INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Rapport Projet Chat System

Programmation Orientée Objet

28 janvier 2022

Etudiants :

BACCAR Rostom 4IR B1

FERCHICHI Wissem 4IR B1

Enseignant :

YANGUI Sami

RAPPORT PROJET CHAT SYSTEM

BACCAR Rostom 4IR B1
FERCHICHI Wissem 4IR B1

PLAN:

Liens utiles :

I. **JIRA :** https://chatsystem.atlassian.net/jira/software/projects/GDP/boards/1/backlog
II. **Git :** https://github.com/rostom-baccar/INSA-POO-Project-Chat-Sytem-4IR
III. **Vidéo de démonstration :** https://youtu.be/O3YKJvjBTYE

Remarques :

- *Des captures d'écran sont disponibles dans la section JIRA concernant cette étape*

Conception

Voir le contenu du dossier Conception de notre dépôt Git.

Installation et Déploiement

Remarque : le readme disponible sur notre dépôt Git propose une version simplifiée de ce paragraphe.

Notre application fonctionne comme ceci : il y a un serveur qui gère tous les clients. Ce serveur devra être déployé sur une seule machine. Les clients quant à eux doivent être déployés sur différentes machines (la machine où est déployé le serveur peut aussi faire office de client).
Il faut donc dans un premier temps lancer le serveur sur une machine puis créer autant de clients qu'on le souhaite.

I. Utilisation du fichier JAR (recommandé)

2 fichiers JAR sont fournis dans le dossier Executables de notre dépôt Git : un fichier **Client.jar** et un fichier **Server.jar.**
Voici les étapes à suivre :

Sous Windows :

- Ouvrir le fichier Server.jar sur la machine serveur
- Ouvrir le fichier Client.jar dans le(s) machine(s) client

Sous Linux :

Sur la machine serveur :

- Ouvrir un terminal dans le dossier qui contient le fichier Server.jar
- Taper la commande suivante : ***java -jar Server.jar***

Sur chacune des machines client :

- Ouvrir un terminal dans le dossier qui contient le fichier Client.jar
- Taper la commande suivante : ***java -jar Client.jar***

II. Déploiement manuel

En cas de problème avec les fichiers JAR fournis, le test de l'application peut simplement se faire en clonant notre dépôt Git et en ouvrant le programme avec un IDE.

Voici les étapes à suivre :

- Exécuter la classe Server sur la machine serveur
- Exécuter la classe Client sur le(s) machine(s) client
-

Remarque importante : à l'ouverture du fichier Client.jar (ou à l'exécution de la classe Client), on demande au client de renseigner l'adresse IP de la machine serveur vu que notre application est basée sur un seul serveur. Dans le cas réel d'une entreprise, cette étape n'est pas nécessaire puisque l'adresse IP de la machine serveur est déjà connue et peut être directement renseignée dans le code. Mais vu qu'on ne connait pas l'adresse IP de la machine serveur sur laquelle notre application sera testée, nous avons opté pour une telle solution

Manuel d'utilisation

I. Description des fonctionnalités développées

1. Coté Serveur

- Arrêt du serveur à n'importe quel moment

2. Côté Client

- Vérification du pseudonyme avant la connexion :

Avant la connexion, on demande à l'utilisateur local de renter un pseudonyme. Celui-ci pourra se connecter à condition que le pseudonyme rentré soit unique : il ne faut pas qu'un utilisateur distant déjà connecté ait le même pseudonyme.

- Affichage dynamique des personnes connectées sur le réseau :

L'utilisateur local a accès à une liste d'utilisateurs distants connectées au réseau. Cette liste est dynamique : si un utilisateur distant se connecte ou se déconnecte, la liste se met à jour automatiquement sans qu'aucune action de la part de l'utilisateur local ne soit requise.

- Changement de pseudonyme en étant connecté et vérification du nouveau pseudonyme :

Une fois connecté avec un pseudonyme unique, l'utilisateur a la possibilité de changer de pseudonyme. Celui-ci doit aussi être unique.

- Group Chat avec toutes les personnes connectées sur le réseau :

L'utilisateur local a la possibilité de communiquer avec l'ensemble des utilisateurs distants connectés à travers un Group Chat qui s'affiche une fois que l'utilisateur se connecte avec un pseudonyme unique.

- Private Chats avec une ou plusieurs personnes en parallèle :

L'utilisateur local a la possibilité de choisir un utilisateur distant connecté dans la liste pour commencer un Private Chat avec lui. L'utilisateur local a la possibilité de discuter avec plusieurs utilisateurs distants en parallèle. Pour cela, il faudra démarrer plusieurs Private Chats avec les personnes souhaitées.

- Réception de notifications :

La réception de notifications se fait à travers des Dialog Box. Celles-ci permettent d'alerter l'utilisateur local dans les cas suivants :

* Lorsque celui-ci saisit un pseudonyme qui n'est pas unique (avant et après connexion)

* Lorsqu'un utilisateur distant se connecte

* Lorsqu'un utilisateur distant se déconnecte

* Lorsqu'un utilisateur distant change de pseudonyme

* Lorsqu'un utilisateur A veut discuter avec un utilisateur B. L'utilisateur B recevra une notification lui indiquant que l'utilisateur A souhaite discuter avec lui. Une fois que l'utilisateur B a lui aussi ouvert une session privée avec l'utilisateur A, l'utilisateur A reçoit une notification lui indiquant que l'utilisateur B a ouvert la fenêtre de chat privée avec lui.

- Rechargement de toutes les fenêtres Private Chats avec le nouveau pseudonyme lorsqu'une personne change son pseudonyme :

Pour l'utilisateur qui a changé de pseudonyme : tous les Private Chats qu'il a démarrés vont se mettre à jour automatiquement avec son nouveau pseudonyme.

Pour l'utilisateur qui a ouvert un Private Chat avec la personne qui a changé de pseudonyme : le Private Chat en question se mettra à jour automatiquement avec le nouveau pseudonyme de la personne qui a changé de pseudonyme.

- Déconnexion à n'importe quel moment :

L'utilisateur a la possibilité de se déconnecter à n'importe quel moment. La déconnexion peut se faire à travers un bouton ou en fermant simplement la fenêtre principale.

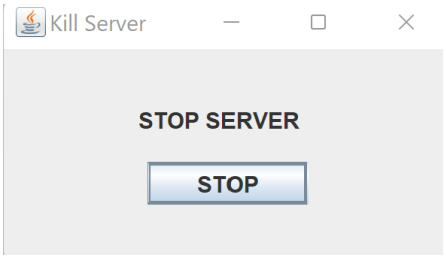
- Tous les Private Chats ouverts affichent l'historique des messages échangés avec la personne en question :
- Tous les messages envoyés et/ou reçus dans le Group Chat ou dans les différents Private Chats sont horodatés.

II. Explication de l'interface utilisateur

1. Coté Serveur

- Kill Server Window

Cette fenêtre est la seule fenêtre affichée coté serveur. Elle permet au clic du bouton « STOP » d'arrêter le serveur.

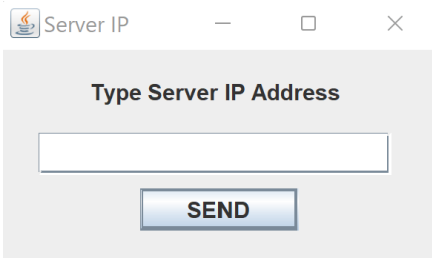


2. Coté Client

Remarque : l'ordre suivant des fenêtres est l'ordre chronologique de leur ouverture pour l'utilisateur local.

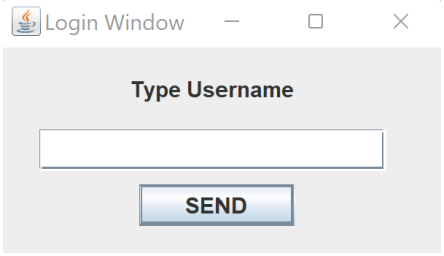
- Server IP Address Window

Cette fenêtre est la première fenêtre qui s'affiche pour l'utilisateur local. Elle permet la saisie de l'adresse IP de la machine serveur.

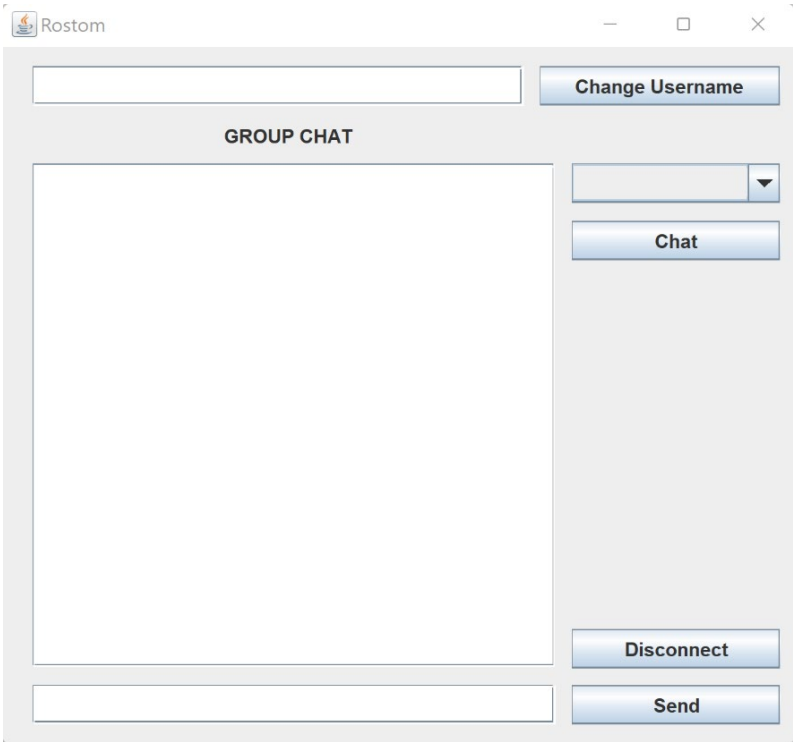


- Login Window

Cette fenêtre permet simplement la saisie du pseudonyme. Si celui-ci est valide, l'utilisateur se connecte avec succès. Sinon, la saisie du pseudonyme est demandée à nouveau.



- Main Window



Cette fenêtre est la fenêtre principale qui permet à l'utilisateur d'utiliser toutes les fonctionnalités développées :

- Le bouton « Change Username » permet de changer de pseudonyme

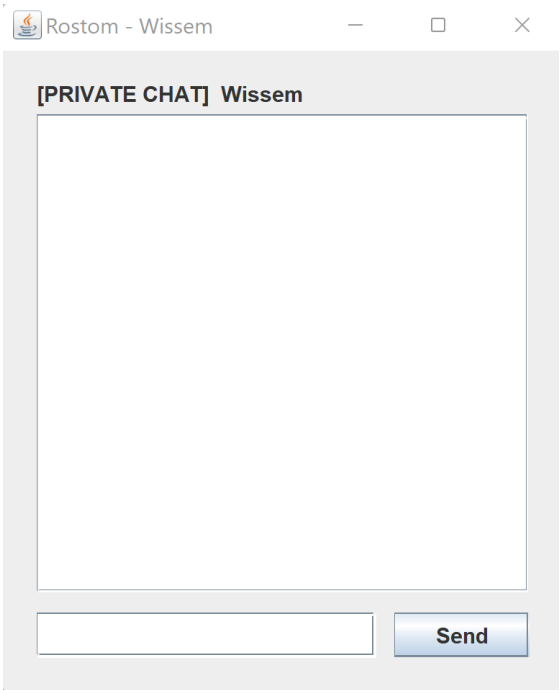
- La liste en haut du bouton « Chat » affiche tous les utilisateurs distants connectés sur le réseau

- Le bouton « Send » permet d'envoyer un message dans le Group Chat

- Le bouton « Chat » permet de démarrer un Private Chat avec l'utilisateur distant choisi dans la liste

- Le bouton « Disconnect » permet de se déconnecter

- Chat Window

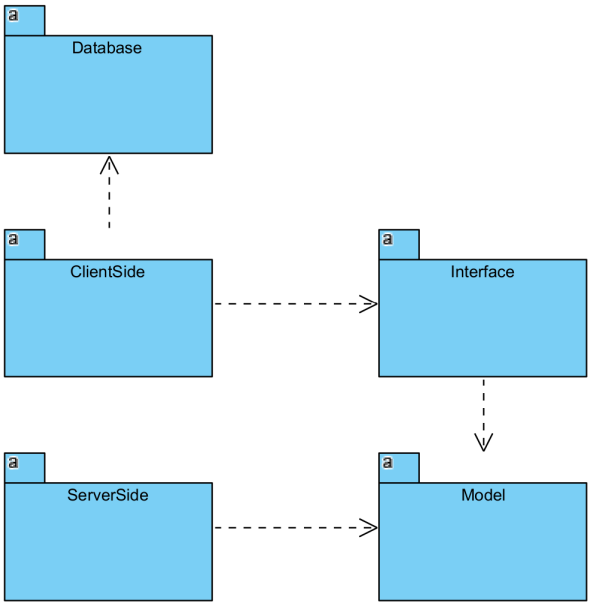


Cette fenêtre s'affiche après que l'utilisateur local a cliqué sur le bouton « Chat » qui permet de commencer un Private Chat avec un utilisateur distant.

Dans l'exemple de la figure de droite, l'utilisateur local Rostom a ouvert un Private Chat avec l'utilisateur distant Wissem.

Explication du programme

I. Explication des packages et des classes



Le programme est divisé en 5 packages différents

*Remarque : pour une meilleure lisibilité, les packages sont **en orange** et les classes qu'elles contiennent sont **en jaune***

1. Model

Ce package contient toutes les classes purement « Objet » qui nous ont aidé dans la modélisation de certaines entités dans notre code. Le package contient 4 classes :

- **RemoteUser**

RemoteUser est une classe modélisant un utilisateur distant par un pseudonyme et une adresse IP qui est l'adresse locale de la machine de l'utilisateur distant.

- **Message**

Message est une classe modélisant les messages que les clients et le serveur s'envoient. Cette classe a permis de rendre le code très optimal. En effet, une version antérieure de l'application ne permettait que l'envoi d'objets de type String entre le client et le serveur. Vu les nombreux types de requêtes qui changent selon la fonctionnalité utilisée, nous avons opté pour un système d'échange d'objets de type Message.

L'objet Message contient les champs type, content, argument1, argument2 et argument3 et des méthodes pour créer des messages selon les attributs spécifiés.

- **MessageType**

MessageType est une simple classe Enum qui énumère plusieurs types de messages, le type de message étant un des champs de la classe Message

- **LocalIPAddress**

LocalIPAddress est une classe qui ne modélise pas d'objet spécifique. La classe contient simplement une méthode qui évite que la machine retourne l'adresse de loopback 127.0.0.1 quand on veut récupérer son adresse locale dans le réseau. C'est un problème auquel nous avons en effet été confronté lors de nos tests du programme sur les machines de l'INSA. Son emplacement dans le package Model est simplement dû au fait qu'elle est utilisée par plusieurs classes et qu'elle n'appartenait pas à une classe en particulier pour qu'on la définisse simplement comme une méthode de cette classe.

2. ClientSide

Ce package contient toutes les classes qui concernent l'utilisateur local : il contient la classe Client et la classe ServerResponseListener qui est un thread créé par la classe Client.

- **Client**

La classe Client est responsable de l'affichage des 2 fenêtres LoginWindow et MainWindow, de la création de la base de données et de la création du thread ServerResponseListener. Cette classe termine son exécution après avoir créé la fenêtre MainWindow.

- **ServerResponseListener**

La classe ServerReponseListener est un thread créé par la classe Client qui gère l'écoute des messages provenant du serveur. A la réception des messages, il peut effectuer des tâches différentes selon le type de message reçu.

3. Interface

Ce package contient 5 classes responsables de l'affichage des fenêtres : KillServerWindow, ServerIPAddressWindow, LoginWindow, MainWindow et ChatWindow.

Côté Serveur :

- **KillServerWindow**

KillServerWindow est la classe qui crée la fenêtre à travers laquelle le serveur peut être arrêté. C'est la seule fenêtre affichée côté serveur.

Côté Client :

- **ServerIPAddressWindow**

ServerIPAddressWindow est la classe qui crée la fenêtre de saisie de l'adresse de la machine serveur.

- **LoginWindow**

LoginWindow est créée par la classe Client. Elle permet simplement la saisie du pseudonyme de l'utilisateur. Elle envoie ensuite cette requête au serveur.

- **MainWindow**

MainWindow est aussi une classe créée par la classe Client une fois que l'étape de connexion est franchie. Elle permet d'envoyer toutes sortes de requêtes au serveur en utilisant les différents boutons, chacun étant responsable d'une certaine fonctionnalité.

- **ChatWindow**

ChatWindow est la classe qui crée la fenêtre qui s'affiche lorsque l'utilisateur local souhaite démarrer un Private Chat avec un utilisateur distant. Elle est créée par la classe MainWindow à l'appui du bouton « Chat ». Elle permet l'envoi d'un seul type de requête au serveur : les Private Chats.

4. ServerSide

Ce package contient toutes les classes qui concernent le serveur de l'application. Il contient la classe Server et les 2 classes ClientHandler et UsernameHandler qui sont 2 threads créés par la classe Server

- **Server**

Cette classe gère le serveur de l'application. Elle établit la connexion socket avec tous les clients voulant se connecter à l'aide d'une boucle while infinie. A chaque fois que le serveur accepte la connexion socket d'un client, le serveur lui crée 2 threads distincts : ClientHandler et UsernameHandler.

- **ClientHandler**

Cette classe est en quelque sorte le porte-parole du client. Elle représente une sorte d'instance de serveur pour le client. Chaque client a son propre ClientHandler. Le rôle du ClientHandler est de réceptionner les requêtes que le client lui envoie à travers les 3 fenêtres LoginWindow, MainWindow et ChatWindow. Après réception, et selon le type de requête, le ClientHandler peut faire plusieurs choses. Par exemple, s'il s'agit d'un Private Chat, le ClientHandler va identifier la personne voulue et lui envoyer le Private Chat. S'il s'agit d'un message écrit dans le Group Chat, il va broadcaster le message en question à tous les utilisateurs connectés.

Le ClientHandler a en effet accès à la liste de tous les utilisateurs connectés, qui est une liste détenue par la classe Server vu qu'elle gère la connexion de tous les clients. Mais toutes les connexions socket ne sont pas forcément ajoutées à la liste des personnes connectées. C'est en effet le rôle de la classe UsernameHandler qui est la 3ème classe de ce package

- **UsernameHandler**

UsernameHandler est un thread créé par la classe Server, à la suite de la création du thread ClientHandler. Lorsque le client saisit un pseudonyme, la requête est réceptionnée par le ClientHandler du client qui décide de la validité du pseudonyme. Le rôle du UsernameHandler est d'attendre simplement que le pseudonyme choisi soit valide pour ajouter le client à la liste des clients connectés.

5. Database

Ce package contient la classe responsable de la base de données de l'application.

- **Database**

Cette classe gère tout ce qui est lié à la base de données : établir la connexion avec la BDD (il s'agit dans notre cas de la BDD de l'INSA), créer la table, insérer une ligne dans la table, charger l'historique à l'ouverture d'un ChatWindow et enfin supprimer la table créée.

II. Déroulement du programme

Remarques :

- *Le programme dispose d'un serveur unique qui gère tous les clients. Ce choix est venu du fait qu'une seule entité qui gère tous les clients était idéal pour avoir accès à tous les logs dans un seul et même endroit. Toutes les requêtes sont gérées par le serveur qui détient aussi la liste de tous les utilisateurs connectés. En cas de problème, le debug est beaucoup plus facile.*
- *L'utilité des threads est venue en s'apercevant qu'il y avait certains bouts de code qu'il fallait exécuter en boucle (boucle infinie). Dans d'autres cas, il s'agissait de fonctions bloquantes qui n'exécutaient rien à moins de recevoir une réponse (comme la variable servant à écouter les messages venant du serveur). Dans ces cas-là, on ne pouvait pas exécuter ces bouts de code alors que la classe en question a encore du code à exécuter. La solution pour éviter de bloquer le programme est donc de lancer des threads qui vont se charger d'exécuter ces bouts de code bloquants en parallèle.*
- *Cette partie du rapport est bien illustrée à travers les diagrammes de séquences. Mais une explication exhaustive est toujours bénéfique.*

1. **Partie établissement de connexion socket et création de threads**

Le programme commence par le lancement du serveur sur la machine serveur. La fenêtre KillServerWindow apparaît côté serveur si on désire à un moment arrêter le serveur. Le serveur est en état d'acceptation de connexions socket en continue.

Ensuite, un client est créé sur la machine client. Sa création entraîne automatiquement l'acceptation de sa demande de connexion socket par le serveur. A ce stade, la classe Server lance deux threads liés au client créé : un ClientHandler et un UsernameHandler. Le ClientHandler va gérer le client créé en écoutant ses messages et en y répondant.

En parallèle, la classe Client crée un thread ServerResponseListener qui va écouter les messages provenant du serveur et agir selon la nature de ces messages.

2. **Partie Login**

Après avoir saisi l'adresse IP de la machine serveur à travers la fenêtre ServerIPAddressWindow, la fenêtre LoginWindow s'affiche à l'écran du client. Celui-ci saisit un pseudonyme. Le LoginWindow envoie la requête de connexion au ClientHandler qui détermine si le pseudonyme rentré est unique grâce à la liste des clients connectés à laquelle il a accès.

Si le pseudonyme n'est pas unique, le ClientHandler renvoie simplement une notification au client. Le ServerResponseListener du client recevra cette notification et l'affichera sous forme de Dialog Box.

Si le pseudonyme est unique, le ClientHandler envoie à tous les utilisateurs connectés une notification de connexion du client en question tout en débloquant le thread UsernameHandler qui était en attente. Le UsernameHandler ajoute donc le client en question à la liste des personnes connectées. Enfin, le ClientHandler renvoie une réponse à son client lui disant qu'il est connecté. Cette réponse est gérée par le ServerResponseListener du client. A sa réception, le ServerResponseListener débloque la classe Client pour qu'elle passe à la suite : l'affichage du MainWindow. Enfin, la classe Client se connecte à la base de données et crée la table msg_history si elle n'a pas été déjà créée.

3. **Partie Principale**

A ce stade, le MainWindow s'affiche au client.
Cette partie n'est pas très différente de la partie précédente, à la différence de certains aspects :

- Le thread UsernameHandler ne sera plus utile puisque le client est à présent connecté.
- Le client peut maintenant envoyer tous types de requêtes à son ClientHandler :
    - Envoi d'un message sur le Group Chat
    - Changement de pseudonyme
    - Ouverture d'un Private Chat avec un utilisateur distant sélectionné dans la liste
    - Déconnexion

Le programme fonctionne donc toujours de la même façon : le client envoie ses requêtes à travers les fenêtres MainWindow et ChatWindow (à l'appui des boutons). Ces requêtes sont gérées par son ClientHandler. Le ClientHandler agit différemment selon la requête reçue, et peut répondre ou non à son client selon la requête reçue. Si le ClientHandler répond au client, c'est le ServerResponseListener du client qui réceptionnera cette réponse et agira en fonction de la nature de cette réponse.

Une des requêtes que le client peut envoyer est l'ouverture d'un Private Chat avec un utilisateur distant sélectionné dans la liste des personnes connectées. Dans ce cas, une fenêtre ChatWindow apparaît contenant l'historique des anciens messages échangés avec l'utilisateur sélectionné s'il y en a. Le client peut échanger à travers cette fenêtre des messages privées avec cet utilisateur. Il est à noter que la connexion entre clients n'est pas directement établie. Pour les Private Chats, comme pour toutes les autres fonctionnalités développées, toutes les requêtes du client sont envoyées au ClientHandler qui, dans le cas d'un Private Chat, redirige simplement le message à l'utilisateur distant selectionné.

ARCHITECHTURES UTILISEES

I. Interface

Nous avons utilisé JAVA Swing pour le développement de notre interface. Ce choix est dû au fait quel les cours et les travaux dirigés en JAVA Swing nous ont permis de nous habituer à cette bibliothèque graphique. De plus, nous avions pu générer notre code automatiquement grâce au plugin Eclipse Window Builder, ce qui nous a beaucoup facilité la tâche.

Voici quelques fonctionnalités que nous avons pu intégrer en utilisant cette bibliothèque :

- Les Text Fields se réinitialisent au clic du bouton auquel ils sont associés.
- Les Text Area ne sont pas modifiables. Elles ne servent qu'à afficher les messages.
- Le bouton par défaut des fenêtres ServerIPAddressWindow, LoginWindow, MainWindow et ChatWindow est le bouton « Send » qui est donc activé par le bouton Enter du clavier.
- Mise en place d'un système de « vu » à travers les Dialog Box indiquant que la personne avec qui on désire discuter a ouvert un Private Chat avec nous.

II. Base de données

Notre projet nécessitait une base de données. Nous avons donc utilisé le serveur MySQL déployé à l'INSA. Notre base de données permet de stocker les historiques des échanges entre les différents utilisateurs.

Le cahier des charges spécifie que chaque conversation entre utilisateurs doit être conservée et affichée à chaque nouvelle session de clavardage. Nous avons créé une table nommée « msg_history », et à chaque envoi de message nous retenons l'adresse IP de la source, l'adresse IP du destinataire, le contenu du message et la date et l'heure de l'envoi.

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	☑	☑	☐	☐	☐	☐	☑	☐	
sender	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐	☐	
receiver	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐	☐	
message	TEXT	☐	☐	☐	☐	☐	☐	☐	☐	NULL
tstamp	DATETIME	☐	☐	☐	☐	☐	☐	☐	☐	NULL
		☐	☐	☐	☐	☐	☐	☐	☐	

Pour accéder à notre base de données depuis un ordinateur INSA ou en utilisant le VPN, il suffit de taper les commandes suivantes dans un terminal :

- **mysql -h srv-bdens.insa-toulouse.fr -u tp_servlet_018 -p**

Un mot de passe sera demandé, il faudra donc rentrer le mot de passe correspondant. Ensuite, pour utiliser notre base de données, il faut taper la commande suivante :

- **USE tp_servlet_018**

Enfin pour visualiser le contenu de la table, il faut taper la commande suivante :

- **SELECT * FROM msg_history**

Voici un exemple du contenu de notre table à la date du 20 janvier 2022 :

	id	sender	receiver	message	tstamp
	1	10.1.5.75	10.1.5.91	hi	2022-01-20 22:58:33
	2	10.1.5.91	10.1.5.75	hello	2022-01-20 22:58:33
	3	10.1.5.75	10.1.5.91	what's up	2022-01-20 22:58:40
	4	10.1.5.91	10.1.5.75	good	2022-01-20 22:58:40
	5	10.1.5.75	10.1.5.91	:)	2022-01-20 22:58:48
▶	6	10.1.5.91	10.1.5.75	k	2022-01-20 22:58:48

ORGANISATION

I. JIRA

Nous avons utilisé la méthode Agile afin de nous organiser dans la réalisation du projet. En effet, le site atlassian.net nous a permis de décomposer notre travail en User Stories bien définies. Chaque User Story comporte une ou plusieurs tâches à accomplir après un certain délai à respecter.

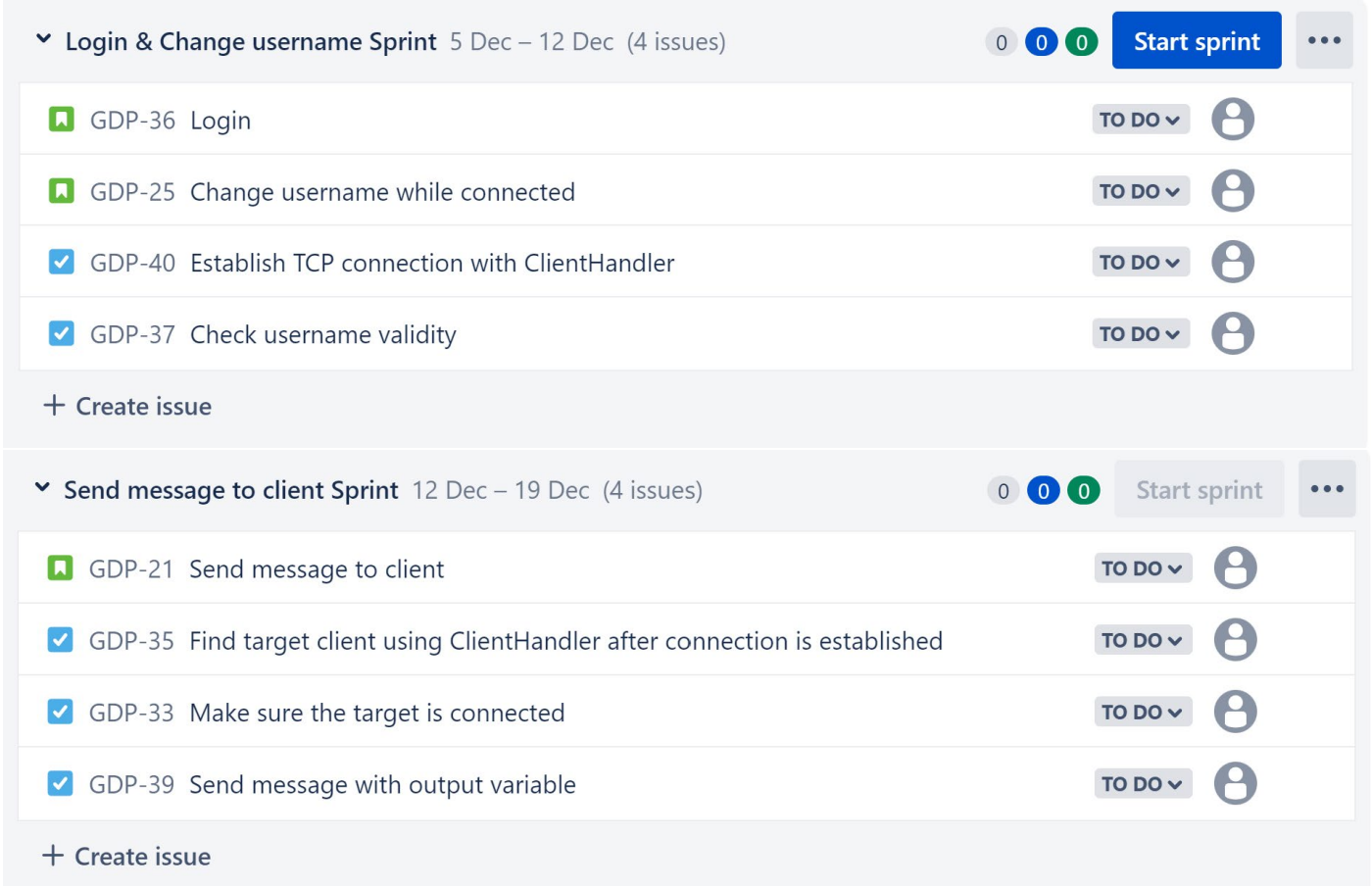
En avançant dans notre projet, nous avons définis plusieurs Speedruns que nous avons commencés et finis, souvent avec un délai non respecté vu que certaines tâches ont pris plus de temps que prévu. A la fin des Speedruns, notre backlog était vide. Nous avons donc recréé, sans les démarrer, tous les Speedruns avec toutes les User Stories et les tâches afin que celles-ci soient visibles en utilisant le lien fourni. Nous avons également laissé les anciens délais que nous nous étions fixés.

La décomposition du travail en User Stories s'est basée sur notre diagramme de cas d'utilisation en premier lieu puis a changé petit à petit au fur et à mesure que le programme évoluait.

Les User Stories expriment donc en général les fonctionnalités développées et expliquées précédemment : Private Chats, Group Chat etc. Les tâches quant à elles représentent ce que nous avions dû développer comme fonctions ou ce que nous avions dû surmonter comme problèmes afin de réaliser la User Story en question.

Le lien JIRA fourni de notre projet ChatSystem montre donc notre backlog avec la liste des User Stories et des tâches répertoriées selon différents Speedruns.

En cas de non-fonctionnement du lien fourni, voici les différents Speedruns que nous avons créés :



Broadcast Sprint 19 Dec – 26 Dec (5 issues) 0 0 0 Start sprint •••

GDP-20	Broadcast message to all clients	TO DO
GDP-24	Disconnect	TO DO
GDP-23	Show all active users	TO DO
GDP-29	Loop over all clients (send message with output variable)	TO DO
GDP-53	Design notification system for each broadcast message	TO DO

+ Create issue

Java Swing 26 Dec – 9 Jan (8 issues) 0 0 0 Start sprint •••

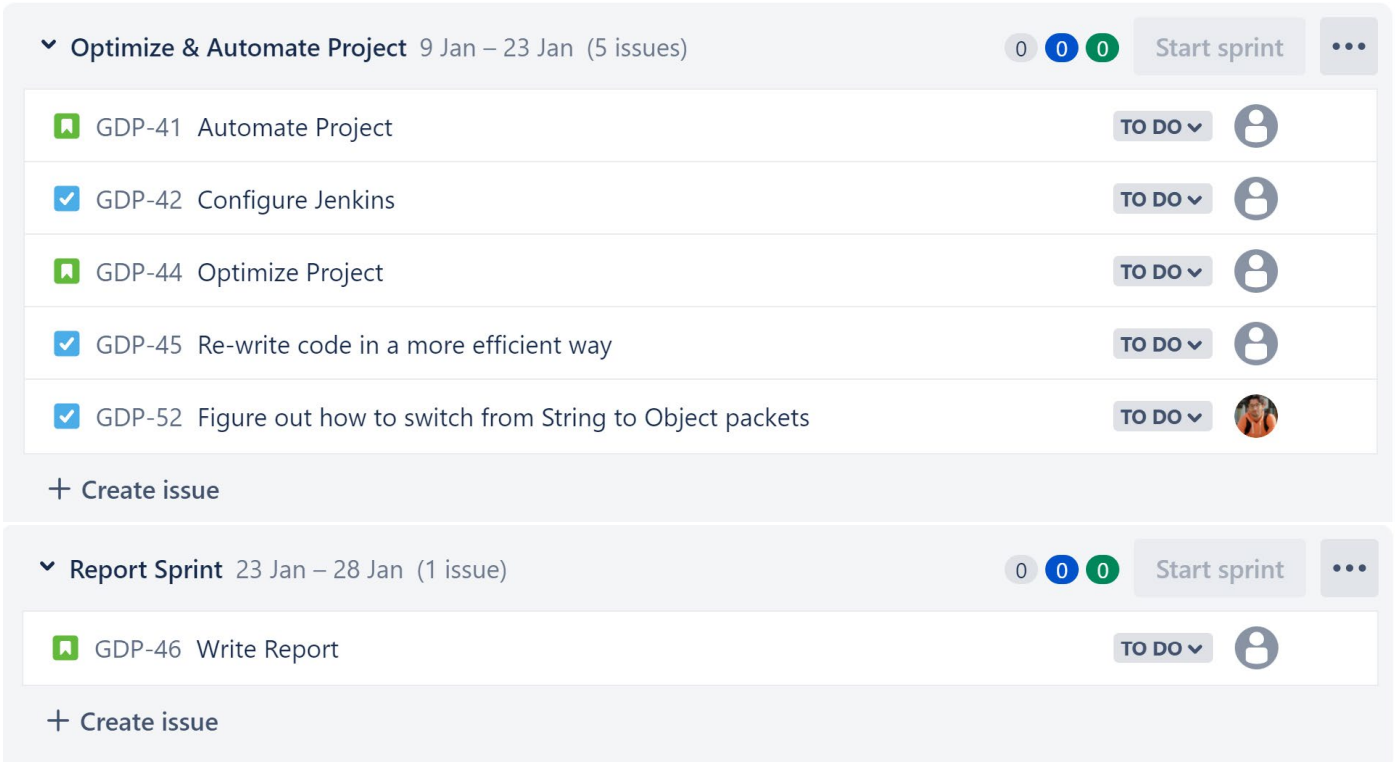
GDP-18	Design different windows	TO DO
GDP-17	Design MainWindow	TO DO
GDP-16	Design ChatWindow	TO DO
GDP-15	Design LoginWindow	TO DO
GDP-22	Link windows to requests	TO DO
GDP-31	Configure action of each button	TO DO
GDP-30	Establish connection between windows and Client class	TO DO
GDP-32	Handle opening and closing of certain windows	TO DO

+ Create issue

DataBase Sprint 26 Dec – 16 Jan (5 issues) 0 0 0 Start sprint •••

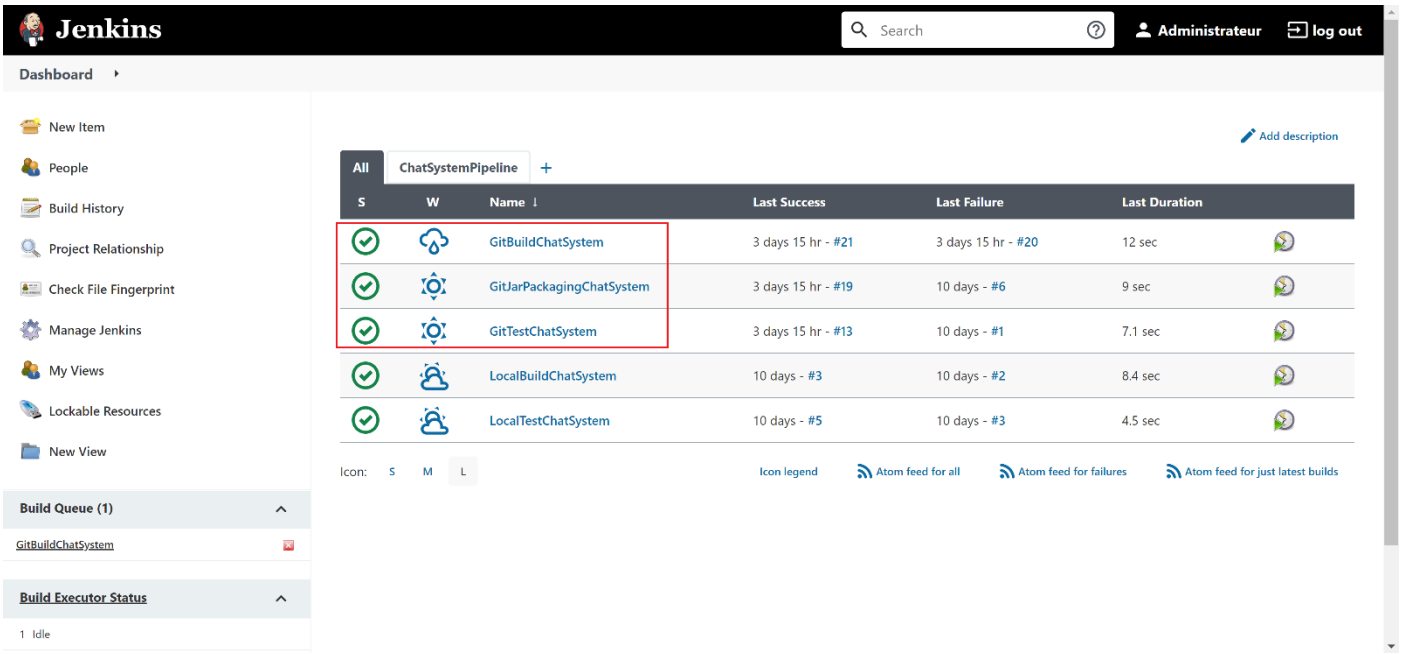
GDP-47	Create Database for Message History	TO DO
GDP-48	Link SQL to Java	TO DO
GDP-50	Design SQL tables	TO DO
GDP-49	Create functions to retrieve messages	TO DO
GDP-51	Create function that loads history and updates Private Chats	TO DO

+ Create issue



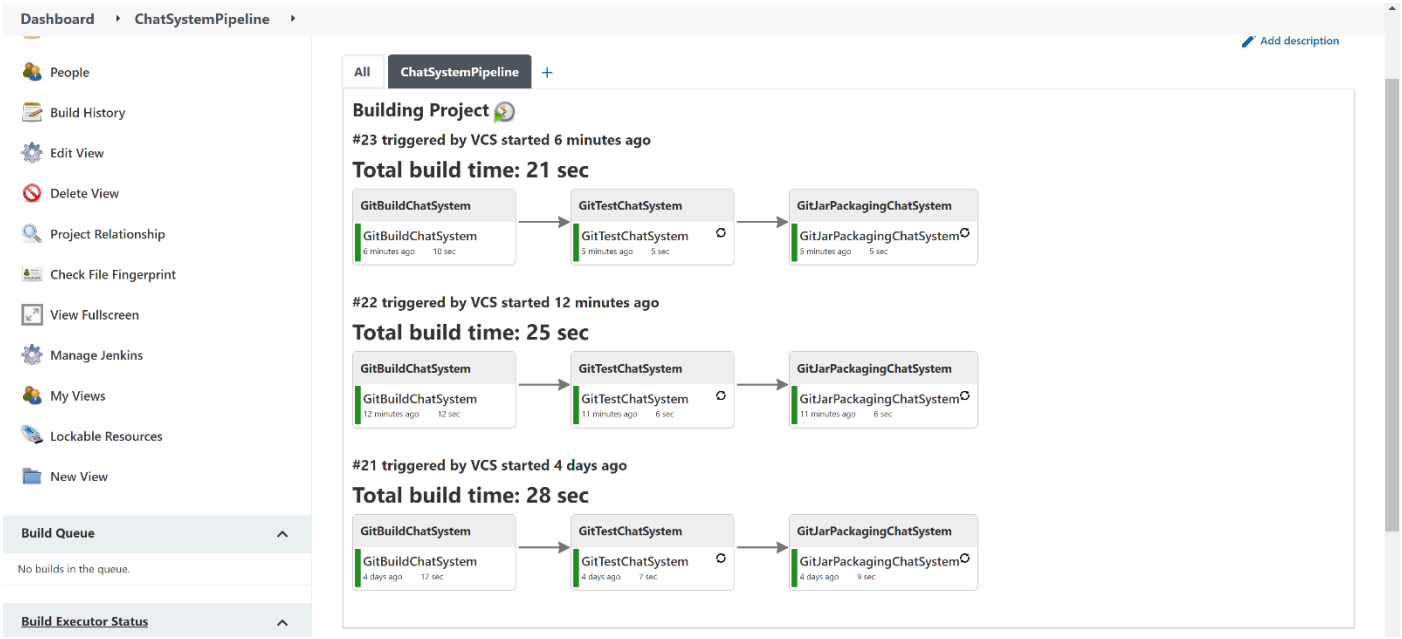
II. Jenkins

Jenkins est un outil qui nous a facilité l'automatisation de la compilation, du test et de la génération des fichiers JAR finaux.



Les Jobs encerclés en rouge sur la figure ci-dessus sont les Jobs créés qui sont déclenchés par un commit Git. Les deux autres Jobs représentent des triggers locaux à la machine. Notre pipeline principale est donc constituée des 3 jobs encerclés et l'ordre d'exécution de ces jobs est le suivant :

- Build
- Test
- JAR Packaging



Nous pouvions donc tester notre application suite au packaging JAR directement après nous être assuré que la partie compilation et test s'est bien déroulée.

Malheureusement, nous n'avons pas eu le temps de développer des tests JUnit rigoureux. Notre phase de test était donc manuelle : imaginer tous les cas possibles puis résoudre d'éventuels problèmes si ceux-ci surviennent.

Vidéo de démonstration

Nous avons réalisé une vidéo de démonstration de 3 minutes qui illustre bien les fonctionnalités développées. La vidéo a été réalisée sur les machines de l'INSA sur 3 comptes personnels différents. Nous avons utilisé la base de données de l'INSA qui nous a été attribuée.

Lien de la vidéo : https://youtu.be/O3YKJvjBTYE

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

Université Fédérale Toulouse Midi-Pyrénées
Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE