URL Shortener API

Este projeto é uma API para encurtar URLs, desenvolvida em **Node.js** e **TypeScript**. A aplicação utiliza **PostgreSQL** como banco de dados e inclui autenticação JWT para permitir que usuários autenticados criem e gerenciem seus próprios URLs encurtados. Este projeto é ideal para aplicações escaláveis e implementa boas práticas de arquitetura com controllers e services.

Funcionalidades

1. **Cadastro e autenticação de usuários** com JWT.
2. **Encurtamento de URLs**, registrando o ID do usuário (caso autenticado).
3. **Contador de acessos** e possibilidade de **soft delete** para URLs.
4. **Endpoints protegidos** por middleware de autenticação.

Tecnologias Utilizadas

- Node.js v20
- TypeScript
- PostgreSQL
- Sequelize ORM
- JWT para autenticação
- Docker e Docker Compose

Pré-requisitos

Para rodar o projeto, é necessário:

- **Docker** e **Docker Compose** instalados na máquina.
- **Node.js** (opcional, caso queira rodar a aplicação localmente fora do Docker e/ou para testes locais)

Configuração Inicial

1. Clone o repositório:

```
git clone https://github.com/alanjso/url-shortener.git
cd url-shortener
```

2. Crie um arquivo `.env` com as seguintes variáveis de ambiente. Essas variáveis configuram o banco de dados e a autenticação:

```
# .env
```

```
# Configurações do banco de dados PostgreSQL
DB_USERNAME=seu_usuario
DB_PASSWORD=sua_senha
DATABASE=nome_do_banco
DB_HOST="localhost" (ao rodar localmente) ou "172.18.0.2" (ao rodar via
docker)

# Configurações da aplicação
JWT_SECRET=seu_jwt_secreto
CUSTOM_ALPHABET="0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
STUVWXYZ"
HOST="http://localhost"
API_VERSION="v1"
PORT="4000"
```

Executando o Projeto com Docker Compose

Para simplificar a configuração, o projeto usa o Docker Compose para subir o ambiente completo com a aplicação e o banco de dados PostgreSQL.

1. **Construa e inicialize os contêineres**:

   ```
   docker-compose up -d --build
   ```

   Isso vai:

   - Construir a imagem Docker para a aplicação Node.js.
   - Inicializar o banco de dados PostgreSQL.
   - Expor a API na porta `4000` e o PostgreSQL na porta `5432`.
   - Roda a aplicação em segundo plano.

Endpoints Principais

Abaixo estão os principais endpoints do projeto.

1. Autenticação

- **Registro de Usuário**: `POST /v1/src/user`
  - Exemplo de payload:

    ```
    {
      "email": "usuario@exemplo.com",
      "password": "senha123",
      "confirm_password": "senha123"
    }
    ```

- **Login de Usuário**: `POST /v1/src/auth/login`
    - Exemplo de payload:

```
{
  "email": "usuario@exemplo.com",
  "password": "senha123"
}
```

    - Retorna um token JWT para autenticação.

2. URL Encurtamento

- **Criar URL Encurtada**: `POST /v1/shorten`
    - Caso uma token JWT no cabeçalho `Authorization` seja enviado, o user automaticamente ficará associado à URL encurtada.
    - Exemplo de payload:

```
{
  "originalUrl": "https://www.google.com.br/"
}
```

- **Acessar URL Original**: `GET /v1/:shortUrl`
    - Redireciona para a URL original e incrementa o contador de acessos.
- **Listar URLs do Usuário (Autenticado)**: `GET /v1/url/list`
    - Retorna todas as URLs encurtadas pelo usuário autenticado.

Estrutura de Diretórios

Abaixo está um resumo da estrutura de diretórios do projeto:

```
.
├── src
│   ├── app
│   │   ├── auth       # Controller e Router de auth
│   │   ├── url        # Model, Service, Controller e Router de url
│   │   └── user       # Model, Service, Controller e Router de user
│   ├── database       # Configuração do banco de dados e Sequelize
│   ├── middleware     # Middlewares, incluindo autenticação JWT
│   ├── index.ts       # Arquivo principal de inicialização do projeto
```

```
│   └── routes.ts      # Arquivo centralizador com todas as rotas do projeto
├── Dockerfile         # Dockerfile para a aplicação Node.js
├── docker-compose.yml # Arquivo de orquestração com Docker Compose
├── .env               # Variáveis de ambiente (não comitar)
├── .env.exemple       # Exemplo de como configurar as variáveis de ambiente
└── README.md          # Documentação do projeto
```

Testes Unitários

Para testes unitários, o projeto utiliza o **Jest**. Você pode rodar todos os testes com o comando:

```
npm run test
```

Obs: Para utilizar o comando, será necessário executar localmente a aplicação, alterando o DB_HOST no .env e sendo necessário instalação localmente do node e instalação de pacotes.

Observações

- **Soft Delete**: URLs podem são desativadas em vez de excluídas definitivamente.
- **Persistência de Dados**: Um volume Docker `pgdata` é configurado para persistir dados do PostgreSQL.
- **Encurtador de URL.json**: Esse arquivo é uma cópia da colelação de URLs, que foi exportado e deixado na raiz do projeto para ser importado na extenção Thunder Client

Pontos de Melhoria para Escalabilidade Horizontal

- **Quebra do sistema em microserviços**
- **Banco de Dados Distribuído**
- **Cache de Dados dar URLs mais acessadas**
- **Balanceador de Carga**
- **Monitoramento de pontos de gargalo do sistema**
- **Centralização de Logs das instancias e microserviços**

Desafios para Escalabilidade

- **Consistência de Dados**: Em um sistema distribuído, manter a consistência entre várias instâncias e réplicas de banco de dados pode ser um desafio. A aplicação precisa ser projetada para lidar com cenários de consistência eventual e resolver conflitos de dados.
- **Gerenciamento de Cache**: Manter o cache atualizado em um sistema escalável é desafiador, especialmente com múltiplas instâncias que podem ter caches diferentes. Estratégias de

invalidação e atualização de cache são essenciais.

- **Manutenção de Conexões de Banco de Dados**: Em sistemas com múltiplas instâncias, o número de conexões simultâneas ao banco de dados pode aumentar rapidamente, podendo gerar problemas de conexão.