🚀 URL Shortener - NestJS & PostgreSQL

Um encurtador de URLs desenvolvido com **NestJS**, **Sequelize** e **PostgreSQL**, oferecendo autenticação JWT, contagem de acessos e soft delete.

📌 Funcionalidades

- ✅ Encurtamento de URLs
- ✅ Autenticação com JWT (usuário opcional)
- ✅ Contagem de acessos a cada redirecionamento
- ✅ Listagem de URLs por usuário
- ✅ Atualização e desativação de URLs
- ✅ Deploy manual ou via Docker Compose

⚙️ Tecnologias Utilizadas

- NestJS - Framework Node.js
- Sequelize - ORM para PostgreSQL
- JWT - Autenticação
- Docker - Containerização
- PostgreSQL - Banco de dados

🛠️ Instalação e Configuração

1️⃣ Clone o Repositório

```
$ git clone https://github.com/alanjso/url_shortener_ns.git
$ cd url_shortener_ns
```

2️⃣ Instale as Dependências

```
$ npm install
```

3️⃣ Configure as Variáveis de Ambiente

Crie um arquivo `.env` na raiz do projeto e adicione:

```
DATABASE_HOST ="localhost para deploy local ou db para deploy com docker
compose"
DATABASE_PORT ="5432"
DATABASE_USER ="your_postgres_user"
DATABASE_PASSWORD ="your_postgres_password"
```

```
DATABASE_NAME = "url_shortener_ns"
PORT = "4000"
CUSTOM_ALPHABET =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
JWT_SECRET = "your_jwt_secret"
JWT_EXPIRES_IN = "24h"
```

4️⃣ Inicie o Servidor

```
$ npm run start
```

💻 API Endpoints

🔗 Criar URL Encurtada

`POST /urls/shorten`

```
{
  "originalUrl": "https://www.exemplo.com"
}
```

🔷 Resposta:

```
{
  "shortUrl": "http://localhost:4000/urls/abc123"
}
```

Obs: Funciona com ou sem login, mas se estiver logado a url criada fica vinculada e pode ser gerenciada pelo user logado

🔗 Redirecionar para URL Original

`GET /urls/:shortUrl`
🔷 **Resposta:** Redireciona para `originalUrl`

Criar Usuário

POST `/users`

Body:

```
{
  "name": "John Doe",
```

```
  "email": "john@example.com",
  "password": "strongpassword"
}
```

Resposta:

```
{
  "id": "uuid",
  "name": "John Doe",
  "email": "john@example.com"
}
```

Login

POST `/auth/login`

Body:

```
{
  "email": "john@example.com",
  "password": "strongpassword"
}
```

Resposta:

```
{ "access_token": "jwt_token" }
```

🔗 **Listar URLs do Usuário** (autenticado)

`GET /urls`

🔷 **Resposta:**

```
{
  "urls": [
    { "id": 1, "originalUrl": "https://exemplo.com", "shortUrl": "abc123",
"accessCount": 5 }
  ]
}
```

🔗 **Atualizar URL**

`PUT /urls/:id`

```
{
  "originalUrl": "https://novoexemplo.com"
}
```

🔷 **Resposta:** URL atualizada

🔗 Desativar URL

`DELETE /urls/:id` 🔷 **Resposta:**

```
{
  "message": "deleted"
}
```

🚢 Deploy com Docker Compose

🚀 Subindo a Aplicação com Docker

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

📌 Isso iniciará o PostgreSQL e o projeto em NestJS em background.

🛑 Parar os Containers

```
$ docker-compose down
```

Observações

- **Soft Delete**: URLs são desativadas em vez de excluídas definitivamente.
- **Persistência de Dados**: Um volume Docker `urlShortenerPgData` é configurado para persistir dados do PostgreSQL.

📌 Considerações Finais

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.