

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

Decidi construir o projeto sobre o ecossistema Spring Boot 4.0.1 utilizando Java 21, uma escolha que considero adequada para criar uma base sólida capaz de suportar as operações críticas de um sistema de gestão de restaurantes. 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.

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.

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.

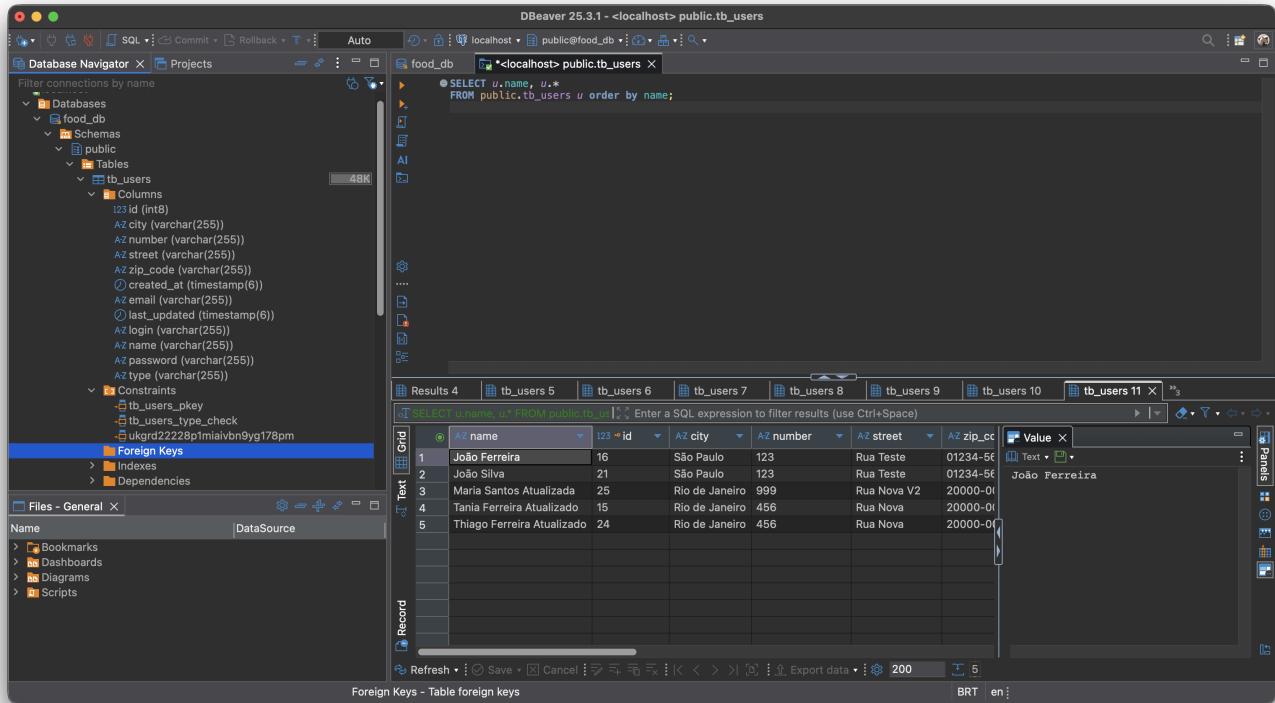
A entidade `Address` foi implementada como `@Embeddable`, permitindo que seja incorporada diretamente na tabela `tb_users` sem necessidade de uma tabela separada. Essa decisão simplifica a persistência e consultas, já que endereço não possui identidade própria e está sempre associado a um usuário específico. 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`).



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:

- Faça login através do endpoint [POST /auth/login](#)
- A sessão é mantida automaticamente através de cookies (JSESSIONID)
- 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 - Stateful 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

[application/json](#)

Example Value Schema

Swagger V1 - Endpoints de autenticação

The screenshot shows the Swagger UI interface for the authentication endpoints. It lists three error responses:

- 200**: Login successful - session created. Example Value:

```
{ "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": {} } }
```
- 400**: Validation error. Example Value:

```
{ "type": "https://api.food-backend.com/problems/resource-not-found", "title": "Resource Not Found", "status": 404, "detail": "User not found or invalid credentials", "properties": { "additionalProp1": {}, "additionalProp2": {}, "additionalProp3": {} } }
```
- 404**: User not found or invalid credentials. Example Value:

```
{ "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": {} } }
```
- 415**: Unsupported media type. Example Value:

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

Swagger V1 - Endpoints de busca

The screenshot shows the Swagger UI interface for the search endpoints. It lists three error responses:

- 415**: Unsupported media type. Example Value:

```
{ "type": "https://api.food-backend.com/problems/resource-not-found", "title": "Resource Not Found", "status": 404, "detail": "Unsupported media type", "properties": { "additionalProp1": {}, "additionalProp2": {}, "additionalProp3": {} } }
```
- 500**: Internal server error. Example Value:

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

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 for a 500 Internal server error, which includes a JSON schema for a 'Resource Not Found' problem. Below this, the 'Users' section is visible, containing several API endpoints:

- 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 the 'PUT /v1/users/{id}' endpoint documentation. Below the endpoint description, there is a 'Schemas' section listing various data types:

- AddressDTO >
- UserUpdateRequest > (highlighted in orange)
- 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-docs/v2](#)

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](http://localhost:8080) [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
id <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

The screenshot shows the Swagger UI interface for a DELETE endpoint at `/v2/#/Users%20V2/delete`. The main area displays the following information:

- Responses**: A table showing response codes and descriptions.
 - Code**: 204, **Description**: User deleted successfully, **Links**: No links.
 - Code**: 400, **Description**: Invalid ID format, **Links**: No links.
 - Code**: 401, **Description**: Unauthorized - JWT token required, **Links**: No links.
- Example Value**: A dropdown menu with the value `*^`.
- Schema**: A JSON schema for the response:

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

Swagger V2 - Respostas de erro

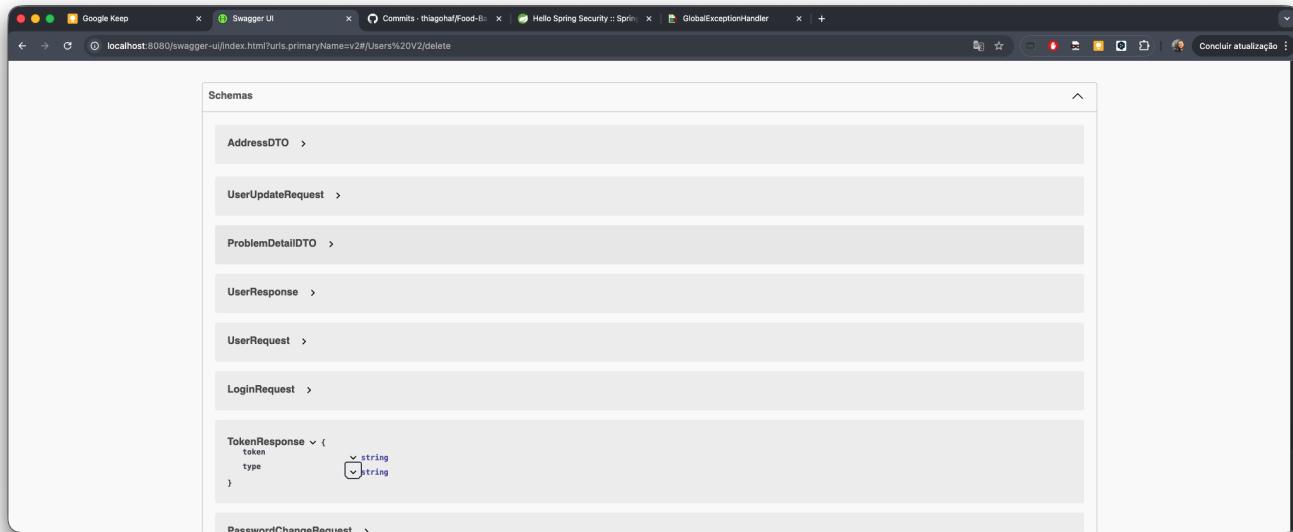
The screenshot shows the Swagger UI interface displaying a list of API endpoints and their corresponding schemas. The endpoints listed are:

- 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)

Below the endpoints, there is a section titled "Schemas" which lists the following:

- 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

Login com usuário não encontrado

The screenshot shows a POST request to `{base_url}/auth/login` with the following JSON body:

```

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

The response status is 404 Not Found, with the following JSON content:

```

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 }
    
```

Acesso não autorizado sem login

The screenshot shows a GET request to `{base_url}/v1/users` with no authentication. The response status is 401 Unauthorized, with the following JSON content:

```

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 }
    
```

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

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case is selected: "GET - Buscar usuário por ID inexistente". The request URL is set to `(base_url)/v1/users/99999`. The response status is 404 Not Found, and the response body is a JSON object with the following content:

```

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 }

```

Testes de Erro - DomainValidationException (400)

Estes testes validam regras de negócio que retornam 400:

Criar usuário com email duplicado

The screenshot shows the Postman interface with a collection named "Food Backend - ProblemDetail Tests". A specific test case is selected: "POST - Criar usuário com email duplicado". The request URL is set to `(base_url)/v1/users`. The request body is a JSON object with the following content:

```

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 }

```

The response status is 400 Bad Request, and the response body is a JSON object with the following content:

```

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 }

```

Testes de Erro - MethodArgumentNotValidException (400)

Estes testes validam as validações de entrada dos DTOs:

Criar usuário com múltiplos campos inválidos

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

```

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

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

```

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    }
  
```

Testes de Casos de Sucesso

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

Criar usuário válido (público)

The screenshot shows the Postman interface with the same collection. A test case titled "POST - Criar usuário válido (público)" is selected. The request URL is `{base_url}/v1/users`. The request body contains the following JSON:

```

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

The response status is 201 Created, with the following JSON response:

```

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

Listar todos os usuários

The screenshot shows the Postman application interface. On the left, the 'Personal Workspace' sidebar lists collections, environments, and flows. A collection named 'Food Backend - ProblemDetail Tests' is selected, containing several sub-folders and endpoints:

- 0 Autenticação
- 1. ResourceNotFoundException (404)
- 2. DomainValidationException (400)
- 3. MethodArgumentNoValidException (400)
- 4. Novos Tratamentos de Erro
- 5. Casos de Sucesso
 - POST POST - Criar usuário válido (público)
 - POST POST - Login (antes de acessar endpoints protegidos)
 - GET GET - Listar todos os usuários
 - GET GET - Buscar usuário por ID
 - GET GET - Buscar usuário por nome
 - GET GET - Buscar usuário por login
 - GET GET - Buscar usuário por email
 - PUT PUT - Atualizar usuário
 - PATCH PATCH - Alterar senha
 - DELETE DELETE - Deletar usuário
- 6. Endpoints V2 - JWT Authentication
- FoodApp
- localhost:8080
- Locatech
- notes

The main workspace displays a successful API call for the endpoint `GET {{base_url}}/v1/users`. The response status is **200 OK**, with a duration of **42 ms** and a size of **1.77 KB**. The response body is as follows:

```
[{"id": 15, "name": "Tania Ferreira", "email": "tania@email.com", "type": "CUSTOMER", "login": "taniasferreira", "addresses": [{"city": "Rio de Janeiro", "number": 456, "street": "Rua Nova", "zipCode": "28000-000"}, {"city": "S\u00e3o Paulo", "number": 123, "street": "Rua Teste", "zipCode": "00000-000"}], "createdAt": "2025-01-02T08:49:35.111Z", "updatedAt": "2025-01-02T08:49:35.111Z"}
```

Testes V2 - Autenticação JWT

A versão 2 utiliza autenticação JWT, e os testes validam o fluxo completo:

Login V2 com sucesso

The screenshot shows the Postman application interface. The left sidebar displays the 'Personal Workspace' with a collection named 'Food Backend - ProblemDetail Tests'. This collection contains several sub-folders and endpoints:

- 0. Autenticação
- 1. ResourceNotFoundException (404)
- 2. DomainValidationException (400)
- 3. MethodArgumentNotValidException (400)
- 4. Novos Tratamentos de Erro
- 5. Casos de Sucesso
- 6. Endpoints V2 - JWT Authentication
- 81 Autenticação V2
 - POST - Login V2 (sucesso)
 - POST - Login V2 (usuário não encontrado)
 - POST - Login V2 (senha incorreta)
 - GET - Acesso não autorizado V2 (sem token)
 - GET - Acesso não autorizado V2 (token inválido)
 - POST - Logout V2 (sucesso)
 - POST - Logout V2 (sem token - erro 401)

The right panel shows a specific endpoint: **POST - Login V2 (sucesso)**. The request method is set to **POST** and the URL is **((base_url)) /v2/auth/login**. The **Body** tab is selected, showing the following raw JSON payload:

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

The response status is **200 OK**, with a duration of **160 ms** and a size of **646 B**. The response body is as follows:

```
1 {
2   "token": "eyJhbGciOiJIUzI1NiJ9.eyJlcnQiOiJxLCjzdWl1O1jb2fc21sdmEiLCJpYXQiOjE3NjgzMjY3MTExMjQwLCJ0Y29sbGVkIjoiZmFyZWJpbmRleCIsImV4cCI6MTc2ODU5MzExMXQ6.0Y71Jep8qoRzsK9731yZUTKwfULRKjMHYIwi1bIVQ01xgPZpCstnVL68jwmKzEBhYtR0_XYqC50",
3   "type": "Bearer"
4 }
```

Acesso não autorizado V2 sem token

The screenshot shows the Postman application interface. On the left, there's a sidebar with sections for 'Collections', 'Environments', 'History', 'Flows', and 'File'. The main area displays a collection named 'Food Backend - ProblemDetail Tests'. Under this, there are several sub-collections: '0. Autenticação', '1 ResourceNotFoundException (404)', '2 DomainValidationException (400)', '3. MethodArgumentNotValidException (400)', '4. Novos Tratamentos de Erro', '5. Casos de Sucesso', and '6. Endpoints V2 - JWT Authentication'. The '6. Endpoints V2 - JWT Authentication' section is expanded, showing a test for 'GET - Acesso não autorizado V2 (sem token)'. The request URL is {{base_url}}/v2/users. The response status is 401 Unauthorized, and the response body is a JSON object:

```

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:8088/v2/users",
7   "properties": {
8     "path": "/v2/users",
9     "method": "GET"
10   }
11 }

```

A aplicação possui cobertura de testes automatizados configurada através do JaCoCo com meta mínima de 80% de cobertura de linhas. Os testes estão organizados no pacote `src/test/java` e incluem testes unitários para services (`UserService`), controllers (`UserControllerV2Test`, `AuthControllerV2Test`), mappers (`UserMapper`), interceptors (`AuthInterceptorTest`), e configurações (`OpenApiConfigTest`). O Maven Surefire Plugin executa os testes durante o build, e o JaCoCo gera relatórios em `target/site/jacoco/` que podem ser verificados para garantir a manutenção da qualidade do código.

5. 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 lists various features like Ask Gordon, Containers, Images, Volumes, Builds, Models, MCP Toolkit, Docker Hub, Docker Scout, and Extensions. The 'Containers' section is selected. At the top, there are statistics for Container CPU usage (0.30% / 1100%) and Container memory usage (372.81MB / 7.47GB). A search bar and a filter option 'Only show running containers' are present. Below is a table listing the three containers:

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

At the bottom, it says 'Showing 3 items'. The status bar at the bottom shows 'Engine running', 'RAM 4.80 GB CPU 0.09%', 'Disk 9.90 GB used (limit 452.13 GB)', and 'Terminal New version available'.

Docker - Detalhes dos containers no IntelliJ IDEA

The screenshot shows the IntelliJ IDEA interface with the Docker Compose Services tool window open. The tree view on the left shows services: Docker-compose: docker, Docker-compose: docker-local, Docker-compose: food-backend, Docker-compose: food-backend_default, and Docker-compose: postgres. The 'food-postgres' service is selected. To the right, a terminal window shows logs for the 'food-postgres' container:

```

2026-01-11 04:58:36.797 UTC [95686] ERROR: syntax error at or near "Maria Santos" at character 44
2026-01-11 04:58:36.797 UTC [95686] STATEMENT: delete from public.tb_users where name is "Maria Santos"
2026-01-11 04:58:59.045 UTC [95686] ERROR: syntax error at or near "23" at character 42
2026-01-11 04:58:59.045 UTC [95686] STATEMENT: delete from public.tb_users where id is 23
2026-01-11 04:53:38.715 UTC [27] LOG: checkpoint starting: time
2026-01-11 04:53:39.191 UTC [27] LOG: checkpoint complete: wrote 5 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.421 s, sync=0.005 s

```

The status bar at the bottom indicates '59:27 (72 chars) LF UTF-8 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 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.

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