Rendu du TP - Realtime Elo Ranker

Table des Matières

Introduction

Ce document présente mon travail réalisé sur le Tp Realtime Elo Ranker. Le projet consiste en une application web permettant de simuler des matchs entre des joueurs et de calculer et afficher leur classement Elo en temps réel. Le projet est composé de plusieurs applications et librairies, notamment le serveur `realtime-elo-ranker-server`, le client `realtime-elo-ranker-client`, et un simulateur de matchs `realtime-elo-ranker-simulator`.

lien du dépot git

https://github.com/KillianOuzet/realtime-elo-ranker-master

Serveur - `realtime-elo-ranker-server`

L'application serveur a été développée en utilisant NestJS, un framework Node.js pour construire des applications serveur. Le serveur gère le calcul des résultats de matchs et le classement des joueurs, et expose une API Web pour fournir les fonctionnalités métiers.

Modules

Module Player

Le module Player gère les opérations liées aux joueurs, telles que la création, la récupération et la mise à jour des joueurs.

- **Routes :**
  - `POST /api/player`: Crée un nouveau joueur. Cette route prend en entrée un objet `CreatePlayerDto` contenant les informations du joueur à créer (par exemple, `id` et `baseRank`). Le service `PlayerService` est utilisé pour ajouter le joueur à la base de données.
  - `GET /api/player`: Récupère la liste de tous les joueurs. Cette route utilise le service `PlayerService` pour obtenir tous les joueurs stockés.
  - `GET /api/player/:id`: Récupère un joueur par son identifiant. Cette route utilise le service `PlayerService` pour trouver un joueur spécifique en fonction de son `id`.

Le service `PlayerService` encapsule la logique métier pour les opérations sur les joueurs. Il interagit avec le repository des joueurs pour effectuer les opérations CRUD (Create, Read, Update).

Module Match

Le module Match gère les opérations liées aux matchs, telles que la publication des résultats de matchs.

- **Routes :**
  - `POST /api/match`: Publie les résultats d'un match pour mettre à jour le classement des joueurs. Cette route prend en entrée un objet `PublishMatchDto` contenant les informations du match (par exemple, les identifiants des joueurs et le résultat du match). Le service `MatchService` est utilisé pour traiter le résultat du match et mettre à jour les classements des joueurs en conséquence.

Le service `MatchService` encapsule la logique métier pour les opérations sur les matchs. Il calcule les nouveaux classements des joueurs en utilisant la formule Elo et met à jour les joueurs dans la base de données en faisant appel au `PlayerService`.

Module Ranking

Le module Ranking gère les opérations liées au classement des joueurs.

- **Routes :**
  - `GET /api/ranking`: Récupère le classement actuel des joueurs. Cette route utilise le service `RankingService` pour obtenir le classement actuel.
  - `GET /api/ranking/events`: Écoute les événements de mise à jour du classement en temps réel via Server-Sent Events (SSE). Cette route permet aux clients de recevoir des mises à jour en temps réel lorsque le classement des joueurs change.

Le service `RankingService` encapsule la logique métier pour les opérations sur le classement. Il interagit avec le repository des joueurs pour obtenir le classement actuel et utilise des événements pour notifier les clients des mises à jour en temps réel. De plus, il garde en liste les classements, comme cela pas besoin de faire des requêtes redondantes vers la bd. En effet, à chaque modification le `PlayerService` va informer le `RankingService` et celui-ci va renvoyer les évènements. À l'initialisation du serveur, le PlayerService envoie tout le classement au RankingService pour remplir sa liste.

Fonctionnement

Chaque module est responsable de ses propres opérations et utilise des services pour encapsuler la logique métier. Les contrôleurs gèrent les requêtes HTTP et délèguent les opérations aux services. Les services interagissent avec les entités et les repositories pour effectuer les opérations sur les données.

- **PlayerController** : Gère les requêtes HTTP pour les opérations sur les joueurs et délègue les opérations au `PlayerService`.
- **MatchController** : Gère les requêtes HTTP pour les opérations sur les matchs et délègue les opérations au `MatchService`.
- **RankingController** : Gère les requêtes HTTP pour les opérations sur le classement et délègue les opérations au `RankingService`.

Les repositories sont responsables de l'accès aux données et de la persistance des entités.

Le serveur utilise également des événements pour notifier le client des mises à jour en temps réel. Par exemple, lorsque le classement des joueurs est mis à jour après un match, un événement est émis pour notifier les clients connectés via SSE.

En résumé, l'application serveur est structurée en modules, chacun responsable d'un aspect spécifique des fonctionnalités métiers. Les contrôleurs gèrent les requêtes HTTP, les services encapsulent la logique métier, et les repositories gèrent l'accès aux données. Je pense et espère avoir réussi à réaliser une architecture correcte

Simulateur de Match - `realtime-elo-ranker-simulator`

Le simulateur de match est un script TypeScript qui récupère les joueurs existants et lance des matchs de manière aléatoire en appelant les routes du serveur.

Fonctionnement

- Le script récupère la liste des joueurs existants depuis le serveur (si aucune données existantes, alors utilise une liste générique de joueurs en les postant aux serveurs).
- Il sélectionne deux joueurs au hasard et simule un match entre eux.
- Les résultats du match sont envoyés au serveur via la route `POST /api/match`.
- Le script répète ce processus pour simuler plusieurs matchs.

Tests

Tests Unitaires

Des tests unitaires ont été ajoutés pour vérifier le bon fonctionnement des différentes fonctions et méthodes des services et des contrôleurs. Les tests unitaires utilisent la librairie Jest.

Tests d'Intégration

Des tests d'intégration ont été ajoutés pour vérifier les interactions entre les différents modules et services. Ces tests permettent de s'assurer que les modules fonctionnent correctement ensemble.

Tests End-to-End (E2E)

Des tests E2E ont été ajoutés pour vérifier le bon fonctionnement de l'application dans son ensemble. Ces tests simulent des scénarios utilisateur complets et vérifient que l'application répond correctement aux requêtes.

Commandes de test

Pour lancer les tests il y a 3 commandes utilisables. Tout d'abord allez dans le dossier `apps/realtime-elo-ranker-server`. Une fois à la racine de ce dossier vous pouvez lancer les tests :

Pour lancer les tests unitaire et d'intégrations

```
pnpm test
```

Pour voir la couverture de tests il faut ajouter :cov

```
pnpm test:cov
```

Pour lancer les tests End to End :

```
pnpm test:e2e
```

Lancer l'Application

Prérequis

- Installer les dépendances des différents packages avec `pnpm` :

```
pnpm install
```

Lancement

Tout d'abord, chaque commande est à lancer simplement à la racine du projet, pas besoin d'aller dans chaque application.

Ensuite il faut suivre un ordre précis, toujours lancer le serveur en premier et le client en dernier ! Donc si vous voulez lancer le swagger, faite le avant le client.

Lancer le Serveur

Pour lancer le serveur, exécutez la commande suivante :

```
pnpm run apps:server:dev
```

Lancer le Swagger

Pour lancer le Swagger (non requis pour le bon fonctionnement), exécutez la commande suivante :

```
pnpm run docs:swagger:start
```

Lancer le Client

Ne pas oublier de générer les libs pour le client si ça n'a pas été fait au préalable :

```
pnpm run libs:ui:build
```

Pour lancer le client, exécutez la commande suivante :

```
pnpm run apps:client:dev
```

Lancer le script de simulation

Pour lancer le script de simulation de match, exécutez la commande suivante :

```
pnpm simulate:matches
```

Celui-ci va tourner indéfiniment tant que vous ne l'arrêtez pas.