UNIVERSITÀ DI PISA

Concurrent and Distributed Systems Project

Reactive Chat

Gabriele Martino

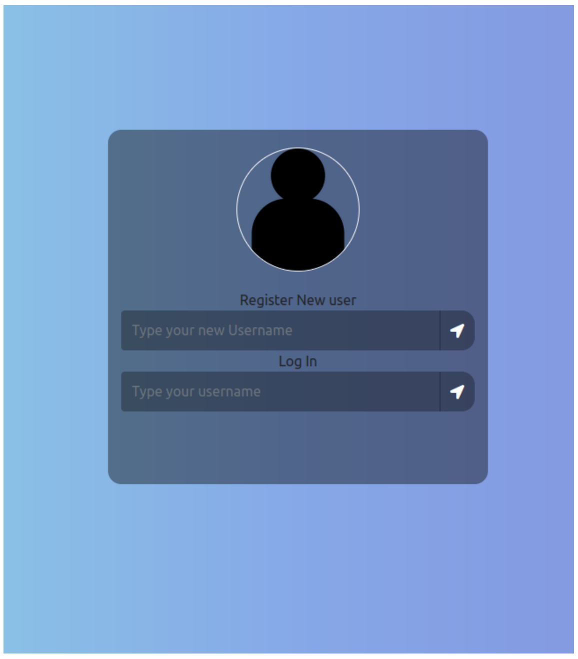
Uses cases

The use cases implemented are made to test mostly all the basic functions provided from the system. The implementation doesn’t cover real uses cases, as security issues or particular situations, because this would require work that goes outside the scope of the project and it’s not explanatory of the main features of the architecture.

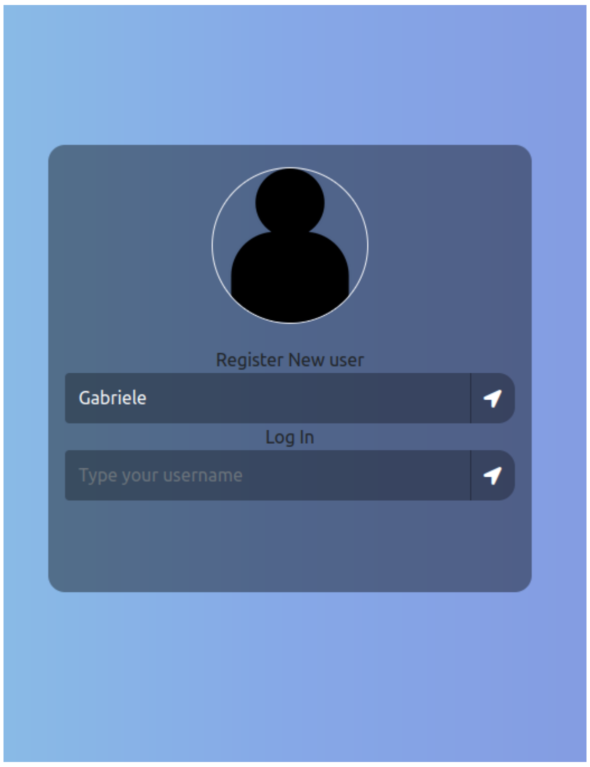
Signup

This use case simply creates an entry in the database. It’s enough to type the name in the specific input and click on the button on the right or type ‘Enter’.

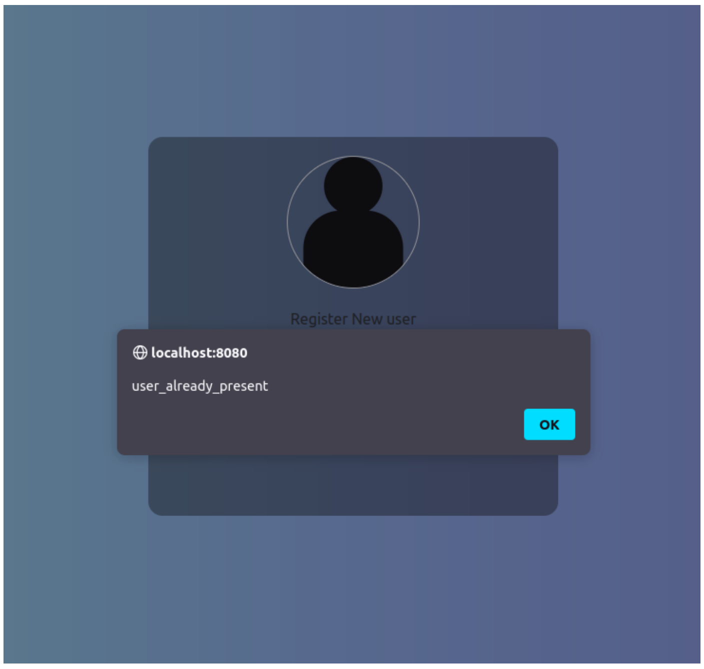
In the case that a user already exists with the same name, this event will be notified.



Login page



Login Page with a typed name for signup

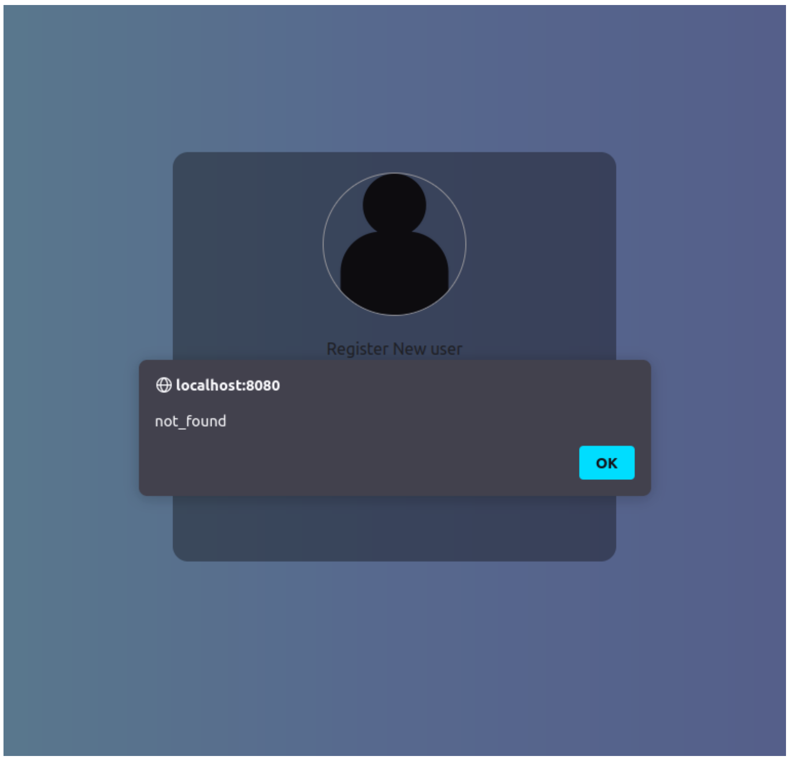


user_already_present

Login page with error for signup

Login

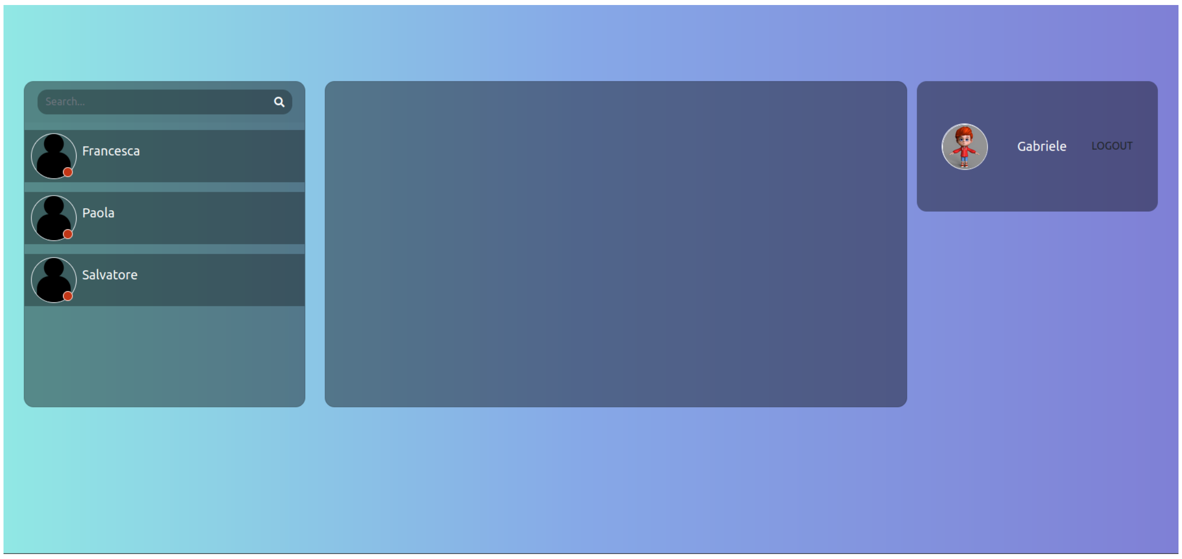
The procedure to login is the same as the signup use case; the difference resides in the input field.
In the case the username doesn't correspond to any already registered user, this event will be notified.



not_found

Login page with error for login

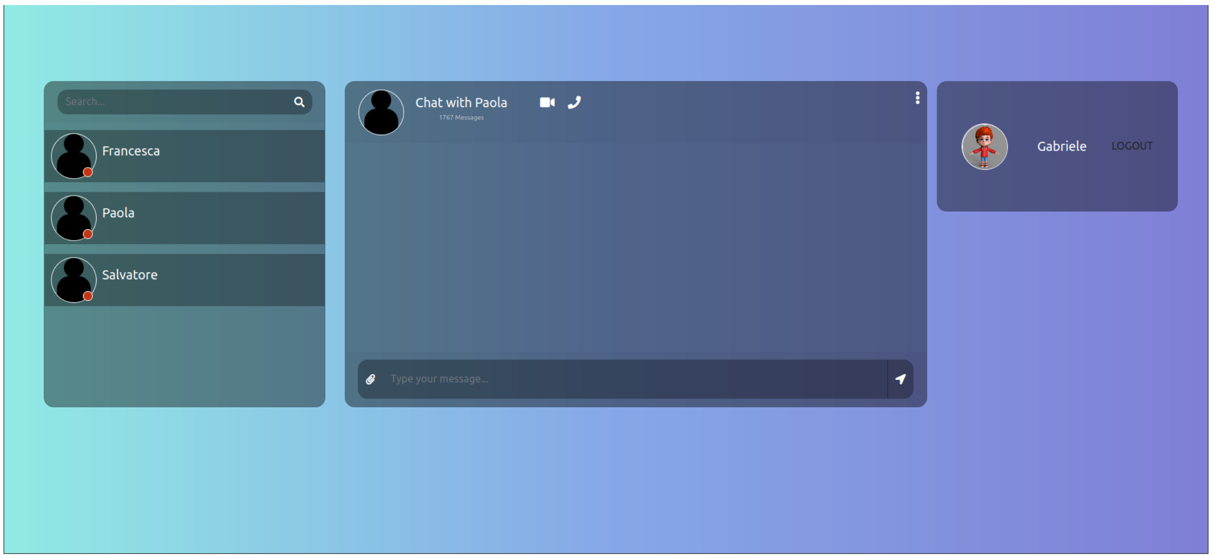
After the login the main page will be shown.
The main page contains three containers with all the utilities for the chat. The left side container contains a list with all the users registered (so it's possible to chat with all the users). The search field for the chat is just a placeholder, the search field has not been implemented. The central container is empty at the startup of the main page, but will contain a chat interface when a user gets selected. The right side container of the UI shows the user's name of the logged user and a Logout button.



Main page

Select a chat

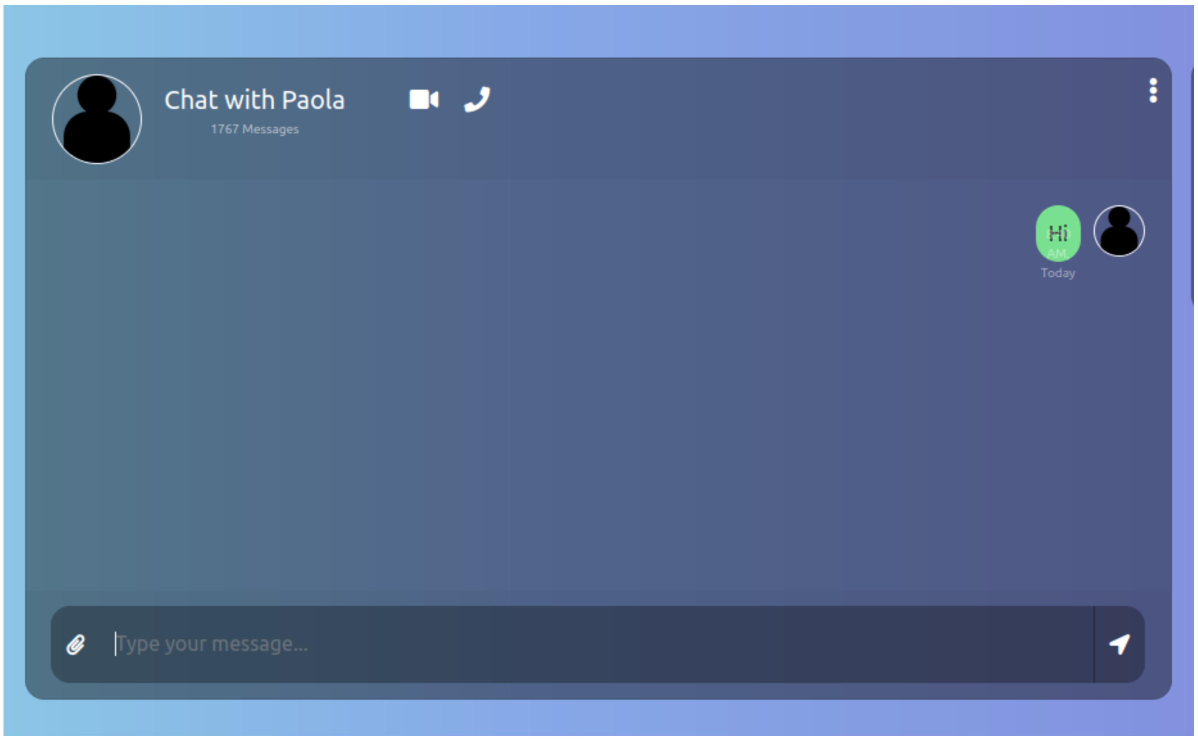
To select a chat is enough to click on the entry of the user you want to chat with, on the chat list on the left of the UI. This action will open the chat on the central container of the interface. It's possible to change the chat at any moment.



Main page with a chat selected

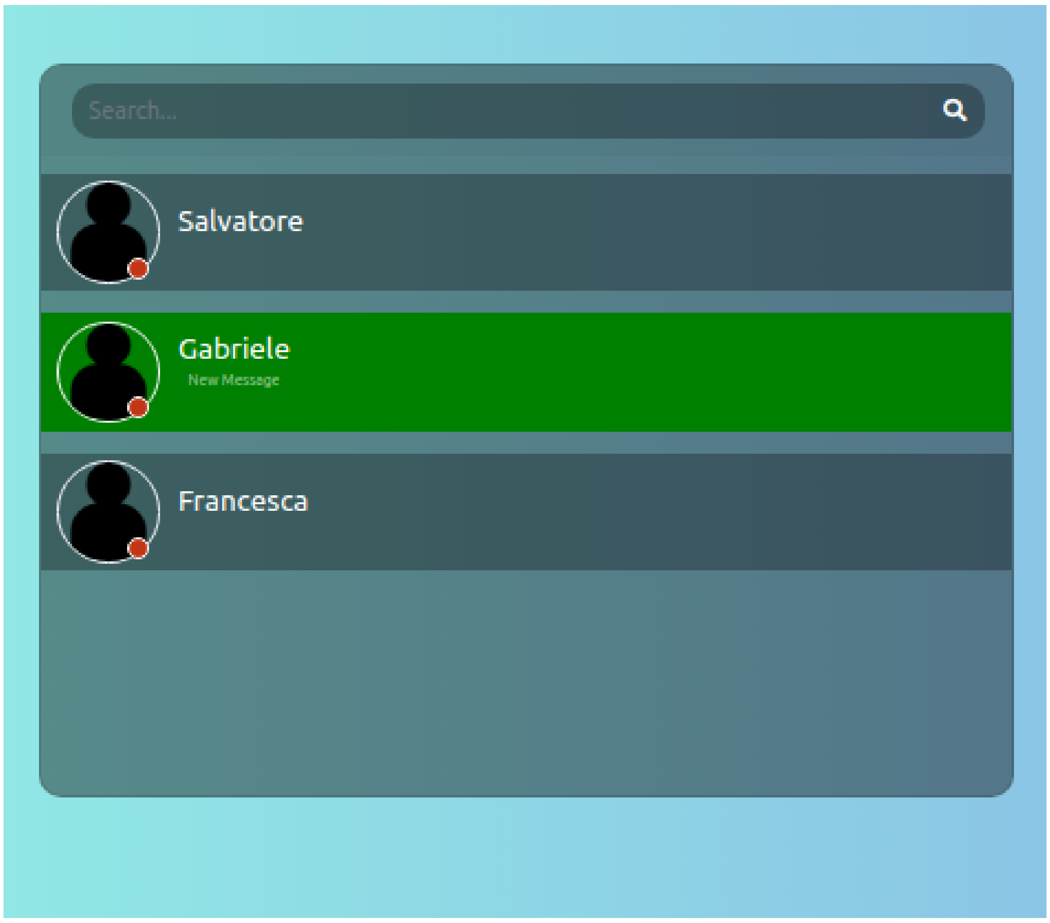
Type a message

Once a chat is selected it's possible to send a message to the user.



Chat container with a message typed

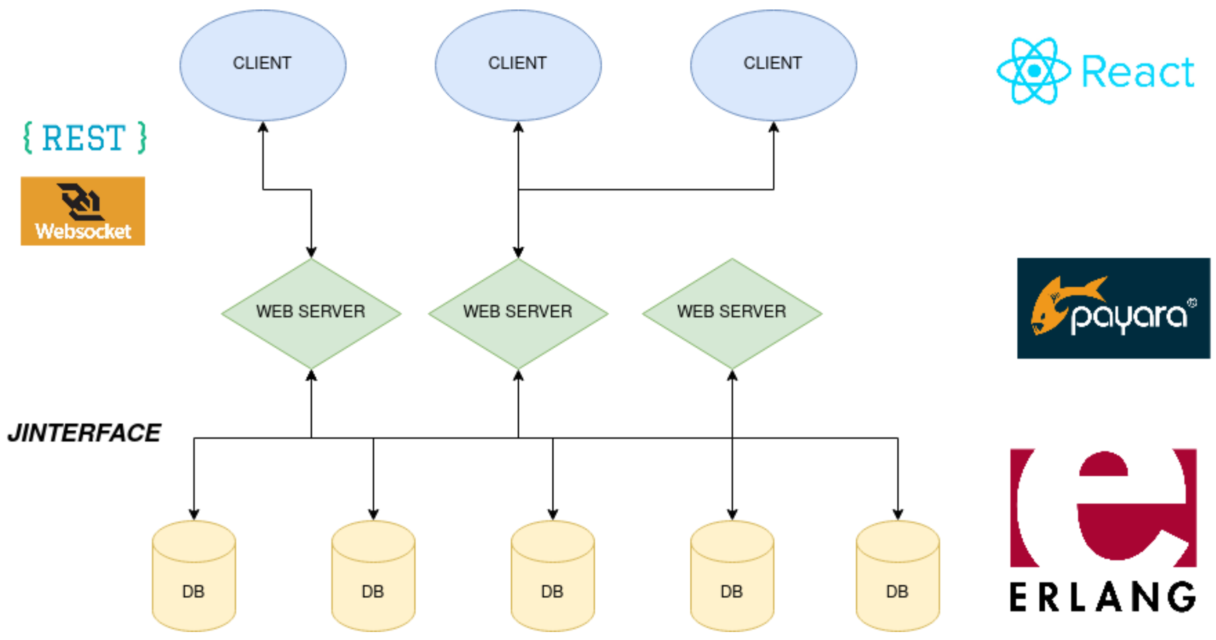
If the user destination of the chat is online at the same time and also his chat is open, the message will be received immediately. Otherwise, if another chat is open a notification will appear.



Notification for a new message arrived

The Architecture

Nowadays the applications need to cope with a huge amount of users connected at the same time. Moreover, the user can be in any location around the world and the response time has become primary in the requirements of the applications. For this reason a distributed approach is mandatory.
The application offered is a WEB application, so it's necessary to develop a UI able to offer all the features promised. For this, we choose React, one of the most used JS frameworks, that allows an easy and reacting change of the interface based on events.
The DB used for this application is Mnesia, a distributed DB built on the top of ERLANG, a functional programming language that helps communications among nodes. This DB allows a distribution of the data almost with no developing effort.
The glue of these two parts are the web servers. The web servers offer the web site to the client and act like dispatcher of the requests between the client and the DB (further more details).



Architecture of the application

Communication Client - WebServer

When developing an application it's important to establish a communication interface between the parts. The communication between the client and the web server that has been developed is of two kind:

- **REST** : all the requests are POST HTTP demultiplexed with URL. This it's used for almost all the user requests.
- **WEBSOCKET**: since this allows a bidirectional channel between client and server, it's perfect for the asynchronous receiving of the messages for the chat.

The payload of all the message is a **JSON** file with a defined structure:

REQUEST
{
    request: *type of request*
    value: *value*
}

RESPONSE
{
    response: *type of the response*
    value: *ok | error*

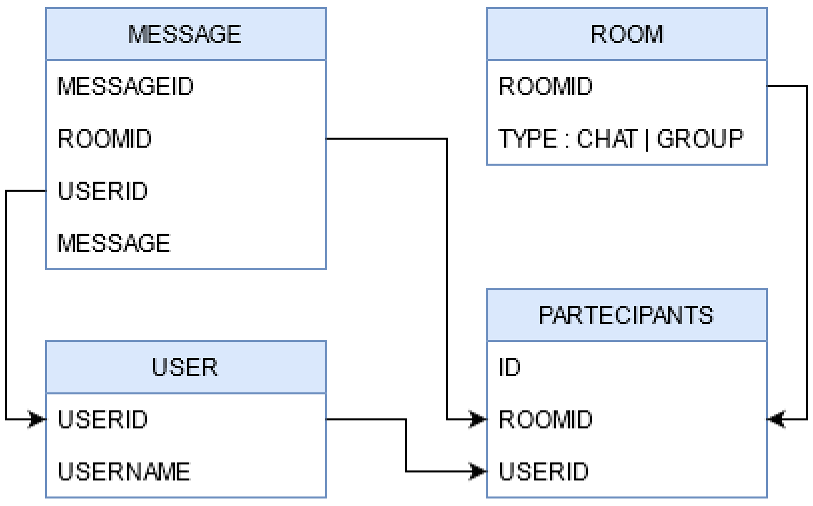
    data: *the payload if value = ok*
}

Communication WebServer - Erlang Nodes

The communication between web servers and erlang nodes is built using ***JINTERFACE***. This is a library that executes a pseudo-erlang node inside a Java environment. This allows communication to erlang nodes transparently. This approach unfortunately causes issues at the network level if used with JAVAEE . The creation of an erlang node makes a new entry of a node name in EPMD (The DNS-kind of app in the system that keeps track of the erlang node in the system), beside a socket port. The JAVA EE environment creates instances of some classes (EJB or Servlet) to allow the thread-pools of the server to easily and faster handle more requests. The multiple creation of theses instance can create an inconsistent multiple creation of an erlang-node inside the Java environment, so for this reason it's primarily important to handle the unicity of these nodes (using *static variables*, *volatile*, *synchronized* methods, *Singleton* annotation and so on).

The Database - MNESIA and Erlang Nodes

Mnesia DB offers an easy handling of a distributed database that matches our requirements for this application. In mnesia, as in others dbs, it's necessary to schematize the database tables on which we want to work on.



The little scheme shown above wants to offer a system that allows the creation of Chat with only two users, or Groups (this has not been implemented). The ***room*** table contains an entry that represents the Chat, that can be of a couple of users or for a group; the ***participants*** table contains the users inside the room (so two entries for the same room if it is a chat, or multiple entry if the room is a group)

Chat creation

In our application everyone can chat with any other user in the system. According to the schema this brings to a huge amount of entries in both participants and room tables (N x N entries in room table, 2*(N x N) entries in participants table). For this reason, so for efficiency, we decided to create a new room only when necessary.
When a logged user clicks on the name of another user to chat with him, he requires, if not already created, the making of a new room. This happens following this procedure:

- **Open a new chat**: if the room between the two users doesn't exist, send a request to server, otherwise simply open it;
- The request is forward to the DB that:
    - **Create an entry in the ROOM table;**
    - Insert two entries in the PARTICIPANTS table, one for the requirer one for the other user;
    - Send back the room information to the requirer;
    - If the other user of the room is logged, send a new event to it.

The event of the creation of a new room for the other user of the chat is particularly important. That's because all the users need an updated list of all the chats to receive a real time notification from the other users (this will be clear later).

Event Handling

The DB servers have been implemented to handle two kinds of events: **new message insertion, new chat creation.**
All the erlang nodes are connected to each other, so it's straightforward to reach any of these endpoints. When one of those requests arrives at one node, **this forwards the event to all the other nodes.**
When one node receives an event from another erlang node, **this checks if it has registered web servers. In that case it dispatches this event to all of them.**

Web Server

Requests

The web server is the most crucial part of this application. We used JAVA EE over **Payara**, the descendent of Glassfish, one of the main Java Servers used in enterprise applications.
The requests of the users can be of two kinds: **REST** and **WEBSOCKET**. The REST requests are handled by two classes "***AuthenticationWebServlet***" and "***UserRequestsWebServlet***".

AuthenticationWebServlet implements:

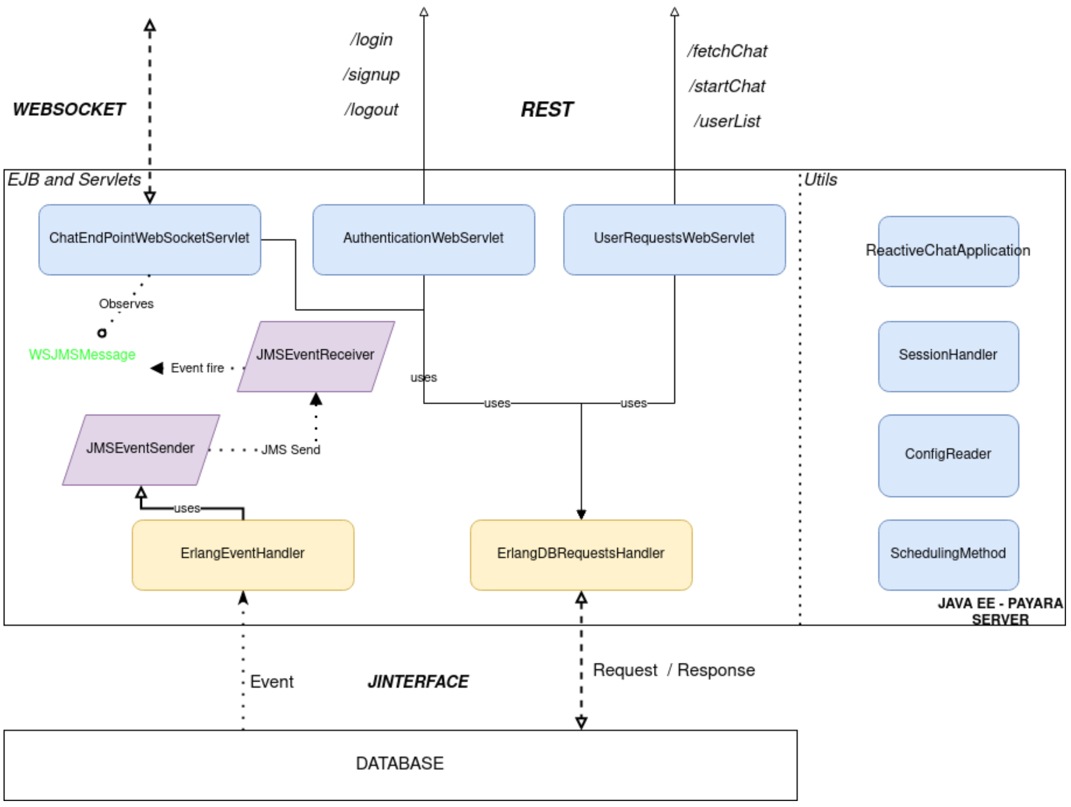
- **/login**
- **/signup**
- **/logout**

UserRequestsWebServlet implements***:***

- ***/fetchChat :*** *require the list of all the chats*
- ***/startChat :*** *create a chat between two users (the requirer and another)*
- ***/userList :*** *require the list of all the users*

When a user logins to the application, it requires the list of the chat and of the users using **/fetchChat** and **/userList.** These two lists are combined together. If the logged user is present in some of these chats, these are used as placeholders for the other peer. In those cases in which it is not present, the information of the peers are taken from the list of users (see Chat creation above).
If the room is not present, but the logged user wants to chat with the other peer, it uses its information to send a **/startChat** request to create a new one.
All of these requests are forwarded to the DB using ***ErlangDBRequestHandler*** EJB.

Besides the REST requests we also use a websocket. This is created at login. Through this socket are sent the messages (in and out) and the new chat event.



Web server structure

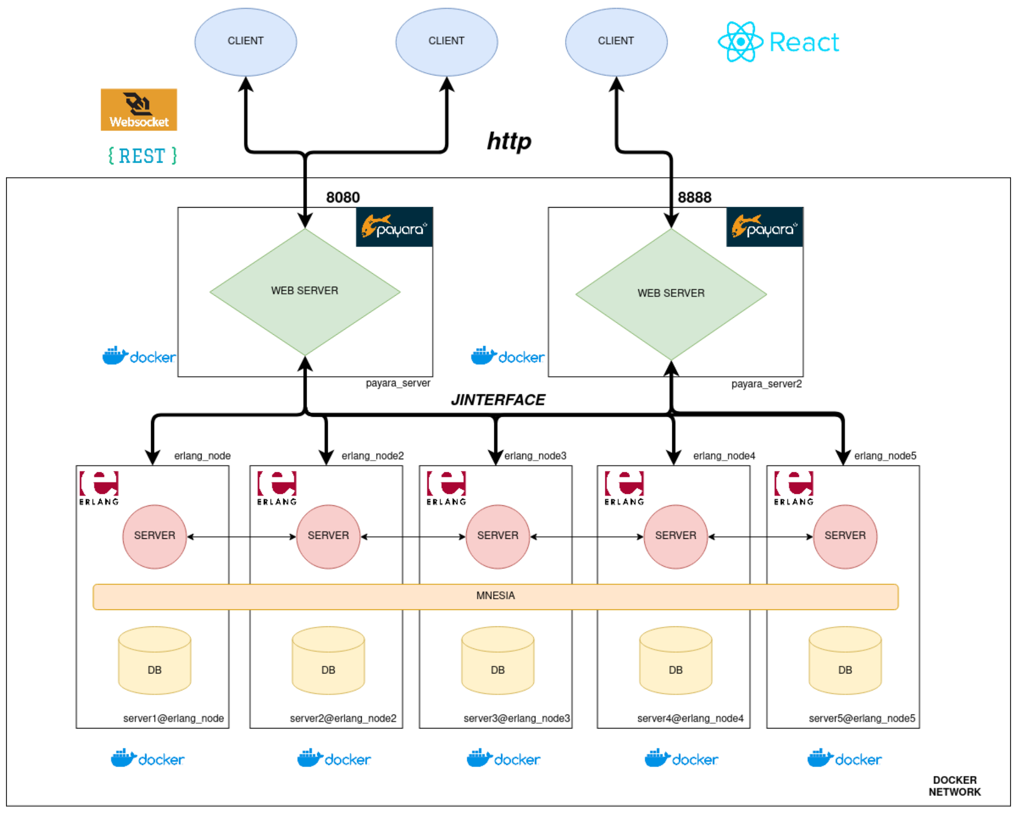
Event handling

The ***ErlangEventHandler*** EJB offers a specific thread that uses ***JINTERFACE*** with a specific erlang node that waits for new events from the DB. These events are decoded from erlang tuples and re-encoded in JSON. Then it uses the JMSEventSender over JMS messages system and arrives at the JSMEventReceiver. This one fires an internal event that is observed by ***ChatEndPointWebSocketServlet***.

This servlet stores a list of all users connected with a websocket and a list of all the rooms in the db. When a new message event arrives, the servlet instance checks if the destination peer of the message is connected at that moment, so it will forward the message to it.

Deployment and Tests

The ideal deployment is to distribute every ideal node into a real one. In our test case, we deployed every node into a docker container. These docker containers are all into a docker network in such way the communications are possible.


CLIENT
CLIENT
CLIENT
React
Websocket
{ REST }
http
8080
8888
payara
WEB SERVER
WEB SERVER
docker
payara_server
payara_server2
JINTERFACE
erlang_node
erlang_node2
erlang_node3
erlang_node4
erlang_node5
ERLANG
SERVER
MNESIA
DB
server1@erlang_node
server2@erlang_node2
server3@erlang_node3
server4@erlang_node4
server5@erlang_node5
DOCKER NETWORK