UNIVERSIDADE FEDERAL DE JUIZ DE FORA DEPARTAMENTO DE CIÊNCIAS DA COMPUTAÇÃO

Filipe Bastos

Como unir o mundo orientado a objetos ao mundo relacional utilizando ferramentas ORM

Juiz de Fora, MG

2016

UNIVERSIDADE FEDERAL DE JUIZ DE FORA DEPARTAMENTO DE CIÊNCIAS DA COMPUTAÇÃO

Filipe Bastos

Como unir o mundo orientado a objetos ao mundo relacional utilizando ferramentas ORM

Monografia de conclusão de curso apresentada ao curso de Pós-graduação em Desenvolvimento de Sistemas com Tecnologia Java do Departamento de Ciência da Computação da Universidade Federal de Juiz de Fora, como requisito parcial à conclusão do curso.

Orientador: Tarcísio de Souza Lima

Juiz de Fora, MG

2016

Filipe Bastos

Como unir o mundo orientado a objetos ao mundo relacional utilizando ferramentas ORM

Monografia de conclusão de curso apresentada ao curso de Pós-graduação em Desenvolvimento de Sistemas com Tecnologia Java do Departamento de Ciência da Computação da Universidade Federal de Juiz de Fora, como requisito parcial à conclusão do curso.

Aprovado em 2 de maio de 2017

Tarcisio de Souza Lima
Examinador UFJF
Orientador

Daves Márcio Silva Martins
Examinador UFJF

Tatiane Ornelas Martins Alves
Examinadora UFJF

Juiz de Fora, MG
2016

Agradecimentos

Minha esposa pela paciência e ajuda em momentos difíceis. Minha mãe que sempre me fez seguir em frente.
Meus professores por alimentar meus conhecimentos.

“A percepção do desconhecido é a mais fascinante das experiências. O homem que não tem os olhos abertos para o misterioso passará pela vida sem ver nada.”

(Albert Einsten)

Resumo

O desenvolvimento de sistemas vem evoluindo a cada dia e fazer integração entre metodologias diferente vem se tornando rotina no dia a dia dos desenvolvedores, o que faz com que, em muitas situações, o foco no negócio seja desviado para tratar de problemas técnicos, qual faz com a produtividade seja diminuída. Um clássico exemplo de metodologias diferentes é a do paradigma de programação orientada a objetos e o tão popular banco de dados relacional. Surgiu então o conceito ORM (Object Relational Mapper) que facilitou o relacionamento e configurações entre os dois paradigmas. Ferramentas foram desenvolvidas, em principal o Hibernate, e logo notou-se a necessidade de criar um padrão foi então que surgiu a especificação do JPA.

Palavras-chave: Persistência, Framework ORM, JPA, Hibernate.

Abstract

The development of systems is evolving every day and make integration of diferent methodologies is becoming routine in the daily lives of developers, which means that, in many situations, the focus on the business to be diverted to deal with technical problems, which makes with productivity is decreased. A classic example of diferent methodologies is the programming paradigm object-oriented and so popular relational database. Then came the concept ORM (Object Relational Mapper) which facilitated the relationships and configurations between the two paradigms. Tools were developed, leading in Hibernate, and soon saw the need to create a pattern was then came the JPA specification.

Keywords: Persistence, Framework ORM, JPA, Hibernate.

Lista de ilustrações

Lista de Tabelas

Lista de abreviaturas e siglas

OO	Orientação a Objetos
ORM	*Object Relational Mapper*
JPA	*Java Persistence API*
POJO	*Plain Old Java Objects*
API	*Application Programming Interface*
JDBC	*Java Database Connectivity*
SQL	*Structured Query Language*
JPQL	*Java Persistence Query Language*
XML	*eXtensible Markup Language*

Sumário

1. Introdução

No dia a dia dos ambientes corporativos, é comum a manipulação de dados em grande escala, dados estes, que em sua grande maioria, são armazenados em banco de dados relacionais e que facilmente são manipulados por seus sistemas gerenciadores que também trabalham no modelo relacional. No passar dos anos o desenvolvimento de aplicações orientadas a objeto em destaque a linguagem Java, que foi a escolhida para descrever este processo, ganhou espaço no mercado, mas ao mesmo tempo trouxe um grande desafio de unir dois paradigmas distintos, orientação a objetos e modelo relacional.

A princípio a criação da API Java, *Database Connectivity* (JDBC), que de acordo com o Oracle, é o padrão da indústria para a conectividade independente de banco de dados entre a linguagem de programação Java, resolveu a questão de envio de instruções SQL aos bancos de dados relacionais, mas rapidamente notou-se uma queda de produtividade, quando o desenvolvedor tornava-se responsável por criar tais instruções, cabendo a ele tratar a mudança do paradigma orientado a objetos para o mapeamento objeto relacional e vice-versa. Nesta fase o desenvolvedor escrevia manualmente todas as instruções que eram necessárias para persistência e recuperação dos dados.

Esta queda de produtividade estimulou o desenvolvimento de mecanismos que ajudassem a melhorar este cenário, o objetivo era criar ferramentas poderosas que cuidassem de toda tarefa árdua de escrever instruções relacionais ou pelo menos abstraísse a maior parte delas, deixando o desenvolvedor com maior tempo para cuidar da lógica de negócio.

Assim foi criado o Hibernate, um *framework* de mapeamento objeto-relacional ou ORM, (do inglês: *Object-relational mapping*), que segundo Bauer, era um projeto ambicioso com pretensão de ser a solução completa aos problemas de gerenciamento de dados persistentes em Java. Sendo que, entre as várias características do Hibernate, a principal é que, ele transforma classes Java em entidades de banco de dados, através de arquivos de configuração no formato XML ou anotações Java, gerando também as chamadas SQL que mantém a aplicação portável a quaisquer bancos de dados.

Após o surgimento do Hibernate e do conceito de ORM outros *frameworks* de persistência foram sendo criados, notou-se então, a necessidade de criar um padrão a qual esses *frameworks* implementariam. Assim, um grupo de desenvolvedores originou uma especificação, baseada em conceitos trazidos pelo Hibernate, surgindo então o ***Java Persistence API*** (**JPA**).

2. JPA e a escolha de uma implementação

Partindo do entendimento que havia muitos tipos de frameworks ORM[1] para resolver o mesmo problema, foi criado o JPA. Conforme explicado por Cordeiro:

> "Justamente com o intuito de resolver essa situação, em 2006, foi criada uma especificação para as bibliotecas ORM, no caso, a Java Persistence API - JPA. Essa nova especificação não representou grandes alterações na forma de usar um ORM, para quem já estava acostumado com o Hibernate. Entre algumas poucas diferenças, destacam-se os nomes dos métodos e das classes que devem ser usadas [...]."
> (CORDEIRO, 2012. P. 12)

Assim, de acordo com Coelho, toda especificação precisa de uma implementação para que possa ser utilizada em um projeto, que são os *frameworks* ORM. Para o JPA, há várias implementações no mercado, como Hibernate, EclipseLink, OpenJPA, Batoo, entre outras, sendo que cada uma delas possui suas vantagens e desvantagens na hora da utilização.

A mais famosa e frequentemente utilizada no mercado é o Hibernate, a qual a JPA baseou-se para iniciar suas normas, por essa razão foi usado o Hibernate implementando o JPA para explicar fundamentos práticos.

> "Uma implementação da JPA deve satisfazer todas as interfaces determinadas pela especificação. Pode acontecer de uma implementação ter 80% de todas as interfaces implementadas e ainda assim lançar uma versão para utilização; fica a critério de cada uma garantir que tudo esteja corretamente implementado. Caso não esteja, usuários escolherão não utilizar. Para implementar a especificação, é necessário implementar as interfaces que estão no pacote javax.persistence.* , que é justamente o papel do Hibernate, EclipseLink etc. Mesmo uma implementação incompleta pode ser considerada válida. Na versão 3.x do Hibernate, caso um desenvolvedor adicionasse a anotação @NamedNativeQuery , que permite que uma query nativa (com a sintaxe do banco de dados) fosse declarada em uma entity (entidade), uma exceção seria lançada pelo Hibernate informando que essa funcionalidade ainda não estava implementada[...]."

[1] ORM é uma técnica de mapeamento objeto relacional que visa criar uma camada de mapeamento entre nosso modelo de objetos (aplicação) e nosso modelo relacional (banco de dados) de forma a abstrair o acesso ao mesmo.

(COELLHO, 2013. P. 6)

3. Ciclo de vida do JPA

Todo objeto tem seu ciclo de vida, desde o momento em que é instanciado até sua falta de existência, na verdade estamos falando dos estados em que um objeto pode se encontrar. No JPA esse estado é definido pela existência do objeto na Entity Manager conforme ilustrado na Figura 1.

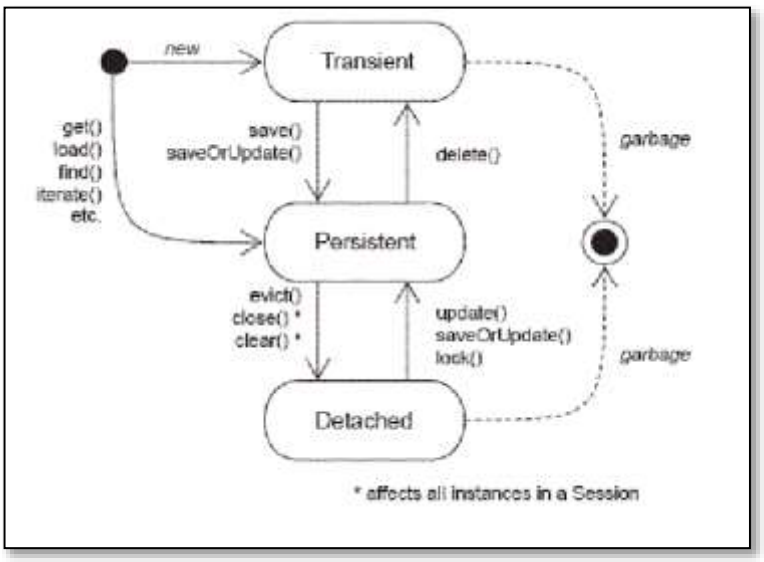


Figura 01 – Fluxograma ciclo de vida do JPA

- Transiente: A instância não é, e nunca foi associada com nenhum contexto persistente, não possui uma identidade persistente (valor da *primary key*).
- Persistente: A instância está atualmente associada a um contexto persistente. Possui uma identidade persistente (valor da *primary key*) e, talvez, corresponda a um registro na base, para um contexto persistente em particular, o JPA garante que a identidade persistente equivale à identidade Java (posição de memória do objeto).
- Detached: A instância foi associada com um contexto persistente, mas foi fechado, ou ocorreu serialização por outro processo. Possui uma identidade persistente, e, talvez, corresponda a um registro no banco de dados, para instâncias desatachadas, o JPA não garante o relacionamento entre identidade persistente e identidade Java.

4. Persistência

Para armazenar os dados precisamos converter nossos objetos em registros de tabelas que nada mais é do que linhas contendo os dados referentes ao objeto em questão. Esse processo de armazenamento é chamado de persistência onde o dado é passado da memória para o disco físico.

Podemos optar por mapear nossas classes em nossa aplicação por XML ou por anotações. Temos que realizar algumas configurações que são necessárias para que a comunicação entre a aplicação e o banco de dados aconteça. No JPA o arquivo que possui as informações sobre essa comunicação é o *persistence.xml*, que deve estar alocado na pasta WEB-INF.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns=http://java.sun.com/xml/ns/persistence
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.x
   sd">
  <persistence-unit name=" conexaoPU " transaction-
  type="RESOURCE_LOCAL">
  <class> </class>
    <properties>
                  </properties
  > </persistence-unit>
</persistence>
```

Abaixo temos um exemplo do arquivo.

Figura 02: Exemplo de aplicação por XML ou por anotações

Veja que no arquivo habilitamos a *persistence-unit* com um nome “conexaoPU”, poderia ser qual qualquer outro, importante lembrar desse nome pois iremos utilizá-lo na classe Java e um tipo de transação no nosso caso

RESOURCE_LOCAL que indica que nós gerenciaremos nossas transações com o banco de dados.

De acordo com as especificações do JPA, ao utilizarmos um servidor de aplicações JavaEE não precisaríamos especificar as entidades nas tags < class > ... < /class > desde que tenhamos apenas uma persistence-unit. Por sua vez, nas tags < properties >>... < /properties > iremos definir sub-tags de propriedades que descrevem informações do banco de dados como o dialeto, o driver, nome do banco, [illegible] banco de dados p

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistenc
     e
     http://java.sun.com/xml/ns/persistence/persistence_2_
     0.xsd">
  <persistence-unit name="conexaoPU" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property
         name="javax.persistence.jdbc.driver"
         value="org.postgresql.Driver"/>
      <property name="javax.persistence.jdbc.url"
         value="jdbc:postgresql://localhost:5432/t
         cc"/>
      <property name="javax.persistence.jdbc.user"
      value="postgres"/> <property
      name="javax.persistence.jdbc.password" value="postgres"/>

      <property name="hibernate.dialect"
         value="org.hibernate.dialect.PostgreSQLDial
         ect"/>
      <property name="hibernate.hbm2ddl.auto"
        value="create"/> <property
        name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql"
        value="true" />
    </properties>
  </persistence-
unit>
</persistence>
```

Figura 03: Demonstração POSTGRESQL como banco de dados

Ainda dentro da tag < *properties* > podemos definir as credenciais de conexão com o banco de dados, driver e etc. Podemos definir ainda, qual o modo de geração das tabelas no banco de dados definidas pelas seguintes opções:

- *create* - as tabelas são criadas pelo próprio Hibernate e excluídas somente no final da execução do programa caso solicitado pelo desenvolvedor;
- *update* - nada é excluído, apenas criado, ou seja, todos os dados são mantidos. Útil para aplicações que estão em produção, das quais nada pode ser apagado.
- validate - não cria e nem exclui nada, apenas verifica se as entidades estão de acordo com as tabelas e, caso não esteja, uma exceção é lançada.

No contexto JPA, as classes que têm vínculo com a camada de persistência são chamadas de entidades, elas terão papel fundamental para elaboração das tabelas e seus atributos serão mapeados como colunas no banco de dados.

5. Mapeamento Objeto-Relacional com *An-notations*

Para transformar uma classe em uma entidade utilizamos a anotação *@Entity*. Os atributos das classes também podem ser anotados, essas informações serão cruciais para definir, por exemplo, se uma coluna poderá receber valores nulos ou não, se terá valor único e outras definições.

```
imports ...;

@Entity
public class Produto implements Serializable {

   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy =
   GenerationType.IDENTITY)private Integer id;

   @Column(nullable = false, unique =
   true) private String descricao;

  /* Getters e Setters
omitidos */ }
```

Figura 04: Exemplo de Anotação @Enitity

Todas as anotações são importadas do pacote "*javax.persistence"*, no qual estão todas as anotações padronizadas pela JPA. O "*@Id"* identifica a chave primária da tabela e o "*@GeneratedValue"* é a geração auto incremental desta chave, assim o JPA criará a tabela e as colunas com o mesmo nome de nossa classe.

Dessa forma, já temos nosso arquivo configurado e a classe modelo, agora vamos testar a persistência de um objeto, para tanto temos que conseguir uma conexão com o banco de dados definido na *persistence-unit* e, utilizaremos a interface *EntityManagerFactory* para sabermos como conseguiremos a conexão.

Assim, para instanciá-la, basta utilizar o método "*createEntityManagerFactory*" da classe Persistence da própria JPA, indicando qual é a persistence-unit que foi definida no *persistence.xml*, que no caso chamamos de "conexaoPU".

Para melhor exemplificar esta questão, usaremos uma classe com o método *main* para testar a persistência. Veja abaixo:

```
public class ProdutoPersist {

   public static void main(String[] args) {

EntityManagerFactory emf =
Persistence.createEntityManagerFactory("conexaoPU");

        EntityManager em = emf.createEntityManager();

        Produto produto = new
        Produto();
        produto.setDescricao("Arroz
        ");

        em.persist(produto);

        em.close
        ();
        emf.clos
        e();

        System.out.println("Produto persistido com
   sucesso!"); }

}
```

Figura 05: Exemplo método *main*.

O código acima será executado sem erros, porém nada será persistido na base. Isso ocorre pelo fato de que toda vez que desejarmos alterar o banco de dados, precisamos que uma transação seja aberta para realizar tal procedimento.

Nesse sentido, ficou definido no *persistence.xml* que iriamos gerenciar nossas transações , conforme exemplificado a seguir :

```
public class ProdutoPersist {

   public static void main(String[] args) {

       EntityManagerFactory emf =
          Persistence.createEntityManagerFactory("conexa
          oPU");

       EntityManager em =
       emf.createEntityManager(); Produto
       produto = new Produto();
       produto.setDescricao("Arroz");
       EntityTransaction tx =
       em.getTransaction(); tx.begin();
       em.persist(produ
       to); tx.commit();

       em.close
       ();
       emf.clos
       e();
              System.out.println("Produto persistido com
   sucesso!"); }

}
```

Figura 06: Exemplo persistence.xml

Dessa vez a execução do código fará com que o objeto produto seja persistido de fato. Nota-se que ao executar pela segunda vez a base de dados, a mesma será removida e reconstruída pelo fato de termos definido o modo de geração das tabelas como *create*.

6. Entidades

Como falado anteriormente, em JPA trabalhamos com o conceito de entidades (Entity), o que significa que nossos POJOs, nossas classes de domínio, irão refletir informações à base de dados, as classes serão anotadas para serem tabelas e os atributos serão colunas. Para tal, a classe precisa seguir três regras:

- Deve ser anotada com *@Entity*;
- Deve ter um campo definido que representa a chave primária da tabela, anotado com *@Id*;
- Deve ter um construtor público sem parâmetros.

```
import
javax.persistence.Entity;
import
javax.persistence.Id;

@Entity
public class Produto {

  @Id
  private Long id;

  private String nome;

  public Long
    getId() {
    return id;
  }

  public void setId(Long
    id) { this.id = id;
  }

  public String
    getNome() { return
    nome;
  }

  public void setNome(String
    nome) { this.nome = nome;

}
```

Figura 07: JPA com o conceito de entidades (Entity)

Na classe “Produto”, não cita-se o nome da tabela e nem de suas colunas, neste caso o JPA mapeia o nome da classe para ser o nome da tabela e os atributos para as colunas. Assim, poderíamos escolher os nomes e alterar os comportamentos padrões nele previsto, sendo alguns, nome de coluna, nome de tabela, nome de relacionamentos, dentre outros. Nessa situação, vamos modificar o comportamento com a utilização da anotação “*@Table”* e “*@Column”*:

```
@Entity
@Table(name = "t_produto")
public class Produto { ... } // Para as tabelas

@Id
@Column(name = "id_produto")
private Long id;     // Com o @Column para as colunas
```

Figura 08: Uso da anotação @Table e @Column

As anotações “*@Table* e *@Column”* definem um relacionamento direto entre a classe e o banco de dados, alterando todos os valores padrões que a JPA seguiria. Estas anotações estão alterando o nome da tabela e nome da coluna no banco de dados através da propriedade “*name”*.

Dessa forma, a propriedade “*name”* é responsável em modificar o nome das tabelas e colunas em relação os valores padrões e, a anotação *@Entity* também tem essa propriedade que, se usada, irá modificar a forma com que o JPA reconhece essa entidade dentro do seu contexto, isso influencia, as consultas JPQL, conforme exemplo a seguir:

```
@Entity(name =
"ProdutoEntity")
@Table(name = "t_produto")
public class Produto {...}  // Na
classse select p from ProdutoEntity
p // Na JPQL
```

Figura 09: JPA na JPQL

A anotação @*Table* além da propriedade *name*, possui também as propriedades: *catalog* - que indica o catálogo de metadados do banco - *schema* - identifica qual *schema* a tabela pertence – *uniqueConstraints* - indica quais constraint únicas devem existir na tabela e pode-se usar, também, para fazer uma constraint composta. É importante atentar que esses valores serão verificados pelo JPA, somente se as tabelas forem criadas.

A anotação @*Column* possui diversas propriedades que podem alterar o comportamento dos valores de uma coluna, além da *name*, vejamos:

- *Unique*: indica se pode haver valores repetidos naquela coluna da tabela;
- *Nullable*: indica se a coluna aceita informações nulos;
- *Length*: indica o tamanho do campo, geralmente aplicado a campos de texto;
- *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 banco de dados pode ser perdida;
- *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 d*ouble* e *float*;
- *Table*: indica a qual tabela aquela coluna pertence.

Outra anotação interessante é a "*@Basic"* utilizada nos atributos. A propriedade *optional* define se o valor pode estar nulo na hora da persistência, similar ao *nullable* na anotação @*Column*, e *fetch* indica se o conteúdo será carregado juntamente com a *Entity* quando ela for carregada no banco de dados. Os tipos de *fetch* são *LAZY*, não é carregado com a *Entity* e *EAGER*, sempre será carregado com a *Entity*, veremos mais a frente sobre esses tipos, veja como ficaria: @Basic(optional = false, fetch = FetchType.LAZY) private String nome;

É interessante usar esta anotação com o *fetch = FetchType.LAZY* , por exemplo, para atributos com carregamento considerados pesado, como foto. Nesse caso conseguiríamos gerenciar quando esse atributo seria chamado, evitando objetos enormes e sem necessidade.

Toda *Entity* precisa de uma chave-primária e já vimos que para determiná-la usamos a anotação @*Id*. Após a determinação da chave-primária, anotamos o campo com @*GeneratedValue* que fará com que os valores sejam

gerados automaticamente, mas para que a geração automática de valores funcione, será preciso determinar a estratégia de geração, que será controlada pelo banco de dados, sendo importante ressaltar que estes valores sempre serão únicos.

Outra possibilidade é optar por não escolher a estratégia explicitamente, neste caso, a não escolha faz com que o JPA determine a geração como sendo do tipo AUTO por padrão. Estes valores serão gerados a partir de uma sequência global de valores, que nunca serão reiniciados, ou seja, a cada persistência uma sequência única disponibilizará o próximo valor disponível.

```
@Id
@Column(name =
"id_produto")
@GeneratedValue
private Long id;

      ou

@Id
@Column(name = "id_produto")
@GeneratedValue(strategy =
GenerationType.AUTO) private Long id;
```

Figura10: JPA – Tipo AUTO por padrão

A estratégia *IDENTITY*, tem geração de valores a partir de uma hierarquia, de modo que os valores gerados são exclusivos apenas por tipo de hierarquia, em outras palavras, a geração é a partir do tipo da classe de entidade.

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

Figura 11: JPA por estratégia IDENTITY

A estratégia *SEQUENCE*, a geração de valores desta estratégia é global e ao contrário da *AUTO* e da *IDENTITY*, ela recebe um valor assim que um

objeto passa a ser persistente, enquanto as duas anteriores recebem seu valor após o *commit* do objeto. Isso é muito válido quando se precisa do valor da chave com maior ganho em desempenho, já que será possível obtê-lo sem precisar ir até o servidor.

Esta estratégia será dividida em duas partes, a primeira consiste em anotar a classe de entidade com *@SequenceGenerator*, que utiliza as propriedades, *name*, (nome da sequência), *initialValue* (determina o valor inicial da sequência, valor padrão é 1) e *allocationsSize* (determina a quantidade de valores armazenados, por padrão o valor é 50). A segunda parte é anotar o campo dado nome da sequência previamente definido.

```
@Entity(name =
"ProdutoEntity")
@Table(name = "t_produto")
@SequenceGenerator(name = "seq", initialValue = 1, allocationSize
= 100) public class Produto {

   @Id
   @Column(name = "id_produto")
   @GeneratedValue(strategy = GenerationType.SEQUENCE,
   generator="seq") private Long id;



}
```

Figura 12: Estratégia em duas partes

Por último temos a estratégia *TABLE*, que salvará todas as chaves em uma tabela. Tabela esta, composta de duas colunas, sendo que uma indica o nome da tabela e a outra, o valor do id atual para aquela tabela.

```
@Id
@Column(name = "id_produto")
@GeneratedValue(strategy =
GenerationType.TABLE) private Long id;
```

Figura 13: Estratégia Table

Há casos em que é preciso determinar dois ou mais campo para compor a chave primária, para tal, usou-se o *composite key* ou chave composta. Existem dois tipos de chaves compostas, simples ou complexas sendo que a classe que utilizar chave composta deve obedecer algumas normas. São elas:

- Deve-se ter um construtor sem argumentos
- Deve-se implementar a interface *java.io.Serializable*
- Deve-se sobrescrever os métodos *equals* e *hashCode*

São chamadas de chaves compostas simples quando apenas os tipos do Java são utilizados como *ID* (String, Integer, Long e etc.). Existem dois modos de utilizar uma chave composta simples, utilizando a anotação *@IdClass* ou *@EmbeddedId*, o uso desta chave pode ser no material apenso a esta monografia.

Podemos definir uma chave composta complexa quando o id de uma classe é composto de outras entidades. Para tanto, foram desenvolvidas quatro modelos desse perfil, para poder melhor exemplificar esse caso (APENDICE B) .

Considerações a serem feitas:

- Utilização da anotação *@OneToOne* (veremos mais adiante, detalhes deste relacionamento), na classe CasaCachorro para determinar um relacionamento um para um com as classes Cachorro e Dono.

- Utilizamos a classe *CasaCachorroPK* como chave primária de *CasaCachorro*, esta chave é composta da Classe Cachorro e da classe Dono, que através do métodos *equals*() apontará para o id de suas respectivas classes.

- Não é necessário anotações sobre a classe *CasaCachorroPK*, por se tratar de uma classe chave.

A determinação do tipo de chave primária ou estratégia de geração de chaves é o ponto inicial para elaboração dos atributos em nossa classe de entidade,

após esta configuração é possível se deparar com várias outras possibilidades de caracterizar a classe. Assim, digamos que nossa entidade contenha um atributo do tipo *Date,* ou um atributo que seja um *Enum*, desta maneira verá que para estas situações usamos duas anotações específicas e se quiser customizar o valor que será inserido na base de dados.

Para um atributo do tipo *Date*, usamos a anotação @*Temporal*, a partir de então temos a possibilidade de salvar no banco de dados, somente a data com o *@Temporal (TemporalType.DATE)*, somente o horário com @*Temporal (TemporalType.TIME)* ou ainda, data e horário com @*Temporal* (*TemporalType.TIMESTAMP*), sendo este último o valor padrão.

Assim, para Cocian, a estrutura de enumeradores é um conjunto de constantes, organizadas em ordem de declaração. Sua funcionalidade principal é agrupar valores com o mesmo sentido dentro de uma única estrutura e também limitar ou determinar um número de valores que podem ser usados na programação.

Para os *Enums* é usada a anotação @*Enumerated*, no qual podemos optar por dois tipos de valor que serão persistidos na base de dados, *EnumType.STRING* string que é o nome do *enum* ou *EnumType.ORDINAL* que é seu valor de ordenação, de acordo com a ordem que o desenvolvedor a declarou no corpo do *Enum*, sendo este do tipo inteiro.

Uma anotação muito útil é a @*ElementCollection* onde é possível definir um relacionamento um para muito, sem a necessidade de se criar uma outra entidade. Dessa forma, seu funcionamento fará com os valores sejam automaticamente armazenados uma tabela separada com chave estrangeira para a entidade “dona”.

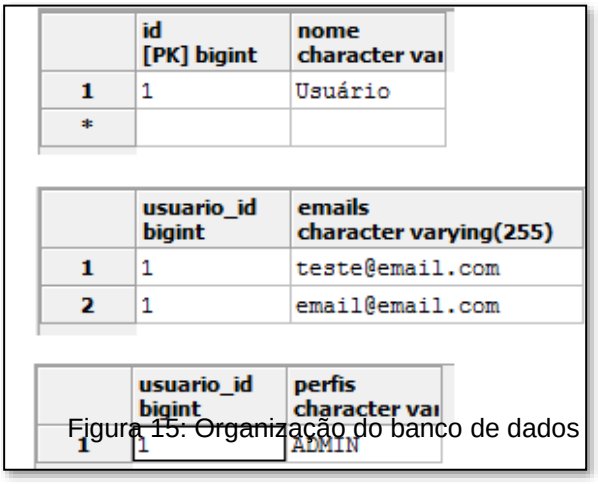
A seguir é possível ver a presença da anotação @*ElementCollection* sobre o atributo e-mails e foi utilizada também a anotação @*CollectionTable* que serve para personalizar em qual tabela as listas devem ser salvas.

```java
@Entity
public class Usuario implements Serializable {

   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue
   private Long
   id;
   private String nome;

   @ElementCollection
   @CollectionTable(name = "T_EMAIL")
   private Collection<String> emails;
   @ElementCollection(targetClass = Perfil.class)
   @Enumerated(EnumType.STRING)
   private Collection<Perfil> perfis;

   // getters and setters

   public Long getId()
      { return id;
   }


   public void setId(Long id)
      { this.id = id;
   }

   public String getNome()
      { return nome;
   }

   public void setNome(String nome) {
      this.nome = nome;
   }



   public Collection<String> getEmails() {
      return emails;
   }
```

```
    public void setEmails(Collection<String> emails) {
        this.emails = emails;
    }

    public Collection<Perfil> getPerfis() {
        return perfis;
    }

    public void setPerfis(Collection<Perfil> perfis) {
        this.perfis = perfis;
    }

}
```

Figura 14: Anotação @ElementCollection e Anotação @CollectionTable

Como padrão a JPA procuraria por uma tabela chamada *usuario_emails* se não for especificado implicitamente o nome da tabela. Vejamos então como nossos dados ficariam organizados neste contexto de relacionamento no banco de dados.



	id [PK] bigint	nome character var
1	1	Usuário
*		

	usuario_id bigint	emails character varying(255)
1	1	teste@email.com
2	1	email@email.com

	usuario_id bigint	perfis character var
1	1	ADMIN

Figura 15: Organização do banco de dados

7. Relacionamentos

De acordo com a JPA, podemos definir quatro tipos de relacionamentos, estes devem ser expressos na modelagem através de associações entre classes, de acordo com a cardinalidade adequada.

- ***One to One*** (Um para Um): Por exemplo, um país é governado por apenas um presidente e um presidente governa um país. Para representar esse relacionamento no JPA usamos a anotação *@OneToOne*,Fig.**??**.

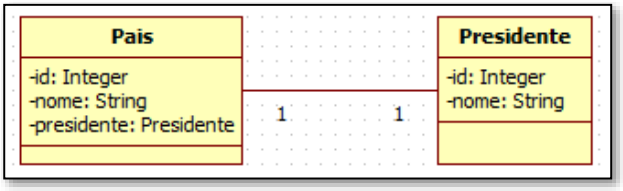


Figura 16: Relação *OneToOne*

- ***One to Many*** **(Um para Muitos) e** ***Many to One*** **(Muitos para Um):** Por exemplo, um setor possui muitos funcionários e um funcionário trabalha em apenas em um setor. Para representar esse relacionamento no JPA usamos a anotação *@OneToMany*, no objeto setor, caso o relacionamento seja bidirecional a anotação *@ManyToOne*, no objeto funcionário, Fig.04

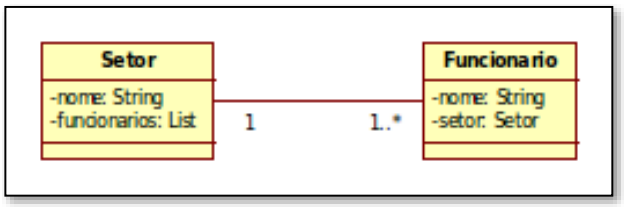


Figura 17: Relação *OneToMany*

- ***Many to Many* (Muitos para Muitos):** Por exemplo, um livro possui muitos autores e um autor possui muitos livros. A anotação é a *@ManyToMany*,

Existe a possibilidade de escolhermos entre definir um relacionamento unidirecional ou bidirecional. Para escolhermos a melhor implementação ao nosso modelo, devemos nos perguntar se necessariamente um lado precisa conhecer o outro. Veremos que ambos os caso são usuais, mas se você puder evitar o relacionamento bidirecional, será de grande ganho em desempenho.

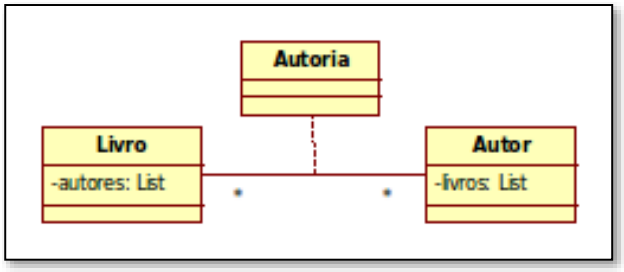


Figura 18: Relação *ManyToMany*

Além da definição da implementação unidirecional ou bidirecional, ao relacionar as entidades pode-se definir, por exemplo, se um relacionamento irá gerar uma terceira coluna, ou ainda, quando um objeto for persistido fará com que seu relacionamento seja persistido em cascata e outros, conforme exemplificado a seguir:

```
@Entity
public class Presidente implements Serializable {


   private static final long serialVersionUID = 1L;
```

```
   @Id
   @GeneratedValue
   private Long
   id;


   private String nome;
```

```
    @OneToOne
    @JoinColumn(name = "PAIS_FK")
    private Pais pais;

    // Getters and Setters
```

```
@Entity
public class Pais implements Serializable {

    private static final long serialVersionUID = 1L;
```

```
    @Id
    @GeneratedValue
    private Long
    id;

    private String nome;

    // Getters and Setters
```

Tabela 01 : Implementação unidirecional ou bidirecional

No código acima foi utilizado à anotação *@OneToOne* para definir o relacionamento um para um, note-se que ao utilizar umas das entidades, *Presidente*, que seria a "dona" do relacionamento, utilizamos também o *@JoinColumn*, que fará que a chave primária da entidade relacionada seja a estrangeira na entidade em questão. Como só anotamos o relacionamento em *Presidente* isso caracterizou uma relação unidirecional a qual só presidente precisa conhecer Pais.

Existe um problema no exemplo proposto e está relacionado ao ciclo de vida da entidade JPA, ao tentarmos persistir um presidente uma exceção do tipo *org.hibernate, a TransientPropertyValueException* será lançada, pois um objeto presidente está relacionado a um objeto pais que não existe no banco de dados, ou seja, o objeto pais não tem um id para ser a chave estrangeira em presidente.

```
public static void main(String[] args) {

      EntityManagerFactory emf =
          Persistence.createEntityManagerFactory("conexaoPU");

      // ============ // ============== //

      EntityManager em = emf.createEntityManager();
      EntityTransaction tx = em.getTransaction();
      tx.begin();

      Pais pais = new Pais();
      pais.setNome("Brasil"
                         );

      Presidente presidente = new Presidente();
      presidente.setNome("JK");
      presidente.setPais(pais);

      em.persist(presidente
      ); tx.commit();

      // ============ // ============== //


      em.close()
      ;
      emf.close(
      );

System.out.println("Persistncia realizada com sucesso!");
```

Tabela 02: TransientPropertyValueException

Basicamente o que o JPA está reclamando é que tentamos inserir um objeto persistente, que é *presidente*, relacionado a outro que está no estado transient que é pais. Podemos resolver isso das seguintes formas:

- Persistir País antes, que fará com que o JPA defina um id para o objeto:

Tabela 03: JPA para definir um objeto

```
public static void main(String[] args) {

        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("conexaoPU");

        // ============ // ============== //

        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();

         Pais pais = new Pais();
        pais.setNome("Brasil");

                em.persist(pais);
                tx.commit();
                // ============ // ============== //

                tx= em.getTransaction();
                tx.begin();

        Presidente presidente = new Presidente();
        presidente.setNome("JK");
        presidente.setPais(pais);

        em.persist(presidente);
        tx.commit();

        // ============ // ============== //

        em.close();
        emf.close();

        System.out.println("Persistncia realizada com sucesso!"); }
```

- Utilizar a persistência por cascata, utilizaremos então a seguinte sintaxe:

```
@OneToOne(cascade = CascadeType.PERSIST).

      @Entity
public class Presidente implements Serializable {


    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private Long
    id;

    private String nome;

    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn(name = "PAIS_FK")
    private Pais pais;

    // Getters and Setters
```

Tabela 04: Persistência por cascata

O *cascade* é uma propriedade que define como o JPA deve-se comportar quanto à entidade relacionada ao executar uma transação no banco de dados. Nota-se que utilizarmos o *Enum CascadeType*, ele pertence ao pacote *javax.persistence*, e detêm os tipos possíveis do cascade.

8. Carregamento LAZY e EAGER

Definir nossa estratégia de busca por entidades, desde o início, nos ajudará ao longo de todo o ciclo de desenvolvimento, não é uma “otimização prematura”, é apenas uma parte natural de qualquer desenho ORM.

Assim, segundo disposto na “*Enterprise JavaBeans*” , os *EntityManager* administram todas as instâncias das entidades, por isso ela também é responsável pelo carregamento do estado dos objetos. Dessa forma, há dois modos de carregar um objeto com os dados obtidos de um banco de dados, são eles o *LAZY* e *EAGER*. No modo *LAZY*, o carregamento é tardio e isso faz com que seu carregamento seja adiado ao máximo na busca dos dados no banco de dados. Já no modo *EAGER*, o carregamento é imediato.

Por padrão, os relacionamentos com as anotações @*ManyToOne* e @*OneToOne*, são buscados ansiosamente (terminadas em “One”), enquanto os relacionamentos @*OneToMany* e @*ManyToMany* (terminados em Many). Esta é a estratégia padrão e o JPA não otimiza magicamente a recuperação de objetos, ela só faz o que é instruída a fazer (APÊNDICE C).

Assim, podemos observar que temos uma Entidade Setor que se relaciona com a entidade *TecnologiaInformacao*, este é um relacionamento *Many to One*, na primeira coluna o código tem como estratégia o carregamento *EAGER* e na segunda o *LAZY*, vamos inserir um objeto para cada estratégia e fazer uma busca na base e analisar o sql gerado.

Hibernate: select setor0_.id as id1_2_1_, setor0_.TI_FK as TI2_2_1, tecnologia1_.id as id1_11_0, tecnologia1_.sala as sala2_11_0_ from Setor setor0_ left outer join T_TEC_INFO tecnologia1_ on setor0_.TI_FK=tecnologia1_.id where setor0_.id=?	Hibernate: select setor0_.id as id1_2_0_, setor0_.TI_FK as TI2_2_0_ from Se Setor setor0_ where setor0_.id=?

Tabela 05: estratégia com o carregamento EAGER e o LAZY

O resultado gerado refere-se respectivamente às estratégias da imagem anterior. No primeiro caso um *left* outer*join*, foi usado para carregar os dados da tabela *T_TEC_INFO*, isso já é o sintona do *EAGER*, em outras palavras, toda vez que uma busca for realizada *SETOR* irá carregar dados desta tabela.

Já na do outro lado, usamos o *LAZY* por isso não JOINs na tabela *T_TEC_INFO*, este é o carregamento tardio.

Agora vamos imaginar que tenhamos em nossa base de dados muitos setores e nossa estratégia de carregamento escolhida inicialmente seja a *EAGER*, se uma busca por todos os setores for realizada, já sabemos que os dados da entidade *TecnologiaInformacao* será carregada juntamente com Setor. Observamos o seguinte, se *TecnologiaInformacao* estiver relacionada com outras entidades e se estes relacionamentos também estiverem com estratégia *EAGER* fará com que o JPA gere muitos JOINs o que irá causar lentidão no sistema.

A escolha da estratégia de carregamento automática com *LAZY* e *EAGER* será de suma importância para o desempenho de sua aplicação. Sendo que, a má escolha poderá afetar toda a funcionalidade relacionada à estratégia em

questão, por isso o melhor conselho é favorecer a estratégia de carregamento manual através de JOINs na *query*.

9. Herança

O conceito de herança é muito comum em sistemas OO, mas requer muito cuidado em seu uso, o alto acoplamento pode se tornar um problema e o processo de manutenção do sistema custoso. Mas, a herança tem também têm vantagens e seu uso faz com que o código sofra reuso e corte de redundância.

Na JPA podemos definir quatro estratégias para o mapeamento com herança, representadas pelas anotações, sendo que cada uma possui vantagens e desvantagens, são elas: *mapped superclass*, *single table*, *joined* e *table per concrete class*.

- ***Mapped superclass:*** Nessa estratégia de herança, a classe pai não é uma entidade logo não pode ser persistida ou consultada. Por boa prática devemos defini-la como com *abstract*. Utilizamos a anotação *@MappedSuperclass* para definir a estratégia. (APÊNDICE D)

- ***Single Table***: neste caso a classe a ser herdada é uma entidade, nesta estratégia os dados são mantidos em uma única estrutura de tabela no banco de dados, o que torna mais fácil de ser analisada. Em geral tem um bom desempenho em consultas, sua desvantagem pode estar na utilização de campos *not null* que pode ocasionar erro ao persistir uma entidade sem valor para o campo em questão.

	id [PK] bigint	sala character varying(255)
1	1	Sala 001
*		

Figura 19: Estratégia Single Table

Utilizamos as anotações @*Inheritance* e @*DiscriminatorColumn*. O tipo de herança *InheritanceType.SINGLE_TABLE* é escolhida na propriedade *strateg (*caso exista uma relação de herança em uma *entity* e a anotação @*Inheritance* não esteja presente), a JPA adotará a estratégia *SINGLE_TABLE* como padrão. A anotação @*DiscriminatorColumn* informa qual o nome da coluna que armazenará a entity "dona" de uma determinada linha no banco de dados.

	class_name character varying(31)	id [PK] bigint	nome character var	cnpj character varying(255)	cpf character varying(255
1	PessoaFisica	1	Filipe Bast		111.111.111-11
2	PessoaJuridica	2	Open TI LTD	22.222.222/2222-22	
*					

Figura 20: Legenda

- ***Joined:*** os dados são armazenados em tabelas separadas para cada entidade e irá seguir o conceito OO, aplicado ao modelo de dados. Mas, poderá ter um *insert* mais custoso uma vez que, para cada entidade no mapeamento teremos um *inser e,* caso exista uma herança C extends B extends A, ao persistir um C, três *inserts* serão realizados um para cada entidade mapeada.

Quanto maior for à hierarquia, maior será o número de JOINs realizado em cada consulta.

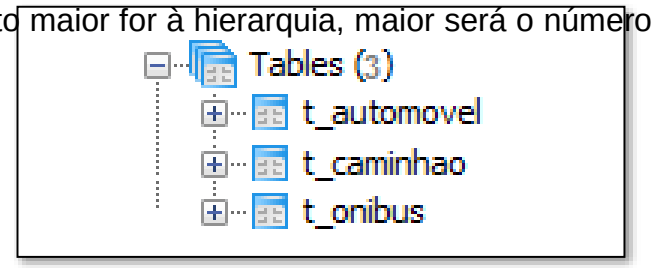


Figura 21: Árvore de tabelas da estratégica *Joined*

- ***Table****:* é a estratégia *per concrete class*, em uma tabela constam apenas os dados relativos à entidade. Quando a consulta é realizada em apenas em uma entidade, seu retorno é mais rápido justamente por ser específica. O problema neste caso é a repetição de colunas, ao realizar uma consulta, pois é

necessário trazer mais de uma entidade da herança, essa pesquisa terá um custo maior, já que será utilizado UNION ou uma consulta por tabela.

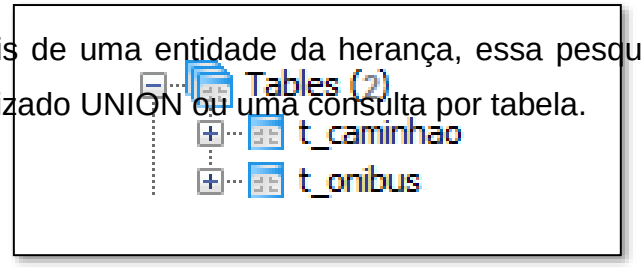


Figura 22: Árvore de tabelas da estratégica *table per concrete*

10. @Embeddable e @Embedded

Ainda de acordo com o Oracle, essas anotações configuram um relacionamento por composição, seu uso ajuda a melhor organizar as entidades e promove uma oportunidade de reaproveitamento de código.

Há casos em que nos deparamos com uma tabela com várias colunas e ao trazer esses dados ao modelo orientado a objetos podemos quebrar esses dados em objetos menores e deixar as informações mais específicas.

Um bom exemplo para este relacionamento seria uma tabela sobre uma pessoa que contém dados de endereço, documentação, filiação, e muitos outros detalhes, vejamos então como podemos utilizar essas anotações.

```java
@Embeddable
public class Endereco
    implements Serializable {

    @Basic(optional = false)
    private String
    logradouro;

    @Basic(optional =
    false) private Integer
    numero;

    @Basic(optional = false)
    private String
    complemento;

    @Basic(optional =
    false) private String
    bairro;

    @Basic(optional = false)
private String cep

    @Basic(optional = false) priv


    /** Getters and Setters */
}
```

```java
@Entity
public class Pessoa implements
    Serializable {

    @Id @GeneratedValue
    private Long id;

    @Basic(optional = false)
    private String nome;

    @Embedded
    private Endereco endereco;



    /** Getters and Setters */
```

Tabela 06: Exemplo de @Embeddable e @Embedded

11. Consultas com JPQL

A JPQL (*Java Persistence Query Language*) é a linguagem de consulta padrão da JPA, que permite escrever consultas portáveis, que funcionam independentemente do banco de dados. Esta linguagem de query usa uma sintaxe parecida com a SQL, para selecionar objetos, valores de entidades e os relacionamentos entre elas. É importante ressaltarmos que JPQL é case sensitive e suas queries **não** serão realizadas em tabelas e sim em entidades.

```
@Entity

@Table(name = "T_CACHORRO") public class
Cachorro implements
Serializable {


private static final long serialVersionUID = 1L;

@Id @GeneratedValue
private Integer id;


private String nome;


public Cachorro() {


}


public Cachorro(String nome) { this.nome =
nome;
}

public static void main(String[] args) {

EntityManagerFactory
emf=Persistence.createEntity
ManagerFactory("conexaoPU"); EntityManager
em = null;

em = emf.createEntityManager();
EntityTransaction tx =
em.getTransaction(); tx.begin();
```

```
public static void main(String[] args) {


    EntityManagerFactory
        emf=Persistence.createEntity
    ManagerFactory("conexaoPU");
    EntityManager em = null;

    em = emf.createEntityManager();
    EntityTransaction tx =
        em.getTransaction();
    tx.begin();
    Cachorro c1 = new Cachorro("Jon");
    em.persist(c1);
    tx.commit();


em = emf.createEntityManager

tx.commit();


}

em = emf.createEntityManager(); tx =
em.getTransaction(); tx.begin();
Cachorro c3 = new Cachorro("Arya");
em.persist(c3);
tx.commit();
```

Cachorro c1 = new Cachorro("Jon");
em.persist(c1);
tx.commit();
 	
em = emf.createEntityManager(); tx = em.getTransaction(); tx.begin();
Cachorro c2 = new Cachorro("Sansa");
em.persist(c2);
tx.commit();
 	
/** Getters e Setters
	Query q1 = em.createQuery("select c from Cachorro c");
	List<Cachorro> list1 = q1.getResultList();
	
	System.out.println("Tamanho da lista: " + list1.size());
	
	em.close(); emf.close();
	}

Tabela 07 : Exemplo de Entity e main

Do lado esquerdo temos a *Entity* Cachorro e do lado direito uma classe com método *main* que demonstra a inclusão de três cachorros. Notamos logo abaixo em nosso código, um exemplo de JPQL, a partir da *EntityManager* utilizamos o método *createQuery* que espera uma *string* que será nossa *query* como parâmetro, o resultado será uma *Query* (poderíamos usa a classe *TypedQuery* ao invés da Query, ambas são do pacote *javax.persistence* do JPA), a partir desse ponto conseguiremos obter o resultado da consulta.

Na escrita da JPQL, o uso do *select*, assim como do alias da entidade, torna-se facultativo, poderíamos então escrever nossa *query* desta forma “from Cachorro”. Podemos usar outros métodos provenientes da *EntityManager* para escrever nossas consultas, até mesmo usar SQL nativo.

Podemos filtrar nossos dados utilizando passagem de parâmetros pela JPQL, para isso mostraremos duas formar de desenvolver, uma pelo índice do atributo e outra pelo nome do atributo.

```
public class Principal9 {


public static void main(String[] args) {


EntityManagerFactory emf = Persistence.createEntityManagerFactory("conexaoPU");
EntityManager em = null;


em = emf.createEntityManager(); EntityTransaction tx = em.getTransaction(); tx.begin();
Cachorro c1 = new Cachorro("Jon"); em.persist(c1);
tx.commit();


em = emf.createEntityManager(); tx = em.getTransaction(); tx.begin();
Cachorro c2 = new Cachorro("Sansa"); em.persist(c2);
tx.commit();


em = emf.createEntityManager(); tx = em.getTransaction(); tx.begin();
Cachorro c3 = new Cachorro("Arya");



em.persist(c3); tx.commit();

Query q1 = em.createQuery("select c from Cachorro c where c.nome = :nome");
q1.setParameter("nome", "Jon");
Cachorro cc1 = (Cachorro) q1.getSingleResult(); System.out.println("O nome do cachorro : " +
cc1.getNome());

// ou


Query q2 = em.createQuery("select c from Cachorro c where c.nome = ?1"); q2.setParameter(1,
"Arya");
Cachorro cc2 = (Cachorro) q2.getSingleResult(); System.out.println("O nome do cachorro : " +
cc2.getNome());

em.close(); emf.close();
}


}
```

Tabela 08: parâmetros pela JPQL

Dessa forma, o JPA ficará responsável por traduzir cada JPQL para a sintaxe correta e o desenvolvedor não precisará tomar conhecimento de qual a sintaxe requerida para cada banco.

12. Conclusão

Uma ferramenta ORM, como é o caso do Hibernate tornou a união do mundo orientado a objetos e o mundo relacional mais viável e levou o desenvolvedor a se preocupar mais com as regras de negócio ao invés de escritas SQL, inserção por cascatas e etc. A normalização das propriedades do Hibernate, através da JPA abriu um vasto caminho para outros *frameworks* fossem sendo desenvolvidos e que deixou ainda melhor a união destes dois paradigmas.

Importante entender que é preciso estar atento ao funcionamento das funções da JPA para utilizá-la da maneira correta. O Mau uso da especificação leva a diversos problemas, o que leva algumas pessoas a criticarem a ferramenta. Temos que ter em mente que JPA veio para auxiliar uma parte do desenvolvimento que perdeu o seu foco quando se deparou com dois mundos diferentes, ela não será a salvação de tudo e sim uma excelente especificação a ser seguida e bem praticada, com isso certamente a produtividade do desenvolvimento e manutenibilidade aumentarão.

Referências

BAUER, C.; KING, G. Java Persistence with Hibernate. [S.l.]: Manning Publications Company, 2015. ISBN 1932394885.

BEGINNING with JPA 2.0 <https://coderevisited.com/beginning-jpa-2-0/>. Acessado em: 17-06-2016.

CAELUM – Apostilas – Java Desenvolvimento Web - Mapeando uma classe Tarefa para nosso Banco de Dados. <https://www. caelum.com.br/apostila-java-web/uma-introducao-pratica-ao-jpa-com-hibernate/ #14-4-mapeando-uma-classe-tarefa-para-nosso-banco-de-dados. Acessado em: 15-06-2016.

COCIAN, LUIS FERNANDO ESPINOSA - Manual Da Linguagem C - 1. Ed. ULBRA, 2004.

COELLHO, H. JPA eficaz: as melhores práticas de persistência de dados em java. 1. Ed. [S.l.]: Casa do Código, 2013.

CORDEIRO, G. Aplicações java para web com JSF e JPA. 1. Ed. [S.l.]: Casa do Código, 2012.

HIBERNATE. <http://hibernate.org/orm/>. Acessado em: 01-06-2016.

INTRODUÇÃO AO ENTITYMANAGER - http://www.devmedia.com.br/introducao-ao-entitymanager/5206 - Acesso em 2/05/2017.

JPA Mini Livro – Primeiros passos e conceitos detalhados. <http://uaihebert.com/jpa-mini-livro-primeiros-passos-e-conceitos-detalhados/>. Acessado em: 05-06-2016.

JPA REFERENCE (JavaDoc). <http://www.objectdb.com/api/java/jpa>. Accessado: 2016-05-20.

KEITH, M.; SCHINCARIOL, M.; KEITH, J. Pro JPA 2: Mastering the JavaTM Persistence API. [S.l.]: Apress, 2011. (Books for professionals by professionals). ISBN 9781430219576.

ORACLE.<http://www.oracle.com/technetwork/java/javase/jdbc/index.html>. Acessado em: 01-06-2016

REVISTA EASY. NET MAGAZINE 28 < http://www.devmedia.com.br/orm-object-relational-mapping-revista-easy-net-magazine-28/27158- > Acesso em: 02-05-2017

APÊNDICES

APÊNDICES A – Uso da chave composta utilizando a anotação @idclass ou @embeddedid

```
@Entity
@Table(name = "t_carro")
public class Carro implements Serializable {

   private static final long serialVersionUID = 1L;

   @EmbeddedId
   private CarroPK carroPK;

   @Column
   private String nome;

   public Carro() {

   }

   public Carro(String
      nome) { this.nome =
      nome;
   }

   public CarroPK
      getCarroPK() { return
      carroPK;
   }

   public void setCarroPK(CarroPK
      carroPK) { this.carroPK = carroPK;
   }
```

```
    public String
        getNome() { return
        nome;
    }

    public void setNome(String
        nome) { this.nome = nome;
    }

}

 @Embeddable
public class CarroPK implements Serializable {

    private static final long serialVersionUID = 1L;

    @Column
    private Integer ano;

    @Column
    private String chassi;

    public Integer
        getAno() { return
        ano;
    }

    public void setAno(Integer
        ano) { this.ano = ano;
    }

    public String
        getChassi() { return
        chassi;
    }

    public void setChassi(String
        chassi) { this.chassi = chassi;
    }
```

```
    @Override
    public int hashCode() {
      return this.ano.hashCode() +
    this.chassi.hashCode(); }

    @Override
    public boolean equals(Object obj) {
      if
        (!CarroPK.class.isInstance(ob
        j)) { return false;
      }
      if
        (!CarroPK.class.cast(obj).ano.equals(this.an
        o)) { return false;
      }
      if
        (!CarroPK.class.cast(obj).chassi.equals(this.chas
        si)) { return false;
      }
      return
    true; }
}
```

APENDICE B – Utilização de chave composta usando o id de uma classe composto

```
@Entity
@Table(name = "T_CACHORRO")
public class Cachorro implements Serializable {

   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedV
   alue
   private Integer id;
   private String nome;

   public Cachorro() {

   }

   public Cachorro(String
      nome) { this.nome =
      nome;
   }


   /** Getters e Setters */

   public Integer
      getId() { return
      id;
   }

   public void setId(Integer
      id) { this.id = id;
   }
   public String
      getNome() { return
      nome;
```

```
    }

    public void setNome(String
        nome) { this.nome = nome;
    }

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

    @Override
    public boolean equals(Object obj) {
        if
            (!Cachorro.class.isInstance(ob
            j)) { return false;

        return
    Cachorro.class.cast(obj).id.equals(this.id); }
}


@Entity
@Table(name = "T_DONO")
public class Dono implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedV
    alue
    private Integer id;

    private String nome;
```

```
    public Integer
      getId() { return
      id;
    }

    public void setId(Integer
      id) { this.id = id;
    }

    public String
      getNome() { return
      nome;
    }

    public void setNome(String
      nome) { this.nome = nome;
    }

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


    @Override
    public boolean equals(Object
      obj) { if
      (!Dono.class.isInstance(obj))
      {
        return
      false; }

      return
    Dono.class.cast(obj).id.equals(this.id); }
}
```

```
@Entity
@IdClass(CasaCachorroPk.cla
ss) @Table(name =
"T_CASA_CACHORRO")
public class CasaCachorro implements Serializable {

   private static final long serialVersionUID = 1L;

   @Id
   @OneTo
   One
   @JoinColumn(name =
   "CACHORRO_FK") private
   Cachorro cachorro;

   @Id
   @OneTo
   One
   @JoinColumn(name =
   "DONO_FK") private Dono
   dono;

   public Cachorro
      getCachorro() { return
      cachorro;
   }

   public void setCachorro(Cachorro
      cachorro) { this.cachorro = cachorro;
   }

   public Dono
      getDono() {
      return dono;
   }

   public void setDono(Dono dono) {
```

```
        this.dono =
    dono; }

}

public class CasaCachorroPk implements Serializable {

    private static final long serialVersionUID = 1L;

    private Cachorro cachorro;

    private Dono dono;

    public Cachorro
        getCachorro() { return
        cachorro;
    }

    public void setCachorro(Cachorro
        cachorro) { this.cachorro = cachorro;
    }

    public Dono
        getDono() {
        return dono;
    }

    public void setDono(Dono
        dono) { this.dono = dono;
    }

    @Override
    public int hashCode() {
        return this.cachorro.hashCode() +
    this.dono.hashCode(); }

    @Override
    public boolean equals(Object obj) {
```

```
    if
        (!CasaCachorroPk.class.isInstance(o
        bj)) { return false;
    }

                                    return
    CasaCachorroPk.class.cast(obj).cachorro.equals(this.cachorro)
        && CasaCachorroPk.class.cast(obj).dono.equals(this.dono);
}
```

APENDICE – C - Carregamento LAZY e EAGER

```
@Entity
public class Setor
    implements Serializable
    {

   private static final
      long
      serialVersionUID =
      8913002777036707562
      L;

   @Id
   @GeneratedValue
   private Long
   id;

   @ManyToOne // O mesmo
   que
      @ManyToOne(fetch=FetchType.EAGER) @JoinColumn(name =
   "TI_FK")
   private TecnologiaInformacao ti;

   public Long getId() { return id;
   }

   public void setId(Long id) { this.id = id;
   }

   public TecnologiaInformacao getTi() {
     return ti; }

   public void setTi(TecnologiaInformacao ti) {
```

```
@Entity
public class Setor
    implements
    Serializable {

   private static
      final long
      serialVersionUI
      D =
      891300277703670
      7562L;

      @Id
      @Generated
      Value
      private
      Long id;

      @ManyToOne(fetch =
      FetchType.LAZY)
```

```
          this.ti = ti; }
                                              }
      }
@JoinColumn(name = "TI_FK") private TecnologiaInformacao ti;

public Long getId() { return id;
}

public void setId(Long id) { this.id = id;
}

public TecnologiaInformacao getTi() {
  return ti; }

public void setTi(TecnologiaInformacao ti) {
  this.ti = ti;
}
```

APENDICE D- Mapped superclass

```
@MappedSuperclass
public abstract class Departamento implements Serializable {

   private static final long serialVersionUID = 1L;

   private String sala;

   public String
      getSala() { return
      sala;
   }

   public void setSala(String
      sala) { this.sala = sala;
   }

}
@Entity
@Table(name = "T_TEC_INFO")
public class TecnologiaInformacao extends Departamento implements
    Serializable {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedVa
   lue private
   Long id;


   public Long
      getId() {
      return id;
   }

   public void setId(Long
      id) { this.id = id;
   }

}
```

APENDICE – E - Single Table

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

   @Id
   @GeneratedVa
   lue private
   Long id;

   private String nome;

   public Long
      getId() {
      return id;
   }

   public void setId(Long
      id) { this.id = id;
   }


   public String
      getNome() { return
      nome;
   }

   public void setNome(String
      nome) { this.nome = nome;
   }
}

@Entity
@DiscriminatorValue("PessoaFi
sica")
public class PessoaFisica extends Pessoa implements Serializable {
```

```
  private static final long serialVersionUID = 1L;

  private String cpf;

  public String
    getCpf() { return
    cpf;
  }

  public void setCpf(String
    cpf) { this.cpf = cpf;
  }
}

@Entity
@DiscriminatorValue("PessoaJuri
dica")
public class PessoaJuridica extends Pessoa implements Serializable {

  private static final long serialVersionUID = 1L;

  private String cnpj;

  public String
    getCnpj() { return
    cnpj;
  }

  public void setCnpj(String
    cnpj) { this.cnpj = cnpj;
  }
```

APÊNDICE F- Joined

```
@Entity
@Table(name = "T_AUTOMOVEL")
@Inheritance(strategy =
InheritanceType.JOINED)
public abstract class Automovel implements Serializable {

   private static final long serialVersionUID = 9137047909661179776L;

   @Id
   private Long id;

   private String chassi;

   public String
      getChassi() { return
      chassi;
   }

   public void setChassi(String
      chassi) { this.chassi = chassi;
 } }

@Entity
@Table(name = "T_CAMINHAO")
public class Caminhao extends Automovel implements Serializable {

   private static final long serialVersionUID = 5154289656498987240L;

   // Outros
atributos }

@Entity
@Table(name = "T_ONIBUS")
public class Onibus extends Automovel implements Serializable {
```

```
    private static final long serialVersionUID = 5154289656498987240L;

    // Outros
atributos }
```

APÊNDICE G - Table

```
@Entity
@Table(name = "T_AUTOMOVEL")
@Inheritance(strategy = InheritanceType.TABLE_PER
CLASS) public abstract class Automovel implements
Serializable {

   private static final long serialVersionUID = 9137047909661179776L;

   @Id
   private Long id;

   private String chassi;

   public String
      getChassi() { return
      chassi;
   }

   public void setChassi(String
      chassi) { this.chassi = chassi;
   }
}


@Entity
@Table(name = "T_CAMINHAO")
public class Caminhao extends Automovel implements Serializable {

   private static final long serialVersionUID = 5154289656498987240L;

   // Outros
atributos }

@Entity
@Table(name = "T_ONIBUS")
public class Onibus extends Automovel implements Serializable {
```

```
    private static final long serialVersionUID = 5154289656498987240L;

    // Outros
atributos }
```