

RailSync Report

Author 1

`nicolas.amadori@studio.unibo.it`

Author 2

`riccardo.mazzi@studio.unibo.it`

March 2025

This report details the design and implementation of RailSync, a full-stack web application for train ticket booking. The project is built on the MEVN (MongoDB, Express.js, Vue.js, Node.js) stack and architected using a Docker-based microservice approach to ensure scalability and resilience. The system comprises multiple containerized services, including two backend instances, a frontend, a Redis database, a MongoDB replica set, and an NGINX reverse proxy for load balancing and fault tolerance.

Key features include secure user authentication and authorization using JSON Web Tokens (JWT) and Role-Based Access Control (RBAC). The application provides a user-friendly interface for searching real-time train solutions by integrating with the official Trenitalia API. A critical component is the seat reservation system, which leverages Redis to implement a distributed locking mechanism, preventing race conditions and ensuring data consistency during concurrent booking attempts.

The project also features a comprehensive administrative backend for user and reservation management. The development process followed a Git flow model with semantic versioning. System reliability was validated through a suite of automated Jest tests covering core functionalities and manual acceptance tests to verify frontend behavior, fault tolerance by simulating service failures, and security token handling. The entire application is packaged and deployed using Docker Compose, simplifying setup and ensuring a reproducible environment.

Disclaimer

During the preparation of this work, the authors used ChatGPT, to assist in correcting grammar and improving the clarity of the English language used throughout the document.

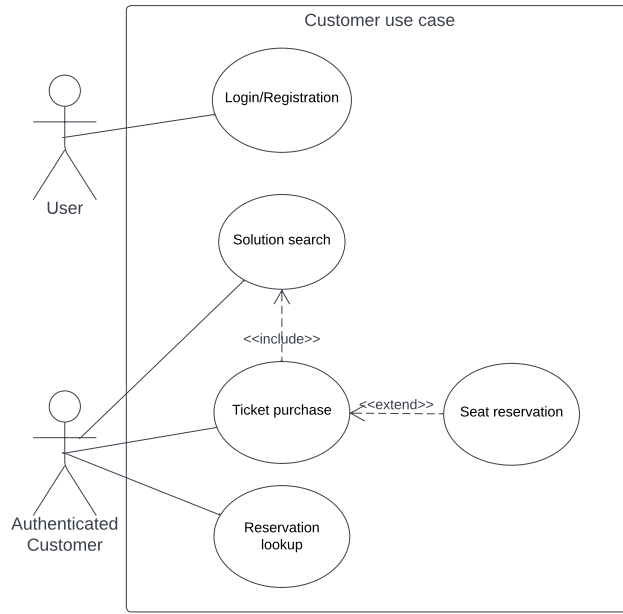


Figure 1: Use case of the customer

After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the final report/artifact.

1 Concept

In our project we developed a distributed web application to manage train booking, called RailSync.

At the moment, users will be connected to the same network, but with proper hosting, the server will be accessible from all over the world.

Our system handles multiple customers that will find trains, reserve seats and see their bookings, while few admin users will monitor the entire system.

Customers will be using our application whenever they need to travel by train and find their best choice, using any device with a browser and connectivity, such as smartphones and computers.

The system will store user's data, in particular their personal information given during the registration and their reservation details, inside a database hosted on the same physical server.

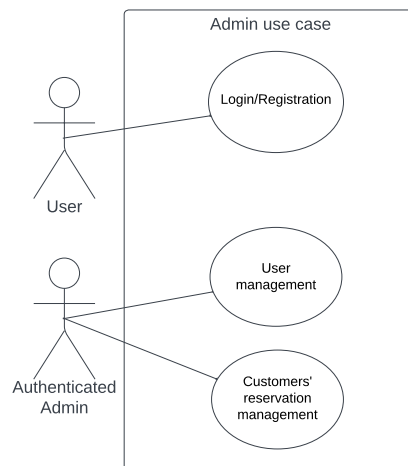


Figure 2: Use case of the admin

1.1 Customer case scenario

The use case shown in Figure 1 represents the interaction of the customers with the web application. Users must authenticate themselves before being able to perform any action.

1.2 Admin case scenario

The use case shown in Figure 2 represents the interaction of the admin with the web application. Admin, like customers, must authenticate themselves before being able to perform any action and can perform any actions available to simple users.

2 Requirements

- **Functional**

1. Login and registration
2. Multiple role access
3. Account and personal settings
4. Homepage with search functionality
5. Ticket purchase with seat reservation
6. Personal reservation details
7. User management (for admins only)
8. Customers' reservation management (for admins only)

- **Non-functional**

1. Concurrent access to trains seats during reservation
2. Usability for inexperienced users
3. Availability of the service h24
4. Load balancing of the requests
5. Fault tolerance
6. High-security system

- **Implementation**

1. RESTful APIs support for third party integration
2. MongoDB, Express.js, Vue.js, Node.js (MEVN) architecture
3. NoSQL database support for booking and user management
4. Reactive Web application for cross-platform access
5. Docker-based structure

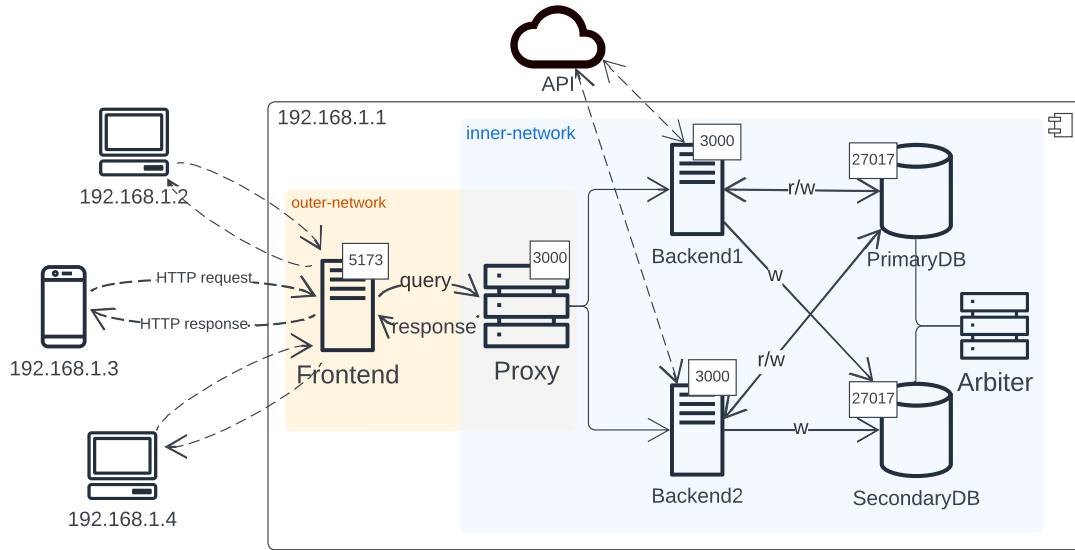


Figure 3: Component Diagram

3 Design

3.1 Architecture

We chose a Client-Server architecture with an implementation based on MVC (Model View Controller), highly used in centralized Web applications.

3.2 Infrastructure

As shown in Figure 3, RailSync will be distributed into several infrastructural components. Clients will access the application through their web browsers. A front-end web server will establish connections with these clients and forward their requests to a proxy server. This proxy will then distribute the workload between two back-end servers, which handle the core application logic. To ensure robust data management, these back-ends will connect to a replicated database consisting of two identical instances.

All servers will be deployed on the same physical machine leveraging dockerization, organized in two docker networks: outer (for backend communications) and inner. The frontend Web server will be accessible by multiple devices (clients) connected on the same local network of the physical machine.

3.3 Modelling

The application domain will contain users, reservations and travel solutions as main entities (Figure 4). Every entity will be stored in the database accessed by the web

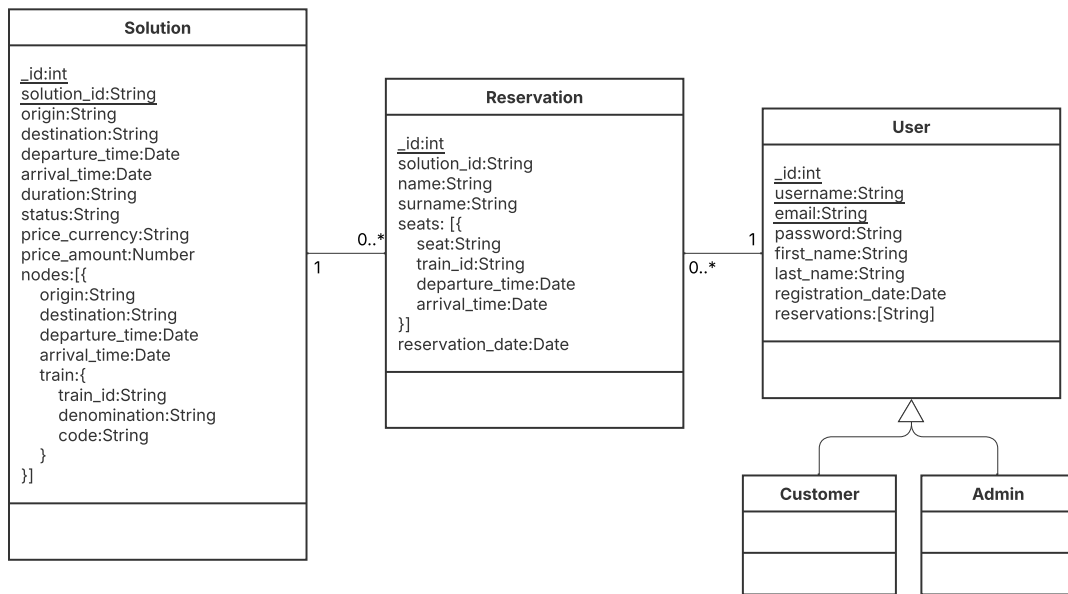


Figure 4: Class Diagram

server, in particular solutions will be collected using requests to third-party API services.

The domain will include many events, such as user registration and login, stations autocompletion, solution search and seat reservation. Additionally, admins will have the authority to remove users, solutions, and reservations, as well as grant administrative privileges to other users.

In the system, different types of messages will be exchanged. From clients to the server, commands (related to user profile, search, booking, and admin actions) and queries (such as retrieving available solutions or reservations) will be exchanged.

The system will manage data from two sources: it will use the user, reservation, and solution (with train data) stored in its internal database, while also fetching station and additional solution data from third-party APIs.

3.4 Interaction

As shown in Figure 5, clients interact with the server by sending commands and queries, the web server processes these requests (interacting with the database or the third party services, if necessary) and responds to the clients, following the request-response pattern. For example, when the user logs in or when it searches for trains.

During the real-time booking scenarios where multiple users are reserving seats of the same train, when one user books or even just selects a seat, all other clients see the real-time change and cannot select that seat following a polling pattern.

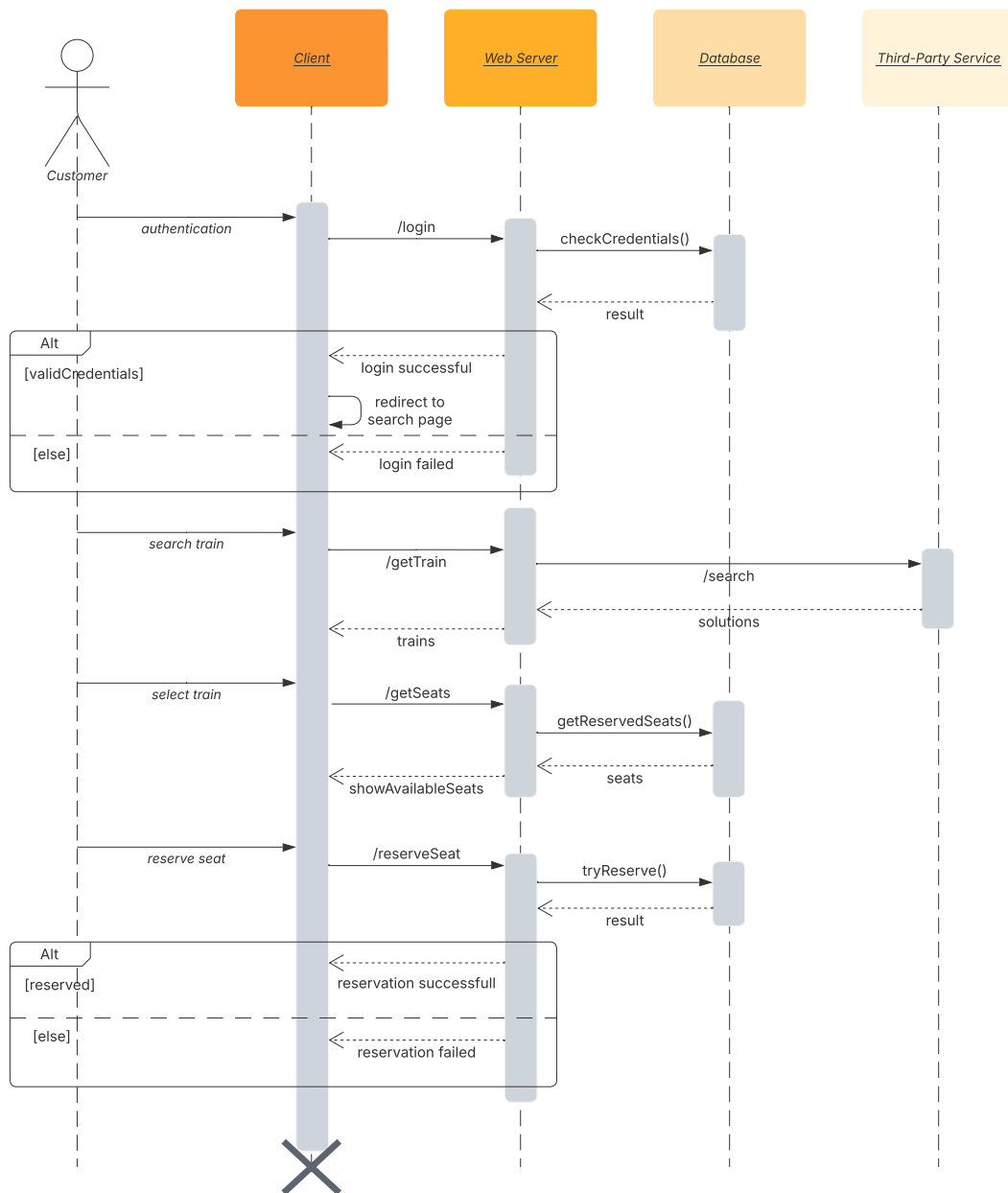


Figure 5: Sequence Diagram

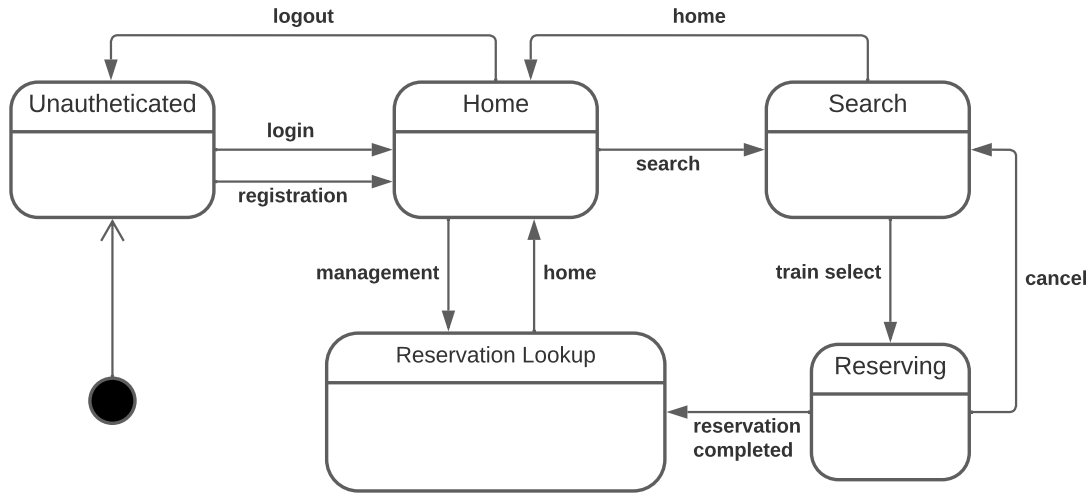


Figure 6: State Diagram

3.5 Behaviour

Each part of RailSync serves a unique function and responds distinctively to events and messages. The system contains both stateful components that maintain system information –like the database and session manager–, and stateless components that process requests without storing data, such as API endpoints.

In RailSync, state updates occur when certain events or actions are triggered by users (customers or admins) by changing user interface (Figure 6).

3.6 Data and Consistency Issues

Data storage is essential for several key pieces of information: user, reservation, solution, and train data. User data is necessary for managing accounts. Solution data is needed to track booking information. Reservation data is necessary to keep track of occupied seats on a booked train during a specific travel time.

Persistent data will be stored in a document-based database. The web server queries the database to retrieve user data and manage bookings. Queries occur during login, data retrieval, and booking updates. Concurrent access to a dedicated server is needed to handle multiple users accessing the same seat during a reservation.

Session information must be shared between the web server and clients to maintain user authentication and session state like the authentication token.

3.7 Fault-Tolerance

We use database replication to ensure that all data is securely backed up. The primary database has the read preference from the web server requests, while all the replicas receive all writes and updates, maintaining a reliable and up-to-date copy of the data.

Replicas continuously monitor the primary database using a heart-beating mechanism. Additionally, the web application attempts to connect to the database multiple times with a timeout before reporting an error if the database is unresponsive.

When the primary database fails, all the replicas plus the arbiter vote to elect a new primary database, restoring all the functionalities of the application.

Since a proxy is used with multiple web servers, if one server fails during a user session, the proxy automatically redistributes the load to the remaining healthy servers, enhancing the application's fault tolerance.

3.8 Availability

To ensure a high level of availability, the project adopts several complementary strategies. Firstly, a basic caching mechanism is implemented at the database level. This allows frequently accessed data to be retrieved more efficiently by leveraging the high speed of RAM, thus reducing latency and minimizing the load on the persistent storage layer.

A critical component of the system architecture is the load balancer, which distributes incoming requests across multiple servers. This approach not only enhances the scalability of the system but also prevents individual servers from becoming overwhelmed. The load balancing is realized through a proxy server that utilizes a round-robin policy to forward requests evenly.

Moreover, the network architecture is designed using partitioning: it is divided into two separate networks. This separation increases the security of the system, as it limits the potential impact of attacks by isolating different components and functionalities.

3.9 Security

Authentication is required to access any part of the application. Unauthenticated users are redirected to the login or registration page. The internal network, which handles the application's logic and sensitive data, exposes several APIs that are accessible only with a valid authentication token.

After authentication, authorization is enforced. For internal network APIs, the system verifies whether the user is an admin or the owner of the requested data, ensuring that sensitive information is protected and accessible only to authorized users.

Authentication is handled using tokens, each with a 24-hour expiration to improve session security. The system verifies tokens on each request to confirm user identity and permission.

To protect user credentials, all passwords are securely hashed before being stored in the database. This prevents the storage of plain-text passwords and mitigates the risk of data exposure in case of breaches.

4 Implementation

HTTP

Used for communication between a client (like a browser) and a server, handling requests (GET, POST, etc.) for fetching or sending data.

REST API

The backend follows REST architectural principles, exposing endpoints grouped by resources (e.g., /api/users, /api/reservations) and supporting standard HTTP methods for CRUD operations. REST APIs are also used to obtain official information about trains, stations and solution from trenitalia endpoints.

JSON

A lightweight data format for structuring data, commonly used for transferring data between the client and server due to its easy readability and integration with JavaScript.

NoSQL

A database model that stores data in non-tabular formats (like documents), suitable for unstructured or flexible data, providing scalability and performance, especially for large volumes of data.

JWT

Authentication is managed using JSON Web Tokens (JWT), which are generated based on the user's ID. Each token is issued upon login and must be included in subsequent requests to access protected resources, enabling stateless and secure user verification.

RBAC

Role-Based Access Control, a method for managing user permissions, ensuring that users can only access resources or perform actions based on their assigned role (e.g., admin, user).

4.1 Technological details

Docker

We use Docker and Docker Compose to efficiently build and run our web application, creating a separate container for each component: the frontend, proxy, each backend instance, database initializer, database arbiter, and each database replica. The frontend container is connected only to the outer network, the proxy one is connected to both the inner and outer networks, redirecting and balancing user traffic. All other components are isolated within the inner network, protected from external access.

MEVN

We implemented our web application using the MongoDB, Express.js, Vue.js, and Node.js (MEVN) architecture, as it offers a modern, efficient, and well-integrated full-stack development approach.

CORS

Cross-Origin Resource Sharing (CORS) is configured in the backend to allow resources on the backend to be requested from a different domain (like the frontend Vue app), ensuring controlled access.

NGINX

A NGINX proxy server is configured to forward API requests from the frontend to the backend server, allowing a load balancing and a certain level of fault tolerance.

Express Middleware

Express middlewares are used to process each requests, such as parsing JSON bodies (body-parser), validating input, authenticating users (e.g., JWT verification), and handling authorization in a centralized way.

Redis

Redis is used as an in-memory data store to manage high-concurrency operations, such as train seat reservations. To prevent race conditions—where multiple users attempt to reserve the same seat or modify the same cart simultaneously—a distributed locking mechanism is implemented. This ensures that only one process can modify a shared resource at a time, maintaining data consistency under concurrent access.

JEST

JEST is a testing framework for JavaScript. It helps check if code works correctly by running tests, catching bugs early, and making development more reliable.

5 Validation

5.1 Automatic Testing

Using Jest, we developed and executed a comprehensive suite of automated tests covering user-related functionalities such as registration and authentication, station retrieval via API, the booking process with proper validation, and Redis mutual exclusion logic to prevent race conditions.

To keep the tests organized and maintainable, we structured them into separate test files, each dedicated to a specific domain (e.g., `user.test.js`, `station.test.js`, `booking.test.js`, `redis.test.js`). Each file contains multiple granular test cases, ensuring both positive and negative scenarios are covered.

All tests can be run collectively by executing `npm test` within one of the two backend containers, or individually by running `npm test {testfilename}` to target a specific test file.

Regardless of the state of the database, each test file is designed to populate it with the specific entities it needs before execution.

Furthermore, the successful execution of the tests also implicitly verifies that the database connection is properly established, acting as an additional safeguard against connectivity issues.

Finally, a healthcheck is configured for the MongoDB container to ensure it is responsive before any test is executed. If the healthcheck fails repeatedly, Docker marks the container as **unhealthy**, though it does not automatically restart it unless managed externally. This mechanism helps to detect and isolate issues related to the database service early during testing.

5.2 Acceptance test

We performed several *manual* tests to validate system behavior in real-world scenarios. In particular, we manually tested the frontend functionalities to ensure that the user interface behaved as expected. This included verifying navigation, form validation, the booking workflow, and real-time updates, as well as ensuring that the layout remained consistent and responsive across interactions and different screen sizes.

Additionally, we manually tested the system's resilience to backend failures by simulating the crash of one backend container. We verified that the reverse proxy correctly rerouted the communication to the remaining operational backend. We simulated the failure of the primary database to verify that the replicas and the arbiter successfully elect a new primary, ensuring uninterrupted service continuity. Automating these kind of test would have involved dynamic container orchestration and real-time fault injection: they are more effectively assessed through direct observation during controlled testing.

We also manually tested the behavior of the system with respect to authentication token invalidation. Since each restart of the Docker environment caused the invalidation of previously issued tokens, we verified that the frontend handled these scenarios gracefully, prompting the user to log in again without crashing or showing errors. Furthermore, we tested how the system responds to deliberately corrupted or tampered tokens by

manually modifying them in the browser's local storage. This allowed us to confirm that the backend correctly rejects invalid tokens and that the frontend reacts appropriately.

6 Release

In summary, the project is organized as a set of interdependent modules, each corresponding to a distinct service within a Docker-based microservice architecture. These include two backend instances, a frontend application, a Redis instance for managing mutual exclusion, a MongoDB replica set (composed of primary, secondary, and arbiter nodes), and a reverse proxy managed by Nginx. Although all source code resides in a single Git repository, the services are logically separated and built independently using their respective Dockerfiles. Each backend container is functionally identical but instantiated separately to allow fault tolerance and load balancing. The proxy handles routing and redirection of traffic in case of backend failure, ensuring high availability.

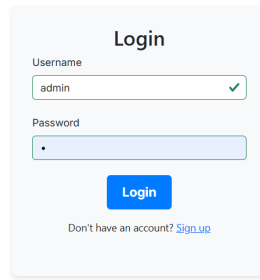
Distribution is handled via Docker images that are built and deployed together using Docker Compose. We opted to package everything into a single versioned repository rather than distributing separate archives. This choice simplifies orchestration and ensures full reproducibility of the environment, especially since the services are closely tied together.

Versioning is managed via GitHub, using a Git flow branching model with semantic versioning tags to mark releases. The final version of the system is tagged as **3.3.0**. Releases are not published on a package registry, but the entire project can be cloned from GitHub and run locally.

7 Deployment

After cloning the repository, the user must launch docker daemon using docker's shell commands or its GUI program. After that, the user initializes the MongoDB volumes launching the script called **FORCE_CREATE_NOVOLUME.sh**, and can then manage the full stack using one of the provided shell scripts. **start.sh** and **stop.sh** respectively launch and stop the full Docker Compose setup, while **restart.sh** calls both in order, ensuring ease of deployment.

If the application is hosted locally, all the users in the local network can access RailSync by connecting to the hosting machine at port 5173.



The image shows a login form titled "Login". It has two input fields: "Username" with the value "admin" and a green checkmark icon, and "Password" with a masked input (dots). Below the fields is a blue "Login" button. At the bottom, there is a link that says "Don't have an account? [Sign up](#)".

Figure 7: Login form

8 Application Guide

8.1 User

Upon connection, any user without a valid authentication token is prompted to either log into an existing account or register a new one. (Figure 7 and Figure 8)

After logging in, users are directed to the homepage, where they can search for trains by filling out a search form. For departure and arrival stations, users only need to type the beginning of the station names, which are autocompleted through API calls to the Trenitalia official endpoint. After selecting the departure time, users can click the "Search" button and view the real-time results displayed below. They can then click the "Continue" button on their chosen train option to proceed with the booking process. (Figure 9)

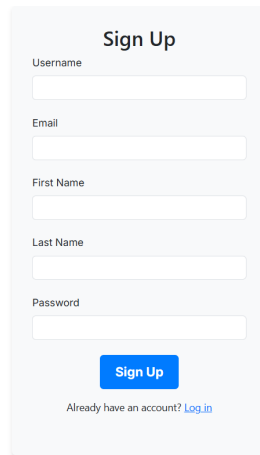
On the booking page, users select their preferred seat for each train in the chosen solution. Seats that are already occupied for the chosen train and time are displayed in red and cannot be selected. If a user is in the process of selecting a seat, it will also appear in red and not selectable to other users, preventing simultaneous selection.

Finally, users enter the passenger's information (name and surname), which is automatically pre-filled with their profile details by default. (Figure 10)

In the "Reservations" page, users will be able to see the details of their past bookings. (Figure 11)

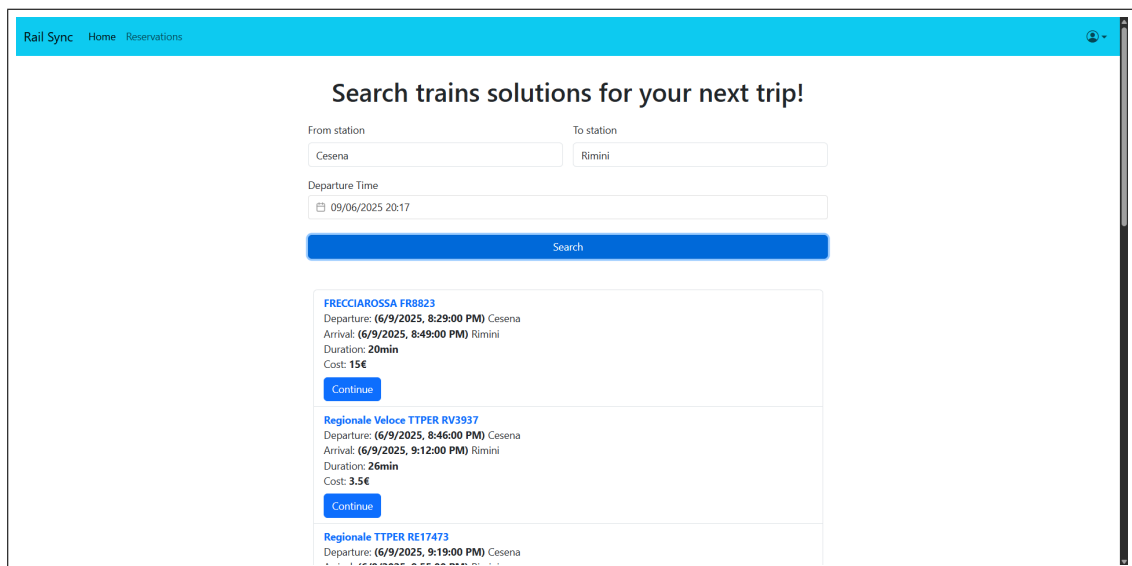
Clicking the profile icon prompts two actions (Figure 12): "Profile", which takes you to your personal profile page where you can update your account information, and "Sign Out", which logs you out of the website.

On the profile page (Figure 13), you can edit your username, email, first name, last name, and password.



A sign-up form titled "Sign Up" with a light gray background. It contains five input fields: "Username", "Email", "First Name", "Last Name", and "Password". Below the fields is a blue "Sign Up" button. At the bottom, there is a link that says "Already have an account? [Log in](#)".

Figure 8: Signup form



The homepage of the Rail Sync application. It features a blue header with the text "Rail Sync" and navigation links "Home" and "Reservations". A user profile icon is in the top right. The main heading is "Search trains solutions for your next trip!". Below this is a search form with "From station" (Cesena) and "To station" (Rimini) dropdowns, a "Departure Time" field (09/06/2025 20:17), and a blue "Search" button. The results section shows three train options, each with a "Continue" button:

Train	Departure	Arrival	Duration	Cost
FRECCIAROSSA FR8823	(6/9/2025, 8:29:00 PM) Cesena	(6/9/2025, 8:49:00 PM) Rimini	20min	15€
Regionale Veloce TTPER RV3937	(6/9/2025, 8:46:00 PM) Cesena	(6/9/2025, 9:12:00 PM) Rimini	26min	3.5€
Regionale TTPER RE17473	(6/9/2025, 9:19:00 PM) Cesena			

Figure 9: Homepage with search form

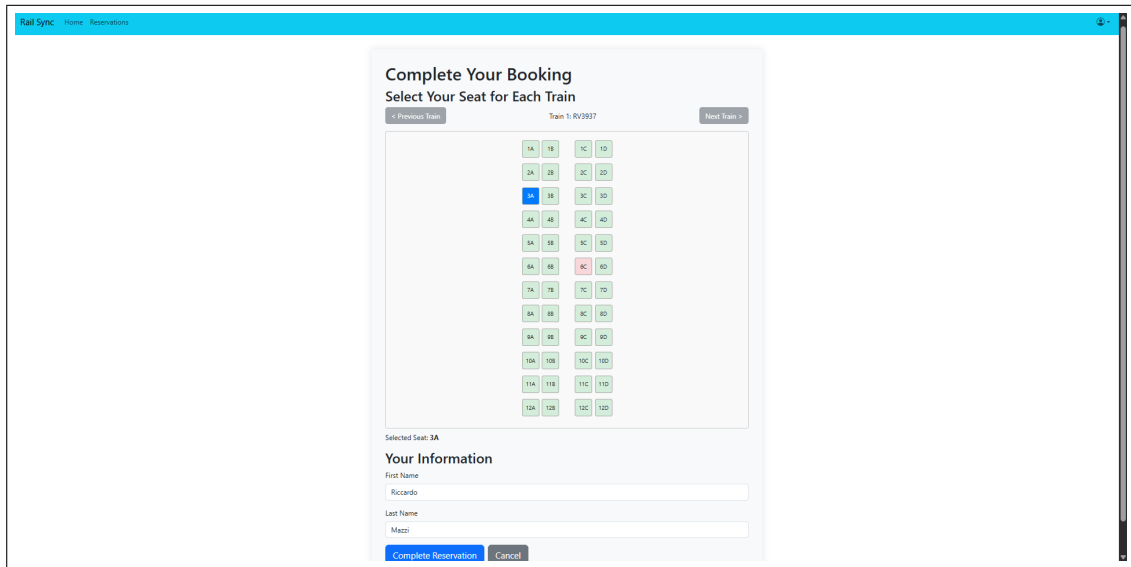


Figure 10: Booking page with seat selection

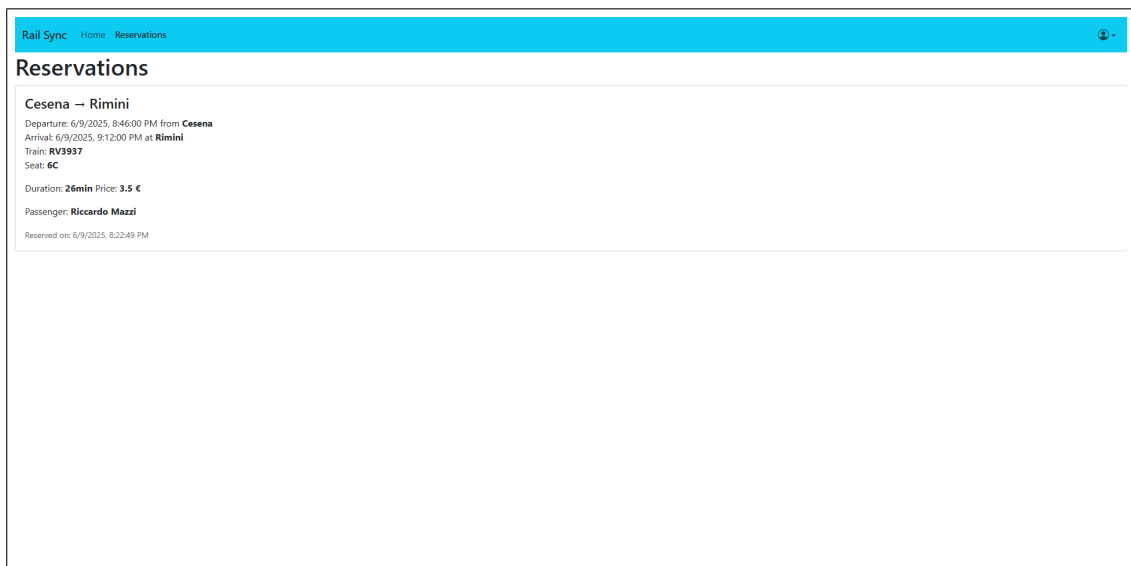


Figure 11: Reservations' details

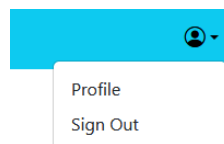


Figure 12: Profile actions

The screenshot shows a web application interface for 'Rail Sync'. At the top, there is a blue header bar with the text 'Rail Sync' and navigation links 'Home' and 'Reservations'. Below the header, the user's profile information is displayed in a form with the following fields: 'Username' (value: nicolas.amadori), 'Email' (value: nicolas.amadori@mail.com), 'First Name' (value: Nicolas), and 'Last Name' (value: Amadori). Each field has a green checkmark icon to its right, indicating it is valid. At the bottom left of the form, there is a blue 'Edit' button.

Figure 13: Profile page

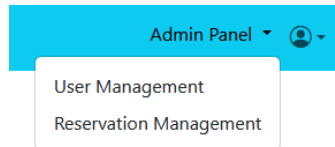


Figure 14: Admin actions menu

8.2 Admin

As an admin, the user gains access to an additional menu (Figure 14) with two options: User Management and Reservation Management. In the User Management section (Figure 15), the admin can view all regular users and delete their accounts if necessary. In the Reservation Management section (Figure 16), the admin can delete individual reservations or clear all train solutions saved in the database that haven't been reserved by any user.

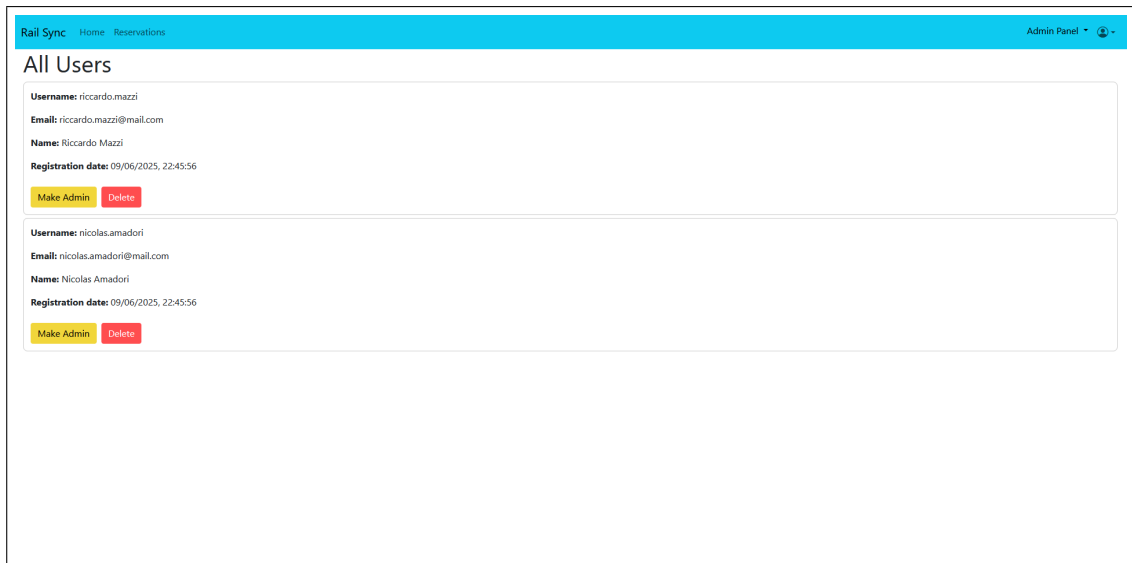


Figure 15: User Management

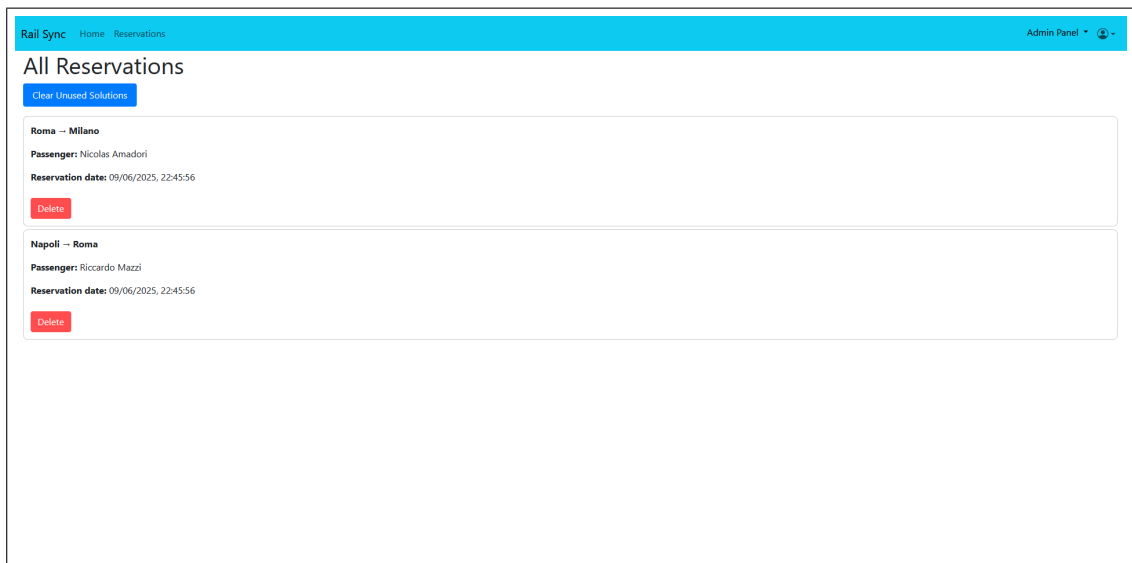


Figure 16: Reservation Management

9 Self-evaluation

9.1 Mazzi Riccardo

I believe RailSync is a well-rounded project, particularly strong in its distributed architecture. One of the most challenging aspects was learning how to effectively divide components using Docker and how to design and interact with APIs. Equally demanding were the implementations that aimed to strengthen the reliability of the system, such as setting up database replicas, deploying multiple back-end instances, and managing concurrency.

Looking ahead, there is definitely room to improve the user interface and overall usability, which would enhance the experience for end users.

After implementing the login system, I deployed the booking and reservation features, ensuring proper handling of occupied seats across different solutions and users. I set up database replicas along with the arbiter and initializer components. Additionally, I conducted booking tests and implemented other minor features, such as allowing admin users to grant admin rights to others.

I believe Nicolas and I worked together efficiently, collaborating in parallel and successfully merging our work without conflicts, even between ourselves.

9.2 Amadori Nicolas

I am very happy with my contribution to the project, as I believe I have successfully developed critical components that significantly enhance both the functionality and reliability of the system.

I was responsible for the implementation of the authentication and authorization system using JWT, including the development of custom Express middleware to secure private routes and enforce user roles.

I also implemented comprehensive backend tests for API endpoints related to user management, station autocompletion, and concurrent seat reservation, ensuring correctness under edge-case scenarios.

On the frontend, I implemented the registration page and the homepage that contains the train solution search feature, which includes an autocomplete system for departure and arrival stations powered by official Trenitalia APIs.

I also set up a proxy server to enable load balancing across backend services, improving the system's scalability and fault tolerance.

Additionally, I integrated a Redis container to handle concurrent booking attempts, ensuring atomic operations and preventing multiple users from reserving the same seat simultaneously.

These contributions addressed key challenges commonly found in distributed systems, such as secure access control, load distribution, and consistency in concurrent operations.

I am very pleased with the work Riccardo and I have done, as we were able to collaborate effectively and find solutions to the challenges we encountered, ultimately developing a complete and robust distributed application.