Présentation du sujet et mise en place

Bienvenue pour cette démonstration du langage Rust et du Framework Axum. Ce projet a pour but de vous faire découvrir Rust et Axum à travers différentes étapes.

Pour ce projet, nous utiliserons une base de données Postgresql et nous allons apprendre à la mettre en place avec Docker dès maintenant. Dans un premier temps, vérifiez qu'un fichier **.env** existe à la racine et qu'il contient les informations suivantes :

```
DATABASE_URL=postgres://postgres:password@localhost/postgres
POSTGRES_PASSWORD=password
POSTGRES_USER=postgres
POSTGRES_DB=postgres
```

Descriptif de la base de données

Nous allons maintenant voir ce que contient notre base de données. Rendez-vous dans le fichier init.sql", au répertoire **db_scripts**. Après une courte observation, une table **tasks** existe et c'est celle-ci que nous allons plus tard utiliser à l'aide de sea-orm.

1.0 - Début du projet

Mise en place du container de base de données

Pour mettre en place la base de données, executez les commandes suivantes en tant qu'utilisateur root.

```
docker compose up
```

En cas de problème, vous pouvez arrêter le container et supprimer la base de données à l'aide des commandes.

```
docker compose down
rm -rf /data -f
```

1.1 - Génération des entités de la base de données avec sea-orm

Nous allons utiliser l'ORM "sea-orm" pour générer nos entités de base de données à partir de notre base de données mise en place. Installons la dépendance de la command-line de sea-orm.

```
cargo install sea-orm-cli
```

sea-orm va utiliser notre base Posgresql, assurez-vous que le container est bien lancé. Nous allons placer nos entités dans le répertoire "src/database".

```
sea-orm-cli generate entity -o src/database
```

1.2 - Cargo run

Maintenant, nous allons vérifier que le programme s'execute correctement.

```
cargo run
```

Cargo.toml

Le programme devrait installer les dépendances présentes dans le fichier Cargo.toml. Ce fichier permet d'importer des**crate** qui sont des librairies externes.

1.3 - Arborescence du projet

L'arborescence d'un projet Rust se décompose en plusieurs modules de fichiers et répertoires grâce à un fichier **lib.rs** qui a pour but de lister toutes les dépendances du projet.

Il sera possible de définir des modules de fichiers dans des sous-arborescences grâce à des fichiers **mod.rs** qui seront eux-mêmes listés dans **lib.rs**.

Dans notre cas, nous garderons les choses légères et importantes dans des fichiers, et nous utiliserons des fichiers**mod.rs** et les sous-arborescences pour les dépendances qui peuvent tendre à s'allourdir.

main.rs

Le fichier main.rs est le point d'entrée de notre programme. Il contient la fonction `main()` qui est appelée au lancement du programme. Nous allons tenter de garder ce fichier le plus lisible possible, c'est pourquoi nous allons déplacer la fonction qui instancie le server `run()` dans un autre fichier **server.rs**.

Petite note : `#[tokio::main]` permet d'indiquer à Rust que nous allons utiliser le runtime Tokio, qui est un runtime asynchrone.

server.rs

Le fichier server.rs contient la fonction `run()` qui permet d'instancier le server et lancer l'application. Mais avant cela, server.rs va permettre de définir les routes de notre application, les states ainsi que la connections en base de données.

router.rs

Le fichier router.rs contient la fonction `create_routes()` qui permet de définir les routes de notre application. C'est dans ce même routeur que nous allons définir les extensions, les middlewares, les states et les controllers.

Petite note : Les routes peuvent être organisés à l'aide de **nest** pour générer des plus petit routeurs qui regroupent plusieurs routes pour un même prefix "/tasks", sur lequel esra appliqué différentes méthodes HTTP associées à différents handlers. Les routes peuvent aussi être organisées de manière plus fines pour une gestion des autorisations par groupe d'utilisateurs.

répertoire routes

Le répertoire routes contient les fichiers qui définissent les handlers de nos routes.

répertoire models

Le répertoire models contient les fichiers qui définissent les structures de données.

répertoire database

Le répertoire database contient les fichiers qui définissent les entités de la base de données généré par sea-orm.

1.4 - Création d'une stucture de données

1.5 - Création d'endpoint : GET /

Handler

Result<>

1.6 - Création d'endpoint : GET /teapots

Status Code

1.7 - Création d'endpoint : POST /tasks

States

Insertion en base de données

1.8 - Validation de la structure de données de données

1.9 - Gestion des erreurs et "null case"

2.0 - Création d'endpoint : GET /tasks

2.1 - Création d'un filtre sur l'endpoint : GET /tasks

2.2 - Création d'endpoint : GET /tasks/{id}

2.3 - Création d'endpoint : PUT /tasks/{id}