Mode d'emploi : Documentation du développement et de la création du code serveur

Table des matières

I. Introduction

Ce document vous guidera à travers les différentes fonctionnalités, la mise en place, et la gestion du serveur qui facilite les communications entre utilisateurs. L'objectif principal de ce serveur est de fournir une plateforme robuste et sécurisée pour la messagerie instantanée, en permettant aux clients de se connecter, de s'inscrire, de participer à des salons de discussion, et d'interagir de manière fluide.

1. Objectif du Serveur

Le serveur constitue le cœur de notre application de chat, agissant comme une infrastructure centralisée qui coordonne les échanges entre les clients. Son rôle essentiel est de maintenir une communication fiable, d'authentifier les utilisateurs, de gérer les salons de discussion, et de garantir une expérience utilisateur harmonieuse.

2. Importance de la Convivialité

L'efficacité de l'application dépend en grande partie de la convivialité du serveur. Une architecture bien conçue et une documentation claire sont cruciales pour assurer une utilisation aisée par les développeurs et les administrateurs. Une interface utilisateur conviviale simplifie la gestion du serveur, permettant ainsi de résoudre rapidement d'éventuels problèmes et d'optimiser les performances.

3. Organisation de la Documentation

La documentation est structurée pour vous accompagner dans toutes les étapes, de l'installation initiale jusqu'à la gestion quotidienne du serveur. Chaque section fournit des informations détaillées, des exemples pratiques et des astuces pour garantir une utilisation optimale de l'application.

II. Création de la base de données

Après avoir installé toutes les dépendances de MySQL et de Python, nous allons procéder à la création de la base de données nécessaire au fonctionnement de l'application de chat. Assurez-vous que votre serveur MySQL est en cours d'exécution et que vous avez les privilèges nécessaires pour créer une nouvelle base de données.

Dans mon cas j'ai deux utilisateurs qui fonctionnent pour effectuer des actions sur cette base de données :

Username	Host	Password
root	localhost	IutC_MySQL !68#
test	localhost	test

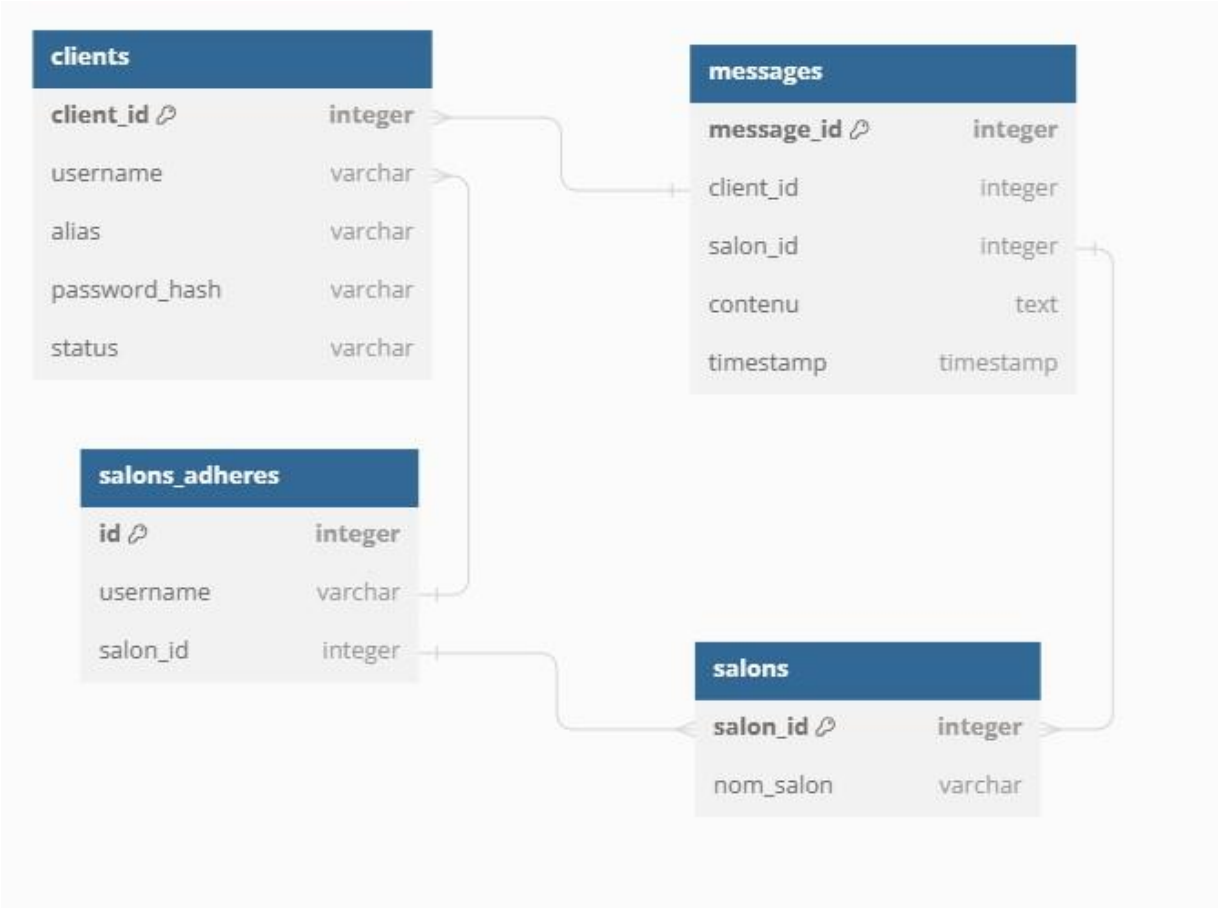
On peut soit recréer la base de données à l'aide de ces commandes :

Commandes	Explication
CREATE TABLE clients (client_id INT AUTO_INCREMENT PRIMARY KEY, username VARCHAR(255) NOT NULL, alias VARCHAR(255) NOT NULL, password_hash VARCHAR(64) NOT NULL, status VARCHAR(20) DEFAULT 'disconnected'); INSERT INTO clients (username, alias, password_hash, status) VALUES ('admin', 'Admin User', 'hashed_admin_password'), ('server', 'Server User', 'hashed_server_password');	Cette commande crée une table clients pour stocker les informations des utilisateurs. • client_id est une clé primaire auto-incrémentée. • username est le nom d'utilisateur unique. • alias est l'alias de l'utilisateur. • password_hash stocke le hash du mot de passe. • status indique le statut de connexion de l'utilisateur (par défaut à "disconnected"). On ajoute de base deux utilisateurs server et admin pour la gestion des commandes sur le serveur. Le reste des utilisateurs pourra se créer après dans l'interface du client avec la fenêtre d'inscription. On les supprimera ultérieurement pour les réinscrire avec le client afin qu'il aient un vrai password hashé.
CREATE TABLE salons (salon_id INT AUTO_INCREMENT PRIMARY KEY, nom_salon VARCHAR(255) NOT NULL); INSERT INTO salons (nom_salon) VALUES ('Général'), ('Blabla'), ('Comptabilité'), ('Informatique'), ('Marketing');	Cette commande crée une table salons pour stocker les informations sur les salons. • salon_id est une clé primaire auto-incrémentée. • nom_salon contient les noms des salons, et des salons prédéfinis sont insérés. Pour les salons on ne peut pas les créer après donc on les instancie directement. Ils sont fixs donc pas besoin de les supprimer ni d'en rajouter.
CREATE INDEX idx_username ON clients(username); CREATE TABLE salons_adheres (id INT AUTO_INCREMENT PRIMARY KEY, username VARCHAR(255) NOT NULL, salon_id INT NOT NULL, FOREIGN KEY (username) REFERENCES clients(username), FOREIGN KEY (salon_id) REFERENCES salons(salon_id));	Cette commande crée un index sur la colonne username dans la table clients, ce qui peut améliorer les performances lors de la recherche par nom d'utilisateur. Cette commande crée une table salons_adheres pour enregistrer les adhésions des utilisateurs aux salons. • id est une clé primaire auto-incrémentée. • username est une clé étrangère, ref à la table clients. • salon_id est une clé étrangère, ref à la table salons.
CREATE TABLE messages (message_id INT AUTO_INCREMENT PRIMARY KEY, client_id INT NOT NULL, salon_id INT NOT NULL, contenu TEXT, timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP, FOREIGN KEY (client_id) REFERENCES clients(client_id), FOREIGN KEY (salon_id) REFERENCES salons(salon_id));	Cette commande crée une table messages pour stocker les messages envoyés par les utilisateurs. • message_id est une clé primaire auto-incrémentée. • client_id est une clé étrangère faisant référence à la table clients. • salon_id est une clé étrangère faisant référence à la table salons. • contenu stocke le texte du message. • timestamp enregistre le moment où le message a été créé (par défaut à la date et l'heure actuelles).

Soit on importe le fichier sql :

Sous Windows : mysql -u[utilisateur] -p [nom_base_de_donnees] < fichier.sql

Sous Linux : mysql nom_base_de_donnees < fichier.sql



III. Installation

L'installation de Python est une étape cruciale pour assurer le bon fonctionnement de l'application de chat. Si Python n'est pas déjà installé sur votre système, suivez ces étapes détaillées pour garantir une installation et pouvoir commencer à développer l'application

1. Étape 1 : Vérification de l'installation de Python

Avant de commencer à coder, assurez-vous d'avoir Python installé sur votre machine. Si ce n'est pas le cas, il faut le télécharger et l'installer ici : Welcome to Python.org

IV. Initialisation et connexion à la base de données

La phase d'initialisation du serveur revêt une importance capitale, mettant en œuvre la configuration de la base de données, l'initialisation des sockets, et la création des structures de données essentielles. Cette section détaille ces étapes cruciales qui forment le fondement du serveur de l'application de chat.

```
class Server:
    def __init__(self):
        """
        Initialise la classe du serveur.

        Configurations de la base de données, initialisation des sockets, et des structures de
données.
        """
        self.db_config = {
            'host': 'localhost',
            'user': 'root',
            'password': 'IutC_MySQL!68#',
            'database': 'test'
        }

        self.db_connection = mysql.connector.connect(**self.db_config)
        self.cursor = self.db_connection.cursor()

        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_address = ('localhost', 5000)
        self.server.bind(self.server_address)

        self.clients = []
        self.pending_requests = {}
        self.banned_users = {}
        self.authenticated_users = set()
        self.is_authenticated = False
```

1. Configuration de la Base de Données

La première étape consiste à définir les paramètres de la base de données. Ces paramètres, contenus dans la variable db_config, incluent des informations telles que l'hôte, l'utilisateur, le mot de passe, et le nom de la base de données. Ces informations sont nécessaires pour établir une connexion avec la base de données MySQL. La connexion à la base de données MySQL est établie à l'aide de la bibliothèque mysql.connector.

2. Initialisation des Sockets

Le serveur utilise des sockets pour communiquer avec les clients. L'initialisation du socket se fait avec les paramètres AF_INET et SOCK_STREAM pour spécifier le type de connexion. Ensuite, le serveur est lié à une adresse spécifique (dans notre cas, 'localhost' sur le port 5000). Ces étapes permettent d'établir une interface de communication pour les clients se connectant au serveur.

3. Structures de Données

Le serveur utilise plusieurs structures de données pour suivre l'état du système. Les clients connectés, les demandes d'adhésion en attente, les utilisateurs bannis, et les utilisateurs authentifiés sont tous surveillés à l'aide de listes, de dictionnaires, et d'ensembles. Ces structures sont initialisées au début du serveur pour garantir une gestion efficace des interactions.

V. Gestion de l'inscription des clients

```
def inscription_handler(self, username, alias, password):
        """
        Gère le processus d'inscription pour un nouvel utilisateur.

        Args:
            username (str): Le nom d'utilisateur choisi par l'utilisateur.
            alias (str): L'alias choisi par l'utilisateur.
            password (str): Le mot de passe choisi par l'utilisateur.

        Returns:
            str: Message indiquant le résultat de l'inscription.
        """
        try:
            if not self.is_username_unique(username):
                return "Nom d'utilisateur déjà utilisé. Choisissez un autre."
            if not self.is_alias_unique(alias):
                return "Alias déjà utilisé. Choisissez un autre."

            if self.authenticate_user(username, password):
                return "Nom d'utilisateur déjà utilisé. Choisissez un autre."

            password_hash = hashlib.sha256(password.encode()).hexdigest()

            query = "INSERT INTO clients (username, alias, password_hash) VALUES (%s, %s, %s)"
            values = (username, alias, password_hash)
            self.cursor.execute(query, values)
            self.db_connection.commit()

            salons_a_integrer = ['Général']
            for salon in salons_a_integrer:
                query = f"INSERT INTO salons_adheres (username, salon_id) VALUES (%s, %s)"
                salon_id = self.get_salon_id(salon)
                values = (username, salon_id)
                self.cursor.execute(query, values)
                self.db_connection.commit()

            return "Inscription réussie!"
        except Exception as e:
            return f"Erreur lors de l'inscription : {e}"

    def is_alias_unique(self, alias):
        """
        Vérifie si l'alias est unique dans la base de données.

        Args:
            alias (str): L'alias à vérifier.
```

```
        Returns:
            bool: True si l'alias est unique, False sinon.
        """
        query_check = f"SELECT * FROM clients WHERE alias = '{alias}'"
        self.cursor.execute(query_check)
        result_check = self.cursor.fetchone()
        return result_check is None

    def is_username_unique(self, username):
        """
        Vérifie si le nom d'utilisateur est unique dans la base de données.

        Args:
            username (str): Le nom d'utilisateur à vérifier.

        Returns:
            bool: True si le nom d'utilisateur est unique, False sinon.
        """
        query_check = f"SELECT * FROM clients WHERE username = '{username}'"
        self.cursor.execute(query_check)
        result_check = self.cursor.fetchone()
        return result_check is None
```

La gestion de l'inscription des clients constitue une étape cruciale dans le fonctionnement du serveur. Cette section détaille le processus d'inscription, les vérifications effectuées pour assurer l'unicité des noms d'utilisateur et des alias, ainsi que la manière dont les informations sont stockées dans la base de données.

1. Méthode inscription_handler

La méthode inscription_handler prend en charge le processus d'inscription pour un nouvel utilisateur.

2. Méthode is_alias_unique

La méthode is_alias_unique vérifie si l'alias est unique dans la base de données. Elle effectue une requête SQL pour rechercher un alias similaire. Si aucun résultat n'est renvoyé, l'alias est considéré comme unique.

3. Méthode is_username_unique

La méthode is_username_unique est similaire à is_alias_unique, mais elle vérifie l'unicité du nom d'utilisateur dans la base de données.

VI. Opérations sur la base de données

```python
    def insert_message(self, client_id, salon_id, contenu):
        """
        Insère un message dans la base de données.

        Args:
            client_id (int): L'ID du client qui envoie le message.
            salon_id (int): L'ID du salon dans lequel le message est envoyé.
            contenu (str): Le contenu du message.
        """
        try:
            query = "INSERT INTO messages (client_id, salon_id, contenu) VALUES (%s, %s, %s)"
            values = (client_id, salon_id, contenu)
            self.cursor.execute(query, values)
            self.db_connection.commit()
        except Exception as e:
            print(f"Erreur lors de l'insertion du message dans la base de données : {e}")

    def get_salons(self):
        """
        Récupère la liste de tous les salons depuis la base de données.
        Returns:
            list: Liste des salons disponibles.
        """
        query = "SELECT * FROM salons"
        self.cursor.execute(query)
        salons = self.cursor.fetchall()
        return salons

    def get_user_salons(self, username):
        """
        Récupère la liste des salons auxquels un utilisateur est inscrit.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur.
        Returns:
            list: Liste des salons auxquels l'utilisateur est inscrit.
        """
        try:
            query = f"SELECT salon_id FROM salons_adheres WHERE username = '{username}'"
            self.cursor.execute(query)
            salons_adheres = self.cursor.fetchall()
            subscribed_salons = []
            for salon_id in salons_adheres:
                salon_name = self.get_salon_name(salon_id[0])
                subscribed_salons.append({"salon_id": salon_id[0], "salon_name": salon_name})
            return subscribed_salons
        except Exception as e:
            print(f"Error retrieving subscribed salons for {username}: {e}")
```

```
            return []
    def get_salon_name(self, salon_id):
        """
        Récupère le nom d'un salon à partir de son ID.
        Args:
            salon_id (int): L'ID du salon.
        Returns:
            str: Le nom du salon.
        """
        query = f"SELECT nom_salon FROM salons WHERE salon_id = {salon_id}"
        self.cursor.execute(query)
        result = self.cursor.fetchone()
        return result[0] if result else None

    def get_client_id(self, username):
        """
        Récupère l'ID du client à partir du nom d'utilisateur.
        Args:
            username (str): Le nom d'utilisateur du client.
        Returns:
            int or None: L'ID du client s'il existe, None sinon.
        """
        query = f"SELECT client_id FROM clients WHERE username = '{username}'"
        self.cursor.execute(query)
        result = self.cursor.fetchone()
        return result[0] if result else None

    def get_salon_id(self, salon):
        """
        Récupère l'ID d'un salon à partir de son nom
        Args:
            salon (str): Le nom du salon.
        Returns:
            int or None: L'ID du salon s'il existe, None sinon.
        """
        query = f"SELECT salon_id FROM salons WHERE nom_salon = '{salon}'"
        self.cursor.execute(query)
        result = self.cursor.fetchone()
        return result[0] if result else None
```

1. Insertion de Message : insert_message

Cette méthode insère un message dans la table "messages" de la base de données. Elle prend l'ID du client, l'ID du salon et le contenu du message en tant qu'arguments, puis exécute une requête SQL pour effectuer l'insertion. En cas d'erreur, elle affiche un message d'erreur.

2. Récupération des Salons : get_salons

Cette méthode récupère tous les salons disponibles à partir de la table "salons" de la base de données et renvoie une liste de résultats

3. Récupération des Salons d'un Utilisateur : get_user_salons

Cette méthode récupère la liste des salons auxquels un utilisateur est inscrit. Elle utilise une requête SQL pour obtenir les ID des salons auxquels l'utilisateur est inscrit dans la table "salons_adheres". Ensuite, elle utilise la méthode get_salon_name pour obtenir le nom de chaque salon à partir de son ID et retourne une liste d'objets contenant l'ID et le nom du salon.

4. Récupération du Nom d'un Salon par ID : get_salon_name

Cette méthode récupère le nom d'un salon à partir de son ID en utilisant une requête SQL sur la table "salons".

5. Récupération de l'ID d'un Client par Nom d'Utilisateur : get_client_id

Cette méthode récupère l'ID d'un client à partir de son nom d'utilisateur en utilisant une requête SQL sur la table "clients".

6. Récupération de l'ID d'un Salon par Nom : get_salon_id

Cette méthode récupère l'ID d'un salon à partir de son nom en utilisant une requête SQL sur la table "salons".

VII. Gestion des Clients et Authentification

1. Authentification et Gestion des Utilisateurs

```
def handle_client(self, client_socket, username, password):
        """
        Gère les interactions avec un client connecté au serveur.

        Args:
            client_socket (socket.socket): Le socket du client.
            username (str): Le nom d'utilisateur du client.
            password (str): Le mot de passe du client.
        """
        try:
            if self.is_user_banned(username):
                client_socket.send("Vous êtes banni. Réessayez dans quelques
minutes.".encode('utf-8'))
                return

            salons = self.get_salons()
            salons_data = json.dumps(salons)
            client_socket.send(salons_data.encode('utf-8'))
            self.update_client_status(username, 'connected')
            salons_adheres = self.get_user_salons(username)
            client_socket.send(json.dumps(salons_adheres).encode('utf-8'))

            if 'Général' in salons:
```

```
                self.pending_requests.pop('Général', None)

            while True:
                data = client_socket.recv(1024).decode('utf-8')

                if not data:
                    break

                elif data.startswith('AUTH:'):
                    _, auth_username, auth_password = data.split(':')
                    if self.authenticate_user(auth_username, auth_password):
                        client_socket.send("CONNECTED".encode('utf-8'))
                    else:
                        client_socket.send("AUTH_FAILED".encode('utf-8'))
                        client_socket.close()

                elif data.startswith('BAN:'):
                    _, target_username = data.split(':')
                    if self.is_user_permanently_banned(target_username):
                        client_socket.send("User is permanently banned.".encode('utf-8'))
                        client_socket.close()
                    else:
                        self.ban_user(target_username)

                elif data.startswith('JOIN:'):
                    salon_name = data.split(':')[1]

                    # Si l'utilisateur est déjà adhéré au salon, ignorer la demande
                    if self.is_user_adhered(username, salon_name):
                        print(f"L'utilisateur {username} est déjà adhéré au salon {salon_name}.")
                        continue

                    # Sinon, traiter la demande d'adhésion
                    self.handle_join_request(username, salon_name)

                elif data.startswith('REGISTER:'):
                    _, username, alias, password = data.split(':')
                    response = self.inscription_handler(username, alias, password)
                    client_socket.send(response.encode('utf-8'))
                    break

                elif data.startswith('MSG ('):
                    _, rest = data.split('(', 1)
                    salon, message = rest.split('):', 1)

                    insert_query = "INSERT INTO messages (client_id, salon_id, contenu) VALUES
(%s, %s, %s)"
```

```
                    insert_values = (self.get_client_id(username), self.get_salon_id(salon),
message)
                    self.cursor.execute(insert_query, insert_values)
                    self.db_connection.commit()

                    self.broadcast(f"{username} dans le salon {salon}: {message}")

                elif data.startswith('PRIVATE:'):
                    _, rest = data.split(':', 1)
                    self.handle_private_message(username, rest)
            except Exception as e:
                print(f"Erreur avec {username}: {e}")
            finally:
                self.update_client_status(username, 'disconnected')
                self.remove_client(client_socket, username)
```

La fonction handle_client gère les interactions avec un client connecté au serveur. Elle vérifie si l'utilisateur est banni, envoie la liste des salons disponibles, met à jour le statut du client et envoie la liste des salons auxquels il est inscrit. Ensuite, elle traite les actions du client telles que l'authentification, le bannissement, l'adhésion à un salon, l'envoi de messages, et les messages privés. En cas d'erreur, elle affiche un message d'erreur. Enfin, elle met à jour le statut du client à "déconnecté" et le retire de la liste des clients connectés.

2. Gestion des Adhésions aux Salons

```
def handle_join_request(self, username, salon_to_join):
        """
        Traite la demande d'adhésion d'un utilisateur à un salon.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur faisant la demande.
            salon_to_join (str): Le nom du salon auquel l'utilisateur veut adhérer.
        """
        if salon_to_join.lower() == "blabla":
            self.add_user_to_salon(username, salon_to_join)
        else:
            if salon_to_join not in self.pending_requests:
                self.pending_requests[salon_to_join] = []

            self.pending_requests[salon_to_join].append(username)
            print(f"Join request from {username} for salon {salon_to_join}.")
```

La méthode handle_join_request gère les demandes d'adhésion d'utilisateurs à des salons spécifiques.

- Validation du Salon : Vérifie si le salon spécifié (salon_to_join) est égal à "blabla" (cas spécial).
- Ajout Direct au Salon : Si le salon est "blabla", l'utilisateur est automatiquement ajouté à ce salon.
- Traitement des Demandes en Attente : Si le salon est différent de "blabla", la méthode vérifie s'il existe déjà des demandes en attente pour ce salon.

- Création de la File d'Attente : Si aucune file d'attente n'existe, elle est créée pour le salon spécifié.
- Ajout à la File d'Attente : Ajoute le nom d'utilisateur à la file d'attente du salon.
- Affichage de la Demande : Affiche un message informant du traitement de la demande, indiquant le nom d'utilisateur et le salon concernés.

3. Vérification de l'Adhésion et Actions Associées

```
def is_user_adhered(self, username, salon_name):
        """
        Vérifie si un utilisateur est adhérent à un salon spécifié.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur à vérifier.
            salon_name (str): Le nom du salon auquel vérifier l'adhésion.
        Returns:
            bool: True si l'utilisateur est adhérent, False sinon.
        """
        try:
            salon_id = self.get_salon_id(salon_name)

            query_check = f"SELECT * FROM salons_adheres WHERE username = '{username}' AND
salon_id = {salon_id}"
            self.cursor.execute(query_check)
            result_check = self.cursor.fetchone()

            return result_check is not None
        except Exception as e:
            print(f"Erreur lors de la vérification de l'adhésion de {username} au salon
{salon_name}: {e}")
            return False

    def accept_join_request(self, username, salon_to_join):
        """
        Accepte la demande d'adhésion d'un utilisateur à un salon spécifié.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur dont la demande est acceptée.
            salon_to_join (str): Le nom du salon auquel l'utilisateur est ajouté.
        """
        if salon_to_join in self.pending_requests and username in
self.pending_requests[salon_to_join]:
            self.pending_requests[salon_to_join].remove(username)
            if salon_to_join.lower() == "blabla":
                self.add_user_to_salon_direct(username, salon_to_join)
            else:
                self.add_user_to_salon(username, salon_to_join)

            message = f"{username} a rejoint le salon {salon_to_join}."
            self.broadcast(message)
```

```
    def reject_join_request(self, username, salon_to_join):
        """
        Rejette la demande d'adhésion d'un utilisateur à un salon spécifié.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur dont la demande est rejetée.
            salon_to_join (str): Le nom du salon dont la demande est rejetée.
        """
        if salon_to_join in self.pending_requests and username in
self.pending_requests[salon_to_join]:
            self.pending_requests[salon_to_join].remove(username)

            message = f"Votre demande d'adhésion au salon {salon_to_join} a été refusée. Accès
restreint"
            self.broadcast(message)
            self.remove_user_from_salon(username, salon_to_join)

    def remove_user_from_salon(self, username, salon_name):
        """
        Retire un utilisateur d'un salon spécifié.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur à retirer.
            salon_name (str): Le nom du salon duquel retirer l'utilisateur.
        """
        try:
            salon_id = self.get_salon_id(salon_name)

            query = f"DELETE FROM salons_adheres WHERE username = '{username}' AND salon_id =
{salon_id}"
            self.cursor.execute(query)
            self.db_connection.commit()

            print(f"Utilisateur {username} retiré du salon {salon_id}.")
        except Exception as e:
            print(f"Erreur lors de la suppression de {username} du salon {salon_name}: {e}")

    def add_user_to_salon(self, username, salon):
        """
        Ajoute un utilisateur à un salon spécifié.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur à ajouter.
            salon (str): Le nom du salon auquel ajouter l'utilisateur.
        """
        try:
            salon_id = self.get_salon_id(salon)

            if not salon_id:
                print(f"Le salon {salon} n'existe pas.")
                return
```

```
            query_check = f"SELECT * FROM salons_adheres WHERE username = '{username}' AND
salon_id = {salon_id}"
            self.cursor.execute(query_check)
            result_check = self.cursor.fetchone()

            if not result_check:
                query = f"INSERT INTO salons_adheres (username, salon_id) VALUES (%s, %s)"
                values = (username, salon_id)
                self.cursor.execute(query, values)
                self.db_connection.commit()

                print(f"Utilisateur {username} ajouté au salon {salon_id}.")

        except Exception as e:
            print(f"Erreur lors de l'ajout de {username} au salon {salon}: {e}")

    def handle_client_disconnect(self, username):
        """
        Gère la déconnexion d'un client en retirant ses demandes d'adhésion.
        Args:
            username (str): Le nom d'utilisateur du client qui se déconnecte.
        """
        try:
            for salon, demandeurs in self.pending_requests.items():
                if username in demandeurs:
                    demandeurs.remove(username)
                    break
        except Exception as e:
            print(f"Erreur lors de la gestion de la déconnexion de {username}: {e}")
```

- is_user_adhered :

Cette méthode vérifie si un utilisateur est adhérent à un salon spécifié. En utilisant le nom d'utilisateur et le nom du salon, elle récupère l'ID du salon, puis exécute une requête SQL pour vérifier si une entrée correspondante existe dans la table des adhésions aux salons. Elle renvoie True si l'utilisateur est adhérent et False sinon.

- accept_join_request :

Lorsqu'un administrateur accepte une demande d'adhésion d'un utilisateur à un salon, cette méthode retire l'utilisateur de la liste des demandes en attente. Si le salon est "blabla", l'utilisateur est ajouté directement à ce salon sans approbation supplémentaire. Sinon, l'utilisateur est ajouté au salon après avoir retiré sa demande en attente. Un message de diffusion informe les autres membres du salon de l'adhésion.

- reject_join_request :

Lorsqu'une demande d'adhésion est rejetée, cette méthode retire l'utilisateur de la liste des demandes en attente, envoie un message personnalisé à l'utilisateur rejeté, et retire cet utilisateur du salon auquel la demande a été refusée.

- remove_user_from_salon :

Cette méthode retire un utilisateur spécifié d'un salon particulier. Elle utilise le nom d'utilisateur et le nom du salon pour obtenir l'ID du salon, puis exécute une requête SQL pour supprimer l'entrée correspondante dans la table des adhésions aux salons.

- add_user_to_salon :

Ajoute un utilisateur à un salon spécifié. Elle vérifie d'abord si l'utilisateur n'est pas déjà adhérent au salon pour éviter les doublons, puis ajoute l'entrée correspondante dans la table des adhésions aux salons.

- handle_client_disconnect :

Lorsqu'un client se déconnecte, cette méthode parcourt la liste des demandes en attente pour chaque salon et retire le client de ces listes, évitant ainsi toute demande en attente liée à ce client.

VIII. Commandes et Opérations du Serveur :

1. Gestion des Clients et Statuts :

```
def remove_client(self, client_socket, username):
        """
        Retire un client de la liste des clients connectés.
        Args:
            client_socket (socket): Le socket du client à retirer.
            username (str): Le nom d'utilisateur du client à retirer.
        """
        try:
            for client, client_username in self.clients:
                if client == client_socket and client_username == username:
                    client_socket.close()
                    self.clients.remove((client_socket, client_username))
                    self.handle_client_disconnect(username)
                    break
        except Exception as e:
            print(f"Erreur lors de la déconnexion de {username}: {e}")

    def update_client_status(self, username, new_status):
        """
        Met à jour le statut d'un client dans la base de données.
        Args:
            username (str): Le nom d'utilisateur du client à mettre à jour.
            new_status (str): Le nouveau statut du client.
```

```
        """
        query = f"UPDATE clients SET status = '{new_status}' WHERE username = '{username}'"
        self.cursor.execute(query)
        self.db_connection.commit()

    def authenticate_user(self, username, password):
        """
        Authentifie un utilisateur en vérifiant le nom d'utilisateur et le mot de passe.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur à authentifier.
            password (str): Le mot de passe de l'utilisateur.
        Returns:
            bool: True si l'authentification réussit, False sinon.
        """
        query = f"SELECT password_hash FROM clients WHERE username = '{username}'"
        self.cursor.execute(query)
        result = self.cursor.fetchone()
        if result:
            stored_password_hash = result[0]
            input_password_hash = hashlib.sha256(password.encode()).hexdigest()
            return stored_password_hash == input_password_hash
        return False
```

Ces trois fonctions sont cruciales pour la gestion des clients connectés, la mise à jour de leurs statuts et l'authentification sécurisée des utilisateurs. Elles contribuent à maintenir l'intégrité du système en assurant une gestion appropriée des connexions et des informations d'authentification.

- remove_client :

Cette fonction gère la déconnexion d'un client en le retirant de la liste des clients connectés. Elle prend en paramètres le socket du client à retirer (client_socket) et le nom d'utilisateur du client (username). Elle parcourt la liste des clients connectés et ferme le socket du client correspondant. Ensuite, elle retire ce client de la liste et déclenche la gestion de la déconnexion du client. En cas d'erreur lors de la déconnexion, un message d'erreur est affiché.

- update_client_status :

Cette fonction met à jour le statut d'un client dans la base de données. Elle prend en paramètres le nom d'utilisateur du client à mettre à jour (username) et le nouveau statut du client (new_status). Elle construit une requête SQL pour mettre à jour le champ "status" de l'utilisateur dans la base de données. Après l'exécution de la requête, les changements sont confirmés dans la base de données.

- authenticate_user :

Cette fonction authentifie un utilisateur en vérifiant le nom d'utilisateur et le mot de passe fourni. Elle prend en paramètres le nom d'utilisateur (username) et le mot de passe (password). Elle effectue une requête SQL pour récupérer le hachage du mot de passe stocké dans la base de données. Ensuite, elle hache le mot de passe fourni et le compare avec le hachage stocké. Si les hachages correspondent, l'authentification réussit et la fonction retourne True, sinon elle retourne False.

2. Actions sur les Utilisateurs

```
def kick_user(self, target_username):
        """
        Expulse un utilisateur du serveur.
        Args:
            target_username (str): Le nom d'utilisateur de l'utilisateur à expulser.
        """
        for client, username in self.clients:
            if username == target_username:
                try:
                    client.send("Vous avez été expulsé du serveur. Vous êtes banni pendant 5
minutes.".encode('utf-8'))
                except Exception as e:
                    print(f"Erreur lors de l'envoi du message à {username}: {e}")
                finally:
                    self.banned_users[target_username] = time.time()
                    self.clients.remove((client, username))
                    client.close()
                break

    def ban_user(self, target_username):
        """
        Banni définitivement un utilisateur du serveur.
        Args:
            target_username (str): Le nom d'utilisateur de l'utilisateur à bannir.
        """
        for client, username in self.clients:
            if username == target_username:
                try:
                    client.send("Vous avez été banni définitivement du serveur !".encode('utf-
8'))
                except Exception as e:
                    print(f"Erreur lors de l'envoi du message à {username}: {e}")
                finally:
                    self.remove_user_from_database(target_username)
                    self.clients.remove((client, username))
                    client.close()
                break

    def remove_user_from_database(self, username):
        """
        Supprime un utilisateur de la base de données, y compris ses adhésions et messages.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur à supprimer.
        """
        try:
            client_id = self.get_client_id(username)
            query_delete_salon = f"DELETE FROM salons_adheres WHERE username = '{username}'"
```

```
            self.cursor.execute(query_delete_salon)
            query_delete_messages = f"DELETE FROM messages WHERE client_id = {client_id}"
            self.cursor.execute(query_delete_messages)
            query_delete_client = f"DELETE FROM clients WHERE username = '{username}'"
            self.cursor.execute(query_delete_client)
            self.db_connection.commit()

            print(f"Utilisateur {username} supprimé de la base de données.")
        except Exception as e:
            print(f"Erreur lors de la suppression de {username} de la base de données: {e}")

    def is_user_banned(self, username):
        """
        Vérifie si un utilisateur est banni du serveur.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur à vérifier.
        Returns:
            bool: True si l'utilisateur est banni, False sinon.
        """
        if username in self.banned_users:
            return time.time() - self.banned_users[username] < 300
        return False

    def kill_server(self):
        """
        Arrête le serveur de manière contrôlée.
        Envoie des messages d'arrêt aux clients, attend 10 secondes, puis ferme les connexions et
arrête le serveur.
        """
        print("Arrêt du serveur par commande KILL.")
        self.broadcast("Le serveur va s'arrêter dans 10 secondes. Merci de vous déconnecter.")
        time.sleep(10)

        for client, _ in self.clients:
            try:
                client.send("Le serveur va s'arrêter. Merci de vous déconnecter.".encode('utf-
8'))
                client.close()
            except Exception as e:
                print(f"Erreur lors de la fermeture de la connexion du client : {e}")

        os._exit(0)

    def handle_commands(self):
        """
        Gère les commandes entrées par l'utilisateur (utilisé dans le thread séparé).
        Les commandes possibles sont KICK, BAN, KILL, ACCEPT, et REJECT.
        """
```

```
        while True:
            command = input("Tapez une commande (KICK/BAN/KILL/ACCEPT/REJECT): ").strip()
            if command.startswith('KICK:'):
                _, target_username = command.split(':')
                self.kick_user(target_username)
            elif command.startswith('BAN:'):
                _, target_username = command.split(':')
                self.ban_user(target_username)
            elif command.startswith('ACCEPT:'):
                _, username, salon_to_join = command.split(':')
                self.accept_join_request(username, salon_to_join)
            elif command.startswith('REJECT:'):
                _, username, salon_to_join = command.split(':')
                self.reject_join_request(username, salon_to_join)
            elif command == 'KILL':
                self.kill_server()

    def broadcast(self, message):
        """
        Diffuse un message à tous les clients connectés.
        Args:
            message (str): Le message à diffuser.
        """
        for client, username in self.clients:
            try:
                client.send(message.encode('utf-8'))
            except Exception as e:
                print(f"Erreur lors de l'envoi du message à {username}: {e}")
                self.remove_client(client, username)

    def update_password(self, username, new_password):
        """
        Met à jour le mot de passe d'un utilisateur dans la base de données.
        Args:
            username (str): Le nom d'utilisateur de l'utilisateur.
            new_password (str): Le nouveau mot de passe.
        Returns:
            str: Un message indiquant le succès ou l'échec de la mise à jour.
        """
        try:
            password_hash = hashlib.sha256(new_password.encode()).hexdigest()
            query = f"UPDATE clients SET password_hash = '{password_hash}' WHERE username =
'{username}'"
            self.cursor.execute(query)
            self.db_connection.commit()
            return "Mot de passe mis à jour avec succès!"
        except Exception as e:
            return f"Erreur lors de la mise à jour du mot de passe : {e}"
```

Dans cette section, nous avons plusieurs méthodes qui gèrent les interactions avec les utilisateurs, que ce soit pour les exclure temporairement, les bannir définitivement, les supprimer de la base de données, vérifier leur statut de bannissement, ou même arrêter le serveur de manière contrôlée.

3. Expulsion Temporaire (kick_user)

La méthode kick_user est responsable de l'expulsion temporaire d'un utilisateur du serveur. Elle prend en paramètre le nom d'utilisateur ciblé. Elle parcourt la liste des clients connectés, envoie un message d'expulsion au client ciblé, enregistre le temps d'expulsion, retire le client de la liste, et ferme la connexion.

4. Bannissement Définitif (ban_user)

La méthode ban_user bannit définitivement un utilisateur du serveur. Tout comme kick_user, elle envoie un message de bannissement au client, supprime l'utilisateur de la base de données et de la liste des clients connectés, et ferme la connexion.

5. Suppression de l'Utilisateur de la BDD (remove_user_from_database)

La méthode remove_user_from_database supprime un utilisateur de la base de données, y compris ses adhésions aux salons et ses messages. Elle utilise le nom d'utilisateur pour récupérer l'ID du client, puis exécute des requêtes SQL pour supprimer les données associées.

6. Vérification du Statut de Bannissement (is_user_banned)

La méthode is_user_banned vérifie si un utilisateur est actuellement banni du serveur. Elle prend le nom d'utilisateur en paramètre, consulte le dictionnaire des utilisateurs bannis, et retourne True si l'utilisateur est banni et le temps écoulé depuis le bannissement est inférieur à 300 secondes.

7. Arrêt Contrôlé du Serveur (kill_server)

La méthode kill_server effectue un arrêt contrôlé du serveur. Elle envoie des messages d'arrêt à tous les clients, attend 10 secondes, puis ferme les connexions et arrête le serveur. Cette méthode est déclenchée par la commande "KILL".

8. Gestion des Commandes Administratives (handle_commands)

La méthode handle_commands permet à l'administrateur du serveur d'entrer des commandes administratives telles que KICK, BAN, ACCEPT, REJECT, et KILL. Elle tourne en boucle, analysant les commandes entrées et appelant les méthodes appropriées en fonction de la commande saisie.

9. Diffusion de Messages à Tous les Clients (broadcast)

La méthode broadcast envoie un message à tous les clients connectés. Elle parcourt la liste des clients et envoie le message à chacun. Si une erreur se produit lors de l'envoi, elle utilise la méthode remove_client pour retirer le client défaillant de la liste.

10. Mise à Jour du Mot de Passe (update_password)

La méthode update_password permet de mettre à jour le mot de passe d'un utilisateur dans la base de données. Elle prend le nom d'utilisateur et le nouveau mot de passe en paramètres, génère le hash correspondant, puis exécute une requête SQL pour effectuer la mise à jour.

IX. Démarrage Sécurisé du Serveur et Gestion des Connexions

La fonction start_server joue un rôle crucial dans le lancement sécurisé du serveur, la gestion de l'authentification initiale, et l'acceptation des connexions entrantes. Cette fonction assure un processus d'initialisation robuste et instaure une surveillance constante des nouveaux clients.

```
def start_server(self):
        """
        Démarre le serveur, gère l'authentification initiale et écoute les connexions entrantes.
        """
        while True:
            username = input("Nom d'utilisateur : ")
            password = input("Mot de passe : ")

            if self.authenticate_user(username, password) and username in ['admin', 'server']:
                print(f"Authentification réussie pour l'utilisateur {username}.")
                self.authenticated_users.add(username)
                break
            else:
                print("Authentification échouée. Veuillez réessayer.")
        self.server.listen(5)
        print("Serveur en attente de connexions...")
        command_handler = threading.Thread(target=self.handle_commands)
        command_handler.start()

        while True:
            client_socket, client_address = self.server.accept()
            credentials = client_socket.recv(1024).decode('utf-8').split(':')

            if len(credentials) >= 2:
                if credentials[0] == "INSCRIPTION":
                    _, username, alias, password = credentials
                    response = self.inscription_handler(username, alias, password)
                    client_socket.send(response.encode('utf-8'))
                    client_socket.close()
                else:
                    username, password = credentials[:2]

                    if self.authenticate_user(username, password):
                        self.clients.append((client_socket, username))
                        client_handler = threading.Thread(target=self.handle_client,
args=(client_socket, username, password))
                        client_handler.start()
                        client_socket.send("CONNECTED".encode('utf-8'))
                    else:
                        client_socket.send("AUTH_FAILED".encode('utf-8'))
                        client_socket.close()
            else:
                print("Erreur : Liste credentials invalide.")
```

1. Authentification Initiale :

La fonction commence par une phase d'authentification initiale où l'administrateur doit fournir un nom d'utilisateur et un mot de passe valides. Cette étape vise à garantir que seule une entité autorisée peut accéder au serveur. En cas de succès, l'utilisateur est ajouté à la liste des utilisateurs authentifiés.

2. Attente des Connexions :

Après l'authentification réussie, le serveur entre en mode d'attente de connexions entrantes. La méthode listen permet d'accepter jusqu'à cinq connexions en attente.

3. Gestion des Commandes Administratives :

Un thread distinct, command_handler, est démarré pour gérer les commandes administratives telles que KICK, BAN, etc. Cette séparation des tâches garantit que les commandes peuvent être traitées indépendamment des connexions entrantes.

4. Traitement des Connexions Entrantes :

Le serveur reste dans une boucle d'acceptation des connexions. Lorsqu'une connexion est établie, les informations d'identification sont reçues du client. Selon le type de connexion (inscription ou authentification), le serveur réagit en conséquence.

- Pour une demande d'inscription, les détails de l'utilisateur sont extraits, le gestionnaire d'inscription est appelé, et la réponse est renvoyée au client.
- Pour une authentification, le serveur vérifie les informations d'identification, ajoute le client à la liste des clients connectés s'il est authentifié, et lance un thread distinct (client_handler) pour gérer les interactions avec ce client.

5. Sécurité et Gestion d'Erreurs :

La fonction intègre des mécanismes de sécurité, notamment la transmission sécurisée des informations d'identification et la gestion appropriée des erreurs. Les erreurs potentielles, telles que des informations d'identification incorrectes, déclenchent des messages explicatifs et des actions appropriées, comme la fermeture de la connexion.

6. Flux de Contrôle :

Le flux de contrôle de la fonction est organisé de manière à maintenir la stabilité du serveur tout en réagissant de manière flexible aux différentes situations. La boucle principale assure une surveillance constante des connexions, tandis que les threads séparés gèrent les commandes administratives et les clients individuels.