Documentação “warren-api-test”

Instalação: O arquivo **.env** na raíz do projeto possui as instruçõe para a instalação.

Comandos via terminal:

O arquivo **package.json** possui os seguintes comandos para serem executados via **npm run:**

test: Executa os testes automatizados desenvolvidos;
start: Inicializa o projeto em ambiente de desenvolvimento na porta definida no arquivo “.env”;
build: Faz a transpiração do código typescript para javascript e armazena na pasta “dist”;
migrate: executa as migrations para a criação do banco de dados;
migrate:revert: reverte a última migration executada;
migrate:revert:all: reverte todas as migrations executadas.

Obs: Este projeto foi desenvolvido em um ambiente com sistema operacional Windows, mas não deve ter nenhum “problema” ao executar no ambiente linux.

Banco de dados:

O banco de dados utilizado durante o desenvolvimento foi o **MySQL**. Como o projeto utilizada um ORM para banco relacionais (sequelize) deve funcionar normalmente em outros bancos de dados como o Postgres por exemplo. Além do MySQL foi testado apenas com o **SQLite** pela facilidade de executar o projeto sem criar um banco de dados mais completo.
Para utilizar o SQLite basta definir a variável de ambiente **DB_DIALECT='sqlite'** no arquivo .env na raíz do projeto.

Front-end da aplicação:

Na raíz do projeto haverá uma pasta **“front”** que basicamente possui um arquivo HTML simples, pois a intenção do projeto é desenvolver uma **API back-end**, com importações externas do **Vue.js** e **Axios** (será necessário estar conectado a internet). Durante a execução do projeto, a rota raíz da API (“/”) apontará para esse arquivo HTML apenas para testabilidade das rotas da API.

Rotas da API:

Na raíz do projeto há uma arquivo **“warren-api.postman_collection.json”** que poderá ser importado para o **Postman** com todas as rotas disponíveis na API.

Arquitetura do projeto:

O projeto foi desenvolvido com Typescript para utilização de técnicas e conceitos que que são melhor aproveitados quando usados em uma linguagem tipada e orientada a objetos, como injeção de dependências e utilização de interfaces.

A arquitetura foi dividida em 3 camadas principais buscando uma melhor organização, manutenção e desacoplamento.

1. **Application**
   Essa camada é a porta de entrada para a API, nela você encontrará as **rotas** definidas e os respectivos **controllers**. Também possui **middlewares** de validação de dados de entrada, além de **serviços de aplicação**, que são assim chamados por apenas orquestraram os recurso oferecidos pela camada de domínio e conterem o mínimo de regras. Essa camada é fortemente ligada ao framework usado para declaração das rotas, framework Express neste projeto.

   Obs: Muitos desenvolvedores costumam deixar as **rotas** e **controllers** em uma camada definida como **interface**, seguindo a orientação de **arquitetura hexagonal**. Não foi usado o conceito dessa camada neste projeto.

2. **Domain**
   Essa camada representa o “core” da API. Nela deverá conter a maior parte das **regras de negócio** definidas e as **entidades** importantes para API. Também é importante que seja mais desacoplada possível das outras camadas, o uso de interfaces para isto é bem importante nessa camada.

3. **Infrastructure**
   Essa camada possui implementações que estão mais ligadas às decisões de tecnologias externas a API. Conexão com **banco de dados** ou envios de emails (não implementado neste projeto), por exemplo, devem ser implantados nessa camada.

   Obs: O **ORM** (Sequelize) escolhido para esse projeto, não é o ideal para manter a camada de infraestrutura desacoplada da camada de domínio. Por se tratar de um ORM do tipo Active Record as entidades devem estender uma classe desse ORM e com isso tornam-se dependentes do ORM. Poderia ser utilizado algum outro ORM ou implementar um “mapeamento” entidade / table, mas por questão de tempo não foi feito neste projeto.