POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Software Engineering II

Implementation Document

Project: Best Bike Paths

Authors: Ianosel Bianca Roberta, Gholami Vajihe, Errigo Simone
Version: 1.0
Date: 01.02.2026
Project Link: Errigo-Gholami-Ianosel (github.com)

Contents

Contents **i**

1 Introduction **1**
1.1 Purpose . . . 1
1.2 Scope . . . 1
1.3 Definitions, Acronyms . . . 1
1.3.1 Definitions . . . 1
1.3.2 Acronyms . . . 2
1.4 Revision History . . . 3
1.5 Document Structure . . . 4

2 Implemented Functionalities and Requirements **5**
2.1 Product Functions . . . 5
2.1.1 User Access and Identity Management . . . 5
2.1.2 Trip Recording and Management . . . 5
2.1.3 Path Discovery, Creation and Selection . . . 6
2.1.4 Map-based Visualization and Navigation Support . . . 6
2.1.5 Report Submission and Confirmation . . . 6
2.1.6 Path Condition Evaluation and Ranking . . . 7
2.1.7 Statistics Computation and Caching . . . 9
2.1.8 Weather Data Acquisition and Enrichment . . . 9
2.2 Requirements . . . 10

3 Adopted Development Frameworks **14**
3.1 Adopted Frameworks . . . 14
3.1.1 Mobile Application . . . 14
3.1.2 Backend . . . 15
3.1.3 Data Layer . . . 16
3.2 Adopted Programming Languages . . . 16

3.3 Development Tools 16
3.4 Technologies Used 17
3.5 API Calls 20

4 Source Code Structure **21**
4.1 Frontend 21
4.2 Backend 23
4.3 Server 26

5 Testing Strategy **28**
5.1 Backend Testing 28
5.2 Frontend Testing 30

6 Installation Instructions **32**
6.1 Prerequisites 32
6.2 Backend Setup 33
6.2.1 Production Deployment 33
6.2.2 Local Backend Execution 34
6.2.3 Backend Services in Local Development 36
6.3 Frontend Setup 36
6.3.1 Deployed Mobile Application 37
6.3.2 Local Mobile Application Execution 37

7 References **38**
7.1 Reference Documents 38
7.2 Software Used 38
7.3 Use of AI Tools 38
7.3.1 Tools Used 39
7.3.2 Typical Prompts 39
7.3.3 Input Provided 40
7.3.4 Constraints Applied 40
7.3.5 Outputs Obtained 40
7.3.6 Refinement Process 41

8 Effort Spent **42**

Bibliography **43**

List of Tables **46**

1 | Introduction

1.1. Purpose

The purpose of the Best Bike Paths (BBP) platform is to help cyclists find and use the most suitable routes by collecting real-world path data, aggregating reports, and ranking alternatives. This document focuses on the technical implementation and architectural choices behind the platform. While the overall functional requirements and high-level design are described in the Requirements Analysis and Specification Document (RASD) and the Design Document (DD),this document focuses specifically on the realization of the system.

1.2. Scope

This document, Implementation and Test Document (ITD), provides a comprehensive description of the implementation and testing phases of the BBP platform. Specifically, it focuses on the functionalities developed, the adopted frameworks, and the structure of the source code. Additionally, it includes a detailed testing strategy, covering the procedures, tools, and methodologies used during the developmentprocess. This document also serves as a guide for installing and running the platform, offering installation instructions and addressing any prerequisites or potential issues. The effort spent by the team members is also summarized to provide insight into the workload distribution.

1.3. Definitions, Acronyms

This section provides definitions and explanations of the terms and acronyms used through out the document, making it easier for readers to understand and reference them.

1.3.1. Definitions

- **Path:** A route created by users (manual drawing or GPS-based creation). A path consists of one or more Path Segments.

- **Path Segment:** A portion of a Path defined by its polyline geometry and linked to adjacent segments.
- **Path Status:** The overall condition of a Path or Path Segment, computed from user reports.
- **Path Score/Ranking:** The value used to order suggested paths, derived from status and distance when searching routes.
- **Trip:** A cycling activity tracked through the BBP application. If the user is logged in, the trip is stored and becomes part of the trip history, including temporal and spatial data (e.g., duration, distance, route).
- **Report:** A submission of path information by a logged-in user. Reports describe obstacles and path condition for a specific segment.
- **Freshness:** A metric used when aggregating reports; newer reports carry more weight than older ones when determining Path Status.
- **Obstacle:** An element on a path that negatively impacts cycling conditions, such as potholes or flooding, as identified by users.
- **Manual Creation Mode:** The creation mode in which a logged-in user defines a new path by drawing it on the map.
- **Automatic Creation Mode:** The creation mode in which a logged-in user defines a new path by cycling along it, allowing the system to reconstruct the path using GPS data.
- **Manual Report:** The functionality where a logged-in user manually creates a report by selecting the path condition and obstacle through the application interface.

1.3.2. Acronyms

- **BBP:** Best Bike Paths.
- **API:** Application Programming Interface.
- **APK:** Android Package.
- **CLI:** Command Line Interface.
- **CRUD:** Create, Read, Update, Delete.

- **DBMS:** Database Management System.
- **DD:** Design Document.
- **EAS:** Expo Application Services.
- **GPS:** Global Positioning System.
- **HTTP:** HyperText Transfer Protocol.
- **IPA:** iOS App Store Package.
- **ITD:** Implementation and Test Document.
- **JSON:** JavaScript Object Notation.
- **JWT:** JSON Web Token.
- **NGINX:** Engine X (reverse proxy).
- **ORM:** Object-Relational Mapping.
- **OSRM:** Open Source Routing Machine.
- **QR Code:** Quick Response Code.
- **RASD:** Requirements Analysis and Specification Document.
- **REST:** Representational State Transfer.
- **SDK:** Software Development Kit.
- **TLS:** Transport Layer Security.
- **UI:** User Interface.
- **URL:** Uniform Resource Locator.
- **UX:** User Experience.
- **VPS:** Virtual Private Server.

1.4. Revision History

- Version 1.0 (01 February 2026);

1.5. Document Structure

Mainly the current document is divided into six chapters:

1. **Introduction:** provides an overview of the document, outlining its purpose, scope, and relevance to the project.
2. **Implemented Functionalities and Requirements:** details the functionalities and requirements that have been implemented in the project.
3. **Adopted Development Frameworks:** describes the development frameworks utilized in the project, explaining their roles and benefits.
4. **Source Code Structure:** outlines the organization and structure of the source code, facilitating understanding and navigation.
5. **Testing Strategy:** presents the testing methodologies and strategies employed to ensure the quality and reliability of the software.
6. **Installation Instructions:** provides step-by-step guidance on how to install and set up the software.
7. **References:** lists the references and resources used in the creation of the document and the project.
8. **Effort Spent:** details the distribution of work and time spent by each team member throughout the project.

2 | Implemented Functionalities and Requirements

2.1. Product Functions

This section describes the core functionalities of the app, organized to support the main goals and requirements defined for the project.

2.1.1. User Access and Identity Management

The app allows users to access the system either as guests or as authenticated users, with different levels of interaction. Guest users can explore bike paths, visualize ranked routes between two locations, start trips, and access general information. Their interaction is limited, as they cannot access restricted areas of the app, submit or validate reports, and any data generated during their usage is not saved.
Guest users can register at any time by providing basic identification information. After logging in, users gain access to the full set of functionalities, including viewing previously recorded trips, accessing personal statistics, managing created bike paths, and contributing to the system by submitting or validating reports.
Authenticated users can also access and modify their personal profile information and application preferences. They can update their profile information, including name, email address, and password. Users can also configure application-level preferences, such as the default visibility of newly created paths and the visual appearance of the app.
The app also provides a help and support feature, allowing users to contact the development team directly.

2.1.2. Trip Recording and Management

Users can start cycling trips both as guests and as authenticated users, but any data generated during the activity as a guest user is not saved once the trip ends.
Logged-in users can record and manage their cycling trips with full data persistence. A trip

represents a single cycling activity and includes information such as duration, distance, and the route followed. Additional contextual information, such as weather conditions, may also be retrieved to provide further context to the recorded activity. When the trip ends, all collected data is saved and becomes part of the user's trip history.
The app continuously monitors the user's movement during normal usage to detect potential cycling activity. Bike activity detection is based on movement speed and consistency over time.
When the app detects that the user is biking without an active trip, it proactively suggests starting a trip. If the user is currently using the app as a guest, the system prompts them to log in or register in order to enable trip monitoring and data persistence.

2.1.3. Path Discovery, Creation and Selection

The app allows users to search and explore bike paths. Users can select an origin and a destination, and the app suggests one or more bike paths connecting the two locations. Suggested paths are ordered according to their overall quality and suitability, based on the information available in the system. Users can select a specific path and view its details, including its condition, length, and any associated reports.
Logged-in users can also create new bike paths to extend the set of available routes. Paths can be created in manual mode, by drawing the route directly on the map, or in automatic mode, by recording a cycling activity along the desired trajectory. When creating a path, users can choose whether it should remain private or be shared with the community. Either way, the created path is stored in the system and can be accessed.

2.1.4. Map-based Visualization and Navigation Support

The app uses an interactive map as the main interface to display bike paths and trips. Suggested paths, selected routes, and recorded trips are shown directly on the map, allowing users to easily understand the layout and characteristics of each route.
While navigating, the app shows the user's current position along the selected path, making it easy to track progress in real time. The map may also display reports, providing useful contextual information while the user is moving.

2.1.5. Report Submission and Confirmation

Logged-in users can contribute information about bike paths conditions by submitting reports. Reports describe obstacles, anomalies, or the overall condition of a path segment and can be created manually by the user during an active trip. In all cases, reports are

linked to a specific path.
To improve the reliability of the collected information, the app allows users to confirm or reject existing reports, while cycling. User feedback helps reduce false positives and outdated data, making the reported information more accurate over time.

2.1.6. Path Condition Evaluation and Ranking

The app evaluates path conditions by combining information collected from user reports. Reports describe the condition of a specific segment and are used to compute both segment and path status. Confirmations and rejections are represented as additional reports and contribute to reliability, while report freshness reduces the influence of older observations. The resulting status is then used to rank paths returned by the search feature.

Path Status Scoring Model

Each report submitted by a user refers to a specific path condition, represented as a status. To enable aggregation and quantitative reasoning, each status is mapped to a numerical score as follows:

Path Status	**Numerical Score**
Optimal	5
Medium	4
Sufficient	3
Requires Maintenance	2
Closed	1

Table 2.1: Mapping of path status to numerical scores

Report Freshness Model

Since real-world path conditions evolve over time, the algorithm assigns a freshness weight to each report in order to prioritize recent information. For a report i, freshness is computed from its creation time as:

$$fresh_i = 2^{-\frac{ageMin_i}{H_{min}}}$$

where $ageMin_i$ represents the age of the report in minutes, computed as the difference between the current time and the report creation timestamp. This exponential decay ensures that older reports progressively lose influence over the aggregation process.

Report Validation Contribution

Reports may undergo validation by other users and can be confirmed or rejected. Each confirmation or rejection is stored as a new report on the same segment with status *CONFIRMED* or *REJECTED*. For each report, two partial scores are computed:

$$confirmedScore = \sum fresh_i \quad \text{for each confirmation}$$

$$rejectedScore = \sum fresh_i \quad \text{for each rejection}$$

Reports in the *CREATED* and *CONFIRMED* states contribute to the confirmed score and those in *REJECTED* contribute to the rejected score.

Report Reliability Computation

The overall reliability of a report is computed by combining confirmation and rejection scores through a weighted difference:

$$reportReliability = clamp(1 + \alpha \cdot confirmedScore - \beta \cdot rejectedScore,\ min,\ max)$$

where α and β are weighting parameters controlling the impact of confirmations and rejections respectively, and min and max define lower and upper bounds for reliability. In the current configuration, $\alpha = 0.6$, $\beta = 0.8$, $min = 0.1$, and $max = 2.5$.
Reports whose reliability falls below a minimum threshold are excluded from further aggregation, as they are considered no longer representative of the current path condition.

Path Status Aggregation

Reports first update the status of the specific path segment they refer to. The segment-level status is derived from report scores weighted by reliability. Rejected reports contribute a negative score to reduce the average. The path status is then computed by combining segment statuses with a weighted mix:

$$PathStatusScore = 0.7 \cdot \overline{score}_{reportedSegments} + 0.3 \cdot \overline{score}_{allSegments}$$

where $\overline{score}_{reportedSegments}$ is the average score of segments that have at least one valid report, and $\overline{score}_{allSegments}$ is the average score over all segments in the path. If no segments have valid reports, the path status defaults to the average over all segments.

This approach avoids overly diluting short-path reports while still keeping long paths stable.

Path Status Determination

The final numerical score is mapped back to a discrete path status according to predefined thresholds:

Path Status	Score Range
Optimal	$[4.5, 5]$
Medium	$[3.5, 4.5)$
Sufficient	$[2.5, 3.5)$
Requires Maintenance	$[1.5, 2.5)$
Closed	$[1, 1.5)$

Table 2.2: Mapping of numerical scores to discrete path statuses

Impact on Path Ranking

The computed path status directly influences the ranking of paths during route discovery. Paths with higher evaluated conditions are prioritized when suggesting routes between an origin and a destination. Since the merging algorithm is continuously updated as new reports and validations are received, path rankings dynamically adapt to evolving real-world conditions.

2.1.7. Statistics Computation and Caching

The app computes statistics to help users better understand their cycling activities and overall performance. Statistics are generated from recorded trip data.
To keep the system efficient, the app avoids unnecessary recomputations. Aggregated statistics are updated only when new trips are recorded, while previously computed results are reused whenever possible. Per-trip statistics are generated when needed and then stored, so they do not have to be recalculated every time they are accessed. As a result, users can access up-to-date statistics without affecting the overall performance of the app.

2.1.8. Weather Data Acquisition and Enrichment

The app retrieves weather information from external meteorological services to provide additional context for cycling activities. Weather data, such as temperature, wind, and

other relevant conditions, is associated with recorded trips based on the time and location of the activity.
Weather information is collected when a trip is completed, ensuring that the recorded conditions accurately reflect the environment in which the activity took place. This allows users to better understand their trips and interpret performance data in relation to external factors.

2.2. Requirements

Rx	Description	Implemented
R1	The system shall allow guest users to create an account by providing personal information and credentials.	Yes
R2	The system shall allow registered users to log into the application using valid credentials.	Yes
R3	The system shall allow logged-in users to view their profile and account settings.	Yes
R4	The system shall allow logged-in users to update their profile and account settings.	Yes
R5	The system shall allow logged-in users to log out of the application, ending their current session.	Yes
R6	The system shall allow guest users to start a cycling trip.	Yes
R7	The system shall allow guest users to stop a currently active trip, but shall not store any trip data after the trip ends.	Yes
R8	The system shall allow the user to start a trip only when their GPS position matches the path origin.	Yes
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.	Yes
R10	The system shall set the current GPS position as trip origin when starting from auto-detection.	Yes
R11	The system shall automatically stop the active trip when the user's GPS position deviates from the selected path within a certain threshold.	Yes

Continued on next page

Rx	Description	Implemented
R12	The system shall allow logged-in users to start a cycling trip in manual or automatic mode.	Partial - Only Manual Mode required
R13	The system shall allow logged-in users to stop a currently active trip and save the recorded data.	Yes
R14	The system shall collect GPS data during trip recording.	Yes
R15	The system shall collect motion sensor data (accelerometer, gyroscope) during trip recording only when Automatic Mode is enabled.	No - Not required
R16	The system shall allow logged-in users to view the list of their recorded trips.	Yes
R17	The system shall allow logged-in users to view a summary of their overall cycling statistics (total distance, total time, average speed, etc.).	Yes
R18	The system shall allow logged-in users to view statistics for each trip (distance, speed, duration, etc.).	Yes
R19	The system shall display the route and reported obstacles associated with a recorded trip.	Yes
R20	The system shall allow logged-in users to delete a recorded trip.	Yes
R21	The system shall communicate with external weather services to retrieve meteorological data related to the time and location of a trip.	Yes
R22	The system shall detect when a user is cycling based on speed and acceleration patterns.	Yes
R23	The system shall detect irregular movements from sensor data that may suggest potholes or surface defects when Automatic Mode is enabled.	No - Not required
R24	The system shall present automatically detected path and obstacle data to the logged-in user for manual confirmation before publishing.	No - Not required
R25	The system shall allow logged-in users to manually create a new bike path by drawing segments.	Yes

Continued on next page

Rx	Description	Implemented
R26	The system shall allow logged-in users to manually report obstacles or problems on a bike path while performing an active trip.	Yes
R27	The system shall allow logged-in users to manually confirm or reject the presence of obstacles reported by other users.	Yes
R28	The system shall allow logged-in users to create a new bike path in automatic mode using GPS tracking.	Yes
R29	The system shall allow logged-in users to delete their previously created paths.	Yes
R30	The system shall allow logged-in users to set the visibility of their created paths as public or private.	Yes
R31	The system shall aggregate multiple user reports referring to the same path segment.	Yes
R32	The system shall evaluate the reliability of each path segment based on the number of confirmations and report freshness.	Yes
R33	The system shall determine the current status of a path (optimal, medium, sufficient, requires maintenance, closed).	Yes
R34	The system shall allow any user (guest or logged-in) to view the detailed status and latest reports of a selected bike path.	Yes
R35	The system shall allow any user (guest or logged-in) to browse available public bike paths on a map.	Yes
R36	The system shall allow any user to search for bike paths connecting two locations.	Yes
R37	The system shall compute suggested routes based on path quality and distance.	Yes
R38	The system shall rank suggested routes according to their safety and quality.	Yes
R39	The system shall display the user's current GPS position during navigation along a selected path.	Yes

Continued on next page

Rx	Description	Implemented
R40	The system shall send pop-ups to warn users about nearby obstacles or closed path segments during an active trip.	Yes
R41	The system shall interface with map and geocoding services to translate addresses into coordinates and render paths.	Yes
R42	The system shall ensure that communication with all external services (map, weather) handles temporary unavailability gracefully.	Yes

Table 2.3: Mapping between BBP Requirements and implemented functionalities

3 | Adopted Development Frameworks

3.1. Adopted Frameworks

The BBP platform relies on a set of technologies selected to cover the needs of the mobile client, the server side, and the persistence layer. In the sections below, we outline the core frameworks used in each area and briefly describe their role, key characteristics, and why they were chosen for this project.

3.1.1. Mobile Application

The BBP mobile application is developed using **React Native** within the **Expo** ecosystem. This choice allows the use of a single codebase to target both Android and iOS platforms, while ensuring consistent access to device features such as GPS. Expo also simplifies build, deployment, and permission management, reducing development overhead and improving iteration speed during testing.

Application navigation is handled through **Expo Router**, which provides a file-based routing system. This approach helps maintain a clear and scalable screen structure as the application grows. Global state management is implemented using **Zustand**, selected for its minimal boilerplate and efficient update mechanism, which is particularly suitable for managing user sessions and cross-screen shared state.

The user interface is built using **React Native Paper**, which offers a Material Design-compliant component library with built-in theming support. This enables a consistent visual appearance across devices and simplifies the implementation of light, dark, and system-defined themes. **Lucide Icons** is used to provide a lightweight and consistent icon set across the application.

Map visualization and interaction represent a core feature of the app and are implemented using **React Native Maps**. This library is used to display bike paths, recorded trips, reports, and ranking-related visual overlays. Location tracking is managed through **Expo Location**, which handles permission requests and continuous GPS updates required for

navigation and trip recording functionalities.
Communication with the backend services is handled through **Axios**, which manages HTTP requests and response handling. Client-side input validation is implemented using **Zod**, which defines schemas for forms such as authentication and profile management and integrates with React Hook Form through submit-time validation. Authentication tokens and sensitive session data are securely stored on the device using **Expo Secure Store**.

3.1.2. Backend

Express.js, or simply Express, is a back end web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs.
The main reason for choosing Express.js as the backend framework is because it is very easy-to-use and flexible, together with its ability to maintain high performance while offering a comprehensive set of features. Express provides pre-built functions, libraries, and tools that help accelerate the web development process.
It has also one of the most powerful and robust routing system that assists your application in responding to a client request via a particular endpoint. With the routing system in Express.js, we were able to split the routing system into manageable files using the framework's router instance. It is very helpful in managing the application structure grouping different routes into a single folder/directory.
ExpressJS uses middleware to process incoming requests before they reach their final destination. This allows us to perform tasks such as authentication, validation, and logging in a reusable and modular manner.
Jest is a testing framework for JavaScript applications, developed and maintained by Meta and distributed as open-source software. It provides an all-in-one solution for writing, executing, and maintaining automated tests, including unit tests and integration tests. Jest includes built-in support for test runners, mocking utilities, and code coverage analysis, reducing the need for additional external dependencies.
Jest was selected primarily for its integration with modern JavaScript and Node.js environments. It allowed us to quickly set up a testing environment with minimal overhead, while still offering advanced configuration options when needed. The framework supports asynchronous testing out of the box, which is essential for backend applications that rely heavily on asynchronous operations such as database access and HTTP requests.

3.1.3. Data Layer

For data storage and management, the BBP platform relies on **PostgreSQL** as its primary **DBMS**. This choice is motivated by its proven reliability and by its relational data model, which fits well with the platform's core entities, such as users, paths, segments, and trips, and supports the enforcement of consistency constraints across related data.
A relational approach is particularly suitable for BBP, where path-related information evolves over time and must remain consistent despite frequent user-generated updates. **PostgreSQL** provides the transactional guarantees and integrity mechanisms required to manage this evolving dataset in a robust and predictable way.
The interaction between the modular **Express** backend and the database is handled through **Prisma**, which is adopted as the **ORM**. **Prisma** generates a type-safe client starting from a single schema definition, ensuring that database queries are strongly typed and aligned with the **TypeScript** types used throughout the application. This approach helps detect data access errors at compile time and keeps the persistence layer consistent as the data model evolves during development.

3.2. Adopted Programming Languages

The platform is mainly developed using **TypeScript**, which is adopted across both the mobile application, built with **React Native** and **Expo**, and the backend services, built with **Express** and **Prisma**. Using a single, strongly typed language across different layers of the system improves maintainability and reduces the likelihood of runtime errors, while also simplifying development workflows.
On the server side, **TypeScript** integrates smoothly with **Prisma**, as the generated client types help identify data access issues at compile time. On the client side, it works well with form validation and API interaction, making input models and component properties explicit and easier to evolve over time.
Overall, this language choice results in clearer and more robust code, supports safer refactoring, and reduces the effort required for developers to work across multiple components of the platform.

3.3. Development Tools

The development workflow is based on **Node.js** and **npm**, which are used to manage dependencies and execute project scripts for both the backend and the mobile application. On the backend side, **Docker** and **Docker Compose** are adopted to build and

run the service in a reproducible and isolated environment, ensuring consistency across development and production setups. Database-related tasks, including client generation and schema migrations, are handled through the **Prisma CLI**.
For the mobile application, development and local testing are carried out using the **Expo CLI**. The **Expo Go** application enables real-time previews on physical devices by scanning the Metro QR code, allowing rapid iteration and immediate feedback during development. For distributable builds, the **EAS CLI** is used to generate **Android** and **iOS** build artifacts, with APK files produced from the Android build output.
In the production environment, the backend is exposed through a shared **Nginx** reverse proxy. HTTPS termination and certificate management are handled using **Cloudflare Origin Certificates**, which centralize TLS configuration and request routing while keeping backend services isolated from direct internet exposure.
To support integration testing and a shared development workflow, **EchoAPI** is used to mock and inspect backend responses. This tool allows team members to validate client-side behavior against expected API outputs, facilitating coordination between frontend and backend development and reducing coupling during implementation phases.

3.4. Technologies Used

The following is a list of the main technologies, libraries, and tools used in the development of the BBP platform:

- **React Native**: framework for building native mobile applications using JavaScript and React, enabling code reuse across Android and iOS platforms.[22]
- **Expo**: development platform built on top of React Native that simplifies configuration, permissions management, build processes, and access to native device features.[10]
- **Expo Router**: file-based routing system used to structure navigation and manage application screens in a scalable and maintainable way.[11]
- **Zustand**: lightweight state management library used to handle global application state such as authentication, user preferences, and shared UI data.[29]
- **React Native Paper**: UI component library implementing Material Design principles, providing accessible and themable components for a consistent user interface.[3]
- **Lucide Icons**: icon library used to supply a coherent and lightweight set of vector icons across the application.[21]

- **React Native Maps**: library used for interactive map rendering, including bike path visualization, trip tracking, and report overlays.[34]
- **Expo Location**: module used to manage location permissions and retrieve continuous GPS updates required for navigation and trip recording features.[13]
- **Axios**: HTTP client used to perform asynchronous requests to backend APIs and handle responses.[2]
- **Zod**: schema validation library used for client-side data validation, integrated with React Hook Form to enforce input constraints in forms.[38]
- **React Hook Form**: form management library used to handle user input efficiently and reduce unnecessary re-renders in complex forms.[33]
- **Expo Secure Store**: secure storage mechanism used to persist authentication tokens and sensitive session data on the device.[14]
- **Joi**: data validation library used to define schemas for request payloads, ensuring that incoming data conforms to expected formats and constraints before being processed by the application.[17]
- **bcrypt**: cryptographic hashing library used to securely store user passwords by applying adaptive hashing algorithms, protecting against brute-force and rainbow table attacks.[18]
- **jsonwebtoken**: library used to implement stateless authentication through JSON Web Tokens (JWT), enabling secure user session management and authorization across API endpoints.[1]
- **cors**: middleware that enables and configures Cross-Origin Resource Sharing, allowing controlled access to backend resources from client applications hosted on different domains.[15]
- **Pino**: high-performance logging library designed for Node.js applications, used to record structured logs for monitoring, debugging, and auditing purposes.[27]
- **pino-pretty**: development tool used to format Pino logs into a human-readable form, improving log readability during debugging and local development.[28]
- **Supertest**: HTTP testing library that allows automated testing of RESTful APIs by simulating HTTP requests and validating responses without requiring a running network server.[19]

- **PostgreSQL**: relational database management system used as the primary data store for the platform. It provides strong transactional guarantees, support for complex relational schemas, and integrity constraints required to manage evolving entities such as users, paths, segments, trips, and reports.[30]
- **Prisma**: Object-Relational Mapping (ORM) tool used to interface the Express backend with the database. Prisma generates a type-safe client from a centralized schema definition, ensuring consistency between the database model and the TypeScript types used in the application, and enabling compile-time detection of data access errors.[32]
- **Node.js**: JavaScript runtime environment based on the V8 engine, enabling the execution of server-side applications using an event-driven, non-blocking I/O model that is well suited for scalable backend services.[26]
- **npm**: package manager used to manage project dependencies and execute predefined scripts for development, testing, and build tasks.[25]
- **Docker**: containerization platform used to package the backend application and its dependencies into isolated and reproducible runtime environments.[6]
- **Docker Compose**: orchestration tool used to define and manage multi-container setups, including backend services and database instances, during local development and deployment.[5]
- **Prisma CLI**: command-line tool used to manage database schemas, generate type-safe clients, and apply migrations in a controlled and consistent manner.[31]
- **Expo CLI**: development tool used to run, test, and debug the mobile application locally within the Expo ecosystem.[8]
- **Expo Go**: companion application that enables real-time testing of the mobile app on physical devices by loading the development bundle via QR code.[12]
- **EAS CLI**: build and deployment tool used to generate distributable Android and iOS application artifacts, including APK files for Android.[9]
- **Nginx**: reverse proxy used in production to expose backend services, handle request routing, and provide an additional security layer between clients and internal services.[24]

- **Cloudflare Origin Certificates**: TLS certificate mechanism used to terminate HTTPS connections at the proxy level while keeping backend services isolated from direct internet exposure [4].
  item **EchoAPI**: API development and testing tool used to perform and inspect HTTP requests against backend endpoints and validate request and response formats [7].

3.5. API Calls

Any API not included in the DD should be mentioned here.

4 | Source Code Structure

4.1. Frontend

The following directory tree provides an overview of the frontend source code structure of the BestBikePaths mobile app. The structure reflects the organization adopted by Expo Router and the separation of concerns between routing, UI components, and utilities.

```
src/
|-- api/                          # Backend API communication
|   |-- client.ts                 # Axios client with interceptors
|   |-- auth.ts                   # Authentication API wrappers
|   |__ ...                       # Other API modules
|-- app/                          # Expo Router navigation structure
|   |-- (auth)/                   # Authentication-related screens
|   |   |-- _layout.tsx           # Auth flow layout
|   |   |-- welcome.tsx           # Welcome screen
|   |   |__ ...                   # Other auth screens
|   |-- (main)/                   # Main application screens
|   |   |-- _layout.tsx           # Bottom navigation and access guards
|   |   |-- home.tsx              # Map-based path search and navigation
|   |   |__ ...                   # Other main screens
|   |-- _layout.tsx               # Root layout with global providers
|   |__ +not-found.tsx            # Fallback screen for unknown routes
|-- android/                      # Android native project
|-- assets/                       # Static assets
|   |-- images/                   # Icons and images
|   |__ fonts/                    # Custom fonts
|-- auth/                         # Authentication and session management
|   |-- authSession.ts            # In-memory session handling
|   |__ storage.ts                # Zustand store + SecureStore integration
|-- components/                   # Reusable UI components
|   |-- ui/                       # UI primitives (buttons, inputs, popups)
```

```
|   |   |-- AppButton.tsx         # Custom button component
|   |   |__ ...                   # Other UI primitives
|   |-- icons/                    # Icon wrappers and helpers
|   |   |__ LucideIcon.tsx        # Lucide icon set integration
|   |__ ...                       # Other components
|-- constants/                    # Static configuration values
|   |-- Colors.ts                 # Color palette definitions
|   |__ Privacy.ts                # Privacy options
|-- hooks/                        # Custom React hooks
|   |-- useBottomNavVisibility.tsx # Bottom navigation visibility
|   |__ ...                       # Other hooks
|-- ios/                          # iOS native project
|-- tests/                        # Automated tests
|   |-- integration/              # Integration tests
|   |-- mocks/                    # Module mocks
|   |-- unit/                     # Unit tests
|   |__ utils/                    # Test helpers
|-- theme/                        # Theming configuration
|   |-- layout.ts                 # Layout helpers
|   |-- mapStyles.ts              # Map style definitions
|   |-- paperTheme.ts             # React Native Paper theme
|   |__ typography.ts             # Typography settings
|-- utils/                        # Utility functions
|   |-- geo.ts                    # Distance and route helpers
|   |-- apiError.ts               # API error normalization
|   |__ ...                       # Other utilities
|-- validation/                   # Zod validation schemas
|   |-- auth.ts                   # Login, signup, profile schemas
|   |__ ...                       # Other validation schemas
|-- .expo/                        # Expo local state
|-- .env                          # Environment variables
|-- .gitignore                    # Git ignore rules
|-- .npmrc                        # NPM configuration
|-- app.config.js                 # Expo configuration
|-- app.json                      # Expo app configuration
|-- babel.config.js               # Babel configuration
|-- eas.json                      # Expo Application Services configuration
```

```
|-- expo-env.d.ts                # Expo TypeScript env definitions
|-- jest.config.js               # Jest configuration
|-- jest.setup.ts                # Global test setup
|-- node_modules/                # Installed dependencies
|-- package.json                 # Dependencies and scripts
|-- package-lock.json            # Locked dependency versions
|__ tsconfig.json                # TypeScript configuration
```

4.2. Backend

The following directory tree outlines the structure of the backend source code of the BestBikePaths system. The backend follows a modular and layered architecture, separating concerns between routing, middleware, business logic, data access, and external service integrations.

```
backend/
|-- prisma/                              # Prisma ORM configuration
|   |-- schema.prisma                    # Database schema definition
|   |-- migrations/                      # Database migrations
|   |__ json.types.d.ts                  # Custom Prisma JSON type definitions
|-- src/                                 # Source code
|   |-- constants/                       # Application-wide constants
|   |   |__ appConfig.ts                 # General app constants
|   |-- errors/                          # Custom error classes
|   |   |-- app.errors.ts                # Application-specific errors
|   |   |__ index.ts                     # Export all error classes
|   |-- managers/                        # Business logic
|   |   |-- auth/                        # Authentication logic
|   |   |-- path/                        #  Path management logic
|   |   |__ ...                          # Other managers
|   |-- middleware/                      # Middlewares
|   |   |-- jwt.auth.ts                  # JWT authentication middleware
|   |   |-- http.logger.ts               # HTTP request logging middleware
|   |   |__ ...                          # Other middlewares
|   |-- routes/                          # API route definitions
|   |   |-- v1/                          # Version 1 of the API
|   |       |-- index.ts                 # API version entry point
|   |       |-- auth.routes.ts           # Authentication routes
```

```
|   |       |-- user.routes.ts          # User routes
|   |       |__ ...                     # Other routes
|   |-- schemas/                        # Validation schemas
|   |   |-- auth.schema.ts              # Auth-related schemas
|   |   |-- user.schema.ts              # User-related schemas
|   |   |__ ...                         # Other schemas
|   |-- services/                       # External service integrations
|   |   |-- weather.service.ts          # OpenMeteo API integration
|   |   |__ ...                         # Other services
|   |-- tests/                          # Test files
|   |   |-- integration/                # Integration tests
|   |   |   |-- auth.integration.test.ts # Auth integration tests
|   |   |   |__ ...                     # Other integration tests
|   |   |-- unit/                       # Unit tests
|   |       |--  auth.manager.test.ts   # Auth manager tests
|   |       |__ ...                     # Other unit tests
|   |-- types/                          # TypeScript type definitions
|   |   |-- express/                    # Express-related types
|   |   |   |-- index.d.ts              # Custom Express types
|   |   |__ coordinate.types.ts         # Coordinate types
|   |   |__ ...                         # Other type definitions
|   |-- utils/                          # Utility functions
|   |   |-- prisma-client.ts            # Prisma client instance
|   |   |-- geo.ts                      # Geospatial utility functions
|   |   |__ ...                         # Other utility functions
|   |__ server.ts                       # Server entry point
|-- .env                                # Environment variables
|-- .gitignore                          # Git ignore rules
|-- docker-compose.yml                  # Docker Compose configuration
|-- docker-compose-dev.yml              # Dev Docker Compose configuration
|-- Dockerfile                          # Dockerfile for backend service
|-- Dockerfile.dev                      # Dev Dockerfile
|-- jest.config.mjs                     # Jest configuration
|-- node_modules/                       # Installed dependencies
|-- package.json                        # Dependencies and scripts
|-- package-lock.json                   # Locked dependency versions
|-- prisma.config.ts                    # Prisma configuration
```

```
|-- setup.test.ts                    # Global test setup
|__ tsconfig.json                    # TypeScript configuration
```

4.3. Server

The following directory structure provides a high-level overview of the server-side deployment environment used to host the BestBikePaths system. It illustrates the organization of reverse proxy configuration, SSL certificates, and Dockerized application services on the target server.

```
/opt/
|-- nginx/                              # Shared NGINX reverse proxy
|   |-- conf.d/                         # Virtual hosts and routing configuration
|   |   |-- 00_cloudflare_realip.conf   # Cloudflare real IP handling
|   |   |-- 00_globals.conf             # Global NGINX settings
|   |   |-- 01_upstreams.conf           # Upstream definitions
|   |   |-- api.conf                    # BBP backend API proxy rules
|   |   |__site.conf                    # Additional hosted site configuration
|   |-- ssl/                            # TLS certificates (Cloudflare Origin)
|   |   |-- bia3ia-origin.crt
|   |   |-- bia3ia-origin.key
|   |   |-- origin.crt
|   |   |__origin.key
|   |-- log/                            # NGINX logs
|   |   |-- access.log
|   |   |__error.log
|   |__update-cloudflare-ips.sh         # Script to update Cloudflare IP ranges
|
|-- bbp-backend/                        # BestBikePaths backend service
|   |-- Dockerfile                      # Backend container definition
|   |-- docker-compose.yml              # Backend service orchestration
|   |-- .env                            # Environment configuration
|   |__...
|
|-- osrm/                               # OSRM routing service
|   |-- docker-compose.yml              # OSRM service orchestration
|   |-- pavia-milano.osm.pbf            # OpenStreetMap extract
|   |-- pavia-milano.osrm*              # Preprocessed routing datasets
|   |__*.osrm.*                         # OSRM auxiliary data files
|
|-- residenza-clas-marina/              # Additional hosted application
```

```
|   |-- docker-compose.yml            # Service orchestration
|   |-- app/                          # Application source code
|   |__nginx/                         # Service-specific NGINX config
|
|__containerd/                        # Container runtime (system-managed)
    |-- bin/
    |__lib/
```

5 | Testing Strategy

5.1. Backend Testing

Backend tests focus on API correctness, business logic reliability, and error handling across the service layer. They are implemented with **Jest** and organized into unit and integration suites under `src/tests/unit` and `src/tests/integration`. Unit tests isolate managers and utilities by mocking persistence and external services, while integration tests exercise the Express app through real HTTP requests using Supertest.

Unit Testing

Unit tests were used to validate the correctness of the backend *business logic* in isolation, with the main goal of detecting defects early and keeping failures easy to debug.
We implemented unit tests using **Jest** (TypeScript support via `ts-jest`). Dependencies that would make tests slow were replaced with **mocks** through `jest.mock`. In particular, we mocked the persistence layer (`QueryManager` / DB access), external services (weather/geocoding/snapping functions), and sensitive utilities (`bcrypt` for password hashing, and the logger). This allowed us to test each method by controlling the returned values and by asserting the correctness of the output (HTTP status and JSON payload), the correctness of side effects (which mocked functions are called and with which parameters) and the correct propagation of errors (via Express `next()`).
To keep unit tests close to the real execution environment, methods were invoked with mocked Express `Request`/`Response` objects and a mocked `next` callback. Test cases cover common edge cases such as missing fields, invalid inputs, unauthorized access, conflicts (duplicate resources), and not-found scenarios.

Integration Testing

Integration tests were used to validate the correctness of component interactions at the *API boundary*, ensuring that routing, middleware, validation, authentication, error handling, and controller/manager orchestration work correctly when exercised through real HTTP calls. In our project, integration tests are implemented using **Supertest** as an

HTTP test driver: instead of manually mocking `Request`/`Response`, Supertest performs requests against the Express application instance (`app`) and verifies the resulting behavior. The focus of these tests is on verifying that each endpoint:

- accepts and rejects payloads consistently (validation and error codes),
- enforces authorization correctly (requests with missing/invalid tokens),
- returns the expected HTTP status codes and response bodies, and
- correctly maps internal failures to the public API error format.

To keep integration tests fast, we stubbed external dependencies where needed. In particular, we mocked the Prisma client layer so that endpoint-to-manager flows can be validated without requiring a real database instance for every test run. Authentication flows are tested by generating JWTs within the test suite (with test secrets configured in the environment `setup.test.ts`), thus simulating authenticated requests realistically while keeping the execution fully automated.

How to run backend tests

Backend tests are executed inside the backend container to ensure a consistent environment. Before running tests, make sure the backend stack is up:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml up -d
```

Then run the Jest suite inside the running container:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml exec api
npm run test
```

To run only integration tests, filter by filename pattern or target a specific file:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml exec api
 npm run test -- integration
docker compose -f docker-compose.yml -f docker-compose.dev.yml exec api
 npm run test -- user.integration.test.ts
```

5.2. Frontend Testing

Frontend tests were used to validate the correctness of the user interface logic, client-side state management, and screen-level interactions of the mobile application.
Tests are implemented using **Jest** in a **React Native** environment and are organized into *unit* and *integration* test suites, located respectively under `src/tests/unit` and `src/tests/integration`. The test environment is configured through `jest.setup.ts`, where Expo modules and native dependencies are mocked.

Unit tests

Frontend unit tests focus on validating isolated pieces of client-side logic without involving UI rendering or navigation flows. These tests target reusable utilities, hooks, API clients, and validation logic that represent the foundation of the application behavior.
Examples of unit-tested components include geospatial utilities, path mapping and transformation helpers, report option builders, authentication storage and session helpers, and API wrapper functions. By isolating these components, unit tests allow precise control over inputs and outputs and make failures easier to diagnose.

Integration tests

Frontend integration tests validate the interaction between components at the screen level, ensuring that UI elements, navigation, and state transitions work correctly when exercised together. These tests are implemented using **React Native Testing Library**, which simulates user interactions and renders components in an environment close to real execution.
Integration tests cover end-to-end UI flows such as authentication (login, signup, profile editing), navigation guards distinguishing guest and authenticated access, path discovery and trip-related flows, report submission and confirmation, and settings and profile management screens. The focus is on verifying that user actions result in the expected UI updates and API calls, and that error states are handled consistently.

How to run frontend tests

Frontend tests can be executed from the mobile application directory. All required dependencies are installed via npm, and the Jest test suite can be run as follows:

```
npm run test
```

It is also possible to execute a subset of tests by filtering on filename patterns or by specifying a specific test file:

```
npm run test -- integration
npm run test -- login.integration.test.tsx
```

6 | Installation Instructions

The BBP platform can be used in different configurations, depending on the intended use case and on whether the user aims to simply run the application or to actively develop and test it.
In particular, three main usage scenarios are supported:

- **Remote backend with prebuilt mobile application**: the user installs a prebuilt APK and interacts with an already deployed backend instance.
- **Fully local execution**: both the backend services and the mobile application are executed locally for development and testing purposes.
- **Hybrid configuration**: the mobile application is run locally while relying on a remotely deployed backend.

6.1. Prerequisites

To run the BBP platform locally, a set of prerequisites must be satisfied to ensure a consistent and reproducible development environment.

- **Node.js and npm**: required to manage dependencies and execute project scripts for both the backend services and the mobile application. All build, test, and development workflows rely on the Node.js runtime.
- **Docker and Docker Compose**: required to run backend services and auxiliary components in isolated containers. In the local setup, Docker Compose is used to start the BBP backend, the PostgreSQL database, and the OSRM routing service, ensuring a configuration that closely mirrors the production environment without requiring manual installation of these components on the host system.
- **Expo CLI**: required to run the mobile application locally during development. It is used to start the Metro bundler and to enable live previews of the application on emulators or physical devices.

- **Expo Go**: required on a physical mobile device to preview the application during development by connecting to the local Metro bundler started via Expo CLI. May be replaced with an emulator.

6.2. Backend Setup

The BBP backend can be executed either as a remotely deployed service or as a locally running instance for development and testing purposes.

6.2.1. Production Deployment

The BBP backend is deployed on a **VPS** using a **container-based architecture** built on Docker and Docker Compose. The service is designed to coexist with other applications hosted on the same server while remaining isolated and not directly exposed to the public network.

The backend application runs as a stateless service inside **Docker containers** and listens on port 3000, which is reachable only within the container network. No backend container exposes public ports on the host system. All incoming traffic is handled by a **shared NGINX reverse proxy**, which is the only component exposing ports 80 and 443.

NGINX acts as the **single entry point** for the system and is responsible for HTTPS termination, request routing, load balancing, and basic traffic filtering. TLS is managed using **Cloudflare Origin Certificates**, with Cloudflare acting as **DNS provider and upstream security layer**. Requests validated by Cloudflare are forwarded to the reverse proxy, which then routes them to the backend service.

The backend is horizontally scalable and is deployed using **multiple container replicas**. In the current configuration, three backend instances are executed in parallel. NGINX resolves backend containers dynamically using **Docker's embedded DNS** and **balances incoming requests** across replicas through an upstream definition. This approach allows the system to scale without introducing a dedicated load balancer and ensures high availability of the API layer.

The deployment process starts by copying the backend project files to the server. This can be performed from a local machine using a secure file transfer mechanism, for example:

```
rsync -avz ./BACKEND/ user@<SERVER_IP>:/opt/bbp-backend/
```

Once the project is available on the server, environment-specific configuration values are provided through a dedicated **.env** file. This file defines runtime parameters such as authentication secrets, database connection details, and service timeouts. Sensitive values are **never committed** to version control and are configured directly on the server.

The backend is built and started using **Docker Compose**. A shared **Docker network** is created once and is used to connect the reverse proxy, backend replicas, and auxiliary services. The backend containers join this network and expose port 3000 only internally. The services can be built and started with the following commands:

```
docker compose build --no-cache
docker compose --profile tools run --rm migrate
docker compose up -d --scale api=3
```

Database **migrations** are executed explicitly through a dedicated migration container and are only run when schema changes are introduced. This avoids unintended schema modifications during routine restarts.
Persistent data storage is handled through **PostgreSQL** accessed via **Prisma Accelerate**. This configuration allows the backend to rely on a **managed database connection layer** without hosting a database instance directly on the server, while still preserving transactional guarantees and schema consistency.
Caching of data relies on **Redis**, which runs alongside the backend in the same server environment and is reachable only inside the internal Docker network. The backend connects using the environment variables **REDIS_HOST** and **REDIS_PORT**, and uses Redis as a cache layer while keeping the database as the source of truth.
In addition to the core backend service, snapping-related functionalities are delegated to a separate **OSRM service**, which runs in its own container. OSRM is responsible for snapping user-defined paths to the OpenStreetMap road network using a cycling profile. The backend communicates with OSRM through an internal HTTP interface and exposes a dedicated API endpoint to the mobile application.
After deployment, the backend can be verified by inspecting container status and logs:

```
docker compose ps
docker logs -f bbp-backend-api-1 --tail=50
```

A dedicated **health endpoint** is exposed and can be used by the reverse proxy to verify service availability. If correctly deployed, the backend responds to authenticated API requests through the public API endpoint exposed by the reverse proxy.

6.2.2. Local Backend Execution

For local development and testing, the BBP backend can be executed using a Docker-based setup that mirrors the production environment, while enabling development-specific features such as live reload. To keep the production configuration unchanged, the local

development setup is defined through an additional Docker Compose file and a dedicated development Dockerfile.
Before starting the backend locally, developers must ensure they are operating from the backend project directory, which contains the production **docker-compose.yml**, the development override **docker-compose.dev.yml**, and the related **Dockerfile.dev**. All commands must be executed from this directory.
The development configuration builds the backend image using **Dockerfile.dev** and runs the service with **NODE_ENV=development**. Project source files are mounted into the container through bind mounts, so changes to the codebase are immediately reflected without rebuilding the image. Node.js dependencies are installed inside the container during the image build and are not required on the host system.
Environment-specific configuration values are provided through a dedicated **.env** file located in the backend project directory. This file defines authentication secrets, database connection parameters, and service endpoints. Additional local-only variables can be added when needed without affecting the production configuration.
To build and start the backend locally, the development override is applied on top of the base configuration using Docker Compose:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml up -d
 --build
```

Once the backend container is running, application logs can be inspected in real time to monitor startup and runtime behavior:

```
docker logs -f bbp-backend-api --tail=50
```

A basic health check endpoint is exposed by the backend and can be used to verify that the service is running correctly:

```
curl -i http://localhost:3000/health
```

In the development configuration, port 3000 is published on the host system. As a result, the backend API is available at **http://localhost:3000** and can be accessed by the mobile application or by API testing tools.
Automated tests can be executed directly inside the running backend container. This allows tests to run in the same environment as the application, without requiring local dependency installation. Tests can be started with:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml
exec api npm run test
```

When the local backend is no longer needed, all running services can be stopped and removed using Docker Compose:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml down
```

This command stops and removes the containers associated with the local development environment without affecting images or configuration files. All backend services can be stopped and restarted at any time without affecting the host system, as the entire runtime environment is isolated within containers.

6.2.3. Backend Services in Local Development

Some backend functionalities depend on external services. In a local development environment, these services behave differently depending on their availability and deployment model.

Caching of trip and statistics data relies on **Redis**, which is required for the backend to operate correctly. In local development, Redis starts together with the backend using Docker Compose. Connection settings are controlled via **REDIS_HOST** and **REDIS_PORT**.

Weather-related features rely on the **Open-Meteo** service. This integration works out of the box in local development, as Open-Meteo provides a public API that does not require authentication or API keys.

Geocoding functionalities are handled through **Nominatim**. In the local setup, no additional configuration is required, as the backend uses the public Nominatim API to resolve textual addresses into geographic coordinates.

Path snapping functionalities depend on **OSRM** (Open Source Routing Machine). Unlike the previous services, OSRM is not a public API and must be explicitly deployed as a separate service. In the production environment, OSRM runs as a dedicated container and is accessed by the backend through an internal HTTP interface. In a local development setup, OSRM is not available by default.

As a result, snapping-related features are disabled locally unless an OSRM instance is explicitly configured. Developers who need to test path snapping locally have to deploy OSRM locally by following the OSRM deployment procedure, which involves downloading OpenStreetMap extracts, preprocessing the dataset, and running the OSRM service inside a Docker container.

6.3. Frontend Setup

The BBP mobile application can be used either as a prebuilt artifact connected to the production backend or as a locally executed application for development and testing

purposes.

6.3.1. Deployed Mobile Application

The deployed version of the BBP mobile application is distributed as a prebuilt Android **APK**. This configuration is intended for demonstration, evaluation, and production usage scenarios where no local development environment is required.
Users can install the APK directly on their Android device and immediately start using the application. In this setup, the mobile client connects to the remotely deployed backend instance exposed through the public API endpoint. All backend services, including authentication, path discovery, trip management, and report handling, are accessed through the production infrastructure.

6.3.2. Local Mobile Application Execution

For development and testing purposes, the BBP mobile application can be executed locally using **Expo CLI**. This configuration enables rapid iteration, hot reload, and debugging during frontend development.
The application can be run either on a physical mobile device using **Expo Go** or on an Android or iOS emulator. In both cases, Expo CLI starts the Metro bundler locally and serves the application bundle to the target device.
Before starting the application, developers must ensure they are operating from the mobile application project directory. Dependencies are installed using npm, and the development server is started as follows:

```
npx expo start
```

During local execution, the mobile application can be configured to target either the production backend or a locally running backend instance. This is controlled through environment variables defined in a `.env` file. In particular, the API base URL must be updated to point to the desired backend endpoint before starting the Expo development server.
When connected to a locally running backend, this setup allows developers to test end-to-end flows, including authentication, path search, trip recording, and report submission. Unlike the prebuilt APK, the locally executed application fully supports configuration changes.

7 References

7.1. Reference Documents

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

- Assignment specification for the ITD 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 [35];
- Slides of the Software Engineering II course available on WeBeep [36].

7.2. 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 [23].
- **LaTeX**: typesetting system used to produce the final RASD document in a consistent format [20].
- **Git**: version control used to track changes and support collaborative development [37].
- **GitHub**: remote repository hosting and collaboration platform used for versioning, reviews, and issue tracking [16].

7.3. Use of AI Tools

AI tools were used during the project as supporting tools, in the same way as other software adopted in the development process. Their role was not to automatically generate content, but to help improve the clarity, structure, and overall quality of the documentation.

Their use was mainly limited to the writing and revision phases. They were helpful in rephrasing sentences, simplifying long or unclear passages, and checking whether explanations could be misunderstood. In some cases, interacting with an AI assistant also helped clarify ideas before writing the final version of the text.

7.3.1. Tools Used

The AI tools employed during the project were:

- Gemini
- ChatGPT
- Copilot

7.3.2. Typical Prompts

AI tools were queried using prompts such as:

- "Rephrase this description to make it clearer."
- "Does this explanation of the backend deployment process contain any ambiguities?"
- "Suggest alternative wording for this technical paragraph to improve flow."
- "Identify any terms in this section that may be inconsistent with the definitions provided earlier in the document."
- "Format this design description or table using LaTeX".
- "What's the difference between interface and type in TypeScript, and in what specific scenarios is one recommended over the other? How do you achieve type safety?"
- "What is the config.ts file and how do you configure it in the best way?"
- "What's the ideal structure for organizing a Node.js/Express project to clearly separate routes, controllers, and business logic?"
- "What are the best practices for structuring a React Native project?"
- "Explain how mocking works in Jest. What are the best practices?"
- "How does a Dockerfile work, and what are the best practices for writing one for a Node.js application?"

- "How does a Docker Compose file work, and what are the best practices to set up Docker Compose for a multi-service Node.js application?"
- "How to monitor logs of Docker containers effectively?"

7.3.3. 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.
- Technical descriptions needing LaTeX formatting.
- Code snippets or configuration files requiring explanations or best practices.

7.3.4. 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.

7.3.5. 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.
- Explanations of technical concepts and best practices.

7.3.6. 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.

8 | 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
Document Preparation	1 hours	1 hours	1 hours	3 hours
Backend Development	1 hours	1 hours	1 hours	3 hours
Frontend Development	1 hours	1 hours	1 hours	3 hours
Deployment Process	1 hours	1 hours	1 hours	3 hours
Testing	1 hours	1 hours	1 hours	3 hours
Total Hours	**500 hours**	**500 hours**	**500 hours**	**500 hours**

Table 8.1: Time spent on document preparation

Bibliography

[1] Auth0. node-jsonwebtoken — json web token implementation for node.js. URL https://github.com/auth0/node-jsonwebtoken.

[2] Axios Contributors. Axios — promise-based http client for browser and node.js. URL https://axios-http.com/.

[3] Callstack. React native paper — material design components for react native. URL https://reactnativepaper.com/.

[4] Cloudflare, Inc. Cloudflare origin certificates — secure origin connections. URL https://developers.cloudflare.com/ssl/origin-configuration/origin-certificates/.

[5] Docker, Inc. Docker compose — multi-container application orchestration, . URL https://docs.docker.com/compose/.

[6] Docker, Inc. Docker — containerization platform, . URL https://docs.docker.com/.

[7] EchoAPI Team. Echoapi — api development, testing, and mocking platform. URL https://www.echoapi.com/.

[8] Expo. Expo cli — command-line tools for expo projects, . URL https://docs.expo.dev/workflow/expo-cli/.

[9] Expo. Eas cli — build and deploy expo applications, . URL https://docs.expo.dev/eas/.

[10] Expo. Expo — universal react native tooling and libraries, . URL https://docs.expo.dev/.

[11] Expo. Expo router — file-based routing for expo and react native, . URL https://docs.expo.dev/router/introduction/.

[12] Expo. Expo go — run expo apps on physical devices, . URL https://docs.expo.dev/get-started/expo-go/.

[13] Expo. Expo location — access device location, . URL https://docs.expo.dev/versions/latest/sdk/location/.

[14] Expo. Expo securestore — secure key-value storage, . URL https://docs.expo.dev/versions/latest/sdk/securestore/.

[15] Express.js Team. cors — cross-origin resource sharing middleware for express.js. URL https://www.npmjs.com/package/cors.

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

[17] Hapi.js. Joi — object schema description language and validator for javascript objects. URL https://joi.dev/.

[18] Kelektiv. bcrypt — node.js bcrypt library. URL https://github.com/kelektiv/node.bcrypt.js.

[19] Lad.js Contributors. Supertest — http assertions made easy. URL https://github.com/ladjs/supertest.

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

[21] Lucide Contributors. Lucide — beautiful & consistent icon toolkit. URL https://lucide.dev/.

[22] Meta Platforms, Inc. React native — learn once, write anywhere. URL https://reactnative.dev/.

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

[24] NGINX, Inc. Nginx — high-performance http server and reverse proxy. URL https://nginx.org/en/docs/.

[25] npm, Inc. npm — node.js package manager. URL https://docs.npmjs.com/.

[26] OpenJS Foundation. Node.js — javascript runtime built on chrome's v8 engine. URL https://nodejs.org/en.

[27] Pino.js Contributors. Pino — high-performance logging library for node.js, . URL https://github.com/pinojs/pino.

[28] Pino.js Contributors. pino-pretty — pretty printer for pino logs, . URL https://github.com/pinojs/pino-pretty.

[29] pmndrs. Zustand — a small, fast and scalable state-management solution. URL https://zustand.docs.pmnd.rs/.

[30] PostgreSQL Global Development Group. Postgresql — open source relational database. URL https://www.postgresql.org/.

[31] Prisma. Prisma cli — schema and migration management, . URL https://www.prisma.io/docs/reference/api-reference/command-reference.

[32] Prisma. Prisma — next-generation orm for node.js and typescript, . URL https://www.prisma.io/docs.

[33] React Hook Form Contributors. React hook form — performant, flexible form library for react. URL https://react-hook-form.com/.

[34] React Native Maps Contributors. react-native-maps — react native mapview component. URL https://github.com/react-native-maps/react-native-maps.

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

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

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

[38] Zod Contributors. Zod — typescript-first schema validation. URL https://zod.dev/.

List of Tables

2.1 Mapping of path status to numerical scores 7
2.2 Mapping of numerical scores to discrete path statuses 9
2.3 Mapping between BBP Requirements and implemented functionalities . . . 13

8.1 Time spent on document preparation 42