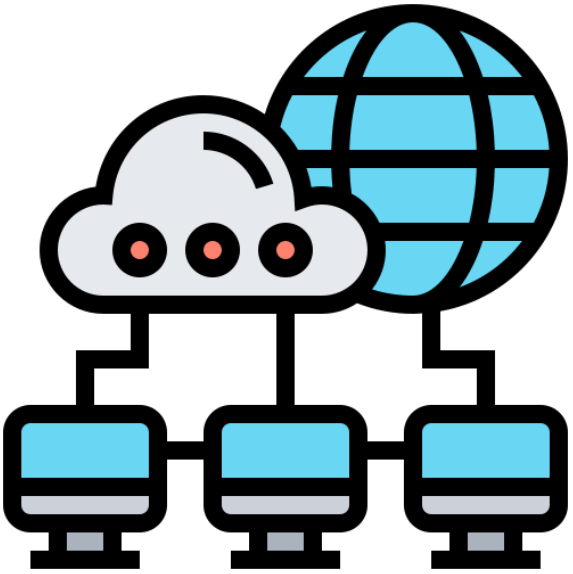
Rapport de projet

STOMP

M1 - MIAGE



William VONDERSCHER

Léo ZANZI

Sofiane CHELH

2021-2022

Introduction

Le but de ce projet est d'implémenter le protocole textuel STOMP, il permet l'interaction entre des clients avec un système de connexion au serveur, d'abonnement/désabonnement à des topics, l'envoi de messages à des topics et recevoir ces derniers lorsque l'on est abonné à ce topic. Dans ce rapport nous allons voir comment nous avons mené le projet.

Nous avons décidé de réaliser une messagerie, les clients qui se connectent au serveur pourront s'abonner à trois sujets différents : Sport, Général et Jeux. Lorsqu'ils enverront un message dans le topic Sport par exemple, alors tous les autres clients abonnés à ce sujet recevront le message.

Architecture du projet

Nous avons réalisé le projet en Typescript avec Node.js. Nous avons fait ce choix car nous sommes habitués à l'utilisation du javascript et nous trouvons qu'il est plus facile à mettre en place qu'un projet Java, surtout avec Node.js. Nous avons utilisé Github pour l'hébergement de notre code.

Dans le but de d'avoir des interactions client/serveur, nous avons utilisé les Websocket.

Notre projet se divise en deux parties. la partie serveur et la partie client. Tout d'abord la partie serveur, elle est dans un fichier nommé "serveur.ts", c'est celui-ci qui va permettre la connexion des clients et la gestion des requêtes STOMP (les recevoir, les traiter et effectuer une réponse appropriée). La partie client qui elle va permettre l'interaction avec le serveur et d'autres clients pour communiquer avec des messages.

Choix techniques

Comme nous l'avons dit précédemment, nous avons utilisé Node.js pour implémenter STOMP.

A. Ce que nous avons implémenté

Pour l'implémentation, nous nous sommes concentrés sur les fonctionnalités de base du protocole, nous n'avons pas implémenté toute la spécificité comme par exemple la gestion des en-têtes optionnels comme les transactions… Nous avons basé notre

- CONNECT or STOMP
  - REQUIRED: accept-version, host
  - OPTIONAL: login, passcode, heart-beat
- CONNECTED
  - REQUIRED: version
  - OPTIONAL: session, server, heart-beat
- SEND
  - REQUIRED: destination
  - OPTIONAL: transaction
- SUBSCRIBE
  - REQUIRED: destination, id
  - OPTIONAL: ack
- UNSUBSCRIBE
  - REQUIRED: id
  - OPTIONAL: none

implémentation sur ce schéma :

donc pour la frame CONNECT, nous avons traité les headers accept-version et host.

a. Côté serveur

A l'aide de la documentation de STOMP nous avons implémenté les différents traitements de requêtes. Nous avons commencé par mettre en place les websockets du serveur avec la création d'une websocket server et sur cette dernière nous ajoutons un listener qui va nous permettre de traiter les messages envoyés par les clients.

Pour gérer les websockets clients, nous avons créé une Map avec en clé l'id du client et en valeur, la websocket. Pour l'id, nous avons modifié l'interface websocket de Node.js :

```
interface ExtWebSocket extends WebSocket {
    id: string;
}
```

On a créé une interface “ExtWebSocket” pour indiquer qu’il s’agit d’une extension de l’interface websocket. Dans celle-ci on ajoute un champ id qui est un string. Maintenant on a un moyen simple de reconnaître les différents clients. Dès qu’un client se connecte au serveur (avec l'événement “connection” du websocket.server, on donne un id au client qu'on génère aléatoirement avec une fonction.

Pour gérer les différents sujets (sport, général & jeux) nous avons aussi créé des Map, une par sujet, nous expliquerons dans les détails plus tard les données que nous stockons dans celles-ci. La clé correspond à l’id du client et la valeur à l’id de son abonnement à ce sujet.

Le but du serveur est de traiter les requêtes des clients et le traitement est assez similaire pour chaque requête. On commence d’abord par regarder le type de requête qu’on reçoit, c’est le but de la fonction “typeRequete” dans notre code. On découpe la requête en plusieurs parties, par rapport aux fins de lignes, c’est à dire que dans l’exemple :

REQUETE

ABC:DEF

Le découpage (split) permet d’avoir dans un tableau les éléments “REQUETE”,”ABC:DEF”. Donc la première chose à faire est de regarder la commande de la frame :

CONNECT

La frame connect permet à un client d’initier le flux entre le client et le serveur, dans notre cas, le client est déjà connecté au serveur, mais dans le but d’accéder aux fonctionnalités du protocole STOMP, il doit commencer par se connecter :

```
CONNECT
accept-version:1.2
host:stomp.github.org

^@
```

Toutes les frames sont formés de cette manière, on à le nom de la commande que le client souhaite effectuer, ici c'est CONNECT, maintenant que nous savons que le client souhaite se connecter, nous vérifions d'abord qu'il ne l'est pas déjà, pour cela nous avons créé une liste de websocket de clients, donc si la websocket qui a envoyé la requête de connexion est déjà dans la liste, pas besoin de la réinscrire dedans, si c'est le cas, on envoie une erreur au client (nous y reviendrons plus tard). Si le client n'est pas connecté, alors on va continuer de traiter la requête. Nous allons maintenant regarder les en têtes, elles sont sous la forme de clé:valeur. Pour que les en-têtes soient valide, il faut qu'elles soient dans le bon ordre (en premier accept-version et en deuxième host). Nous commençons par vérifier que le premier en-tête est accept-version, on découpe l'en-tête par rapport au caractère ":", la première partie avant ":" doit être "accept-version" et la partie après doit être "1.0", "1.1" ou "1.2", si c'est le cas, on vérifie le deuxième en-tête host de la même manière, dans notre cas, la partie après ";" doit être "localhost" car nous sommes en local sur la machine. Si les deux en-têtes sont bien formés, alors on ajoute le client à la map de clients si un problème est détecté avec la formation de la requête, on envoie une erreur.

DISCONNECT

Lorsque le client souhaite partir du site, il envoie une requête DISCONNECT au serveur :

```
DISCONNECT
receipt:77
^@
```

Dans le traitement, on commence par vérifier qu'il est dans la Map des clients connectés aux serveurs, sinon cela veut dire qu'il n'est pas connecté de base. La particularité de cette requête, c'est qu'elle n'a pas d'en-tête obligatoire, alors le client peut juste envoyer "DISCONNECT" avec "^@" à la fin pour dire que la requête est terminée et il sera déconnecté. Il y a un en-tête optionnel "receipt",il permet au client de recevoir un identifiant qu'il a passé à la requête dans le but de confirmer que sa requête est bien traitée. Nous avons décidé d'implémenter la possibilité d'utiliser cet en-tête. Pour vérifier, on regarde si dans la requête, il y a un élément qui commence par "receipt:" (avec la fonction find de javascript) alors on renvoie une frame RECEIPT qui contient un en-tête "receipt-id" qui aura comme valeur ce que le client avait mit dans l'en-tête "receipt" de la requête DISCONNECT.

On supprime le client de la Map des connectés du serveur ainsi que de chacune des Map sujets (on vérifie s'ils en font partie et si c'est le cas, on le supprime.

Le client est alors déconnecté !

SUBSCRIBE

Un client qui souhaite s'abonner à un sujet va envoyer une frame SUBSCRIBE au serveur :

```
SUBSCRIBE
id:0
destination:/queue/foo
ack:client

^@
```

Pour cette requête nous avons traité les deux en-têtes requis id et destination. On effectue une vérification similaire à la frame CONNECT où l'on coupe les en-têtes en deux parties (par rapport a ":"). Le premier est l'id, il correspond à l'identifiant de l'abonnement qui est unique, c'est à dire que chaque abonnement du client à un identifiant unique. Par exemple. le client s'abonne en premier à au topic jeux avec un id = 0, il veut ensuite s'abonner au topic général, il devra utiliser un id autre que 0, comme 1. Donc on vérifie que le premier argument est bien id et que la valeur n'est pas déjà un abonnement du client qui existe. La vérification se fait dans les Map des topics, ces dernières stockent comme clé : l'id du client et en valeur : l'id de l'abonnement.

On regarde aussi si le deuxième en-tête "destination" est bien formé. Il correspond au topic de destination, dans notre projet, une valeur correct serait : "topic/jeux", ici on veut s'abonner au sujet jeux.

Une fois que tout est analysé, on peut ajouter l'abonnement dans la map qui correspond au topic de la destination.

Si le client dont l'identifiant est "toto" envoie une frame SUBSCRIBE avec en id = 5 (qui n'existe pas déjà) et la destination "topic/general", alors on ajoute à la map correspondant au sujet général : la clé "toto" et la valeur 5. Le client est maintenant abonné au topic !

UNSUBSCRIBE

Un client qui souhaite se désinscrire d'un topic peut utiliser la frame UNSUBSCRIBE pour réaliser l'action :

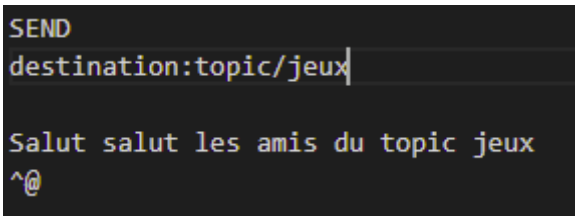
```
UNSUBSCRIBE
id:0

^@
```

C'est une frame très simple, le client donne une en-tête qui contient l'identifiant unique de son abonnement. Par exemple, le client est abonné au topic jeux avec l'id = 5, alors il va utiliser id:5 dans la frame ci-dessus.

Dans le fonction qui traite la frame, on commence par regarder si l'en-tête "id" est présent, si c'est le cas, on regarde dans les 3 maps des topics si la clé contenant l'id du client existe, et si cela correspond à l'id de l'abonnement, si on réunit ces deux conditions alors on supprime le couple de la map. Le client est alors désinscrit.

SEND

Un client qui souhaite envoyer un message aux abonnés d'un topic, va envoyer une frame SEND au serveur :



Avec le serveur nous allons traiter cette requête dans le but de renvoyer le message à tous les clients abonnés au topic précisé. L'en-tête que nous regardons est destination, comme pour la frame SUBSCRIBE c'est là ou le message doit aller. On récupère alors cette destination et on la stock dans une variable.

Le but maintenant c'est de récupérer le message, c'est le body du message (la deuxième partie) ici on sépare les en-têtes au body grâce à un saut de ligne. Dans l'exemple du dessus le body est "salut salut les amis du topic jeux". Pour détecter le body, nous avons considéré qu'il se trouve après le saut de ligne. On rappelle que la requete que nous traitons dans nos fonctions est une liste dont la séparation est la fin de ligne, donc chaque élément de la liste est une ligne de la frame. On regarde dans cette liste, la première ligne qui contient "" donc la ligne vide. A partir de cet élément, on rassemble le reste pour remettre le body en forme. Pour finir on regarde si les deux derniers caractères du body sont "^@" (élément de fin de frame) s' ils sont présents alors on les enlève sinon c'est que la requête est mal formée et on envoie une erreur.

Maintenant que nous avons le message a envoyer et la destination, on vérifie si la destination existe et on envoie une frame MESSAGE aux clients (on itère sur chaque clients de la map du topic concerné et on envoie le message).

MESSAGE

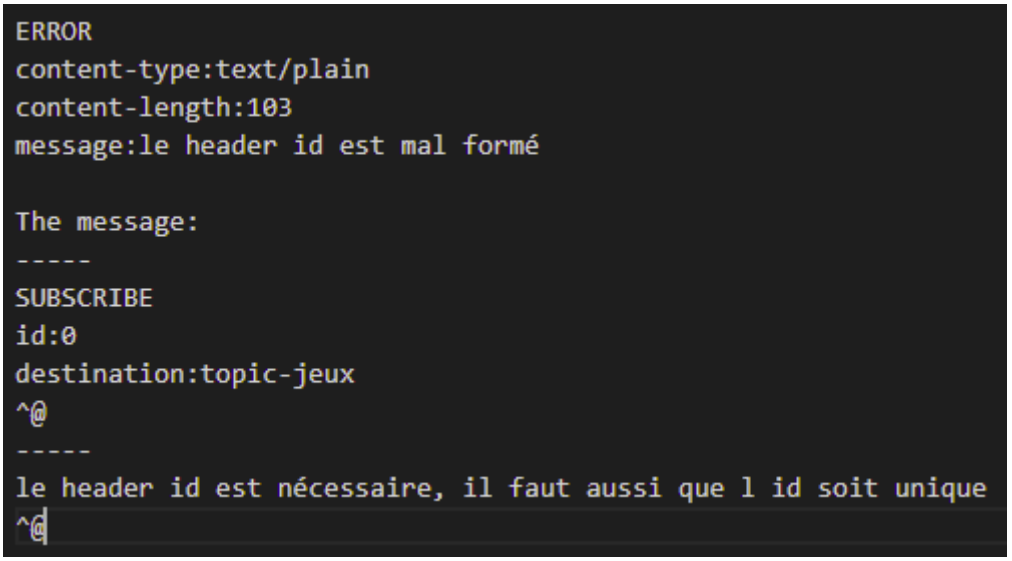
La frame MESSAGE est envoyée par le serveur aux clients. Elle est directement envoyée après la réception d'une frame SEND :

```
MESSAGE
subscription:0
message-id:007
destination:/queue/a
content-type:text/plain

hello queue a^@
```

Cette frame à comme premier en-tête "subscription" qui est l'id d'abonnement unique du client destinataire, on récupère cette information quand on traite la frame SEND (point précédent) comme on itère sur la map des clients abonnés au topic, on l'information pour chaque client de l'id de leur abonnement au topic. Le deuxième en-tête est "message-id", c'est un identifiant unique qui est attribué au message, dans le code du serveur, nous avons avons un entier dont la valeur est 0, puis pour chaque message envoyé, on incrémente de 1, comme cela on a un id unique pour chaque message. L'en-tête destination contient le topic concerné par le message et le dernier en-tête content-type indique qu'on utilise du texte pour le bod. Finalement, on a le body du message qui correspond au même body que celui de la frame SEND. Il n'y a pas de traitement particulier, on remplit juste les différents éléments qu'on a besoin (subscription, message-id, destination & body) puis on les envoie aux clients destinataires.

ERROR

La dernière frame que nous avons implémenté est la frame ERROR, celle-ci est envoyé du serveur au client lorsqu'un problème arrive lors du traitement des frames :



Ci-dessus un exemple du contenu de la frame ERROR, dans cet exemple il s'agit d'une erreur qu'on retourne lorsque le client envoie une frame SUBSCRIBE avec un id qui existe déjà pour ses abonnements.Les deux premiers en-têtes sont des informations sur le texte, c'est la première fois que nous voyons l'en-tête "content-length" il contient le nombre de caractères du body. Le dernier en-tête "message" est une courte description du problème. Dans le body, on indique la frame qui a posé problème (ici la frame SUBSCRIBE) et en dessous on met un message d'erreur plus développé.

Nous avons vu tout ce que nous avons implémenté de STOMP dans cette partie, nous allons maintenant voir comment fonctionne le client.

b. Côté client

En ce qui concerne l'aspect client, que nous avons développé dans un second temps, nous avons choisi d'élaborer une interface assez simple d'utilisation pour les utilisateurs.

CONNECT & DISCONNECT

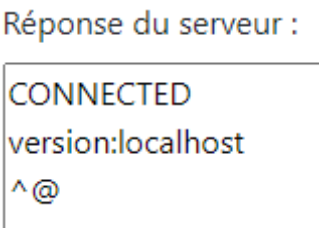
L'interface comprend un bouton qui permet de se connecter et se déconnecter au serveur, pour ces deux actions, nous avons fait le choix de mettre en place les actions en cliquant sur les boutons. Les requêtes sont envoyées au serveur en cliquant sur les boutons et non en écrivant en brut la frame connexion. :

Lorsqu'on clique sur le bouton "Se connecter" alors une action se déclenche en faisant l'appel au serveur. Le WebSocket récupère alors en paramètre l'adresse que nous avons mis dans le champ (ici localhost) puis il est envoyé au serveur la frame de connexion expliqué dans la partie précédente :

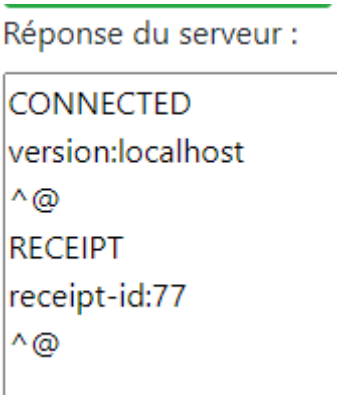
```
$("#send").click(function(){
    socket = new WebSocket($("#url").val());
    socket.onopen = function(event){
        socket.send("CONNECT\n" +
            "accept-version:1.2\n" +
            "host:localhost\n" +
            "\n" +
            "^@");
    }
```

Nous avons mis en place un TextArea qui retourne les réponses du serveur.

Lorsque la connexion s'est bien déroulé alors nous avons le message de retour du serveur nous indiquant que la connexion s'est bien déroulé :



De même pour la déconnexion au serveur :

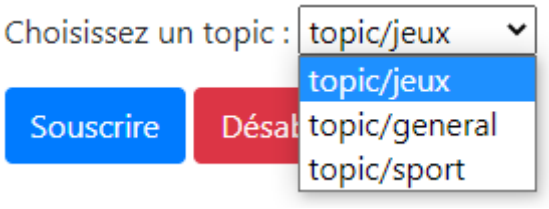


SEND

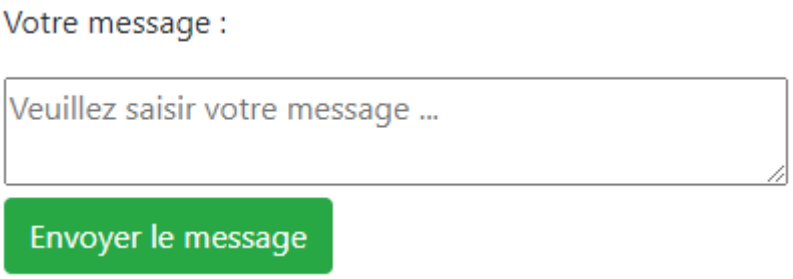
En ce qui concerne la frame SEND, elle fait appel au message et au topic.

Il fallait récupérer ces deux variables:

Pour le topic nous avons choisi de mettre une liste déroulante qui comprend les 3 topics (général, sport et jeux). Nous avons fait ce choix car cela nous permettait d'éviter les erreurs que l'utilisateur pouvait rencontrer en écrivant un topic qui n'existait pas par exemple.



Pour le message, c'est un textarea qui récupère le message puis le stocke dans une variable.

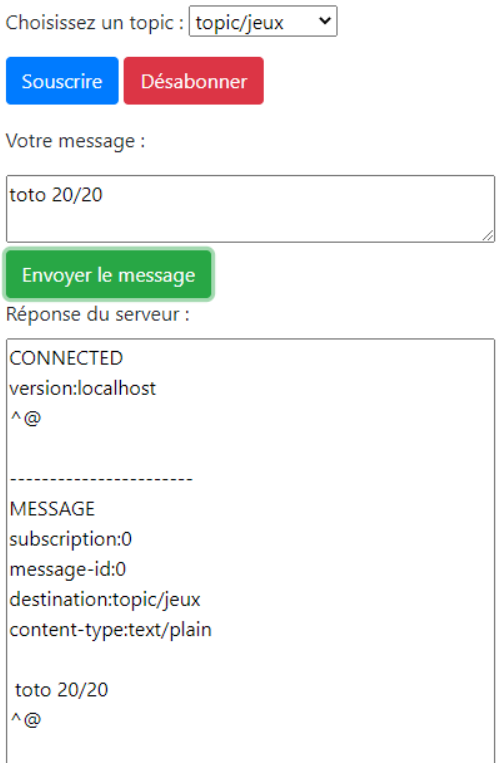


Au niveau de l'implémentation, pour cette fonctionnalité nous avons :

```
$("#envoyer").click(function(){
    socket.send("SEND\n" +
        "destination:" + $("#topic-select").val() +
        "\n" +
        "content-type:text/plain\n" +
        "\n" +
        $("#message").val() +
        "^@");
    socket.onmessage = function(event){
        $("#responseServer").append("\n----------------------\n");
        $("#responseServer").append(event.data);
        console.log(event.data);
    }
})
```

Lorsque l'utilisateur clique sur le bouton envoyer le message, la destination est récupérée grâce au select qui permet de savoir sur quel topic l'utilisateur souhaite envoyer le message.

Puis le message est ajouté juste avant le ^@.



Bien évidemment si l'utilisateur n'a pas souscrit au topic alors il lui sera impossible d'envoyer un message à ce topic en question. De même pour l'utilisateur s'étant désabonné d'un topic.

SUBSCRIBE / UNSUBSCRIBE

Pour ce qui est de la fonctionnalité subscribe, nous avons mis en place un item sélectif permettant de choisir les topics auxquels vous voulez accéder.

Il vous suffit juste de choisir le topic que vous souhaitez et de cliquer sur "souscrire".

Pour ce qui est du désabonnement, c'est exactement le même principe, vous choisissez le topic et vous cliquer sur désabonner. Pour ce qui est du code :

```
$("#subscribe").click(function(){
    socket.send("SUBSCRIBE\n" +
        "id:" + idTopics + "\n" +
        "destination:" + $("#topic-select").val() +
        "\n" +
        "^@");
    map.set($("#topic-select").val(), idTopics);
    idTopics++;
})

$("#unsubscribe").click(function(){
    socket.send("UNSUBSCRIBE\n" +
        "id:" + map.get($("#topic-select").val()) + "\n" +
        "\n" +
        "^@");
})
```

idTopics est une variable jouant le rôle d'identifiant, elle est incrémentée à chaque fois que le client souscrit à un abonnement pour éviter d'avoir plusieurs fois le même id.

Pour ce qui est de "destination:" nous récupérons tout simplement le texte qui est dans l'item select (soit topic/jeux, topic/general ou topic/sport).

Une fois l'envoie de la requête, nous utilisons un map permettant de lier le nom du topic avec l'identifiant qui lui a été attribué et nous incrémentons notre idTopics (pour éviter les doublons comme dit précédemment).

Pour ce qui est du désabonnement, c'est tout simple, nous envoyant notre requête en récupérant l'identifiant associé au topic sélectionner (d'où l'intérêt d'avoir utilisé un mapping clé/valeur (dans notre cas nom de topic / identifiant)).

B. Ce que nous n'avons pas implémenté

Nous n'avons pas implémenté toute la spécificité de STOMP, tout d'abord dans les frames que nous avons implémenté, tous les en-têtes ne sont pas pris en compte, nous avons utilisé les en-têtes obligatoires pour les frames. par exemple les en-têtes d'authentification.

ACK/NACK

ACK permet de reconnaître le bon traitement d'un message. Cette frame contient un id qui correspond à l'id du message qui est concerné. On peut aussi ajouter l'en-tête transaction si jamais la frame doit faire partie d'une transaction.

NACK est l'inverse de ACK, il permet de dire au serveur que le client n'a pas eu le message. Elle a les mêmes en-têtes que ACK.

RECEIPT

Nous avons implémenté cette frame pour la frame DISCONNECT, elle permet au client de recevoir une confirmation que la frame qu'il a envoyé a bien été traité (en-tête receipt a mettre dans la frame dont l'on souhaite avoir un retour). Elle a en en-tête l'id du receipt qui été en en-tête de la frame que le client a envoyé avant.

TRANSACTION

Les transactions permettent de s'assurer de l'atomicité d'une suite de messages (tout est envoyé ou rien du tout). il y a trois frames qui sont en rapport avec les transactions : BEGIN qui permet de commencer une transaction dans laquelle on y met le header "transaction" qui correspond à l'id de la transaction ; il y a COMMIT qui marque la fin d'une transaction et enfin ABORT qui permet de revenir à l'état de début de transaction.

Les difficultés rencontrées

A. Le serveur

Lors du développement du serveur, nous avons rencontré quelques difficultés. Tout d'abord pour le stockage des clients, à la base nous souhaitions avoir une simple liste des clients qui étaient connectés au serveur. Le problème c'était pour différencier les websockets clients entre elles. Pour régler cette complication, nous avons décidé d'écrire une interface qui dérive de la classe Websocket, en réalisant cela on peut ajouter des attributs, dans notre cas on ajoute un id qui est un string et cet id est unique et est donné dès qu'un client se connecte au serveur. Maintenant nous pouvons manipuler les websockets plus facilement.

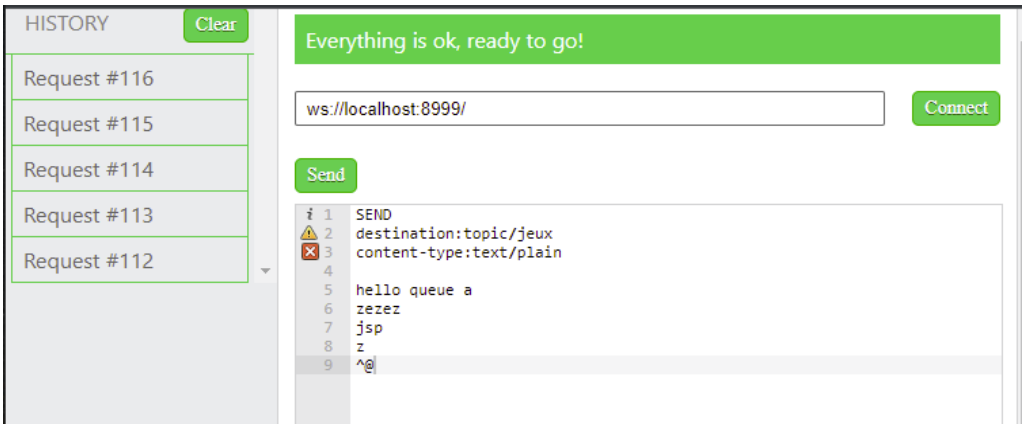
Un autre problème concerne les différents topics auxquels on peut s'abonner et comment on gère les abonnements. On rappelle que quand un client s'abonne a un topic, il le fait avec un id unique ce qui pose problème puisque d'après nous, le plus simple c'est de seulement créer une liste par topic et d'ajouter le client dès qu'il s'abonne. Dans notre cas il faut stocker l'id de l'abonnement, pour régler ce problème nous avons créé une map par topic, cette map a pour clé l'id du client et en valeur l'id de l'abonnement, cela nous permet de connaître le client et son id d'abonnement pour les topics.

Répartitions des tâches

Au début de projet, nous avons commencé par travailler sur la partie serveur car nous utilisions l'extension Smart Websocket Client pour simuler un client. Nous avons travaillé ensemble au début sur le serveur car nous devions comprendre comment le projet fonctionnait. Nous avons développé ensemble les fonctionnalités suivantes : connexion, déconnexion du serveur, ainsi que l'abonnement et désabonnement à un topic. William a terminé le serveur avec la gestion de la réception des messages client et l'envoie de ces derniers aux clients d'un topic. Pour finir Léo et Sofiane ont développé le client qui permet l'envoie des frames/messages au serveur.

Tests du projet

Dans un premier temps, lorsque nous avons réalisé le projet, nous avons commencé par développer le serveur, nous avons utilisé une extension sur Google Chrome appelée Smart Websocket Client. Cette extension nous a permis de simuler un client tout au long du développement, à chaque fois que nous avions fini de traiter une frame on la testait :



Tout d'abord on commence par indiquer l'adresse du serveur websocket, ici il est en localhost sur le port 8999, on clique sur le bouton "connect" et si tout est bon, il indique qu'il est bien connecté au serveur. Ensuite grâce à la zone de texte en dessous, on peut écrire nos requêtes et appuyer sur le bouton "Send" et le message est envoyé au serveur. Il y a une autre zone de texte en dessous avec la réponse qui sera renvoyée du serveur au client. Pour tester le multi-client, il suffit d'ouvrir plusieurs chrome et d'utiliser l'extension, chaque chrome sera un client différent.

Lorsque nous avons développé le client, nous avons fait le reste des tests avec notre propre client. Pour tester les requêtes en général, nous avons testé le plus de cas possibles pour s'assurer que le serveur traite correctement les requêtes. Par exemple, on s'assure qu'aucune requête ne fonctionne si le client ne s'est pas connecté au préalable avec la frame CONNECT. Pour le multi-client, nous avons ouverts plusieurs chrome avec l'extension et avec les clients on s'est inscrit à différents topics et lorsqu'un des clients envoie un message sur un topic où plusieurs autres clients sont inscrits, alors on doit bien avoir les messages dans la zone de texte retour des clients.

Utilisation créative du protocole STOMP

Le protocole STOMP pourrait être utilisé comme une sorte de forum web pour des personnes jouant à des jeux vidéo. La connexion permettrait de s'identifier et donc de ne pas être anonyme, le fait de pouvoir s'abonner à des topics / sujets permettrait aux utilisateurs de suivre le contenu qu'ils aiment et le fait de pouvoir envoyer des messages leur permettrait d'interagir avec les autres utilisateurs.