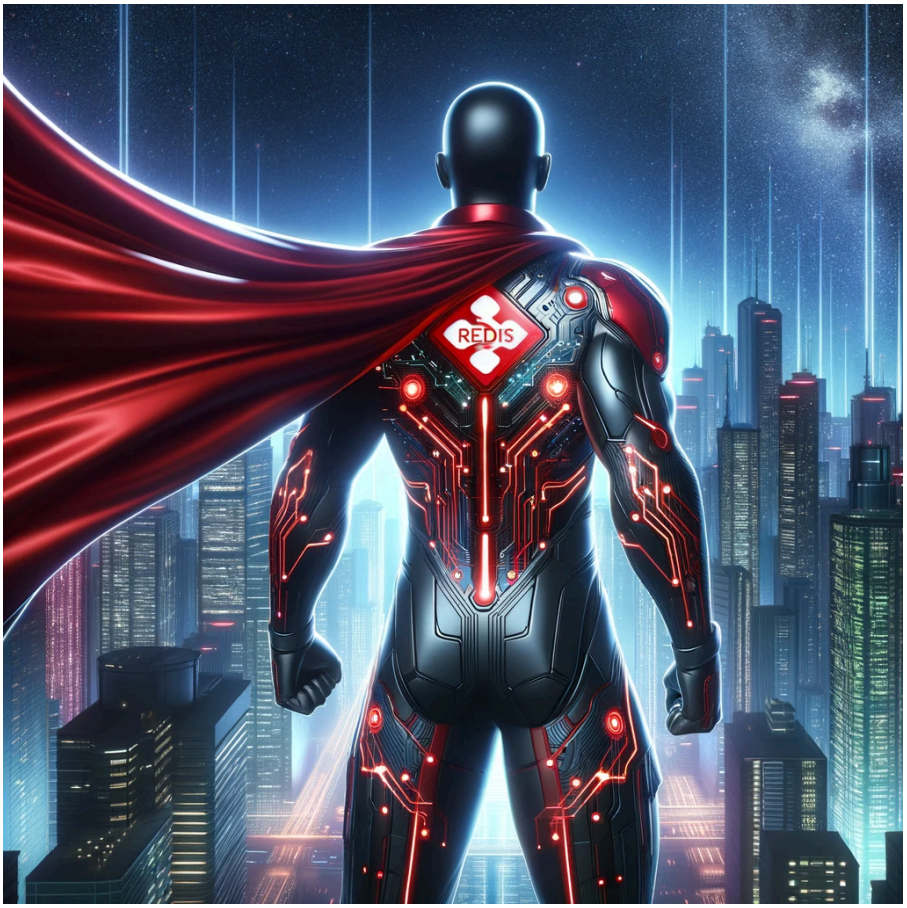
Redis pour votre architecture de messagerie Discord.



Objectifs du TP :

- Mise en place d'un base de données Redis.
- Créer un client Java Swing pour l'échange de messages.
- Manipuler le message brocker de Redis.
- Manipuler le stockage clé/valeur de Redis.

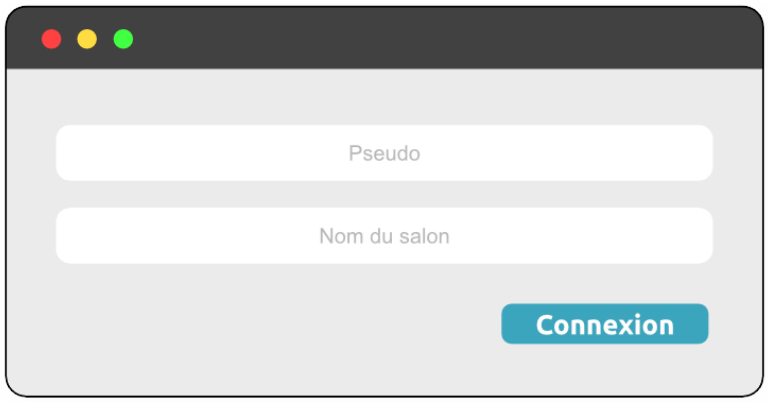
Partie 1 : Prérequis et présentation du projet

Pour ce TP, vous allez réaliser une application semblable à Discord. L'application a pour but d'échanger des messages avec des utilisateurs se trouvant dans le même salon de discussion.

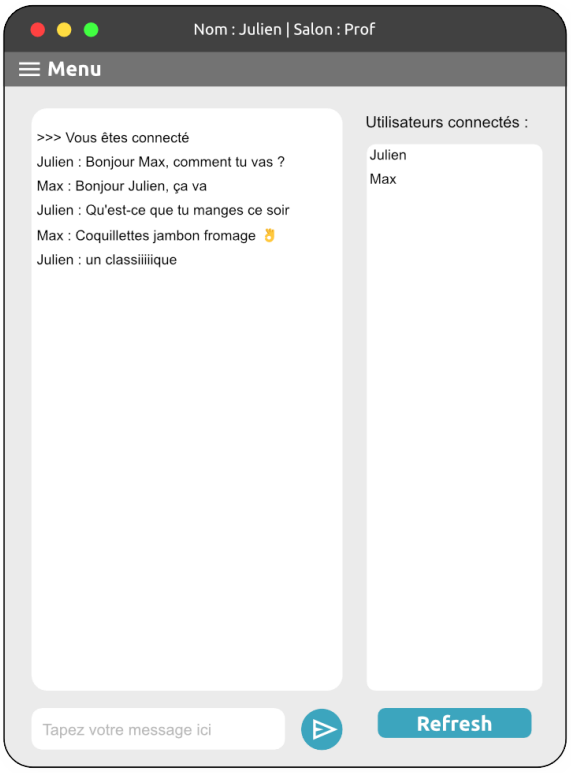
Pour pouvoir réaliser ce TP, vous devez remplir quelques prérequis :

- Avoir installé une version Java sur votre poste
- Avoir réalisé le TP Redis précédent

Vous devez donc créer une nouveau projet Java Maven en suivant les étapes présentées dans le précédent TP. Votre travail s'attachera à reproduire les maquettes présentées ci-dessous en utilisant les composants Swing.



Interface de connexion d'un utilisateur pour rejoindre un salon. Celle-ci se décompose en deux champs : un pour le pseudo, l'autre pour le nom du salon. L'interface se termine par un bouton de connexion qui ouvre la fenêtre de dialogue.



<u>Interface de dialogue avec les utilisateurs ayant rejoint le salon.</u> Celle-ci possède un titre de fenêtre comprenant le nom de l'utilisateur ainsi que le nom du salon.

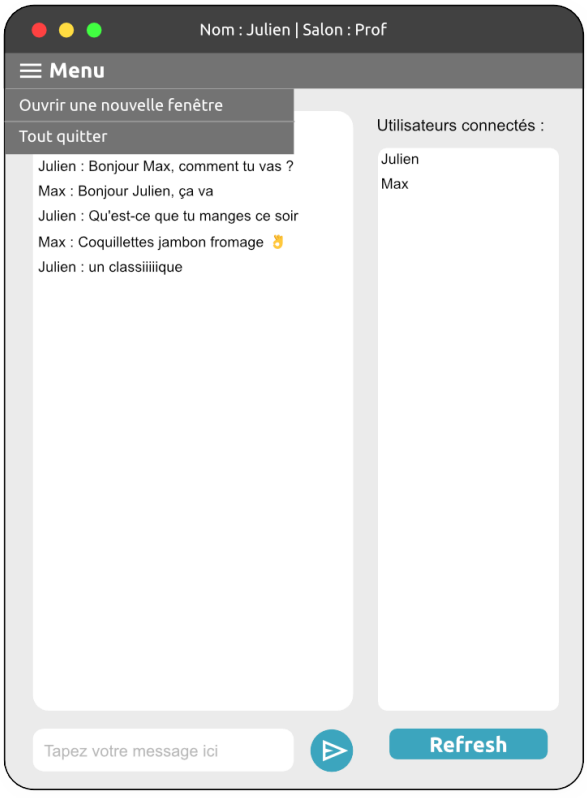
On trouve ensuite les éléments permettant le dialogue avec : un champ texte en lecture uniquement contenant les messages, un champ texte éditable permettant de rédiger un message et pour terminer, le bouton pour envoyer le texte.

Sur la droite de la fenêtre, on retrouve la liste de tous les utilisateurs connectés sur le serveur (tous les canaux confondus). Le bouton « refresh » en dessous permet d'actualiser les valeurs de cette liste.

En haut de notre fenêtre se trouve un menu. Lors d'un clic dessus, une liste d'actions déroulante s'affiche avec deux options :

« Ouvrir une nouvelle fenêtre » permet d'afficher une fenêtre de connexion afin d'ouvrir une nouvelle connexion sur le serveur.

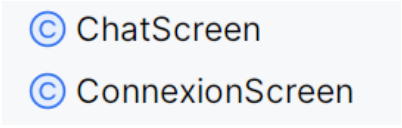
« Tout quitter » doit permettre de fermer toutes les fenêtres ouvertes.



Ce TP est organisé de façon incrémentale dans la réalisation de l'application. Nous vous conseillons de ne pas sauter des étapes et de développer uniquement les fonctionnalités demandées.

Partie 2 : « *Bienvenue dans le monde réel* »

Pour cette première partie, nous allons commencer par les bases de l'application. Codez en Java Swing les interfaces graphiques. Vous ne coderez que les éléments visuels sans implémenter les comportements. Les deux interfaces graphiques donneront naissance à deux classes différentes. Vous devez créer un fichier main afin de visualiser ces interfaces. L'image ci-dessous illustre le résultat obtenu :



Maintenant que vos interfaces graphiques sont finalisées, nous allons attaquer la première brique logique de l'application. Celle-ci consistera à publier un message sur le message broker de Redis. Vous modifierez en conséquence « ConnexionScreen » afin de permettre à l'utilisateur de publier dans un salon défini. Voici la User Story associée :

> En tant qu'utilisateur identifié et connecté dans un salon, lorsque je rentre un message et que je clique sur le bouton envoyer, un message est publié sur le broker Redis.

Pour réaliser cette US (User Story), nous vous conseillons :

1. De fixer, dans un premier temps, en « dur » dans le code le nom de l'utilisateur et le nom du salon. Nous rendrons ça dynamique plus tard.
2. De vous inspirer du code « Publisher » que vous avez vu dans le précédent TP
3. D'utiliser le «AppEcouteur » du précédent TP pour vérifier que votre envoi fonctionne

Une fois le publisher implémenté avec succès, vous allez vous attaquer au listener. L'idée est de boucler la boucle en affichant les messages envoyés. Vous vous inspirerez encore une fois du TP précédent. Voici les deux US :

> En tant qu'utilisateur identifié et connecté dans un salon, lorsque la connexion avec redis est établie, le message suivant est affiché dans le chat: « >>> Vous êtes connecté »

> En tant qu'utilisateur identifié et connecté dans un salon, lorsque qu'un message est publié dans ce même salon, le message est affiché en dernier dans la liste des messages. Le message aura la forme suivante : *pseudo* : *message*

Partie 3 : « Un grand pouvoir implique de grandes responsabilités. »

Dans cette nouvelle partie, nous allons rendre dynamique le pseudo et le nom du salon dans « ChatScreen ». Cela va ainsi permettre à notre programme d'ouvrir plusieurs connexions avec notre serveur Redis. L'utilisateur doit ainsi pouvoir se connecter avec un autre pseudo dans des salons différents. Voici les US que vous devez intégrer :

En tant qu'utilisateur non connecté, lorsque je lance l'application pour la première fois, je dois voir la fenêtre de connexion.

En tant qu'utilisateur non connecté, lorsque je valide mes informations de connexion, je dois rejoindre la fenêtre de chat avec les informations entrées.

En tant qu'utilisateur déjà connecté dans un salon, lorsque je clique sur « Ouvrir une nouvelle fenêtre », je dois voir une nouvelle fenêtre de connexion.

Nous allons maintenant aborder la liste des utilisateurs connectés sur le serveur. La liste doit contenir le pseudo de toutes les personnes qui sont présentes dans au moins un salon. Voici les US que vous devez intégrer :

En tant qu'utilisateur connecté dans un salon, lorsque je rejoins le salon ou appuis sur le bouton refresh, je dois voir mon pseudo ainsi que toutes les personnes connectées sur le serveur.

En tant qu'utilisateur connecté dans un salon, lorsque je me déconnecte du salon, mon pseudo doit disparaître de la liste des personnes connectées.

En tant qu'utilisateur connecté dans deux salons différents, lors de ma déconnexion du premier salon, mon pseudo restera dans la liste des utilisateurs étant donné que j'ai toujours une connexion d'ouverte dans le deuxième salon.

Pour implémenter cette fonctionnalité, vous utiliserez le stockage de Redis. Notamment pour la clé « users » vous accéderez à un set dans lequel vous ajouterez et enlèverez des pseudos (sous forme de string) suivant les cas présentés ci-dessus.

Nous supposerons qu'il n'est pas possible d'avoir deux utilisateurs différents avec le même pseudo.

Pour terminer cette partie, nous allons implémenter le comportement du bouton « Tout quitter ». Voici l'US que vous devez intégrer :

En tant qu'utilisateur connecté dans un ou plusieurs salons, lorsque je clique sur « tout fermer », toutes les fenêtres doivent se fermer et ainsi terminer le programme.

La fermeture de toutes les fenêtres respectera le comportement de déconnexion.

Partie 4 : « *Tout le monde peut cuisiner, mais le véritable génie, n'appartient qu'aux audacieux.* »

Lors d'échanges d'informations entre deux systèmes, il est important d'utiliser le même formalisme. En effet, si l'API s'attend à recevoir deux champs qui sont : identifiant et mot de passe, vous ne devez pas être en capacité d'envoyer, par exemple, uniquement une date.

Dans le cas ou vous ne feriez pas ces contrôles, un ensemble de comportements non désirés pourraient survenir. Allant de la simple erreur si un élément attendu n'existe pas, jusqu'à l'injection de code malveillant.

> Pour aller plus loin, vous pouvez explorer la faille Log4j qui permet de prendre le contrôle d'une machine à distance. Cette vulnérabilité en date du 10 décembre 2021 aura fait trembler internet, illustrant à la perfection une faille de sécurité dans l'échange de données entre deux systèmes.

Pour assurer un contrôle sur les échanges d'informations, les développeurs ont donc créés différents outils répondant aux problématiques précédentes. L'un d'entre eux et le Data Transfert Object (DTO) : il s'agit d'une classe (Java dans notre cas) représentant le formalisme d'échange entre les différents acteurs. Cette classe fait office de contrat entre les acteurs lors d'un échange d'informations.

On vous propose donc dans cette partie d'implémenter le fonctionnement des DTO pour l'échange de messages dans votre application. Pour faire cela, il vous faudra créer une classe avec les attributs suivants :

- « Eméteur » qui sera un String représentant le pseudo de l'utilisateur
- « Contenu » qui sera un String et contiendra le message

Via Maven, vous utiliserez la librairie Gson développée par Google afin de sérialiser et dé-sérialiser les instances de votre classe DTO vers du JSON. Le JSON sera le contenu à envoyer sur le message broker de Redis.