# JPA

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande
Divisão de Computação

# Introdução

## Motivação

Diversas vezes, ao trabalhar com JDBC, nos deparamos com rotinas chatas e entediantes ao escrever comandos SQL.

## Exemplo

- Diversos WHILES para popular um objeto;
- Erro de sintaxe ao escrever um INSERT e;
- Até mesmo problemas de relacionamento de chaves ao fazer um delete.

Veja o código de JDBC a seguir....

# Introdução

```
private Usuario buscaUsuario(int id) throws SQLException {
    Connection conexaoComBanco = null;
    PreparedStatement preparedStatement = null;
    Usuario usuarioEncontrado = null;
    String consulta = "SELECT EMAIL, NOME  FROM  USUARIO WHERE ID = ?";
     try {
         conexaoComBanco = getConexaoComBanco();
         preparedStatement = conexaoComBanco.prepareStatement(consulta);
         preparedStatement.setInt(1, id);
         ResultSet rs = preparedStatement.executeQuery();
         usuarioEncontrado = new Usuario();
         while (rs.next()) {
             String email = rs.getString("EMAIL");
             String nome = rs.getString("NOME");
             usuarioEncontrado.setNome(nome);
             usuarioEncontrado.setId(id);
             usuarioEncontrado.setEmail(email);
         }
    } finally {
         if (preparedStatement != null) {
             preparedStatement.close();
         }
         if (conexaoComBanco != null) {
             conexaoComBanco.close();
```

# Introdução

Para evitar toda essa *burocracia* do JDBC, surgiu a JPA:

## Benefícios

- Pode aumentar consideravelmente o tempo de desenvolvimento da equipe
- Facilitar muito a implementação do código que manipula o banco de dados.

Quando corretamente aplicado, a JPA se torna uma ferramenta que ajuda em:

- Todas as funções de um CRUD;
- Além de contribuir com diversas otimizações de performance e consistência de dados.

# Introdução

É nesse exato momento que podemos entender por que as siglas *ORM* são tão faladas.

## Conceito Object Relational Mapping (ORM)

- Transformar as informações de um banco de dados, que estão no modelo relacional para classes Java, no paradigma Orientado a Objetos de um modo fácil.

# Introdução

Um framework ORM vai ajudá-lo na hora de:

- Realizar consultas;
- Manipular dados e;
- Até mesmo a retirar relatórios complexos

A **burocracia** de um SQL com JDBC não será necessária, nem mesmo os incansáveis loops que fazemos para popular objetos.

Tudo isso já está pronto. Só precisamos saber usar.

# Introdução

Outra vantagem que veremos é a portabilidade da aplicação entre banco de dados, tratamento de acesso simultâneo, acesso às funções, dentre outras mais.

```
private Usuario buscaUsuario(int id){
    EntityManager entityManager = getEntityManager();
    Usuario usuario = entityManager.find(Usuario.class, id);
    entityManager.close();
}
```

Note quão simples e prático ficou nosso código.

# Introdução

E para salvar uma informação no banco de dados?

Será que teremos que fazer muito código?

Na prática, não é necessário criar *queries* longas e tediosas.

**Bastaria fazer:**

```
private void salvarUsuario(Usuario usuario) {
    EntityManager entityManager = getEntityManager();
    entityManager.getTransaction().begin();
    entityManager.persist(usuario);
    entityManager.getTransaction().commit();
    entityManager.close();
}
```

# Introdução

É possível perceber que o contato com o banco de dados ficou mais simples e de fácil utilização.

Rotinas que incluíam escrever tediosos e verbosos comandos SQL agora viraram apenas chamadas a uma API simples.

É claro que a vantagem não fica só aí. Vai muito mais além

# Escolha uma Implementação

A JPA nada mais é do que um conjunto de regras, normas e interfaces para definir um comportamento também conhecido como especificação.

- Toda especificação precisa de uma implementação para que possa ser usada num projeto.

No caso da JPA, a implementação mais famosa e utilizada no mercado é o **Hibernate**, mas existem mais no mercado como OpenJPA, **EclipseLink**, Batoo e outras.

- Cada implementação tem suas vantagens e desvantagens na hora da utilização.

# Escolha uma Implementação

- O *Hibernate* contém diversas funcionalidades como Engenharia Reversa, Exportação de Schema facilitado e anotações próprias para simplificar a utilização da API.
- Já da OpenJPA é possível dizer que sua simplicidade e facilidade de usar os fazem bem próximos da API da JPA;
  - ao estudar a JPA facilmente poderia se entender a OpenJPA e suas anotações extras.

# Escolha uma Implementação

Para implementar a especificação, é necessário implementar as interfaces que estão no pacote *javax.persistence.** , que é justamente o papel do *Hibernate*, *EclipseLink* etc.

Além das interfaces implementadas da JPA, cada implementação pode ter as suas próprias anotações/classes para adicionar mais comportamentos, indo além do que prevê a especificação.

# Escolha uma Implementação

## A vantagem de ter um projeto usando apenas anotações da JPA:

É que ela se torna automaticamente portável, possibilitando a troca de *Hibernate* para o *EclipseLink*, por exemplo.

## A desvantagem em utilizar uma aplicação com anotações da JPA:

Perder as facilidades que a implementação pode fornecer.

- A anotação @ForeignKey, por exemplo (*Hibernate*).

# Escolha uma Implementação

```
public class Pessoa{
  @OneToOne
  @JoinColumn(name = "CREATE_BY")
  @ForeignKey(name="FK_USUARIO")// anotação do hibernate
  protected Usuario usuario;
}
```

É possível ver no código anterior que agora a chave estrangeira terá um nome determinado pela aplicação.

Para quem utilizar a JPA 2.1, a anotação @ForeignKey estará incorporada, fazendo com que a do *Hibernate* seja *deprecated*.

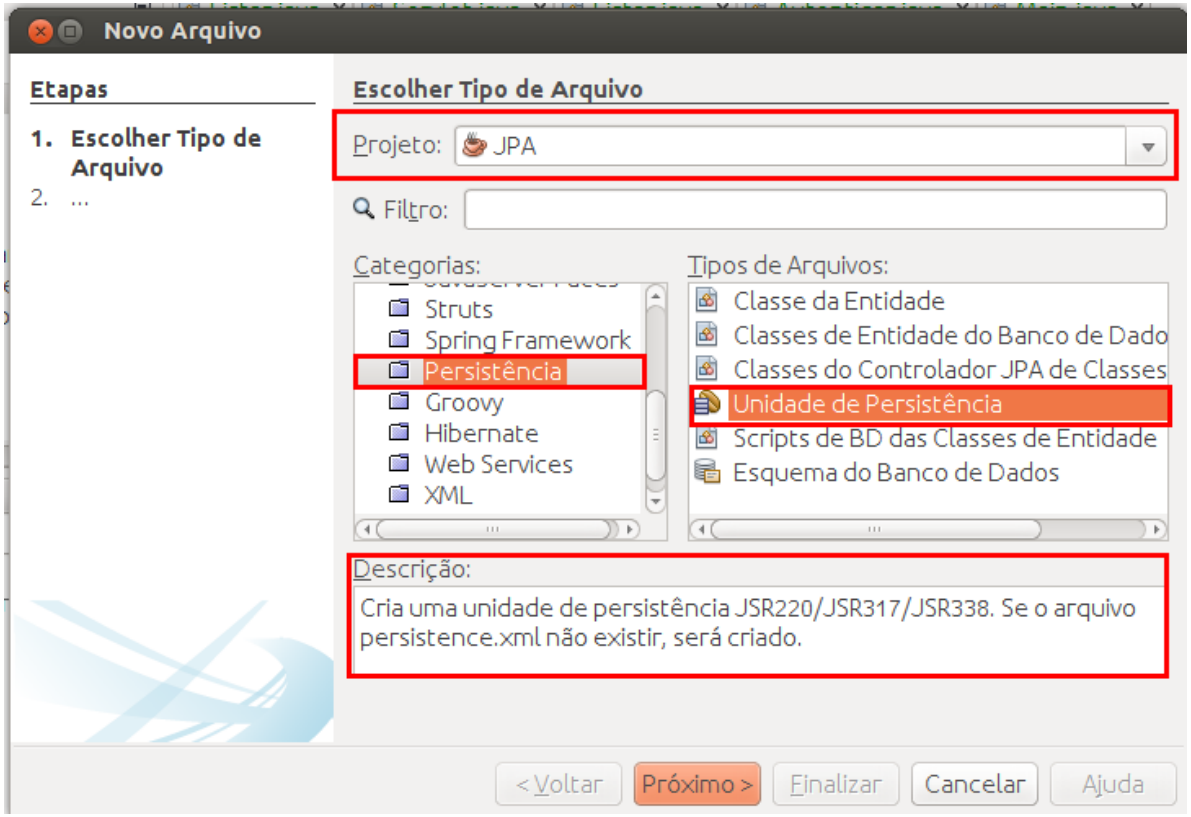
# Escolha uma Implementação

É preciso sempre ter em mente o seguinte: se a API nativa JPA não atende às suas necessidades a solução seria procurar alguma implementação que tenha as funções de que você necessita.

É uma via de duas mãos, pois o desenvolvedor terá acesso a uma implementação, mas, ao tentar mudá-la muito código terá que ser analisado e refeito.

# Como compor meu persistence.xml?

## Como criar o arquivo persistence.xml no NetBeans?



# Como compor meu persistencie.xml?

## Persistence.xml depois de criado no NetBeans

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w
  <persistence-unit name="MeuPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>negocio.Pessoa</class>
    <class>negocio.Dependente</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/prova"/>
      <property name="javax.persistence.jdbc.user" value="postgres"/>
      <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>
      <property name="javax.persistence.jdbc.password" value="postgres"/>
    </properties>
  </persistence-unit>
</persistence>
```

# EntityManager

```
public class Main {
    private static EntityManagerFactory entityManagerFactory =
            Persistence.createEntityManagerFactory("MeuPU");

    public static void main(String[] args) throws Exception {
        EntityManager entityManager =
                            entityManagerFactory.createEntityManager();
        entityManager.getTransaction().begin();
        // faz algo
        entityManager.getTransaction().commit();
        entityManager.close();
    }
}
```

# EntityManager

Note que a ação de criar um *EntityManagerFactory* é uma operação muito custosa e não vale a pena repeti-la diversas vezes.

Nesse caso o ideal é deixar o *EntityManagerFactory* como *static*, ou seja, uma instância para a aplicação inteira.

# EntityManager

**Você pode até se perguntar:**

## Como ele se comporta em um ambiente muti-thread?

O *EntityManagerFactory* é *thread-safe*, em outras palavras, ele não mandará um *EntityManager* para uma outra *thread* de outro usuário.

## Conclusão

Com isso é seguro ter sempre o *EntityManagerFactory* como estático.

# EntityManager

Note que temos um *EntityManagerFactory* fixo agora e o único objeto a chamar o método *close* é o *EntityManager*.

Entretanto em uma aplicação profissional não fica bom chamar uma *EntityManager* de uma classe com um método *main*.

É muito comum encontrar pela internet uma classe chamada de *JpaUtil*. Ela seria utilizada em diversos locais do sistema.

*JpaUtil* não é um *pattern* oficializado, é apenas um modo de diminuir o acoplamento da aplicação.

# Mapeamentos

```
@Entity
@Table(name = "pessoa")
public class Pessoa implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Integer id;
    @Column(name = "nome")
    private String nome;
    @Column(name = "sobrenome")
    private String sobrenome;
    @OneToMany(mappedBy = "pessoa", cascade = CascadeType.REMOVE)
    private Collection<Dependente> dependenteCollection;
```

# Mapeamentos

```
@Entity
@Table(name = "dependente")
public class Dependente implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id")
    private Integer id;
    @Column(name = "nome")
    private String nome;
    @Column(name = "sobrenome")
    private String sobrenome;
    @JoinColumn(name = "pessoa_id", referencedColumnName = "id")
    @ManyToOne
    private Pessoa pessoa;
```

# @Table

A anotação **@Table** possui o atributo **name** que define o tabela caso não siga o mesmo nome da entity; **catolog** indica o catálogo de metadados do banco; **schema**, a qual esquema a tabela pertence; **uniqueConstraints** indica quais **Constraints** únicas devem existir na tabela (Esse valor só é verificado se a JPA for criar tabelas).

```
@Table(name = "PESSOA_TB",
       catalog = "db_catalog",
       schema = "table_schema",
       uniqueConstraints={
         @UniqueConstraint(
            columnNames={"codigo", "nome"}
         ),
         @UniqueConstraint(
            columnNames={"outroCampo", "outroNome"}
```

# @Column

## + Atributos de @Column

- **name**: indica o nome da coluna do banco de dados para aquele atributo;
- **length**: indica o tamanho do campo, geralmente aplicado a campos de texto;
- **unique**: indica se pode haver valores repetidos naquela coluna;
- **nullable**: indica se a coluna aceita informações **null**;
- **columnDefinition**: permite definir a declaração de como é a coluna, usando a sintaxe específica do banco de dados;
  - O problema de definir essa opção é que a portabilidade entre bancos pode ser perdida;

# @Column

## Atributos de @Column

- **insertable** e **updatable**: indica se aquele campo pode ser alterado ou ter valor inserido;
- **precision** e **scale**: servem para tratar números com pontos flutuantes, como **double** e **float**;
- **table**: indica qual tabela aquela coluna pertence;

# @Basic

Para a anotação @Basic temos dois atributos:

- **optional**: define se o valor pode estar **null** na hora da persistência
- **fetch**: indica se o conteúdo será carregado juntamente com a *Entity* quando ela for buscada no banco de dados

Essa anotação é padrão para todo atributo de uma classe, ou seja, anotar um atributo com @Basic é o mesmo que não anotar.

```
@Basic(optional = true, fetch = FetchType.EAGER)
```

# @Transient

A anotação **@Transient** informa que esse campo não deve ser persistido pelo JPA, desse modo podemos ter sempre o valor atualizado dinamicamente;

```
@Entity
public class Aluno {
    @Transient
    private long notaTotal;

    public Aluno(String nome, int idade, long notaTotal) {
        this.nome = nome;
        this.idade = idade;
        this.notaTotal = notaTotal;
    }

    // outras coisas
}
```

# Como mapear herança da melhor maneira?

Para trabalhar com hierarquias entre classes, a JPA fornece diversas alternativas.

Para herança é possível utilizar as estratégias:

- **mapped superclass**
- **single table**
- **joined**
- **table per concrete class**

Cada tipo de herança tem suas vantagens e desvantagens que podem influenciar no tipo de manutenção dos dados, investigação de informações e na estruturação das classes.

# Mapped Superclass

Pode acontecer que uma entity herde de uma classe que não seja entity.

```
@MappedSuperclass
public abstract class Departamento {

    private String nome;

    public abstract void calcularDespesasDoMes();

    // get e set
}
```

Veja que **Departamento** está anotada com **@MappedSuperclass** que indica que pode ser utilizada em herança de entity.

# Mapped Superclass

Tem-se por boa prática deixar uma **@MappedSuperclass** como abstrata, apenas para definir uma herança.

- Como esse tipo de classe não pode ser consultada ou persistida não existe motivo para deixá-la como concreta.

Uma **@MappedSuperclass** é ideal para quando se tem uma herança na qual a primeira classe da hierarquia não é persistida e nem consultada diretamente por HQL/JPQL

# Table per class

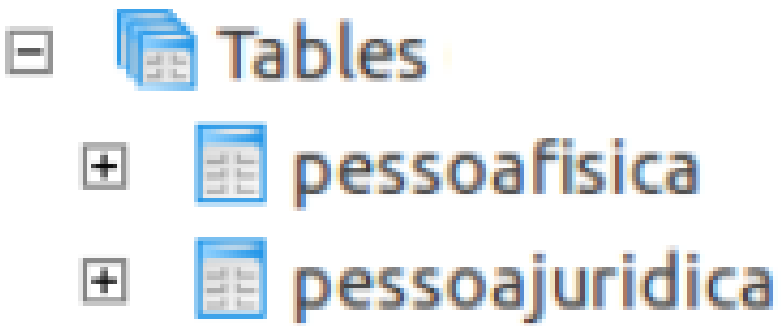
**TABLE PER CLASS** trabalha com a ideia de uma tabela por classe concreta.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Pessoa { ... }
```

# Table per class

```
@Entity
public class PessoaFisica extends Pessoa{ ... }

@Entity
public class PessoaJuridica extends Pessoa{ ... }
```

# Table per class



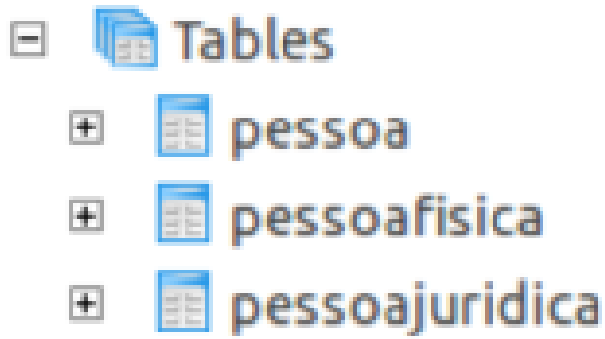
# Joined

A estratégia **JOINED** utiliza da abordagem de uma tabela para cada entity da herança.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Pessoa { ... }

@Entity
public class PessoaFisica extends Pessoa { ... }

@Entity
public class PessoaJuridica extends Pessoa { ... }
```

## Joined

É possível ver nas classes **Pessoa**, **PessoaJuridica** e **PessoaFisica**, apenas o tipo de herança foi definido na superclasse e nada mais.




O tipo **JOINED** tem por característica criar uma tabela por **entity**, então desta vez teremos 3 tabelas no banco de dados.

# Joined

```
select * from pessoa
```

| id integer | nome character varying(255) |
|---|---|
| 1 | Pessoa Fisica 01 |
| 2 | Pessoa Fisica 02 |
| 3 | Pessoa Juridica |
| 4 | Pessoa Juridica |
| 5 | Pessoa Juridica |

```
select * from pessoafisica
```

| id integer | cpf character varying(255) |
|---|---|
| 1 | CPF 11111111 |
| 2 | CPF 22222222 |

# Single Table

É possível ter toda a herança persistida e com seus dados em uma única tabela.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "pertence_a_classe")
public abstract class Pessoa {

    @Id
    @GeneratedValue
    private int id;
    private String nome;
```

**@DiscriminatorColumn** informa qual o nome da coluna que armazenará a entity "dona"de uma determinada linha no banco de dados.

# Single Table

```java
@Entity
@DiscriminatorValue("PessoaJuridica")
public class PessoaJuridica extends Pessoa{
    private String CNPJ;
    // outras coisas
}
```

```java
@Entity
@DiscriminatorValue("PessoaFisica")
public class PessoaFisica extends Pessoa{
    private String CPF;
    // outras coisas
}
```

**@DiscriminatorValue**: valor que determinará cada classe na tabela do banco de dados.

# Single Table

```
select * from pessoa
```

Output pane

Data Output | Explain | Messages | History

| | id integer | pertence_a_classe character varying(31) | nome character varying(255) | cpf character varying(255) | cnpj character varying(255) |
|---|---|---|---|---|---|
| 1 | 1 | PessoaFisica | Pessoa Fisica 01 | CPF 11111111 | |
| 2 | 2 | PessoaFisica | Pessoa Fisica 02 | CPF 22222222 | |
| 3 | 3 | PessoaJuridica | Pessoa Juridica | | CNPJ 44444444 |
| 4 | 4 | PessoaJuridica | Pessoa Juridica | | CNPJ 55555555 |
| 5 | 5 | PessoaJuridica | Pessoa Juridica | | CNPJ 66666666 |

# Relacionamentos

- @OneToOne
- @OneToMany e @ManyToOne
- @ManyToMany

# @OneToOne

```
@Entity
public class Pessoa {
    @Id
    private id;
    private nome;

    @OneToOne
    private Endereco endereco;
    // get and set
}
@Entity
 public class Endereco {
     @Id
     private id;
     private String nomeRua;
 }
```

# @OneToMany e @ManyToOne

```
@Entity
public class Pessoa{
    // outras informações
    @OneToMany
    private List<Cachorro> cachorros;
}
```

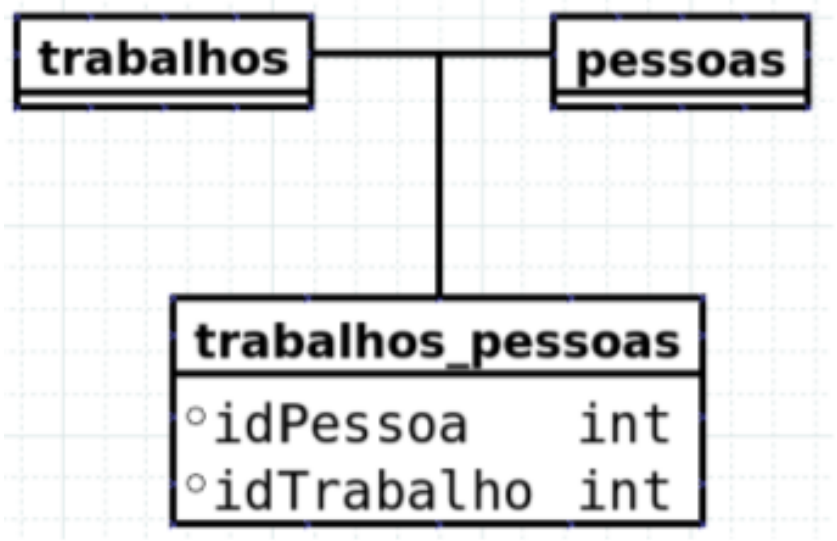
# @OneToMany e @ManyToOne

```
@Entity
public class Cachorro {
    @Id
    @GeneratedValue
    private int id;

    private String nome;

    @ManyToOne
    @JoinColumn(name = "pessoa_id")
    private Pessoa pessoa;
}
```

Veja que agora a entidade `Cachorro` tem uma referência para `Pessoa` e com a anotação `@JoinColumn`. E apenas um pequeno ajuste precisa ser feito na entidade `Pessoa`:

```
@Entity
public class Pessoa {
    // outras coisas
    @OneToMany(mappedBy = "pessoa")
    private List<Cachorro> cachorros;
}
```

# @ManyToMany


trabalhos
pessoas
trabalhos_pessoas
idPessoa int
idTrabalho int

# @ManyToMany

```java
@Entity
public class Pessoa{
    // outras informações
    @ManyToMany
    @JoinTable(name = "trabalhos_pessoas")
    private List<Trabalho> trabalhos;
}
```

- A entidade Pessoa está sendo marcada como dona do relacionamento pela anotação @JoinTable.
- Deste modo como está mapeado anteriormente, o relacionamento é unidirecional com uma tabela extra que armazenará as chaves de cada tabela.

# @ManyToMany

E se precisássemos criar o relacionamento do lado de *Trabalho* também?

Para definir qual o lado não dominante em um relacionamento bidirecional basta fazer como a seguir:

```
@Entity
public class Trabalho {

    @Id
    @GeneratedValue
    private int id;
    private String nome;
    @ManyToMany(mappedBy = "trabalhos")
    private List<Pessoa> funcionarios;

}
```

# Esqueça SQL! Abuse da JPQL

Uma grande facilidade da JPA é justamente a portabilidade.

- É possível migrar o banco de dados sem necessariamente precisar alterar os códigos escritos.
- Quem já sofreu dando manutenção a uma aplicação que tem como requisito rodar em diversos bancos de dados diferentes sabe a dificuldade que é.

# Esqueça SQL! Abuse da JPQL

Imagine um sistema de controle de chamados. Ele registra dados das ligações, dos problemas levantados pelos clientes e o status atual desse chamado.

Esse sistema rodará dentro do cliente, e cada cliente pode usar a infraestrutura que quiser, e isso inclui o banco de dados que ele achar melhor.

Note que nossa aplicação poderá ter contato com uma grande variedade de **fabricantes** de banco de dados.

# Esqueça SQL! Abuse da JPQL

O problema de a mesma aplicação rodar em diversos bancos é justamente a sintaxe de cada *query* a ser executada.

Uma *query* simples funciona em qualquer banco de dados:

- *select * from cachorros*

Mas tudo pode ficar complicado quando precisarmos limitar a quantidade de linhas retornadas em cada consulta.

# Esqueça SQL! Abuse da JPQL

Cada banco de dados faz isso à sua maneira:

```
# MySQL
select * from cachorros LIMIT 10;
# Postgres
select * from cachorros LIMIT 10;
# MS SQLServer
select top 10 * from cachorros;
# Oracle
select * from cachorros where rownum <= 10;
```

Veja que a simples ação de limitar a quantidade de linhas retornadas já geraria bastante dor de cabeça para a aplicação que utiliza diversos bancos.

# Esqueça SQL! Abuse da JPQL

Para contornar esse problema seria necessário:

1. Criar uma consulta para cada tipo de banco diferente dentro do *DAO/Repository* ou;
2. Fazer uma série de *if* para verificar qual banco está sendo utilizado ou;
3. Utilizar um arquivo externo com a sintaxe de cada banco.
   - Nesse caso teríamos que, a cada nova consulta criada na aplicação, replicar as consultas para cada arquivo de consulta de cada banco.

# Esqueça SQL! Abuse da JPQL

Para ajudar nesse problema de diferentes sintaxes para os diferentes bancos, a JPA vem com a linguagem chamada JPQL (*Java Persistence Query Language*).

A sintaxe da JPQL se assemelha muito a uma *query* normal.

# Parâmetros com JPQL

Para isso, é possível passar valores para a cláusula *where* da sua consulta através da sintaxe da *JPQL* para parametrização:

```
select c from Cachorro c where c.idade = :idade
```

Chamamos essa abordagem de parametrização de parâmetros nomeados. Para passar o valor para a consulta, basta fazer como:

```
String consulta = "select c from Cachorro c where c.idade = :idade";
TypedQuery<Cachorro> query =
        entityManager.createQuery(consulta, Cachorro.class);
query.setParameter("idade", 33);
```

# Parâmetros com JPQL

Vamos analisar por partes essa consulta: **select c from Cachorro c** realiza uma busca no banco de dados trazendo todos os cachorros cadastrados.

Note que não existe o * , mas existe a letra **c** que informa quais objetos serão retornados, e não campos. Nesse caso, o alias **c** foi dado aos objetos do tipo Cachorro.

# Parâmetros com JPQL

Uma das vantagens de utilizar JPQL é que a JPA já converte o resultado da con- sulta em objetos. Não é necessário buscar linha por linha e coluna por coluna no objeto como no JDBC. Caso um atributo a mais seja inserido na entidade, a consulta já virá com esse campo populado.

# Parâmetros com JPQL

A JPQL na verdade é uma linguagem baseada em objetos. Ao invés de descrevermos como ficará a ligação das tabelas em uma query, escrevemos como os objetos se relacionam.

Por isso que, em uma JPQL, é descrito algo como select p from Pessoa p join p.emails e where e.titulo = "ABC", e a sintaxe da query é feita toda em cima de como o objeto e seus atributos estão escritos.

# Parâmetros com JPQL

```
String consulta = "select c from Cachorro c
                   where c.dataRegistro < :data";
TypedQuery<Cachorro> query =
        entityManager.createQuery(consulta, Cachorro.class);
Data dataAtual = new Date();
query.setParameter("data", dataAtual);
```

```
String consulta = "select c from Cachorro c where c.dono = :dono";
TypedQuery<Cachorro> query =
        entityManager.createQuery(consulta, Cachorro.class);
query.setParameter("dono", dono);
```

# Navegação nas Pesquisas

```
#1 select p from Pessoa p join p.carros
#2 select p from Pessoa p left join p.carros
#3 select p from Pessoa p left join fetch p.carros
```

```
String consulta = "select p from Pessoa p join p.carros";
TypedQuery<Pessoa> query =
          entityManager.createQuery(consulta, Pessoa.class);
List<Pessoa> resultado = query.getResultList();
```

# Name Queries

Imagine um sistema em que um desenvolvedor precise consultar a data de nascimento de uma pessoa para exibir sua idade.

- Em outro momento, outro desenvolvedor precisará buscar pessoas pela data de nascimento para calcular o valor do plano de saúde.

Note que se não tivermos um lugar para concentrar essas consultas facilmente teremos consultas repetidas em nossa aplicação.

# Name Queries

Para ajudar a evitar repetição de consultas, podemos utilizar a chamada *NamedQuery*; uma *NamedQuery* deve ser declarada em uma *entity* e seu nome deve ser único.

```
@Entity
@NamedQueries({
    @NamedQuery(name = Aluno.BUSCAR_POR_NOME,
                query = "select a from Aluno a where a.nome = :nome"),
    @NamedQuery(name = Aluno.BUSCAR_POR_IDADE,
                query = "select a from Aluno a where a.idade = :idade")
})
public class Aluno {

  public static final String BUSCAR_POR_NOME = "Aluno.BuscarPorNome";
  public static final String BUSCAR_POR_IDADE = "Aluno.BuscarPorIdade";
    // outras coisas
}
```

# Name Queries

```
TypedQuery<Aluno> query =
    entityManager.createNamedQuery(Aluno.BUSCAR_POR_NOME, Aluno.class);
query.setParameter("nome", nome);
Aluno aluno = query.getSingleResult();
```

Veja que não foi necessário digitar a *JPQL* novamente, bastou fazer referência ao seu nome através do método *createNamedQuery*.

# Queries Nativas

Ao utilizar JPA é necessário ter em mente que, um dos motivos pelo qual a JPA foi desenvolvida é: permitir a portabilidade entre bancos.

A **desvantagem** é que recursos e otimizações específicas do banco de dados não podem ser utilizados.

# Queries Nativas

Imagine um sistema em que a senha do aluno será salva em *MD5*. Ao tentar fazer o login, nós devemos comparar a senha que o usuário digitou com a senha salva.

Para facilitar, vamos utilizar a função *md5* do banco *postgres*:

```
String consulta = "SELECT * FROM Aluno a where
                   md5(a.senha) = :senha";

Query query = entityManager.createNativeQuery(consulta, Aluno.class);
query.setParameter("senha", senha);
Aluno aluno = (Aluno) query.getSingleResult();
```

# Queries Nativas

Veja no exemplo que a função *md5(...)* foi chamada de dentro da consulta.

A sintaxe de uma *NativeQuery* é a mesma do banco em que está sendo executada, lembrando que para criá-la, o método chamado é *createNativeQuery*.

Perceba que ele não retorna uma *TypedQuery*, portanto será necessário fazer *cast* do objeto retornado pela consulta.

# Queries Nativas

## Importante

Apesar da facilidade de uso, é preciso ficar atento a uma eventual mudança de banco de dados, pois podem ocorrer erros nas consultas, já que elas foram pensadas em um tipo de banco de dados específico.

# Entenda as queries programáticas com Criteria

Imagine uma tela que tenha onde diversas informações poderiam ou não participar de uma consulta.

Note que o botão de pesquisa pode ser acionado com uma ou mais opções selecionadas.

# Entenda as queries programáticas com Criterias queries

Ao utilizar JPQL com consulta dinâmica teremos um problema bem chato de criar a String . Veja o código a seguir:

```
String consulta = "select c from Cachorro c where 1=1 " +
if(nome != null){
    consulta += " and c.nome = :nome"
}
if(idade != null){
    consulta += " and c.idade = :idade"
}
// outros ifs
TypedQuery<Cachorro> query =
                  entityManager.createQuery(consulta, Cachorro.class);

 if(nome != null){
     query.setParameter("nome", nome);
 }
```

# Entenda as queries programáticas com Criterias queries

Outro problema é que se a query tiver os dois parâmetros.

```
String consulta = "select c from Cachorro c where 1=1 " +
if(nome != null){
    consulta += " and c.nome = :nome"
}
if(idade != null){
    consulta += " and c.idade = :idade"
}
// outros ifs
TypedQuery<Cachorro> query =
                  entityManager.createQuery(consulta, Cachorro.class);

 if(nome != null){
     query.setParameter("nome", nome);
 }

 if(idade != null){
     query.setParameter("idade",idade);
 }
```

# Entenda as queries programáticas com Criterias queries

Para esse tipo de situação em que uma consulta pode ter tantas possibilidades de erro na sintaxe, existe a chamada Criteria. É uma ótima solução, mas infelizmente de uso bem complexo.

# Entenda as queries programáticas com Criterias queries

```
EntityManager entityManager = emf.createEntityManager();

CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaQuery<Cachorro> criteriaQuery =
        criteriaBuilder.createQuery(Cachorro.class);
Root<Cachorro> root = criteriaQuery.from(Cachorro.class);

List<Predicate> condicoes = new ArrayList<Predicate>();

if(nome != null){
    Path<String> atributoNome = root.get("name");
    Predicate whereNome = criteriaBuilder.equal(atributoNome, nome);
    condicoes.add(whereNome);
}

if(idade != null){
    Path<Integer> atributoIdade = root.get("idade");
    Predicate whereNome = criteriaBuilder.equal(atributoIdade, idade);
    condicoes.add(whereNome);
}
```

# Entenda as queries programáticas com Criterias queries

Apesar do código extremamente extenso, é possível ver agora como a consulta dinâmica começa a facilitar a vida. Não foi necessário adicionar o parâmetro e depois o valor. Automaticamente a própria JPA fez essa tarefa.

É justamente nesse enorme código acima que foi possível ver a desvantagem da Criteria da JPA. É muito código e muita complexidade para realizar apenas uma consulta com parâmetros dinâmicos.

# EasyCriteria

Devido a esse problema de Criteria, o framework EasyCriteria trabalha com a Criteria da JPA de modo simples

O objetivo do framework é utilizar o Criteria da JPA e ainda assim manter a portabilidade do projeto entre suas maiores implementações: OpenJPA, EclipseLink e Hibernate.

# EasyCriteria

```
EntityManager em = getEntityManagerFactory().createEntityManager();

EasyCriteria easyCriteria =
          EasyCriteriaFactory.createQueryCriteria(em, Cachorro.class);

if(nome != null){
    easyCriteria.andEquals("nome", nome);
}

if(idade != null){
    easyCriteria.andEquals("idade", idade);
}

List<Cachorro> resultList = easyCriteria.getResultList();
```

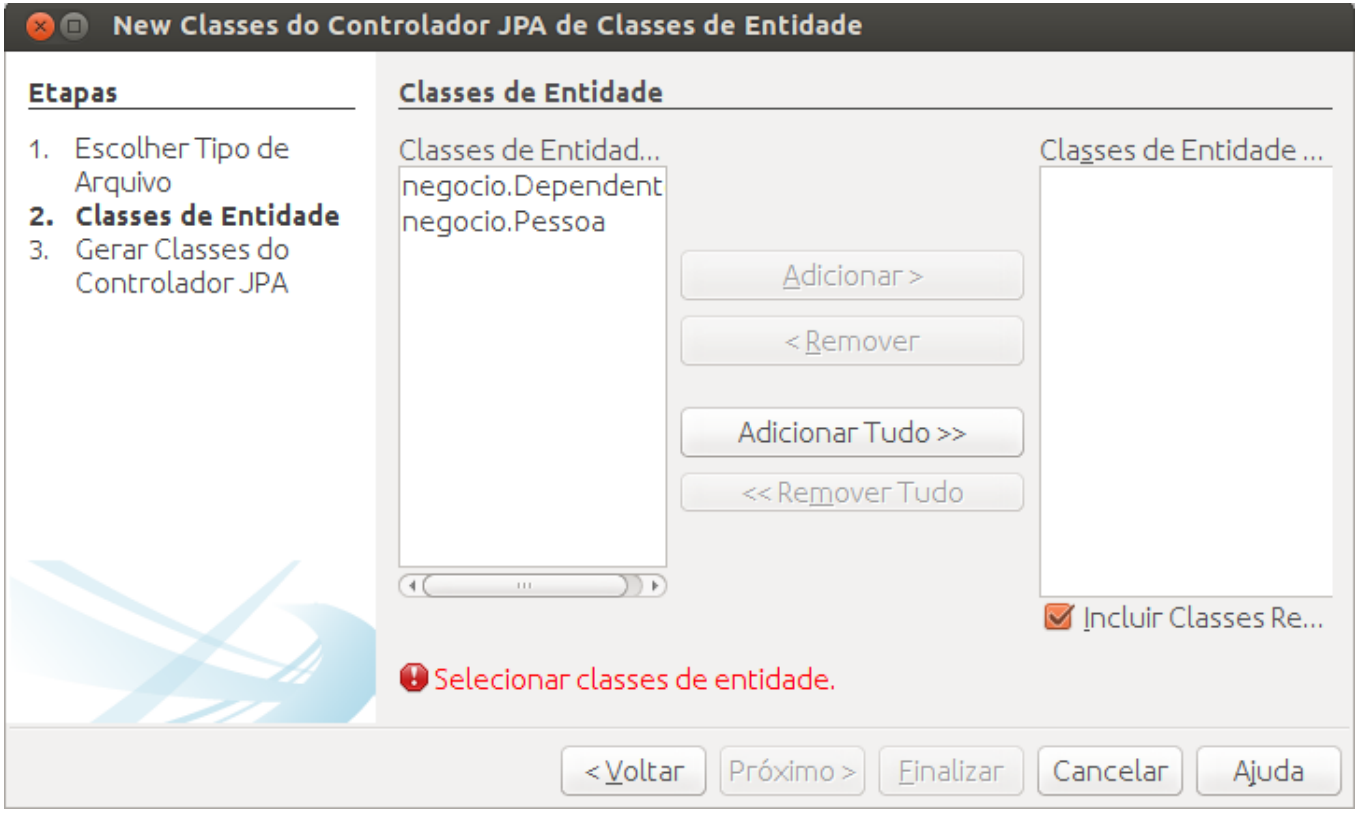
# EasyCriteria

Um detalhe interessante do EasyCriteria é que ele pode ser baixado diretamente do Maven com a dependência que segue:

```
<dependency>
    <groupId>uaihebert.com</groupId>
    <artifactId>EasyCriteria</artifactId>
    <version>3.0.0</version>
</dependency>
```

Seu site oficial é: `http://easycriteria.uaihebert.com`

# JPAController



# JPAController

```
public class PessoaJpaController implements Serializable {

    public PessoaJpaController(EntityManagerFactory emf) {
        this.emf = emf;
    }
    private EntityManagerFactory emf = null;

    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }

    public void create(Pessoa pessoa) {
        EntityManager em = null;
        try {
            em = getEntityManager();
            em.getTransaction().begin();
            em.persist(pessoa);
            em.getTransaction().commit();
        } finally {
            if (em != null) {
                em.close();
            }
        }
    }

    public void edit(Pessoa pessoa) throws NonexistentEntityException, Exception {
        EntityManager em = null;
        try {
            em = getEntityManager();
```

# JPAController

```
public class Main {
  private static final EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory(

  public static void main(String[] args) throws NonexistentEntityException {
      EntityManager entityManager = entityManagerFactory.createEntityManager();
      entityManager.getTransaction().begin();
      List<Pessoa> vetPessoa = new PessoaJpaController(entityManagerFactory).findPessoaEntities();
      for (int i = 0; i < vetPessoa.size(); i++) {
          Pessoa pessoa = vetPessoa.get(i);
          System.out.println("Pessoa:"+pessoa.getNome());
      }
      entityManager.getTransaction().commit();
      entityManager.close();
  }
}
```

# JPA

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Câmpus Rio Grande
Divisão de Computação