JPA - Java Persistence API

Msc. Roberto Ribeiro Rocha

Banco de Dados II

2022

Roteiro

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Objetivos

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Apresentar as características básicas da JPA.
- Entender como é feito o mapeamento objeto-relacional.
- Entender como criar consultas utilizando HQL - *Hibernate Query Language*.
- Praticar a persistência e consulta de dados.

Motivação

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Armazenar e buscar objetos em um banco relacional.
- Usar as facilidades de uma consulta “orientada a objetos”.
- **Problema atual:** discrepância entre as linguagens orientadas a objetos e os bancos relacionais.
- **Solução:** usar uma camada intermediária para resolver esse problema.

Anotações

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- É um recurso da linguagem Java que provê informações/dados sobre um programa
- Não é parte do programa em si. Não tem efeito direto na execução do código que está anotado.
- Usos das anotações:
  - → Informação para o compilador (erros e *suppress warnings*);
  - → Geração de código, configurações, etc;
  - → Processamento extra em *runtime* (utilizadas na execução).
- Elas podem ser aplicadas em: classes, atributos e métodos.
- A anotação deve aparecer **antes** do elemento desejado.

Exemplos de anotações

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

```
@FunctionalInterface
interface DisplayInterface {
    void display(String message);
}

@Configuration(option = "managed", type = "memory", timeout = 15)
class MyClass() {

    @Override
    public String toString() { ... }

    @Deprecated
    public void method01() { ... }

    @SuppressWarnings("unchecked")
    void method02() { ... }
```

Processamento de anotações

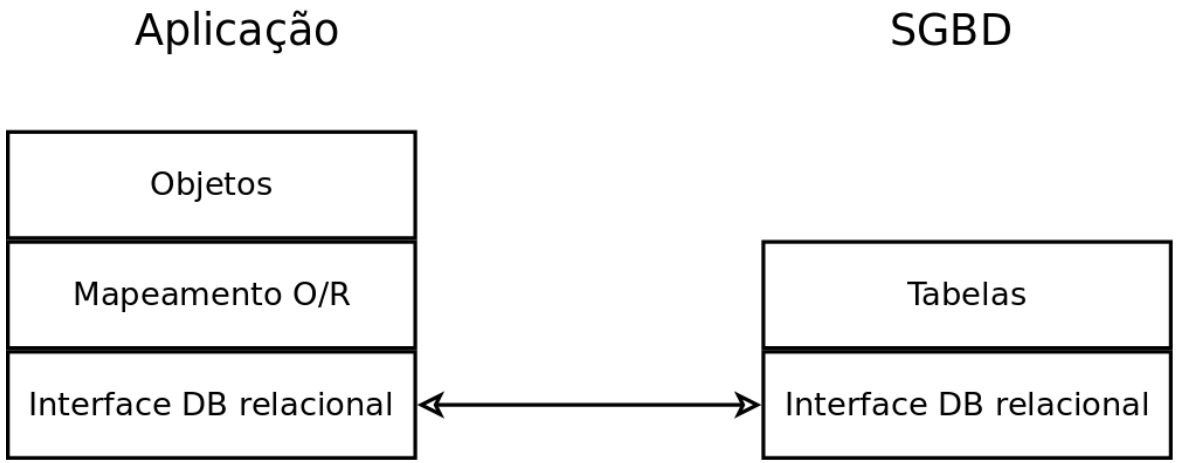
UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- O Java permite que programadores definam suas próprias anotações
- Também permite escrever processadores de anotações, que leem as anotações do código e executem ações baseadas nestas anotações.
- Esse recurso é nativo a partir do Java 6.
- Este recurso é utilizado no mapeamento Objeto-Relacional.

Mapeamento Objeto-Relacional - ORM

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Provê uma solução para armazenar objetos na tecnologia relacional.
- A ideia da persistência Objeto-Relacional (O/R):
  - → reunir as vantagens do **modelo orientado a objetos**, com o desempenho e a confiabilidade do **banco de dados relacional**.
- A persistência é uma abstração de alto-nível sobre JDBC.



JPA - Java Persistence API

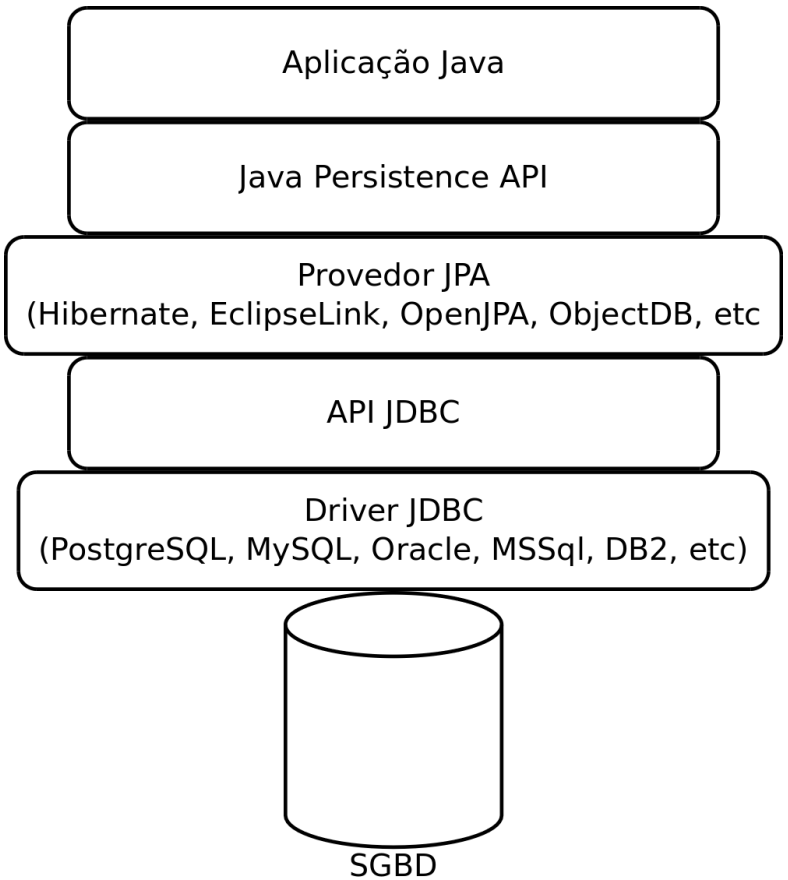
UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- É uma especificação padrão para mapeamento objeto-relacional e gerenciamento de persistência da plataforma Java.
- Ela define a padronização da camada de persistência.
- A JPA permite descrever de modo declarativo:
  - → o mapeamento O/R,
  - → uma linguagem de consulta e
  - → ferramentas para manipular entidades.
- Essa padronização permite um aumento relevante da produtividade.

JPA - Java Persistence API

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Camadas envolvidas quando uma aplicação utiliza a JPA para acessar o banco de dados.



Componentes JPA

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Componentes para o funcionamento do JPA:

- **Entity Manager**: responsável por praticamente todas operações de persistência e gerenciamento de objetos.
- **Persistence Context**: mantém os objetos que são manipulados pelo *EntityManager*.
- **Persistence Unit**: provê a configuração (JDBC) do banco de dados para o provedor JPA.
- **Provedores**: *frameworks* de persistência que implementam a JPA. Os mais comuns no mercado são:
  - → Hibernate (`http://jpa.hibernate.org`)
  - → TopLink/EclipseLink (Oracle)
  - → OpenJPA (`http://openjpa.apache.org`)

O *Framework* Hibernate

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- O **Hibernate** é um *framework* ORM para ambientes Java e outras linguagens.
- Ele implementa a JPA, e possui alguns recursos extras.
- Segundo a documentação oficial: *“o Hibernate pretende retirar do desenvolvedor cerca de 95 % das tarefas mais comuns de persistência de dados"*.

O *Framework* Hibernate

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Algumas características:
  - → Suporta classes com agregações, herança, polimorfismo, composições e coleções;
  - → Permite a escrita de consultas através de HQL[1] e SQL;
  - → É um *framework* não intrusivo (não restringe a arquitetura);
  - → Implementa a especificação Java Persistente API (JPA);
  - → Comunidade grande e ativa, mantido pela Jboss sob a licença LGPL.
- Os *jar's* do Hibernate devem ser adicionados no projeto Java.
- O *jar* do JDBC do banco de dados também deve ser utilizado.

[1] *Hibernate Query Language*

O *Framework* Hibernate

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Lista de *jar's* do Hibernate 3.5 (deve-se colocar no *classpath*)

- `hibernate-core-3.5.0-Final.jar`
- `hibernate-annotations-3.5.0-Final.jar`
- `hibernate-commons-annotations-3.2.0.Final.jar`
- `hibernate-entitymanager-3.5.0-Final.jar`
- `hibernate-jpa-2.0-api-1.0.1.Final.jar`
- `antlr-2.7.6.jar`
- `asm-3.1.jar`
- `cglib-2.2.jar`
- `commons-collections-3.1.jar`
- `dom4j-1.6.1.jar`
- `javassist-3.9.0.GA.jar`
- `jta-1.1.jar`
- `slf4j-api-1.5.8.jar`
- `slf4j-simple-1.4.2.jar`
- `xml-apis-1.0.b2.jar`
- `postgresql-8.2-512.jdbc4.jar`, o JDBC deve ser trocado caso seja usado outro banco.
- `hsqldb.jar`, o JDBC deve ser trocado caso seja usado outro banco.

Outra opção é utilizar o **Maven** para gerenciar essas dependências.

Prática 1 - Criar e configurar um projeto...

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Criar um Maven Project com as seguintes informações:
  - → Archetype: quickstart
  - → groupId: ‘nome do grupo’
  - → artifactId: ‘nome do projeto’
  - → version: 1.0
- Adicionar a dependência ‘hibernate-core’;
- Adicionar a dependência do driver JDBC do banco de dados;
- Criar a classe HibernateUtil.java;
- Criar o arquivo persistence.xml dentro da pasta META-INF.

Gerenciamento e uso das Entidades

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

A classe `HibernateUtil.java`

- Ela foi criada para se obter facilmente uma instância de `EntityManager` (ver projeto da aula).
- Ela cria objetos `EntityManager` que utiliza um *persistence context*.
- Esse contexto é configurado no arquivo `persistence.xml`:
  - → é colocado junto com o projeto (ver pasta `META-INF`).
  - → serve para facilitar a alteração das configurações de contexto com mínimas alterações no código Java.
  - → deve-se configurar: usuário, senha, url, *driver* JDBC e dialeto, específicos para cada banco de dados.

Gerenciamento e uso das Entidades

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Para fazer uso das entidades, existem classes da API que auxiliam na manipulação e criação do ambiente JPA, são elas:

- EntityManager: gerencia as entidades (controla a criação e remoção de entidades);
- EntityManagerFactory: classe que cria um *EntityManager*;
- Persistence: fornece dados de banco para a fábrica.

Gerenciamento e uso das Entidades

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Uma *persistence unit* define o conjunto de configurações utilizadas para a conexão com o banco de dados dentre outras propriedades (configuradas no arquivo `persistence.xml`)
- O código usa a classe `HibernateUtil` para obter uma instância do `EntityManager` para manipular as entidades.
- Mas antes vamos dar uma olhada em alguns métodos do `EntityManager`.

EntityManager

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Um objeto *EntityManager* permite criar, remover e buscar objetos.
- Principais métodos da interface `EntityManager`:

```
void persist(Object entity) // Gerencia e persiste uma instância.
void remove(Object entity) // Remove a instância da entidade.
<T> merge(T entity) // Faz o merge/update da entidade no contexto atual.

<T> find(Class<T> entityClass, Object primaryKey) // Busca pela PK.

// Cria uma instância de Query para executar uma SQL nativa.
Query createNativeQuery(String sqlString)
//Cria uma instância de TypedQuery a partir da execução da query.
TypedQuery<T> createQuery(String qlString, Class<T> resultClass)

// Obtém o objeto que controla a transação.
EntityTransaction getTransaction()
```

Mapeamento das entidades

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- É feito através das anotações (@) nos POJO's[2]
- A classe anotada se torna uma **entidade**.
- As anotações são metadados para o mapeamento entre objetos e o banco (de forma transparente para o desenvolvedor).
- As classes, interfaces e anotações necessárias estão no pacote **javax.persistence**.
- As configurações são feitas por **exceção**
  - → somente é necessário configurar o que estiver fora do padrão, pois a omissão de configuração indica ao *framework* de persistência a configuração *default*.

[2] *Plain-Old Java Object*

Mapeamento do POJO (entidade)

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Algumas características de uma classe POJO:
  - → A classe deve ser anotada com **@Entity**;
  - → Deve ter um construtor sem argumentos, público ou protegido, mas pode ter outros construtores;
  - → Não deve ser final (podem ser extensíveis);
  - → Deve implementar a interface *Serializable*.
  - → Deve ter os métodos `set` e `get` para cada atributo.
- Esses POJO's são chamados de *beans de entidade* ou *entity beans* ou simplesmente *entity*.
- Eles devem implementar os métodos `equals` e `hashcode`.

Mapeamento das entidades

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- O mapeamento entre o modelo Orientado a Objetos e o modelo Relacional possui a seguinte relação:
  - Classe $\rightarrow$ Tabela
  - Objeto $\rightarrow$ Linha da tabela
  - Atributo $\rightarrow$ Coluna da tabela
  - Associação $\rightarrow$ Chave estrangeira
- As principais anotações são: **@Entity**, **@Table**, **@Column**, **@Id**, **@GeneratedValue**, **@SequenceGenerator**, **@Temporal**, **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@JoinColumn**, **@Transient**.
- Veremos um pouco de cada uma delas...

Anotações de mapeamento

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

@Entity: Indica que uma classe Java será tratada como uma entidade. Essa anotação deve ser colocada antes do nome da classe.

```
@Entity
public class Cargo implements Serializable {
   ...
}
```

Anotações de mapeamento

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

@Id: Indica que o atributo é chave primária. Exemplo:

```
@Entity
public class Cargo implements Serializable {
  @Id
  private Integer codigo;
  ...
}
```

Prática 2 - Criando uma Entity e uma tabela

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

```java
@Entity
public class Cargo implements Serializable { //nosso POJO

  @Id
  private Integer codigo;
  private String nome;
  private float salario;

  //Implementar o construtor default
  //Implementar os sets e gets...
```

```sql
CREATE TABLE CARGO (
  CODIGO INTEGER NOT NULL,
  NOME VARCHAR(30) NOT NULL,
  SALARIO FLOAT NOT NULL,
  PRIMARY KEY (CODIGO)
);
```

Prática 2.1 - Persistindo objetos...

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

```
//cria um objeto de teste
Cargo programador = new Cargo();
programador.setCodigo(7);
programador.setNome("Programador");
programador.setSalario(3000f);

//obter uma instância do EntityManager
EntityManager em;
em = HibernateUtil.getEntityManager();

//persistir o objeto (dentro de uma transação)
em.getTransaction().begin();//inicia a transação
em.persist(programador);
em.getTransaction().commit();//finaliza a transação
```

Anotações de mapeamento

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

@Table: Indica que essa classe será mapeada para uma tabela.

```
@Entity
@Table(name="CARGOS")
public class Cargo implements Serializable {
   ...
}
```

Importante: Por padrão, a JPA assume que o nome da entidade (classe) seja o mesmo nome da tabela quando essa anotação não for definida.

Prática 2.2 - Testando o @Table

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Modificar o nome da tabela:

```
ALTER TABLE CARGO RENAME TO POSITION;
```

- Executar o main para ver o erro relacionado com a incompatibilidade entre o nome da classe e o nome da tabela.
- Incluir o @Table(name="POSITION") na classe
- Limpar a tabela, caso necessário (evitar *duplicate key*)
- Executar o main novamente e verificar o cargo inserido.

Anotações de mapeamento

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

@Column: Define configurações específicas para o atributo da classe de acordo com sua respectiva coluna da tabela.
Propriedades mais comuns:

- **name** que indica o nome da coluna na tabela,
- **nullable** que indica se o campo pode ser *null* e
- **length** que indica o tamanho da coluna.

Os dois últimos permitem fazer a validação antes de persistir os dados.

```
@Entity
public class Cargo implements Serializable {
    ...
  @Column(name="CODE")
  private Integer codigo;

  @Column(name = "NAME", nullable = false, length=30)
  private String nome;
  ...
```

Prática 2.3 - Testando o @Column

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Modificar o nome das colunas da tabela:

```
ALTER TABLE POSITION RENAME COLUMN CODIGO TO CODE;
ALTER TABLE POSITION RENAME COLUMN NOME TO NAME;
ALTER TABLE POSITION RENAME COLUMN SALARIO TO WAGE;
```

- Executar o main para ver o erro devido à modificação dos nomes das colunas.
- Incluir as anotações `@Column` nos respectivos atributos da classe.
- Executar o main novamente e verificar o cargo inserido.

Anotações de mapeamento

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

@Enumerated: mapeaia atributos do tipo `enum` (constantes).
Possui 2 opções de `EnumType` para armazenar o valor no banco:

- ORDINAL (*default*): armazena o valor numérico da constante
- STRING: armazena o nome da constante

```
public enum TipoCargo { CLT, PJ }
```

O atributo `type` do cargo será mapeado automaticamente:

```
@Entity
public class Cargo implements Serializable {
  ...
  @Enumerated(EnumType.ORDINAL)
  private TipoCargo type;
  ...
```

Prática 2.4 - Testando o @Enumerated

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Modificar a tabela `POSITION`:

```
ALTER TABLE POSITION ADD COLUMN TYPE INTEGER NULL;
```

- Criar o *enum*.

```
public enum TipoCargo { CLT, PJ }
```

- Criar o atributo `type` na classe `Cargo`:

```
@Enumerated(EnumType.ORDINAL)
private TipoCargo type; //criar os set's e gets
```

- Modificar o main para setar o valor do tipo do cargo.
- Executar o main novamente e verificar o cargo inserido.
- (Extra) Testar com `EnumType.STRING` e a coluna `TYPE` com `VARCHAR(10)`.

Campos gerados automaticamente (auto-incremento

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

@GeneratedValue: indica que o atributo possui geração automática de valores (auto-incremento ou *sequence*).
Ele possui 2 atributos:

- **strategy**: tipo de geração (`sequence` ou `auto-incremento`)
- **generator** (opcional): nome do elemento gerador (usado em conjunto com outra anotação).

Exemplo de anotação com auto-incremento:

```
@Entity
public class Department implements Serializable {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer code;
}
```

Prática 2.5 - Inserindo com auto-incremento

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Criar a tabela DEPARTMENT:

```
CREATE TABLE DEPARTMENT (
  CODE SERIAL NOT NULL,
  NAME VARCHAR(30) NOT NULL,
  PRIMARY KEY (CODE)
);
```

Criar a classe Department:

```
@Entity
public class Department implements Serializable {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY) //auto-incremento
  private Integer code;
  private String name;
  //Criar o construtor default e implementar os sets e gets.
```

Implementar um main para instanciar e persistir um `Department`.

Campos gerados automaticamente

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Usando Sequence

@SequenceGenerator: é um possível gerador, definido junto com o *@GeneratedValue*. Associa o atributo a uma *sequence*.
Ele possui 3 atributos:

- **name**: indica o nome do gerador usado na anotação anterior (`@GeneratedValue`),
- **sequenceName**: é o nome da *sequence* no banco e
- **allocationSize**: indica a quantidade[3] a ser incrementada.

[3] `http://royontechnology.blogspot.com.br/2010/04/note-on-allocationsize-parameter-of.html`

Campos gerados automaticamente

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Usando Sequence

Exemplo usando sequence:

```
@Entity
public class Funcionario implements Serializable {

  @Id
  @GeneratedValue(strategy=GenerationType.SEQUENCE,
                  generator="regSeqGenerator")
  @SequenceGenerator(name="regSeqGenerator",
                     sequenceName="SEQ_FUNCIONARIO", //sequence do banco
                     allocationSize=1)
  private Integer registro;
  ...
}
```

Prática 2.6 - Inserindo objetos com sequence

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Criar a sequence e testá-la:

```
CREATE SEQUENCE SEQ_FUNCIONARIO START WITH 1;

--Consulta um valor da sequence (no PostgreSQL)
SELECT NEXTVAL('SEQ_FUNCIONARIO');
```

Criar a tabela `Funcionario`:

```
CREATE TABLE FUNCIONARIO (
  REGISTRO INTEGER NOT NULL,
  NOME VARCHAR(30) NOT NULL,
  SEXO VARCHAR(1) NOT NULL,
  PRIMARY KEY (REGISTRO)
);
```

Prática 2.7 - Inserindo objetos com sequence

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Criar a classe `Funcionario`:

```
@Entity
public class Funcionario implements Serializable {
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="regSeqGen")
@SequenceGenerator(name="regSeqGen", sequenceName="SEQ_FUNCIONARIO",
                   allocationSize=1) //usando sequence
  private Integer registro;
  private String nome;
  private String sexo;
  //Criar o construtor default e implementar os sets e gets.
```

Implementar um método main para instanciar um objeto da classe `Funcionario` e persistir ele no banco de dados.

Anotações de mapeamento

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

@Temporal: usado para informações relacionadas ao tempo. Deve possuir o tipo adequado, de acordo com o tipo de dado da coluna correspondente no banco:

- DATE mapeia para a classe `java.sql.Date`
- TIME mapeia para a classe `java.sql.Time`
- TIMESTAMP mapeia para a classe `java.sql.Timestamp` ou `java.util.Date`.

```
...
@Temporal(TemporalType.TIMESTAMP)
private Date nascimento; //possui data e hora
...
```

Observação: O tipo do atributo é `java.util.Date`

Anotações de mapeamento

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

@Transient: indica que o atributo não será mapeado para o banco de dados.

```
@Entity
public class Funcionario implements Serializable {
  ...
  @Transient
  private double valorTemporario; //não será persistido
  ...
```

Observação: Essas e outras anotações de atributos também podem ser colocadas nos métodos *get's* de seus respectivos atributos.

Prática 2.8 - Datas e ignorando atributos

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Incluir o campo `NASCIMENTO` a tabela `FUNCIONARIO`:

```
ALTER TABLE FUNCIONARIO ADD COLUMN NASCIMENTO TIMESTAMP NULL;
```

- Incluir os dois atributos na classe `Funcionario`:

```
@Temporal(TemporalType.TIMESTAMP)
private Date nascimento;
@Transient
private int idade;
//Implementar os sets e gets destes 2 atributos.
```

- No main, incluir uma data de nascimento no objeto `Funcionario`.
- Executar o método main para instanciar e persistir um `Funcionario` no banco de dados.

Consultas - Buscando objetos...

O JPA possui recursos que facilitam a busca no banco de dados.

O EntityManager possui métodos que nos auxiliam nas consultas:

- Método `find(entityClass, primaryKey)`: busca pela PK.
- Método `createQuery(qlString, resultClass)`: veremos mais adiante.
- Médoto `createNativeQuery(sqlString)`: veremos mais adiante.

Prática 3.1 - Buscando objetos...

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

```
EntityManager em = HibernateUtil.getEntityManager();

Cargo objCargo = em.find(Cargo.class, 1); //busca cargo com pk 1
System.out.println("Cargo: " + objCargo);

Department objDep = em.find(Department.class, 1);//busca dep com pk 1
System.out.println("Departamento: " + objDep);

Funcionario objFun = em.find(Funcionario.class, 1);//busca func com pk 1
if (objFun == null) {
 System.out.println("Não há funcionário com código 1. objFun is null.");
} else {
 System.out.println("Funcionario: " + objFun);
}
```

Observação: para facilitar a impressão dos valores, implementar o método `toString()` na classe.

Editando e removendo objetos...

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Após buscar um objeto, podemos fazer update ou remover ele.

- Método `remove(entity)`: remove a instância daquela entidade.
- Método `merge(entity)`: faz o update da entidade no contexto atual.

Relembrando: toda a operação que altera o banco de dados deve ser feita dentro de uma **transação**.

- `em.getTransaction().begin();`
- `//código que modifica o banco`
- `em.getTransaction().commit();`

Prática 3.2 - Buscando objetos e fazendo update...

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

```
EntityManager em = HibernateUtil.getEntityManager();

int pkCargo = 1;
Cargo objCargo = em.find(Cargo.class, pkCargo);// busca o cargo pela pk
if (objCargo != null) { // se achar, faz update
  objCargo.setNome("Gerente"); // modifica o nome e salário
  objCargo.setSalario(5000f);

  em.getTransaction().begin();
  em.merge(objCargo); // faz o update
  em.getTransaction().commit();
} else {
  System.out.println("Cargo " + pkCargo + " não encontrado.");
}
```

Prática 3.3 - Buscando objetos e deletando...

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

```
EntityManager em = HibernateUtil.getEntityManager();

int pkCargo = 1;
Cargo objCargo = em.find(Cargo.class, pkCargo);// busca o cargo pela pk
if (objCargo != null) { // se achar, faz delete
  em.getTransaction().begin();
  em.remove(objCargo); // remove o cargo
  em.getTransaction().commit();
} else {
  System.out.println("Cargo " + pkCargo + " não encontrado.");
}
```

Consultas - Buscando objetos...

Voltando nas consultas, o JPA possui recursos que facilitam a busca mais de um objeto (lista) no banco de dados.

O EntityManager possui dois métodos importantes para as consultas:

- Método `createQuery(qlString, resultClass)`: cria uma instância de `TypedQuery` a partir da execução da query.
- Médoto `createNativeQuery(sqlString))`: cria uma instância de `Query` para executar uma SQL nativa.

Prática 3.4 - Fazendo consultas simples

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

```
EntityManager em = HibernateUtil.getEntityManager();

String ql = "select c from Cargo c"; // define a consulta
TypedQuery<Cargo> query = em.createQuery(ql, Cargo.class); // cria a
    query
List<Cargo> cargos = query.getResultList();// executa a consulta

System.out.println("Cargos: " + cargos); // imprimir os cargos
```

Observações sobre o *select*:

- Essa consulta é orientada a objetos:
- a consulta está buscando objetos da classe `Cargo`.
- a letra `c` é um objeto da classe `Cargo`.
- `select c` significa trazer os objetos da classe `Cargo`.

Prática 3.5 - Fazendo consultas com filtros

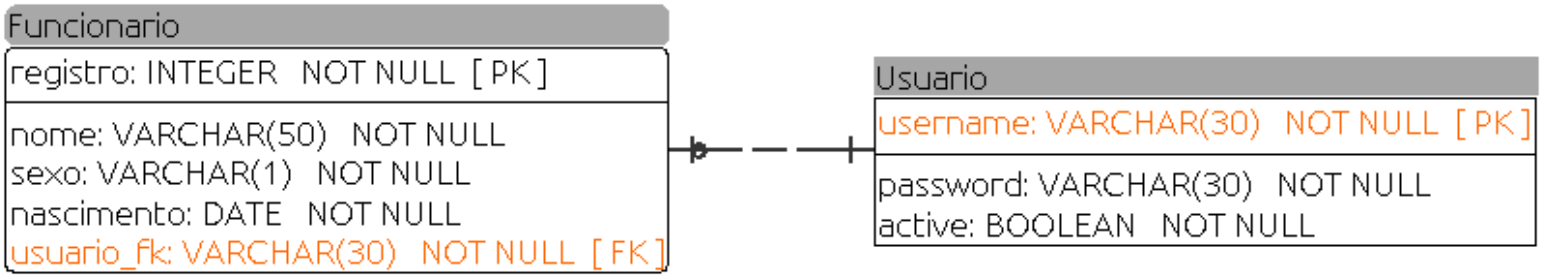
UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

```
EntityManager em = HibernateUtil.getEntityManager();

float salario = 1000f;
String ql = "select c from Cargo c where c.salario > :filtro_salario";
TypedQuery<Cargo> query = em.createQuery(ql, Cargo.class);
query.setParameter("filtro_salario", salario); //seta o filtro de salá
    rio

List<Cargo> cargos = query.getResultList();
System.out.println("Cargos: " + cargos); // imprimir os cargos
```

Anotação `@OneToOne` - contexto relacionamento 1:1

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Considerar o seguinte cenário:



A tabela `Funcionario` possui uma chave estrangeira de `Usuario`.
A classe de mapeamento da tabela `usuario` fica assim:

```
@Entity
public class Usuario implements Serializable {

  @Id
  private String username;
  private String password;
  private boolean active;
    ...
```

Anotação `@OneToOne` - definição

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Usada em um relacionamento um-para-um (1:1).
- É uma anontação do atributo
- Permite a entidade da classe atual acessar a entidade do atributo.

Exemplo:

```
@Entity
public class Funcionario implements Serializable {
  ...
  @OneToOne
  @JoinColumn(name="USUARIO_FK")
  private Usuario usuario;
```

A anotação `@JoinColumn` indica o campo da **chave estrangeira** da tabela `Funcionario` que está associado com a chave primária (`@Id`) da classe `Usuario`.

Anotação `@OneToOne`

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Ela pode ser usada de ambos lados em um relacionamento 1:1. Veja a classe `Usuario` atualizada:

```
@Entity
public class Usuario implements Serializable {

  @Id
  private String username;
  private String password;
  private boolean active;

  @OneToOne(mappedBy="usuario")
  private Funcionario funcionario;
```

O **mappedBy** indica que o atributo `usuario` definido na classe `Funcionario` possui a especificação da chave estrangeira.

Prática 4.1.1 - @OneToOne - Tabelas

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Criar a tabela USUARIO:

```
CREATE TABLE USUARIO (
  USERNAME VARCHAR(30) NOT NULL,
  PASSWORD VARCHAR(30) NOT NULL,
  ACTIVE BOOLEAN NOT NULL,
  PRIMARY KEY (USERNAME)
);
```

Incluir uma chave estrangeira na tabela FUNCIONARIO:

```
DELETE FROM FUNCIONARIO; --para facilitar a criação da FK

ALTER TABLE FUNCIONARIO
                ADD COLUMN USUARIO_FK VARCHAR(30) NOT NULL;

ALTER TABLE FUNCIONARIO
      ADD CONSTRAINT FUNCIONARIO_USUARIO_FK
      FOREIGN KEY (USUARIO_FK) REFERENCES USUARIO(USERNAME);
```

Prática 4.1.2 - @OneToOne - Classes

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Criar a classe Usuario:

```
@Entity
public class Usuario implements Serializable {
  @Id
  private String username;
  private String password;
  private boolean active;

  @OneToOne(mappedBy="usuario")
  private Funcionario Funcionario;
  //criar os sets e gets de todos os atributos e o construtor
```

Adicionar o atributo usuario na classe Funcionario:

```
@OneToOne
@JoinColumn(name="USUARIO_FK")
private Usuario usuario;
//criar os sets e gets
```

Prática 4.1.3 - `@OneToOne` - Salvando objetos

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Executar o main (Prática 2.6) para persistir um `Funcionario` e verificar o erro da execução.
- Modificar o main para executar os seguintes passos:

```
// criar um objeto Usuario
// criar um objeto Funcionario
// setar o usuário dentro do funcionário
  //exemplo: objFuncionario.setUsuario(objUsuario);
// obter o EntityManager
// persistir o objeto Usuario
// persistir o objeto Funcionario
```

- Executar o main para persistir os dois objetos.
- Verificar os valores armazenados nas duas tabelas.

Prática 4.1.4 - `@OneToOne` - Consultando objetos

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Criar um main para executar os seguintes passos:

```
//buscar um usuário (usar o find)
//imprimir o usuário
//buscar um funcionario do usuário consultado anteriormente **
  //Funcionario objFun = objUser.getFuncionario();
//imprime o funcionario
```

** Observe que não foi preciso fazer o find do `Funcionario`.
** A busca do `Usuario` já trouxe o `Funcionario` junto.

- Veja que a consulta dos dois objetos foi feito em um único select.

Prática 4.1.5 - `@OneToOne` - Consultando objetos

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Criar um main para executar os seguintes passos:

```
// buscar um funcionario (usar o find)
// imprimir o funcionario
// buscar um usuário do funcionario consultado anteriormente **
  //Usuario objUser = objFun.getUsuario();
// imprime o usuário
```

** A busca do `Funcionario` já trouxe o `Usuario` junto.

- A consulta dos dois objetos também foi feito em um único select.
- Incluir a configuração abaixo na anotação `@OneToOne` na classe `Funcionario`:

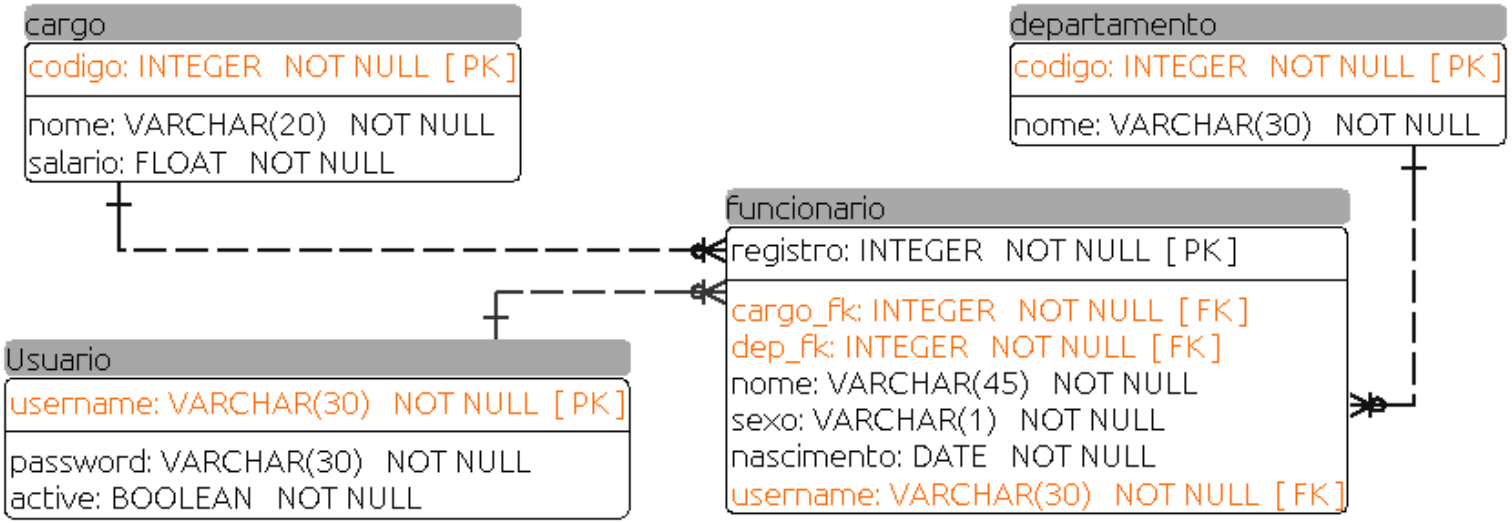
```
fetch=FetchType.LAZY
```

- Executar o main novamente e verificar os *selects* executados.

Anotações de Mapeamento - `@ManyToOne`

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

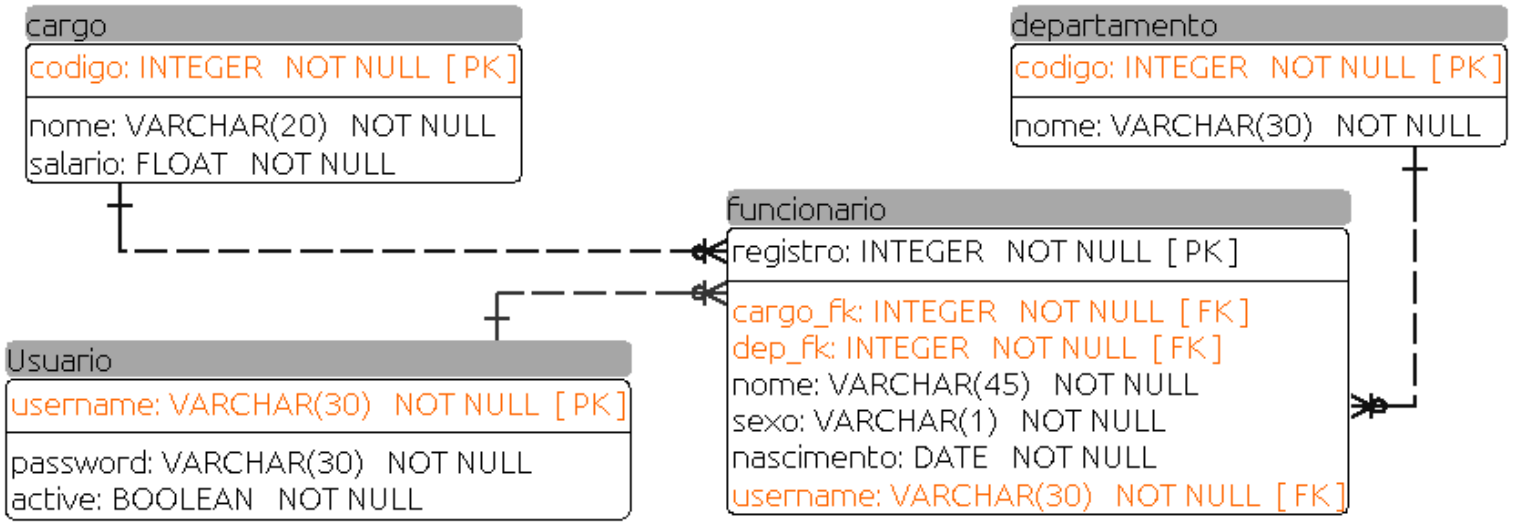
Considerar o seguinte cenário:



Temos um relacionamento 1:N entre as tabelas `Cargo` / `Funcionario` e `Departamento` / `Funcionario`.

- Usada em um relacionamento muitos-para-um (N-1).
- Essa anotação fica na **classe** do lado **N** em um **atributo** que mapeia uma tabela do lado **1**.
- Indica que a entidade **N** consegue acessar a entidade do lado **1**.

Anotações de Mapeamento - `@ManyToOne`



A classe de mapeamento da tabela `Funcionario` fica assim:

```
@Entity
public class Funcionario implements Serializable {
  ...
  @ManyToOne
  @JoinColumn(name="CARGO_FK") //define a FK
  private Cargo cargo;
  ...
```

Anotação de `@OneToMany`

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Usada em um relacionamento um-para-muitos (1-N).
- Essa anotação fica na **classe** do lado **1** em um **atributo** que mapeia uma tabela do lado **N**.
- Indica que a entidade **1** consegue acessar a entidade do lado **N**.

Exemplo:

```
@Entity
public class Cargo implements Serializable {
  ...
  @OneToMany(mappedBy="cargo") //o fetch padrão é LAZY
  private Set<Funcionario> funcionarios;
```

O atributo **mappedBy** indica o atributo `cargo` definido na classe `Funcionario` (lado **N**). Lá no atributo `cargo` está definida a chave estrangeira.

Prática 5.1.1 - 1:N - Salvando objetos

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Atualizar as tabelas:

```
DELETE FROM FUNCIONARIO; --para facilitar a criação da FK
ALTER TABLE FUNCIONARIO ADD COLUMN CARGO_FK INTEGER NOT NULL;

ALTER TABLE FUNCIONARIO ADD CONSTRAINT FUNCIONARIO_CARGO_FK
FOREIGN KEY (CARGO_FK) REFERENCES POSITION (CODE);
```

Atualizar a classe `Funcionario`:

```
@ManyToOne //o fetch padrão é EAGER
@JoinColumn(name="CARGO_FK")
private Cargo cargo; //criar o set e get
```

Atualizar a classe `Cargo`:

```
@OneToMany(mappedBy="cargo") //o padrão é LAZY
private Set<Funcionario> funcionarios; //criar o set e get
```

Prática 5.1.2 - 1:N - Salvando objetos

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Testar o main para inserir um cargo (Prática 2).
- Testar o main para inserir um funcionário (Prática 4).
- Implementar o main para salvar um `Funcionario`:

```
//buscar o cargo
//criar um objeto usuário / ou buscar um usuário existente
//criar um objeto funcionário
//setar o usuário dentro do funcionário
  objFuncionario.setUsuario(objUsuario);
//setar o cargo dentro do funcionário
  objFuncionario.setCargo(objCargo);
//dentro de apenas uma transaction
  //persistir o usuário e
  //persistir o funcionario
```

Prática 5.2 - 1:N - Consultando objetos

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Implementar o main para buscar um `Funcionario`:

```
// buscar o funcionário - usar find
//testar com fetch default
//imprimir o funcionário
//obter/buscar o cargo deste funcionário
//imprimir o cargo
```

- Executar o main e verificar os selects executados no banco.
- Incluir o `fetch=FetchType.LAZY` na anotação `ManyToOne` no atributo `cargo` da classe `Funcionario`.
- Executar o main novamente e verificar os selects executados no banco.

Prática 5.3 - 1:N - Consultando objetos

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

- Implementar o main para buscar um `Cargo`:

```
//buscar o cargo
//imprimir o cargo
//obter os funcionários daquele cargo
//imprimir os funcionários
```

- Executar o main e verificar os selects executados no banco.
- Incluir o `fetch=FetchType.EAGER` na anotação `OneToMany` no atributo `funcionarios` da classe `Cargo`.
- Executar o main novamente e verificar os selects executados no banco.

Prática 5.4 - 1:N - Exercício Proposto

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Executar as práticas anteriores para fazer o mapeamento e testar o relacionamento entre as classes `Funcionario` e `Department`.

Prática 6.1 - Consultas "orientadas a objeto" (funcionário X usuário)

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

A SQL nativa para trazer todos os funcionários ativos é:

```
select f.* from Funcionario f
join Usuario u on u.username = f.usuario_fk
where u.active = true
```

Uma consulta **não nativa** equivalente seria:

```
select f from Funcionario f where f.usuario.active = true
```

Repare que nesta consulta usa classes, objetos e atributos.

- Implementar uma consulta usando `TypedQuery` para consultar uma lista de `Funcionario`s.
- Analise o *select* nativo que o *framework* executou. Repare no *join* gerado devido a este fragmento da consulta: `f.usuario`

Prática 6.2 - Consultas “orientadas a objeto” (funcionário X usuário X cargo)

UNIVAS
UNIVERSIDADE DO VALE DO SAPUCAÍ

Parte 1: consultar os funcionarios ativos do cargo “QA”

- Inserir pelo menos um funcionario ativo “QA” (caso necessáriio)
- Testar a seguinte consulta (setar o filtro adequadamente):

```
select f from Funcionario f
where f.usuario.active = true AND f.cargo.nome = :nome_cargo
```

Parte 2: consultar os cargos que possuem funcionários ativos

- Testar a seguinte consulta[4]:

```
select distinct f.cargo from Funcionario f
where f.usuario.active = true
```

Analise o *select* nativo que o *framework* executou.

[4] a consulta “não-nativa", não é permitido navegar do lado muitos (N) para o lado 1

Referências

http://www.hibernate.org/docs

https://goalkicker.com/HibernateBook/

http://en.wikibooks.org/wiki/Java_Persistence

http://www.objectdb.com/java/jpa

http://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL

Fim