BandTec
DIGITAL SCHOOL

Curso de Tecnologia em Análise e Desenvolvimento de Sistemas

Técnicas de Programação Web

Spring Data JPA - Relacionamentos e Query - Guia de Referência

Sumário

Relacionamentos entre Entidades

É muito difícil imaginar um sistema de informação, por menor que seja, que use apenas 1 (uma) tabela de um banco de dados relacional. Aliás, os bancos de dados relacionais possuem esse nome pela capacidade explícita de relacionar várias tabelas para garantir, entre outras coisas, a normalização e a consistência de dados. O relacionamento entre tabelas também pode ser mapeado para objetos com o JPA e veremos como implementar os principais tipos de relacionamentos neste tópico.

Anteriormente, usamos como exemplo uma classe chamada **Estabelecimento**. Vamos supor que agora precisamos agrupar os estabelecimentos do sistema por tipos para facilitar as pesquisas dos usuários e a geração de relatórios. Assim, entra a tabela **tipo_estabelecimento**, que está ligada à **estabelecimento** por meio de uma **FK** (**Foreign Key** ou **Chave Estrangeira**). O relacionamento entre essas tabelas está representado na Figura 1.

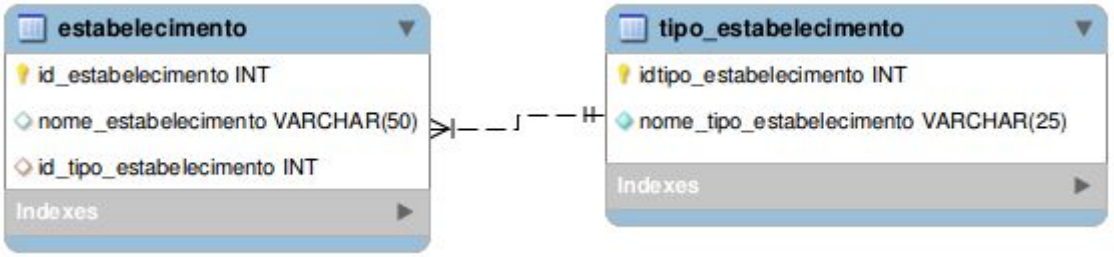

Figura 1: Diagrama entidade-relacionamento entre “estabelecimento” e “tipo_estabelecimento”

Ou seja, o mapeamento configura que um estabelecimento é de um **tipo_ estabalecimento** e que um **tipo_estabelecimento** pode ser de vários **estabelecimento**. Vejamos a seguir como mapear esse relacionamento em entidades JPA.

Muitos para Um (ManyToOne)

Primeiro, vamos criar a classe de ORM para a tabela **tipo_estabelecimento**. Um nome apropriado para a classe seria **TipoEstabelecimento**, conforme já estudamos anteriormente. Um exemplo de como seu código ficaria está no no código fonte seguir.

```
@Entity
@Table(name = "tbl_tipo_estabelecimento")
public class TipoEstabelecimento {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "id_tipo_estabelecimento")
	private Integer id;

	@Column(name = "nome_tipo_estabelecimento", length=25, nullable=false)
	private String nome;

	// construtores, getters e setters

}
```

Vejamos agora a alteração que precisa ser feita da classe **Estabelecimento**, para que o JPA conheça o relacionamento entre ela e **TipoEstabelecimento** no código fonte a seguir.

```
// imports que já existiam

import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;

public class Estabelecimento {

    // atributos que já existiam

   @JoinColumn(name = "id_tipo_estabelecimento")
   @ManyToOne
   private TipoEstabelecimento tipo;

   // construtores, getters e setters
}
```

A configuração do relacionamento foi feita por meio das anotações **@ManyToOne** e **@JoinColumn** sobre o atributo **tipo**, do tipo **TipoEstabelecimento.**

Com a anotação **@ManyToOne**, indicamos que podem haver **muitos** objetos do tipo **Estabelecimento** associados a **um** mesmo **TipoEstabelecimento**. Afinal, do estabelecimento de exemplo *"Hotel Familiar da Luz"* e *"Mega Hotel da Sombra"* seriam ambos do tipo de estabelecimento *"Hotel"*.

Com a anotação **@JoinColumn**, indicamos qual o campo na tabela **estabelecimento** deve estar na chave estrangeira para a tabela **tipo_estabelecimento**. O nome do campo está no atributo **name** dessa anotação.

Caso alguma ou nenhuma das tabelas exista ou a chave estrangeira não existir, o Hibernate enviar as instruções SQL necessárias para a criação das tabelas e/ou da

chave estrangeira, caso a propriedade **spring.jpa.hibernate.ddl-auto** do projeto esteja com o valor **update**.

Indicando o TipoEstabelecimento de um Estabelecimento

A entidade **Estabelecimento**, de forma muito mais natural, possui um **TipoEstabelecimento**. Logo, se, ao usarmos o método **setTipo()** e usarmos como argumento um objeto do tipo **TipoEstabelecimento** recuperado do banco de dados pelo JPA, o Hibernate cria e envia a instrução necessária para que o registro da tabela **estabelecimento** possua o valor de **id_tipo_estabelecimento** correto. Seriam enviados um **update** ou um **insert**, caso o registro de estabelecimento já exista ou se estamos criando um novo, respectivamente.

Consultando o TipoEstabelecimento de um Estabelecimento

A entidade **Estabelecimento**, de forma muito mais natural, possui um **TipoEstabelecimento**. Logo, ao invocar o método **getTipo()** de um objeto do tipo **Estabelecimento**, obteremos, de forma transparente, o registro da tabela **tipo_estabelecimento** que possui a chave estrangeira para o campo **id_tipo_estabelecimento** na tabela **estabelecimento**. O Hibernate é quem cria e envia para o banco as instruções ***select*** necessárias para que isso fique fácil no código Java.

Um para Muitos (OneToMany)

Assim como podemos dizer que um **Estabelecimento** é de um **TipoEstabelecimento**, podemos dizer que um **TipoEstebalecimento** pode ser usado por vários **Estabelecimento**, certo? Como isso é uma associação comum e também é frequente a necessidade da recuperação dos N itens associados a um determinado registro via chave estrangeira, o JPA oferece uma configuração simples para recuperar esse tipo de relacionamento. Basta usar a anotação **@OneToMany**. Um exemplo de como seu código ficaria está no no código fonte a seguir, onde acrescentamos mais um atributo à classe **TipoEstabelecimento**.

```
import javax.persistence.OneToMany;
import java.util.List;

// anotações já existentes
public class TipoEstabelecimento {

	// atributos já existentes

	@OneToMany(mappedBy = "tipo")
	private List<Estabelecimento> estabelecimentos;
	// construtores, getters e setters
}
```

Na anotação **@OneToMany** usamos o atributo **mappedBy**, que indica por qual nome o **TipoEstabelecimento** é conhecido em **Estabalecimento**. No código fonte anterior podemos ver que o atributo foi chamado de **tipo**, por isso usamos essa palavra no **mappedBy**.

Note que o atributo estabelecimentos é uma **Collection**. Isso serve para representar de forma orientada a objetos a cardinalidade *"um mesmo tipo de estabelecimento pode ser de vários estabelecimentos"*. Afinal, podem haver vários "hotel", "escola", "academia"... E com esse atributo devidamente anotado com **@OneToMany**, sempre que recuperarmos um **TipoEstabelecimento** usando um **EntityManager** e invocarmos set **getEstabelecimentos()**, o Hibernate enviará automaticamente para o banco um novo ***select*** solicitando todos os registros da tabela **estabelecimento** relacionados com o registro de **tipo_estabelecimento** em questão e preencherá a coleção com instâncias de Estabelecimento preenchidas conforme os registros encontrados ou com um a **coleção vazia**, caso não exista nenhum registro relacionado.

Chave Primária Composta

Algumas tabelas possuem suas chaves primárias (*primary keys*) compostas, ou seja, não é apenas 1 (um) campo que compõe a chave primária. Uma chave composta pode ter dois ou mais campos, conforme necessário.

Quando surge essa necessidade, é preciso criar uma classe de ORM exclusivamente para representar a composição de uma chave primária composta. Feito isso, é necessário indicar na entidade mapeada para a tabela que está usando uma classe de chave composta.

Tomemos como exemplo a seguinte situação: Temos a tabela **avaliacao**, cuja chave primária é composta pelos campos **id_usuario** e **id_estabelecimento** os quais também são chaves estrangeiras para as tabelas **usuario** e **estabelecimento**, respectivamente. Detalhes sobre essa tabela na Figura 2.

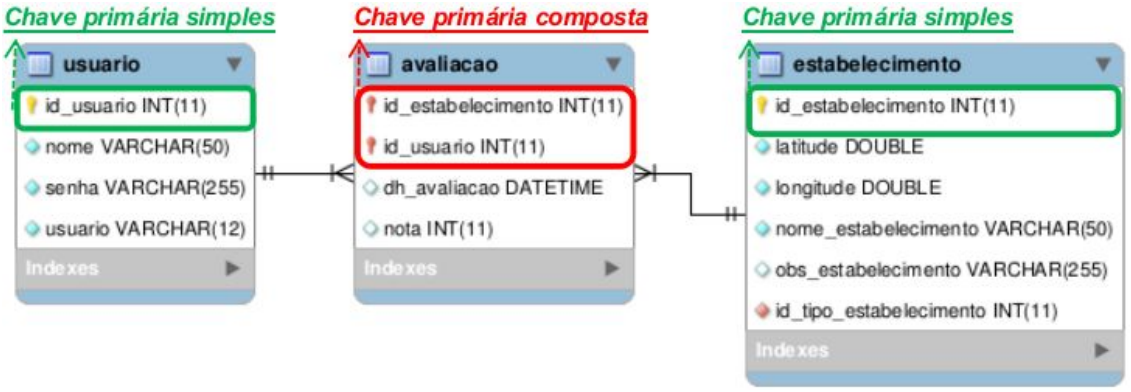

Figura 2: Tabela “avaliacao”, sua chave primária e seus relacionam

Para fazer o mapeamento objeto relacional desse exemplo, vamos primeiro criar a classe que irá mapear a chave primária da tabela avaliação. Para isso, vamos criar a classe **AvaliacaoId** no pacote de entidades. Vide o código fonte a seguir.

```
import javax.persistence.Embeddable;

@Embeddable
public class AvaliacaoId implements Serializable {

	@Column(name = "id_usuario")
	private Integer usuarioId;

	@Column(name = "id_estabelecimento")
	private Integer estabelecimentoId;

	// getters e setters
	// equals() e hashCode()
}
```

O que torna essa classe numa entidade de mapeamento de chave primária composta é a anotação **@Embeddable** sobre a assinatura da classe. Seu caminho completo está no **import** do início do código.

Note que ela implementa a interface **Serializable**. Isso é um **requisito obrigatório** para classes anotadas com **@Embeddable**.

Um detalhe muito importante é que toda classe de mapeamento de chave primária composta **deve, obrigatoriamente, sobrescrever** os métodos **equals()** e **hashCode()**. O Intellij possui um assistente que faz isso com Alt+Insert e escolher as opções **hashCode() and equals()**.

O código gerado por esse assistente será como o do código fonte a seguir.

```
@Override
public int hashCode() {
	final int prime = 31;
	int result = 1;
	result = prime * result + ((estabelecimentoId == null) ? 0 :
estabelecimentoId.hashCode());
	result = prime * result + ((usuarioId == null) ? 0 : usuarioId.hashCode());
	return result;
}

@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	AvaliacaoId other = (AvaliacaoId) obj;
	if (estabelecimentoId == null) {
		if (other.estabelecimentoId != null)
			return false;
	} else if (!estabelecimentoId.equals(other.estabelecimentoId))
		return false;
	if (usuarioId == null) {
		if (other.usuarioId != null)
			return false;
	} else if (!usuarioId.equals(other.usuarioId))
		return false;
	return true;
}
```

Os métodos **equals()** e **hashCode()** são usados pelo Java para saber se duas instâncias de objetos são o “mesmo” objeto. Se não sobrescritos, ele considera se os objetos comparados são o mesmo objeto em memória. Se sobrescritos, como foi feito

aqui, ele usa o critério programado neles para saber se dois objetos são “iguais”. Se não forem sobrescritos em todas as classes que mapeiam chaves primárias compostas, o JPA lança uma exceção logo no início da execução do programa.

Após mapear a chave primária composta, devemos indicar na entidade que mapeia a tabela **avaliacao** que ela está usando esse mapeamento. Faremos isso na classe **Avaliacao**, cujo código está no código fonte a seguir.

```
import javax.persistence.EmbeddedId;

@Entity
@Table(name = "tlb_avaliacao")
public class Avaliacao {

	@EmbeddedId
	private AvaliacaoId id;

	@JoinColumn(name = "id_usuario", insertable = false, updatable = false)
	@ManyToOne(optional = false)
	private Usuario usuario;

	@JoinColumn(name = "id_estabelecimento", insertable = false, updatable = false)
	@ManyToOne(optional = false)
	private Estabelecimento estabelecimento;

	private Integer nota;

	// @CreationTimestamp
	@Temporal(TemporalType.TIMESTAMP)
	@Column(name = "dh_avaliacao")
	private Calendar dataAvaliacao;

    // getters e setters
}
```

O que faz com que essa classe use como chave primária a **AvaliacaoId** é a anotação **@EmbeddedId** sobre o atributo **id**. Seu caminho completo está no **import** do início do código.

Um detalhe muito importante é os relacionamentos com **Usuario** e **Estabelecimento** precisam ser configurados com **insertable=false** e **updatable=false**. Isso é imposição do JPA, para que não haja ambiguidade sobre onde configurar os valores do **id** de **Usuario** e **Estabelecimento**, pois só podem ser informados no atributo **id**, que é uma instância de **AvaliacaoId**. Se esses dois atributos não forem configurados, o JPA lançará uma exceção assim que o programa iniciar.

Para recuperar os valores do **Usuario** e do **Estabelecimento** associados a uma **Avaliacao**, basta usar os métodos **getUsuario()** e **getEstabelecimento()**, respectivamente.

A entidade **Avaliacao** está no código fonte a seguir.

```
@Entity
@Table(name = "tbl_usuario")
public class Usuario {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "id_usuario")
	private Integer id;

	@Column(length=50, nullable=false)
	private String nome;

	@Column(length=12, nullable=false)
	private String usuario;

	private String senha;

	// construtores, getters e setters
}
```

Como usar entidades com chave composta

Já vimos como mapear uma chave composta com JPA. Veremos agora como os atributos de **Avaliacao** e **AvaliacaoId** deveriam ser preenchidos para que a chave composta e os relacionamentos funcionem corretamente.

Quando se deseja **criar** um registro de **Avaliacao**, deve-se primeiro criar uma instância de **AvaliacaoId** e preencher seus atributos com ids de instâncias de **Usuario** e **Estabelecimento** recém criados ou recuperados do banco de dados. O próximo passo é usar a instância de **AvaliacaoId** recém criada como o atributo **id** de uma instância de **Avaliacao,** conforme o código fonte a seguir.

```
Usuario usuario = // novo ou recuperado
Estabelecimento estabelecimento = // novo ou recuperado

AvaliacaoId idNovo = new AvaliacaoId();
idNovo.setUsuarioId(usuario.getId());
idNovo.setEstabelecimentoId(estabelecimento.getId());

Avaliacao novaAvaliacao = new Avaliacao();
novaAvaliacao.setId(idNovo);
```

Assim, o objeto **novaAvaliacao** está pronto para ser persistido no banco de dados pelo JPA. Note que não é preciso atribuir os valores dos atributos **usuario** e **estabelecimento** para a criação de um novo registro de **Avaliacao**.

Herança

Um dos recursos mais interessantes nas linguagens orientadas a objetos, como Java, é a possibilidade de herdar características entre classes. Quando queremos que uma classe herde de outra, criamos uma **subclasse** (nomenclatura usada pelo Java). A classe da qual herda é sua **superclasse** (nomenclatura usada pelo Java).

Podemos usar esse recurso também no ORM. Isso é usado quando temos parecidas e/ou quando tabelas foram "divididas" em várias por questões de normalização, mas esse divisões só tornam a programação mais trabalhosa em linguagens orientadas a objetos.

Um exemplo clássico disso é quando é o cenário "um sistema precisa cadastrar clientes. Um cliente pode ser pessoa física ou jurídica. Se pessoa física, cadastrar seu estado civil e escolaridade. Se pessoa jurídica, cadastrar sua inscrição estadual e nome fantasia". Nesse cenário, os profissionais de banco de dados costumam criar 3 tabelas com nomes como estes: **cliente**, **cliente_pf**, **cliente_pj**, conforme o diagrama da Figura 3.

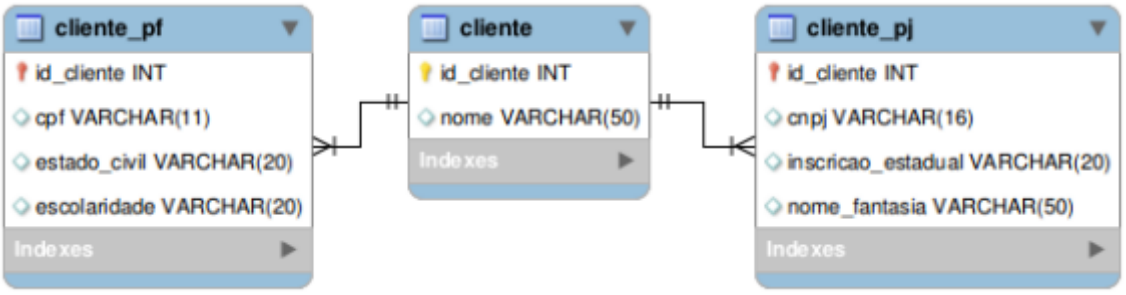

Figura 3: Conceito de cliente "dividido" em 3 tabelas para fins de normalização

Para fazer o mapeamento objeto relacional desse exemplo, podemos usar 3 (três) estratégias que o JPA nos oferece. Uma que poderia usar as tabelas exatamente como estão na Figura 3 e outras que levariam a diferentes propostas de modelagem.

Independente da estratégia adotada, precisaremos de 4 classes: **Cliente**, **ClientePf** e **ClientePf**, descritas nos códigos fonte a seguir.

```
import javax.persistence.Inheritance;

@Entity
@Table(name="tbl_cliente")
@Inheritance
public class Cliente {


        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "id_cliente")
        private Integer id;
```

```
        @Column(length = 50)
        private String nome;

        // construtores, getters e setters
 }
```

```
@Entity
@Table(name="tbl_cliente_pf")
public class ClientePf extends Cliente {

        @Column(name="estado_civil", length = 20)
        private String estadoCivil;

        @Column(length = 20)
        private String escolaridade;

    // contrutores, getters e setters
}
```

```
@Entity
@Table(name="tbl_cliente_pj")
public class ClientePj extends Cliente {

        @Column(name="inscricao_estadual", length = 20)
        private String inscricaoEstadual;

        @Column(length = 50)
        private String nomeFantasia;

        // contrutores, getters e setters

}
```

Quanto a entidade **Cliente**, note que ela está anotada com **@Inheritance**. Essa anotação indica que essa classe pode ser estendida por outra(s). Ela possui um atributo **strategy**, que é o que define a estratégia de herança usada. Ele será explicado e exemplificado na sequência deste tópico.

As entidades **ClientePj** e **ClientePj** estendem de **Cliente**, sendo, portanto, subclasses de **Cliente**.

A seguir, vejamos as diferentes formas de programar um relacionamento usando herança com JPA.

Herança com a estratégia JOINED

Para o mapeamento espere que as tabelas sejam como as da figura 4.1, usamos a herança de estratégia **JOINED**. Para ver como indicar essa estratégia na anotação **@Inheritance** veja o código fonte a seguir.

```
import javax.persistence.InheritanceType;

@Entity
@Table(name="tbl_cliente")
@Inheritance(strategy=InheritanceType.JOINED)
public class Cliente {

	// atributos e métodos já existentes
 }
```

Ou seja, a estratégia **JOINED** define que cada classe fica mapeada para uma tabela diferente, porém, **os atributos da tabela da super classe não são repetidos nas tabelas das subclasses**. Nas tabelas das sub classes, a **chave primária** acaba sendo também uma **chave estrangeira** para a tabela da super classe.

Herança com a estratégia TABLE_PER_CLASS

A estratégia **TABLE_PER_CLASS** (exemplificada no código fonte a seguir) funciona de forma muito parecida com a **JOINED**. A diferença é que todos os atributos nas diferentes tabelas se repetem, não importa que representem "a mesma" informação.

```
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Cliente {

	// atributos e métodos já existentes
 }
```

Com a **TABLE_PER_CLASS**, as tabelas ficariam como na Figura 4, onde é possível notar que **não é criado nenhum relacionamento entre as tabelas**.

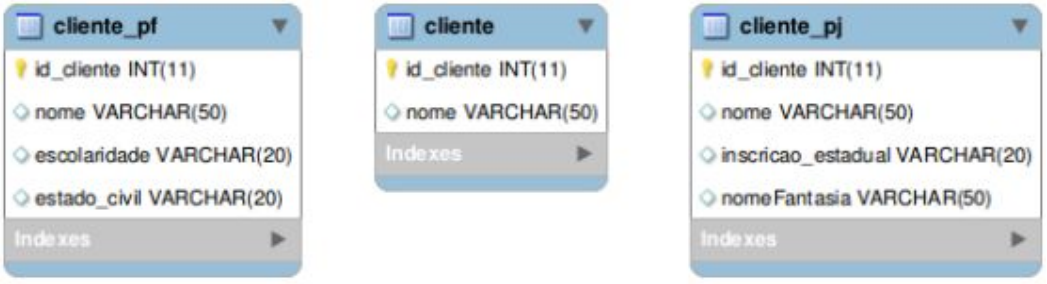

Figura 4: Tabelas "cliente", "cliente_pf" e "cliente_pj" compatíveis com herança "TABLE_PER_CLASS"

Herança com a estratégia SINGLE_TABLE

A estratégia **SINGLE_TABLE**, como o nome sugere, indica que apenas **1 (uma)** tabela ficará mapeada para a superclasse e suas sub classes, cujo nome será o indicado na **@Table** da super classe. Veja no código fonte a seguir como indicar essa estratégia. Essa é a estratégia padrão do JPA caso o atributo **strategy** seja omitido na anotação **@Inheritance**.

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Cliente {

	// atributos e métodos já existentes
}
```

Com a **SINGLE_TABLE**, as tabelas ficariam como na Figura 5, onde é possível notar que **apenas uma tabela inclui todos os campos de todas as classes envolvidas no mapeamento com herança**. A anotação **@Table** nas subclasses é ignorada nessa estratégia.

Figura 5: Tabela "cliente" compatível com herança "SINGLE_TABLE"

Note que há um campo chamado **DTYPE** na tabela cliente ele serve para indicar o "tipo" de cliente, de acordo com a sub classe usada para a criação do registro. Como temos **ClientePf** e **ClientePj**, os valores desse campo nos registros poderiam ser, literalmente, "ClientePf" e "ClientePj". A Figura 6 contém o resultado da consulta aos registros de uma tabela **cliente** criada para ser compatível com a estratégia **SINGLE_TABLE**. Observe os valores do campo DTYPE na primeira coluna.

#	DTYPE	id_cliente	nome	inscricao_estadual	nomeFantasia	escolaridade	estado_civil
1	ClientePj	1	NULL	6876787866-3	Loja Bá	NULL	NULL
2	ClientePf	2	NULL	NULL	NULL	Superior Completo	Solteiro

Figura 6: Registros de uma tabela "cliente" compatível com herança "SINGLE_TABLE"

Caso você queira que o campo que represente o tipo tenha outro nome, como **tipo_pessoa**, por exemplo, basta usar a anotação **@DiscriminatorColumn** sobre a entidade da super classe (no caso, a Cliente). Exemplo no código fonte a seguir.

```
import javax.persistence.DiscriminatorColumn;

@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipo_pessoa")
public class Cliente {

	// atributos e métodos já existentes
 }
```

Custom Queries

Na medida em que vamos acrescentando mais entidades e mais relacionamentos, as possibilidades de operações vão se multiplicando em nosso projeto com Spring Data e JPA/Hibernate. Assim, é comum surgir a necessidade de instruções junto ao SGBD que o uso de métodos padronizados numa **Repository** pode não ser o suficiente. Para esses casos, podemos criar **Custom Queries**, ou seja, instruções personalizadas.

É possível criar **Custom Queries** que adotam o **JPQL** ou SQL Nativo. A seguir, vejamos o que vem a ser **JPQL**.

Introdução à JPQL

A **JPQL** (***J**ava **P**ersistence **Q**uery **L**anguage*) é uma linguagem de instruções **orientada a objetos** para entidades de mapeamento objeto-relacional **independente de fornecedor de banco de dados**. Com ela escrevemos instruções de **consulta** (select), **exclusão** (delete) e **alteração** (update). Não há suporte para "insert" nela.

Ser **orientada a objetos** significa que devemos usar **os nomes das classes e atributos das Entidades** e não das tabelas nas instruções JPQL.

Ser **independente de banco de dados**, significa que o JPA traduz instruções JPQL para o SQL específico de cada fabricante de banco de dados sem que o desenvolvedor se preocupe com isso. Isso é um grande elemento facilitador, pois há muitas diferenças entre os SQLs, como conversão de tipos e paginação de resultados. A Figura 7 ilustra esse processo para alguns servidores de banco.

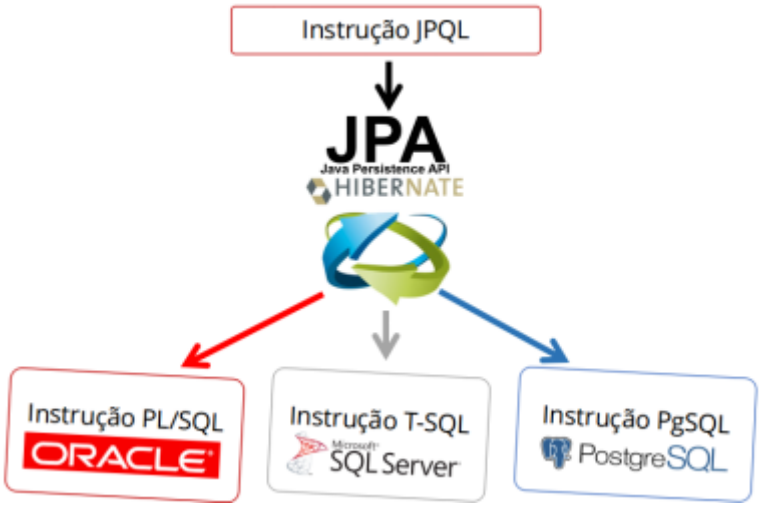

Figura 7: Tradução de JPQL para SQL de diferentes fabricantes de banco de dados

Outra grande vantagem dessa independência de banco de dados, é que o projeto pode acessar, por exemplo, o ***PostgreSQL*** durante o desenvolvimento e o ***Oracle*** em produção, ambos sendo acessados pelas **mesmas instruções JPQL**.

Exemplos de instruções JPQL

A seguir, veremos como realizar operações com JPQL considerando 2 entidades que já estudamos: **Estabelecimento** e **TipoEstabelecimento**.

Para conhecer melhor o mapeamento objeto-relacional, observe o Quadro 1 para **Estabelecimento** e o Quadro 2 para **TipoEstabelecimento**.

	Tabela	**Entidade**
Nome	estabelecimento	Estabelecimento
Campos x Atributos	id_estabelecimento	id
	nome_estabelecimento	nome
	cnpj	cnpj
	dh_criacao	dataCriacao
	id_tipo_estabelecimento (chave estrangeira)	tipo (relacionamento)

Quadro 1: ORM da entidade “Estabelecimento”

	Tabela	**Entidade**
Nome	tipo_estabelecimento	TipoEstabelecimento
Campos x Atributos	id_tipo_estabelecimento	id
	nome_tipo_estabelecimento	nome

Quadro 2: ORM da entidade “TipoEstabelecimento”

O clássico *“select * from”* do SQL fica assim com JPQL, supondo que queremos todos os registros de **TipoEstabelecimento**:

```
from TipoEstabelecimento
```

Ou, se quisermos deixar a instrução mais explícita:

```
select t from TipoEstabelecimento t
```

Nessa última instrução, usamos **t** letra e para ser o alias da entidade **TipoEstabelecimento**, mas poderia ser qualquer letra ou palavra com letra minúscula e, preferencialmente, no padrão **camelCase**.

Note ainda que as instruções confirmam o que já afirmamos: Não usamos o nome da tabela e sim da classe da entidade.

Outro exemplo, envolvendo uma cláusula "*where*":

```
from Estabelecimento where nome = 'oi lolla'
```

Ou, se quisermos deixar a instrução mais explícita:

```
select e from Estabelecimento e where e.nome = 'oi lolla'
```

Em ambas, solicitamos uma consulta por estabelecimentos com o nome igual a 'oi lolla'. Podem ser usadas para retornar uma instancia ou uma coleção de objetos do tipo **Estabelecimento**.

Consultas com relacionamentos

Por ser orientado a objetos, o JPQL facilita imensamente consultas entre entidades relacionadas. Vejamos alguns exemplos:

Recuperar estabelecimentos usando critérios de TipoEstabelecimento

```
select e from Estabelecimento e where e.tipo.nome = 'bar'
```

Nessa instrução, queremos todos os registros de **Estabelecimento**, cujo atributo **nome** de seu **tipo** (que é um **TipoEstabelecimento**) é *'bar'*. Essa consulta pode ser usada para retornar uma instancia ou uma coleção de objetos do tipo **Estabelecimento**.

Recuperar tipos de estabelecimentos usando critérios de Estabelecimento

```
select e.tipo from Estabelecimento e where e.cnpj is null
```

Nessa instrução, queremos todos os registros de **TipoEstabelecimento**, eassociados a registros de **Estabelecimento** cujo atributo **cnpj** não é nulo. Essa consulta pode ser usada para retornar uma instancia ou uma coleção de objetos do tipo **TipoEstabelecimento**, pois, logo após a instrução select temos **e.tipo**.

Em instruções como essas, o JPA irá criar a instrução SQL apropriada para o fornecedor de banco de dados com o uso dos JOINS possíveis.

Todos esse benefícios do uso do relacionamento oferecido pelo mapeamento objeto-relacional podem ser usufruídos em instruções de **update** e **delete**.

As cláusulas e palavras chave da JPQL são **case insensitive**.

Operadores JPQL

A imensa maioria dos operadores do SQL padrão é idêntico no JPQL. A diferença está na comparação de "*diferente*". No SQL usamos **<>**, mas no JPQL é **!=**.

Quanto às funções, existe um conjunto de funções no JPQL, explicadas no tópico a seguir.

Funções JPQL

As funções da JPQL são:

upper(String s): Converte o argumento para CAIXA ALTA.

lower(String s): Converte o argumento para caixa baixa.

current_date(): Recupera a data atual do SGBD.

current_time(): Recupera a hora atual do SGBD.

current_timestamp(): Recupera o *timestamp* atual do SGBD.

substring(String s, int inicio, int tamanho): Recupera os "tamanho" caracteres a partir da posição "inicio" do argumento "s".

trim(String s): Faz o *trim* no argumento.

length(String s): Recupera a quantidade de caracteres do argumento.

locate(String s, String termo, int inicio): Recupera a posição onde determinado "termo" estão no argumento "s" a partir de uma posição "inicio".

abs(Numeric n): Retorna o valor absoluto do argumento.

sqrt(Numeric n): Retorna a raiz quadrada do argumento.

mod(Numeric dividendo, Numeric divisor): Retorna o resto da divisão de "dividendo" pelo "divisor".

treat(x as Tipo): Tenta fazer um *cast* do argumento "x" para o "Tipo" indicado.

size(c): Retorna o tamanho da coleção (*Collection*, *List*, *Set* etc) no argumento.

Recomendação para aprofundamento em JPQL

O estudo de JPQL é muito grande, cabendo sozinho num livro de centenas de páginas. Algumas sugestões de materiais completos e atualizados para estudo on-line dessa tecnologia são:

JPA Queries - JPQL (JPA Query Language) and Criteria API
https://www.objectdb.com/java/jpa/query

Ultimate Guide to JPQL Queries with JPA and Hibernate
https://thoughts-on-java.org/jpql/

Custom Queries em Repository

O uso de Custom Queries em interfaces Repository é uma boa prática e simples de fazer. A seguir, veremos as diferentes formas de fazê-lo.

Consultas com @Query

É possível criar um método que retorna um objeto ou uma coleção de objetos num método de uma Repository anotando ele com **@Query** (pacote **org.springframework.data.jpa.repository**).

@Query usando JPQL

Caso você queira usar JPQL em instruções num **@Query,** basta fazer como no exemplo a seguir.

```
@Query("from Estabelecimento where nome != null")
List<Estabelecimento> listaDosComNome();
```

A instrução JPQL vai direto na **@Query**. Note que, quando usamos esse recurso, não somos obrigados a usar o padrão **findBy** no início do nome do método.

@Query usando SQL Nativo

Caso você queira usar SQL Nativo em instruções num **@Query,** basta fazer como no exemplo a seguir.

```
@Query(value="select * from tbl_estabelecimento where nome_estabelecimento is not null",
        nativeQuery = true)
List<Estabelecimento> listaDosSemNome();
```

Note que, bastou usarmos o atributo **nativeQuery=true** na anotação **@Query**. Assim, o instrução SQL Nativo deve ser colocada no atributo **value**.

Consultas parametrizadas com @Query

Caso você precise de consultas com parâmetros, pode-se recorrer a 2 técnicas: parâmetros **numerados** ou **nomeados**. Independente da técnica utilizada, o JPA faz as conversões e ajustes necessários para a criação da instrução SQL que será enviada ao

SGBD. Assim, não precisamos nos preocupar se o parâmetro é String, Double, Calendar, List etc.

Consultas com parâmetros numerados

São aquelas onde apenas numeramos os parâmetros, que são substituídos pelos argumentos do método seguindo sua ordem. A numeração começa do **número 1 (um)**. Vejamos o exemplo a seguir.

```
@Query("from Estabelecimento where nome like ?1 or cnpj like ?2")
List<Estabelecimento> pesquisarPorNomeCnpj(String nomeContendo, String cnpjContendo);
```

No último código, o valor do argumento **nomeContendo** irá substituir o argumento **?1** e do **cnpjContendo**, o **?2**.

Essa técnica também pode ser usada quando usamos SQL Nativo.

Consultas com parâmetros nomeados

São aquelas onde damos nome aos parâmetros, que são substituídos pelos argumentos do método de acordo com a anotação **@Param** (pacote **org.springframework.data.repository.query**). Vejamos o exemplo a seguir.

```
@Query("from Estabelecimento where nome like :valorNome or cnpj like :valorCnpj")
List<Estabelecimento> pesquisarPorNomeCnpj(
                            @Param("valorNome") String nomeContendo,
                            @Param("valorCnpj") String cnpjContendo);
```

No último código, o valor do argumento **nomeContendo** irá substituir o argumento **:valorNome** e do **cnpjContendo**, o **:valorCnpj**.

Essa técnica também pode ser usada quando usamos SQL Nativo.

Independente da técnica de parametrização utilizada, também podemos usar coleções (Collection, Lst, Set etc) como parâmetros. Vejamos os exemplos a seguir.

```
@Query("from Estabelecimento where cnpj in ?1")
List<Estabelecimento> pesquisarPorCnpjs(List<String> cnpjs);
```

Nesse código, o parâmetro **?1** será substituído pelo argumento **cnpjs**.

```
@Query("from Estabelecimento where cnpj in :cnpjs")
List<Estabelecimento> pesquisarPorCnpjs(@Param("cnpjs") List<String> cnpjs);
```

Nesse código, o parâmetro **:cnpjs** será substituído pelo argumento **cnpjs**.

Atualizações e exclusões com @Query e @Modifying

É possível criar um método que executa instruções DML de update ou delete num método de uma Repository anotando ele com **@Query** e **@Modifying** (ambas do pacote **org.springframework.data.jpa.repository**).

Importante! Se a **@Modifying** não for usada em método com a intenção de executar atualizações ou exclusões, o método irá lançar uma exceção em tempo de execução, assim que for invocado.

Observação! Caso o método anotado com **@Query** e **@Modifying** não for invocado a partir de uma **Service** (o que não é uma boa prática, mas pode acontecer), você também deve anotar o método com **@Transactional** (pacote **org.springframework.transaction.annotation**).

No mais, quase tudo que aprendemos sobre o uso da **@Query** para consultas vale para operações de atualizações e exclusões. A diferença é que os métodos que fazem isso só podem ter retornos do tipo **int** (quando queremos saber quantos registros foram afetados) ou **void** (quando apenas queremos executar a instrução). Vejamos nos exemplos a seguir.

```
@Modifying
@Query("delete from Estabelecimento where cnpj is null")
int excluirSemCnpj();
```

Nesse código, o retorno do método será um número inteiro com a quantidade de registros excluídos pelo "*delete*" executado.

```
@Modifying
@Query("update Estabelecimento set cnpj="-" where cnpj is null")
void preencherSemCnpj();
```

Nesse código, o método provocará o envio da instrução *"update"* para o SGBD, mas não será possível saber quantos registros foram afetados, uma vez que o retorno do método e **void**.

Inserts personalizados com @Query e SQL Nativo

A JPQL não tem a instrução "insert". Porém, é possível criar um registro de forma personalizada numa entidade usando SQL Nativo na **@Query** e os parâmetros necessários. Vejamos os exemplos a seguir.

```
@Query(value="insert into tbl_estabelecimento (nome_estabelecimento) values (?1)",
        nativeQuery=true)
void cadastroRapido(String nome);
```

Nesse código, o parâmetro **?1** será substituído pelo argumento **nome** e a invocação do código não vai retornar a quantidade de registros afetados.

```
@Query(value="insert into tbl_estabelecimento (nome_estabelecimento) values (?:nome)",
        nativeQuery=true)
int cadastroRapido(String nome);
```

Nesse código, o parâmetro **:nome** será substituído pelo argumento **nome** e a invocação do código vai retornar a quantidade de registros afetados.

Bibliografia

GUTIERREZ, Felipe. Pro Spring Boot. New York (EUA): Apress, 2016.

JBOSS.ORG. Hibernate ORM 5.2.12.Final User Guide. Disponível em: <https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html>. Acesso em: 24/03/2019.

JENDROCK, Eric. Persistence - The Java EE5 Tutorial. Disponível em: <https://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html>. Acesso em: 20/03/2019.

PANDA, Debu; RAHMAN, Reza; CUPRAK, Ryan; REMIJAN, Michael. EJB 3 in Action, Second Edition. Shelter Island, NY, EUA: Manning Publications, 2014.