JPA

Linguagem Orientada a Objetos

Prof. Rafael Tápia

1. Estrutura do Projeto

Modelo de Camadas: Nosso projeto está organizado em diferentes camadas para separar responsabilidades. Essas camadas são:

- **Controller**: Lida com as **requisições HTTP** e responde ao usuário.
- **Service**: Contém a **lógica de negócio**, ou seja, o que o sistema realmente faz.
- **Repository**: Faz a **comunicação com o banco de dados**.

Essa divisão nos ajuda a manter o código organizado, e cada parte do sistema tem sua responsabilidade clara.

Além disso, se precisarmos mudar a lógica de negócio ou o banco de dados, podemos fazer isso sem afetar todo o sistema.

2. Conexão com o Banco de Dados

Spring Data JPA: É uma ferramenta que facilita o acesso ao banco de dados. Com ela, não precisamos escrever SQL manualmente toda vez que queremos salvar ou buscar dados. O JPA (Java Persistence API) cuida disso por nós.

MariaDB: É o banco de dados que estamos utilizando. Ele armazena as informações dos clientes, como nome, e-mail, telefone, etc.

MariaDB é praticamente igual ao MySQL porque ambos compartilham a mesma base e funcionam de maneira muito semelhante. A principal diferença é que ***MariaDB é totalmente open source****, enquanto o* ***MySQL é controlado pela Oracle****. Para a maioria dos usos, eles são intercambiáveis.*

Como nos conectamos ao banco?

No arquivo de configuração (**application.properties**), configuramos o endereço do banco, o nome do banco de dados, usuário e senha. O Spring Boot usa essas informações para fazer a conexão automática.

Aqui está um **exemplo** de configuração:

spring.datasource.url=jdbc:mariadb://localhost:3307/DBClientes
spring.datasource.username=root
spring.datasource.password=123

3. Classes e Camadas no Projeto

Cliente

A classe **Cliente** representa a tabela Clientes no banco de dados. Cada objeto Cliente é uma linha dessa tabela.

Quando falamos de um objeto, podemos pensar em um cliente específico, com nome, e-mail, e outros dados.

3. Classes e Camadas no Projeto

Repository

O **ClienteRepository** é a interface que comunica nossa aplicação com o banco de dados. Ela é responsável por buscar, salvar, e deletar informações no banco. Tudo isso é feito automaticamente com o Spring Data JPA.

Exemplo de uso:

```
clienteRepository.findById(1); // Busca o cliente com ID 1 no banco de dados
```

3. Classes e Camadas no Projeto

Service

A camada de Serviço (**Service**) organiza a lógica do sistema. A classe **ClienteService**, por exemplo, define as regras para buscar ou criar um cliente e lida com possíveis erros, como quando um cliente não é encontrado.

Antes, utilizamos a nomenclatura "**Core**" para essas classes, mas agora usamos "**Service**" propositalmente.

Não se prenda aos nomes, o importante é entender as funcionalidades.

Poderíamos também usar "**Business**", "**Domain**", "**Manager**", "**Handler**", entre outros, todos com o mesmo objetivo: **controlar as regras de negócio**.

3. Classes e Camadas no Projeto

Controller

O **Controller** é o responsável por receber as requisições do usuário.

Por exemplo, quando você acessa uma URL para buscar ou cadastrar um cliente, o Controller lida com isso.

Ele chama o Service, que por sua vez fala com o Repository para interagir com o banco de dados.

4. Por que Precisamos de Diferentes Classes?

Separação de Responsabilidades: Cada classe tem um papel específico. Isso torna o código mais fácil de entender e de modificar no futuro.

- Controller só lida com a entrada e saída de dados (requisições HTTP).
- Service cuida da lógica de negócios, ou seja, o que a aplicação deve fazer.
- Repository é responsável por interagir com o banco de dados.

Essa separação também facilita a **manutenção** e o **reuso** de código.

5. Como o JPA Funciona?

O **JPA (Java Persistence API)** é uma ferramenta que mapeia objetos Java para tabelas do banco de dados. Com ele, você pode trabalhar com **objetos Java** e **o JPA traduz** essas operações para SQL de forma automática.

No nosso caso, a classe **Cliente** está anotada com **@Entity**, que diz ao JPA que essa classe está relacionada a uma tabela no banco de dados.

Quando salvamos um Cliente, por exemplo, o JPA transforma esse objeto em um comando SQL que insere os dados na tabela.

Spring Data JPA é uma extensão que já faz muito do trabalho pesado por nós, como criar os métodos de consulta e salvar dados, sem que precisemos escrever SQL.

5. Como o JPA Funciona?

Exemplo:

A classe **Cliente** é uma entidade, mapeada para a **tabela clientes**. Quando chamamos o método **clienteRepository.save(cliente),** o JPA cria um comando SQL para inserir ou atualizar esse cliente no banco de dados.

6. Resumindo o Fluxo do Sistema

1. O usuário faz uma requisição HTTP (por exemplo, acessando uma URL para buscar um cliente).
2. O **Controller** recebe essa requisição e chama o Service.
3. O **Service** decide o que deve ser feito (buscar, salvar, ou atualizar um cliente).
4. O **Service** usa o **Repository** para falar com o banco de dados e executar as operações.
5. O **JPA** transforma os **objetos Java** em comandos **SQL** e interage com o **banco**.
6. O resultado é enviado de volta ao usuário.

Exemplos

Inserção de cliente:

Quando criamos um novo cliente e chamamos o método **save**(), o **JPA** gera um **INSERT INTO** no banco.

Consulta de cliente:

Quando buscamos um cliente pelo **ID**, o **JPA** gera um **SELECT FROM** na **tabela** e retorna o cliente correspondente como um objeto Java.

EXERCÍCIOS

Inclusão de Novos Atributos na Classe "Cliente"

Agora que já compreendemos a estrutura básica do projeto e o funcionamento da conexão com o banco de dados, é hora de expandir a nossa classe Cliente com novos atributos. Atualmente, a classe possui os seguintes campos:

- id (Integer)
- nome (String)
- email (String)
- telefone (String)

Tarefa:

Incluir os seguintes atributos na classe Cliente:

- **cpf** (String): Número do CPF (Cadastro de Pessoa Física).
- **endereco** (String): Endereço completo do cliente.
- **dataDeNascimento** (LocalDate): Data de nascimento do cliente.
- **dataDeCadastro** (LocalDateTime): Data e hora em que o cliente foi cadastrado no sistema.

Passos para a Solução:

Modificar a Classe Cliente:

- Adicionar os novos atributos à classe Cliente.
- Utilizar anotações do JPA (como @Column) para mapear esses novos atributos ao banco de dados.

Atualizar o Banco de Dados:

- O Spring Data JPA vai gerar automaticamente as colunas adicionais no banco de dados, devido à configuração **spring.jpa.hibernate.ddl-auto=update**. Certifique-se de que essa propriedade está ativada no seu arquivo **application.properties**.

Testar as Novas Funcionalidades:

- Após incluir os novos atributos, crie um novo registro de cliente, preenchendo todos os campos, para verificar se os novos dados estão sendo salvos corretamente no banco.

Dicas

Para o **cpf**, utilize uma **String**, mas lembre-se que em um cenário real pode ser interessante validar o formato do CPF.

O campo **endereco** pode ser apenas um texto simples, mas, em sistemas mais complexos, poderia ser uma classe separada com vários detalhes (rua, número, cidade, etc.).

Para o campo **dataDeNascimento**, use o tipo **LocalDate** do Java 8, que é ideal para trabalhar com datas.

CRUD Simples de Clientes

Implemente um CRUD (**Create, Read, Update, Delete**) básico para a entidade **Cliente**.

Crie os métodos de **CRUD** no **ClienteController** e associe-os às rotas HTTP adequadas (**GET**, **POST**, **PUT**, **DELETE**).

Implemente os métodos no **ClienteService** e no **ClienteRepository**.

Exemplo de Rotas:

- **GET** /clientes/{id}: Buscar cliente por ID.
- **POST** /clientes: Criar novo cliente.
- **PUT** /clientes/{id}: Atualizar um cliente existente.
- **DELETE** /clientes/{id}: Remover um cliente.