INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

PROJET POO

clavardage

MONIER Romain,
PAILLÉ Célestine
Promo 56, 4IR-I-A2

Date de Dépôt :
28 Janvier 2022

Année 2021-2022

TABLE DES MATIERES

I- Processus de développement

Tout au long de notre projet et de nos améliorations de code, nous avons eu une structure nous permettant une approche DevOps en utilisant des outils d'intégration continue ainsi que de déploiement continu. Nous avons utilisé trois principaux outils de gestion dont nous allons détailler ici les fonctionnalités.

A. Git, choix des branches

Tout d'abord, la gestion de version a été réalisée grâce à Git, nous avons pu ainsi gérer le projet et ses différentes versions. Nous pensions de prime abord partir sur deux branches principales, une branche *main* qui serait la branche correspondant à la version la plus récente du *build* de notre application ainsi qu'une branche *dev* qui serait celle où les toutes dernières fonctionnalités seraient ajoutées. Nous aurions alors créé une nouvelle branche à chaque développement de fonctionnalité et aurions pu effectuer une *Pull Request* sur la branche *dev* pour y intégrer les changements. Finalement, il aurait fallu *merge* la branche *dev* dans la branche *main* pour ensuite récupérer le binaire déployé.

Nous avons toutefois laissé tomber cette approche assez tôt. En effet, nous avons travaillé principalement sur deux packages différents dans notre équipe de deux, donc il y ne pouvait que très rarement avoir de conflits, et nous pouvions éviter d'avoir plusieurs branches et des *Pull Request* à gérer. Nous avons donc pris la décision de ne garder qu'une seule branche, la branche *main*, de façon à gagner du temps pour le développement. Nous n'avons d'ailleurs pas eu de soucis de conflits majeurs, donc ce choix s'est avéré être pertinent.

Au cours du développement, dès lors que nous sommes arrivés à une version suffisamment complète pour être exécutée et utilisée, nous avons commencé à les considérer comme des versions bêtas de l'application en suivant la norme de version sémantique. Ainsi, nous avons taggué à l'aide du système de tag de Git chaque commit de la branche main qui ajoutait une fonctionnalité ou venait patcher un bug. Nous avons eu 3 versions bêta, jusqu'à la 0.3.0.

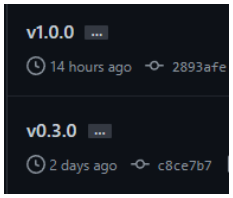

FIGURE 1 : DIFFERENTS TAGS

Pour ce qui est du dépôt, nous avons utilisé GitHub. Cette solution s'est révélée être extrêmement pratique pour des raisons que nous décrirons plus tard. De plus, l'affichage du *README* permet de rendre le guide d'utilisation plus ergonomique avec une table des matières permettant de trouver directement l'information recherchée.

B. Jira, suivi des tâches

L'outil suivant que nous avons utilisé est Jira, qui nous a permis de suivre notre cahier des charges et les tâches à faire tout en les répartissant par ordre de priorité et de difficulté, et ceci chaque semaine. Nous avons en effet défini des sprints en prenant des *user stories* découpées en tâches depuis notre *backlog*. Cela a permis un suivi efficace de l'avancement du projet, nous a aussi permis de réorganiser certaines fonctionnalités qui ont pu prendre plus de temps que prévu, mais aussi de se rendre compte de la difficulté ou facilité de certaines tâches que l'on pensait être bien plus simples ou difficiles.

FIGURE 2 : DIFFERENTES TACHES DU BACKLOG

Il est aussi important de noter que nous aurions pu utiliser l'outil *Projects* directement intégré sur GitHub, cependant la nouvelle version permettant de se rapprocher de Jira (donc avec des dates et des priorités) est toujours en bêta, nous sommes donc restés sur la solution Jira qui est à l'heure actuelle plus complète.

C. GitHub Actions et Gradle, outils CI/CD

Enfin, pour le CI/CD, donc l'intégration continue et le déploiement continu, nous sommes partis sur une solution directement intégrée à GitHub, GitHub Actions, associé à un *build tool*, Gradle. De cette façon, nous avons pu synchroniser beaucoup plus efficacement notre code de notre dépôt aux *workflows* continus, nous avons donc préféré cette solution à Jenkins.

Gradle est un outil qui permet un déploiement sur les machines de développement en une ligne de commande seulement, sans même avoir besoin de l'installer au préalable comme c'est le cas pour d'autres outils tel que Maven. Cependant, pour atteindre une telle efficacité, il brise une règle importante de l'utilisation des gestionnaires de version, il est en effet nécessaire de stocker une partie de ses scripts wrapper dans son dépôt, dont un fichier JAR.

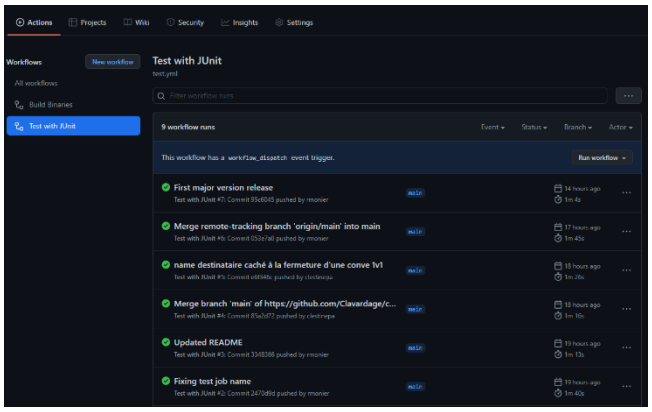

FIGURE 3 : TESTS JUNIT IMPLEMENTES DANS GIT

Pour l'aspect CI, nous avons automatisé une vérification des tests unitaires avec JUnit 5 à chaque push sur Git. Les résultats sont visibles dans l'onglet Actions. De plus, les tests sont faisables manuellement sur notre propre machine grâce au *build tool* et package manager que nous avons utilisé, Gradle. Une fois le dépôt Git cloné sur une machine, il suffit de lancer la commande de test pour que les tests unitaires s'exécutent. C'est aussi cette commande qui est utilisée lors de l'action automatique à chaque nouveau push.

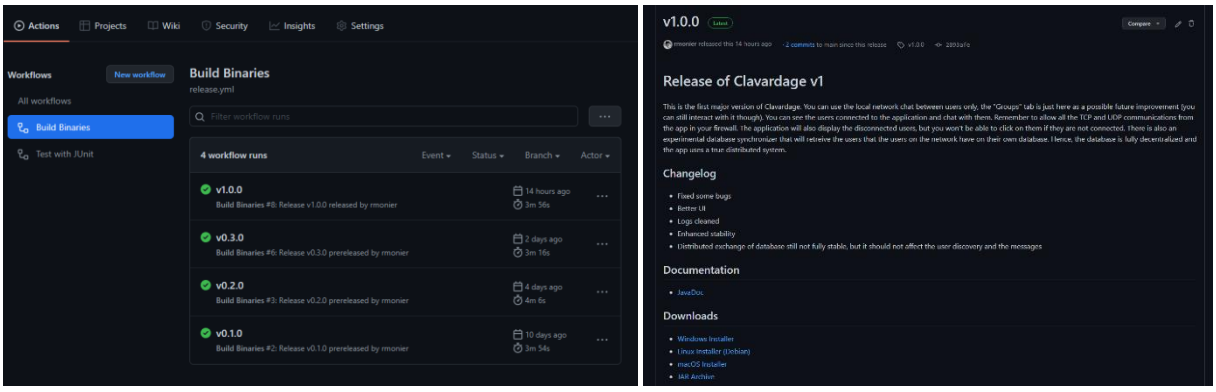

FIGURE 4 : BUILD PIPELINE

Pour créer une nouvelle version de notre application, à l'aide des tags Git, nous avons utilisé l'outil Release de GitHub. À chaque nouvelle version créée, GitHub Actions lance un pipeline d'actions qui vont permettre de réaliser plusieurs tâches. Le code des automations se trouve dans le répertoire *.github/workflows* du dépôt. Ce code se charge de lancer des tâches Gradle pour générer la *javadoc*, l'uploader sur la page de la release, générer l'archive JAR ainsi que les fichiers binaires exécutables pour les plateformes 64 bits Windows, macOS et Linux (Debian seulement). En effet, dans la recette de déploiement continu nous avons demandé l'exécution de la tâche permettant de créer des binaires sous ces 3 OS différents. La création de binaires se fait à l'aide de JPackage, un utilitaire intégré à Java 17 permettant de packager notre application dans un exécutable. Cet exécutable est ensuite wrappé dans un installateur créé par JPackage qui se chargera aussi de mettre une image de la

JVM utilisée qui se retrouvera aux côtés de l'application. Ainsi, l'utilisateur final n'aura même pas besoin de posséder Java sur sa machine pour exécuter l'application.

II- Technologies utilisées et application

A. Application

Notre application se découpe en suivant le design pattern MVC. De plus, nous avons utilisé Gradle, il y a donc une arborescence spécifique à adopter. La racine est composée des scripts Gradle décrits précédemment qui permettent d'exécuter l'application en une ligne de commande, du *README* qui décrit comment utiliser le dépôt en mode développement et d'un dossier clavardage qui contient la recette de build et les sources. Il y a aussi un dossier .github qui contient les workflows de CI/CD pour GitHub Actions. Le dossier source contient le code et les ressources de l'application dans le répertoire main, et les tests unitaires associés dans le répertoire test. L'application est découpée en 3 principaux packages : *model*, *view* et *controller*.

Le package *model* est composé des managers qui permettent d'interagir avec la base de données locale, des DTO (data transfert objects) qui seront instanciés par les managers et des exceptions personnalisées. Le package *controller* contient le point d'entrée de l'application c'est-à-dire le routeur (Clavardage.java) ainsi que d'autres packages qui correspondent aux différentes fonctionnalités de base de l'application. Le package *view* contient tout ce qui est lié à l'interface utilisateur.

B. Technologies

Clavardage se base sur l'OpenJDK 17. Java 17 est donc nécessaire pour développer l'application, et le build de l'OpenJDK utilisé pour les binaires exécutables distribués est celui de Eclipse Temurin. Tout autre build devrait fonctionner de manière équivalente pour exécuter le JAR ou développer avec Gradle.

Pour ce qui est de la base de données, nous avions pour objectif de la rendre complétement décentralisée, donc nous sommes partis sur du SQLite, nous permettant d'avoir un petit fichier local placé sur la machine de l'utilisateur.

C. Dépendances

Notre application se base sur plusieurs dépendances. La première est le driver SQLite JDBC pour pouvoir utiliser la base de données. Il y a aussi *AppDirs*, une dépendance permettant de facilement trouver le chemin pour stocker des données sur chaque OS, permettant d'y placer le fichier de base de données. Pour la gestion du *hashage* des mots de passes des utilisateurs, nous avons préféré l'algorithme *Bcrypt* au PBKDF2 inclut par défaut dans Java pour des raisons de sécurité, le *bcrypt* étant plus performant, nous avons donc utilisé une librairie pour cela. Enfin, pour vérifier facilement si l'email entré est valide et vérifier via une *RegEx* si le login correspond à nos contraintes, nous avons utilisé Apache Commons qui fournit de nombreuses classes utilitaires.

D. Détails techniques

L'application possède deux interfaces principales. La première est celle de connexion, qui attend que l'on se connecte ou se créé un compte. Nous sommes partis du principe que Clavardage étant à la destination d'entreprises, chaque utilisateur aurait un seul et unique mail et qu'il pourrait se connecter en l'utilisant comme identifiant unique. De plus, un UUID, identifiant unique généré aléatoirement, est associé à son compte. En arrière-plan, l'application utilise un Synchronizer qui va chercher les autres applications connectées sur le réseau local, et va venir récupérer leur base de données d'utilisateurs, permettant à chaque utilisateur de se connecter depuis n'importe quelle machine du moment que la

base de données décentralisée a été mise à jour. Une expression régulière est utilisée pour vérifier les noms d'utilisateur et emails.

Une fois authentifié, l'utilisateur dispose d'une interface qui affiche chaque user connu, donc qui a été connecté auparavant, et dès lors que l'un d'eux se connecte, une pastille verte s'affiche et il devient cliquable. La conversation s'ouvre alors si l'action est effectuée, la pastille devient bleue et l'historique des messages s'affiche. La découverte des utilisateurs s'effectue en UDP toutes les 10 secondes tandis que l'échange de message est en TCP. Dès lors qu'un message est envoyé ou reçu, il est sauvegardé dans la base de données. De plus, le Synchronizer effectue un envoi de la base de données qu'il possède toutes les 60 secondes en UDP à toutes les instances de l'application connectées au réseau local. Un système de base la plus récente était prévu, cependant nous n'avons pas eu le temps de l'implémenter. Lorsqu'un utilisateur change de nom, chaque application connectée est prévenue par une fenêtre modale afin que l'utilisateur ne soit pas perdu.

L'utilisateur connecté est géré dans la classe AuthOperations qui possède des méthodes statiques permettant de garder un seul et unique utilisateur sur l'instance courante. Les communications réseaux sont gérées par plusieurs daemons qui lancent chacun un thread permettant de s'occuper de différentes tâches. La connectivité est gérée en 3 couches distinctes. Le ConnectivityDaemon lie les activités et appels base de données aux services réseaux, les Services s'occupent de gérer des appels base de données et d'utiliser les Connectors qui eux font des appels TCP et UDP.

III- Manuel d'utilisation

Dans cette partie, nous verrons comment l'utilisateur peut naviguer et utiliser l'application. A l'ouverture de celle-ci, l'utilisateur se retrouve face à la fenêtre de connexion (voir A.).

Pour que l'application fonctionne, il faut impérativement que l'administrateur réseau de la machine de l'utilisateur ait configuré le pare-feu pour qu'il autorise les connexions UDP et TCP entrantes et sortantes pour l'application.

A. Fenêtre de connexion

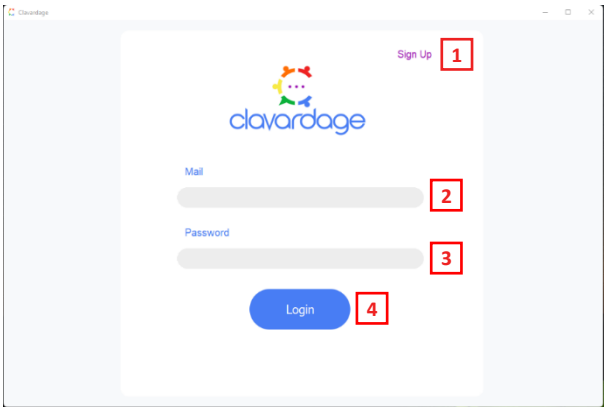

FIGURE 5 : FENETRE DE CONNEXION

Cette fenêtre permet à l'utilisateur de se connecter à son compte afin d'accéder à la fenêtre de messagerie (voir C.). S'il ne possède pas encore de compte, il doit s'en créer un. Pour cela, il lui suffit d'appuyer sur le bouton *Sign Up* [1] afin d'ouvrir la fenêtre de création de compte (voir B.). S'il possède un compte, il peut se connecter en rentrant le mail et le mot de passe associés à celui-ci dans les champs dédiés [2 et 3] et valider. L'utilisateur peut valider en cliquant sur le bouton *Login* [4] ou en tapant sur la touche *ENTREE* de son clavier. Si l'utilisateur ne rentre pas correctement les informations de son compte, il est averti et reste sur cette fenêtre. Si les informations sont cohérentes, l'utilisateur accède à sa messagerie.

B. Fenêtre de création de compte

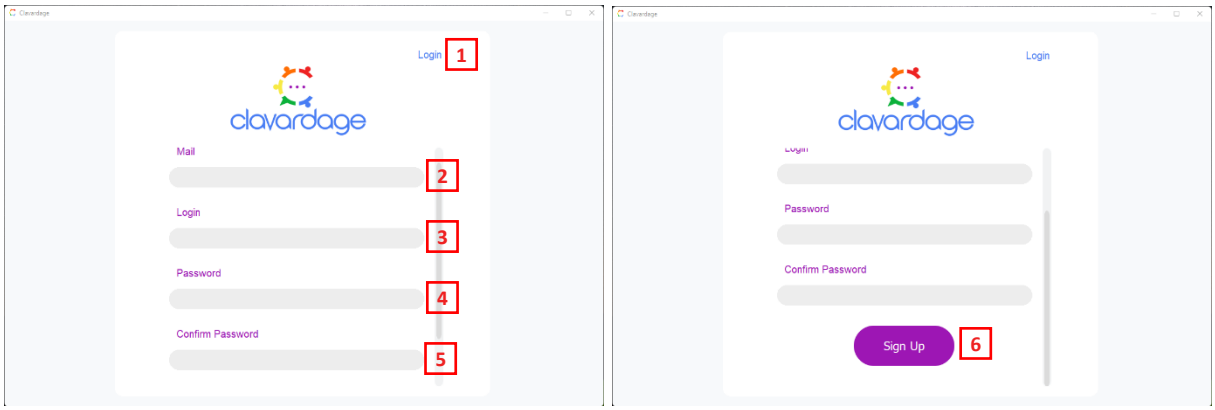

FIGURE 6: FENETRE DE CREATION DE COMPTE

Cette fenêtre permet à l'utilisateur de se créer un compte et de s'y connecter afin d'accéder à la fenêtre de messagerie (voir C.). Si l'utilisateur possède déjà un compte, il lui suffit d'appuyer sur le bouton *Login* [1] afin d'ouvrir la fenêtre de connexion (voir A.). Si l'utilisateur souhaite en créer un, il doit rentrer un mail [2] et choisir un nom d'utilisateur [3]. Le mail ne doit pas déjà être associé à un compte Clavardage et le login doit se composer de 3 à 20 caractères alphanumériques et/ou underscore. L'utilisateur choisit un mot de passe qu'il devra confirmer [4 et 5] puis il doit valider ces informations. L'utilisateur peut valider en cliquant sur le bouton *Sign Up* [6] ou en tapant sur la touche *ENTREE* de son clavier. Si l'utilisateur ne rentre pas correctement les informations, il est averti et reste sur cette fenêtre. Si les informations sont cohérentes, l'utilisateur accède à sa messagerie.

C. Fenêtre de messagerie

Cette fenêtre permet à l'utilisateur de naviguer dans sa messagerie. A l'ouverture, aucune conversation n'est ouverte. L'utilisateur peut accéder aux paramètres de l'application [1] dans lesquels il peut choisir son thème, entre clair (par défaut) et sombre, et changer la langue de l'application (anglais par défaut). L'utilisateur peut également ouvrir les paramètres du compte [2] dans lesquels il peut changer son nom d'utilisateur ou se déconnecter. Ces deux actions affichent un message d'alerte (voir c.).

FIGURE 7 : PARAMETRES GENERAUX

a. Aucune conversation affichée

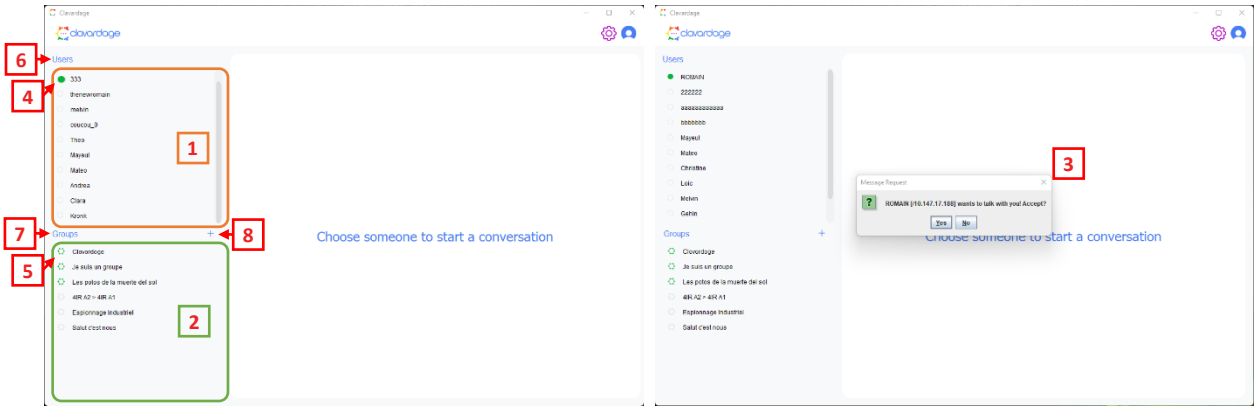

FIGURE 8 : FENETRE DE MESSAGERIE SANS CONVERSATION AFFICHEE ET DEMANDE DE DISCUSSION

L'utilisateur peut ouvrir une conversation en cliquant sur le nom des autres utilisateurs [1] ou des groupes [2] (ce qui envoie une demande de discussion au destinataire) ou en acceptant une demande de discussion [3] (qui apparaît quand un destinataire le demande). Ouvrir une conversation l'affichera automatiquement. Nous parlons de conversation ouverte quand l'échange de messages est possible

entre l'utilisateur et l'entité destinataire. Pour cela, il faut que les deux correspondants soient connectés et qu'une demande de discussion ait été acceptée.

Les entités connectées sont en haut de leur liste et marquées d'une pastille verte [4 et 5]. La connectivité des utilisateurs est vérifiée et mise à jour toute les dix secondes mais la connectivité des groupes n'est pas gérée. L'utilisateur peut cliquer sur les titres *Users* [6] et *Groups* [7] afin d'accéder directement au haut de la liste associée. L'utilisateur peut également créer un nouveau groupe en cliquant sur le bouton *+* [8]. Cette action ouvre et affiche automatiquement la conversation associée à ce nouveau groupe et l'utilisateur peut directement modifier son nom (voir c.).

b. Conversation affichée

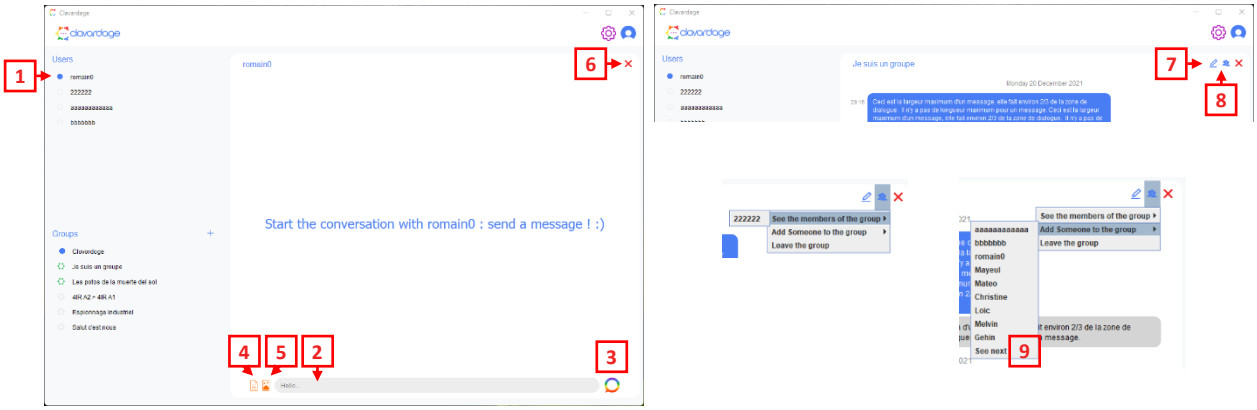

FIGURE 9 : FENETRE DE MESSAGERIE AVEC UNE CONVERSATION AFFICHEE ET PARAMETRES DE GROUPES

Plusieurs conversations peuvent être ouvertes en même temps et toute conversation affichée à l'écran est forcément ouverte. Les conversations en cours (c'est-à-dire qui sont ouvertes mais pas forcément affichées à l'écran) sont en haut de leur liste et marquées d'une pastille bleue [1]. Une fois une conversation affichée, l'utilisateur peut discuter avec l'entité. Pour cela, l'utilisateur doit écrire le message dans le champ dédié [2] et l'envoyer. L'envoi se fait en cliquant sur la bulle [3] ou en tapant sur la touche *ENTREE* du clavier. Seules les conversations entre deux utilisateurs sont reliées au réseau, l'envoi de messages dans une conversation de groupe est purement graphique. La réception de message et l'historique de ceux-ci (sauvegardé pour les futures connexions) sont donc réservés aux conversations entre deux utilisateurs.

L'utilisateur ne déclenchera aucune action en cliquant sur les boutons d'envoi de fichier [4] ou d'image [5]. Il peut fermer la conversation en cliquant sur la croix [6], ce qui affiche un message d'alerte (voir c.).

Lorsque la conversation affichée est celle d'un groupe, l'utilisateur est en mesure de modifier le nom du groupe en cliquant sur le bouton dédié [7]. Il peut alors directement taper sur son clavier pour modifier le nom. Le nom doit être inférieur à 40 caractères et ne peut être vide ou identique à l'ancien nom. Pour valider le changement, l'utilisateur peut cliquer dans une autre zone de l'application ou taper sur la touche *ENTREE* de son clavier. Si le nom choisit respecte les contraintes, le changement se fait, l'utilisateur en est averti. Si une contrainte n'est pas respectée, aucun changement ne se fait. L'utilisateur peut également ouvrir les paramètres de groupes [8] afin de voir les utilisateurs présents dans la conversation, en ajouter un ou quitter le groupe.

En cliquant sur *See the members of the group* ou sur *Add Someone to the group*, l'utilisateur peut parcourir la liste qui lui est proposé. Pour cela, il peut cliquer sur *See Back* ou *See Next* [9] (présents seulement quand nécessaire) qui lui permet d'afficher les utilisateurs précédant ou suivant. Lorsque l'utilisateur parcours la liste des membres du groupe, cliquer sur un des noms ne fait rien. Lorsqu'il parcourt la liste des utilisateurs qui ne sont pas dans le groupe, cliquer sur un des noms affichent un message d'alerte (voir c.). En cliquant sur *Leave the group*, un message d'alerte s'affiche (voir c.).

c. Message d'alerte

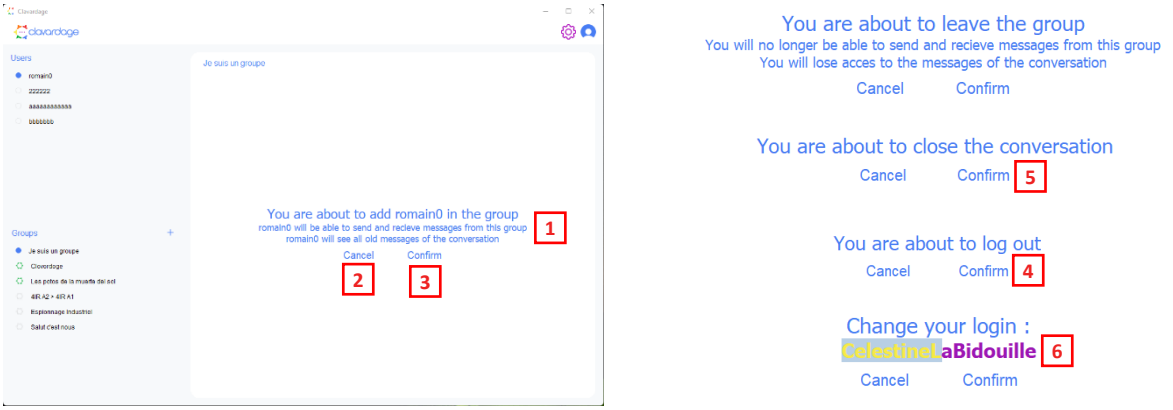

FIGURE 10 : FENETRE DE MESSAGERIE AVEC UN MESSAGE D'ALERTE ET LES DIFFERENTS MESSAGES D'ALERTE

Tant qu'un message d'alerte est affiché, l'utilisateur n'est plus en mesure d'ajouter un groupe ou de cliquer sur les noms des entités pour ouvrir une conversation (voir a.).

Tout message d'alerte est composé d'informations précisant à l'utilisateur la nature de l'alerte [1] et de deux boutons : *Cancel* [2], qui ferme le message d'alerte et ramène l'utilisateur dans le contexte qui précède son affichage, et *Confirm* [3].

Le bouton *Confirm* ferme le message d'alerte et déclenche l'action décrite dans l'alerte. En général, ce bouton ramène également l'utilisateur dans le contexte qui précède l'affichage du message d'alerte. Ce n'est pas le cas pour la déconnexion [4] ou la fermeture de la conversation [5]. La déconnexion ramène l'utilisateur sur la fenêtre de connexion (voir A.) et la fermeture de la conversation ramène l'utilisateur dans un contexte sans conversation ouverte (voir a.).

Le message d'alerte du changement de nom d'utilisateur permet à l'utilisateur d'écrire son nouveau login directement dedans [6]. Le login doit se composer de 3 à 20 caractères alphanumériques et/ou underscore. Si l'utilisateur rentre un nouveau login qui ne respecte pas les contraintes, cliquer sur *Confirm* ne validera pas le changement, l'utilisateur est averti de l'erreur et reste sur le message d'alerte.

Lorsque qu'un autre utilisateur change son nom, l'utilisateur en ai informé par l'apparition d'une fenêtre popup [7]. Il ne pourra pas réaccéder à sa fenêtre de messagerie tant qu'il n'aura pas fermé cette popup.

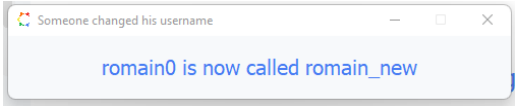

FIGURE 11 : POPUP D'UN CHANGEMENT DE LOGIN

IV- Conception UML

A. Brouillon des différents diagrammes

a. Diagrammes de classe

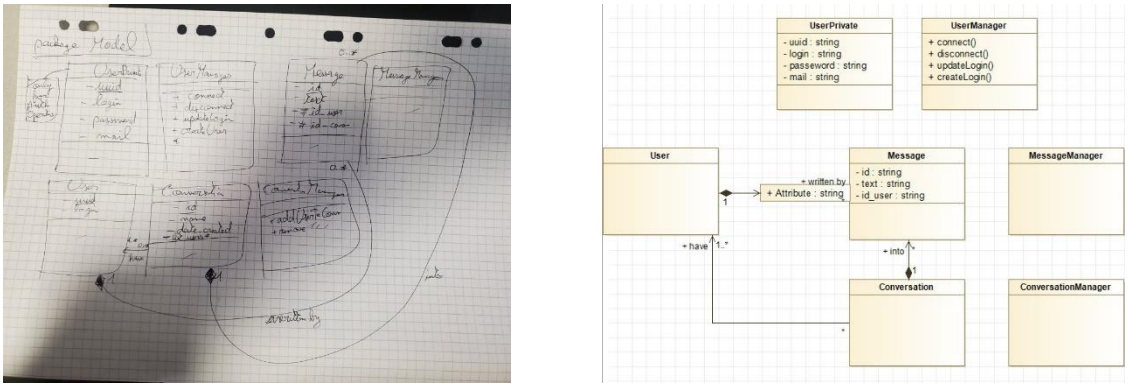

FIGURE 12 : BROUILLON DU DIAGRAMME DE CLASSE

b. Diagramme de cas d'utilisation et table de la base de données

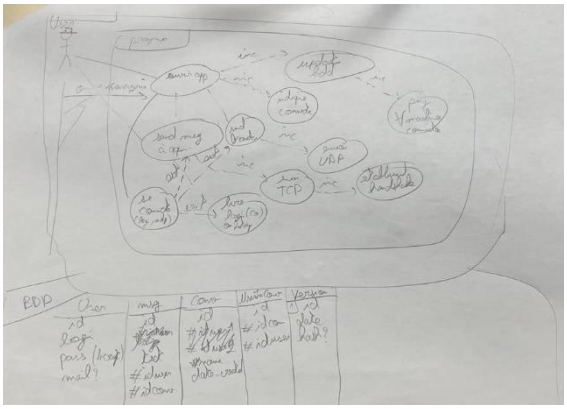

FIGURE 13 : BROUILLON DU DIAGRAMME DE CAS D'UTILISATION ET DE LA TABLE DE LA BASE DE DONNEE

B. Diagrammes finaux

a. Diagramme de classe des objets

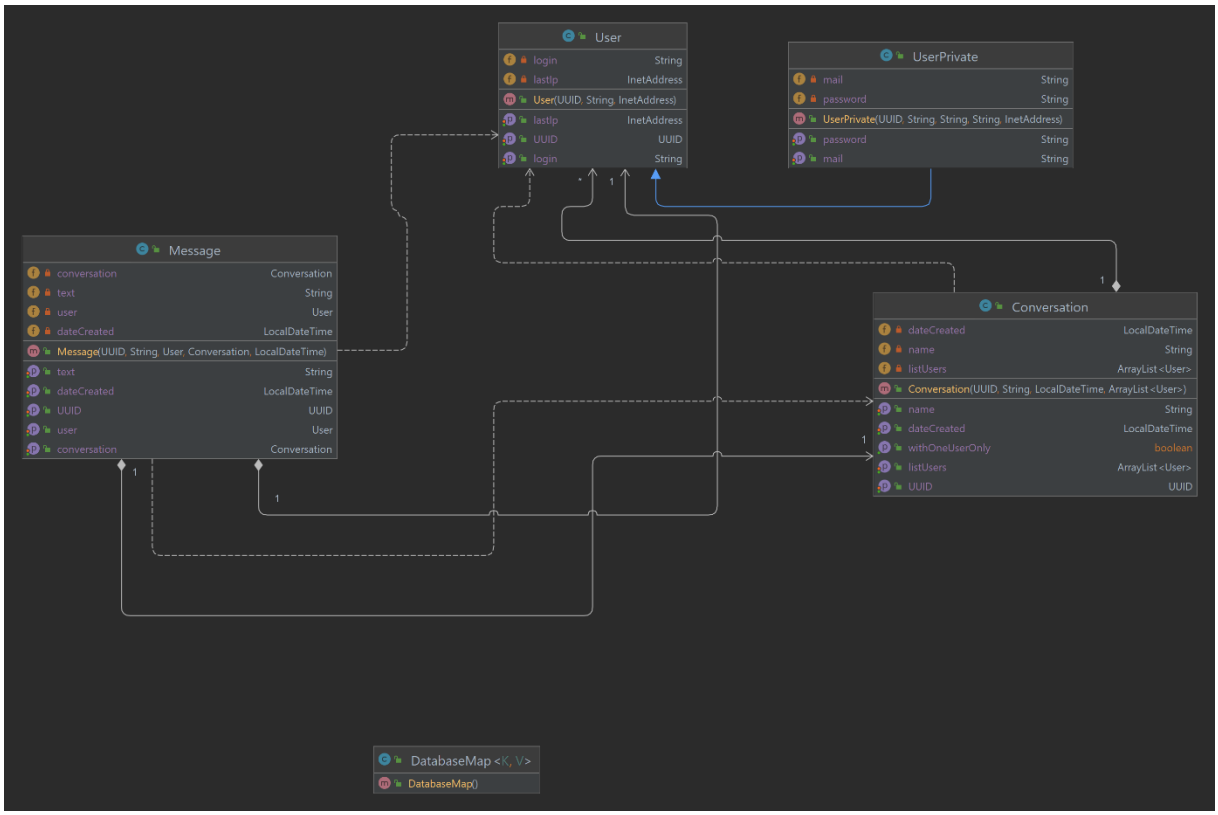

FIGURE 14 : DIAGRAMME DE CLASSE DES OBJETS

b. Diagramme de classe des connexions

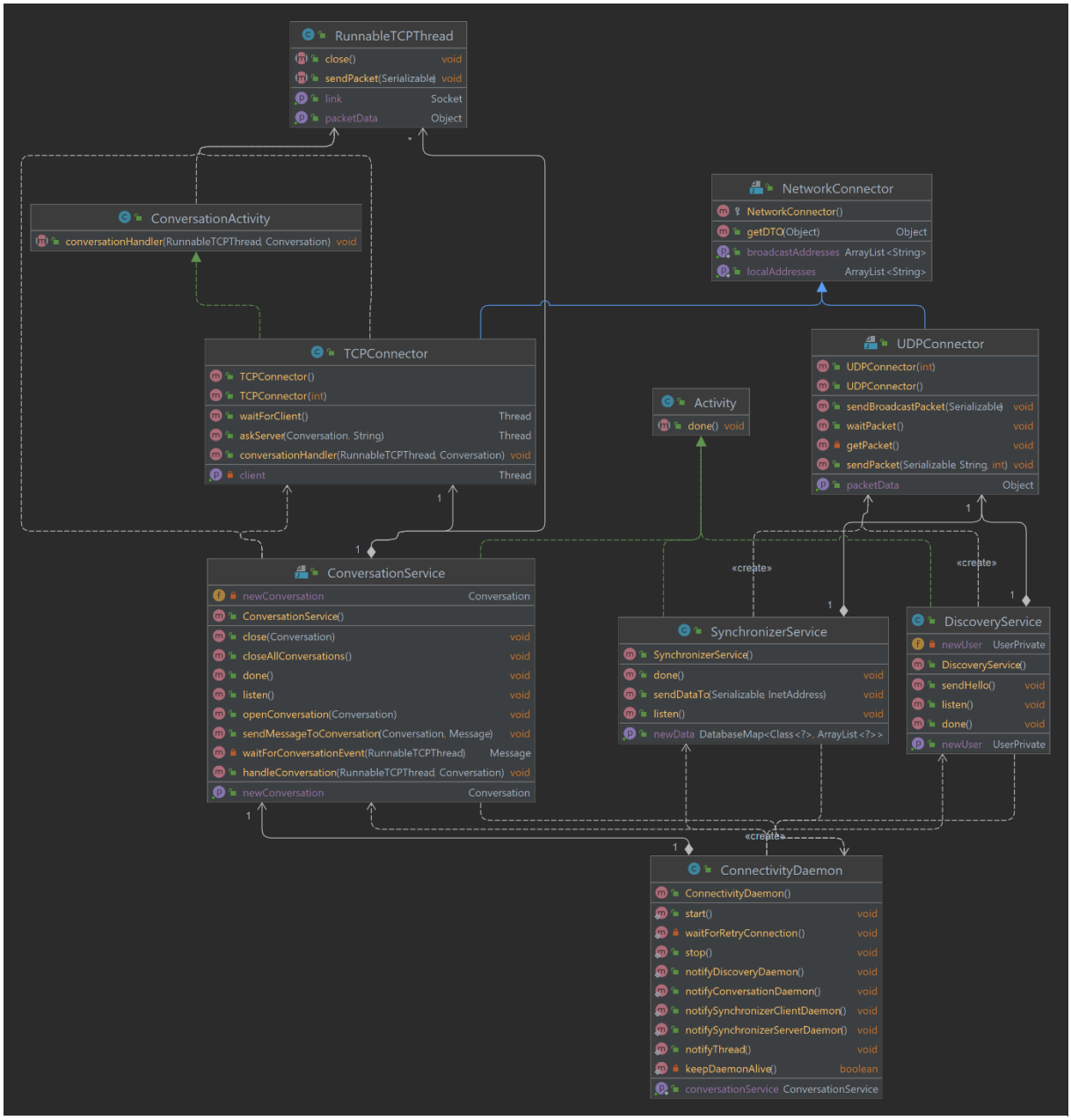

FIGURE 15 : DIAGRAMME DE CLASSE DES CONNEXIONS

c. Diagramme de classe de la base de donnée

FIGURE 16 : DIAGRAMME DE CLASSE DE LA BASE DE DONNEE

d. Diagramme de cas d'utilisation

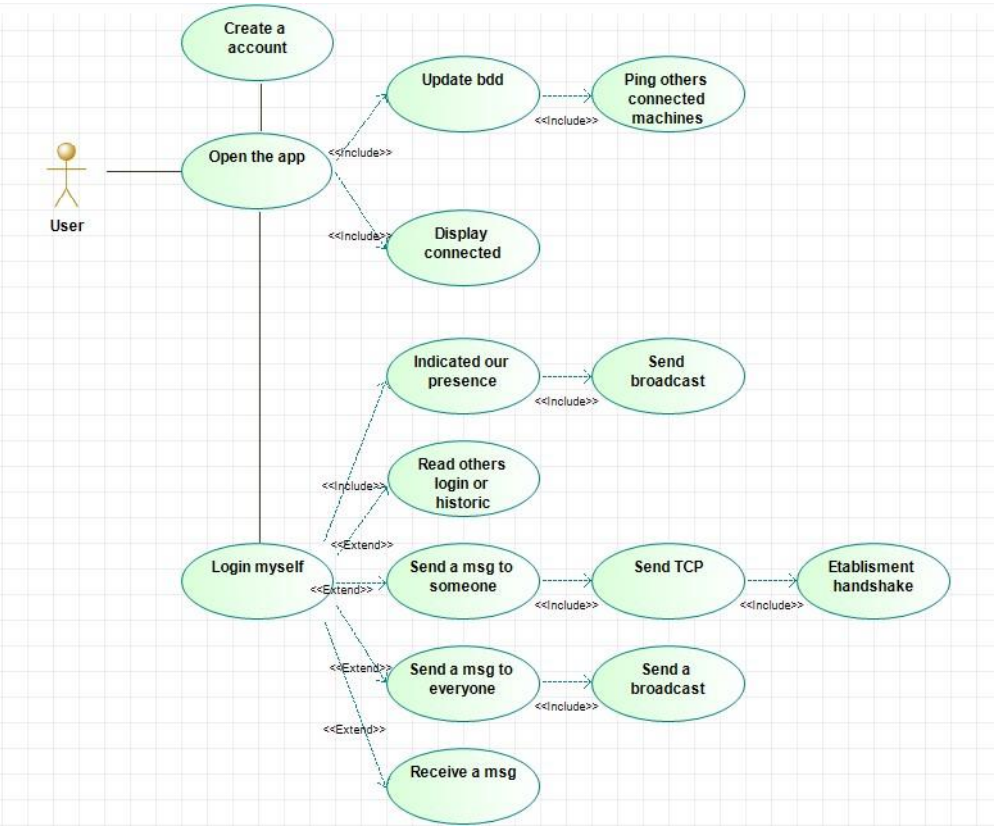

FIGURE 17 : DIAGRAMME DE CAS D'UTILISATION

e. Diagrammes de séquence

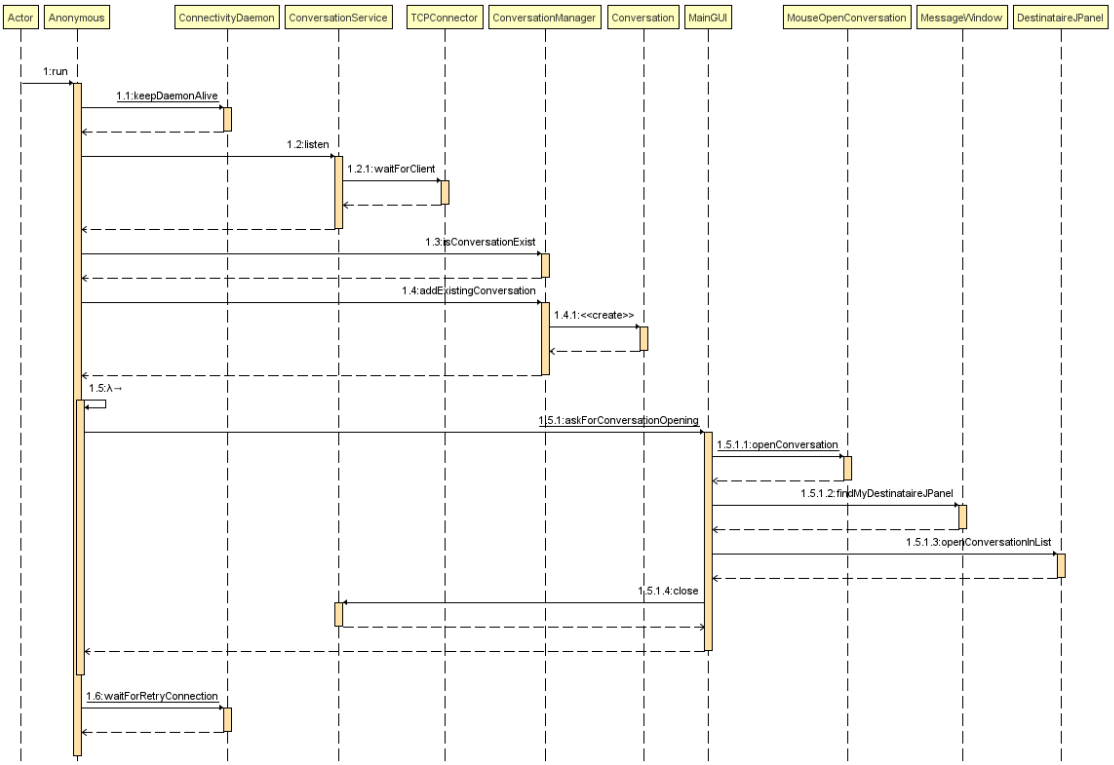

FIGURE 18 : DIAGRAMME DE SEQUENCE DE DE SERVICE DE CONVERSATION

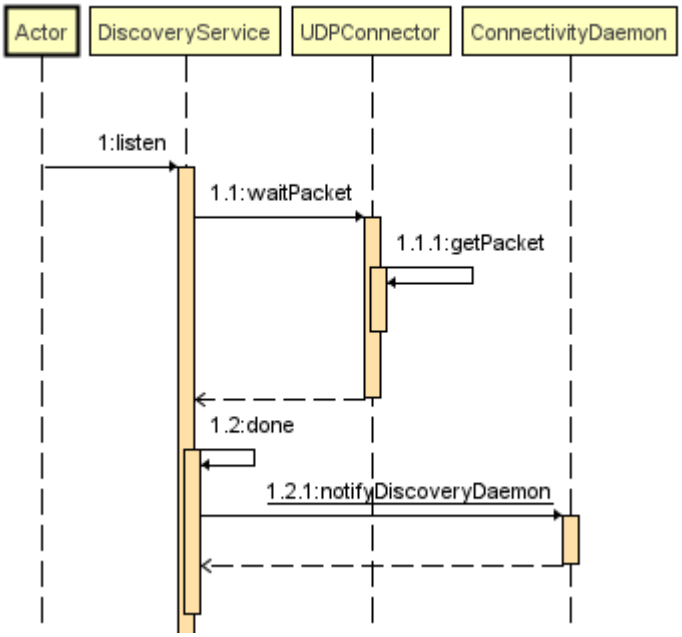

FIGURE 19 : DIAGRAMME DE SEQUENCE DE DE LA DECOUVERTE DES UTILISATEURS

Table des illustrations