28/12/2023

Documentation Programmeur

FAST CHAT

Thibaut COSENZA RT222
FAST CHAT

Sommaire :

1. **Présentation du code 1**
   a) **Fichiers présents sur le repository GitHub 1**
2. **Code du Serveur (Serveur.py) : 1**
3. **Code du Client (Client.py) : 8**

1. Présentation du code

a) Fichiers présents sur le repository GitHub

Pour cette application de chat, le nombre de fichiers à télécharger est au nombre de trois ou quatre :

- Le fichier **Serveur.py** (contenant le code du serveur).
- Le fichier **Client.py** (contenant le code du client).
- **La Base de données** (MySQL) (contenant la base de données de l'application).
- **Le fichier de packages** à installer pour le Client et le Serveur.

2. Code du Serveur (Serveur.py) :

Le code du **serveur** est d'abord **composé** de plusieurs **classes** qui sont :

- **AuthDialog :** Fenêtre de dialogue pour l'authentification des administrateurs.
- **UserSignal :** Classe pour la mise à jour des utilisateurs connectés.
- **UserManager :** Gestionnaire des utilisateurs : messages et actions administratives.
- **ServerThread :** Thread principal du serveur qui accepte les connexions des clients et les gère.
- **ServerGUI :** Interface graphique pour l'administration du serveur.
- **ClientHandler :** Gestionnaire de messages pour chaque client connecté.

Le serveur a donc pour rôle d'administrer les différents clients, et de recevoir les différents messages de ceux-ci. Il s'occupe notamment de renvoyer les messages reçus par les différents clients, mais aussi d'administrer les clients avec des fonctions : « Kill », « Kick » et « Ban ».

Voici donc le **code du serveur**, avec **docstring** incluse, permettant de comprendre comment le code est **architecturé :**

```
Classes :
- AuthDialog : Fenêtre de dialogue pour l'authentification des administrateurs.
- UserSignal : Classe pour la mise à jour des utilisateurs connectés.
- UserManager : Gestionnaire des utilisateurs : messages et actions administratives.
- ServerThread : Thread principal du serveur qui accepte les connexions des clients et les gère.
- ServerGUI : Interface graphique pour l'administration du serveur.
- ClientHandler : Gestionnaire de messages pour chaque client connecté.

Le code utilise PyQt6 pour l'interface graphique et le multithreading pour gérer les connexions des clients
de manière asynchrone. Il utilise également une base de données MySQL pour stocker les messages
et les utilisateurs bannis.

Note : Assurez-vous d'installer PyQt6 et le module MySQL Connector avant d'exécuter le code.

Utilisation :
    1. Lancer l'application.
    2. Connectez-vous avec l'identifiant et le mot de passe donné.
    3. Administrez votre application de messagerie

Author:
    COSENZA Thibaut

Date:
    31/12/2023
"""
```

Import des différents packages :


```python
import sys
import socket
import threading
import time
import queue
import mysql.connector
from functools import partial
from PyQt6.QtGui import QPalette, QColor, QLinearGradient
from PyQt6.QtCore import QTimer, QObject, pyqtSignal, QThread
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QListWidget, QDialog, QLabel, QLineEdit, QDialog, QMessageBox
```

Première classe du code du serveur (AuthDialog) permettant **l'authentification** de l'administrateur à la palette de commande :

```python
class AuthDialog(QDialog):
    """
        Fenêtre de dialogue pour l'authentification des administrateurs.
    """
    def __init__(self):
        """
        Initialise la fenêtre de dialogue et ses composants.
        """
        super().__init__()

        self.setWindowTitle("Authentification du serveur")
        self.setGeometry(300, 300, 300, 150)

        layout = QVBoxLayout()
        # QLabel et QlineEdit pour le champs de saisie du nom d'utilisateur.
        self.username_label = QLabel("Nom d'utilisateur:")
        self.username_input = QLineEdit(self)

        # QLabel et QlineEdit pour le champs de saisie du mot de passe administrateur.
        self.password_label = QLabel("Mot de passe:")
        self.password_input = QLineEdit(self)
        self.password_input.setEchoMode(QLineEdit.EchoMode.Password)

        # QPushButton pour déclencher le processus d'authentification.
        self.login_button = QPushButton("Connexion", self)
        self.login_button.clicked.connect(self.authenticate)
```

Méthode permettant l'authentification du serveur :

```python
# Méthode appelée lorsqu'un utilisateur tente de s'authentifier.
def authenticate(self):
    # Nom d'utilisateur et mot de passe à utiliser pour l'administrateur.
    correct_username = "admin"
    correct_password = "serv2024!"

    # Récupère le nom d'utilisateur et le mot de passe saisis.
    entered_username = self.username_input.text()
    entered_password = self.password_input.text()

    # Vérifie si les informations d'authentification sont correctes.
    if entered_username == correct_username and entered_password == correct_password:
        self.accept()
    else:
        QMessageBox.warning(self, "Authentification échouée", "Nom d'utilisateur ou mot de passe incorrect.")
```

Deuxième classe du code du serveur (UserManager) permettant la gestion des utilisateurs, que cela soit au niveau des messages, au niveau des outils d'administrations, ou encore à la connexion à la base de données :


```python
class UserManager:
    """
        Gestionnaire des utilisateurs : messages et actions administratives.
    """
    def __init__(self, user_signal):
        """
        Attributs :
            - connected_users: Dictionnaire des utilisateurs connectés.
            - message_queues: Dictionnaire des files d'attente de messages pour chaque utilisateur.
            - lock: Verrou pour assurer la synchronisation des opérations sur les utilisateurs.
            - user_signal: Instance de la classe UserSignal pour émettre des signaux de mise à jour des utilisateurs.
            - kicked_users: Dictionnaire des utilisateurs actuellement kickés et leur durée.
            - banned_users: Dictionnaire des utilisateurs bannis.
            - shutdown_requested: Booléen indiquant si l'arrêt du serveur a été demandé.
            - db_connection: Connexion à la base de données MySQL.
        """
        self.connected_users = {}
        self.message_queues = {}
        self.lock = threading.Lock()
        self.user_signal = user_signal
        self.kicked_users = {}
        self.banned_users = {}
        self.shutdown_requested = False

        # Configuration de la connexion à la base de données MySQL.
        self.db_connection = mysql.connector.connect(
            host="127.0.0.1",
            user="serv302",
            password="serv2024",
            database="sae302"
```

Différentes fonctions, permettant, l'envoi de messages en broadcast, l'envoi de messages d'arrêt du serveur, ou encore la création de la table « messages » dans la base de données :

```python
# Fonction diffusant le message reçu au serveur à tous les utilisateurs connectés, sauf à l'expéditeur.
def broadcast_message(self, message, sender_username):
    with self.lock:
        for user_data in self.connected_users.values():
            user_socket = user_data['socket']
            target_username = self.get_username_from_socket(user_socket)
            if target_username != sender_username and target_username not in self.kicked_users and target_username not in self.banned_users:
                user_socket.send(message.encode('utf-8'))

# Fonction diffusant un message d'arrêt du serveur à tous les utilisateurs lorsque la commande "kill" est appelé"
def broadcast_server_shutdown(self):
    shutdown_message = "Attention le serveur va s'arrêter !"
    self.broadcast_message(shutdown_message, sender_username="Server")
    with self.lock:
        for user_data in self.connected_users.values():
            user_socket = user_data['socket']
            user_socket.send(shutdown_message.encode('utf-8'))

# Fonction créant la table des messages dans la base de données.
def create_messages_table(self):
    with self.db_connection.cursor() as cursor:
        cursor.execute("""
                    CREATE TABLE IF NOT EXISTS messages (
                        id INT AUTO_INCREMENT PRIMARY KEY,
                        username VARCHAR(255),
                        address VARCHAR(255),
                        message TEXT,
                        timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

Fonctions en lien avec la connexion à la base de données, ou encore à l'ajout ou a la suppression d'utilisateurs connectés :


```python
# Fonction ajoutant le message reçu par le serveur à la base de données.
def add_message_to_db(self, username, address, message):
    with self.db_connection.cursor() as cursor:
        cursor.execute("""
                INSERT INTO messages (username, address, message) VALUES (%s, %s, %s)
            """, (username, address, message))
    self.db_connection.commit()

# Fonction créant la table des utilisateurs bannis dans la base de données.
def create_banned_users_table(self):
    with self.db_connection.cursor() as cursor:
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS banned_users (
                id INT AUTO_INCREMENT PRIMARY KEY,
                username VARCHAR(255) UNIQUE,
                timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
        """)
# Fonction ajoutant un utilisateur à la liste des utilisateurs connectés.
def add_user(self, username, address, client_socket):
    self.connected_users[username] = {'address': address, 'socket': client_socket}
    self.message_queues[username] = queue.Queue()
    self.user_signal.user_updated.emit()

# Fonction supprimant un utilisateur de la liste des utilisateurs connectés.
def remove_user(self, username):
    if username in self.connected_users:
        del self.connected_users[username]
        del self.message_queues[username]
        self.user_signal.user_updated.emit()
```

Fonctions permettant l'administration avec les commandes : **« Kick », « Ban » et « Kill » :**

```python
# Fonction "Kickant" un utilisateur pour une durée spécifiée.
def kick_user(self, username, duration):
    kick_message = f"Server : {username} a été kick pendant {duration}"
    self.broadcast_message(kick_message, sender_username="Server")
    self.kicked_users[username] = time.time() + self.parse_duration(duration)

    # Lance un minuteur dans un thread séparé pour ne pas bloquer le thread principal.
    threading.Thread(target=self.kick_timer, args=(username,)).start()
    # Marquer l'utilisateur comme "kické" pour la durée spécifiée.
    self.user_signal.user_updated.emit()

# Fonction "Timer" pour lever le kick après la durée spécifiée.
def kick_timer(self, username):
    time.sleep(self.kicked_users[username] - time.time())
    if username in self.kicked_users:
        del self.kicked_users[username]
        self.user_signal.user_updated.emit()

# Fonction analysant une chaîne de durée au format "1h", "5m", "10s" en secondes.
def parse_duration(self, duration):
    unit = duration[-1].lower()
    value = int(duration[:-1])
    if unit == 'h':
        return value * 3600
    elif unit == 'm':
        return value * 60
    elif unit == 's':
        return value
```

Troisième classe du code du serveur (ServerThread), permettant l'acceptation des connexions clients et du management des threads :

```python
class ServerThread(QThread):
    """
            Thread principal du serveur qui accepte les connexions des clients et les gère.
            Attribut :
            - user_manager: Instance de la classe UserManager pour la gestion des utilisateurs.
    """
    def __init__(self, user_manager):
        """
            __init__: Initialise le thread du serveur.
        """
        super().__init__()
        self.user_manager = user_manager

    # Fonction gérant les messages reçus par le client.
    def handle_client_messages(self, client_socket, username):
        try:
            address = client_socket.getpeername()
            print(f"{username} connecté à l'adresse {address}")
            while True:
                data = client_socket.recv(1024)
                if not data:
                    break
                message = data.decode('utf-8')
                print(f"Reçu de {username}: {message}")

                if "@ServerShutdown@" in message:
                    print("Arrêt du serveur initié. Fermeture de la connexion client.")
                    break
```

Fonctions permettant de gérer la connexion du serveur et du thread, mais aussi les messages reçus par les clients :

```python
    # Méthode principale du thread qui accepte les connexions et gère les clients.
    def run(self):
        host = "0.0.0.0"
        port = 9000
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_socket.bind((host, port))
        server_socket.listen(5)

        print(f"Serveur à l'écoute sur {host}:{port}")

        try:
            while not self.user_manager.shutdown_requested:
                client_socket, address = server_socket.accept()
                username = client_socket.recv(1024).decode()
                print(f"{username} connecté à l'adresse {address}")
                self.user_manager.add_user(username, address, client_socket)

                # Envoie l'accusé de réception après l'ajout de l'utilisateur.
                client_socket.send("ACK_USERNAME".encode())

                if not self.user_manager.shutdown_requested:
                    # Créer une instance de ClientHandler.
                    client_handler = ClientHandler(username, client_socket, self.user_manager)

                    # Envoie ClientHandler vers un autre thread.
                    thread = QThread(self)
                    client_handler.moveToThread(thread)
```

```python
    # Fonction gérant un message reçu par le client.
    def handle_message_received(self, message):
        print(f"Message reçu : {message}")
        username, _, user_message = message.partition(':')
        self.user_manager.add_message_to_db(username.strip(), "", user_message.strip())

        # Envoi le message en broadcast à tout les clients.
        self.user_manager.broadcast_message(message, sender_username=username)
```

Quatrième classe du code du serveur (ServerGUI), créant l'interface graphique du serveur pour l'administration :

```python
class ServerGUI(QWidget):
    """
        Interface graphique pour l'administration du serveur.

        Attributs:
        - selected_user: Nom de l'utilisateur sélectionné dans l'interface.
        - user_manager: Instance de la classe UserManager pour la gestion des utilisateurs.
        - authenticated: Booléen indiquant si l'utilisateur est authentifié.
    """
    selected_user = None
    def __init__(self, user_manager):
        """
            __init__: Initialise l'interface graphique.
        """
        super().__init__()
        self.user_manager = user_manager
        self.authenticated = False

        # Définition du titre de la fenêtre et de sa taille.
        self.setWindowTitle("Administration")
        self.setGeometry(300, 300, 400, 200)

        # Déclaration des boutons (kick, ban, kill) comme des attributs de classe.
        self.kick_button = QPushButton("Kicker l'utilisateur sélectionné", self)
        self.ban_button = QPushButton("Bannir l'utilisateur sélectionné", self)
        self.kill_button = QPushButton("Kill le serveur", self)
        self.init_ui()
```

Plusieurs fonctions et méthodes permettant la connexion de l'administrateur à l'interface graphique, la mise à jour des messages, la sélection des utilisateurs dans l'interface, et les différentes commandes d'administration :

```python
# Méthode pour l'authentification de l'administrateur.
def authenticate(self):
    # Nom d'utilisateur et mot de passe à utiliser pour l'administrateur.
    correct_username = "admin"
    correct_password = "serv2024!"

    # Vérifie si l'utilisateur est authentifié avant de traiter les messages.
    if not self.authenticated:
        auth_dialog = AuthDialog()
        if auth_dialog.exec() == QDialog.DialogCode.Accepted:
            self.authenticated = True
        else:
            sys.exit(0)

# Met à jour l'affichage des messages dans l'interface.
def update_messages(self, username, message):
    # Vérifie si l'utilisateur est authentifié avant de traiter les messages.
    if self.authenticated:
        print(message)

# Fonction enregistrant l'utilisateur sélectionné dans la variable de classe.
def select_user(self, item):
    ServerGUI.selected_user = item.text()

# Fonction rafraichissant la liste des utilisateurs connectés dans l'interface.
def refresh_user_list(self):
    current_item = self.user_list.currentItem()
    self.user_list.clear()
    users = list(self.user_manager.connected_users.keys())
    self.user_list.addItems(users)
```

```python
# Fonction "Kickant" l'utilisateur sélectionné.
def kick_user(self):
    # Vérifie si l'utilisateur est authentifié avant d'exécuter la commande.
    if self.authenticated:
        selected_item = self.user_list.currentItem()
        if selected_item is not None:
            ServerGUI.selected_user = selected_item.text()
            duration = "1h"
            self.user_manager.kick_user(ServerGUI.selected_user, duration)
            self.refresh_user_list()
        else:
            print("Aucun utilisateur sélectionné.")

# Fonction "Bannissant" l'utilisateur sélectionné.
def ban_user(self):
    # Vérifie si l'utilisateur est authentifié avant d'exécuter la commande.
    if self.authenticated:
        selected_user = self.user_list.currentItem().text()
        self.user_manager.ban_user(selected_user)

# Fonction arrêtant le serveur et l'application.
def kill_server(self):
    # Vérifie si l'utilisateur est authentifié avant d'exécuter la commande
    if self.authenticated:
        self.user_manager.kill_server()
        sys.exit()
```

Cinquième et dernière classe du code du serveur (ClientHandler), permettant de gérer les messages pour chaque client connecté :

```python
class ClientHandler(QObject):
    """
        Gestionnaire de messages pour chaque client connecté.

        Attributs:
        - username: Nom d'utilisateur du client.
        - client_socket: Socket de communication avec le client.
        - user_manager: Instance de la classe UserManager pour la gestion des utilisateurs.

        Signal:
        - message_received: Signal émis lorsqu'un message est reçu du client.
    """
    message_received = pyqtSignal(str)
    def __init__(self, username, client_socket, user_manager):
        """
            __init__: Initialise le gestionnaire de messages.
        """
        super().__init__()
        self.username = username
        self.client_socket = client_socket
        self.user_manager = user_manager
```

Fonctions permettant l'envoie de message au serveur, les messages reçus du client, mais aussi la méthode principale exécutée dans un thread pour gérer les clients :

```python
# Fonction envoyant un message au serveur.
def send_message_to_server(self, message):
    try:
        self.client_socket.send(message.encode('utf-8'))
    except Exception as e:
        print(f"Erreur lors de l'envoi d'un message au serveur pour {self.username}: {e}")

# Fonction gèrant les messages reçus du client.
def handle_client_messages(self):
    try:
        address = self.client_socket.getpeername()
        print(f"{self.username} connecté à l'adresse {address}")

        while True:
            data = self.client_socket.recv(1024)
            if not data:
                break
            message = data.decode('utf-8')
            print(f"Reçu de {self.username}: {message}")

            if "@ServerShutdown@" in message:
                print("Arrêt du serveur initié. Fermeture de la connexion client.")
                break

            # Formatage du message.
            formatted_message = f"@{self.username}: {message}"
            print(f"Message formaté: {formatted_message}")

            self.user_manager.add_message_to_db(self.username, "", message)
```

```python
# Méthode principale exécutée dans un thread pour gérer les clients.
def run(self):
    try:
        address = self.client_socket.getpeername()
        print(f"{self.username} connecté depuis {address}")

        while True:
            # Vérifie si l'utilisateur est "kické" ou "banni".
            if self.user_manager.is_kicked(self.username):
                kick_message = "Vous avez été Kické, veuillez réessayer plus tard."
                self.client_socket.send(kick_message.encode())
                break

            if self.user_manager.is_banned(self.username):
                ban_message = "Vous avez été banni du serveur."
                self.client_socket.send(ban_message.encode())
                break

            self.client_socket.send("ACK_USERNAME".encode())
            self.handle_client_messages()

    except Exception as e:
        print(f"Erreur de gestion du client {self.username}: {e}")
    finally:
        self.user_manager.remove_user(self.username)
        print(f"{self.username} déconnecté.")
        self.client_socket.close()
        self.user_manager.user_signal.user_updated.emit()
```

Enfin, la **dernière fonction du serveur** permettant de lancer l'application entière avec les paramètres fournis :

```python
def main():
    """
        Fonction principale pour lancer l'application du serveur de chat asynchrone.

        Crée une instance de l'application PyQt, du gestionnaire d'utilisateurs et du thread du serveur.
        Lance l'interface graphique et démarre le thread du serveur.
    """
    app = QApplication(sys.argv)

    # Authentification réussie, créer le gestionnaire d'utilisateurs et démarre le serveur.
    user_signal = UserSignal()
    user_manager = UserManager(user_signal)

    # Démarre le thread du serveur.
    server_thread = ServerThread(user_manager)
    server_thread.start()

    server_gui = ServerGUI(user_manager=user_manager)
    server_gui.authenticate()

    server_gui.show()

    sys.exit(app.exec())

if __name__ == "__main__":
    main()
```

3. Code du Client (Client.py) :

Le code du **client** est d'abord **composé** de plusieurs **classes** qui sont :

- **ClientThread :** classe basée sur QThread chargée de gérer la communication entre le client et le serveur.
- **RoomWindow :** La fenêtre principale représentant l'espace de discussion où les utilisateurs peuvent interagir.
- **RegistrationWindow :** gère l'inscription des utilisateurs.
- **LoginWindow :** gère les fonctions de connexion des utilisateurs.
- **ChatClient :** La fenêtre principale de l'application qui contrôle l'ensemble du processus.

Le client a donc pour rôle de pouvoir créer un utilisateur, avec un nom d'utilisateur et un mot de passe, puis de se connecter avec celui-ci pour pouvoir envoyer des messages via une interface graphique dédiée avec différents canaux de discussions.

Voici donc le **code du client**, avec **docstring** incluse, permettant de comprendre comment le code est **architecturé :**

```python
"""
Application de messagerie (côté client)

Cette application met en œuvre un client de chat utilisant PyQté pour l'interface utilisateur graphique (GUI) et des sockets pour la communication
Le client permet aux utilisateurs de s'inscrire, de se connecter, de rejoindre différents salons de discussion et d'envoyer des messages à d'autres

Classes:
    - ClientThread: classe basée sur QThread chargée de gérer la communication entre le client et le serveur.
    - RoomWindow: La fenêtre principale représentant l'espace de discussion où les utilisateurs peuvent interagir.
    - RegistrationWindow: gère l'inscription des utilisateurs.
    - LoginWindow: gère les fonctions de connexion des utilisateurs.
    - ChatClient: La fenêtre principale de l'application qui contrôle l'ensemble du processus.

Utilisation :
    1. Lancer l'application.
    2. Définissez l'adresse IP du serveur sur la page d'accueil.
    3. Connectez-vous ou créez un nouveau compte utilisateur.
    4. Rejoignez un salon de discussion et commencez à interagir avec d'autres utilisateurs.

Author:
    COSENZA Thibaut

Date:
    31/12/2023
"""
```

Import des différents packages :

```
import sys
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout, QLabel, QLineEdit, QPushButton, QDialog, QStackedWidget, QListWidget,
from PyQt6.QtCore import Qt, pyqtSignal, QThread
from PyQt6.QtGui import QTextCursor, QPalette, QColor, QLinearGradient
import hashlib
import socket
import mysql.connector
import threading
```

Première classe du code du client (ClientThread), permettant la communication côté client avec le serveur :

```
    # Fonction factice qui peut être remplacée par la fonction réelle gérant les messages de l'interface utilisateur.
def get_gui_message():
    pass

class ClientThread(QThread):
    message_received = pyqtSignal(str)
    """
        Thread responsable de la gestion de la communication côté client avec le serveur.
    """
    def __init__(self, host, port, username, password, room):

        """
            Initialise ClientThread avec les paramètres fournis.
            Paramètres :
            - host (str) : Adresse de l'hôte du serveur.
            - port (int) : Numéro de port du serveur.
            - username (str) : Nom d'utilisateur choisi par l'utilisateur.
            - password (str) : Mot de passe de l'utilisateur.
            - room (str) : Channel initial de l'utilisateur.
        """
```

Fonction principale du thread responsable de la connexion au serveur :

```
    def run(self):

        """
            Méthode principale du thread, responsable de la connexion au serveur,
            de l'authentification de l'utilisateur et de la réception continue des messages du serveur.
        """
        try:
            self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.client_socket.connect((self.host, self.port))
            self.client_socket.send(self.username.encode())

            # Reçois l'accusé de réception du serveur.
            ack = self.client_socket.recv(1024).decode()
            if ack != "ACK_USERNAME":
                print("Erreur : Le serveur n'a pas reconnu le nom d'utilisateur.")
                return

            # Envoie le mot de passe et le channel.
            auth_data = f"{self.password};{self.room}"
            self.client_socket.send(auth_data.encode())

            # Démarrer un thread séparé pour gérer les entrées de l'utilisateur et envoyer des messages.
            input_thread = threading.Thread(target=self.send_user_input)
            input_thread.start()

            while True:
                message = self.client_socket.recv(1024).decode()
                if not message:
                    break
                self.message_received.emit(message)
```

Méthodes pour envoyer et récupérer les messages saisis par l'utilisateur dans la zone de texte :

```
    # Méthode pour envoyer les messages saisis par l'utilisateur au serveur.
    def send_user_input(self):
        try:
            while True:
                message = yield from get_gui_message()
                if not message:
                    break
                full_message = f"{self.username}: {message}"
                # Envoyer le message au serveur pour la diffusion
                self.client_socket.send(full_message.encode())
        except Exception as e:
            print(f"Erreur lors de l'envoi des données de l'utilisateur : {e}")

    # Méthode pour récupérer le message depuis la zone de texte de saisie et l'envoyer au serveur.
    def send_message(self, message):
        if self.client_socket:
            self.client_socket.send(message.encode())
            print(f"Message envoyé: {message}")
```

Deuxième classe du code du client (RoomWindow), représentant la fenêtre principale dans laquelle les utilisateurs interagissent avec l'espace de discussion :

```
class RoomWindow(QWidget):
    """
        Représente la fenêtre principale dans laquelle les utilisateurs interagissent avec l'espace de discussion.
    """
    message_received = pyqtSignal(str)
    logout_requested = pyqtSignal()

    def __init__(self, username, client_thread, parent, user_manager):
        """
            Initialise la fenêtre RoomWindow avec les paramètres fournis.

            Paramètres :
            - nom d'utilisateur (str) : Nom d'utilisateur de l'utilisateur.
            - client_thread (ClientThread) : Le thread du client qui gère la communication.
            - parent : Le widget parent.
            - user_manager : Instance du gestionnaire d'utilisateurs.
        """
        super().__init__()

        self.username = username
        self.client_thread = client_thread
        self.parent = parent
        self.user_manager = user_manager
        self.setWindowTitle(f"Interface de {username}")
        self.setMinimumSize(300, 250)
        self.room_list = QComboBox(self)
        self.init_ui()
```

Différentes **fonctions**, permettant d'effacer le contenu de la zone de texte, d'émettre des signaux de déconnexion, de gérer le changement de salon, d'afficher le message envoyé par l'utilisateur dans la zone de texte graphique, mais aussi de récupérer le message et de l'envoyer au serveur :

```
# Fonction effaçant le contenu de la zone de texte d'entrée.
def clear_message_input(self):
    self.input_edit.clear()

# Fonction émettant le signal de déconnexion.
def logout(self):
    self.logout_requested.emit()

# Fonction gérant le changement de salon.
def room_changed(self, index):
    selected_room = self.room_list.currentText()
    self.update_title(selected_room)
    # Envoie un message dans le salon sélectionné.
    self.message_received.emit(f"@{self.username}: Change de salon vers {selected_room}")
    # Met à jour le statut de l'utilisateur dans la base de données.
    self.user_manager.update_user_status(self.username, "connected" if selected_room else "disconnected")

# Fonction affichant le message dans la zone de texte principale.
def display_message(self, message):
    self.message_edit.append(message)

# Fonction récupèrant le message depuis la zone de texte de saisie et envoie au serveur.
def send_message(self):
    message_text = self.input_edit.text()
    # Émet le signal pour informer les autres parties de l'application.
    self.message_received.emit(f"@{self.username}: {message_text}")
```

Troisième classe du code du client (RegistrationWindow), permettant l'enregistrement pour les nouveaux utilisateurs :

```
class RegistrationWindow(QDialog):
    """
        Représente la fenêtre d'enregistrement pour les nouveaux utilisateurs.
    """
    registration_successful_signal = pyqtSignal(str)

    def __init__(self, parent, user_manager):
        """
            Initialise la fenêtre d'enregistrement avec les paramètres fournis.

            Paramètres :
            - parent : Le widget parent.
            - user_manager : Instance du gestionnaire d'utilisateurs.
        """
        super().__init__()

        self.parent = parent
        self.user_manager = user_manager
        self.setWindowTitle("Inscription")
        self.setGeometry(200, 200, 400, 200)

        self.init_ui()
```

Fonctions permettant le processus d'enregistrement d'un utilisateur et de gérer le bouton de retour :

```python
# Fonction gérant le processus d'enregistrement de l'utilisateur.
def on_registration(self):
    username = self.username_input.text()
    password = self.password_input.text()

    if username and password:
        # Stocke le mot de passe en clair dans la base de données.
        success = self.user_manager.register_user(username, password)
        if success:
            print("Inscription réussie!")
            self.registration_successful_signal.emit(username)
            self.accept()
        else:
            print("Le nom d'utilisateur existe déjà. Veuillez choisir un nom d'utilisateur différent.")
    else:
        print("Veuillez remplir tous les champs.")

# Fonction gèrant le retour à la page précédente.
def on_back(self):
    self.parent.show_home_page()
```

Quatrième classe du code client (LoginWindow), permettant la connexion pour les utilisateurs existants :

```python
class LoginWindow(QWidget):
    """
        Représente la fenêtre de connexion pour les utilisateurs existants.
    """
    login_successful = pyqtSignal(str)
    login_failed = pyqtSignal()

    def __init__(self, parent, user_manager, host, port, room):
        """
            Initialise la fenêtre de connexion avec les paramètres fournis.

            Paramètres :
            - parent : Le widget parent.
            - user_manager : Instance du gestionnaire d'utilisateurs.
            - host (str) : Adresse de l'hôte du serveur.
            - port (int) : Numéro de port du serveur.
            - room (str) : Channel initial pour l'utilisateur.
        """
        super().__init__()

        self.parent = parent
        self.host = host
        self.port = port
        self.room = room
        self.user_manager = user_manager
        self.setWindowTitle("Connexion")
        self.setGeometry(200, 200, 400, 200)
        self.init_ui()
```

Méthode et fonction permettant de définir l'adresse IP du serveur, mais aussi de gérer le processus de connexion de l'utilisateur :

```python
    # Méthode pour définir l'adresse IP du serveur.
    def set_server_ip(self):
        new_ip = self.ip_input.text()
        if new_ip:
            self.parent.set_server_ip(new_ip)
            print(f"Adresse IP du serveur définie sur {new_ip}")
        else:
            print("Veuillez entrer une adresse IP valide.")

    # Fonction gérant le processus de connexion de l'utilisateur.
    def on_login(self):
        username = self.username_input.text()
        password = self.password_input.text()

        # Récupère le mot de passe "hashé" stocké dans la base de données.
        stored_password = self.user_manager.get_user_password(username)

        if stored_password:
            entered_hashed_password = hashlib.sha256(password.encode()).hexdigest()

            # Compare le mot de passe entré avec celui stocké dans la base de données.
            if entered_hashed_password == stored_password:
                print("Connexion Réussie !")
                self.login_successful.emit(username)
            else:
                print("Mot de passe incorrect.")
                self.login_failed.emit()
        else:
            print("Utilisateur non enregistré.")
```

Cinquième classe du code client (ChatClient), permettant de gérer les différentes pages et les interactions avec l'utilisateur :

```python
class ChatClient(QMainWindow):
    """
        Fenêtre principale de l'application qui gère les différentes pages et les interactions avec l'utilisateur.
    """
    def __init__(self):
        """
            Initialise l'application ChatClient.
        """
        super().__init__()

        # Définition des widgets pour la fenêtre d'accueil.
        self.home_page = HomePage(self)
        self.stacked_widget = QStackedWidget(self)
        self.stacked_widget.addWidget(self.home_page)
        self.setCentralWidget(self.stacked_widget)

        # Définition des attributs host, port, et room.
        self.host = host
        self.port = 9000
        self.room = "Général"

        # Initialise le gestionnaire d'utilisateurs
        self.user_manager = UserManager()

        self.home_page.login_requested.connect(self.show_login_window)
        self.home_page.registration_requested.connect(self.show_registration_window)
```

Différentes fonctions, permettant de définir l'adresse IP du serveur, d'afficher la fenêtre d'enregistrement, de gérer les actions après une instruction réussie, d'afficher la fenêtre de connexion et de gérer les actions après une connexion réussie :

```python
# Fonction définissant l'adresse IP du serveur.
def set_server_ip(self, new_ip):
    self.host = new_ip
    self.dbhost = new_ip

# Fonction affichant la fenêtre d'enregistrement.
def show_registration_window(self):
    self.registration_window = RegistrationWindow(self, self.user_manager)
    self.registration_window.registration_successful_signal.connect(self.handle_registration_successful)
    self.stacked_widget.addWidget(self.registration_window)
    self.stacked_widget.setCurrentWidget(self.registration_window)

# Fonction gérant les actions après une inscription réussie.
def handle_registration_successful(self, username):
    print(f"Inscription réussie pour {username}")
    self.show_home_page()

# Fonction affichant la fenêtre de connexion.
def show_login_window(self):
    self.login_window = LoginWindow(self, self.user_manager, host, port, room)
    self.login_window.login_successful.connect(self.handle_login_successful)
    self.stacked_widget.addWidget(self.login_window)
    self.stacked_widget.setCurrentWidget(self.login_window)
    self.login_window.login_failed.connect(self.handle_login_failed)

# Fonction gérant les actions après une connexion réussie.
def handle_login_successful(self, username):
    print(f"Connexion réussie pour {username}")
```

Différentes fonctions, permettant de gérer les actions après une connexion échouée, d'afficher la page d'accueil, la fenêtre principale de la discussion et de gérer la déconnexion de l'utilisateur :

```python
# Fonction gérant les actions après une connexion échouée.
def handle_login_failed(self):
    print("Connexion échouée. Veuillez vérifier votre nom d'utilisateur ou votre mot de passe")

# Fonction affichant la page d'accueil.
def show_home_page(self):
    self.stacked_widget.setCurrentWidget(self.home_page)
    self.stacked_widget.addWidget(self.login_window)

# Fonction affichant la fenêtre principale de la discussion.
def show_room_window(self, username, channel="Général"):
    self.room_window = RoomWindow(username, self.client_thread, self, self.user_manager)
    self.stacked_widget.addWidget(self.room_window)
    self.stacked_widget.setCurrentWidget(self.room_window)
    self.room_window.message_received.connect(self.send_message_to_server)

    # Connexion du signal "currentIndexChanged" de "room_list" à "handle_room_change".
    self.room_window.room_list.currentIndexChanged.connect(self.handle_room_change)

    # Connexion du signal "logout_requested" de "RoomWindow" à "handle_logout_requested".
    self.room_window.logout_requested.connect(self.handle_logout_requested)

    # Met à jour le titre et envoie un message pour le salon initial.
    self.room_window.update_title(channel)
    self.client_thread.send_message(f"@{username}: Rejoint le salon {channel}")

# Fonction gérant la déconnexion de l'utilisateur.
def handle_logout_requested(self):
    self.client_thread.send_message(f"@{self.room_window.username}: Quitte le salon")
    self.client_thread.terminate()
```

Différentes méthodes et fonctions permettant de gérer le changement de salon, d'effectuer des actions après une inscription réussie, d'envoyer un message au serveur, ou encore de définir le thread du client :

```
    # Méthode gérant les changements de salon.
    def handle_room_change(self, index):
        selected_room = self.room_window.room_list.currentText()
        self.room_window.update_title(selected_room)
        self.client_thread.send_message(f"@{self.room_window.username}: Change de salon vers {selected_room}")

    # Fonction effectuant des actions après une inscription réussie.
    def registration_successful(self, username):
        self.stacked_widget.setCurrentWidget(self.home_page)
        self.show_room_window(username, "Général")

    # Méthode appelée lorsque la page actuelle change.
    def page_changed(self, index):
        pass

    # Fonction envoyant un message au serveur.
    def send_message_to_server(self, message):
        try:
            if self.client_thread:
                self.client_thread.send_message(message)
                print(f"Message envoyé au serveur : {message}")
            else:
                print("Erreur: Thread client non initialisé.")
        except Exception as e:
            print(f"Erreur lors de l'envoi du message au serveur : {e}")

    # Fonction définissant le thread client.
    def set_client_thread(self, client_thread):
        self.client_thread = client_thread
```

Sixième classe du code client (HomePage), permettant d'afficher la page d'accueil où les utilisateurs peuvent s'enregistrer ou se connecter :

```
class HomePage(QWidget):
    """
        Représente la page d'accueil de l'application où les utilisateurs peuvent se connecter ou s'enregistrer.
    """
    registration_requested = pyqtSignal()
    login_requested = pyqtSignal()
    set_ip_requested = pyqtSignal(str)

    def __init__(self, parent):
        """
            Initialise la page d'accueil avec les paramètres fournis.

            Paramètres :
            - parent : Le widget parent.
        """
        super().__init__()

        self.parent = parent
        self.setWindowTitle("Connexion / Inscription à l'application de messagerie")
        self.setGeometry(300, 200, 400, 200)
        self.init_ui()
```

Méthodes et fonctions permettant de définir l'adresse IP du serveur, d'émettre le signal de demande de connexion ou d'enregistrement :

```python
# Méthode pour définir l'adresse IP du serveur.
def set_db_host(self):
    new_ip = self.ip_input.text()
    if new_ip:
        self.set_ip_requested.emit(new_ip)
        print(f"Adresse IP du serveur définie sur {new_ip}")
    else:
        print("Veuillez entrer une adresse IP valide.")

# Fonction émettant le signal de demande de connexion.
def on_login_clicked(self):
    self.login_requested.emit()

# Fonction émettant le signal de demande d'enregistrement.
def on_register_clicked(self):
    self.registration_requested.emit()

# Méthode pour définir l'adresse IP du serveur.
def set_server_ip(self):
    new_ip = self.ip_input.text()
    if new_ip:
        self.set_ip_requested.emit(new_ip)
        print(f"Adresse IP du serveur définie sur {new_ip}")
    else:
        print("Veuillez entrer une adresse IP valide.")
```

Septième classe du code client (UserManager), permettant de gérer les opérations liées à l'utilisateur telles que l'enregistrement, la connexion ou encore l'interaction avec la base de données :

```python
class UserManager:
    """
        Gère les opérations liées à l'utilisateur telles que l'enregistrement, la connexion et l'interaction avec la base de données.
    """
    def __init__(self):
        """
            Initialise le gestionnaire d'utilisateurs (UserManager).
        """
        self.connection = None
        self.cursor = None

    # Fonction établissant la connexion à la base de données avec l'adresse IP fournie.
    def set_db_host(self, host):
        self.connection = mysql.connector.connect(
                host=host,
                user="serv302",
                password="serv2024",
                database="sae302"
            )

        if self.connection.is_connected():
            print(f"Connexion à la base de données en {host}")
            self.cursor = self.connection.cursor()
            self.create_table_if_not_exists()
        else:
            print("Erreur de connexion à la base de données")
```

Fonctions, permettant de récupérer le mot de passe de l'utilisateur depuis la base de données, d'enregistrer un nouvel utilisateur dans la base de données, ou encore de vérifier les informations d'identifications dans la base de données :

```python
# Fonction récupérant le mot de passe de l'utilisateur depuis la base de données.
def get_user_password(self, username):
    # Récupère le mot de passe de l'utilisateur depuis la base de données.
    self.cursor.execute("SELECT password FROM user WHERE username = %s", (username,))
    result = self.cursor.fetchone()

    if result:
        return result[0]
    else:
        return None

# Fonction enregistrant un nouvel utilisateur dans la base de données.
def register_user(self, username, password):
    # Vérifie si l'utilisateur existe déjà.
    self.cursor.execute("SELECT * FROM user WHERE username = %s", (username,))
    if self.cursor.fetchone():
        return False
    hashed_password = hashlib.sha256(password.encode()).hexdigest()

    # Insère l'utilisateur dans la base de données avec le mot de passe hashé.
    self.cursor.execute("INSERT INTO user (username, password, alias) VALUES (%s, %s, %s)",
                        (username, hashed_password, 'DefaultAlias'))
    self.connection.commit()
    return True

# Fonction vérifiant les informations d'identification dans la base de données.
def check_credentials(self, username, entered_password):
    # Vérifie les informations d'identification dans la base de données.
    self.cursor.execute("SELECT * FROM user WHERE username = %s AND password = %s", (username, entered_password))
    return bool(self.cursor.fetchone())
```

Fonction permettant de mettre à jour le statut de l'utilisateur dans la base de données, et lancement de l'application du client et de l'interface graphique :

```python
    # Fonction mettant à jour le statut de l'utilisateur dans la base de données.
    def update_user_status(self, username, status):
        # Vérifie si l'utilisateur existe.
        if self.get_user_password(username) is not None:
            # Met à jour le statut de l'utilisateur dans la base de données.
            self.cursor.execute("UPDATE user SET status = %s WHERE username = %s", (status, username))
            self.connection.commit()
        else:
            print(f"Utilisateur {username} non trouvé. Impossible de mettre à jour le statut.")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    app.setApplicationDisplayName("Application FASTCHAT")
    # Ajout des détails du serveur
    host = "127.0.0.1"
    port = 9000
    room = "Général"
    client = ChatClient()
    client.show()
    sys.exit(app.exec())
```