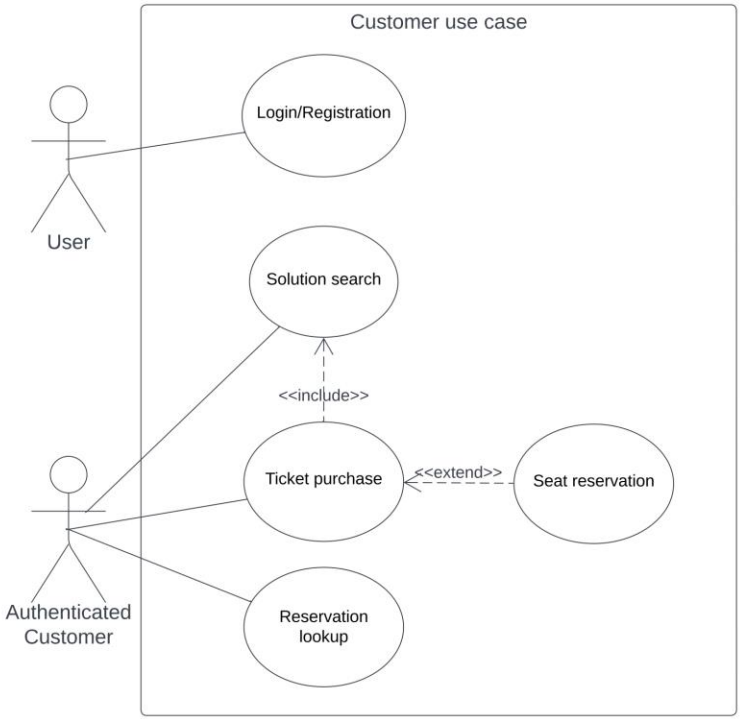
RAILSYNC

DISTRIBUTED TRAIN BOOKING

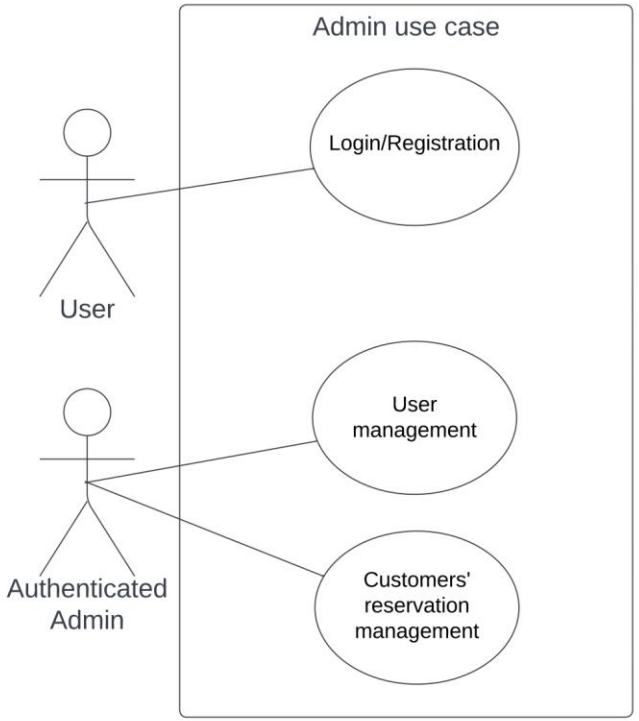
NICOLAS AMADORI, RICCARDO MAZZI

CONCEPT

CUSTOMER CASE SCENARIO



ADMIN CASE SCENARIO



REQUIREMENTS

Functional

- Login and registration
- Multiple role access
- Account and personal settings
- Homepage with search functionality
- Ticket purchase with seat reservation
- Personal reservation details
- User management (for admins only)
- Customers' reservation management (for admins only)

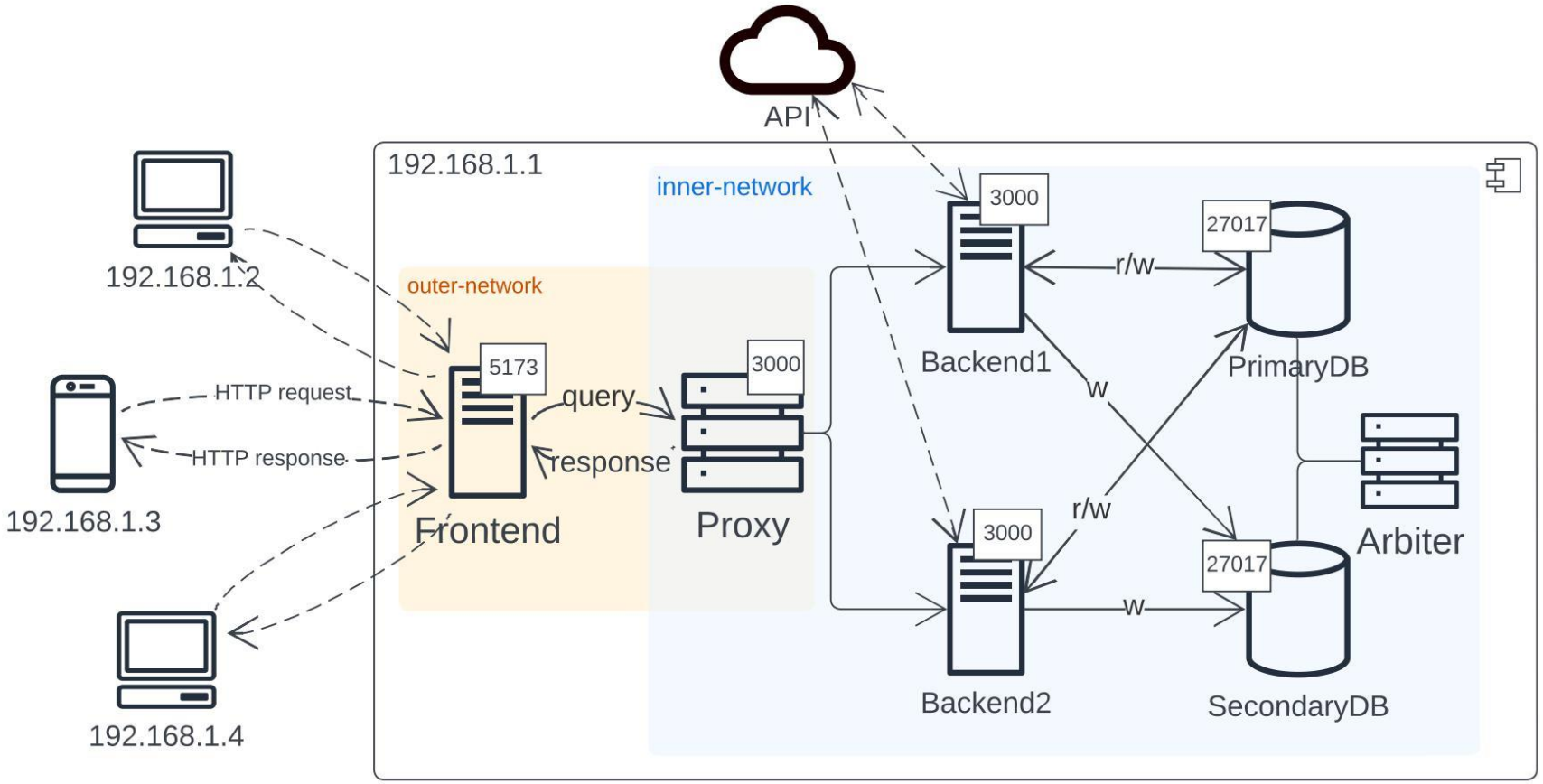
Non-functional

- Concurrent access to trains seats during reservation
- Usability for inexperienced users
- Availability of the service h24
- Load balancing of the requests
- Fault tolerance
- High-security system

Implementation

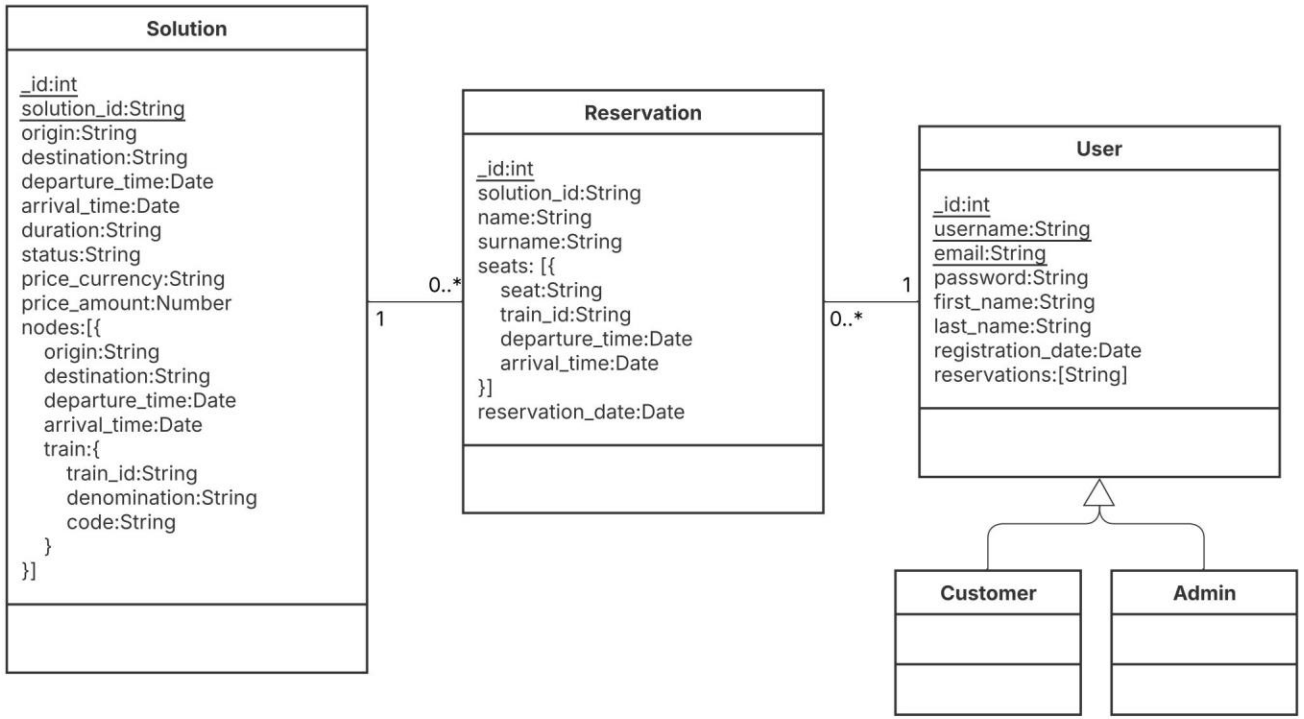
- RESTful APIs support for third party integration
- MongoDB, Express.js, Vue.js, Node.js (MEVN) architecture
- NoSQL database support for booking and user management
- Reactive Web application for cross-platform access
- Docker-based structure

INFRASTRUCTURE



- **Network partitioning**: separates internal and external networks for enhanced security and isolation

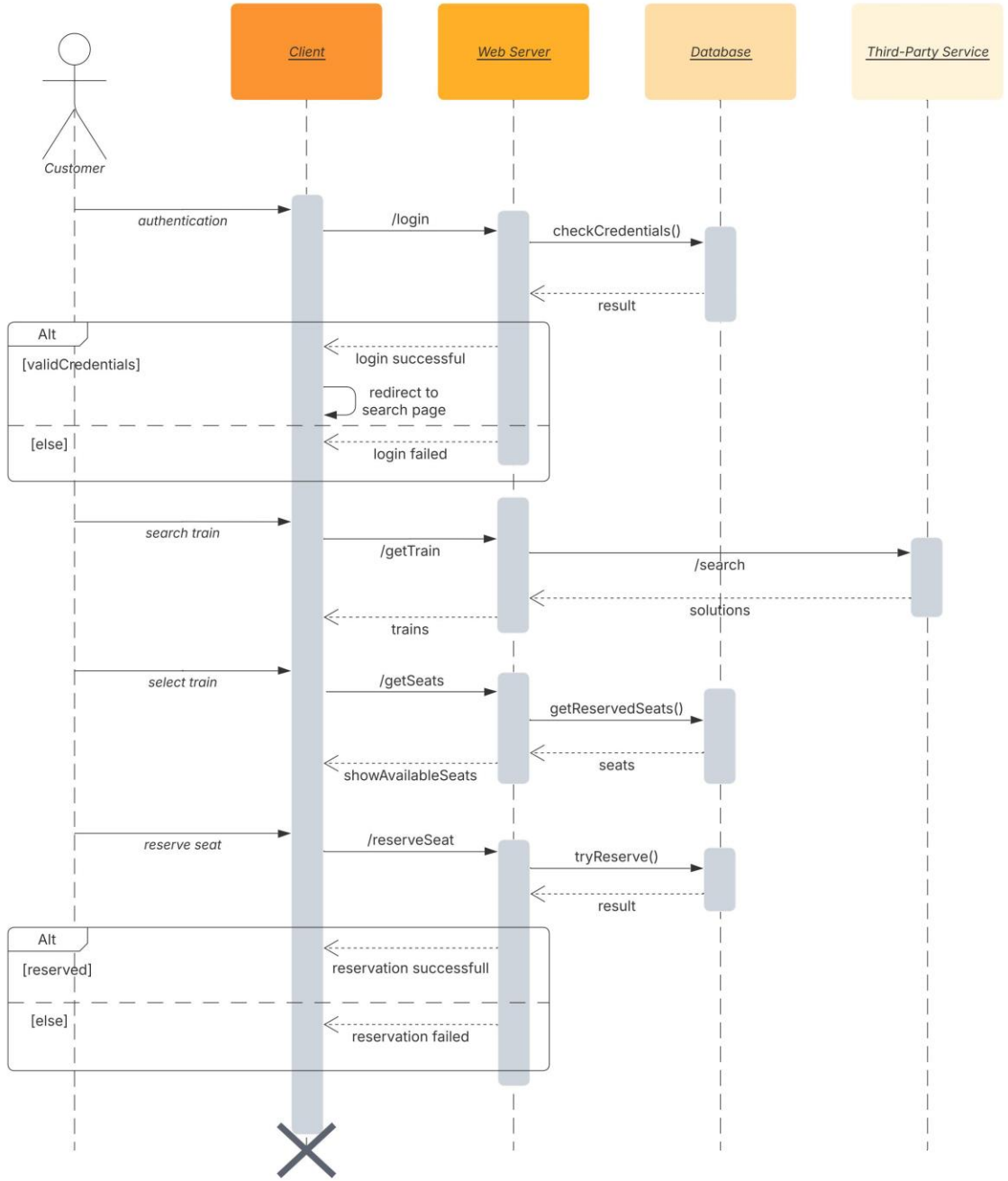
MODELLING



- **Entities**: Users, Reservations, Travel Solutions
- **Data sources**: Internal DB + external APIs (stations, solutions)
- **Events**: Registration, login, search, booking
- **Admin** actions: Manage users, data, and roles
- **Messages**:
  - **Commands**: actions like booking or deleting
  - **Queries**: fetch data (solutions, reservations)

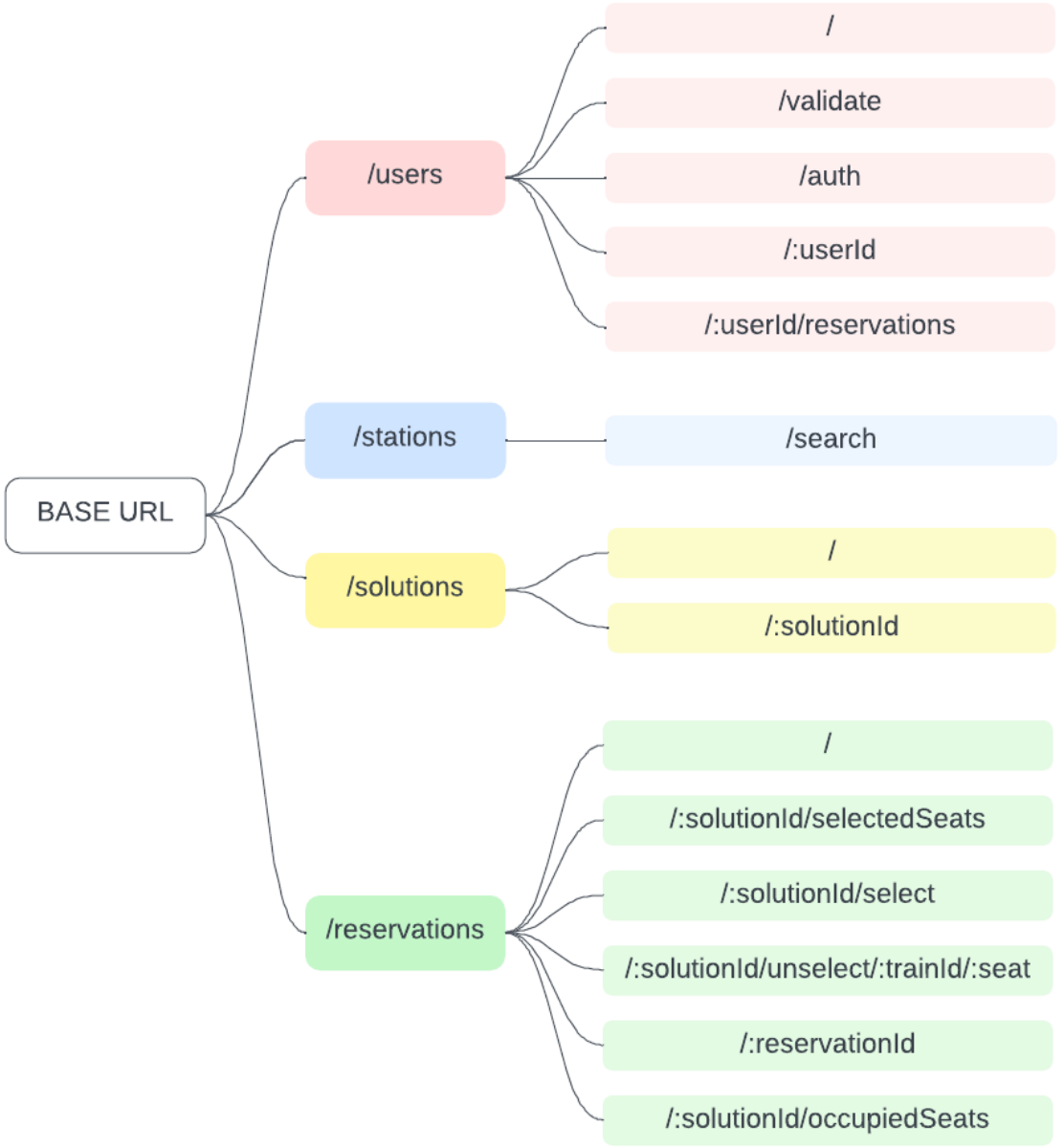
INTERACTION

- DB & third-party interactions handling
- Real-time booking: seat selection updates for all users
- **Polling** pattern to sync seat availability



RESTFUL APIs

- APIs offers access to backend **system resources** from the frontend application, ensuring **authorization** through authentication token.
- Each endpoint exposes one or more of these **HTTP methods**:
  - GET
  - POST
  - PATCH
  - DELETE

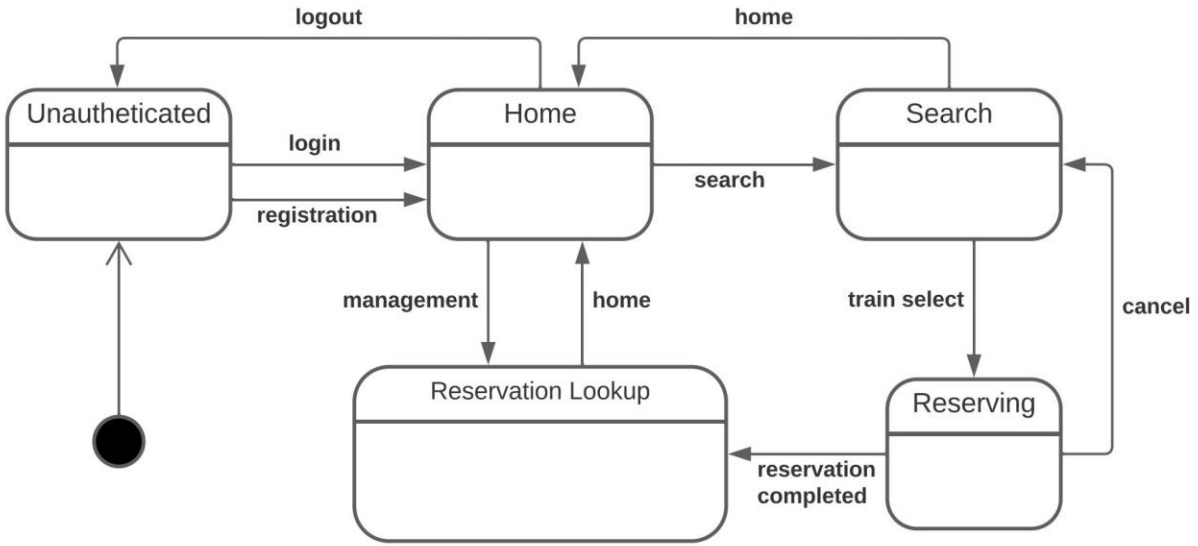


TRENITALIA APIs

- We used official Trenitalia APIs (**lefrecce**) to get **real-time** data to:
  - autocomplete stations name
  - find possible solutions for the user's research using specified departure/arrival stations and departure time

https://www.lefrecce.it/Channels.Website.BFF.WEB/website/locations/search?**name=ces**

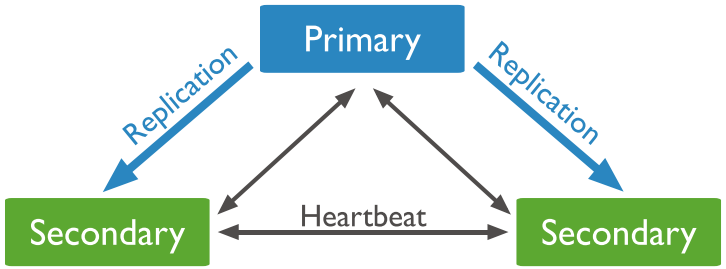
BEHAVIOUR



- **Stateful**: DB, session manager (track system info)
- **Stateless**: API endpoints (process w/o storing)
- **State updates**: triggered by user/admin actions
- Changes happen via user interface events

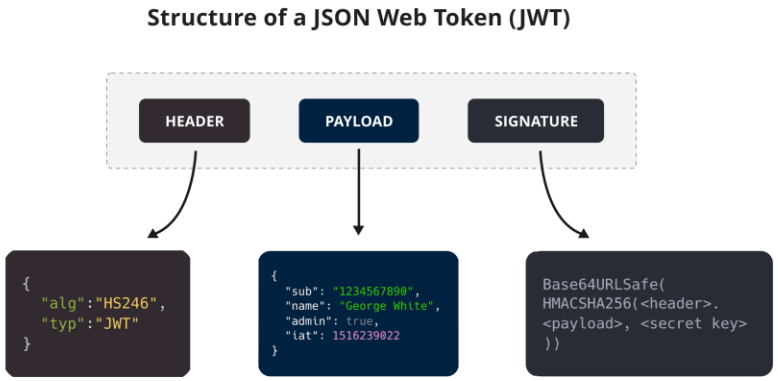
DATABASE FAULT-TOLERANCE

- **Database replication** ensures secure backups and high availability.
- **Primary DB** handles reads; **replicas** sync all writes/updates.
- **Heartbeat monitoring** detects primary DB failure.
- On failure, **replicas + arbiter elect a new primary** automatically.
- Web app retries DB connection before showing errors.
- **Database caching**: speeds up frequent queries, reduces DB load



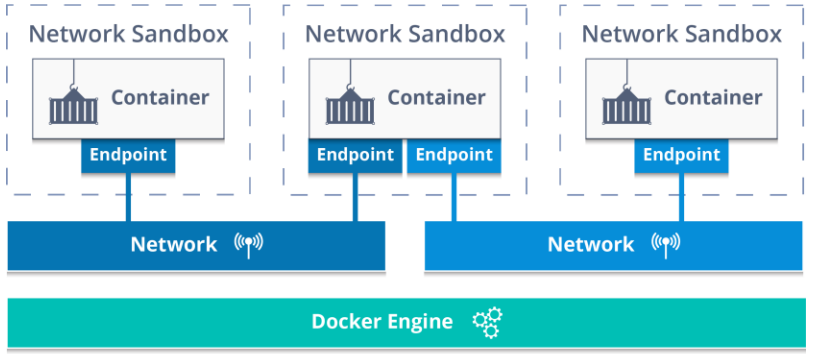
SECURITY

- **Authentication required** for all app access
- **Token-based auth** (JWT Token) with 24h expiry, verified on each request
- **Authorization (RBAC)**: only admins or data owners can access sensitive APIs
- **Internal APIs** protected via token and network isolation
- **Passwords hashed** before storage – no plain-text credentials



DOCKER

- We use **Docker & Docker Compose** to manage our web app components.
- Each part (frontend, proxy, backend, DB, etc.) runs in a **separate container**.
- **Network isolation**:
  - *Frontend* → outer network only
  - *Proxy* → connects inner & outer, manages traffic
  - *Others* → inner network only, **not externally accessible**

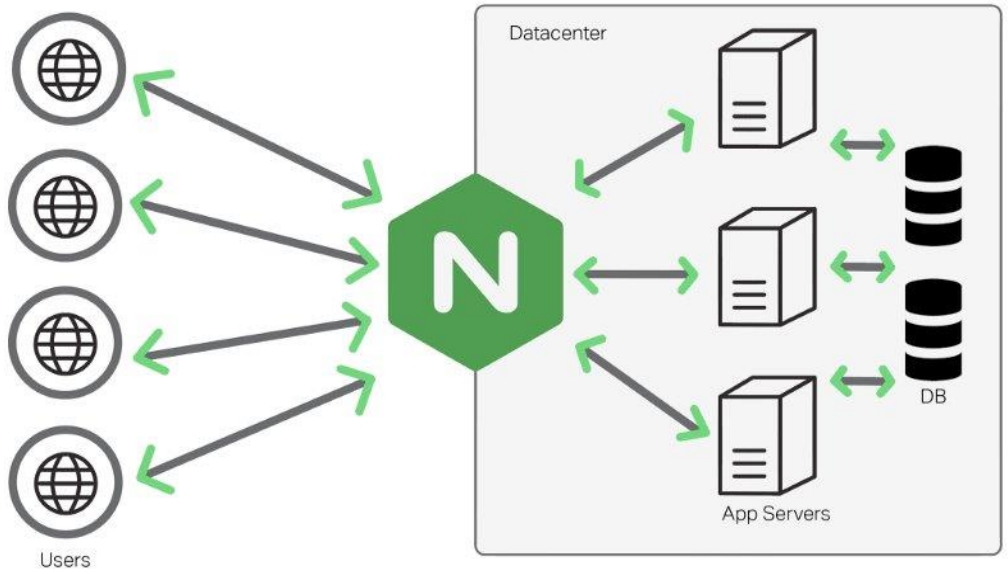


KEY TECHNOLOGIES

- **MEVN**: Full-stack app built with **MongoDB, Express.js, Vue.js, Node.js** for seamless and modern development.
- **CORS**: Backend allows secure cross-origin requests from the frontend.
- **Express Middleware**: Handles parsing, validation, authentication (e.g., JWT), and authorization.
- **Redis**: Used for **high-concurrency operations** with distributed locks to avoid race conditions (e.g., seat reservations).
- **JEST**: Ensures code reliability through automated **unit testing**.

LOAD BALANCING

- Implemented using an **NGINX proxy server**.
- **Distributes incoming requests** across two backend instances using **round-robin.**
- Enables **scalability** and **fault tolerance**.
- Ensures efficient use of resources and reduces server overload.
- Transparently handles **API routing** from the frontend to the backend.



AUTOMATIC TESTING

- Tests built with **Jest**, covering user, station, booking, and Redis logic.
- Organized in **separate test files** by domain (e.g., user.test.js, booking.test.js).
- Tests **populate the database** before running, ensuring independence.
- Tests verify **database connectivity** implicitly.
- **MongoDB healthcheck** ensures DB responsiveness before tests; Docker marks unhealthy containers.

DEPLOYMENT

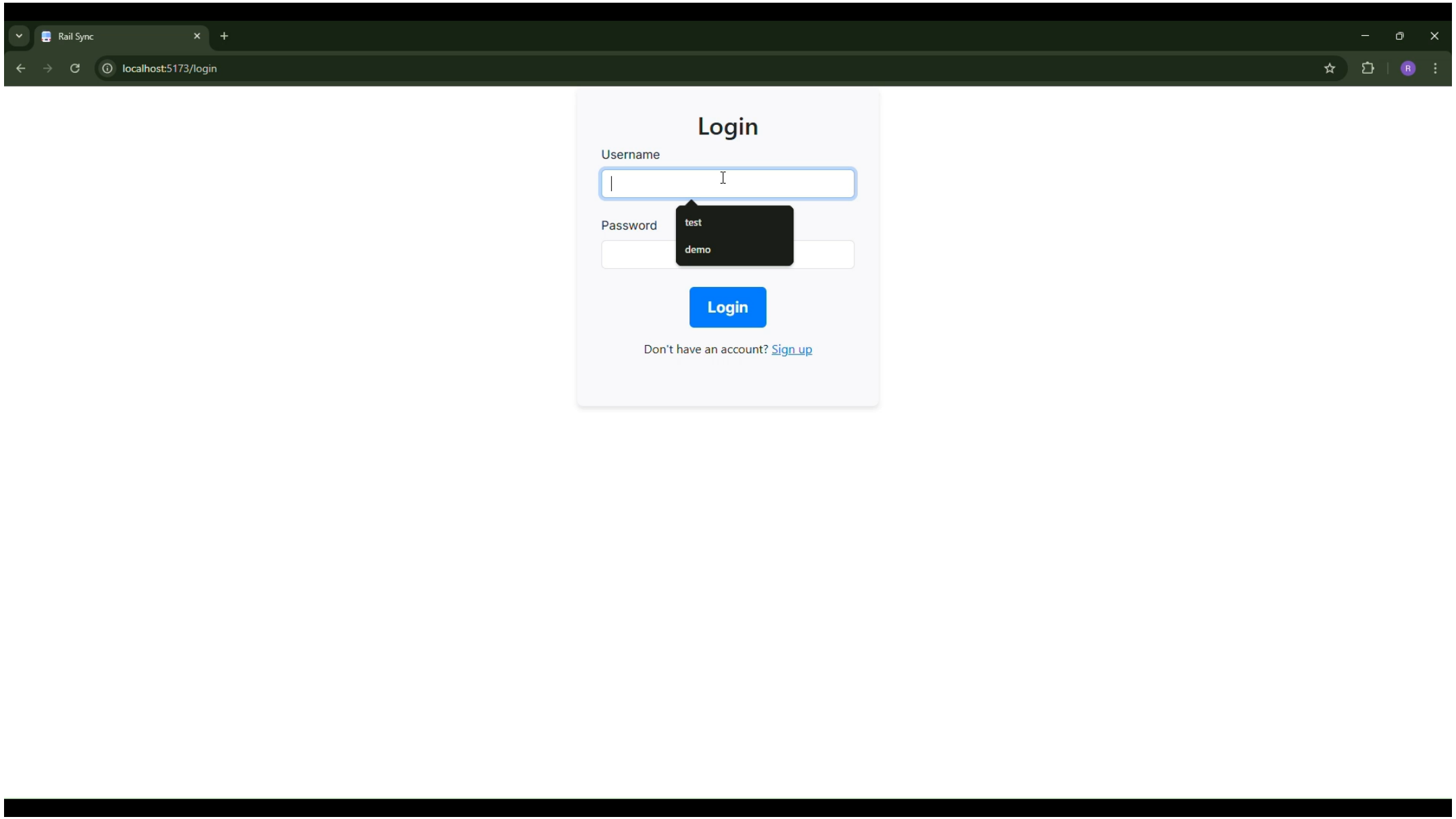
- Start by launching the **Docker daemon** (via shell or GUI).
- Initialize MongoDB volumes with the script: `FORCE_CREATE_NOVOLUME.sh`.
- Manage the full stack using provided scripts:
- `start.sh` to launch
- `stop.sh` to stop
- `restart.sh` to restart the setup.
- If hosted locally, users access the app at: `http://<host-machine>:5173`

DEMO

NICOLAS AMADORI

RICCARDO MAZZI

DEMO


Rail Sync
localhost:5173/login
Login
Username
Password
test
demo
Login
Don't have an account? Sign up