INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
TOULOUSE

Chat application

Object Oriented Design

Project by :

CHALHOUB Sandro & MOUSSET Sarah
4IR - A2

- 25/01/2023 -

Sommaire

Introduction

The aim of the Chat Application project is to design a communication system that allows individuals to chat with each other over a local network. The system must be able to adapt to different scenarios and to be deployable on various environments.

To begin this project, based on the provided requirements, we chose to design our system prior to trying any kind of implementation. In this view, we used several approaches, thereby creating different kinds of UML diagrams, in pursuit of a better comprehension of the aforementioned requirements and their resulting object-oriented interactions. When it comes to the programming itself, we chose to adopt the Agile method and organised our work in sprints. We also took the time to identify the limitations and difficulties of the project.

There's no doubt that designing our application prior to any programming attempt made it easier for us to start the latter. However, and unsurprisingly, programming our application allowed us to discover obstacles that we hadn't foreseen during the design stage. As a result, modifications were made to our initial design, which are included in this final version of the report.

1 User Manual

1.1 Login window

When the application is launched, the LoginStage window appears, and the user is asked to either sign up or log in, depending on whether they've used the application before or not. If it's the user's first time using ChatApp, the username and password entered will be the expected username and password for future logins.

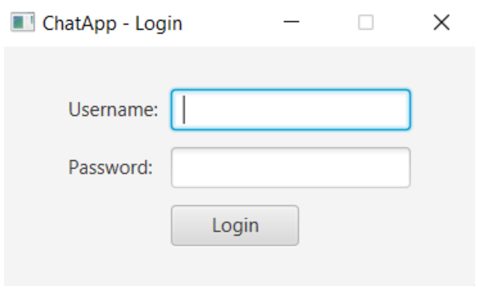


FIGURE 1 – Login Window

1.2 Main window

Once connected, the ChatAppStage, which is the main window of our application, appears, and the LoginStage window disappears. The lists of all online and offline users appear separately. A user can read their chat with a certain other user by double-clicking on the latter's username in the list. A user can also request to start a chat with another online user by clicking on the *Invite* button. Two users will only be able to chat if one has sent a chat request to another, and the latter accepted the request. However, a user can unilaterally end an active chat by clicking on the *End* button for that chat.

Closing the main window is equivalent to disconnecting from the application, thus triggering the closure of all ChatApp-related windows and the termination of all relevant processes. Finally, a user can request the modification of their username from the main window by clicking on the Change Username button, which pops up a separate window that allows them to choose the new username.

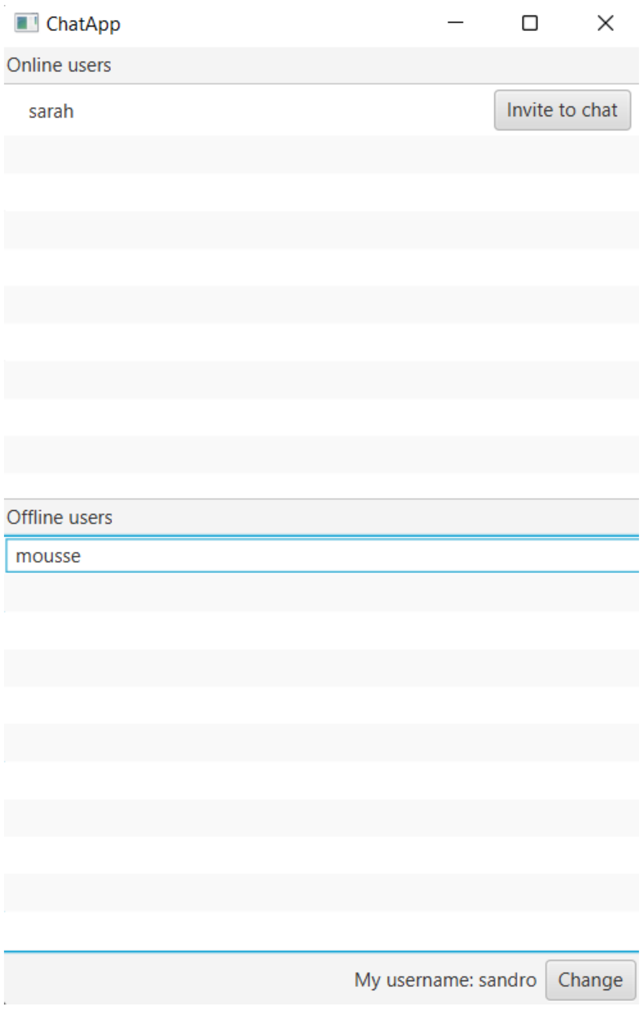


FIGURE 2 – Main Window

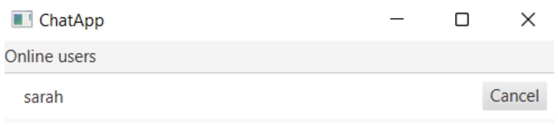


FIGURE 3 – Chat request sent

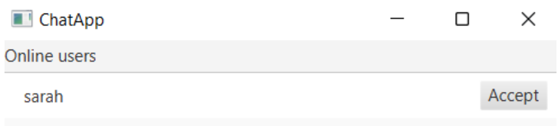
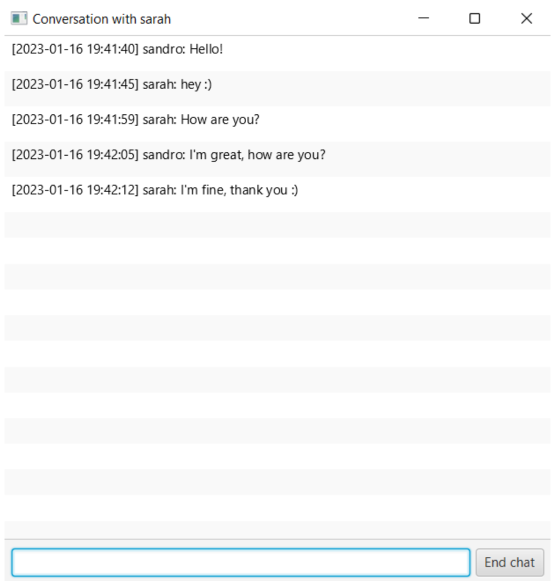
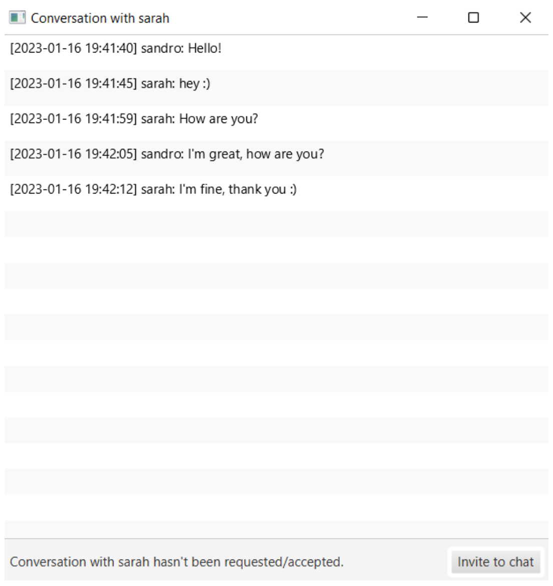


FIGURE 4 – Chat request received



```
[2023-01-16 19:41:40] sandro: Hello!
[2023-01-16 19:41:45] sarah: hey :)
[2023-01-16 19:41:59] sarah: How are you?
[2023-01-16 19:42:05] sandro: I'm great, how are you?
[2023-01-16 19:42:12] sarah: I'm fine, thank you :)
```

FIGURE 5 – Ongoing chat

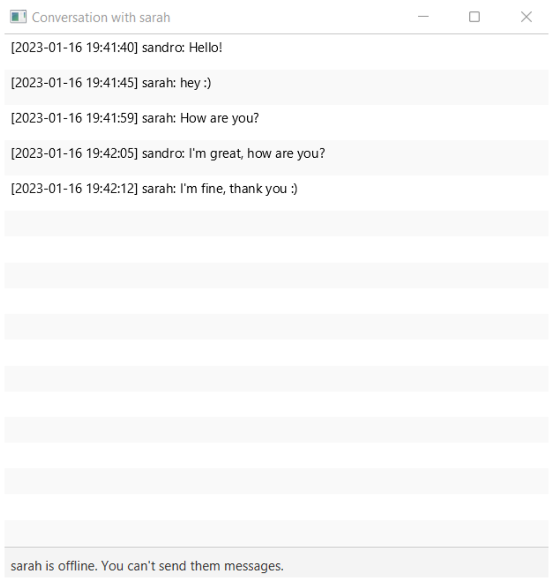


```
[2023-01-16 19:41:40] sandro: Hello!
[2023-01-16 19:41:45] sarah: hey :)
[2023-01-16 19:41:59] sarah: How are you?
[2023-01-16 19:42:05] sandro: I'm great, how are you?
[2023-01-16 19:42:12] sarah: I'm fine, thank you :)
Conversation with sarah hasn't been requested/accepted.
```

FIGURE 6 – Inactive chat

1.3 Chat window

The chat window pops up when a user double clicks on the username of another user in the list. The history of their previous chats automatically appears in the right chronological order. Each message shows the sender, time and date. A text field at the bottom of the window lets the user write messages. Sending the typed message is done by hitting on Enter.



```
[2023-01-16 19:41:40] sandro: Hello!
[2023-01-16 19:41:45] sarah: hey :)
[2023-01-16 19:41:59] sarah: How are you?
[2023-01-16 19:42:05] sandro: I'm great, how are you?
[2023-01-16 19:42:12] sarah: I'm fine, thank you :)
sarah is offline. You can't send them messages.
```

FIGURE 7 – History of chat with offline user

1.4 Username modification window

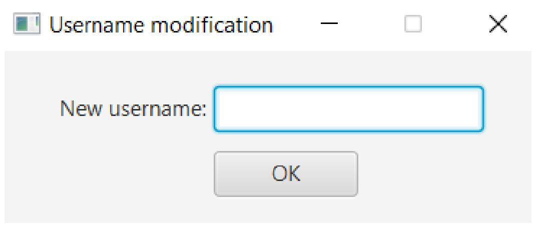
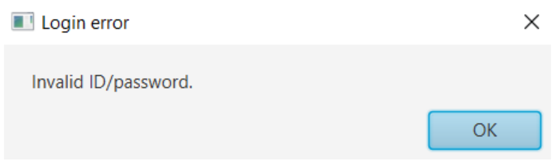


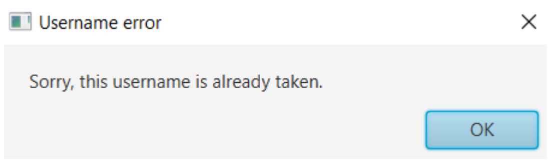
FIGURE 8 – Username modification window

1.5 Error messages



Invalid ID/password.

FIGURE 9 – Invalid username or password



Sorry, this username is already taken.

FIGURE 10 – Username modification error

2 Use-case diagram

The first diagram we designed is a use-case diagram, which allows for a better understanding of how and why someone may want to use our application.

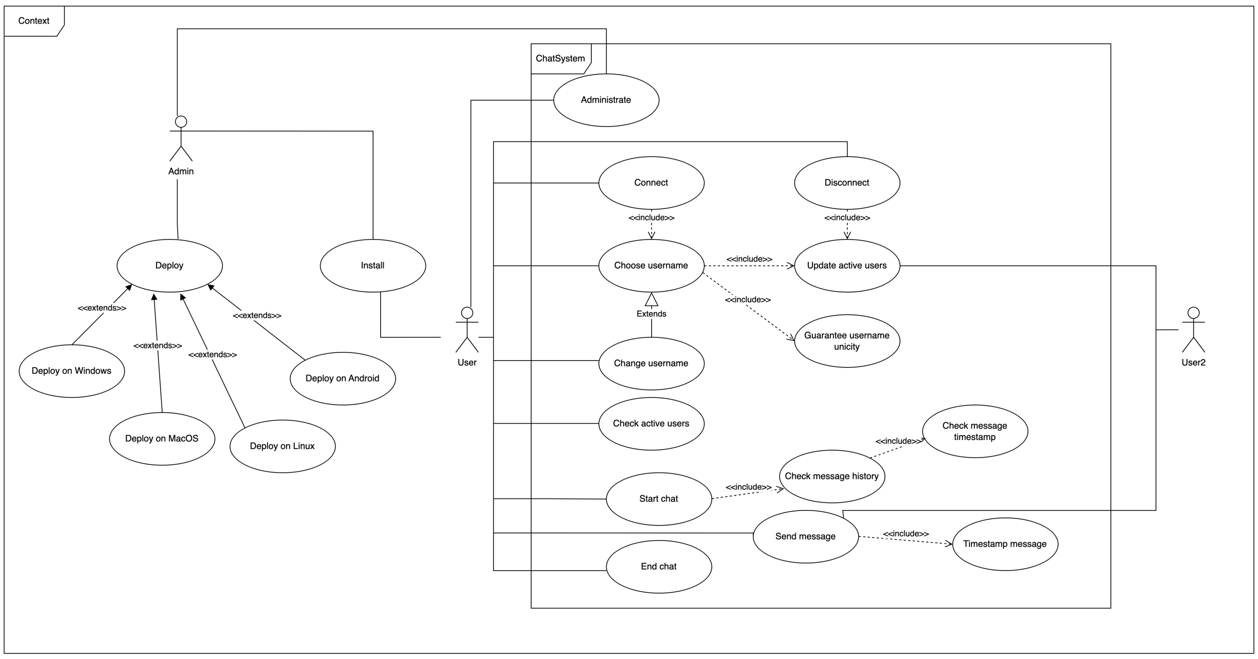


Figure 11 – Use-case diagram

We've identified two main actors : the system administrator and the primary user. We've also identified a secondary actor, representing another active user. The administrator manages the compatibility of the agent with the operating system in which it is deployed, while the primary user is the one who interacts with the agent itself. In this precise vision of the application, the other user is considered 'idle' : they simply receive a message or update their knowledge of the active actors. They would effectively take on the role of the main user if they came to take an action themself.

3 Class diagram

The following class diagram allows us to represent the dependencies and various interactions existing between the classes in our design. We've chosen to follow an MVC (Model View Controller) architecture, by organising the classes in packages depending on their respective responsibilities.

From a general point of view, the application works as follows : the user communicates via the View classes which, on the occurrence of certain events, can trigger listener methods by calling the fire methods in ListenerManager. The latter manages all our application listeners. As a result, any action performed on the GUI generates specific responses from listener implementations, depending on the listeners triggered by the action and the classes implementing those listeners.

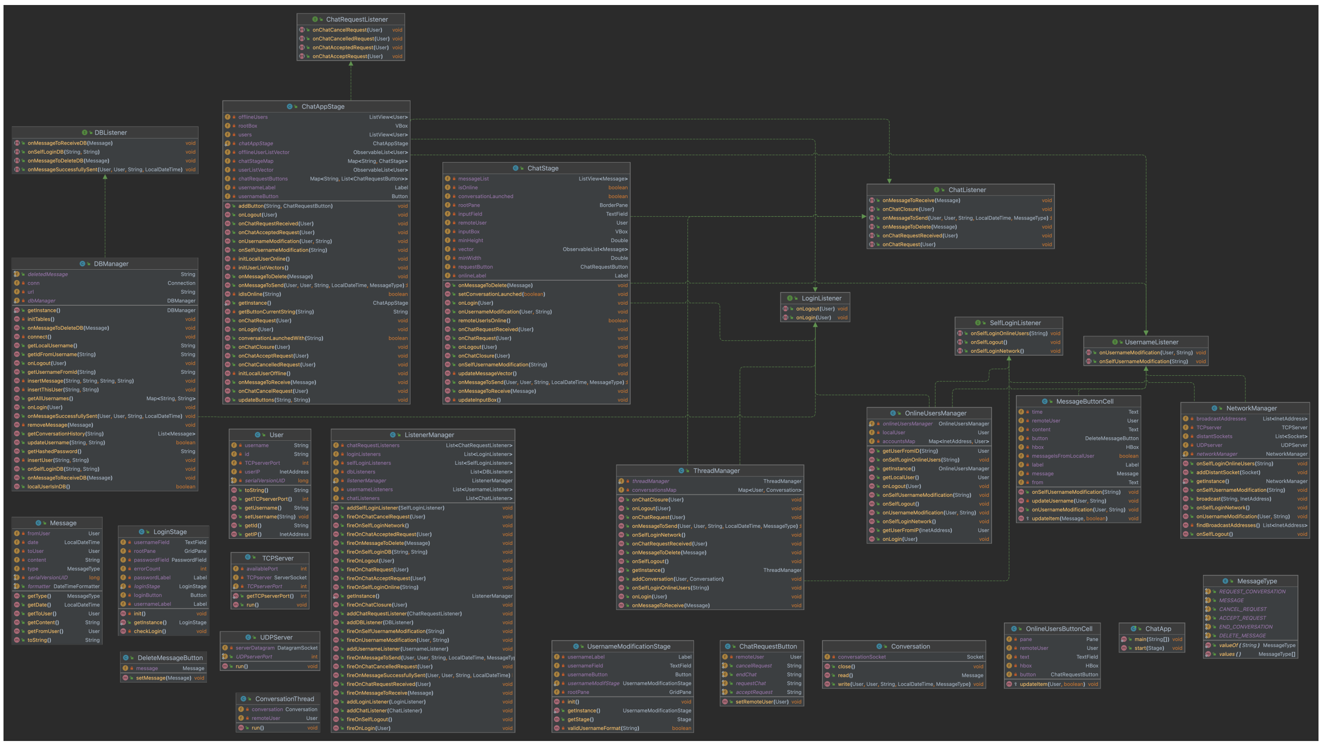


FIGURE 12 – Class diagram

- The SelfLoginListener is associated with the local user's login and logout events.
- The LoginListener is associated with remote users' login and logout events, from the local user's point of view.
- The DBListener is associated with database-related events (credentials verification, adding, retrieving and deleting messages, etc.)
- The ChatListener is associated with chat-related events (chat request, chat request acceptance, chat request cancellation, sending and receiving messages).
- The ThreadManager manages the local user's active conversations.
- The Networkmanager manages the local user's UDP and TCP servers.
- The OnlineUsersManager locally manages the online users list.
- The DBManager manages the local database. It makes it possible to add and delete messages, and to load chat histories. It also makes it possible to verify the credentials entered by the user during login.

4 Sequence diagrams

The following sequence diagrams allow for a better understanding of the interactions between the different objects, following a user's actions.

4.1 Signup & login from the local user's point of view

When the user launches ChatApp, they are prompted to enter their username and password. If it's the user's first time using ChatApp, the chosen username and password are stored in the local database. Indeed, the Login button pressed by the user is interpreted as an event which triggers the onSelfLoginDB method of DBListener. The implementation of this method by the DBManager stores the new user's credentials in their local database, and then broadcasts online status to the rest of online users. However, if the user isn't using ChatApp for the first time, the username and password entered are, instead, compared to those already existent in the user's local database.

If the credentials don't match, an error message is displayed, and the user can try to log in again if they wish. Otherwise, the user is connected : the methods of the SelfLoginListener, implemented by OnlineUsersManager and NetworkManager, among others, are fired by consequence, and allow the list of active users of the local user to be updated, and to broadcast the active status of the latter to the other online users of the application. As soon as the user is logged in, the signup / login window is automatically closed, and the main application window, where all active users are listed, is displayed.

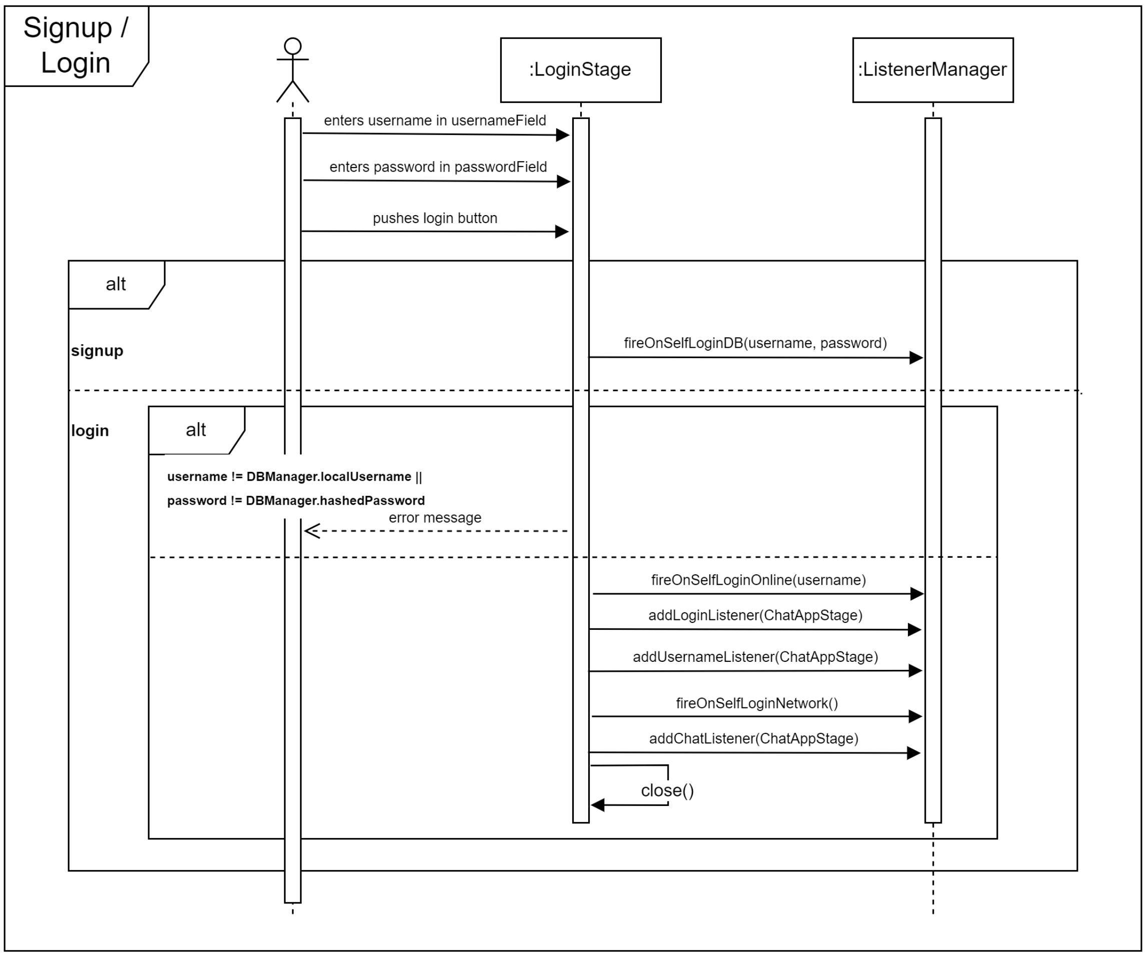

Signup / Login
:LoginStage
:ListenerManager
enters username in usernameField
enters password in passwordField
pushes login button
alt
signup
fireOnSelfLoginDB(username, password)
login
alt
username != DBManager.localUsername ||
password != DBManager.hashedPassword
error message
fireOnSelfLoginOnline(username)
addLoginListener(ChatAppStage)
addUsernameListener(ChatAppStage)
fireOnSelfLoginNetwork()
addChatListener(ChatAppStage)
close()

Figure 13 – Sequence diagram of the signup & login procedures

4.2 Signup & login from the remote user's point of view

As we've seen, logging in implies firing, among other listener methods, the onSelfLoginNetwork method, which leads to the broadcast of the user's online status. The reception of that broadcast message will be managed by each remote user's UDPServer instance, which will fire the onLogin listener method. This method is implemented by DBManager, OnlineUsersManager and ChatAppStage.

The DBManager makes sure that the username of the user logging in is up to date in the local database ; the OnlineUsersManager adds the user to the locally-managed map of online users ; the ChatAppStage removes the user from the offline users list view, and adds them to the online users list view.

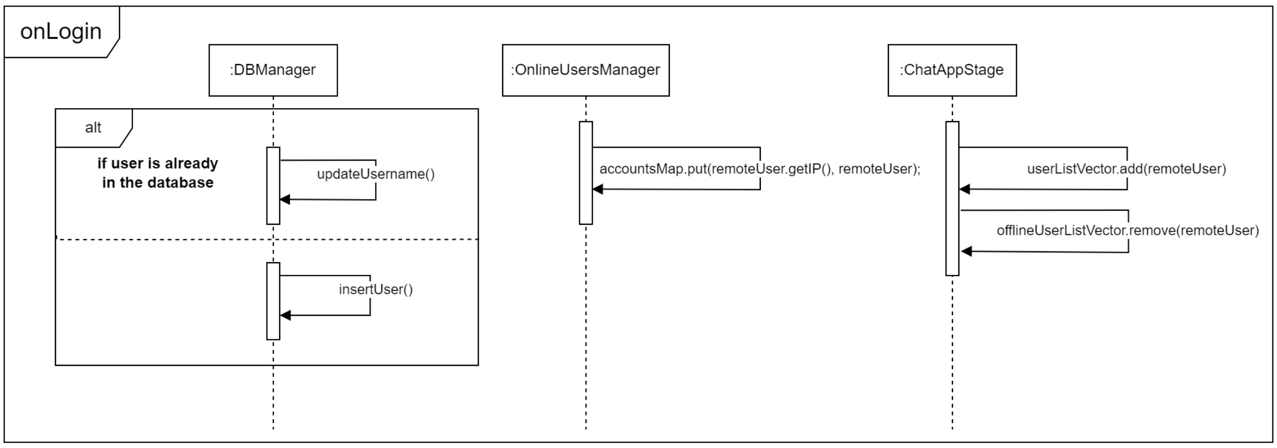

onLogin
:DBManager
:OnlineUsersManager
:ChatAppStage
alt
if user is already in the database
updateUsername()
insertUser()
accountsMap.put(remoteUser.getIP(), remoteUser);
userListVector.add(remoteUser)
offlineUserListVector.remove(remoteUser)

Figure 14 – Sequence diagram of the *onLogin* method

4.3 Logout from the local user's point of view

Closing the main window leads to the user's disconnection. This event will trigger all the necessary actions to properly disconnect the user. Indeed, the SelfLoginListener's onSelfLogout method is fired. The latter is implemented by several managers, including, but not limited to : the NetworkManager, which broadcasts the logout information to all active users, so that the local user no longer appears as active on everybody else's windows ; and the ThreadManager, which closes all conversations left active by the user prior to their logging out.

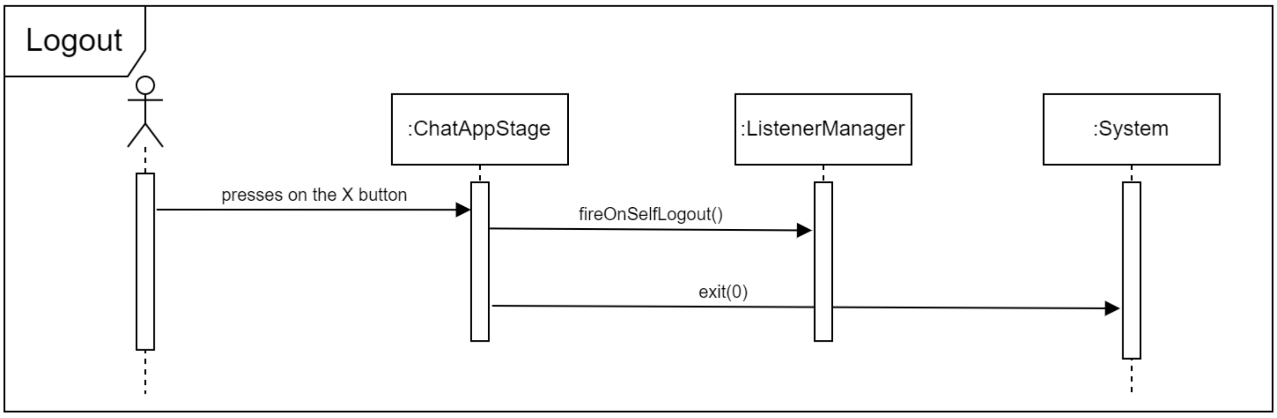


Figure 15 – Sequence diagram of the logout from the local user's point of view

4.4 Logout from the remote user's point of view

When a user logs out, a logout broadcast is sent to all the active ChatApp users, and is managed by those users' UDPServer instances. As a consequence, the onLogout listener method is fired, and is implemented by ThreadManager, OnlineUsersManager and ChatAppStage.

The ThreadManager makes sure that any active conversation with the logging out user is properly closed ; the OnlineUsersManager removes the logging out user from the locally-managed online users map ; the ChatAppStage removes the user from the online users list view, and adds them to the offline users list view.

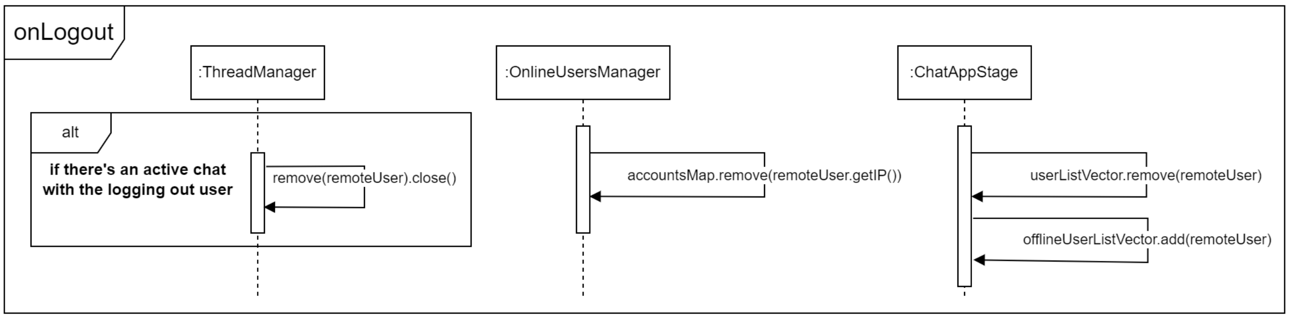

onLogout
:ThreadManager
:OnlineUsersManager
:ChatAppStage
alt
if there's an active chat with the logging out user
remove(remoteUser).close()
accountsMap.remove(remoteUser.getIP())
userListVector.remove(remoteUser)
offlineUserListVector.add(remoteUser)

FIGURE 16 – Sequence diagram of the onLogout method

4.5 Sending a chat request

A user can request to chat with another online user. To do so, all they need to do is click on the Invite button on the right side of the username in question. When that button is clicked, both the onChatRequest and onMessageToSend listener methods are fired.

The first one is implemented by ThreadManager, which will request a TCP connection to the listening server of the remote user, connect to the redirected TCP server, and then create a Conversation object (basically consisting of a socket) with that user. It is also implemented by ChatAppStage in order to update the text and functionality of the button pressed from *Invite* to *Cancel.*

As for the second listener method, onMessageToSend, it is fired with 'REQUEST CONVERSATION' as one of its arguments, as for the message, which is actually empty, to be interpreted by the remote user's ConversationThread object as a conversation request - more on that in the next sequence diagram.

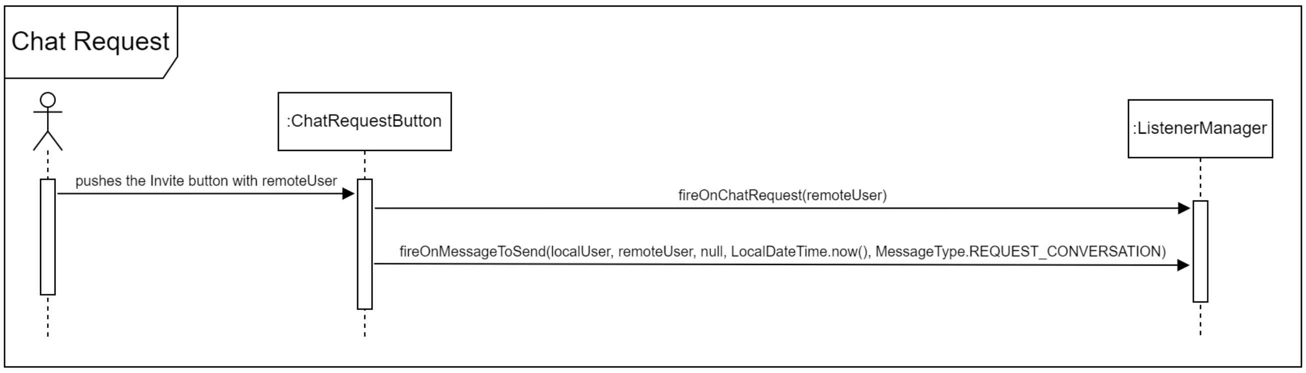

Chat Request
:ChatRequestButton
:ListenerManager
pushes the Invite button with remoteUser
fireOnChatRequest(remoteUser)
fireOnMessageToSend(localUser, remoteUser, null, LocalDateTime.now(), MessageType.REQUEST_CONVERSATION)

FIGURE 17 – Sending a chat request sequence diagram

4.6 Receiving and accepting a chat request & ending the chat

On reception of the aforementioned REQUEST_CONVERSATION message from a certain user, thanks to the conversation.read() call in the ConversationThread's run() method, the listener method onChatRequestReceived is fired. This method is only implemented by ChatAppStage, allowing for the modification of the button next to the requesting user from *Invite* to *Accept*. As such, when and if the *Accept* button is clicked, the listener methods onChatAcceptRequest and onMessageToSend are fired.

The first one, again, updates the button to the *End* functionality, allowing for the eventual termination of the chat. The second method, for its part, is fired with ACCEPT_REQUEST as one of its parameters, as for the message to be interpreted as an acceptance for the chat launch by the remote user's ConversationThread object (which will basically update the button to the *End* functionality for the remote user as well). At this point, the chat is up and running.

If, at any point, one of both users presses on the *End* button, the onChatClosure and onMessageToSend methods are fired. onChatClosure is implemented by the ThreadManager as to induce the closure of the Conversation socket and thread. onMessageToSend, for its part, is fired with END_CONVERSATION as one of its arguments, which will be interpreted by the remote user's ConversationThread as a mandatory request to end the chat - so it will also fire onChatClosure for the remote user.

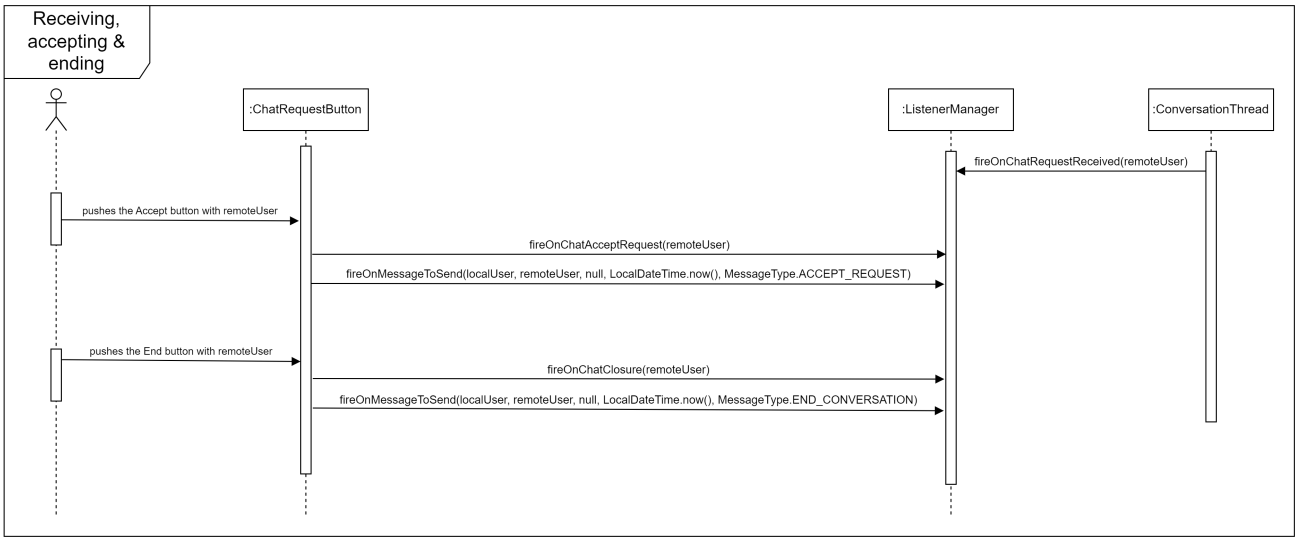

Receiving, accepting & ending
:ChatRequestButton
:ListenerManager
:ConversationThread
fireOnChatRequestReceived(remoteUser)
pushes the Accept button with remoteUser
fireOnChatAcceptRequest(remoteUser)
fireOnMessageToSend(localUser, remoteUser, null, LocalDateTime.now(), MessageType.ACCEPT_REQUEST)
pushes the End button with remoteUser
fireOnChatClosure(remoteUser)
fireOnMessageToSend(localUser, remoteUser, null, LocalDateTime.now(), MessageType.END_CONVERSATION)

Figure 18 – Receiving and accepting a chat request & ending the chat sequence diagram

4.7 Sending a message

Once a chat between two users is up and running, a user is able to send and receive messages. In order to send a message, the user enters some text in the ChatStage text field, and presses Enter whenever they wish to actually send the message. It is worth noting that the empty string is not an allowed input - nothing happens if a user tries to press Enter without having typed anything prior to that. When Enter is pushed, the ChatStage fires the onMessageToSend listener method with the typed text as input parameter, and type MESSAGE. As such, it will be interpreted by the remote user's ConversationThread object as a message to receive. onMessageToSend is implemented, among others, by the ChatStage itself, in order to add the message to the buttom of the local user's ChatStage view.

On reception of a message over TCP, a user's ConversationThread will fire the onMessageReceived listener method. This method will trigger the addition of the message to the conversation map, to the chat view, and to the local database.

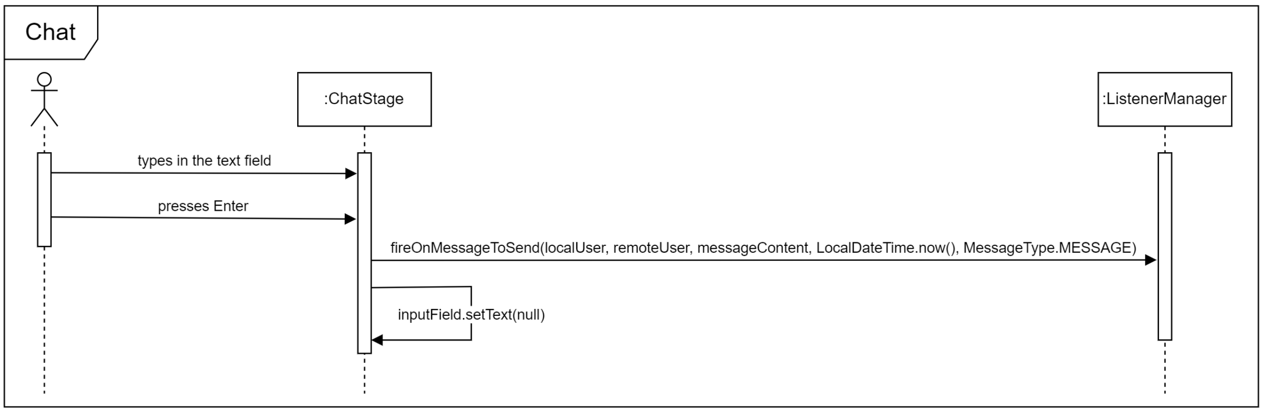

Chat
:ChatStage
:ListenerManager
types in the text field
presses Enter
fireOnMessageToSend(localUser, remoteUser, messageContent, LocalDateTime.now(), MessageType.MESSAGE)
inputField.setText(null)

Figure 19 – Chatting sequence diagram

4.8 Username modification

At any moment, a user can change their username through the UsernameModificationStage. The username must have a valid format, and must not be used by any other ChatApp user. As such, two verifications happen prior to the modification taking effect. Once done, the UsernameModificationStage disappears and the onSelfUsernameModification listener method is fired. The method is implemented by the different views in order to update the local user's username on their own screen ; but more importantly, it's implemented by the NetworkManager in order to broadcast the username change to all online users. DBManager also updates the local database using its updateUsername method. On reception of the broadcast, the UDPServer of the respective online remote users fires onUsernameModification which leads to the modification of the different views, as well as the database, of those remote users.

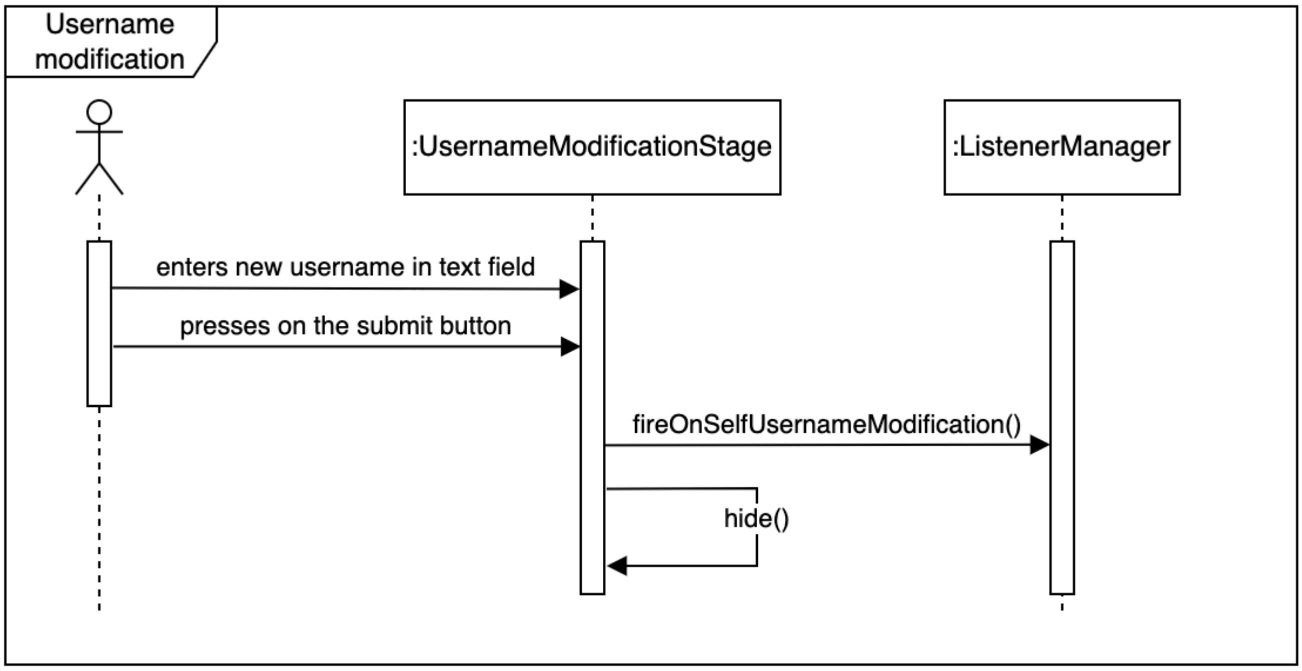

Username modification
:UsernameModificationStage
:ListenerManager
enters new username in text field
presses on the submit button
fireOnSelfUsernameModification()
hide()

Figure 20 – Username modification sequence diagram

5 Composite diagram

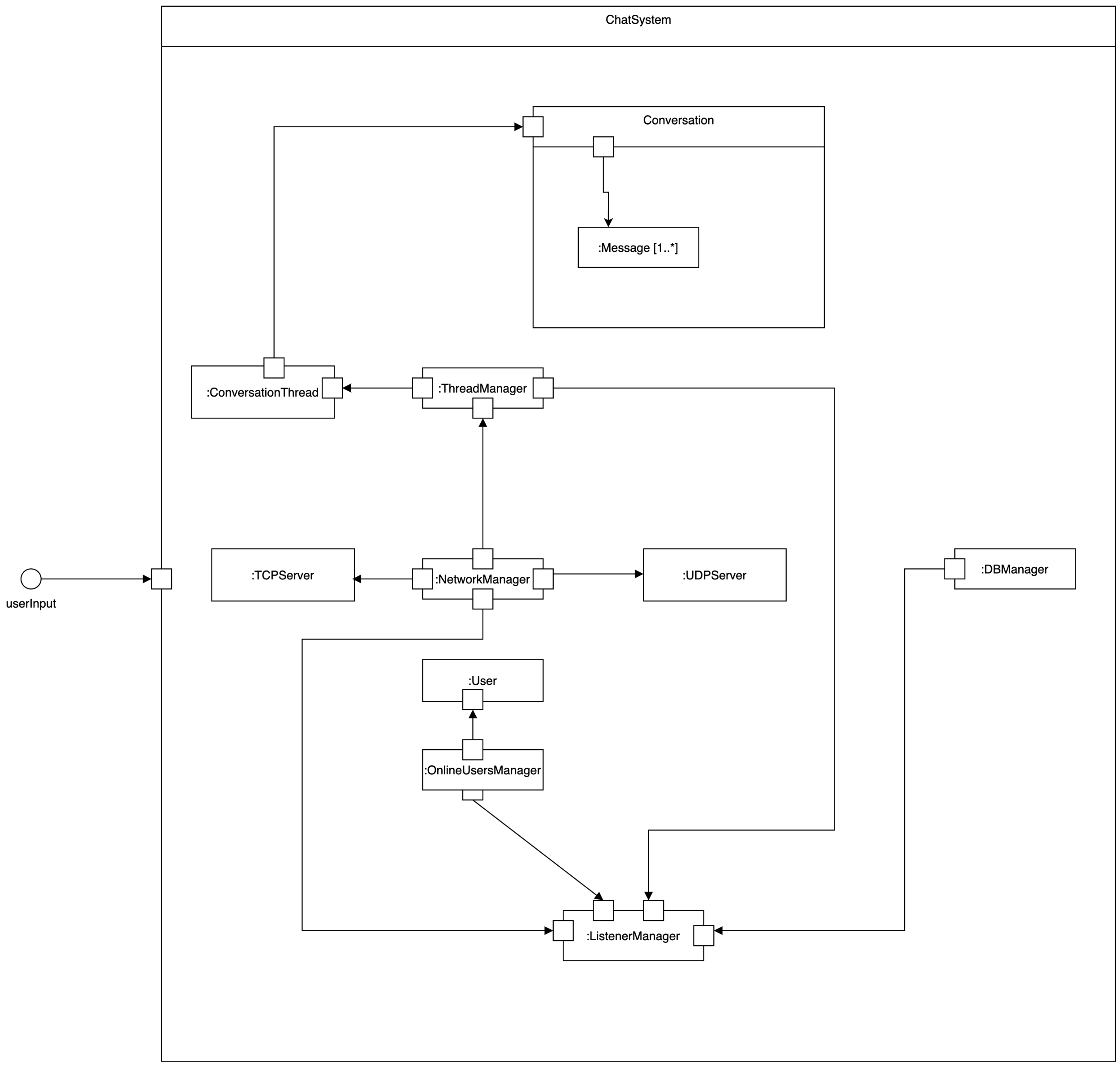


FIGURE 21 – Composite diagram

6 Entity-relationship database diagram

The decentralised database has two main functions : to store the user's credentials on sign-up in order to make login possible in the future, and to store the messages of all their chats, making it possible to load chat histories. Obviously, two users can only have a single chat, but a user can chat with several other users. Similarly, a chat may contain several messages, but a message belongs to a single chat. Our relational diagram, which we chose to design using the entity-relationship model, contains two entities : Message and User. The *send* relationship will be reflected in the database implementation by the presence of the fromID key of User as a foreign key in the Message table, which makes it possible to identify both the sender and receiver of each message sent or received by the local user.

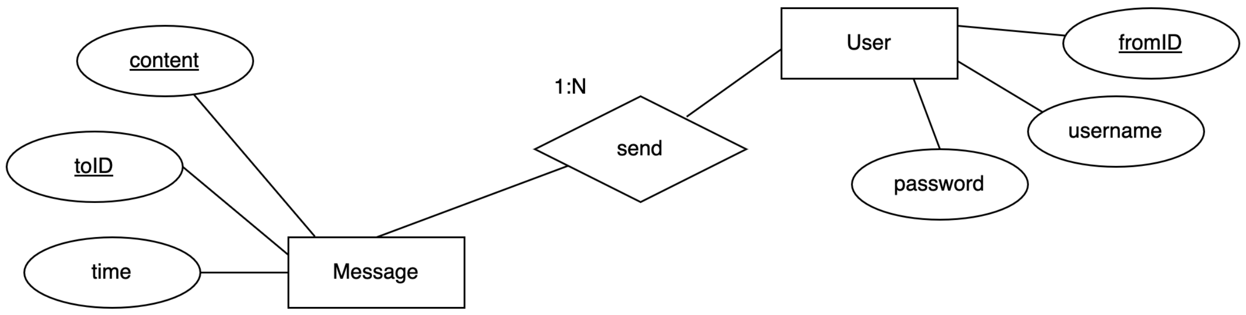


Figure 22 – Entity-relationship database diagram

Conclusion

In a nutshell, designing our application, while taking into consideration all the constraints specified in the functional and non-functional requirements, allowed us to have a better idea of how we could eventually program it. Still, we've come across obstacles that we hadn't envisioned prior to the start of the implementation phase, which made us modify multiple aspects of the initial design.

Moreover, this project was an opportunity for us to get a first serious hands-on approach to the object-oriented programming paradigm, to the Agile software project management methodology, as well as to automation tools such as Maven. We're a lot more confident in the use of these methods and tools today than we were prior to the realisation of this project.