UNIVERSIDADE PAULISTA

DAVID MEDEIROS FERNANDES

EDUARDO HENRIQUE DA SILVA NOGUEIRA ELER

EVANDRO REMOLLI TEIXEIRA

JOÃO ROBERTO DE ANDRADE BERTOLUCCI

KELVIN DOS SANTOS PEDROZA

DOCUMENTAÇÃO TÉCNICA DA API

ACESSIBILIDADE EM WEBSITE PARA INCLUSÃO DE PCD

Marília

2024

SUMÁRIO

1 INTRODUÇÃO

Esta API REST oferece funcionalidades para gerenciar usuários e seus respectivos comentários. A autenticação é realizada através de tokens JWT (JSON Web Token) e as senhas são criptografadas utilizando o algoritmo bcrypt.

1.1 Docker

Utilizamos o Docker para facilitar a criação do banco de dados. Ele permite criar, gerenciar e executar containers. Container é um processo no sistema operacional, sendo ele mais leve. Cada container tem tudo que se precisa para executar um determinado programa. No nosso caso utilizamos o banco de dados PostgreSQL.

Para executar o container é necessário estar na pasta raiz do projeto e executar o comando “docker compose up”. Assim o Docker vai iniciar o container e a conexão com o banco de dados.

2 AUTENTICAÇÃO

- **Método:** JWT (JSON Web Token)
- **Endpoint de autenticação:** POST /auth/login
- **Corpo da requisição:**
  - email: O email do usuário.
  - password: A senha do usuário em texto plano.
- **Resposta:**
  - Em caso de sucesso, retorna um token JWT no cabeçalho Authorization com o prefixo Bearer.
  - Em caso de falha, retorna um código de status 401 (Unauthorized) com uma mensagem de erro.

3 RECURSOS

3.1 Usuários

- **GET /users:** Retorna uma lista de todos os usuários.
- **POST /users:** Cria um novo usuário.
    - **Corpo da requisição:**
        - name: Nome do usuário.
        - userEmail: Email do usuário (único).
        - password: Senha do usuário em texto plano (será criptografada com bcrypt).
        - FirstQuestion: A primeira pergunta de segurança requerida para a realização da troca de senha.
        - SecondQuestion: A segunda pergunta de segurança requerida para realização da troca de senha.
- **GET /users/{id}**: Retorna um usuário específico pelo seu ID.
- **PUT /users/{id}**: Atualiza um usuário específico.
- **DELETE /users/{id}**: Exclui um usuário específico.

3.2 Comentários

- **GET /comments:** Retorna uma lista de todos os comentários.
- **POST /comments:** Cria um novo comentário.
    - **Corpo da requisição:**
        - descption: Conteúdo do comentário.
        - userName: Nome do usuário.
        - userId: ID do usuário que está fazendo o comentário.
- **GET /comments/{id}**: Retorna um comentário específico pelo seu ID.
- **PUT /comments/{id}**: Atualiza um comentário específico.
- **DELETE /comments/{id}**: Exclui um comentário específico.

4 ESTRUTURA DOS DADOS

4.1 Usuário

JSON

```
{
    "name": "John Doe",
    "email": "johndoe@example.com",
    "firstQuestion": "Exemplo 1",
    "secoundQuestion": "Exemplo 2",
    "password": "1234567!aA",
}
```

4.2 Comentário

JSON

```
{
    "id": 1,
    "description": "Este é um comentário de exemplo.",
    "userId": 1,
    "createdAt": "2023-11-07T12:34:56Z"
}
```

5 SERVIÇOS

Existe a camada de serviço que fica responsável pela manipulação e gestão dos dados, assim separando as responsabilidades entre todas as camadas do sistema. Irei citar alguns tópicos em relação essa camada e suas responsabilidades.

- **Orquestrar a lógica de negócios:** É aqui que as regras de negócio são definidas e aplicadas, como o exemplo a criação de novos usuários e novos comentários. Essa camada é responsável pela comunicação com o banco de dados.
- **Abstrair a complexidade**: Esconde a complexidade das operações de baixo nível, como consultas SQL, chamadas a APIs externas.

6 CONTROLLERS

Essa camada atua como ponto de contado com o mundo externo. É daqui que as requisições dos usuários chegam primeiro, junto com a camada de serviços que formam a parte principal do sistema. Suas reponsabilidades são:

- **Receber as requisições:** É a primeira camada a receber as requisições HTTP dos usuários.
- **Extrair dados:** A partir da requisição, o controller extrai os dados necessários para atender à solicitação do cliente, como parâmetros da URL, dados do corpo da requisição.
- **Preparar a resposta**: Após a camada de serviços executar a lógica de negócio, o controller recebe o resultado e o formata em uma resposta adequada, que será enviada ao cliente.
- **Chamar a camada de serviços**: Delega a lógica de negócio para a camada de serviços.

7 MIGRAÇÕES COM TYPEORM

As migrações do TypeORM são um mecanismo poderoso para gerenciar as mudanças estruturais no seu banco de dados ao longo do tempo. Elas permitem que seja possível definir as alterações necessárias no esquema de forma declarativa, e o TypeORM se encarrega de aplicar essas mudanças de forma segura e controlada.

7.1 Motivos para utilizar as migrações

- **Controle de versões:** As migrações criam um histórico das mudanças ocorridas no banco de dados, facilitando assim o acompanhamento das evoluções do mesmo. Em caso de algum erro é possível rodar um rollback, assim voltando o banco de dados a estado original.
- **Colaboração:** Diferentes pessoas podem trabalhar paralelamente, criando migrações de forma independente.
- **Testes:** As migrações podem ser utilizadas para criar ambientes de teste com dados específicos.

Figura 1 – Exemplo de migração

```
import { MigrationInterface, QueryRunner } from "typeorm";

export class CreateUserTable1628849534983 implements MigrationInterface {
  public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(`
      CREATE TABLE users (
        id uuid NOT NULL DEFAULT uuid_generate_v4(),
        username varchar(255) NOT NULL,
        password_hash varchar(256) NOT NULL,
        email varchar(256) UNIQUE NOT NULL,
        firstquestion varchar(256),
        secondquestion varchar(256),
        recoverPassword varchar(256),
        created_at timestamptz NOT NULL DEFAULT NOW(),
        updated_at timestamptz NULL DEFAULT NOW(),
        CONSTRAINT user_pk PRIMARY KEY (id),
        CONSTRAINT user_un_username UNIQUE (username)
      );
    `);
  }

  public async down(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(` DROP TABLE IF EXISTS users`);
  }
}
```

Fonte: Autor.

8 AUTENTICAÇÃO NAS REQUISIÇÕES

Para acessar endpoints protegidos, inclua o token JWT no cabeçalho “Authorization” de todas as requisições, utilizando o formato Bearer <token>.

9 ERROS DO PROTOCOLO HTTP

- **401 Unauthorized:** Autenticação falhou ou token inválido.
- **404 Not Found:** Recurso não encontrado.
- **403 Forbidden:** Acesso negado.
- **400 Bad Request:** Requisição mal formatada.
- **500 Internal Server Error:** Erro interno do servidor.

10 OBSERVAÇÕES

- **Relação entre usuários e comentários:** Cada comentário pertence a um único usuário, representado pela chave estrangeira “userId”.
- **Criptografia:** As senhas dos usuários são armazenadas de forma criptografada utilizando o algoritmo “bcrypt”.
- **Data de criação:** O campo “createdAt” indica a data e hora em que o recurso foi criado.
- **Paginação:** Para consultas com um grande número de registros, pode ser implementada a paginação para otimizar a performance.
- **Validação:** É importante implementar validação nos dados de entrada para garantir a integridade dos dados.
- **Perguntas de segurança:** As perguntas de segurança são criptografadas ao serem inseridas no sistema para a maior confiabilidade e caso de vazamentos e para não comprometer as informações pessoais dos usuários.