

# Relatório Técnico - Sistema de Gestão de Restaurantes

---

**Autor:** Thiago Henrique Alves Ferreira

**Instituição:** FIAP - Faculdade de Informática e Administração Paulista

**Curso:** Arquitetura e Desenvolvimento Java

**Contato:** rm369442@fiap.com.br

**Repositório:** <https://github.com/thiagohaf/Food-Backend>

---

## 1. Visão Geral e Arquitetura

### Tecnologias Utilizadas

O projeto foi desenvolvido utilizando um conjunto moderno de tecnologias Java que garantem robustez, segurança e facilidade de manutenção. A seleção das tecnologias priorizou frameworks maduros e amplamente adotados no ecossistema Spring, permitindo aproveitar toda a comunidade e documentação disponível.

#### Framework e Core:

- **Spring Boot 4.0.1:** Framework principal escolhido por sua capacidade de reduzir significativamente a configuração inicial através de convenções sensatas e autoconfiguração. O Spring Boot facilita muito o desenvolvimento através da automação de configuração e gestão de dependências, o que reduz significativamente o tempo de desenvolvimento e minimiza erros de configuração.
- **Java 21:** Linguagem de programação utilizada, escolhida por ser uma versão LTS (Long Term Support) que oferece recursos modernos como Records, Pattern Matching e melhorias de performance.
- **Maven:** Gerenciador de dependências e ferramenta de build utilizada. O projeto inclui Maven Wrapper (`mvnw`) para garantir consistência entre diferentes ambientes de desenvolvimento.

#### Persistência de Dados:

- **Spring Data JPA:** Camada de persistência que abstrai o acesso a dados através de interfaces Repository, reduzindo código boilerplate e simplificando operações CRUD.
- **PostgreSQL 16:** Banco de dados relacional escolhido por sua robustez, conformidade com SQL, suporte a ACID e performance. A escolha por PostgreSQL permite garantir integridade referencial e transações críticas para operações como criação de usuário e alteração de senha.
- **Hibernate:** ORM (Object-Relational Mapping) utilizado pelo Spring Data JPA, que mapeia entidades Java para tabelas do banco de dados e gerencia o ciclo de vida das entidades.

#### Validação e Documentação:

- **Bean Validation (JSR 303/380):** Especificação Java para validação de dados através de anotações, integrada ao Spring Boot Starter Validation. Utilizada para validações de entrada nos DTOs.
- **SpringDoc OpenAPI 3 (v2.7.0):** Biblioteca utilizada para gerar documentação automática da API em formato OpenAPI 3.0, disponibilizando interface Swagger UI interativa.

- **Swagger Annotations (v2.2.22)**: Anotações para enriquecer a documentação da API com descrições detalhadas, exemplos e esquemas de resposta.

## Segurança:

- **jBCrypt (v0.4)**: Biblioteca para hashing de senhas utilizando o algoritmo BCrypt, que gera salt automático para cada senha, aumentando a segurança contra ataques de força bruta.
- **HttpSession**: Mecanismo de sessão HTTP utilizado na versão 1 (V1) para autenticação stateful baseada em sessão.
- **Spring Security**: Framework de segurança do ecossistema Spring utilizado na versão 2 (V2) para gerenciar autenticação e autorização.
- **JWT (JSON Web Tokens) - jjwt (v0.12.5)**: Biblioteca utilizada para implementar autenticação stateless através de tokens JWT na versão 2 (V2). A versão 0.12.5 foi escolhida por ser compatível com Java 21 e Spring Boot 4.0.1.

## Utilitários:

- **Lombok**: Biblioteca que reduz código boilerplate através de anotações processadas em tempo de compilação (ex: `@RequiredArgsConstructor`, `@Getter`, `@Setter`), mantendo o código mais limpo e legível.

## Testes e Qualidade:

- **JUnit 5**: Framework de testes padrão da plataforma Java, incluído no Spring Boot Starter Test. Utilizado para escrever testes unitários e de integração.
- **JaCoCo (v0.8.11)**: Ferramenta de análise de cobertura de código que instrumenta o código durante a execução dos testes e gera relatórios detalhados. Configurado com meta mínima de 80% de cobertura de linhas.
- **Maven Surefire Plugin**: Plugin Maven que executa testes durante o build, integrado com o JaCoCo para coletar dados de cobertura.

## Containerização:

- **Docker**: Tecnologia de containerização utilizada para empacotar a aplicação e suas dependências em uma imagem isolada.
- **Docker Compose**: Ferramenta para orquestrar múltiplos containers (aplicação Spring Boot e PostgreSQL), facilitando a execução local e garantindo consistência entre ambientes.

A escolha dessas tecnologias modernas como Spring Boot 4.0.1, Java 21, PostgreSQL e Docker garante que a aplicação esteja preparada para escalabilidade e manutenção futura, aproveitando as melhores práticas e padrões estabelecidos na comunidade Java.

A arquitetura adotada segue um padrão em camadas (Layered Architecture) com separação clara de responsabilidades. Optei por isolar as regras de negócio na camada de serviços (`UserService`), evitando que os controllers (`UserController`, `UserControllerV2`) fiquem poluídos com lógica de domínio. Os controllers focam exclusivamente em receber requisições HTTP, delegar processamento aos services e formatar respostas.

Entre a camada de apresentação e a de domínio, implementei uma camada de DTOs (`UserRequest`, `UserResponse`, `UserUpdateRequest`, `PasswordChangeRequest`) que protege as entidades de domínio

(User, Address) de exposição direta na API. A transformação entre DTOs e entidades é realizada pelo UserMapper, componente que centraliza essa lógica de conversão. Essa decisão de design evita acoplamento entre o contrato da API e a estrutura interna do banco de dados, facilitando futuras evoluções sem quebrar contratos já estabelecidos.

Para padronização de respostas de erro, implementei o tratamento de exceções através do GlobalExceptionHandler, que converte todas as exceções em objetos ProblemDetail conforme o RFC 7807. Isso garante que qualquer erro retornado pela API siga um formato consistente com os campos type, title, status, detail e propriedades customizadas. Exceções de domínio como DomainValidationException e ResourceNotFoundException são capturadas e transformadas em respostas HTTP apropriadas.

A aplicação suporta duas versões de autenticação coexistentes. A versão 1 (V1) utiliza autenticação stateful baseada em HttpSession, implementada manualmente através do AuthInterceptor que intercepta requisições e verifica a existência de sessão válida. A versão 2 (V2) migra para autenticação stateless usando JWT (JSON Web Tokens) gerenciada pelo Spring Security através da classe SecurityConfig e do filtro JwtAuthenticationFilter. Essa coexistência permite migração gradual sem impactar clientes existentes.

## Estrutura do Projeto

A organização do código segue uma estrutura modular que reflete a arquitetura em camadas adotada. O projeto está organizado no diretório food-backend/ e segue o padrão Maven de estrutura de diretórios. A seguir, apresento a estrutura completa do projeto:

```

food-backend/
└── src/
    └── main/
        └── java/com/thiagoferreira/food_backend/
            ├── Application.java                                # Classe principal da
            │
            └── aplicação
                └── (HttpSession)
                    ├── controllers/
                    │   ├── AuthController.java                      # Controladores REST
                    │   └── AuthControllerV2.java                     # Autenticação V1
                    └── (JWT)
                        ├── UserController.java                  # Autenticação V2
                        └── UserControllerV2.java
                            # Gerenciamento de
                            # Gerenciamento de
                            # Interceptadores HTTP
                            # Interceptor de
            └── autenticação V1
                └── domain/
                    └── dto/
                        # Data Transfer
            └── Objects
                └── AddressDTO.java
                └── LoginRequest.java
                └── PasswordChangeRequest.java
                └── ProblemDetailDTO.java

```

```

    |
    |
    |   ├── TokenResponse.java
    |   ├── UserRequest.java
    |   ├── UserResponse.java
    |   └── UserUpdateRequest.java
    |
    |   └── entities/          # Entidades JPA
    |       ├── Address.java
    |       └── User.java
    |
    |   └── enums/            # Enumeradores
    |       ├── ErrorMessages.java
    |       └── UserType.java
    |
    |   └── exceptions/       # Tratamento de
    |
    |   ├── exceções
    |   |   ├── DomainValidationException.java
    |   |   ├── GlobalExceptionHandler.java
    |   |   ├── ResourceNotFoundException.java
    |   |   └── UnauthorizedException.java
    |   |
    |   |   └── infraestructure/
    |   |       ├── config/
    |   |           └── OpenApiConfig.java
    |   |
    |   |       └── WebConfig.java      # Configuração web
    |
    |   └── Swagger/OpenAPI
    |       |   └── (interceptors)
    |       |       ├── repositories/
    |       |           └── UserRepository.java      # Repositórios JPA
    |       |       └── security/                  # Configurações de
    |
    |   └── segurança V2
    |       |   ├── JwtAuthenticationFilter.java
    |       |   ├── JwtService.java
    |       |   ├── SecurityConfig.java
    |       |   ├── SecurityProblemDetailAccessDeniedHandler.java
    |       |   ├── SecurityProblemDetailEntryPoint.java
    |       |   └── UserDetailsServiceImpl.java
    |       |
    |       |   └── mappers/
    |           └── UserMapper.java      # Mappers DTO/Entity
    |       |
    |       |   └── services/
    |           └── UserService.java    # Lógica de negócio
    |
    |       └── resources/
    |           └── application.properties      # Configurações da
    |
    |   └── aplicação
    |       └── test/                   # Testes automatizados
    |           └── java/com/thiagoferreira/food_backend/
    |               ├── controllers/
    |               ├── exceptions/
    |               ├── infraestructure/
    |               ├── interceptors/
    |               ├── mappers/
    |               └── services/
    |
    |   └── docker-compose.yml        # Configuração Docker
    |
    Compose
    └── Dockerfile                 # Imagem Docker
    └── pom.xml                     # Configuração Maven
    └── README.md                   # Documentação do
    projeto

```

Esta estrutura organiza o código seguindo os princípios de separação de responsabilidades e modularidade:

- **controllers/**: Contém os controladores REST separados por versão (V1 e V2), cada um implementando seus respectivos endpoints e tipos de autenticação
- **domain/**: Agrupa as classes de domínio em subpacotes (**dto**, **entities**, **enums**), mantendo as entidades JPA separadas dos DTOs que expõem a API
- **services/**: Contém a lógica de negócio isolada dos controllers, facilitando reutilização e testes
- **infrastructure/**: Separa aspectos técnicos da infraestrutura (**config**, **repositories**, **security**) das regras de negócio, seguindo o princípio de Inversão de Dependências
- **exceptions/**: Centraliza o tratamento de exceções e define exceções customizadas de domínio
- **mappers/**: Centraliza a lógica de transformação entre DTOs e entidades, evitando acoplamento
- **interceptors/**: Contém interceptadores HTTP específicos para a autenticação V1

A separação entre V1 e V2 está presente principalmente nos controllers, mantendo o restante do código (services, repositories, entities) compartilhado entre as duas versões. Isso demonstra que a coexistência de duas versões de autenticação não resultou em duplicação de código de negócio, apenas na camada de apresentação.

## 2. Modelagem de Dados e Entidades

O modelo de dados foi projetado com foco nas necessidades específicas de um sistema de gestão de restaurantes. A entidade principal **User**, mapeada para a tabela **tb\_users**, concentra as informações de usuários do sistema. O campo **email** foi definido como único (`@Column(unique = true)`) na entidade para evitar duplicidade de cadastro a nível de banco, complementando a validação de negócio no `UserService.createUser()` que verifica a existência prévia através do `UserRepository.existsByEmail()`. A mesma estratégia foi aplicada ao campo **login**, garantindo que cada usuário tenha um identificador único.

A entidade **Address** foi implementada como `@Embeddable`, permitindo que seja incorporada diretamente na tabela **tb\_users** sem necessidade de uma tabela separada. Essa decisão simplifica a persistência e consultas, já que endereço não possui identidade própria e está sempre associado a um usuário específico. Os campos **street**, **number**, **city** e **zipCode** compõem a estrutura básica necessária para entrega e localização.

O enum **UserType** define dois tipos de usuário: **OWNER** (proprietário do restaurante) e **CUSTOMER** (cliente). Essa distinção permite futuras expansões de regras de negócio específicas por perfil sem necessidade de refatoração estrutural.

Para auditoria temporal, a entidade **User** utiliza JPA Auditing (`@EnableJpaAuditing` na classe **Application**) para preencher automaticamente os campos **createdAt** e **lastUpdated**. O **createdAt** é marcado como `updatable = false` para garantir que nunca seja alterado após a criação, enquanto **lastUpdated** é atualizado automaticamente pelo Hibernate a cada modificação.

Escolhemos PostgreSQL como banco de dados relacional porque o modelo de dados é altamente estruturado, com relacionamentos claros e necessidade de garantias ACID. A integridade referencial e transações são fundamentais para operações críticas como criação de usuário (onde validamos email/login únicos) e alteração de senha (onde validamos senha atual antes de atualizar). O Hibernate foi configurado

com `spring.jpa.hibernate.ddl-auto=update` para desenvolvimento, permitindo evolução do schema automaticamente baseado nas entidades, enquanto mantemos `spring.jpa.open-in-view=false` para evitar problemas de lazy loading em contextos fora de transação.

Durante o desenvolvimento, utilizei o DBeaver para visualizar e validar a estrutura do banco de dados PostgreSQL. A figura abaixo mostra a estrutura da tabela `tb_users` criada pelo Hibernate, onde podemos observar todos os campos da entidade `User`, incluindo os campos de endereço incorporados e os campos de auditoria (`created_at` e `last_updated`).

## Estrutura do Banco de Dados PostgreSQL - DBeaver

The screenshot shows the DBeaver interface with the following details:

- Database Navigator:** Shows the connection to `localhost` and the database `food_db`.
- Table Structure:** The `tb_users` table is selected. It has the following columns:
  - `id` (int8)
  - `city` (varchar(255))
  - `number` (varchar(255))
  - `street` (varchar(255))
  - `zip_code` (varchar(255))
  - `created_at` (timestamp(6))
  - `email` (varchar(255))
  - `last_updated` (timestamp(6))
  - `login` (varchar(255))
  - `name` (varchar(255))
  - `password` (varchar(255))
  - `type` (varchar(255))
- Foreign Keys:** Shows a constraint named `tb_users_pkey` and another named `tb_users_type_check`.
- Data Grid:** Displays the following data for the `tb_users` table:

	AZ name	123 id	AZ city	AZ number	AZ street	AZ zip_cc
1	João Ferreira	16	São Paulo	123	Rua Teste	01234-56
2	João Silva	21	São Paulo	123	Rua Teste	01234-56
3	Maria Santos Atualizada	25	Rio de Janeiro	999	Rua Nova V2	20000-01
4	Tania Ferreira Atualizado	15	Rio de Janeiro	456	Rua Nova	20000-01
5	Thiago Ferreira Atualizado	24	Rio de Janeiro	456	Rua Nova	20000-01

## 3. API e Endpoints

A API está organizada em duas versões principais que coexistem na mesma aplicação. A versão 1 (`/v1/**`) utiliza autenticação baseada em sessão HTTP, enquanto a versão 2 (`/v2/**`) migra para JWT através do Spring Security.

O fluxo principal de gerenciamento de usuários na V1 inicia com o cadastro público através de `POST /v1/users`, que não requer autenticação. Após cadastro, o usuário autentica-se via `POST /auth/login`, que cria uma sessão HTTP e armazena o ID do usuário no atributo `USER_ID`. Com sessão válida, o usuário pode listar todos os usuários (`GET /v1/users`), buscar por ID (`GET /v1/users/{id}`), realizar buscas por nome (`GET /v1/users/search/name?name={nome}`), por login (`GET /v1/users/search/login?login={login}`) ou por email (`GET /v1/users/search/email?email={email}`). A busca por nome utiliza o método `findByNameContainingIgnoreCaseOrderByNameAsc` do repositório, que realiza correspondência parcial case-insensitive e ordena alfabeticamente.

**Ponto crítico de segurança:** Separei explicitamente a atualização de dados cadastrais da alteração de senha em rotas distintas. O endpoint `PUT /v1/users/{id}` (e `PUT /v2/users/{id}` na V2) aceita apenas `UserUpdateRequest` contendo `name` e `address`, sendo que o método `updateEntityFromDto`

do `UserMapper` intencionalmente não processa campos de senha. A alteração de senha é realizada exclusivamente através de `PATCH /v1/users/{id}/password` (e `PATCH /v2/users/{id}/password` na V2), que requer `PasswordChangeRequest` com `currentPassword` e `newPassword`. O `UserService.changePassword()` valida a senha atual usando `BCrypt.checkpw()` antes de aplicar a nova senha, garantindo que apenas o próprio usuário autenticado possa alterar sua senha com conhecimento da senha atual. Essa separação evita que campos de senha sejam acidentalmente expostos em payloads de atualização cadastral e permite políticas de segurança distintas para cada operação.

Na versão 2, o fluxo de autenticação utiliza JWT. O endpoint público `POST /v2/auth/login` autentica o usuário através do `UserService.authenticate()` e retorna um token JWT gerado pelo `JwtService.generateToken()`. O token é encapsulado em um `TokenResponse` no formato `{"token": "...", "type": "Bearer"}`. Para acessar endpoints protegidos da V2, o cliente deve incluir o header `Authorization: Bearer {token}`. O `JwtAuthenticationFilter` intercepta requisições para `/v2/**`, extrai e valida o token antes de permitir o acesso aos controllers.

O endpoint `POST /v2/auth/logout` permite que o usuário faça logout. Como JWT tokens são stateless por natureza, o logout não invalida o token no servidor (diferente do logout V1 que invalida a sessão HTTP). O endpoint simplesmente retorna 200 OK, sinalizando ao cliente que ele deve descartar o token localmente. Esta é uma abordagem comum em sistemas stateless, onde o token permanece válido até sua expiração natural (configurável através da propriedade `jwt.expiration`). Em produção, para maior segurança, pode-se implementar uma blacklist de tokens invalidados, armazenando os tokens em cache (Redis, por exemplo) e verificando sua presença durante a validação no `JwtAuthenticationFilter`.

A exclusão de usuários é realizada através de `DELETE /v1/users/{id}` ou `DELETE /v2/users/{id}`, que invoca `UserService.deleteUser()` que verifica existência antes de remover, lançando `ResourceNotFoundException` se o ID não existir.

## 4. Documentação e Testabilidade

A API segue a especificação OpenAPI 3.0 para documentação automática, configurada através da classe `OpenApiConfig`. Utilizei o SpringDoc OpenAPI 2.7.0 que gera automaticamente a documentação interativa acessível em `/swagger-ui.html`. A configuração separa as APIs em dois grupos: "v1" (para endpoints `/v1/**` e `/auth/**`) e "v2" (para endpoints `/v2/**`), cada um com suas próprias descrições contextualizando o tipo de autenticação utilizado.

Os controllers são anotados com `@Tag` para organização no Swagger, e cada endpoint possui `@Operation` com `summary` e `description` detalhados. As respostas são documentadas através de `@ApiResponse`s que especificam códigos HTTP e esquemas de resposta, incluindo os casos de erro que retornam `ProblemDetailDTO`. Para os endpoints V2, adicionei `@SecurityRequirement(name = "bearerAuth")` que integra o botão de autenticação no Swagger UI, permitindo que desenvolvedores testem endpoints protegidos diretamente na interface.

### Documentação Swagger - Versão 1

A documentação da API versão 1 está organizada no Swagger UI, mostrando todos os endpoints disponíveis com autenticação baseada em sessão HTTP. As imagens abaixo mostram algumas das principais visualizações da documentação:

#### Swagger V1 - Visão geral

**API Core - V1 (Legado)** v1.0 OAS 3.0

[/api-docs/v1](#)

**API RESTful para o sistema de gestão de restaurantes Food App - Versão 1.**

**Autenticação**

Esta versão utiliza autenticação stateful baseada em HttpSession.  
Endpoints: [/v1/\\*\\*](#) e [/auth/\\*\\*](#)  
Como usar:  
1. Faça login através do endpoint [POST /auth/login](#)  
2. A sessão é mantida automaticamente através de cookies (JSESSIONID)  
3. Não é necessário enviar tokens em requisições subsequentes

**Observações Importantes**

- Endpoints públicos: O endpoint [POST /v1/users](#) (cadastro de usuário) é público e **não requer autenticação**
- Autenticação obrigatória: Para acessar os demais endpoints protegidos, é necessário autenticar-se previamente

**Nota sobre Versão**

Esta é a versão legada da API. Recomendamos a migração para a API Core - V2 que utiliza autenticação JWT e oferece melhor escalabilidade.  
Contact Thiago Ferreira  
MIT

## Swagger V1 - Endpoints de usuários

Servers

[http://localhost:8080 - Generated server url](http://localhost:8080) [Authorize](#)

**Authentication** Authentication APIs - Stateless session-based authentication

[POST /auth/logout](#) Logout user and invalidate session

[POST /auth/login](#) Authenticate user and create session

**Users** User management APIs. Most endpoints require authentication (valid session). Only POST /v1/users (create user) is public and does not require authentication.

[DELETE /v1/users/{id}](#) Delete a user

[GET /v1/users/{id}](#) Search users by id

[GET /v1/users](#) Search users

[GET /v1/users/search/name](#) Search users by name

[GET /v1/users/search/login](#) Search users by login

## Swagger V1 - Detalhes do endpoint de criação

[POST /auth/login](#) Authenticate user and create session

Authenticates a user with login and password. On success, creates an HTTP session and stores the user ID. The session is maintained via cookies (JSESSIONID). This endpoint is public and does not require authentication.

**Parameters**

No parameters

**Request body** required

[application/json](#)

**Example Value** Schema

```
{
  "login": "string",
  "password": "string"
}
```

**Responses**

Code	Description	Links
200	Login successful - session created	No links
400	Validation error	No links

**Media type**

[\\*/\\*](#)

**Example Value** Schema

## Swagger V1 - Endpoints de autenticação

The screenshot shows the Swagger UI interface for a Spring Boot application. It displays four error responses:

- 200 Login successful - session created**:  
Validation error  
Media type:   
Example Value | Schema
- 400 Validation error**  
Media type:   
Example Value | Schema
- 404 User not found or invalid credentials**  
Media type:   
Example Value | Schema
- 415 Unsupported media type**  
Media type:   
Example Value | Schema

## Swagger V1 - Endpoints de busca

The screenshot shows the Swagger UI interface for a Spring Boot application. It displays three error responses:

- 415 Unsupported media type**  
Media type:   
Example Value | Schema
- 500 Internal server error**  
Media type:   
Example Value | Schema

## Swagger V1 - Respostas de erro

The screenshot shows a browser window with the URL `localhost:8080/swagger-ui/index.html?url.primaryName=v1#/Authentication/login`. At the top, there is an error message: "500 Internal server error" with a dropdown menu for "Media type". Below it, there is a "Schema" section containing a JSON object representing a problem detail for a resource not found. The JSON is as follows:

```
{
  "type": "https://api.food-backend.com/problems/resource-not-found",
  "title": "Resource Not Found",
  "status": 404,
  "detail": "User not found with ID: 123",
  "properties": {
    "additionalProp1": {},
    "additionalProp2": {},
    "additionalProp3": {}
  }
}
```

Below the schema, there is a "Users" section with a description: "User management APIs. Most endpoints require authentication (valid session). Only POST /v1/users (create user) is public and does not require authentication." It lists several endpoints with their methods and URLs:

- DELETE** /v1/users/{id} Delete a user
- GET** /v1/users/{id} Search users by id
- GET** /v1/users Search users
- GET** /v1/users/search/name Search users by name
- GET** /v1/users/search/login Search users by login
- GET** /v1/users/search/emails Search users by email

## Swagger V1 - Schema de resposta

The screenshot shows a browser window with the same URL as the previous one. The main content area is focused on the "PUT /v1/users/{id} Update user info (except password)" endpoint. To the right, there is a sidebar titled "Schemas" which lists several schema definitions:

- AddressDTO >
- UserUpdateRequest > (highlighted in blue)
- ProblemDetailDTO >
- UserResponse >
- UserRequest >
- LoginRequest >
- PasswordChangeRequest >

## Documentação Swagger - Versão 2

A versão 2 da API utiliza autenticação JWT e está documentada separadamente no Swagger. As imagens abaixo ilustram a documentação dos endpoints V2, incluindo o recurso de autenticação Bearer Token:

## Swagger V2 - Visão geral

**API Core - V2** v2.0 OAS 3.0

**API RESTful para o sistema de gestão de restaurantes Food App - Versão 2.**

**🔑 Autenticação**

Esta versão utiliza autenticação **stateless** baseada em **JWT** (JSON Web Token).

Endpoints: [/v2/\\*\\*](#)

Como usar:

- Faça login através do endpoint [POST /v2/auth/login](#)
- Copie o token JWT retornado na resposta
- Inclua o token no header **Authorization** de todas as requisições:

```
Authorization: Bearer {seu_token_aqui}
```

**⚠️ Observações Importantes**

- Endpoints públicos: O endpoint [POST /v2/users](#) (cadastro de usuário) é público e **não requer autenticação**
- Autenticação obrigatória: Para acessar os demais endpoints protegidos, é necessário autenticar-se previamente
- Token JWT: Lembre-se de incluir o token Bearer no header **Authorization** em todas as requisições autenticadas

**🚀 Primeiros Passos**

- Cadastre um novo usuário através de [POST /v2/users](#) (público)

## Swagger V2 - Autenticação JWT

**⚠️ Observações Importantes**

- Endpoints públicos: O endpoint [POST /v2/users](#) (cadastro de usuário) é público e **não requer autenticação**
- Autenticação obrigatória: Para acessar os demais endpoints protegidos, é necessário autenticar-se previamente
- Token JWT: Lembre-se de incluir o token Bearer no header **Authorization** em todas as requisições autenticadas

**🚀 Primeiros Passos**

- Cadastre um novo usuário através de [POST /v2/users](#) (público)
- Autentique-se via [POST /v2/auth/login](#) com suas credenciais
- Utilize o token JWT retornado para acessar os endpoints protegidos

**💡 Vantagens da V2**

- Autenticação stateless (sem necessidade de sessão no servidor)
- Melhor escalabilidade e performance
- Tokens podem ser facilmente revogados
- Suporte a múltiplos dispositivos simultâneos

Contact Thiago Ferreira  
MIT  
[Project README](#)

Servers: <http://localhost:8080 - Generated server url> [Authorize](#)

**Authentication V2** Authentication APIs V2 - JWT token-based authentication

## Swagger V2 - Autenticação JWT

**Users V2** User management APIs V2. Most endpoints require JWT authentication. Only POST /v2/users (create user) is public and does not require authentication. Use 'Bearer (token)' in Authorization header for protected endpoints.

**DELETE /v2/users/{id}** Delete a user

Deletes a user by ID. Requires JWT authentication.

**Parameters**

Name	Description
<b>id</b> <small>required</small>	<small>integer(\$int64)</small> id (path)

**Responses**

Code	Description	Links
204	User deleted successfully	No links
400	Invalid ID format	No links

Media type: [\\*/\\*](#)

Example Value | Schema

## Swagger V2 - Detalhes do endpoint

**Responses**

Code	Description	Links
204	User deleted successfully	No links
400	Invalid ID format	No links
401	Unauthorized - JWT token required	No links

**Example Value** | **Schema**

```
{
  "type": "https://api.food-backend.com/problems/resource-not-found",
  "title": "Resource Not Found",
  "status": 404,
  "detail": "User not found with ID: 123",
  "properties": {
    "additionalProp1": {},
    "additionalProp2": {},
    "additionalProp3": {}
  }
}
```

## Swagger V2 - Respostas de erro

**GET** /v2/users/{id} Search users by id

**GET** /v2/users Search users

**GET** /v2/users/search/name Search users by name

**GET** /v2/users/search/login Search users by login

**GET** /v2/users/search/email Search users by email

**PATCH** /v2/users/{id}/password Change user password

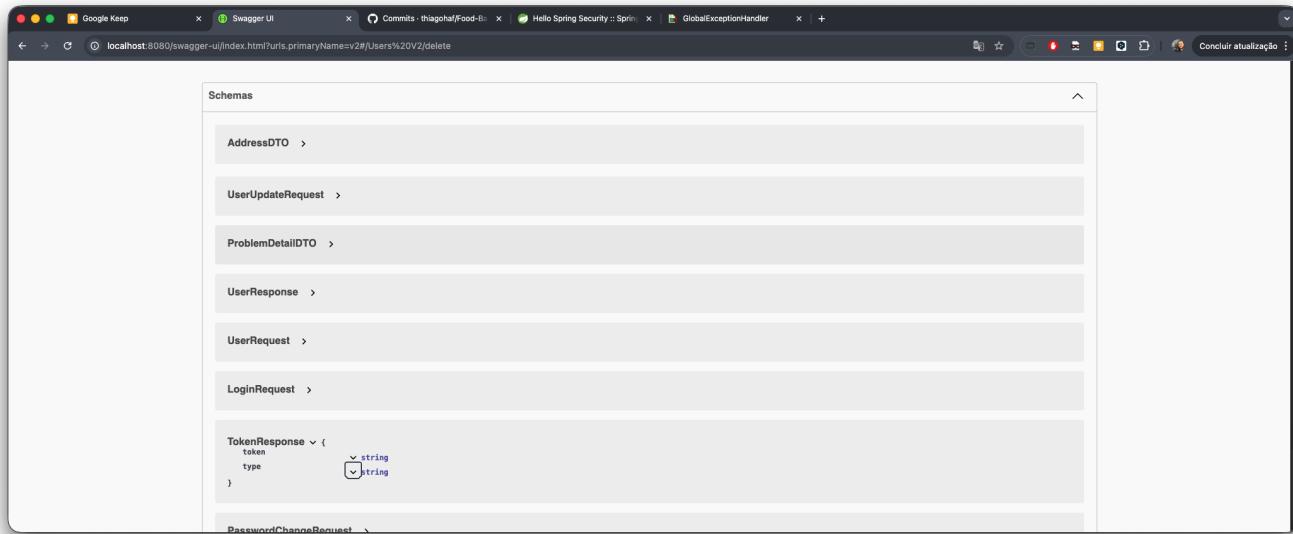
**POST** /v2/users Create a new user

**PUT** /v2/users/{id} Update user info (except password)

**Schemas**

- AddressDTO >
- UserUpdateRequest >

## Swagger V2 - Schemas



Para validação sistemática dos cenários de borda, criei uma coleção completa do Postman ([Food\\_Backend\\_ProblemDetail\\_Tests.postman\\_collection.json](#)) que cobre diversos casos de teste, incluindo login inválido (retorna 404 com ProblemDetail), tentativa de cadastro com email duplicado (retorna 400 com ProblemDetail), requisições sem autenticação em endpoints protegidos (retorna 401), validações de campos obrigatórios, e fluxos completos de CRUD. A coleção utiliza variáveis de ambiente (`{{base_url}}`, `{{user_id}}`, `{{jwt_token}}`) para facilitar execução em diferentes ambientes.

A coleção está organizada em pastas que seguem a estrutura dos testes, facilitando a navegação e execução dos cenários. Durante o desenvolvimento, testei todos os endpoints manualmente através do Postman para garantir que as respostas de erro seguem o padrão RFC 7807 (Problem Details). A seguir, apresento alguns exemplos dos testes realizados:

## Testes de Autenticação

Os testes de autenticação validam o fluxo de login e logout, tanto na versão 1 (sessão HTTP) quanto na versão 2 (JWT). Alguns exemplos:

### Login com sucesso

The screenshot shows the Postman application interface. On the left, there's a sidebar with 'Personal Workspace' and a list of collections, environments, and flows. The main area shows a collection named 'Food Backend - ProblemDetail Tests' with a sub-collection '0. Autenticação'. Under this, there are several test cases: 'POST - Login (sucesso)', 'POST - Login (usuário não encontrado)', 'POST - Login (senha incorreta)', 'POST - Login (validação - login...)', 'POST - Login (validação - senha...)', and 'POST - Logout'. The 'POST - Login (sucesso)' case is selected. The request details show a POST method to '({{base\_url}})/auth/login' with a JSON body containing 'login' and 'password'. The response pane shows a 200 OK status with a response body of '1'. The bottom navigation bar includes 'File', 'Cloud View', 'Find and replace', 'Console', 'Terminal', 'Import Complete', 'Runner', 'Start Proxy', 'Cookies', 'Vault', 'Trash', and 'AI'.

## Acesso não autorizado após logout

The screenshot shows the Postman interface with a successful API call. The URL is `{base_url}/v1/users`. The response status is **200 OK**, and the response body is:

```

1 {
2   "detail": "Authentication required. Please log in to access this resource.",
3   "instance": "http://localhost:8080/v1/users",
4   "status": 401,
5   "title": "Unauthorized",
6   "type": "https://api.food-backend.com/problems/unauthorized",
7   "path": "/v1/users",
8   "method": "GET"
9 }

```

## Acesso não autorizado sem login

The screenshot shows the Postman interface with a successful API call. The URL is `{base_url}/v1/users`. The response status is **200 OK**, and the response body is:

```

1 {
2   "detail": "Authentication required. Please log in to access this resource.",
3   "instance": "http://localhost:8080/v1/users",
4   "status": 401,
5   "title": "Unauthorized",
6   "type": "https://api.food-backend.com/problems/unauthorized",
7   "path": "/v1/users",
8   "method": "GET"
9 }

```

## Login senha incorreta

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case "POST - Login (senha incorreta)" is selected. The request URL is `(base_url)/auth/login`. The "Body" tab is active, showing the following JSON payload:

```

1 {
2   "login": "joaosilva",
3   "password": "senha_errada"
4 }

```

The response status is 404 Not Found, with a response time of 101 ms and a body size of 625 B. The response body is:

```

1 {
2   "detail": "User not found with the provided details.",
3   "instance": "/auth/login",
4   "status": 404,
5   "title": "Resource Not Found",
6   "type": "https://api.food-backend.com/problems/resource-not-found"
7 }

```

## Login com usuário não encontrado

The screenshot shows the Postman interface with the same collection and test case. The request URL is `(base_url)/auth/login`. The "Body" tab is active, showing the following JSON payload:

```

1 {
2   "login": "usuario_inexistente",
3   "password": "senha123"
4 }

```

The response status is 404 Not Found, with a response time of 33 ms and a body size of 625 B. The response body is identical to the previous one.

## Login com validação de login vazio

The screenshot shows the Postman interface with the same collection and test case. The request URL is `(base_url)/auth/login`. The "Body" tab is active, showing the following JSON payload:

```

1 {
2   "login": "",
3   "password": "senha123"
4 }

```

The response status is 400 Bad Request, with a response time of 29 ms and a body size of 609 B. The response body is:

```

1 {
2   "detail": "Validation failed",
3   "instance": "/auth/login",
4   "status": 400,
5   "title": "Validation Error",
6   "type": "https://api.food-backend.com/problems/validation-error",
7   "errors": [
8     {
9       "login": "Login is required"
10    }
11 }

```

## Login com validação de senha vazia

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A POST request is being made to `(base_url)/auth/login`. The Body tab contains the following JSON:

```

1 {
2   "login": "joaosilva",
3   "password": ""
4 }
    
```

The response status is 400 Bad Request, with the following JSON body:

```

1 {
2   "detail": "Validation failed",
3   "instance": "/auth/login",
4   "status": 400,
5   "title": "Validation Error",
6   "type": "https://api.food-backend.com/problems/validation-error",
7   "errors": [
8     "password": "Password is required"
9   ]
10 }
    
```

## Logout

The screenshot shows the Postman interface with the same collection. A POST request is being made to `(base_url)/auth/logout`. The Body tab contains the following JSON:

```

1
    
```

The response status is 200 OK, with the following JSON body:

```

1
    
```

## Testes de Erro - ResourceNotFoundException (404)

Estes testes validam que a API retorna corretamente o status 404 quando um recurso não é encontrado:

### Buscar usuário por ID inexistente

Food Backend - ProblemDetail Tests / 1. ResourceNotFoundException (404) / GET - Buscar usuário por ID inexistente

GET `(base_url)/v1/users/99999`

404 Not Found | 43 ms | 617 B | Save Response | ↻

Key	Value	Description
Key	Value	Description

```

1 {
2   "detail": "User not found with ID: 99999",
3   "instance": "/v1/users/99999",
4   "status": 404,
5   "title": "Resource Not Found",
6   "type": "https://api.food-backend.com/problems/resource-not-found"
7 }
  
```

## Buscar usuário por email inexistente

Food Backend - ProblemDetail Tests / 1. ResourceNotFoundException (404) / GET - Buscar usuário por email inexistente

GET `(base_url)/v1/users/search/email=email_inexistente@teste.com`

404 Not Found | 51 ms | 636 B | Save Response | ↻

Key	Value	Description
<input checked="" type="checkbox"/> Key	email_inexistente@teste.com	Description
<input checked="" type="checkbox"/> email	Value	Description
Key	Value	Description

```

1 {
2   "detail": "User not found with the provided details.",
3   "instance": "/v1/users/search/email",
4   "status": 404,
5   "title": "Resource Not Found",
6   "type": "https://api.food-backend.com/problems/resource-not-found"
7 }
  
```

## Buscar usuário por login inexistente

Food Backend - ProblemDetail Tests / 1. ResourceNotFoundException (404) / GET - Buscar usuário por login inexistente

GET `(base_url)/v1/users/search/login=login_inexistente`

404 Not Found | 49 ms | 636 B | Save Response | ↻

Key	Value	Description
<input checked="" type="checkbox"/> Key	login_inexistente	Description
<input checked="" type="checkbox"/> login	Value	Description
Key	Value	Description

```

1 {
2   "detail": "User not found with the provided details.",
3   "instance": "/v1/users/search/login",
4   "status": 404,
5   "title": "Resource Not Found",
6   "type": "https://api.food-backend.com/problems/resource-not-found"
7 }
  
```

## Deletar usuário inexistente

The screenshot shows the Postman application interface. In the center, there is a request card for a DELETE operation. The URL is set to `(base_url) /v1/users/99999`. The method dropdown shows "DELETE". Below the URL, there are sections for "Docs", "Params", "Authorization", "Headers (8)", "Body", "Scripts", "Tests", and "Settings". Under "Headers (8)", there is a "Query Params" table with one row: "Key" (Key) and "Value" (Value). In the "Body" section, the "JSON" tab is selected, showing a JSON object with fields: "detail": "User not found with the provided details.", "instance": "/v1/users/99999", "status": 404, "title": "Resource Not Found", and "type": "https://api.food-backend.com/problems/resource-not-found". The status bar at the bottom indicates a 404 Not Found response with 59 ms and 629 B.

## Atualizar usuário inexistente

The screenshot shows the Postman application interface. In the center, there is a request card for a PUT operation. The URL is set to `(base_url) /v1/users/99999`. The method dropdown shows "PUT". Below the URL, there are sections for "Docs", "Params", "Authorization", "Headers (11)", "Body", "Scripts", "Tests", and "Settings". Under "Headers (11)", the "Content-Type" is set to "application/json". In the "Body" section, the "JSON" tab is selected, showing a JSON object with fields: "name": "João Silva", "address": {"street": "Rua Teste", "number": "123"}, "city": "São Paulo", and "zipCode": "01234-567". The status bar at the bottom indicates a 404 Not Found response with 26 ms and 617 B.

## Alterar senha de usuário inexistente

**PATCH - {{base\_url}}/v1/users/99999/password**

```

1 {
2   "currentPassword": "senha123",
3   "newPassword": "novasenha456"
4 }
    
```

**Body Cookies (0) Headers (14) Test Results**

```

1 {
2   "detail": "User not found with ID: 99999",
3   "instance": "/v1/users/99999/password",
4   "status": 404,
5   "title": "Resource Not Found",
6   "type": "https://api.food-backend.com/problems/resource-not-found"
7 }
    
```

**404 Not Found** • 12 ms • 626 B

## Testes de Erro - DomainValidationException (400)

Estes testes validam regras de negócio que retornam 400:

### Criar usuário com email duplicado

**POST - {{base\_url}}/v1/users**

```

1 {
2   "name": "Maria Santos",
3   "email": "teste@email.com",
4   "login": "mariasantos",
5   "password": "senha123",
6   "type": "CUSTOMER",
7   "address": {
8     "street": "Rua Teste",
9     "number": "456",
10    "city": "São Paulo",
11    "zipCode": "01234-567"
12  }
13 }
    
```

**Body Cookies (0) Headers (19) Test Results**

```

1 {
2   "detail": "The email provided is already registered.",
3   "instance": "/v1/users",
4   "status": 400,
5   "title": "Domain Validation Error",
6   "type": "https://api.food-backend.com/problems/domain-validation-error"
7 }
    
```

**400 Bad Request** • 13 ms • 606 B

### Alterar senha com senha atual igual a senha nova

## Testes de Erro - MethodArgumentNotValidException (400)

Estes testes validam as validações de entrada dos DTOs:

### Alterar senha com campos vazios

### Alterar senha com nova senha muito curta

**PATCH - Alterar senha (nova senha muito curta)**

```

1 {
2   "currentPassword": "senha123",
3   "newPassword": "123"
4 }
    
```

**Body** Cookies (1) Headers (13) Test Results

**400 Bad Request** 5 ms · 649 B Save Response

```

1 {
2   "detail": "Validation failed",
3   "instance": "/v1/users/1/password",
4   "status": 400,
5   "title": "Validation Error",
6   "type": "https://api.food-backend.com/problems/validation-error",
7   "errors": [
8     "newPassword": "New password must be at least 6 characters"
9   ]
10 }
    
```

## Criar usuário com body vazio

**POST - Criar usuário (body vazio)**

```

1 {}
    
```

**Body** Cookies (1) Headers (13) Test Results

**400 Bad Request** 18 ms · 726 B Save Response

```

1 {
2   "detail": "Validation failed",
3   "instance": "/v1/users",
4   "status": 400,
5   "title": "Validation Error",
6   "type": "https://api.food-backend.com/problems/validation-error",
7   "errors": [
8     "password": "Password is required",
9     "name": "Name is required",
10    "type": "User type is required",
11    "login": "Login is required",
12    "email": "Email is required"
13  ]
14 }
    
```

## Criar usuário com email inválido

**POST - Criar usuário (email inválido)**

```

1 {
2   "name": "João Silva",
3   "email": "email-inválido",
4   "login": "joao.silva",
5   "password": "senha123",
6   "type": "CUSTOMER",
7   "address": {
8     "street": "Rua Teste",
9     "number": "123",
10    "city": "São Paulo",
11    "zipCode": "01234-567"
12  }
13 }
    
```

**Body** Cookies (1) Headers (13) Test Results

**400 Bad Request** 20 ms · 610 B Save Response

```

1 {
2   "detail": "Validation failed",
3   "instance": "/v1/users",
4   "status": 400,
5   "title": "Validation Error",
6   "type": "https://api.food-backend.com/problems/validation-error",
7   "errors": [
8     "email": "Invalid email format"
9   ]
10 }
    
```

## Criar usuário com múltiplos campos inválidos

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case "POST - Criar usuário (múltiplos campos inválidos)" is selected. The request URL is `{base_url}/v1/users`. The request body is:

```

1 {
2   "name": "",
3   "email": "",
4   "login": "test",
5   "password": "123",
6   "type": null
7 }

```

The response status is 400 Bad Request, with the following JSON content:

```

1 {
2   "detail": "Validation failed",
3   "instance": "/v1/users",
4   "status": 400,
5   "title": "validation Error",
6   "type": "https://api.food-backend.com/problems/validation-error",
7   "errors": [
8     {
9       "password": "Password must be at least 6 characters",
10      "name": "Name is required",
11      "type": "User type is required",
12      "login": "Login is required",
13      "email": "Email is required"
14    }
15 }

```

## Criar usuário com name vazio

The screenshot shows the Postman interface with the same collection and test case. The request body is:

```

1 {
2   "name": "",
3   "email": "teste@email.com",
4   "login": "teste",
5   "password": "senha123",
6   "type": "CUSTOMER",
7   "address": "Avenida Paulista",
8   "street": "Rua Teste",
9   "number": "123",
10  "city": "São Paulo",
11  "zipcode": "01234-567"
12 }
13

```

The response status is 400 Bad Request, with the following JSON content:

```

1 {
2   "detail": "Validation failed",
3   "instance": "/v1/users",
4   "status": 400,
5   "title": "validation Error",
6   "type": "https://api.food-backend.com/problems/validation-error",
7   "errors": [
8     {
9       "name": "Name is required"
10    }
11 }

```

## Criar usuário com senha muito curta

Food Backend - ProblemDetail Tests / 3. MethodArgumentNotValidException (400) / POST - Criar usuário (senha muito curta)

```

POST {{base_url}}/v1/users
{
  "name": "João Silva",
  "email": "joao@mail.com",
  "login": "joaosilva",
  "password": "123",
  "type": "STAFFER",
  "address": "Avenida Paulista",
  "street": "Rua Teste",
  "number": "123",
  "city": "São Paulo",
  "zipCode": "01234-567"
}
  
```

Body Cookies (1) Headers (13) Test Results

400 Bad Request 19 ms 631 B Save Response

## Atualizar usuário com name vazio

Food Backend - ProblemDetail Tests / 3. MethodArgumentNotValidException (400) / PUT - Atualizar usuário (name vazio)

```

PUT {{base_url}}/v1/users/{user_id}
{
  "name": "",
  "address": {
    "street": "Rua Teste",
    "number": "123",
    "city": "São Paulo",
    "zipCode": "01234-567"
  }
}
  
```

Body Cookies (1) Headers (13) Test Results

400 Bad Request 13 ms 607 B Save Response

## Outros Testes de Erro

Estes testes validam o tratamento de erros adicionais relacionados a requisições malformadas, endpoints inexistentes, tipos de conteúdo não suportados e outros cenários de erro:

### Endpoint inexistente (GET)

Food Backend - ProblemDetail Tests / Novos Tratamentos de Erro / GET - Endpoint inexistente

GET `(base_url) /v1/users/inexistente/rota`

404 Not Found • 13 ms • 701 B

```

1 {
  "detail": "No handler found for GET /v1/users/inexistente/rota",
  "instance": "/v1/users/inexistente/rota",
  "status": 404,
  "title": "Endpoint Not Found",
  "type": "https://api.food-backend.com/problems/endpoint-not-found",
  "method": "GET",
  "path": "/v1/users/inexistente/rota"
}

```

## Endpoint inexistente (POST)

Food Backend - ProblemDetail Tests / Novos Tratamentos de Erro / POST - Endpoint inexistente

POST `(base_url) /v1/users/rota/que/nao/existe`

404 Not Found • 8 ms • 712 B

```

1 {
  "detail": "No handler found for POST /v1/users/rota/que/nao/existe",
  "instance": "/v1/users/rota/que/nao/existe",
  "status": 404,
  "title": "Endpoint Not Found",
  "type": "https://api.food-backend.com/problems/endpoint-not-found",
  "method": "POST",
  "path": "/v1/users/rota/que/nao/existe"
}

```

## ID com tipo inválido (string) - GET

Food Backend - ProblemDetail Tests / Novos Tratamentos de Erro / GET - ID com tipo Inválido (string)

GET `(base_url) /v1/users/abc`

400 Bad Request • 22 ms • 669 B

```

1 [
  {
    "detail": "Invalid value 'abc' for parameter 'id'. Expected type: Long",
    "instance": "/v1/users/abc",
    "status": 400,
    "title": "Type Mismatch",
    "type": "https://api.food-backend.com/problems/type-mismatch",
    "parameter": "id",
    "expectedType": "long",
    "providedValue": "abc"
  }
]

```

## ID com tipo inválido - PUT

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case titled "PUT - ID com tipo inválido" is selected. The request method is PUT, and the URL is `{base_url}/v1/users/xyz`. The request body is a JSON object with a single field "name": "João Silva". The response status is 400 Bad Request, and the response body contains a detailed error message:

```

1 {
2   "detail": "Invalid value 'xyz' for parameter 'id'. Expected type: Long",
3   "instance": "/v1/users/xyz",
4   "status": 400,
5   "title": "Type Mismatch",
6   "type": "https://api.food-backend.com/problems/type-mismatch",
7   "parameter": "id",
8   "expectedType": "Long",
9   "providedValue": "xyz"
10 }

```

## Parâmetro email faltando

The screenshot shows the Postman interface with the same collection and test case. The request method is GET, and the URL is `{base_url}/v1/users/search/email`. The request includes a "Query Params" table with one row: "Key" (empty) and "Value" (empty). The response status is 400 Bad Request, and the response body contains a detailed error message:

```

1 {
2   "detail": "Required parameter 'email' is missing",
3   "instance": "/v1/users/search/email",
4   "status": 400,
5   "title": "Missing Required Parameter",
6   "type": "https://api.food-backend.com/problems/missing-parameter",
7   "parameter": "email"
8 }

```

## Parâmetro login faltando

Food Backend - ProblemDetail Tests / 4. Novos Tratamentos de Erro / GET - Parâmetro login faltando

GET `(base_url) /v1/users/search/login`

Body Cookies (1) Headers (13) Test Results

```
{
  "detail": "Required parameter 'login' is missing",
  "instance": "/v1/users/search/login",
  "status": 400,
  "title": "Missing Required Parameter",
  "type": "https://api.food-backend.com/problems/missing-parameter",
  "parameter": "login"
}
```

## Parâmetro name faltando

Food Backend - ProblemDetail Tests / 4. Novos Tratamentos de Erro / GET - Parâmetro name faltando

GET `(base_url) /v1/users/search/name`

Body Cookies (1) Headers (13) Test Results

```
{
  "detail": "Required parameter 'name' is missing",
  "instance": "/v1/users/search/name",
  "status": 400,
  "title": "Missing Required Parameter",
  "type": "https://api.food-backend.com/problems/missing-parameter",
  "parameter": "name"
}
```

## Body vazio quando obrigatório

Food Backend - ProblemDetail Tests / 4. Novos Tratamentos de Erro / POST - Body vazio quando obrigatório

POST `(base_url) /v1/users`

Body Cookies (1) Headers (13) Test Results

```
{
  "detail": "Request body is required but was not provided",
  "instance": "/v1/users",
  "status": 400,
  "title": "Malformed Request",
  "type": "https://api.food-backend.com/problems/malformed-request"
}
```

## Content-Type não suportado (XML) - POST

The screenshot shows the Postman interface with a POST request to `{base_url}/v1/users`. The request body contains XML data:

```
<user><name>João</name></user>
```

The response status is 415 Unsupported Media Type, with a message indicating that the media type 'application/xml; charset=UTF-8' is not supported.

## Content-Type não suportado (text/plain) - PUT

The screenshot shows the Postman interface with a PUT request to `{base_url}/v1/users/1`. The request body contains plain text data:

```
name=João
```

The response status is 415 Unsupported Media Type, with a message indicating that the media type 'text/plain; charset=UTF-8' is not supported.

## JSON malformado

Food Backend - ProblemDetail Tests / Novos Tratamentos de Erro / POST - JSON malformado

```
POST {base_url}/v1/users
{
  "name": "João Silva",
  "email": "joao@email.com",
  "login": "joaosilva",
  "password": "senha123",
  "type": "CUSTOMER"
}

```

Body Cookies Headers Test Results

400 Bad Request 22 ms 621 B Save Response

## Método HTTP não suportado (tentar POST em GET)

Food Backend - ProblemDetail Tests / Novos Tratamentos de Erro / POST - Método HTTP não suportado (tentar POST em GET)

```
POST {base_url}/v1/users/search/name
{
}

```

Body Cookies Headers Test Results

405 Method Not Allowed 33 ms 696 B Save Response

## Testes de Casos de Sucesso

Os testes de casos de sucesso validam que os endpoints funcionam corretamente quando recebem dados válidos:

### Criar usuário válido (endpoint público)

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case "POST - Criar usuário válido (público)" is selected. The request method is POST, and the URL is `{base_url}/v1/users`. The Body tab contains the following JSON payload:

```

1 {
2   "name": "Thiago Ferreira",
3   "email": "thiago@email.com",
4   "login": "thiagoferreira",
5   "password": "senha123",
6   "type": "USER",
7   "address": {
8     "number": "123",
9     "street": "Rua Teste",
10    "city": "São Paulo",
11    "zipCode": "01234-567"
12  }
13 }

```

The response status is 201 Created, and the response body shows the created user's details, including an ID of 24 and a last update timestamp.

## Login (antes de acessar endpoints protegidos)

The screenshot shows the Postman interface with the same collection. A test case "POST - Login (antes de acessar endpoints protegidos)" is selected. The request method is POST, and the URL is `{base_url}/auth/login`. The Body tab contains the following JSON payload:

```

1 {
2   "login": "joaosilva",
3   "password": "senha123"
4 }

```

The response status is 200 OK, and the response body includes a session cookie named JSESSIONID.

## Listar todos os usuários

The screenshot shows the Postman interface with the same collection. A test case "GET - Listar todos os usuários" is selected. The request method is GET, and the URL is `{base_url}/v1/users`. The Headers tab shows the presence of the session cookie JSESSIONID. The response status is 200 OK, and the response body is a JSON array of user objects, including one for Tania Ferreira.

## Buscar usuário por ID

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case titled "GET - Buscar usuário por ID" is selected. The request URL is `{base_url}/vt/users/{user_id}`. The response status is 200 OK, and the response body is a JSON object representing a user:

```

1 [
2   {
3     "address": {
4       "city": "São Paulo",
5       "number": "123",
6       "street": "Rua Teste",
7       "zipCode": "01234-567"
8     },
9     "email": "thiago@email.com",
10    "id": 24,
11    "lastUpdate": "2026-01-17T01:38:47.892296",
12    "login": "thiagoferreira",
13    "name": "Thiago Ferreira",
14    "type": "OWNER"
15  }
16 ]

```

## Buscar usuário por nome

The screenshot shows the Postman interface with the same collection and test case. The request URL is `{base_url}/vt/users/search?name=João`. The response status is 200 OK, and the response body contains two users matching the name "João".

```

1 [
2   {
3     "address": {
4       "city": "São Paulo",
5       "number": "123",
6       "street": "Rua Teste",
7       "zipCode": "01234-567"
8     },
9     "email": "joao@email.com",
10    "id": 16,
11    "lastUpdate": "2026-01-02T01:55:52.768172",
12    "login": "joao",
13    "name": "João Ferreira",
14    "type": "CUSTOMER"
15  },
16  {
17    "address": {
18      "city": "São Paulo",
19    }
20  }
21 ]

```

## Buscar usuário por email

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case titled "5. Casos de Sucesso" is selected, showing a successful GET request to the endpoint `(base_url)/v1/users/search/email?email=jao@email.com`. The response status is 200 OK, and the response body is a JSON object containing user details.

```

{
  "id": 21,
  "name": "Jo\u00e3o Silva",
  "email": "jao@email.com",
  "address": {
    "city": "S\u00e3o Paulo",
    "number": "123",
    "street": "Rua Teste",
    "zipCode": "01234-567"
  },
  "login": "joaosilva",
  "type": "CUSTOMER"
}

```

## Buscar usuário por login

The screenshot shows the Postman interface with the same collection and test case. A new GET request is made to the endpoint `(base_url)/v1/users/search/login?login=joaosilva`. The response status is 200 OK, and the response body is identical to the previous one.

```

{
  "id": 21,
  "name": "Jo\u00e3o Silva",
  "email": "jao@email.com",
  "address": {
    "city": "S\u00e3o Paulo",
    "number": "123",
    "street": "Rua Teste",
    "zipCode": "01234-567"
  },
  "login": "joaosilva",
  "type": "CUSTOMER"
}

```

## Atualizar usuário

The screenshot shows the Postman interface with the same collection and test case. A PUT request is made to the endpoint `(base_url)/v1/users/{user_id}` with the raw JSON body:

```

{
  "name": "Thiago Ferreira Atualizado",
  "address": {
    "street": "Rua Nova",
    "number": "456",
    "city": "Rio de Janeiro",
    "zipCode": "20000-000"
  }
}

```

The response status is 200 OK, and the response body shows the updated user information.

```

{
  "id": 24,
  "name": "Thiago Ferreira Atualizado",
  "email": "thiag@email.com",
  "address": {
    "city": "Rio de Janeiro",
    "number": "456",
    "street": "Rua Nova",
    "zipCode": "20000-000"
  },
  "login": "thiagoferreira",
  "name": "Thiago Ferreira Atualizado",
  "type": "OWNER"
}

```

## Alterar senha

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case titled "PATCH - Alterar senha" is selected. The request method is PATCH, and the URL is `{base_url}/v1/users/{user_id}/password`. The Body tab contains the following JSON payload:

```

1 {
2   "currentPassword": "senha123",
3   "newPassword": "novasenha456"
4 }

```

The response status is 204 No Content, with a duration of 178 ms and a total size of 371 B.

## Deletar usuário

The screenshot shows the Postman interface with the same collection and test case. The request method is DELETE, and the URL is `{base_url}/v1/users/{user_id}`. The Headers tab shows "Content-Type: application/json". The response status is 204 No Content, with a duration of 50 ms and a total size of 371 B.

## Testes V2 - Autenticação JWT

A versão 2 utiliza autenticação JWT através do Spring Security. Os testes estão organizados em três categorias principais que validam o fluxo completo de autenticação, casos de sucesso e tratamento de erros.

### Autenticação V2

Os testes de autenticação V2 validam o fluxo de login e logout com JWT, além de cenários de acesso não autorizado:

#### Login V2 com sucesso

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A test case "POST - Login V2 (sucesso)" is selected, which sends a POST request to `{base_url}/v2/auth/login` with the following JSON body:

```

1 {
  "login": "joaosilva",
  "password": "senha123"
}

```

The response status is 200 OK, and the response body contains a token and type information.

## Login V2 com usuário não encontrado

The screenshot shows the Postman interface with the same collection. A test case "POST - Login V2 (usuário não encontrado)" is selected, which sends a POST request to `{base_url}/v2/auth/login` with the following JSON body:

```

1 {
  "login": "usuario_inexistente",
  "password": "senha123"
}

```

The response status is 404 Not Found, and the response body indicates that the user was not found.

## Login V2 com senha incorreta

The screenshot shows the Postman interface with the same collection. A test case "POST - Login V2 (senha incorreta)" is selected, which sends a POST request to `{base_url}/v2/auth/login` with the following JSON body:

```

1 {
  "login": "joaosilva",
  "password": "senha_errada"
}

```

The response status is 404 Not Found, and the response body indicates that the user was not found.

## Acesso não autorizado V2 sem token

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case titled "GET - Acesso não autorizado V2 (sem token)" is selected. The request URL is `{(base_url)} /v2/users`. The response status is 401 Unauthorized, and the response body is a JSON object indicating an unauthorized access error.

```

1 {
2   "type": "https://api.food-backend.com/problems/unauthorized",
3   "title": "Unauthorized",
4   "status": 401,
5   "detail": "Authentication required. Please provide a valid JWT token in the Authorization header.",
6   "instance": "http://localhost:8080/v2/users",
7   "properties": [
8     {
9       "path": "/v2/users",
10      "method": "GET"
11    }
12  ]
}
  
```

## Acesso não autorizado V2 com token inválido

The screenshot shows the Postman interface with the same collection and test case. This time, the Authorization header is set to "Bearer token\_invalid,12345". The response status is 401 Unauthorized, and the response body is a JSON object indicating an unauthorized access error.

```

1 {
2   "type": "https://api.food-backend.com/problems/unauthorized",
3   "title": "Unauthorized",
4   "status": 401,
5   "detail": "Authentication required. Please provide a valid JWT token in the Authorization header.",
6   "instance": "http://localhost:8080/v2/users",
7   "properties": [
8     {
9       "path": "/v2/users",
10      "method": "GET"
11    }
12  ]
}
  
```

## Logout V2 com sucesso

The screenshot shows the Postman interface with the following details:

- Left Sidebar:** Personal Workspace, Collections (Food Backend - ProblemDetail Tests), Environments, History, Flows, Notes, Files (BETA).
- Top Bar:** Home, Workspaces, API Network, Search Postman.
- Central Area:**
  - Collection: Food Backend - ProblemDetail Tests
  - Test Case: 6.1 Autenticação V2
    - POST POST - Login V2 (sucesso)
    - POST POST - Login V2 (usuário não encontrado)
    - POST POST - Login V2 (senha incorreta)
    - GET GET - Acesso não autorizado V2 (sem token)
    - GET GET - Acesso não autorizado V2 (token inválido)
    - POST POST - Logout V2 (sucesso)
    - POST POST - Logout V2 (sem token - erro 401)
    - 6.2 Casos de Sucesso V2
    - 6.3 Erros V2 (ProblemDetail)
  - FoodApp, localhost:8080, Locatech, notes.
- Bottom Bar:** Cloud View, Find and replace, Console, Terminal, Import Complete, Runner, Start Proxy, Cookies, Vault, Trash.

## Logout V2 sem token (erro 401)

The screenshot shows the Postman interface with the following details:

- Left Sidebar:** Personal Workspace, Collections (Food Backend - ProblemDetail Tests), Environments, History, Flows, Notes, Files (BETA).
- Top Bar:** Home, Workspaces, API Network, Search Postman.
- Central Area:**
  - Collection: Food Backend - ProblemDetail Tests
  - Test Case: 6.1 Autenticação V2
    - POST POST - Logout V2 (sem token - erro 401)
  - FoodApp, localhost:8080, Locatech, notes.
- Bottom Bar:** Cloud View, Find and replace, Console, Terminal, Import Complete, Runner, Start Proxy, Cookies, Vault, Trash.

**Test Result:**

```
401 Unauthorized | 22 ms | 727 B | Pass the correct auth credentials
```

**Body (JSON):**

```
1: { "type": "https://api.food-backend.com/problems/unauthorized",
2:   "title": "Unauthorized",
3:   "status": 401,
4:   "detail": "Authentication required. Please provide a valid JWT token in the Authorization header.",
5:   "instance": "http://localhost:8080/v2/auth/logout",
6:   "properties": {
7:     "path": "/v2/auth/logout",
8:     "method": "POST"
9:   }
10: }
```

## Casos de Sucesso V2

Os testes de casos de sucesso V2 validam que os endpoints protegidos funcionam corretamente quando autenticados com JWT:

### Criar usuário V2 (endpoint público)

Food Backend - ProblemDetail Tests / 6. Endpoints V2 - JWT Authentication / 6.2 Casos de Sucesso V2 / POST - Criar usuário V2 (público)

**POST** `{base_url}/v2/users`

Body: `x-www-form-urlencoded`

```

1 {
  "name": "Maria Santos",
  "email": "maria@email.com",
  "login": "mariasantos",
  "password": "senha123",
  "type": "CUSTOMER",
  "address": {
    "street": "Rua V2",
    "number": "789",
    "city": "São Paulo",
    "zipCode": "01234-567"
  },
  "email": "maria@email.com",
  "id": 25,
  "lastUpdate": "2026-01-17T01:51:21.87909",
  "login": "mariasantos",
  "name": "Maria Santos"
}

```

Body Cookies (1) Headers (14) Test Results

201 Created - 129 ms - 653 B

## Listar todos os usuários V2

Food Backend - ProblemDetail Tests / 6. Endpoints V2 - JWT Authentication / 6.2 Casos de Sucesso V2 / GET - Listar todos os usuários V2

**GET** `{base_url}/v2/users`

Query Params:

Key	Value	Description
Key	Value	Description

Body Cookies (1) Headers (15) Test Results

200 OK - 47 ms - 1.63 KB

## Buscar usuário por ID V2

Food Backend - ProblemDetail Tests / 6. Endpoints V2 - JWT Authentication / 6.2 Casos de Sucesso V2 / GET - Buscar usuário por ID V2

**GET** `{base_url}/v2/users/{user_id}`

Query Params:

Key	Value	Description
Key	Value	Description

Body Cookies (1) Headers (15) Test Results

200 OK - 37 ms - 745 B

## Buscar usuários por nome V2

Food Backend - ProblemDetail Tests / 6. Endpoints V2 ~ JWT Authentication / 6.2 Casos de Sucesso V2 / GET - Buscar usuários por nome V2

GET `({base_url})/v2/users/search/name?name=María`

Query Params

name	María
------	-------

Body

```
{
  "id": 25,
  "name": "Maria Santos",
  "login": "mariasantos",
  "email": "maria@email.com",
  "type": "CUSTOMER",
  "street": "Rua V2",
  "number": "89",
  "city": "São Paulo",
  "zipCode": "01234-567"
}
```

200 OK · 720 ms · 725 B

## Atualizar usuário V2

Food Backend - ProblemDetail Tests / 6. Endpoints V2 ~ JWT Authentication / 6.2 Casos de Sucesso V2 / PUT - Atualizar usuário V2

PUT `({base_url})/v2/users/((user_id))`

Body

```
{
  "name": "Maria Santos Atualizada",
  "address": {
    "street": "Rua Nova V2",
    "number": "999",
    "city": "Rio de Janeiro",
    "zipCode": "20000-000"
  }
}
```

Body

```
{
  "id": 25,
  "name": "Maria Santos Atualizada",
  "login": "mariasantos",
  "email": "maria@email.com",
  "type": "CUSTOMER",
  "street": "Rua V2",
  "number": "89",
  "city": "São Paulo",
  "zipCode": "01234-567"
}
```

200 OK · 67 ms · 744 B

## Erros V2 (ProblemDetail)

Os testes de erros V2 validam que os erros retornados seguem o padrão RFC 7807 (ProblemDetail):

### Criar usuário V2 com email duplicado

**POST - Criar usuário V2 (email duplicado)**

```

1 {
2   "name": "Teste Duplicado",
3   "email": "maria@email.com",
4   "login": "testeduplicado",
5   "password": "senha123",
6   "type": "CUSTOMER"
7 }
  
```

**Body Cookies Headers Test Results**

**400 Bad Request** 28 ms 606 B Save Response

```

1 {
2   "detail": "The email provided is already registered.",
3   "instance": "/v2/users",
4   "status": 400,
5   "title": "Domain Validation Error",
6   "type": "https://api.food-backend.com/problems/domain-validation-error"
7 }
  
```

## Criar usuário V2 com validação (campos inválidos)

**POST - Criar usuário V2 (validação - campos inválidos)**

```

1 {
2   "name": "",
3   "email": "email-invalido",
4   "login": "",
5   "password": "123"
6 }
  
```

**Body Cookies Headers Test Results**

**400 Bad Request** 19 ms 747 B Save Response

```

1 {
2   "detail": "Validation failed",
3   "instance": "/v2/users",
4   "status": 400,
5   "title": "Validation Error",
6   "type": "https://api.food-backend.com/problems/validation-error",
7   "errors": [
8     {
9       "password": "Password must be at least 6 characters",
10      "name": "Name is required",
11      "type": "User type is required",
12      "login": "Login is required",
13      "email": "Invalid email format"
14    }
15 ]
  
```

## Usuário inexistente V2

**GET - Usuário inexistente V2**

**Query Params**

Key	Value	Description
Key	Value	Description

**Body Cookies Headers Test Results**

**404 Not Found** 52 ms 692 B Save Response

```

1 {
2   "detail": "User not found with ID: 99999",
3   "instance": "/v2/users/99999",
4   "status": 404,
5   "title": "Resource Not Found",
6   "type": "https://api.food-backend.com/problems/resource-not-found"
7 }
  
```

## Testes Automatizados e Cobertura de Código (JaCoCo)

A aplicação possui uma suíte completa de testes automatizados organizados no pacote `src/test/java` seguindo a estrutura do código de produção. Os testes incluem testes unitários para services (`UserService`), controllers (`UserControllerV2Test`, `AuthControllerV2Test`), mappers (`UserMapper`), interceptors (`AuthInterceptorTest`), configurações (`OpenApiConfigTest`), filtros de segurança (`JwtAuthenticationFilterTest`) e handlers de exceção (`GlobalExceptionHandlerTest`). O Maven Surefire Plugin executa os testes durante o build, garantindo que todas as alterações no código sejam validadas automaticamente.

Para garantir a qualidade do código e identificar áreas não testadas, implementei a análise de cobertura de código utilizando o **JaCoCo (Java Code Coverage)** versão 0.8.11. O JaCoCo é configurado através do plugin Maven `jacoco-maven-plugin` no `pom.xml` com três execuções principais:

### 1. Prepare Agent (`prepare-agent`):

- Configurado para instrumentar as classes durante a execução dos testes
- Exclui classes do sistema (`sun/**`, `jdk/**`, `java/**`, `com/sun/**`) e a classe principal da aplicação (`Application.class`) da análise de cobertura, focando apenas no código de negócios

### 2. Report (`report`):

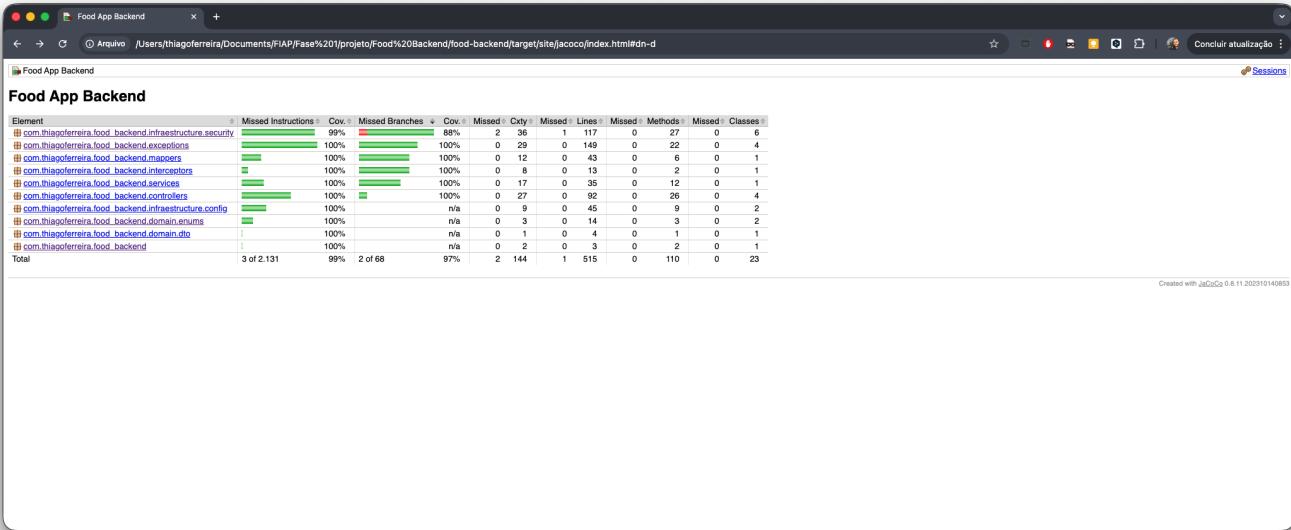
- Executado na fase `test` do Maven
- Gera relatórios HTML detalhados em `target/site/jacoco/` com métricas de cobertura por classe, método e linha
- Os relatórios incluem informações sobre instruções, branches, linhas, métodos e classes cobertas

### 3. Check (`jacoco-check`):

- Valida se a cobertura de código atende aos critérios mínimos estabelecidos
- Configurado com meta mínima de **80% de cobertura de linhas** (`COVEREDRATIO` de 0.80) para o bundle completo
- O build falha se a cobertura ficar abaixo do mínimo, garantindo que novos códigos mantenham ou melhorem a qualidade

Durante o desenvolvimento, executei os testes e gerei os relatórios de cobertura para validar que todas as funcionalidades críticas estão adequadamente testadas. A figura abaixo mostra o relatório de cobertura do JaCoCo, onde podemos observar a cobertura geral do projeto e os detalhes por pacote e classe.

### Relatório de Cobertura JaCoCo



A integração do JaCoCo com o ciclo de build garante que a qualidade do código seja mantida continuamente. Toda vez que os testes são executados através de `mvn test` ou `mvn clean package`, o JaCoCo coleta dados de cobertura, gera relatórios e valida se os critérios mínimos são atendidos. Isso permite identificar rapidamente áreas do código que precisam de testes adicionais e ajuda a manter a confiabilidade do sistema durante evoluções futuras.

A suíte de testes implementada abrange diferentes tipos de testes para garantir a qualidade em múltiplas camadas:

- **Testes unitários:** Testes isolados de componentes individuais como services, mappers e utilitários, utilizando mocks para isolar dependências
- **Testes de integração:** Testes que utilizam o contexto completo do Spring Boot (`@SpringBootTest`) para validar interações entre componentes
- **Testes de controladores:** Testes focados nos endpoints REST utilizando `MockMvc` para simular requisições HTTP e validar respostas, códigos de status e estrutura de dados
- **Testes de serviços:** Testes da lógica de negócios isolada, validando regras de domínio e transformações de dados
- **Testes de exceções:** Testes específicos para o tratamento de erros, garantindo que exceções são capturadas corretamente pelo `GlobalExceptionHandler` e convertidas em respostas `ProblemDetail` apropriadas

## 5. Validações Implementadas

A aplicação implementa um sistema abrangente de validações em múltiplas camadas para garantir a integridade e consistência dos dados. As validações são aplicadas tanto no nível de entrada (DTOs) quanto no nível de negócio (services), criando uma defesa em profundidade contra dados inválidos.

### Validações de Entrada (Bean Validation)

As validações de entrada são implementadas utilizando Bean Validation (JSR 303/380) através de anotações nos DTOs. O Spring Boot Starter Validation ativa automaticamente a validação e o `GlobalExceptionHandler` captura `MethodArgumentNotValidException`, convertendo-as em respostas `ProblemDetail` com detalhamento dos erros.

## Validações implementadas nos DTOs:

- **Email:** Formato válido de email através de `@Email` e `@NotBlank`, garantindo que apenas endereços de email válidos sejam aceitos
- **Password:** Mínimo de 6 caracteres através de `@Size(min = 6)`, assegurando senhas com complexidade mínima
- **Campos obrigatórios:** Name, Email, Login, Password e UserType são marcados como `@NotBlank` ou `@NotNull`, garantindo que dados essenciais estejam presentes
- **Tipos de dados:** Validações de tipo garantem que os dados recebidos correspondem aos tipos esperados (String, Enum, etc.)

Essas validações são executadas automaticamente pelo Spring antes que a requisição chegue ao controller, retornando `400 Bad Request` com detalhes dos campos inválidos caso algum dado não atenda aos critérios.

## Validações de Domínio (Business Rules)

Além das validações de entrada, a aplicação implementa validações de negócio na camada de serviços, que garantem regras de domínio mais complexas:

- **Unicidade de Email:** O `UserService.createUser()` verifica através de `UserRepository.existsByEmail()` se o email já está cadastrado antes de criar um novo usuário. Se duplicado, lança `DomainValidationException` com status 400
- **Unicidade de Login:** Similarmente, o login é verificado através de `UserRepository.existsByLogin()` para garantir que cada usuário tenha um identificador único
- **Validação de Senha na Alteração:** O método `UserService.changePassword()` valida que a senha atual fornecida corresponde à senha armazenada usando `BCrypt.checkpw()` antes de aplicar a nova senha. Também verifica que a senha atual e nova senha sejam diferentes, lançando `DomainValidationException` se forem iguais
- **Existência de Recurso:** Operações de atualização, exclusão e busca por ID verificam a existência do usuário através de `UserRepository.findById()` antes de proceder, lançando `ResourceNotFoundException` (404) se o recurso não existir
- **Autenticação de Senha:** No login, a senha fornecida é verificada contra o hash armazenado usando BCrypt, garantindo que apenas usuários com credenciais corretas possam autenticar

Essas validações de domínio são críticas para manter a integridade dos dados e garantir que operações sensíveis (como alteração de senha) sejam realizadas apenas quando todas as condições de negócio são atendidas. A separação entre validações de entrada (formato e estrutura) e validações de domínio (regras de negócio) segue boas práticas de arquitetura, permitindo que cada camada tenha responsabilidades bem definidas.

## 6. Guia de Infraestrutura (Docker)

O projeto foi containerizado utilizando Docker e Docker Compose para facilitar a execução local e garantir consistência entre ambientes de desenvolvimento. O `docker-compose.yml` define dois serviços que trabalham em conjunto.

Durante o desenvolvimento e testes, utilizei o Docker Desktop para gerenciar os containers e o IntelliJ IDEA para executar a aplicação quando necessário. As imagens abaixo mostram os containers em execução:

## Docker - Containers em execução no Docker Desktop

The screenshot shows the Docker Desktop interface. On the left, a sidebar includes options like Ask Gordon (BETA), Containers (selected), Images, Volumes, Builds, Models (BETA), MCP Toolkit (BETA), Docker Hub, Docker Scout, and Extensions. The main area is titled 'Containers' with a 'Give feedback' link. It displays container CPU usage (0.30% / 1100%) and memory usage (372.81MB / 7.47GB). A search bar and a 'Show charts' button are at the top right. Below is a table listing three containers:

	Name	Container ID	Image	Port(s)	CPU (%)	Last start...	Actions
<input type="checkbox"/>	food-backend	-	-	-	0.3%	14 days ago	<span style="color: blue;">[ ]</span> <span style="color: blue;">[ ... ]</span> <span style="color: red;">[ ]</span>
<input type="checkbox"/>	food-app	c161c3b71391	food-backend-app	8081:8080	0.11%	14 days ago	<span style="color: blue;">[ ]</span> <span style="color: blue;">[ ... ]</span> <span style="color: red;">[ ]</span>
<input type="checkbox"/>	food-postgres	7dfdb8fbad17	postgres:16-alpine	5432:5432	0.19%	14 days ago	<span style="color: blue;">[ ]</span> <span style="color: blue;">[ ... ]</span> <span style="color: red;">[ ]</span>

At the bottom, status indicators show 'Engine running', system resources (RAM 4.80 GB, CPU 0.09%, Disk 9.90 GB used / limit 452.13 GB), and links to 'Terminal' and 'New version available'.

## Docker - Detalhes dos containers no IntelliJ IDEA

The screenshot shows the IntelliJ IDEA interface with the Docker Compose service tree on the left. Services listed include Docker-compose: docker, Docker-compose: docker-local, Docker-compose: food-backend (with app and postgres sub-containers), food-backend-default, and Docker-compose: spring-boot. The food-postgres container is selected. The central pane shows its logs:

```

Log Dashboard
food-postgres 7dfdb8fb postgres:16-alpine Docker-compose: food-backend
2026-01-11 04:50:36.797 UTC [95686] ERROR: syntax error at or near "Maria Santos" at character 44
2026-01-11 04:50:36.797 UTC [95686] STATEMENT: delete from public.tb_users where name is Maria Santos
2026-01-11 04:50:59.045 UTC [95686] ERROR: syntax error at or near "23" at character 42
2026-01-11 04:50:59.045 UTC [95686] STATEMENT: delete from public.tb_users where id is 23
2026-01-11 04:53:38.735 UTC [27] LOG: checkpoint starting: time
2026-01-11 04:53:39.151 UTC [27] LOG: checkpoint complete: wrote 5 buffers (0.8%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.421 s, sync=0.005 s

```

Buttons for 'Open Project', 'Restart', 'Stop', and 'Terminal' are at the top right. Status bar at the bottom indicates 59:27 (72 chars), LF, UTF-8, and 4 spaces.

## Serviços Configurados

### PostgreSQL (postgres):

- Imagem: **postgres:16-alpine** (versão leve e atualizada)
- Container: **food-postgres**
- Porta: **5432:5432** (mapeada para o host)
- Variáveis de ambiente:
  - **POSTGRES\_DB=food\_db**
  - **POSTGRES\_USER=postgres**
  - **POSTGRES\_PASSWORD=postgres**
- Volume persistente: **postgres\_data** montado em **/var/lib/postgresql/data** para preservar dados entre reinicializações
- Health check configurado: **pg\_isready -U postgres** verifica a disponibilidade a cada 10 segundos

## Aplicação Spring Boot (app):

- Build: Utiliza o **Dockerfile** local para construir a imagem
- Container: **food-app**
- Dependência: Aguarda o serviço **postgres** estar saudável (**condition: service\_healthy**)
- Variáveis de ambiente:
  - **DB\_HOST=postgres** (nome do serviço no Docker Compose)
  - **DB\_PORT=5432**
  - **DB\_NAME=food\_db**
  - **DB\_USER=postgres**
  - **DB\_PASSWORD=postgres**
  - **SERVER\_PORT=8080** (porta interna do container)
- Porta: **8081:8080** (mapeia porta 8081 do host para 8080 do container)

## Dockerfile

O **Dockerfile** utiliza multi-stage build para otimizar o tamanho da imagem final:

### Stage 1 (build):

- Base: **maven:3.9-eclipse-temurin-21**
- Copia **pom.xml** primeiro (aproveita cache do Docker)
- Copia código fonte
- Executa **mvn clean package -DskipTests** para construir o JAR

### Stage 2 (runtime):

- Base: **eclipse-temurin:21-jre** (apenas JRE, menor que JDK completo)
- Copia o JAR gerado do stage anterior
- Expõe porta 8080
- Define **ENTRYPOINT** para executar o JAR com **java -jar app.jar**

## Comandos para Execução

### 1. Construir e iniciar todos os serviços:

```
cd food-backend
docker-compose up --build
```

O flag **--build** força a reconstrução das imagens, útil na primeira execução ou após alterações no código.

### 2. Executar em background (detached mode):

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

### 3. Visualizar logs:

```
# Logs de todos os serviços  
docker-compose logs -f  
  
# Logs apenas da aplicação  
docker-compose logs -f app  
  
# Logs apenas do PostgreSQL  
docker-compose logs -f postgres
```

#### 4. Parar os serviços:

```
docker-compose down
```

#### 5. Parar e remover volumes (apaga dados do banco):

```
docker-compose down -v
```

#### 6. Reconstruir apenas a aplicação (útil durante desenvolvimento):

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

### Verificação da Aplicação

Após subir os containers, aguarde alguns segundos para a aplicação inicializar completamente. Acesse:

- **API Swagger UI:** <http://localhost:8081/swagger-ui.html>
- **OpenAPI JSON:** <http://localhost:8081/api-docs>
- **Health check manual:** `curl http://localhost:8081/v1/users` (deve retornar 401 se não autenticado, confirmando que a API está respondendo)

### Troubleshooting

Se a aplicação falhar ao conectar ao banco, verifique:

```
# Status dos containers  
docker-compose ps  
  
# Logs da aplicação procurando por erros de conexão  
docker-compose logs app | grep -i "database\|connection\|jdbc"  
  
# Testar conectividade do container app para o postgres  
docker-compose exec app ping postgres
```

---

Para limpar completamente e recomeçar:

```
docker-compose down -v  
docker system prune -f  
docker-compose up --build
```

---

## Considerações Finais

Este relatório técnico documenta o desenvolvimento do sistema de gestão de restaurantes, desde a arquitetura e modelagem de dados até a infraestrutura Docker. O projeto demonstra a aplicação de conceitos importantes como separação de responsabilidades, tratamento padronizado de erros, documentação automática de APIs, validações em múltiplas camadas e containerização.

A escolha de tecnologias modernas como Spring Boot 4.0.1, Java 21, PostgreSQL e Docker garante que a aplicação esteja preparada para escalabilidade e manutenção futura. A coexistência de duas versões da API (V1 com sessão HTTP e V2 com JWT) permite migração gradual sem impactar clientes existentes. O sistema abrangente de validações implementado (Bean Validation e regras de domínio) garante a integridade dos dados em todas as camadas da aplicação.

Todos os códigos-fonte, documentação adicional e coleções de testes estão disponíveis no repositório público do GitHub: <https://github.com/thiagohaf/Food-Backend>