RAPPORT DE PROJET
Programmation Réseaux

Nizar Karroud

Année scolaire 2024-2025

Table de matière

Introduction

Objectif

InSync est une application de chat web développée pour le département informatique de notre université. Elle vise à fournir une plateforme conviviale où étudiants et enseignants peuvent collaborer, échanger des idées et partager des ressources dans un environnement sécurisé et facile à utiliser.

Fonctionnalités principales

- Discussions en temps réel entre étudiants et enseignants

- Création de groupes de discussion pour projets ou matières spécifiques

- Partage sécurisé de fichiers et de ressources

- Notifications pour les messages importants et les annonces

Public cible

Cette application est conçue pour les étudiants, les enseignants et le personnel administratif du département informatique.

Technologies utilisées

- Frontend : HTML , CSS , React , Javascript

- Backend : Flask REST API / Websocket

- Base de données : MongoDB , PostrgreSQL

Installation et mise en route

1. Clonage du Dépôt GitHub

Commencez par cloner le dépôt depuis GitHub ou installer le zip.

https://github.com/NizarKarroud/InSync

2. Backend

1. Accédez au répertoire du backend avec la commande `cd chatapp`.

2. Installez les dépendances Python nécessaires en exécutant :
```
pip install -r requirements.txt
```
3. Personnalisez le fichier `.env` selon vos besoins, notamment pour les configurations de base de données et les clés secrètes.

4. Assurez-vous que PostgreSQL et MongoDB sont installés et configurés sur votre machine.

3. Frontend

1. Installez npm et Node.js si ce n'est pas déjà fait.

2. Mettez à jour les adresses IP dans tous le dossier notamment dans le fichier `package-lock.json` pour refléter votre configuration locale

4. Lancer le Backend

Pour démarrer le backend, exécutez simplement le fichier `app.py` en utilisant Python :
```
python app.py
```

5. Lancer le Frontend

Pour démarrer le frontend, accédez au répertoire du frontend avec `cd chatapp` et exécutez :
```
```

npm start
```

# Architecture du Système

L'application InSync utilise une architecture en trois tiers (3-tier architecture) afin de séparer clairement les différentes responsabilités du système et de faciliter la maintenance, l'évolutivité et la sécurité.

L'architecture en trois tiers est divisée en trois couches principales :

## Couche de Présentation (Front-End)

Cette couche gère l'interface utilisateur et les interactions avec l'utilisateur final.

Technologies utilisées : HTML, CSS, JavaScript (React).

## Couche Logique (Back-End)

Cette couche contient la logique de traitement des données et les règles applicatives.

Elle reçoit les requêtes de l'utilisateur, traite les données, et renvoie les résultats à la couche de présentation.

Technologies utilisées : Python Flask et Websockets

## Couche de Données (Base de Données)

Cette couche est responsable du stockage des données.

Elle interagit avec des systèmes de gestion de base de données pour r récupérer, stocker et manipuler les informations.

Technologies utilisées : PostgreSQL MongoDB .
```

Communication entre les Couches

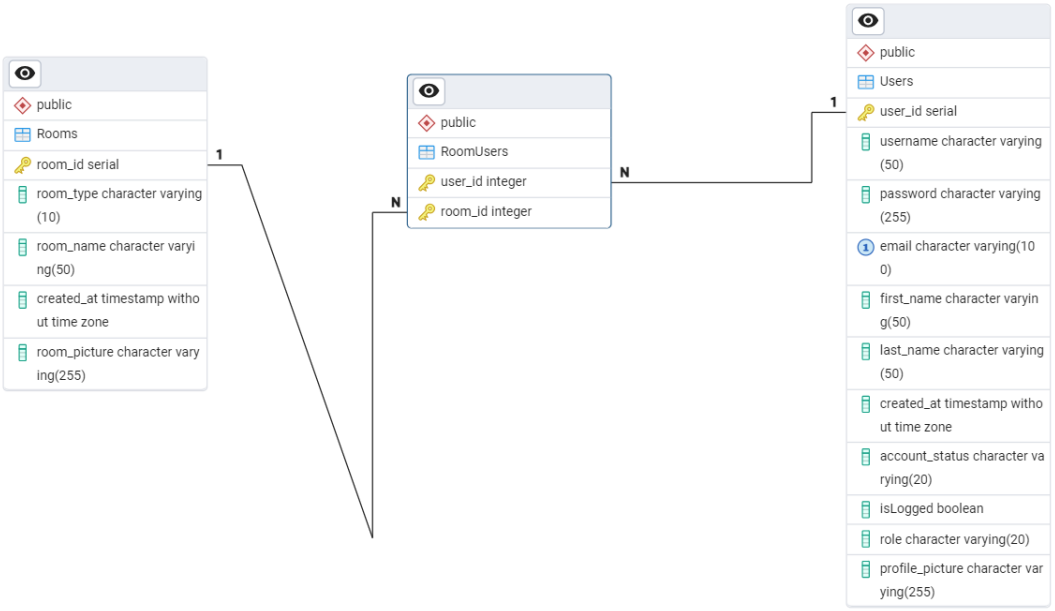
Les trois couches communiquent de manière distincte :

Couche de présentation ↔ Couche logique : La couche de présentation envoie des requêtes HTTP ou des WebSocket vers le back-end pour récupérer ou envoyer des données.

Couche logique↔ Couche de données : Le back-end interagit avec la base de données à l'aide de l'ORM (Object–relational mapping) SQLAlchemy et PyMongo pour la gestion des données non relationnelles

Schéma de la base de données

PostgreSQL Database Structure:



MongoDB Collections Structure

MongoDB offre la possibilité de travailler avec des collections de type timeseries, une fonctionnalité optimisée pour le stockage et l'analyse de données chronologiques.

Ce type de collection est particulièrement utile pour enregistrer des événements qui se produisent sur une période, comme les logs, les données de capteurs, les transactions financières, ou les événements de connexion des utilisateurs.

Les collections timeseries sont structurées autour de deux éléments principaux :

- **timestamp** : Un champ qui enregistre la date et l'heure de chaque document, essentiel pour organiser et interroger les données chronologiques.
- **meta** : Un champ optionnel permettant de regrouper les données par un identifiant ou une catégorie, ce qui peut être utile pour des analyses spécifiques (par exemple, le nom d'un utilisateur ou un identifiant de chat pour suivre les messages).

1. Registration Collection

Field Name	Data Type	Description
username	String	The username of the user registering.
email	String	The email address of the user.
password	String	The hashed password for the user's account.
first_name	String	The first name of the user.
last_name	String	The last name of the user.
registration_date	DateTime	The date and time the user registered (in UTC).
status	String	The current status of the registration (`pending`, `approved`, `rejected`).
admin_comments	String	Comments or feedback provided by an administrator.
role	String	The role assigned to the user (e.g., `student`, `teacher`, `admin`).

2. LoginAttempt Timeseries Collection

Field Name	Data Type	Description
attempt_time	DateTime	The date and time of the login attempt (in UTC).
username	String	The username provided during the login attempt.
ip_address	String	The IP address from which the login attempt was made.
success	Boolean	Indicates whether the login attempt was successful (`true`/`false`).

3. Notifications Timeseries Collection

Field Name	Data Type	Description
user_id	String	The unique identifier of the user .
room_id	String	The unique identifier of the chat room where the message was sent.
timestamp	DateTime	The date and time the message was sent (in UTC).
room_name	String	The room name (groupe name or the name of the other user if it's a direct message)

4. UserMessages Timeseries Collection

Field Name	Data Type	Description
sender_id	String	The unique identifier of the user sending the message.
room_id	String	The unique identifier of the chat room where the

		message is sent.
timestamp	DateTime	The date and time the message was sent (in UTC).
message_text	String	The text content of the message.
message_type	String	The type of message (`text`, `image`, `file`).
attachments	String	Path to the file or resource attached to the message.

5. OnlineUsers Collection

Field Name	Data Type	Description
user_id	String	The unique identifier of the user connected

6. UserSockets Collection

Field Name	Data Type	Description
user_id	String	The unique identifier of the user connected
socket_id	string	The unique identifier of the users socket

7. JoinedUsers Collection

Field Name	Data Type	Description
user_id	String	The unique identifier of the user connected
room_id	string	

Structure du code

Organisation Générale

Le code est divisé en plusieurs modules pour assurer une séparation claire des préoccupations. Cette organisation permet une gestion simplifiée des différentes fonctionnalités du projet

Arborescence des Fichiers

L'arborescence du projet InSync est organisée comme suit :

```
└────chatapp
   │  package-lock.json
   │  package.json
   │
   │
   ├────api
   │   │  .env
   │   │  app.py
   │   │  models.py
   │   │  requirements.txt
   │   ├────static
   │   │   │  openapi.json
   │   │   │
   │   │   └────files
   │   │      ├────rooms
   │   │      │
   │   │      └────users
   │   │
   │   ├────templates
```

```
│   │       new-email.html
│   │
├──────public
│       index.html
│
└──────src
    │   index.js
    │
    ├──────components
    │   ├──────dashboard
    │   │           chats.js
    │   │           dashboard.js
    │   │           dms.js
    │   │           groupdial.js
    │   │           groups.js
    │   │           settingdial.js
    │   │           socket.js
    │   │           userdial.js
    │   │
    │   └──────login
    │           forgotpwdsection.js
    │           loginpage.js
    │           loginsection.js
    │           registersection.js
    │           reset_password.js
    │
    └──────styles
                login.css
```

Documentation de l’API

La documentation complète de l'API est accessible via la route suivante : /api/docs. Cette documentation fournit une vue détaillée de toutes les routes disponibles, des paramètres requis, des réponses attendues, ainsi que des exemples d'utilisation pour chaque endpoint.

Testing

Utilisation de Postman pour Tester l'API

Dans le cadre du développement de l'API, **Postman** a été utilisé comme outil principal pour effectuer des tests approfondis des endpoints. Cette plateforme intuitive a permis de :

- **Valider les fonctionnalités des endpoints :** En envoyant des requêtes HTTP (GET, POST, PUT, DELETE) avec différents paramètres et charges utiles pour s'assurer qu'ils fonctionnent comme prévu.
- **Tester la gestion des erreurs :** Vérifier que l'API renvoie les codes d'état HTTP appropriés (ex. 200 pour succès, 400 pour requêtes mal formées, 401 pour accès non autorisé).
- **Examiner les réponses :** Inspecter les formats JSON renvoyés pour s'assurer qu'ils sont conformes aux attentes.
- **Tester les fonctionnalités de sécurité :** Utiliser des tokens JWT pour simuler des authentifications et valider les permissions et l'accès restreint à certains endpoints.

Tests Manuels par les Utilisateurs

En parallèle, des **tests manuels** ont été réalisés par un groupe restreint d'utilisateurs. Ces tests avaient pour objectif de :

- Identifier les bugs ou incohérences non détectés par les tests techniques.
- Tester les fonctionnalités de manière intuitive pour garantir une expérience utilisateur fluide.
- Récolter des retours sur l'ergonomie et les performances de l'application.

Ces tests ont permis de simuler des scénarios réels, comme la création de comptes, la gestion des groupes, et l'envoi de messages , etc…..

Automatisation des Tests avec Pytest (Futur Plan)

Pour optimiser le processus de test et garantir la fiabilité de l'API sur le long terme, une stratégie d'automatisation est prévue en utilisant **Pytest**. Cette approche permettra de :

- **Automatiser les tests unitaires :** Pour valider chaque fonction et module individuellement.
- **Réaliser des tests d'intégration :** Vérifier que les composants de l'API fonctionnent correctement ensemble.
- **Effectuer des tests de régression :** S'assurer que les nouvelles modifications n'introduisent pas de bugs dans les fonctionnalités existantes.
- **Gagner du temps :** Réduire l'effort manuel, en particulier lors des cycles de mise à jour fréquents.

Considérations de sécurité

Hashage des Mots de Passe

La bibliothèque werkzeug.security, offre une solution simple et sécurisée pour la gestion des mots de passe. Elle permet de hacher les mots de passe en utilisant PBKDF2 avec SHA-256, une méthode standard et robuste.

Pourquoi PBKDF2 avec SHA-256 ?

Protection contre les attaques par force brute : PBKDF2 (Password-Based Key Derivation Function 2) est conçu pour rendre le calcul des hachages coûteux en termes de temps et de ressources. Cela ralentit considérablement les attaques automatisées visant à deviner un mot de passe.

SHA-256 est un algorithme cryptographique reconnu et largement adopté pour sa sécurité.

Le mot de passe en clair n'est jamais enregistré ni transmis

Seul le hachage du mot de passe est enregistré dans la base de données. Cela garantit que même en cas de compromission, les mots de passe réels des utilisateurs restent protégés.

JWT (JSON Web Tokens)

Structure d'un JSON Web Token (JWT)

Un JSON Web Token (JWT) est un format compact et sécurisé utilisé pour transmettre des informations entre deux parties, généralement un client et un serveur. Ces informations sont codées sous forme d'un token signé numériquement, garantissant son intégrité et son authenticité.

Composants d'un JWT

Un JWT est composé de trois parties principales : l'en-tête (Header), la charge utile (Payload) et la signature.

- **Header (En-tête)** :
  L'en-tête spécifie l'algorithme de signature utilisé, par exemple HMAC SHA256, et le type de token, qui dans ce cas est un JWT.
  Exemple d'en-tête :

{
"alg": "HS256",
"typ": "JWT"
}

- **Payload (Charge utile)** :
  Le payload contient les revendications, qui sont des informations encodées sous forme de JSON.

  J'ai inclut l'ID utilisateur et le nom d'utilisateur, ainsi que des données supplémentaires comme la date d'expiration (24 heures).
  Exemple de payload :
  {
  "user_id": "12345",
  "username": "johndoe",
  "exp": 1632751294
  }

- **Signature** :
  La signature est utilisée pour garantir l'intégrité et l'authenticité du token. Elle est générée en combinant l'en-tête et le payload, puis en les signant à l'aide d'une clé secrète.

Clé Secrète et Configuration

La clé secrète utilisée pour signer les JWT est stockée dans `app.config["JWT_SECRET_KEY"]`. Elle est utilisée lors de la création du token pour garantir que celui-ci est valide et n'a pas été modifié. Cette clé doit être stockée de manière sécurisée et ne doit pas être exposée.

Logging des Tentatives de Connexion

Enregistrement des tentatives de connexion, qu'elles soient réussies ou échouées dans la collection login_attempts

Limitation du Taux de Requêtes (Rate Limiting)

Chaque endpoint est soumis à des limites de requêtes par addresse IP Intégré via Flask-Limiter , pour éviter les abus, par exemple les attaques par déni de service (DoS) ou par force brute.

Validation des Fichiers Téléchargés

Validation des extensions autorisées pour éviter l'exécution de fichiers malveillants.
Limitation de la taille des fichiers pour prévenir les attaques visant à épuiser les ressources serveur.

Prévention des Injections SQL

SQLAlchemy : Un ORM (Object-Relational Mapping) qui protège automatiquement contre les injections SQL en utilisant des requêtes paramétrées.
MongoDB : Pour les bases NoSQL, validation des schémas pour réduire les risques d'injection.

Contrôle d'Accès aux Groupes via Codes Chiffrés

Pour rejoindre un groupe, l'utilisateur doit fournir un code d'accès chiffré (room_code). Ce code est généré en combinant des informations sensibles sur le groupe, comme l'identifiant du groupe (room_id) et son nom (room_name), puis en chiffrant ces données avec une clé secrète stockée de manière sécurisée.

Formation du Code :
Le code d'accès est une chaîne chiffrée qui encapsule :

- L'ID du groupe : Identifiant unique du groupe dans la base de données.
- Le nom du groupe : Pour identifier le groupe de manière lisible et explicite.

Ces informations sont sérialisées en format JSON, puis chiffrées avec un algorithme de chiffrement symétrique via Fernet de la biblio cryptography.

Assurance d'un Accès Sécurisé :

- Seuls les utilisateurs authentifiés peuvent tenter de rejoindre un groupe.
- Lorsque l'utilisateur soumet un code, il est d'abord déchiffré pour extraire les informations qu'il contient. Si le code est invalide, mal formé, ou falsifié, l'accès est immédiatement refusé.
- Une fois le code validé, le système vérifie si l'utilisateur est déjà membre du groupe. Si l'utilisateur est déjà inscrit, l'opération est bloquée.

- Seules les personnes ayant reçu un code valide peuvent rejoindre le groupe, empêchant les accès non autorisés.

Conclusion

L'application InSync a été conçue pour offrir une plateforme de communication moderne, sécurisée et efficace pour le département informatique de notre université. Grâce à une architecture en trois tiers bien définie, InSync permet une gestion fluide des interactions entre les utilisateurs tout en garantissant la sécurité des échanges grâce à des mécanismes de hachage des mots de passe et d'authentification avec JWT.

Le projet a mis en œuvre des technologies robustes, comme React, Flask, PostgreSQL et MongoDB, afin de répondre aux besoins de notre public cible — étudiants, enseignants et personnel administratif. L'intégration de fonctionnalités telles que la messagerie en temps réel, la gestion des groupes, le partage sécurisé de fichiers et la gestion des notifications a permis de créer un environnement interactif et productif.