Projet-Conception-Logicielle

Introduction

Ce projet consiste en la création d'un service de messagerie distribuée à l'instar de Teams ou discord. Il s'articulera autour d'un client et d'un serveur.
Le serveur est en charge de la distribution des groupes, messages et responsable de l'authentification des utilisateurs.
Le client permet lui à un utilisateur l'envoi de requêtes via une IHM, lui permettant de communiquer avec les autres utilisateurs.

Membre du projet : Alexandre Froehlich, Guillaume Leinen, Jean-Noël Clink, Erwan Aubry, Ayrwan Guillermo

Serveur : Alexandre, Erwan

Client : Guillaume, Jean-Noël, Ayrwan

Fonctionnalités et patterns implémentés

Fonctionnalités

Le serveur est capable de :

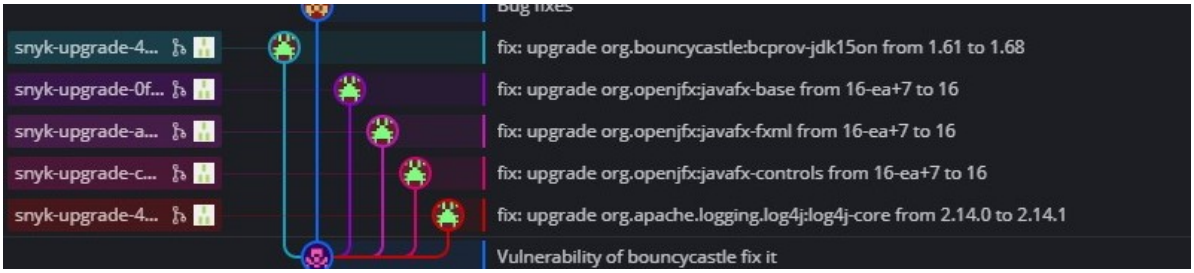
- Importer les données d'une base de donnée dans le serveur.
- Répondre à une requête du client (format JSON) avec une fonctionnalité d'exclusion mutuelle implémenté par une machine à états finis : en bloquant le serveur sur un état défini, on empêche la création de requêtes en doublon et on évite la saturation du serveur (SPAM ou DDOS).
- D'accepter de multiples connexions clients le tout en concurrence
- De vérifier l'authenticité d'un client au travers d'un jeton de sécurité généré et vérifié par le serveur
- De modifier la base de donnée en fonction des requêtes client

Le client peut quand à lui :

- permettre à l'utilisateur de **s'authentifier** par un login et mdp
- accéder aux discussions auquel l'utilisateur est inscrit
- créer une nouvelle discussion ainsi que de modifier les paramètres d'une discussion
- retrouver les messages précédemment envoyés par soi-même ou d'autres utilisateurs
- et enfin d'envoyer un message dans une discussion

De manière commune, un **logger** permet un retour d'information clair dans le terminal exécution pour chaque application.

Il est important de noter que la **sécurité** à été un point d'intérêt particulier au cours de ce projet. Ainsi, les mots de passes sont **hashés** et ne transite pas en clair par le réseau, et l'identifiant est lui transité par une chaîne de caractère aléatoire** créé lors de la création d'un profil. Aussi le programme est analysé par un bot hébergé sur GitHub qui renseigne sur les failles : SNYK



Ici le bot à corrigé les vulnérabilités liés à des bibliothèques non définitives ou pas à jour.

Enfin, les profils, groupes de discussions et messages sont tous stockés sur une **base de donnée SQLite**.

Patterns

Pour ce projet nous avons utilisé un certains nombre de patterns de programmations :

- serveur TCP concurrent : implémenté au niveau du serveur. Permettant de créer le serveur en lui même et écouter la connexion de divers clients.
- motif sujet-observateur et délégation : ce motif est implémenté au niveau du client. L'envoi de message n'est pas supervisé par le contrôleur de discussion mais par une classe observateur. Le Contrôleur est un observable et un clic sur le bouton "ENTRER" va enclencher une notification envoyé via l'API Java à observateur "MessageSender". C'est lui qui se charge de l'envoi de message. On a donc à la fois une classe observatrice qui interagit pas avec l'élément observé, et une classe qui permet une délégation d'une fonctionnalité.
- motif stratégie : implémenté au niveau du serveur. Permettant de classer et faciliter le développement des différents protocoles présents, afin que l'utilisateur client puisse interagir avec le serveur.
- motif composant-composite : implémenté au niveau du serveur. Permet de créer une machine à état fini afin de gérer le fait que le serveur accepte la connexion, envoie des messages, reçois des messages et ferme la connexion.

Conception : Cartes CRC

Nous nous sommes servis de fiches CRC que nous avons complété au fur et à mesure des implémentations du projet.

Ces cartes permettent de représenter les interactions entre les classes, voici un exemple :

State
Responsabilités : -représente un état de la machine à états finis - etats possibles : idle (non connecté), wait(connecté au serveur) ,sending (envoi de message)
Linked to classes : ACTION FSM

MACHINE A ETATS FINIS [NOT IMPLEMENTED]

FSM
Responsabilités : -connait l'état du client au regard de la connection (idle, wait, sending) -responsable du changement d'etat par la méthode transit()
Linked to classes : STATE ACTION MAIN CONTROLLER

L'ensemble des cartes CRC est consultable dans `docs/CRC`

Conception : Tests unitaires

Nous avons testé un certains nombre de fonctionnalités qui étaient à notre portée et clés pour le fonctionnement du projet. Nous nous sommes donc concentrés sur les tests du serveur, vu que le client est essentiellement une IHM et qu'il est difficile de tester des fonctionnalités @fxml (en tout cas avec JUnit).

Vous pouvez retrouver les tests dans le dossier "test" de chaque projet respectif.

Voici les coverages relatifs à ces tests :

Serveur :

Current scope: all classes

Overall Coverage Summary

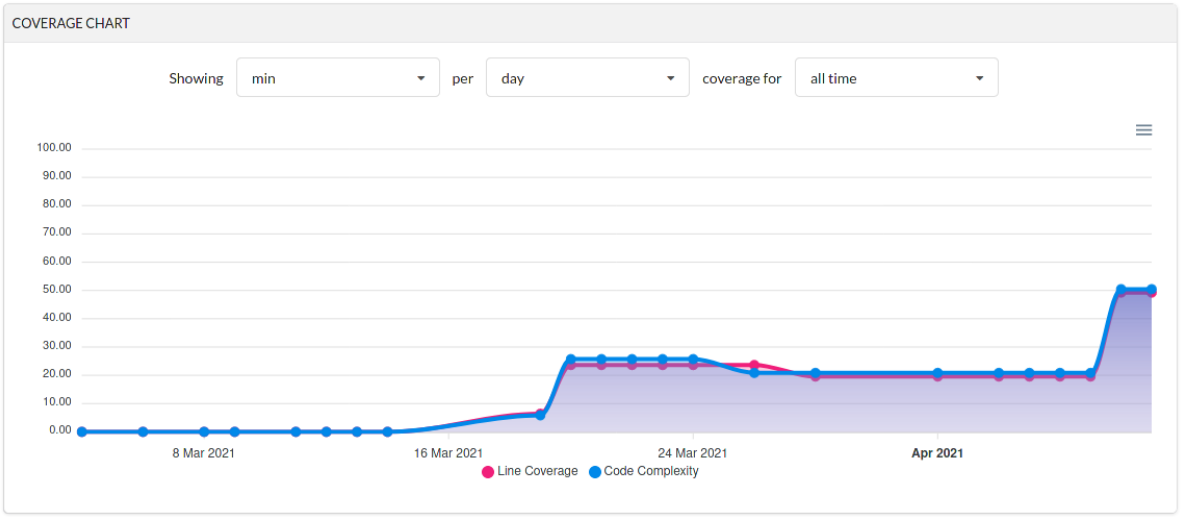
Package	Class, %	Method, %	Line, %
all classes	67,6% (23/34)	58,9% (99/168)	52,7% (414/786)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
<empty package name>	0% (0/1)	0% (0/2)	0% (0/24)
context	100% (1/1)	100% (2/2)	100% (4/4)
database	50% (2/4)	76,2% (16/21)	75,3% (70/93)
database.entities	100% (4/4)	84,8% (28/33)	80,8% (63/78)
fsm	0% (0/6)	0% (0/25)	0% (0/97)
protocols	100% (12/12)	100% (36/36)	85,2% (218/256)
server	66,7% (4/6)	34,7% (17/49)	25,2% (59/234)

generated on 2021-04-08 21:42

Coverage en fonction des commit :



Client :

Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	16% (4/25)	9,2% (17/184)	5,9% (54/911)

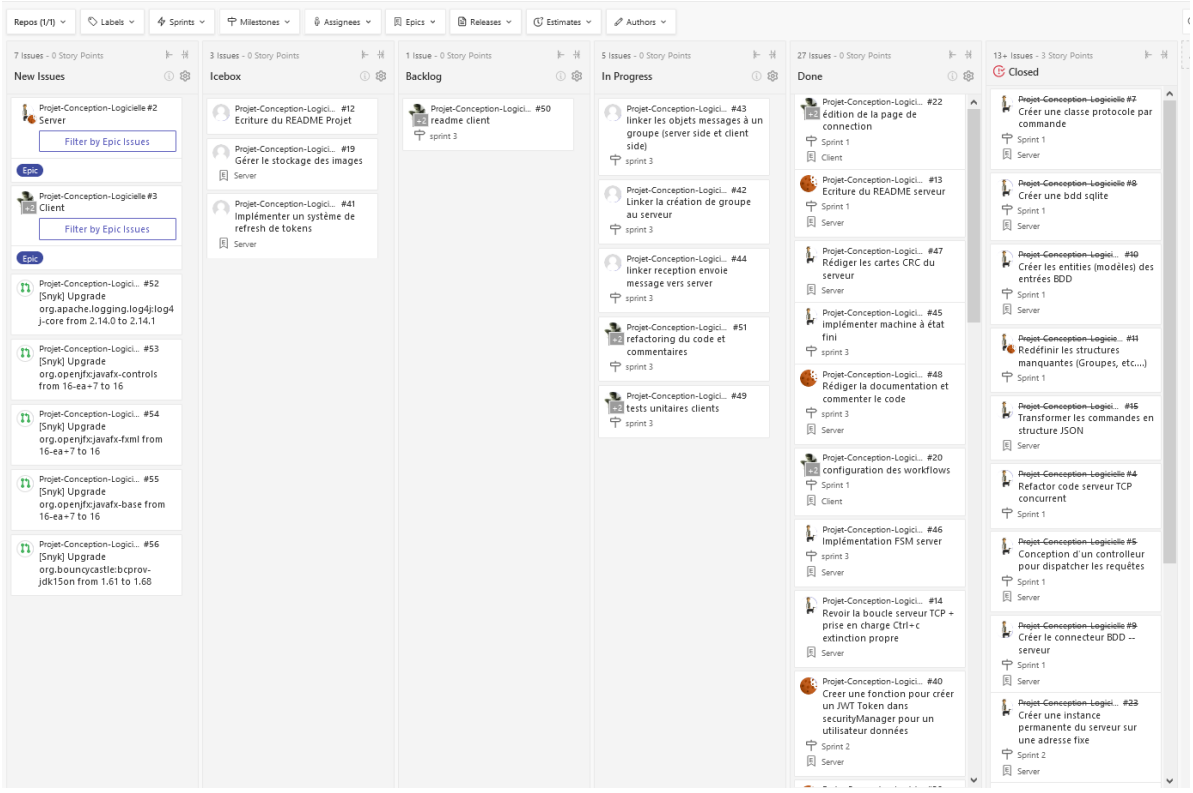
Coverage Breakdown

Package	Class, %	Method, %	Line, %
<empty package name>	0% (0/1)	0% (0/4)	0% (0/11)
fsm	100% (3/3)	87,5% (14/16)	65,3% (47/72)
networking	50% (1/2)	37,5% (3/8)	14,6% (7/48)
pageManagement	0% (0/19)	0% (0/156)	0% (0/780)

generated on 2021-04-08 21:37

Conception: suivi Agile

Le suivi de projet, en mode Agile, a été effectué par l'utilitaire Zenhub.



Il permet une répartition des taches entre les membres du groupe.

Utilisation

génération d'un Jar :

```
mvn clean compile install package
```

Pour le lancement de l'application

```
java -jar target/serveur-xxx.jar # a changer selon le cas
```

Si le binaire à été téléchargé depuis moodle ou github il suffit de lancer :

```
java -jar server-binaire.jar
```

Pensez bien à lancer le jar dans le même dossier que la BDD et le fichier config.json

Pour vous connecter plusieurs utilisateurs sont a votre disposition :

Login	**Password**
test	test
alexandre	alexandre
guillaume	guillaume

> **Attention** : le binaire client à besoin des fichiers ressources (.fxml) pour correctement s'exécuter. Il faut donc le lancer dans le dossier `client/`.

Développement futur

Nous avons implémenté une Machine à États Finis (FSM) sur le serveur pour le protéger contre le doublement de l'information ou le spam de requête.

La machine dispose de 4 états : **idle, sending, receiving, closing**

Nous avons ajouter sur le client une FSM à 3 états : **idle, waiting, sending.**

Cependant nous n'avons pas encore implémenté la FSM dans le client. Toutefois les classes existent et sont testés donc il reste juste l'implémentation.

Il aurait été aussi souhaitable de **hasher** les messages, mais nous sommes limités en caractères pour un message et un hash en augmentant trop la taille.