

Projeto: ShareFood

Alessandro Schneider - RM 363307
Eduardo Fernando Serafim Santos - RM 361523
Raquel Morabito - RM 362871
Henrique Danzo Baria - RM 363256
Natan Campos - RM 364605

1. Introdução

Descrição do problema

Diante do alto custo de desenvolvimento de sistemas individuais para cada cliente, o sistema proposto visa criar um *hub* de gestão para diversos restaurantes. Isso deve permitir que clientes e donos de restaurantes compartilhem uma só plataforma de acesso único, porém personalizada para cada cliente e restaurante.

Objetivo do projeto

Desenvolver um backend robusto utilizando Java e Spring para gerenciar restaurantes, itens do cardápio e usuários, atendendo aos requisitos definidos pelos *stakeholders* do projeto.

2. Arquitetura do Sistema

Descrição da Arquitetura

O sistema será desenvolvido utilizando a linguagem Java e Spring Framework, utilizando a **separação em camadas** conforme definido nas boas práticas de “Clean Architecture”, no projeto utilizamos 3 pacotes principais, sendo:

- **application** - é responsável por manter a lógica e orquestração para funcionamento da aplicação, bem como sua correta integração com as demais camadas.
- **domain** - contém apenas o necessário para que a aplicação utilize as regras e os modelos definidos aqui, é isolada e não possui acesso às camadas externas a ela.
- **infra** - aqui foram definidas as estruturas de apoio ao funcionamento e integração da aplicação, como bancos de dados, controladores REST, segurança e outras configurações de *frameworks*.

APPLICATION

- **Input/Output** - será responsável por receber das camadas exteriores as entradas que serão utilizadas pelos *use cases*, da mesma forma, enviarão os dados no formato *output* para a saída, inibindo qualquer acoplamento com a *domain* ou com o DTO's da camada de apresentação;
- **Gateways** - aqui são criadas as rotas de utilização obrigatória por parte da *infra* e da *application* para acessar outros pontos da aplicação (ou até de outras aplicações), é através do gateway que são transitados os dados e realizadas solicitações inter-camadas, as injeções sempre tratam com interfaces, apenas esperando uma especificação e não a implementação concreta, isso reduz o acoplamento e facilita a mudança em partes específicas do projeto caso necessário.
- **Exceptions** - são exceções lançadas pelos *use cases* como regras de validação ou para tratamentos de erros.
- **UseCases** - aqui temos todas regras de negócio de cada entidade, cada *use case* é responsável por executar uma função, conforme princípios SOLID, recebendo em seu construtor o *gateway* que poderá utilizar para acessar outras camadas da aplicação. Note que o *use case* não tem injeção ou vínculos com *data sources*, já que este não sabe para onde será enviada a solicitação, tampouco sabe quem irá implementar o *gateway* para o qual envia suas necessidades. Caso haja necessidade por exemplo de implementar um novo *data source* a camada *application* não será afetada.

DOMAIN

- **Interfaces** - aqui definimos as interfaces necessárias para os *gateways* e os *data sources* que serão usados na camada de application;
- **Models** - são o coração da aplicação, são utilizados pela camada *application*, porém não tem conhecimento de nada externo ao seu pacote; apenas definem validações próprias de criação ou outras regras exclusivas ao seu modelo;

INFRA

- **Config** - aqui foram definidas configurações gerais para o funcionamento da infra, como criação de *factorys*, *swagger* e *security* do Spring;
- **Controller** - é responsável por toda a comunicação com a interface que consumirá a API (*client*), toda requisição direcionada ao *app* deve ser tratada inicialmente aqui e repassada para as demais camadas, recebem e devolvem DTO's. Aqui temos a conversão da *request* como DTO para a entrada definida pela *application*, *input*, e também o processo inverso, recebemos a saída (*output*) da *application* e convertemos em um novo DTO para devolver ao usuário.
- **Database** - até o momento temos neste projeto uma conexão com banco relacional Postgres, que foi definida aqui através com uso de JPA e Hibernate, porém caso hajam novas implementações, como MongoDB, um servidor de arquivos, ou outro, podemos criar aqui.
 - Entities - entidades do JPA, recebem ;
 - Repositories - repositórios do JPA, que implementam a interface conforme o

data source, aqui temos a conversão das entidades entre camadas, realizando a conversão para entidades do banco de dados e devolvendo para o gateway o objeto convertido novamente; aqui separamos o pacote em adapters e jpa, ficando o jpa responsável pelo acesso direto ao banco utilizando framework e as classes do adapter responsáveis pela implementação e conversão de objetos;

- Mappers - classes auxiliar para conversão entidades domínio <-> jpa utilizando o MapStruct;
- **DTO's** - fornecerá a interface entre as entidades reais e os clientes, desta forma mantemos o baixo acoplamento e evitamos quebras de contrato entre API e clientes que já consomem os *endpoints*.
- **DOC** - para evitar a poluição dos *rest controllers* com as anotações do *swagger* em cada endpoint, criamos o pacote *doc* para que seja definida aqui a documentação, assim as definições do *swagger* já estarão na classe pai e classe do controller apenas preocupa-se em tratar as chamadas recebidas;
- **Security** - neste pacote estão definidos utilitários para autenticação/autorização e geração e validação de tokens.

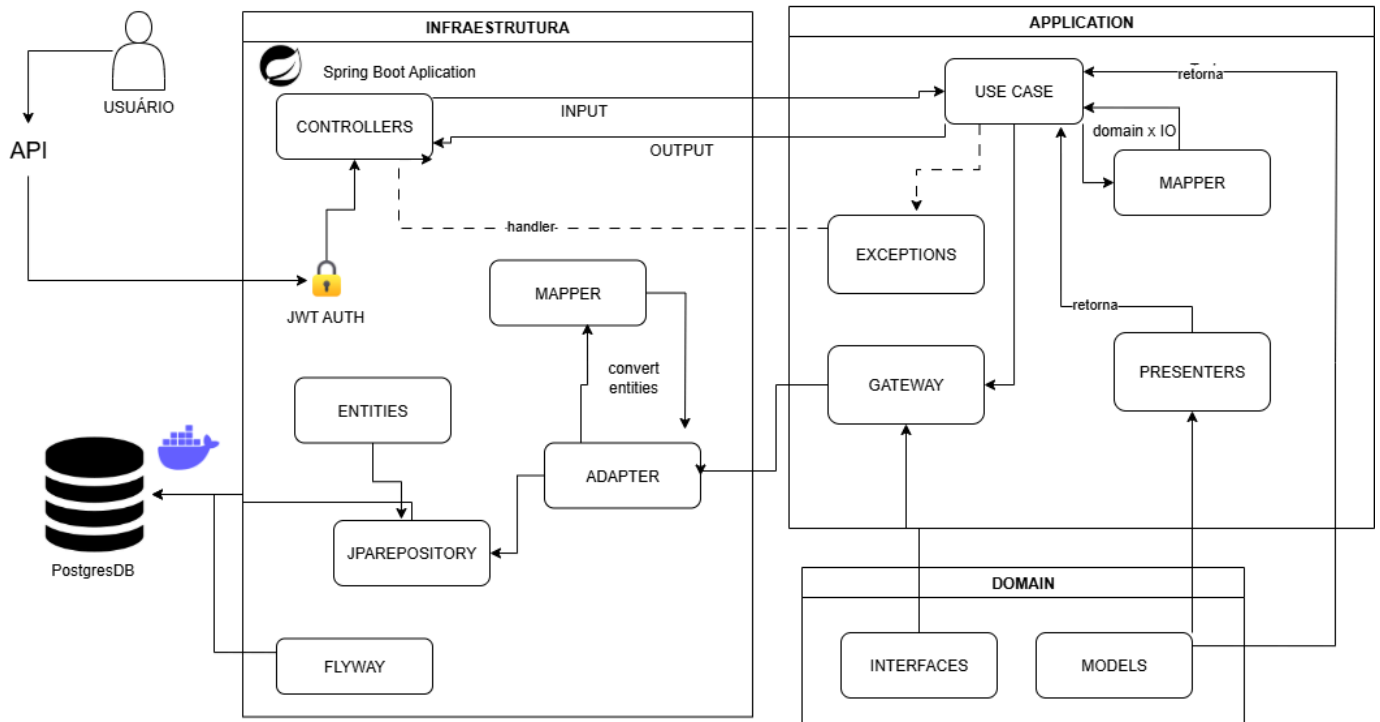
Além disso, foi utilizada injeção de dependência para inversão do controle (*IoC*), onde o próprio Spring faz o gerenciamento dos objetos injetados (*gateways*, *repositories*), com isso temos um código mais limpo, flexível e independente.

Para dar agilidade à configuração da aplicação foram utilizados alguns componentes essenciais no projeto:

- **Maven** - com o gerenciamento de dependências fornecido pelo Maven todas as configurações necessárias quanto a isso serão facilmente configuradas no arquivo *pom.xml*, agilizando o trabalho da equipe e evitando quebra do código em diferentes ambientes;
- **Spring Boot** - o projeto inicial foi criado com o Spring Boot, abstraindo grande parte das configurações iniciais para rodar o projeto, como um servidor embutido dentre outras facilidades;
- **Docker** - para que a aplicação seja escalável e, independente da configuração de ambiente foi utilizado o docker para sua execução, assim toda a aplicação pode ser empacotada e compartilhada.
- **Flyway** - o Flyway foi utilizado para criação inicial do banco de dados bem como versionamento da modelagem do banco, não foi utilizada a geração automática de tabelas do Hibernate. Com o Flyway a tabela de usuários foi previamente populada com alguns dados iniciais para teste.
- **MapStruct** - este gerador de código facilita muito o trabalho ao realizarmos conversões entre as classes, após definirmos as conversões necessárias (fora do padrão) ou o que deve ser ignorado, ele gera todo o código de conversão, tornando o processo transparente.

- **Mockito** - foi utilizado na geração de testes;
- **AssertJ** - Biblioteca que facilita as validações na hora dos testes
- **Rest Assured** - Biblioteca utilizada para fazer os testes de integração na camada Controller da aplicação

Diagrama da Arquitetura



3. Descrição dos endpoints da API

AUTENTICAÇÃO		
ENDPOINT	MÉTODO	DESCRIÇÃO
/auth/login	POST	Recebe um login/senha para realizar a autenticação do usuário.

/auth/validar-token	POST	Recebe o token via header authorization e verifica se o mesmo está vigente ou não.
USUÁRIO		
ENDPOINT	MÉTODO	DESCRIÇÃO
/usuario	GET	Retorna uma lista com todos os usuários cadastrados.
/usuario/{id}	GET	Retorna o usuário com o id enviado via param.
/usuario	POST	Recebe no corpo da requisição um json contendo os dados para cadastro de um novo usuário.
/usuario/{id}	PUT	Recebe no corpo da requisição os dados para atualização de um usuário, enviando na request como parâmetro o ID do usuário a ser atualizado.
/usuario/{id}	DELETE	Recebe via parâmetro da requisição o ID do usuário a ser deletado.
/usuario/{id}/mudar-senha	PATCH	Recebe via parâmetro da requisição o ID do usuário que deve ter a senha alterada e no corpo da requisição a senha antiga e a nova.
PERFIL DE USUÁRIO		
ENDPOINT	MÉTODO	DESCRIÇÃO
/usuario/{id}/perfil	GET	Recebe via parâmetro da requisição o ID e retorna todos os perfis que atualmente estão atribuídos a ele.
/usuario/{id}/perfil	POST	Recebe via parâmetro da requisição o ID do usuário a e no corpo da requisição uma lista com os perfis a serem atribuídos para ele.
/usuario/{id}/perfil	DELETE	Recebe via parâmetro da requisição o ID do usuário a e no corpo da requisição uma lista com os perfis a que devem ser removidos do usuário.
RESTAURANTE		
ENDPOINT	MÉTODO	DESCRIÇÃO
/restaurante	GET	Retorna uma lista com todos os restaurantes cadastrados.
/restaurante	POST	Recebe no corpo da requisição um json contendo os dados para cadastro de um novo restaurante.

/restaurante/{id}	PUT	Recebe no corpo da requisição os dados para atualização de um restaurante, enviando na request como parâmetro o ID do restaurante a ser atualizado.
/restaurante/{id}	DELETE	Recebe via parâmetro da requisição o ID do restaurante a ser deletado.

CARDÁPIO		
ENDPOINT	MÉTODO	DESCRIÇÃO
/restaurante/{id}/cardapio	GET	Retorna uma lista com todos os itens no cardápio do restaurante, parâmetro ID refere-se ao restaurante do qual estamos buscando o cardápio.
/restaurante/{id}/cardapio	POST	Recebe no corpo da requisição um json contendo os dados do item do cardápio para cadastro no menu do restaurante.
/restaurante/{id}/cardapio/{idItem}	PUT	Recebe no corpo da requisição um json contendo os dados do item do cardápio para atualização e também o ID do restaurante, só é permitido editar itens do próprio restaurante.
/restaurante/{id}/cardapio/{idItem}	DELETE	Recebe via parâmetro da requisição o ID do item a ser deletado, só é permitido alterar itens do próprio restaurante.

Exemplos de requisição e resposta

ENDPOINT	MÉTODO	RESPOSTA
/<entidade>	GET	200 - OK - Lista de <entidade>
/<entidade>	POST	201 - CREATED - Objeto <entidade>
/<entidade>/{id}	PUT	200 - OK - Objeto <entidade>
/<entidade>/{id}	DELETE	200 - OK - se deletado

ENDPOINT	MÉTODO	RESPOSTA
/<entidade>	GET	200 - OK - Lista de <entidade>
/<entidade>	POST	201 - CREATED - Objeto <entidade>
/usuario/mudar-senha/{id}	PATCH	204 - NO_CONTENT - se alterada
/auth/login	POST	200 - OK - JSON contendo token e informações do usuário.
/auth/validar-token	GET	200 - OK 401 - UNAUTHORIZED
Todos endpoints	*	400 - BAD_REQUEST - erros de validação 400 - BAD_REQUEST - em caso de erros 401 - UNAUTHORIZED - login/senha incorretos ou não autorizado

4. Configuração do Projeto

Configuração do Docker Compose

Para facilitar a execução e testes do projeto localmente foram criados 2 arquivos docker-compose. O primeiro pode ser executado para subir o projeto e o segundo caso seja necessário utilizar o pgAdmin para visualizar/gerenciar o banco de dados.

O arquivo principal **docker-compose.yml** contém todas as instruções para que o projeto seja executado sem que sejam necessárias configurações adicionais de porta, banco de dados ou da aplicação.

Neste definimos a criação das seguintes imagens:

- **postgres** - utilizando a imagem do *postgres:15*, este container servirá como base de dados para a aplicação, foi exposta na porta 5432 com usuário e senha padrão, apenas definimos o nome do banco de dados como **restaurante**;
- **app** - imagem da aplicação Java criada, será exposta na porta 8080, depende da criação do container **postgres** para que seja executado;
- **volumes - pgdata** foi montado para que as informações do banco de dados sejam persistidas mesmo após um container ser excluído ou reiniciado;
- **networks** - esta definição foi necessária para que haja a correta comunicação entre os containers definidos acima, sem uma rede própria todos os *containers* ficariam isolados e portanto não seria possível por exemplo acessar o pgAdmin do banco de dados.

No arquivo **Dockerfile** foram definidas as regras para empacotamento e compilação da versão final do projeto.

Instruções para execução local

Utilize o docker com o comando abaixo para subir o projeto.

> **docker compose up -d**

URL inicial da aplicação.

<http://localhost:8080/>

URL da documentação do Swagger.

<http://localhost:8080/swagger-ui/index.html>

5. Qualidade do código

Boas Práticas Utilizadas

- **Clean Architecture** - A Clean Architecture promove a separação clara entre responsabilidades, baixo acoplamento e alta coesão. Funcionando da seguinte maneira: A lógica de negócio está isolada na camada *domain*, os casos de uso são organizados na *application*, e a infra contém as implementações técnicas como controllers, repositórios e configurações. Dessa forma, ela permite com que o sistema possa evoluir com um impacto baixo entre as partes.
- **Convenções Spring** - o Spring adota o lema de convenções sobre configurações, utilizando o Spring a detecção de componentes é automática quando utilizamos os padrões definidos por ele, como no caso de *Controllers*, *Services*, *Repositories*, *Component* e *DTO's*, isso evita configurações manuais, apesar de serem possíveis.
- **Spring JPA** - a padronização de nomes nos *repositories* também facilita a criação de consultas, dispensando por vezes a criação de *queries* manuais.
- **SOLID**
 - Single Responsibility Principle (SRP) - separação em camadas como já comentado nas convenções do Spring (*controller*, *service*, *repositories*)
 - Dependency Inversion Principle (DIP) - com as anotações para injeção de dependência o Spring cuidará da criação da classe no momento correto
 - Outros princípios serão aplicados conforme o projeto evoluir.

- **Injeção de dependência** - o Spring fornece injeção de dependências, dispensando o controle de criação de classes, basta utilizar o `@Autowired`.
 - **BCrypt** - importante elemento de segurança para armazenar senhas criptografadas,, o **bcrypt** implementa boas práticas ao realizar **salt** em seu algoritmo, assim cada hash gerado é diferente mesmo que a entrada seja igual.
 - **REST Full** - como boa prática não guardamos o estado da sessão no servidor, sendo todas as informações necessárias enviadas a cada requisição.
 - **DTO - Data Transfer Object Pattern** - este padrão foi utilizado para que não sejam expostos dados sensíveis para o *client* que está consumindo a API, bem como para abstrair informações diversas em um só objeto de retorno
 - **Chain of Responsibility** - o padrão de *filters* que o Spring utiliza para autenticação e autorização nos módulos *security* e *web* implementa este padrão, repassando a responsabilidade para cada filtro decidir se deve ou não processar ou passar para frente a solicitação.
-

6. Collections para teste

Link para a collection do Postman

Dentro da pasta “**docs**” do projeto foram disponibilizadas as *collections* para importação no Postman através de um arquivo JSON, nele estão configuradas as *requests* para execução, com alguns dados de exemplo.

Descrição dos testes manuais

Para validar os *endpoints* é necessário realizar a importação no Postman do arquivo JSON disponibilizado, com isso os parâmetros iniciais estarão preenchidos e as *requests* disponíveis. Será necessário realizar a autenticação inicial para que as outras URL's possam ser requisitadas, através da URI “/auth”.

7. Testes automatizados

O projeto apresenta uma estrutura de testes automatizados que cobrem os principais fluxos da aplicação. Estes testes estão em conformidade aos princípios da Clean Architecture. Os testes cobrem de forma abrangente os casos de uso na camada *application*, os *adapters* de persistência em *infra.database.repositories.adapter* e os controladores REST na *infra.controller*.

As bibliotecas utilizadas incluem JUnit 5, Mockito, AssertJ, Rest Assured e Spring MockMvc, com apoio de uma classe utilitária (Helper) que padroniza a criação de entidades e dados para testes. A cobertura de código foi avaliada e atende à meta mínima exigida de 80% de teste unitários, sem considerar ainda os testes de integração realizados via Postman. Para rodar os testes, é necessário que o comando seja o:

> mvn verify

com esse comando, o Maven executará todos os testes unitários, se baseando nas classes que terminam com o sufixo “Test” e os testes de integração, se baseando nas classes que terminam com o sufixo “IT”.

8. Repositório do código-fonte

Disponível em: <https://github.com/aschneider12/tech21>

9. Vídeo de apresentação

Disponível em: [Google Drive - Vídeo de apresentação](#)

https://drive.google.com/file/d/1W_WagXnEuwkzYLaVTdG9mHI6NxvmKy3k/view?usp=sharing