JPA

Introdução

- Qualquer aplicativo corporativo executa operações de banco de dados.
- Os desenvolvedores de aplicativos normalmente lutam para executar operações de banco de dados de forma eficiente.
- Usando ORM, a carga de interagir com o banco de dados reduz significativamente.
- Forma uma ponte entre modelos de objeto e modelos relacionais .

Incompatibilidade Objeto – Tabela Relacional

- Granularidade: Modelo de objeto tem mais granularidade do que o modelo relacional.
- Herança: não são suportados por todos os tipos de bancos de dados relacionais.
- Identidade: Modelo relacional não prevê identidade ao escrever igualdade.

Incompatibilidade Objeto – Tabela Relacional

- Associações: modelos relacionais não podem determinar vários relacionamentos ao olhar para um modelo de objeto.
- Navegação de dados: dados de navegação entre os objetos em uma rede de objeto é diferente em ambos os modelos.

JPA

- Java Persistence API é a especificação de uma coleção de classes e métodos para armazenar persistentemente as vastas quantidades de dados em um banco de dados.
- As versões anteriores do EJB, a camada de persistência era definida combinada com a camada de lógica de negócios usando javax.ejb.EntityBean interface.
- Na introdução do EJB 3.0, a camada de persistência foi separada e especificada como JPA 1.0 (Java Persistence API).

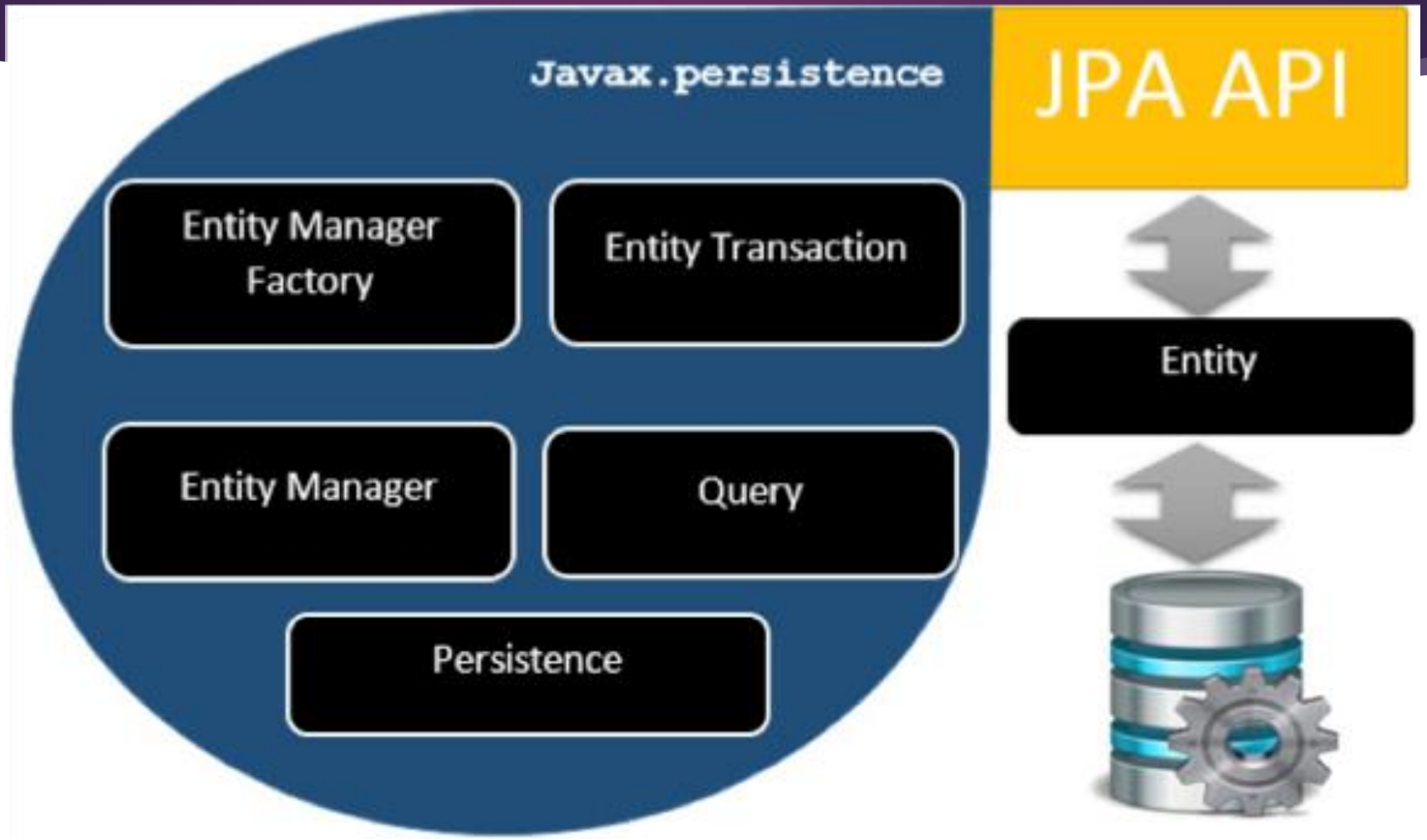
JPA

- As especificações desta API foram liberadas junto com as especificações do JAVA EE5 em 11 de maio de 2006 usando a JSR 220.
- A JPA 2.0 foi lançada com as especificações do Java EE6 em 10 de dezembro de 2009, como parte do Java Community Process JSR 317.
- A JPA 2.1 foi lançada com a especificação do JAVA EE7 em 22 de abril de 2013 usando JSR 338.

JPA

- Para utilizar a JPA, é necessário um provedor.
- Este provedor é uma implementação da especificação JPA.
- Alguns provedores (frameworks) existentes no mercado:
- Hibernate, EclipseLink, Toplink, OpenJPA, Spring Data

Arquitetura



Arquitetura

- Entity:
  - Uma entidade é um objeto de domínio de persistência leve.
  - Normalmente, uma entidade representa uma tabela em um banco de dados relacional e cada instância de entidade corresponde a uma linha nessa tabela.

Arquitetura

- O principal artefato de programação de uma entidade é a classe de entidade, embora as entidades possam usar classes auxiliares.
- O estado persistente de uma entidade é representado por campos persistentes ou propriedades persistentes. Esses campos ou propriedades usam anotações de mapeamento objeto-relacional para mapear as entidades e as relações entre entidades para os dados relacionais.

Arquitetura

- EntityManager:
- As entidades são gerenciadas pelo EntityManager, que é representado pelas instâncias javax.persistence.EntityManager.
- Cada instância do EntityManager está associada a um contexto de persistência: um conjunto de instâncias de entidade gerenciada que existem em um armazenamento de dados específico.

Arquitetura

- Um contexto de persistência define o escopo sob o qual determinadas instâncias de entidade são criadas, persistentes e removidas.
- A interface EntityManager define os métodos que são usados para interagir com o contexto de persistência.

Arquitetura

- EntityManagerFactory:
  - Esta é uma classe de fábrica de EntityManager. Ele cria e gerencia várias instâncias de EntityManager.
- Persistence:
  - Esta classe contém métodos estáticos para obter instâncias de EntityManagerFactory em uma estrutura independente de frameworks.

Arquitetura

- Query :
  - A interface javax.persistence.Query é implementada por cada framework para encontrar objetos persistentes que estão de acordo com determinado critério. A JPA padronizou o suporte a consultas usando tanto a Java Persistence Query Language (JPQL) e a Structured Query Language (SQL). As instâncias de Query são obtidas de um EntityManager.

Arquitetura

- EntityTransaction:
  - Cada EntityManager possui um relacionamento um-para-um com uma única javax.persistence.EntityTransaction. EntityTransactions permitem operações em dados persistidos serem agrupados em unidades de trabalho, que ou são completamente concluídas com sucesso ou falham completamente, deixando o banco de dados no seu estado original. Estas operações tudo ou nada são importantes para manter a integridade dos dados.

ORM

- Um objeto é armazenado no banco de dados relacional em três fases.
- A primeira fase, denominada fase de dados do objeto, contém classes POJO, interfaces de serviço e classes. É a principal camada de componentes de negócios, que possui operações de lógica de negócios e atributos. Por exemplo, um banco de dados de empregados:

ORM

- A classe POJO do empregado contém atributos como ID, nome, salário e designação. E métodos setter e getter desses atributos.
- As classes DAO / Service do empregado contêm métodos de serviço como criar empregado, encontrar empregado e excluir empregado.

ORM

- A segunda fase é chamada de mapeamento ou fase de persistência que contém o provedor JPA, o arquivo de mapeamento (ORM.xml), o JPA Loader e o Object Grid.
- Provedor JPA: o produto do fornecedor que implementa o JPA (javax.persistence). Por exemplo, Eclipselink, Toplink, Hibernate, etc.
- Arquivo de mapeamento: O arquivo de mapeamento (ORM.xml) contém configuração de mapeamento entre os dados em uma classe POJO e dados em um banco de dados relacional.

ORM

- Object Grid: A Object Grid é um local temporário que pode armazenar a cópia de dados relacionais, ou seja, como uma memória cache. Todas as consultas contra o banco de dados é primeiro efetuada nos dados na Object Grid. Somente depois que ele é confirmado, ele efetua o banco de dados principal.
- JPA Loader: O JPA Loader carrega os dados da Object Grid. Ele funciona como uma cópia do banco de dados para interagir com classes de serviço para dados POJO (Atributos da classe POJO).

ORM

- A terceira fase é a fase Relacional de dados. Ele contém os dados relacionais que estão logicamente conectados ao componente de negócios. Somente quando o componente de negócios confirma os dados, ele é armazenado no banco de dados fisicamente. Até então, os dados modificados são armazenados em uma memória cache como um formato de Grid. O mesmo é o processo para obter dados.
- O mecanismo da interação programática das três fases é chamado como mapeamento relacional de objetos.

Mapping.xml

- O arquivo mapping.xml deve instruir o fornecedor JPA para mapear as classes Entity com tabelas de banco de dados.
- Ex: entidade empregado que contém quatro atributos.
- A classe POJO da entidade Employee denominada Employee.java é a seguinte:

Mapping.xml

```
public class Employee {

    private int eid;
    private String ename;
    private double salary;
    private String deg;

    public Employee(int eid, String ename, double salary, String deg) {
        this.eid = eid;
        this.ename = ename;
        this.salary = salary;
        this.deg = deg;
    }
}
```

Mapping.xml

<? xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns=“http://java.sun.com/xml/ns/persistence/orm" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation= “http://java.sun.com/xml/ns/persistence/ormhttp://java.sun.com/xml/ns/persistence/orm_1_0.xsd version="1.0">

<description>XML Mapping file </description><entity class="Employee">
<table name="EMPLOYEETABLE"/>

<attributes>

<id name="eid"> <generated-value strategy="TABLE"/></id>

<basic name="ename"><column name="EMP_NAME" length="100"/>

</basic>

<basic name="salary"></basic>

<basic name="deg"></basic>

</attributes></entity></entity-mappings>

Mapping.xml

- O script acima mapeia a classe de entidade com a tabela de banco de dados. Neste arquivo:
- <Entity-mappings>: tag define a definição do esquema para permitir tags de entidade no arquivo xml.
- <Description>: tag define a descrição sobre a aplicação.
- <Entity>: tag define a classe de entidade que você deseja converter em tabela em um banco de dados. A classe de atributo define o nome da classe de entidade POJO.

Mapping.xml

- <Table>: tag define o nome da tabela. Se você quiser manter o nome da classe como nome da tabela, então essa tag não é necessária.
- <Attributes>: tag define os atributos (campos em uma tabela).
- <Id>: tag define a chave primária da tabela. A tag <generated-value> define como atribuir o valor da chave primária.
- <Basic>: tag é usada para definir os atributos restantes para a tabela.
- <Column-name>: tag é usada para definir o nome do campo da tabela definida pelo usuário.

Annotations

- Geralmente, os arquivos xml são usados para configurar componentes específicos ou mapear duas especificações diferentes de componentes.
- Em nosso caso, temos que manter o xml separadamente em um framework.
- Isso significa que ao escrever um arquivo xml de mapeamento precisamos comparar os atributos de classe POJO com tags de entidade no arquivo mapping.xml.

Annotations

- Solução: Na definição de classe, podemos escrever a parte de configuração usando anotações.
- As anotações são usadas para classes, atributos e métodos.
- As anotações começam com o símbolo '@‘ e são declaradas antes da classe, atributo ou método. Todas as anotações do JPA são definidas no pacote javax.persistence.

Annotations

- @Entity declarar a classe como entidade ou uma tabela.
- @Table declarar o nome da tabela.
- @Basic especifica atributos.
- @Embedded Esta anotação especifica um atributo que é uma instância de uma classe @Embeddable

Annotations

- @Id chave primária.
- @GeneratedValue Esta anotação especifica como o a chave primária pode ser inicializada: Automático, manual ou valor tirado da tabela de seqüência.
- @Transient Essa anotação especifica o atributo que não é persistente, ou seja, o valor nunca é armazenado no banco de dados.

Annotations

- @Column Esta anotação é usada para especificar um nome de coluna.
- @SequenceGenerator Esta anotação é usada para definir o valor da propriedade que é especificada na anotação @GeneratedValue. Ele cria uma seqüência.
- @TableGenerator Esta anotação é usada para especificar o gerador de valor para a propriedade especificada na anotação @GeneratedValue. Ele cria uma tabela para geração de valor.

Annotations

- @AccessType Este tipo de anotação é usado para definir o tipo de acesso. Se você definir @AccessType (FIELD), será acessado o conteúdo do atributo em uma consulta. Se você definir @AccessType (PROPERTY), então o atributo não será acessado em um get, e sim o método get marcado com essa anotação, podendo ter valores fixos.
- @JoinColumn Isso é usado em associações de muitos para um e um para muitos.

Annotations

- @UniqueConstraint Especificar que o valor de um atributo é único.
- @ColumnResult Esta anotação faz referência ao nome de uma coluna em uma consulta consulta SQL usando a cláusula select.
- @ManyToMany Definir uma relação muitos-para-muitos entre as tabelas join.
- @ManyToOne Definir uma relação muitos-para-um entre as tabelas join.

Annotations

- @OneToMany Definir uma relação um-para-muitos entre as tabelas join.
- @OneToOne Definir uma relação um-para-um entre as tabelas join.
- @NamedQueries Esta anotação é usada para especificar uma lista de consultas nomeadas.
- @NamedQuery Essa anotação é usada para especificar uma consulta usando o nome estático.

Padrões de Java Bean

- Uma classe java encapsula os valores de atributos e métodos em uma única unidade chamada objeto. Java Bean é um armazenamento temporário e um componente reutilizável ou um objeto.
- É uma classe serializável que tem o construtor padrão e os métodos getter e setter para inicializar os atributos de instância individualmente.

Padrões de Java Bean

- Um bean contém o construtor padrão ou um arquivo que contém uma instância serializada. Portanto, um bean pode instanciar um bean.
- As propriedades de um bean podem ser segregadas em propriedades Booleanas e propriedades não Booleanas.
- A propriedade não-booleana contém métodos getter e setter.
- Propriedade booleana contém setter e o método is.

Padrões de Beans

- O método getter de qualquer propriedade deve começar com letras pequenas 'get' (convenção de método java) e continua com um nome de campo que começa com letra maiúscula. Por exemplo. O nome do campo é 'salary', portanto, o método getter deste campo é 'getSalary ()'.

Padrões de Beans

- O método setter de qualquer propriedade deve começar com um pequeno conjunto de letras (convenção do método java), continuado com um nome de campo que começa com letra maiúscula e o valor do argumento para definir como campo. Por exemplo. O nome do campo é 'salary', portanto, o método setter deste campo é 'setSalary (double sal)'.
- Para propriedade Booleana, o método para verificar se ele é verdadeiro ou falso é is. Por exemplo, para a propriedade Booleana ‘Empty', o método is deste campo é 'isEmpty ()'.

Persistence.xml

- Documento utilizado para configurar o acesso ao provedor JPA
- Nele vão dados como:
  - Configuração do provedor JPA
  - Acesso ao Sistema Gerenciador de Banco de Dados
  - Classes mapeadas para o SGBD

Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?><persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

<persistence-unit name="teste" transaction-type="RESOURCE_LOCAL">

<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

<class>Pessoa</class>
```

Persistence.xml

```
<properties>
<property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>

<property name="javax.persistence.jdbc.user" value="root"/>

<property name="javax.persistence.jdbc.password" value="ifsul2017"/>

<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/4k ou i"/>
```

Persistence.xml

```
<property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5InnoDBDialect" />
<property name="hibernate.show_sql" value=“true" />
<property name="hibernate.format_sql" value="true" />
<property name="hibernate.use_sql_comments" value="false"/>
<property name="hibernate.hbm2ddl.auto" value="update" />
</properties></persistence-unit></persistence>
```

Descrição dos elementos xml

- <?xml version="1.0" encoding="UTF-8"?><persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"></persistence>

- Cabeçalho e definição da versão do JPA

Descrição dos elementos xml

- <persistence-unit name="teste" transaction-type="RESOURCE_LOCAL"></persistence-unit>

- Define o nome da unidade de persistência (neste caso o nome é teste) e o tipo de transação que será realizada:
  - RESOURSE_LOCAL: Utilizado para JAVASE, ou seja, o programa não rodará em um servidor de aplicação JAVAEE.
  - JTA: Utilzado para JAVAEE, através do Java Transaction API (JTA).

Descrição dos elementos xml

- <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
  - Define o provedor JPA, neste caso o hibernate.
- <class>Pessoa</class>
  - Define o nome da classe que será persistida usando o mapeamento objeto-relacional.
- <properties></properties>
  - Marcação do conjunto de propriedades de configuração

Descrição dos elementos xml

- <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
  - Define o driver JDBC (neste caso MySQL)
- <property name="javax.persistence.jdbc.user" value="root"/>
  - Define o usuário do banco de dados
- <property name="javax.persistence.jdbc.password" value="ifsul2017"/>
  - Define a senha do banco de dados

Descrição dos elementos xml

- <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/4k"/>
  - Define a URL (localhost:3306) e nome do banco de dados (4k)
- <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
  - Define o dialeto a ser usado (INNODB)
- <property name="hibernate.show_sql" value="true" />
  - Define se deseja visualizar no console como o hibernate está compondo o comando SQL a ser executado no banco de dados.

Descrição dos elementos xml

- <property name="hibernate.format_sql" value="true" />
  - Define se deseja identar o comando SQL
- <property name="hibernate.use_sql_comments" value="false"/>
  - Define se deseja que o hibernate inclua comentários no SQL explicando o que o comando faz

Descrição dos elementos xml

- <property name="hibernate.hbm2ddl.auto" value="update" />
  - **validate**: valida se o banco está de acordo com o código, sem alterar o banco de dados
  - **update**: atualiza o banco com os dados novos do programa
  - **create**: cria as tabelas e deleta qualquer dado previamente existente
  - **create-drop**: cria as tabelas como no modo create, mas deleta tudo ao final. Usado para testes.

Classe Pessoa

```
public class Pessoa {
@Id
@GeneratedValue
@Column(name="ID_PESSOA")
private Long id;
@Basic
private String nome;
Gets, sets e construtor
}
```

Classe Executavel (Salvar)

```
public class Executavel {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("teste");
EntityManager em = emf.createEntityManager();
Pessoa a = new Pessoa(“Joao”);
em.getTransaction().begin();
em.persist(a);
em.getTransaction().commit();
em.close();
emf.close();
}}
```

Classe Executavel (Buscar)

```
public class Executavel {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("teste");
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Pessoa pessoaJ = em.find(Pessoa.class,1L);
System.out.println(pessoaJ.getNome());
em.getTransaction().commit();
em.close();
emf.close();
}}
```

@Basic

- A anotação @Basic pode ter algumas propriedades de configuração
  - @Basic(fetch=FetchType.LAZY, optional=false)
    - Fetch = Identifica a forma de carregar os dados do BD
    - EAGER é o padrão, que ao relacionar o objeto a um registro do BD, já carrega os dados do registo no objeto.
    - LAZY carrega os dados do objeto somente quando atributo do objeto vai ser acessado. Pode ajudar a diminuir a carga do BD.
    - optional= Define que a coluna será NOT NULL. O padrão é true.

@Column

- A anotação @Column pode ter algumas propriedades de configuração
  - @Column(name="SSN", unique=true, nullable=false, description="description« , length=100, scale=9, precision=2, columnDefinition ="TEXT")
  - Name define o nome da coluna no BD
  - Unique define se a coluna poderá ter dados repetidos
  - Nullable define se a coluna poderá ter valores nulos
  - Description define um texto de descrição da coluna
  - Length define o tamanho da coluna
  - Scale, Precision Define para números BigDecimal, o número de dígitos à direita da vírgula e o número de dígitos significativos, respectivamente.

@Column

- A anotação @Column pode ter algumas propriedades de configuração
  - @Column(name="SSN", unique=true, nullable=false, description="description« , length=100, scale=9, precision=2, columnDefinition ="TEXT")
  - Scale define para números BigDecimal o número de dígitos à direita da vírgula
  - Precision define para números BigDecimal o número de dígitos significativos
  - columnDefinition define algum parâmetro de configuração do SQL da coluna.

@SequenceValue

- A anotação @SequenceValue pode ter a propriedade de configuração strategy
  - @SequenceValue(strategy="GenerationType.AUTO")
  - @SequenceValue(strategy="GenerationType.IDENTITY")
  - @SequenceValue(strategy="GenerationType.SEQUENCE", generator ="")
  - @SequenceValue(strategy="GenerationType.TABLE", generator="")

GenerationType.SEQUENCE

- A propriedade SEQUENCE exige anteriormente a criação de um SequenceGenerator
  - @SequenceGenerator(name = nome, sequenceName = nomenobanco, initialValue = 1, allocationSize = 100)
    - Name = nome do generator (este nome irá na propriedade generator do @SequenceValue)
    - sequenceName = nome do sequence no banco de dados
    - initialValue = id inicial
    - allocationSize = o JPA irá criar um bloco de IDs, do id inicial até + allocationSize. Quando acabar o bloco o JPA cria mais um bloco com esse tamanho. Isso visa aumentar o desempenho.

GenerationType.TABLE

- A propriedade TABLE exige anteriormente a criação de um TableGenerator
  - @TableGenerator(name=“nome", table=“nomedatabela", pkColumnName = “sequencias", valueColumnName = “proximo", pkColumnValue=“gen_id_pessoa", allocationSize=30)
  - Name = nome do tablegenerator (este nome irá na propriedade generator do @SequenceValue)
  - Table = nome da tabela interna para gerenciar os ids

GenerationType.TABLE

- pkColumnName = nome da coluna onde ficam os nomes das sequencias
- valueColumnName = nome da coluna que armazena o ultimo valor gerado da sequencia
- pkColumnValue = valor da sequencia
- allocationSize = tamanho do bloco

Classe composta por várias tabelas

- Mapear uma Entity em várias tabelas
- Supondo que os atributos da classe Pessoa esteja dividida em três tabelas:
  - Pbasica e Pcomplementar

```
@Entity
@Table(name=“PESSOA")
@SecondaryTables({
   @SecondaryTable(name=“PBASICA",
pkJoinColumns={@PrimaryKeyJoinColumn(name="ID_BAS")}),
   @SecondaryTable(name=“PCOMPLEMENTAR",
pkJoinColumns={@PrimaryKeyJoinColumn(name="ID_COMP")})
})
```

Várias classes em uma Entity

- Ex: Entidade Pessoa ter nome, idade e Endereço (rua, cep, cidade etc) na mesma tabela

```
@Embeddable
public class Endereco {

    private String logradouro;
    @Column(name="endereco")
    private String nome;
    private int numero;
}
```

```
@Entity
public class Pessoa {
@Id
private int id;
private String nome;
@Embedded
private Endereco endereco;
```

Tipos básicos

Java type	Database type
String (char, char[])	VARCHAR (CHAR, VARCHAR2, CLOB, TEXT)
Number (BigDecimal, BigInteger, Integer, Double, Long, Float, Short, Byte)	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
int, long, float, double, short, byte	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
byte[]	VARBINARY (BINARY, BLOB)
boolean (Boolean)	BOOLEAN (BIT, SMALLINT, INT, NUMBER)
java.util.Calendar	TIMESTAMP (DATETIME, DATE)
java.lang.Enum	NUMERIC (VARCHAR, CHAR)
java.util.Serializable	VARBINARY (BINARY, BLOB)

Chave composta simples

- Quando são usados tipos básicos para formação da chave
- @IdClass
  - Quando se referencia a classe que possui a estrutura da chave composta
- @EmbeddedId
  - Quando se insere dentro da classe a classe que possui a chave composta

@IdClass

```
@Entity
@IdClass(Cidades.class)
public class Prefeito {
@Id
private string estado;
@Id
private string cidade;

private string nome;
```

@IdClass

public class Cidades implements Serializable{

private String estado;
private String cidade;

Gets, sets, constructor vazio, construtor completo, hashcode e equals

@EmbeddedId

```
@Entity
public class Prefeito {
@EmbeddedId
private Cidades id;

private string nome;
```

@EmbeddedId

@Embeddable
public class Cidades implements Serializable{

private String estado;
private String cidade;

Gets, sets, constructor vazio, construtor completo, hashcode e equals

Chave composta simples Consulta

```
EntityManager em = …

Cidades c = new Cidades(“RS”, “Sapucaia do Sul");

Prefeito p = em.find(Prefeito.class, c);

System.out.println(p.getNome() + “ - ” + p.getCidade() + " / " + p.getEstado());
```

Tipo enumerado

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

@Enumerated(EnumType.STRING)
    private Sexo sexo;
}
```

```
public enum Sexo {
    MASCULINO, FEMININO
}
```

Coleção

```
@Entity
public class Pessoa {
    @Id
    @GeneratedValue
    private int id;
    private String name;

    @ElementCollection
    @CollectionTable(name = "pessoa_tem_emails")
    private Set<String> emails;
```

Mapeamento Um para Um Unidirecional

```
@Entity
public class Pessoa {
    @Id
    @GeneratedValue
    private int id;
    private String nome;
    @OneToOne
    @JoinColumn(name="celular_id" , unique=true)
    private Celular celular;
```

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

 }
```

Mapeamento Um para Um Bidirecional

```
@Entity
public class Pessoa {
    @Id
    @GeneratedValue
    private int id;
    private String nome;
    @OneToOne
    @JoinColumn(name="celular_id",
 unique=true)
    private Celular celular;
}
```

```
@Entity
public class Celular {
    @Id
    @GeneratedValue
    private int id;
    private int numero;
    @OneToOne(mappedBy="celular")
    private Pessoa proprietario;
 }
```

Mapeamento Um para Um Bidirecional

- mappedBy indica que a classe não deve recriar uma chave estrangeira, que a chave já foi criada no atributo celular da classe Pessoa.
- @JoinColumn indica o nome da chave estrangeira.

- Ao criar uma pessoa e um celular, ambos devem ser setados
  - p.setCelular(c);
  - c.setProprietario(p);

Chave composta complexa

```
@Entity
public class Casa {
    @Id
    @OneToOne
    @JoinColumn(name = “PROP_ID")
    private Pessoa proprietario;
    private String cor;
```

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

Chave composta complexa

```
@Entity
public class Casa {
    @Id
    private int idProprietario;
    @MapsId
    @OneToOne
    @JoinColumn(name = “PROP_ID")
    private Pessoa proprietario;
    private String cor;
```

Chave composta complexa

- Usado quando a chave composta não é formada por tipos de dados simples.
- Ao invés de acessar o id da casa buscando o id da pessoa (casa.getProprietario().getId()), pode se fazer um link com um atributo simples
- @MapsId relaciona o id do proprietário com o atributo idProprietario

Chave composta complexa

```
 public class CasaId implements Serializable{
private static final long serialVersionUID = 1L;
private int proprietario;
private int inquilino;
@Override
public int hashCode() {
        return proprietario + inquilino;
    }
```

```
@Override
   public boolean equals(Object obj) {
       if(obj instanceof CasaId){
           CasaId casaId = (CasaId) obj;
           return casaId.proprietario == proprietário
&& CasaId.inquilino == inquilino;
       }
       return false;
   }
```

Chave composta complexa
Id composto por mais de uma entidade

- Supondo:
- Uma casa de aluguel possui:
  - um proprietário
  - um inquilino.

Chave composta complexa
Id composto por mais de uma entidade

```
@Entity
@IdClass(CasaId.class)
public class Casa {
  @Id
  @OneToOne
  @JoinColumn(name = “PROP_ID")
   private Proprietario proprietario;

  @Id
  @OneToOne
  @JoinColumn(name = “INQ_ID")
    private Inquilino inquilino;
    private String cor;
```

Chave composta complexa
Id composto por mais de uma entidade

- A classe contém dois atributos inteiros apontando para as entidades do relacionamento.
- O nome dos atributos da Casa são exatamente iguais aos nomes dos atributos da CasaId. Se o atributo na entidade Casa fosse proprietarioDaCasa, o nome dentro da classe de id teria que ser proprietarioDaCasa ao invés de proprietario.
- Para que uma classe possa ser utilizada como id, ela deve:
  - Ter um construtor público sem argumentos
  - Implementar a interface Serializable
  - Sobrescrever os métodos hashCode/equals

Mapeamento Um para Muitos

```
@Entity
public class Celular {
    @Id
    @GeneratedValue
    private int id;
  @OneToMany(mappedBy = "celular")
    private List<Chamadas> chamadas;
    private int numero;
}
```

Mapeamento Um para Muitos

```
@Entity
public class Chamada {
    @Id
    @GeneratedValue
    private int id;
    @ManyToOne
    @JoinColumn(name = "celular_id")
    private Celular celular;
    private long duracao;
```

- Não existe autoreferenciamento
- Deve-se fazer:
  - chamada.setCelular(celular)
  - celular.setChamadas(chamadas)

Mapeamento Muitos para Muitos

```
@Entity
public class Aluno {
    @Id  @GeneratedValue
    private int id;
    private String nome;
    @ManyToMany
    @JoinTable(name = “aluno_turma", joinColumns = @JoinColumn(name =
“aluno_id"), inverseJoinColumns = @JoinColumn(name = “turma_id"))
    private List<Turmas> turmas;
```

Mapeamento Muitos para Muitos

```
@Entity
public class Turma {
    @Id
    @GeneratedValue
    private int id;
    private String nome;
    @ManyToMany(mappedBy=“turmas")
    private List<Aluno> alunos;
```

Mapeamento Muitos para Muitos com campos adicionais

- Imagine que ao matricular um aluno numa turma, fique registrado a data da matrícula
- Esse horário deve ser vinculado ao relacionamento, e não nas entidades aluno e turma.
- Para esse tipo de situação se usa a classe associativa ou entidade associativa.
- Com as classes associativas é possível armazenar informações extras ao criar um relacionamento de muitos para muitos.

Mapeamento Muitos para Muitos com campos adicionais

```
@Entity
public class Aluno {
     @Id
    @GeneratedValue
    private int id;
     private String nome;
     @OneToMany(mappedBy = “aluno")
    private List<AlunoTurma> turmas;
```

Mapeamento Muitos para Muitos com campos adicionais

```
@Entity
public class Turma {
    @Id
    @GeneratedValue
    private int id;
    private String nome;
     @OneToMany(mappedBy = “turma")
    private List<AlunoTurma> alunos;
```

Mapeamento Muitos para Muitos com campos adicionais

```
@Entity
@IdClass(AlunoTurmaId.class)
public class AlunoTurma {
    @Id
    @ManyToOne
    @JoinColumn(name="aluno_id")
    private Aluno aluno;
```

```
@Id
    @ManyToOne
    @JoinColumn(name="turma_id")
    private Turma turma;
    @Temporal(TemporalType.DATE)
    private Date data;
```

Mapeamento Muitos para Muitos com campos adicionais

```
public class AlunoTurmaId implements Serializable {
    private static final long serialVersionUID = 1L;
    private int aluno;
    private int turma;

  // gets, sets, constructor vazio, equals e hashcode
```

Mapeamento Muitos para Muitos com campos adicionais

- Os atributos presentes na classe AlunoTurmaId tem que ter o mesmo nome dos atributos Aluno e Turma encontrados na entidade AlunoTurma.
- O nome deve ser exatamente igual ao atributo

Cascade

```
EntityManager em = …
Carro c = new Carro(“Fusca");
Endereco e = new Endereco(“Rua copacabana”);
em.getTransaction().begin();
Pessoa p = em.find(Pessoa.class, 1);
p.setCarro(c);
p.setEndereco(e);
em.getTransaction().commit();
```

Cascade

- O código apresentará um erro de execução
- O objeto p, da classe Pessoa, de id 1, é buscado do banco
  - Attached: A entidade está dentro da transação, é monitorado pelo JPA
- O hibernate não conhece os objetos da classe Carro e Endereco que tentou-se fazer update em p.
  - Detached: Objetos “p” e “e” não foram associados ao contexto de persistência.
- Solução: CASCADE
  - @OneToOne, @OneToMany e @ManyToMany
  - Ex: @OneToOne(cascade = CascadeType.PERSIST)

Cascade

- Vantagem: propagação automática da ação configurada nos relacionamentos da entidade.
- Configurações:
- CascadeType.DETACH: Quando uma entidade for retirada do Persistence Context (o que provoca que ela esteja detached), essa ação será refletida nos relacionamentos.
  - Ocorre por Persistence Context finalizado ou por comando específico: entityManager.detach(), entityManager.clear().

Cascade

- CascadeType.MERGE: Quando uma entidade tiver alguma informação alterada (update), essa ação será refletida nos relacionamentos.
  - Ocorre quando a entidade for alterada e a transação finalizada ou por comando específico: entityManager.merge().
- CascadeType.PERSIST: Quando uma entidade for nova e inserida no banco de dados, essa ação será refletida nos relacionamentos.
  - Ocorre quando uma transação finalizada ou por comando específico: entityManager.persist().

Cascade

- CascadeType.REFRESH: Quando uma entidade tiver seus dados sincronizados com o banco de dados, essa ação será refletida nos relacionamentos.
  - Ocorre por comando específico: entityManager.refresh().
- CascadeType.REMOVE: Quando uma entidade for removida (apagada) do banco de dados, essa ação será refletida nos relacionamentos.
  - Ocorre por comando específico: entityManager.remove().
- CascadeType.ALL: Quando qualquer ação citada acima for invocada pelo JPA ou por comando, essa ação será refletida no objeto.
  - Ocorre por qualquer ação ou comando listados acima.

Cascade

- Observações:
  - É preciso ter bastante cuidado ao anotar um relacionamento com CascadeType.ALL. Uma vez que se a entidade for removida o seu relacionamento também é removido do banco de dados.
    - Ex: Ao excluir a pessoa p, o carro c também seria excluído do banco de dados.

Cascade

- Observações:
  - Se o carro atribuído ao relacionamento for um carro que já exista no banco mas está detached, o Cascade não irá ajudar nesse caso.
    - Ex:Ao executar o comando em.persist(p), o JPA tentará executar o persist nos objetos do relacionamento marcados com CascadeType.PERSIST (em.persist(c)).
    - Caso o carro já exista no banco, o JPA tentará inseri-lo novamente e uma mensagem de erro será exibida.
    - Seria necessário um objeto atualizado do banco de dados e o melhor modo de se fazer é utilizando o método getReferenceOnly().

Cascade

```
@Entity
public class Pessoa {
@Id
private int id;
private String nome;
@OneToOne(mappedBy="pessoa")
private Carro carro;
```

```
@Entity
public class Carro{
@Id
@GeneratedValue
private int id;
private String nome;
@OneToOne(cascade = CascadeType.PERSIST)
private Pessoa pessoa;
```

Cascade

- Forma correta de persistir:
  - em.persist(c)
- Erro:
  - Em.persist(p)
- O cascade só é acionado pelo JPA quando a ação for executada pela entidade em que o Cascade foi definido.
- No exemplo, apenas a classe Carro definiu Cascade para Pessoa
- Então apenas a classe Carro irá disparar a função de Cascade.

OrphanRemoval

- @OneToOne(orphanRemoval=true) ou OneToMany(orphanRemoval=true)
- Usado quando uma entidade é usada exclusivamente dentro de outra.
- Equivalente ao CascadeType.REMOVE
- Porém, considerando o exemplo de uma entidade pessoa que se relaciona com uma entidade endereco:
- Em um outro momento o endereco será removido do relacionamento.
  - pessoa.setEndereco(null);
- Quando a entidade pessoa for atualizada no BD, a entidade endereco será excluída do BD

Excluir

- em.remove(objeto);
- Ex: pessoa possui carro. Ao remover ocorreria um erro.
  - em.remove(pessoa);
- Solução:
  - CascadeType.REMOVE
  - OrphanRemoval
  - Colocar o relacionamento para “null” antes de realizar a exclusão

```
pessoa.setCarro(null);
em.remove(pessoa);
```

Lazy - Eager

```
@Entity
public class Carro {
   @Id
   @GeneratedValue
   private int id;
   private String nome;
   @ManyToOne
   private Pessoa pessoa;
```

```
@Entity
public class Pessoa {
   @Id
   private int id;     private String nome;
   @OneToMany(mappedBy = "pessoa", fetch = FetchType.LAZY)
   private List<Carro> carros;
   @Lob
   @Basic(fetch = FetchType.LAZY)
   private Byte[] figuraGrande;
```

Lazy - Eager

- Lazy indica que o relacionamento/atributo não será carregado do banco de dados de modo natural.
- Ao realizar o Pessoa p = em.find(Pessoa.class,1), a lista de carros e a figuraGrande não serão carregados.
  - Query mais leve e tráfego de dados na rede menor.
  - O atributo somente será carregado com um get.
  - p.getFiguraGrande() gerará um novo select no BD;

Lazy - Eager

- Por padrão o atributo será sempre eager.
- Relacionamentos que terminam em One serão EAGER por padrão: @OneToOne e @ManyToOne.
- Relacionamentos que terminam em Many serão LAZY por padrão: @OneToMany e @ManyToMany.

Erro: “cannot simultaneously fetch multiple bags”

- Acontece quando o JPA busca uma entidade com mais de uma coleção no banco de dados de modo EAGER.

Erro: “cannot simultaneously fetch multiple bags”

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

```
@OneToMany(mappedBy = "pessoa", fetch =
FetchType.EAGER, cascade = CascadeType.ALL)
    private List<Carro> carros;

    @OneToMany(fetch = FetchType.EAGER)
    private List<Cao> caes;
```

Erro: “cannot simultaneously fetch multiple bags”

- Ocorrerá o erro
- “javax.persistence.PersistenceException: org.hibernate.HibernateException: cannot simultaneously fetch multiple bags”.
- Uma lista (List, Collection) também pode ser conhecida por “bag”.
- Esse erro acontece porque o Hibernate tenta igualar o número de resultados vindo em uma consulta.
- Caso o SQL gerado retorne como resultado 2 linhas para a lista de caes, e uma para carros, o Hibernate irá repetir o carro para igualar as linhas do resultado dos caes.

Erro: “cannot simultaneously fetch multiple bags”

- Ex:
  - Carro
  - id: 4, cor: preto
  - Cao
  - id:7, nome: totó
  - Id: 8, nome: fifi
  - Pessoa
  - id:1, nome: Joao
  - em.find(Pessoa.class,1)

p.id	p.nome	cao.id	cao.nome	carro.id	carro.cor
1	Joao	7	toto	4	Preto
1	Joao	8	fifi	4	preto

Erro: “cannot simultaneously fetch multiple bags”

- 4 possíveis soluções:
  - Utilizar java.util.Set ao invés das outras coleções.
  - Utilizar EclipseLink.
  - Utilizar FetchType.LAZY ao invés de EAGER
    - solução parcial, caso uma consulta seja feita e seja utilizada join fetch nas coleções, o problema volta a acontecer.
  - Utilizar @LazyCollection ou @IndexColumn do próprio Hibernate na coleção

Herança - @MappedSuperClass

- Usado quando a classe pai não é uma entidade

@MappedSuperclass
public abstract class Pessoa {
  private String nome;
	@Entity
	public class PessoaFisica extends Pessoa {
	  @Id
	  private int id;
	  private String cpf;

Herança – Single_Table

- Usado quando ocorre herança entre classes e todos os dados devem ser persistidos.
- A estratégia Single Table utilizará apenas uma tabela para armazenar todas as classes da herança.

Herança – Single_Table

```
@Entity
@Table(name = “CAO")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = “NOME_CLASSE_CAO")


public abstract class Cao {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String nome;
```

Herança – Single_Table

```
@Entity
@DiscriminatorValue(“CAO_PEQUENO")
public class CaoPequeno extends Cao {
    private String latidoFino;
```

```
@Entity
@DiscriminatorValue(“CAO_GRANDE")
public class CaoGrande extends Cao {
    private int pesoCocoGrande;
```

Herança – Single_Table

- @Inheritance(strategy = InheritanceType.SINGLE_TABLE): deve ser utilizada na classe classe pai.
- @DiscriminatorColumn(name = “NOME_CLASSE_CAO”): cada linha da tabela pertencerá a uma entidade. Esta anotação define a coluna que irá conter a descrição da entidade de cada linha.
- @DiscriminatorValue: Define qual o valor a ser salvo na coluna descrita na anotação @DiscriminatorColumn.
- O ID é definido apenas na classe pai. Não é permitido sobrescrever o id de uma classe na hierarquia.

Herança – Single_Table

id	nome_ classe_cao	nome	latido_fino	pesoCocoGrande
1	CAO_PEQUENO	Gigante	Au	
2	CAO_PEQUENO	Monstro	Hunf	
3	CAO_PEQUENO	Hulk	Grrr	
4	CAO_GRANDE	Mindinho		1
5	CAO_GRANDE	Tiquinho		2
6	CAO_GRANDE	Formiga		1

Herança – Single_Table

- É possível utilizar o campo de descrição da classe com um inteiro
- @DiscriminatorColumn(name = “NOME_CLASSE_CAO”, discriminatorType = DiscriminatorType.INTEGER).
- @DiscriminatorValue(“1″)
  - O valor a ser salvo deve ser alterado na Entity também.
  - O número será salvo ao invés do texto.

Herança – Joined

- cada entidade terá seus dados em uma tabela específica.

```
@Entity
@Table(name = “CAO")
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "NOME_CLASSE_CAO")
public abstract class Cao {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String nome;
```

Herança – Joined

```
@Entity
@DiscriminatorValue(“CAO_GRANDE")
public class CaoGrande extends Cao {
    private int pesoCocoGrande;
```

```
@Entity
@DiscriminatorValue(“CAO_PEQUENO")
public class CaoPequeno extends Cao {
    private String latidoFino;
```

Herança – Joined

CAO		
id	nome_ classe_cao	nome
1	CAO_PEQUENO	Gigante
2	CAO_PEQUENO	Monstro
3	CAO_PEQUENO	Hulk
4	CAO_GRANDE	Mindinho
5	CAO_GRANDE	Tiquinho
6	CAO_GRANDE	Formiga

CAO_GRANDE	
id	pesoCocoGrande
4	1
5	2
6	1

CAO_PEQUENO	
id	latido_fino
1	Au
2	Hunf
3	Grrr

Herança – Tabela Por Classe Concreta

- Uma tabela por entidade concreta.
- Caso uma entidade abstrata seja encontrada em uma hierarquia a ser persistida, essas informações serão salvas nas classes concretas abaixo dela.
- Não existe mais a necessidade da anotação: @DiscriminatorColumn(name = “NOME_CLASSE_CAO”). Cada classe concreta terá todos os seus dados, com isso não existirá mais uma tabela com dados que não pertençam a ela.
- Não existe mais a necessidade da anotação: @DiscriminatorValue pelo mesmo motivo explicado acima.

Herança – Tabela Por Classe Concreta

```
@Entity
@Table(name = “CAO")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Cao {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String nome;
```

Herança – Tabela Por Classe Concreta

```
@Entity
public class CaoPequeno extends Cao {
    private String latidoFino;
```

```
@Entity
public class CaoGrande extends Cao {
    private int pesoCocoGrande;
```

Herança – Tabela Por Classe Concreta

CAO_GRANDE		
id	nome	pesoCocoGrande
4	Mindinho	1
5	Tiquinho	2
6	Formiga	1

CAO_PEQUENO		
id	nome	latido_fino
1	Gigante	Au
2	Monstro	Hunf
3	Hulk	Grrr

Mapear Herança – Prós e Contras
SINGLE_TABLE - Prós

- A estrutura da tabela no banco de dados fica mais fácil de analisar. Apenas uma tabela é necessária
- Os atributos das entidades poderão ser encontrados em uma única tabela.
- Todos os dados persistidos serão encontrados em uma única tabela.
- Em geral tem um bom desempenho.

Mapear Herança – Prós e Contras
SINGLE_TABLE - Contras

- Não pode ter campos “não nulos”.
- Imagine um caso onde a classe CaoPequeno tivesse o atributo não nulo “corDoPelo” no banco de dados.
- Ao persistir CaoGrande que não tem esse atributo, uma mensagem de erro avisando que “corDoPelo” estava nulo seria enviada pelo banco de dados.

Mapear Herança – Prós e Contras
JOINED - Prós

- Cada entidade terá uma tabela contendo seus dados.
- Irá seguir o conceito de orientação a objetos aplicado no modelo de dados

Mapear Herança – Prós e Contras
JOINED - Contras

- O insert é mais custoso.
- Um insert deverá ser feito para cada entidade utilizada na hierarquia.
- Se existe uma herança C extends B extends A, ao persistir um C, três inserts serão realizados – um em cada entidade mapeada.
- Quanto maior for a hierarquia, maior será o número de JOINs realizado em cada consulta.

Mapear Herança – Prós e Contras TABLE_PER_CLASS - Prós

- Quando a consulta é realizada em apenas uma entidade seu retorno é mais rápido.
- Em uma tabela constam apenas os dados da classe.

Mapear Herança – Prós e Contras
TABLE_PER_CLASS - Contras

- Colunas serão repetidas.
- As colunas referentes aos atributos das classes abstratas serão repetidas nas tabelas das classes filhas.
- Quando uma consulta é realizada para trazer mais de uma entidade da herança, essa pesquisa terá um custo maior.
- Será utilizado UNION ou uma consulta por tabela.

Referências

- JPA Tutorial. Disponível em http://www.w3ii.com/jpa/default.html
- Coelho, H. JPA Mini Livro – Primeiros passos e conceitos detalhados. Disponível em: http://uaihebert.com/jpa-mini-livro-primeiros-passos-e-conceitos-detalhados