UNIVERSITÀ DI PISA

MSc in Artificial Intelligence and Data Engineering
MSc in Computer Engineering

Distributed Systems and Middleware Technologies

Project Documentation

Fabio Cognata

Federica Perrone

Luca di Giacomo

A.Y. 2023-2024

https://github.com/FabioCognata/DistributedSystems

Summary

1. Project Specifications

1.1. Introduction

The main purpose of this application is to allow the employees of a company to chat with each other. The employees have a dedicated page in which they can retrieve their past chats and start new ones. Furthermore, each employee can look at the messages received while not online via a notification system.

1.2. Functional and Non-functional Requirements

In our web-application, one type of actor exists, and it is the **user**.

The operation that the user can perform are the following:

- Create an account
- Login/logout
- Browse employees of the company
- Search for a specific employee by username or full name
- Filter employees based on the department in which they work
- Create new chat with an employee
- Browse his past chats
- Delete his past chats
- Browse his notifications containing messages received while he is offline
- Browse the messages of a specific chat

For what concerns **non-functional requirements** we have:

- Concurrent service accesses management
- Strong consistency for users, chats and messages and notifications data stored in MySQL DB
- Horizontal scaling
- High service availability for the web app server
- Interface responsiveness
- Soft real time
- Usability

1.3. Synchronization and communication issues

- Initialization of a Connection type resource in the Servlet Context, at the Context Initialization to share the MySQL database connection among all the Servlets.
- Internal coordination between erlang PIDs using registries for Real Time synchronization of messages and notifications.
- Initialization of a handler for MySQL queries in Erlang to manage the persistency of messages and notifications without blocking the registries from handling other incoming messages and still be able to coordinate eventual confirm/errors from the mysql handler process.
- Coordination between Chat Registry and Notification Registry to manage the scenario where the user is not in the chatroom (forwarding of the request to the Notification Registry for the notification storage and answering to the WebSocket).

- Access Tokens handling over multiple Tomcat instances by hashing the IP addresses in the Nginx load balancer.

2. System architecture

2.1. Introduction

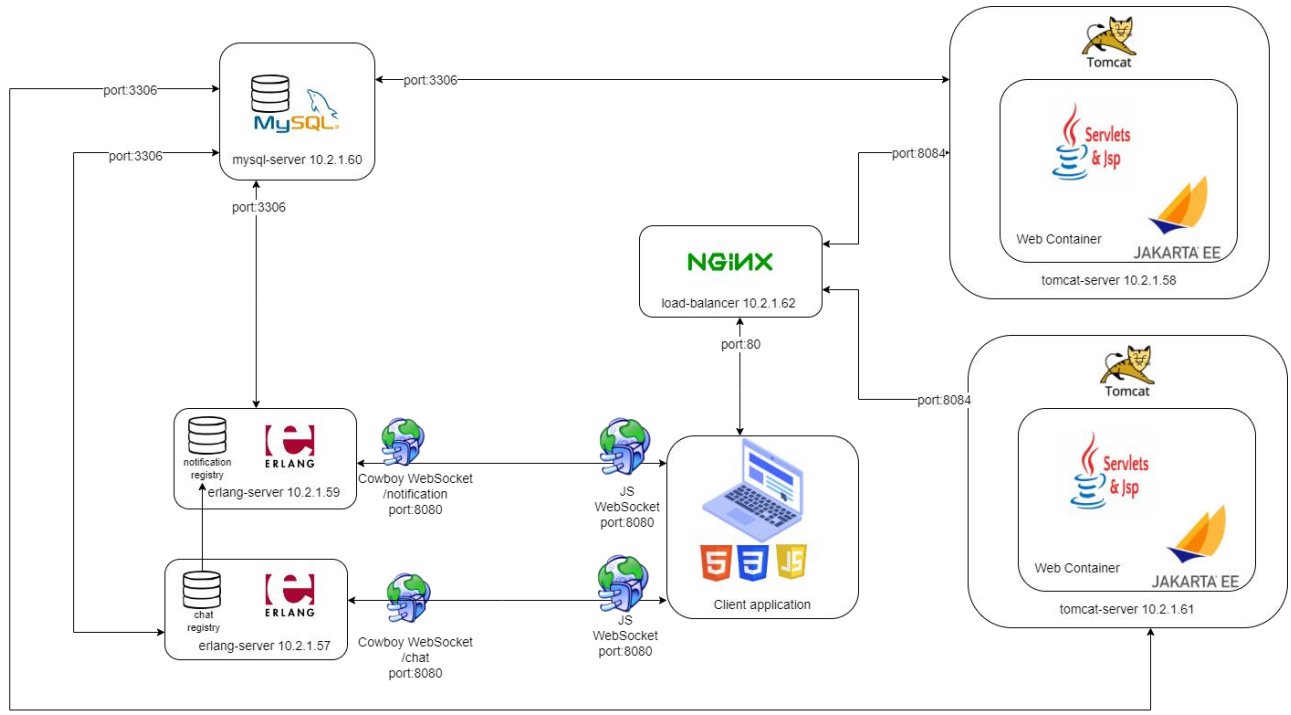


Figure 2.1 - System Architecture

As shown in figure, the web application is composed of two **Tomcat servers** managed by a **Load Balancer** developed via **Nginx**, two **Erlang servers** each one exposing specific functionalities, specifically one for handling the chat and the second one for the notification in order to separate and reduce the overall load of requests and for last a **MySQL server** to handle the data in a consistent and permanent manner.

2.2. Erlang

In our project, we designed the chat and notification features using **Erlang**. To fully utilize Erlang's built-in capabilities for handling multiple tasks simultaneously and ensuring robustness, we implemented the two functionalities in a distributed fashion across two separate containers, each running its own Erlang instance, with distinct IP addresses (10.2.1.57 and 10.2.1.59).

Every server node provides an access point for connecting to users' web browsers using the **WebSocket** protocol. This approach enables the server to accept chat messages from a user, relay them to other online users and save them into the Database. It allows also to handle the notifications pushing it in the client. All this has been done ensuring that the user doesn't need to manually refresh the webpage.

For the Erlang-side WebSocket communication, we harnessed **Cowboy**, an efficient and contemporary HTTP server tailored for Erlang/OTP. This choice enabled us to direct our attention solely towards the business logic and remote deployment, sparing us from the complexities of low-level HTTP protocols. In contrast, on the client side, WebSocket communication was implemented using **JavaScript**. Each message exchanged between the client and server was serialized in JSON format, with the Erlang implementation utilizing the **jsone** library.

For last **rebar3** has been used as building tool for the Erlang application.

2.2.1. Chat Server

The chat server application leverages the **supervisor behavior**. Upon startup, the application process initiates the creation of a supervisor process, known as the "**chat server sup**" callback module. This supervisor is responsible for overseeing the lifecycle of the process that operates the Cowboy listener. The process responsible for running Cowboy is designed as a **gen server** and employs the "**cowboy listener**" module as its callback module. The Cowboy server listens on port 8080, awaiting incoming WebSocket connection requests. Once a request is received, it spawns a new ad hoc process to manage that specific connection. As a result, each user is assigned to a distinct Erlang process. When the process starts a "**socket listener**" module and a "**chat registry**" together with "**MySQL listener**" modules are initiated.

Then the chat registry is charged to coordinate the socket connections through the erlang messages.

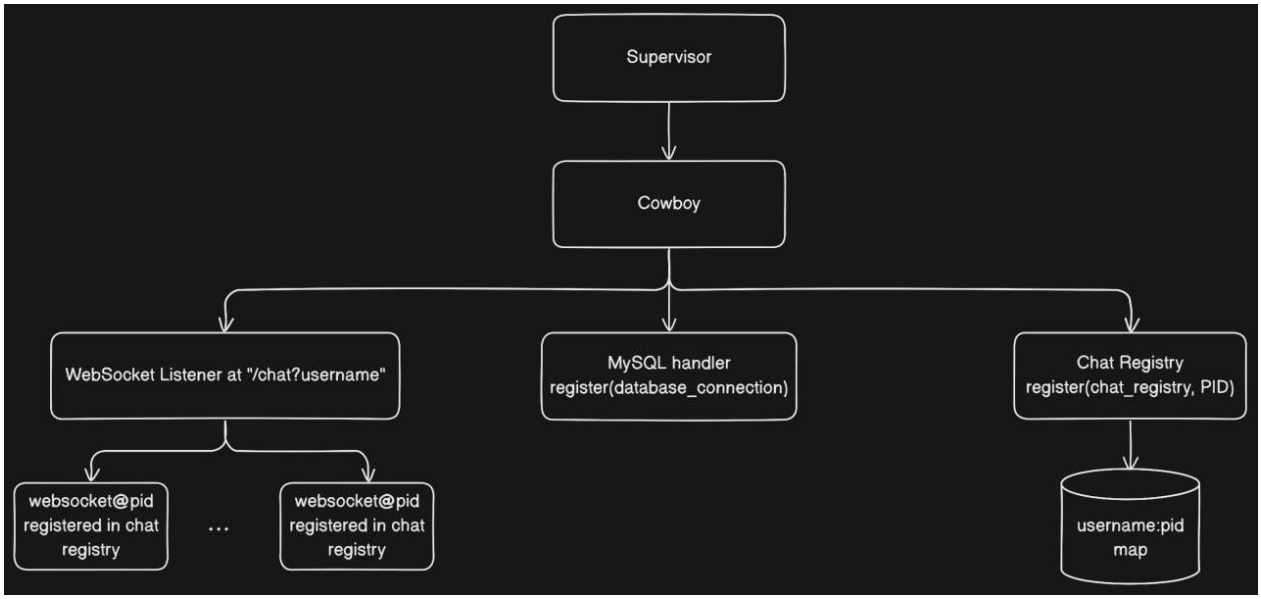


Figure 2.2 - Chat Server Architecture

Socket listener module basic operation

- *init/1*: This function is invoked every time a request is received, spawning a new process for the Cowboy WebSocket.
- *websocket_init/1*: Once called it passes its related PID and the username for the registration.
- *websocket handle/2*: Whenever a frame is received from the client, this function comes into play. It is responsible for handling the reception and deserialization of JSON objects sent by the client.
- *websocket info/2*: When an Erlang message is received by the registry process, it forwards chat messages to the client's browser linked to this particular process.
- *terminate/3*: This function is executed when the WebSocket connection is closed. Its main duty is to remove the associated user from the list of currently active ones, by sending an unregister message to the registry process.

Chat registry module message operation

- *Register*: Insert a key value pair for the username and its associated WebSocket PID in a map.
- *Forward*: This operation is useful to forward a message. Specifically, it looks for the destination PID in the map, then spawn a new process in order to save the message into the database and

eventually choosing if forward the message to an already opened chat WebSocket, or send it to the notification server, in the case the specific WebSocket is not found.

- *Unregister*: Remove the specific key value pair from the map.

2.2.2. Notification Server

Similarly to the Chat Server, a **notification registry** and a **MySQL handler processes** are initiated in the cowboy start link, following the same overall flow as described before.

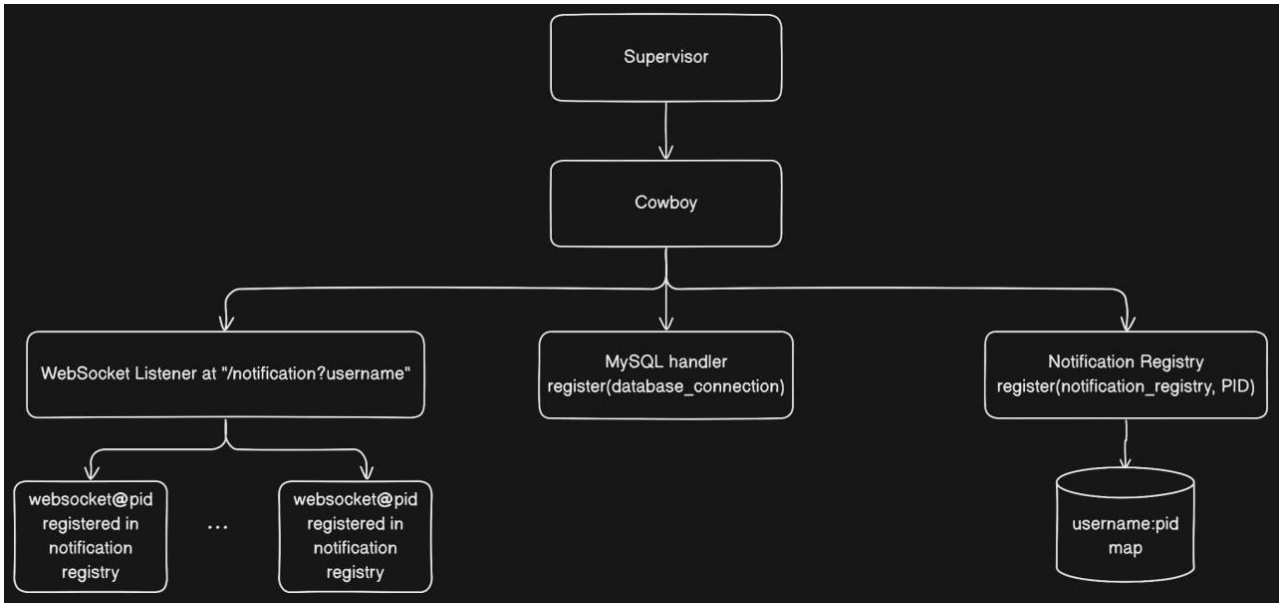


Figure 2.3 - Notification Server Architecture

Notification registry module message operation

- *Register*: If the record of a user is not found in the map, it adds a new one with a PID, notifying all the currently connected users that a new one is online.
- *Increase*: useful operation that allow to increase the notification badge counter by pushing a new notification in the database through MySQL handler process exploiting the same strategy complementary to the chat operation.
- *Delete chat*: Used to notify a user that the chat page he is in no longer exists.
- *Unregister*: It's called in the logout phase in order to delete the registry record related to the user, notifying all connected users.

2.3. Tomcat

2.3.1. Data folders

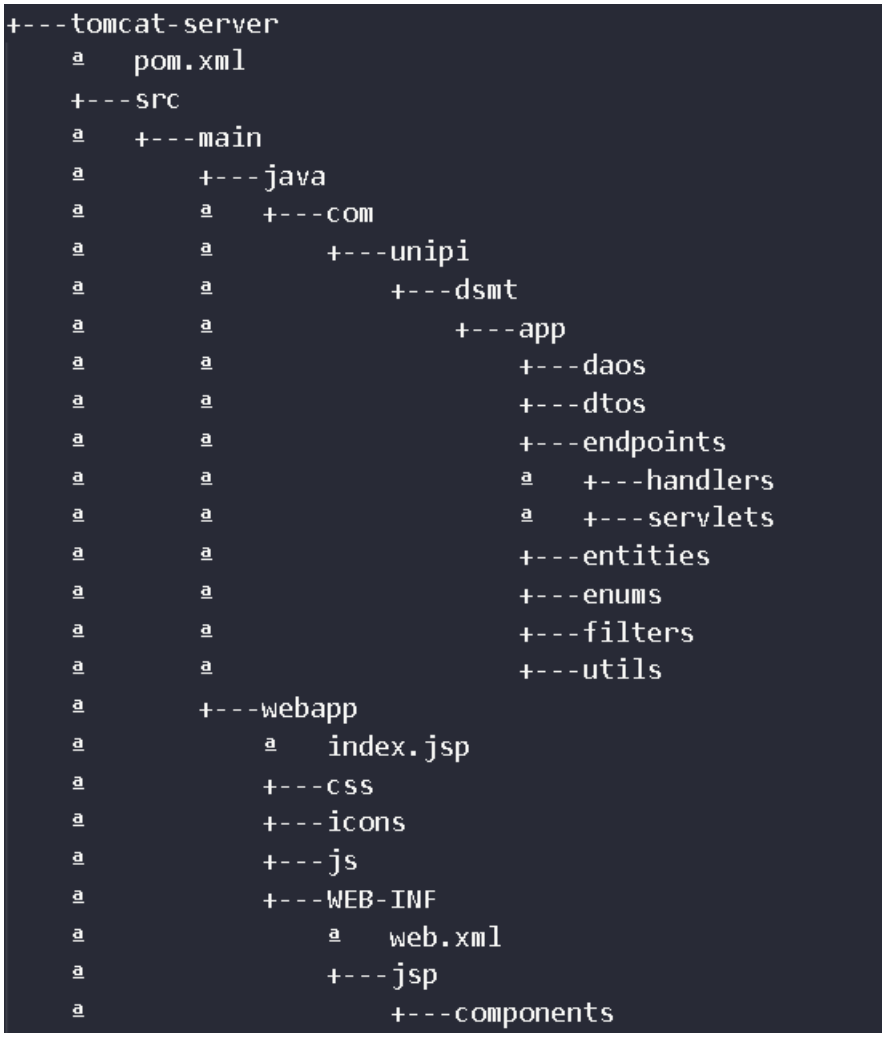


Figure 2.4 - Data Folder Structure

The project is structured in the following ways:

- *daos*: This folder contains Data Access Objects (DAOs) used to initialize and interact with the database or other data sources.
- *dtos*: This folder contains Data Transfer Objects (DTOs) that represent data objects used for transferring data between different parts of the application.
- *endpoints*: This folder contains classes related to application endpoints or routes
    - *handlers*: Subfolder holds request handlers for specific endpoints.
    - *servlets*: Subfolder containing servlet classes for handling web requests.
- *entities*: This folder holds entity classes representing objects in the application domain in order to handle more easily object useful for the database.
- *enums*: This folder includes enumerated constants used in the front end page for department selection during the registration phase
- *filters*: This folder contains filter classes used to intercept request in particular the access filter class that validate the authorization token.

- *utils*: This folder includes utility classes that provide various helper functions and methods for the application, such as error handler and access control functions.
- *webapp*: This folder holds web-related resources.
- *WEB-INF*: This is a standard folder for configuration files and resources. It contains the JSP components and pages.

2.3.2. Java

Java interacts with the database through **DAOs** (Data Access Objects) exploiting the entities java files to model MySQL tables. Each DAO exposes the main operations towards the database.

It interacts with the client application through the **DTOs** (Data Transfer Objects), modelling the request and the response parameters.

Java exploits an app **context listener** to initialize the database connection and store it in the servlet context so that it is shared among all the servlets.

Servlet Endpoint

SERVLETS TABLE			
Name	Endpoint	Methods	Description
ChatID Servlet	"/chatID"	GET	Retrieve the chat ID using the username of the participants
Chat Servlet	"/chat"	GET	Verify that the specific user is allowed to the chat and then retrieve all the related chat messages and deletes all the notifications about the chat and renders the chat jsp
		POST	Verify that the chat between the two specific users doesn't' exist, if so, a new chat will be generated and the users will be redirect to the chat page only after the chat ID is retrieved
		DELETE	Check if the chat already exists, if so, the chat will be deleted and a notification will be sent to the user staying on the chat page
Department Servlet	"/department"	GET	Retrieve a list of user belonging to a specific department rendering the department jsp
Home Servlet	"/home"	GET	Check which users are online returning a list of the user ordered by online flag and renders the home jsp
Login Servlet	"/login"	GET	Check if there already exist a authorization token, if there is the user will be redirect to the home servlet otherwise it will render the login jsp
		POST	Check and validate user credential and eventually set a new token for a new access and redirect to the home servlet
Logout Servlet	"/logout"	GET	Set the online flag to false, making no longer valid the session and redirect to the login servlet
NotificationCount Servlet	"/notificationcount"	GET	Allow to retrieve the total number of notifications related to a specific user
Notification Servlet	"/notification"	GET	Take the notifications grouped by user and render the notification jsp

Profile Servlet	“/profile”	GET	Once the user information and the existing chat for that user are retrieved it will render the profile jsp
SignUp Servlet	“/signup”	GET	Render the signup jsp
		POST	Collect the inserted information push them into the database for a new registration and generate a new authentication token then redirects to the home servlet

2.3.3. Webapp

The project section relative to the webapp contains some useful sources to format and build the pages such as the CSS and JavaScript files and secondary sources like icons and sounds.

The most important content is the WEB-INF folder with the **jsp folder** with inside all the necessary jsp files, and a component one that expose the **navbar** actually present in all the pages of the application, fundamental component since it handles all the functionalities about the notification system in an interactive way.

JavaScript key functions
At the beginning of the *chat.js file*, a **WebSocket connection** relative to the specific logged user is opened.

This file contains important functions to handle the WebSocket operations such as

- *handleSend*: Allows to send a message in the classic JSON format through the connection already established.
- *onmessage*: Handles the incoming messages.

The two functions are tightly coupled: when a user sends a new message with the “*handleSend*” function, it is first added to the chat and displayed to the user. Subsequently, the Erlang server receives the message and attempts to save and send it successfully. In the event of Erlang encountering an issue, it will return an error. The "*onmessage*" function, capable of detecting the error, will then highlight the current message in a different color, indicating that it was not sent.

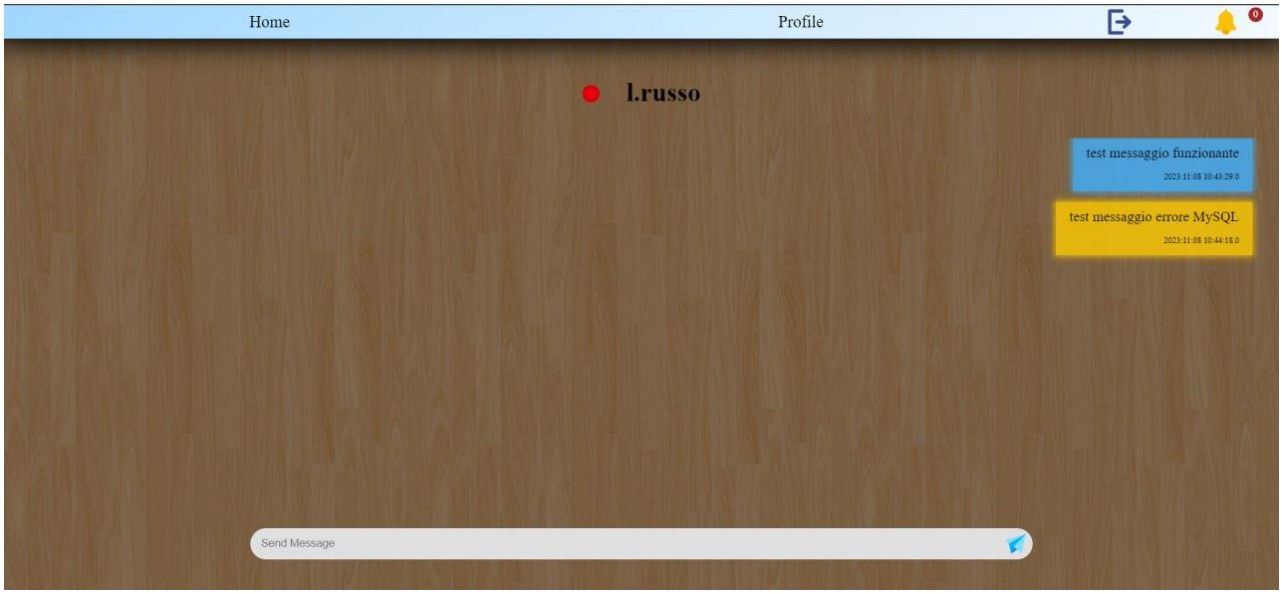


Figure 2.5 - Error message

As seen before, also for the *navbar.js file* a new **WebSocket connection** is established at the beginning.

The main function of this file is the *onmessage* function, it handles several types of notification received by the erlang notification endpoint:

- *online notification*: Received when a new user logged into the app, used to change the online flag of the relative user.
- *offline notification*: Received when a new user logged out the app, used to change the online flag of the relative user.
- *message notification*: Encountered when a user send a new message and the current user is not on the relative chat page, used to update the notification badge counter.
- *chat deletion*: Notify the current user on the chat page that the other user deleted the chat.

The JavaScript sources are processed in **deferred mode**, loading them at the outset. It's only after the HTML page is rendered that the JavaScript files are compiled.

2.4. MySQL Server

To ensure consistency, availability of data and permanence **MySQL** was used as Database hosted in a server with IP address 10.2.1.60.

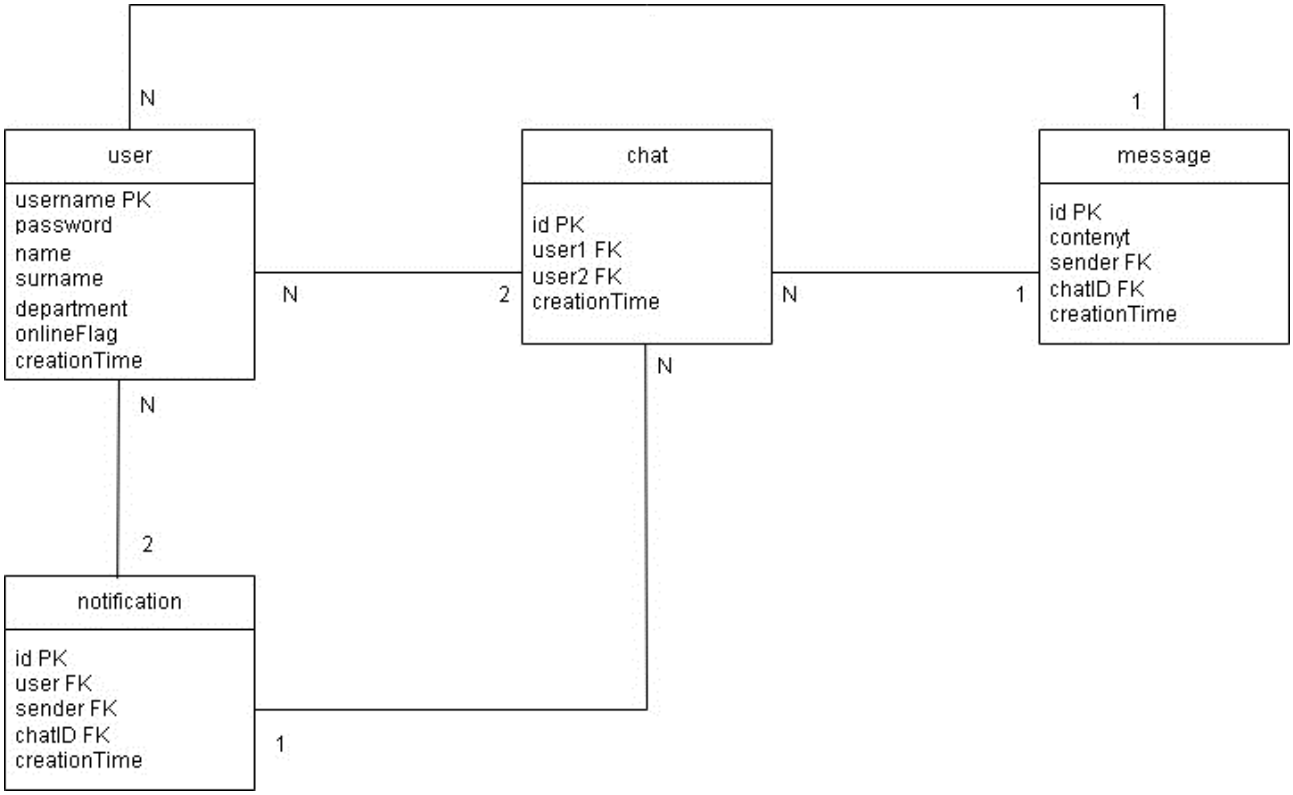


Figure 2.6 - Entity-Relation diagram of MySQL DB

The final structure of the database is described in Figure 2.6.

The database provides only one type of actor for the user modelling the potential employee of a company that can interact with other ones with the possibility to handle its relative notifications and chats.

The table are characterized by the following features:

- **User**: identified by its username chosen at the registration phase, there is associated with the user an encrypted password and an online Flag useful to handle in a dynamic and interactive way the notification system and by a department belonging to.
- **Chat**: this entity acts as a collector for notifications and messages for a user.
- **Message**: entity representing the message text exchanged between two possible users and it's used for persistent messages handled in Erlang.
- **Notification**: entity modelling the possible notification related to two user and a chat, handled like for message entity by Erlang.

3. User manual

3.1. Login page

The root page of the web-app allows a user to login, by entering its username and password.

After that the user clicks the login button, if the login operation has been successful, the user will get redirect to the home page, otherwise an error message will appear.

It is also available a link to the sign-up page for unregistered users.

Figure 3.1 - Login page

3.2. Sign-up page

This page allow unregistered users to sign up to the app.

It can be accessed from login page and shows a list of fields that the user is required to fill with its personal information in order to sign up.

After that the user clicks the sign-up button, if the sign-up operation has been successful, the user will get redirect to the home page, otherwise an error message will appear.

It is also available a link to allow the user to return in the login page.

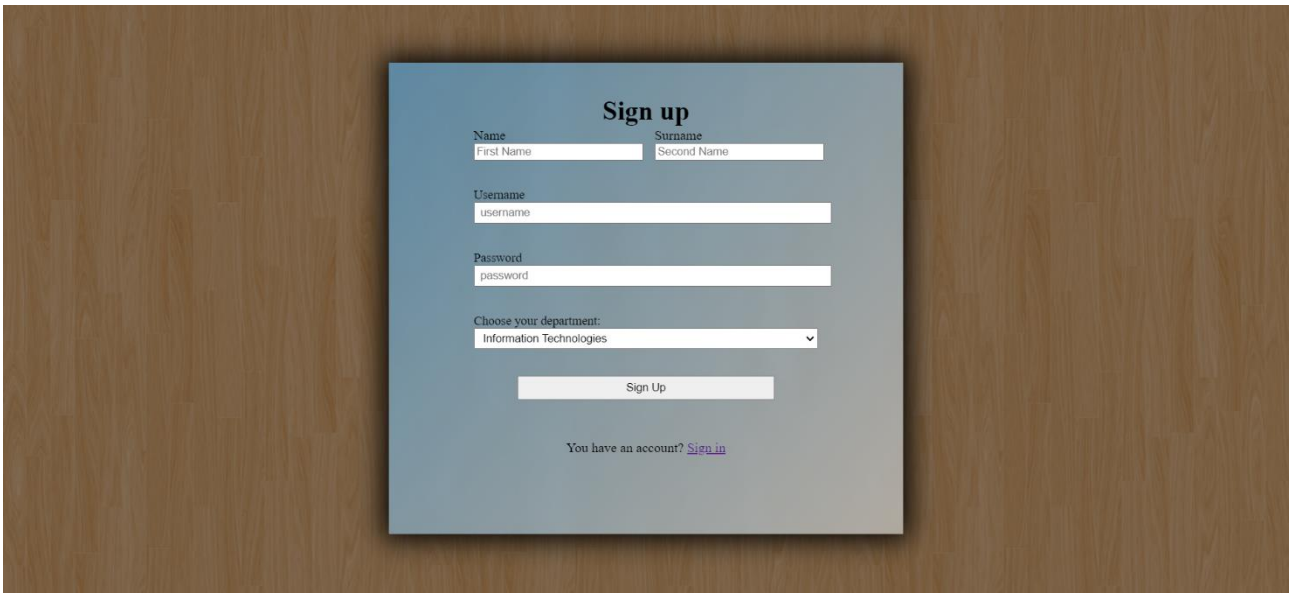


Figure 3.2 - Sign-up page

3.3. Home page

This page shows the user home page.

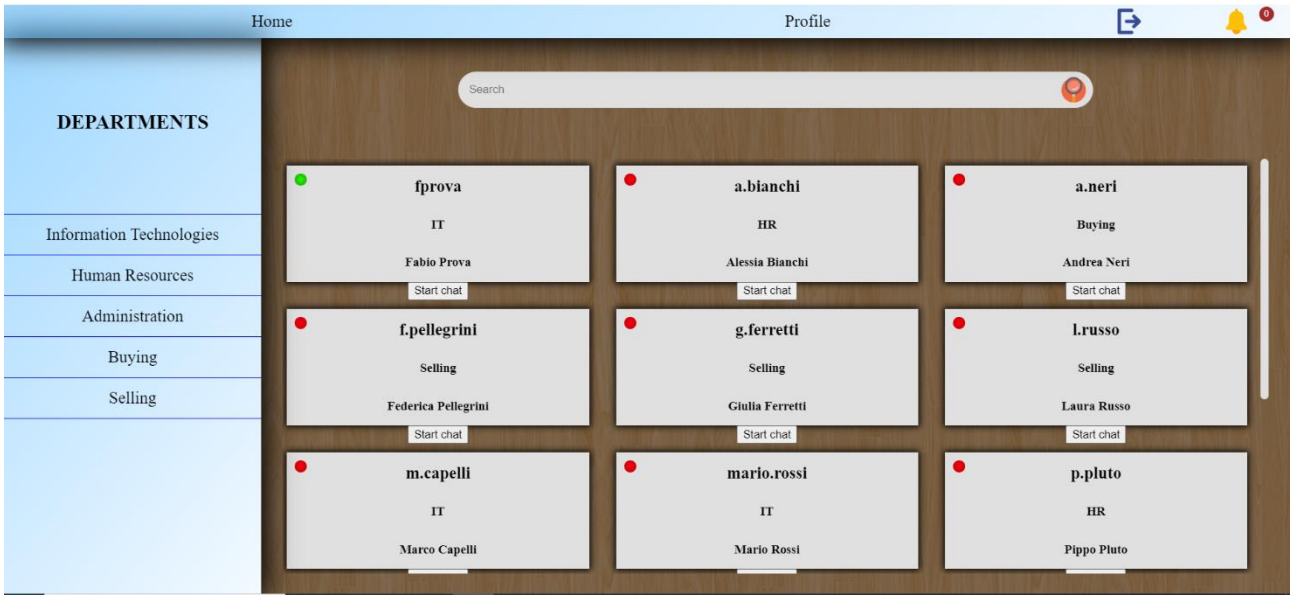
It displays all the company employees with their state (online/offline).

The user can start a chat with a specific employee by simply click on the start chat button.

The user can search for a specific employee, using the search bar, searching by username or full name.

The user can filter the company employees based on the department in which they work.

In the home page, there is a navigation bar (at the top of the page), that is also present in all the other pages. Through this navigation bar, a user can return to this page, go to its profile page, view its notification or logout.



3.4. Profile page

This page shows the user personal information and its recent chats.

The user can browse the list of messages related to a specific chat by simply click on the chat box.

The user can also delete a chat by simply click on the trash icon.

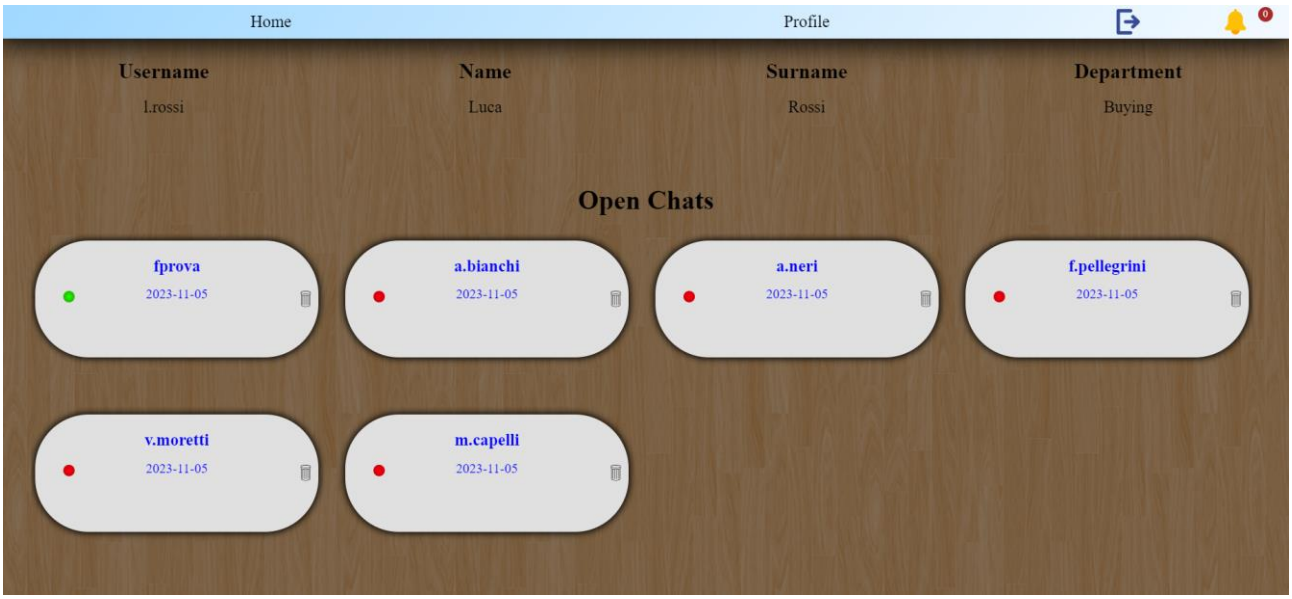


Figure 3.4 - Profile page

3.5. Chat page

This page shows the list of messages related to a specific chat.

The user can send a new message through the provided input field.

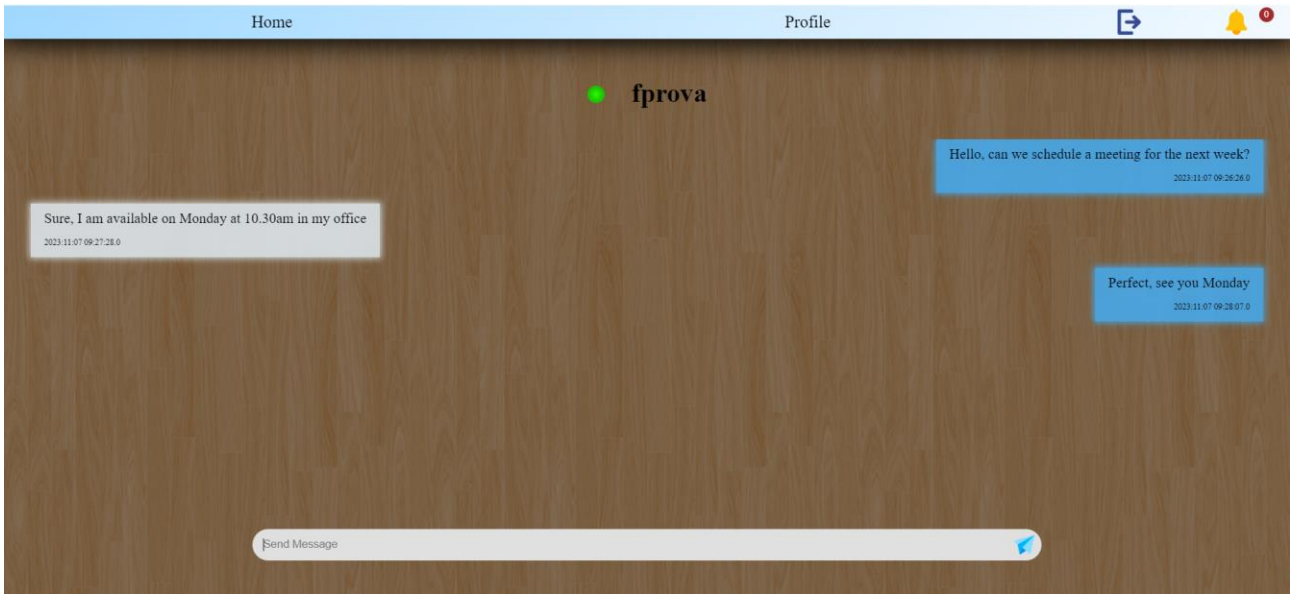


Figure 3.5 - Chat page

3.6. Notification page

This page shows the list of notifications about messages received when the user was not online.

The user can click on a notification in order to open the related chat and see the new messages.

Once the user clicks on the notification, it is deleted and will no longer visible.

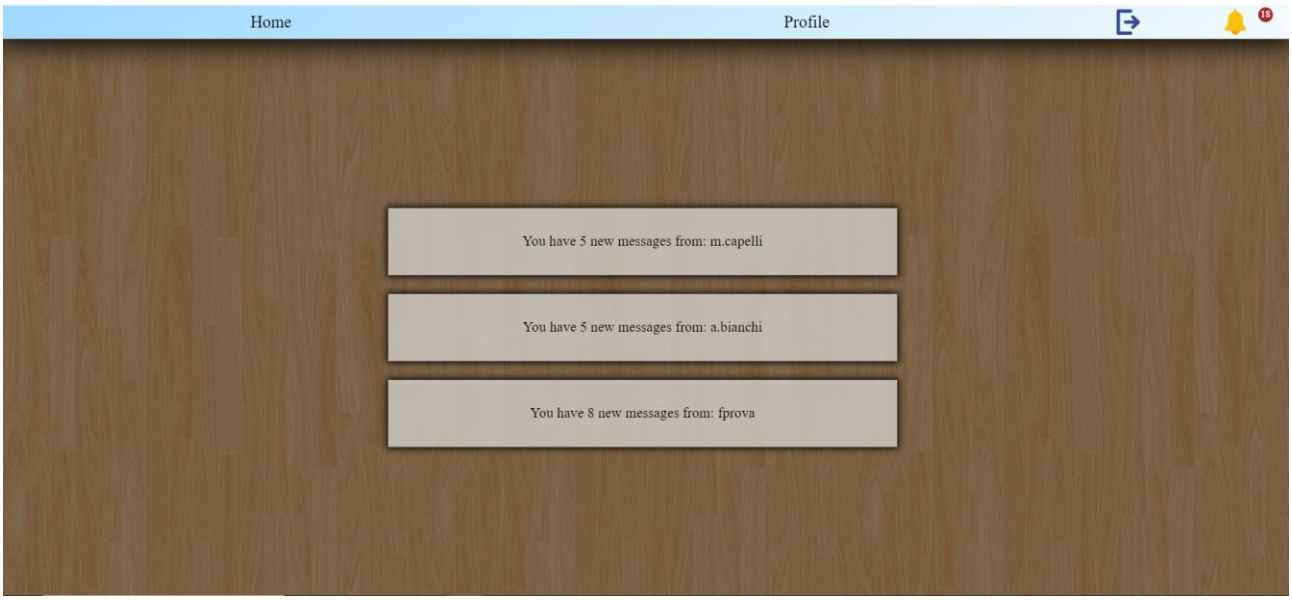


Figure 3.6 - Notification page