Task 1 Report

Federico Fregosi, Mirko Laruina,
Riccardo Mancini, Gianmarco Petrelli

11/11/2019

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*. Login is carried out using username and password selected by the user when registering. Username and password are also the only two information necessary for the registration and they can be free selected by the user.

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

- Send a message to a chat
- Read chat messages
- Create a private chat
- Create a group chat

A list of his chats is permanently shown to the user. Both creating a private chat or a group chat require the username of the correspondents. When an user create a group chat, he automatically become admin of the chat in question.

A **group admin** is the only one who must be able to:

- Add users to the group
- Remove users from the group

Adding an user to the group requires once again to specify the username of the correspondent. However, an admin can't leave the group without deleting the group itself.

Lastly, an user can close the session just by logging out pushing the specific button.

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

1.3.2 Non-Functional requirements

TODO

2 Design

2.1 Use-case diagram

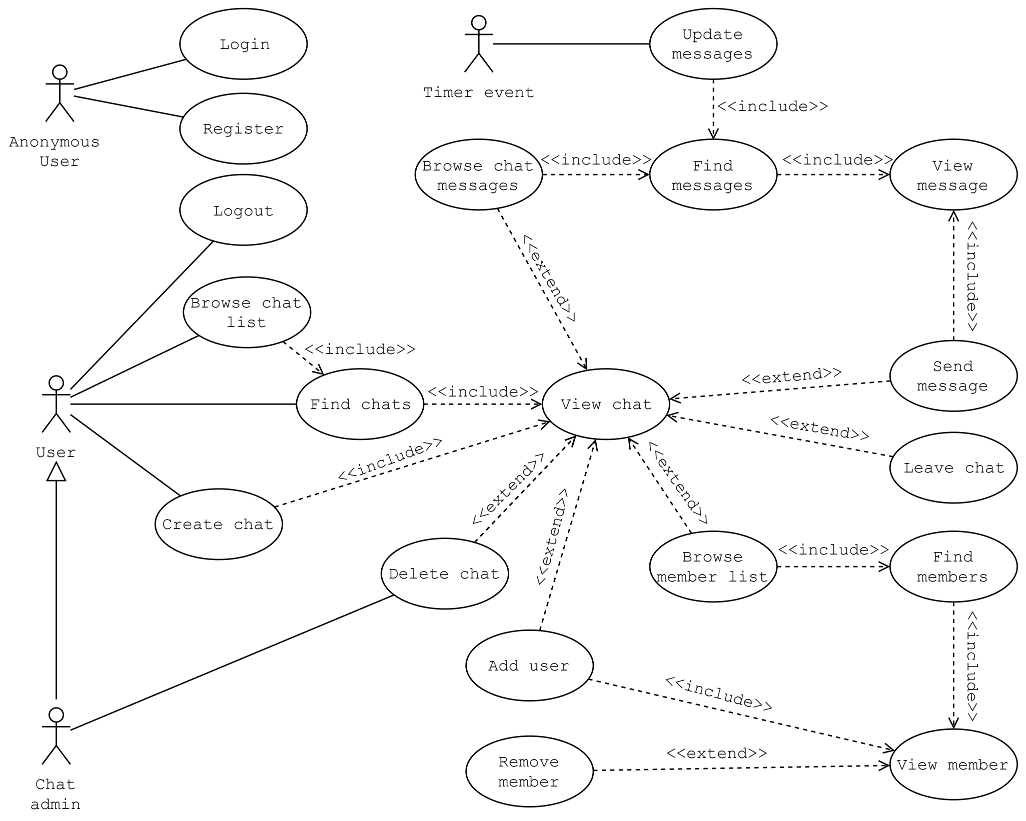

Figure 1: Use-case diagram

Figure 1 shows the use-case diagram for the project. Please note that the `Add user` and `Remove member` operations are only allowed to the `Chat admin`.

2.2 Class diagram

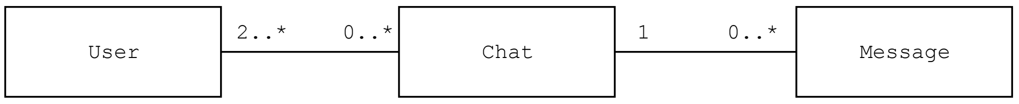

Figure 2: Class diagram for the identified entities.

Figure 2 shows the class diagram derived from the specifications. 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

TODO

The proposed software architecture is a classic three-layer architecture:

1. database (MySQL)
2. server back-end (Java + Spring)
3. user web app (ReactJS)

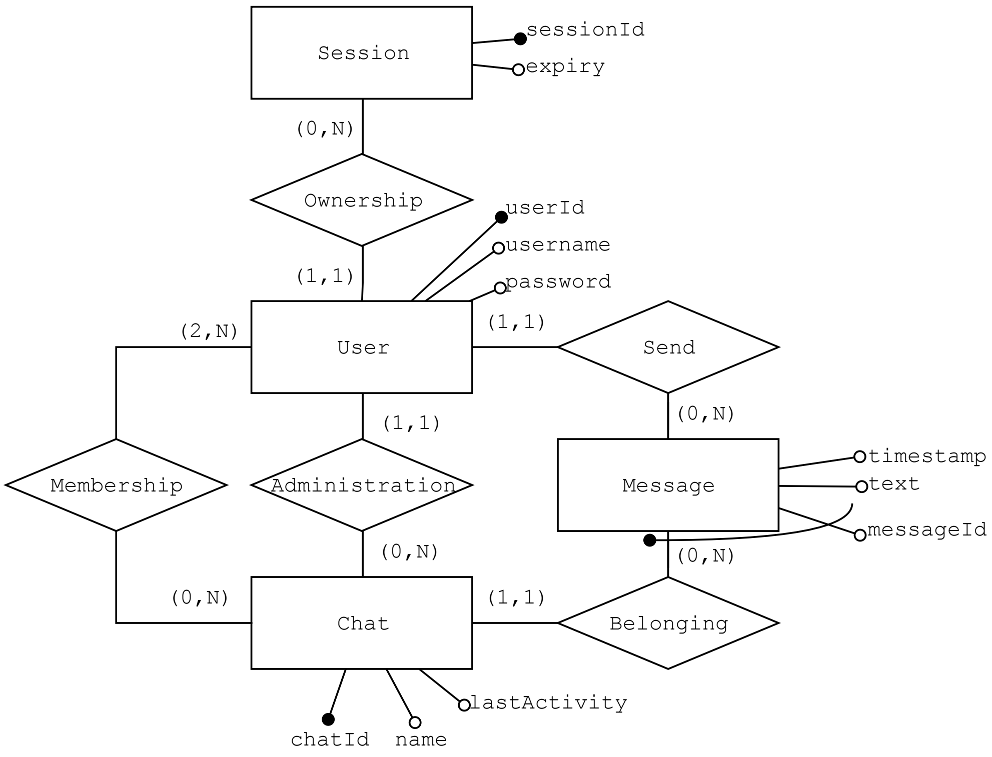

Figure 3: ER diagram for the database.

3 Implementation

3.1 Database

3.1.1 Relational DB Design

Figure 3 shows the ER diagram of the MySQL database. 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 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.

3.1.2 JPA

Given the database, the Java implementation using JPA was very fast, especially compared to using plain SQL. We made use of both one-to-many and many-to-many relationships as it can be seen from the ER diagram.

TODO

3.2 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 this Task with plain SQL, JPA and levelDB implementations). The database backend can be set from a configuration file, where some other db-specific settings can be set.

3.3 ReactJS frontend

The fronted 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.

3.4 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.

4 Testing and Evaluation

The database APIs have been tested using some test units through *JUnit4* [5] before being integrated with the ReactJS frontend.

TODO

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

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

[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.

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