UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Distributed Systems and Middleware Technologies

TherAPPist

Workgroup project documentation

Pietro Tempesti, Benedetta Tessa

Content

INTRODUCTION

TherAPPist is a chat web-application whose purpose is to help people with their mental health by providing them a free therapist they can chat it completely anonymously.

First, the user has to register by specifying an issue they need help with and choose from a list of therapists specialized in that issue.

Similarly, a therapist can register by specifying the fields they are specialized in and the maximum number of patients they want to work with.

Once they register, they have to wait for the admin approval since they need to check the goodness of the new therapist.

Requirements

Functional requirements

An *Unregistered User* can:

- Register either as a Therapist or as a Patient

A *Registered (not logged) User* can:

- Login using their credentials

A *Logged User* can:

- Send a new message the contact.
- Receive a new message.
- Update their information.

A *(logged) patient* can:

- If they haven't already chosen a therapist, they can choose one among a list of suitable therapists according to their specialization.
- Terminate the therapy.

A *(logged) therapist* can:

- Wait for their approval from the administration
- Choose the patient to chat with.

An *Admin* can:

- Decide whether or not admit a therapist to the website
- Upgrade a therapist to Admin

Size and scope of the application

Our application is intended to be available for general purpose, as a web service: Each person who wants a light psychological support in the form of a chat can sign in. Also, therapists who want to help people as volunteers can apply for the therapist role of the application: their applications will be evaluated by the administrator.

The selected technologies ensure the following requirements:

- No message can ever be lost, regardless the fact that the receiving user is online
- The application is totally portable and reachable from any web browser
- The GUI provides a user-friendly experience and makes application easy to use

Architecture

This application is client-server and follows a model-view-controller pattern.

It was, as a matter of fact, implemented with the help of Spring Boot MVC. We implemented the controller using the Spring Controllers, the module making use of the Spring services and the view exploiting JSP and JSTL. Also, we decided to implement webSockets in order to handle the real-time chat system.

But now let's talk about the client and the server.

Client

From an architectural point of view, the client is only in charge of providing the user a GUI and the communication with the server. The client does store information about users and we make use of sessions in order to make http stateful since it's stateless by default.

Messages are exchanged thanks to Web Sockets. The reason why we chose them is because they run over TCP and provide a low-latency low-level communication by reducing the overhead of each message. They also are thread safe which means the server can send data to the client at any time without synchronization issues.
This could be easily done by configuring the client to receive Web Sockets and with the help of JavaScript, which provides methods to open the sockets, send and show messages on receive.

Server

Role of the server

The server is in charge of:

- Register user data at registration time, remembering username, password and other personal information.
- Login users by checking username and password
- Forward any message to the correct client.
- Register every in-transit message in order to permit the restoring of the chat list for every client.
- Queuing correctly the messages that are destined to the same client.

Implementation

As specified before, our server is implemented by using Spring Boot MVC and an Erlang module adopting the gen-server behaviour which handle the connection with one of our persistent databases.

Synchronization management

By default, both Spring Controllers and Services aren't thread-safe since they are singleton, but this is not a critical issue in our case: in fact, exploiting the HttpSession, we don't have any shared object in our controller layer. In the module layer we implemented ConcurrentHashMaps in order to handle the shared collections of objects. When looping over such ConcurrentHashMaps, we made use of the iterators of such collections since they provide weakly consistency.

The Objects that we put into ConcurrentHashMaps are the WebSocketSessions, because we have to send messages from a session to another, and the ErlangConnections, because we have to store the connections of each logged user, and retrieve them when an user sends a message in order to store it into Mnesia.

Also the connections to the erlang server, which is in charge to storing the chat history, are thread safe, because we use an ExecutorService dedicated to each client.

Persistent data storing

We exploited Spring JPA and JPA-based repositories thanks to which we could define custom *find* queries of which Spring will provide the implementation automatically.
We also exploited Mnesia, a relational database completely handled in Erlang.

Spring JPA

We use Spring JPA to connect our application with a MySQL database, in which we store the information related to therapists and patients.

We choose this database to handle without struggles the complex queries needed by our system.

We create two tables, one for the patients and one for the therapists:

Patient

- Username (varchar55, primary key)
- Email (varchar55)
- Password (varchar55)
- FullName (varchar55)
- dateOfBirth (dateTime)
- issue (varchar55)
- therapist (varchar55)

Therapist

- Username (varchar55, primary key)
- Email (varchar55)
- Password (varchar55)
- FullName (varchar55)
- dateOfBirth (dateTime)
- specialization1 (varchar55)
- specialization2 (varchar55)
- specialization3 (varchar55)
- biography (varchar255)
- state (varchar55)
- maxPatients (int)
- acceptedPatients (int)
- gender (varchar55)

Mnesia

For storing the information related to the messages we decided to use Mnesia, which is designed with requirements of speed in simple operations, needed in our application to ensure low latency in retrieving the chat history. Also, we choose to handle the message storing with erlang in order to better handle the concurrent message storing, thanks to the Mnesia concurrent access handler.

We store the messages in a single table, called messages, with the following structure:

```
-record(messages, {timestamp,
                   sender,
                   receiver,
                   text}).
```

In order to ensure the persistence of the data, the tables are saved using the *disc_copies* options, thus we maintain the records stored in memory and on the disc. Every time the server starts Mnesia it waits if the tables are present: if they are then we simply load them, otherwise we create the tables again.

Rebar3

We decided to use rebar3, a build tool and package manager for erlang applications, to build and launch properly the erlang side of the server.

To compile and run the project we use the command

```
$ rebar3 shell --sname therappist_server@localhost --setcookie therappist --script
src/start.escript
```