

# Rapport Projet Clavardage

Gabin Noblet, Paul Thebault

15 Février 2021

# Table des matières

<b>Introduction</b>	<b>2</b>
<b>1 Conception</b>	<b>3</b>
1.1 Conception au début du projet . . . . .	3
1.2 Programmation tout au long du projet . . . . .	4
1.3 État de la structure à la fin du projet . . . . .	4
<b>2 Manuel d'utilisation</b>	<b>5</b>
2.1 Utilisation du client . . . . .	5
2.2 Lancement du serveur de base de données . . . . .	6
2.3 Lancement du servlet . . . . .	6
<b>3 Motivations de choix</b>	<b>7</b>
3.1 Hypothèses . . . . .	7
3.2 Interface graphique . . . . .	7
3.3 Modèle utilisé . . . . .	8
3.3.1 User . . . . .	8
3.3.2 Address . . . . .	8
3.3.3 Conversation . . . . .	8
3.4 Réseau . . . . .	9
3.4.1 <i>CCP - Clavardage Control Protocol</i> . . . . .	9
3.4.2 Envoi de messages . . . . .	10
3.4.3 Envoi de fichier . . . . .	10
3.4.4 <i>ListenerPool</i> . . . . .	10
3.5 <i>Servlet</i> . . . . .	11
3.6 Base de données . . . . .	11
3.7 Fichier de configuration . . . . .	12
<b>Conclusion</b>	<b>13</b>

# Introduction

Ce rapport présente notre projet d'application de clavardage en Java. Dans un premier temps nous avons réfléchi à la conception en accord avec le cahier des charges en effectuant différents diagrammes (Diagrammes des cas d'utilisation et diagrammes de séquence). Puis dans un second temps nous avons mis en pratique et développé cette application de clavardage décentralisée.

**Environnement de développement et de test** Pour développer nous avons utilisé l'IDE IntelliJ de JetBrains. Pour ce qui est des tests de communication entre deux clients, nous avons utilisé une machine virtuelle inter-connectée avec notre OS (Windows avec VM Linux). Ensuite la base de données et le serveur *Tomcat* sont implémentés sur un *Raspberry Pi*.

# Chapitre 1

## Conception

### 1.1 Conception au début du projet

Avant de commencer le projet, nous avons commencé un travail de conception objet. Nous avons alors tenté de décrire le fonctionnement de l'application de chat, telle qu'on la voulait. De ce travail a découlé deux diagrammes de séquences, ainsi qu'un diagramme de cas d'utilisation.

Le [premier](#), décrit grossièrement les interactions entre l'utilisateur et l'application, telle qu'on les imaginait.

Le [second](#) divise l'application en plusieurs classes, et décrit les interactions entre ces classes ainsi qu'avec l'utilisateur <sup>1</sup>.

Enfin, le diagramme de cas d'utilisation décrit lui les actions que peut effectuer l'utilisateur par l'application.

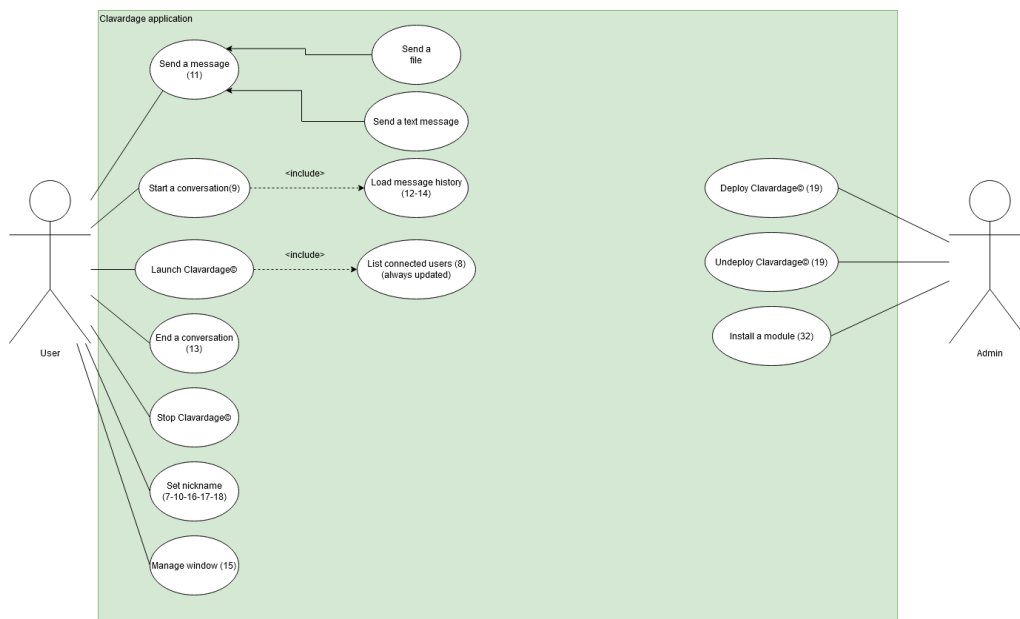


FIGURE 1.1 – Diagramme de cas d'utilisation initial

---

1. Par le biais de l'interface graphique

## 1.2 Programmation tout au long du projet

Après ce travail initial, nous nous sommes attelés à la programmation de l'application. Nous avons fait le choix de sous-diviser chaque classe du diagramme de classe détaillé, en plusieurs classes, à la fois pour éviter d'avoir des classes *boîtes noires*, et aussi pour rendre la maintenance facile, et éviter d'avoir trop de classes à rallonges.

Par conséquent, nous créons de nouvelles classes au long du projet, ajoutons de nouvelles interactions entre ces classes. Notre conception de l'application a également évolué avec notre compréhension du cahier des charges.

Malgré cela, nous nous sommes bien rendus compte que le travail de conception initial nous a fait gagner du temps, puisque qu'il nous a donné une idée claire de la topologie de l'application.

## 1.3 État de la structure à la fin du projet

Nous avons donc refait des diagrammes UML à la fin du projet, afin de décrire la structure de l'application, ainsi que les interactions entre ses différents composants. Tous ces diagrammes sont disponibles dans le [dépôt du projet sur GitHub](#).

Nous avons donc pu refaire notre diagramme de cas d'utilisation pour qu'il corresponde aux actions que peut effectuer l'utilisateur avec l'application finale.

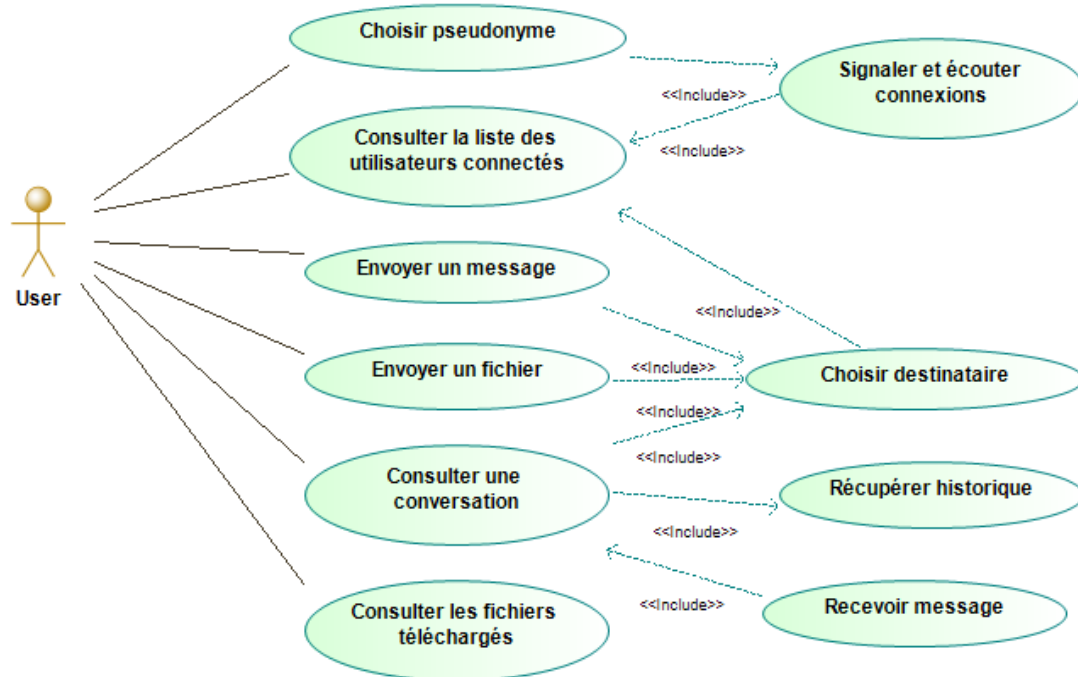


FIGURE 1.2 – Diagramme de cas d'utilisation initial

## Chapitre 2

# Manuel d'utilisation

### 2.1 Utilisation du client

#### Lancement de l'application

1. Télécharger [clavardage.jar](#) sur le dépôt GitHub.
2. Créer un fichier client.config dans le même dossier que le fichier clavardage.jar. Un [exemple](#) est disponible sur le dépôt, et un fichier valide permettant d'utiliser une base de données et un servlet hébergés en permanence a été envoyé par mail à M. Yangui.
3. Exécuter *clavardage.jar*.
4. Entrer un pseudonyme dans la fenêtre qui s'affiche et valider.

La fenêtre principale devrait alors apparaître.



FIGURE 2.1 – Fenêtre principale de l'application

**Liste des utilisateurs** Sur la gauche de la fenêtre principale, devrait se trouver une liste des utilisateurs connectés (éventuellement vide). Double-cliquer sur un utilisateur permet d'ouvrir la conversation avec cet utilisateur.

**Conversations avec des utilisateurs** La partie principale de la fenêtre est l'espace réservé aux conversations. Elle contient tout les messages déjà envoyés, et sera mise à jour à l'arrivée et à l'envoi de nouveaux messages. En haut de cette partie s'affichent les onglets correspondant aux conversations ouvertes. En bas de cette partie se trouve un champ de texte permettant d'écrire un message, un bouton pour l'envoyer, ainsi qu'un bouton pour envoyer un fichier et enfin un bouton pour ouvrir le dossier des fichiers téléchargés.

**Résolution des problèmes** Si les autres utilisateurs n'apparaissent pas, vérifiez bien que le réseau sur lequel vous souhaitez communiquer est le même par lequel vous accédez à Internet.

## 2.2 Lancement du serveur de base de données

La structure de la base de données est disponible dans le fichier [clavardage.sql](#). Ensuite vous devez définir un utilisateur qui puisse accéder à la base de donnée depuis un client distant.

```
MariaDB [(none)]> create user '<username>'@'%' identified by '<password>';
```

Et enfin vous devez donner les droits à cet utilisateur. Pour des raisons de simplicité, nous avons créé un utilisateur avec tous les droits mais cette pratique est à éviter.

```
MariaDB [(none)]> grant all privileges to '<username>'@'%' on <db_name>;
```

## 2.3 Lancement du servlet

Le servlet est matérialisé par le fichier [clavardage.war](#). Pour fonctionner, le servlet doit être mis sur un serveur *apache-tomcat*. La version utilisée pour ce projet est *tomcat9*. Sur la page de management de votre serveur (<http://server.addr:8080/manage/html>) vous pouvez déposer le *servlet.war* et le déployer. Une fois déployé le servlet sera accessible sur l'adresse <http://server.addr:8080/servlet>

## Chapitre 3

# Motivations de choix

### 3.1 Hypothèses

Pour le développement de ce projet, nous avons émis plusieurs hypothèses :

1. Chaque utilisateur sur le réseau utilise la même machine ;
2. Chaque machine possède une adresse IP fixe ;
3. Un utilisateur doit pouvoir changer de pseudo, il n'est donc pas identifié par ce dernier.

### 3.2 Interface graphique

Nous avons choisi Java Swing pour l'interface graphique puisque c'est la bibliothèque graphique que nous avons étudiée en cours. Elle est également très documentée avec de nombreux exemples dans la documentation officielle de Java.

Pour modéliser l'application, nous avons d'abord choisi une fenêtre `JOptionPane` pour le choix du pseudonyme. Cette classe est très simple et permet de faire exactement ce que l'on cherche avec un code concis.

Ensuite, la fenêtre principale est composée de deux parties : une `JList` représentant la liste des utilisateurs connectés, et permettant d'en sélectionner un facilement ; un `JTabbedPane` contenant toutes les conversations ouvertes, permettant de revenir simplement à une conversation précédemment ouverte.

Chaque conversation contient un `JTextArea` pour afficher l'historique des messages, un `JTextField` pour écrire un message, et plusieurs `JButton` permettant d'envoyer un message, d'envoyer un fichier, ainsi que d'afficher les fichiers précédemment téléchargés.

Pour que l'interface graphique soit mise à jour à la réception de nouveaux messages ou lors de la connexion d'un nouvel utilisateur, nous avons utilisé le design pattern Observer.



### 3.3 Modèle utilisé

#### 3.3.1 User

Commençons par l'un des objets les plus importants de cette application, l'objet **User**. Au vu des hypothèses présentées précédemment, l'utilisateur ne peut pas être représenté par son pseudo. Néanmoins, comme nous faisons l'hypothèse de l'adresse IP fixe pour chaque utilisateur, nous pouvons identifier ces derniers par leur adresse IP.

Nous avons donc fait le choix d'identifier les utilisateurs par un objet **Address** représentant un couple *@IP : port*.

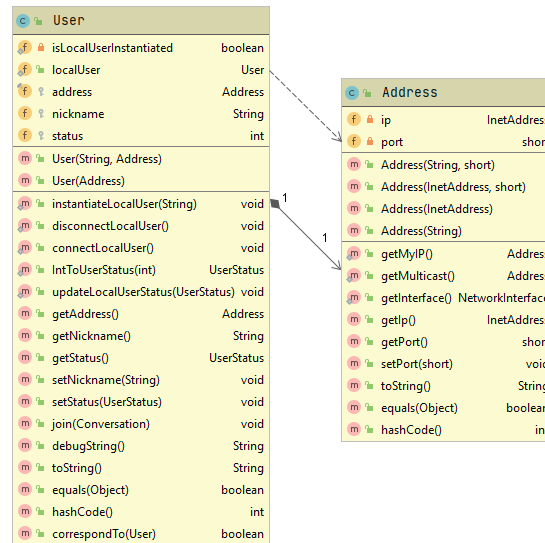


FIGURE 3.1 – Classes **Address** et **User**

#### 3.3.2 Address

L'objet **Address** représente un couple adresse IP et port, l'adresse d'un socket. La *Figure 3.3.1* représente la structure de l'objet **Address**. Cet objet rassemble aussi des fonctions statiques relatives à la machine de l'utilisateur. La fonction `getMyIp()` permet de récupérer l'adresse ip de la machine et le port sur lequel l'application est en écoute de nouveaux messages.

La fonction `getMulticast()` renvoie deux adresses au choix. Premièrement, elle permet de renvoyer l'adresse *multicast* du groupe de l'application. Nous verrons plus tard pourquoi nous avons choisis le multicast. Comme vu lors des expérimentations, il est possible que le réseau ou certaines machines (ayant une distribution Windows notamment) ne prenne pas en compte le multicast. Pour cela, il est possible dans le fichier de configuration de mettre *MULTICAST = false*. Dans ce cas là l'application va fonctionner avec du broadcast et cette fonction renverra l'adresse de broadcast sur le réseau courant.

Enfin la fonction `getInterface()` permet de récupérer un objet **NetworkInterface** représentant l'interface réseau de la machine.

#### 3.3.3 Conversation

La modélisation d'une conversation est assez simple. Comme le montre la *Figure 3.3.3*, une conversation est composée d'un identifiant, d'un nom, d'une liste de participant et de l'historique

des messages. Premièrement, il avait été choisis de ne pas mettre de nom, ni de liste de participants. Mais, il s'avère que cette modélisation est plus pratique car elle permet de prendre en charge des conversations à plusieurs utilisateurs (ce n'est pas possible à ce jour, mais si on souhaite mettre ça en place ce sera plus simple).

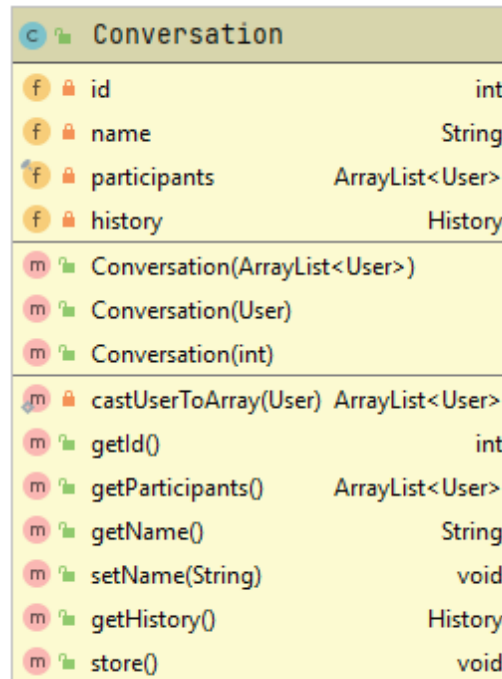


FIGURE 3.2 – Classes `Conversation`

## 3.4 Réseau

L'interface réseau est le coeur du logiciel et de nombreux choix ont donc été faits pour répondre aux besoins du cahier des charges.

### 3.4.1 *CCP - Clavardage Control Protocol*

La première chose à avoir été mise en place a été la découverte des utilisateurs connectés sur le réseau. Pour ce faire, nous avons élaboré un protocole de contrôle *Clavardage Control Protocol* (*CCP*) qui prend en charge tous les échanges réseaux qui servent au bon fonctionnement de l'application mais qui ne sont pas des messages. Ce protocole est basé sur des sockets UDP (port 1921 par défaut).

Lors du démarrage de l'application, un message *CCP\_DISCOVERY* est envoyé. Au début nous avons décidé d'envoyer ce message en *broadcast* pour toucher toutes les machines du réseau local. Très vite, nous avons fait le choix d'utiliser le *multicast* pour pouvoir atteindre seulement les machines qui implémentent l'application et ce, sur n'importe quel sous réseau de l'entreprise (si les routers sont compatibles *multicast*).

Si une machine reçoit un message *CCP\_DISCOVERY*, elle stocke les informations de cet utilisateur et lui transmet ses informations en retour avec un *CCP\_REPLY*. Enfin, il y a le type *CCP\_DISCONNECT* qui est envoyé lorsque l'utilisateur quitte son application ou qu'il se change

CCP packet identifier	CCP type	Payload
CCP	0	[Alice=192.168.2.12 :1921]

FIGURE 3.3 – Structure d'un paquet CCP

en statut *Invisible*. Ce message est envoyé en *multicast* pour prévenir tous les utilisateurs de la déconnexion de l'utilisateur.

### 3.4.2 Envoi de messages

L'envoi des message est effectué en TCP. Si Alice veut envoyer un message à Bob elle ouvre une connection TCP avec Bob, lui envoie le message et ferme la connexion. Nous avons préféré faire ça plutôt que de maintenir une connexion tant que la conversation est ouverte.

### 3.4.3 Envoi de fichier

Tout d'abord, dans le fichier de configuration, il y a l'option *FILE\_DIRECTORY*. Ce champ sert à renseigner le chemin du dossier dans lequel vous souhaitez recevoir les fichiers reçus. Pour reprendre l'exemple précédent, lorsque Alice veut envoyer un fichier à Bob, elle envoie un message de type *FILE* dont le contenu est le nom du fichier. Sur réception du message Bob ouvre un port de transfert de fichier (1922 par défaut) et répond à Alice en lui envoyant un message de type *FILE\_ACK* indiquant qu'il est prêt à recevoir un fichier. Sur réception de cette confirmation, Alice peut envoyer le fichier à Bob.

### 3.4.4 *ListenerPool*

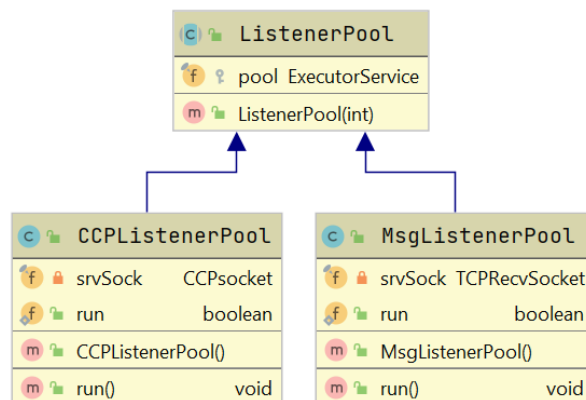


FIGURE 3.4 – Structure ListenerPool

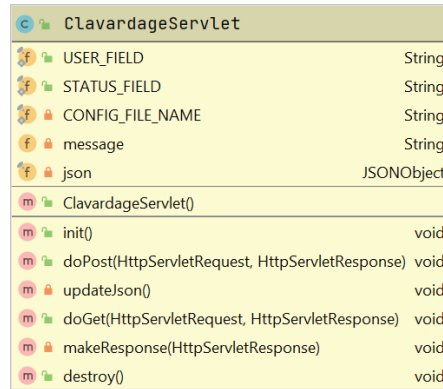
Les *ListenerPool* sont des ensembles de *threads* qui nous permettent de gérer plusieurs connexions simultanées sur notre application. Par exemple, pour pouvoir prendre en charge le phénomène d'amplification lié au *CCP\_DISCOVERY*<sup>1</sup>, nous avons un groupe de threads qui prend en charge le traitement des différents paquets. Le fonctionnement est analogue pour la réception des messages.

La taille de ces *ListenerPool* (nombre de threads) est configurable dans le fichier de configuration.

1. Lorsque l'on envoie ce message tous les utilisateurs nous répondent et donc plus le nombre de clients sur le réseau augmente plus le nombre de *CCP\_REPLY* va augmenter

### 3.5 Servlet

Lors du développement du projet, il a été demandé de mettre en place un serveur permettant de stocker le statut des utilisateurs. Pour ce faire, nous utilisons un serveur *tomcat9*. Ce servlet a un fonctionnement assez simple, il est composé d'une seule classe (voir *Figure 3.5*).



ClavardageServlet	
USER_FIELD	String
STATUS_FIELD	String
CONFIG_FILE_NAME	String
message	String
json	JSONObject
ClavardageServlet()	
init()	void
doPost(HttpServletRequest, HttpServletResponse)	void
updateJson()	void
doGet(HttpServletRequest, HttpServletResponse)	void
makeResponse(HttpServletResponse)	void
destroy()	void

FIGURE 3.5 – Clavardage Servlet

Dans le reste du projet, nous avons dû créer notre propre protocole pour l'envoi des messages, des fichiers et des paquets CCP. Ici nous utilisons le protocole *HTTP*. Pour récupérer la liste des utilisateurs et leur statut, il suffit de faire une requête *HTTP-GET* qui renvoie du *JSON*. Pour mettre à jour son statut, un utilisateur fait une requête *HTTP-POST* avec son identifiant (*@IP : port*) et son statut.

Les trois statuts disponibles sont :

- *Online*, assigné par défaut
- *Away*, pour signaler un moment d'absence
- *Invisible*, pour apparaître déconnecté tout en gardant l'application ouverte

### 3.6 Base de données

Le moteur de base de donnée utilisé est *MariaDB* (une extension de *MySQL*). Pour des raisons de praticité, on a décidé de stocker les tables nécessaires au fonctionnement de l'application et du *Servlet* sur une seule et même base de donnée mais elles n'ont rien en commun. Cependant en pratique, il faudrait avoir deux bases différentes. Sur la figure suivante vous pouvez voir la structure de la base de donnée des messages.

Nom de la table	Champs de la table
message_type	type
chat_room	id ; name ; users_number
chat_participant	id ; room_id ; user_id
chat_message	id ; room_id ; sender ; message ; created_at ; type

Cette base de donnée est conçue de sorte qu'elle puisse être compatible avec des conversations de groupe (3 participants ou plus, comme vu avec la modélisation d'une conversation précédemment).

La base de donnée du Servlet est beaucoup plus simple, elle est enregistre juste le statut des différents utilisateurs.

Nom de la table	Champs de la table
status_type	id ; description
user_status	id ; status

### 3.7 Fichier de configuration

Pour la simplicité, nous avons mis en place un fichier de configuration qui permet de changer facilement des paramètres et qui facilite ainsi le déploiement.

```
NETWORK_CLAVARDAGE_PORT = 1921
NETWORK_FILE_TRANSFERT_PORT = 1922
CCP_LISTENER_POOL_SIZE = 5
MSG_LISTENER_POOL_SIZE = 5
MULTICAST = true
MULTICAST_GROUP = 224.0.0.121

# /\ Warning, for Windows paths you need to escape colons, e.g. C\:/Path/To/Prefered/
Directory/
# Must end with a slash
FILE_DIRECTORY = <absolute_path>

DB_ADDR = <adresse>
DB_PORT = <port>
DB_DATABASE = clavardage
DB_LOGIN = <login>
DB_PASSWORD = <password>

# Servlet address: @IP:Port (BackSlash needed to escape the colon)
SERVLET_ADDR = <adresse>
```

FIGURE 3.6 – Fichier de configuration

# Conclusion

Lors de ce projet, nous avons acquis de nombreuses compétences tant en technique (en *Java* en particulier mais aussi *Git* et *Maven*) qu'en conception et en organisation de projet.

**Conception** Lors de l'analyse du cahier des charges nous avons dessiné des premiers diagrammes qui nous ont aidés à démarrer sur le projet, mais lors de l'implémentation beaucoup d'éléments se sont rajoutés et c'est ce qui complexifie les diagrammes finaux.

**Programmation** Durant l'implémentation nous avons dû résoudre beaucoup de problèmes qui ont menés à ce qu'est l'application aujourd'hui. Nous avons aussi essayé de rendre cette application optimisée avec par exemple l'utilisation du *multicast* à la place du *broadcast*. Mais nous avons aussi essayé au maximum d'adapter notre code pour des améliorations futures comme pour les conversations de groupes.