

Projeto: ShareFood

Equipe

Alessandro Schneider - RM 363307
Eduardo Fernando Serafim Santos - RM 361523
Raquel Morabito - RM 362871
Henrique Danzo Baria - 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 usuários e atender 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** em: Controller, Service, Repository, Entities e DTO.

- **Controller** - será 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;
- **Service** - a lógica de negócios é aplicada nesta camada, servirá como intermediadora da camada externa (*controller*) com a camada de manipulação de dados (*repository*);
- **Repository** - a camada de manipulação dos dados deve ter seu acesso restrito, o *controller* não deverá acessar a camada *repository*, somente a camada *service*, isso é necessário para que seja mantido o baixo acoplamento do projeto.
- **DTO** - 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*.

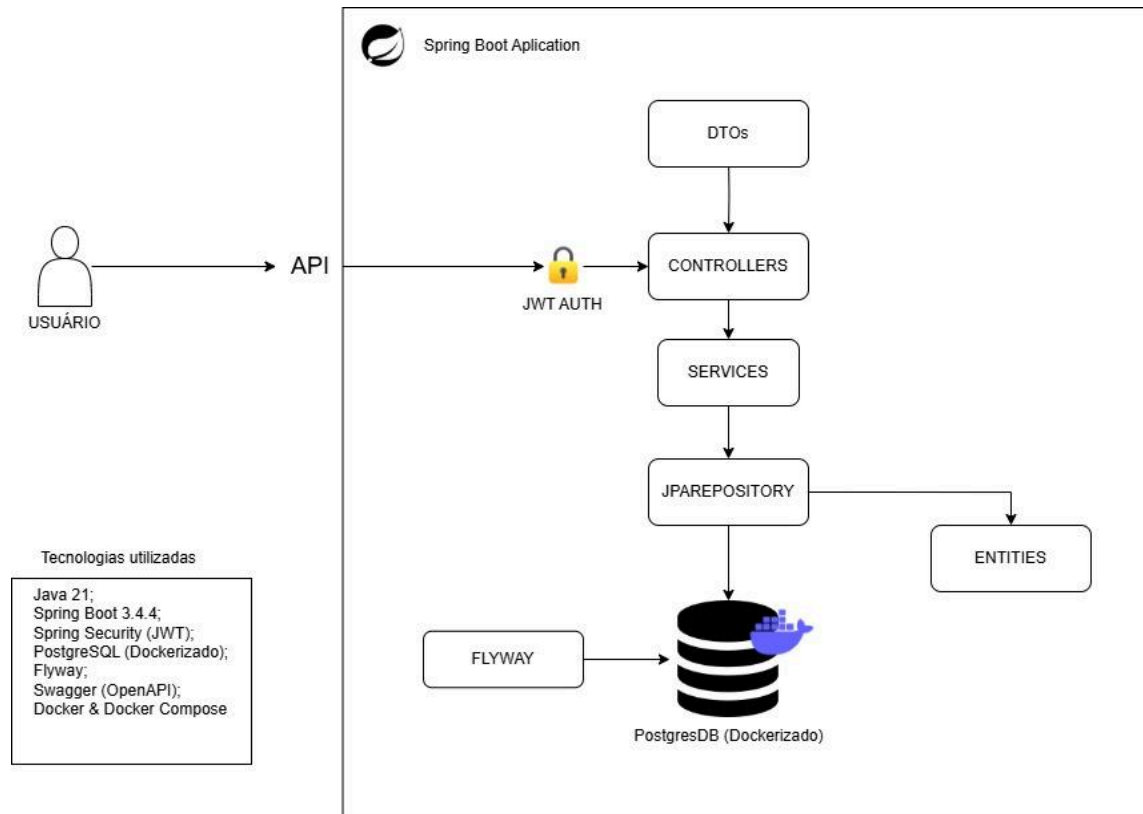
- **Entities** - neste pacote serão adicionadas representações em objetos dos modelos de dados do banco relacional, utilizando o JPA / Hibernate para mapeamento dos atributos.

Além disso, será utilizada injeção de dependência para inversão do controle (*IoC*), onde o próprio Spring faz o gerenciamento dos objetos injetados (*services, 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.

Diagrama da Arquitetura



3. Descrição dos Endpoints da API

Tabela de Endpoints

ENDPOINT	MÉTODO	DESCRIÇÃO
/usuario	GET	Retorna uma lista com todos os usuários cadastrados.
/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.

ENDPOINT	MÉTODO	DESCRIÇÃO
/usuario	GET	Retorna uma lista com todos os usuários cadastrados.
/usuario	POST	Recebe no corpo da requisição um json contendo os dados para cadastro de um novo usuário.
/mudar-senha/{id}	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.
/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.

Exemplos de requisição e resposta

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

URL inicial da aplicação.

<http://localhost:8080/>

URL da documentação do Swagger.

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

URL do endpoint de usuários.

<http://localhost:8080/usuario>

5. Qualidade do Código

Boas Práticas Utilizadas

- **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.

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.

7. Repositório do Código

URL do repositório

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