UNIVERSIDADE FEDERAL DA FRONTEIRA SUL - *CAMPUS* CHAPECÓ
CURSO DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO
ENGENHARIA DE SOFTWARE I
DOCENTE: NEIMAR MARCOS ASSMANN

MARCO ANTONIO BERNARDELI DA VEIGA

PROJETO INTEGRADOR:

RELATÓRIO TÉCNICO DO BACKEND

SUMÁRIO

1 Arquitetura Geral

O sistema foi projetado com uma arquitetura de microsserviços desacoplada, separando completamente as responsabilidades do backend e do frontend. A comunicação entre as camadas é feita exclusivamente através de requisições HTTP, com o backend expondo endpoints REST que o frontend utiliza para buscar, criar, editar e deletar dados.

1.1 Backend (Java/Spring Boot)

O foco deste relatório, o backend foi desenvolvido como uma API RESTful. Ele é o cérebro da aplicação, responsável por toda a lógica de negócio, segurança, manipulação de dados e comunicação com o banco de dados.

1.2 Frontend (React)

O cliente que consome a API. Ele é responsável por toda a interface do usuário, desde a homepage pública até o painel de gerenciamento administrativo.

1.3 Banco de Dados (PostgreSQL/H2)

O sistema foi projetado para ser compatível com o PostgreSQL para produção, utilizando um banco de dados em memória H2 para testes e desenvolvimento inicial, configurado através do Spring Boot.

2 Detalhes da Implementação do Backend

A seguir, uma análise aprofundada das decisões, lógicas e componentes que estruturam a aplicação backend.

2.1 Configuração do Ambiente e Dependências

A fundação do projeto é gerenciada pelo **Maven**. O arquivo `pom.xml` é onde declaramos todas as bibliotecas que nosso projeto precisa. As dependências primárias do Spring Boot escolhidas foram:

- `spring-boot-starter-web`: Fornece um servidor web **Tomcat embutido** e o suporte essencial para a criação de APIs REST.
- `spring-boot-starter-data-jpa`: Abstrai a complexidade da comunicação com o banco de dados, integrando o **Hibernate** como provedor de ORM.

- `spring-boot-starter-security`: Fornece a base de segurança, permitindo a criação da entidade `Usuario` e preparando o terreno para futuras funcionalidades de login.

As configurações de ambiente são definidas no arquivo **src/main/resources/application.properties**.

2.2 Estrutura de Pacotes: A Arquitetura em Camadas

A organização do código seguiu o padrão de arquitetura em camadas para garantir a separação de responsabilidades.

- **`model.classes` (Camada de Modelo):** Representa as tabelas do banco de dados como classes Java (`@Entity`).
- **`repositories` (Camada de Repositório):** Contém as interfaces que estendem `JpaRepository`, sendo a única camada que acessa diretamente o banco de dados.
- **`service` (Camada de Serviço):** Orquestra a lógica de negócio da aplicação.
- **`controller` (Camada de Controle):** Expõe os endpoints da API para o frontend.
- **`dto` (Data Transfer Objects):** Serve como um "contrato"de dados entre o backend e o frontend.
- **`config` (Configurações):** Centraliza configurações transversais, como CORS e o serviço de arquivos estáticos.

2.3 Modelo de Dados e Decisões de Persistência

O design do banco de dados foi pensado para ser flexível e robusto.

- **Atributo Embutido (`@Embedded`):** A decisão de usar `@Embedded` para a classe `Endereco` foi baseada no requisito de que um `Imovel` e um `Proprietario` teriam apenas um endereço. Endereço foi criado com a intenção de representar um atributo multivalorado.
- **Relacionamento Muitos-para-Muitos (`@ManyToMany`):** A relação entre `Imovel` e `Caracteristica` é resolvida através de uma tabela de junção, criada automaticamente pelo Hibernate.
- **Relacionamento Um-para-Muitos (`@OneToMany`):** A relação entre `Imovel` e `Imagem` é um exemplo de um-para-muitos, onde um imóvel pode ter várias imagens.

2.4 Funcionalidades de Negócio e Endpoints da API

O backend oferece uma API REST completa para todas as operações CRUD.

- **`ImovelController`:** Fornece os endpoints padrão para gerenciar imóveis e uma busca customizada (`/por-proprietario/{proprietarioId}`).
- **`FileUploadController`:** Criado especificamente para lidar com o upload de imagens.
- **Controllers Adicionais:** Foram criados controllers para todas as outras entidades (`Usuario`, `Bairro`, etc.), garantindo uma API consistente.

2.5 Configurações Transversais (CORS e Servir Arquivos)

A classe `WebConfig` foi utilizada para configurar a integração com o frontend.

- **CORS (Cross-Origin Resource Sharing):** O método `addCorsMappings` foi configurado para permitir a comunicação entre o frontend e o backend, incluindo métodos como `PUT`, `DELETE` e `PATCH`.
- **Servir Imagens Estáticas:** O método `addResourceHandlers` foi implementado para mapear a URL `/uploads/**` para o diretório físico no servidor onde as imagens são salvas.

3 Banco de Dados

A persistência de dados do sistema é gerenciada pelo **PostgreSQL**, com o **Hibernate** (via Spring Data JPA) atuando como a camada de abstração (ORM).

3.1 Gerenciamento do Esquema

A propriedade `spring.jpa.hibernate.ddl-auto: update` no arquivo `application.properties` foi uma decisão estratégica para agilizar o desenvolvimento. Ela instrui o Hibernate a comparar o modelo de entidades Java com o esquema do banco de dados a cada inicialização e a aplicar as alterações necessárias automaticamente.

3.2 Mapeamento de Tipos: Java vs. SQL

Houve uma tradução de tipos entre a linguagem Java e o SQL do banco de dados:

- `Long` em Java é mapeado para `BIGINT` no PostgreSQL.
- `String` em Java é mapeado para `VARCHAR(255)` por padrão.

- `Double` em Java é mapeado para `DOUBLE PRECISION`.
- `Boolean` em Java é mapeado para `BOOLEAN`.

3.3 Representação de Relacionamentos no Banco

A seguir, como as anotações JPA se traduzem para a estrutura do banco:

- **`@Embedded` (`Endereco`):** Como `Endereco` é uma classe embutida, ela **não gera uma tabela separada**. Seus campos são criados como colunas diretamente dentro das tabelas `imovel` e `proprietario`.
- **`@ManyToOne` (`Imovel -> Proprietario`):** Este relacionamento é representado por uma única coluna de chave estrangeira na tabela "muitos". A tabela `imovel` contém uma coluna `proprietario_id` que armazena o `id` do proprietário correspondente.
- **`@ManyToMany` (`Imovel <-> Caracteristica`):** Este relacionamento é resolvido através de uma **tabela de junção** intermediária, chamada `imovel_caracteristica`, que contém apenas as colunas `imovel_id` e `caracteristica_id`.

4 Conclusão

O desenvolvimento do backend para o sistema de gestão imobiliária foi um exercício prático e aprofundado na construção de APIs REST com Spring Boot. As decisões de arquitetura, como a separação em camadas e o uso de DTOs, provaram ser eficazes para criar um código organizado, seguro e fácil de manter. Os desafios encontrados, como a configuração de CORS, o tratamento de relacionamentos complexos e a lógica de upload de arquivos, foram oportunidades valiosas de aprendizado que resultaram em um backend robusto e funcional, pronto para servir como uma base de uma possível aplicação completa no futuro. Link para o repositório no GitHub onde está todo o projeto, assim como o restante da documentação: https://github.com/Panoise/Projeto-Integrador/tree/main