Adrien MARCHAND
Julien MATHIEU

PROJET JAVA : DOCTOR CHAT

@github : https://github.com/Belgarel/Doctor_Chat

Table des matières

Adrien MARCHAND
Julien MATHIEU

1. Introduction

Notre projet, nommé « Dr Chat », avait pour objectif la conception et la réalisation d'une application client-serveur JAVA de type « chat » incluant les principales fonctionnalités suivantes :

- Se connecter au client ou à défaut pouvoir créer un nouveau compte.
- Ajouter des contacts existants dans la BDD.
- Démarrer une conversation avec un ou plusieurs contacts.
- Archiver en BDD les messages, les conversations, les contacts de l'utilisateur.

2. Fonctionnalités prévues

Écran de login

- Connection à un serveur, avec identifiant et mot de passe correspondant à un utilisateur enregistré dans la base de données du serveur.
- Contrôle d'erreur côté client : format de l'adresse du serveur, présence du login.
- Inscription d'un nouvel utilisateur (affichage d'une erreur si le login est déjà pris).

Écran de chat

- Chargement des contacts, des conversations et des messages existants dans la BDD.
- Ajout de contacts parmi les utilisateurs existants (affichage d'une erreur si le login n'existe pas ou s'il s'agit déjà d'un contact de l'utilisateur).
- Création de conversations avec un ou plusieurs contacts (affichage d'une fenêtre par conversation).
- Envoi de messages dans la conversation, enregistrés par le serveur et reçus par tous les membres de la conversation.

Features abandonnées

Au cours du projet, nous avons eu besoin de réduire le scope de celui-ci, principalement par manque de temps.

- Le stockage des mots de passe dans la base de données aurait dû être chiffré. Les mots de passe sont stockés en clair.
- Il aurait dû être possible de créer des conversations à plusieurs participants, et de les sélectionner via un système d'onglets. Cette feature a été abandonnée.
- Il aurait dû être possible de supprimer des contacts ou des messages postés d'une conversation. Cette feature a été abandonnée.

Adrien MARCHAND
Julien MATHIEU

3. Fonctionnement de l'application

Base de données

Le server a pour principal rôle de communiquer avec les clients d'une part, avec la couche persistante d'autre part.
La couche persistante est la base de données Oracle de l'université. Nous utilisons le driver officiel *ojdbc* téléchargeable sur le site d'Oracle pour nous connecter.

Structure de la BDD

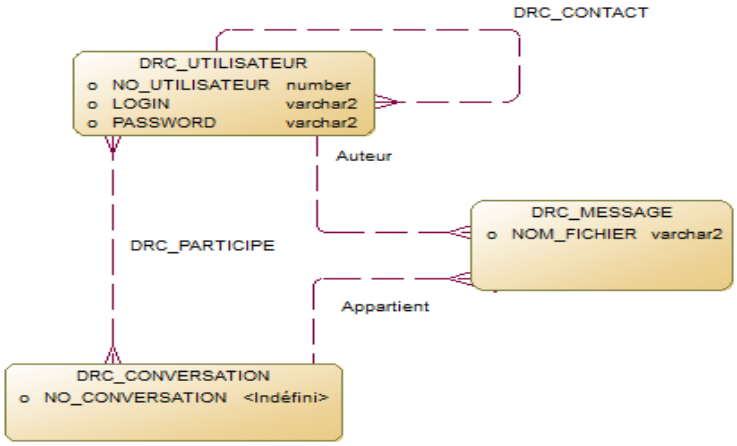
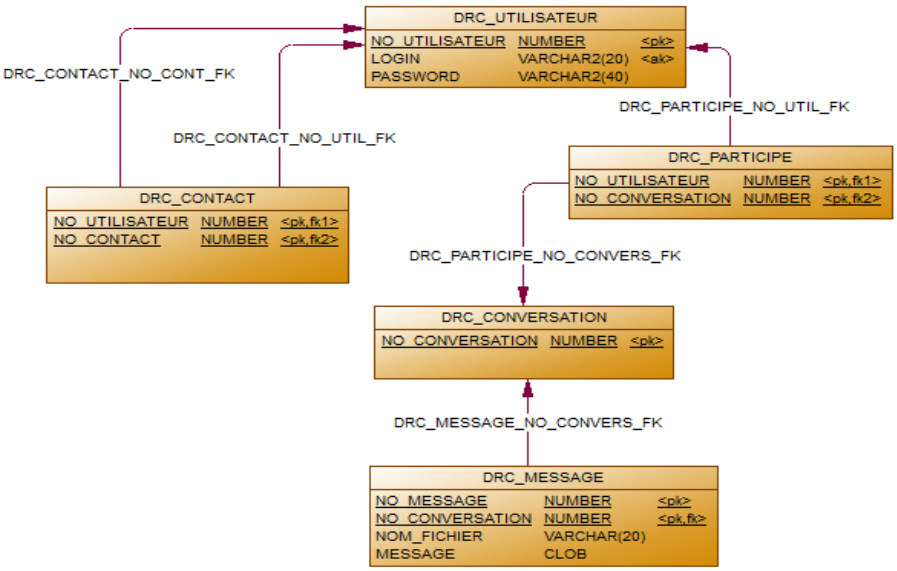


Schéma physique de la BDD



Côté serveur, des classes *xxxService* sont chargées de la communication avec les tables. Elles possèdent une implémentation de fonction de bases telles que « *findByLogin* », « *createMessage* », « *deleteConversation* » selon nos besoins.

Communication client-serveur

La communication client-serveur s'effectue par l'envoi de messages typés entre les deux tiers.

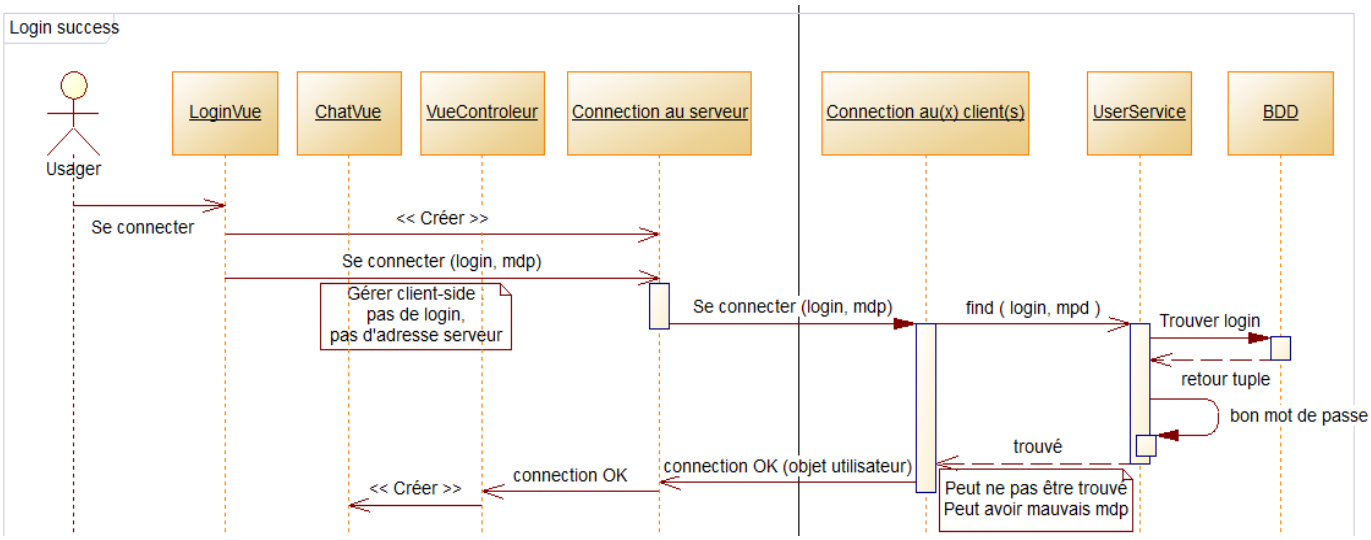
Il existe deux types de message : les messages en provenance des clients, héritant de la classe « *ClientMessage* » et les messages en provenance du serveur héritant de la classe « *ServerMessage* ».

Chaque type de message possède un contrat, une responsabilité qui lui est propre :

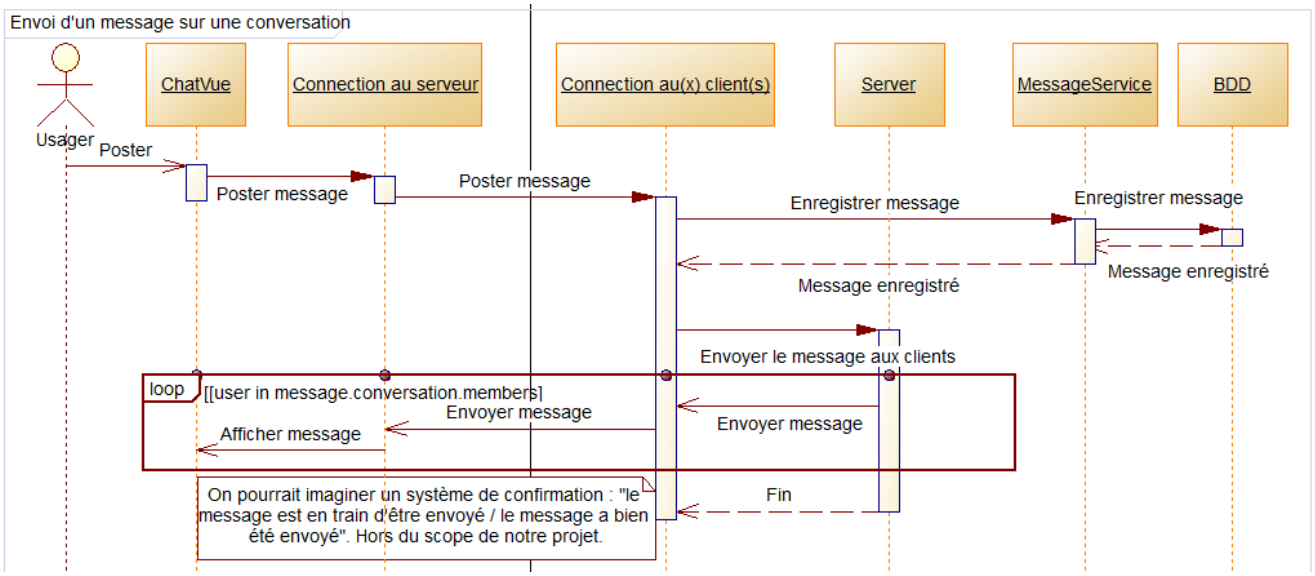
- Pour les clients : demande de connexion, demande d'ajout de contact, invitation à rejoindre une conversation, ajout d'un message…
- Pour le serveur : réussite/échec de la création d'un nouveau contact, réussite/échec de l'authentification du client…

Voici deux illustrations de ce fonctionnement :

- Le cas du **login** (scénario nominal) est assez simple.



- Le cas d'un **envoi de message** est plus complexe.



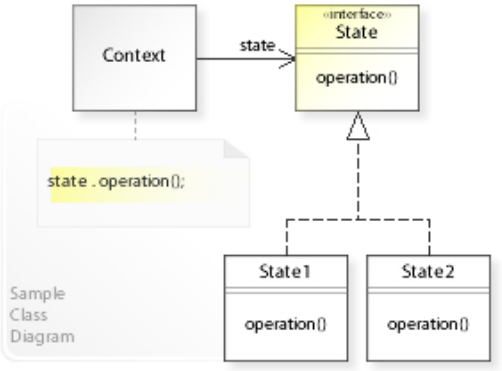
Côté vue : javafx

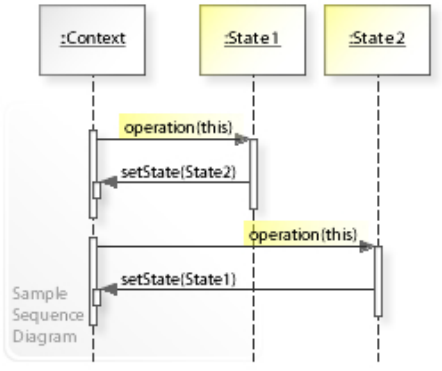
Chaque écran est décrit par un fichier fxml (listant les éléments statiques) construit avec SceneBuilder, auquel est associé une classe Controller qui prend en charge les éléments dynamiques et les comportements. La navigation entre écrans est orchestrée par la classe *« ViewController »*.

Côté vue : le design pattern état

Le design pattern État intervient en général lorsqu'un diagramme d'états-transitions ou d'automates à états est possible.

Le principe est simple, il doit être possible de représenter n'importe quel état et le traiter indifféremment d'un autre. Un état doit pouvoir recevoir des évènements qu'ils puissent les traiter et éventuellement muter vers d'autres états.





Adrien MARCHAND
Julien MATHIEU

Traitement des différents messages échangés entre le client et le serveur

Quand un message est émis par le client, il se retrouve dans la fonction *« run() »* de la classe *« server.connection.ConnectedClient ».* Plus spécifiquement, il y a un thread par instance de *« ConnectedClient »*, qui est chargé d'écouter et traiter les messages qui lui sont adressés.

Si le message n'est pas *« null »*, autrement dit, s'il ne s'agit pas d'une déconnection, il est orienté vers une fonction spécifique de cette classe selon son type à l'aide du mot-clé *« instanceof ».* Cette fonction effectuera alors un traitement différent suivant le type de message : mise à jour de la couche persistante, envoi de messages à d'autres clients ou réponse à l'envoyeur.

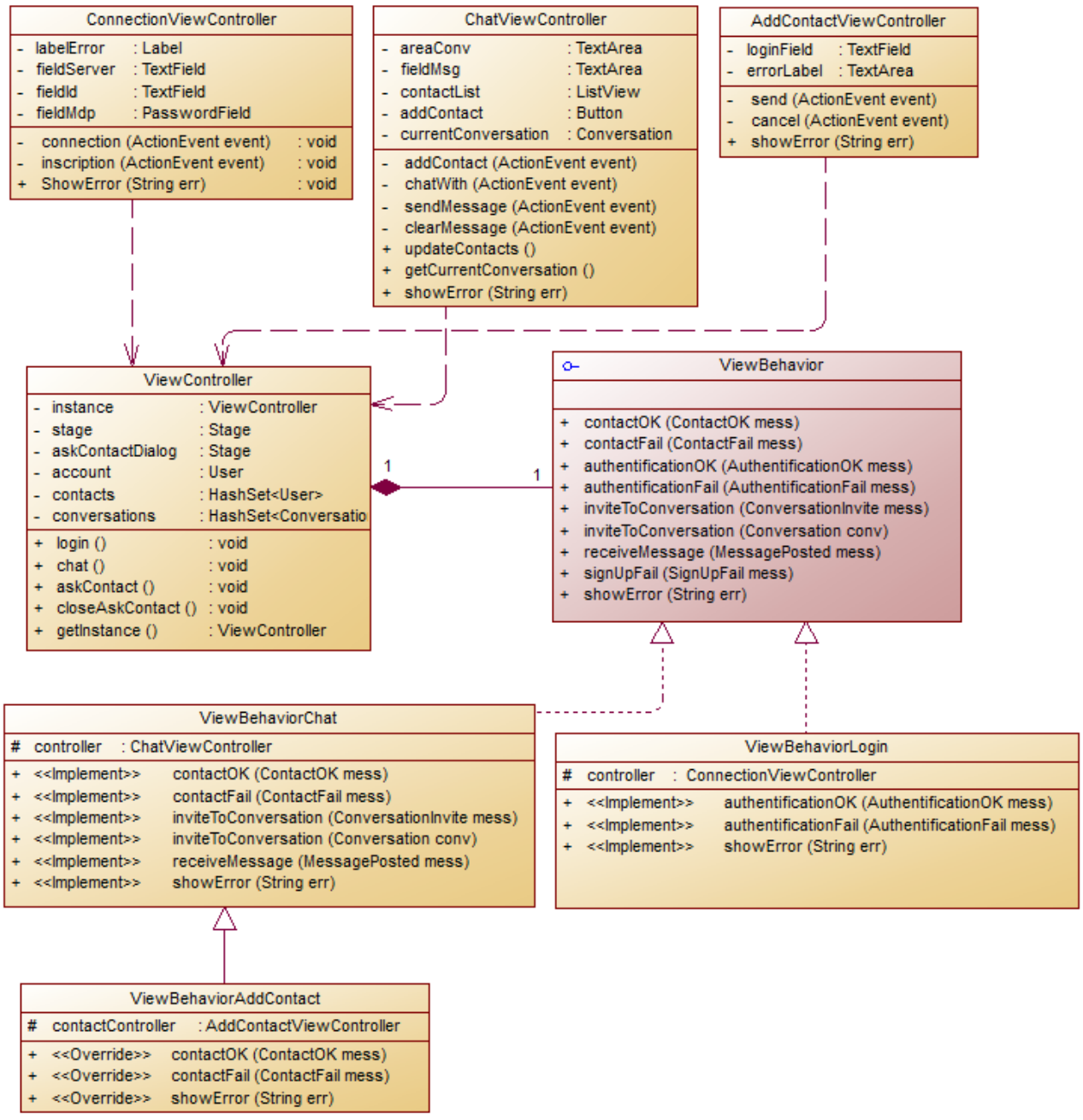
Lorsqu'un message est émis par le serveur, il est capté par la fonction *« run() »* de la classe *« client.connection.ClientReceive »*. Le rôle de cette classe est, comme ConnectedClient, d'écouter le serveur. Si le message n'est pas *« null »*, il est, selon son type, aiguillé vers les fonctions de traitement.

Ces fonctions de traitement sont déléguées à l'état (« *ViewBehavior »)* du contrôleur de la vue *(« ViewController »*) car la réponse à apporter dépend de l'état du logiciel : connexion, chat ou ajout de contact.

Par exemple, si un message autorisant la connexion est reçu alors que l'utilisateur est déjà connecté, il n'est pas adapté de déconnecter l'utilisateur courant pour reconnecter le nouvel utilisateur, le message est donc considéré comme un bug et ignoré (il serait aussi possible de le logger à des fins de débogage).

Adrien MARCHAND
Julien MATHIEU

Diagramme de classe contrôleurs / états (package client)



4. Organisation du groupe

En raison de notre différence de niveau, on ne s'en cache pas, c'est Adrien qui a réalisé la majeure partie du projet et pris les décisions. L'objectif pour Julien était de comprendre et pratiquer autant que possible, en essayant d'accomplir les tâches que déléguerait Adrien.

Pour la base de données, les décisions ont été prises en commun, et l'essentiel du SQL a été écrit par Julien.

Nous avons, pour une partie du projet, décidé que Julien essaierait d'établir la base de l'architecture client-serveur tandis qu'Adrien mettait en place la communication entre le serveur et la base de données (connexion et services).

Adrien a exploré un peu javafx en créant la vue de login, avant de déléguer la vue chat à Julien.

Adrien a aussi testé le workflow des messages client-serveur avec le login, avant d'en déléguer des parties à Julien (notamment ce qui concernait l'ajout d'un contact). Julien a aussi fixé des bugs qu'identifiait Adrien.

Ce que j'ai appris – Adrien

Je n'avais encore jamais monté un aussi gros projet de A à Z. J'ai la mauvaise habitude de prendre les décisions en faisant, c'est-à-dire que je comprends l'application non pas *avant* de la réaliser, mais *en* la réalisant. Ce projet m'a permis de planifier un peu plus, et de communiquer avec mon partenaire sur mes décisions.

Je n'avais jamais réalisé de connexion à une base de données oracle auparavant. Il y a des erreurs que je ne ferai pas deux fois.

Rétrospectivement, je me suis rendu compte que les classes Service (pour le Serveur) utilisaient beaucoup de fois le même genre de code. Spring aurait accéléré les choses, mais je voulais les garder accessibles à mon partenaire. Cependant, même ainsi, je pense que j'aurais pu créer des fonctions qui auraient minimisé la quantité de code dupliqué (telles que "effectuer une requête d'insertion par exemple).

La communication entre client et server, c'était aussi du nouveau pour moi. Je suis content de la décision d'avoir créé une classe par message, ça marche bien. La nomenclature aussi est assez claire.

Dans la vue, j'ai mis en place un des *design patterns* (état) récemment appris en UML, et j'ai trouvé ça pratique, solide, extensible, logique... en somme, bien pratique. Rétrospectivement, ça n'en valait peut-être pas tant la peine à l'échelle d'un si petit projet, mais c'était une bonne occasion de faire l'exercice et d'en comprendre l'utilité.

Java FX était totalement nouveau pour moi, c'est donc là que j'ai appris le plus. Le gros défi a été de faire le lien entre les fichiers fxml générés par SceneBuilder (et chargés par le loader) et le Controller.

Si le projet était à refaire, je crois que je serais capable de plus planifier, pour mieux communiquer dès le départ.

Ce que j'ai appris - Julien

Notre projet m'a permis de découvrir ce que peut-être le travail d'une équipe de développement avec un objectif commun, un scope et des délais à tenir.

J'ai pu mettre en pratique et approfondir les concepts abordés dans plusieurs modules : répartition des tâches et gestion du planning, importance de la conception avec l'utilisation de design patterns, création et utilisation d'une base de données.

Concernant le langage JAVA, j'ai pu approfondir certains points vus en cours et en découvrir d'autres :

- Construction d'une architecture client-serveur avec l'utilisation de threads,
- Envoi et traitements des messages entre le client et le serveur,
- Utilisation d'une BDD dans une application Java,
- Création d'interfaces graphiques avec l'outil SceneBuilder et compréhension du modèle MVC,
- Application des principes de la POO avec le design pattern l'utilisation du design pattern « État ».

D'autre part j'ai pu apprendre les rudiments de l'utilisation de git.

Je tiens à remercier Adrien pour avoir été à la fois le leader de notre projet ainsi qu'un très bon formateur et un coéquipier patient et compréhensif.