Task 1 Report

Federico Fregosi, Mirko Laruina,
Riccardo Mancini, Gianmarco Petrelli

11/11/2019

Contents

1 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 Functional specifications

An **anonymous user** must be able to register in order to become a *registered 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 **group admin** must be able to:

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

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

1.4 Software Architecture

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

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

2 Analysis

2.1 Use-case diagram

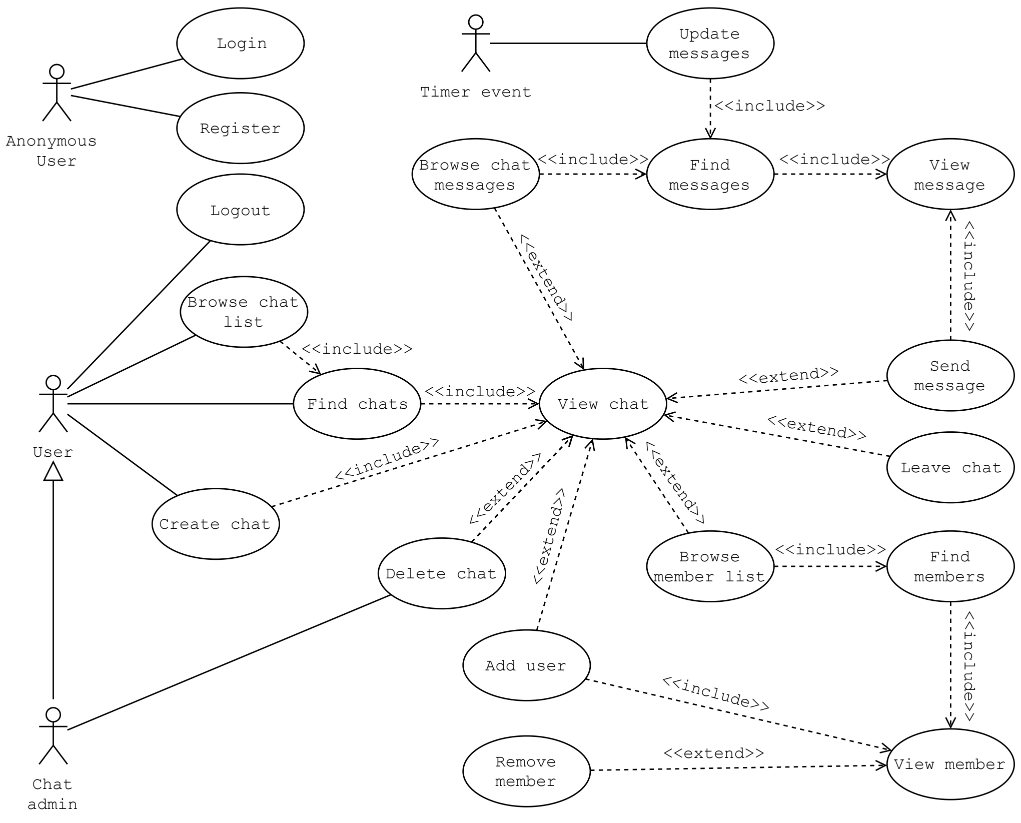

Figure 1: Use-case diagram

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

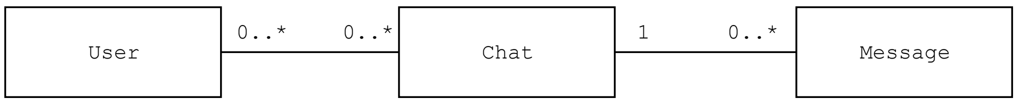

Figure 2: Class diagram for the identified entities.

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.

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 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.

The database APIs have been tested using some test units through *JUnit4* [3] before being integrated with the ReactJS frontend.

3.2 ReactJS frontend

The fronted has been developed using the *ReactJS* framework[4]. 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"[5] by the client.

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

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

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

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

[5] 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.

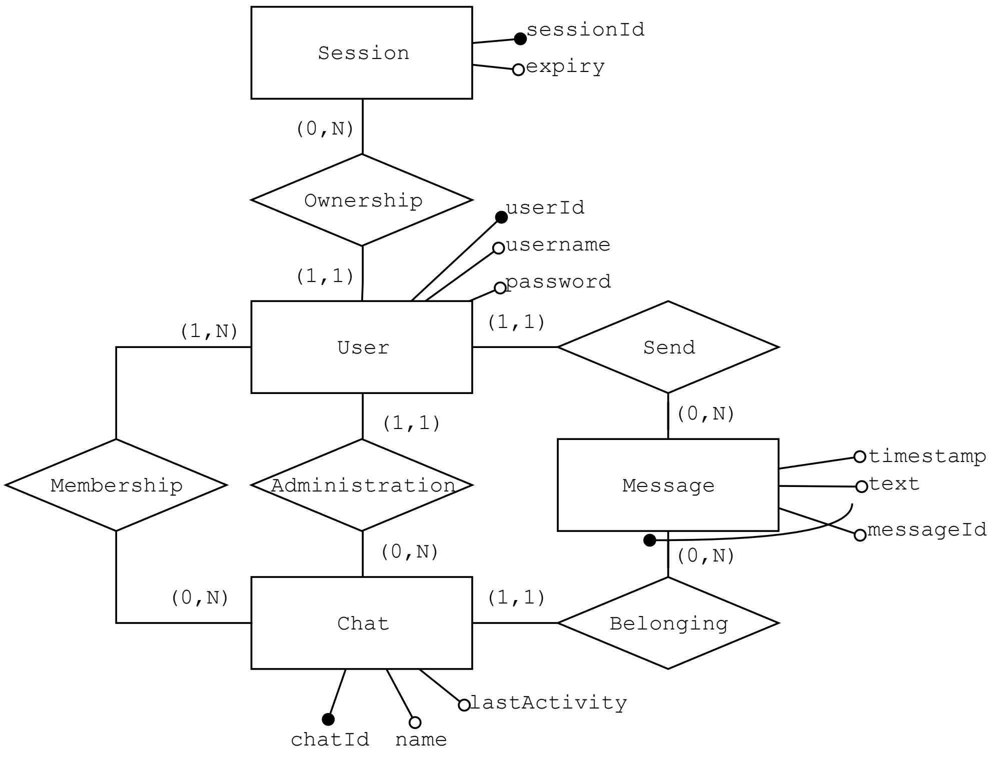

Figure 3: ER diagram for the database.

4 Database

4.1 SQL

Figure 3 shows the ER diagram of the SQL 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.

4.1.1 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.

4.2 Key-Value

4.2.1 Feasability study

Naive design Starting from the entity model and the use-cases, we first drafted a naive key-value implementation for the database and then we checked againts the use-cases whether it was efficient and we made some tweaks to it in order to improve it. The naive conversion is shown in the next table:

Key	Value
user:$userId:username	Users.username
user:$userId:password	Users.password
session:$sessionId:expiry	Sessions.expiry
session:$sessionId:userId	Sessions.userId
users:nextId	Next id for a user
chat:$chatId:name	Chats.name
chat:$chatId:lastActivity	Chats.lastActivity
chat:$chatId:members	List of chat members
chat:$chatId:admin	Chats.adminId
chat:$chatId:messages:nextId	Next id for a chat
chat:$chatId:message:$messageId:text	Messages.text
chat:$chatId:message:$messageId:timestamp	Messages.timestamp
chat:$chatId:message:$messageId:sender	Messages.senderUserId
chats:nextId	Next id for a chat

Where we decided to generate incremental ids using the nextId fields to store the id of the next generated entity.

Use-cases analysis Since mapping between use-cases and DB-abstraction-layer methods is straightforward, this analysis will take into consideration the methods of the database abstraction layer interface. Let's first list the methods whose implementation is trivial and presents no problems using the presented structure.

```
List<User> getChatMembers(long chatId);
boolean addChatMember(long chatId, long userId);
boolean removeChatMember(long chatId, long userId);
boolean checkChatMember(long chatId, long userId);
long addChatMessage(Message message);
long createChat(String name, long adminId, List<Long> userIds);
boolean deleteChat(long chatId);
Chat getChat(long chatId);
long createUser(User user);
long getUserFromSession(String sessionId);
boolean setUserSession(UserSession user);
boolean removeSession(String sessionId);
```

Let's now analyze the more tricky methods:

```
/**
* Returns the list of chats for the user identified by
* the given userId.
*
* @return the list of chats or null in case of error.
```

```java
*/
List<Chat> getChats(long userId);
```

Since one of the use-cases is retrieving the chats of one user, the naive conversion is not efficient since a linear search is required. Therefore, we should add a new list to map user chats, i.e.:

user:$userId:chats	List of chat ids where the user is present

```java
/**
* Returns the list of chats for the user identified by
* the given userId.
*
* @return the list of chats or null in case of error.
*/
List<Chat> getChats(long userId);
```

Since one of the use-cases is retrieving the chats of one user, the naive conversion is not efficient since a linear search is required. Therefore, we should add a new list to map user chats, i.e.:

user:$userId:chats	List of chat ids where the user is present

```java
/**
* Returns user identified by the given username.
*
* @return the user in case of success, null otherwise.
*/
User getUser(String username);
```

This method is inefficient using the proposed naive implementation. The use of a reverse index would make it more efficient, i.e.:

username:$username:userId	Users.userId

```java
/**
* Returns true if there exists a private chat between user1 and
* user2.
*
* NB: a private chat is a chat with only two members.
*
* @return {@code true} if it exists,
*         {@code false} otherwise or in case of errors.
*/
boolean existsChat(long user1, long user2);
```

This method can be implemented in an inefficent way iterating through the user's chats and its members. A more efficient way would be inserting a redundant list of private chats the user is in. However, since this is a proof-of-concept implementation, we decided not to make this optimization.

```java
/**
* Returns a list of messages for the given chat, in the given
* time range, up to the given number of elements, sorted in
* ascending message sending time.
*
```

```
* @param chatId id of the chat whose messages are to be retrieved
* @param from start of the time range (included). It can be null,
*        whose meaning is that there is no lower bound.
* @param to end of the time range (excluded). It can be null,
*        whose meaning is that there is no upper bound.
* @param n maximum number of elements to return.
*        If from is not null, messages are counted from
*        {@code from} up to n or {@code to}. If from is null,
*        messages are counted from {@code to} up to n or {@code from}.
* @return the list of messages or null in case of error.
*/
List<Message> getChatMessages(long chatId, long from, long to, int n);
```

Even though it may look tricky, this query is in fact quite trivial since message ids are strictly monotone, due to the id generation strategy we imposed.

To tell the truth, this API was originally taking a time range, making its implementation very complicated using a key-value database.

Another problem with this API was its slowness due to the three required operations required to read the message from the database (there are 3 fields to be fetched). This problem was discovered thanks to the benchmarks of section 5. The solution has been to aggregate the three fields under one single key since they are always fetched together.

4.2.2 Final design

Key	**Value**
user:$userId:username	Users.username
user:$userId:password	Users.password
user:$userId:chats	List of chat ids where the user is present
username:$username:userId	Users.userId
session:$sessionId:expiry	Sessions.expiry
session:$sessionId:userId	Sessions.userId
users:nextId	Next id for a user
chat:$chatId:name	Chats.name
chat:$chatId:lastActivity	Chats.lastActivity
chat:$chatId:members	List of chat members
chat:$chatId:admin	Chats.adminId
chat:$chatId:messages:nextId	Next id for a chat
chat:$chatId:message:$messageId	Messages.*
chats:nextId	Next id for a chat

4.2.3 Implementation

The Java implementation has been done using *LevelDB*[6]. It is a simple Key-Value database, providing just three methods to access the store: *put*, *get* and *delete*.

[6]More information at: `https://github.com/google/leveldb`

5 Benchmarking

Since our Java backend provides a convenient HTTP API, we were able to use the open-source software Siege[7] in order to benchmark the different database backends.

We used one concurrent user and measured the average *transactions per second* (tps) for 5000 requests over three runs.

Disclaimer The tests were not done in a rigorous way but they were executed just to get an idea on the capabilities of the different database technologies.

Database/Implementation	get chat list	send messages	get messages
SQL/plain	397 (309)	217	13.05
SQL/JPA	278 (225)	136	9.69
LevelDB/no_aggregation	1022 (587)	1050	3.53
LevelDB/aggregated	1033 (527)	950	12.81

In the first column, the value between parenthesis represents the value for the first run, which is the minimum and has been exluded from the average, since it is the only one that didn't take advantage of caching (as the other two runs).

From the results it can be seen that:

- plain SQL is slightly better than JPA.
- Message field aggregation in LevelDB is more than 3x faster in getting the list of messages.
- LevelDB is much faster than SQL in the first two operations, while shows similar performances in reading many values (as in the last operation).

We also tried increasing the number of concurrent users, obtaining an overall better tps (obviously), but showing the same ratios between different backends.

[7]More information at: `https://www.joedog.org/siege-home/`