Second Cycle Degree in Computer Engineering

Connect for Help
Distributed System and Middleware Technology

Leo Maltese, Antonio Di Noia, Alberto Caruso

Academic Year 2020/2021

Contents

1 Introduction

In this period of pandemic, the weakest sections of the community have remained isolated. Disconnected and unable to access the most common services, such as the purchase of basic necessities (food, clothes, etc.). Connect For Help is a web application where the requester users, those who in some way need services, can post their requests in a bulletin board and where the performer users try to satisfy them. Everything is managed by an easy to use platform. Connection For Help also implements a chat where the requester and the performer can coordinate and exchange information before the requested service is satisfied.

2 Architecture

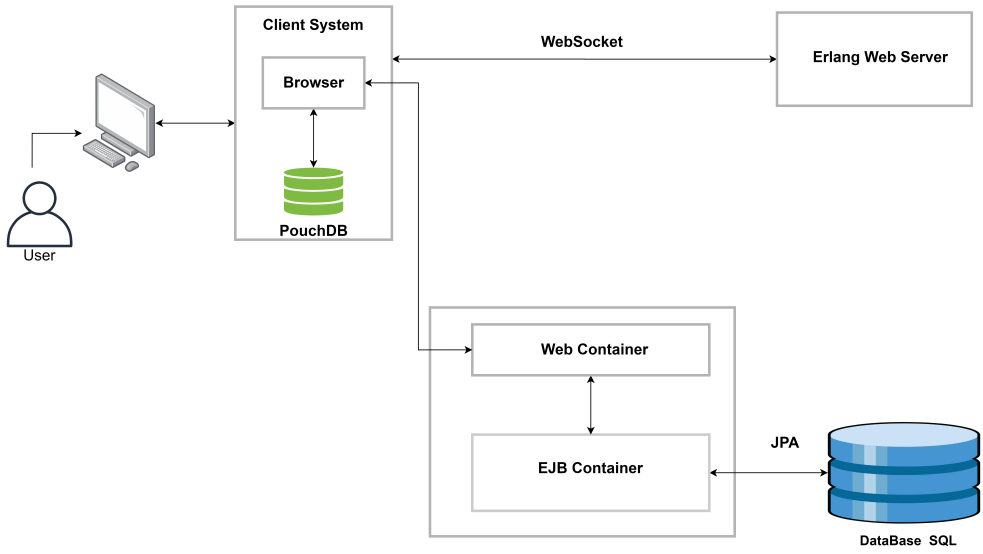

Figure 1: Architecture Structure

The web application architecture consists of: a Client System, an Erlang Web Server and an Application Server that communicated with a relational database. The Client System is the system that interfaces with the user, who, to use the service offered, will browse a website (implemented with the use of html / css / javascript) via a browser. Inside the System client there is also PouchDB, this tool is a NOSQL database that is used to keep track of the messages that the user exchanges in chats in a single session. Continuing from this last point, the communication between two users, and therefore the exchange of messages through a chat, is allowed with the use of Erlang Web Server (obviously implemented in Erlang language) which takes care of exchanging messages between

users and, moreover, of keeping track of the users are online in the site. The communication between Client System and Erlang Web Server takes place thanks to Web Socket. This technology is defined as a messaging protocol that allows asynchronous and full-duplex communication over a TCP connection, in particular, Web Sockets are not http connections even if they use http to initiate the connection. Abut the connection, this was managed via javascript web socket as regards the client side, as regards the server side, he connection was managed with the use of web socket but with the help of an external library called Cowboy. Therefore the exchange of messages between two users takes place via the following path: client $\rightarrow$ server / server $\rightarrow$ client.

For the connection between the Client System and the Application Server (Glassfish 5.1.0), the web application uses the REST services to allow communication and recovery of the various resources / services offered. The main characteristics of this type of communication are:

- A REST service provides resources and not methods.
- The format used, in this case, to make the resources available is JSON.
- The REST model is usually implemented via the HTTP protocol, and therefore on a Client-Server type architecture. The operation provides a URL structure that uniquely identifies a resource or a set of resources and the use of specific http methods for retrieving information or for modification (GET. POST, PUT, DELETE etc.).

About the Application Server, briefly, through the use of EJBs, it deals with managing the resources made available to the user.

Finally, a relational DB (MySQL) has been inserted into our architecture, to allow the storage, for example, of user data, of the services that are available, etc. JPA and JDBC Connection Pool technology was used for communication between the application server and the DB.

The various components introduced in this chapter and used in the architecture will be explained in more detail below.

3 Implementation

In this chapter we will focus on the implementation choices made for the deployment of the application.

3.1 Database

As already explained above, for the back end, MySQL was chosen as DBMS.

3.1.1 Database schema

In Figure 2 the E-R diagram of the database schema is shown. The entities are: *User*, *Service* e *Category*.

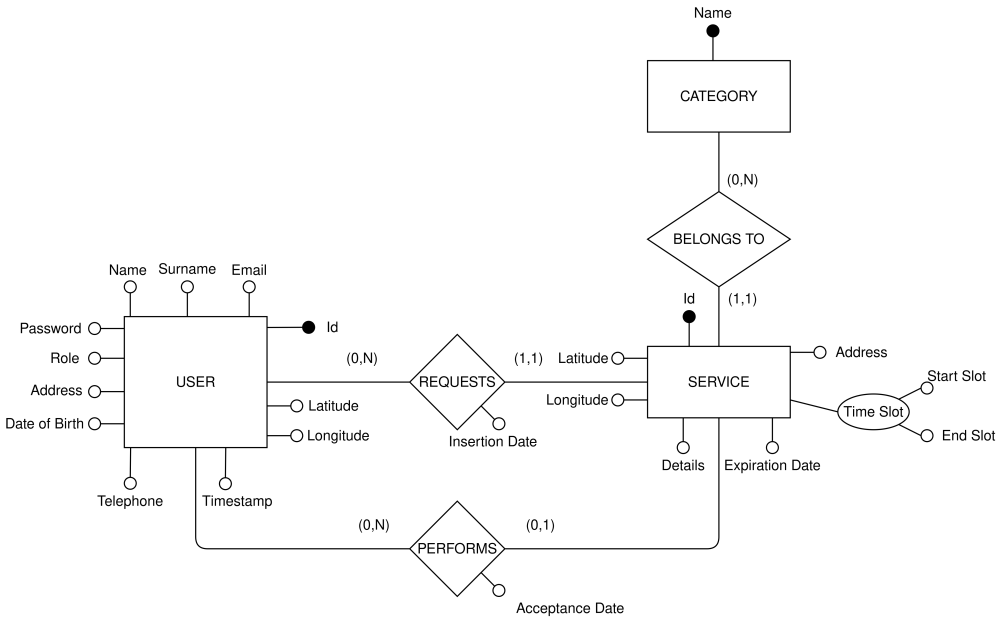

Figure 2: E-R Diagram

The *User* entity represents the users who have been thought as users of the application. In order to distinguish the two different types of users, a field, *Role*, has been inserted and it can only assume P (for the Performer) or R (for the Requester) as values. In this table a constraint of uniqueness is present on the *Email* attribute.

The *Service* entity represents the services that a Requester user can request, and a Performer user can perform. These services have a time window, *Time Slot*, chosen by the requester when the service is created, in which they must be performed, with the constraint that this time window is placed within a single day. Moreover, there is an *Expiration Date* field which represents the deadline for subscribing as a performer of a certain service. In this case, the constraint is, obviously enough, that this deadline has to occur before the start of the time window previously described.

Finally, the *Category* entity represents the type to which a particular service belongs. Examples of categories are the following: "shopping", "keep company" etc.

3.1.2 Persistence: JPA

For the management of data in our SQL database, we used the Java Persistence API which represents a Java specification for accessing, persisting and managing data between Java Objects and relational database (in our case MySQL).

The use of this tool has allowed us, thanks to annotations, to carry out a

mapping between the Java classes and the tables of our relational database, in order to make data recovery easier. Below there are two examples: an example of how the "User" class has been mapped to its table in the relational DB and an example of a possible query.

```
@Entity
@Table(name="user")
@Inheritance(strategy= InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="Role",discriminatorType =
    DiscriminatorType.STRING,length = 1)
public abstract class User implements Serializable {
    private int idUser;
    private String name;
    private String surname;
    private String email;
    private String password;
    private String address;
    private Date dateOfBirth;
    private String role;
    private String telephone;
    private Double latitude;
    private Double longitude;
    private Timestamp timeStamp;

    public User() {
    }

    public User(int idUser, String name, String surname, String
    email, String password, String address, Date dateOfBirth,
    String role, String telephone) {
        this.idUser = idUser;
        this.name = name;
        this.surname = surname;
        this.email = email;
        this.password = password;
        this.address = address;
        this.dateOfBirth = dateOfBirth;
        this.role = role;
        this.telephone = telephone;
    }

    @Id
    @Column(name = "idUser")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    public int getIdUser() {
        return idUser;
    }

    public void setIdUser(int idUser) {
        this.idUser = idUser;
    }

    @Basic
    @Column(name = "Name")
    public String getName() {
        return name;
    }
```

```
50
51      public void setName(String name) {
52          this.name = name;
53      }
54
55      @Basic
56      @Column(name = "Surname")
57      public String getSurname() {
58          return surname;
59      }
60
61      public void setSurname(String surname) {
62          this.surname = surname;
63      }
64
65      @Basic
66      @Column(name = "Email",unique=true)
67      public String getEmail() {
68          return email;
69      }
70
71      public void setEmail(String email) {
72          this.email = email;
73      }
74
75      @Basic
76      @Column(name = "Password")
77      public String getPassword() {
78          return password;
79      }
80
81      public void setPassword(String password) {
82          this.password = password;
83      }
84
85      @Basic
86      @Column(name = "Address")
87      public String getAddress() {
88          return address;
89      }
90
91      public void setAddress(String address) {
92          this.address = address;
93      }
94
95      @Basic
96      @Column(name = "DateOfBirth")
97      public Date getDateOfBirth() {
98          return dateOfBirth;
99      }
100
101     public void setDateOfBirth(Date dateOfBirth) {
102         this.dateOfBirth = dateOfBirth;
103     }
104     @Basic
105     @Column(name = "Latitude")
106     public Double getLatitude() {
```

```
107         return latitude;
108     }
109 
110     public void setLatitude(Double latitude) {
111         this.latitude = latitude;
112     }
113 
114     @Basic
115     @Column(name = "Longitude")
116     public Double getLongitude() {
117         return longitude;
118     }
119 
120     public void setLongitude(Double longitude) {
121         this.longitude = longitude;
122     }
123 
124     @Basic
125     @Column(name = "Role")
126     public String getRole() {
127         return role;
128     }
129 
130     public void setRole(String role) {
131         this.role = role;
132     }
133 
134     @Basic
135     @Column(name = "Telephone")
136     public String getTelephone() {
137         return telephone;
138     }
139 
140     public void setTelephone(String telephone) {
141         this.telephone = telephone;
142     }
143 
144     @Basic
145     @Column(name = "TimeStamp")
146     public Timestamp getTimeStamp() {
147         return timeStamp;
148     }
149 
150     public void setTimeStamp(Timestamp timeStamp) {
151         this.timeStamp = timeStamp;
152     }
```

Listing 1: Snippet Persistence User

```
1     @PersistenceContext(unitName = "MainPersistenceUnit")
2     private EntityManager em;
3 
4     @Override
5     public User getUser(int id) {
6         Query q = this.em.createQuery(
7                 "SELECT u FROM User u where u.idUser=:id");
8         q.setParameter("id", id);
```

```
9           try {
10              return (User) q.getSingleResult();
11          } catch (NoResultException exc){
12              return null;
13          }
14      }
```

Listing 2: Query to retrieve User data

3.2 Application Server

The first choice taken was the one regarding the middleware technology to use. In this case the choice fell on a Jakarta EE Application Server, and in particular on its reference implementation Glassfish.

Application servers are useful in the development of enterprise applications as they automatically address a set of problems, thanks to the use of the EJBs technology and of the EJB containers.

The EJBs are a server-side component (a Java class) that encapsulates the business logic of an application, that is the code that fulfills the purpose of the application. The EJBs "live" within the EJB container, which in turn "live" within the application server. Those EJB container provides the EJBs within it with system-level services such as: EJB life-cycle management, transaction and resource management, scalability, persistance and so on. In this way EJB programmers can focus on the business logic.

Session beans are of three types: stateful, stateless, and singleton. In the application described here only stateless EJBs were used.

3.2.1 Use of stateless EJBs and solution to synchronization problems

Stateless session bean does not store session or client state information between invocations. At most, a stateless session bean may store state for the duration of a method invocation. When a method completes, state information is not retained. Any instance of a stateless session bean can serve any client,any instance is equivalent. Thanks to this, EJB container can pool stateless EJBs, and each time a method invocation is called, it is possible to choose one EJB out of the pool without any issue.

After a close study of the problem, this came out to be the most useful and performing type of EJB for the purpose of the application, given the fact that within the application it was not required to save any particular state information between a given client and a server.

Thanks to the fact that each stateless EJB instance can serve any client, it was not needed to handle synchronization problems, since each client, when performing a method call, will obtain a different instance of the EJB.The only problem might arise if some of the methods of the EJB perform write operations to the database. In this case, the code was properly written to avoid any inconsistency in the database.

One example of such an issue in the application described here is the acceptance of a requested service by a performer user. Since only one performer can subscribe to perform a service, and this implies to write the identifier of the performer in the corresponding service, if, by chance, two users try to subscribe to the same service through the appropriate method call, the one that will actually get the subscription will be the one whose method code is executed first, while the second will find a *non-null* value inside the performer identifier field inside that particular service inside the database and this was handled by simply not performing the write operation, leaving the database in a consistent state.

3.2.2 JDBC connection pooling

To store, organize, and retrieve data, most applications, including the one reported here, use relational databases. Java EE applications access relational databases through the JDBC API. A JDBC resource (called a data source) provides applications with a means of connecting to a database and it is associated to a connection pool, that is a group of reusable connections for a particular database.

For the purpose of this application, one JDBC was created, specifying a unique JNDI name that identifies the resource. Since all resource JNDI names are in the *java:comp/env* subcontext, when specifying the JNDI name of a JDBC resource in the Glassfish Admin Console (reachable at the address *ip_of_the_application_server:4848*) only entering *jdbc/name_of_the_resource* is needed. In this case the chosen name was *jdbc/Connect4Help*.

The advantage of exploiting a connection pool arise from the fact that creating each new physical connection when a database operation is requested is time and computing consuming. Using a JDBC connection pool, the server will instead keep a pool of already available connections (reachable through the JNDI name of the corresponding JDBC resource) to increase the performance: when an application requests a connection, it obtains one from the pool; when the application closes a connection, the connection is returned to the pool, avoiding to spend time in continuously opening and closing connections.

3.3 Structure of the web application

In this paragraph we will show how the application is structured at the implementation level. In fact, seeing the image just below, Figure 3, it is possible observe the different modules, which were useful both for the realization and fro the deployment of the application.

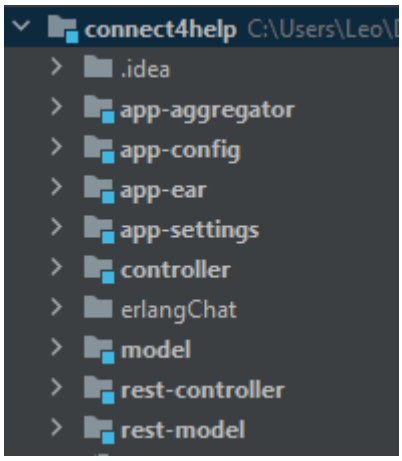

Figure 3: General structure of Application

The first four modules, i.e. *app-aggregator*, *app-config*, *app-ear* and *app-setting*, were used to combine the various modules and to do deploy in a simple and efficient way. In particular:

- **app-aggregator:** in this module there is a particular pom.xml file, called parent pom.xml. The goal of this file is to group all maven projects in order to facilitate the build process.
- **app-config:** this module is used to set some persistence settings.
- **app-ear:** this module is used to assemble and group the various modules into an EAR (Enterprise Archive) file. Therefore the purpose of this module is to make the installation in our application server simultaneous and consistent, in fact it is only necessary to deploy this module to allow the application to be used.
- **app-setting:**module where inside there is a pom.xml file used to set some parameters, in particular those relating to the connection to the DB and the data to recover the jdbc connection pool.

As already said, the modules analyzed above are used for the "construction" of our application, below, however, we will discuss the other modules that represents the main part of the project.

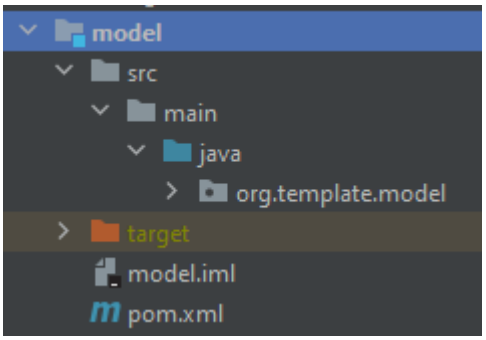

Figure 4: model Module

In Figure 4 shows the simple structure of the *model* module. The single package contains the classes that represent the respective tables in the DB useful for the JPA (User, Service, etc..).

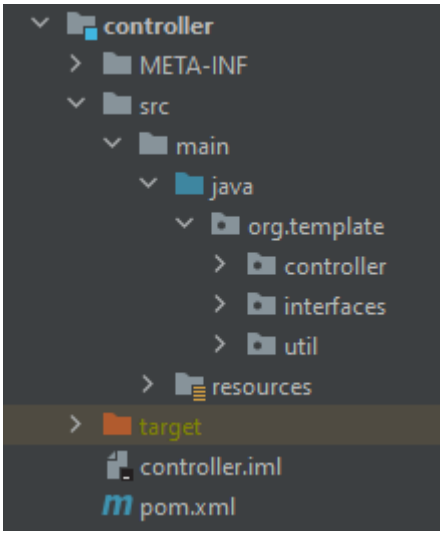

Figure 5: controller Module

In Figure 5 shows the structure of the *controller* module.This module contains three package, in particular:

- **interfaces:** this package contains the interfaces of stateless EJBs used to take specific values from the DB.
- **controller:** this package contains the classes that implement their interfaces.

- **util:** this package contains utility classes for calculating latitude and longitude values.

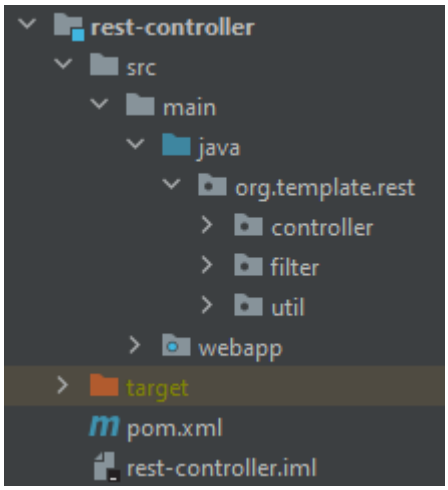

Figure 6: restController Module

In Figure 6 the *rest-controller* module, it represents the web module, in fact inside it there are classes useful for its implementation, in particular:

- **controller:** in this package contains the java classes for the construction of a RESTFUL application, in fact each type of "resource" offered is mapped witch a different path.
- **filter:** in this package are implemented special annotations that verify that the access to a specific resource is made only by a particular type of user through the token mechanism (it will explain later).
- **util:** int this package there are utility classes for implementing *OAuth2* authentication.

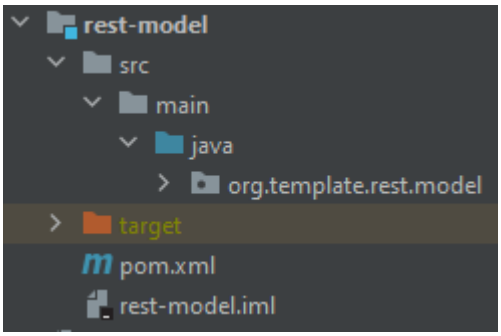

Figure 7: restModel Module

Finally the last module, *rest-controller*, in Figure 7, contains classes that are exploited to retrieve information by front-end, such as: the information of the service, the result of an operation occurred correctly, etc.

3.4 Web app

This module is used for organize the endpoints (API) exposed for User Interface, receive request from UI and forward to application server, receive response to application server and forward to UI. The endpoints exposed are:

- GET /users/ return all users inside the DB
- POST /users/ create user and put inside the DB
- GET /users/id return user with id equal to id if it is inside the DB
- POST /users/login authenticate user and return token,id,role, address very usefull in UI
- GET /users/id/services find the services create by the Requester or the services performed by performer
- POST /users/id/services create service by requester with id equal to id
- DELETE /users/id/services/idService if the user with id equal to id corresponds to Requester, the services with id equals to idServices will be deleted otherwise if the user is a Performer, The service became available for other Performer.
- GET /services?radius=radius;address=address find services that match query.
- PUT /services/idServices/users/idUser The performer with id equals to idUsers is registered to service with id equal to idServices

3.5 Erlang Web Server

The messaging service is implemented using the Erlang language, in particular a simple server is created, which manages the connections coming from the web module, takes care of keeping track of online users, and correctly sending messages to the recivier. The server is divided into modules:

- chat handler
- chat controller
- chat server

3.5.1 Chat Handler

The module is a web server built through the use of Cowboy, is a small, fast and modern HTTP server for Erlang / OTP (https://github.com/ninenines/cowboy). The management of the connections to the web module's web sockets (opening and closing), the sending and receiving of messages from one user to another are his task. Using the API of the chat server module it connects to the chat controller module which takes care of keeping track of users online, then adding or removing.

3.5.2 chat server

Is a simple gen server behaviour offer by OTP libraries, it exposes the API for chat handler module (Listining 2: row 7,11,15), and by then call the function inside the chat controller module

3.5.3 chat controller

Is a simple gen server behaviour offer by OTP libraries. It mantains an ets table(Listining 3:row 5), used to keep track of online user and their associated PID and Id, infact every PID is associated to a websocket. The operation to forwarded a message to the client(Listining 1: row 24), is a job of chat handler module thanks to pid save inside ets table.

Messages from the web module must have a specific format:

- !PING every 20 second the web module send a keep alive message, to maintain active the connection beetween client and server. (Listining 1: row 7)
- !ENTER–>IdSender used for register the user in ETS table, if the operation had success the client is online. (Listining 1: row 10)
- msgFromUser–>IdReceiver used for sending the message to IdReceiver, first the IdReceiver is searched inside the ETS table, if the client is online the message is sent. (Listining 1: row 16)

- !EXIT used for remove the specific client from ETS table. (Listining 1: row 13)

Erlang web server is very simple, it is not maintain the messsage but only the client online, so if the message is sent to a client not online, that message is lost.

Below is a snippet of the chat handler module:

```
websocket_handle({text, Msg}, State) ->
  Details=string:lexemes(Msg, "-->"), % msg

  NickReciver=lists:last(Details),
  Msg_User=hd(Details),
  if
    Msg_User == <<"!PING">> ->
      S=State,
      ok;
    Msg_User == <<"!ENTER">> ->
      S=NickReciver,
      chat_server:enter({self(),NickReciver});
    Msg_User == <<"!EXIT">> ->
      S=State,
      chat_server:leave(self());
    true ->
      S=State,
      chat_server:send_message({self(),{NickReciver,S}},Msg_User)
  end,
  {ok, S};
websocket_handle(_Data, State) ->
  {[], State}.

websocket_info({send_message, _ServerPid, Msg}, State) ->
    {reply, {text, Msg}, State};
websocket_info({timeout, _Ref, Msg}, State) ->
  %erlang:start_timer(1000, self(), <<"How' you doin'?">>),
  {[{text, Msg}], State};
websocket_info(_Info, State) ->
  {[], State}.
```

Listing 3: Snippet Chat Handler module

```

enter({Pid, Id}) ->
    io:format("~p Joine! ~n", [Pid]),
    gen_server:cast(?SERVER, {enter,{Pid,Id}}).

leave(Pid) ->
    io:format("~p leave! ~n", [Pid]),
    gen_server:cast(?SERVER, {leave, Pid}).

send_message({Pid_Sender,{Id_Dest,Id_Sender}}, Message) ->
    gen_server:cast(?SERVER, {send_message, {Pid_Sender,{Id_Dest,
    Id_Sender}}, Message}).
```

Listing 4: Snippet Chat Server module

```

```

```
2  init([]) ->
3      % ets table is used to keep track of online users (nicks)
4      % and their associated pid.
5      Users = ets:new(users,[set]),
6      {ok, Users}.
7
8  nick_list(Pid, Users) ->
9      Nicks = [N ++ " " || {N, S} <- ets:tab2list(Users), S =/= Pid],
10     Pid ! {send_message, Pid, "Online people: " ++ Nicks ++ "\n"}.
11
12 private_message({Id_Dest,Id_Sender}, Pid_Sender, Msg, Users) ->
13     FormatMsg = format_message(Id_Sender, Msg),
14     case ets:lookup(Users, Id_Dest) of
15         [] ->
16             Pid_Sender ! {send_message, Pid_Sender,"Users not
       available."};
17         [{_,Pid}] ->
18             Pid ! {send_message, Pid_Sender, FormatMsg}
19     end.
20
21 check_nick(Nick, Users, Pid) ->
22     case ets:insert_new(Users, {Nick, Pid}) of
23         true ->
24             ok;
25         false ->
26             nick_in_use
27     end.
28
29 disconnect_nick(Pid, Users) ->
30     ets:match_delete(Users, {'_',Pid}),
31     {ok, Users}.
```

Listing 5: Snippet Chat Controller module

In Listining 3, we report the most important function:

- row:8 nicklist user for retrieve all the online client.
- row:12 privatemessage used for forward message to a specific chat handler
- row:21 checknick used for manage the client inside ETS table.
- row:29 disconnectnick used for remove item from ETS table.

The OTP libraries used for build the modules of Erlang Web Server use the concepts of supervisor. It is responsible for starting, stopping, and monitoring its child processes. The basic idea of a supervisor is that it must keep its child processes alive by restarting them when necessary.

For simulate the server we put the Erlang web server inside a virtual machine with a specific IP address and port, in our case we use 8080 port number.

3.6 Front-end

The front-end was implemented through the use of HTML, CSS and JavaScript files. We are not going to focus on the analysis of each of those files used to

realize the graphical interface, but, instead, on how the communication with the application server and the communication with the Erlang server, dedicated to the exchange of textual messages between users, has been allowed.

3.6.1 Communication to the Application Server

As explained above, the request of certain information/resources, coming from the front-end, toward the middleware/back-end, occurs through the use of specific endpoints (API). Such requests has been implemented using the JavaScript language.

```
1  var elements = document.getElementById("signInForm").elements;
2      var obj ={};
3      for(var i = 0 ; i < (elements.length-1) ; i++){
4          var item = elements.item(i);
5          obj[item.name] = item.value;
6      }
7
8      var raw= JSON.stringify(obj);
9      var myHeaders = new Headers();
10     myHeaders.append("Content-Type", "application/json");
11     var requestOptions = {
12         method: 'POST',
13         headers: myHeaders,
14         body:raw,
15         redirect: 'follow'
16     };
17
18     fetch("http://localhost:8080/rest/api/users/login",
       requestOptions)
19         .then(response => response.json()
20             .then(jsonBody => ({
21                 token: response.headers.get('Authorization').
       replace("Bearer", ""),
22                 status: response.status,
23                 jsonBody
24         })))
25             .then(result => setUserInformation(result.token, result
       .jsonBody, result.status))
```

The snippet of code reported above shows an example of how the endpoints made available are used, and, in that particular piece of code, how a user is authenticated during the login phase. From row 1 to row 8 there is the creation of the JSON object to be inserted in the body of the request. From row 9 to row 16 some important values are set, more precisely: the content type of the message, the HTTP request method, the body etc.. Between lines 18 and 25 the request is sent and the relative response received is handled. In order to issue the request, the *Fetch API*, which provides a JavaScript interface for accessing and handling the HTTP pipeline, both requests and relative responses. As can be seen on line 18, a global *fetch()* method, which provides a simple and logical way to retrieve resources, was used. Then, a request was made to the dedicated

endpoint for user verification and the relative response was handled through the then *then(...)* methods provided, again, by the Fetch API.

As can be easily seen, this communication model has been used also for the other requests issued by the front-end, so it is thought unnecessary the explanation of further examples.

3.6.2 Communication to the Erlang Server

Communication between the client and the Erlang server, as already anticipated, takes place through the use of WebSocket, implemented respectively with Javascript and with the Cowboy library. Below it will be shown how WebSocket were implemented on JavaScript.

```
1  window.addEventListener("load", toggleConnection, false);
2  var websocket;
3  var content = document.getElementById("status");
4
5  var ENTER="!ENTER-->"+localStorage.getItem("id");
6  var EXIT="!EXIT";
7  var PING="!PING";
8
9  var server = "ws://alpha:8080";
10
11 var timerID = 0;
12 function keepAlive() {
13     var timeout = 20000;
14     if (websocket.readyState == websocket.OPEN) {
15         websocket.send(PING);
16     }
17     timerId = setTimeout(keepAlive, timeout);
18 }
19
20 function connect()
21 {
22     websocket = new WebSocket(server);
23     showScreen('<b>Connecting to chat</b>');
24     websocket.onopen = function(evt) { onOpen(evt) };
25     websocket.onclose = function(evt) { onClose(evt) };
26     websocket.onmessage = function(evt) { onMessage(evt) };
27     websocket.onerror = function(evt) { onError(evt) };
28 };
29
30 function toggleConnection(){
31     if (websocket && websocket.readyState == websocket.OPEN) {
32         disconnect();
33     } else {
34         connect();
35     };
36 };
37
38 function disconnect() {
39     websocket.close();
40 };
```

]

In the first part of the code, the connection to the server is handled. In addition to the initialization of some global variables, useful in order to set the server address or to set the main commands, on line 1 the function *toogleConnection* is called as soon as the user enters the chat page. In particular, this function has to check if the WebSocket is open, otherwise it calls the *connect* function which is responsible for creating it and, moreover, associating certain functions to different events that occur on the WebSockets. One thing to pay attention to is the *keepAlive* function, which has the task of sending a "PING" to the Erlang server every 20 seconds. Implementing this kind of behaviour was paramount, given the fact that, by default, WebSockets drop the connection after a given amount of time. Basically the *keepAlive* function forces the connection to stay up in order to allow the user to keep messaging with other users.

```
1  function cancelKeepAlive() {
2      if (timerId) {
3          clearTimeout(timerId);
4      }
5  }
6
7  function sendTxt(msg) {
8      if (websocket.readyState == websocket.OPEN) {
9          websocket.send(msg);
10         showScreen('sending: ' + msg);
11     } else {
12         showScreen('websocket is not connected');
13     };
14 };
15
16 function onOpen(evt) {
17     showScreen('<span style="color: green;">CONNECTED </span>');
18     sendTxt(ENTER);
19     keepAlive();
20 };
21
22 function onClose(evt) {
23     showScreen('<span style="color: red;">DISCONNECTED</span>');
24 };
25
26 function onMessage(evt) {
27     const words = evt.data.split('^'); //[0]->ora [1]->recieve
       [2]->Msg
28     if(words.length==3){
29         var dateMillis = parseInt(words[0].replace(/[\(\)']+/g,''))
       ;
30
31         addMessage(parseInt(words[1]),dateMillis,words[2],false);
32         showScreen('<span style="color: blue;">New message</span>')
       ;
33     } else{
34         showScreen('<span style="color: blue;">RESPONSE: ' + evt.
       data + '</span>');
35     }
36 };
```

The second part of the code highlights the functions that are called when an event is caught from WebSockets. Among the most important functions, the *onOpen* function sends a certain message, "ENTER", to the Erlang server as soon as the WebSocket is created. This function, in addition to asking to establish a connection, it also sends information about who is the user who wants to connect through a specific identifier. The *onMessage* function is invoked whenever a new message is received, and it takes care of retrieving the content of the message and to managing it based on the type of content extracted from it.

So, through the implementation and usage of these functions, and of the WebSocket technology, it was possible to establish a connection between client and server (Erlang)

3.6.3 Pouch DB

Regarding the front-end, Pounch DB was also used, this is a non-relational database "in-browser", it uses the browser, on which the code is executed, to locally store the saved data. The language used to query the database is JavaScript.

```
function getChat(idReceiver){
    db.createIndex({
        index: {fields: ['timestamp']}
    }).then(function () {
        db.find({
            selector: {
                timestamp:{$gt:null},
                idReceiver: idReceiver
            },
            fields: ['idReceiver','timestamp','msg','sender'],
            sort: [{'timestamp':'asc'}]
        }).then(function (result) {
            console.log(result);
            for(var i=0;i<result.docs.length;i++){
                var aux=result.docs[i];
                var messageStyle=createMessageSent(aux.msg, aux.
    sender, aux.timestamp);
                var chatBox = document.getElementById("chat");
                chatBox.appendChild(messageStyle);
            }

        }).catch(function (err) {
            console.log(err);
        });
    });

}
```

In out case, this database is used to to store the messages that the user sent and those he received, in order to be able to save the conversation that the user was holding. In fact, as soon as the user changes the chat, through a query to

the Pouch DB database, the messages relating to that chat are reloaded and shown in the appropriate box.