Computer Engineering

Distributed Systems and Middleware Technologies

HereThePaw

Project Documentation

Tommaso Giorgi
Matteo Guidotti
Clarissa Polidori

Academic Year: 2021/2022

Table of Contents

1 | Introduction

HereThePaw is a webapp designed to combat the problem of pet abandonment. It's aimed at those who can't afford to take their pets to an overpriced pet boarding house and animal lovers who want to make some money or just spend time with different animals. The application will allow users to log in with two different roles. We will call *pet sitters* those users who wish to offer the service of keeping animals for a period of time, and we will call *pet owners* those users who are looking for a pet sitter to whom they can entrust their pet. After logging in, a pet owner has the opportunity to make a reservation for a certain period for his pet to a pet sitter of his choice. The booking must be accepted by the pet sitter to be considered effective. Both users can access the list of their bookings, the pet sitter can of course accept or decline the booking for him. In order to decide which pet sitter to book, a pet owner can read the reviews left by other users and see the average rating. If he has already booked a service with that pet sitter, he can also leave a review.
HereThePaw also provides a chat room where pet owners and pet sitters can exchange information about the exchange of the pet or subsequently inform each other about the progress of the stay.

2 | Architecture

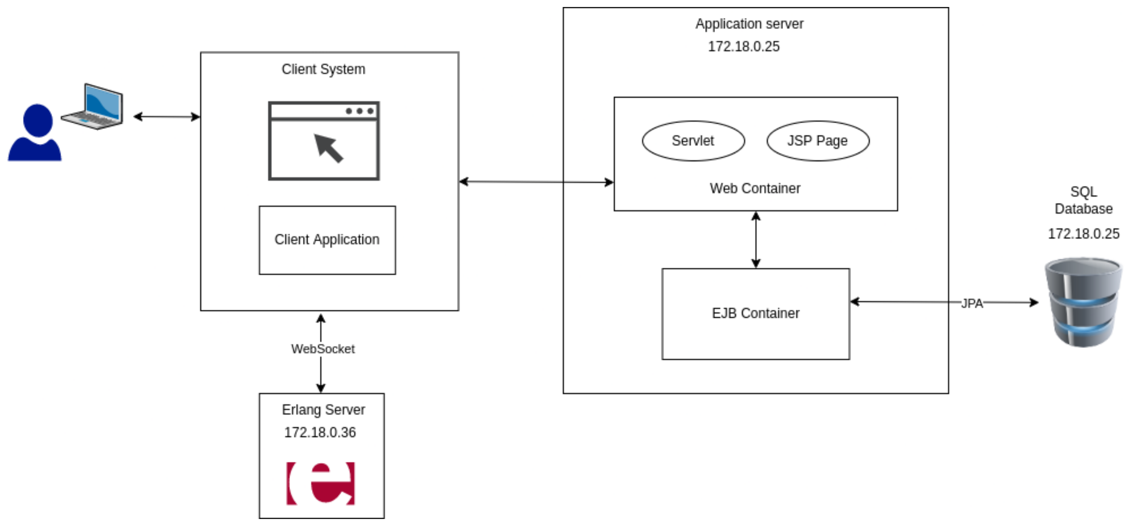


Figure 2.1: Architecture structure

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. To write dynamic, data-driven pages for our Java web application, we used *JavaServer Pages (JSP)*, which is a standard Java technology, together with *Java servlets*.
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 two users and, moreover, of keeping track of the users that 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 communication over a TCP connection; in particular, WebSockets are not http connections even if they use http to initiate the connection. About the connection, this was managed via javascript *WebSocket* at the client side; instead, regarding the server side, the connection was managed with the use of websocket 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 6.2.1*), the web application uses the REST services to allow communication and recovery of the various resources / services offered. 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.
By means of the UNIPI VPN, we have been able to deploy the Erlang server at the address 172.18.0.36, listening at the port 3307, while the Glassfish server, together with the MySQL server, is in execution at the address 172.18.0.25, attending for requests at the port 8080. The various components introduced in this chapter and used in the architecture will be explained in more detail below.

3 | Implementation

3.1 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 the application described here only stateless EJBs were used.

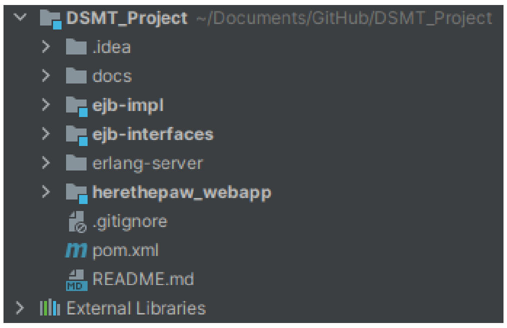
3.2 EJBs

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 and concurrency problems, since each client, when performing a method call, will obtain a different instance of the EJB, thanks to the EJB container.

3.3 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/dsmt_project. 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.

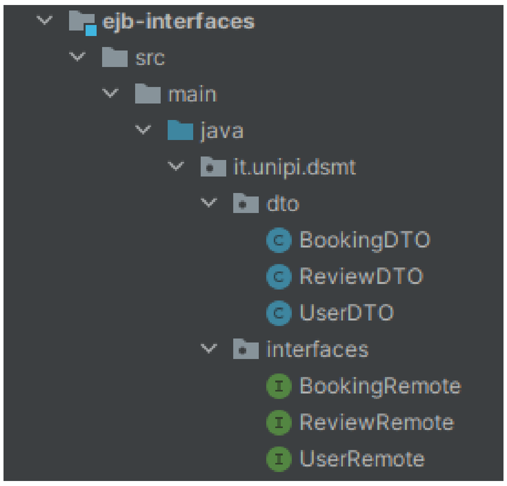
3.4 Project Structure



```
DSMT_Project ~/Documents/GitHub/DSMT_Project
  .idea
  docs
  ejb-impl
  ejb-interfaces
  erlang-server
  herethepaw_webapp
  .gitignore
  pom.xml
  README.md
External Libraries
```

Figure 3.1: Project structure

3.5 EJB modules



```
ejb-interfaces
  src
    main
      java
        it.unipi.dsmt
          dto
            BookingDTO
            ReviewDTO
            UserDTO
          interfaces
            BookingRemote
            ReviewRemote
            UserRemote
```

Figure 3.2: Module ejb_interfaces structure

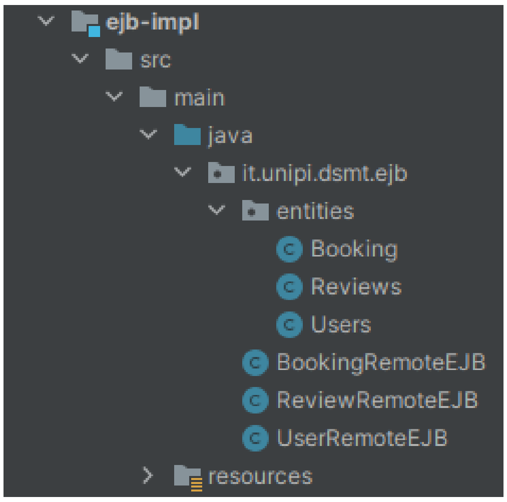


Figure 3.3: Module ejb_impl structure

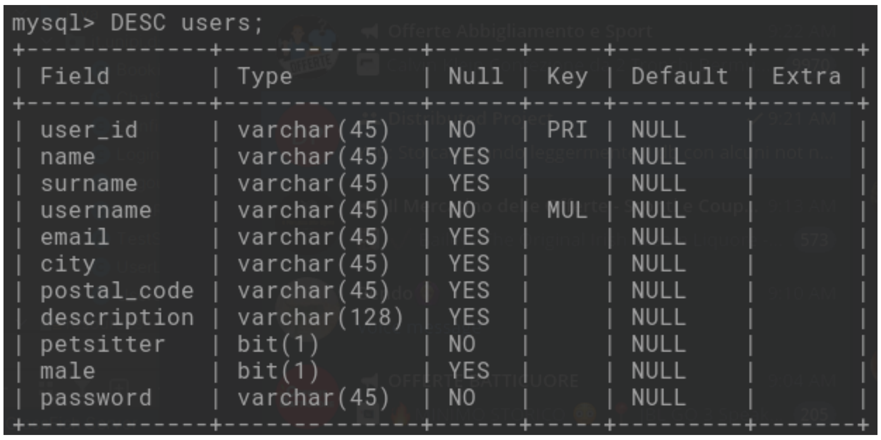
As far as EJB management is concerned, we have two modules, ejb_interfaces which, as its name suggests, contains the interfaces that provide the schema for the methods implemented by the classes in the ejb_impl module. In addition, this module contains the Data Type Objects (DTO) corresponding to the entities contained in the ejb_impl module.

The ejb_impl module implements the interfaces and contains a package named *entities* containing the entities used to transform the tables of our database into objects on which we can execute queries using JPA; this part will be described in details in the following paragraphs. It also contains the implementation of the beans that execute the database queries.

3.5.1 Database and JPA

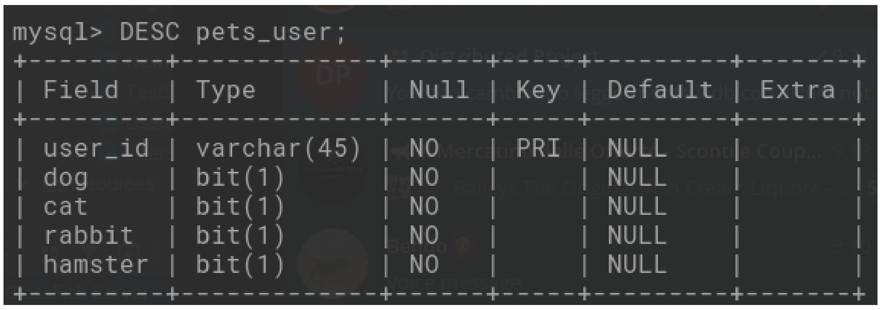
We used a relational SQL database with **four tables**. The tables are the following:

- *"users"* table: this table is used to store the information of a user registered to the system.
- *"pets_user"* table: this table is used to map a user_id to the pets that he/she owns .
- *"booking"* table: this table is used to store all the booking requests from every pet owner to every pet sitter, includes pending and confirmed requests.
- *"review"* table: this table is used to store reviews composed of a text and a 1-5 number rating written to pet sitters.



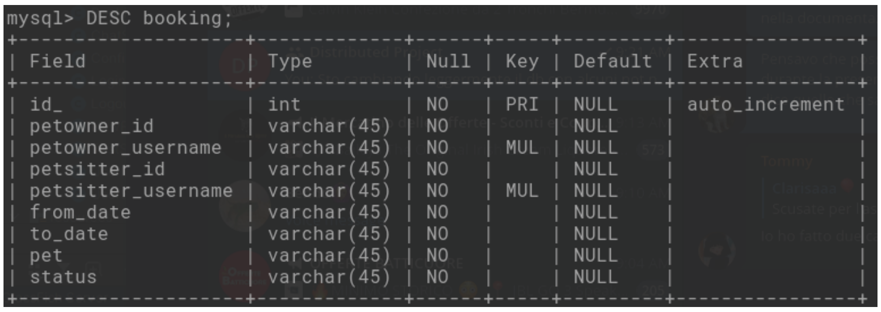
```
mysql> DESC users;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| user_id     | varchar(45)  | NO   | PRI | NULL    |       |
| name        | varchar(45)  | YES  |     | NULL    |       |
| surname     | varchar(45)  | YES  |     | NULL    |       |
| username    | varchar(45)  | NO   | MUL | NULL    |       |
| email       | varchar(45)  | YES  |     | NULL    |       |
| city        | varchar(45)  | YES  |     | NULL    |       |
| postal_code | varchar(45)  | YES  |     | NULL    |       |
| description | varchar(128) | YES  |     | NULL    |       |
| petsitter   | bit(1)       | NO   |     | NULL    |       |
| male        | bit(1)       | YES  |     | NULL    |       |
| password    | varchar(45)  | NO   |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
```

Figure 3.4: *"users"* table



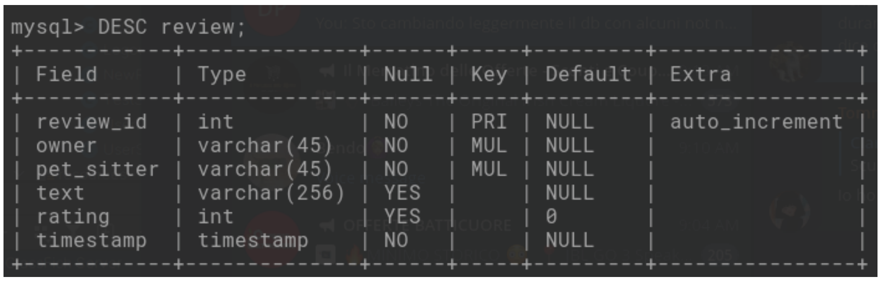
```
mysql> DESC pets_user;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| user_id | varchar(45) | NO   | PRI | NULL    |       |
| dog     | bit(1)      | NO   |     | NULL    |       |
| cat     | bit(1)      | NO   |     | NULL    |       |
| rabbit  | bit(1)      | NO   |     | NULL    |       |
| hamster | bit(1)      | NO   |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

Figure 3.5: *"pets_user"* table



```
mysql> DESC booking;
+--------------------+-------------+------+-----+---------+----------------+
| Field              | Type        | Null | Key | Default | Extra          |
+--------------------+-------------+------+-----+---------+----------------+
| id_                | int         | NO   | PRI | NULL    | auto_increment |
| petowner_id        | varchar(45) | NO   |     | NULL    |                |
| petowner_username  | varchar(45) | NO   | MUL | NULL    |                |
| petsitter_id       | varchar(45) | NO   |     | NULL    |                |
| petsitter_username | varchar(45) | NO   | MUL | NULL    |                |
| from_date          | varchar(45) | NO   |     | NULL    |                |
| to_date            | varchar(45) | NO   |     | NULL    |                |
| pet                | varchar(45) | NO   |     | NULL    |                |
| status             | varchar(45) | NO   |     | NULL    |                |
+--------------------+-------------+------+-----+---------+----------------+
```

Figure 3.6: *"booking"* table



```
mysql> DESC review;
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| review_id  | int          | NO   | PRI | NULL    | auto_increment |
| owner      | varchar(45)  | NO   | MUL | NULL    |                |
| pet_sitter | varchar(45)  | NO   | MUL | NULL    |                |
| text       | varchar(256) | YES  |     | NULL    |                |
| rating     | int          | YES  |     | 0       |                |
| timestamp  | timestamp    | NO   |     | NULL    |                |
+------------+--------------+------+-----+---------+----------------+
```

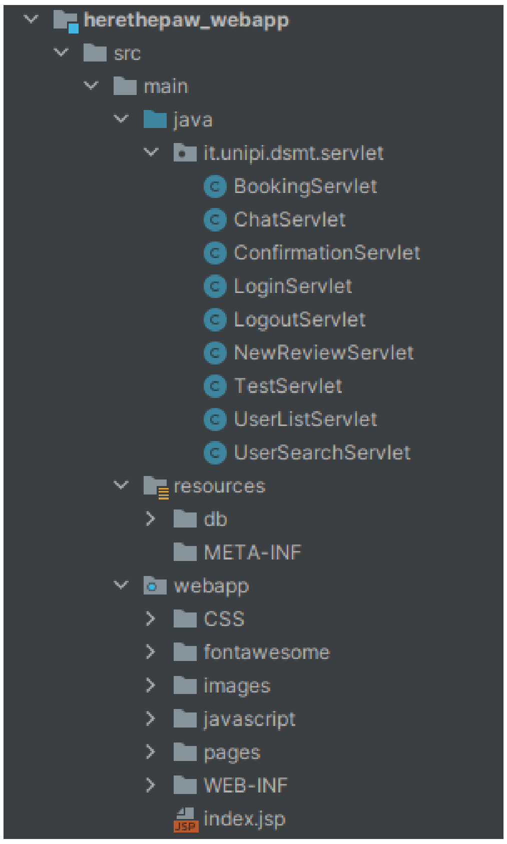
Figure 3.7: *"review"* table

After we developed all the queries, we decide to add some **indexes** to speed up the most intesive queries in our web application. The indexes that we add are the following:

- Column "username" of users table: we added this index to speed up the login query and every query that involves the search of an user by its username
- Column "petsitter_username" of booking table: we added this index to speed up the query to display all the booking request of a target pet sitter that are typically done using the username.
- Column "petowner_username" of booking table: same as before but useful if we are trying to display them for a pet owner user.

For most of the queries we decided to use **Java Persistence API (JPA)** to create an easy link between SQL tables and Java Objects. For this purpose, we added an entity class for each table. Then the entity objects are converted to the corresponding DTO classes for safety reasons.

3.6 Web app Module



```
herethepaw_webapp
  src
    main
      java
        it.unipi.dsmt.servlet
          BookingServlet
          ChatServlet
          ConfirmationServlet
          LoginServlet
          LogoutServlet
          NewReviewServlet
          TestServlet
          UserListServlet
          UserSearchServlet
      resources
        db
        META-INF
      webapp
        css
        fontawesome
        images
        javascript
        pages
        WEB-INF
        index.jsp
```

Figure 3.8: Module herethepaw_webapp structure

This module contains the web app, i.e. the set of servlets, objects written in Java language that operate within a web server, these are used to handle different client requests in a concurrent and distributed way. They allow the generation of dynamic web pages depending on the request parameters sent by the user's client browser to the server. In our application, we do not program servlets directly, but prefer to use JavaServer Pages (JSP) that implement the servlet specification, which are then translated (compiled) into servlets at runtime. Requests coming from the client may be POST or GET, these are handled differently depending on the corresponding servlet.

3.6.1 Servlet

Research Servlet

A servlet has been used to handle the pet sitter search in the main page of the web application. In this search the client can insert two parameters:

- Pet: client selects from a menù the pet that he/she wants to assign to a pet sitter.
- City: client inserts a city in order to find pet sitters who live near him. This parameter can also be omitted and if so, a search is done including all the cities.

Once these parameters has been inserted, a GET request is sent to the search servlet. An example of request is the following:

```
/search_user?city=Pisa&pet=dog
```

The servlet will retrieve these two parameters and will forward them to the usersearch.jsp that will use these parameters to perform a SQL query on the *Users* Table using a UserRemoteEJB instance.

Userpage Servlet

In our application the "Username" of a user is unique and so it is the only information needed by this servlet to display the corresponding user page. The username is passed to the servlet using a GET request, an example of this GET request is the following:

```
/UserListServlet?username=Tomawk
```

This request is then redirected to the userpage.jsp that will use an UserRemoteEJB instance to find the user with that username in the database in order to create a dynamic page with all the information of that user retrieved from the database too. If that user is a pet sitter, additional elements are displayed to allow the booking with him.

Login/Logout Servlet

Two different servlets are used for handling the Login and the Logout of an user in our application. Let's start with the Login. Once an username and a password are inserted into the corresponding field and a submit button is clicked, a POST request is sent to the login servlet. The login servlet will retrieve both these parameters and using the *loginUser* method of a UserRemoteEJB instance will search in the Users Table of our database the corresponding <username,password> pair. If such pair is found, a user object instance (UserDTO) is fulfilled with all the information of that user (retrieved from the database) and a HttpSession attribute (*"logged_user"*) is added to the HttpSession current session and assigned to that UserDTO instance. So if there are no logged users, this session attribute is assigned to *null*, otherwise to a UserDTO instance of the user found in the database. So, the logout becomes very easy and does not require parameters. Once the logout button is clicked, the session is invalidated and so all the information regarding the previous logged user are discarded.

Booking Servlets

To handle the booking functionality of the web application we used two different servlet, the first one is called *BookingServlet* and it is used to handle the pending booking requests from the pet owners instead the other servlet named *ConfirmationServlet* is used to handle the acceptance or the refusal of a pending booking request from a pet sitter. Both receive POST requests from the client with all the information needed to insert a pending booking request and to confirm it. The information needed to insert a new pending booking request are the following:

- Pet: the pet that we want to ask a booking for.
- Pet sitter username and id: these are implicit and don't need to be inserted manually by the pet owner, they are retrieved from the user page in which we are filling in the other fields for the booking.
- Pet owner username and id: also these information are implicit and are got from the logged user instance.
- Date from: We are booking a pet sitter from this date to another.
- Date to: the date that will end our booking with the pet sitter.

These information are got from the POST request by BookingServlet and with a BookingEJB instance (including different booking methods) are inserted in the *booking* table on our SQL database, the status column is set to "pending" in this case.
When a pending request is inserted, then it can be accepted or refused by the target pet sitter. This can be done simply changing the status column in the database from "pending" to "accepted" or "declined".

When the pet sitter clicks on the "Accept" or "Decline" button of a target pending booking request, a POST request is sent to ConfirmationServlet containing the booking_id of that booking request that is important to identify the right entry in the *booking* table to update the status column.

Review Servlets

To manage the insertion of a new review by a user we use the servlet named NewReviewServlet. It receives POST requests from the client with all the information needed to insert a new review. The necessary information is as follows:

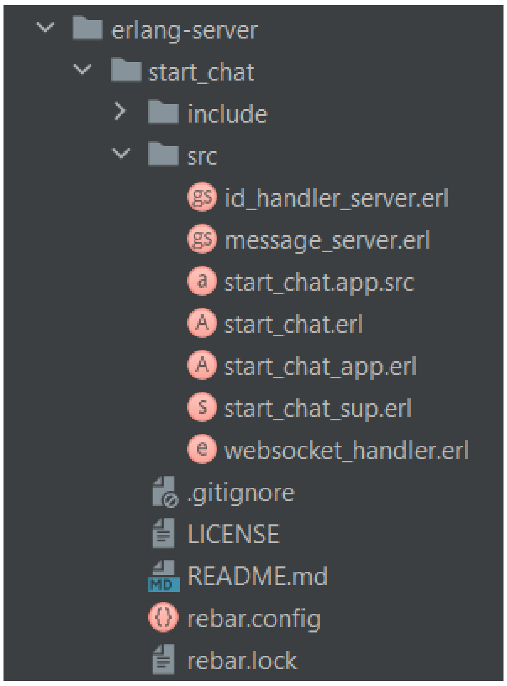
- review: possible review text
- pet_owner: reviewer
- pet_sitter: user reviewed
- rating review: rating

With an instance of ReviewEJB through the insertReview method they are inserted in the table *review* of our database.

3.6.2 Additional Information

- We did not hash passwords and so they are stored in clear.
- We did not provide a registration form, the users inserted in the application were added manually.
- Pet owner can not become pet sitters, so a pet owner can only request other pet sitter for a booking but can not accept requests from other users.
- Most of the services provided by the application require a login. If the user is not logged, he can only search pet sitters using the search and check their information.
- Reviews can be inserted only if the pet owner has booked the corresponding pet sitter

3.7 Erlang server



```
erlang-server
  start_chat
    include
    src
      id_handler_server.erl
      message_server.erl
      start_chat.app.src
      start_chat.erl
      start_chat_app.erl
      start_chat_sup.erl
      websocket_handler.erl
    .gitignore
    LICENSE
    README.md
    rebar.config
    rebar.lock
```

Figure 3.9: Erlang server structure

The chat service is implemented by using an Erlang server, that will have the role to forward messages sent from a user that is using the client application, to another one. In order to be able to do that, both the users have to be online (a user is considered online and ready to chat with others, when it is entered in the chat page). At the moment in which a logged user is entering in the chat page of the application, a websocket connection between the client (implemented in Javascript) and the server is established.

3.7.1 WebSocket Handler

The Erlang server is composed by three main modules, namely id_handler_server, message_server and websocket_handler. The websocket_handler is the module in which all the websocket communications take place. The module can receive 4 different types of messages from the client, that will be handled by the websocket_handle method:

- "&LOGIN:<username>": The user has just entered the chat section and wants to be registered at the server side. When the module receives this message, it calls a cast provided by the other modules, sending the PID of the process that is handling the connection (simply obtained by calling self()) together with the username of the user that has entered the chat. In that way, each time in which that username has to receive a message, the server will forward it by using the process identified by that PID
- "&LOGOUT": The user leaved the chat page, so its username and related PID has to be removed by the online users at served side. The websocket_handler calls the relative cast provided by the other modules of the application
- "&PING": When the websocket handler receives this message, it means that the client side wants to know the list of online users. It is provided by calling a cast provided by the other modules
- "<messageText>:<senderUserName>:<receiverUserName>": When the first part of the arrived string is not one of the three options showed above, the handler knows that it is a message to be forwarded to another user. The handler will decompose the string and send the retrieved values to the message_server module, by using a cast

The module can also receive Erlang messages from the other modules of our application. These messages will be handled by the websocket_info method, that will simply forward the message to the client connected through the websocket.

3.7.2 Id Handler

The module id_handler_server has the responsibility of storing and handling the online users. The users are stored by means of an ets structure, that is an Erlang built-in term storage, with which a process can create a table that is able to store tuples. In our case the tuples will be formatted like {UserName, Pid}. The id_handler_server is a gen_server module that provides a call function to retrieve the Pid of the process that handles the connection with a specified user (specified with its username), a call to insert a new username with the related Pid (if this username is not online yet), a call to remove a username that is logging out and a call handler that is able to retrieve a string that contains the usernames of all the online users, separated by means of '|'.

3.7.3 Message Handler

The module message_server is a gen_server module that consent to our Erlang application to listen for new messages, format and forward them to the relative receivers. The init(_) function is responsbile for starting to listen requests arriving to a given port (in our case, it is 3307), and it also indicates what is the module that has to dispatch the requests, that in our case is the websocket_handler module. Then, the module provides different cast handlers in order to fulfill the needs of the dispatcher module. The first one is the login handler, that simply calls the id_handler_module for inserting a new user; it will be returned an atom that could be *ok* or *nickname_in_use* depending on if the username sent is already registered or not. If it is not, a welcome message is sent to the user, otherwise the error will be signalized. In the same way, message_server is able to handle the logout of an user by asking to the id_handler_server to remove the user that is logging out. Another functionality of the module is to call the id_handler_server for retrieving the online users list and to send it to the client that has requested it. Finally, message_server consent to the application to forward messages sent by a client to another client. In order to do that, at first it asks to the id_handler_server to retrieve the PID of the process that handles the communication with the specified username, than it sends to that PID the string message formatted like "<UserName_Sender>:<MessageText>". This message will be received by a process that is running the websocket_handler module and, in particular, it will be handled by the websocket_info, that is the function call responsible for handling the receiving of Erlang messages. This handler will simply return a tuple like {reply, {text, MessageText}, State}, that consent to send the MessageText to the client connected through the websocket.

3.7.4 Compiling and Executing the Application

In order to make available to our erlang project the required dependencies we used Rebar3, that is an Erlang tool that makes it easy to create, develop, and release Erlang libraries, applications, and systems in a repeatable manner. By using it, we are able to denote the required dependencies in the rebar.config file (in our case, it was the cowboy library) and compile the application, by using the *rebar3 compile* command. Now, we can run a shell by means of the *rebar3 shell* command; this shell will be ran in an environment in which all the dependecy modules are executed, so that we can start to execute our application. The server is started by calling the function *start* of the module start_chat. This module has an application behaviour that provides the start and stop methods, useful to start and stop all the application components.