**CURSO – JAVA: TRABALHANDO COM LAMBDAS, STREAMS E SPRING FRAMEWORK**

**MÓDULO 1 – UM NOVO PROJETO**

O Jackson/Databinding tem o objetivo de serializar e desserializar um dado.

A anotation **@JsonAlias**("Title") permite pegar uma “chave” de um objeto recebido na requisição (por exemplo) e transformar esse nome title em outro parâmetro qualquer como título. Ou seja, ele irá ler “Title” e escrever “título”.

A annotation **@JsonProperty**(“Algo”) serve para serializar e desserializar um dado, ou seja, ele irá ler “algo” e escrever “algo”.

**@JsonAlias**

O @JsonAlias é usado para definir um ou mais apelidos para o nome da propriedade JSON associada ao campo Java.

Ao desserializar, o @JsonAlias permite que a biblioteca encontre o valor JSON correspondente, mesmo que o nome da propriedade no JSON não corresponda exatamente ao nome do campo Java.

Por exemplo, se você tiver uma classe Java com a propriedade "nomeCompleto" e o JSON usa "nome" em vez disso, você pode usar o @JsonAlias("nome") para mapear corretamente a propriedade. Dessa forma, tanto "nomeCompleto" quanto "nome" serão aceitos ao fazer o mapeamento:

public class Pessoa {

@JsonAlias({"nomeCompleto", "nome"})

private String nomeCompleto;

}

**@JsonProperty**

Essa anotação é usada para definir o nome da propriedade JSON que está associada ao campo Java.

Quando o JSON é serializado, isto é, convertido de objetos Java para JSON, o nome especificado em @JsonProperty será usado como a chave para o campo no JSON de saída. Da mesma forma, quando o JSON é desserializado (convertido de JSON para objetos Java), a biblioteca procura pelo nome especificado em @JsonProperty para mapear o valor JSON para o campo Java.

Então, se você quiser usar um nome diferente no JSON, pode usar o @JsonProperty(“nomeNoJson”). Isso é útil, por exemplo, quando você precisa trabalhar com um JSON que segue uma convenção de nomenclatura diferente da sua classe Java.

Por exemplo, se você tiver uma classe Java com a propriedade "nomeCompleto" e quiser que ela seja mapeada para o JSON como "nome", você pode usar o @JsonProperty("nome") para especificar o nome correto no JSON:

public class Pessoa {

@JsonProperty("nome")

private String nomeCompleto;

}

Para converter um dado em outro com o Jackson, nós vamos precisar da classe ObjectMapper.

Criamos a interface como ainda não sabemos o retorno do método, vamos usar o genérics <T>.

public interface **IConverteDados** {

<T> T **obterDados**(String json, Class<T> classe);

}

Implementando a interface em uma classe:

public class **ConverteDados** implements **IConverteDados** {

private **ObjectMapper** mapper = new **ObjectMapper**();

@Override

**Na chamada desse método, o 2º parâmetro deve ser a classe que eu quero que o mapper transforme, dessa forma temos um método genérico!**

public <T> T **obterDados**(String json, Class<T> classe) {

try {

return mapper.readValue(json, classe);

} catch (JsonProcessingException e) {

throw new RuntimeException(e);

}

}

}

Aplicando:

ConverteDados conversor = new **ConverteDados**();

**DadosSerie** dados = conversor.**obterDados**(json, **DadosSerie.class**);

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

**CURSO – JAVA: PERSISTÊNCIA DE DADOS E CONSULTAS COM SPRING DATA JPA**

**MÓDULO 3 – MAPEANDO RELACIONAMENTOS**

**Fetch type:** Define qual a estratégia será utilizada para carregar os dados do banco para sua aplicação.

FetchType.**Eager** 🡪 Traz os relacionamentos mesmo se não for solicitado, ou seja, pega todos os dados relacionados ao mesmo tempo.

FetchType.**Lazy** 🡪 Pega apenas o que precisa, quando precisa, ou seja, só busca os outros itens quando realmente vai utilizá-los.

A estratégia de carregamento afeta diretamente o desempenho da aplicação. Um carregamento Eager pode parecer eficiente, pois tudo já está carregado de uma vez. No entanto, se a relação envolver muitos dados, isso pode causar problemas de desempenho, além de consumir muito mais memória, uma vez que estamos carregando mais dados do que realmente precisamos.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

**MÓDULO 4 – BUSCANDO INFORMAÇÕES DO BANCO**

Vamos utilizar o **Derived Queries** para personalizar nossas buscas no banco de dados, onde a partir do nome do método criado no repositório, a JPA conseguirá inferir isso e transformar em consultas no banco.

O nome do método deve ser de acordo com a **propriedade da classe** e não de acordo com o tipo desse dado.

**Ordenação de dados:**

Para trabalhar com a ordenação de registros, podemos utilizar o **OrderBy** para ordenar por algum atributo, como a série pela avaliação. Também podemos encadear atributos. Se uma Série tem um Ator e queremos ordenar pelo nome do ator, podemos utilizar OrderByAtorNome, por exemplo.

Além do OrderBy, ainda existem alguns outros recursos de filtros que podem ser utilizados:

**Distinct**, para remover dados duplicados

**First**, para pegar o primeiro registro

**Top**, para limitar o número de dados

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

**MÓDULO 5 – CONHECENDO JPQL**

As vezes temos métodos com nomes enormes usando a derived queries (ex: findByTotalTemporadasLessThanEqualAndAvaliacaoGreaterThanEqual (totalTemporadas, avaliacao), então é melhor personalizar a query e colocar um nome mais adequado.

select \* from series WHERE series.total\_temporadas <= 5 AND series.avaliacao >= 8.5;

**Serie 🡪 Nome da entidade**

@Query("SELECT s FROM **Serie** s WHERE **s**.totalTemporadas <= :totalTemporadas AND **s**.avaliacao >= : avaliacao ")

List<Serie> seriesPorTemporadaEAvaliacao(int totalTemporadas, double avaliacao);

O :NomeParâmetro significa que vamos utilizar o parâmetro do método.

Para usar query nativa do SQL, invés de JPQL, devemos devemos usar o atributo nativeQuery = true.

A principal diferença entre JPQL e SQL Nativo é, portanto, o nível de abstração. A JPQL abstrai os detalhes do banco de dados, permitindo que você trabalhe no nível do modelo de domínio (classe). **Se você precisar mudar o banco de dados para outro, a maioria das suas consultas JPQL continuarão funcionando sem alterações**.