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

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, Abbreviations

This section provides definitions and explanations of the terms, acronyms, and abbreviations used through out the document, making it easier for readers to understand and reference them. The definitions shared between this document and the RASD document are reported in the following list:

1.3.1. Definitions

- **Path:** A route defined either by data collected from users or by paths manually created within BBP. A Bike Path consists of one or more Path Segments.
- **Path Segment:** A portion of a Path defined between two consecutive waypoints. Each segment may have specific attributes.
- **Path Status:** The overall condition of a Bike Path or Path Segment, as determined by the system's aggregation and merging process. Statuses include: *Optimal*, *Medium*, *Sufficient*, *Requires Maintenance*, and *Closed*.
- **Path Score:** The final ranking value computed by the system, based on the path's aggregated status and effectiveness, used to recommend the best alternatives between an origin and a destination.
- **Trip:** A cycling activity tracked through the BBP application. If the user is logged-in, the trip is stored and becomes part of the user's trip history. Each trip includes temporal, spatial, and contextual data (e.g., duration, distance, speed).
- **Report:** A submission of path information by a logged-in user. A report contains details on path status and obstacles and can be either Manually Entered or Automatically Detected (sensor-detected).
- **Freshness:** A metric used when merging reports; newer reports carry more weight than older ones when determining Path Status.
- **Obstacles:** Elements on a path that negatively impact cycling conditions, such as potholes or flooding, as identified by users or automatic sensors.
- **Manual Creation Mode:** The creation mode in which a logged-in user defines a new path by drawing its geometry on the map or selecting the map segments composing it.
- **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 system functionality where the logged-in user manually makes a report, inserting the bike path's condition and obstacle, through the application interface.

- **Automatic Report:** The system functionality where, during an active trip with Automatic Mode enabled, the system analyzes sensor data to detect anomalies (e.g., potholes) and presents them to the user for confirmation before generating a report.

1.3.2. Acronyms

- **API:** Application Programming Interface.
- **APK:** Android Package
- **DBMS:** DataBase Management System.
- **DD:** Design Document.
- **DOM:** Document Object Model.
- **DTO:** Data Transfer Object, represents a link between the user input and a Java Object.
- **HTTP:** HyperText Transfer Protocol.
- **IPA:** iOS App Store Package.
- **JPA:** Java Persistence API.
- **JS:** JavaScript.
- **QR Code:** Quick Response Code.
- **REST:** REpresentational State Transfer (see DD).
- **RASD:** Requirements Analysis and Specification Document.
- **DD:** Design Document.
- **ITD:** Implementation and Test Document.
- **S2B:** Software To Be.
- **UI:** User Interface.
- **URL:** Uniform Resource Locator.
- **UX:** User eXperience.
- **ORM:** Object-Relational Mapping.

- **GPS:** Global Positioning System.
- **JSON:** JavaScript Object Notation.
- **CRUD:** Create, Read, Update, Delete.

1.3.3. Abbreviations

- **BBP:** Best Bike Paths.

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.

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.

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 by drawing the route directly on the map. 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 by the user.

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 the condition of bike paths by combining information collected from user reports and recorded trips. Each path is associated with indicators that describe its current condition, taking into account reported issues, detected obstacles, confirmations provided by multiple users over time and their freshness. This allows the app to keep an updated and reliable view of path quality.

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 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 confirmation 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. Only validated interactions contribute to the reliability of a report. 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 *IGNORED* state do not contribute to either score, while those in the *CREATED* state contribute to the confirmed 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 and freshness. 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
R12	The system shall allow logged-in users to start a cycling trip in manual or automatic mode.	Partial - Only Manual Mode was 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

Continued on next page

Rx	Description	Implemented
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
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

Continued on next page

Rx	Description	Implemented
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
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

Continued on next page

Rx	Description	Implemented
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 App

The BBP mobile application is developed with React Native in the Expo ecosystem, allowing a single codebase to target both iOS and Android while keeping GPS access consistent across devices. Expo also simplifies build and deployment steps, which reduces overhead during development and testing.
Navigation is handled through Expo Router, which offers file-based routing and keeps screen structure easy to maintain as the app grows. Global state is managed with Zustand, selected for its minimal boilerplate and efficient updates, which are important for user session handling.
The user interface relies on React Native Paper to provide a Material Design-compliant component library with built-in theming support. Lucide Icons supplies a lightweight and consistent icon set, while React Native Maps is used for the core map experience, including path visualization and score overlays. Expo Linear Gradient is used in selected screens to provide branded visual accents in the interface.
Location tracking is implemented via Expo Location, which handles permission requests and continuous GPS updates needed for map views and trip-related features.
For data exchange, Axios manages HTTP requests to the backend. Zod defines schemas for client-side input validation (e.g., authentication and profile forms) and integrates with React Hook Form via resolvers to enforce those constraints in the UI. Expo Secure Store persists authentication tokens on device.

3.1.2. Backend

Express.js 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. It has been called the de facto standard server framework for Node.js.[17]
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.

Technologies used in the Backend

The backend relies on the following technologies and libraries:

- **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[1].
- **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[4].

- **bcrypt**: cryptographic hashing library used to securely store user passwords by applying adaptive hashing algorithms, protecting against brute-force and rainbow table attacks[2].
- **jsonwebtoken**: library used to implement stateless authentication through JSON Web Tokens (JWT), enabling secure user session management and authorization across API endpoints[5].
- **cors**: middleware that enables and configures Cross-Origin Resource Sharing, allowing controlled access to backend resources from client applications hosted on different domains[3].
- **Pino**: high-performance logging library designed for Node.js applications, used to record structured logs for monitoring, debugging, and auditing purposes[6].
- **pino-pretty**: development tool used to format Pino logs into a human-readable form, improving log readability during debugging and local development[7].
- **Supertest**: HTTP testing library that allows automated testing of RESTful APIs by simulating HTTP requests and validating responses without requiring a running network server[8].

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 **NestJS** 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 **NestJS** 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. 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.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
|   |-- 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
|   |   |-- openmeteo.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
|-- Dockerfile                          # Dockerfile for backend service
|-- jest.config.mjs                     # Jest configuration
|-- node_modules/                       # Installed dependencies
|-- notes.md                            # Project notes
|-- 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. 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.

How to run unit tests

All tests are executed via Jest. In our repository, the test script is already configured in `package.json`:

```
npm install
npm run test
```

Optionally, tests can be executed in watch mode in order to re-run them automatically upon file changes:

```
npm run test:watch
```

In general (independently from our setup), you can also run Jest directly, or target a single test file:

```
npm test path.manager.test.ts
```

Test coverage

Coverage reports help identify untested branches and edge cases, and they were used as a guidance tool to refine the test suite during development.
Coverage can be collected by running Jest with the `-coverage` flag. In our setup this can be done through NPM as follows:

```
npm run test -- --coverage
```

It is also possible to collect coverage for a specific subset of tests:

```
npm run test -- path.manager.test.ts --coverage
```

The command generates a textual summary in the terminal which can be inspected to identify uncovered lines, functions, and branches.

5.2. 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 integration tests

Integration tests are part of the standard Jest suite, therefore they can be executed with the same commands:

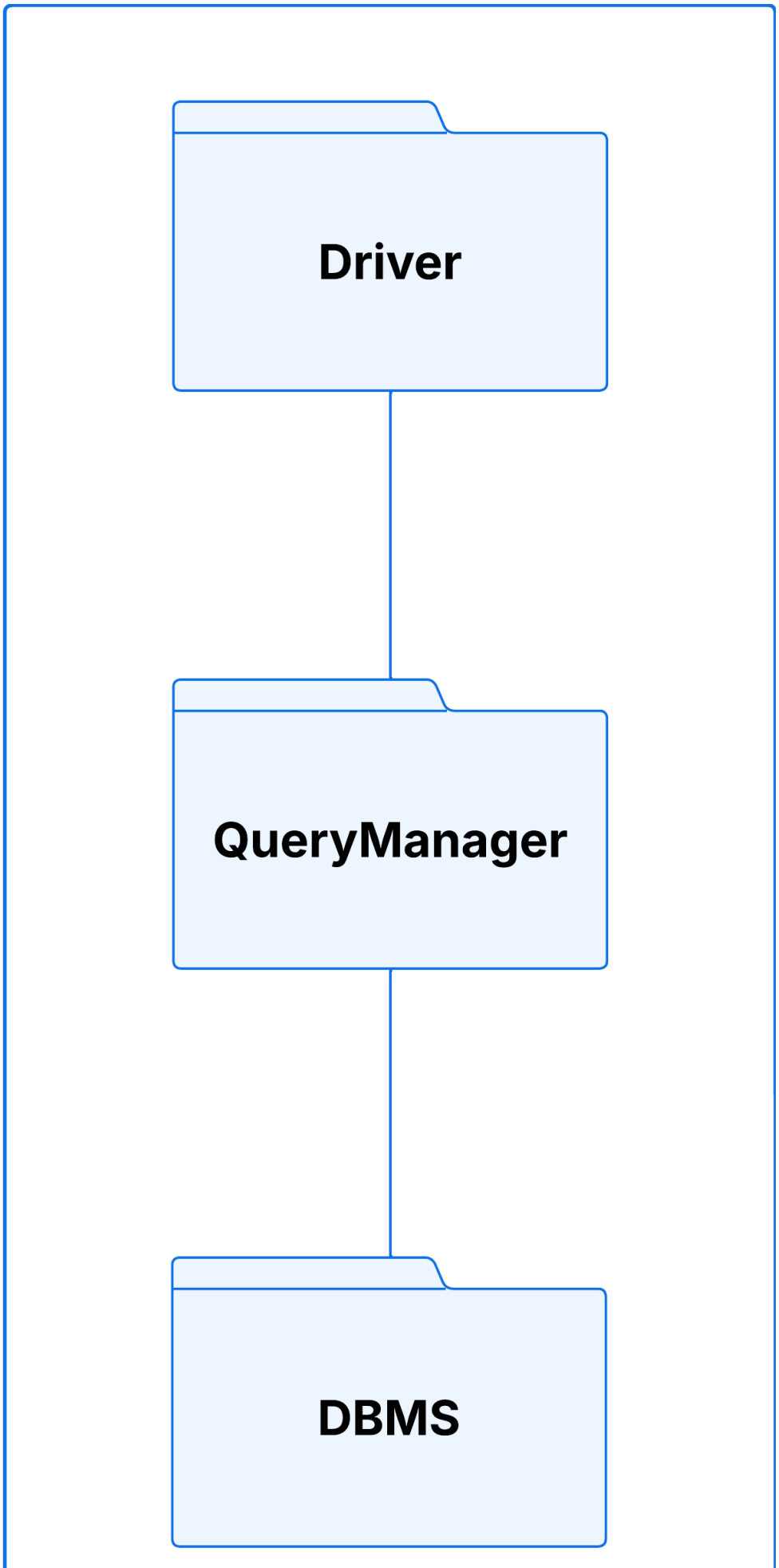
```
npm run test
```

To run only integration tests, a common approach is filtering by filename pattern for example:

```
npm test integration
```

or directly by specifying a single integration test file:

```
npm test user.integration.test.ts
```


Driver
QueryManager
DBMS

6 Installation Instructions

6.1. Prerequisites

Node npm

6.2. Backend Setup

Is running on a Personal Server managed via Docker. If you want to run it locally, you need to have PostgreSQL installed and running. Then, clone the repository, install the dependencies with npm install, set up the .env file with the necessary environment variables, and run the migrations with Prisma.

6.3. Frontend Setup

You can build the app, run it on a simulator or physical device using Expo CLI. You can install the apk on Android devices directly.
If you want to use a local server you should change the API URL in the .env file and build it or run it on the emulator, since the built app points to the production server.

7 References

7.1. Reference Documents

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

- IEEE Standard for Software Requirement Specifications [10];
- 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 [14];
- Slides of the Software Engineering II course available on WeBeep [15].

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 [13].
- **LaTeX**: typesetting system used to produce the final RASD document in a consistent format [11].
- **Git**: version control used to track changes and support collaborative development [16].
- **GitHub**: remote repository hosting and collaboration platform used for versioning, reviews, and issue tracking [9].
- **Lucidchart**: creation of UML diagrams (use case diagrams, state diagrams, domain class diagram) [12].

7.3. Use of AI Tools

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

7.3.1. Tools Used

The AI tools employed during the project were:

- Gemini
- ChatGPT

7.3.2. Typical Prompts

AI tools were queried using prompts such as:

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

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.

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.

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
Introduction	4 hours	4 hours	5 hours	13 hours
Overall Description	11 hours	7 hours	10 hours	28 hours
Specific Requirements	19 hours	8 hours	12 hours	39 hours
Formal Analysis	7 hours	21 hours	11 hours	39 hours
Final Review & Editing	3 hours	3 hours	3 hours	9 hours
Total Hours	**44 hours**	**43 hours**	**41 hours**	**128 hours**

Table 8.1: Time spent on document preparation

Bibliography

[1] Node.js — the v8 javascript engine, . URL https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine.

[2] Bcrypt - crypt library, . URL https://www.npmjs.com/package/bcrypt.

[3] Cors - middleware for express.js, . URL https://www.npmjs.com/package/cors.

[4] Joi - object schema description language and validator for javascript objects, . URL https://joi.dev/.

[5] jsonwebtoken - authentication library, . URL https://www.npmjs.com/package/jsonwebtoken.

[6] Pino - logging library, . URL https://github.com/pinojs/pino.git.

[7] pino-pretty - tool to pretty print logs generated by pino, . URL https://github.com/pinojs/pino-pretty.git.

[8] Supertest - http assertions made easy via superagent, . URL https://www.npmjs.com/package/supertest.

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

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

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

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

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

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

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

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

[17] Wikipedia contributors. Express.js — Wikipedia, the free encyclopedia, 2026. URL https://en.wikipedia.org/w/index.php?title=Express.js&oldid=1331088521. [Online; accessed 17-January-2026].

List of Figures

List of Tables