# Usu-backend

Equipe:Matheus Lourenço RM: 367182

## 1. Introdução

### Descrição do problema

Na nossa região, um grupo de restaurantes decidiu contratar estudantes para construir um sistema de gestão para seus estabelecimentos. Essa decisão foi motivada pelo alto custo de sistemas individuais, o que levou os restaurantes a se unirem para desenvolver um sistema único e compartilhado. Esse sistema permitirá que os clientes escolham restaurantes com base na comida oferecida, em vez de se basearem na qualidade do sistema de gestão.

O objetivo é criar um sistema robusto que permita a todos os restaurantes gerenciar eficientemente suas operações, enquanto os clientes poderão consultar informações, deixar avaliações e fazer pedidos online. Devido à limitação de recursos financeiros, foi acordado que a entrega do sistema será realizada em fases, garantindo que cada etapa seja desenvolvida de forma cuidadosa e eficaz.

A divisão em fases possibilitará uma implementação gradual e controlada, permitindo ajustes e melhorias contínuas conforme o sistema for sendo utilizado e avaliado tanto pelos restaurantes quanto pelos clientes.

### Objetivo do projeto

Desenvolver um backend robusto utilizando Spring Boot para gerenciar usuários e atender aos requisitos definidos.

## 2. Arquitetura do Sistema

### Descrição da Arquitetura

O sistema foi desenvolvido seguindo uma estrutura em camadas, um modelo bastante comum por deixar o código organizado e fácil de manter. As principais camadas são: Controller, Service, Repository, DTO, Entity, Exceptions, Handlers, Mappers, Security e Config.

A camada Controller recebe as requisições e envia as respostas ao cliente. A Service contém as regras de negócio do sistema, enquanto a Repository faz a comunicação com o banco de dados. A DTO serve para transferir dados sem expor diretamente as entidades, e as Entities representam as tabelas do banco. A parte de Exceptions define erros personalizados, tratados pelos Handlers. Os Mappers fazem a conversão entre entidades e DTOs, a Security cuida da autenticação e dos tokens, e a Config reúne as configurações gerais do projeto.

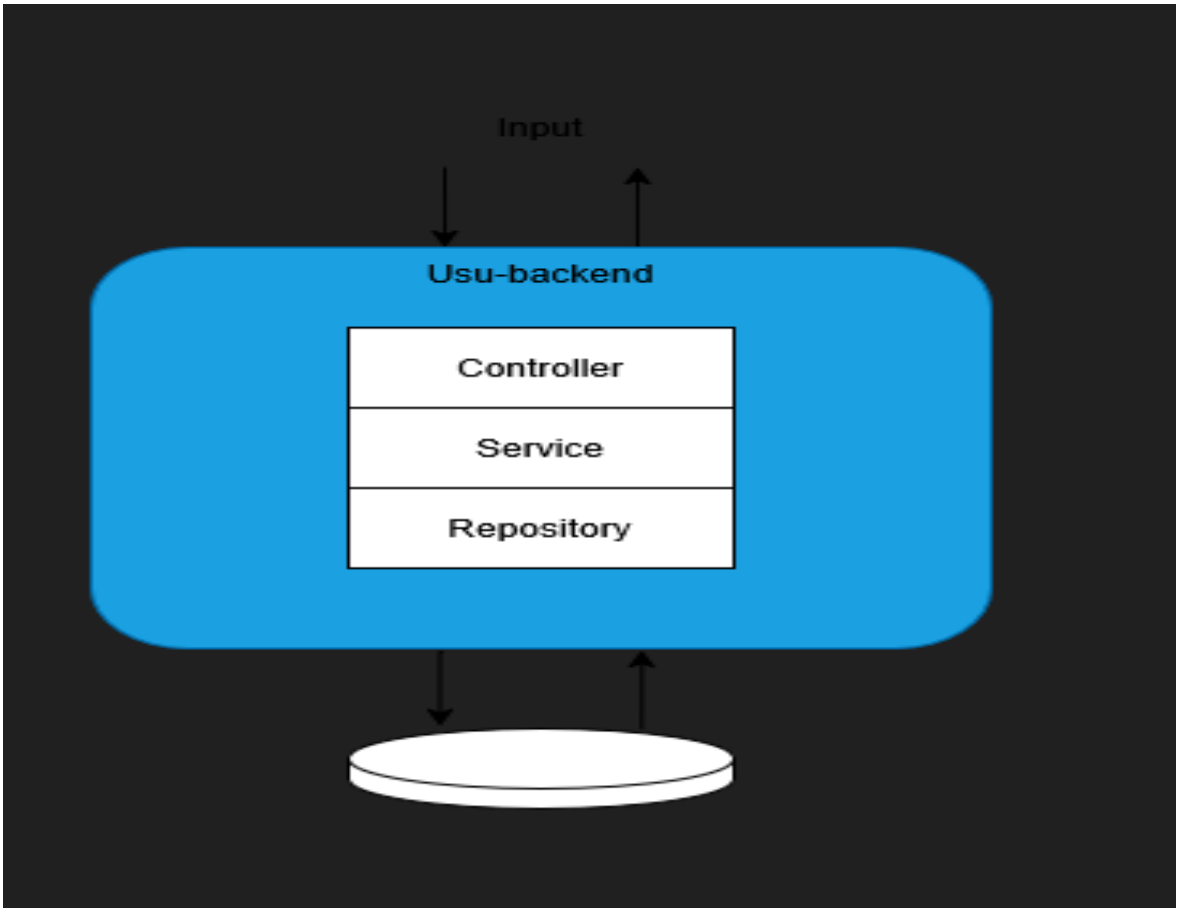
O banco de dados escolhido foi o postgres 16, por ser uma versão estável, bem documentada e com material de apoio disponível.

O Docker foi utilizado para facilitar a execução e o empacotamento do projeto, garantindo que todas as dependências e configurações sejam reproduzidas de forma padronizada em qualquer ambiente. A aplicação foi construída com um Dockerfile de múltiplas etapas (multi-stage build), o que reduz o tamanho final da imagem e melhora o desempenho, utilizando as versões Alpine do Java para ocupar menos espaço.

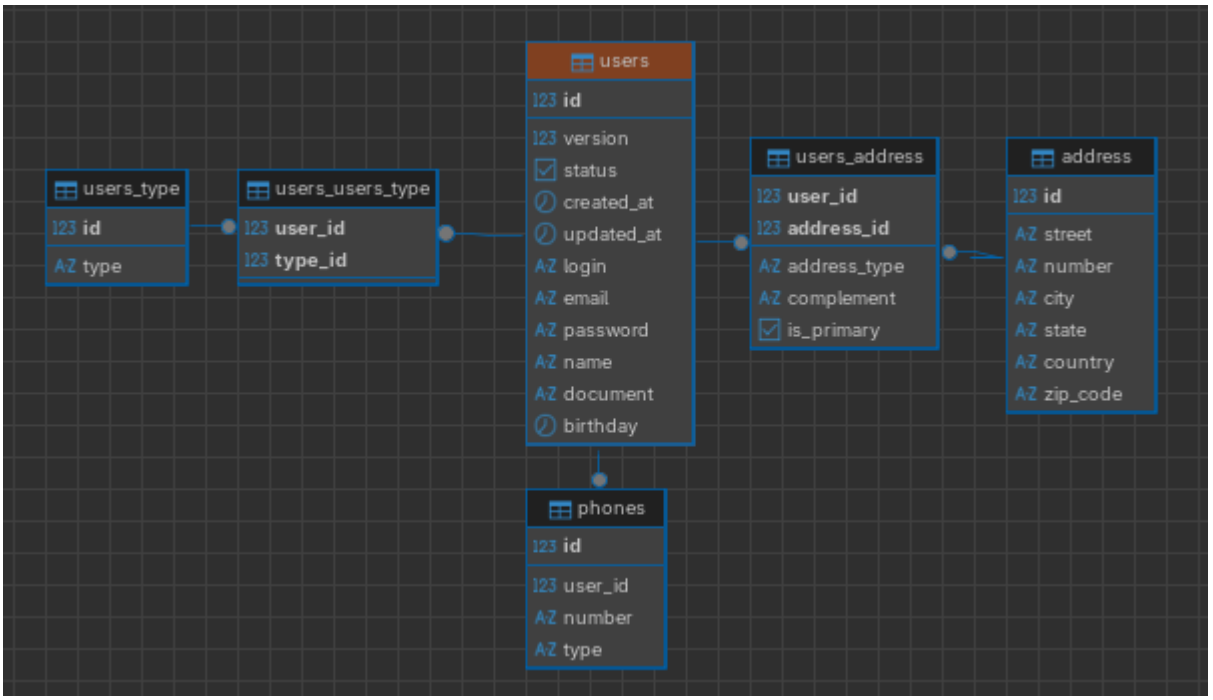
O arquivo docker-compose.yml define dois serviços principais: o backend, que executa a aplicação Java, e o postgres responsável pelo banco de dados. O Docker também realiza a comunicação entre esses serviços por meio de uma rede interna e mapeia as portas necessárias para o acesso externo.

As variáveis sensíveis, como as credenciais do banco e o secret usado para gerar o token JWT, são armazenadas em um arquivo .env, mantendo a segurança e a organização das configurações. Além disso, o uso de volumes garante a persistência dos dados do banco mesmo após a reinicialização dos containers.

## Diagrama da Arquitetura




## Diagrama das entidades



### 3. Descrição dos Endpoints da API

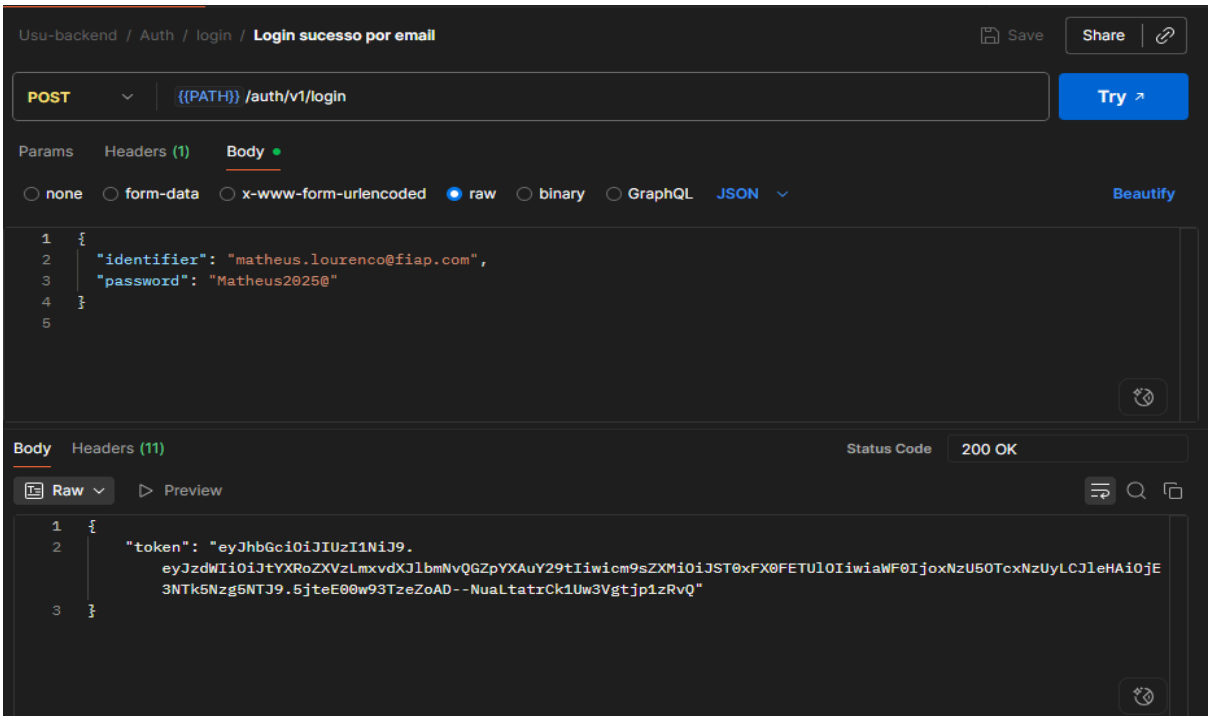
**Tabela de Endpoints**

| Endpoint | Método | Descrição | Parâmetros |
|---|---|---|---|
| /auth/v1/login | POST | Gera um token de autenticação a partir do e-mail e senha do usuário. | - |
| /users/v1 | POST | Cadastrar um novo usuário no sistema. | - |
| /users/v1 | GET | Listar os usuários cadastrados(paginado e com filtro opcional por nome). | name |
| /users/v1/{id} | GET | Consultar dados de um único usuário pelo id. | id |
| /users/v1/{id} | PATCH | Atualizar dados de um usuário existente. | id |
| /users/v1/{id} | DELETE | Excluir um usuário do sistema. | id |
| /users/v1/password | PATCH | Atualizar a senha de um usuário autenticado. | - |

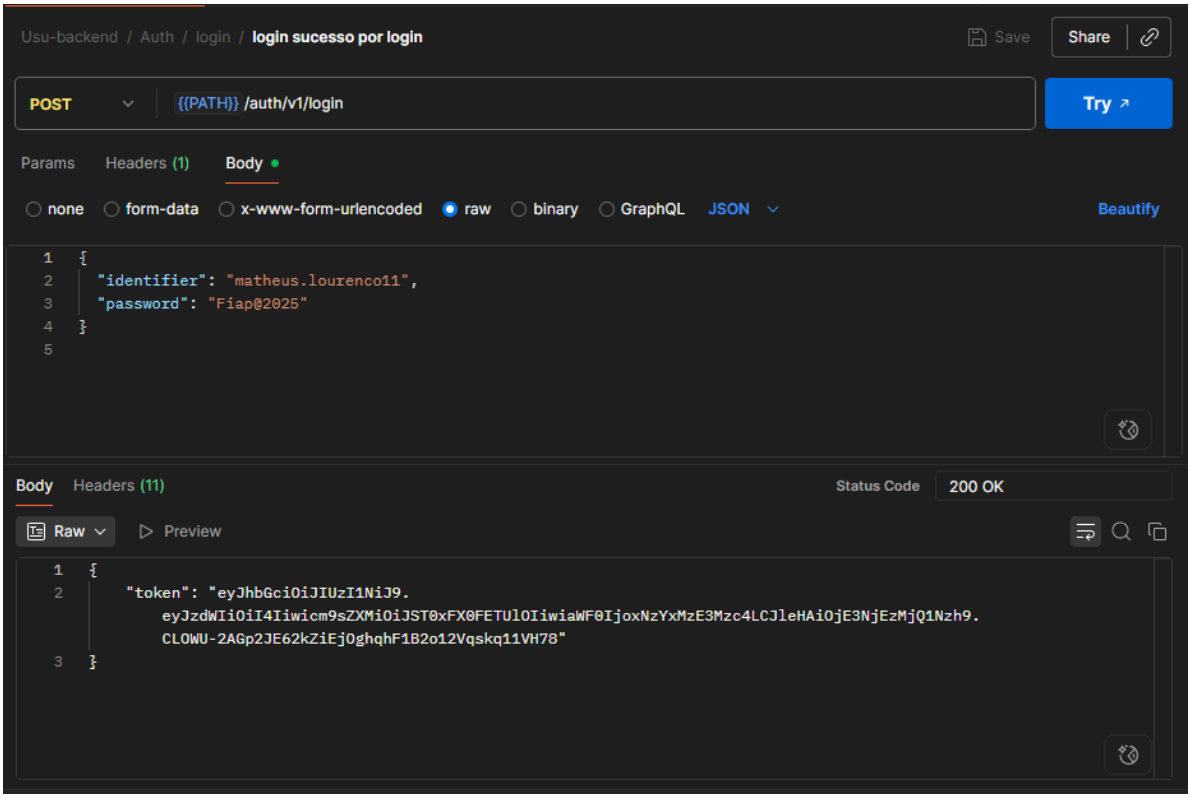
## Exemplos de requisição e resposta

### 3.1. Login

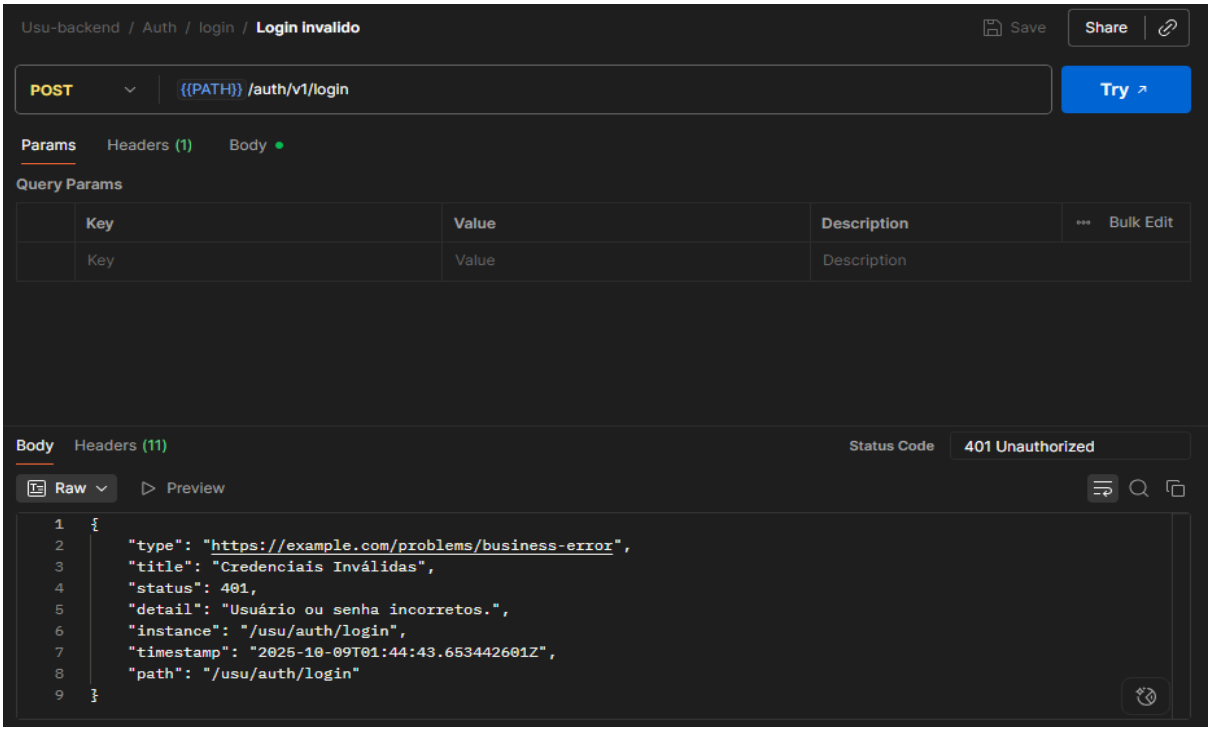
#### 3.1.1 Login usando email




#### 3.1.2 Login usando login

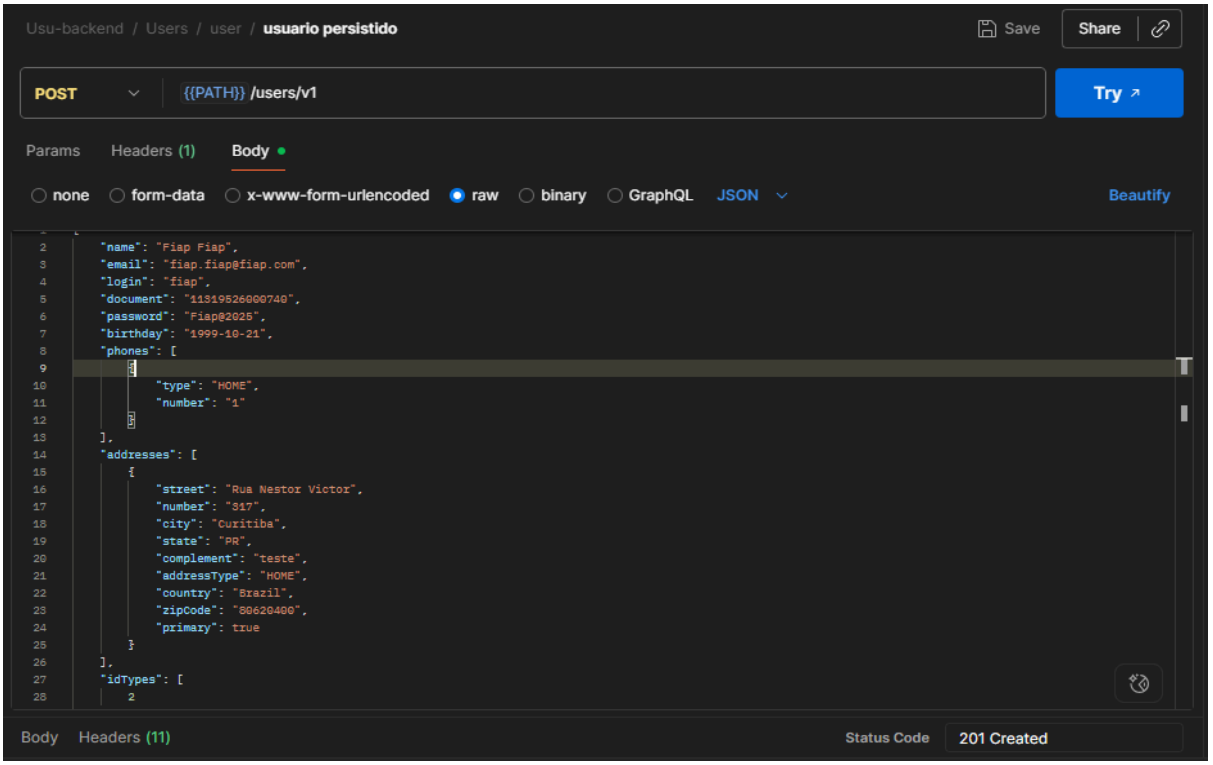


### 3.1.3 Dados inválidos

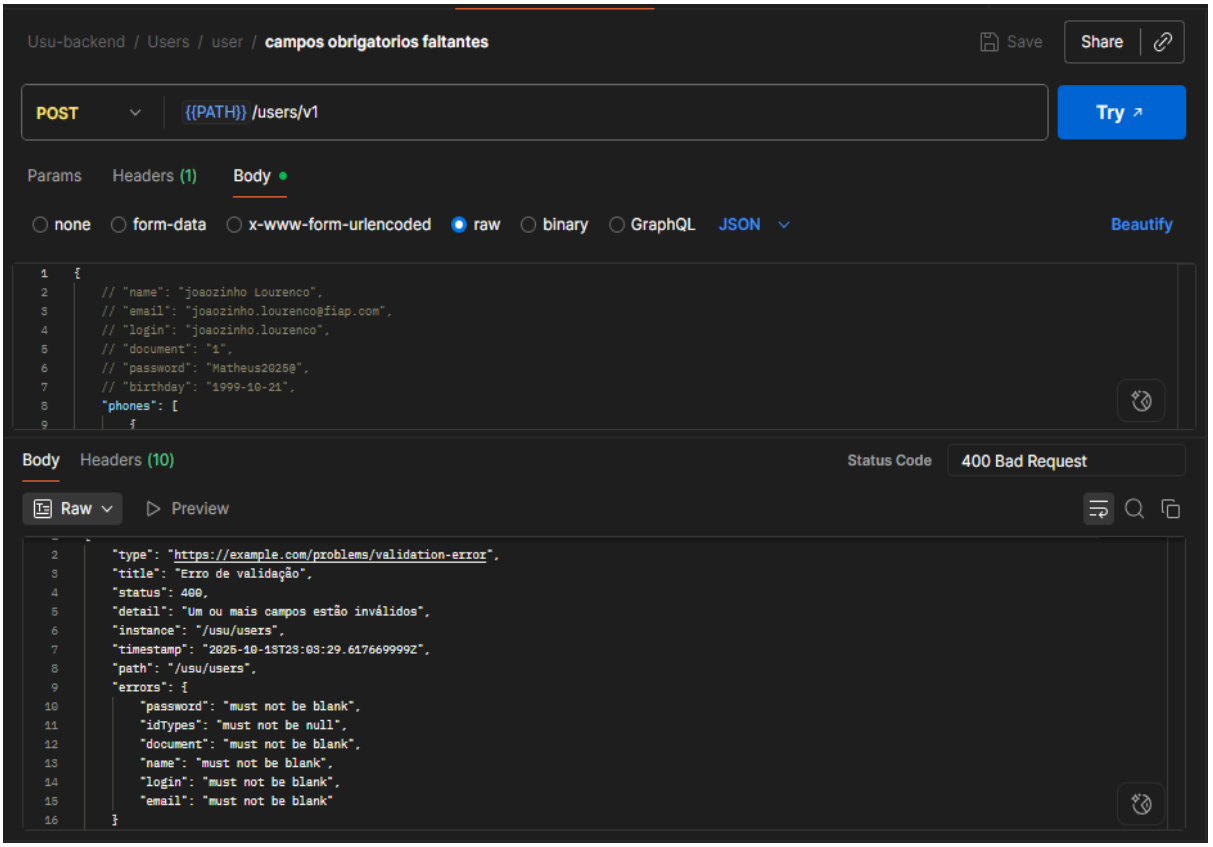



## 3.2. Salvar usuário

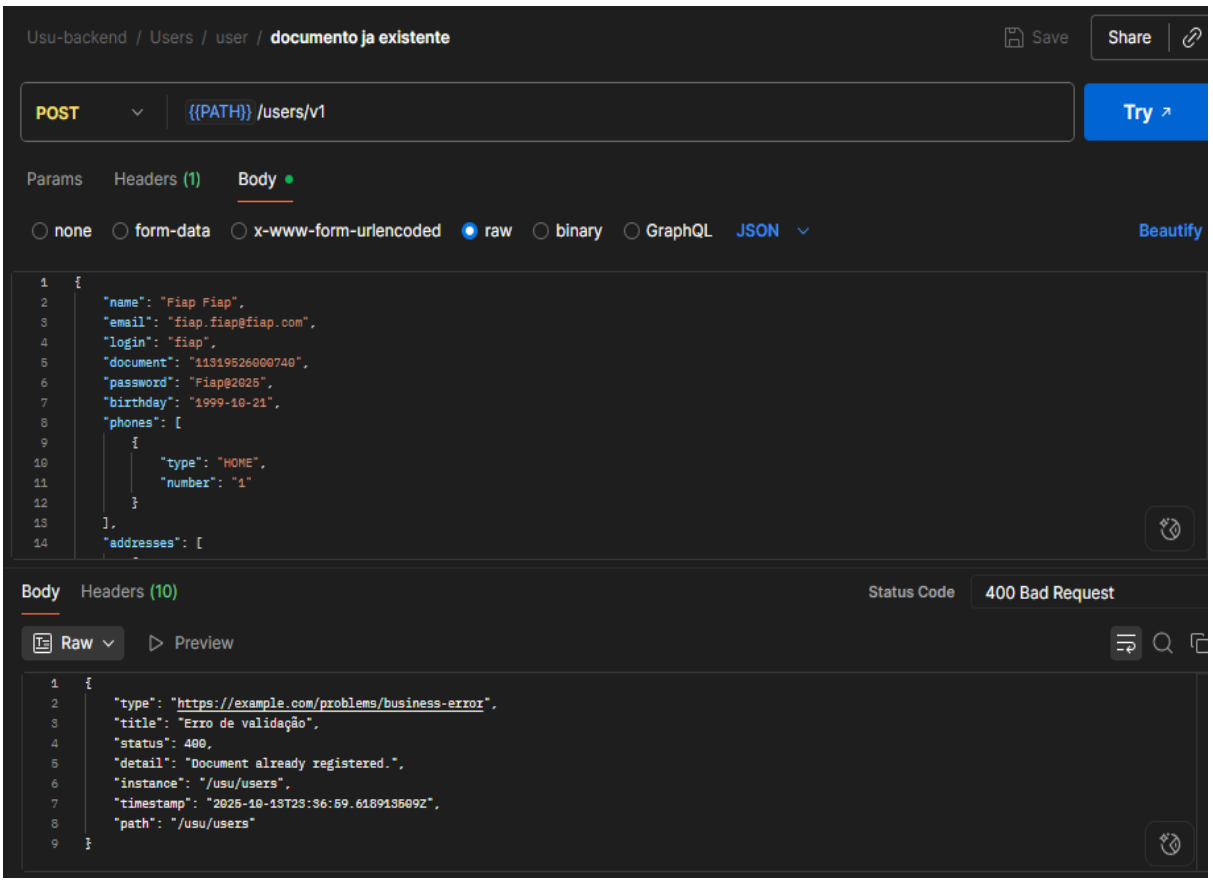
### 3.2.1 Persistir usuário



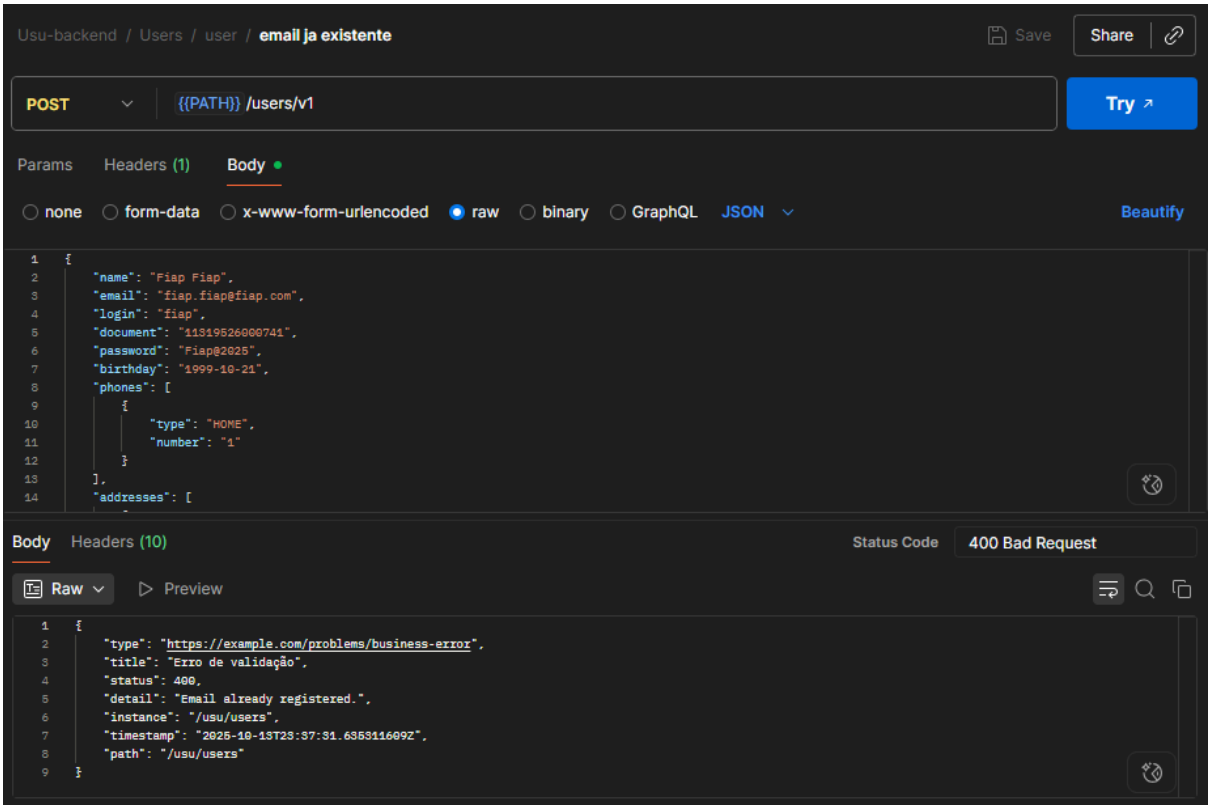
### 3.2.2 Campos obrigatórios não informados



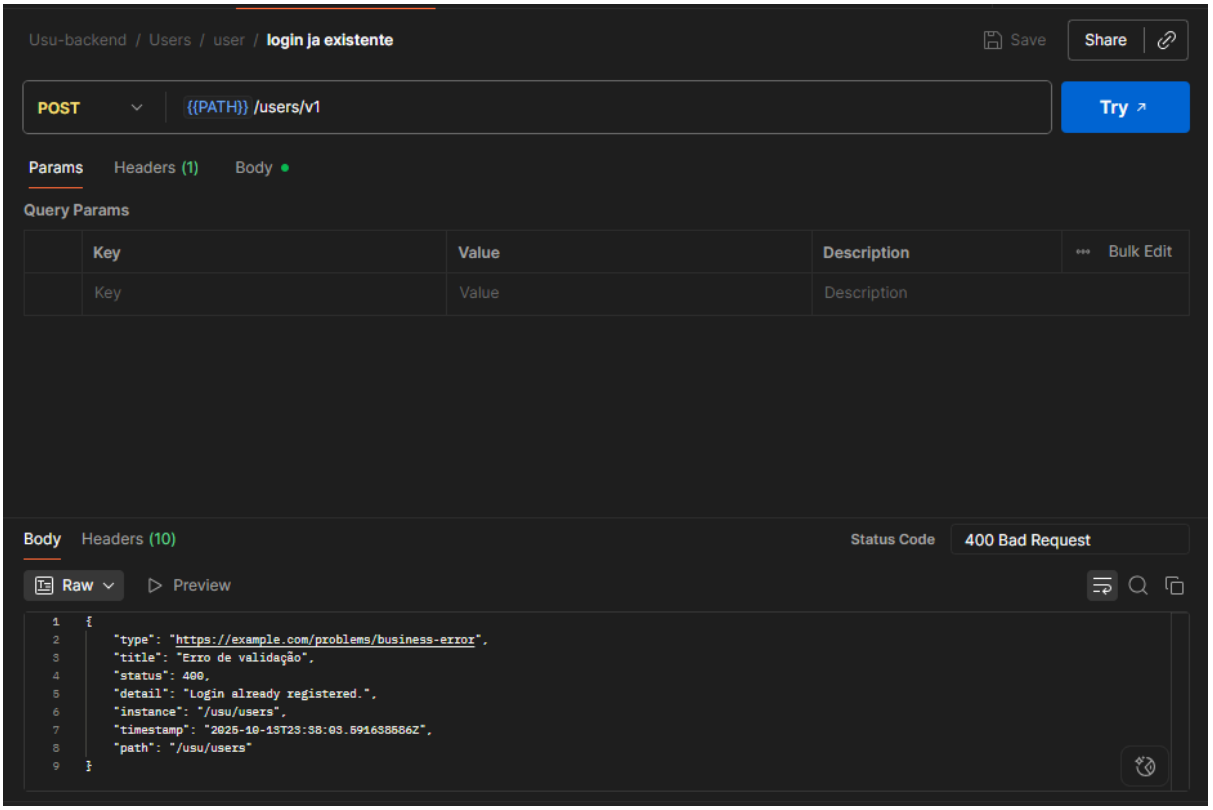

### 3.2.3 Documento já registrado



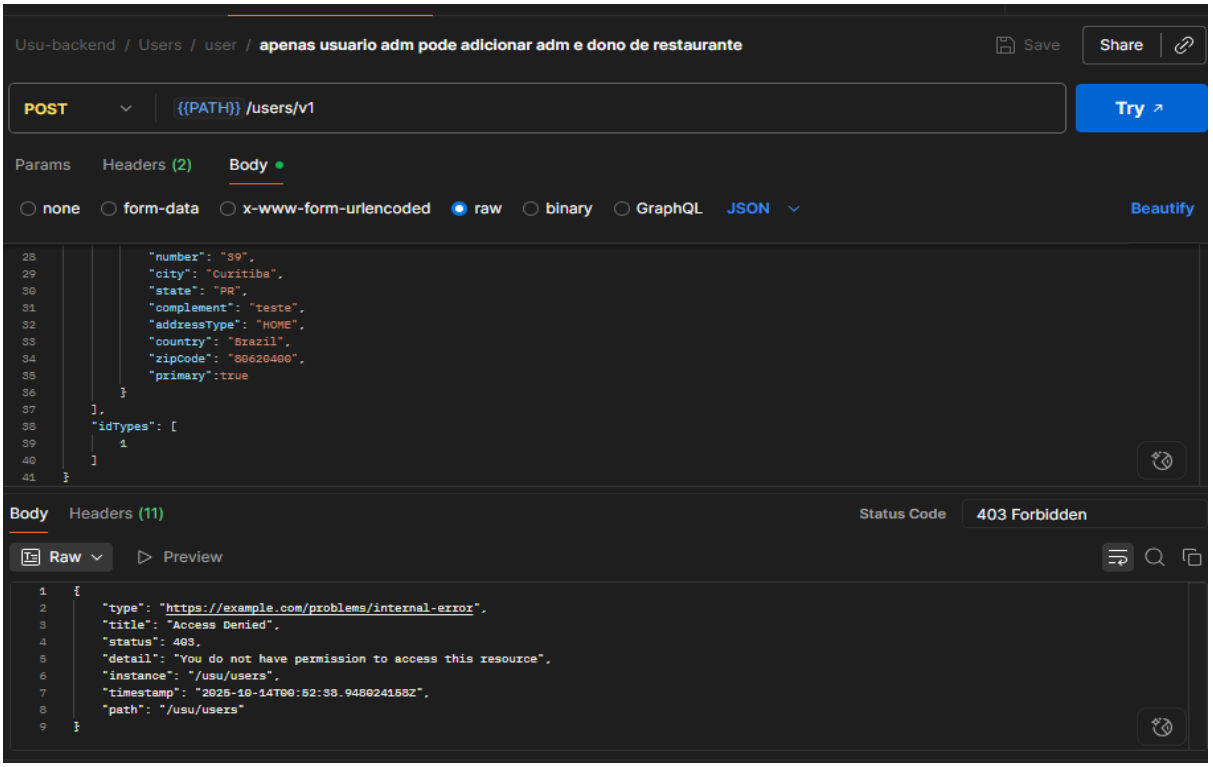
### 3.2.4 Email já registrado




### 3.2.5 Login já registrado

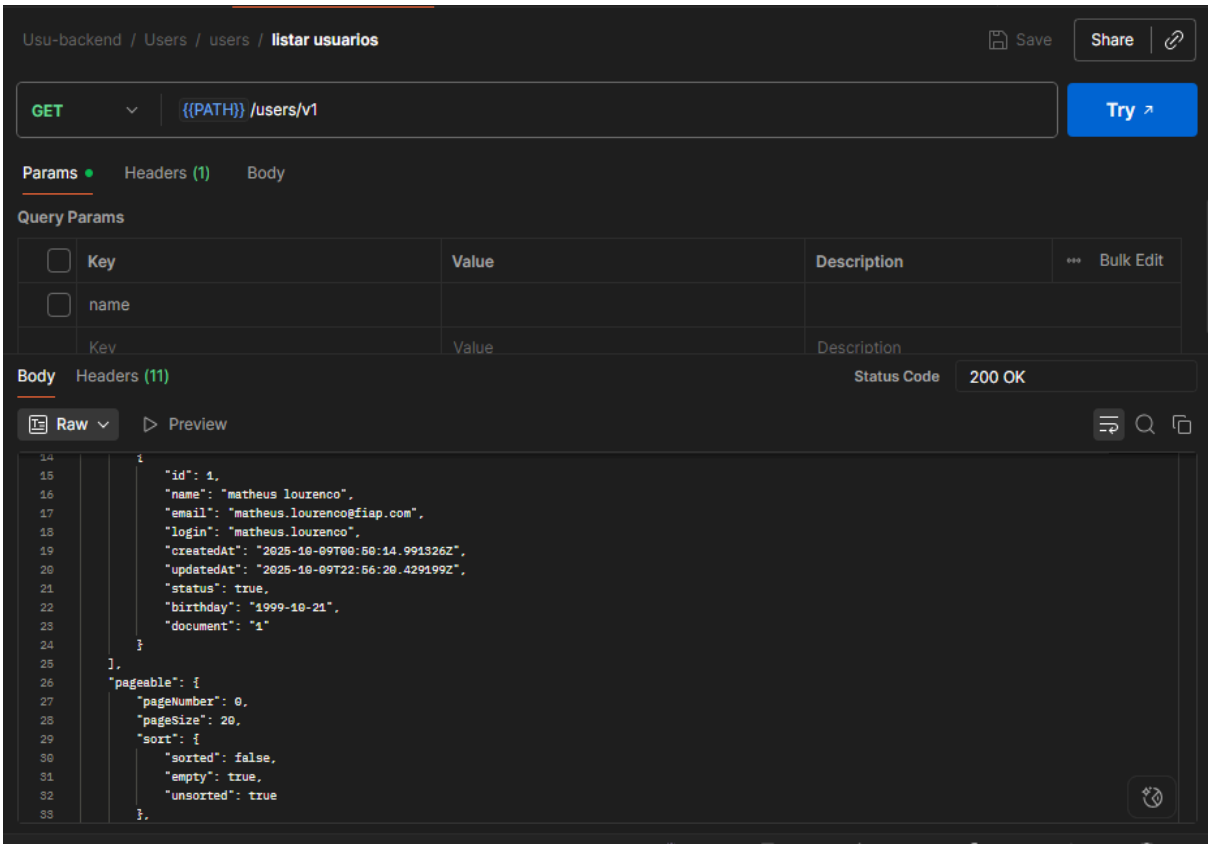


### 3.2.6 Apenas usuário admin pode adicionar dono de restaurante e adm

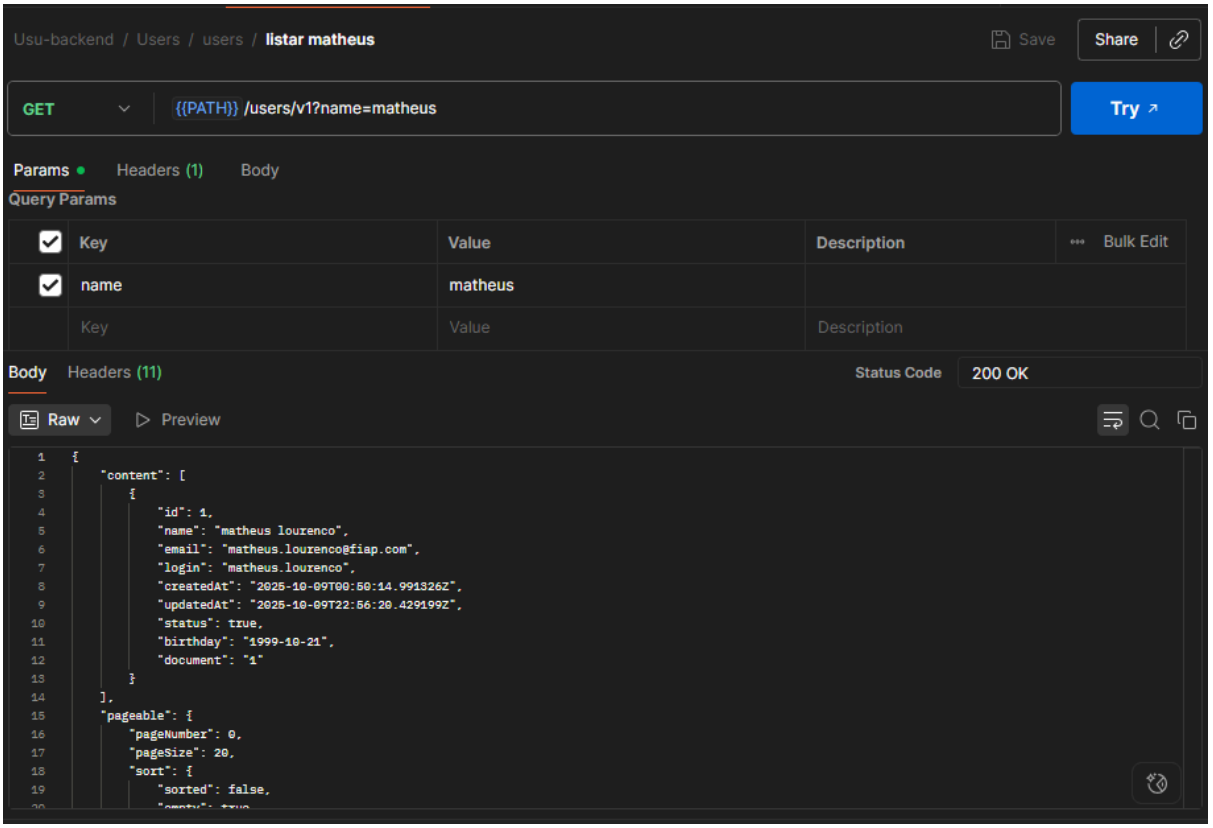



## 3.3. Listar usuários

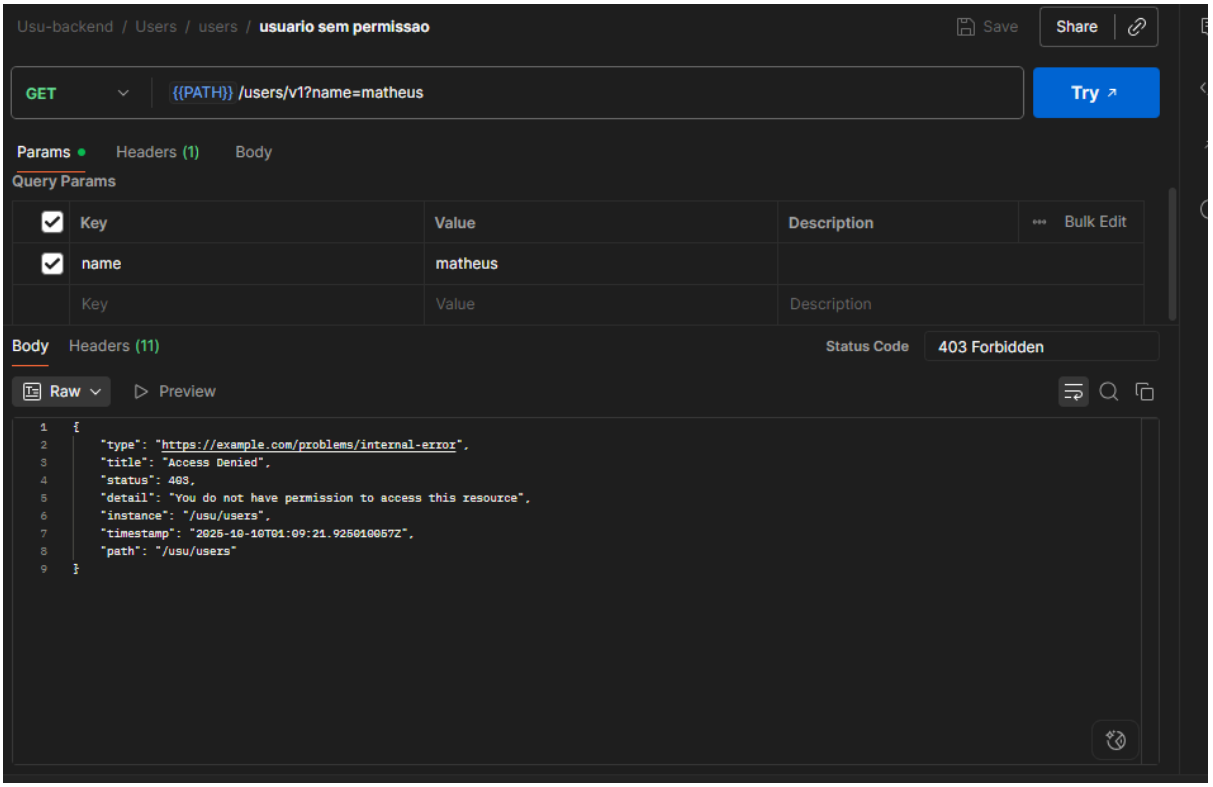
### 3.3.1 Listar todos usuários



### 3.3.2 Listar usuários por nome

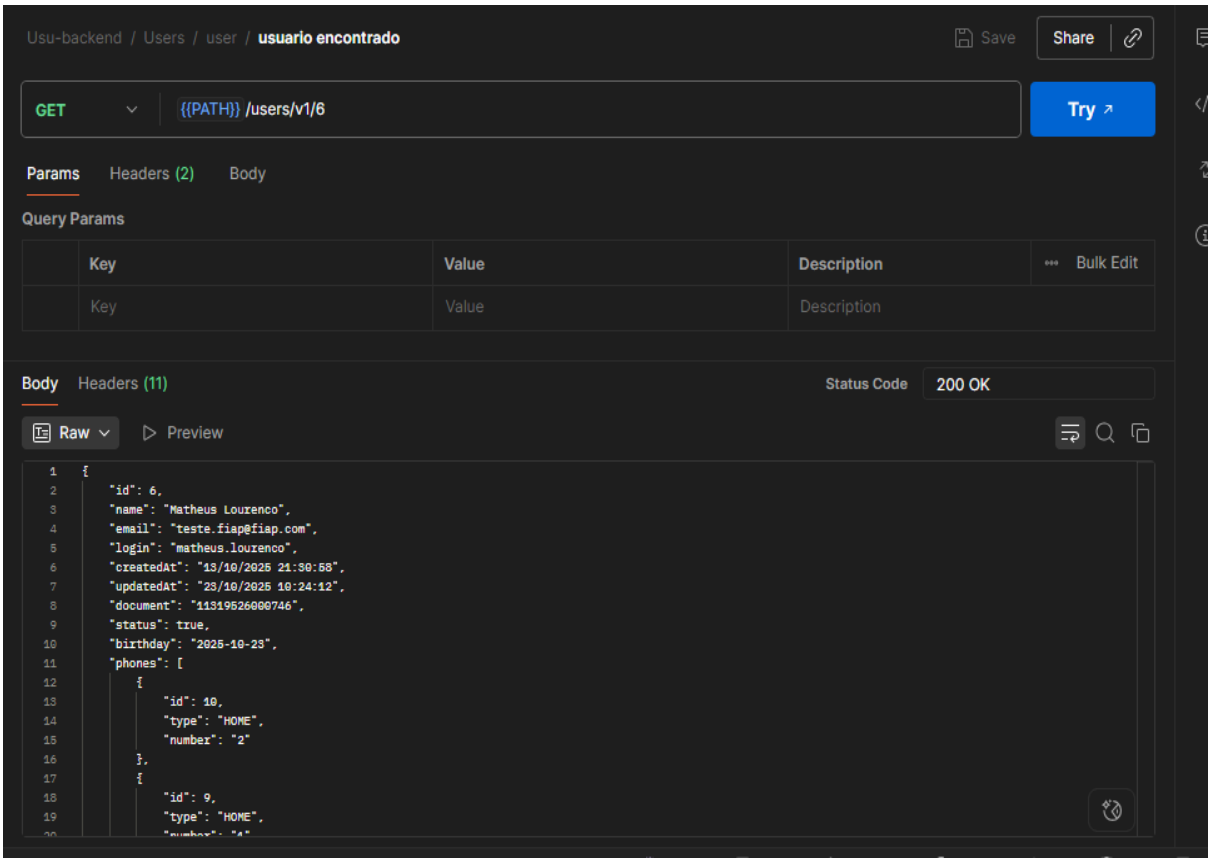



### 3.3.3 Usuário sem permissão

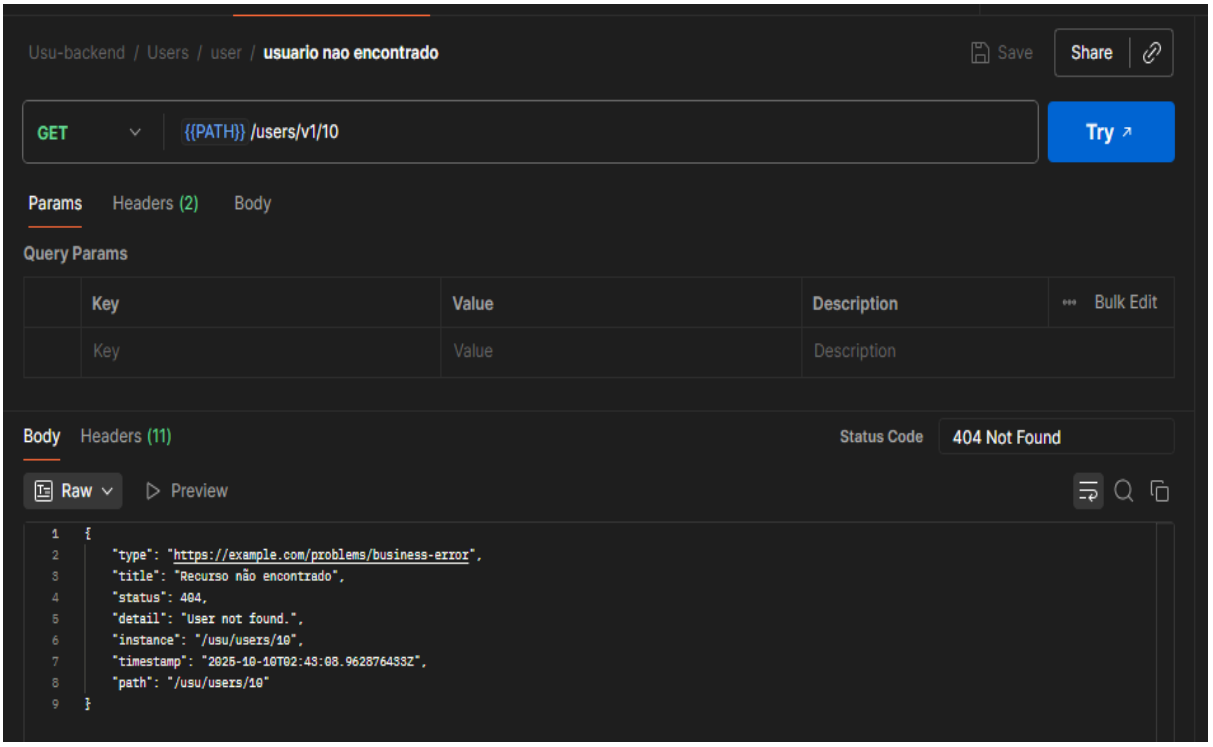


## 3.4. Consultar usuário por ID

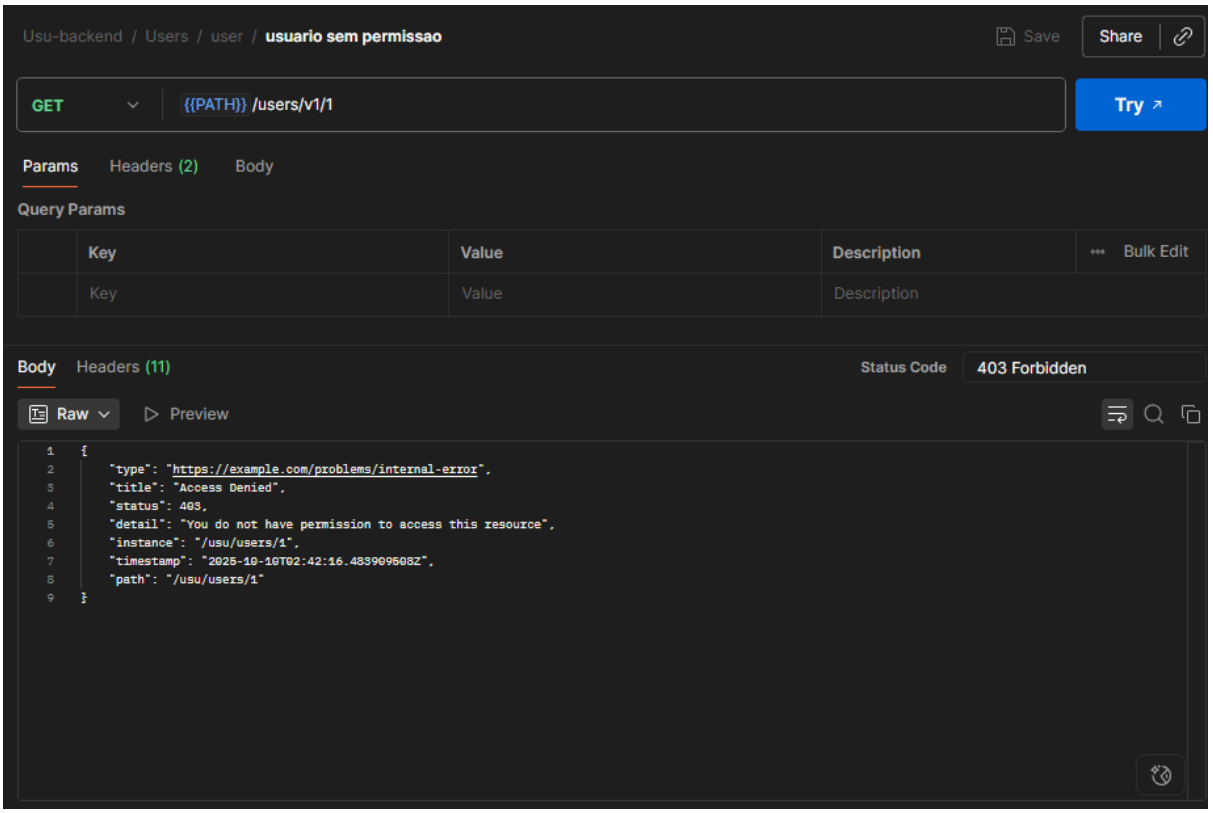
### 3.4.1 Usuário encontrado




### 3.4.2 Usuário não encontrado

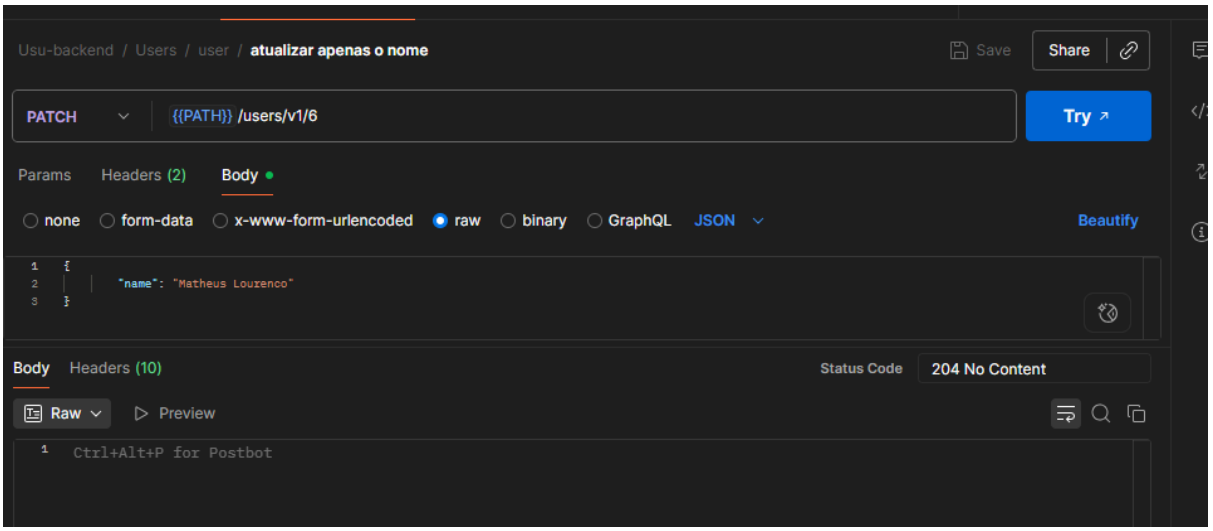


### 3.4.3 Usuário não tem permissão para acessar o recurso

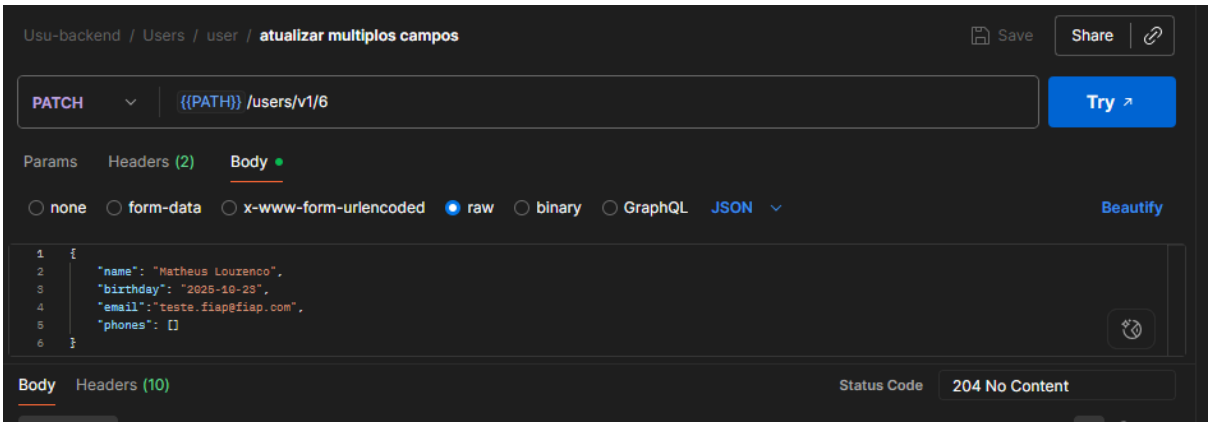



## 3.5. Atualizar usuário

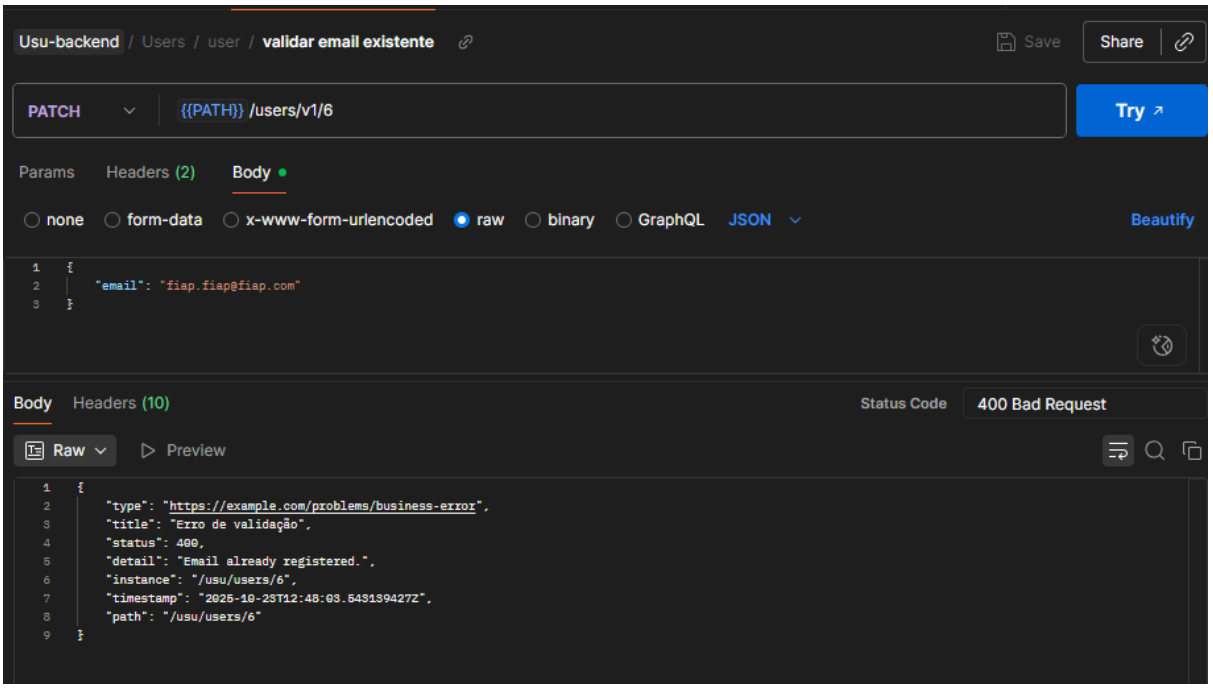
### 3.5.1 Atualizar apenas um campo



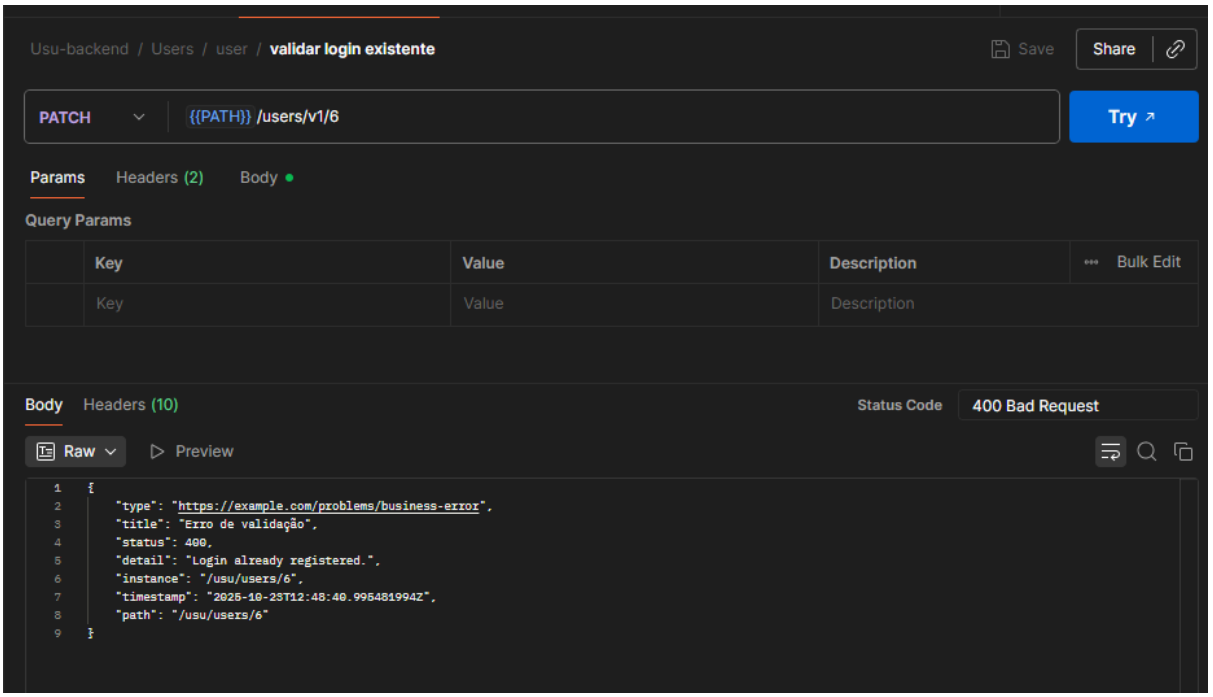
### 3.5.2 Atualizar múltiplos campos



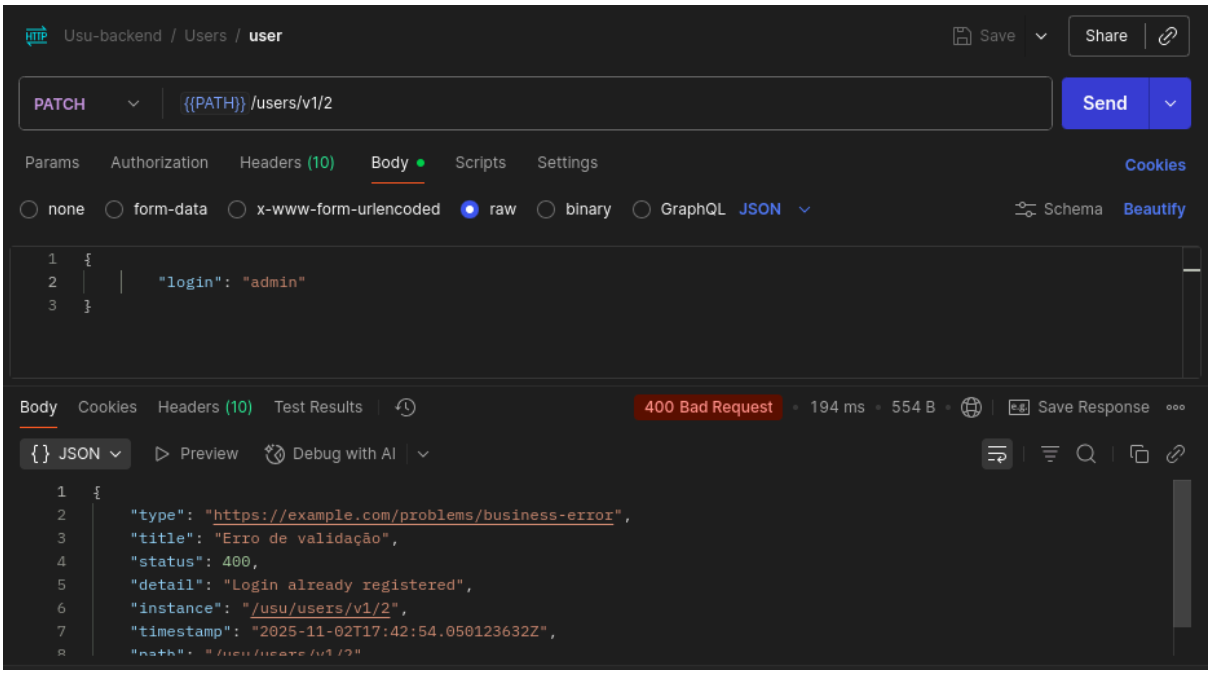

### 3.5.3 Atualizar email para um já existente



### 3.5.4 Atualizar login para um já existente

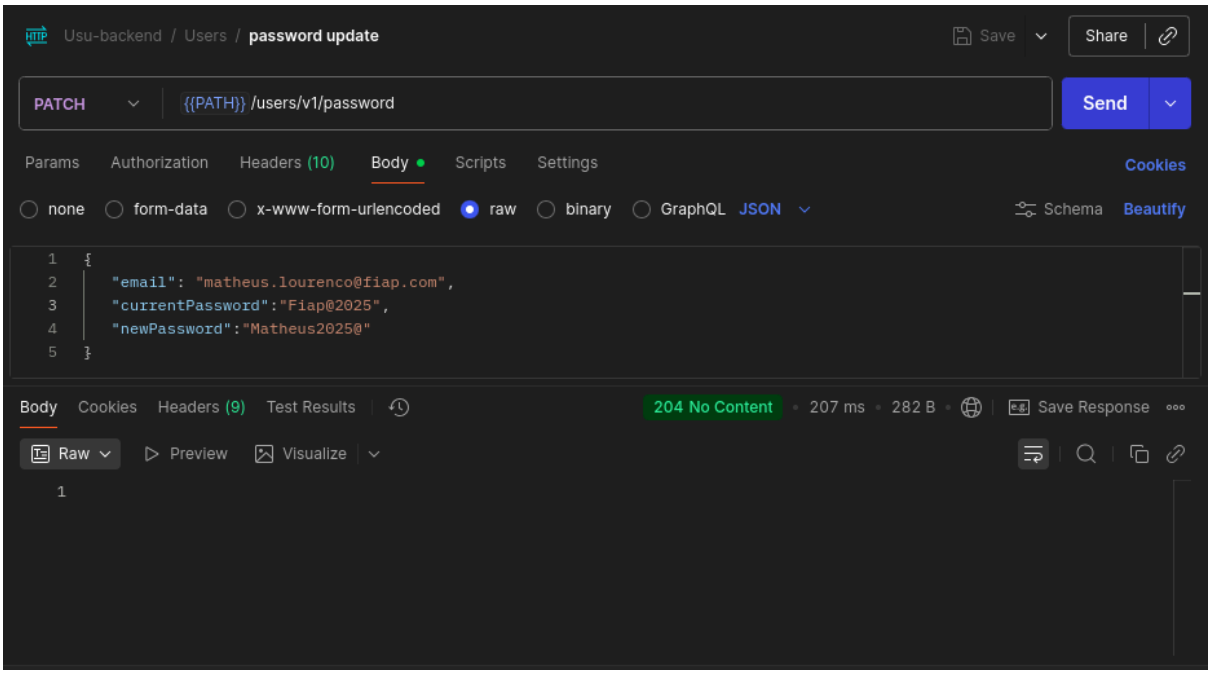



### 3.5.5 Atualizar login para um já existente

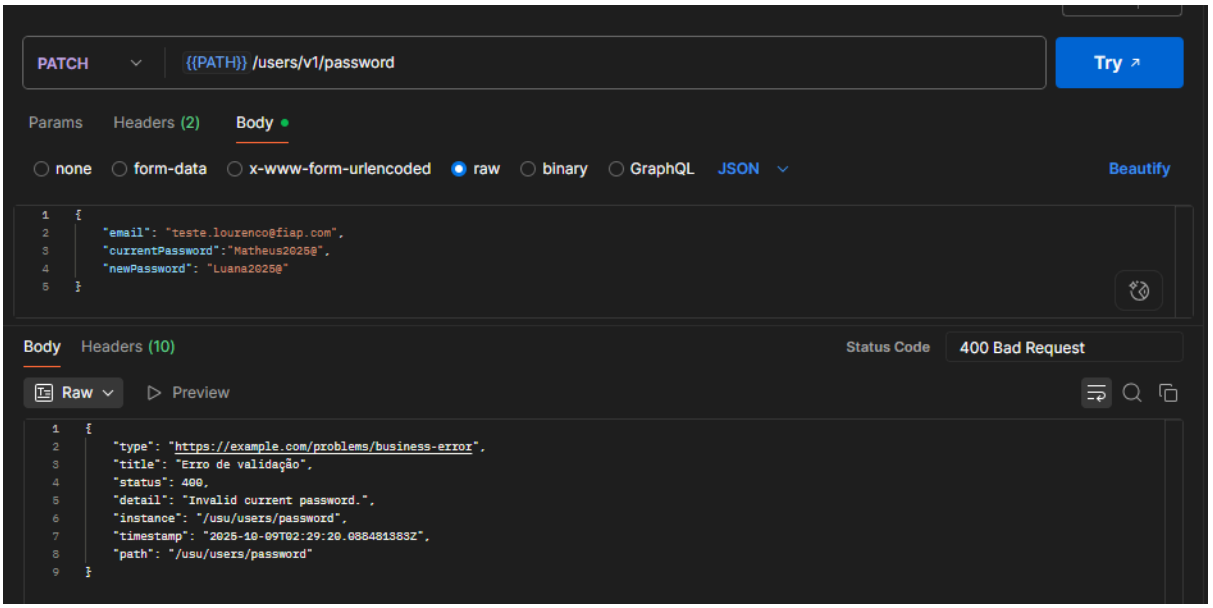


## 3.6. Atualizar senha

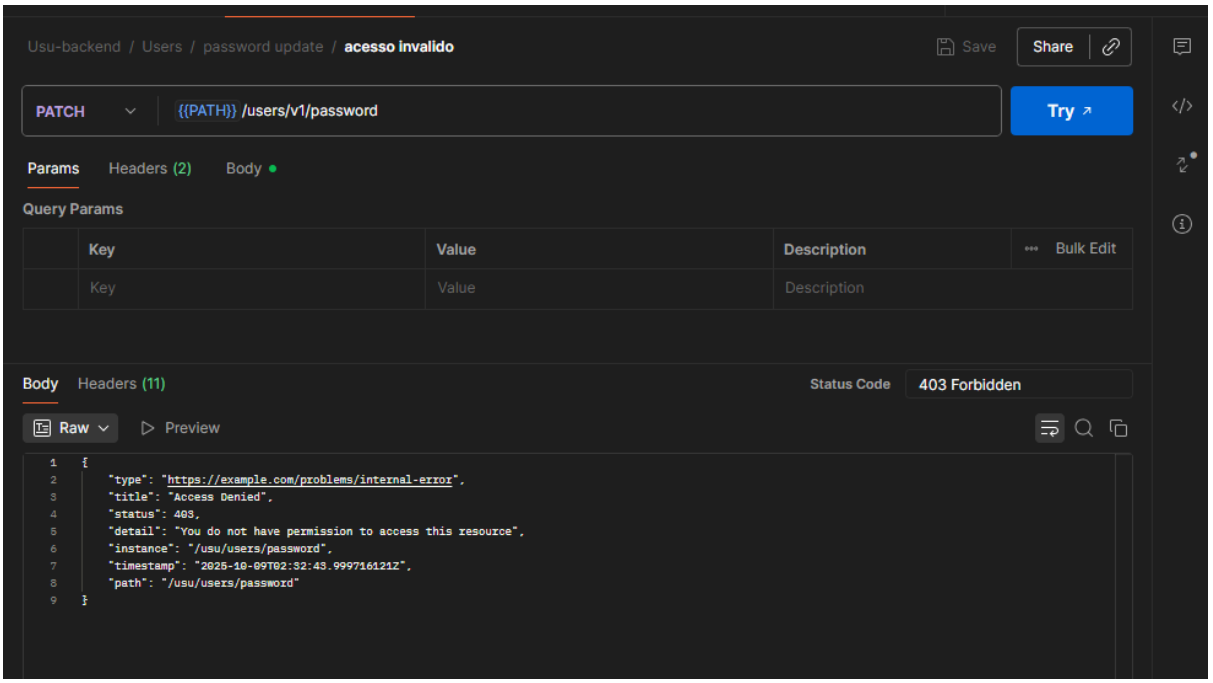
### 3.6.1 Atualizar senha



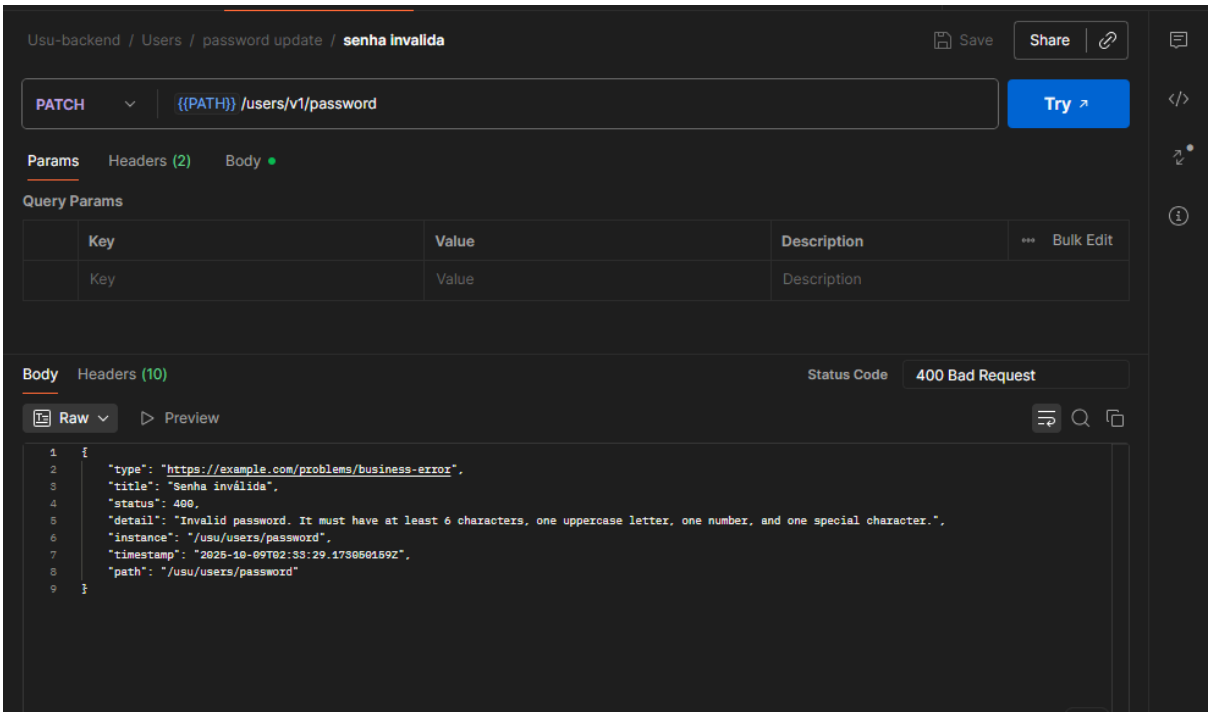

### 3.6.2 Senha atual inválida



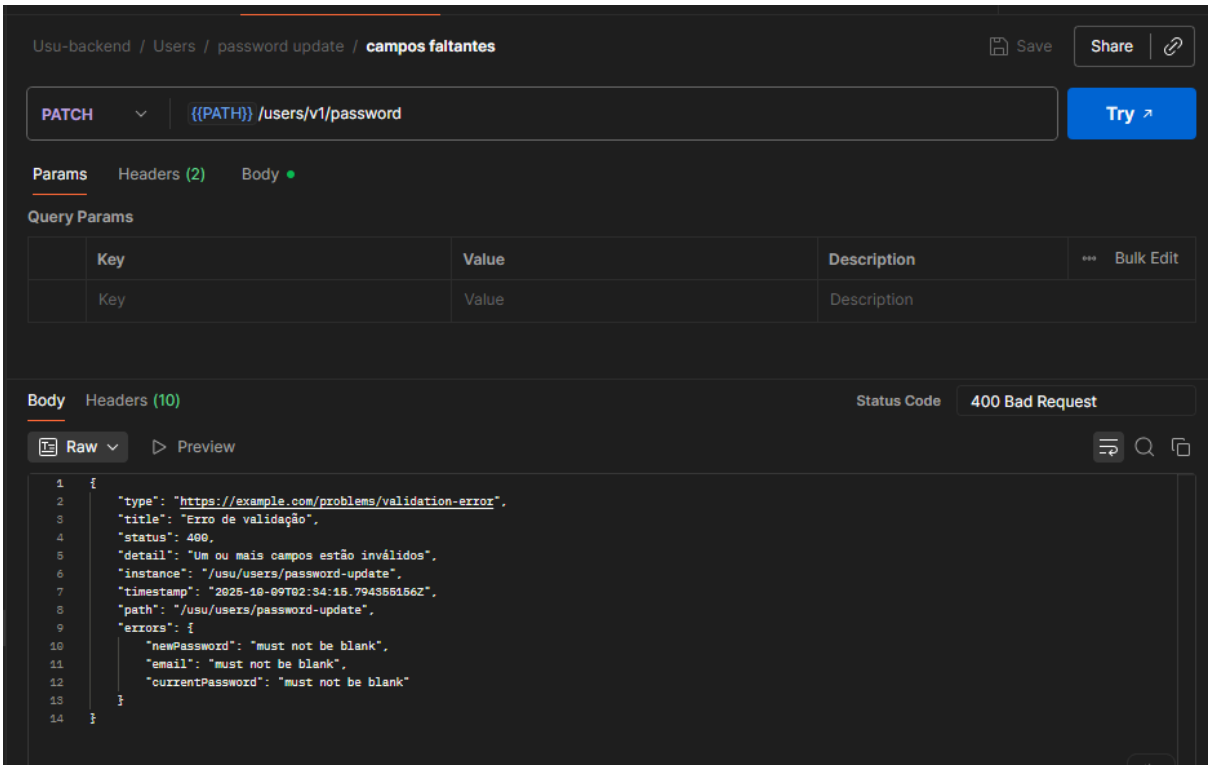
### 3.6.3 Acesso invalido



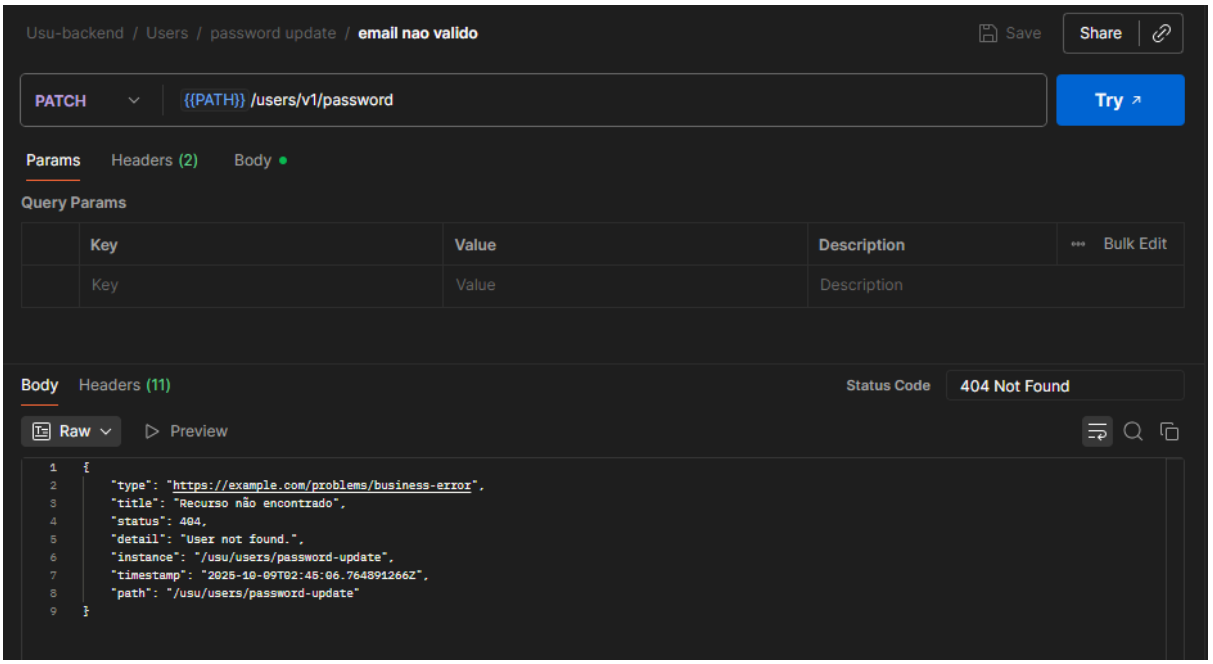

### 3.6.4 Senha inválida



### 3.6.5 Campos obrigatórios não informados

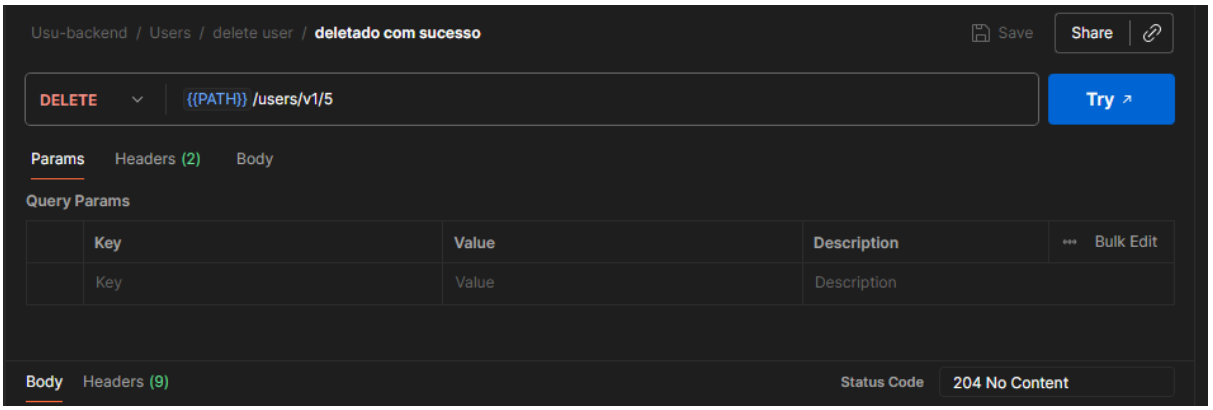



### 3.6.6 Email invalido

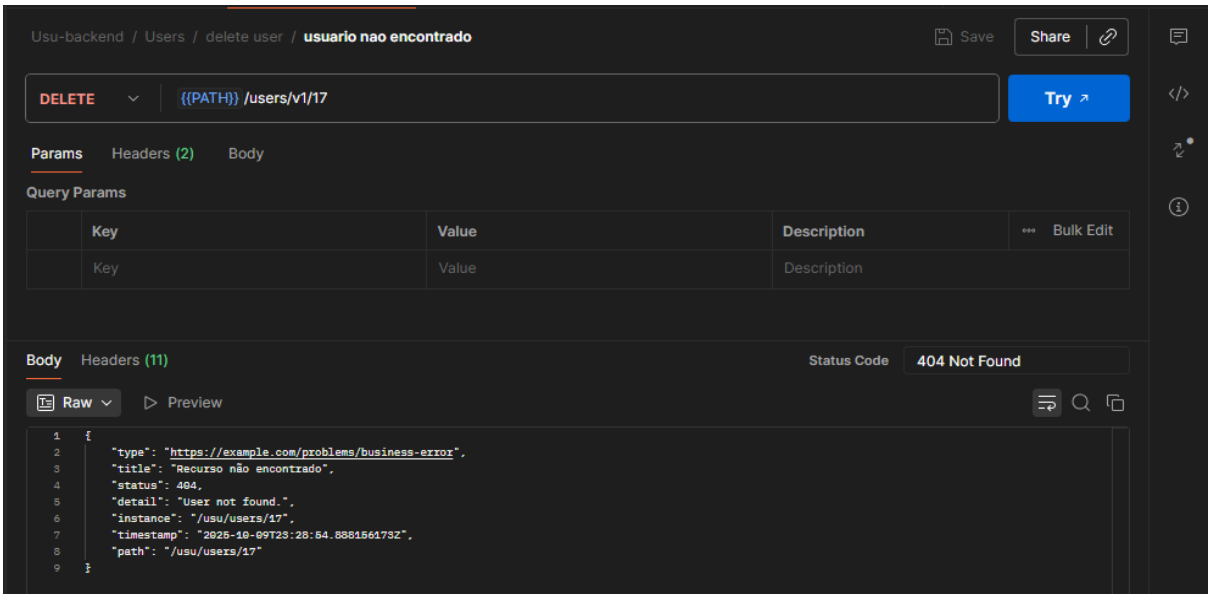


## 3.7. Excluir usuário

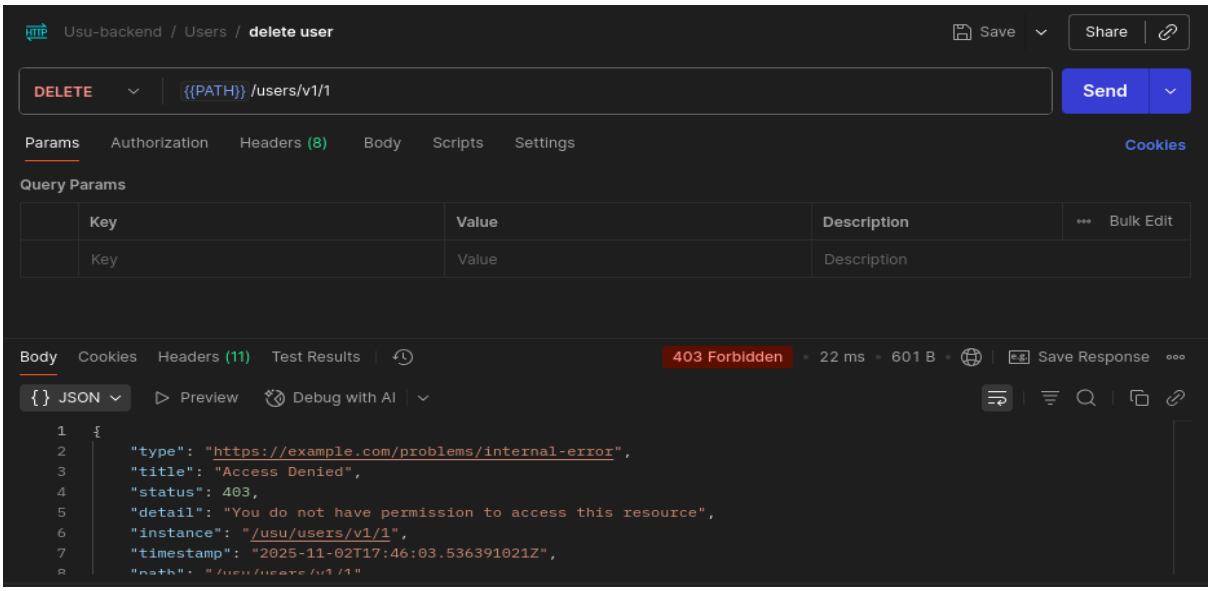
### 3.7.1 Deletado




### 3.7.2 Não encontrado




### 3.7.3 Sem permissão



## 4. Configuração do Projeto

### Configuração do Docker Compose

O arquivo docker-compose.yml define e orquestra múltiplos serviços Docker. Neste projeto são dois serviços, a aplicação em Java (usu_backend) e o banco de dados (postgres). Permitindo que ambos sejam executados e se comuniquem de forma integrada e automatizada.

Cada serviço representa um container que será iniciado com suas configurações específicas. Os serviços são configurados de formas independentes:

**Postgres:**

- Image: Usa a imagem oficial do postgres na versão 16, baseada em Alpine.
- Container_name: Nome do container para facilitar a identificação.
- Restart: Garante que o container será reiniciado automaticamente caso pare por algum motivo.
- Env_file: carrega o arquivo onde contém as variáveis de ambiente.
- Environment: Passa as variáveis para o banco, criando o banco e o usuário inicial.
- Ports: Faz o mapeamento entre a porta do host e a porta interna do container (5432).
- Volumes: utilizado para armazenar os dados do banco.
- Networks: Conecta na mesma rede Docker que o banco, permitindo que o backend acesse o postgres via hostname.

**Usu_backend:**

- Build:Constrói a imagem da aplicação Java usando o Dockerfile presente na raiz.
- Image: Nome da imagem resultante.
- Restart: Garante que o container será reiniciado automaticamente caso pare por algum motivo.
- env_file: Carrega o arquivo onde contém as variáveis de ambiente.
- Depends_on: Garante que o container do postgres seja iniciado antes do backend.
- ports: Expõe a aplicação.

**Instruções para execução local**

O projeto pode ser executado de duas formas distintas: localmente sem o uso do Docker ou utilizando containers via Docker Compose. Abaixo, são descritos os procedimentos necessários para cada abordagem.

**Execução Local (sem Docker)**

1. **Instalação da versão java**

   Caso o java 21 não esteja instalado na máquina, é necessário realizar sua instalação.

2. **Instalação do postgres**

   Caso o postgres não esteja instalado na máquina, é necessário realizar sua instalação.

   Uma alternativa seria a executar o banco de dados em um container Docker com o comando:

docker run --name my-postgres -e POSTGRES_PASSWORD=fiap -p 5432:5432 -d postgres:16-alpine

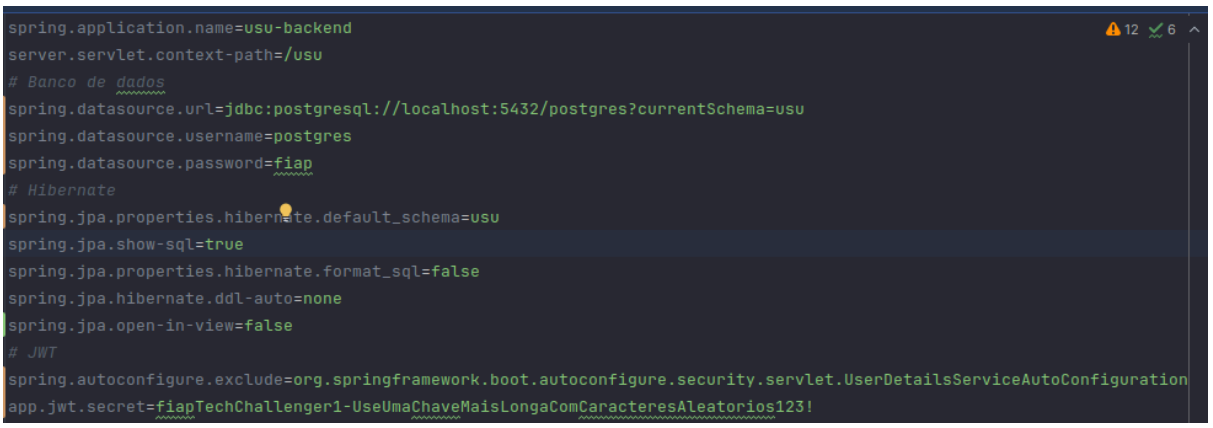
3. **Criação das tabelas e dados iniciais**

   Após a instalação do banco de dados, deve-se executar o script init.sql, localizado na pasta postgres do projeto.

   Esse script é responsável por criar as tabelas, índices e o usuário padrão da aplicação.

4. **Configuracao do arquivo application.properties**

Atualize as variáveis relacionadas à conexão com o banco e ao segredo utilizado para geração do token JWT.




5. **Execução da aplicação**

   Execute a aplicação na sua idle.

**Execução via Docker Compose**

1. **Instalação do docker e docker compose**
   Caso o docker não esteja instalado na máquina, é necessário realizar sua instalação.
2. **Abrir o terminal na pasta do projeto**
   Navegue até o diretório raiz do projeto onde se encontra o arquivo docker-compose.yml.
3. **Subir o Container da aplicação**
   Execute o seguinte comando para construir a imagem e iniciar os containers da aplicação e do banco de dados:
   docker compose up –build

Por se tratar de um projeto acadêmico, tanto a execução via Docker quanto a execução local utilizam as mesmas portas padrão: 8080 para a aplicação e 5432 para o banco de dados postgres.

## 5. Qualidade do Código

### 5.1. Injeção de dependências

O serviço utiliza injeção de dependências via construtor, evitando a criação direta de objetos dentro da classe. Isso torna o código mais modular, desacoplado e testável.

```
public UserService(
        UserRepository userRepository,
        UserPhoneService phoneService,
        UserAddressService userAddressService,
        UserTypeRepository userTypeRepository,
        SecurityValidator securityValidator
) {
```

### 5.2. Transações declarativas

Operações críticas no banco de dados estão anotadas com @Transactional, garantindo que todas as alterações sejam aplicadas de forma atômica, prevenindo inconsistências.

```
    @Transactional  1 usage  Matheus Lourenco
    public Long create(UserCreateDto dto) {
```

### 5.3. Controle de acesso e segurança

A anotação @PreAuthorize é utilizada para garantir que apenas usuários autorizados possam executar determinadas ações, aumentando a segurança do sistema.

```
@Transactional  1 usage  Matheus Lourenco
@PreAuthorize("#id == principal.id or hasAnyRole('ADMIN')")
public Long update(Long id, UserUpdateDto dto) {
```

### 5.4. Validações centralizadas

Regras de validação são encapsuladas em métodos específicos e em classes de validação (SecurityValidator), seguindo o princípio DRY (Don’t Repeat Yourself).

```
    private void validadePersistUser(UserCreateDto dto) {  1 usage  Matheus Lourenco
        // Apenas admin pode adicionar ADMIN e RESTAURANT_OWNER
```

### 5.5. Uso de logs

O código registra eventos importantes utilizando diferentes níveis de log (info, debug, warn), o que auxilia na depuração, monitoramento e auditoria.

```
        log.info("Iniciando criação do usuário: {}", dto.email());

        log.debug("Validação do usuário concluída para: {}", dto.email());
```

### 5.6. Mapeamento entre DTO e entidade

A separação entre DTOs e entidades é realizada por meio de Mappers, evitando o acoplamento entre a camada de apresentação e a persistência.

```
        User user = UserMapper.mapPersistenceEntity(dto);
        validadePersistUser(dto);
```

### 5.7. Tratamento de exceções específico

O código lança exceções customizadas (UserAlreadyExistsException, InvalidPasswordException, ResourceNotFoundException) para situações específicas, proporcionando mensagens de erro claras e tratamento adequado.

```
if (userRepository.existsByDocument(dto.document())) {
    throw new UserAlreadyExistsException(ValidationMessages.DOCUMENT_EXISTS);
}
```

### 5.8. Separação de responsabilidades

Serviços específicos, como UserPhoneService e UserAddressService, são responsáveis apenas por suas respectivas funcionalidades, seguindo o princípio Single Responsibility do SOLID.

```
        User saved = userRepository.save(user);
        log.info("Usuário salvo com ID: {}", saved.getId());

        userAddressService.persistUserAddresses(user, dto.addresses());
        phoneService.saveUserPhone(user, dto.phones());
```

## 6. Collections para Teste

### Link para a Collection do Postman

https://app.getpostman.com/join-team?invite_code=b6e9a6e7714d2866e63797164ad534ddca0687c9ec8ac06df6ee1155c6c2ad93&target_code=b4df7c400c8c96fc2b7b66906fb57e7c

### Descrição dos Testes Manuais

Para a validação manual dos endpoints do sistema, foi desenvolvida uma collection no Postman contendo todos os cenários necessários para testar as funcionalidades da API. Essa collection utiliza variáveis de ambiente para padronizar as requisições e facilitar a execução dos testes.

A variável PATH representa o caminho base da aplicação, definido como http://localhost:8080/usu. Já a variável token é preenchida automaticamente por um script interno após a autenticação bem-sucedida, sendo utilizada nos endpoints que exigem autenticação.

A autenticação é necessária para todos os endpoints, com exceção do POST /users/v1. O token de acesso é obtido através do endpoint /auth/v1/login, responsável por autenticar o usuário e retornar um token JWT.

Para simplificar o processo de validação, foi criado um usuário padrão no sistema com as seguintes credenciais:

- Login: admin
- E-mail: admin@fiap.com
- Senha: Fiap@2025

O token gerado por meio do login é automaticamente armazenado na variável token e incluído no cabeçalho das requisições no formato:

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

O endpoint POST /users/v1 requer autenticação apenas quando utilizado para criar usuários do tipo ADMIN ou RESTAURANT_OWNER.

Por padrão, as respostas da API são retornadas em inglês. Para exibir as mensagens de erro e respostas em português, deve-se adicionar o cabeçalho:

```
Accept-Language: pt_BR
```

Qualquer duvida sobre os endpoints é possível validar através do Swagger do projeto http://localhost:8080/usu/swagger-ui/index.html .

### 6.1. Login(POST /auth/v1/login)

O endpoint de autenticação tem como objetivo gerar um token JWT a partir das credenciais informadas.

Foram realizados testes para os seguintes cenários:

- Login com e-mail: autenticação utilizando o campo identifier como e-mail do usuário.

- Login com login de usuário: autenticação utilizando o campo identifier como login.

- Credenciais inválidas: tentativa de autenticação com dados incorretos, resultando em resposta 401 Unauthorized com mensagem padronizada.

Após um login bem-sucedido, o token é automaticamente armazenado na variável {{token}}, que passa a ser utilizada nos demais endpoints autenticados.

### 6.2. Criar Usuário (POST /users/v1)

O endpoint de criação de usuários permite cadastrar novos registros no sistema. Os testes realizados incluíram:

- Cadastro bem-sucedido: criação de um usuário completo, retornando 201 Created.

- Campos obrigatórios ausentes: validação de erro 400 Bad Request com retorno detalhado por campo inválido.

- Campos aninhados inválidos: verificação de inconsistências em listas de telefones e endereços.

- Regras de negócio: bloqueio de duplicidade para document, email e login.

- Controle de acesso: tentativa de criação de usuários do tipo ADMIN e RESTAURANT_OWNER por perfis não autorizados, retornando 403 Forbidden.

### 6.3. Listar Usuários (GET /users/v1)

Esse endpoint permite consultar usuários cadastrados de forma paginada. Foram validados os seguintes cenários:

- Listagem geral: retorno de todos os usuários com status 200 OK.
- Filtro por nome: uso do parâmetro name para pesquisa específica.
- Usuário sem permissão: bloqueio de acesso a perfis não autorizados, retornando 403 Forbidden.

### 6.4. Consultar Usuário por ID (GET /users/v1/{id})

Permite a obtenção dos detalhes de um usuário específico. Os testes realizados incluíram:

- Usuário encontrado: retorno das informações completas, incluindo endereços e telefones.
- Usuário inexistente: resposta 404 Not Found.
- Acesso restrito: tentativa de acesso por perfil não autorizado, resultando em 403 Forbidden.

### 6.5. Atualizar Usuário (PATCH /users/v1/{id})

Responsável pela atualização parcial ou total dos dados de um usuário existente. Foram testados os seguintes casos:

- Atualização de um único campo: alteração de um atributo isolado, como e-mail.
- Atualização múltipla: modificação de mais de um campo na mesma requisição.
- Conflito de dados: tentativa de atualização com e-mail ou login já cadastrados, retornando 400 Bad Request.

### 6.6. Atualizar Senha (PATCH /users/v1/password)

Permite que o usuário autenticado altere sua senha. Foram contemplados os seguintes testes:

- Atualização bem-sucedida: senha alterada com sucesso, respeitando os critérios de segurança.
- Senha atual inválida: erro 400 Bad Request.
- Campos ausentes: verificação de validação de obrigatoriedade.
- Formato inválido: tentativa de uso de senha fora do padrão exigido.
- Acesso sem token: erro 401 Unauthorized.

### 6.7. Excluir Usuário (DELETE /users/v1/{id})

Realiza a exclusão de um usuário existente. Os cenários testados foram:

- Exclusão com sucesso: resposta 204 No Content.
- Usuário inexistente: erro 404 Not Found.
- Usuário sem permissão: tentativa de exclusão por perfil não autorizado, resultando em 403 Forbidden.

## 7. Repositório do Código

### URL do Repositório

https://github.com/MLourenco111/usu-backend ou use a collection que esta salva na pasta resource do projeto.