Rapport de l'application de chat en temps réel

SBAI Sami
IKRAM Iddouch
ZAROIL Khadija

I : Architecture générale

Cette application de chat en temps réel est construite selon une architecture moderne client-serveur avec plusieurs couches technologiques qui travaillent ensemble pour offrir une expérience utilisateur fluide et réactive.

1 : Stack technologique

- **Backend** : Node.js avec Express.js
- **Frontend** : React avec gestion d'état via Zustand
- **Base de données** : MongoDB (avec Mongoose comme ORM)
- **Temps réel** : Socket.IO pour les communications bidirectionnelles
- **Stockage d'état temporaire** : Redis pour la gestion des utilisateurs en ligne

II : Architecture générale

Le backend suit une architecture MVC (Modèle-Vue-Contrôleur) modifiée, avec une séparation claire des responsabilités :

1 : Points d'entrée et configuration

L'application est initialisée dans index.js, qui configure les middleware Express essentiels (CORS, parsing JSON, cookies), définit les routes API et démarre le serveur. Le serveur HTTP est partagé avec Socket.IO pour permettre des communications en temps réel.

2 : Système d'authentification

L'authentification utilise une approche basée sur JWT (JSON Web Tokens) stockés dans des cookies. Cette méthode offre plusieurs avantages :

- Sécurité améliorée par rapport au stockage local
- Persistance des sessions entre les rechargements de page
- Protection contre les attaques CSRF grâce aux options de cookie

Le processus d'authentification comporte plusieurs étapes :

1. L'utilisateur s'inscrit ou se connecte via les endpoints API

2. Le serveur valide les informations, crée/retrouve l'utilisateur dans la base de données
3. Un JWT est généré et stocké dans un cookie HTTP-only
4. Le middleware protectRoute vérifie ce token pour les routes protégées

3 : Gestion des messages

Le système de messagerie est conçu pour permettre des conversations bidirectionnelles entre utilisateurs. Les contrôleurs gèrent :

- La récupération des utilisateurs disponibles pour discuter
- La récupération des messages entre deux utilisateurs spécifiques
- L'envoi de nouveaux messages (texte et images)

4 : Modèles de données

L'application utilise deux modèles de données principaux :

- **User** : Stocke les informations utilisateur (email, nom, mot de passe hashé, photo de profil)
- **Message** : Enregistre les messages avec leurs métadonnées (expéditeur, destinataire, contenu, horodatage)

5 : Middleware

Le middleware d'authentification protectRoute joue un rôle crucial en :

- Vérifiant la présence et la validité du token JWT
- Récupérant les informations utilisateur associées
- Attachant ces informations à l'objet requête pour les contrôleurs

III : Mécanisme de temps réel

1 : Socket.IO

Socket.IO est utilisé pour créer une connexion bidirectionnelle persistante entre le client et le serveur, permettant :

- Une réception instantanée des messages
- Des mises à jour en temps réel du statut des utilisateurs (en ligne/hors ligne)
- Des notifications de connexion/déconnexion

La configuration de Socket.IO met en place :

- Des gestionnaires de connexion/déconnexion
- Des mappages entre ID utilisateur et ID socket
- Des émetteurs d'événements pour informer les clients des changements

2 : Redis pour la gestion des statuts en ligne

Redis est utilisé comme solution de stockage éphémère pour gérer les statuts "en ligne". Ce choix présente plusieurs avantages :

- Performances élevées (opérations en mémoire)
- Fonctionnalités spécialisées comme les ensembles et les expirations automatiques
- Persistance entre les redémarrages du serveur
- Scalabilité horizontale potentielle

Le système de détection de présence fonctionne selon ces principes :

1. Quand un utilisateur se connecte, son ID est ajouté à un ensemble Redis avec un TTL de 5 minutes
2. Le client envoie périodiquement des "pings" pour renouveler ce TTL
3. En cas de déconnexion normale, l'ID est retiré immédiatement
4. En cas de déconnexion anormale, l'entrée expire automatiquement après 5 minutes

IV : Structure du frontend

1 : Gestion d'état avec Zustand

Zustand est utilisé comme gestionnaire d'état, offrant une alternative légère à Redux. L'application utilise deux stores principaux :

- **useAuthStore** : Gère l'état d'authentification, les connexions Socket.IO et les utilisateurs en ligne
- **useChatStore** : Gère les messages, les utilisateurs et la sélection de conversations

Cette approche permet un découplage propre entre l'interface utilisateur et la logique métier.

2 : Composants React

L'interface est structurée autour de composants clés :

- **App** : Composant racine, gère les routes et l'initialisation
- **ChatContainer** : Affiche les conversations, incluant l'historique des messages
- **MessageInput** : Gère la saisie et l'envoi de messages (texte et images)
- **Composants auxiliaires** : Navbar, ChatHeader, etc.

3 : Flux des données

Le flux de données dans l'application suit un modèle unidirectionnel :

1. Les actions utilisateur déclenchent des fonctions dans les stores
2. Ces fonctions effectuent des appels API via Axios ou des émissions Socket.IO
3. Les réponses mettent à jour l'état dans les stores
4. Les composants sont re-rendus avec les nouvelles données

V : Fonctionnalités clés

1 : Système d'authentification

- Inscription avec validation des entrées
- Connexion sécurisée avec comparaison des hashs de mots de passe
- Maintien de session via cookies JWT
- Vérification automatique de l'authentification au chargement

2 : Messagerie en temps réel

- Envoi et réception instantanés des messages
- Support pour le texte et les images (via Cloudinary)
- Affichage des messages avec distinction expéditeur/destinataire
- Formatage de l'horodatage des messages

3 : Indicateurs de présence

- Affichage des utilisateurs actuellement en ligne
- Mise à jour en temps réel du statut de connexion
- Mécanisme de "heartbeat" pour maintenir le statut en ligne

4 : Gestion des profils

- Mise à jour de la photo de profil (stockée sur Cloudinary)
- Affichage des informations utilisateur dans l'interface

VI : Mécanismes de sécurité

L'application implémente plusieurs mécanismes de sécurité :

- Hachage des mots de passe avec bcrypt (avec salt)
- Cookies JWT HTTP-only pour prévenir les attaques XSS
- Protection des routes sensibles via middleware d'authentification
- Validation des entrées côté serveur
- Stockage sécurisé des images via Cloudinary

VII : Expérience utilisateur

L'interface utilisateur est construite pour optimiser l'expérience :

- Navigation conditionnelle basée sur l'état d'authentification
- Indicateurs de chargement pendant les opérations asynchrones
- Notifications toast pour les confirmations et erreurs
- Interfaces réactives pour différentes tailles d'écran
- Thème personnalisable (via useThemeStore)

VIII : Points d'amélioration potentiels

Bien que l'application soit bien conçue, quelques améliorations pourraient être envisagées :

1 : Fonctionnalités techniques

- Implémenter le défilement automatique aux nouveaux messages
- Ajouter la synchronisation de l'état entre différents onglets/appareils
- Améliorer la gestion des erreurs réseau et des reconnexions Socket.IO
- Mettre en œuvre un système de notification pour les messages non lus

2 : Sécurité et performance

- Ajouter une limitation de débit pour prévenir les abus
- Implementer l'authentification à deux facteurs

- Optimiser le chargement des messages (pagination, chargement à la demande)
- Ajouter un cryptage de bout en bout pour les messages

IX : Flux des interactions en temps réel

Pour illustrer le fonctionnement du mécanisme de temps réel, voici un exemple détaillé du flux d'une conversation :

1 : Connexion d'un utilisateur

1. L'utilisateur se connecte via le formulaire de login
2. Le serveur authentifie et renvoie un token JWT
3. Une connexion Socket.IO est établie avec l'ID utilisateur
4. Le serveur ajoute l'utilisateur à Redis comme "en ligne"
5. Tous les clients connectés sont notifiés du nouvel utilisateur en ligne

2 : Envoi d'un message

1. L'utilisateur compose et envoie un message
2. Le client fait une requête HTTP POST au serveur
3. Le serveur enregistre le message dans MongoDB
4. Le serveur utilise Socket.IO pour notifier le destinataire
5. L'interface du destinataire met à jour la conversation en temps réel

3 : Maintien du statut "en ligne"

1. Le client envoie un "ping" toutes les minutes via Socket.IO
2. Le serveur renouvelle le TTL de l'utilisateur dans Redis
3. Si un utilisateur ferme son navigateur sans déconnexion propre
4. Après 5 minutes sans ping, Redis expire automatiquement l'entrée
5. Le serveur détecte ce changement et notifie les autres utilisateurs

X : Conclusion

Cette application de chat en temps réel démontre une architecture bien pensée qui combine des technologies modernes pour offrir une expérience utilisateur fluide. L'utilisation de Socket.IO et Redis pour les fonctionnalités en temps réel, couplée à MongoDB pour la persistance des données et à React/Zustand pour l'interface utilisateur, crée un système robuste et réactif.

La séparation claire des responsabilités entre le backend et le frontend, ainsi que l'utilisation de patterns de conception éprouvés, rendent le code maintenable et extensible. Les mécanismes de sécurité implémentés protègent les données utilisateur tout en permettant une expérience utilisateur sans friction.