# Task 1 Report

Federico Fregosi, Mirko Laruina,
Riccardo Mancini, Gianmarco Petrelli

18/12/2019

## Contents

# 1 Application Specifications

## 1.1 Application overview

The application is a messaging system where registered users can create an account, exchange text messages and make groups.

A registered user can initiate a chat with another user, create a new group chat (of which he becomes the admin) and send messages to the chats he belongs to, as well as receiving messages from those chats. He can also leave a group.

A group admin can add and remove new users to the group. He cannot assign his powers to another user in the group and if he leaves the group, the latter is deleted.

Everytime a user views a chat, all the latest messages from the chat are fetched from the server and shown to the user.

## 1.2 Actors

Anonymous user, registered user, group admin and a time-based event.

## 1.3 Requirement Analysis

### 1.3.1 Functional requirements

An **anonymous user** must be able to register in order to become a *registered user*. Registration is done by providing a username and a password. Once registered, the user can login using the provided credentials. Every user must have a different username.

Log-out of a user is performed through a dedicated button. If the user closes the application without logging-out, he will not need to perform another login when reopening the application within 5 days.

A **registered user** must be able to:

- Retrieve the list of chats she's a member of
- Read the messages of a chat
- Send a message to a chat
- Create a new private chat
- Create a new group chat
- Leave a group chat

In order to create a private or group chat, the user must type in the username of the other participants. When an user create a new group chat, he automatically becomes the admin of the chat in question.

A **group admin** is the only chat member who is able to:

- Add members to the group chat
- Remove users from the group chat
- Delete the group chat

Adding an user to the group requires once again to specify the username of the correspondent.

The administrator of a chat cannot leave the chat (he can only delete it).

The **time-based event** updates the user interface on regular intervals to show chat updates, i.e. new chats or new messages from the current chat, if any.

#### 1.3.2 Non-Functional requirements

- **Data concurrency:** the application must be able to handle concurrent requests to the database remaining consistent between every operation even with high volume of data.
- **Real-time updates:** A (soft) real-time updating component that pulls updates from the server is also required for this type of application in order to to ensure a comfortable and efficient use.
- **Responsiveness and cross-platform:** Client-side application must provide a responsive view both for pc, laptops and mobile devices.

# 2 Design

## 2.1 Use-case diagram

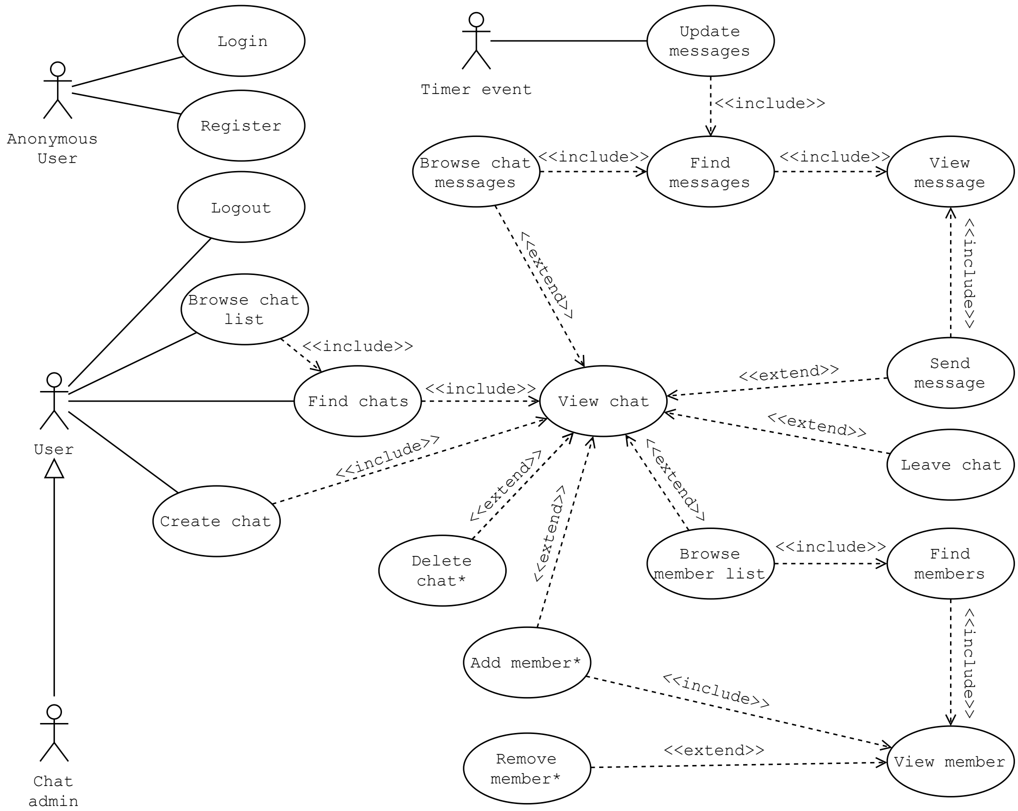


Figure 1: Use-case diagram. Use-cases indicated with a star (*) require the user to be the chat administrator.

## 2.2 Class diagram


Figure 2: Class diagram for the identified entities

It can be seen that we chose to keep it as simple as possible by not making any distinction between *private chats* and *group chats*, creating a single *Chat* entity.

## 2.3 Software Architecture

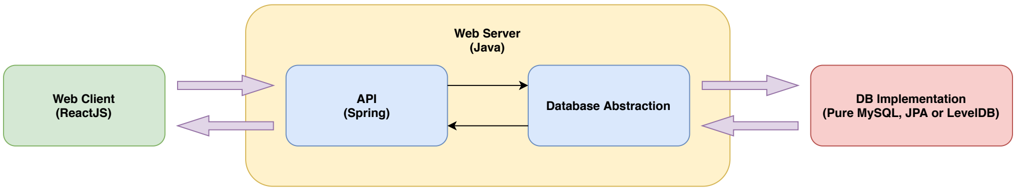


Figure 3: EasyChat software architecture

As can be seen, a Java server back-end is the core of the proposed architecture, providing both the APIs, through the open source Spring Framework, and the database interface, which has to allow a high flexibility in terms of database implementation.

Client-side, a ReactJS front-end has been chosen for the implementation of the web app the user will interact with.

# 3 Implementation

## 3.1 Java backend

The Java backend provides simple REST APIs for managing the chat application. In order to do so, it uses the *Spring* framework[1]. The list of the APIs and their description can be found in the `docs/api.md` file. The APIs return a JSON document, generated using *Google GSON*[2], which is interpreted and shown to the user by the ReactJS frontend. These APIs are implemented using a simple database abstraction layer which provides corresponding APIs to the database. This is implemented in Java using a generic *DatabaseAdapter* interface that can be implemented to provide access to different databases (as we have done in task 0 with plain SQL, and in this task with JPA and levelDB implementations). The database backend can be set from a configuration file, where some other db-specific settings can be set.

You can find a synthetic UML class diagram in the attachments.

### 3.1.1 JPA

Using JPA and Hibernate, the following classes were made:

- *Chat*
- *Message*
- *User*
- *UserSession*

Furthermore, relationships between the classes were set as follows:

- *Chat* has a one to many relationship with *Message*;
- *Chat* has a many to many relationship with *User*;
- *User* has a one to many relationship with *Message*;
- *User* has a one to many relationship with *UserSession*;

A more detailed explanation of the JPA implementation can be found in the tutorial.

The resulting design can be summarized with the ER diagram of figure 4. Every *User* is identified a *userId* and has got a unique *username* and a *password*. The user can be a member of many *Chats* and can be the administrator of many *Chats*. On the other hand, a *Chat* can be administered by only one *User*. A *User* can send a *Message* to a *Chat*. Each *User* and each *Chat* can have many *Messages* while a *Message* can belong to one *Chat* and one *User* only. The *Session* represents a logged user session. Each *User* can have many open *Sessions*.

Once the classes and the design were set, the Java implementation of the methods using JPA was very easy and fast. A new implementation of the *DatabaseAdapter* class has been done, along with JPA implementations of the data model interfaces.

Once the database had been created, we filled it with random test data using the free service available at `http://filldb.info/`. Generated data is not perfect, since some more complex functional constraints could not be included in the generation. However, that was sufficient for the first functional tests of the application.

[1] More information at `https://spring.io/`

[2] More information at `https://github.com/google/gson`

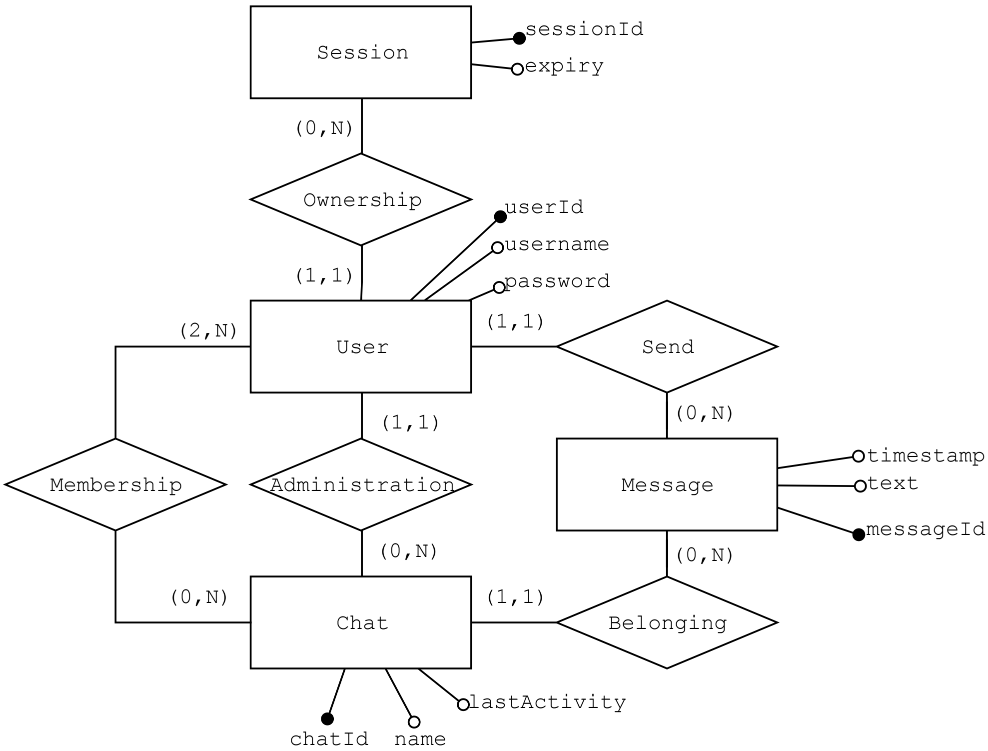


Figure 4: ER diagram for the database.

## 3.2 ReactJS frontend

The frontend has been developed using the *ReactJS* framework[3]. An overview of the main functions is available in the user guide (`docs/user_guide/user_guide.pdf`).

Regarding the implementation, the most notable thing is the timer-based event that updates the UI. In particular, the client makes a request to the server every second for updating the list of chats and every half second for updating the list of messages in the current chat.

## 3.3 Limitations

**Passwords** For the sake of simplicity, password hashing has not been implemented into the application. However, this could be simply integrated with a future update.

**Polling** In the current implementation, the server is polled every second for updating the list of chats and every half second for updating the list of messages. This is indeed a great load on the server in case there are many clients connected at the same moment. This problem has been alleviated by requesting only a subset of the chat messages, based on the timestamp on the latest received message. However, a more appropriate way to handle it would be having a kind of notification API that can be "long polled"[4] by the client.

[3] More information at `https://reactjs.org/`

[4] A "long poll" is when the client makes a request to the server and the server does not respond until new information is available, at the cost of timing out the connection, at which point a new request is made.

# 4 Testing and Evaluation

The application has been tested using unit tests and a final integration test.

## 4.1 Unit tests

The database APIs have been tested using unit tests through *JUnit4* [5] before being integrated with the API backend code. In particular, every method in *DatabaseAdapter* is tested against sample data, verifying its result.

These automated unit tests made it possible to identify bugs at their root cause, saving a lot of time in debugging.

## 4.2 Integration test

In the end, once everything had been put together, we performed a final test in order to verify the correctness of all use cases.

[5] More information at `https://junit.org/junit4/`