MSc Computer Engineering

Distributed Systems and Middleware Technologies

Claudio Daka

Edoardo Pantè

Academic Year 2023/24

Project Specifications..... 3
Project Idea..... 3
Functional Requirements..... 3
Non-Functional Requirements..... 4
Erlang Usage..... 4
Use Cases..... 5
System Architecture..... 6
Overview..... 6
Implementation..... 7
Java Web Server..... 7
Erlang..... 8
Master Node..... 8
Worker Node..... 9
Chat Node..... 9
Chat Server..... 9
Mnesia Database..... 10
Client-Server Communication..... 10
Client Side..... 11
JavaScript WebSocket..... 11
RQLite Database..... 12
Load Balancer..... 13
User Manual..... 14
Login page..... 14
Sign-Up page..... 14
Index page..... 15
Search functionality..... 15
Ride functionality..... 15
Stations page..... 16
Station page..... 16
Chat page..... 17
Chat Inbox page..... 17
Maintainer page..... 18

Project Specifications

Project Idea

MyRide is a platform that allows users to find a bike and use it for a desired period of time. At the end of this period, the user has to leave the bike in a specific station and pay the amount required. If a user encounters a problem with a bike, he can start a chat with one of the maintainers.

Functional Requirements

The users of the application will be divided into three categories: unregistered users, registered users and maintainers. Registered users will be allowed to use the main functionalities of the application. Will be provided a login system using username and password, through which the user will be correctly identified. A registration form will allow new users to register within the application as registered users.

Unregistered user:

- An unregistered user can register to the platform.

Registered user:

- A registered user can login to the platform with his username and password.
- A registered user can logout from the platform.
- A registered user can view the available stations.
- A registered user can unlock a bike.
- A registered user can stop a ride.
- A registered user can create a chat with a maintainer for a specific bike.

Maintainer:

- A maintainer can login to the platform with his username and password.
- A maintainer can logout from the platform.
- A maintainer can add a new station.
- A maintainer can remove a specific station.
- A maintainer can add a new bike to a specific station.
- A maintainer user can remove a bike from a specific station.
- A maintainer can view the active chat requests.
- A maintainer can chat with a registered user.

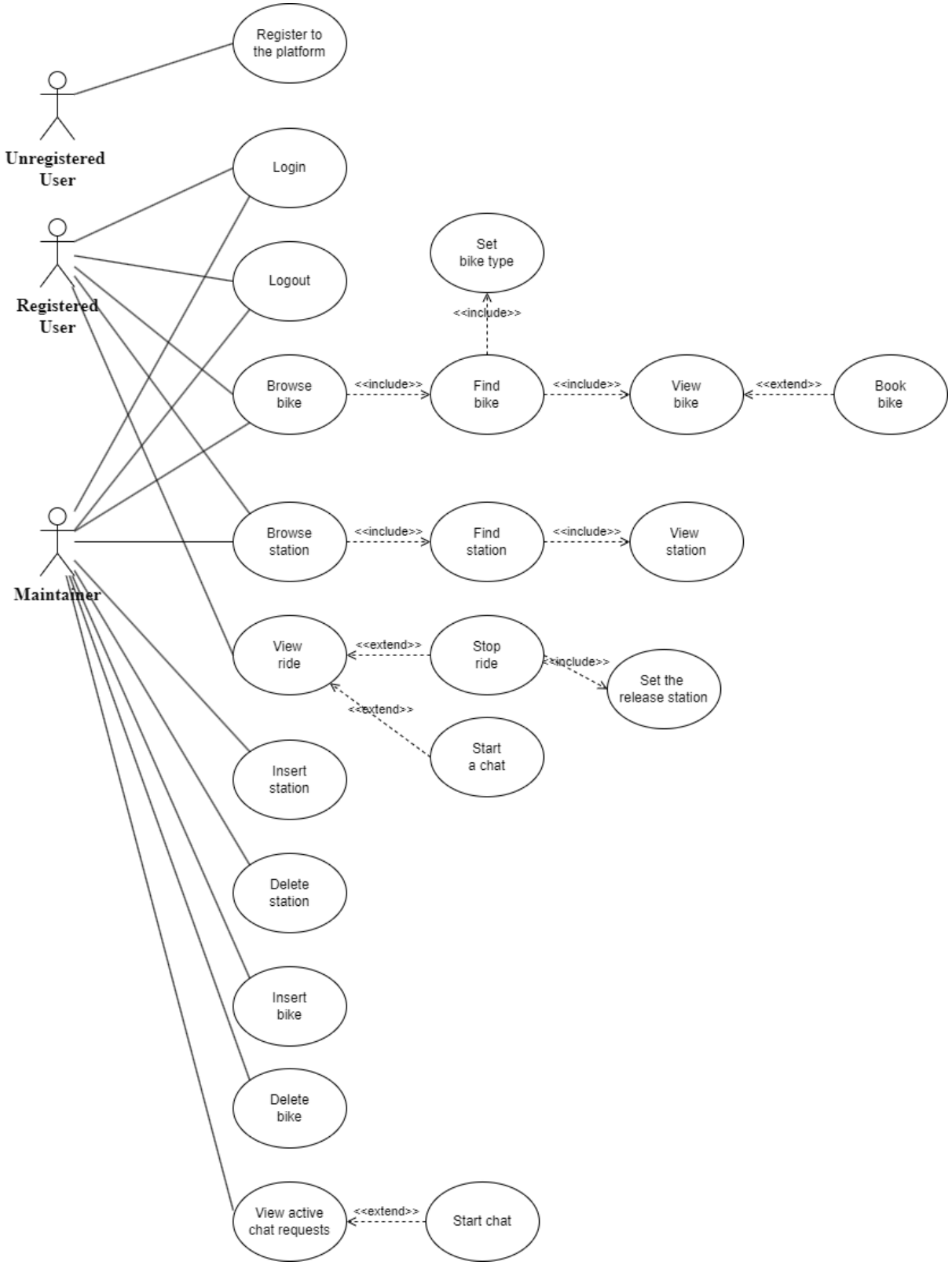
Non-Functional Requirements

- Every ride has a duration time.
- Every bike is identified by an id.
- Every bike has its own type.
- Every bike has a price per hour.
- Every bike can be left at any station.
- Every station has a number of available bikes.
- Every station has a position.
- Registered users are identified by their username.
- The client side and server side must use HTTP protocol to communicate.
- **Usability**: The application needs to be user friendly, providing a GUI.
- **Maintainability**: The code shall be readable and easy to maintain.
- **Concurrency**: The application must handle multiple users at the same time.
- **Persistency**: The application must achieve data persistence.
- The chat service needs to provide low latency, high availability and tolerance to single points of failures (**SPOF**) and network partitions, for which the chat functionality will be designed in order to prefer the Availability (**A**) and Partition Protection (**P**).

Erlang Usage

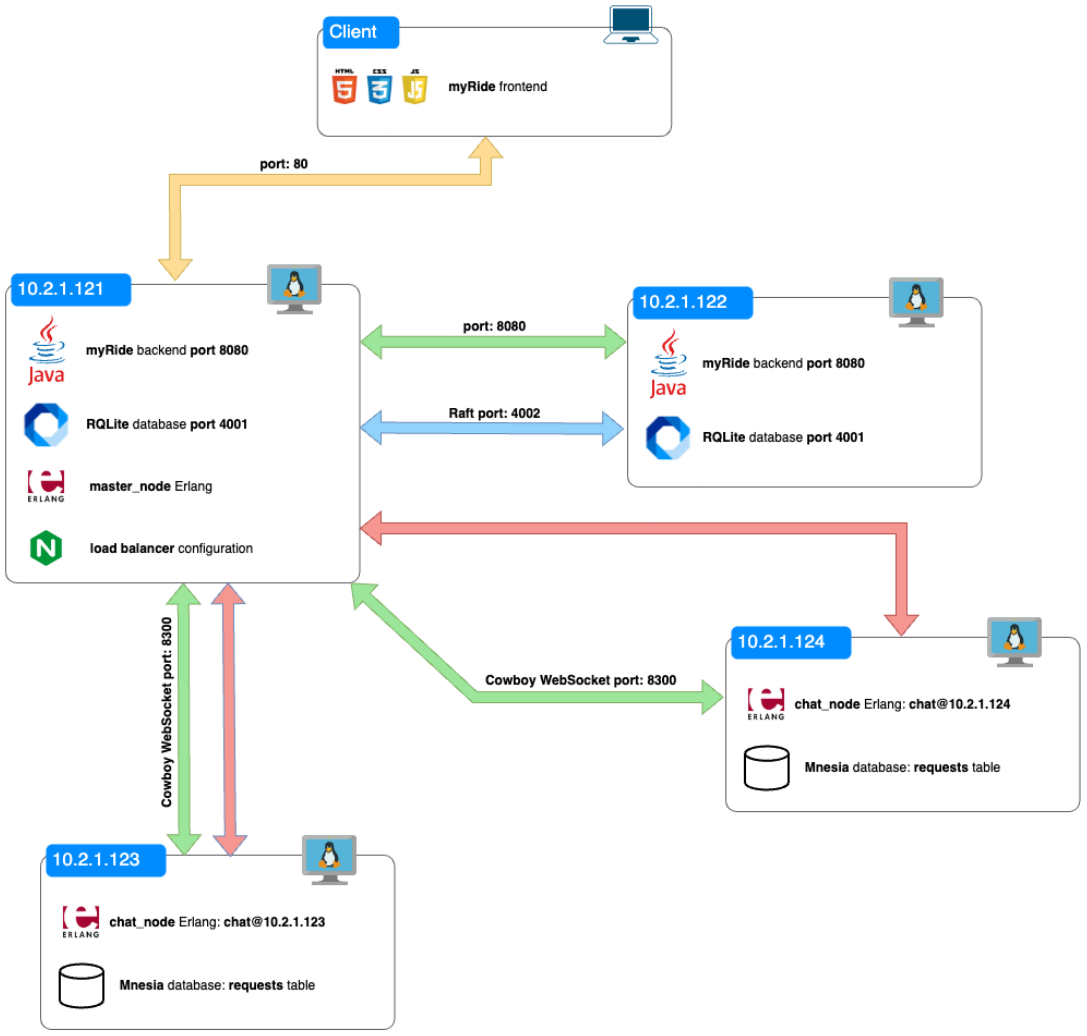
The chat functionality will be developed using Erlang. To take advantage of Erlang's features and to meet our requirements we will deploy the chat in a distributed manner, on multiple nodes. Distributed Erlang nodes will share chat and user status information through Mnesia, a distributed database. A load balancer will be also deployed to distribute the incoming requests to the available Erlang server nodes.

Use Cases


Register to the platform
Unregistered User
Login
Registered User
Logout
Set bike type
<<include>>
Browse bike
<<include>>
Find bike
<<include>>
View bike
<<extend>>
Book bike
Browse station
<<include>>
Find station
<<include>>
View station
Maintainer
View ride
<<extend>>
Stop ride
<<include>>
Set the release station
<<extend>>
Start a chat
Insert station
Delete station
Insert bike
Delete bike
View active chat requests
<<extend>>
Start chat

System Architecture

Overview



The web application, as we can see in figure, consists of a **Client Application**, an **Application Server**, and multiple **Erlang servers**. The Application Server, which implements most of the functionality of the application, uses **TomCat** as its reference implementation. The Java Web Server communicates with a **distributed Rqlite** database that allows the presence of a sqlite relational database on multiple nodes. Erlang Servers, which are managed by a **Load Balancer (Nginx)**, handle the chat service of the application through an **HTTP websocket communication**. The load balancer also handles the load of the Application Servers. The services in Erlang use the **Mnesia database**, which is shared between the master node and the worker node instances. As shown, it is a virtually fully distributed and replicated system that goes to make up for any failure of one of the components of the entire architecture. The only service that could not be distributed is the master node in Erlang because of its supervisor role.

Implementation

Java Web Server

The web server part is developed using a Spring Framework: Spring Boot. Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that can be "just run".

There is no need to deploy a WAR file on an external server because it includes embedded servlet containers like Tomcat (used in the project). There is a minimal configuration setup and no requirement for XML configuration. All the application parameters can be set on external application.properties file.

From a practical point of view, it is fast and easy to learn. It lets the programmer focus on the business logic of the application increasing the productivity. It is also highly flexible and offers the possibility to integrate external libraries and extensions.

In the following lines are exposed some of the most important aspects of the implemented web server:

- **Embedded Tomcat**: It's the default servlet container configured by Spring Boot.
- **Web/REST Server**: Java server that has both the role of web server and REST API server.
- **Spring MVC**: The web application is based on the paradigm Model-View-Controller to separate the business logic from the presentation layer. Inside the class methods (Get / Post) and paths are specified to tell Spring which request it should handle. In this project the 3 logical components are separated in different packages.
- **Database Manager**: Class implemented via the Singleton pattern to manage and maintain the connection to the database.
- **MasterNode.java**: Static class added to handle outgoing requests for the master node when the active chats are requested.
- **DTO classes**: Classes defined to manage communication with the frontend.

Erlang

Within the project, the chat service was developed using Erlang.

To take full advantage of Erlang's support for concurrency and scalability and to meet our requirements for high availability and fault tolerance, we deployed the chat service in a distributed manner, on multiple nodes. For this project, we configured two different containers to run their own Erlang instance (**10.2.1.123**, **10.2.1.124**).

Each node exposes an endpoint to connect to the users' browsers via the WebSocket protocol. In this way, the server can receive chat messages from a user and forward them to the receiver, meanwhile the user is not required to refresh the webpage.

The Erlang-side WebSocket communication has been implemented by leveraging **Cowboy**, an **HTTP server** for Erlang/OTP. This allows it to focus only on the business logic and the remote deployment, without having to deal with low level HTTP protocols. On the other hand, the client-side WebSocket communication has been implemented in **JS**. Every message exchanged between client and server is serialized in **JSON** format. The **jsone** library has been used for the Erlang implementation.

Distributed Erlang nodes share chat requests information through **Mnesia**, a distributed database, integrated by default into Erlang/OTP and based on ETS (in memory) and DETS (on-disk) as storage mechanisms.

A load balancer (**Nginx**) has been also deployed to distribute the incoming WebSocket requests to the available Erlang servers. The container hosting the load balancer (**10.2.1.121**) also runs an Erlang process (**master node**) responsible for configuring the Mnesia cluster and spawning the chat server application processes on the remote Erlang nodes. In addition, the master node handles communication via HTTP with the Java server.

Master Node

To correctly configure the Mnesia database, launch the remote Erlang nodes and handle communication via HTTP with the Java server, has been developed a master application which executes the following operations:

- It connects to the remote nodes by **net_kernel:connect_node/1**.

- It starts its local Mnesia process, creates a **persistent schema** with the other Erlang nodes (if it doesn't already exist) and creates the **requests table** inside the schema (if it doesn't already exist). The new table only resides on the chat server nodes.
- It spawns the chat node application processes on the remote nodes.
- It handles communication via HTTP with the Java server.

The endpoint it exposes is:

- **GET /chats**: this endpoint is contacted to get the list of all the active chat requests.

Worker Node

Chat Node

The chat node application is implemented by exploiting the **supervisor behavior**: at startup, the application process spawns a supervisor process (callback module **chat_node_sup**) which will be responsible to manage the lifecycle of the process which runs the **Cowboy HTTP server**.

The process running Cowboy is implemented as a gen server using the module **cowboy_listener** as a callback module. The Cowboy server will listen at the port 8300 waiting for new WebSocket connection requests, and it will spawn a new ad hoc process to handle the request once received. Thus, every user will be assigned to a different Erlang process. The new spawned process will call the functions belonging to the **chat_websocket** module:

- **init/1**: called whenever a request is received to establish a websocket connection.
- **websocket_handle/2**: called whenever a frame arrives from the client. It will handle the reception and deserialization of the JSON objects coming from the client.
- **websocket_info/2**: called whenever an Erlang message arrives (i.e. a message from another Erlang process). It will forward a chat message to the client browser assigned to this process.
- **terminate/3**: called whenever the WebSocket connection is closed. It will remove its assigned user from the list of the user logged in the chat (maintainer or regular user).

Chat Server

The **chat_server** is the module that implements the main functionality of the application, handling all the main operations such as starting a chat, sending a message, and exiting the chat itself. Worker nodes expose the endpoint on port **8300** to receive messages via WebSocket. Once a message arrives, via the opcode field of the received message, the handler will implement a different handling.

In the following lines is exposed the structure of the chat node module:

- **chat_node_app**: It's the first module that is executed, implements the application behavior and executes the **supervisor**.

- **chat_node_sup**: The supervisor callback module. It is responsible for spawning a process that will manage the lifecycle of the process running the Cowboy server. It implements the behavior of the supervisor.
- **cowboy_listener**: This is a **gen_server** module that starts an instance of Cowboy listening on port 8300 to handle all incoming messages via WebSocket.
- **chat_websocket**: This is the module that implements all the message handling logic via WebSocket: it receives the message, parses it to get the information, and finally assigns the actual handling to the **chat_server** module. Once it has finished handling informs the **chat_websocket** module to send a response message.
- **chat_server**: This is the module that implements the actual logic of the application. It implements all the logic for controlling, saving and managing the chat requests. In case some operations also affect the database, it will contact the **mnesia_manager** module to perform them.
- **mnesia_manager**: This is the module that handles all operations involving the Mnesia database. It's contacted exclusively by the **chat_server** and executes the queries requested.

Mnesia Database

Mnesia is a true distributed DBMS that we used to store chat requests information in a table. These informations were distributed over all the erlang nodes, using ram copies as a storage option. As previously anticipated, there is only one table in the database, **requests**, whose type is bag. The attributes of this table are **user_pid**, **username** and **bike_id**. The data in this table are used to keep track of all chat requests made by users. More in detail, the queries implemented are:

- **start_chat**: Adds a chat request.
- **end_chat**: Removes a user from a given chat.
- **get_bike_pids**: Returns the pids of the users currently online in a chat related to a specific bike. This is a particularly useful query for figuring out to which erlang processes forward messages.
- **remove_users_by_username**: Recursively removes all users connected to this server.

In fact, this last query allows us to guarantee a state of eventual consistency after a node crash, which would otherwise introduce duplicates when it restarted.

Client-Server Communication

During the WebSocket session, all messages exchanged between the client browser and the Cowboy WebSocket handler are encoded in JSON format. Meanwhile JavaScript natively supports JSON encoding/decoding inside the browser, the Erlang application requires an external dependency such as the **jsone** library.
The client browser can send three different kinds of JSON messages:

- Request for starting a chat.
- A heartbeat message.
- Chat message.

The JavaScript client encodes these messages in the following format:

{**opcode**: "START",
username: <username of the current user>,
bike_id: <id of the related bike>}

{**opcode**: "HEARTBEAT",
username: <username of the current user>,
bike_id: <id of the related bike>}

{**opcode**: "MESSAGE",
text: <text of message>}

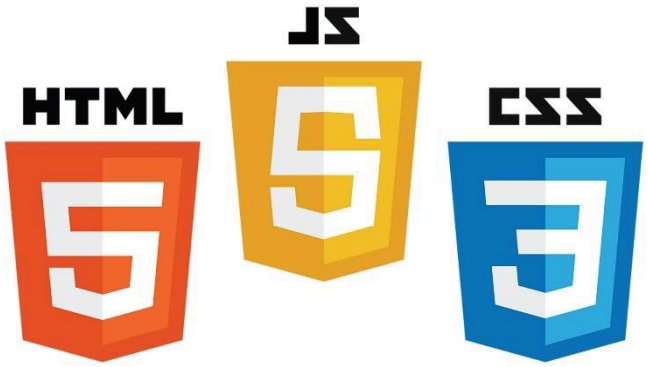
On the other hand, the Cowboy server can send back only one kind of JSON message:

- Chat message

The encoded format is the following:

{**opcode**: "MESSAGE",
sender: <sender's username>,
text: <text of message>}

Client Side



JavaScript WebSocket

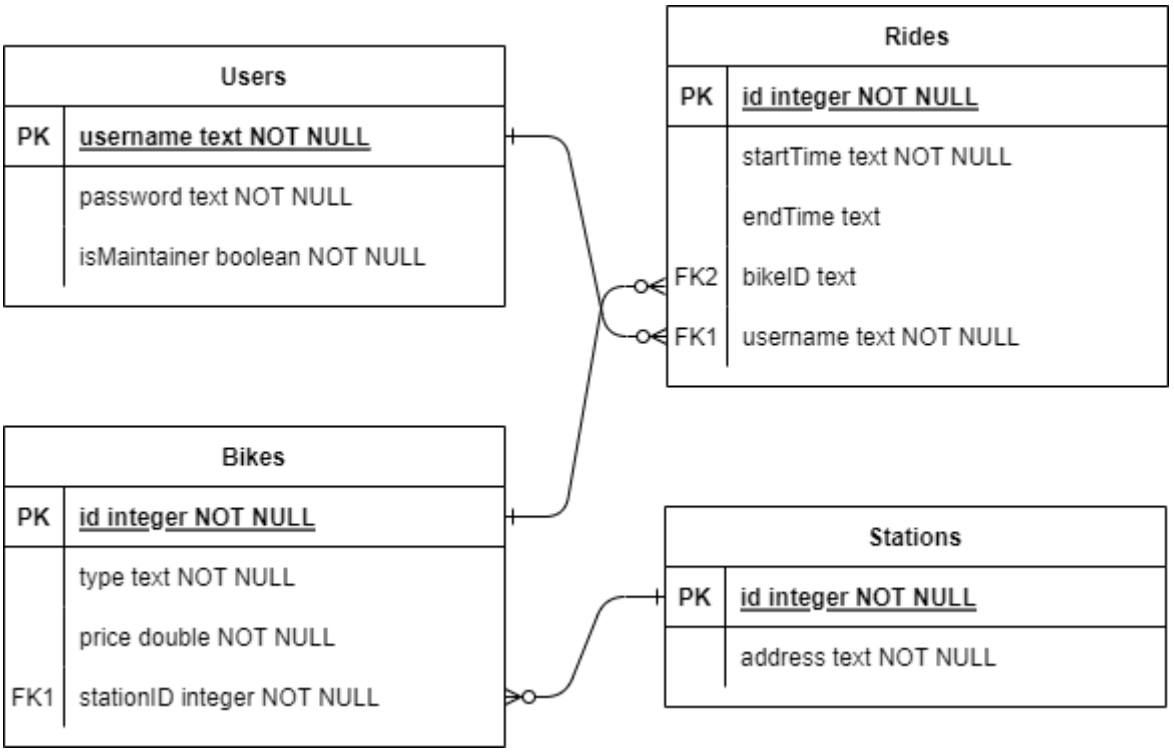
As already mentioned, communication between the client and the worker nodes takes place via WebSocket. This takes place on the page dedicated to a chat related to a specific bike. The connection management uses the WebSocket object offered by JavaScript and implements the functionality of sending and receiving messages. When messages are received, they are parsed in order to extract the content and attach it on the chat window.

RQLite Database

rqlite

The lightweight, distributed relational database

In addition to Mnesia, the Java server uses a **Rqlite database** to maintain information on users, bikes, stations and rides. RQLite is a wrapper for **SQLite** that implements the functionality to have a **distributed database cluster**. This allows the distribution of the database in two nodes. In addition, a very interesting feature is that it is completely **transparent** to the programmer, as we do not need to manage the cluster at the code level as any faults or crashes are handled directly by the **DBMS**. Furthermore, it was very useful to deal with SQLite anyway, as it meant that we did not have to rewrite queries or implement new management mechanisms. As far as the entities in the database are concerned, the structure can be seen in the following image:



Load Balancer

To fairly distribute the load relative to the chat service and fully exploit the benefits deriving from a distributed Erlang deploy (such as high availability, fault tolerance, scalability...), a **load balancer** has been deployed on node **10.2.1.121**.

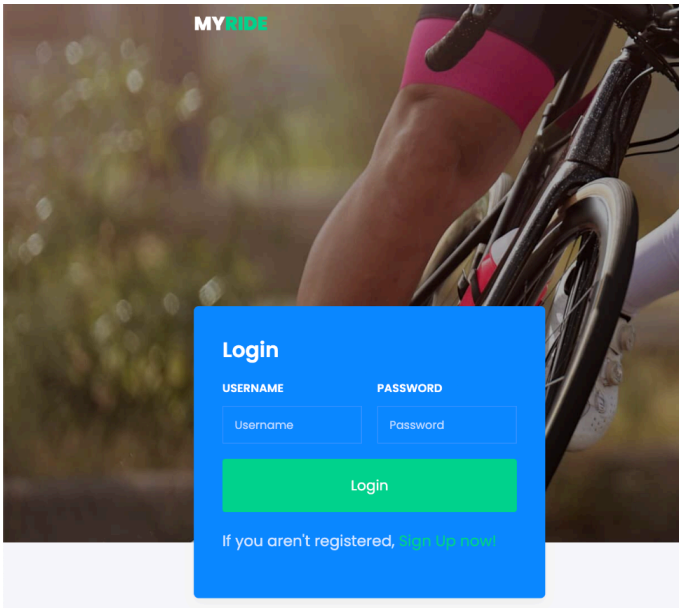
Nginx has been chosen for this purpose since it natively supports the WebSocket protocol both as reverse proxy and load balancer, so it's a perfect match for our requirements. The Nginx service has been configured with two proxies. The first is responsible for balancing the web servers load and is listening on port **80**. The second is responsible for balancing every WebSocket connection to one of the remote chat server nodes and is listening on port **8300**.

```
worker_processes 1;
worker_rlimit_nofile 8192;

events {
  worker_connections  1024;
}

http {

    map $http_upgrade $connection_upgrade {
        default upgrade;
        '' close;
    }

    upstream myride {
        least_conn;
        server 10.2.1.121:8080;
        server 10.2.1.122:8080;
    }

    upstream websocket {
        server 10.2.1.123:8300;
        server 10.2.1.124:8300;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://myride;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
            proxy_set_header Host $host;
        }
    }

    server {
        listen 8300;
        location / {
            proxy_pass http://websocket;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
            proxy_set_header Host $host;
        }
    }
}
```

User Manual

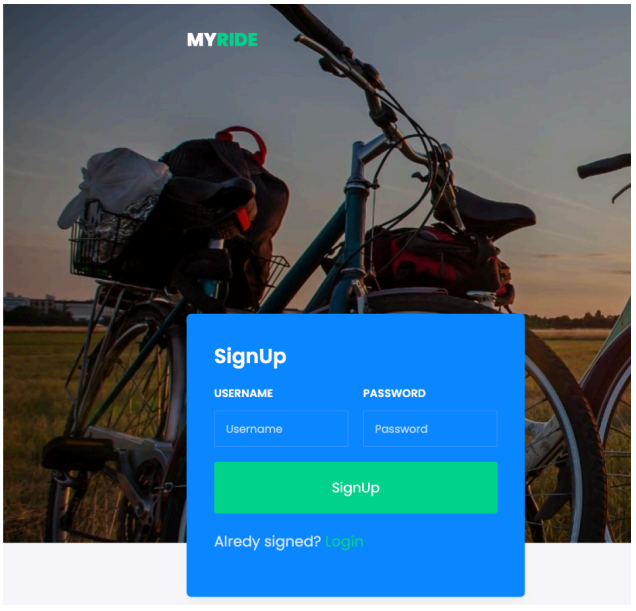
Login page

On this page, the user can log in by entering a username and password. If the user is not already registered, he can do it through the signup page.



Sign-Up page

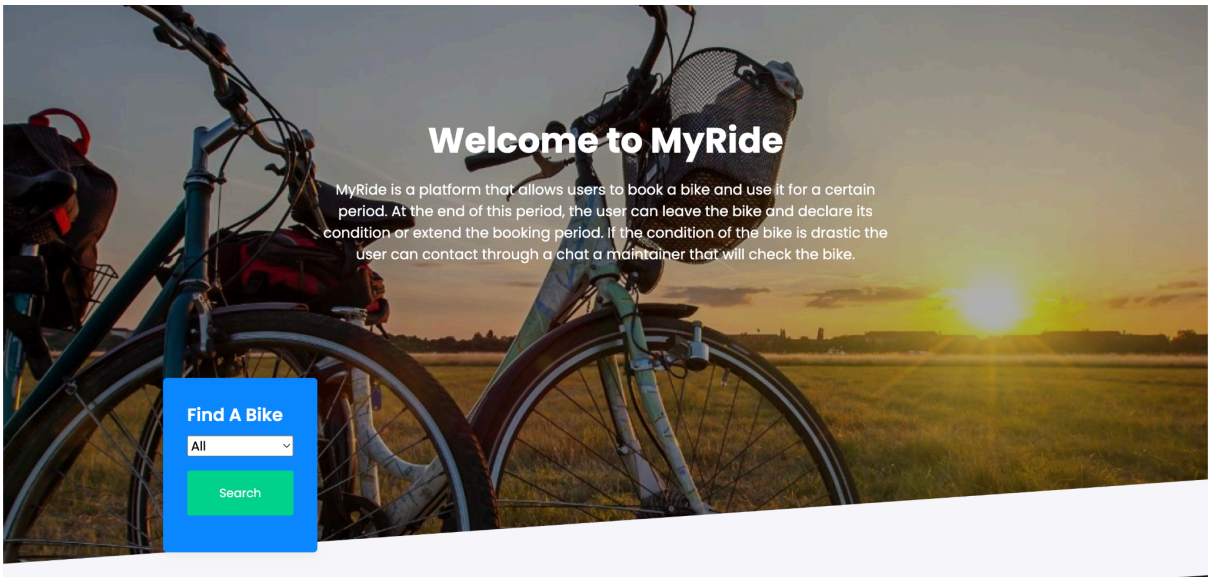
On this page, an unregistered user can register on the platform if not already registered.



Index page

Search functionality

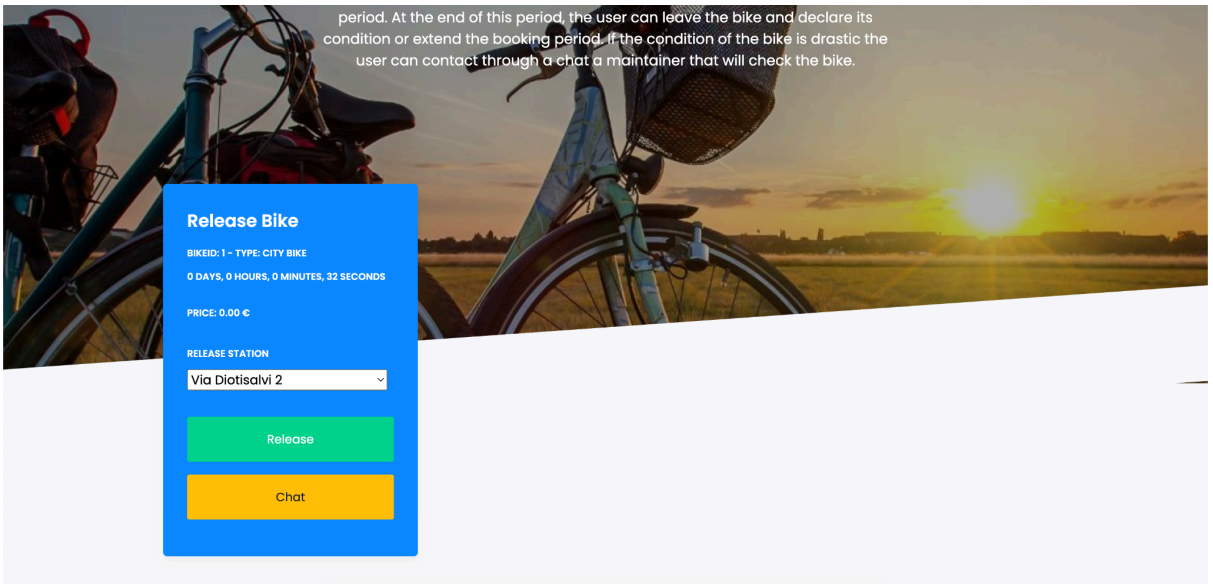
On this page, a user can search a bike, specifying the type if necessary.



Ride functionality

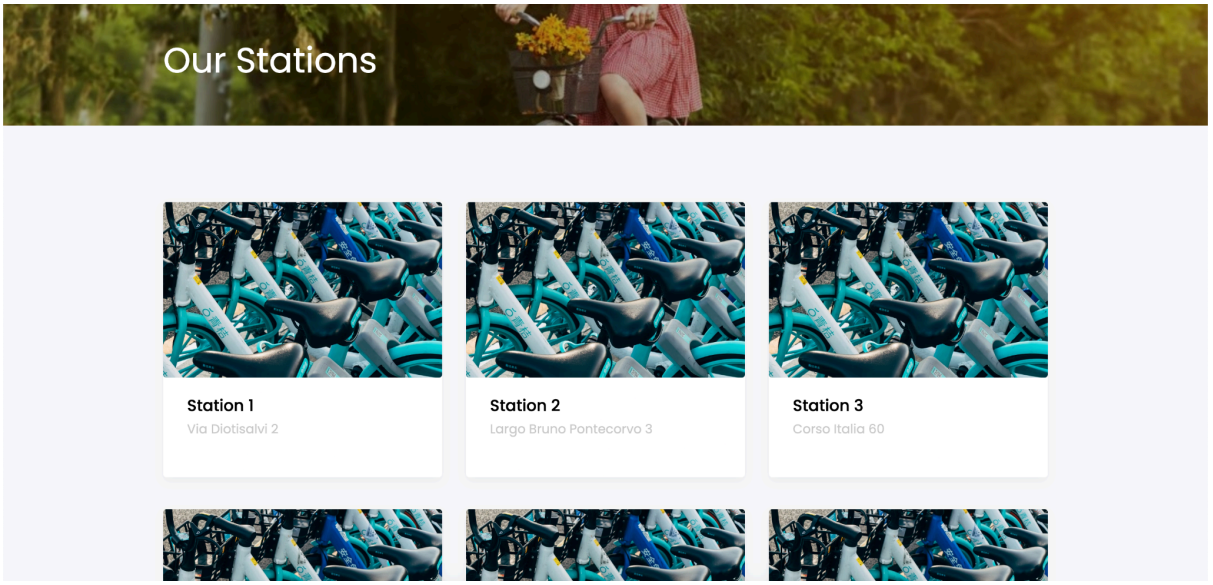
On this page, a user can:

- Release the booked bike in a specific station.
- Start a chat for a specific bike.



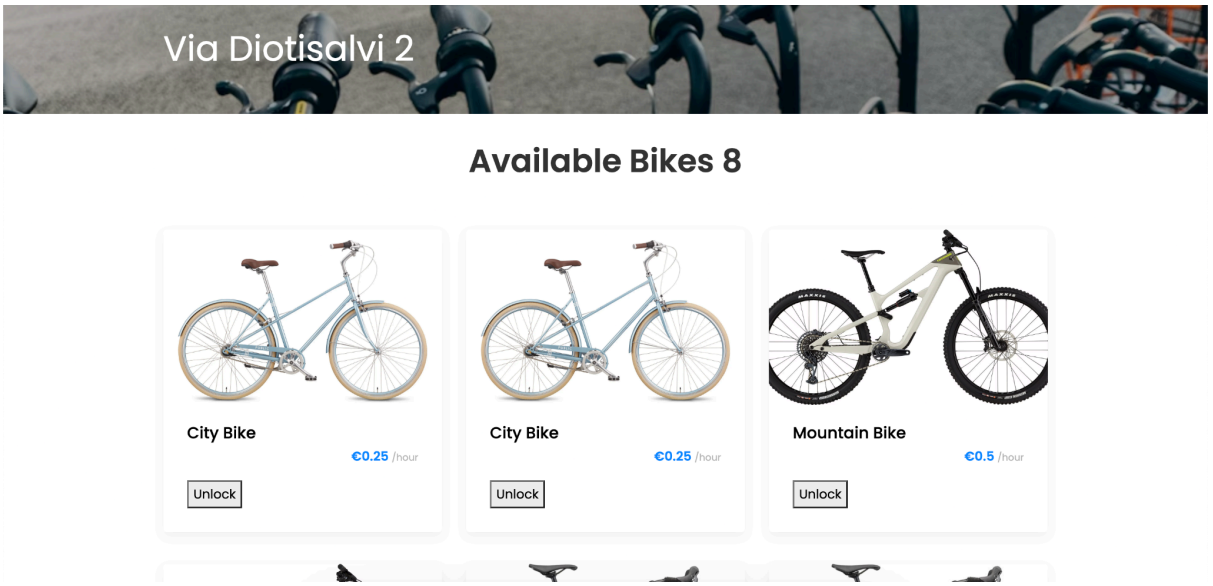
Stations page

On this page, a user can view all the available stations.



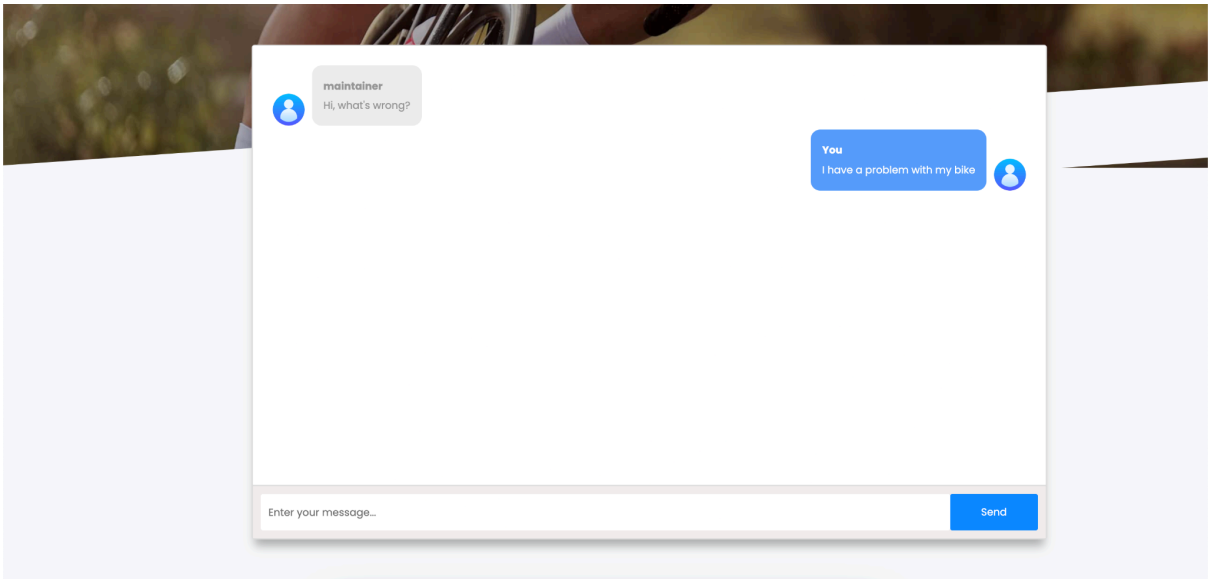
Station page

On this page, a user can view all the available bikes in a specific station and unlock the desidered one.



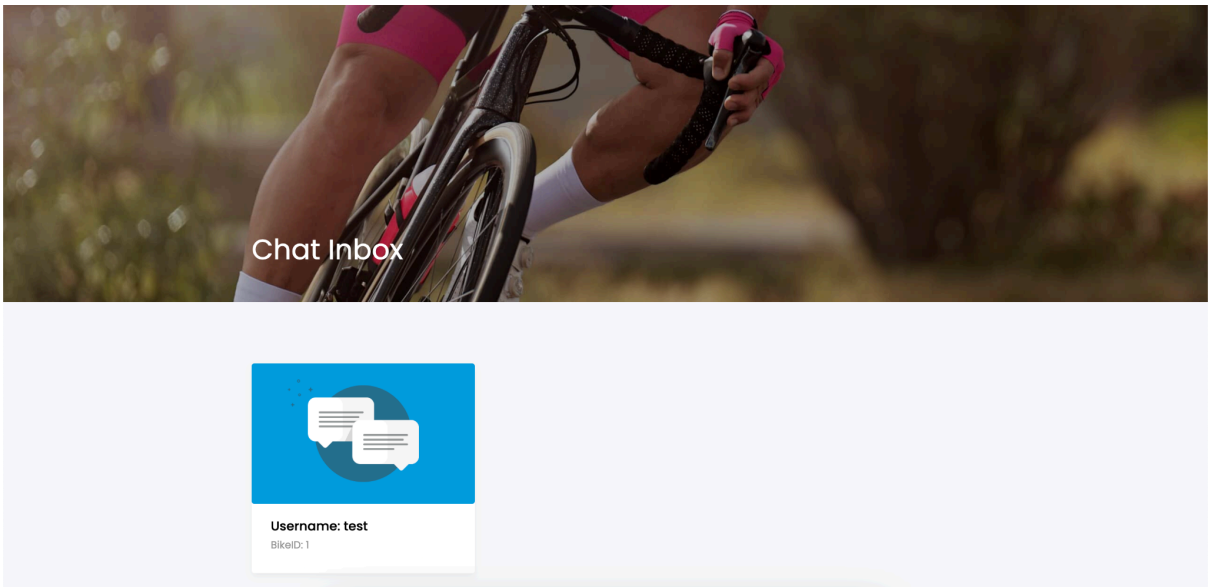
Chat page

On this page, a user can chat with a maintainer, and vice versa.



Chat Inbox page

On this page, a maintainer can view all the active chat requests.



Maintainer page

On this page, a maintainer can:

- Insert new station.
- Insert new bikes in a specific station.
- Delete a specific station.
- Delete a specific bike.