POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Design Document

Best Bike Paths

Authors: **Matteo Caldognetto – Claudia Salone**

Student IDs: 11097854 – 10827565
Academic Year: 2025-26

Contents

1 | Introduction

A. Purpose

Best Bike Paths is an innovative platform designed to offer an intuitive and engaging user experience for cyclists of all levels. The app's design aims to create an accessible and easy-to-navigate environment, allowing cyclists to record their routes, view detailed maps, and quickly access personalized statistics in a clear manner.

The interface is crafted to ensure smooth use on mobile devices, integrating features that facilitate access even for less technological-oriented users, thanks to layout choices, colors, and interactions aimed at enhancing usability.

The system is designed to encourage collaborative and sustainable activity by users, fostering a dynamic and involved community. The design emphasizes the usability of its features, maintaining transparency and control in operations, while allowing everyone to use the app simply and intuitively.

B. Scope

Best Bike Paths aims to provide a digital platform that facilitates smooth interaction among cyclists through an architectural design that supports the recording and sharing of data on cycling routes. The app is structured to be modular and scalable, managing both manually entered data and data automatically acquired via mobile sensors, ensuring information reliability through user confirmation or correction.

The application is designed to offer an intuitive and accessible interface that allows all users to view interactive maps with suggested cycling routes ranked by scores and reviews, thereby enhancing route navigation and trip planning. The design also integrates features for managing dynamic data such as detailed travel statistics and up-to-date weather information.

The goal is to create a system that guarantees usability, safety, accessibility, and ease of maintenance, promoting sustainable mobility and active community participation. Particular attention is given to user experience, efficient data management, and the adoption of accessibility standards to reach a broad and diverse audience.

C. Definitions, Acronyms, Abbreviations

C.1. Definitions

Trip: Represents a recorded bike trip with associated routes and metadata.

TripRoute: A segment of a trip corresponding to a specific street or path section.

Path: A published, community-accessible bike path that can be discovered by all users.

Street: A real-world street segment with geometry and condition status.

PathReport: A crowdsourced condition report submitted by users for trip routes or streets.

ObstacleReport: A report of physical obstacles (potholes, debris, construction, etc.) encountered during trips.

PathSegment: A link between a path and a street, defining the ordered composition of paths from streets.

TripRating: User's rating (1-5) of their trip experience, required before publishing.

Manual Mode: bikers insert the data manually, specifying the name of the streets in the path and their status.

Automated Mode: bikers let BBP acquire data from their mobile devices while they bike. BBP should guess the user is biking given their speed; it should collect GPS information to reconstruct the followed path, and, at the same time, it should acquire data from the mobile device's accelerometer and gyroscope to keep track of any significant movement of the device itself that suggests the presence of potholes or other problems. Since there is the possibility of having false positives (e.g., non-existent potholes), the user will have to confirm or correct the information acquired by BBP before this is made available to the community.

C.2. Acronyms

BBP: Best Bike Paths – the name of the platform.

GPS: Global Positioning System – is one of the global navigation satellite systems (GNSS) that provide geolocation and time information to a GPS receiver anywhere on or near the Earth where signal quality permits [7].

C.3. Abbreviations

G: Goal.

WP: World Phenomena.

SP: Shared Phenomena.

R: Requirements.

D. Revision history

- Version 1.0: 07/01/2026

E. Reference Documents

[1] M Rossi M Camilli E Di Nitto. *Dynamic Analysis, Testing*. Presentazione in slide. Slide della lezione, Corso di Software Engineering II, Politecnico di Milano.

[2] M Rossi M Camilli E Di Nitto. *Introduction to Architectural Styles*. Presentazione in slide. Slide della lezione, Corso di Software Engineering II, Politecnico di Milano.

[3] M Rossi M Camilli E Di Nitto. *Introduction to Software Design*. Presentazione in slide. Slide della lezione, Corso di Software Engineering II, Politecnico di Milano.

[4] M Rossi M Camilli E Di Nitto. *Software Qualities and Software Architecture*. Presentazione in slide. Slide della lezione, Corso di Software Engineering II, Politecnico di Milano.

[5] M Rossi M Camilli E Di Nitto. *Structure of a Design Document*. Presentazione in slide. Slide della lezione, Corso di Software Engineering II, Politecnico di Milano.

[6] M Rossi M Camilli E Di Nitto. *VV terminology, Static vs Dynamic, Analysis Data Flow Analysis*. Presentazione in slide. Slide della lezione, Corso di Software Engineering II, Politecnico di Milano.

[7] Wikipedia contributors. *Global Positioning System — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Global_Positioning_System&oldid=1318018964`. [Online; accessed 4-November-2025]. 2025.

F. Document Structure

Section 1: Introduction

This section provides a general overview of the project, explaining the motivations and goals to be achieved, as well as offering a summary of the approach chosen for developing the platform.

Section 2: Architectural Design

This part describes the architectural structure of the platform at various levels of detail, presenting the main components, their interactions, and the design choices adopted.

Section 3: User Interface Design

This section presents the design of the user interfaces, with related diagrams and wireframes illustrating navigation and the organization of features.

Section 4: Requirements Traceability

This section explains how the objectives and requirements set are reflected in the system, linking functionalities with the design and operational components.

Section 5: Implementation, Integration and Test Plan

Details concerning the implementation, integration, and testing phases are provided, with information useful for those involved in the platform's development and management.

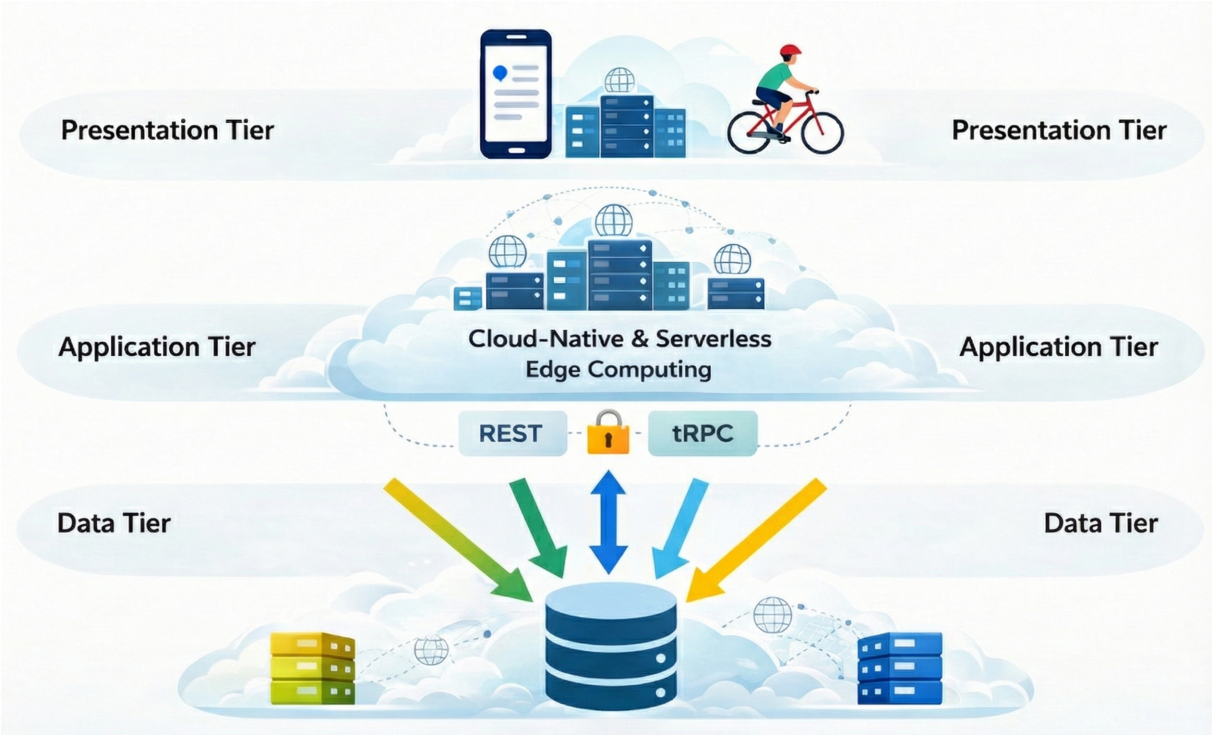
Section 6: Effort Spent

The overall effort dedicated to the creation of the document and related activities is reported.

2 | Architectural Design

A. Overview

This section provides a high-level description of the overall architecture of Best Bike Paths, illustrating the structural components and the interaction patterns that characterize the system. The platform shifts away from traditional monolithic server paradigms to adopt a modern Cloud-Native and Serverless architecture. While logically organized into distinct tiers (Presentation, Application, and Data), the system leverages Edge Computing infrastructure to execute business logic globally, ensuring low latency, automatic scalability, and high availability without manual server management.



A defining architectural choice is the adoption of strongly typed Remote Procedure Calls (RPC) for client-server communication, implemented via tRPC. This approach establishes a strict, schema-based contract between the frontend and the backend. By sharing type definitions across the entire stack, the architecture ensures rigorous interface consistency, robust data validation, and a streamlined development workflow.

This design guarantees modularity and maintainability while supporting advanced client-side capabilities, such as sensor simulation and local state management, effectively distributing the computational load between the user device and the edge infrastructure.future new features.

Presentation Tier

The user interface is not merely a passive display layer but is implemented as a "Smart Client" using a Single Page Application approach built with React. This component acts as a comprehensive runtime environment within the user's browser, taking on responsibilities that go far beyond simple visualization. Specifically, the presentation tier is designed to handle complex logic such as the local simulation of hardware sensors, generating synthetic GPS and accelerometer telemetry to demonstrate automated reporting features directly on desktop environments.

Furthermore, the client actively manages local state and performs rigorous preliminary data validation using strict schemas shared with the backend. This active role significantly reduces the need for constant server round-trips and ensures that the user experience remains fluid and responsive even during complex interactions.

Application Tier

The backend architecture represents a significant departure from traditional monolithic application servers, as it is hosted entirely on a global Edge Computing infrastructure provided by Cloudflare Workers. This serverless design allows the business logic to be executed physically closer to the user, drastically reducing latency and ensuring automatic scalability during high-traffic periods without the need for manual infrastructure management.

The core logic is organized into distinct functional modules exposed via tRPC routers, which implement a Service Layer pattern to orchestrate complex operations such as route calculation, authentication verification, and the aggregation of community data. This tier also acts as a secure gateway that mediates all interactions with external third-party services, such as OSRM for routing and weather APIs, encapsulating API keys and connection details to protect sensitive credentials from being exposed to the client.

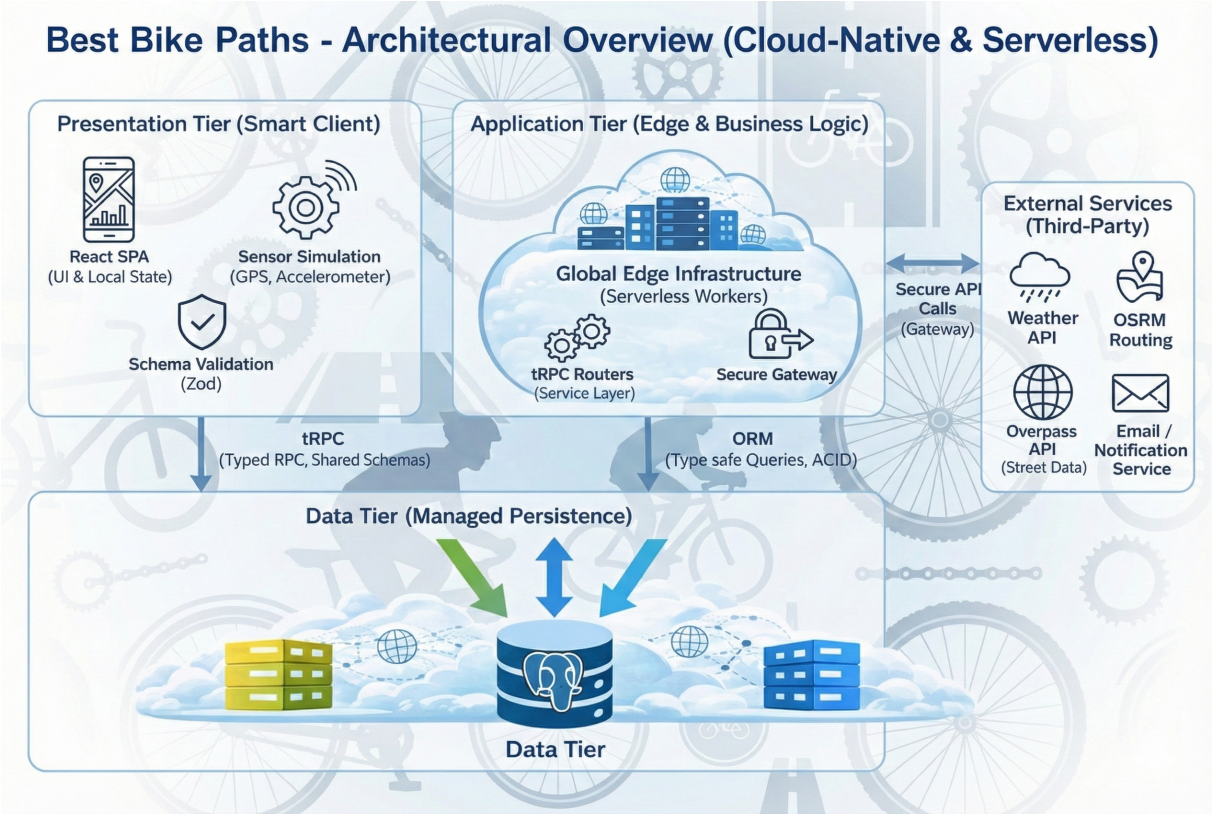
Data Tier

Persistent storage employs a relational database managing structured data through schema-defined tables and relationships. The data model reflects domain entities with precision: trips capture route geometries and performance metrics; streets represent cyclable segments with condition status; paths aggregate streets into published routes discoverable by the community.

User-generated content flows through a validation pipeline where manual reports and automated sensor detections converge into aggregated condition scores, influencing path rankings in search results. Data integrity mechanisms ensure consistency across related entities. Publishing a trip as a community path triggers the creation of path segment records linking the path to constituent streets, maintaining referential relationships.

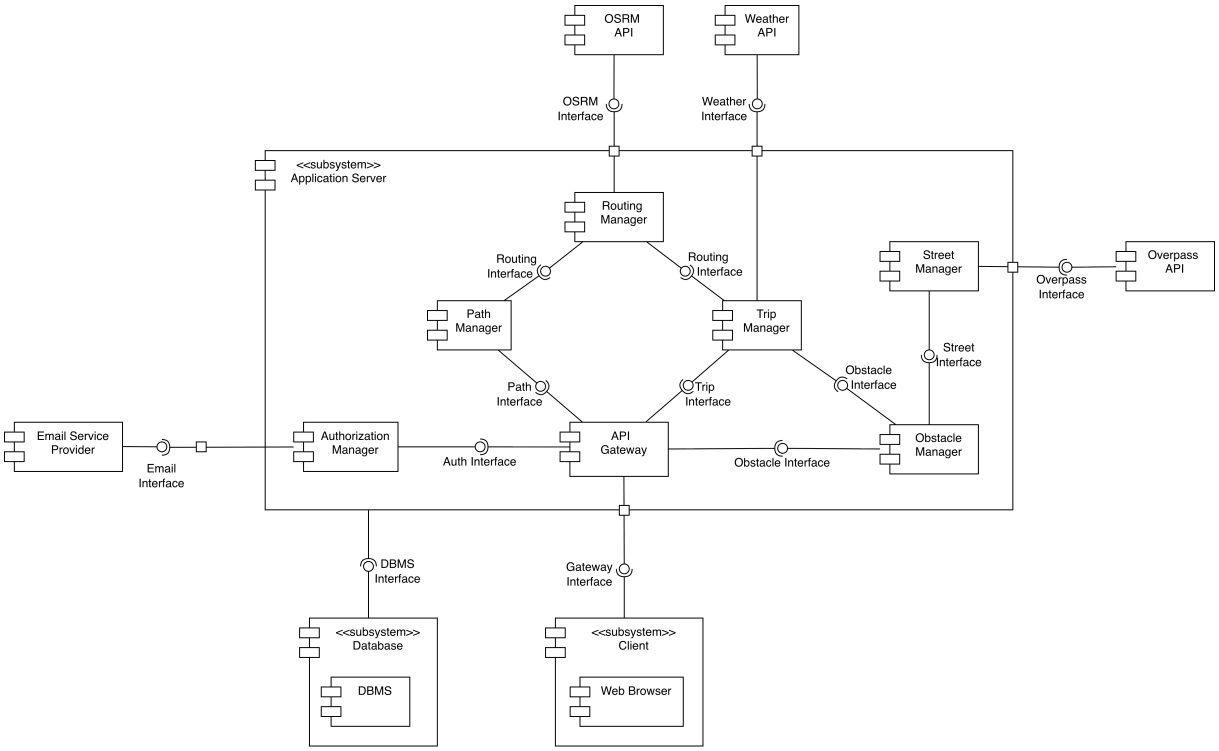
Deletion operations cascade through dependent records, preventing orphaned data. The

schema accommodates both structured relational data for core entities and semi-structured data for variable-content fields like sensor readings and weather conditions, stored as JSON within appropriate columns. The database design supports concurrent operations through transaction isolation and optimistic locking strategies. Read-heavy operations for path discovery leverage indexing on frequently queried fields, while write operations for trip recording and reporting maintain ACID guarantees. This foundation enables reliable service delivery under varying load conditions, from individual users recording trips to community-wide path browsing during peak usage periods.



All client-server communication is handled through typed RPCs targeting the Application Tier, which mediates all access to the Data Tier through an ORM-based persistence layer.

B. Component view



The following sections describe the main components of the system, organized into Back-end, Front-end, and External Services.

B.1. Back-end components

The back-end is hosted within an application server that acts as the entry point and manages the routing of requests toward specific services.

API Gateway

This component acts as the single entry point for all client requests. It intercepts HTTP calls, handles the request context (including session validation), and routes the call to the appropriate Router (Service).

Authentication Management Service

Responsible for handling user identities, this component integrates an external library to manage authentication flows (login, logout, session handling) and maintain active sessions. It provides the application context with information about the currently authenticated user in order to authorize protected operations.

Trip Management Service

This service (corresponding to TripRouter) manages the lifecycle of users' personal trips. Its responsibilities include creating and saving new trips (recording GPS tracks), managing

trip simulations, editing, deleting, and retrieving details of past trips and verifying whether a trip has already been published as a public path.

Path Management Service

Handles the inventory of public cycling paths. It allows users to search for existing paths, view their details, and convert personal trips (Trip) into public cycling paths (Path). It also supports manual creation of paths by defining a sequence of streets.

Routing Service

Provides route calculation functionalities. It interacts with the external routing engine to compute the optimal route between two coordinates or across a series of waypoints (streets or addresses). It includes the logic to validate coordinates and compute geospatial distances.

Street Service

This component aggregates the functionalities for managing reports (PathReportRouter and StreetRouter). It collects manual or sensor-based road condition reports, associates reports with specific streets or path segments and allows users to view and manage their own submitted reports.

External Integration Services (Weather and Geocoding)

A set of utility modules providing adapters to third-party APIs:

- **Weather Service:** Enriches trip data with weather information retrieved according to timestamp and coordinates.
- **Geocoding Service:** Manages translation between textual addresses and geographic coordinates, implementing caching logic to optimize performance.

Database Management Service (DBMS)

Represents the data persistence layer. The system uses a relational database (PostgreSQL) accessed through an Object-Relational Mapper (ORM). It manages structured entities and complex geospatial data (GeoJSON) for route and position storage.

B.2. Front-end Components

The client is a responsive web application (React) designed to deliver a smooth user experience similar to a native app.

Web App and Router

Main container that handles the application lifecycle and client-side navigation. It manages the page structure, dynamic routing, and protection barriers for user-restricted areas.

State Management and Caching Layer

A cross-cutting layer for data management:

- Server State: Manages cached data coming from APIs to optimize performance and reduce network calls.
- Client State: Manages local and volatile UI state.

Map Visualizer

Core graphical component responsible for geospatial rendering. It displays saved paths (GeoJSON), selected streets, and points of interest. It offers interactive functionalities such as zooming, panning, and coordinate selection on the map to define origins and destinations.

Trip Recorder

Module managing trip acquisition logic. In Automated mode, since the app runs in a browser, this component software simulates the behavior of physical sensors (GPS, accelerometer), generating synthetic GPX tracks and sending simulated telemetry data to the back-end. It displays real-time statistics of the ongoing trip.

Path Reporting Builder

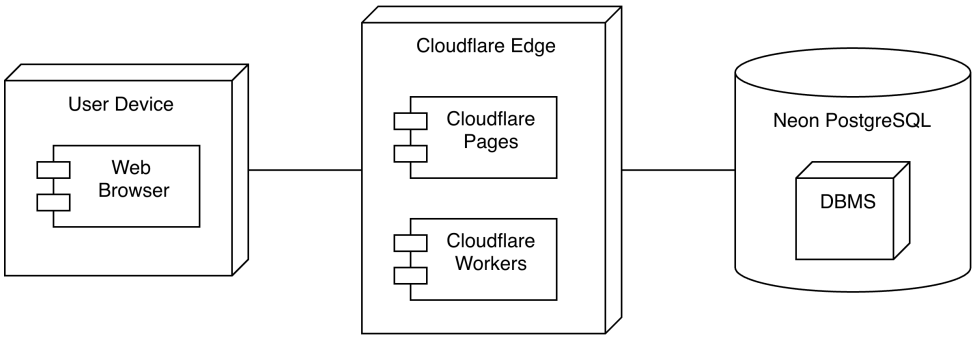
Interface dedicated to manual data entry (ManualPathBuilder). It allows the user to compose a path by selecting a sequence of streets or map points and completing detailed forms to report road surface conditions or the presence of obstacles.

Route Discovery

Search and navigation interface (RouteFinder). It enables users to input search criteria (origin/destination), view route options computed by the server ordered by Score, and consult details of available public paths.

C. Deployment view

This chapter presents the Deployment View of the Best Bike Paths platform, which is graphically represented in the figure below. This view aims to describe the execution environment of the system, detailing the physical and logical distribution of the hardware components and cloud services that support the application software. Unlike traditional architectures based on centralized monolithic servers, BBP adopts a modern Cloud-Native and Serverless architectural approach. This design choice allows the management of the underlying infrastructure to be delegated to specialized providers, while ensuring high availability, automatic scalability, and reduced global latency through the use of Edge Computing technologies.



The following section provides additional details concerning the elements shown in the deployment diagram, analyzing the role of each node and the interactions that occur among them.

User Device

This node represents the physical terminal used by the end user to interact with the system, which may be a personal computer, a smartphone, or a tablet. The fundamental software requirement for this node is the presence of a modern web browser. When a user accesses the platform's URL, the device does not simply display static pages but downloads and executes the entire Single Page Application (SPA) developed in React. Consequently, the browser acts as a true runtime environment for the presentation logic: it handles the rendering of the user interface, local state management, and preliminary form validation. In the specific context of the BBP demo, this node also plays a critical role in sensor simulation: the JavaScript code running on the client simulates the behavior of GPS and accelerometer sensors (normally present on mobile hardware) to generate telemetry data required for the "Automated Reporting" functionality. All outgoing communications from this node to the system occur over the secure HTTPS protocol.

Cloudflare Edge

This node constitutes the core of the backend infrastructure. Instead of residing on a single physical server located in a specific geographic area, the application is deployed on Cloudflare's global Edge Network. Two logical subcomponents can be identified within this environment:

- Component Pages that acts as a Content Delivery Network (CDN). Its purpose is to host the static assets of the application (HTML files, CSS stylesheets, and compiled JavaScript bundles) and serve them to users from the geographically closest server, drastically reducing initial loading times.
- Component Workers that hosts the actual API Server. Here, the Hono framework and the tRPC routers responsible for the business logic are executed. Whenever the user's device sends a data request (for example, to save a route or perform login), it is intercepted and processed by an ephemeral "serverless" instance. This approach ensures that computing resources are dynamically allocated only when needed, handling traffic spikes without manual intervention.

PostgreSQL (Database)

This node represents the data persistence layer (Data Tier). To guarantee the integrity and security of information, the database is decoupled from the application layer and hosted on a managed cloud service. It is a PostgreSQL instance optimized for the cloud, responsible for storing all domain entities: user profiles, authentication sessions, road condition reports, and, most importantly, complex geospatial data (GeoJSON) related to cycling routes. The application running on Cloudflare Edge communicates with this database node via a secure SSL-encrypted TCP connection, using a connection pool optimized for serverless environments. This separation ensures that data remains persistent and protected regardless of the lifecycle of the application processes.

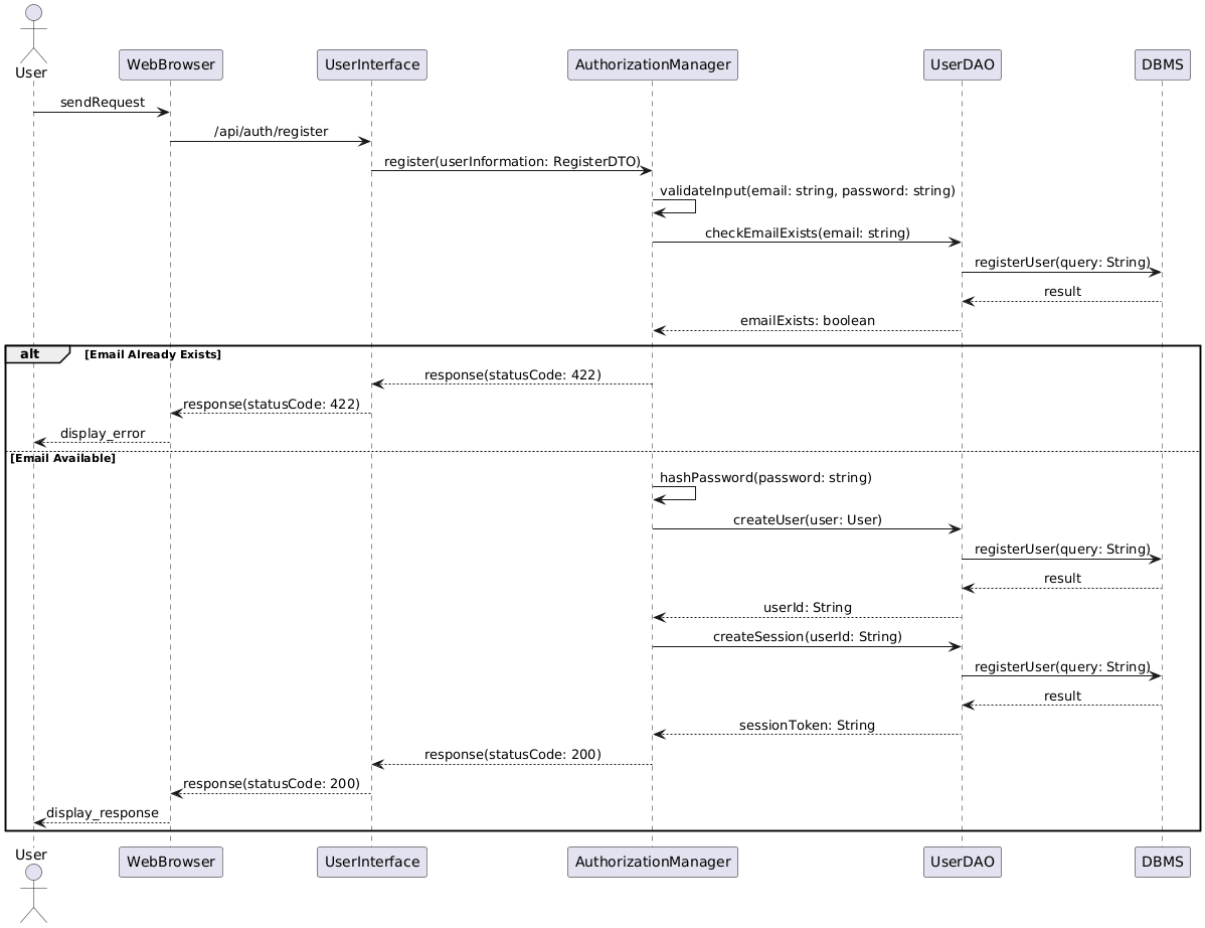
External Services

The BBP system does not operate in isolation but integrates with an ecosystem of third-party services to provide advanced functionality without the need to reimplement complex solutions from scratch.

- OSRM (Open Source Routing Machine): The public routing engine queried by the backend to compute optimal cycling routes between two points and to perform the map-matching of raw GPS traces.
- Overpass: A service based on OpenStreetMap used for geocoding operations, i.e., translating textual addresses into geographic coordinates and vice versa.
- OpenWeather: An external weather API consulted to enrich saved trip data with the atmospheric conditions (temperature, wind, and weather) recorded at the time of the activity. It is important to note that, for security and architectural reasons, it is always the Cloudflare Edge node that interacts with these external services, acting as an intermediary for the client. This pattern hides the complexity of third-party APIs from the user's device and protects any private access keys.

D. Runtime View

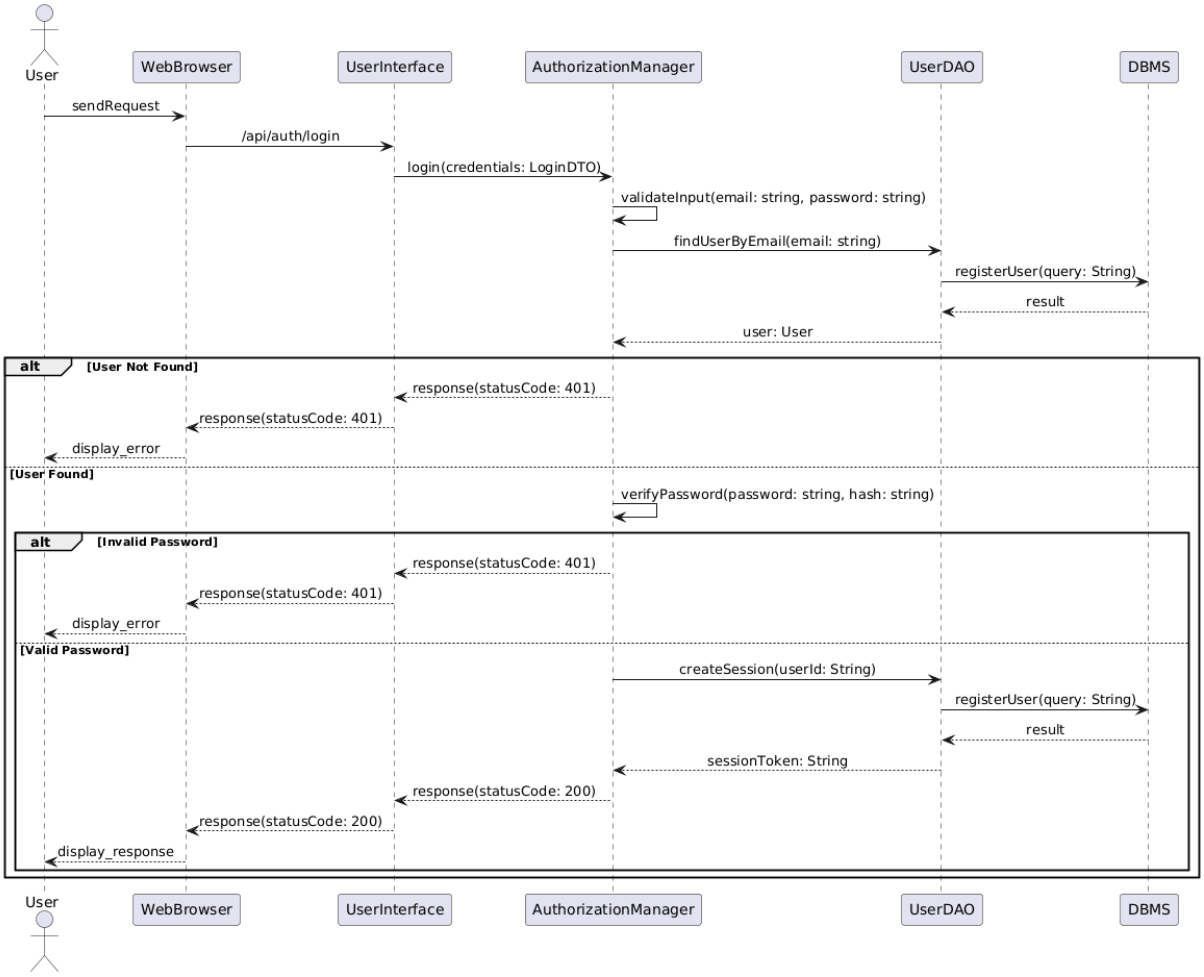
Signup Cyclist



User submits the registration form in the browser, which is sent to '/api/auth/register' and forwarded to the Authorization Manager. The manager validates the input and checks via 'UserDAO.checkEmailExists(email)' whether the email is already in the database.

If the email exists, it returns an error response 422. If not, it hashes the password, creates the user via 'UserDAO.createUser(...)', creates a session via 'createSession(userId)', and returns success 200 with a session token so the user is registered and logged in.

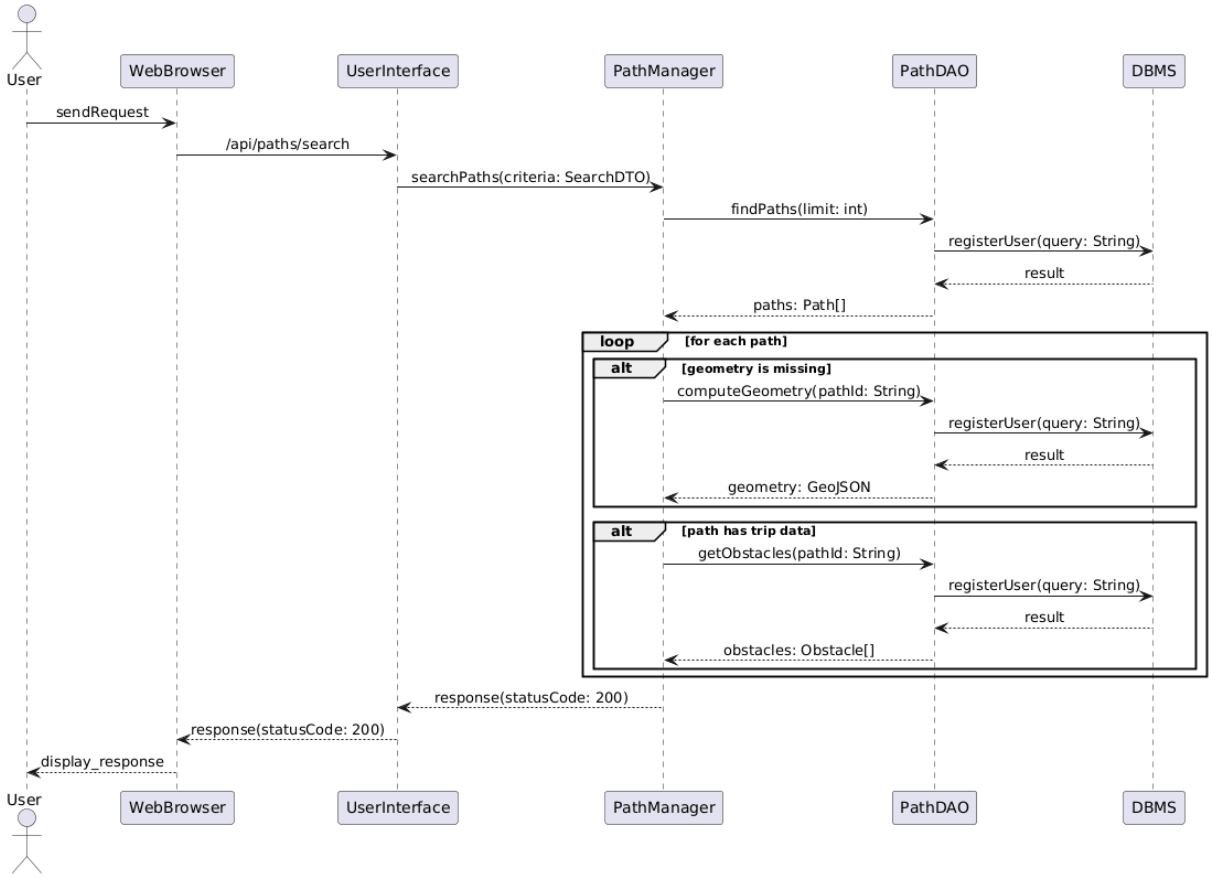
Signin Cyclist



User submits login credentials to '/api/auth/login', which the User Interface forwards to the Authorization Manager. The manager validates the input, fetches the user by email via 'UserDAO.findUserByEmail(email)', and returns 401 if the user doesn't exist.

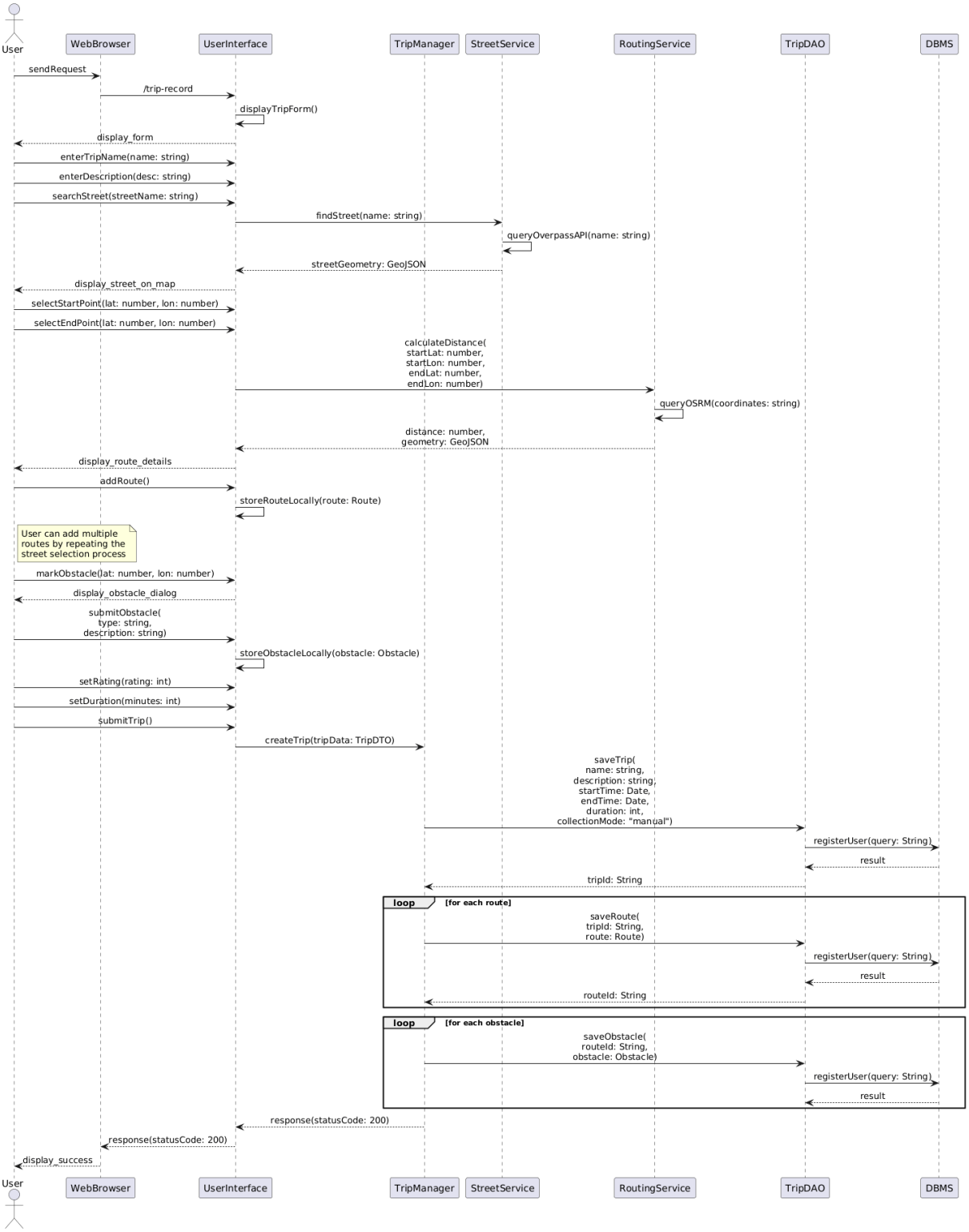
If the user exists, it verifies the password against the stored hash; on failure it returns 401. On success, it creates a session via 'createSession(userId)' (stored by the DAO), returns 200 with a session token, and the user is logged in.

Paths Lookup



The browser calls '/api/paths/search', and the User Interface forwards the request to the Path Manager, which asks the PathDAO for the most relevant paths from the database. For each returned path, the Path Manager fills in missing route geometry by computing it from the underlying street segments and, when trip data is present, also loads any recorded obstacles for that path. It then returns the completed path collection with status code 200 (OK), which indicates the request succeeded, and the UI renders the paths and condition information on the interactive map.

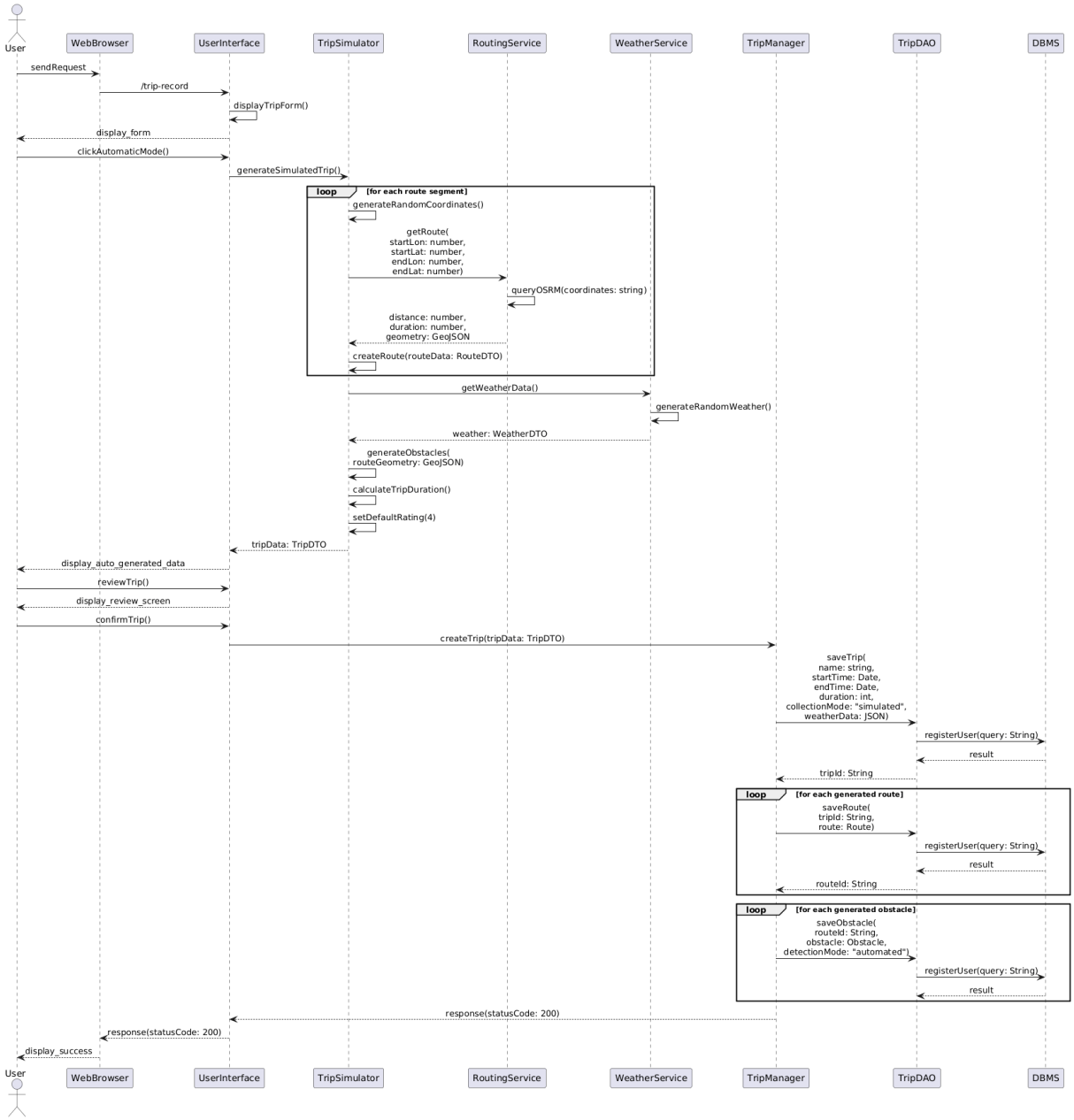
Record manually a trip



On the trip recording page, the user enters basic trip details and searches for a street, and the Street Service uses the Overpass API to fetch the selected OpenStreetMap street geometry so it can be shown on the map. After the user clicks start and end points

on that geometry, the UI calls the Routing Service, which queries OSRM to compute a real cycling route between the two coordinates and returns route distance and geometry rather than a straight-line estimate. The user can repeat this to build a multi-segment trip, optionally mark obstacles on the map, then submit the trip so the backend stores the trip, its routes, and its obstacles in the database and returns a success response.

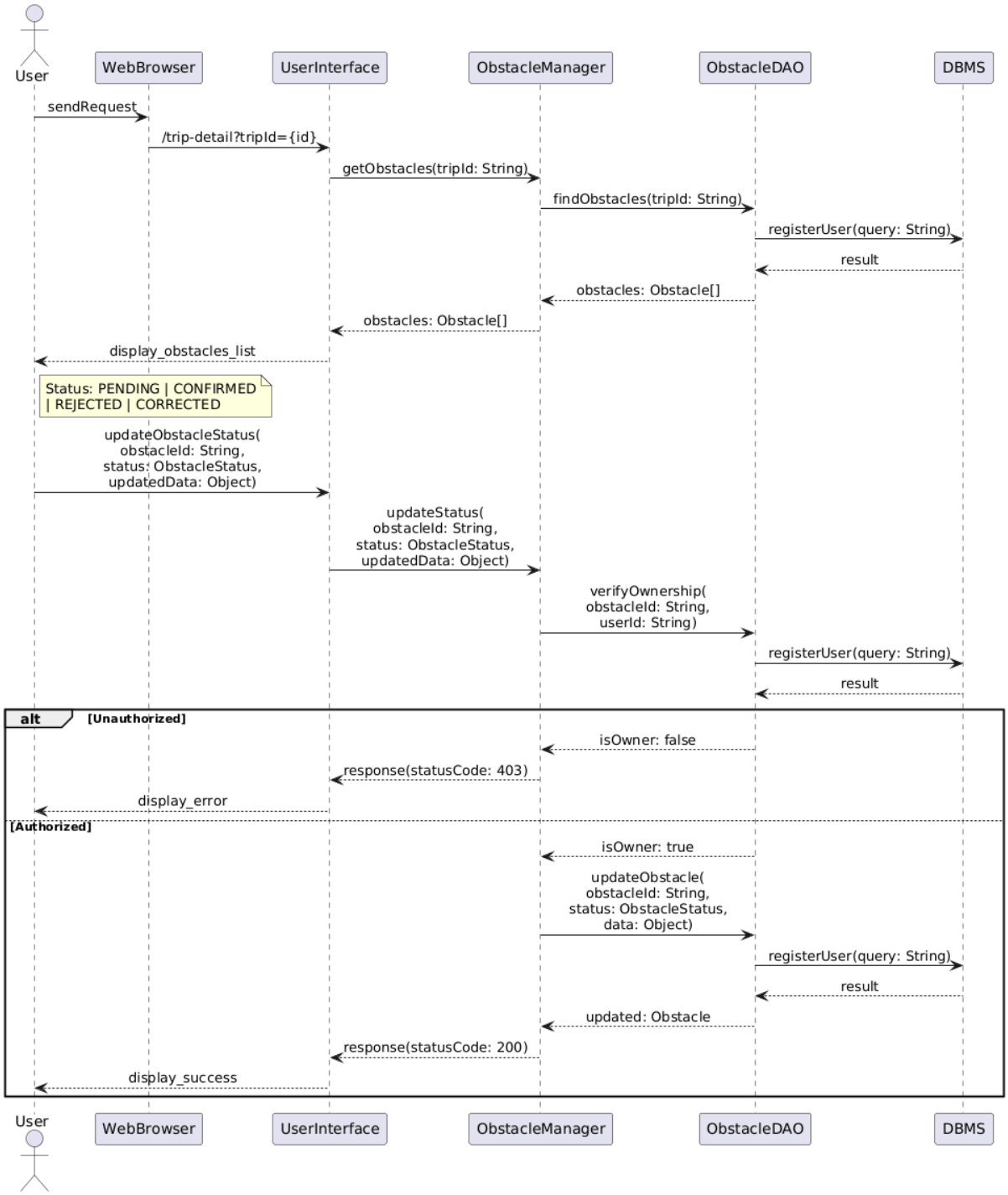
Record automatically a trip



The user opens the trip recording page, switches on Automatic Mode, and the User Interface hands trip generation to a Trip Simulator that creates a simulated ride by generating multiple route segments from random coordinate pairs. For each segment, the simulator calls the Routing Service, which queries OSRM to return realistic routing results such as route distance, duration, and route geometry, so the simulated trip still follows the street network. The simulator then generates plausible weather values, places obstacles by sampling points along the computed route geometry, totals the trip duration by summing segment durations, and sets a default rating before showing everything in a summary modal for confirmation. After the user confirms ("Keep These Routes"), the Trip Manager persists the trip as "simulated" (including weather JSON), saves each route

and obstacle, marks obstacles as “automated,” and returns a success response to notify the user that the simulated trip was recorded.

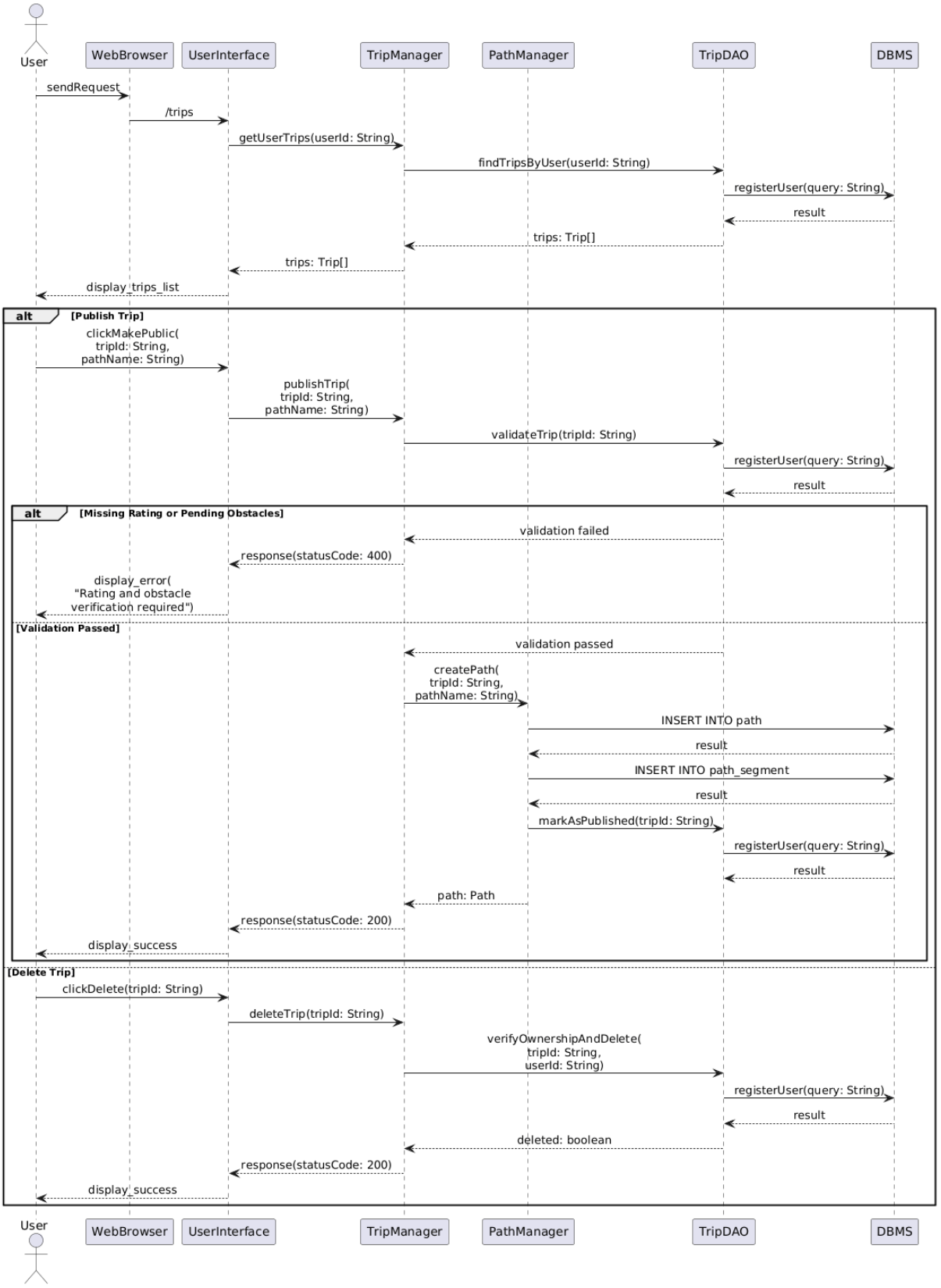
Verify detected obstacles



The user navigates to the trip detail page where the User Interface requests all obstacles associated with the trip from the Obstacle Manager. The system retrieves obstacles from the database and displays them with their current status: PENDING (awaiting verification), CONFIRMED (verified), REJECTED (false positive), or CORRECTED (updated details). When the user updates an obstacle status—whether confirming its

existence, rejecting it as a false positive, or correcting its details—the system follows a unified workflow. The Obstacle Manager first verifies that the authenticated user owns the obstacle by querying the database. If ownership verification fails, a 403 Unauthorized error is returned. If successful, the system updates the obstacle record with the new status and any associated data modifications (such as corrected coordinates or descriptions), then returns a success response.

Paths Browsing



The user navigates to their personal paths area where the User Interface requests all trips belonging to the authenticated user from the Trip Manager. The system queries the database and displays the user's trip collection with statistics. When publishing a trip to the community, the user provides a path name and initiates the publication. The Trip Manager validates the trip by checking that a rating exists and all obstacles have been verified. If validation fails due to missing rating or pending obstacles, a 400 Bad Request error is returned prompting the user to complete these requirements. If validation passes, the Path Manager creates a public path record in the database, links it to the trip's street segments through path_segment records, and marks the trip as published. For trip deletion, the system verifies user ownership and executes the delete operation, which cascades to remove all associated routes and obstacles through database foreign key constraints.

E. Component Interfaces

E.1. HTTP endpoints

These are direct HTTP routes accessible via standard HTTP methods:

System Information

- GET /api - Returns API metadata and available endpoints
- GET /health - Health check endpoint returning system status

OSRM Routing Proxy

- GET /api/osrm/route/:profile/:coordinates - Proxies routing requests to Open Source Routing Machine

Authentication (Better Auth)

- GET /api/auth/* - Authentication endpoints (login, signup, session management)
- POST /api/auth/* - Authentication mutations (handled by Better Auth library)

Authentication (Better Auth)

- GET /api/auth/* - Authentication endpoints (login, signup, session management)
- POST /api/auth/* - Authentication mutations (handled by Better Auth library)

E.2. tRPC API Procedures

All tRPC procedures are accessible via /api/trpc endpoint with batching support enabled.

User Router (user.*) User profile and account management.

- get_me() - Returns current authenticated user profile
- put_updateProfile(name?) - Updates user profile information

Trip Router (trips.*) Bike trip recording, route management, and obstacle reporting.

Trip Management

- post_create(name, description?, startTime, endTime, collectionMode, distance?, avgSpeed?, duration?, weatherData?) - Creates a new trip
- get_list() - Lists all trips for the current user, ordered by creation date
- get_detail(tripId) - Gets trip details with all routes ordered by routeIndex
- get_isPublished(tripId) - Checks if a trip has been published as a community path
- put_updateTripStats(tripId, distance?, avgSpeed?, duration?) - Updates trip statistics from route data
- post_delete(tripId) - Deletes a trip and all associated routes, obstacles, and path records

Route Management

- post_addRoute(tripId, name, geometry, distance, startLat, startLon, endLat, endLon) - Adds a route segment to a trip
- post_removeRoute(tripId, routeIndex) - Removes a route and reindexes remaining routes
- post_reorderRoutes(tripId, newOrder) - Reorders routes within a trip
- post_simulate(tripId, numRoutes?) - Generates simulated routes for testing (default: 5 routes)

Rating System

- post_addRating(tripId, rating, notes?) - Adds or updates trip rating (1-5 scale, required before publishing per RASD)
- get_getRating(tripId) - Retrieves trip rating if exists

Obstacle Reporting

- post_addObstacle(tripRouteId, type, description?, lat, lon, detectionMode, sensorData?, status?) - Reports an obstacle on a trip route
- get_getObstacles(tripId) - Lists all obstacles for a trip with route context
- put_updateObstacleStatus(obstacleId, status, description?, lat?, lon?) - Updates obstacle lifecycle status (confirm/reject/correct per RASD)

Publishing

- post_publish(tripId, pathName, pathDescription?) - Publishes trip as a public community path (requires rating, per RASD)

Path Router (path.*) Community path discovery, creation, and reporting.

Path Discovery

- get_search(query?, limit?) - Searches for paths by name, ordered by score (public, limit: 1-50, default: 20)
- get_getDetails(id) - Gets path details with constituent streets and obstacles (public)
- get_getPathStreets(pathId) - Lists streets composing a path in order (public)

Path Creation

- post_upsertPath(name, description?, geometry) - Creates or retrieves path by geometry (protected)
- post_createManualPath(name, description?, streetIds, geometry?) - Creates path from ordered street list (geometry computed from streets)
- post_publishTripAsPath(tripId, pathName?) - Converts private trip to public path with street linking
- Path Reporting
- post_submitReport(pathId, rating?, streetReports, obstacles?) - Submits condition report for path streets
- streetReports: Array of streetId, status
- obstacles: Array of type, description, lat, lon

Path Report Router (pathReport.*) Crowdsourced condition reporting for trip routes and streets.

Report Management

- post_create(tripRouteId?, streetName?, lat?, lon?, status, obstacles?, isPublishable?, collectionMode?, rating?) - Creates new report (trip-based or standalone street report)
- get_list(status?, limit?, offset?) - Lists published reports with pagination (public, limit: 1-100, default: 50)
- get_detail(reportId) - Gets report detail with route, trip, and user context (public, only publishable reports)
- put_update(reportId, status?, obstacles?, isPublishable?, isConfirmed?) - Updates report properties
- post_delete(reportId) - Deletes user's own report
- get_myReports() - Lists current user's reports including drafts

Street Router (street.*) Street discovery, search, and condition reporting.

Street Discovery

- get_getDetails(id) - Gets street details with recent reports (last 30 days, public)
- get_list(cyclableOnly?, city?, limit?) - Lists streets with filters (public, limit: 1-100, default: 50, cyclableOnly: default true)

Street Reporting

- post_createReport(streetId, status, collectionMode, obstacles?, sensorData?, isPublishable?, isConfirmed?) - Creates condition report for street
- get_myReports(limit?) - Lists user's street reports (limit: 1-50, default: 20)

Routing Router (routing.*) Route finding and navigation between locations.

- get_findRoutes(startStreetName, endStreetName) - Finds routes between two streets using existing path data (public)
- get_getRouteDetails(routeId) - Gets detailed route information with streets and recent reports (public)

F. Selected architectural styles and patterns

The design of Best Bike Paths (BBP) is not limited to simple functional implementation but adopts established architectural styles and specific design patterns to ensure modularity, maintainability, and a clear separation of responsibilities. The following sections analyze the main structural choices adopted.

F.1. Architectural Styles

Layered Architecture (N-Tier) To guarantee a clear separation of concerns, the system's back-end has been structured according to the Layered Architecture style. This organization provides for a unidirectional and hierarchical data flow, where each layer depends exclusively on the one below it, remaining agnostic to the implementation details of the layer above. Specifically, the architecture is divided into four logical layers:

- **Presentation Layer:** Managed by the Hono framework and tRPC procedures, this layer is responsible for exposing API endpoints, validating incoming inputs, and handling the serialization of responses sent to the client.
- **Business Logic Layer:** Composed of "Services" (or logical Routers), this layer encapsulates the application's domain rules, such as calculating path scores or managing user sessions.
- **Data Access Layer:** This layer abstracts the interaction with the relational database. Thanks to the use of Drizzle ORM, SQL queries are encapsulated within type-safe methods, protecting the business logic from the complexity of the underlying query language.
- **External Integration Layer:** A layer dedicated to interacting with third-party services (OSRM, Nominatim, OpenWeather), acting as an adapter to normalize external data before it enters the system.

Client-Server with RPC (Remote Procedure Call)

Unlike the classic RESTful style, BBP adopts an approach based on strongly typed RPC (Remote Procedure Call), implemented via the tRPC library. In this model, the client

does not invoke generic resources via HTTP verbs, but directly calls "procedures" defined on the server (e.g., trip.create, path.getBest). This architectural choice was dictated by the need to maintain strict consistency between the Frontend (React) and the Backend. Since both share TypeScript type definitions, the system ensures that any modification to a function signature on the server is immediately detected as a compilation error on the client, drastically increasing Reliability and development speed.

Component-Based Architecture (Frontend)

The user interface strictly follows a component-based architecture. The application is not a monolithic block of HTML and JavaScript but is composed of independent, reusable, and encapsulated units (React Components). This style allows the presentation logic of each element (such as a path card or a reporting form) to be isolated, making the system easily extensible. The adoption of this architecture is supported by the Compound Components pattern (used in the shadcn/ui library), which allows for the construction of complex interfaces by assembling primitive components in a flexible manner, improving the readability and maintainability of the source code.

F.2. Backend Design Patterns

Service Layer Pattern

To prevent business logic from residing within HTTP request handlers (or tRPC procedures), a Service Layer was introduced. Specific classes or modules, such as the RoutingService, contain the pure application logic. For example, the mathematical calculation to aggregate user reports and determine the condition of a street does not occur at the API entry point but is delegated to the aggregation service. This pattern fosters code reuse and simplifies unit testing, as services can be tested independently of the web infrastructure.

Strategy Pattern

The requirement to support different modes of trip data acquisition (manual entry, simulated automated detection, or pure routing) was resolved by implementing the Strategy Pattern. The system defines a common interface for creating a "Trip" but the concrete algorithm varies based on the collectionMode parameter.

- In the "Manual" case, the strategy is limited to saving the data entered by the user.
- In the "Simulated" case, the strategy invokes the synthetic track generator.
- In the "OSRM" case, the strategy consults the external routing service. This allows for the addition of new acquisition modes in the future without modifying the existing code that handles trip saving.

Middleware Pattern

Cross-cutting request management is entrusted to a chain of Middleware. Before a request reaches the business logic, it passes through a series of intermediate functions responsible for injecting necessary dependencies. One middleware injects the database instance (db), another handles authentication by injecting the user object (auth), and yet another han-

dles logging. This approach centralizes the management of cross-cutting concerns such as security and configuration, avoiding code duplication in every single procedure.

F.3. Frontend Design Patterns

Container/Presentational Pattern

In the frontend, the decision was made to separate data retrieval logic from visualization logic. "Container" components (often coinciding with Navigation Routes, e.g., TripsPage) are responsible for interacting with APIs and managing loading states and errors. Once the data is obtained, it is passed via props to "Presentational" components (e.g., TripList), which are pure and solely responsible for how information is displayed on the screen. This separation improves the testability of graphical components and the clarity of the data flow.

Segregated State Management

Application state management is not monolithic but clearly distinguishes between Server State and Client State.

- For data originating from the server (e.g., trip lists, weather details), a Data Fetching/Caching pattern is used (via TanStack Query). This automatically handles cache validity, background refetching, and request deduplication.
- For volatile interface state (e.g., active search filters, unsubmitted forms), an Atomic pattern is used (via Jotai), where the state is fragmented into small independent units ("atoms") that avoid unnecessary re-renders of the entire page.

G. Other Design Decisions

This section illustrates further design decisions that, while not strictly falling under classic pattern definitions, proved fundamental to satisfying the specific requirements of the BBP project.

Monorepo with Workspaces

The source code has been organized into a single repository (Monorepo) managed via Bun workspaces. This structure allows hosting the frontend application (/apps/app), the backend (/apps/api), and shared packages (/packages/core, /db) within the same development environment. The main motivation for this choice lies in the ability to share code and type definitions between client and server without having to publish external libraries. For instance, the database schema or utility functions for coordinate calculation are defined once and imported directly where needed, ensuring there are never mismatches between the two parts of the system.

Schema-First API Design and Type Safety

To ensure interface robustness, a Schema-First approach based on the Zod validation library was adopted. Instead of manually writing input control logic, each API endpoint pre-defines a strict schema of expected data. This schema is used both at runtime to

validate incoming requests (automatically rejecting malformed data) and at compile-time to generate TypeScript types. This design decision creates a binding contract between frontend and backend: if the backend modifies a schema, the frontend stops compiling until it is updated, preventing a vast class of runtime bugs typical of web applications.

Client-Side Sensor Simulation

A critical decision for the demo development was to implement sensor simulation (GPS, accelerometer) directly in the user's browser (User Device node) rather than on the server. Since the demo application runs in a desktop web environment that lacks actual motion sensors or moving GPS, the frontend includes a simulation module ("Virtual Sensors"). This module generates synthetic GPX tracks and plausible telemetry data locally. This architectural choice allows for faithful demonstration of the "Automated Data Collection" requirement defined in the RASD without requiring dedicated mobile hardware for the testing and presentation phase, shifting the simulation complexity to the client to avoid burdening the backend with dummy data.

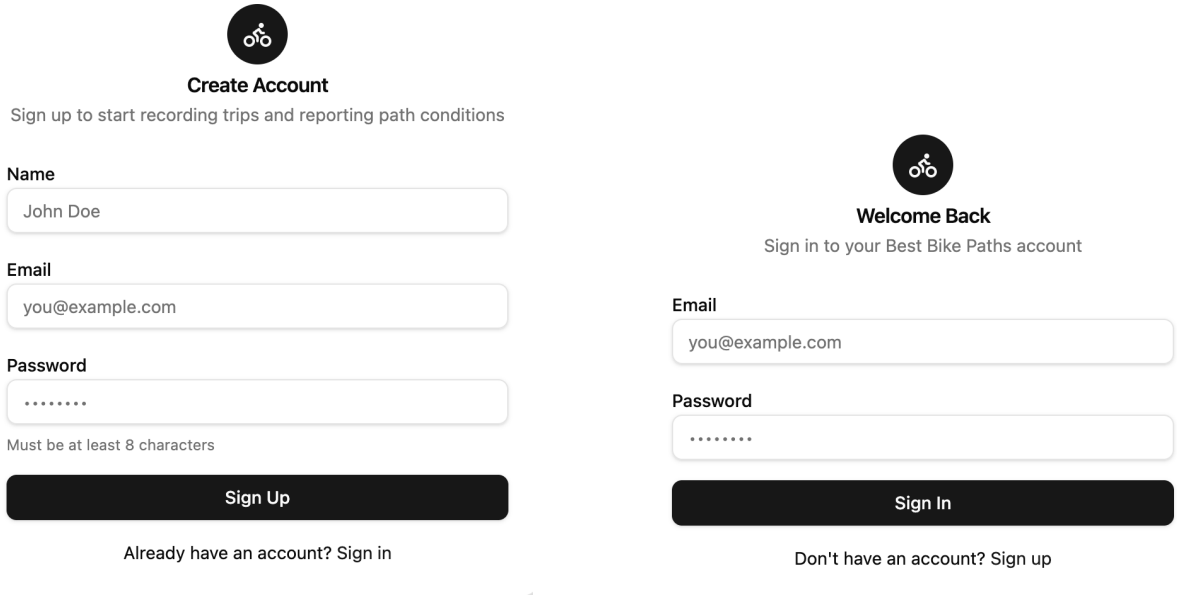
Dependency Injection via Context

To improve testability and decoupling, the backend uses a manual Dependency Injection pattern via the tRPC context. Instead of directly importing global instances (such as the database connection or Redis client) inside functions, these are injected into the request "Context" at startup. During testing, this allows the real database to be easily substituted with a mock or an in-memory database, completely isolating the unit of code under test from external infrastructures.

3 | User interface design

This chapter outlines the User Interface and User Experience architecture of the BBP platform, illustrating the visual layout of key pages and the functional flow triggered by user interactions.

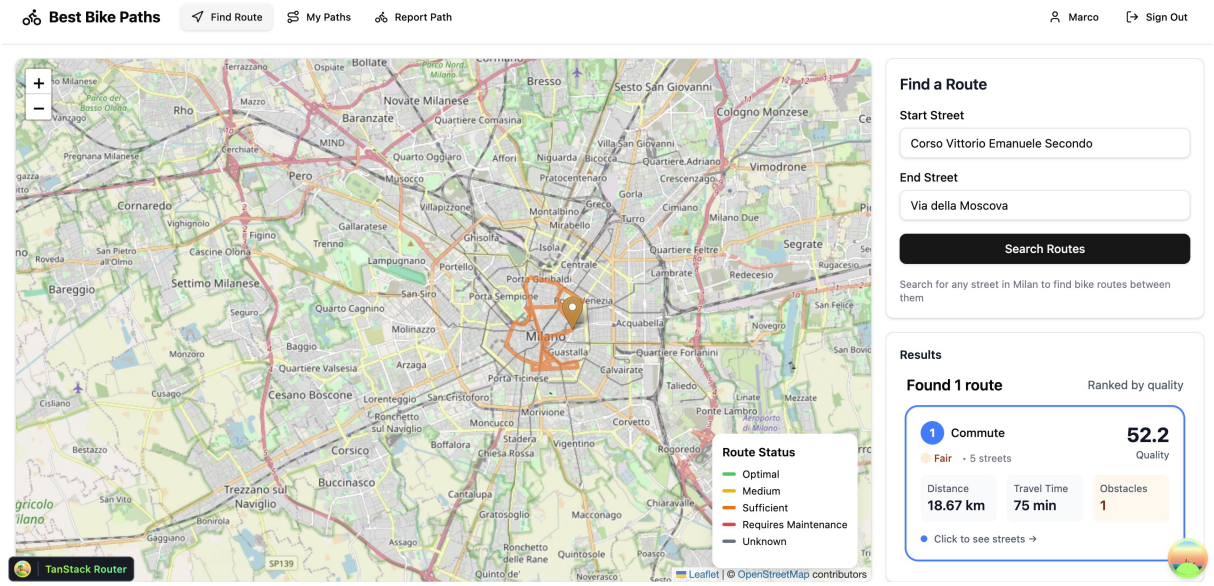
SignUp and Login



The registration and login interface presents a simple and clear form with fields for email and password, along with specific error messages for invalid inputs (e.g., incorrect email format).

Registration requires implicit user consent for access to device sensors and for personal data and privacy processing. The process provides instant feedback to inform the user of success or any registration issues.

Homepage

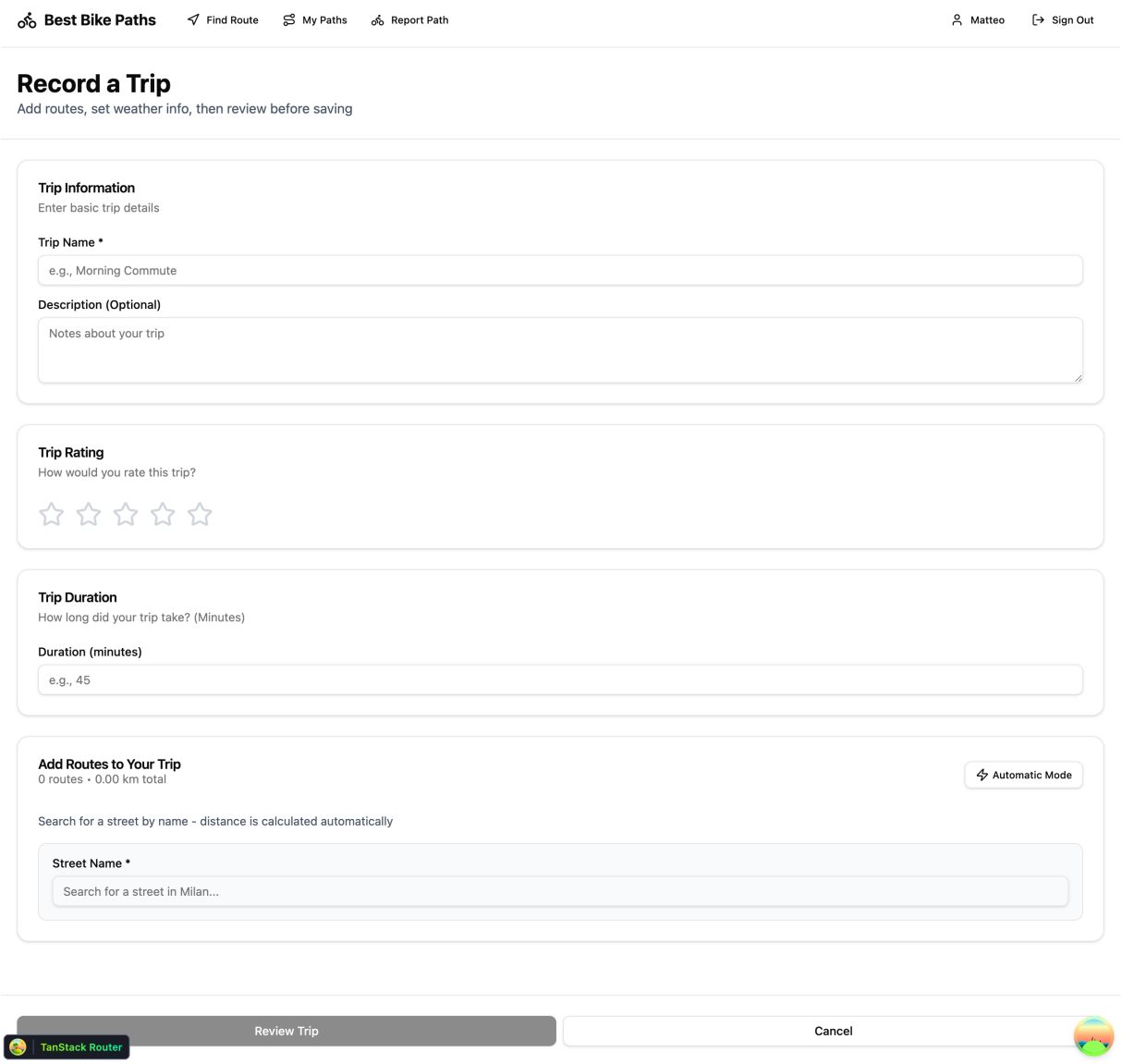


The homepage centers on the "Find Route" section, featuring an interactive map that displays only roads recognized by the system as cycling paths. The interface includes a search bar with two text fields for quickly entering a starting and destination address.

After data entry, a "Search Routes" button initiates the path search. The map supports standard zoom and pan operations and features interactive icons highlighting points of interest. At the top, icons for other app sections are present, such as "My Paths", "Report Path", as well as areas dedicated to the user profile, login, and sign out.

The interface is visible to both registered and unregistered users, with limitations on other areas for the latter.

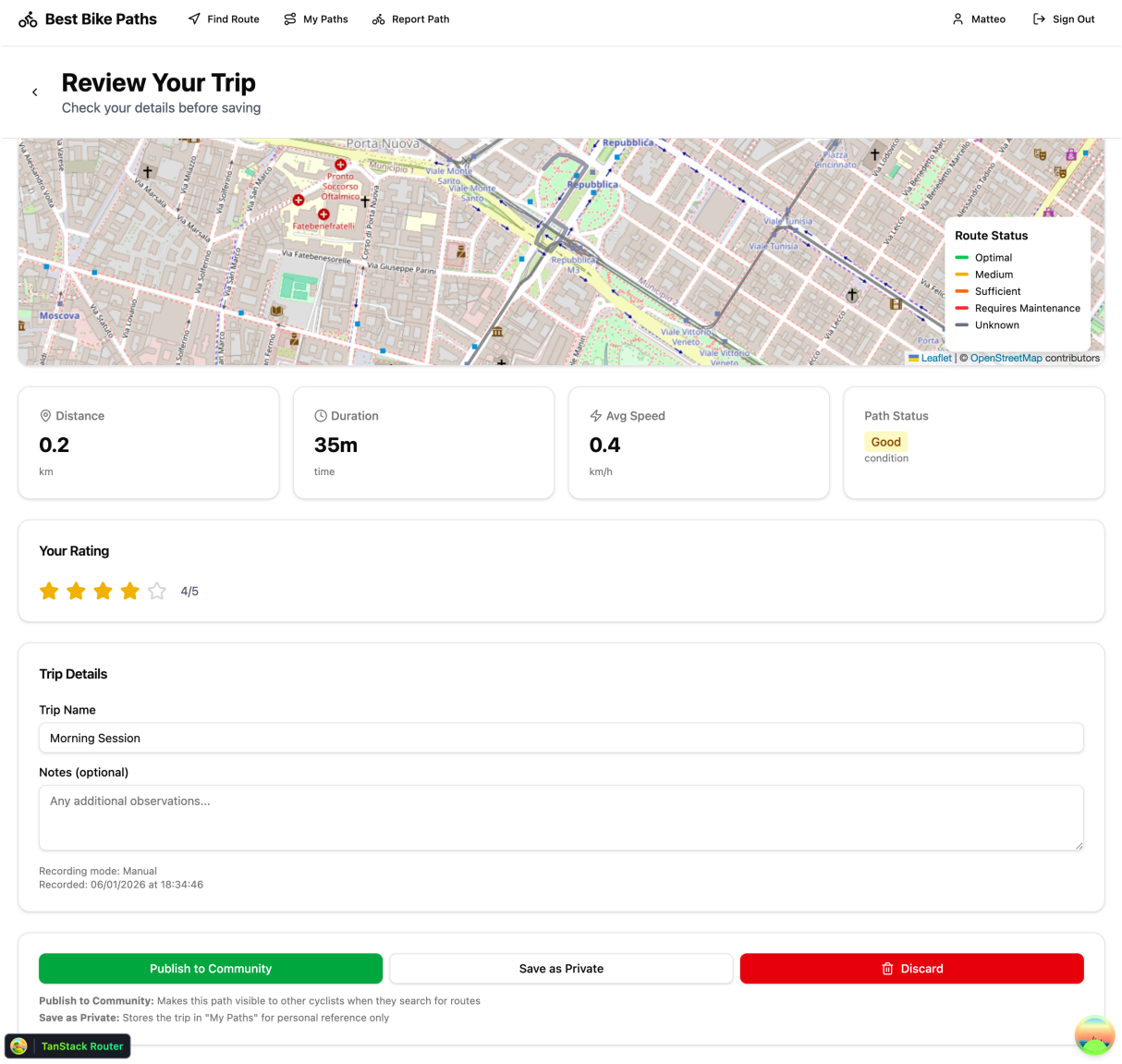
Manual route entry



The manual route entry interface, which can be accessed from "My Paths" via the "+ Record Path" button, allows a registered user to add traveled cycling streets through a search bar where available street names can be typed or selected. Streets not recognized by the system as cycling paths are not accepted.

For each street, the user can indicate the start and end points of the traversal and its condition by choosing from a list of basic options (e.g., optimal, fair, sufficient, needs maintenance). On the map, obstacles can be reported via the "Mark Obstacles" button, which allows the user to place them and add a brief description.

The interface includes fields for entering the path name, an optional description, travel time, and a star rating from 1 to 5. A final button allows saving the entered data and viewing a path summary with relevant automatically calculated statistics (such as distance traveled and average speed). At that point, the user can choose to save the path in "My Paths" or publish it. This section is available only to registered cyclists.



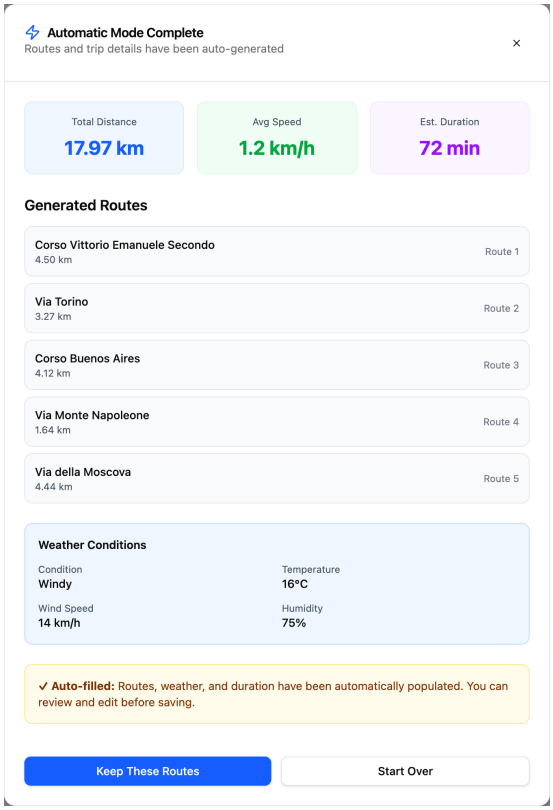
Automatic Route Entry

The automatic route entry interface activates automatically during recording using the registered user's device sensors. In the web app, it can be simulated in the automatic recording area via the "Automatic Mode" button.

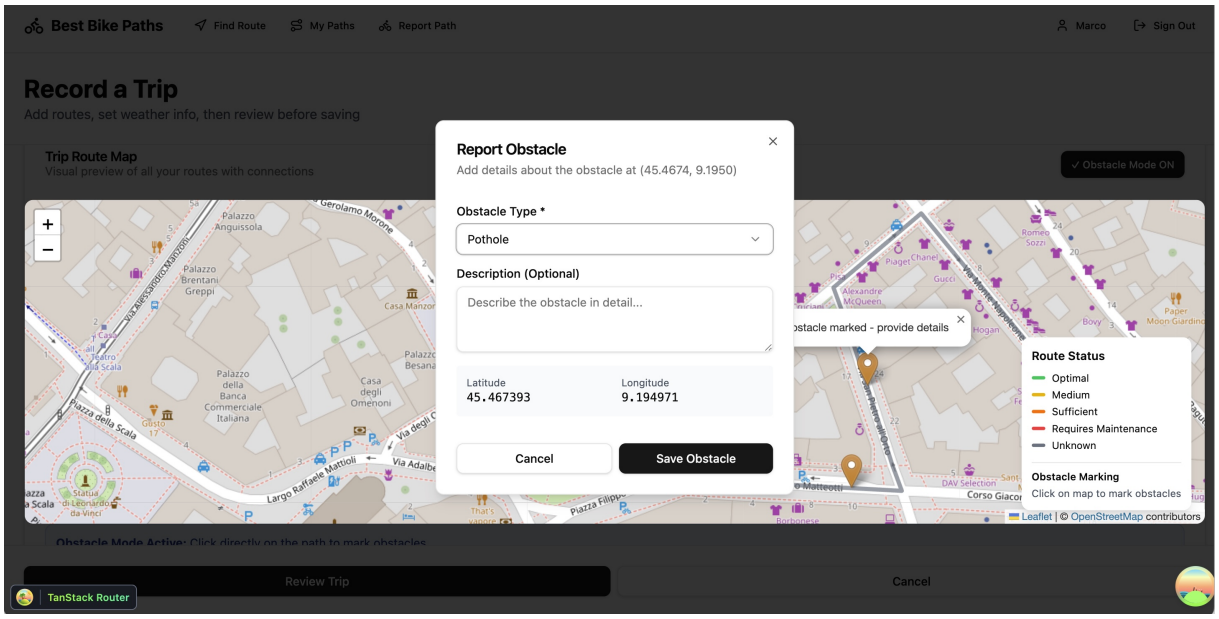
At the end of the route, performance statistics and a list of detected streets to be confirmed or rejected are shown. Upon confirmation, a summary is displayed featuring a map highlighting the traced route with obstacles automatically detected by the sensors, streets with their respective status, star ratings, statistics, and a box with weather conditions.

The user can confirm, delete, or modify the data. They can then add a description and must provide a name for the path. The user can also publish the route, in addition to saving it in their personal area.

The automatic calculation of the status. This section is available only to registered cyclists.



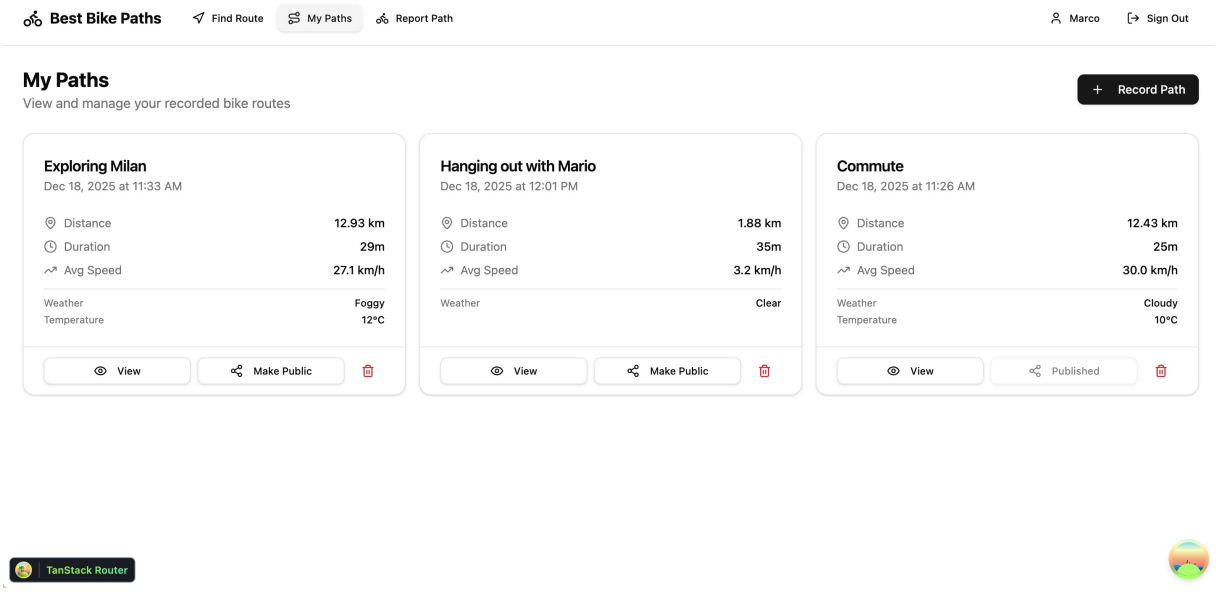
Mark obstacles



This interface allows users to report obstacles along routes within the various reports. Registered cyclists can use the "Mark obstacles" feature to pinpoint the exact location of hazards directly on the map path.

Users can specify the obstacle type (e.g., water, potholes, other) and include additional notes. Once registered, obstacles are displayed as markers on the map and listed below it, showing their respective types and coordinates.

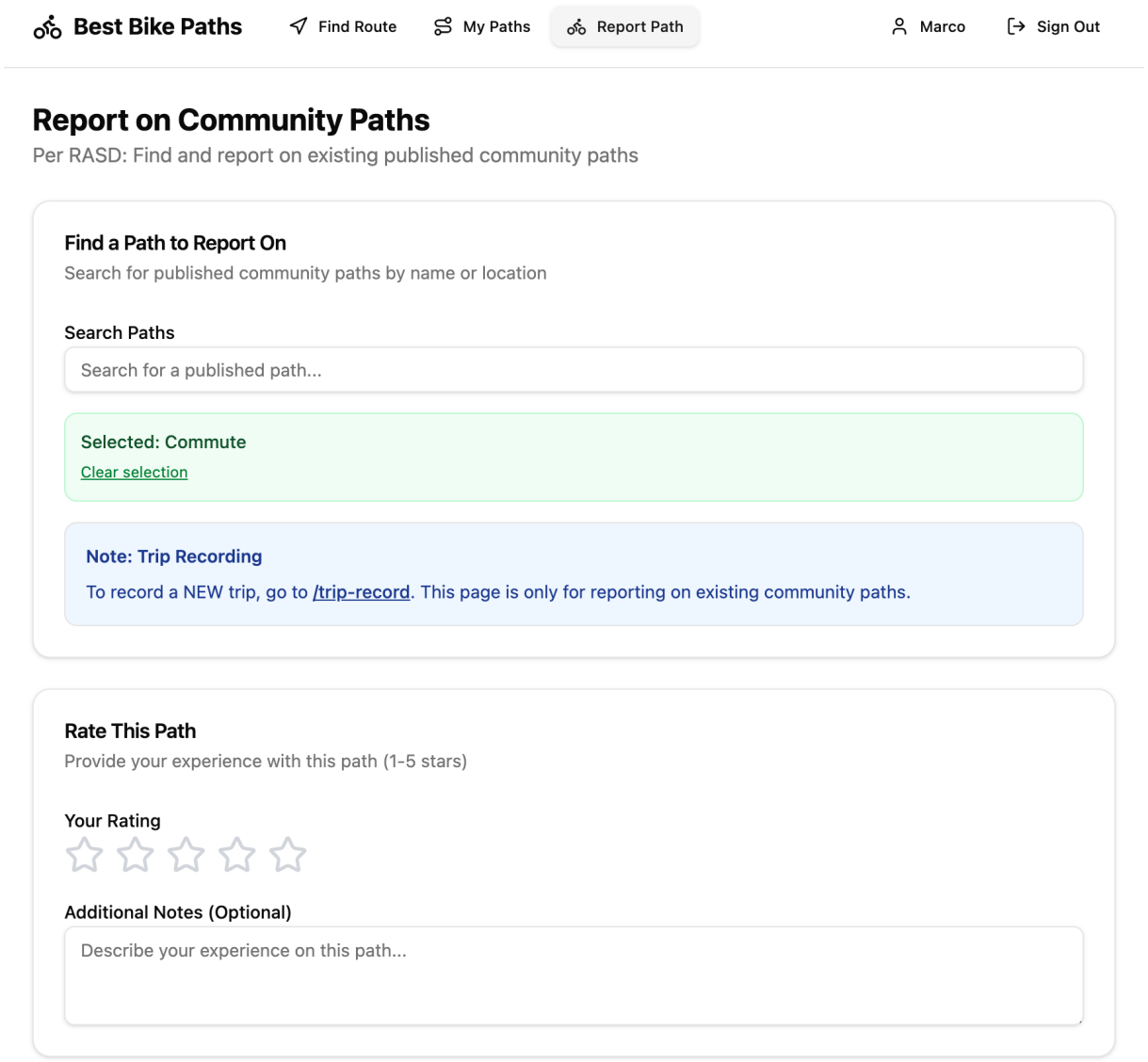
MyPaths



The “My Paths” interface allows users to view their saved routes in chronological order. Each entry shows information such as name, date, statistics, and any weather conditions. Simple buttons are available to delete, publish, or view the paths.

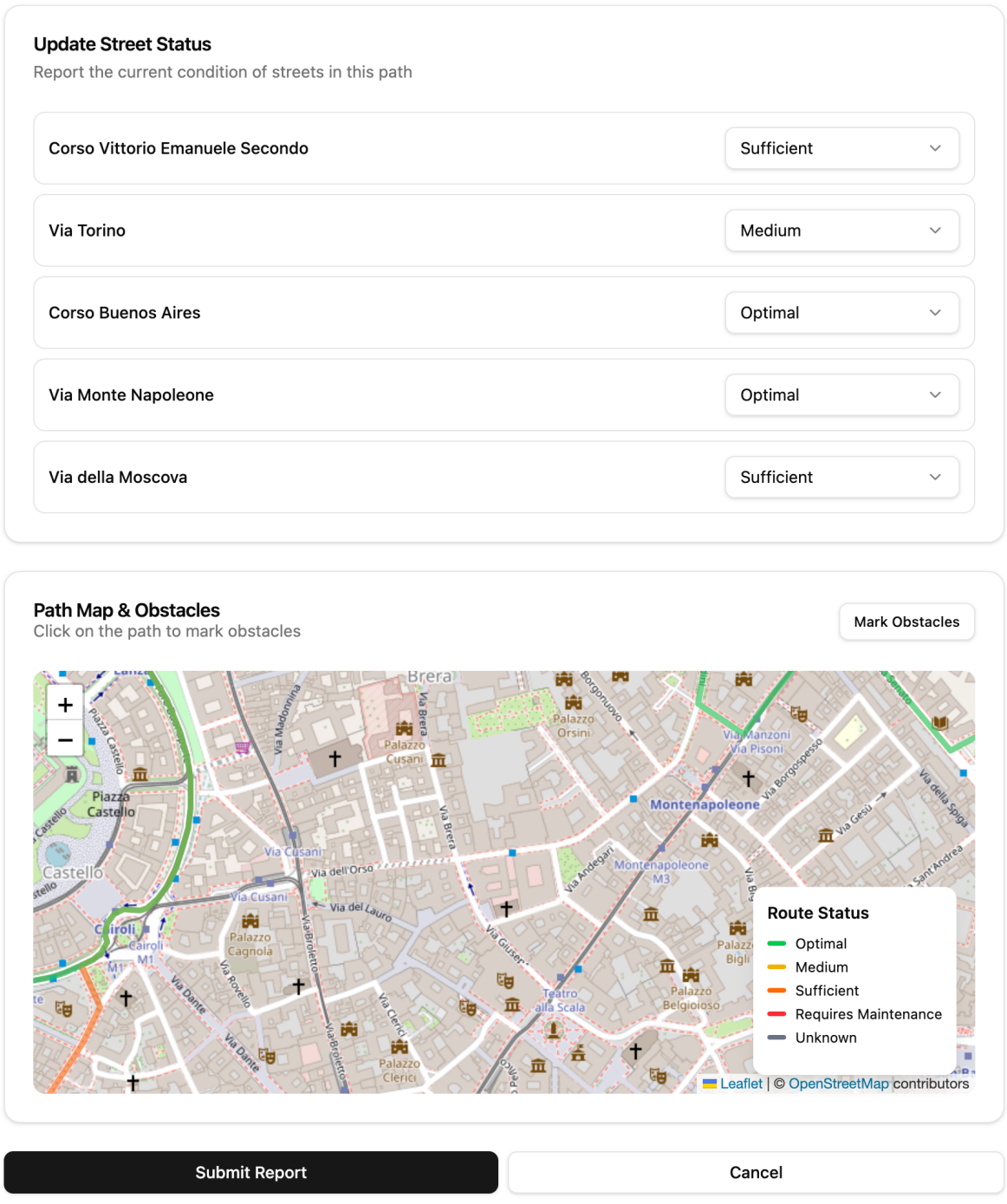
By choosing to view a route, it is displayed on the map along with detected obstacles. From there, the user can return to the path list. Deletions require confirmation to prevent accidental removal. This section is available only to registered cyclists.

Report route



This interface allows for the re-review of an existing path previously published and created by another user. A search bar allows the user to find the existing path by name. Once the itinerary is selected, the interface displays its current details and enables the user to submit a new report.

The cyclist can update the status of individual streets (e.g., optimal, fair, sufficient, needs maintenance), manage obstacles via the "Mark Obstacles" button, add an optional description, and provide a new star score. Upon completion, the "Submit Report" button sends the data; the system confirms the success of the operation and automatically recalculates the overall path status, updating the information on the map for the entire community. Conversely, the "Cancel" button discards the report.



4 | Requirements Traceability

This section breaks down how each RASD requirement is addressed by specific design elements. The tables below provide a clear mapping of how our design fulfills each requirement.

A. User Management and Authentication

Requirements	**R1**: The system shall allow a new user to create an account by providing necessary information. **R2**: The system shall allow a registered user to log in by providing their credentials.
Components	Web Browser, API Gateway, Authorization Manager, DBMS, Email Service Provider

Requirements	**R4**: The system shall provide an interface for a logged-in user to view and edit their profile information. **R5**: The system shall allow a user to log out of their account on a device.
Components	Web Browser, API Gateway, Authorization Manager, DBMS

Requirements	**R33**: The system shall provide a section where a logged-in user can view their past recorded trips.
Components	Web Browser, API Gateway, DBMS, Trip Manager

B. Trip Recording and Sensor Tracking

Requirements	**R6**: The system should attempt to detect when the user is cycling and start recording. **R7**: While recording, the system shall log the sequence of GPS coordinates that represent the user's path. **R17**: The system shall support continuing recording even if the user switches apps or turns off the screen, once a recording is started.
Components	Web Browser

Requirements	**R18**: While a trip recording is active (started by the user via R7), the system shall collect accelerometer and gyroscope data continuously in the background.
Components	Web Browser

Requirements	**R8**: The system shall automatically compute summary statistics for the trip. **R9**: After stopping the recording and computing stats, the system shall show the user a summary of their ride. **R10**: The system shall save the recorded trip data to the user's account.
Components	Web Browser, API Gateway, DBMS, Trip Manager

Requirements	**R11**: When a trip is recorded, the system shall attempt to retrieve weather data for the ride. **R12**: If weather data was successfully obtained for a trip, the system shall display it along with the other stats on the trip summary view (R11).
Components	Trip Manager, Weather API, Web Browser

C. Community Data, Safety and Obstacles

Requirements	**R13**: The system shall provide a feature for users to add a bike path entry manually. **R14**: The system shall allow the user to mark any known obstacles or hazards along the path. **R15**: The system shall prompt the user whether to publish this information to the community or keep it private.
Components	Web Browser, API Gateway, Path Manager, Obstacle Manager

Requirements	**R19**: The system shall analyze the sensor data in real-time (or in short batches) to detect possible road anomalies. **R20**: For each detected potential obstacle or anomaly during the ride, the system shall log it internally in the context of the current recording.
Components	API Gateway, Obstacle Manager, DBMS

Requirements	**R21**: When the trip is finished, if the system has any automatically detected events (R20), it shall prompt the user to review them. **R22**: The system shall present the detected events in a user-friendly way. **R23**: The user shall be able to confirm, reject or modifies the events. **R24**: After the user has finished reviewing and confirming the data from an automated collection ride, the system shall save the verified information.
Components	Web Browser, API Gateway, DBMS, Obstacle Manager

Requirements	**R25**: The system shall maintain a central repository of all published bike path entries. **R26**: The system shall allow updates to existing path entries. **R28**: The system shall ensure that published path data does not include personal information about cyclists.
Components	Path Manager, DBMS

Requirements	**R27**: Whenever new path information is published, the system shall integrate it into the dataset used for route queries.
Components	Path Manager, Street Manager, Overpass API

D. Route Discovery and Navigation

Requirements	**R3**: The system shall permit users who are not logged in (guests) to use core viewing functions.
Components	Web Browser, API Gateway, Path Manager

Requirements	**R29**: The system shall provide an interface for the user to specify a start and end location for their desired trip (route query). **R31**: FThe system shall display the calculated route(s) on the map UI for the user.
Components	Web Browser

Requirements	**R30**: The system shall compute a score or rating for each route found, to communicate its quality to the user. **R32**: For each route, the system shall provide a summary of key details.
Components	API Gateway, Routing Manager, OSRM API, Path Manager

5 | Implementation, Integration and Test Plan

A. Overview and Implementation Plan

This chapter describes the implementation strategy, integration approach, and comprehensive testing plan for the Best Bike Paths platform. The development follows a systematic bottom-up methodology that ensures each component is thoroughly validated before integration into the larger system.

The bottom-up implementation strategy begins with the fundamental data layer components and progressively builds toward the user-facing presentation tier. This approach allows development teams to work on independent modules simultaneously while maintaining clear integration checkpoints. Each newly developed module undergoes isolated testing with purpose-built drivers before being integrated into the growing system. Once integrated, the module replaces its corresponding driver and becomes part of a validated subsystem that serves as a foundation for subsequent components.

This incremental integration methodology offers several advantages particularly relevant to the Best Bike Paths architecture. The modular nature of the tRPC-based backend and component-based React frontend naturally supports parallel development across features. Testing occurs on manageable subsystems rather than attempting to validate the entire system at once, making bug identification and resolution more efficient. The clear dependency hierarchy between features ensures that foundational capabilities like authentication and data persistence are solid before building community features like path sharing and obstacle reporting on top of them.

The implementation leverages the monorepo structure to maintain type safety across the frontend-backend boundary throughout development. Drizzle ORM's type-safe database queries ensure that schema changes immediately surface as TypeScript compilation errors in dependent services. The tRPC procedures provide compile-time verification that frontend calls match backend implementations, eliminating an entire class of integration bugs before runtime. This architectural foundation allows the bottom-up strategy to proceed with confidence that type mismatches will be caught during development rather than discovered during integration testing.

B. Features Identification

The Best Bike Paths platform functionality decomposes into distinct feature sets that can be implemented and integrated following clear dependency relationships. Each feature builds upon previously implemented capabilities, establishing a natural implementation order that minimizes integration complexity.

[F1] Authentication and Session Management. This foundational feature set encompasses user registration, login, session creation, and authentication middleware. Every subsequent feature in the platform depends on reliable user identity management, making this the first implementation priority. The authentication system uses Better Auth with session-based cookies for security, implementing password hashing with industry-standard algorithms and secure session token generation. This feature integrates directly with the PostgreSQL database for user credential storage and session persistence, requiring only the data layer to be operational before implementation can begin. The frontend components include registration forms with client-side validation, login interfaces with clear error messaging, and session state management through authentication context providers.

[F2] Trip Recording Infrastructure. Once users can authenticate, the system must provide the core value proposition of recording cycling trips. This feature set divides into two distinct but related capabilities: manual trip recording and automated trip simulation. The manual recording mode allows users to construct routes by selecting street segments obtained through Overpass API integration, with OSRM calculating distances between waypoints. The automated mode generates simulated trips with realistic route geometries, weather conditions, and obstacle placements for demonstration and testing purposes. Both modes persist trip data including start and end times, distance and duration metrics, route geometries as GeoJSON, and optional weather information. This feature requires authentication to associate trips with users and relies on the database schema for trip, route, and obstacle entities. The implementation includes the Trip Manager service on the backend and Trip Recording components with map visualization on the frontend.

[F3] Map Visualization and Interaction. The platform's geographic nature demands robust map rendering and interaction capabilities. This feature implements the Leaflet-based map interface that displays trip routes, path geometries, and obstacle locations. Users interact with the map to select waypoints during manual trip construction, view completed trips with rendered routes, and visualize community paths with condition indicators. The map components handle coordinate transformations, zoom level management, marker placement, and polyline rendering for route geometries. This feature depends on trip recording to have data to visualize and integrates with both Overpass for street geometry display and OSRM for route preview during trip construction. The implementation requires careful attention to performance when rendering large route geometries and managing map tile loading efficiently.

[F4] Obstacle Reporting and Verification. Building on trip recording, this feature enables users to mark obstacles encountered during rides and manage their lifecycle through the verification workflow. During manual trip recording, users can click map locations to place obstacle markers, specifying type and description. Automated trip simulation generates obstacles based on route geometry, starting in PENDING status awaiting user confirmation. The trip detail view displays all obstacles associated with a trip, allowing the owner to transition them through CONFIRMED, REJECTED, or CORRECTED states. This feature requires the Trip Recording infrastructure to exist before implementation since obstacles attach to specific routes within trips. The Obstacle Manager service enforces status lifecycle rules and ownership verification, while the frontend provides intuitive interfaces for obstacle manipulation with clear visual feedback on verification status.

[F5] Path Publishing and Discovery. This feature transforms personal trips into community-shared paths, enabling the collaborative aspect central to the platform's mission. Users can publish trips that meet quality criteria—having a rating and verified obstacles—converting them into public paths visible on the community map. The publishing process creates path records linked to the trip's street segments through path_segment relationships, calculating aggregate condition scores from obstacle data. Path discovery allows users to search for routes by geographic area or origin-destination pairs, viewing paths ranked by community ratings and condition assessments. This feature depends on Trip Recording to have content to publish and Obstacle Verification to ensure published paths have validated hazard information. The implementation includes the Path Manager service, path search and ranking algorithms, and discovery interfaces with filtering capabilities.

[F6] Rating and Review System. The quality of community-contributed paths relies on user feedback mechanisms. This feature allows users to rate trips before publishing, providing the community with quality signals about route desirability. The rating system captures numeric scores and optional textual reviews that surface in path detail views. Aggregate ratings influence path ranking in search results, prioritizing well-regarded routes. This feature integrates with Path Publishing since ratings are required before publication, and enhances Path Discovery by providing sorting and filtering dimensions. The implementation requires extending trip entities with rating storage, creating rating capture interfaces in the trip detail view, and incorporating ratings into the path scoring algorithm within the Aggregation Engine.

[F7] External Service Integration. Throughout the feature implementations described above, integration with external services provides critical capabilities the platform doesn't implement internally. OSRM integration calculates optimal cycling routes between coordinates, providing realistic distance and duration estimates along with detailed route geometries. Overpass API delivers OpenStreetMap street data for manual route construction, enabling users to search for streets by name and retrieve their precise coordinates. Weather service integration enriches trip records with atmospheric conditions at the time of recording. These integrations require robust error handling since

external service availability cannot be guaranteed. The implementation includes adapter services that normalize external API responses, implement retry logic with exponential backoff, and provide graceful degradation when services are unavailable. Circuit breaker patterns prevent cascading failures when dependencies experience extended outages.

[F8] User Profile and Statistics. Users need visibility into their cycling activity and contributions to the community. This feature provides personal areas displaying trip history, published paths, obstacle reports, and aggregate statistics like total distance cycled and paths contributed. The profile view shows user information and activity summaries, while detailed statistics pages break down metrics by time period and activity type. This feature depends on Trip Recording, Path Publishing, and Obstacle Reporting features to have data to aggregate and display. The implementation includes specialized query functions that efficiently calculate statistics across potentially large datasets, caching mechanisms to avoid repeated expensive computations, and visualization components that present metrics in user-friendly formats.

C. Integration Strategy

The integration of system components proceeds systematically from the data persistence layer upward through business logic services to the user interface, following the bottom-up methodology outlined in the implementation overview. Integration begins immediately once the database management system and hosting infrastructure are operational. External service connections are established progressively as features requiring them reach integration.

The foundational integration establishes the database schema and ORM layer. The PostgreSQL instance with PostGIS extension is deployed and configured with connection pooling appropriate for serverless environments. Drizzle ORM integration is validated through migration execution and basic CRUD operations on test data. The schema supporting users, sessions, trips, routes, obstacles, paths, and path segments is created with proper indexes for geospatial queries and foreign key relationships ensuring referential integrity. Testing at this level uses database clients to verify schema correctness, constraint enforcement, and index effectiveness through query analysis. Once validated, the database layer provides the persistence foundation for all subsequent integrations.

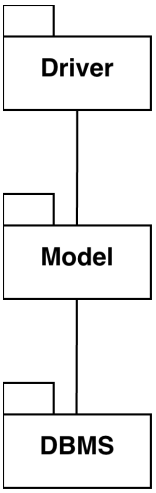


Figure 5.1: Database layer integration with Drizzle ORM

Building on the persistence layer, the authentication system components are integrated next. The Better Auth library is configured with the database connection, implementing user registration with password hashing, login with credential verification, and session creation with secure cookie handling. The authentication middleware is integrated into the Hono HTTP framework, providing request context enrichment with authenticated user information. Testing involves driver programs that invoke registration and login procedures, verifying password hashing produces different outputs for identical passwords, session tokens are cryptographically secure, and session validation correctly accepts valid tokens while rejecting invalid or expired ones. Frontend authentication components are integrated with the backend tRPC procedures, establishing the session management flow that protects subsequent features.

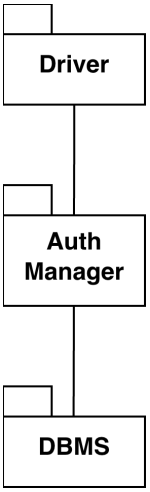


Figure 5.2: Authentication system integration

The Trip Manager service and its associated data access objects are integrated after authentication is validated. This integration includes trip creation procedures that accept

manual and simulated collection modes, route addition functions that persist GeoJSON geometries, and obstacle creation with status lifecycle management. External service adapters for OSRM and Overpass are implemented and tested with mock responses before connecting to actual external APIs. Testing drivers invoke trip creation with various data combinations, verify cascade deletes remove associated routes and obstacles when trips are deleted, and confirm GeoJSON storage and retrieval maintains coordinate precision. The frontend trip recording components are integrated incrementally, starting with basic trip information forms, then adding map interaction for route construction, and finally incorporating obstacle marking capabilities.

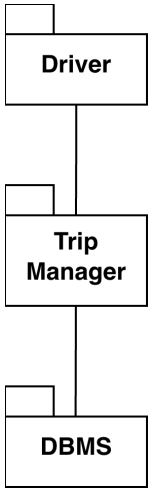


Figure 5.3: Trip recording feature integration

Map visualization components integrate alongside trip recording since route display is essential for user verification of constructed trips. The Leaflet library is configured with appropriate tile providers and the map container is embedded in trip recording and detail views. Route rendering functions convert GeoJSON LineStrings into Leaflet polylines with styling that indicates route quality. Marker placement for obstacles uses custom icons that visually distinguish between PENDING, CONFIRMED, and other status values. Testing involves rendering various route geometries including edge cases like single-point routes and very long routes with thousands of coordinates, verifying zoom and pan operations maintain performance, and confirming marker click events trigger appropriate detail displays.

The obstacle verification workflow integrates after trip recording and display are operational. The trip detail view is extended with obstacle lists showing current status and providing actions for confirmation, rejection, and correction. The Obstacle Manager service implements state transition validation, rejecting invalid transitions and ensuring only trip owners can modify obstacle status. Testing validates that status changes persist correctly, transition rules are enforced preventing EXPIRED obstacles from returning to PENDING, and ownership verification prevents users from modifying obstacles they don't own. Integration testing covers the complete flow from trip creation with simulated obstacles through user verification to final CONFIRMED or REJECTED states.

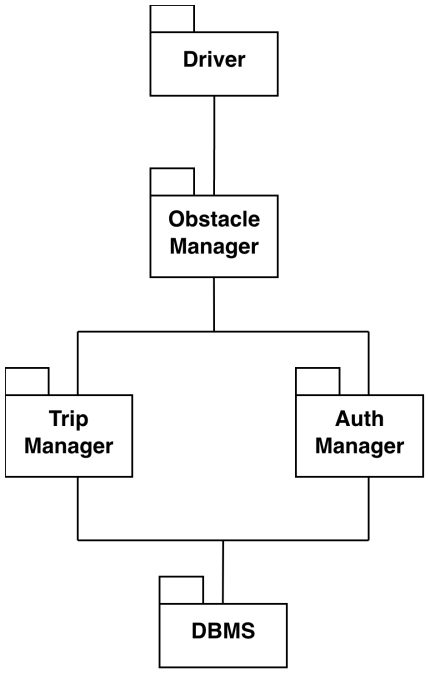


Figure 5.4: Obstacle verification integration

Path publishing functionality integrates once trip recording and obstacle verification are reliable. The Path Manager service implements the publishing logic that validates trip eligibility, creates path records, and establishes path segment relationships linking paths to constituent streets. The Aggregation Engine integrates to calculate path scores from obstacle data and user ratings. Testing verifies published trips create corresponding path records, path segments correctly reference street segments from routes, trips already published cannot be republished, and deleting a path cascades to remove path segments while preserving the original trip. Frontend integration includes publishing buttons in the trip list view, confirmation dialogs explaining publishing consequences, and validation error messages when requirements aren't met.

Path discovery features integrate after publishing provides content to discover. The path search procedure implements spatial queries using PostGIS to find paths within geographic bounds or along specified corridors. Search results include aggregate ratings, condition scores from the Aggregation Engine, and summary statistics like total distance and elevation gain. The frontend discovery interface provides map-based search where users define areas of interest, list views ranking paths by relevance, and detail views showing complete path information with review summaries. Testing validates search returns appropriate results for various geographic queries, ranking algorithms surface high-quality paths, and detail views correctly display path geometry and obstacles.

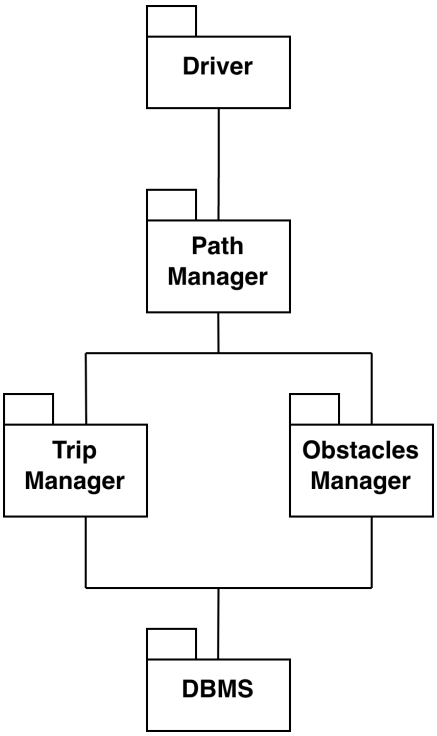


Figure 5.5: Path publishing and discovery integration

Rating and review features integrate alongside publishing since they're required for publication. The trip detail view gains rating interfaces allowing numeric score selection and optional review text entry. The rating data persists with trip records and appears in path detail views after publication. The Aggregation Engine incorporates ratings into path scoring algorithms, weighting recent ratings more heavily than older ones and considering rating distribution to identify controversial paths. Testing confirms ratings persist correctly, aggregate calculations accurately reflect individual ratings, and published paths display rating summaries that update when new ratings are added.

User profile and statistics features integrate after the core trip and path workflows are operational, since they aggregate data from these features. The profile view queries for user trips, published paths, and obstacle reports, presenting summaries with key metrics. Statistics calculations efficiently aggregate data across potentially thousands of trips using database indexes and caching to avoid performance degradation. The personal trips area integrates trip listing, detail viewing, and management actions like deletion and publishing. Testing validates statistics calculations match expected values for test data sets, profile views efficiently load even for very active users, and deletion operations properly cascade through related data.

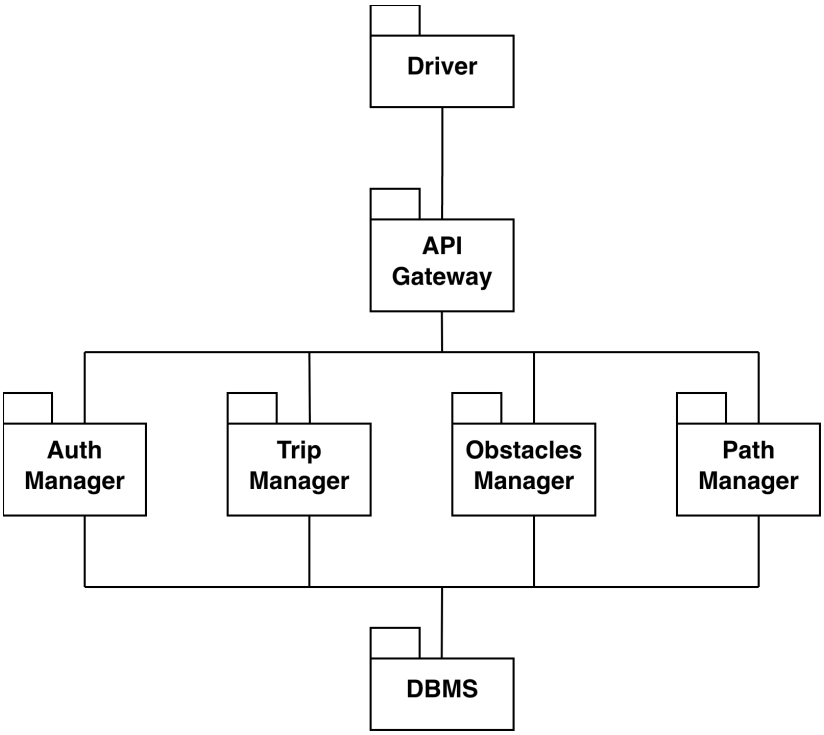


Figure 5.6: Complete integrated system

External service integrations occur progressively as features requiring them reach implementation. OSRM integration begins during trip recording implementation, initially using mock responses for testing before connecting to the actual public OSRM instance or a self-hosted deployment. Overpass integration follows a similar pattern, with mock street data enabling frontend development before live API integration. Weather service integration occurs late since it's non-critical, allowing implementation to focus on core features before adding contextual enhancements. Each external integration includes error handling that gracefully degrades functionality when services are unavailable, preventing external dependencies from blocking platform operation.

Throughout the integration process, the tRPC procedures serve as clear integration boundaries between frontend and backend. The type safety provided by shared TypeScript definitions ensures that frontend components always call backend procedures with correct parameter types and handle responses appropriately. This compile-time verification catches integration errors immediately during development rather than discovering them during runtime testing. The TanStack Query integration on the frontend provides caching and optimistic updates that improve perceived performance and reduce unnecessary backend requests, while maintaining consistency when data changes.

D. System Testing Strategy

System testing is used to ensure that each newly developed component works correctly on its own and that, once integrated, the overall platform still satisfies the requirements

described in the RASD.

Before integration, each backend service and key frontend module is validated in isolation using dedicated drivers (i.e., small test clients that invoke the component's public interfaces). After integration, tests are repeated with new drivers to verify correct interactions between modules and to ensure that end-to-end workflows remain consistent.

The strategy follows a layered approach (many fast unit/component tests, fewer integration tests, and a limited set of end-to-end tests for critical user journeys). For components with stateful lifecycles (e.g., obstacle verification), state-transition test cases are used to validate both valid and invalid status changes.

The following test types are applied:

- **Component (Unit) Testing:** Validates business logic and edge cases in isolation, using drivers and test doubles for external dependencies.
- **Integration Testing:** Verifies that subsystems cooperate correctly and that data is stored/retrieved consistently.
- **Functional (End-to-End) Testing:** Checks that main workflows (registration, trip recording, obstacle reporting/verification, path discovery) match the RASD scenarios from the user perspective.
- **Load and Performance Testing:** Measures responsiveness and stability under expected traffic, identifying bottlenecks and capacity limits.
- **Stress Testing:** Pushes the system beyond normal conditions to observe failure behavior and recovery capabilities.
- **Security and UI Testing:** Ensures authentication/authorization rules are enforced and the web interface remains usable and accessible across common environments.

Most automated tests run continuously in CI to provide rapid feedback, while broader end-to-end, performance, and security checks are executed on dedicated environments before releases.

6 | Effort Spent

	Chapter 1	Chapter 2	Chapter 3	Chapter 4	Chapter 5	Revision
Caldognetto	3	9	1	1	4	3
Salone	2	6	3	3	1	3