PROJETO TECH CHALLENGE

POS TECH – FIAP

Aluno:João Victor Torres Araujo (rm369354)

1 Descrição da Arquitetura da Aplicação

O sistema foi implementado utilizando uma **arquitetura em DDD (Domain-Driven Design)** com separação por **módulos de domínio**, seguindo boas práticas de organização de projetos Spring Boot. A estrutura do código favorece **manutenibilidade**, **baixo acoplamento** e **clareza de responsabilidades**, separando o fluxo da aplicação em partes como: apresentação (controllers), serviço (regras e orquestração), persistência (repositories) e infraestrutura/configuração (security, filtros, handlers, inicialização de dados, etc.).

A aplicação está organizada principalmente dentro do pacote:

`com.techchallenge`

E seus domain principais ficam dentro de:

`com.techchallenge.domain`

1.1 Controllers (Entrada – API REST)

Responsável por expor os endpoints REST para acesso externo via HTTP, recebendo requisições e retornando respostas no formato JSON.

Os controllers são definidos com `@RestController`, e também possuem documentação detalhada via **Swagger/OpenAPI** com anotações como `@Operation`, `@ApiResponses`, `@ExampleObject`, etc.

Principais controllers do projeto:

- **AuthController**
  Pacote: `com.techchallenge.domain.auth.controller`
  Responsável pelos endpoints de autenticação, como login e obtenção de dados do usuário autenticado.
- **UsuarioController**
  Pacote: `com.techchallenge.domain.usuario.controller`
  Responsável pelos endpoints de gerenciamento de usuários (criar, atualizar, buscar, deletar, atualizar role, alterar senha, etc.).
  Possui proteção por autenticação e autorização, usando:
  - `@SecurityRequirement(name = "bearerAuth")`
  - `@PreAuthorize(...)`

Além da proteção por Spring Security, o `UsuarioController` também implementa validações adicionais para garantir que:

- O **ADMIN** possui acesso total
- Usuários comuns só acessam seus próprios dados
  Isso é feito via leitura das informações inseridas no request pelo filtro JWT (`request.getAttribute("email")` e `request.getAttribute("role")`).

1.2 Services (Aplicação / Regras de Negócio)

O Serviços contém a lógica de negócio e a orquestração das operações do sistema. Ela é responsável por validar regras, aplicar transformações e chamar os repositórios.

Os serviços são classes anotadas com `@Service`.

Principais services do projeto:

- **UsuarioService**
  Pacote: `com.techchallenge.domain.usuario.service`
  Responsável por toda a regra de negócio relacionada ao usuário, como:
  - criar usuário com validação de email único
  - atualizar dados
  - buscar por ID
  - buscar por email
  - buscar por nome (filtro)
  - alterar senha validando senha atual
  - atualizar role validando enum permitido
  - deletar usuário

  Também integra criptografia de senha com `PasswordEncoder (BCrypt)`.
- **AuthService**
  Pacote: `com.techchallenge.domain.auth.service`
  Responsável pelo processo de autenticação e geração de tokens:
  - valida login e senha
  - gera JWT com `subject=email` e claim `role`
  - retorna token para o cliente
  - interpreta token para o endpoint `/me`
  - possui método para refresh token

Concentra as regras de negócio, e não depende diretamente de HTTP, mantendo o foco em processos e operações do sistema.

1.3 Domain (Domínio – Modelos e Regras Centrais)

O domínio representa o núcleo do sistema, contendo as entidades, enums e objetos que definem o “modelo de negócio”, ela esta dividida nos pacotes de dto e entity.

Ela está estruturada principalmente em:

`com.techchallenge.domain.usuario.entity`

Entidade principal do projeto:

- **Usuario**
    - Campos:
        - id
        - nome
        - email (único)
        - senha
        - endereco
        - role (enum)
        - ultimaAtualizacao

Além disso, existe o enum:

- **UsuarioRole**
    - Define perfis de acesso como `ADMIN`, `CLIENT`, `DONO.`

Contém **DTOs**, que são objetos usados para entrada/saída da API de forma controlada, evitando expor diretamente a entidade do banco.

Exemplos de DTOs do projeto:

- `UsuarioCreateDTO`
- `UsuarioUpdateDTO`
- `UsuarioUpdateSenhaDTO`
- `UsuarioUpdateRoleDTO`
- `UsuarioResponseDTO`
- `LoginRequestDTO`
- `LoginResponseDTO`
- `UserInfoDTO`

1.4 Repositories (Persistência – Acesso ao Banco)

A persistência do sistema utiliza **Spring Data JPA**, conectando a aplicação ao banco de dados **MySQL**.

Os repositórios ficam em:

`com.techchallenge.domain.usuario.repository`

O principal repositório é:

- **UsuarioRepository**
    - fornece métodos como:
        - `findByEmail(...)`

- `existsByEmail(...)`
- `findByNomeContainingIgnoreCase(...)`
- `findById(...)`
- `findAll(...)`
- `save(...)`
- `deleteById(...)`

Responsável por interagir diretamente com o banco, sem conter lógica de negócio. As regras são aplicadas no Service.

1.5 Factory (Transformação / Conversão de Dados)

O projeto utiliza uma classe de fábrica (Factory) para converter DTOs em entidades e vice-versa, deixando o código mais limpo e centralizando a lógica de mapeamento.

Arquivo principal:

- **UsuarioFactory**
  Pacote: `com.techchallenge.domain.usuario.factory`
  Funções comuns:
  - `fromCreateDTO(...)`
  - `toResponseDTO(...)`
  - `applyUpdate(...)`
  - `applySenhaUpdate(...)`
  - `applyUpdateUserRole(...)`

Isso reduz repetição nos controllers e services, mantendo o mapeamento bem organizado.

1.6 Segurança (JWT + Spring Security)

A segurança do sistema foi implementada com **Spring Security + JWT**, garantindo autenticação por token e controle de acesso por role.

Os principais classes de segurança são:

1.6.1 SecurityConfig (Configuração do filtro e permissões)

Pacote: `com.techchallenge.domain.auth.config`

- Define quais endpoints são públicos e quais precisam de autenticação.
- Desabilita CSRF e habilita CORS.
- Registra o `AuthFilter` antes do `UsernamePasswordAuthenticationFilter`.

Endpoints públicos no projeto:

- `/v1/api/auth/login`

- `POST /v1/api/usuarios/registrar`
- swagger `/swagger-ui/**` e `/v3/api-docs/**`

Todos os demais exigem token JWT válido.

1.6.2 AuthFilter (Filtro JWT)

Pacote: `com.techchallenge.domain.auth.filter`

O filtro intercepta requisições e:

- valida se existe `Authorization: Bearer <token>`
- decodifica JWT com o segredo configurado em `app.auth.jwtSecret`
- extrai o email (subject) e role (claim)
- injeta os dados na request:
    - `request.setAttribute("email", email)`
    - `request.setAttribute("role", role)`
- cria autenticação no contexto do Spring:
    - `SecurityContextHolder.getContext().setAuthentication(...)`

Se o token estiver inválido ou expirado, o filtro retorna erro JSON padronizado.

1.6.3 PasswordEncoder (BCrypt)

Pacote: `com.techchallenge.domain.usuario.security`

A classe `SecurityBeansConfig` expõe o bean:

`PasswordEncoder passwordEncoder()`

usando:

`BCryptPasswordEncoder`

Isso garante que senhas sejam armazenadas de forma segura no banco.

1.7 Tratamento Global de Erros (Exception Handler)

O projeto possui um handler global para capturar exceções e transformar em respostas padronizadas.

Classe:

- **GlobalExceptionHandler**
  Pacote: `com.techchallenge.configuration.handler`

Ele trata erros como:

- credenciais inválidas
- token expirado/inválido

- validações inválidas (DTOs)
- entidades não encontradas
- permissões inválidas (403)
- role inválida
- erros genéricos (500)

Isso padroniza o retorno da API com um JSON de erro como:

- status HTTP
- message

1.8 Inicialização de Dados (Seed / Admin automático)

O projeto possui um componente de inicialização automática para criar usuários administrativos no banco ao iniciar o sistema.

Classe:

- **DataInitializer**
  Pacote: `com.techchallenge.configuration.initializer`

Ao subir a aplicação, ele cria dois admins:

- um admin com `SHA-256 + BCrypt`
- um admin “legacy” com `BCrypt direto`

Esse comportamento facilita testes iniciais e uso da aplicação sem precisar cadastrar um admin manualmente.

1.9 Infraestrutura e Containerização (Docker + Compose)

O projeto foi preparado para execução completa em ambiente isolado usando Docker.

Arquivos principais:

- `Dockerfile`
- `docker-compose.yml`
- `.env.exemplo`

Banco de dados

- MySQL 8.2 rodando no container `tech_db`
- Porta local: `3307 -> 3306`

Aplicação

- Container `tech_app`
- Porta local: `8080`

- Porta debug liberada: `5005`

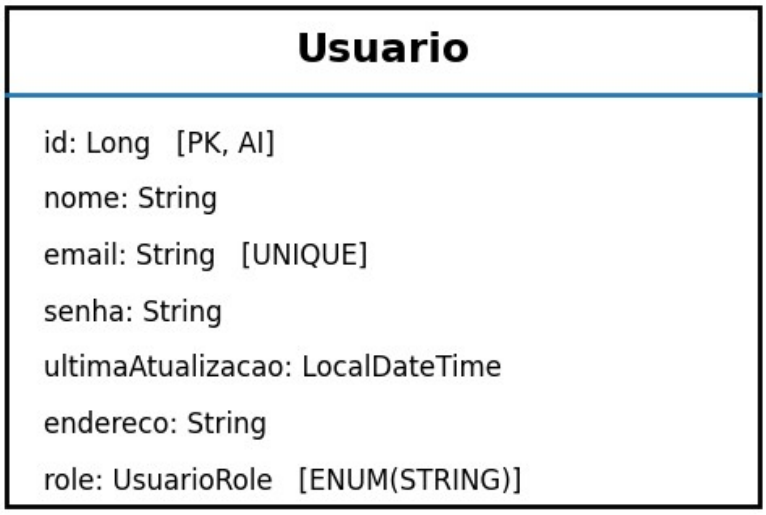
O build é feito em duas etapas:

- **build** (JDK 21) gera o `.jar`
- **run** (JRE 21) executa a aplicação

1.10 Fluxo Geral de Execução da Aplicação

1. O cliente chama um endpoint REST do sistema.
2. O `AuthFilter` valida o token JWT (se for endpoint protegido).
3. O Controller recebe a requisição e chama o Service.
4. O Service aplica regras e validações e chama o Repository.
5. O Repository acessa o banco via JPA/Hibernate.
6. O retorno é convertido para DTO pela Factory.
7. A resposta final é enviada ao cliente.
8. Erros são interceptados pelo `GlobalExceptionHandler`, retornando JSON padronizado.

2 Modelagem das entidades



Legenda: PK = Primary Key | AI = Auto Increment

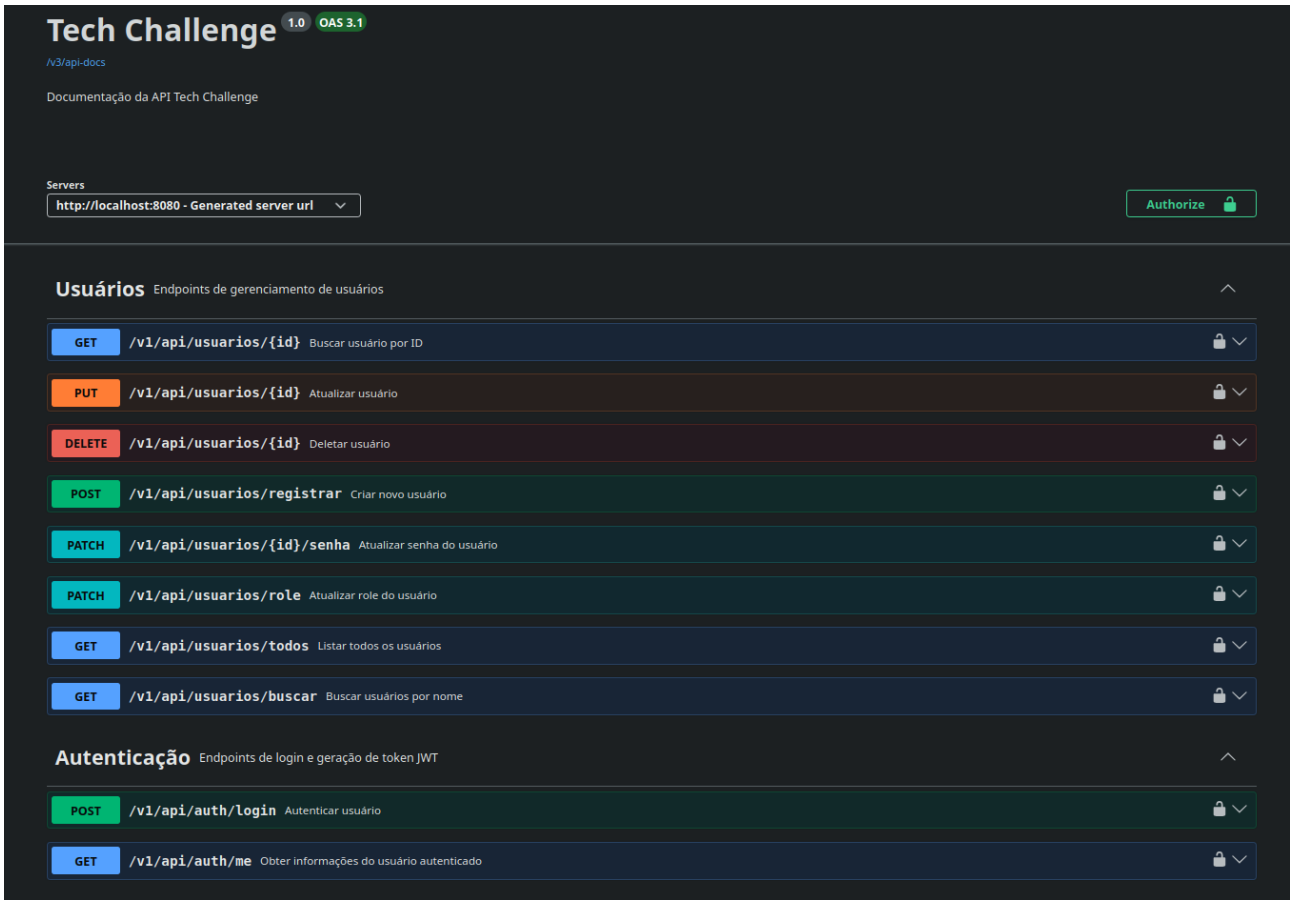
3 Descrição dos endpoints

A API REST expõe endpoints para **autenticação** e **gerenciamento de usuários**, conforme solicitado nos demandas do projeto.
O contexto base utilizado nas requisições é:

Base URL: `http://localhost:8080`
Contexto da API: `/v1/api`

Endpoint	Método	Descrição
`http://localhost:8080/v1/api/auth/login`	**POST**	Autentica um usuário com email e senha, retornando um **token JWT** válido.
`http://localhost:8080/v1/api/auth/me`	**GET**	Retorna os dados do usuário autenticado (requer JWT).
`http://localhost:8080/v1/api/usuarios/registrar`	**POST**	Realiza o cadastro de um novo usuário (endpoint público).
`http://localhost:8080/v1/api/usuarios/{id}`	**GET**	Busca um usuário pelo **ID** (requer JWT).
`http://localhost:8080/v1/api/usuarios/{id}`	**PUT**	Atualiza os dados do usuário (nome/email/endereço) (requer JWT).
`http://localhost:8080/v1/api/usuarios/{id}/senha`	**PATCH**	Troca a senha do usuário (requer JWT).
`http://localhost:8080/v1/api/usuarios/buscar?nome={nome}`	**GET**	Busca usuários por nome (**requer JWT** e normalmente é restrito a ADMIN).
`http://localhost:8080/v1/api/usuarios/role`	**PATCH**	Atualiza a role de um usuário (**requer JWT** e normalmente é restrito a ADMIN).
`http://localhost:8080/v1/api/usuarios/{id}`	**DELETE**	Remove um usuário pelo ID (**requer JWT**).

4 Descrição da documentação Swagger



A aplicação disponibiliza documentação automática da API utilizando **Swagger/OpenAPI**, permitindo a visualização interativa de todos os endpoints REST expostos pelo sistema. Essa documentação facilita o entendimento, testes e validação das operações disponíveis, pois descreve métodos HTTP, rotas, parâmetros, corpo das requisições, códigos de resposta e exemplos de payload.

No projeto, a documentação Swagger é gerada através do componente **SpringDoc OpenAPI**, habilitado via dependência no Gradle (`springdoc-openapi-starter-webmvc-ui`). Ao iniciar a aplicação, o Swagger UI fica acessível pelo navegador, permitindo testar os endpoints sem necessidade de ferramentas externas, como Postman.

4.1 Acesso ao Swagger UI

Após subir o sistema (via Docker ou execução local), a documentação pode ser acessada no endereço:

Swagger UI:
`http://localhost:8080/swagger-ui/index.html`

Além disso, o JSON padrão da especificação OpenAPI é gerado automaticamente pelo SpringDoc em:

OpenAPI JSON:
`http://localhost:8080/v3/api-docs`

4.2 Organização da Documentação por Tags

A documentação está organizada por **tags**, agrupando os endpoints por domínio funcional. No projeto, os grupos principais são:

- **Auth** → endpoints relacionados à autenticação e geração de token JWT.
- **Usuários** → endpoints relacionados ao gerenciamento de usuários.

Essa divisão torna a navegação mais intuitiva e facilita o entendimento do fluxo da aplicação.

4.3 Endpoints Documentados

A interface Swagger exibe os principais endpoints REST do sistema, incluindo:

4.3.1 Autenticação

Endpoints responsáveis por login e obtenção de informações do usuário autenticado:

- **POST** `/v1/api/auth/login`
  Realiza autenticação usando email e senha e retorna um token JWT.
- **GET** `/v1/api/auth/me`
  Retorna informações do usuário autenticado, com base no token JWT enviado.

4.3.2 Usuários

Endpoints responsáveis pela criação e gerenciamento de usuários no sistema:

- **POST** `/v1/api/usuarios/registrar`
  Criação de um novo usuário (endpoint público).
- **GET** `/v1/api/usuarios/{id}`
  Busca usuário por ID.
- **PUT** `/v1/api/usuarios/{id}`
  Atualiza dados do usuário por ID.
- **PATCH** `/v1/api/usuarios/{id}/senha`
  Atualiza a senha do usuário autenticado.
- **PATCH** `/v1/api/usuarios/role`
  Atualiza a role de um usuário (normalmente restrito a ADMIN).
- **GET** `/v1/api/usuarios/buscar`
  Endpoint de busca por nome utilizando query param.
- **DELETE** `/v1/api/usuarios/{id}`
  Remove um usuário do sistema.

4.4 Autenticação no Swagger (JWT – Bearer Token)

O projeto utiliza autenticação via **JWT**, e o Swagger foi configurado para permitir testes em endpoints protegidos.
Após realizar login, o token retornado pode ser inserido no botão **Authorize** do Swagger UI.

O formato utilizado é:

```
Authorization: Bearer {token}
```

Dessa forma, o Swagger passa a incluir automaticamente o header em cada requisição, permitindo testar endpoints protegidos diretamente pela interface.

4.5 Respostas e Códigos HTTP Documentados

O Swagger exibe para cada endpoint:

- **200 OK** → sucesso em requisições comuns (ex.: GET e PUT)
- **201 Created** → criação de recurso (ex.: cadastro)
- **204 No Content** → sucesso sem corpo (ex.: alteração de senha, delete em alguns casos)
- **400 Bad Request** → erro de validação ou parâmetros inválidos
- **401 Unauthorized** → token ausente ou inválido
- **403 Forbidden** → usuário autenticado sem permissão
- **404 Not Found** → recurso não encontrado
- **500 Internal Server Error** → erro inesperado no servidor

Além disso, como o projeto possui um **GlobalExceptionHandler**, os erros são padronizados e retornam mensagens consistentes para facilitar depuração e validação dos testes.

5 Descrição da Coleção Postman

O projeto disponibiliza uma **coleção Postman** (arquivo JSON) com o objetivo de facilitar a execução e validação dos principais cenários da API durante os testes. Essa coleção contém requisições pré-configuradas para autenticação e gerenciamento de usuários, permitindo verificar o funcionamento dos endpoints e o comportamento esperado do sistema em situações de sucesso e erro.

No repositório, a coleção está incluída no arquivo:

`colletion.json`

A utilização dessa coleção permite que qualquer pessoa execute rapidamente as requisições da API sem precisar montar manualmente cada request, garantindo padronização de testes e maior produtividade no processo de validação do Tech Challenge.

5.1 Estrutura Geral da Coleção

A coleção está organizada em um conjunto de requisições que cobrem fluxos essenciais do sistema, principalmente relacionados aos módulos:

- **Auth (Autenticação)**
- **Usuários (Gerenciamento de usuários)**

Além disso, as requisições estão configuradas para utilizar o host local da aplicação:

Base URL: `http://localhost:8080`
Contexto da API: `/v1/api`

5.2 Cenários Cobertos pela Coleção

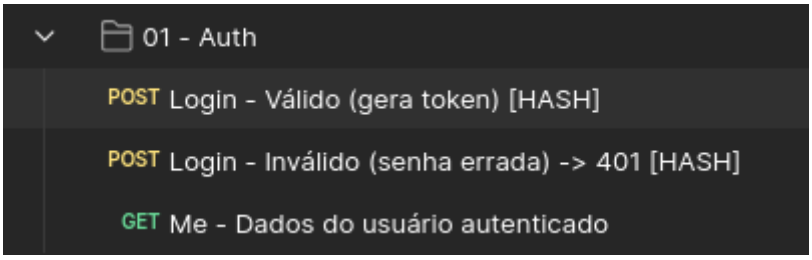
A coleção contempla os cenários principais exigidos para testes da aplicação, incluindo operações de cadastro, login e manutenção de usuários.

5.2.1 Autenticação (Auth)

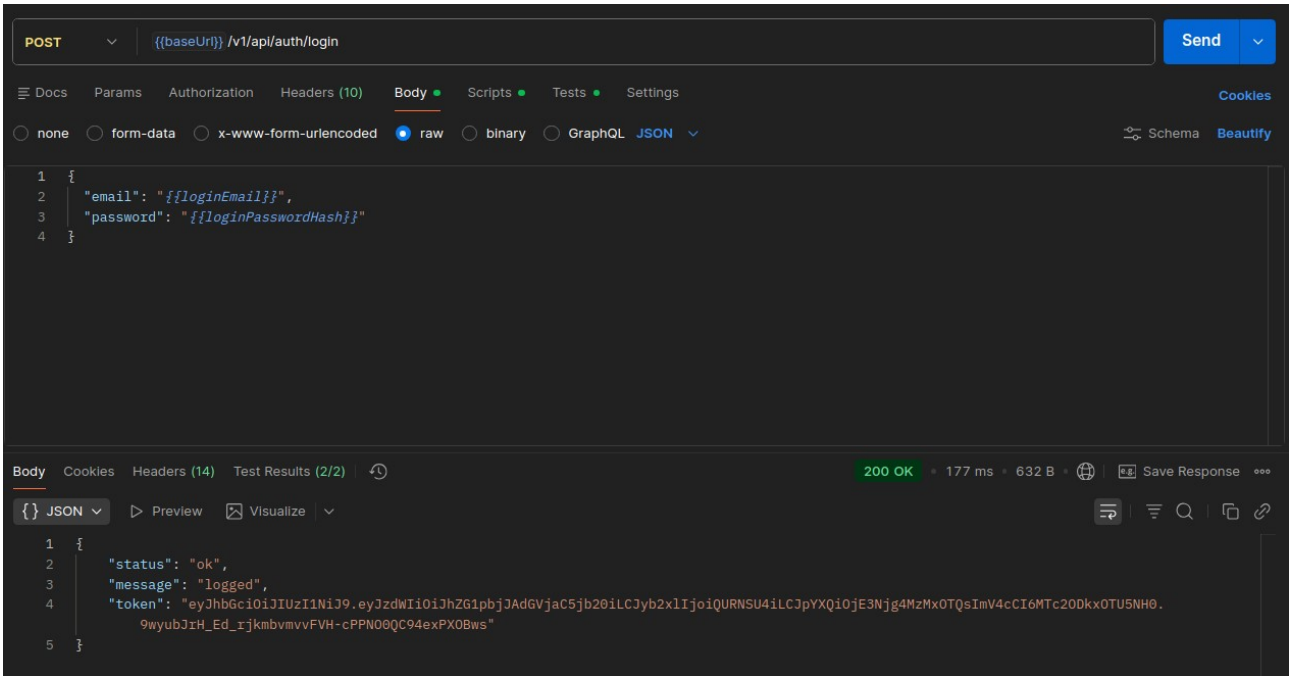
A coleção possui requisições para validar o fluxo de autenticação via JWT:

- **Login (POST /auth/login)**
  Realiza autenticação usando email e senha e retorna um token JWT.
- **Me (GET /auth/me)**
  Retorna os dados do usuário autenticado, utilizando o token JWT no header Authorization.

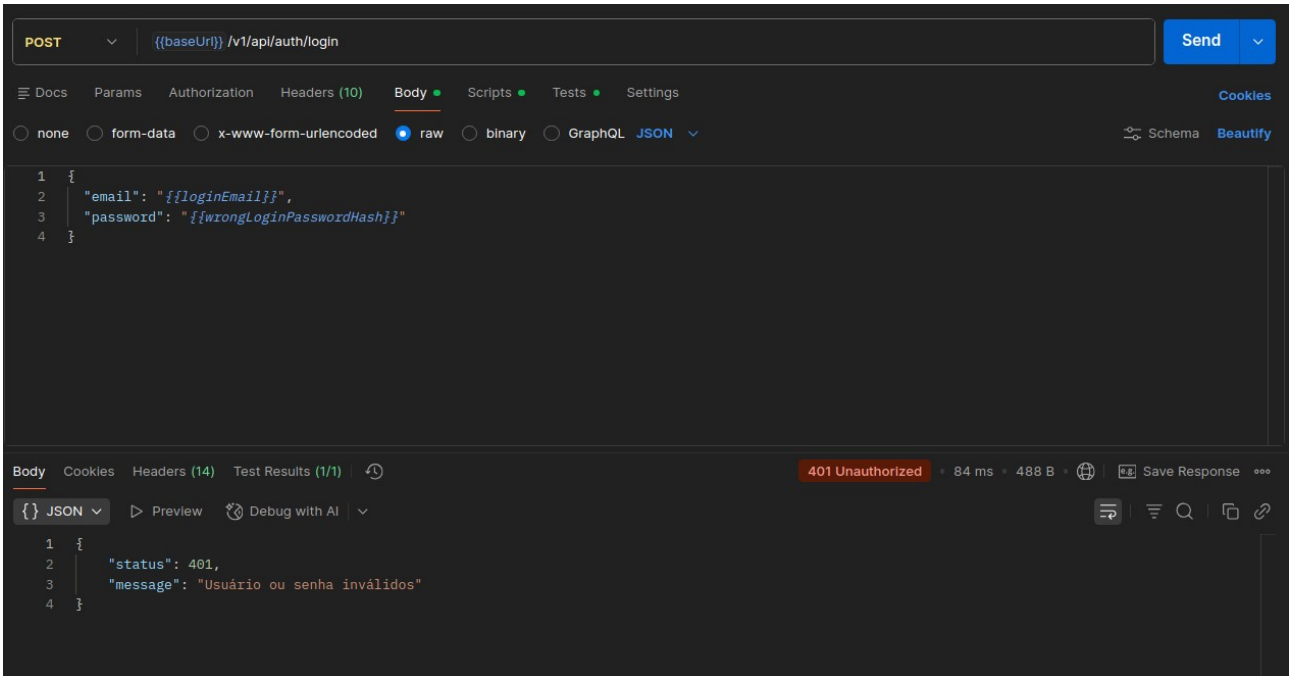
Esse fluxo é essencial pois o sistema exige token JWT para consumir a maioria dos endpoints protegidos.



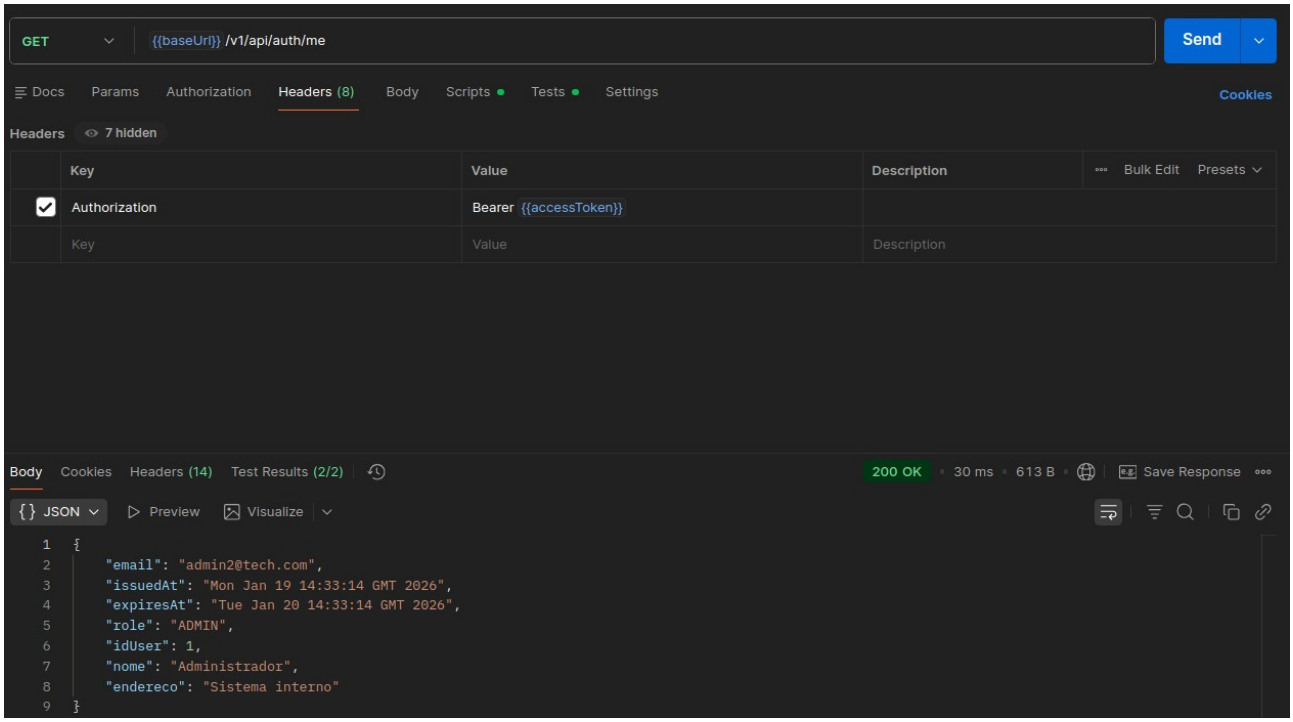
Login - Válido (gera token) [HASH]



Login - Inválido (senha errada) -> 401 [HASH]



Me - Dados do usuário autenticado



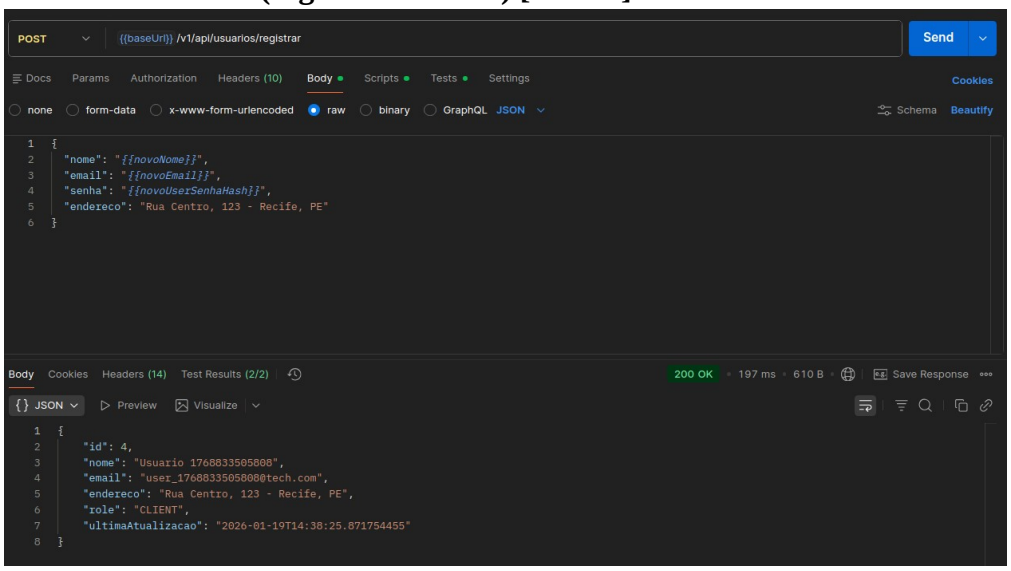
5.2.2 Cadastro de Usuários

A coleção também contém requisições para criação de usuários no sistema:

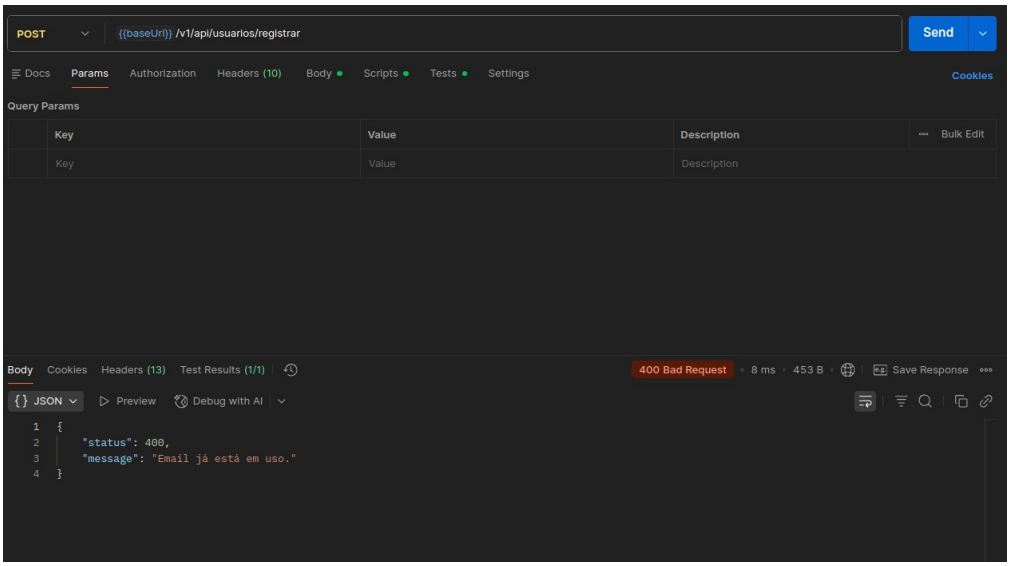
- **Registrar usuário (POST /usuarios/registrar)**
  Cria um novo usuário enviando nome, email, senha e endereço no corpo da requisição.

Essa operação é um endpoint público, permitindo que um usuário seja cadastrado antes da autenticação.

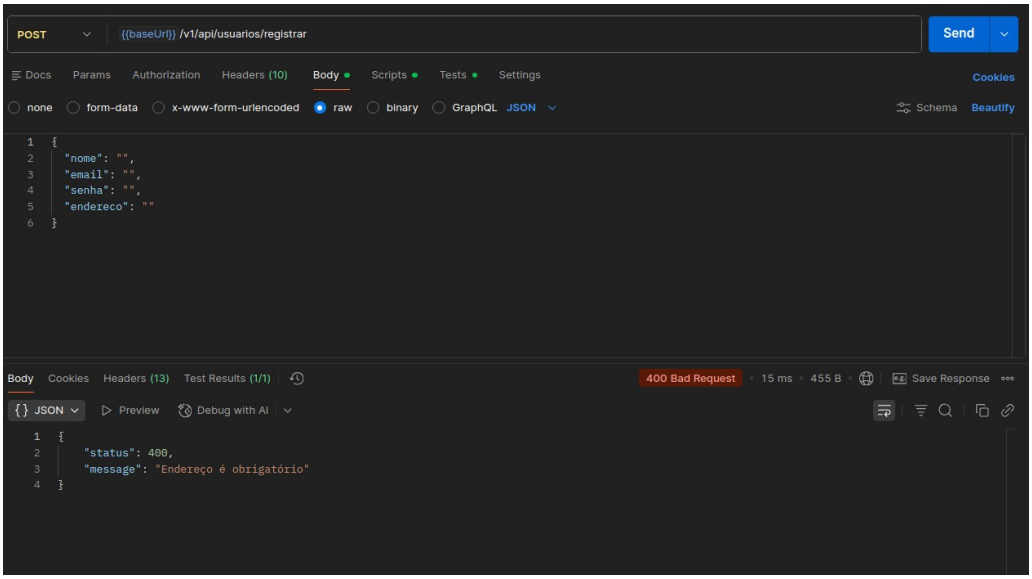
Cadastro - Válido (registrar usuário) [HASH]



Cadastro - Inválido (email duplicado) [HASH]



Cadastro - Inválido (campos obrigatórios faltando)

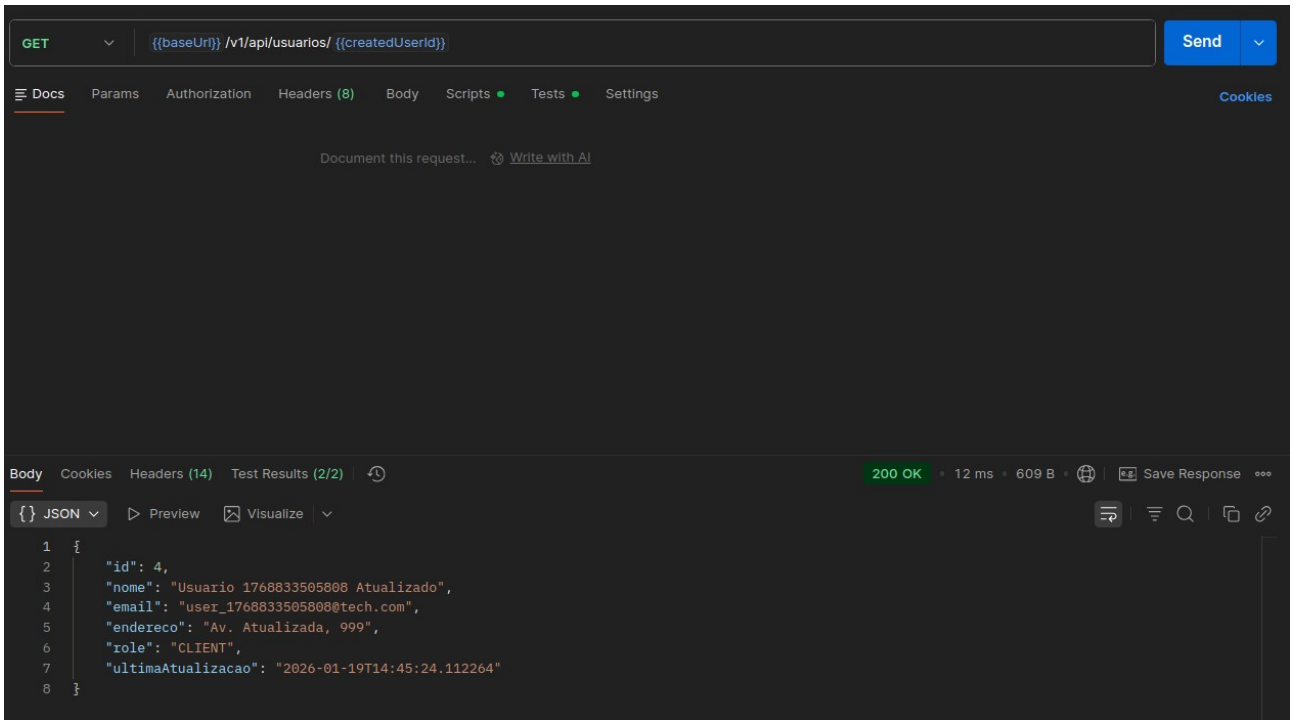


5.2.3 Atualização e Manutenção de Usuários

Para garantir o suporte completo às operações de gestão do usuário, a coleção contém endpoints para:

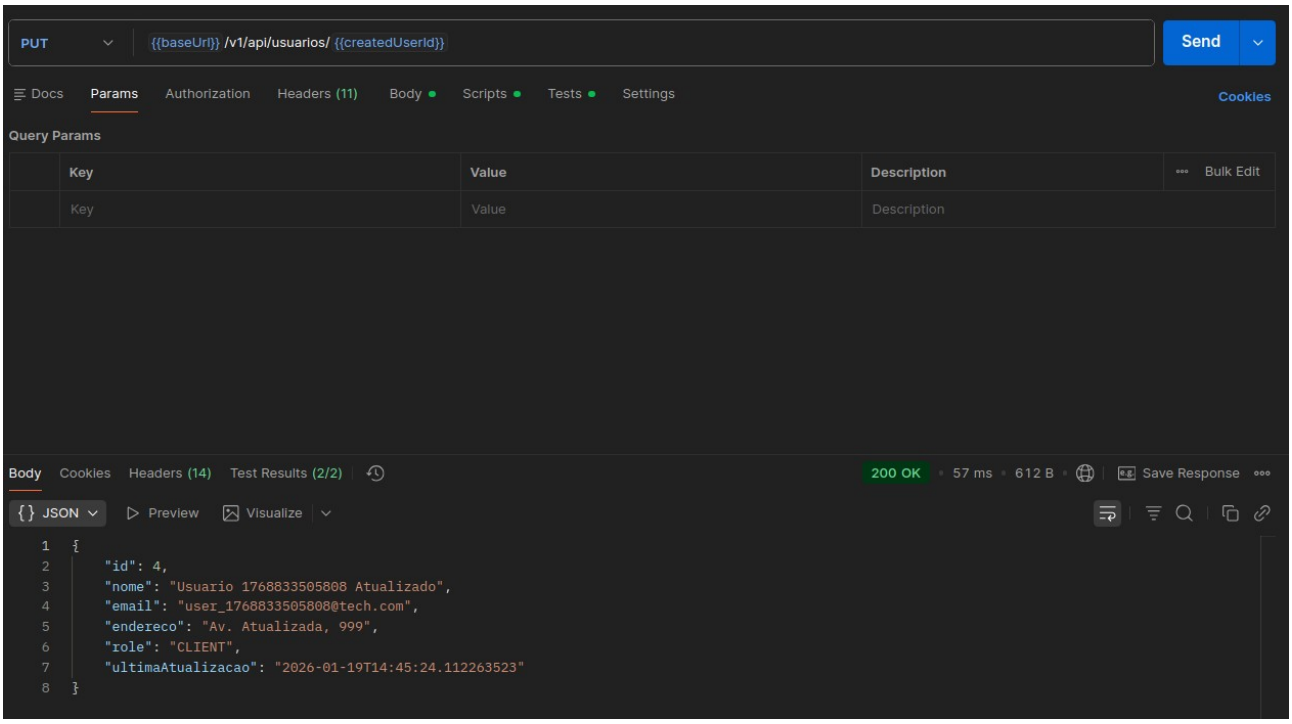
- **Buscar usuário por ID (GET /usuarios/{id})**

Buscar Usuário por ID (GET /{id})

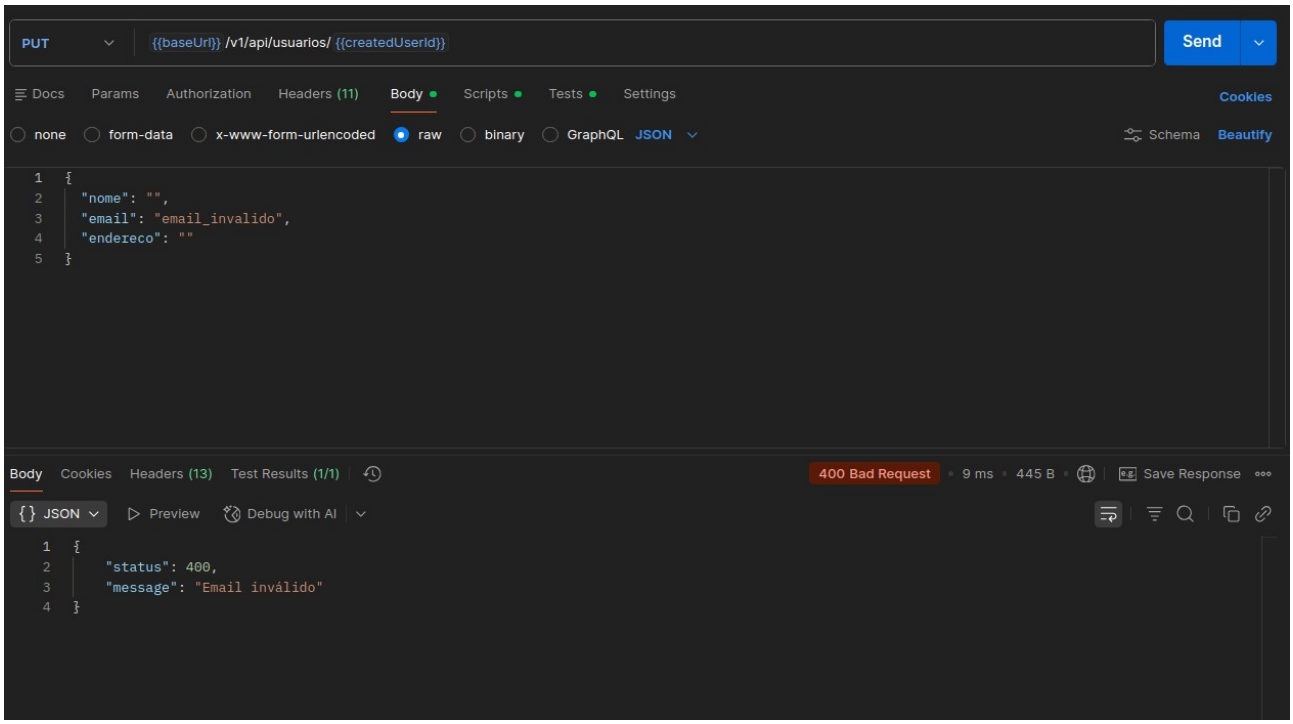


- **Atualizar usuário (PUT /usuarios/{id})**

Atualizar Usuário - Sucesso (PUT)

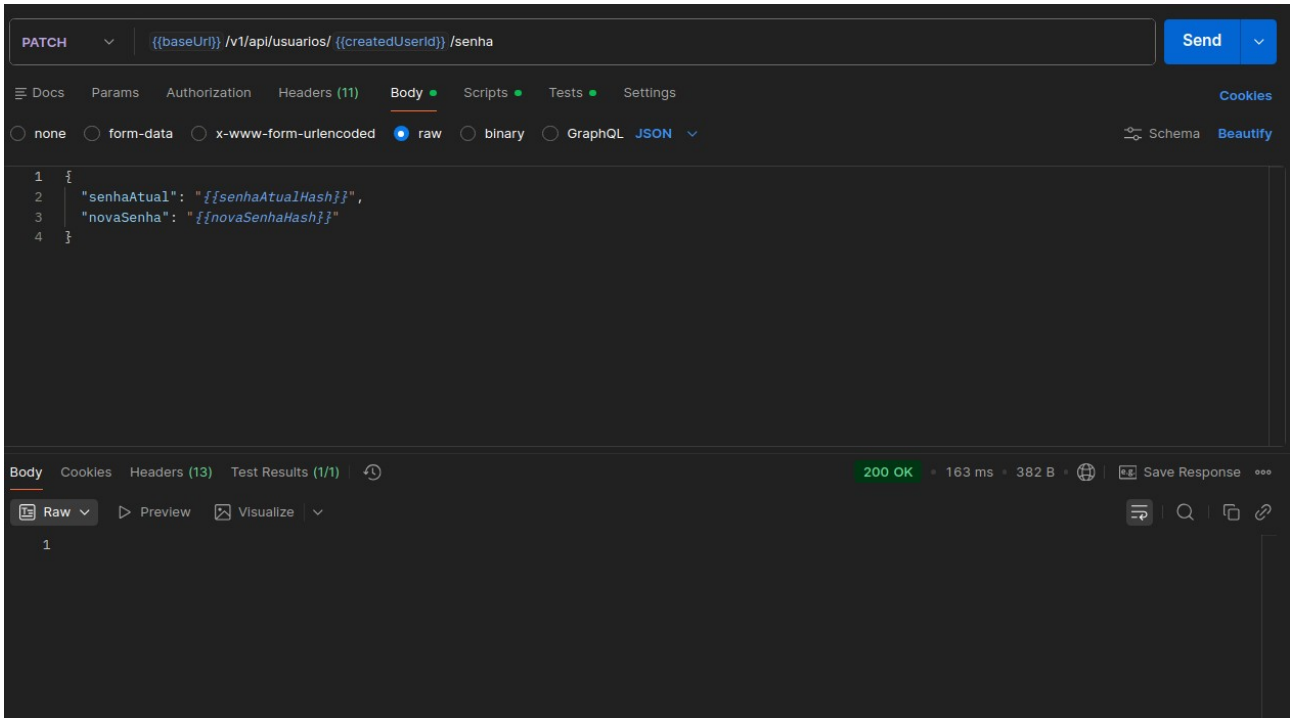


Atualizar Usuário - Erro (PUT com dados inválidos)

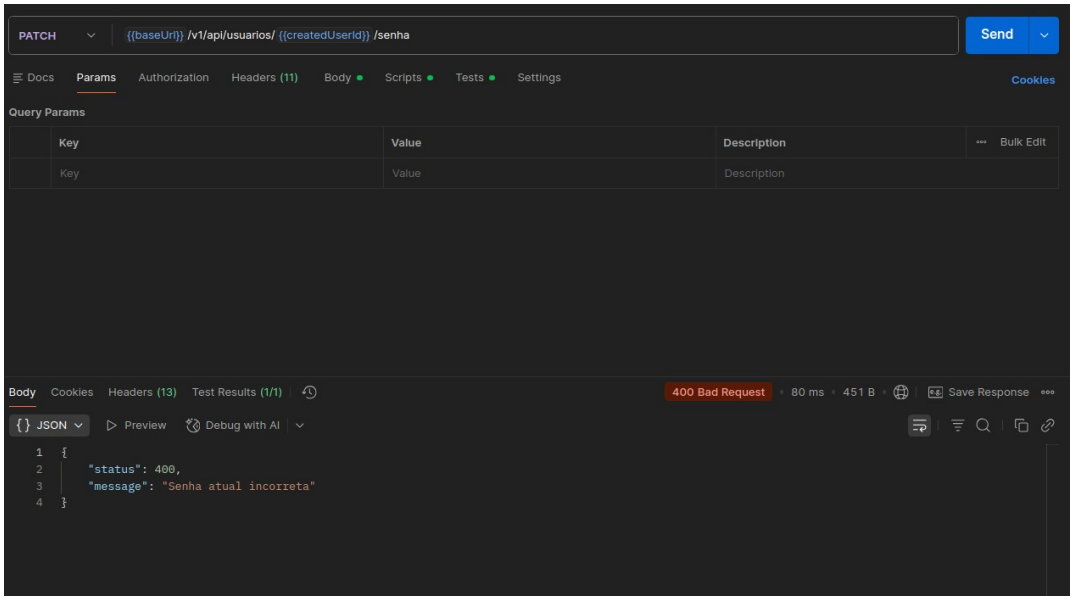


- **Alterar senha (PATCH /usuarios/{id}/senha)**

Alterar Senha - Sucesso (PATCH /{id}/senha) [HASH]

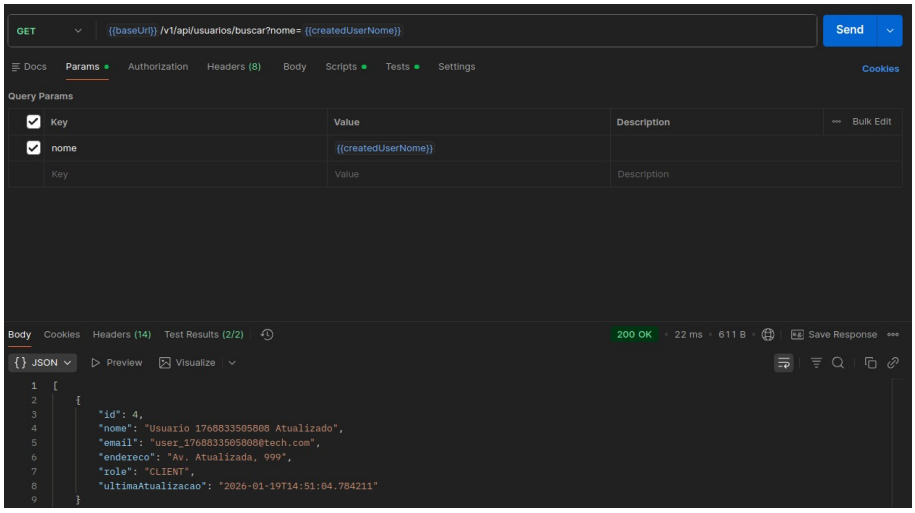


Alterar Senha - Erro (senha atual errada) [HASH]

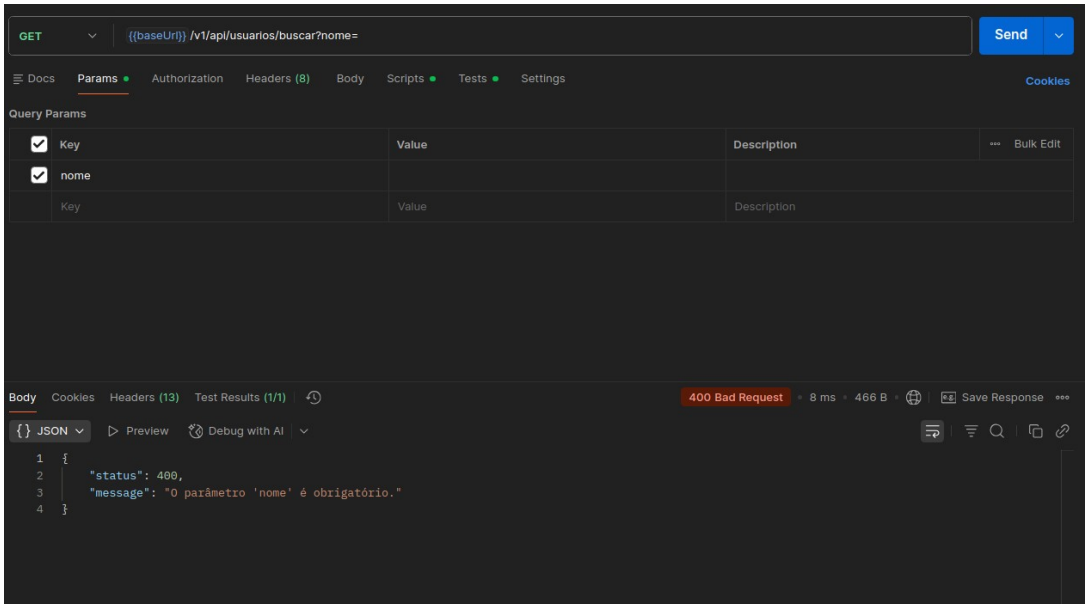


- **Buscar usuários por nome (GET /usuarios/buscar?nome=…)**

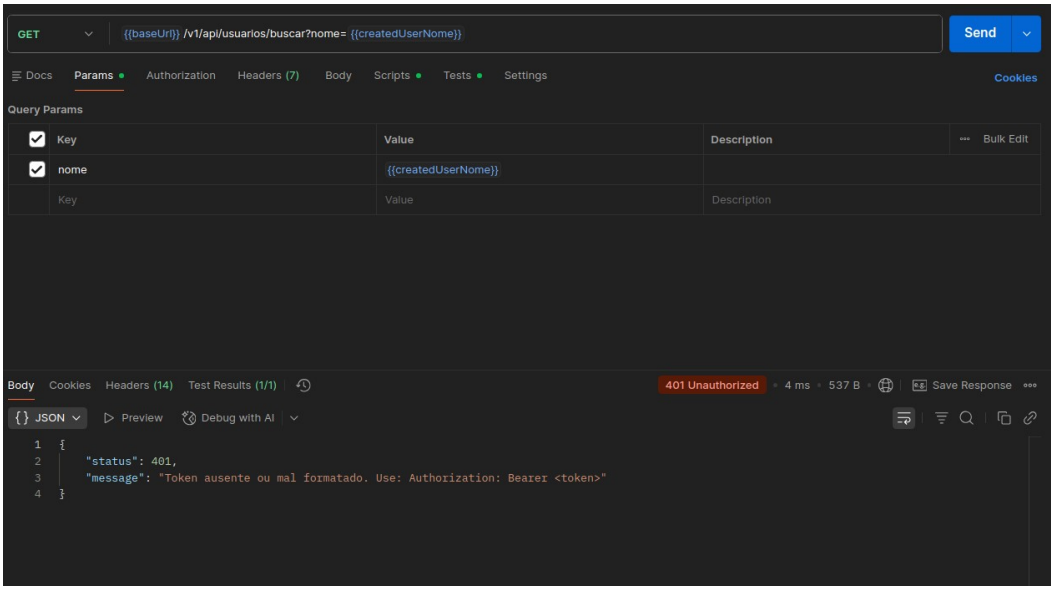
Buscar Usuários por Nome - Sucesso (GET /buscar?nome=...)



Buscar Usuários por Nome - Falha (nome vazio) -> 400

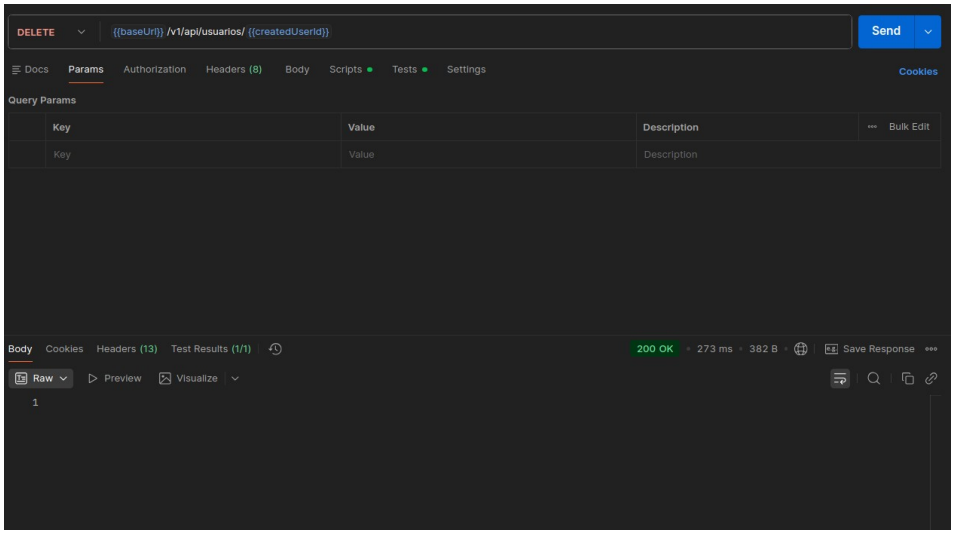


Buscar Usuários por Nome - Falha (sem token) -> 401/403

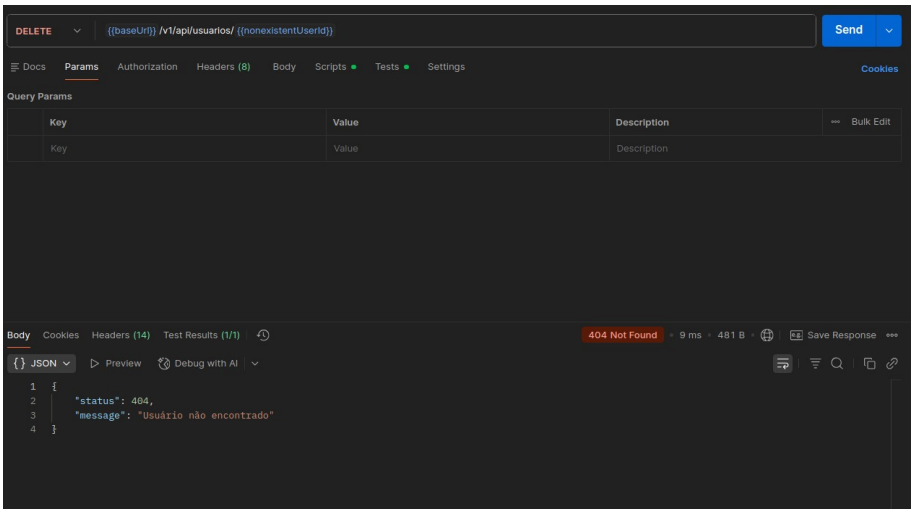


- **Deletar usuário (DELETE /usuarios/{id})**

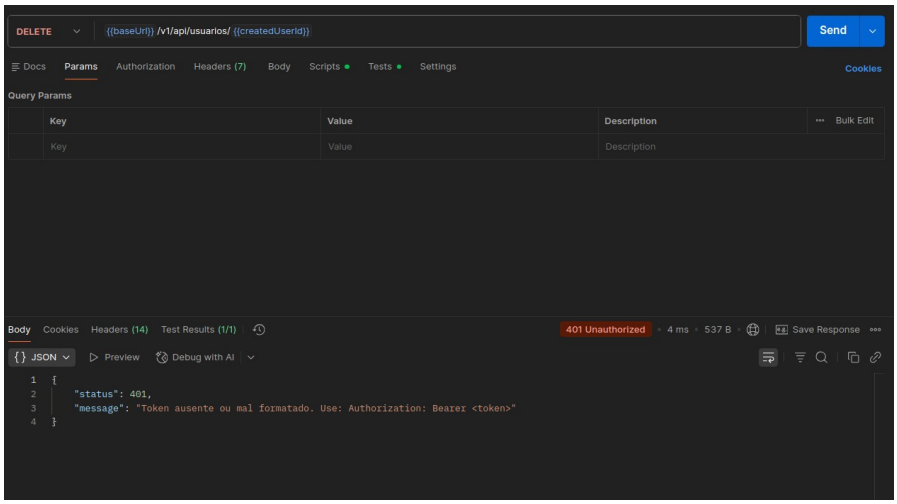
Deletar Usuário - Sucesso (DELETE /{id})



Deletar Usuário - Falha (usuário não existe) -> 404



Deletar Usuário - Falha (sem token) -> 401/403

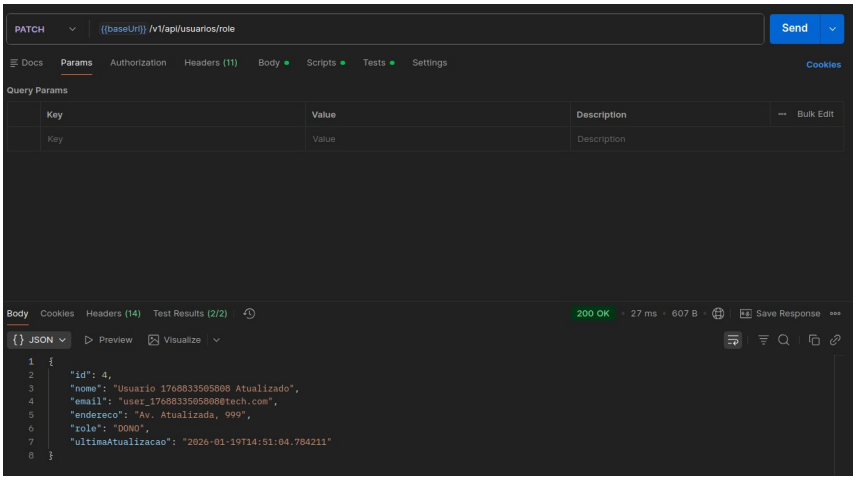


5.2.4 Endpoints Administrativos

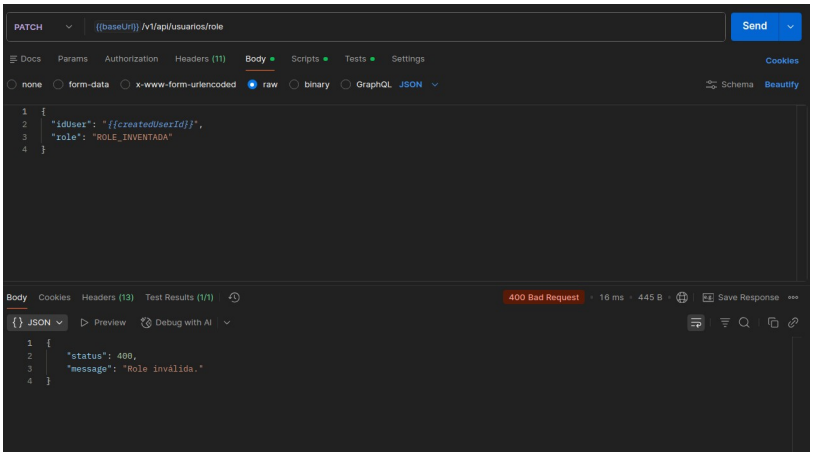
O projeto também possui operações tipicamente restritas a usuários com privilégios de administração, e essas requisições estão incluídas na coleção:

- **Atualizar role de usuário (PATCH /usuarios/role)**
  Permite modificar o perfil (role) de um usuário existente, sendo um endpoint geralmente restrito ao perfil ADMIN.

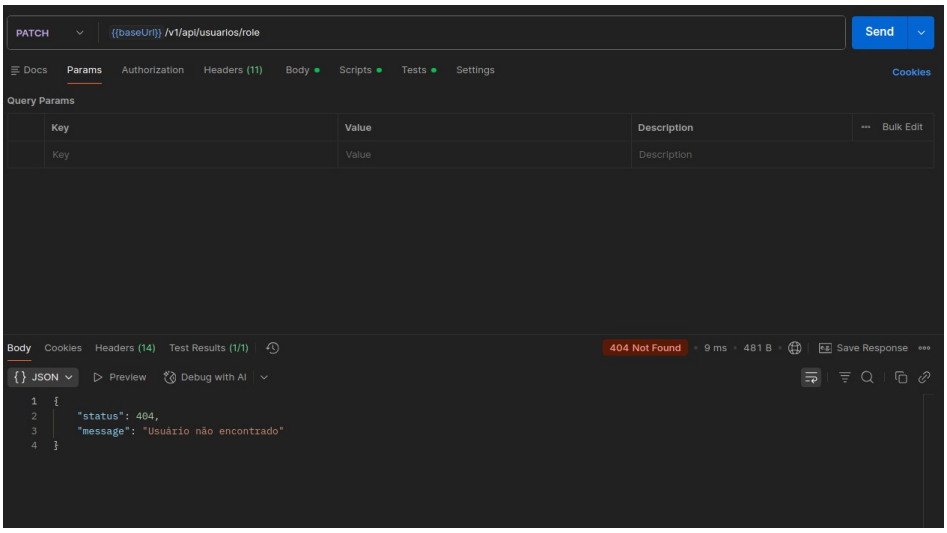
Atualizar Role - Sucesso (PATCH /role) [ADMIN]



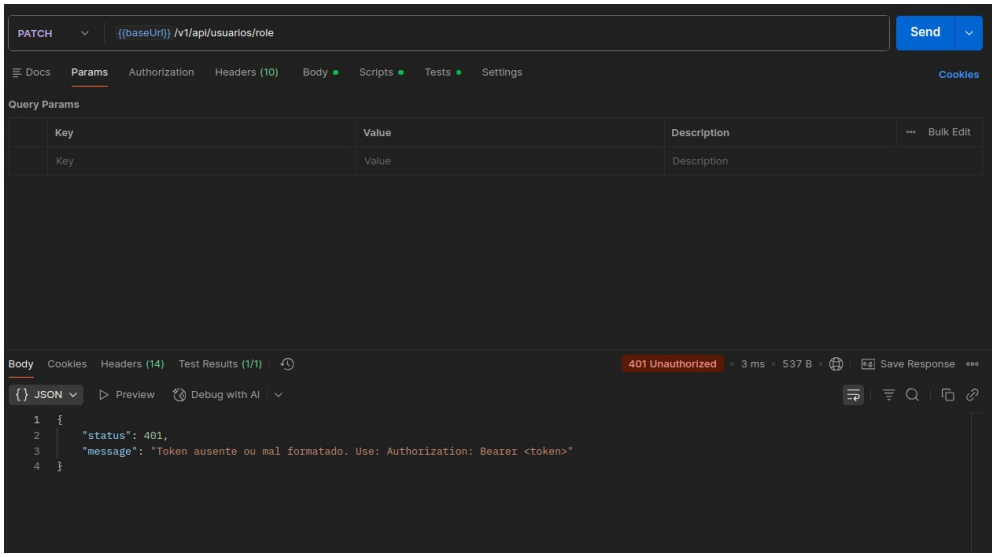
Atualizar Role - Falha (role inválida) -> 400



Atualizar Role - Falha (usuário não existe) -> 404



Atualizar Role - Falha (sem token) -> 401/403



5.3 Uso do Token JWT na Coleção

A coleção foi preparada para trabalhar com autenticação JWT. Após realizar login, o token retornado deve ser inserido no header das requisições protegidas utilizando o padrão:

`Authorization: Bearer {token}`

Esse mecanismo garante que as requisições representem o funcionamento real da aplicação, validando autenticação e autorização conforme configurado no Spring Security e no filtro JWT.

6 Estrutura do Banco de Dados

A aplicação utiliza um banco de dados relacional **MySQL**, executado via Docker (conforme o arquivo `docker-compose.yml`) e acessado pelo Spring Boot através do **Spring Data JPA / Hibernate**. A estrutura das tabelas é gerada automaticamente com base nas entidades anotadas com `@Entity`, utilizando a configuração:

```
spring.jpa.hibernate.ddl-auto=update
```

Isso significa que o Hibernate cria/atualiza as tabelas automaticamente em tempo de execução, conforme a estrutura das classes Java no projeto.

6.1 Banco de Dados e Conexão

O banco é executado no container **tech_db**, utilizando a imagem `mysql:8.2.0`, e é exposto localmente na porta:

- **MySQL**: `localhost:3307` (mapeada para `3306` no container)

As credenciais e o nome do banco são definidos via variáveis de ambiente no arquivo `.env` (modelo em `.env.exemplo`), por exemplo:

- `MYSQL_DATABASE`
- `MYSQL_USER`
- `MYSQL_PASSWORD`

O Spring Boot se conecta ao banco utilizando:

- `SPRING_DATASOURCE_URL`
- `SPRING_DATASOURCE_USERNAME`
- `SPRING_DATASOURCE_PASSWORD`

6.2 Estrutura das Tabelas

No projeto enviado, existe **uma entidade principal persistida no banco**, chamada **Usuario**, localizada em:

```
com.techchallenge.domain.usuario.entity.Usuario
```

Essa entidade é responsável por armazenar os dados do usuário do sistema, incluindo informações de login, perfil e endereço.

6.3 Tabela: USUARIO

A entidade `Usuario` é mapeada automaticamente para uma tabela no banco (por padrão, o Hibernate utiliza o nome da classe como referência).
A tabela possui os seguintes campos:

6.3.1 Campos da Tabela

Coluna	Tipo (JPA)	Restrições	Descrição
`id`	Long	**PK** / Auto Increment	Identificador único do usuário
`nome`	String	—	Nome completo do usuário
`email`	String	**UNIQUE**	Email único utilizado para login
`senha`	String	—	Senha do usuário (armazenada criptografada/encodada)
`ultima_atualizacao`	LocalDateTime	—	Data e hora da última atualização do usuário
`endereco`	String	—	Endereço do usuário
`role`	Enum (STRING)	—	Perfil do usuário (ex.: ADMIN, CLIENT, DONO etc.)

Observação: o campo `role` é um `enum` persistido como **texto**, pois está configurado como:

```
@Enumerated(EnumType.STRING)
```

6.4 Regras e Integridade

A integridade do banco é garantida principalmente por:

- **Chave primária** em `id`
- **Restrição de unicidade** no campo `email` (`@Column(unique = true)`)

Essa regra impede que dois usuários possuam o mesmo email no sistema, o que é essencial para manter consistência no processo de autenticação.

7.Passo a passo para executar a aplicação com Docker Compose

A aplicação foi preparada para rodar de forma totalmente containerizada utilizando **Docker e Docker Compose**, garantindo portabilidade, isolamento de dependências e facilidade de execução em qualquer máquina. O ambiente inclui dois serviços principais:

- **tech_app** → aplicação Spring Boot (Java 21)
- **tech_db** → banco de dados MySQL 8.2

A execução é feita através do arquivo `docker-compose.yml`, que automatiza a criação da rede, inicialização do banco, construção da imagem da aplicação e exposição das portas necessárias.

7.1 Pré-requisitos

Antes de iniciar a aplicação, é necessário ter instalado:

7.1.1 Softwares obrigatórios

- **Docker Engine** (Docker instalado e funcionando)
- **Docker Compose** (geralmente já vem junto no Docker Desktop / Docker Engine)

7.1.2 Recomendado (para desenvolvimento/testes)

- **Git** (para clonar o repositório)
- **Postman** (para executar a coleção de testes)
- **Navegador** (para acessar o Swagger)
- **Java 21** (somente necessário se for rodar sem Docker)

7.2 Clonando o repositório

Para obter o projeto na máquina local, execute:

```
git clone https://github.com/torresvictor100/techchallengecontainer.git
cd techchallengecontainer
```

7.3 Configuração do arquivo `.env`

O projeto utiliza variáveis de ambiente para configurar aplicação e banco de dados. Essas variáveis são lidas automaticamente pelo `docker-compose.yml`.

Crie um arquivo chamado: **`.env`**

E insira as configurações abaixo (exemplo real utilizado no ambiente):

```
# Configurações da aplicação
SERVER_PORT=8080

# DATABASE (MySQL)
MYSQL_ROOT_PASSWORD=root
MYSQL_DATABASE=techchallenge
MYSQL_USER=user
MYSQL_PASSWORD=user123

# Autenticação
APP_AUTH_EMAIL=admin@tech.com
APP_AUTH_PASSWORD=123456

# JWT
APP_AUTH_JWT_SECRET=MinhaChaveSuperSecreta1234567890
APP_AUTH_JWT_EXPIRATION_MS=86400000

# SPRING DATASOURCE
SPRING_DATASOURCE_URL=jdbc:mysql://db:3306/techchallenge?
useSSL=false&allowPublicKeyRetrieval=true&serverTimezone=UTC
SPRING_DATASOURCE_USERNAME=user
SPRING_DATASOURCE_PASSWORD=user123

# JPA / HIBERNATE
SPRING_JPA_HIBERNATE_DDL_AUTO=update
SPRING_JPA_SHOW_SQL=true
SPRING_JPA_HIBERNATE_DIALECT=org.hibernate.dialect.MySQL8Dialect
```

Observações importantes:

- O host do banco no datasource é **db**, pois esse é o nome do serviço MySQL no Docker Compose (rede interna do Docker).
- A porta da aplicação será **8080**, conforme `SERVER_PORT`.

- O `ddl-auto=update` permite que o Hibernate crie/atualize automaticamente as tabelas ao subir o sistema.

7.4 Subindo a aplicação com Docker Compose

Com o `.env` configurado, execute o comando:

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

Esse comando realiza:

1. Construção da imagem do Spring Boot via `Dockerfile`
2. Inicialização do container do MySQL
3. Inicialização da aplicação
4. Mapeamento das portas para acesso externo

Após a execução, os serviços estarão ativos e acessíveis.

7.5 Conferindo se os containers estão rodando

Para verificar se os containers subiram corretamente:

```
docker ps
```

Você deverá ver containers semelhantes a:

- `tech_app`
- `tech_db`

7.6 Acessando a aplicação em execução

Com os containers rodando, a aplicação pode ser acessada pelos seguintes endereços:

7.6.1 Swagger (documentação da API)

`http://localhost:8080/swagger-ui/index.html`

7.6.2 Endpoint base da API

`http://localhost:8080/v1/api`

7.8 Parando a aplicação

Para parar os containers:

```
docker compose down
```

Se quiser remover volumes (apagando dados do banco):

```
docker compose down -v
```