Sorbonne Université
Faculté de Science et d'ingénierie
Département Informatique

Rapport du PC3R
Informatique

Spécialité :
Science et Technologie Logiciel

TME

Messagerie en ligne pour les trains de la SNCF

Encadré par

- R.Demangeon

Réalisé par

- Tabellout Yanis
- Tabellout Salim

le : **24/04/2024**

TABLE DES MATIÈRES

TABLE DES FIGURES

CHAPITRE 1

CHOIX GÉNÉRAUX

Dans ce projet, nous avons implémenté et deployé [1] un système de messagerie en ligne pour le réseaux de la **SNCF**. La messagerie peut être pour un groupe de personnes au sein du même train mais aussi pour une messagerie entre deux personnes (private chat). Nous avons prêté de l'attention à bien sécuriser nos API grâce à un système d'authentification grâce au JWT token.

1. Schéma global, langage et framework

Dans ce projet on a essayé d'avoir une variété dans les langages de programmation nous permettant ainsi de mieux comprendre comment la communication entre client et serveur.

- **Frontend** : Nous avons utilisé React avec son framework **NextJs** et pas mal des librairies de UI comme **TailwindCsS, ShadcnUi** mais aussi des librairies nous permettant de gérer le contexte dans notre siteweb comme **Zustand**.
- **Backend** : Nous avons opté une architecture en **orienté ressource donc une architecture rest** pour le backend. Nous avons choisi comme imposé **Golang** avec le package net/http pour nous permettre de créer les handlers pour les endpoints de notre API. Nous avons utilisé certaines bibliothèques notamment **cron** pour nous permettre de lancer periodiquement les appels vers l'API de la SNCF. Comme nous avons opté pour une authentification **Oauth**, nous avons utilisé la bibliothèque **JWT et cors**. Pour les handlers nous nous sommes inspiré de [2] pour créer les handlers de golang.
- **Base de données** : Nous avons choisi d'utiliser la base de données **PostgreSql** avec **l'ORM Prisma** pour faciliter nos requêtes vers la BDD. La base données a été hosté sur **Supabase**.

- **Websockets** : comme imposé, nous nous sommes inspiré de [3] et de [4] pour gérer les sockets avec le principe de Hub ou bien Room [1]
- **API SNCF** : l'API de la SNCF est une api assez populaire et très bien documenté et bien structuré. Cette api rend en temps réelle toutes les informations sur les lignes de la SNCF.
- **Cloudinary** : Nous avons utilisé le service coud pour permettre une utilisation très efficace pour sauvegarder les images des utilisateurs

La figure suivante modélise notre système au complet :

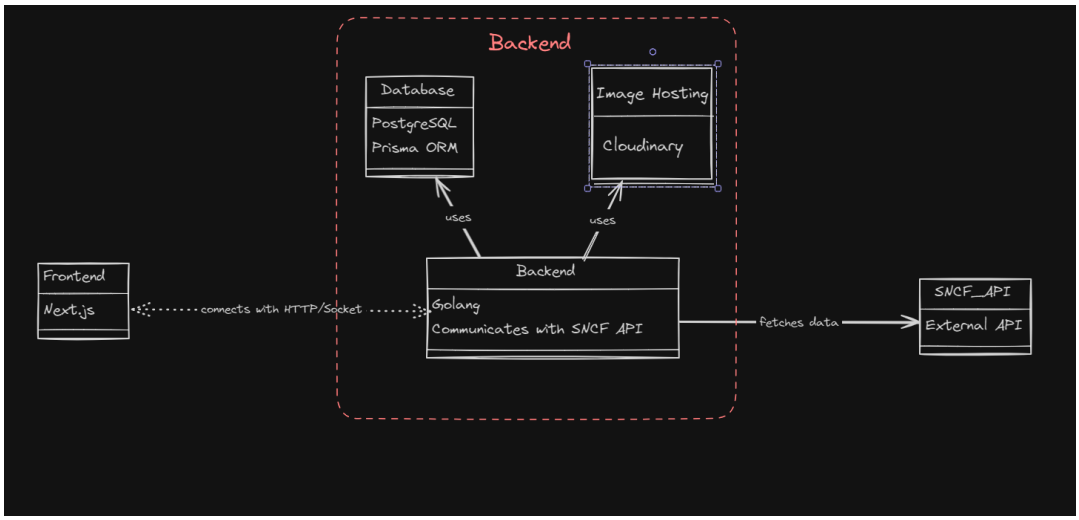


FIGURE 1.1 – Schéma global

2. Structure de fichiers

- **Backend** : contient tout les fichiers relié avec le backend, notamment le **main.go** le point d'entrée du backend
- **Frontend** : contient tout les fichiers relié avec le frontend.
- **Fichiers shell** : Nous avons trois fichiers shell, le **setup** permet de configurer le backend et le frontend et le **runserver et runclient** permettent de lancer le serveur et le client respectivement.

3. Déploiement

Nous avons déployé notre application web en trois partie notamment :

- **Base de données :** Nous avons hosté la base de données sur **Supabase** qui est un SGBD **postgres** très complet. Nous n'avons pas utilisé les fonctionnalités proposés car nous nous contentons de notre propre authentification

1. Room : concept de Hub mais dans socket.io

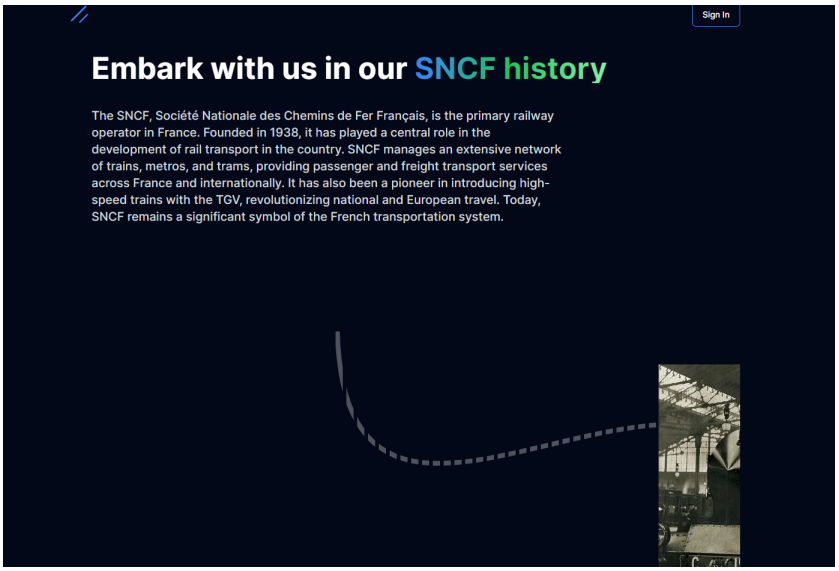
- **Serveur et Frontend :** Nous avons utilisé la plateforme **railway** pour nous permettre de déployer le serveur et le client. Nous avons choisi railway pour sa simplicité mais surtout car contrairement aux plateformes serverless comme vercel, railway permet donc d'établir et de maintenir une connexion avec websockets. Nous avons utilisé aussi Docker pour configurer le déploiement sur railway surtout pour le serveur.

4. Installation et lancement

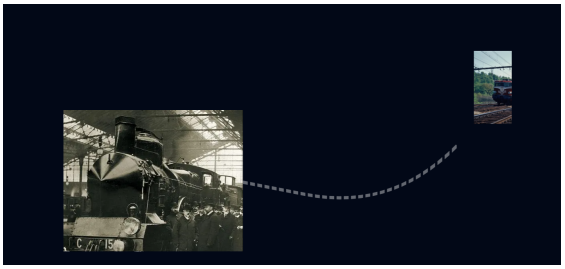
Notre application **est déjà deployé** [1], cependant si vous souhaitez le lancer localement, on suppose que vous avez déjà nodejs et typescript préalablement installé, ainsi que golang. Dans un terminal veuillez lancer le fichier **setup.sh** et lancer le fichier **run-server.sh**, et ouvrir un nouveau terminal pour lancer le shell pour lancer le client.

5. Utilisation

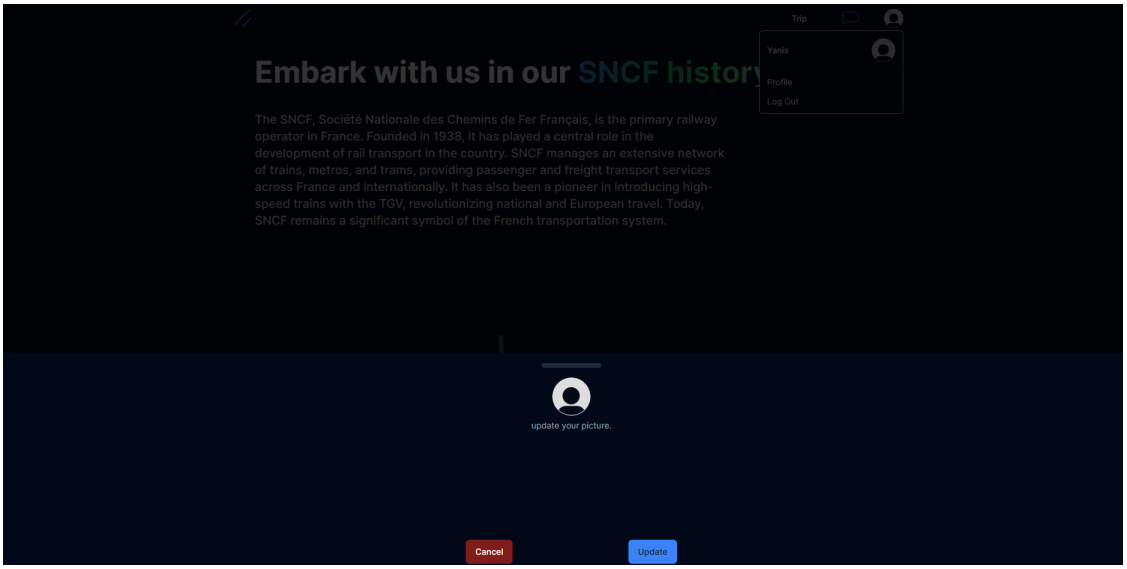
5.1. Page principale



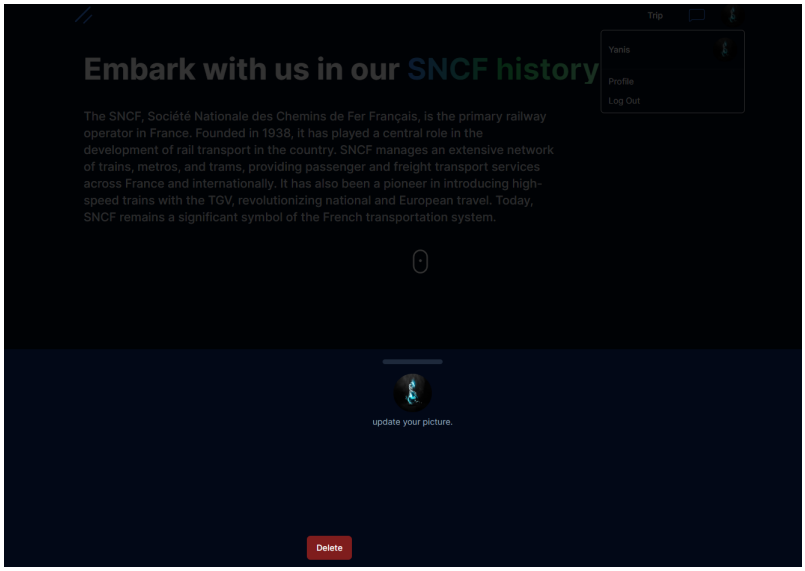
(a) Page Principale



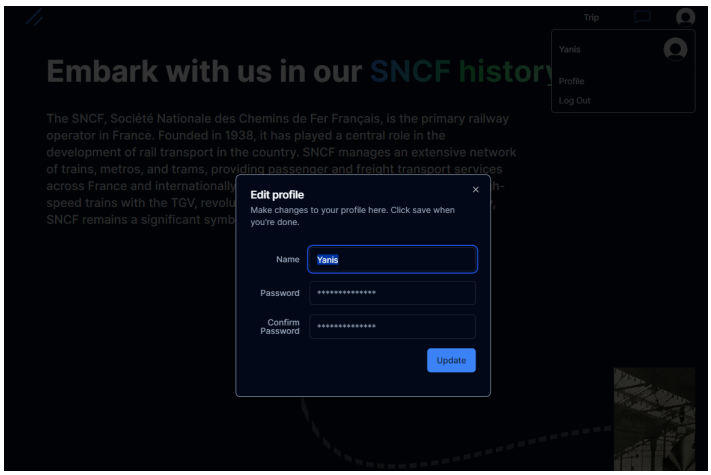
(b) Animation



(c) Mis à jour de la photo du compte



(d) Photo de profile mis à jours



(e) Mis à jour des détails du profile

FIGURE 1.2 – Page principale

5.2. Authentification

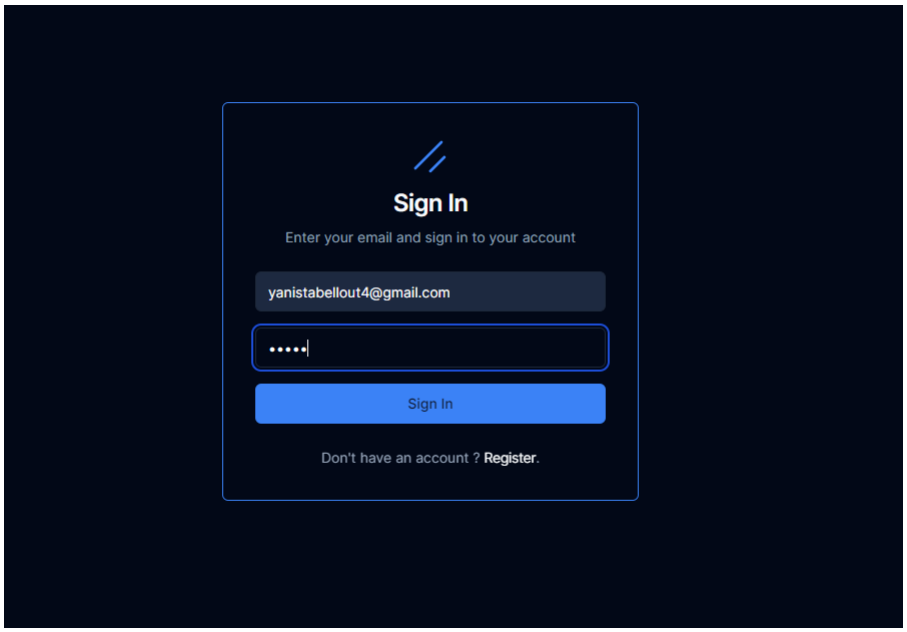
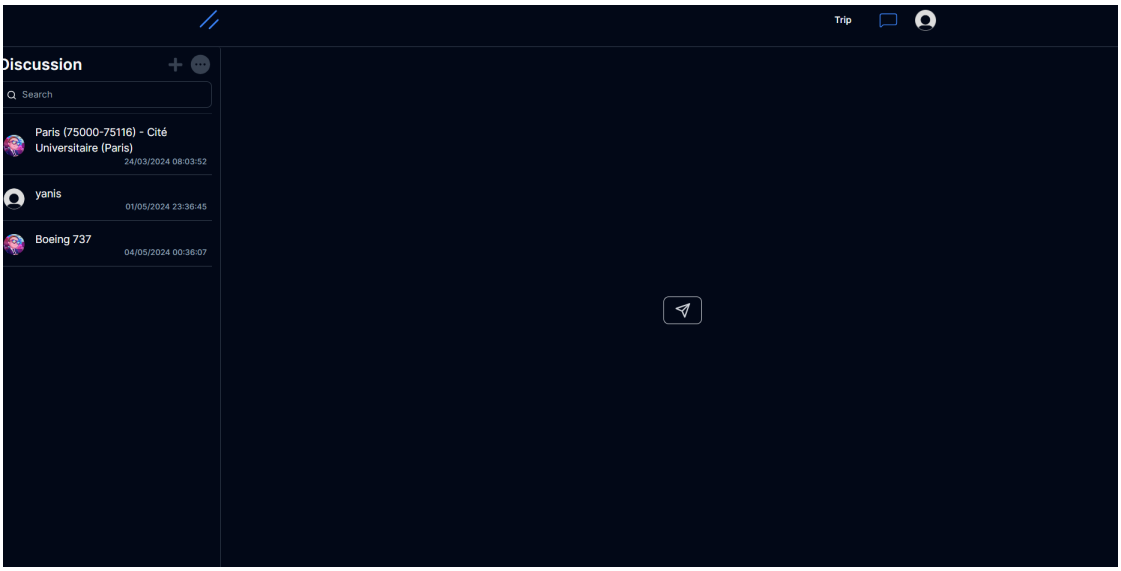


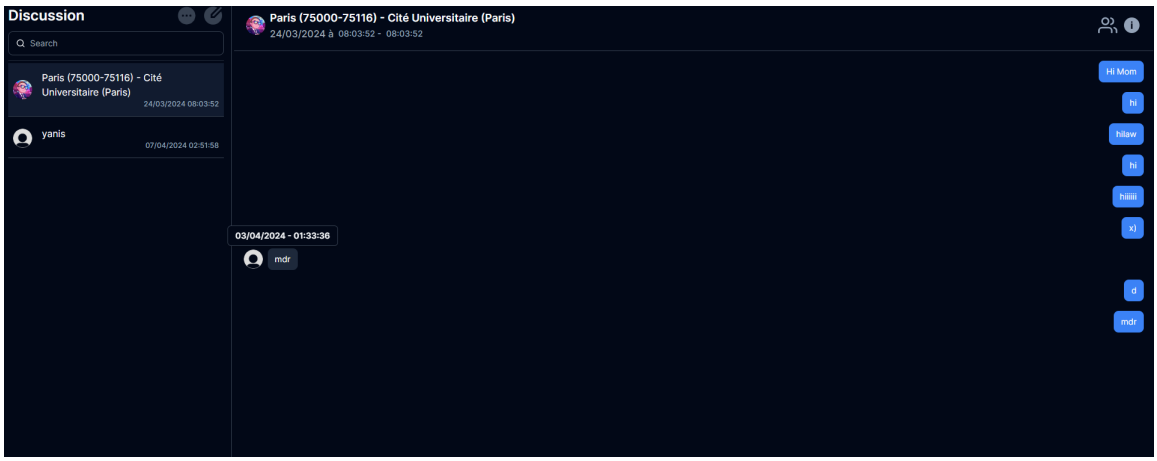
FIGURE 1.3 – Page Authentification

5.3. Chat

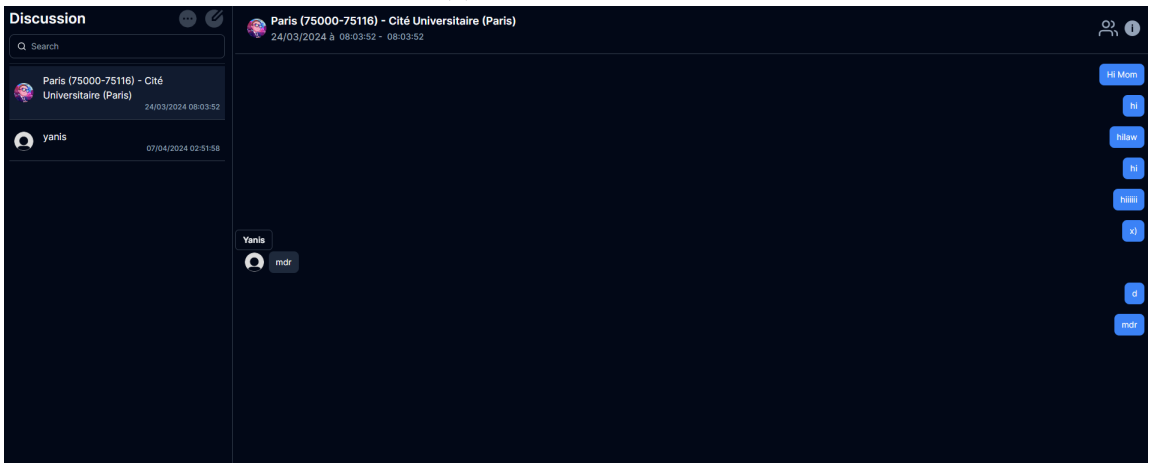


(a) Chat

(b) Chat principale



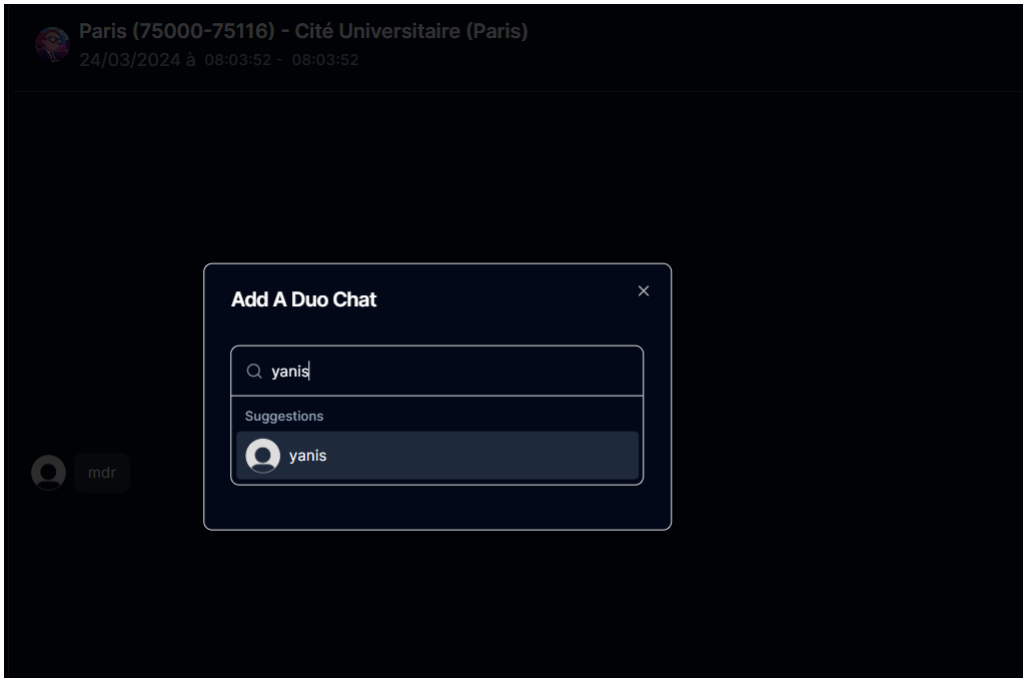
(c) Heure de message



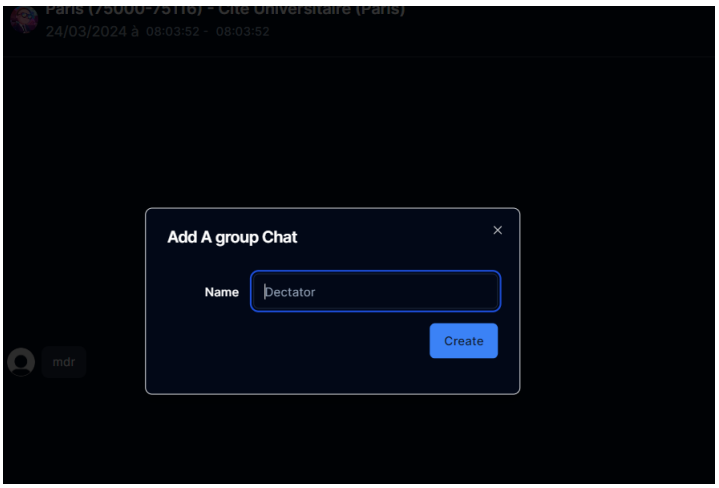
(d) Appuyer sur l'image de l'utilisateur pour discuter avec une personne en privé

(e) Message privé

(f) Créer une nouvelle conversation

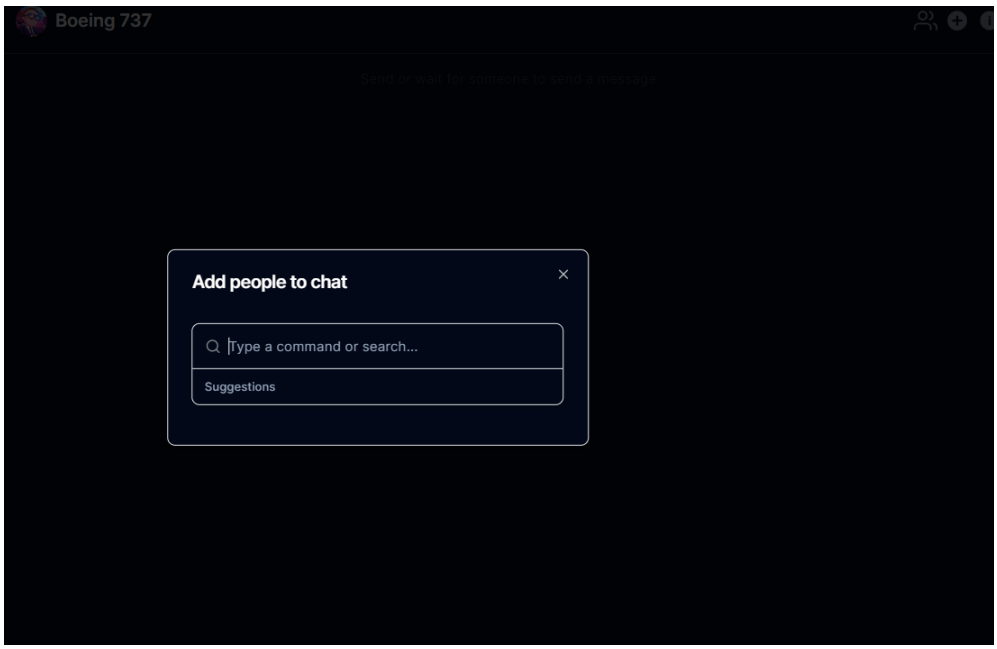


(g) Créer une conversation privée



(h) Créer un groupe

(i) Appuyer sur le button add dans le chat header pour ajouter des personnes



(j) Chercher et cliquez sur la personne

FIGURE 1.4 – Page principale

5.4. Programme journaliers

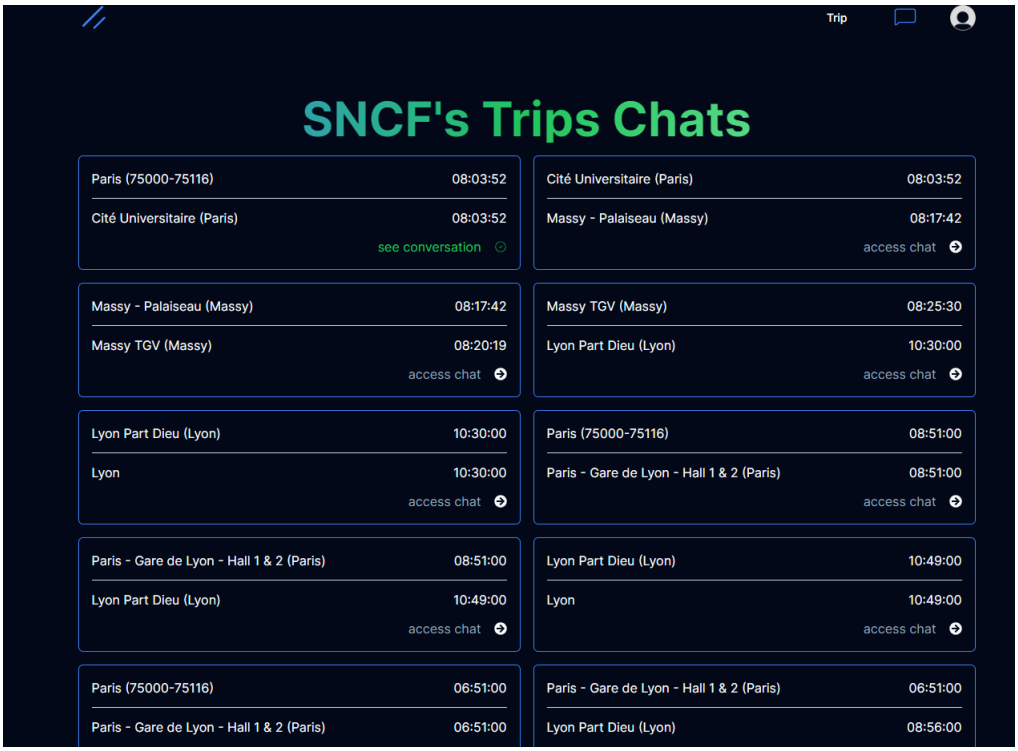


FIGURE 1.5 – Programme journaliers

CHAPITRE 2

FONCATIONNALITÉ ET TECHNICALITÉ

1. Foncationnalité : Use case

Notre application contient pas mal de fonctionnalité :

1.1. Authentification

Un utilisateur peut créer son compte avec son email, nom et peut accéder au site qu'en s'authentifiant.

1.2. Chercher des voyages

L'utilisateur peut trouver les voyages du jours pour accéder à une de messageries disponibles.

1.3. Messagerie

L'utilisateur peut rejoindre les messageries de chaque voyage, mais aussi de contacter un utilisateur en message privé, mais aussi de créer des groupes entre personnes.

1.4. Messagerie personnalisé

Un utilisateur peut créer un groupe et ajouter des personnes, il peut aussi communiquer en privé avec une personne en particulier.

1.5. Mis à jour du profil

L'utilisateur mettre à jour les informations de son profile, comme le nom prénom, mot de passe et sa photo.

1.6. Mis à jours de la base de données

Le serveur de manière automatisé chaque jour à minuit, fait une requête à l'API de la SNCF pour récupérer les données par une API call.

2. Technicalité : Backend

2.1. Schéma de la base données et mis à jours des données

Voici le schéma de notre base de données :

- **User** (id,name,$\cdots$)
- **Trip** (id,from,to,departure_time,arrival_time,**chat_id***,)
- **Chat** (id,name,photo,type [1],$\cdots$)
- **Message** (id,content,chat_id,user_id,$\cdots$)

Nous avons utilisé l'outil **cron** qui permet de lancer périodiquement ou à un moment particulier la mis à jours des données, dans notre cas nous avons pris le choix de mettre à jours les données Chaque soir à minuit. Nous avons ensuite mis en place une fonction qui se charge de récupérer les données de l'API de la SNCF et de mettre à jours les données directement dans la base de données.

2.2. Middleware

Nous avons définit des middlewares qui permettent de factoriser un peu de code notamment pour les handlers, on peut citer le **JsonContentMiddleware** qui permet donc de spécifier que le serveur rend du Json. Le **AllowedMethodsMiddleware** permet de définir quelles sont les méthodes (*Get, Post* $\cdots$) autorisé dans un certain endpoint.

2.3. Authentification

Un token est généré à chaque fois qu'un utilisateur se connecte (sign in), se token sera utilisé pour accéder à des endpoint de l'api protégé grâce à un middleware qui vérifie si le token envoyé dans le header de la requête http a été envoyé puis vérifie sa validité.

2.4. Websocket

Nous avons impléménté comme indiqué dans [3] et [4] un endpoint sur le serveur pour les websocket. Ce endpoint reçoit des requêtes HTTP qui sont "upgrader" à une connection

1. l'attribut type dans le chat est une énumération : *trip, duo, group*

temps réelle entre le client et le serveur tout en gardant l'authentification de la socket. Chaque socket qui se connecte au serveur est associé à un type particulier qu'on appelle socket client.

2.5. Messagerie

Nous avons pris l'approche avec des hubs [4] qui permet de gérer les chats. Chaque hub gère un seul chat, et le hub est représenté par un thread auquelles un ou plusieurs socket client s'abonner (register et unregister). Une socket qui se connecte à un chat (register) pourra diffuser (broadcast) un message et le hub se chargera de diffuser le message pour les autres sockets clients. Le mécanisme de broadcast et d'envoie de message a été implémenté grâce aux canaux synchrones de go ou chaque socket client est associé à un canal ou elle peut recevoir des messages avant de les envoyer au client web (le frontend) permettant ainsi une meilleure réutilisablité du code.

3. Frontend

Pour la partie frontend nous avons utilisé le langage TypeScript, React ainsi que son framework **NextJs**. Pour les requêtes entre client et le serveur nous avons utilisé **Axios** couplé à une librairie qui le **TanStack query**[**5**] avec le **React Query** ce qui nous permet de simplifier le caching, error handling $\cdots$ des requêtes Asynchrone. Pour la partie UI, nous avons utilisé **TailwindCss** ainsi que **Shadcn UI** [6]. TypeScript implique qu'on devrait avoir une application au minimum typesafe, donc nous avons défini les types qu'on devrait intéragir. Pour le state management dans notre application, nous avons utilisé **Zustand**[7] qui nécessite beaucoup moins de configuration contrairement à redux ou react contexte et qui est beaucoup plus intuitif. NextJs suit une certaine structure de fichiers particulière du moins pour les pages ou le path dans le file structure consitue l'url dans le navigateur (e.g *:/app/chat/page.tsx est représenté par l'url : http ://... :3000/chat*). Le principe de react est de séparer les composants pour qu'ils soient les plus réutilisables possible, c'est pour celà que nous avons défini un **folder components** contenant tout les composants réutilisables. React permet aussi d'utiliser des **hooks** qui est du code qui sera lancer quand un composant est dit **mounted ou unmounted**. Pour la connection avec les websocket du serveur, nous nous sommes inspiré de [8] pour implémenter une classe de socket mais aussi de [9] pour nous permettre d'envoyer le token d'authentification au serveur.

BIBLIOGRAPHIE

[1] https://pc3r-production-fba8.up.railway.app.

[2] https://www.digitalocean.com/community/tutorials/how-to-make-http-requests-in-go#making-a-get-request.

[3] https://github.com/snassr/blog-goreactsockets/blob/master/backend/main.go.

[4] https://github.com/gorilla/websocket/blob/main/examples/chat/hub.go.

[5] https://tanstack.com/query/latest.

[6] https://ui.shadcn.com/.

[7] https://zustand-demo.pmnd.rs/.

[8] https://medium.com/@snassr/websockets-react-go-be6330ad547d.

[9] https://stackoverflow.com/questions/4361173/http-headers-in-websockets-client-api.