

# *Projet : ChatLPSIC*

## *Introduction*

ChatLPSIC est une application client-serveur développée en Python qui permet à différentes personnes (clients) connectées à un même serveur de s'envoyer des messages.

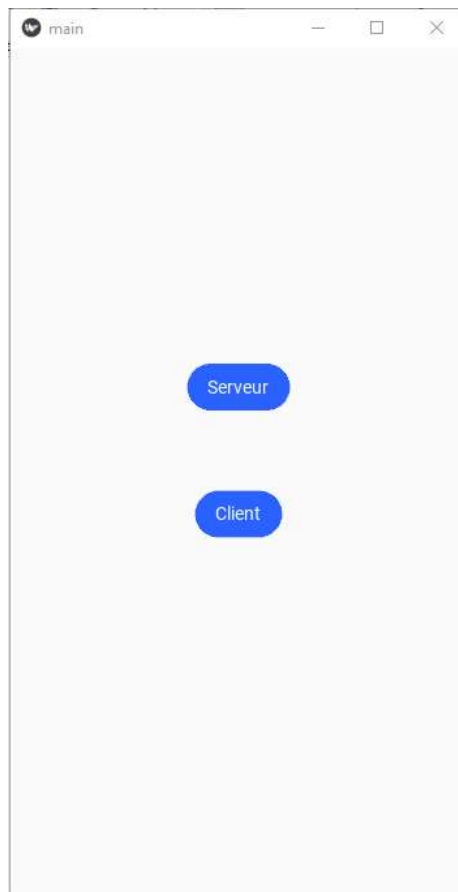
Pour cela, il est nécessaire de se trouver dans le même réseau, que ce soit en local ou sur Internet.

## *Fonctionnement de l'application : en Front-end*

### Accueil

Au début, l'application nous présente 2 boutons :

- Le bouton serveur : pour la configuration du serveur auquel les clients se connecteront.
- Le bouton client : pour se connecter au serveur.



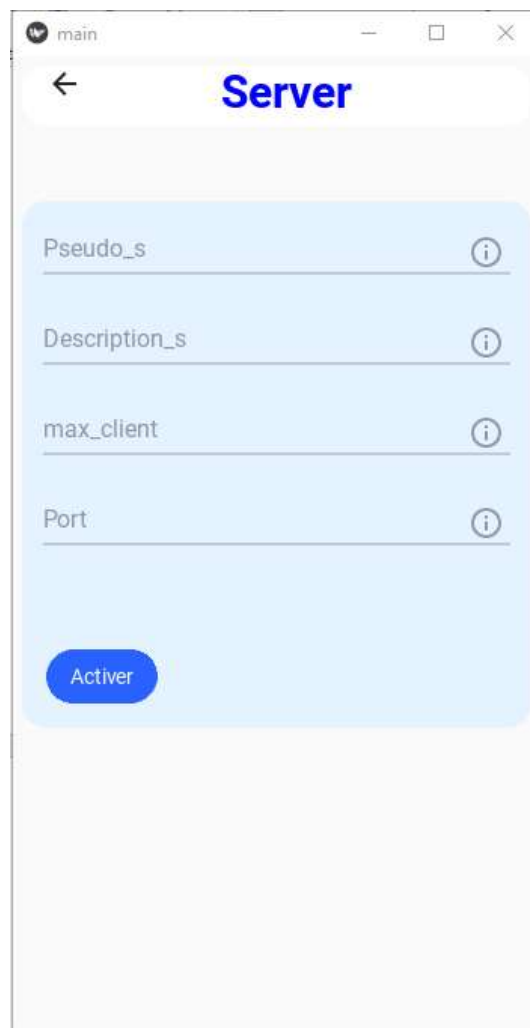
## Configuration du Serveur

Pour la configuration du serveur, on renseigne :

- Pseudo\_s : le pseudonyme du serveur dans le réseau, car il peut y avoir plusieurs serveurs dans un même réseau.
- Description\_s : la description du serveur, sa spécificité.
- Max\_client : le nombre maximal de personnes qui peuvent se connecter au serveur. Les expressions régulières (regex) de cet input fixent le minimum à 2 et le maximum à 16.
- Port : le port du serveur.

Il est à noter l'absence de l'adresse IP. Cela est dû au fait que l'adresse IP de l'appareil sur lequel le serveur est activé est récupérée directement depuis l'appareil.

Il ne faut pas oublier que chaque champ de saisie possède des expressions régulières spécifiques qui régulent les entrées.



The screenshot shows a mobile application window titled "main" with a "Server" configuration screen. The screen has a light blue background and a white header bar with a back arrow and the title "Server". Below the header, there are four input fields, each with a label and an information icon (i) on the right:

- Pseudo\_s
- Description\_s
- max\_client
- Port

At the bottom of the form, there is a blue button labeled "Activer".

## Information du serveur

Après la configuration du serveur, les informations du serveur sont affichées à l'écran suivant.



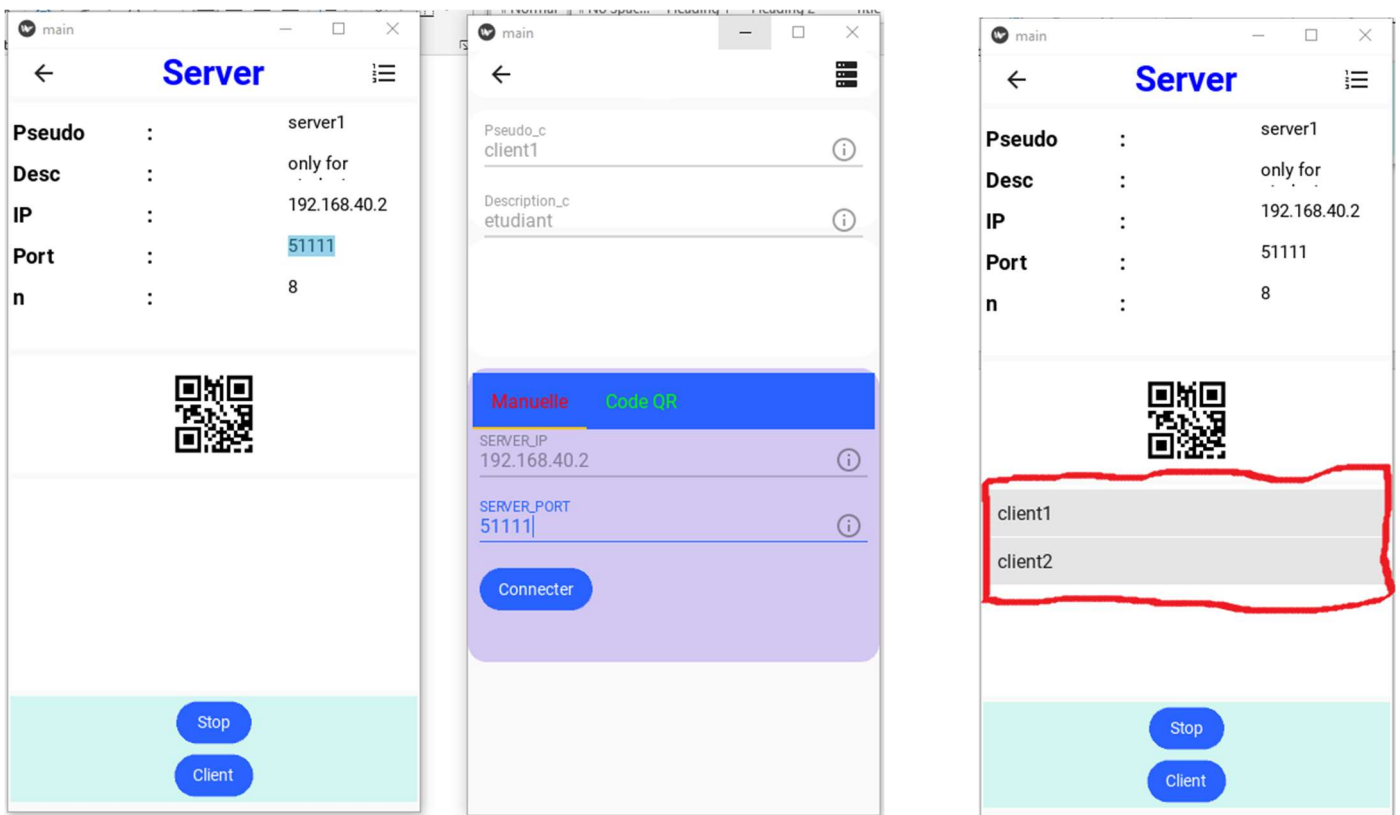
On a le choix d'arrêter le serveur avec le bouton 'Stop' ou de se connecter au serveur en tant que client via le bouton 'Client'.

## Connexion au serveur

On renseigne :

- Pseudo\_c : pour le pseudonyme du client.
- Description\_c : pour la description du client.
- Server\_ip : l'adresse IP du serveur.
- Server\_port : le port du serveur

On note également la présence d'un 'code QR' : pour scanner le code QR du serveur



De plus quand une personne se connecter, cette partie des informations du serveur se mettre à jour pour lister la liste de tous les clients connecter au serveur.

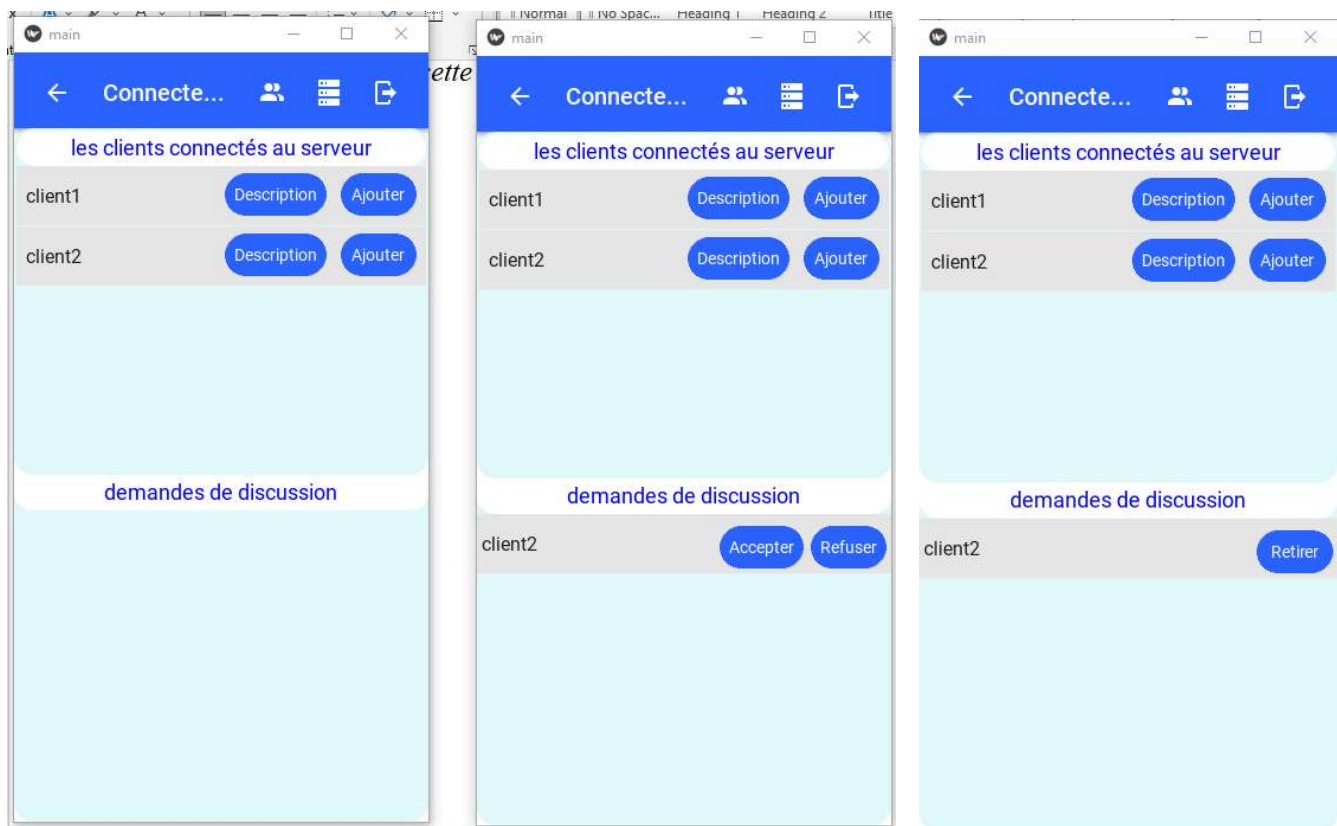
## Post-connexion

Après la connexion au serveur, on accède à l'écran où s'affichent les informations des clients connectés et du serveur.

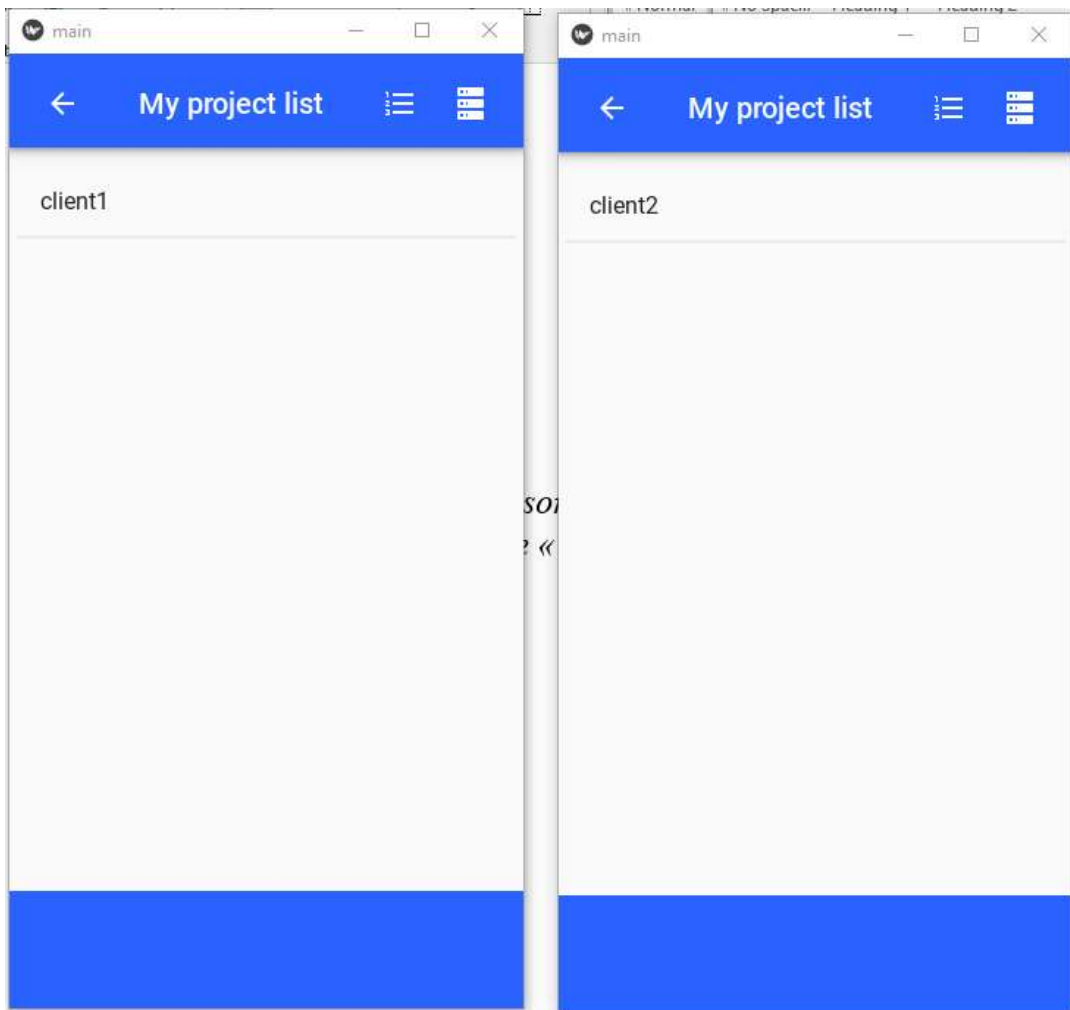
Au départ, on retrouve la liste de tous les clients connectés au serveur. On peut voir la description de chaque client, ou l'ajouter (l'inviter) via respectivement les boutons 'description' et 'ajouter'.

Lorsque l'on appuie sur le bouton 'ajouter' d'un client, ce dernier est notifié et une nouvelle 'card' s'ajoute dans la section 'demande de discussion'. Cette personne peut alors accepter ou refuser la demande via les boutons 'accepter' et 'refuser'.

Si la personne accepte, les boutons 'accepter' et 'refuser' sont retirés et remplacés par le bouton 'retirer', permettant ainsi de retirer la personne acceptée.



Une fois que l'on a accepté des personnes avec qui discuter ou qu'on a été accepté, on passe à l'écran 'accepted' où se trouvent tous les clients avec lesquels on a accepté de discuter.



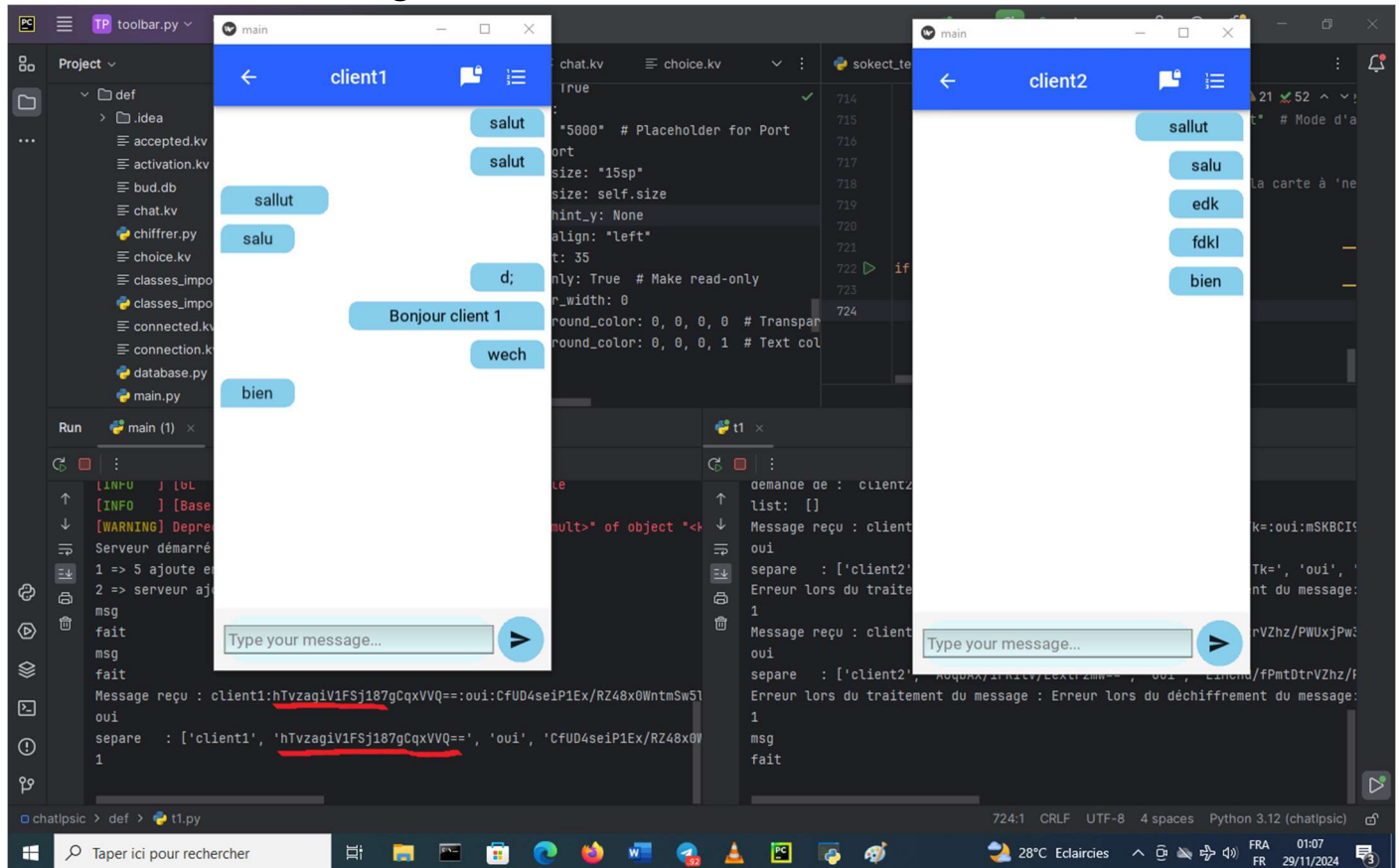
On click sur le pseudo de la personne avec qui l'on veut entrer en communication pour aller sur le « screen » de chating.

## Chating

C'est ici qu'on envoie et reçoit les messages !

Par défaut, les messages sont chiffrés, mais on peut décider de ne pas les chiffrer.

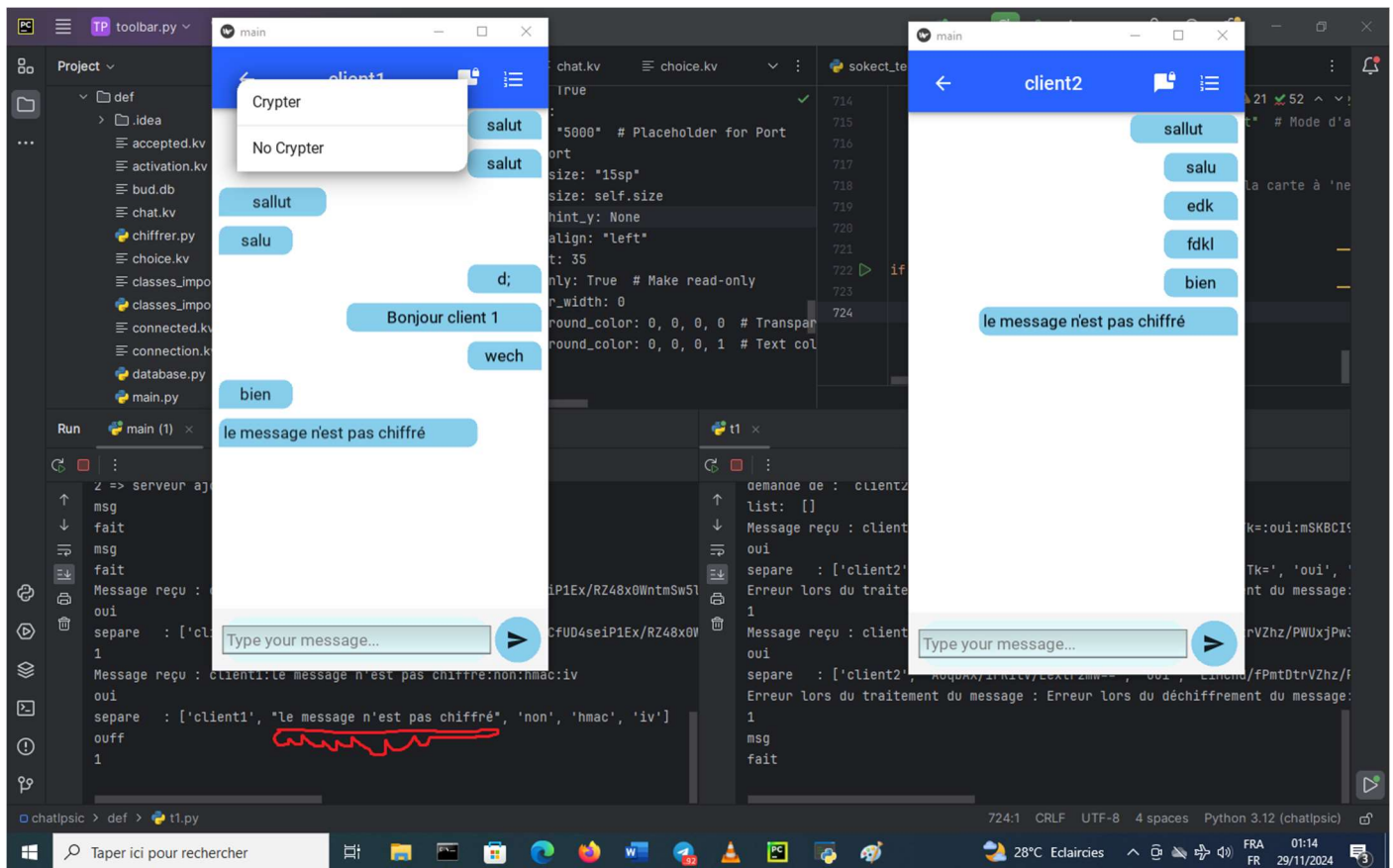
Dans ce cas, les messages sont chiffrés.



Mais on peut décider de ne pas chiffrer les messages en choisissant l'option : 'No Crypter'



Avec ça les messages vont passer en clair dans le réseau :



## *Fonctionnement de l'application : en Back-end*

### *Structure de fichier*

Le projet contient des fichiers Python, des fichiers Kivy (kv) pour l'interface graphique. Lors du fonctionnement de l'application, une image (.png) et une base de données SQLite seront générées. Au total, nous aurons les fichiers suivants :

- *base\_de\_donnée.db*,
- *server\_qr\_code.png*,
- *accepted.kv*,
- *activation.kv*, *chat.kv*,
- *classes\_import.kv*,
- *classes\_import.py*,
- *connected.kv*,
- *connexion.kv*,
- *Database.py*,
- *main.py*,
- *screen\_classe.py*,
- *socket\_ter.py*.



## *activation.kv*

L'écran "Server Activation" est conçu pour permettre l'activation d'un serveur. Il contient un fond de couleur douce et une disposition en grille (MDGridLayout) qui inclut un titre de carte avec un bouton pour revenir à l'écran précédent et une étiquette "Server". Un autre élément de carte permet à l'utilisateur de saisir des informations comme le pseudonyme, la description, le nombre maximum de clients et le port pour activer le serveur. L'activation est effectuée via un bouton qui, une fois pressé, lance une fonction pour démarrer le serveur.

L'écran "Server Details" est dédié à l'affichage des détails du serveur une fois celui-ci activé. Il présente une carte avec un bouton de retour et un titre, ainsi qu'une autre carte qui affiche des informations comme le pseudonyme, la description, l'adresse IP et le port du serveur sous forme de champs en lecture seule. Les informations sont organisées en colonnes pour une présentation claire et structurée.

## *accepted.kv*

L'écran "accepted" affiche la liste des clients avec lesquels on a accepté de discuter. Il utilise un fond clair et une disposition verticale, avec une barre d'en-tête pour naviguer entre les vues. La liste des clients défile verticalement, et une barre de navigation inférieure permet d'accéder à d'autres sections. Des boutons de navigation facilitent la transition entre les écrans.

## *chat.kv*

L'écran "chat" est une interface de chat avec cryptographie et envoi de messages. Il comprend une barre d'en-tête (MDTopAppBar) permettant de naviguer entre les écrans, avec des boutons pour activer la cryptographie et accéder à l'écran des utilisateurs connectés. La zone de chat est présentée avec une liste défilante où les messages sont ajoutés dynamiquement, permettant un défilement vertical fluide. En dessous, un champ de saisie au design arrondi permet à l'utilisateur de rédiger son message. Un bouton d'envoi permet d'envoyer le message, et des interactions sont associées pour activer la cryptographie et naviguer vers d'autres écrans.

## *classes\_import.kv*

Le code décrit plusieurs éléments d'interface utilisateur pour une application, incluant des champs de texte et des boutons interactifs. Le Command et Response sont des éléments avec des arrière-plans arrondis et colorés, offrant un design attrayant pour les messages envoyés et reçus. Le <MyHoverCard> est un composant avec une élévation, utilisé pour afficher un fond coloré par défaut. Un champ de texte personnalisé avec des boutons pour insérer des emojis, des fichiers et enregistrer des messages vocaux est également défini, intégrant des icônes interactives. Enfin, la QRCodeScanScreen présente une interface de lecture de QR codes avec un aperçu vidéo, une carte pour afficher le résultat du scan (IP:PORT), et des éléments de navigation pour revenir à un écran précédent.

## *screen\_classe.py*

Le fichier `screen_classes.py` contient les classes de tous les écrans (screen) de l'application. Il regroupe les différents paramètres et attributs liés à ces classes. Ce fichier est lié au code Kivy. Chaque écran a une classe spécifique dans le fichier `screen_classes.py`, et ces classes sont modifiées dans les fichiers Kivy correspondants. Chaque classe d'écran a son propre fichier Kivy. Les modifications se font à travers les identifiants (id).

## *connected.kv*

Le code décrit l'interface d'un écran nommé **ConnectedClientsScreen**, qui permet de visualiser les utilisateurs connectés à un serveur. L'écran est structuré en deux sections principales : une pour afficher les clients connectés et une autre pour les demandes de discussion. L'écran possède une **MDTopAppBar** avec des boutons de navigation, incluant un retour à l'écran de connexion, des options pour accéder à d'autres écrans et un bouton pour se déconnecter. La première section présente une liste déroulante des clients connectés, tandis que la seconde affiche les demandes de discussion dans un autre conteneur à défilement. Les différentes sections utilisent des **MDCard** et des **ScrollView** pour un affichage soigné et une navigation fluide.

## *connection.kv*

L'écran `ClientConnectionScreen` permet à l'utilisateur de se connecter au serveur via une connexion manuelle ou un QR code. Il comprend une barre de navigation, des champs de saisie pour le pseudo et la description, ainsi qu'un **MDTabs** pour choisir entre la connexion manuelle ou QR code. Des **MDCard** et **MDLabel** structurent les informations, avec un bouton de connexion dédié.

## *main.py*

Le fichier **main.py** est le centre de l'application. Il utilise tous les autres fichiers pour faire fonctionner l'application. Il récupère les informations du backend et les associe aux éléments du frontend. Il récupère également les informations des utilisateurs et les envoie au backend, puis récupère les données du backend pour les afficher graphiquement.

## *Database.py*

La classe `DatabaseManager` permet de gérer les interactions avec une base de données SQLite pour un système de messagerie. Elle établit une connexion à la base de données, crée une table pour stocker les messages, et permet d'insérer et de récupérer des messages. La méthode `connect` gère la connexion, `create_table` crée la table "messages" si elle n'existe pas, `store_message` insère un message dans la base, et `get_messages` permet de récupérer des messages selon une requête spécifiée. La gestion des erreurs est intégrée pour assurer une utilisation fiable.

## *Chiffre.py*

La classe MessageSecurity gère la sécurité des messages en utilisant des techniques de chiffrement asymétrique RSA et symétrique AES. Elle permet de générer des clés RSA, ainsi que des clés AES pour le chiffrement de messages. La classe permet de chiffrer et déchiffrer des messages à l'aide de la clé AES, tout en utilisant un HMAC pour garantir l'intégrité et l'authenticité des messages. Elle gère également le chiffrement et le déchiffrement de la clé AES elle-même via RSA. Des méthodes de génération de clés, de chiffrement/déchiffrement de messages, et de vérification de l'intégrité sont incluses, tout en assurant la sécurité des données via des mécanismes comme le padding et le calcul de HMAC.

## *classes\_import.py*

Ce code définit plusieurs classes personnalisées, telles que MyHoverCard, qui change de couleur au survol de la souris, et Tab, qui permet de gérer des onglets. D'autres classes incluent Command et Response, qui sont des étiquettes avec des propriétés personnalisables, et ClickableTextFieldRound, un champ de texte avec un indice cliquable qui ouvre des boîtes de dialogue d'information en fonction du texte affiché. Ces éléments sont intégrés dans une mise en page flexible, permettant une interface interactive et réactive pour l'utilisateur.

## *socket\_ter.py*

Le code présente une implémentation d'un serveur et d'un client utilisant les sockets en Python pour gérer une communication entre plusieurs clients. Le serveur écoute sur un port spécifique, accepte les connexions des clients et les gère dans des threads séparés. Lorsqu'un client se connecte, son pseudonyme est enregistré et associé à son socket, ce qui permet au serveur de diriger les messages entre les clients. Les messages sont envoyés dans un format structuré avec des balises <START> et <END>. Le serveur prend en charge différentes commandes telles que l'ajout de contacts, l'envoi de messages et la gestion des déconnexions. Les clients, une fois connectés, peuvent envoyer et recevoir des messages en utilisant des pseudonymes pour l'identification. Le serveur utilise aussi un mécanisme de diffusion pour envoyer des messages à tous les clients ou à un client spécifique. Le serveur et les clients peuvent être arrêtés proprement, avec la fermeture des connexions et des ressources.

## *Fonctionnement du code*

### **Connexion :**

Au lancement de l'application, une paire de clés RSA (publique et privée) de 3072 bits est générée.

Lorsqu'un serveur est activé, il reste en écoute pour accepter les connexions des clients jusqu'à atteindre la limite fixée. À chaque connexion, le serveur récupère les informations du client (pseudonyme, description, clé publique et socket) et les stocke dans des dictionnaires et listes appropriés. Il effectue ensuite un 'broadcast' pour transmettre les informations du nouveau client à tous les clients connectés.

Une clé AES de 256 bits est générée pour chaque client connecté. Lorsqu'on invite un client via le bouton "Ajouter", une invitation lui est envoyée. Si le client accepte, la clé AES est cryptée avec sa clé publique RSA et lui est transmise. Les deux parties associent la clé AES à leurs pseudonymes dans un dictionnaire respectif.

### **Envoi de messages :**

Lorsqu'on veut envoyer un message, la clé AES correspondante est récupérée via le pseudonyme du destinataire pour chiffrer le message. Un IV (Initialization Vector) est généré, et un HMAC (Hash-based Message Authentication Code) est calculé pour garantir l'intégrité et l'authentification des données. Le message chiffré et le HMAC sont encodés en base64, concaténés avec d'autres informations, puis prêts à être envoyés au client.

À la réception, le client décode les données en base64, vérifie le HMAC pour s'assurer de l'intégrité du message, puis le déchiffre avec sa clé AES récupérée via le pseudonyme. Enfin, le message est affiché à l'écran et stocké dans la base de données pour un usage ultérieur.