

## Lab 5 – JPA e Hibernate

Neste laboratório iremos mudar a forma com que é persistido os dados, trocaremos o JDBC pelo JPA/Hibernate. Modificaremos somente os Projetos Banco-Modelo fazendo o mapeamento objeto relacional e modificaremos também o projeto Banco-Service fazendo com que este faça todas as operações necessárias utilizando JPA/Hibernate.

### Exercícios

**Exercício 1:** Adicionar JPA e Hibernate a aplicação

**Exercício 2:** Realizar o mapeamento de objeto relacional

**Exercício 3:** Relacionamento OneToOne de entidades

**Exercício 4:** Relacionamento OneToMany e ManyToOne

**Exercício 5:** Relacionamento ManyToMany

**Exercício 6 –** Consultas Tipadas com Criteria

### Exercício 1 – Adicionar JPA e Hibernate a aplicação

1. Para adicionar o JPA e o Hibernate adicione as seguintes dependências no pom.xml do projeto.

```
<!-- JPA Hibernate -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>4.3.11.Final</version>
</dependency>
```

2. Agora vamos adicionar a configuração do JPA no projeto. Crie na pasta **src/main/resources** → **META-INF** o arquivo **persistence.xml** e escreva o seguinte código.

```
<persistence 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"
version="2.1">
  <persistence-unit name="pu-testes" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>

      <property name="javax.persistence.jdbc.url"
        value="jdbc:postgresql://localhost:5432/triway"/>
      <property name="javax.persistence.jdbc.user" value="postgres" />
      <property name="javax.persistence.jdbc.password" value="123456" />
      <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />

      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="validate" />

    </properties>
  </persistence-unit>
</persistence>
```

```
<!-- Configuring Connection Pool -->
<property name="hibernate.c3p0.min_size" value="5" />
<property name="hibernate.c3p0.max_size" value="20" />
<property name="hibernate.c3p0.timeout" value="500" />
<property name="hibernate.c3p0.max_statements" value="50" />
<property name="hibernate.c3p0.idle_test_period" value="2000" />
</properties>
</persistence-unit>
</persistence>
```

Note que o **URL**, o **user** e o **password** escritos acima são os mesmos que você usa para acessar o seu banco de dados. Assim, seu projeto esta setado para usar JPA Hibernate, vamos ver alguns exemplos agora.

## Exercício 2 – Realizar o mapeamento de objeto relacional

1. Vamos criar um módulo de armazenamento de endereços. Primeiro, crie a tabela no seu banco de dados (no caso, o Postgres).

```
CREATE TABLE endereco
(
  id serial NOT NULL,
  rua character varying(32) NOT NULL,
  cidade character varying(32) NOT NULL,
  estado character varying(32) NOT NULL,
  pais character varying(32) NOT NULL,
  cep character varying(10) NOT NULL
)
```

2. Agora, vamos criar uma entidade **Endereço** para fazer o mapeamento das informações a serem recebidas.

```
import javax.persistence.*;

@Entity
@Table(name="endereco")
public class Endereço {

    @Id
    @GeneratedValue
    private Integer id;

    private String rua;
    private String cidade;
    private String estado;
    private String pais;
    private String cep;

    //getters e setters omitidos
}
```

3. Para iniciar e operar o JPA, precisamos de um objeto **PersistenceManager**, usaremos um enum para termos uma implementação do padrão SINGLETON, assim garantimos que teremos um única instância deste objeto para cada requisição.

```
import javax.persistence.*;

public enum PersistenceManager {
    INSTANCE;

    private EntityManagerFactory emFactory;
```

```

private PersistenceManager(){
    emFactory = Persistence.createEntityManagerFactory("pu-testes");
}

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

public void close(){
    emFactory.close();
}
}

```

4. Vamos agora testar se nossa entidade **Endereco** está funcionando corretamente e se o JPA está enviando os dados adquiridos para o banco de dados. Crie uma classe **Main** para testar a entidade, assim como no código abaixo.

```

import javax.persistence.EntityManager;

public class Main {

    public static void main(String[] args){
        Endereco endereco = new Endereco();
        endereco.setCidade("Goiânia");
        endereco.setPais("Brazil");
        endereco.setEstado("Goiás");
        endereco.setCep("74000-000");
        endereco.setRua("Quarta Radial");

        EntityManager em = PersistenceManager.INSTANCE.getEntityManager();
        em.getTransaction().begin();
        em.persist(endereco);
        em.getTransaction().commit();

        em.close();
        PersistenceManager.INSTANCE.close();
    }
}

```

Teste a classe **Main** como uma Java Application e verifique se está funcionando. No console o Hibernate irá iniciar e aparecerá a seguinte mensagem

```

Hibernate:
insert
into
    adress
(cep, cidade, estado, pais, rua)
values
    (?, ?, ?, ?, ?)

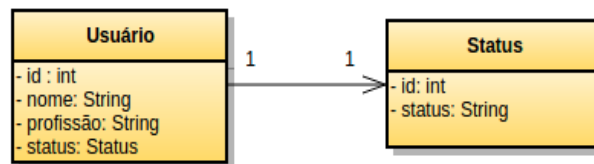
```

Aparecendo essa mensagem e nenhuma mensagem de erro, vá ao Postgres e veja se sua tabela foi atualizada.

	id serial	rua character varyin	cidade character	estado character	pais character va	cep character
1	3	Poribagh	Dhaka	Bongaon	Bangladesh	1000
2	4	Poribagh	Dhaka	Bongaon	Bangladesh	1000
3	5	Quarta Radial	Goiânia	Goiás	Brazil	00000-000

### Exercício 3 – Relacionamento OneToOne de entidades

1. Agora faremos um relacionamento de um-para-um entre entidades, o que pode ser feito através do JPA Hibernate. Neste exemplo definimos que um **Usuário** possui um **Status**, então desta forma o Usuário sabe qual é seu status, mas o contrario não existe, o Status não tem a necessidade de conhecer qual o Usuário que está associado a ele, ou seja, temos um relacionamento unidirecional.



2. Antes, crie as tabelas **usuario** e **status** no Postgres para fazer a associação JPA.

```
CREATE TABLE usuario(  
    id serial NOT NULL,  
    nome character varying(20) NOT NULL,  
    profissao character varying(20) NOT NULL,  
    status_id integer,  
    CONSTRAINT usuario_pkey PRIMARY KEY (id)  
)
```

```
CREATE TABLE status (  
    id serial NOT NULL,  
    status character varying(10) NOT NULL,  
    CONSTRAINT status_pkey PRIMARY KEY (id)  
)
```

3. Crie os objetos **Usuário** e **Status** assim como mostrado no UML acima e no código abaixo.

Classe **Usuario**

```
package login;  
import javax.persistence.*;  
@Entity  
public class Usuario {
```

```

@Id
@GeneratedValue
private int id;

private String nome;
private String profissao;

@OneToOne(cascade=CascadeType.ALL)
private Status status;

// Getters e Setters omitidos
}

```

#### Classe Status

```

package login;

import javax.persistence.*;

@Entity
public class Status {

    @Id
    @GeneratedValue
    private int id;
    private String status;
    // Getters e Setters omitidos
}

```

4. Concluído esse passo, vamos criar a nossa classe **Login** que vai passar as informações para as entidades e consequentemente, para o nosso banco de dados criado. (Utilizar o **PersistenceManager** criado no exercício passado)

```

package login;

import javax.persistence.EntityManager;
import teste.PersistenceManager;

public class Login {
    public static void main(String[] args){

        Usuario usr = new Usuario();

        usr.setNome("João");
        usr.setProfissao("Programador");

        Status status = new Status();
        status.setStatus("Ativo");

        usr.setStatus(status);

        EntityManager em = PersistenceManager.INSTANCE.getEntityManager();

        em.getTransaction().begin();
        em.persist(usr);
        em.persist(status);
        em.getTransaction().commit();
        em.close();
        PersistenceManager.INSTANCE.close();
    }
}

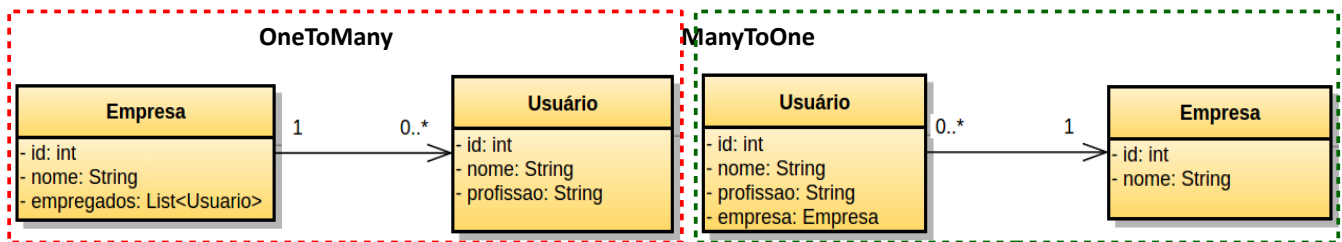
```

5. Execute a classe **Login** e verifique o resultado no seu console e no seu banco de dados.

usuario					status		
	id [PK] serial	nome character	profissao character varyin	status_id integer		id [PK] serial	status character
1	4	Anderson	Radialista	1	1	1	Inativo
2	5	Evandro	Chefe/Pajé	2	2	2	Inativo
3	6	Greice	Administradora	3	3	3	Ativo
4	7	João	Programador	4	4	4	Ativo

## Exercício 4 – Relacionamento OneToMany e ManyToOne

1. Agora faremos um relacionamento de **um-para-muitos** e **muitos-para-um** entre entidades, o que pode ser feito através do JPA Hibernate. Neste exemplo definiremos que uma **Empresa** que possui vários **Usuários**, então desta forma o Usuário sabe qual é sua empresa, mas o contrario não existe, a empresa não tem a necessidade de conhecer qual o Usuário que está associado a ele, ou seja, temos um relacionamento unidirecional que pode funcionar de ambos os lados.



2. Altere sua tabela **Usuário** e crie uma tabela **Empresa** com id e nome da empresa, assim como mostrado no código abaixo.

```
CREATE TABLE usuario(
    id serial NOT NULL,
    nome character varying(20) NOT NULL,
    profissao character varying(20) NOT NULL,
    empresa_id integer,
    CONSTRAINT usuario_pkey PRIMARY KEY (id)
)
```

```
CREATE TABLE empresa(
    id serial NOT NULL,
    nome character varying(20) NOT NULL,
    CONSTRAINT empresa_pkey PRIMARY KEY (id)
)
```

3. Agora, altere sua classe **Usuário** assim como mostrado na UML **OneToMany**, onde será feita a associação do lado da classe **Empresa**. Segue o código das classes a seguir, preste muita atenção nos detalhes e anotações.

```
@Entity
public class Usuario {

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

    // Getters e Setters omitidos

}

@Entity
public class Empresa {
    @Id
    @GeneratedValue
    private int id;
    private String nome_empresa;

    @OneToMany(cascade= CascadeType.ALL)
    @JoinColumn(name="empresa_id")
    private List<Usuario> empregados;

    // Getters e Setters omitidos

}
```

4. Agora vamos fazer o teste. Crie a classe **Cadastro.java** que insira na tabela **empresa** o nome da empresa e a lista de empregados; e na tabela **usuario**, insira o nome e a profissão do empregado. Segue o código abaixo:

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

        Empresa empresa = new Empresa();

        empresa.setNome_empresa("3way Networks");

        Usuario usr1 = new Usuario();

        usr1.setNome("João");
        usr1.setProfissao("Programador");

        Usuario usr2 = new Usuario();

        usr2.setNome("Mário");
        usr2.setProfissao("Técnico");

        List<Usuario> empregados = new ArrayList<Usuario>();

        empregados.add(usr1);
        empregados.add(usr2);
        empresa.setEmpregados(empregados);

        EntityManager em = PersistenceManager.INSTANCE.getEntityManager();

        em.getTransaction().begin();
        em.persist(empresa);
    }
}
```

```


        em.persist(usr1);
        em.persist(usr2);
        em.getTransaction().commit();
        em.close();
        PersistenceManager.INSTANCE.close();
    }
}

```

5. Execute o código acima e veja o resultado na sua tabela. Deverá parecer como a imagem abaixo.

	id [PK] serial	nome_empresa character varying(
1	1	3way Networks
2	2	Aperture Science

	id [PK] serial	nome character varyi	profissao character vary	empresa_id integer
1	1	Enesbaldo	Empresário	1
2	2	Josismar	Programador	1
3	3	Caroline	Assistente	2
4	4	Cave Jhonson	Fundador	2



6. Agora, vamos aprender a fazer o caminho de volta, onde você vai de muitos-para-um. Vamos alterar só alguns componentes dos códigos anteriores. Volte a classe Usuário e Empresa e faça a seguinte alteração.

```

@Entity
public class Usuario {

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

    @ManyToOne(cascade=CascadeType.ALL)
    private Empresa empresa;

    // Getters e Setters omitidos
}

@Entity
public class Empresa {
    @Id
    @GeneratedValue
    private int id;
    private String nome_empresa;

    // Getters e Setters omitidos
}

```

7. Volte a classe **Cadastro** e, ao invés de adicionar empregados à empresa, agora você vai adicionar a empresa nas informações do usuário. Preste bem atenção no código abaixo e altere os usuários e a empresa para ver diferenças.

```

public class Cadastro {
    public static void main(String[] args){

```



```
Empresa empresa = new Empresa();

empresa.setNome_empresa("3way Networks");

Usuario usr1 = new Usuario();

usr1.setNome("João");
usr1.setProfissao("Programador");
usr1.setEmpresa(empresa);

Usuario usr2 = new Usuario();

usr2.setNome("Mário");
usr2.setProfissao("Técnico");
usr2.setEmpresa(empresa);

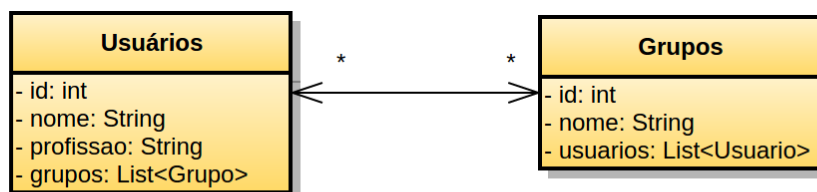
EntityManager em = PersistenceManager.INSTANCE.getEntityManager();

em.getTransaction().begin();
em.persist(empresa);
em.persist(usr1);
em.persist(usr2);
em.getTransaction().commit();
em.close();
PersistenceManager.INSTANCE.close();
}
}
```

Execute o código acima e veja o resultado na sua tabela. Perceba que o resultado será o mesmo do exemplo **OneToMany**. Como foi explicado, são formas diferentes de fazer a mesma associação de tabelas.

## Exercício 5 – Relacionamento ManyToMany

1. Em Hibernate, podemos fazer a associação de muitos-para-muitos também. No nosso exemplo, temos muitos **Usuários** e cada um deles podem participar de vários **Grupos**, sendo que cada grupo também pode conter vários usuários, sendo composta de uma associação **ManyToMany** entre tabelas. Observe o UML abaixo para a criação das classes **Usuário** e **Grupo**.



2. Antes de começar a criar as classes, vamos criar nossas tabelas no postgres. Podemos reutilizar a tabela de usuarios, mas precisamos criar uma nova tabela para os **grupos** e uma outra tabela para fazer essa associação entre grupos e usuarios. Vamos chamá-la de `grupo_usuario`, assim como mostrado no código abaixo.

```
CREATE TABLE usuario(  
    id serial NOT NULL,  
    nome character varying(20) NOT NULL,  
    profissao character varying(20) NOT NULL,  
    CONSTRAINT usuario_pkey PRIMARY KEY (id)  
)
```

```
CREATE TABLE grupo(  
    id serial NOT NULL,  
    nome character varying(20) NOT NULL,  
    CONSTRAINT grupo_pkey PRIMARY KEY (id)  
)
```

```
CREATE TABLE grupo_usuario(  
    grupo_id integer,  
    usuario_id integer,  
)
```

3. Reutilizando a mesma classe **Usuário** dos exercícios anteriores, vamos dar somente uma alteração em sua estrutura. Note também que na anotação **ManyToMany**, você precisa especificar o lado principal dessa associação, como no caso, usamos a tabela **Grupo** como mandante. Segue o código abaixo. Preste bastante atenção na anotação **@ManyToMany** e **@JoinTable** porque são muito importantes e qualquer erro, pode bagunçar toda sua tabela.

```
@Entity  
public class Grupo {  
    @Id  
    @GeneratedValue  
    private int id;  
    private String nome;  
    @ManyToMany  
    @JoinTable(name="grupo_usuario",  
        joinColumns=@JoinColumn(name="grupo_id"),  
        inverseJoinColumns=@JoinColumn(name="usuario_id"))  
    private List<Usuario> usuarios;
```

```
// Getters e Setters omitidos

}

@Entity
public class Usuario {

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

    @ManyToMany(mappedBy="usuarios")
    private List<Grupo> grupos;

    // Getters e Setters omitidos

}
```

3. Agora, vamos testar essas associações adicionando grupos e usuarios em sua tabela. Crie a classe **VerGrupos** assim como mostrado abaixo.

```
public class VerGrupos {

    public static void main(String[] args) {

        Usuario usr1 = new Usuario();

        usr1.setNome("João");
        usr1.setProfissao("Entregador");

        Usuario usr2 = new Usuario();

        usr2.setNome("José");
        usr2.setProfissao("Chefe");

        Usuario usr3 = new Usuario();

        usr3.setNome("Mário");
        usr3.setProfissao("Dono");

        List<Usuario> grupoA = new ArrayList<Usuario>();
        grupoA.add(usr2);
        grupoA.add(usr3);

        List<Usuario> grupoTot = new ArrayList<Usuario>();
        grupoTot.add(usr1);
        grupoTot.add(usr2);
        grupoTot.add(usr3);

        List<Usuario> grupoB = new ArrayList<Usuario>();
        grupoB.add(usr1);
        grupoB.add(usr2);

        Grupo g1 = new Grupo();
        g1.setNome("Admin");

        Grupo g2 = new Grupo();
        g2.setNome("Todos");

        Grupo g3 = new Grupo();
```

```

g3.setNome("Entregas");

List<Grupo> usuario1 = new ArrayList<Grupo>();
usuario1.add(g2);
usuario1.add(g3);

List<Grupo> usuario2 = new ArrayList<Grupo>();
usuario2.add(g1);
usuario2.add(g2);
usuario2.add(g3);

List<Grupo> usuario3 = new ArrayList<Grupo>();
usuario3.add(g1);
usuario3.add(g2);

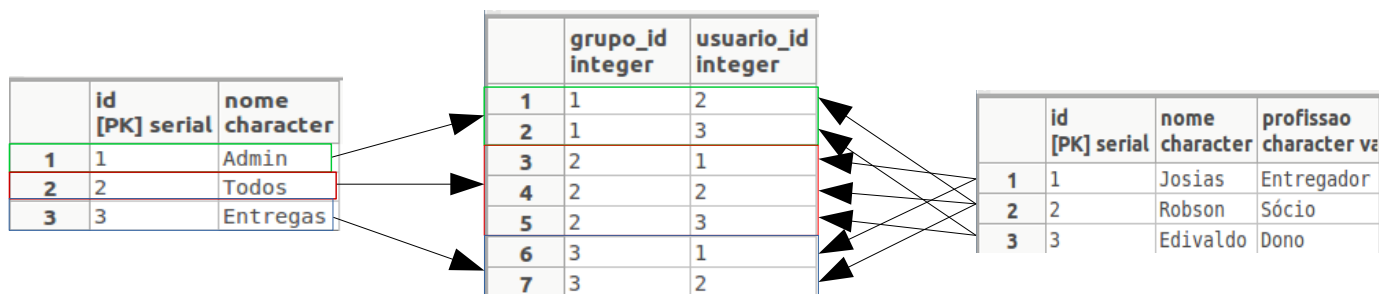
usr1.setGrupos(usuario1);
usr2.setGrupos(usuario2);
usr3.setGrupos(usuario3);

EntityManager em = PersistenceManager.INSTANCE.getEntityManager();
em.getTransaction().begin();
em.persist(usr1);
em.persist(usr2);
em.persist(usr3);
em.persist(g1);
em.persist(g2);
em.persist(g3);
em.getTransaction().commit();

em.close();
PersistenceManager.INSTANCE.close();
}
}

```

4. Execute o programa e verifique o resultado dessa associação no seu banco de dados.



## Exercício 6 – Consultas Tipadas com Criteria

1. Tendo as tabelas e as entidades, vamos ver agora um método de consulta de tabelas. Para isso, vamos usar a API Criteria, que possui a interface `org.hibernate.Criteria` disponível com todos os métodos fornecidas pela API. Através dessa API podemos construir consultas de forma programática. Para testá-la, vamos criar uma classe **UsuarioDAO** e fazer uso do `CriteriaBuilder` como mostrado abaixo.

```
public class UsuarioDAO {

    @PersistenceContext
    private EntityManager em;

    public void listar() {

        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Usuario> q = cb.createQuery(Usuario.class);
        Root<Usuario> root = q.from(Usuario.class);
        TypedQuery<Usuario> query = em.createQuery(q);
        List<Usuario> usuarios = query.getResultList();

        usuarios.forEach(System.out::println);
    }
}
```

2. Conseguimos recuperar os usuários que temos no banco de dados, mas vamos além. Agora vamos pegar somente a profissão de cada usuário. Para isso, é necessário um `Path`, um caminho sinalizando de onde queremos essa informação. Faça as alterações no seu código como mostrado abaixo.

```
Root<Usuario> root = q.from(Usuario.class);
Path<String> path = root.get("profissao");
q.select(path);
TypedQuery<String> query = em.createQuery(q);
List<String> usuarios = query.getResultList();
usuarios.forEach(System.out::println);
```

3. Com o Criteria, podemos fazer uma pesquisa pelo banco, buscando somente aquela informação que você deseja, equivalente à expressão **SELECT user FROM Usuario user WHERE user.id = p**. Segue o código abaixo, faça o teste alterando os valores de `id`.

```
CriteriaQuery<String> q = cb.createQuery(String.class);
Root<Usuario> user = q.from(Usuario.class);
Path<String> path = user.get("nome");
ParameterExpression<Integer> p = cb.parameter(Integer.class);
q.select(path).where(cb.equal(user.get("id"), p));
TypedQuery<String> query = em.createQuery(q);
query.setParameter(p, 2);
String usuario = query.getSingleResult();
```

```
System.out.println(usuario);
```

4. Assim como pesquisar o usuario correspondente ao id informado, você também tem a liberdade de fazer outras formas de pesquisa, como:

- `q.select(path).where(cb.gt(user.get("id"), p));` → id's maiores que p
- `q.select(path).where(cb.lt(user.get("id"), p));` → id's menores que p
- `q.select(user).where(cb.like(user.get("nome"), "%J%"));` → nomes que tem J

Faça alguns testes e veja os resultados. Todas as formas de filtrar uma consulta pode ser feita com o Criteria.

5. O Criteria também nos oferece a possibilidade de filtrar fazendo um **Join** entre tabelas, como veremos a seguir. Vamos recorrer a nossa tabela de usuario e de status para fazer esse exercício. Veja o código abaixo e faça as alterações necessárias.

```
Root<Usuario> user = q.from(Usuario.class);  
q.select(user).where(cb.equal(user.join("status").get("id"), p));  
TypedQuery<Usuario> query = em.createQuery(q);  
List<Usuario> usuarios = query.getResultList();  
System.out.println(usuarios);
```