Scooter Monitor - Final Report

Nicola Ferrarese
nicola.ferrarese@studio.unibo.it

August 2024

This report provides an in-depth overview of the development process for a single-page application (SPA) specifically designed for the reservation and utilization of electric scooters. The primary objective of this project was to create an application that serves as a Minimum Viable Product (MVP) for a comprehensive urban mobility solution.

The application is intended to offer a seamless and user-friendly experience, enabling users to efficiently locate, reserve, and use electric scooters for their urban transportation needs. Through the application, users can access an interactive map that displays all available scooters in their vicinity, providing real-time information on the scooters' locations and availability.

The application includes robust monitoring features that allow users to track and analyze their scooter usage. These functionalities provide detailed statistics, such as the duration of scooter use, distances traveled, and other relevant metrics. This data can help users better understand their usage patterns and make more informed decisions regarding their mobility options.

Contents

1 Goal/Requirements

The primary goal of the ScooterMonitor project is to develop an intuitive and user-friendly application that provides real-time status and location information for a fleet of scooters, along with detailed usage statistics. The key requirements for the system are:

- *Intuitive and easy access to the application*: The user interface must be simple and user-friendly, even for less experienced users.
- *Real-time visualization of individual scooter status and location*: The application must provide an interactive map displaying the current location and status of scooters.
- *Display of individual scooter usage statistics*: Each user should be able to view usage statistics for the scooters they have used.
- *Display of overall scooter usage statistics*: The application should offer aggregated statistics on the usage of all available scooters.
- *Account creation and management*: Users must be able to create new accounts, modify their information, and manage their reserved scooters.
- *Scooter locking and unlocking*: Users must have the ability to lock and unlock scooters through the application.

The expected outcomes of the project are a fully functional and user-friendly software system that meets the outlined requirements, as well as the consolidation of knowledge and skills related to developing intuitive, real-time tracking applications.

1.1 Scenarios

The ScooterMonitor system is designed to provide users with real-time status and location information for a fleet of electric scooters. Users can access this information through a web-based interface, which allows them to monitor scooter availability, analyze usage statistics, and make informed decisions based on accurate and up-to-date data. The following are the use scenarios for the system:

1.1.1 User Types and Their Specific Requirements

The primary users of the application are divided into three main categories: urban citizens, charging and maintenance staff, and market analysts and operators. Each user category has specific use cases and derives different value from the application's features.

Urban Citizens

- **Use Case**: Urban citizens use the application to book and use electric scooters for short trips within the city. They can view available scooters on a map, book a scooter, and track their journey in real-time.

- **Value of Information**: For urban citizens, the most valuable information includes the availability and location of scooters, battery status, and the ability to view trip routes and duration. This information enhances the user experience by making the service more reliable and easy to use.
- **Added Value**: End users find value in the application for its ability to provide a fast, eco-friendly, and convenient means of transportation. The application facilitates scooter access, reducing wait times and improving urban mobility.

Charging and Maintenance Staff

- **Use Case**: Charging and maintenance staff use the application to monitor the status of scooters, plan charging and maintenance operations, and respond promptly to breakdowns or technical issues.
- **Value of Information**: For these users, having access to detailed information on the status of each scooter, including battery levels, maintenance history, and fault notifications, is crucial. This information allows for more efficient fleet management and improves service quality.
- **Added Value:** The application offers value to charging and maintenance staff through advanced monitoring tools and real-time notifications, facilitating operational management and reducing scooter downtime.

Market Analysts and Operators

- **Use Case**: Market analysts and operators use the application to monitor service performance, analyze usage data, and make informed decisions to improve service efficiency and effectiveness.
- **Value of Information**: For these users, the value lies in access to aggregated and analytical data, such as usage rates, movement patterns, user feedback, and operational KPIs. This data is essential for optimizing fleet management, identifying areas for improvement, and planning future expansions.
- **Added Value**: The application provides added value through customizable dashboards and detailed reports that support strategic decisions, enhancing service competitiveness and sustainability.

1.2 Use Scenarios

- **Registration and Login**: To access the features of the ScooterMonitor system, users are required to register and create an account. Once registered, users can log in to their account using their credentials.

- **Viewing the Data Dashboard**: The primary purpose of the ScooterMonitor system is to provide users with real-time and historical data about scooter availability and usage. Users can view this data on the dashboard, which includes information such as scooter location, battery status, and trip history.
- **Viewing Scooter Status and Location**: The system provides an interactive map displaying the current location and status of scooters. Users can see which scooters are available for use, their battery levels, and other relevant details.
- **Viewing Usage Statistics**: Users can view usage statistics for the scooters they have used, as well as aggregated statistics for all scooters. This includes data on trip duration, distance traveled, and usage frequency.
- **Scooter Locking and Unlocking**: Users must have the ability to lock and unlock scooters through the application, ensuring secure access and usage of the scooters.

The expected outcomes of the project are a fully functional and user-friendly software system that meets the outlined requirements, as well as the consolidation of knowledge and skills related to developing intuitive, real-time tracking applications.

2 Requirements Analysis

2.1 Implicit Requirements

Upon analyzing the project requirements, some implicit requirements were identified:

- A simulated Scooter that resembles a real one has to be implemented.
- The system should be able to handle multiple users simultaneously accessing the web application.
- The system should provide real-time updates to the web application interface that will match the real-time status of all device and their current position/status.

2.2 Implicit Hypotheses

The project requirements also make some implicit assumptions:

- The Scooters will be continuously transmitting data but may fail and stop sending data at some point.
- The system will have a reliable network connection between the Scooters, message broker, and web application.

2.3 Non-Functional Requirements

In addition to the explicit functional requirements, there are also some non-functional requirements implied by the project which will have to be taken into account during the design phase:

.The system should have low latency and high availability to ensure real-time data processing and display. The system should have an authentication system.

2.4 Structure

2.4.1 Entities

Based on the end goal and the requirements identified in the previous sections, the key components that need to be modeled in the system are:

- **Scooter**: Represents the electric scooter with all its relevant properties, including GPS location, battery status, device ID, and availability status (free, in use, under maintenance).
  Each scooter has been modelled as a single thread, with it's own. idependent control flow. They communicate to an edge server recieving commands and sending updates utilizing Apache Kafka.
- **MessageBroker and MessageConsumer**: Those entities has to be modelled and implemented both in the scooters and in the backend, in order to allow communication between the two entities

- **Database**: Stores the collected scooter data for historical analysis and retrieval. It also stores user data and user request data.
- **WebServer**: Incapsulate all the logic to not only handle updates and send commands to the devices while updating and fetching information from the MongoDB database
- **WebClient**: Client for the web application that allows users to view and interact with the system, see the real-time status of all devices and have access to specific metrics for each scooter.

2.5 Technology stack choices

The chosen technologies for the development of the system are the following:

- **Apache Kafka** as the message broker. Kafka was chosen for its high throughput and scalability, making it ideal for handling large volumes of data.
- **Node.js + Express.js** for the web application back-end development. Node.js was chosen because it is a lightweight and scalable platform for building server-side applications, and its non-blocking I/O model is ideal for handling multiple concurrent connections. Express.js was chosen as the web framework because it is simple, flexible, and widely used in the Node.js community.
- **Vue.js** for the web application front-end development. Vue.js was chosen because it is a lightweight and easy-to-learn framework for building dynamic user interfaces. It also has good performance and a growing ecosystem of plugins and tools.
- **MongoDB** as the database. MongoDB was chosen because it is a scalable and flexible NoSQL database that can handle large volumes of unstructured data. It also has good performance and supports rich querying capabilities. On top of that, its change streams feature is very suitable for delivering real-time updates.
- **Docker-compose** for the container orchestration. Docker-compose was chosen over Kubernetes because it is simpler to use and more suitable for small-scale deployments. It also integrates well with Docker and allows for easy management of multi-container applications.

2.6 Components

The system architecture overview shown in Figure 10 illustrates the high-level components of the ScooterMonitor system and their interaction protocols. Each of the components is deployed as an individual Docker container, allowing for easy scalability, development, and management of the system.

The scooter and consumer scripts are implemented using Python and are responsible for collecting and processing scooter data, respectively. The scooter script collects data

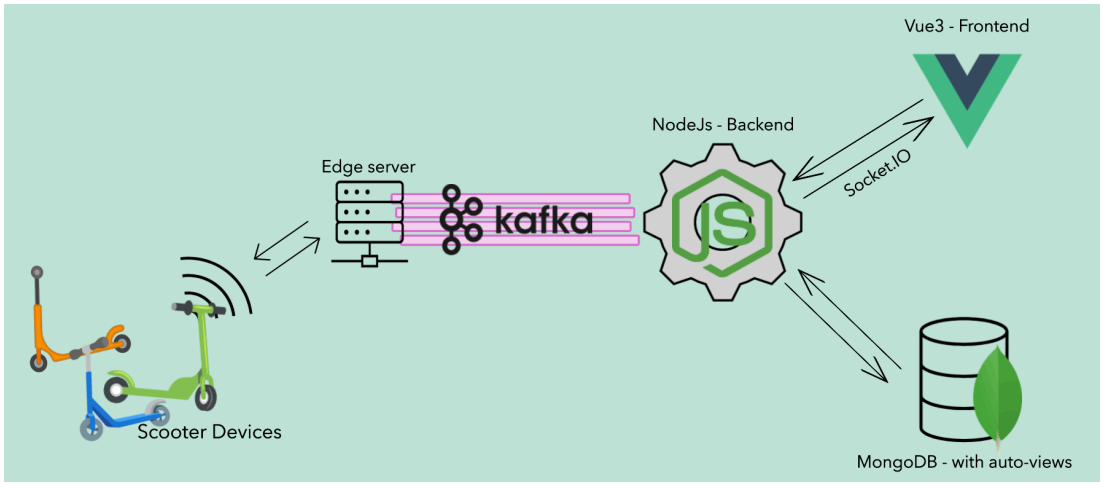

Figure 1: System architecture overview

from the scooters and sends it to the Kafka message broker. The consumer script receives the data from the message broker, processes it, and stores it in the MongoDB database.

The web application is composed of a frontend and a backend, implemented using Vue.js and Node.js with Express.js, respectively. The frontend provides a user interface for visualizing and analyzing the collected scooter data, while the backend provides a RESTful API for the frontend to interact with the system.

The Kafka message broker acts as an intermediary between the scooter and consumer scripts, allowing for asynchronous communication and decoupling between the components. This enables the system to handle large volumes of data and provides fault tolerance in case of component failures.

The MongoDB database is used to store the collected scooter data and provides a scalable and flexible data storage solution. The database is designed to handle large volumes of data and provides efficient querying and indexing capabilities.

The use of Docker containers for deploying the components provides a scalable and portable solution for the system. Each component is deployed as an individual container, allowing for easy management and scaling of the system. The containers are connected using a local network, providing a simple and efficient communication between the components.

2.6.1 Frontend - Vue

The frontend of the application is developed using Vue.js, a progressive JavaScript framework for building user interfaces. The technologies and principles used include:

- **State Management**: Application state management is handled using Vuex, the official state management library for Vue.js. Vuex allows for predictable and organized central state management, facilitating information handling among various components.

- **SCSS**: SCSS (Sassy CSS) is used for styling. SCSS is an extension of CSS that allows the use of variables, nesting, mixins, and other features that make CSS code more modular and maintainable.
- **Component Modeling**: The application is divided into modular and reusable components. Each component represents a specific part of the user interface, such as the map, scooter details, and statistics. This approach facilitates development, maintenance, and future expansion of the application.
- **Responsiveness and Dark Mode**: The application design is responsive, ensuring a good user experience on devices of various sizes. Additionally, a dark mode is implemented to enhance usability in low-light conditions and reduce eye strain.

2.6.2 Backend - Node

The backend of the application is developed using Node.js, a JavaScript runtime platform that enables building scalable and performant server-side applications. The connection between the frontend and backend is realized through various middleware, including:

- **Socket.io for Frontend**: Socket.io is used for real-time communication between the frontend and backend. This allows the map and scooter information to be updated instantly without reloading the page.
- **MongoDB Operations**: MongoDB is used as the database for data management. CRUD (Create, Read, Update, Delete) operations are managed via MongoDB middleware, ensuring efficient interaction with the database.
- **Kafka Consumer/Producer**: Kafka is used for asynchronous communication between the backend and scooters. Each scooter is modeled as a thread worker in a separate container, communicating with the backend via Kafka. This approach ensures high scalability and reliability of the system.
- **Security** To ensure application security, JWT (JSON Web Token) is implemented. JWT is a secure method for representing information compactly and independently between the client and server. It is used for user authentication and authorization, ensuring that only authorized users can access certain features and data of the application.

2.6.3 Kafka - Metrics Retrieval

Kafka is used mainly to allow communication between the Scooter devices and the Backend, allowing Scooters to send updates circa their position and status and to recieve commands such as block/unblock from the users.

2.6.4 MongoDB

MongoDB is the chosen database for its flexibility and scalability. The main functionalities implemented include:

- **Views**: Views in MongoDB are used to create reports and display detailed trip statistics. This allows users to explore data related to their trips and obtain useful information to improve the service's efficiency and effectiveness.

2.7 Behaviour

- **Scooter**: Scooters are modelled as reactive entities. Each Scooter object (that model the real, physical, device) connects with the infrastructure via a Kafka producer and consumer.
  By design, each scooter has it's own state, that define *where* the scooter is and *what* the device is doing, if it's blocked and waiting or if it's being utilized.
  The scooters always streams via *position* Kafka's topic their actual position, and routines in the Back-end ensure that the information is reflected both in the database and in the frontend.
  The scooters responds to commands, in particular, once an user is successfully logged and if the device is not being used, the scooter it's unlocked and the ride starts.
  Why does the Scooter needs to know when it's unlocked? In order to make the scooter simulation realistic, once an user "starts" a ride, the scooter, that internally has a graph-structure of a given city, will pick a random destination and will start to move towards it, since the position and the scooter's status are being published on Kafka topic, this will allow to have near-real time data of all the scooters inside the application.

- **Message Broker**
  Kafka per-se is just a message broker middleware that's been utilized to permit communication between the scooters and the back-end. The communication happens in two separate and independent ways:
  - Scooter to Application: As presented above, in any given time, the scooters stream their status and position as well as trip information to the backend, witch will update accordingly the database.
  - Application to Scooter: Once users wants to start or stop a ride, these commands will be published and sent to the specific scooter,witch will react accordingly

- **Database**:
  The database for the application is implemented using MongoDB, a popular NoSQL document database. MongoDB was chosen for its flexibility, scalability, and for the ability of compute materialized view over the collections.
  The database schema consists of the following collections:

- **Scooter**:
  Each scooter is saved together with it's attributes, such as current position, battery remaining, last block/unblock, it's availability and if it's being used, the trip ID of the ride is being performing.
- **Users**:
  Each users is stored together with username, ID, hashed password and current ride if it's riding.
- **TripSegments**:
  Due to the simulated nature of the project, the scooters when riding emits what has been chosen to call "trip segments". In order to "navigate" a Scooter needs to move in a graph structure, where each node represent an intersection between roads. After each Intersection that the scooter passes, a new trip segments containing segment distance, duration, timestamp and Trip ID is produced, sent via Kafka and stored, thanks to Node.JS in the Database.
- **Trips**:
  Leveraging the views functionality offered by Mongo, TripSegments collection is used to create this view, where, for each Trip ID, information regarding total distance, duration, cost, and ownership (scooter and user) can be accessed.

- **Web-Server**
  Heart of the application, the Back-end server is implemented using Node.js. At application Start-up it waits to be connected to both the database and the message broker, prior to start serving requests. It's main functionalities can be enclosed in the following areas:
  - **Front-end Communication**: Every User interaction passes through the front-end, moreover, many of the non functional requirements relies in a near-real-time coupling between the state of the application itself (considered including the scooters) and the map representing all the scooters that is always visible to the user. To achieve such tasks, all the communication between the front and the backend is dealt using Socket.IO for asynchronous message passing.
  - **Scooter Communication**: in order to control and perceive the scooters, communication in ensured using Kafka's consumer and producers, with callbacks logic to handle properly the messages.
  - **Database Communication**: After every interaction with the user or after a received message from the scooters, data inside the database has to be updated or fetched. To achieve such functionalities Mongoose has being chosen, to allow for streamlined collections handling.

2.8 Interaction

In Figure 2 a diagram showing the logic of the application and the user's interaction paths. Once accessed via browser, the application presents itself showing a map with

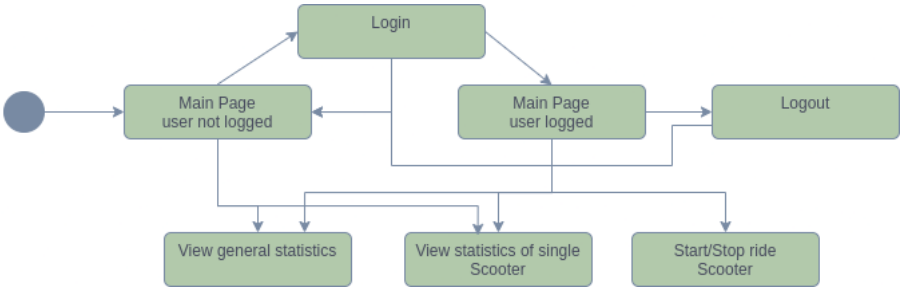

Figure 2: System interaction overview

live data of the scooter.
An anonymous user can click on a scooter to see if it's free or not, thing that is indicated also by the color of the placeholder, moreover is possible to access metrics of all scooter or of the specific device trips, that can be sorted per date, duration or total cost.
An user that instead proceeds with the login can also ride a scooter and stop a ride, other that being able to see statistics as the non-authenticated user.

3 Observability

3.1 Importance of Observability

Observability is crucial to ensure the proper functioning, security, and performance of the application. Constantly monitoring the application allows for the quick identification and resolution of potential issues, improving performance and ensuring an optimal user experience. Various tools and methods are used to monitor the application through dashboards and monitoring systems.

3.2 Kafka-UI

Kafka-UI is used for interaction with the Kafka message broker and monitoring of topics. This tool provides a detailed view of sent and received messages, allowing the data flow between the various components of the application to be monitored.

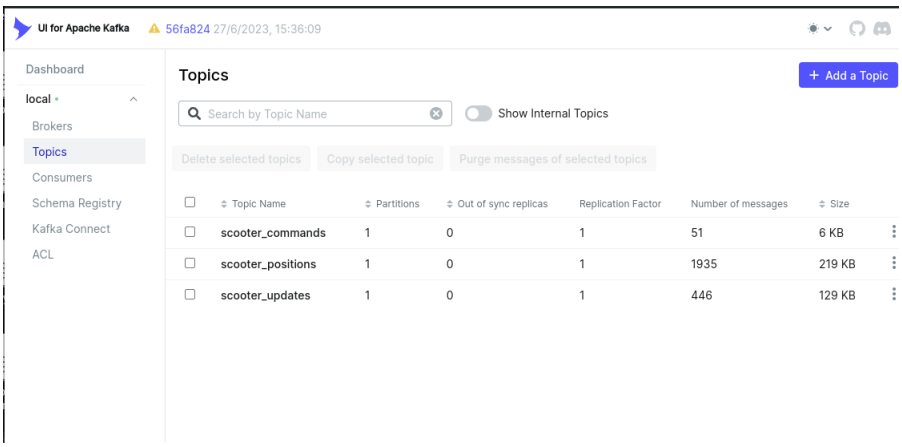

Figure 3: Kafka-UI homepage

3.3 Portainer

Portainer is used for stack orchestration and real-time metric monitoring. It offers a user-friendly interface to manage containers, view resource usage, and access logs, facilitating the management of the Docker infrastructure.

3.4 Mongo-UI

Mongo-UI is used to monitor the state of the MongoDB database and real-time transactions. This tool allows viewing of database operations, analyzing queries, and ensuring that the database is functioning correctly.

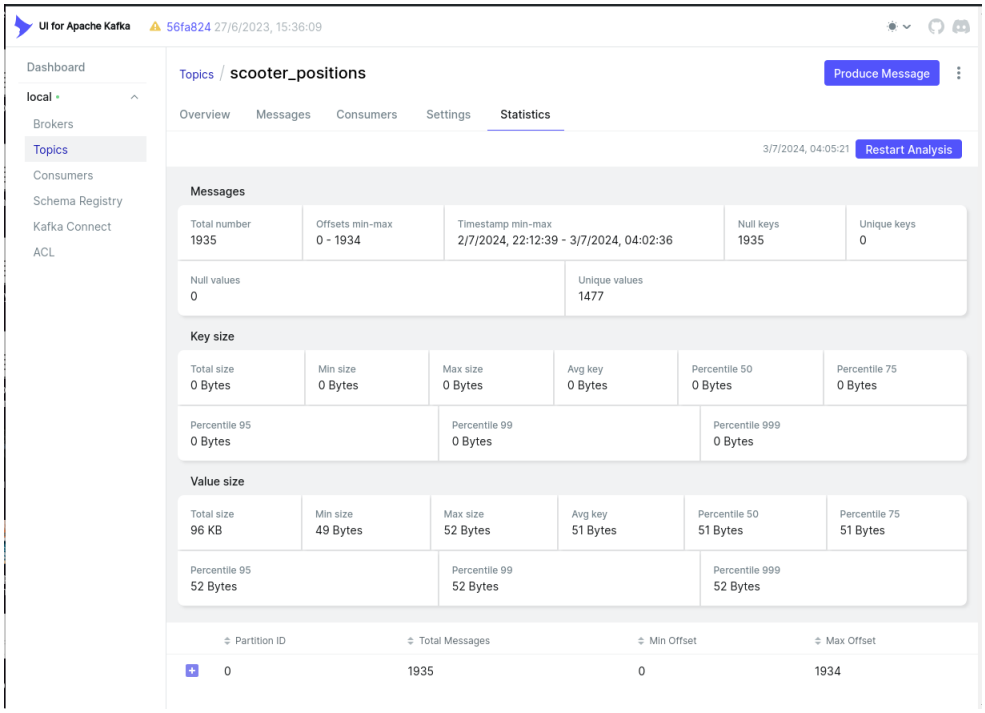

Figure 4: Topic statistics

3.5 Service Provider Metrics

The application uses open-source services and key-based APIs for mapping, such as Leaflet. These service providers offer detailed metrics on access and geolocation of the user base, allowing monitoring of map usage and optimization of performance.

3.6 Statistical View

Metrics exposed through the statistical view allow analysis of data related to scooter trips. Users can explore trips by sorting them by date, distance, or cost. The flexible architecture of the application makes it trivial to integrate additional metrics in the future, ensuring continuous system evolution.

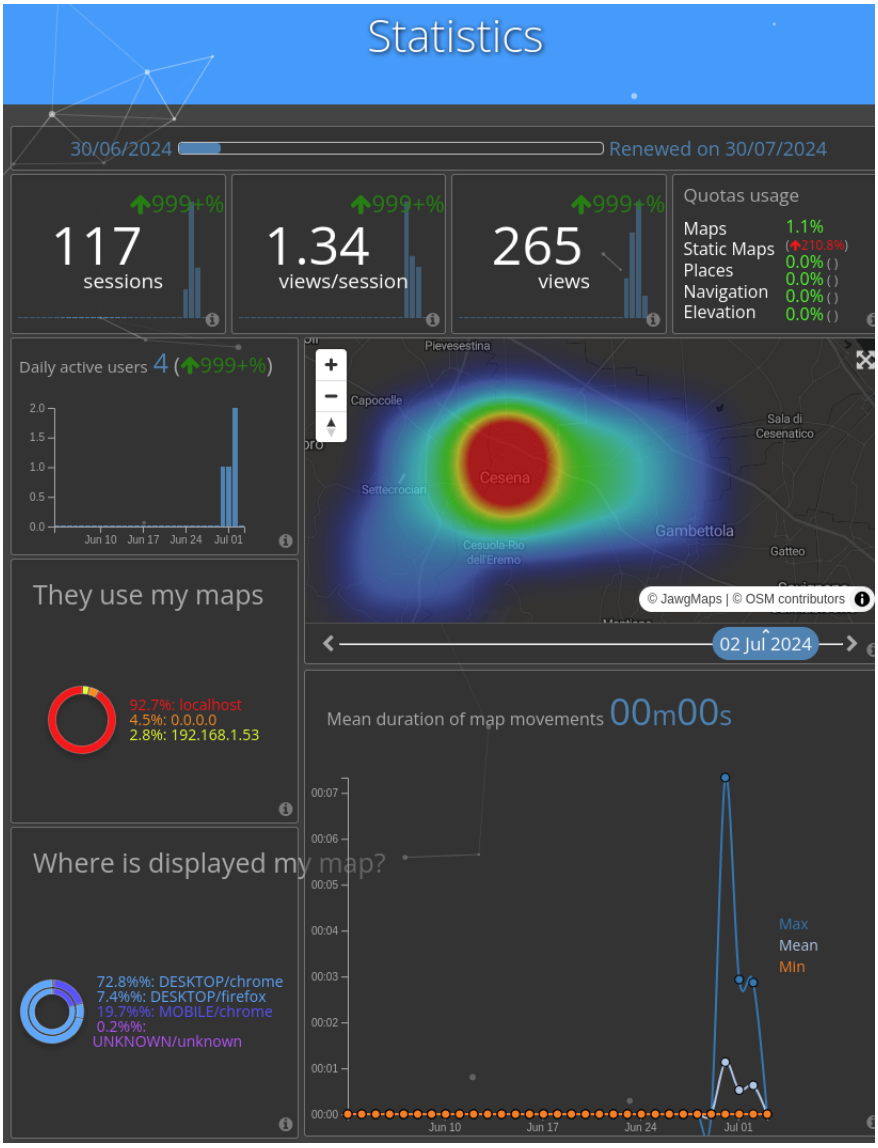

Figure 5: Tile loading information

4 Deployment

4.1 Docker

Each service of the application is implemented as a separate Docker container. This approach allows for greater modularity and ease of management. Services are connected to each other via a private Docker network, ensuring secure and isolated communication. The specific services are:

- **Frontend**: The container hosting the Vue.js application.

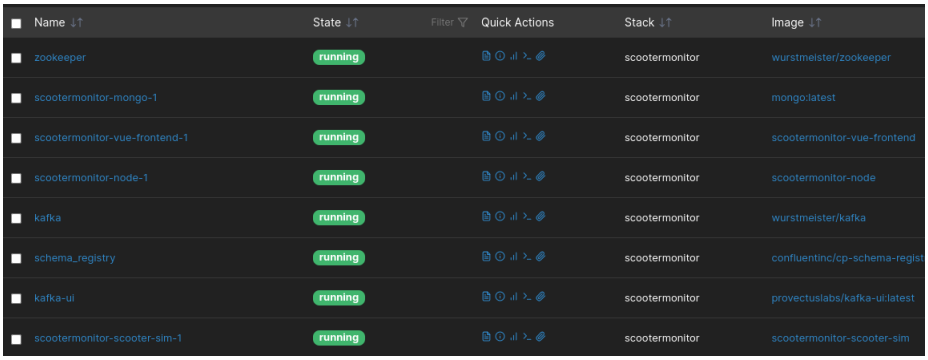

Figure 6: Application Stack

- **Backend**: The container running the Node.js server.
- **MongoDB**: The container for the MongoDB database.
- **Kafka (4 containers)**: The containers for the various Kafka components.
- **Scooters**: Container containing thread workers that simulate the scooters.

The project structure is formed such that, for each service, a docker image should be created, then on the project root, docker compose up to deploy the application.

4.2 Network and Communication

Services within the containers communicate with each other using Docker DNS. This allows each service to resolve the names of other services within the private Docker network, facilitating communication. The private Docker network ensures that services are isolated and protected from unauthorized external access.

4.3 Monitoring and Management

For continuous monitoring and management of the application, Portainer is used, a management platform for Docker containers that offers stack orchestration features, real-time metric monitoring, and allows access to container logs.

5 Usage Examples

Once landed on the page, the user sees the Live maps representing the real-time status of all devices, available in dark or light mode:

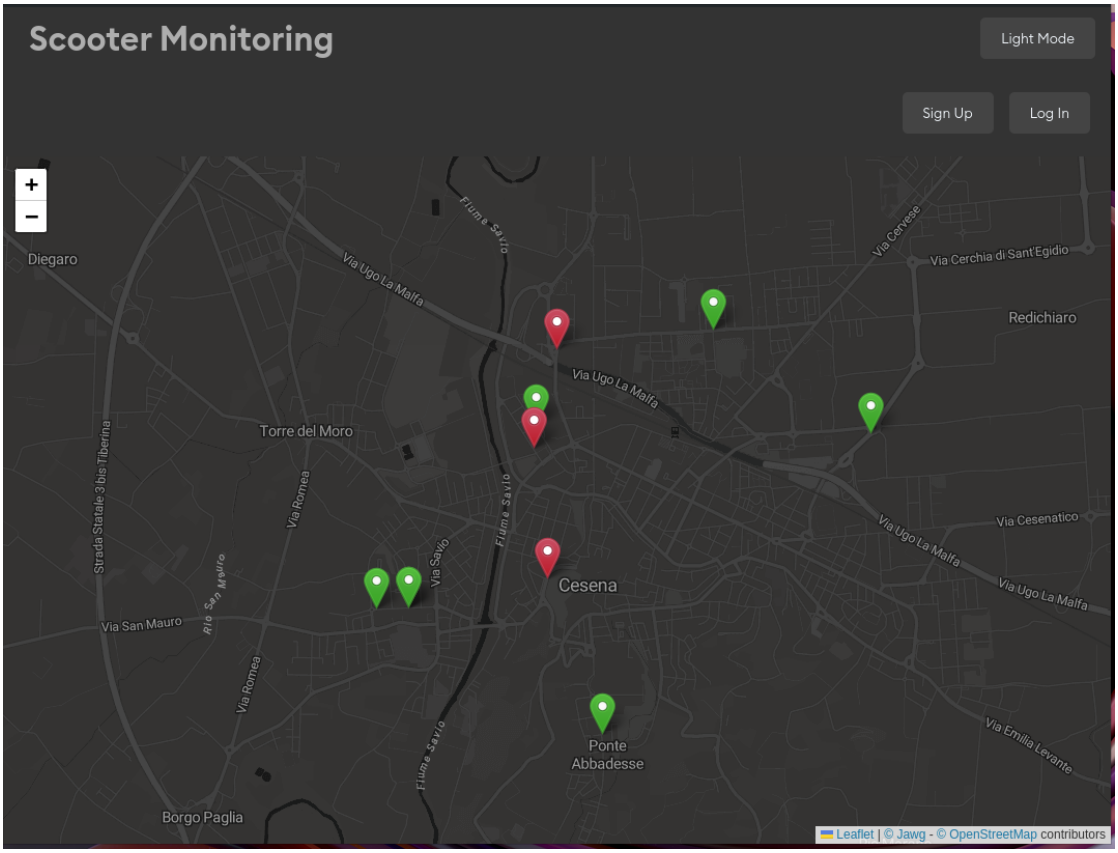

Figure 7: Home page, dark, only the map is visible

Once logged in, the user can select a scooter to unlock it, and get information of the specific scooter:

Is also possible to get more information and statistics:

The application is designed to be responsive, to adapt to various display sizes:

]

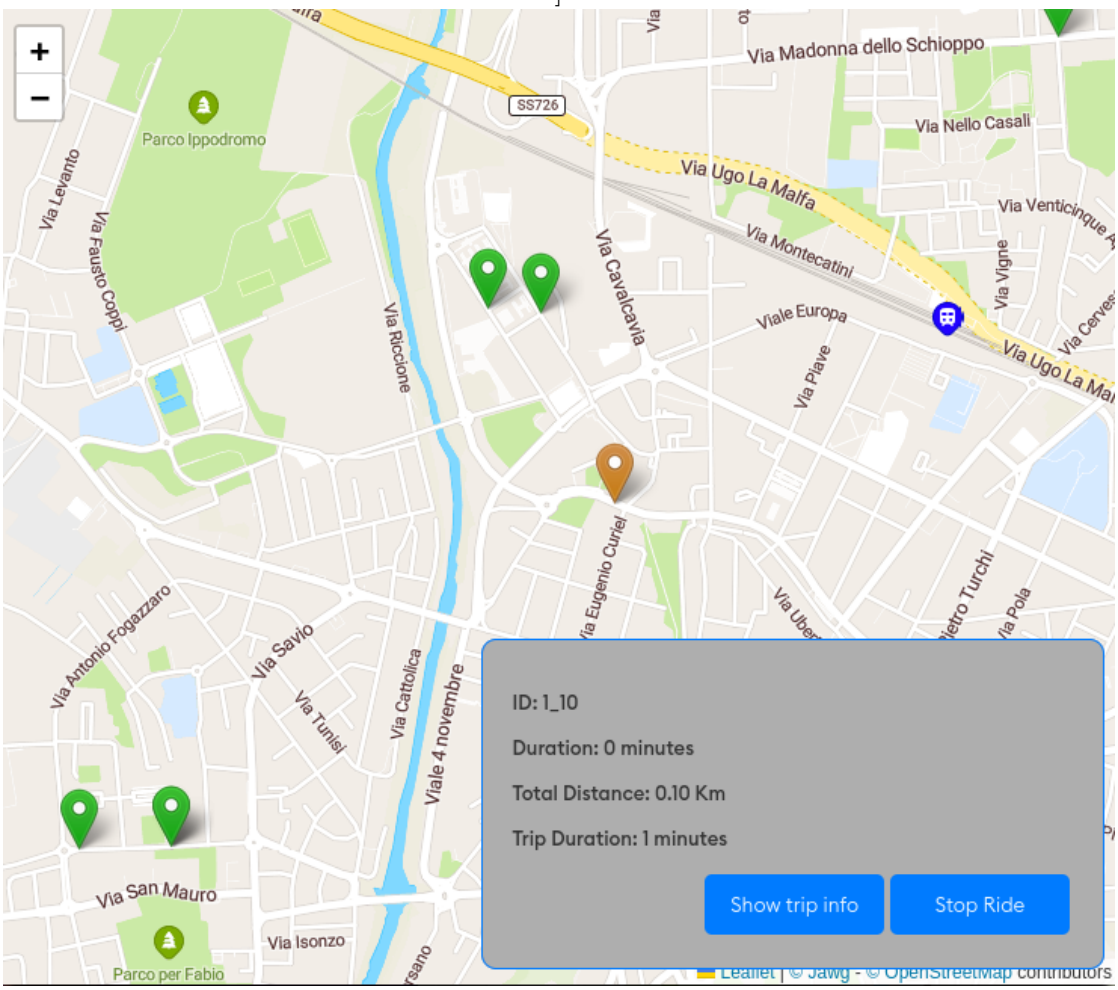

Figure 8: *Information* of the specific scooter

6 Conclusions

It's all about abstraction

In the Scooter Monitor project, the design emphasized a high level of abstraction, where scooters were modeled as simulated entities within the system. This approach allowed for a clear separation of concerns and facilitated the development of modular components. By using the concept of images, each representing a snapshot of the scooter's state, the system can potentially handle replication in future iterations to address the CAP theorem. This abstraction layer simplifies the handling of consistency, availability, and partition tolerance in a distributed system context.

Further abstraction and future work could be achieved by employing orchestrator

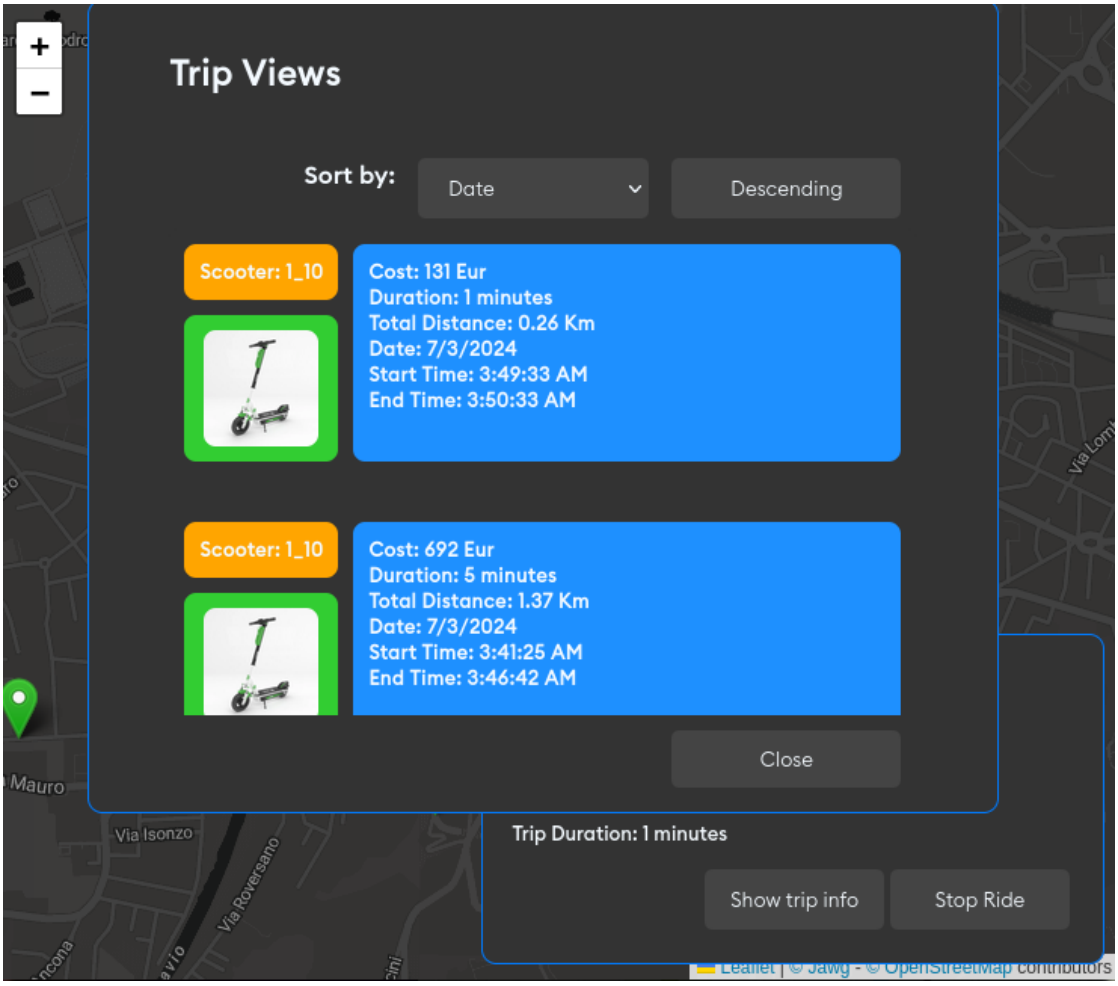

Figure 9: historical Info of trips of a specific device

tools such as Kubernetes for application orchestration. Kubernetes provides advanced features for managing containerized applications at scale, including automated deployment, scaling, and operations of application containers. By abstracting the underlying infrastructure, Kubernetes enables seamless scaling and management, thereby enhancing the robustness and reliability of the system. This would allow for more efficient handling of load balancing, fault tolerance, and rolling updates, ensuring minimal downtime and improved user experience.

6.1 Future Works

The application wanted to be a MVP (minimum valuable product) of a shared mobility solution. The intent was to showcase methodologies and technologies to deploy such

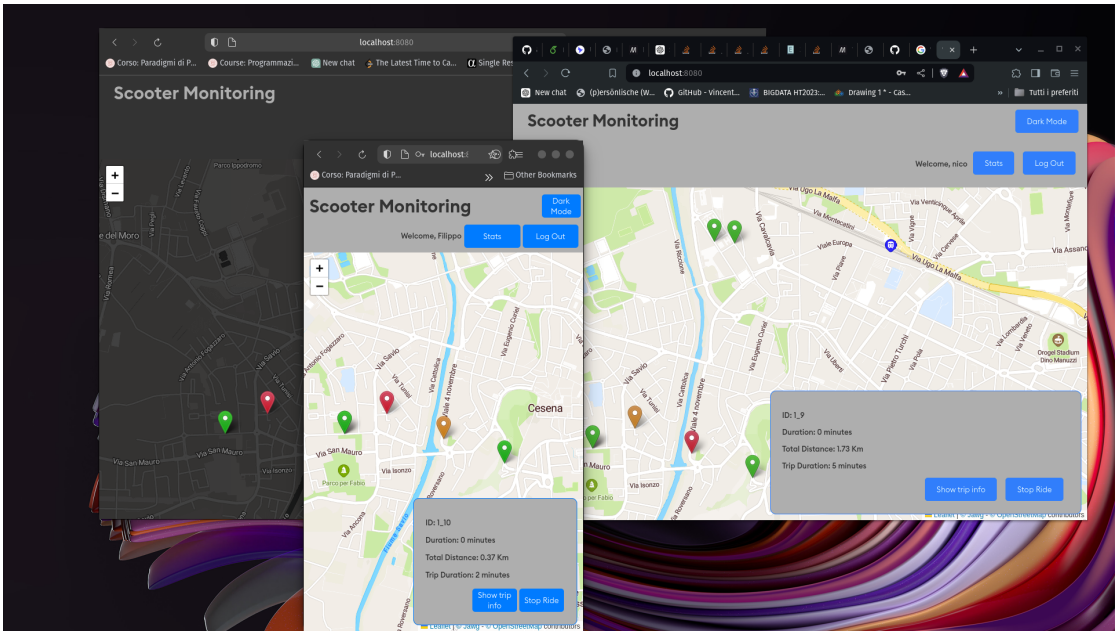

Figure 10: Responsive layout showcase

application.
Although satisfied, the application's requirements do not encompass the architecture itself, future work could address the deployment strategy, switching from docker to a more robust orchestrator like kubernetes.
Due to limited resources, no test nor build automation was used apart from the building and deployment of docker images, integration of webHooks and CI-CD technologies like GitHub actions will be addressed in future works, to ensure a smoother deployment and development application life-cycle.
The application does not distinguish between type of users, one of the most critical feature will be to add different user types and provide custom interfaces and interaction for each of them, to align to the specific needs of the user's type.