



INSA

INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
TOULOUSE

Florian LEON
Nahom BELAY
4IR
Groupe A1
Semestre 1
Année universitaire 2020-2021

Rapport POO/COO
Rapport de Projet

Florian LEON
Nahom Belay
4IR
Groupe A1
Semestre 1
Année universitaire 2020-2021

Rapport POO/COO
Rapport de Projet

Table of Contents

<i>Introduction.....</i>	<i>1</i>
<i>I. Installation manual.....</i>	<i>2</i>
A) Fetching the files	2
B) Importing the project on Eclipse.....	2
C) Importing the Web Application on Eclipse	5
<i>II. User Manual.....</i>	<i>6</i>
<i>III. Notable choices we made during the project.....</i>	<i>8</i>
A) JavaFX and Scene builder for the UI	8
B) Databases	8
C) Connexion protocols	10
D) Server implementation	10
<i>IV. UML Diagrams</i>	<i>13</i>
A) Initial UML diagrams we had designed	13
B) Class Diagram generated from our Java Project	16
<i>Conclusion.....</i>	<i>17</i>

Introduction

In our COO/POO project, we, Florian LEON and Nahom BELAY, created a chat application using Java.

In this report, we explain how to install and use our application, the tools we used and why we picked those tools and finally we present the UML diagrams we had implemented during the design phase.

I. Installation manual

A) Fetching the files

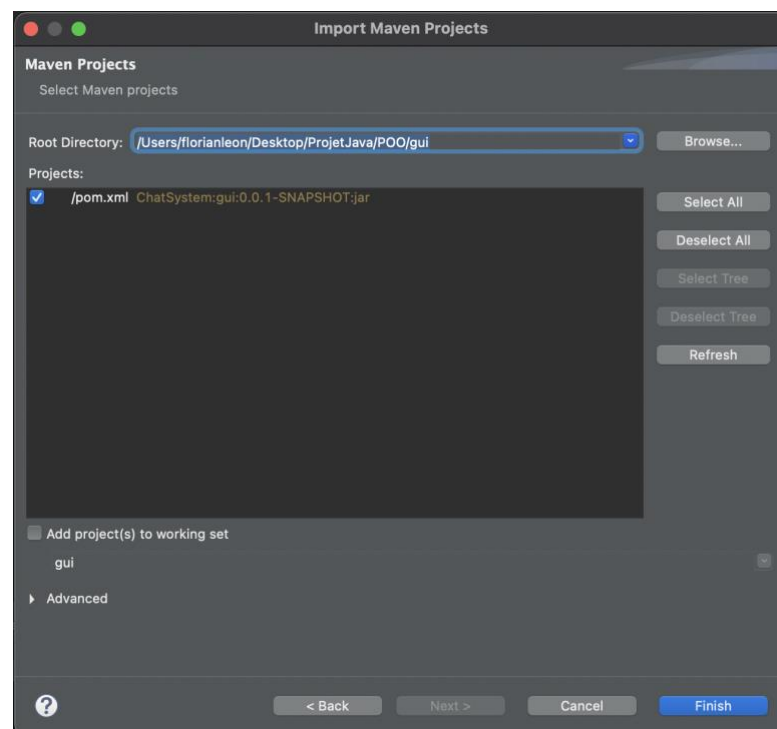
- Get the project from GitHub using this URL:
<https://github.com/nahombelay/ProjetJava> and clone it in your file system of both tests computers.
- Download the project as a ZIP using this URL:
<https://github.com/nahombelay/ProjetJava>. Click on « Code » and then on « Download as ZIP ». Unzip it in the location of your choice.

B) Importing the project on Eclipse

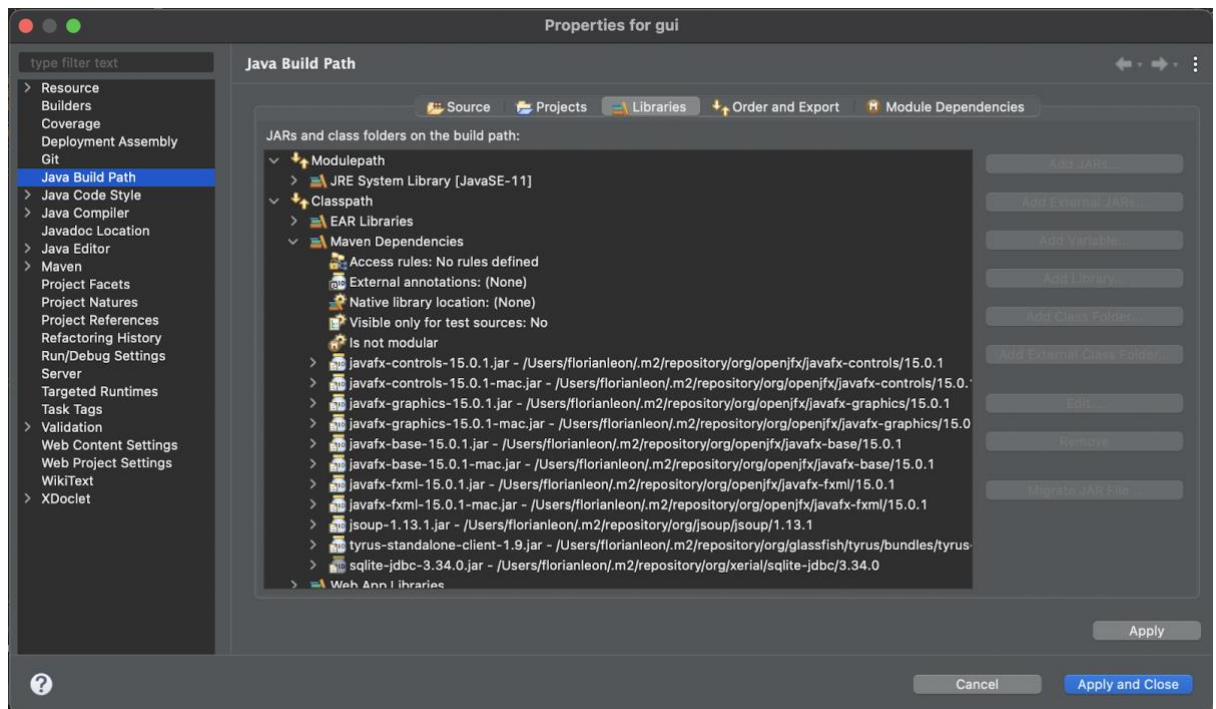
We did not successfully export the project to a jar file due to the multiple dependencies and the lack of time at the end. We used Eclipse for coding and testing so we will only detail how to install it on Eclipse. The Initial project has been completely converted to a maven project so you will not need to use any external Jars. Every dependencies should be automatically downloaded with maven.

The project has been tested and works with JDK 15 but it should work with JRE 11 and 14.

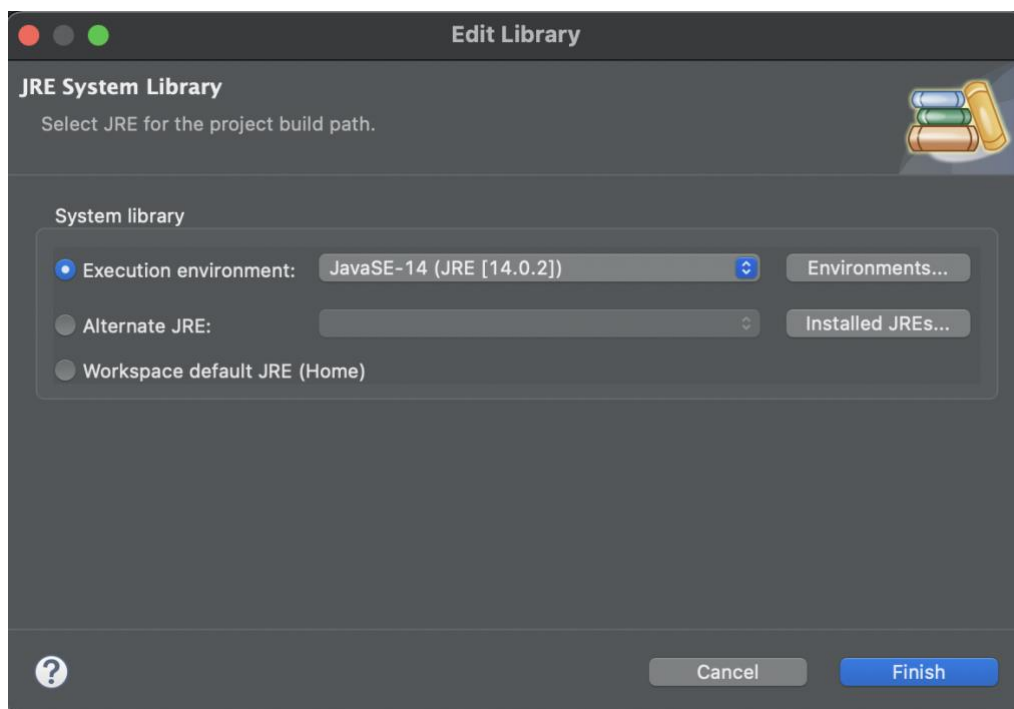
- Open Eclipse (Last version preferred)
 - Choose the folder POO as the Workplace's directory (Not compulsory)
 - Import the project in the workplace : File —> Import... —> Maven —> Existing Maven Project.
- Then, click on Browse... and open the folder gui, and Finish.



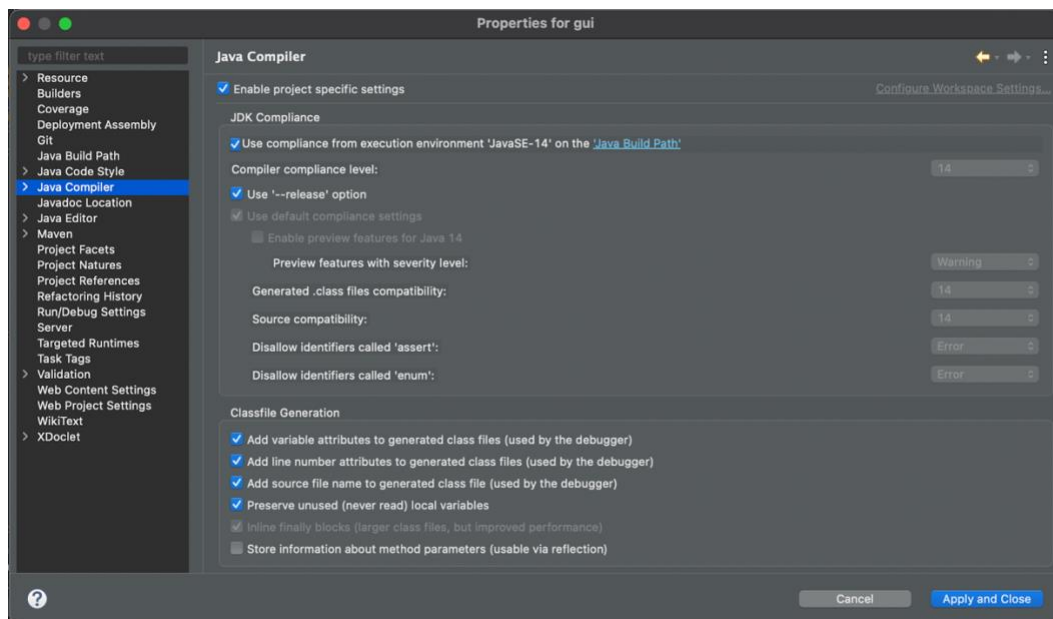
- Right-click on the project gui, Built Path → Configure Build Path...
Develop the Maven Dependencies and make sure that all of these Jars are here. (See Figure Below)



- Then, click on the JRE and click Edit...
Select JavaSE-14 (JRE [14.0.2]) in the Execution environment list or click on Workspace default JRE if you have JDK 15 installed on your computer and click Finish and then Apply.

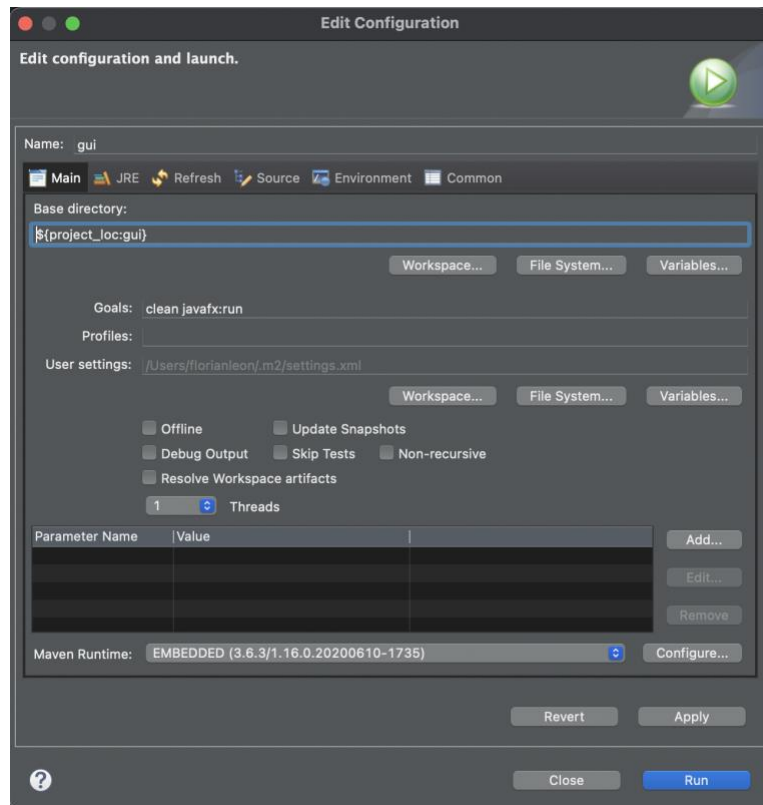


- On the left click on Java Compiler. Make sure that all information is filed as pictured on the figure below. If you choose to use JDK 15, uncheck Use compliance from execution environment and put the Compiler compliance level on 14.



Click on Apply and Close.

- Open the class MessageDB.java located at gui —> src/main/java —> ChatSystem.database. Right-click on the file : Run As —> Java Application or click run on the toolbar. This message should appear in the console « [class ChatSystem.database.MessagesDB] Connection OK ».
- Close the class, we will no longer need it. This step is to make sure that all of our messages we used for testing are deleted. /!\ Do not run it if you did not intend to delete all messages.
- Open the class Main.java located at gui —> src/main/java —> ChatSystem.gui
- Right-click on the gui project : Run As —> Maven Build...
Name it as you want (in our case gui)
Into Goals, type « clean javafx:run » (see Figure Below), click Apply and then Run.



- From now on, to run the project, just right-click on the project gui : Run As → Maven Build and It will automatically select the latest configuration.

C) Importing the Web Application on Eclipse

The web application is located in the POO folder under the name “ChatSystemWebsockets”. Import the project by clicking on Import → General → Existing Projects from Workspace and selecting the project from where you stored it.

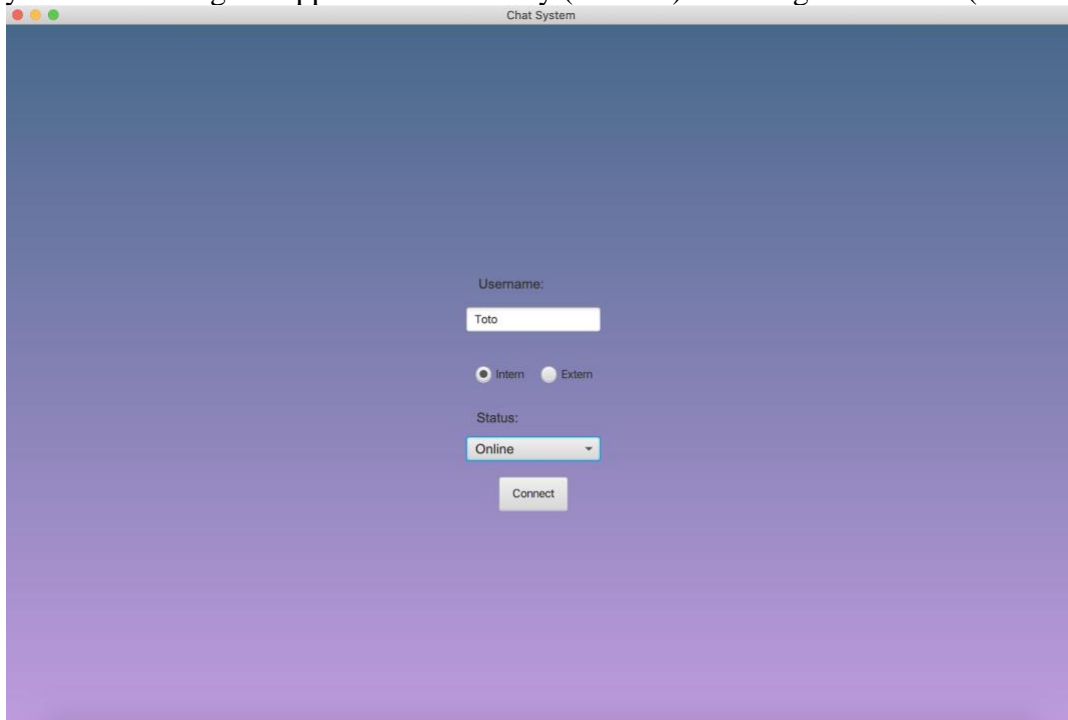
The web.xml file specifies on which port the client should connect. By default it is on port 8080 but if you decide to change it, make sure to modify the web.xml file as well as the method *websocketThread()* in the ExternalUser class.

Export the project as a WAR and deploy it on a Tomcat server that is running on your local computer. When testing with multiple computers, make sure that the computers that are not running the server connect to the server using the correct IP address and not localhost. The line that should be modified is in the method *websocketThread()* in the ExternalUser class.

When deploying the server, make sure to connect to the INSA VPN in order to access INSA’s SQL sever.

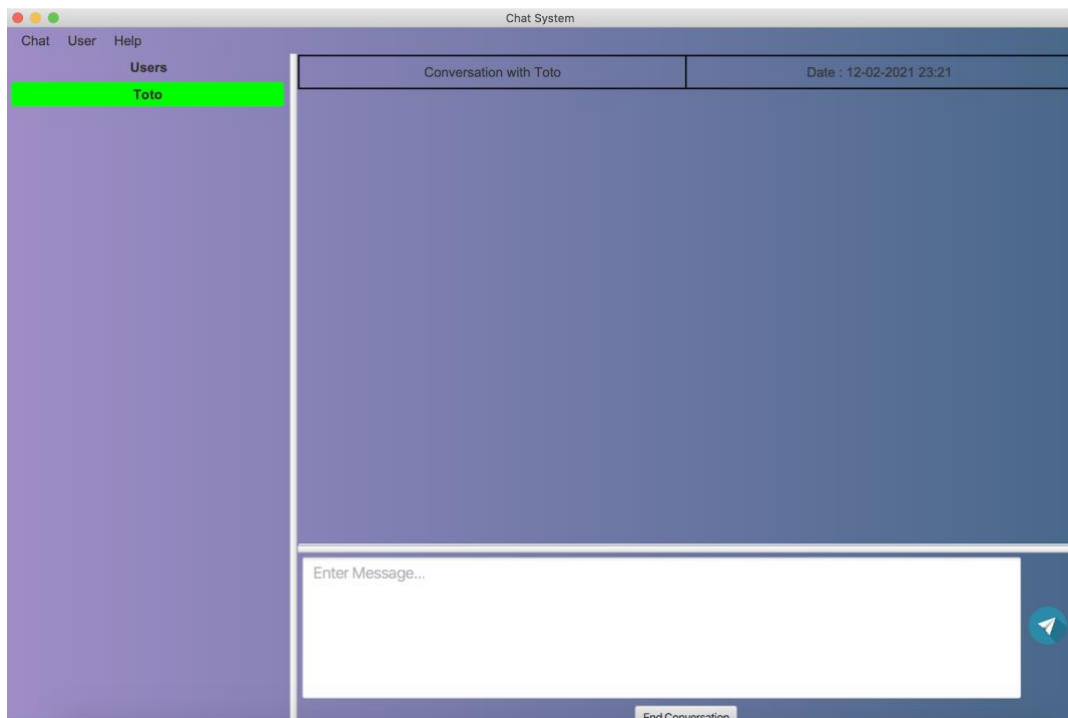
II. User Manual

When launching the application, the user is asked to choose a username, a status and if they are either using the application internally (Internal) or through the server (External).



A screenshot of a web application window titled "Chat System". The background is a gradient of blue and purple. In the center, there is a login form. It starts with a label "Username:" followed by a text input field containing the text "Toto". Below this is a row of two radio buttons: "Intern" (which is selected) and "Extern". Underneath the radio buttons is a label "Status:" followed by a dropdown menu currently showing "Online". At the bottom of the form is a "Connect" button.

After clicking on the connect button they are redirected to the main page.



A screenshot of the "Chat System" main interface. The window has a title bar "Chat System" and a menu bar with "Chat", "User", and "Help". On the left is a sidebar with a "Users" section containing a list with "Toto" highlighted in green. The main area is divided into two parts: a top header bar showing "Conversation with Toto" and "Date : 12-02-2021 23:21", and a large chat area below it. At the bottom of the chat area is a text input field labeled "Enter Message..." and a blue circular button with a white paper plane icon. Below the input field is an "End Conversation" button.

On the left side we have a column with every connected user. Whenever a new user connects the network, their name will automatically be added, and it will be removed when they leave. To start a conversation, you simply need to click on their name. All that is left to do is type any message and click on the send button right next to the text field. If you want to end the conversation, simply click on the end conversation button below the text field.

The menu bar was three columns:

- By clicking on Chat and Quit, you can't close the application properly.
- By clicking on User, you can either change your username or status.
- By clicking on Help and about a small pop up appears with our names and the technologies we used to build the interface.

III. Notable choices we made during the project

A) JavaFX and Scene builder for the UI






For the user interface, we chose to use JavaFx and Scene Builder. We made that choice because we were short in time for the project and we were looking for a solution more intuitive and definitely faster than starting the whole interface from scratch with swing. We started to think about the UI once all the testing for the internal users were made. So, it is not as intuitive as we wanted it to be. Scene Builder was faster and much easier to design the interface. Although, it has been a little bit difficult to understand how it worked but with the correct tutorial and then our own experience it definitely was the best choice to make. As for JavaFx we did not have the choice as it was use to handle Scene Builder. Its syntax was close to swing.

We organized the UI between 2 main screens : the connexion screen and the chat screen both respectively handled by ConnexionController and ChatController classes.

B) Databases

For the databases, we were advised by the teacher to use Sqlite as we wanted to use a local database. This local DB would be placed on each computer running the application. There are 3 tables in our database : Users, ActiveUsers and Messages. Users was supposed to be used to store all users that have ever used the applications but at the end we never used this table. We preferred to use ActiveUsers as it was more relevant and it inherited from Users. This table is used to store all active users that have the application open. Both Users and ActiveUsers are described as below :





We made the assumption that each user will not change its computer and keep the same IP address (whether they are using the application through the server or on the local network).

Table name: <input type="text" value="ActiveUsers"/> <input type="checkbox"/> WITHOUT ROWID									
	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Default value
1	ip	VARCHAR							NULL
2	username	VARCHAR							NULL
3	status	VARCHAR							Online

- ip: It is the user's IP store as a string and the primary key of the table. It must be unique and not null as we use the IP to fetch lines in the database.
- username: It is the user's username that must be unique and not null. This way, we guarantee that each user has a unique username. When we first open the application, our default username is User/ip. As ip addresses are unique the username at each first connection is unique too.
- status: It is the user's status. In our code, we specified 3 values only : « Online », « Do Not Disturb » and « Offline ». In ActiveUsers, it is automatically defined as « Online » while it is « Offline » in Users.

We implemented listeners in this class to send a notification to our ChatController when there is a change. We notify the interface when adding a new user, deleting a user, the user's status change, a user change username. It sends the user's ip, and the user's username it all notify except at the last case. There, it sends the old username and the new one.

We also use this database to store every message the user sends or receives in the table Messages.

Table name: Messages <input type="checkbox"/> WITHOUT ROWID									
	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Default value
1	peer	STRING							NULL
2	isSender	BOOLEAN							NULL
3	text	VARCHAR (640)							NULL
4	timestamp	DATETIME							datetime(CURRENT_TIMESTAMP,'localtime')

- peer: It is an IP address of the person we are talking to.
- isSender: It is a boolean that is true if we sent the message or else otherwise. We mostly use it to display the message correctly.
- text: It is the content of the message store as a string with a limit of 640 characters.
- timestamp: It is a Datetime use to keep the timestamp of each message. In our code we only use its default value as described above.

In addition to store each message, we also use this table to get the message history of a conversation using this query : "SELECT text, timestamp, isSender FROM Messages WHERE peer=?; ». Then, each field is stored as an ArrayList. We return an ArrayList of the 3 ArrayLists previously created. This method is called by the ChatController and allows the user to see the complete message history of a conversation and when each message was sent.

Tests

We did not make JUnits tests but instead we tested each database class within the main method. Our tests consisted of inserting new rows in the tables. Then, inserting them twice in the user database to make sure that duplicates entries are ignored. Finally, we tried to update relevant fields. The changeStatus method is coded so that it only accepts a few options and the rest is ignored.

C) Connexion protocols

When communicated on the local network, we decided to use both TCP and UDP.

We chose TCP to send the packets carrying the message because it provides a more reliable service. We chose a fixed port number through which a user expects a new connexion. After a connexion is established, the two users can communicate through the agreed upon socket.

We chose UDP to send the packets containing information about the users (connexion, changing usernames...) because the packets are broadcasted and using UDP reduces the number of packets that are sent on the network. We chose a fixed port number through which a user expects to receive all the relevant updates from their peers.

Tests

At the beginning, we use our own computer for all the testing. We used the terminal and the netcat command to listen or send a message to our program. When we first launch the program, it sends a broadcast with the tag [1BD] for first connexion and a message we built. When the program receives a [1BD] it sends a UDP datagram to the sender as an acknowledgment along with the users personal information. To make sure we receive and send the right messages, we listen on the UDP socket using the « nc -u -l 20000 » command. Then, in order to test out that the program has the right behavior we built and sent our own messages using the terminal and the « nc -u @senderIP 20000 ».

Later on, we had access to another computer in the same local network so we were able to run the program on both computers and print our results. This way, we were sure that the broadcast within the same local network worked correctly.

D) Server implementation

We created a Web application that runs on INSA's Tomcat Server (we called it ChatSystemWebSocket). We chose not to use Java HTTP Servlets because we had the impression that it would be difficult to have a proxy that uses HTTP Servlets, our Web application when transferring messages from one user to another would have to play the role of a server and client. We instead opted for websockets which allows full duplex communications in a more convenient manner.

Server side

In our WebSocket endpoint we used a hashtable to memorise connected users and all the relevant information needed to contact them (their session keys and sessions) and used a MySQL database to keep track of connected users with their IP address, usernames, status and session keys. Using these two data structures we can easily notify everyone about user updates and forward any incoming message to the right person. Our servers correctly distinguishes internal and external users. We did not have enough time to connect internal users to the server, but we did manage to have two external users communicate through the server using our application.

There are four main methods in our server endpoint:

- OnConnection which adds a new user to our hash table and memorises their session
- OnClose which removes any trace of the user closing its websocket connection
- OnError which deals with any possible errors that may occur
- OnMessage which is called whenever there is an incoming message

When communicating with the Websocket, we have three message formats:

The first one is the first message is of type NewUser and has this format:

[NewUser]:ip:username:status:type the type being whether they are an internal or external user. The server then responds by sending an html table with the list of connected users (if we have an internal user, it only sends a table containing the info of external users but if we have an external user it sends the info of both internal and external users). The arrival of a new user is then broadcasted to connected users following the same logic by distinguishing if they are internal or external users.

The second one if of type UpdateUser and has this format:

[UpdateUser]:ip:username:status:type. Upon receiving this the server updates its own database and broadcasts the updates. If a user has switched their status to offline, we consider that they are no longer reachable and so remove them from the database.

The last one if of type forward and has this format

[Forward]:SourceIP:DestinationIP:message. The server then transfers the message to the appropriate user.

Client side

On the client side, we had to implement a client endpoint that initiates the connection with the server. Once connected, it receives the messages sent from the server through the OnMessage method and sends info to the server using sendMessage.

Other than that, not much changes on the client side the incoming messages are treated the same way and the outgoing messages just have to follow the formatted dictated by the server.

In order to test our code, we used a small html page which connected to our WebApplication that we hosted locally and used a textbox in which we could write the messages we wanted to send. We tested out all the different scenarios and everything worked well.

We didn't have time to have internal users communicate with the server. In order to use the server, you have to be an external user. External Users can communicate with each other through the server.

Tests:

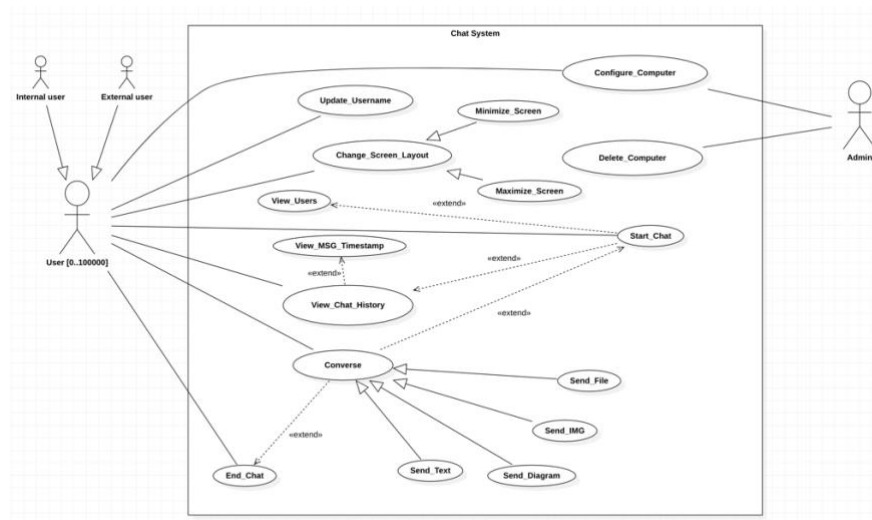
In order to test our code, we used a small html page which connected to our WebApplication that we hosted locally and used a textbox in which we could write the messages we wanted to send. We tested out all the different scenarios and everything worked well.

IV. UML Diagrams

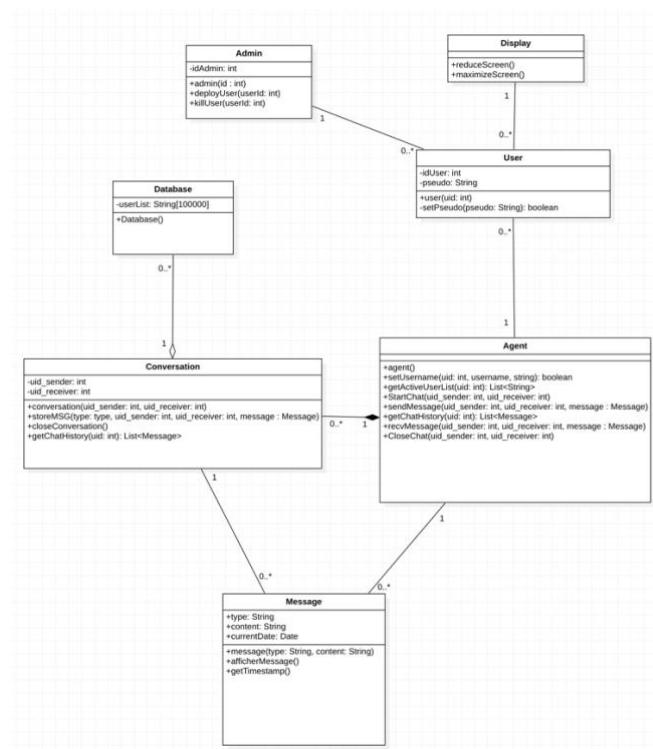
Our UMLs can be found in the COO folder of our git but please not that some of them appear to be empty when you open them with StarUML which is why we also added screenshots to the folder.

A) Initial UML diagrams we had designed

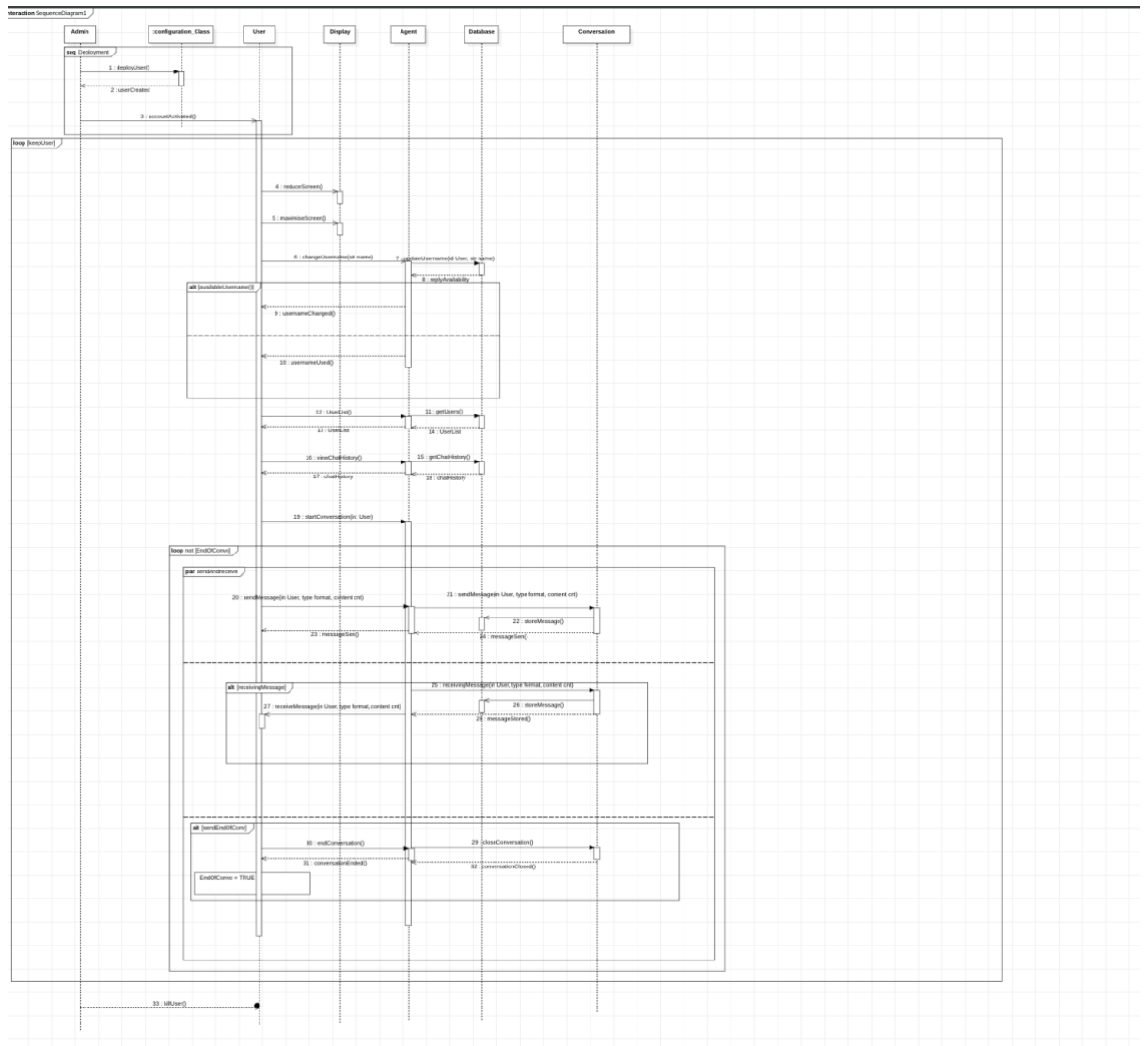
Use Case Diagram



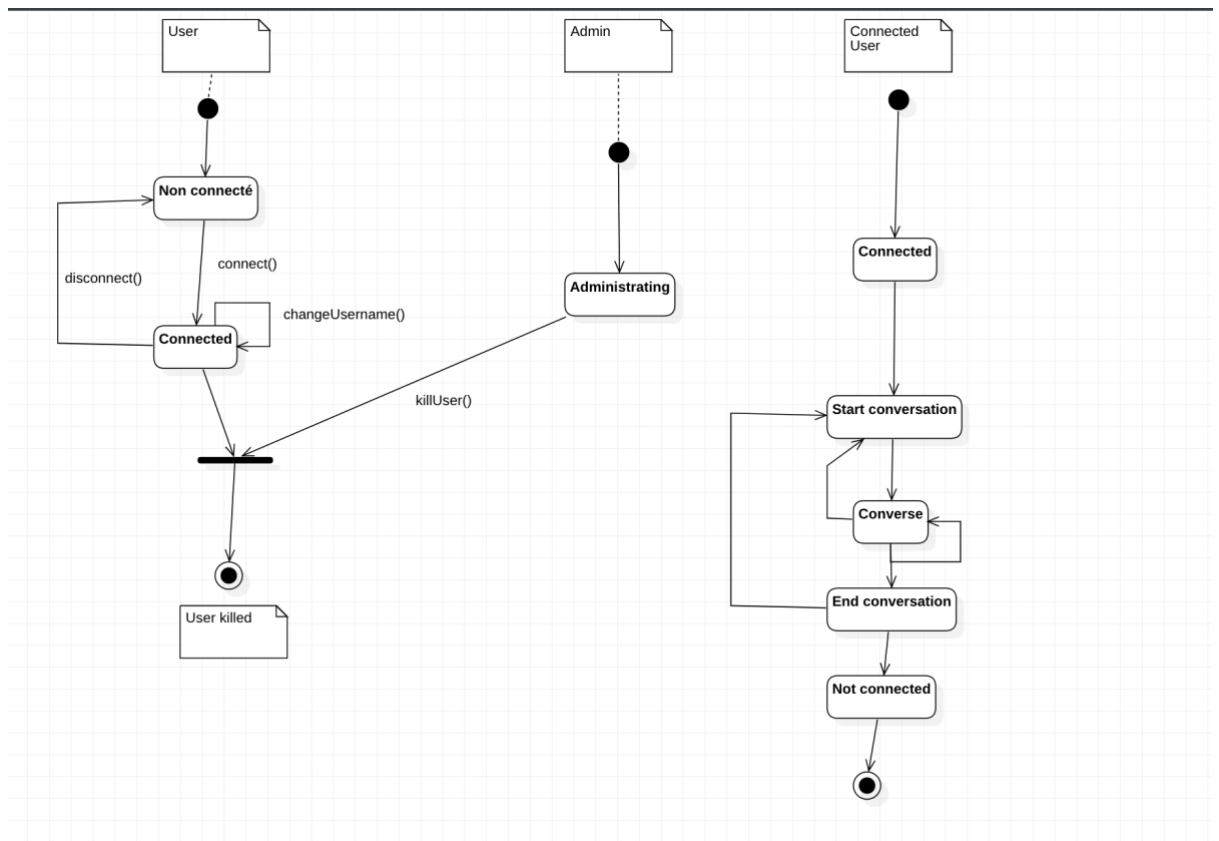
Class Diagram



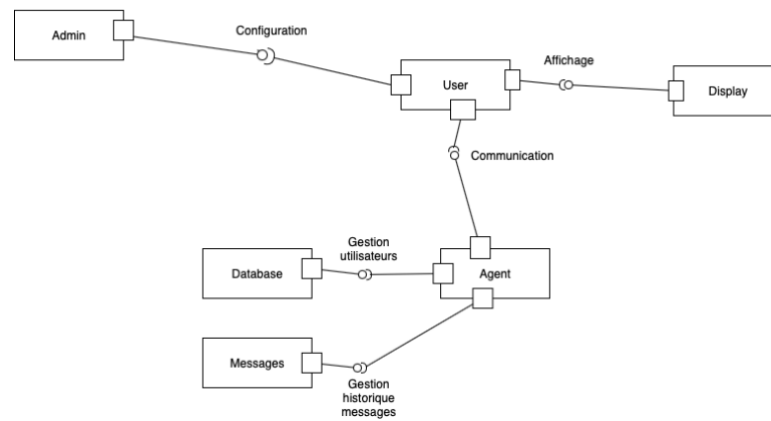
Sequence Diagram



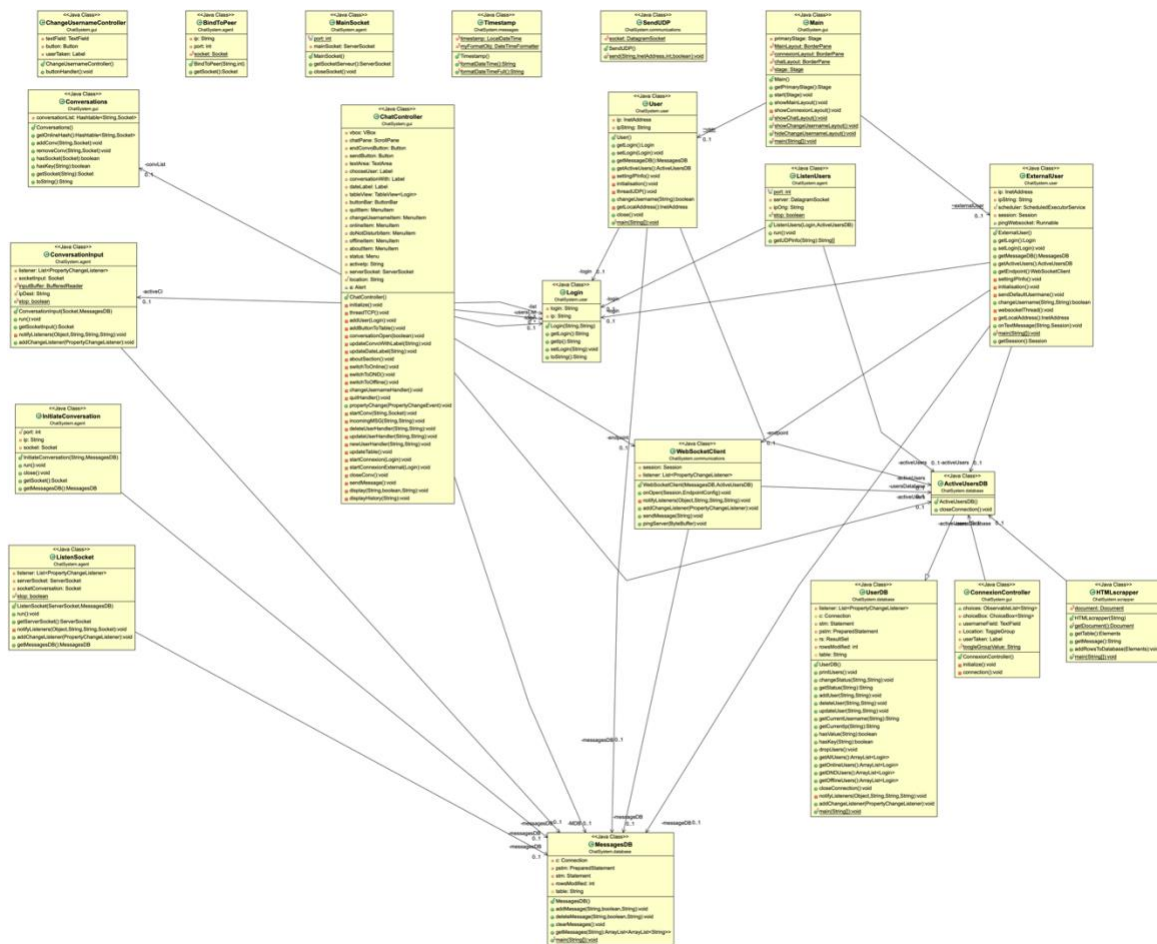
Statechart



Composite Structure Diagram



B) Class Diagram generated from our Java Project



Conclusion

It was an interesting project and given the circumstances, we are satisfied with our work.

The project took up a lot of our time outside of the dedicated lab works but we still enjoyed it.

There are still

INSA Toulouse

135, avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr



MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE