Faculté des Sciences et Ingénierie

Parcours Science et Technologie du Logiciel (STL)

Programmation Concurrente Réactive, Répartie et Réticulaire (PC3R) : Site de chat esport

Étudiants :
Hichem BOUZOURINE, N°21319982,
Rajith RAVINDRAN, N°21304977,

Encadrants :
Romain Demangeon, Romain.Demangeon@lip6.fr

26 mai 2024

Établissement / Formation : Sorbonne Université, Master 1 STL
Année de Formation : Semestre 2, 2023-2024
Mots clés : Caneaux synchrones, messagerie, react, golang, http, client-server

Table des matières

Table des figures

1 Introduction

L'application visée est un site dédié aux résultats d'eSport professionnel, offrant un accès en temps réel et historique aux résultats de diverses compétitions d'eSport à travers le monde.

2 Schéma global

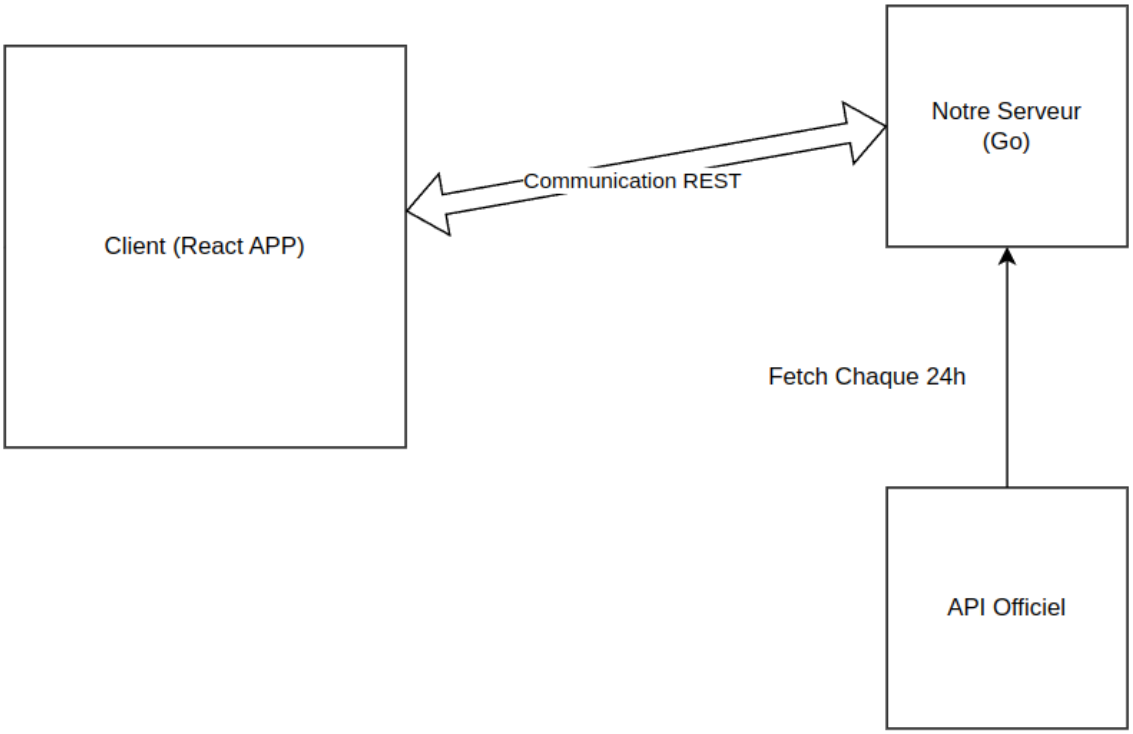

FIGURE 1 – `Schéma global de l'application`

3 Installation du projet

```
git clone https://stl.algo-prog.info/21319982/projet-pc3r
cd projet-pc3r
bash run.sh
```

4 Déploiement en ligne

Nous avons choisi le service *Render & Railway* qui donnent la possibilité de déployer un site web dynamique gratuitement, nous nous excusons si le service est un peu lent.

4.1 Étapes de déploiement

— **Backend** : Nous avons référencé notre *github* pour cloner une image de notre code, puis nous avons ajouté les **variables d'environnement** essentielles pour lancer le serveur, comme il y a un fort support avec le langage **Go** il suffit d'ajouter un Dockerfile puis le backend se lance.

— **Frontend** : Pour lancer notre application React après avoir ajouté les variables d'environnement, nous devrons utiliser le package *http-server* pour tourner l'application indéfiniment.

4.2 Lien de déploiement

Nous tenons à vous dire que nous avons équipé notre déploiement d'un service de logs. Ainsi, lorsque l'application tombe en panne, nous savons pourquoi et il nous est facile de la maintenir.

Voici le lien : **https://valorant-chat-production-70db.up.railway.app/**

Voici le deuxième lien en cas de panne dans le premier site :
https://valorant-chat-1.onrender.com/

5 L'API Source

L'API Web choisie pour ce projet propose des données sur les compétitions d'eSport Valorant, incluant les scores en direct, les calendriers des matchs, les informations sur les équipes et le résultat des matchs. Notre application utilisera principalement les données de scores en direct et historiques pour informer les utilisateurs des résultats des matchs.

Voici l'API en question : **https://github.com/axsddlr/vlrggapi**

6 Use Cases

Cas d'un nouveau utilisateur qui veut rejoindre la plate-forme :
- L'utilisateur clique sur "**Register**" pour créer un nouveau compte.
- L'utilisateur entre son Nom, Email et Mot de passe.
- L'tilisateur clique sur le botton "**Register**".
- Le serveur confirme que l'utilisateur n'existe pas et crée un nouveau user dans la base de données.
- L'utilisateur clique sur "**Go to Matchs**" pour consulter les matchs.
- L'utilisateur choisit un match et le chat s'affiche.
- Le serveur crée une socket de communication
- L'utilisateur envoie un message.
- Le serveur sauvegarde le message et notifie les autres sockets connectés.

7 Conception de la base de données

Le fichier qui permet de définir la base de données et trouvé dans l'emplacement suivant **'/backend/prisma/schema.prisma'**.

Les tables sont les suivantes :
- **MatchResult**(**id**, team1, team2, score1, score2, flag1, flag2, time_completed, round_info, tournament_name, match_page, tournament_icon, *chat_id, createdAt, updatedAt).
- **User**(**id**, name, email, password, photo, createdAt, updatedAt).
- **Chat**(**id**, date, name, photo, *match).
- **Message**(**id**, content, chat_id*, user_id*, createdAt, updatedAt).
- **NewsLetter**(**id**, email, createdAt, updatedAt).

8 Fonctionnalités

— Regarder les résultats de chaque match récent

— Chercher les matchs d'une équipe

— Interagir avec les autres utilisateurs dans un **chat**

— Voir la carte d'un utilisateur

— Inscrire dans la **Newsletter**

9 Backend

9.1 Fonctionnement global du backend

— On récupère les résultats des matchs depuis l'**API** de base chaque **24h** pour rafraichir notre base de données grâce à une librairie open-source qui s'appelle *gocron*[3], il s'agit d'une fonction de scheduler qui s'applique chaque **X** temps.
— Pour les matchs, on crée une fonction qui récupère les matchs depuis la BDD, et on appelle cette fonction dans le *http.ServeMux* qui elle le responsable d'attacher pour chaque *API URL* une fonction.

9.2 Structure et architecture des fichiers

Nous avons structurés notre projet dans 2 dossiers pour séparer le code du serveur de celui du client,

9.2.1 Backend

— Conception base de données : dans '/**prisma**' on retrouve l'ORM qui permet de gérer la base de données.
— Recupération des données : dans le sous dossier '/**matchesResult**' nous définissons le code qui appelle l'API externe et récupère les matchs récents et dans le sous dossier '/**database**' nous transmettons les matchs dans la base de données.
— Divers services : dans le '/**services**' on retrouve les fichers qui définissent les fonctions de l'authentification, la messagerie, la newsletter.

9.2.2 Frontend

— L'entrée principale : dans '/**src**' on retrouve *main.tsx* qui est le point où l'application se lance.
— Pages : dans le sous dossier '/**src**/**pages**' nous définissons 3 fichiers qui représentent les 3 pages disponibles dans notre application.
— composants réutilisables : pour respecter le pattern du React, dans le '/**src**/**components**' on retrouve les fichers qui définissent les petits composants qu'on utilise souvent.
— Divers services : dans le '/**src**/**services**' on retrouve le code qui permet de faire des requêtes au serveur.

9.3 Asynchronie

Nous avons implémenté le principe d'asynchronie vu en cours à plusieurs endroits de notre code, par exemple nous avons choisi de lancer le serveur dans un **Thread** séparé pour maximiser la concurrence, le même principe est utilisé dans la fonction *Scheduler* qui s'exécute à chaque X fois.

9.4 Validation

Pour garantir que notre application est bien réalisée, nous avons choisi de valider les saisies d'authentification, par exemple pour l'email et le mot de passe nous validons chacun par rapport à l'expression régulière.

9.5 Handlers

Dans notre application, nous avons plusieurs handlers, brièvement voici la liste des APIs pour chaque Table. Pour chaque structure, on va avoir des requêtes *HTTP* de type **POST**/**GET**, avec le lien imbriqué pour une meilleur compréhension, et une liste de paramètres qui donne une réponse *HTTP* qui doit être soit la structure demandé ou un message d'erreur.

9.5.1 User

"**/api/auth/login**" :

```
POST        "/" : (email, password) => ({User, Token} || Error)
```

"**/api/auth/signup**" :

```
POST        "/" : (nom, email, password) => ({User, Token} || Error)
```

9.5.2 Chat

: "**/api/chat**" :

```
GET         "/:id" : () => (Chat || Error)
```

9.5.3 Matchs

: "**/api/matchesResult**" :

```
GET         "/:id" : () => (MatchResult[] || Error)
```

9.5.4 Newsletter

: "**/api/subscribeNewsletter**" :

```
GET         "/" : (email) => ("Inscription à la newsletter réussie" || Error)
```

9.6 Websockets

Les Websockets[2] sont l'outil qui nous permet de gérer efficacement la communication en temps réel. Nous avons défini la structure de notre socket **sans utiliser de bibliothèque externe**[6], nous avons donc un **Hub** qui représente une salle de discussion qui inclut les clients dans sa structure et **3 canaux de communication**, Diffusion, inscription et désinscription pour envoyer un message à tous les clients, ajouter ou supprimer un client. Nous garantissons ces fonctionnalités en utilisant la syntaxe de **select** vue en cours pour nous assurer de pouvoir écouter plusieurs événements en même temps.

Nous avons défini les 3 fonctions de base, *s'abonner à une discussion, envoyer un message* et *se désinscrire d'une discussion*, et nous les attachons à des routeurs de type websockets.

L'implémentation complète de *nos propres websockets* se trouve dans le dossier '/**webSocket**' dans le backend.

10 Frontend

Notre application React TS est **multi-pages**, le design est réalisé avec soin pour que l'application soit **responsive**, c'est à dire que le design soit adapté aux utilisateurs sur **grands écrans** et également aux utilisateurs **mobiles**.

10.1 Expérience mobile

De nos jours, la navigation mobile est largement utilisée, c'est pourquoi nous avons investi pour créer une interface utilisateur utilisable pour les utilisateurs mobiles

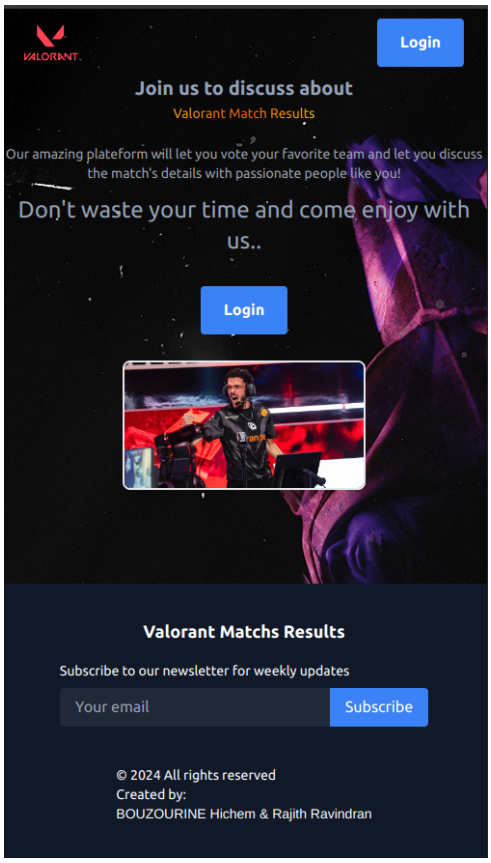

FIGURE 2 – Le design de la page d'accueil dans le mobile

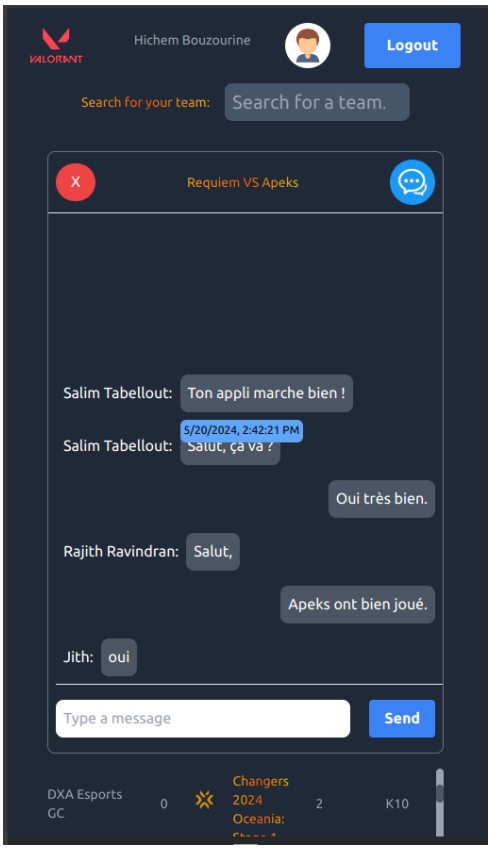

FIGURE 3 – Le design de la page du chat dans le mobile

10.2 Expérience sur les grands écrans

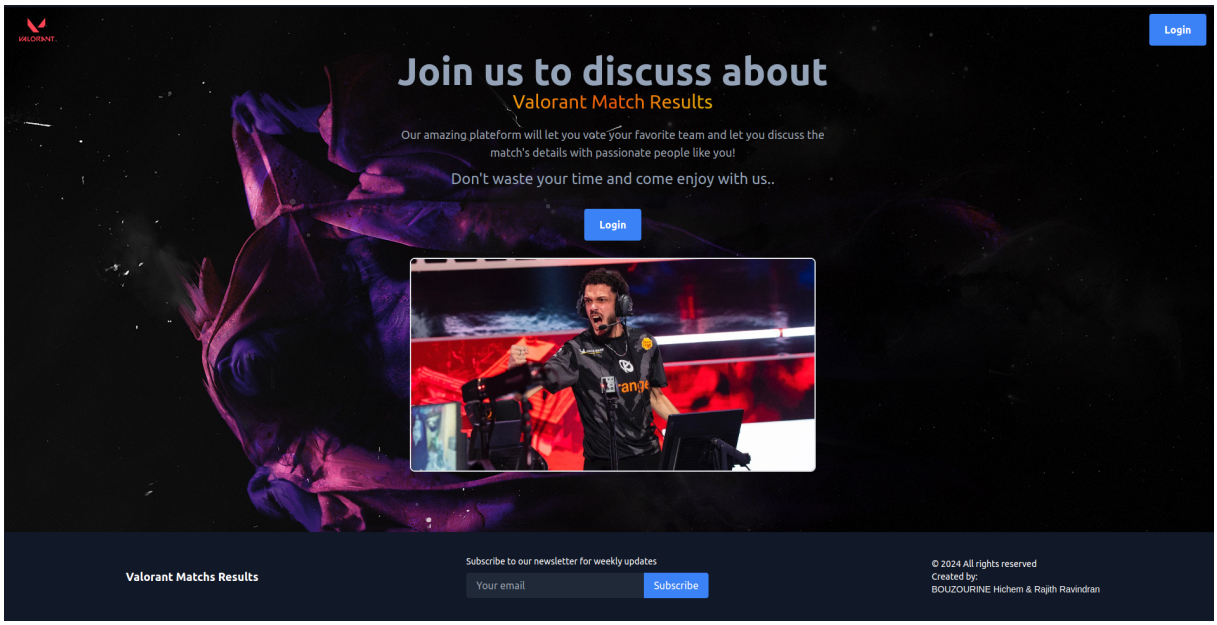

FIGURE 4 – Le design de la page d'accueil dans le desktop

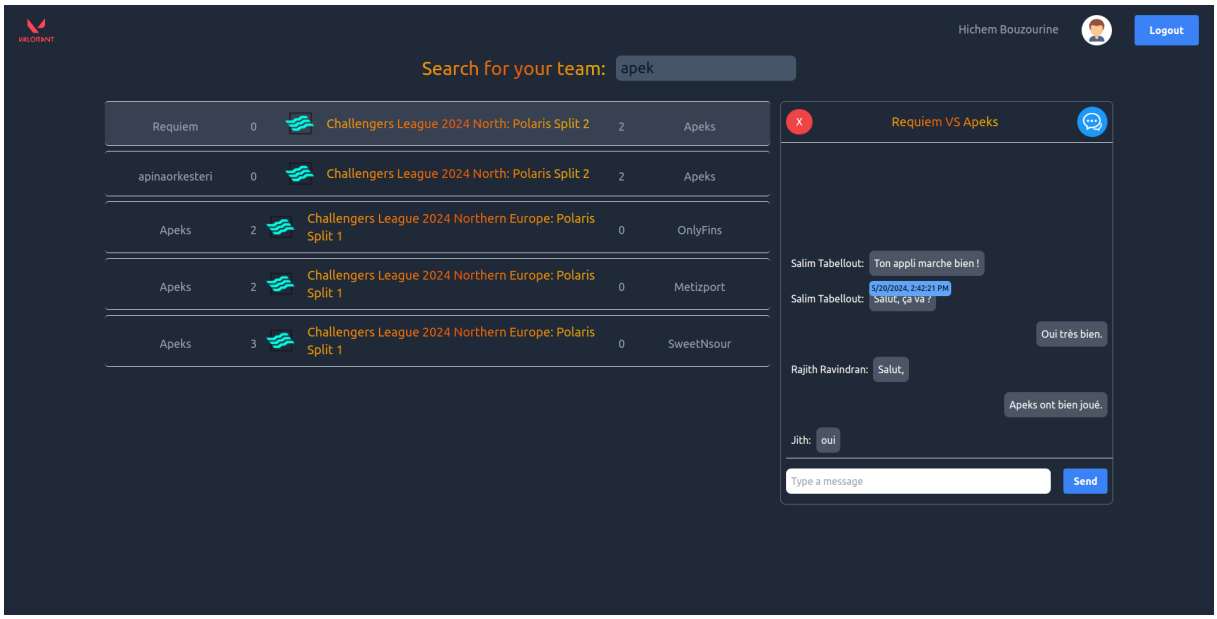

FIGURE 5 – Le design de la page du chat dans le desktop

11 Perspectives

L'idée nous intéresse beaucoup, malheureusement si nous n'avions pas de critère de temps, nous ajouterions un système de notation des messages comme **Reddit**.

12 Ressources

— **VSC Live Share :**
L'emploi de l'extension VSC Live Share [5] s'est révélé d'une grande utilité lors de nos séances de travail en TME, grandement facilitant la collaboration et le partage d'informations entre les membres de l'équipe.

— **ChatGPT pour l'Optimisation de Code en Go :**
Nous avons utilisé ChatGPT [4] principalement pour optimiser et raffiner notre code Go. Cela a entraîné une meilleure compréhension des pratiques de codage efficaces, réduisant la longueur des scripts et améliorant leur lisibilité tout en éliminant les redondances. Ce processus d'apprentissage a été crucial pour améliorer la qualité générale et la maintenabilité de notre code.

— **Collaboration avec nos camarades de classe :**
Nos camarades, Yanis et Salim Tabellout, nous ont apporté une aide précieuse lors des séances de TME. Leur soutien a été instrumental pour surmonter divers défis techniques. Des annotations dans notre code marquent les points spécifiques où leur assistance a été bénéfique, facilitant la révision et l'apprentissage collaboratif.

Références

[1] *«Go Documentation»*. `https://go.dev/doc/`.

[2] *«Websockets»*. Quinn Rivenwell. `https://github.com/nhooyr/websocket`.

[3] *«A Golang Job Scheduling Package»*. gocron. `github.com/jasonlvhit/gocron`.

[4] *«ChatGPT»*. OpenAI. `https://chat.openai.com`.

[5] *«Use Microsoft Live Share to Collaborate with Visual Studio Code.»*. Microsoft. `https://code.visualstudio.com/learn/collaboration/live-share.`.

[6] *«How To Build A Chat And Data Feed With WebSockets In Golang ? .»*. Anthony GG. `https://www.youtube.com/watch?v=JuUAEYLkGbM&t=431s`.