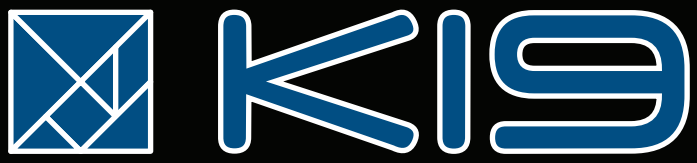


TREINAMENTOS

Persistência com JPA2 e Hibernate

Persistência com JPA 2 e Hibernate

14 de março de 2011

Sumário

Capítulo 1

Introdução

1.1 Persistência

Aplicações corporativas manipulam dados em grande quantidade. Na maioria dos casos, esses dados são armazenados em banco de dados relacionais pois os principais sistema gerenciadores de banco de dados do mercado utilizam o modelo relacional. Por outro lado, hoje em dia, as aplicações corporativas costumam ser desenvolvidas com linguagens orientadas a objetos.

Como o modelo relacional e o modelo orientado a objetos diferem no modo de estruturar os dados, uma transformação deve ocorrer toda vez que alguma informação trafegar da aplicação para o banco de dados ou vice-versa. Essa transformação não é simples pois os dois modelos são bem diferentes.

No contexto das aplicações Java, para facilitar o processo de transformação dos dados que trafegam entre as aplicações e os banco de dados, podemos utilizar algumas ferramentas de persistência como o **Hibernate** ou o **TopLink**.

Essas ferramentas funcionam como intermediários entre as aplicações e os banco de dados, automatizando diversos processos importantes relacionados à persistência dos dados. Elas são chamadas de ferramentas ORM (Object Relational Mapping)

Com o intuito de facilitar a utilização dessas ferramentas e torná-las compatíveis com os outros recursos da plataforma Java, elas são padronizadas pela especificação **Java Persistence API (JPA)**.

Veremos nesse capítulo, os passos principais para utilizar uma implementação da JPA. No caso, utilizaremos o Hibernate e o banco de dados MySQL.

1.2 Configuração

Antes de começar a utilizar o Hibernate, é necessário baixar do site oficial o **bundle** que inclui os jar's do hibernate e todas as suas dependências. Neste curso, utilizaremos a versão 3.5.1. A url do site oficial do Hibernate é `http://www.hibernate.org/`.

Veja também um artigo da K19 sobre configuração do Hibernate no seguinte endereço `http://www.k19.com.br/artigos/configurando-hibernate-com-mysql/`.

Para configurar o Hibernate em uma aplicação, devemos criar um arquivo chamado **persistence.xml**. O conteúdo desse arquivo contém informações sobre o banco de dados, como a url

de conexão, usuário e senha. Além de dados sobre a implementação de JPA que será utilizada.

O arquivo PERSISTENCE.XML deve ser salvo em uma pasta chamada **META-INF**, que deve estar no classpath da aplicação. Veja abaixo um exemplo de configuração para o PERSISTENCE.XML:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/↩
           ns/persistence/persistence_1_0.xsd"
5      version="1.0">
6
7      <persistence-unit name="K19" transaction-type="RESOURCE_LOCAL">
8          <provider>org.hibernate.ejb.HibernatePersistence</provider>
9          <properties>
10             <property name="hibernate.dialect" value="org.hibernate.dialect.↩
                   MySQL5InnoDBDialect"/>
11             <property name="hibernate.hbm2ddl.auto" value="update"/>
12             <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver↩
                   "/>
13             <property name="javax.persistence.jdbc.user" value="usuario"/>
14             <property name="javax.persistence.jdbc.password" value="senha"/>
15             <property name="javax.persistence.jdbc.url" value="jdbc:mysql://↩
                   localhost:3306/k19"/>
16         </properties>
17     </persistence-unit>
18
19 </persistence>
```

1.3 Mapeamento

Um dos principais objetivos dos frameworks ORM é estabelecer o mapeamento entre os conceitos do modelo orientado a objetos e os conceitos do modelo entidade relacionamento. Este mapeamento pode ser definido através de xml ou de maneira mais prática com anotações Java. Quando utilizamos anotações, evitamos a criação de extensos arquivos em xml.

A seguir veremos as principais anotações Java de mapeamento do JPA. Essas anotações estão no pacote **javax.persistence**.

@Entity É a principal anotação do JPA. Ela que deve aparecer antes do nome de uma classe. E deve ser definida em todas as classes que terão objetos persistidos no banco de dados.

As classes anotadas com @ENTITY são mapeadas para tabelas. Por convenção, as tabelas possuem os mesmos nomes das classes. Mas, podemos alterar esse comportamento utilizando a anotação @TABLE.

Os atributos declarados em uma classe anotada com @ENTITY são mapeados para colunas na tabela correspondente à classe. Outra vez, por convenção, as colunas possuem os mesmos nomes dos atributos. E novamente, podemos alterar esse padrão utilizando para isso a anotação @COLUMN.

@Id Utilizada para indicar qual atributo de uma classe anotada com @ENTITY será mapeado para a chave primária da tabela correspondente à classe. Geralmente o atributo anotado com @ID é do tipo LONG.

@GeneratedValue Geralmente vem acompanhado da anotação @ID. Serve para indicar que o valor de um atributo que compõe uma chave primária deve ser gerado pelo banco no momento em que um novo registro é inserido.

1.4 Gerando o banco

Uma das vantagens de utilizar o Hibernate, é que ele é capaz de gerar as tabelas no banco de dados. Ele faz isso de acordo com as anotações colocadas nas classes e as informações presentes no PERSISTENCE.XML.

As tabelas são geradas através de método da classe PERSISTENCE, o CREATEENTITYMANAGERFACTORY(STRING PERSISTENCEUNIT). O parâmetro PERSISTENCEUNIT permite escolher, pelo nome, uma unidade de persistência definida no PERSISTENCE.XML.

```
1 Persistence.createEntityManagerFactory("K19");
```

1.5 Exercícios

1. Crie um projeto no eclipse chamado **JPA2-Hibernate**.

2. Crie uma pasta chamada **lib** dentro do projeto **JPA2-Hibernate**.

3. Entre na pasta **K19-Arquivos/Hibernate** da Área de Trabalho e copie os jar's do Hibernate para a pasta **lib** do projeto **JPA2-Hibernate**.

4. Entre na pasta **K19-Arquivos/MySQL-Connector-JDBC** da Área de Trabalho e copie o arquivo MYSQL-CONNECTOR-JAVA-5.1.13.BIN.JAR para pasta **lib** do projeto **JPA2-Hibernate**.

5. Entre na pasta **K19-Arquivos/SLF4J** da Área de Trabalho e copie os jar's para pasta **lib** do projeto **JPA2-Hibernate**.

6. Entre na pasta **K19-Arquivos/Log4J** da Área de Trabalho e copie o arquivo LOG4J-1.2.16.JAR para pasta **lib** do projeto **JPA2-Hibernate**.

7. Adicione os jar's da pasta **lib** ao **build path** do projeto **JPA2-Hibernate**. Peça orientação do instrutor se for necessário.

8. Crie uma pasta chamada **META-INF** dentro da pasta **src** do projeto **JPA2-Hibernate**.

9. Crie o arquivo de configurações **persistence.xml** na pasta **META-INF**. Para não ter que digitar todo o código copie o modelo **persistence.xml** da pasta **K19-Arquivos/modelos** da sua Área de Trabalho.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <persistence-unit name="K21_livraria" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <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="root"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/K21_livraria"/>
        </properties>
    </persistence-unit>
</persistence>
```

10. Crie uma classe para modelar as editoras da nossa livraria e acrescente as anotações necessárias para fazer o mapeamento. Obs: As anotações devem ser importadas do pacote JAVAX.PERSISTENCE.

```java
@Entity
public class Editora {
    @Id @GeneratedValue
    private Long id;

    private String nome;

    private String email;

    // GETTERS AND SETTERS
}
```

11. Apague a base dados **K21_livraria** se ela existir através do MySQL Query Browser. Peça orientação do instrutor se for necessário.

12. Crie a base dados **K21_livraria** através do MySQL Query Browser. Peça orientação do instrutor se for necessário.

13. Configure o Log4J criando um arquivo chamado **log4j.properties** na pasta **src** do projeto **JPA2-Hibernate**.

```
log4j.rootCategory=INFO, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%r [%t] %-5p %c - %m%n
```

14. Gere as tabelas através da classe PERSISTENCE. Para isso, crie uma classe com método MAIN. Obs: As classes devem ser importadas do pacote JAVAX.PERSISTENCE.

```
1 public class GeraTabelas {
2     public static void main(String[] args) {
3         EntityManagerFactory factory =
4             Persistence.createEntityManagerFactory("livraria");
5
6         factory.close()
7     }
8 }
```

Através do MySQL Query Browser verifique se a tabela EDITORA foi criada corretamente.

1.6 Manipulando entidades

Para manipular as entidades da nossa aplicação, devemos utilizar um ENTITYMANAGER que é obtido através de uma ENTITYMANAGERFACTORY.

```
1 EntityManagerFactory factory =
2     Persistence.createEntityManagerFactory("K19");
3
4 EntityManager manager = factory.createEntityManager();
```

1.6.1 Persistindo

Para armazenar as informações de um objeto no banco de dados basta utilizar o método PERSIST() do ENTITYMANAGER.

```
1 Editora novaEditora = new Editora();
2 novaEditora.setNome("K19 - Livros")
3 novaEditora.setEmail("contato@k19.com.br");
4
5 manager.persist(novaEditora);
```

1.6.2 Buscando

Para obter um objeto que contenha informações do banco de dados basta utilizar o método FIND() ou o GETREFERENCE() do ENTITYMANAGER.

```
1 Editora editora1 = manager.find(Editora.class, 1L);
2 Editora editora2 = manager.getReference(Editora.class, 2L);
```

A diferença entre os dois métodos básicos de busca FIND() e GETREFERENCE() é que o primeiro recupera os dados desejados imediatamente já o segundo posterga até a primeira chamada de um método GET do objeto.

1.6.3 Removendo

Para remover um registro correspondente a um objeto basta utilizar o método REMOVE() do ENTITYMANAGER.

```
1 Editora editora1 = manager.find(Editora.class, 1L);
2 manager.remove(editora1);
```

1.6.4 Atualizando

Para alterar os dados de um registro correspondente a um objeto basta utilizar os próprios métodos setters desse objeto.

```
1 Editora editora1 = manager.find(Editora.class, 1L);
2 editora.setNome("K19 - Livros e Publicações");
```

1.6.5 Listando

Para obter uma listagem com todos os objetos referentes aos registros de uma tabela, devemos utilizar a linguagem de consulta do JPA, a **JPQL** que é muito parecida com a linguagem SQL. A vantagem do JPQL em relação ao SQL é que a sintaxe é a mesma para bancos de dados diferentes.

```
1 Query query = manager.createQuery("SELECT e FROM Editora e");
2 List<Editora> editoras = query.getResultList();
```

1.6.6 Transações

As modificações realizadas nos objetos administrados por um ENTITYMANAGER são mantidas em memória. Em certos momentos, é necessário sincronizar os dados da memória com os dados do banco de dados. Essa sincronização deve ser realizada através de uma transação JPA criada pelo ENTITYMANAGER que administra os objetos que desejamos sincronizar.

Para abrir uma transação utilizamos o método BEGIN().

```
1 manager.getTransaction().begin();
```

Com a transação aberta podemos sincronizar os dados com o banco através do método FLUSH() ou COMMIT().

```
1 Editora editora1 = manager.find(Editora.class, 1L);
2 editora.setNome("K19 - Livros e Publicações");
3
4 manager.getTransaction().begin();
5 manager.flush();
```

```
Editora editora1 = manager.find(Editora.class, 1L);
editora.setNome("K19 - Livros e Publicações");

manager.getTransaction().begin();
manager.getTransaction().commit();
```

1.7 Exercícios

15. Crie um teste para inserir editoras no banco de dados.

```
public class InsereEditoraComJPA {

    public static void main(String[] args) {
        EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("livraria");

        EntityManager manager = factory.createEntityManager();

        Editora novaEditora = new Editora();

        Scanner entrada = new Scanner(System.in);

        System.out.println("Digite o nome da editora: ");
        novaEditora.setNome(entrada.nextLine());

        System.out.println("Digite o email da editora: ");
        novaEditora.setEmail(entrada.nextLine());

        manager.persist(novaEditora);

        manager.getTransaction().begin();
        manager.getTransaction().commit();

        factory.close();
    }
}
```

16. Crie um teste para listar as editoras inseridas no banco de dados.

```
public class ListaEditorasComJPA {

    public static void main(String[] args) {
        EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("livraria");

        EntityManager manager = factory.createEntityManager();

        Query query = manager.createQuery("SELECT e FROM Editora e");
        List<Editora> editoras = query.getResultList();

        for(Editora e : editoras) {
            System.out.println("EDITORA: " + e.getNome() + " - " + e.getEmail());
        }
    }
}
```

Capítulo 2

Mapeamento

O mapeamento objeto relacional é o coração do Hibernate e das outras implementações de JPA pois ele define quais transformações devem ser realizadas nos dados para que essas informações possam navegar da aplicação para o banco de dados ou do banco de dados para a aplicação. Além disso, o mapeamento determina como a ferramenta ORM fará consultas complexas envolvendo mais do que uma tabela.

2.1 Entidades

As classes da nossa aplicação que devem ser mapeadas para tabelas do banco de dados são anotadas com **@Entity**. Cada instância de uma classe anotada com @ENTITY deve possuir um identificador único. Em geral, esse identificador é um atributo numérico que deve ser anotado com **@Id**.

```
1 @Entity
2 class Pessoa {
3     @Id
4     private Long id;
5 }
```

Por convenção, a classe PESSOA será mapeada para uma tabela com o mesmo nome (Pessoa). O atributo ID será mapeado para uma coluna com o mesmo nome (id) na tabela Pessoa. Podemos não seguir o comportamento padrão e aplicar nomes diferentes para as tabelas e colunas utilizando as anotações **@Table** e **@Column**.

A coluna correspondente ao atributo ID será a chave primária da tabela PESSOA por causa da anotação @ID.

```
1 @Entity
2 @Table(name = "tbl_pessoas")
3 class Pessoa {
4     @Id
5     @Column(name = "col_id")
6     private Long id;
7 }
```

2.2 Definindo Restrições

Podemos definir algumas restrições para os atributos das nossas entidades através das propriedades da anotação @COLUMN. Veja as principais propriedades abaixo:

length	Limita a quantidade de caracteres de um valor string
nullable	Determina se o campo pode possuir valores NULL ou não
unique	Determina se uma coluna pode ter valores repetidos ou não

```
1 @Entity
2 class Pessoa {
3     @Id
4     private Long id;
5
6     @Column(length=30,nullable=false,unique=true)
7     private String nome;
8 }
```

2.3 @GeneratedValue

Em geral, os bancos de dados oferecem algum mecanismo para gerar os valores de uma chave primária simples e numérica. Do ponto de vista do desenvolvedor JPA, para deixar com o banco de dados a responsabilidade de gerar os valores de uma chave primária simples e numérica, basta aplicar a anotação **@GeneratedValue**.

```
1 @Entity
2 class Pessoa {
3     @Id
4     @GeneratedValue
5     private Long id;
6 }
```

2.4 Mapeamento Automático

Alguns tipos do Java são mapeados automaticamente para tipos correspondentes do banco de dados tirando dos desenvolvedores esse trabalho. Eis uma listagem dos tipos que são mapeados automaticamente:

- Tipos primitivos (byte, short, char, int, long, float, double e boolean)
- Classes Wrappers (Byte, Short, Character, Integer, Long, Float, Double e Boolean)
- String
- BigInteger e BigDecimal
- java.util.Date e java.util.Calendar

- java.sql.Date, java.sql.Time e java.sql.Timestamp
- Array de byte ou char
- Enums
- Serializables

2.5 Large Objects (LOB)

Eventualmente, dados maiores do que o comum devem ser armazenados no banco de dados. Por exemplo, uma imagem, uma música ou um texto com muitas palavras. Para esses casos, os bancos de dados oferecem tipos de dados específicos. Do ponto de vista do desenvolvedor JPA, basta aplicar a anotação **@LOB** em atributos do tipo STRING, BYTE[], BYTE[], CHAR[] ou CHARACTER[] que o provedor (Hibernate, TopLink ou outra implementação de JPA) utilizará os procedimentos adequados para manipular esses dados.

```
1 @Entity
2 class Pessoa {
3     @Id
4     @GeneratedValue
5     private Long id;
6
7     @Lob
8     private byte[] avatar;
9 }
```

2.6 Data e Hora

Comumente, as aplicações Java utilizam as classes JAVA.UTIL.DATE e JAVA.UTIL.CALENDAR para trabalhar com datas e horas. Essas classes são mapeadas automaticamente para tipos adequados no banco de dados. Portanto, basta declarar os atributos utilizando um desses dois tipos nas classes que serão mapeadas para tabelas.

```
1 @Entity
2 class Pessoa {
3     @Id
4     @GeneratedValue
5     private Long id;
6
7     private Calendar nascimento;
8 }
```

Por padrão, quando aplicamos o tipo JAVA.UTIL.DATE ou JAVA.UTIL.CALENDAR, tanto data quanto hora serão armazenados no banco de dados. Para mudar esse comportamento, devemos aplicar a anotação **@Temporal** escolhendo uma das três opções abaixo:

TemporalType.DATE : Apenas data (dia, mês e ano).

TemporalType.TIME : Apenas horário (hora, minuto e segundo)

TemporalType.TIMESTAMP (Padrão): Data e Horário

```
1  @Entity
2  class Pessoa {
3      @Id
4      @GeneratedValue
5      private Long id;
6
7      @Temporal(TemporalType.DATE)
8      private Calendar nascimento;
9  }
```

2.7 Dados Transientes

Eventualmente, não desejamos que alguns atributos de um determinado grupo de objetos sejam persistidos no banco de dados. Nesse caso, devemos aplicar o modificador **transient** ou a anotação **@Transient**.

```
1  @Entity
2  class Pessoa {
3      @Id
4      @GeneratedValue
5      private Long id;
6
7      @Temporal(TemporalType.DATE)
8      private Calendar nascimento;
9
10     @Transient
11     private int idade;
12 }
```

2.8 Field Access e Property Access

Os provedores de JPA precisam ter acesso ao estado das entidades para poder administrá-las. Por exemplo, quando persistimos uma instância de uma entidade, o provedor deve "pegar" os dados desse objeto e armazená-los no banco. Quando buscamos uma instância de uma entidade, o provedor recupera as informações correspondentes do banco de dados e "guarda" em um objeto.

O JPA 2 define dois modos de acesso ao estado das instâncias das entidades: **Field Access** e **Property Access**. Quando colocamos as anotações de mapeamento nos atributos, estamos optando pelo modo Field Access. Quando colocamos as anotações de mapeamento nos métodos getters, estamos optando pelo modo Property Accesss.

No modo Field Access, os atributos dos objetos são acessados diretamente através de **reflection** e não é necessário implementar métodos getters e setters. No modo Property Access, os métodos getters e setters devem ser implementados pelo desenvolvedor e serão utilizados pelo provedor para que ele possa acessar e modificar o estado dos objetos.

2.9 Exercícios

1. Crie um projeto no eclipse chamado **Mapeamento**. Copie a pasta **lib** do projeto **JPA2-Hibernate** para o projeto **Mapeamento**. Depois selecione todos os jar's e os adicione no classpath.

2. Abra o **MySQL Query Browser** e apague a base de dados **K21_mapeamento** se existir. Depois crie a base de dados **K21_mapeamento**.

3. Copie a pasta **META-INF** do projeto **JPA2-Hibernate** para dentro da pasta **src** do projeto **Mapeamento**. Altere o arquivo **persistence.xml** do projeto **Mapeamento**, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <persistence-unit name="K21_mapeamento" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <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="root"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/K21_mapeamento"/>
        </properties>
    </persistence-unit>
</persistence>
```

4. Crie uma entidade para modelar os usuários de uma rede social dentro de um pacote chamado **modelo**.

```
@Entity
public class Usuario {
    @Id
    @GeneratedValue
    private Long id;

    @Column(unique=true)
    private String email;

    @Temporal(TemporalType.DATE)
    private Calendar dataDeCadastro;

    @Lob
    private byte[] foto;

    // GETTERS AND SETTERS
}
```

5. Adicione um usuário no banco de dados através do seguinte teste.

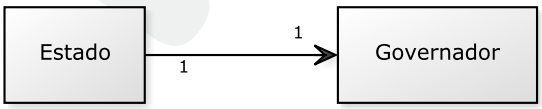
```
1  public class AdicionaUsuario {
2      public static void main(String[] args) {
3          EntityManagerFactory factory =
4                  Persistence.createEntityManagerFactory("K21_mapeamento");
5          EntityManager manager = factory.createEntityManager();
6
7          manager.getTransaction().begin();
8
9          Usuario usuario = new Usuario();
10         usuario.setEmail("contato@k19.com.br");
11         usuario.setDataDeCadastro(Calendar.getInstance());
12
13         manager.persist(usuario);
14
15         manager.getTransaction().commit();
16
17         manager.close();
18         factory.close();
19     }
20 }
```

6. Abra o MySQL Query Browser e observe as propriedades da tabela **Usuario** da base de dados **K21_mapeamento**.

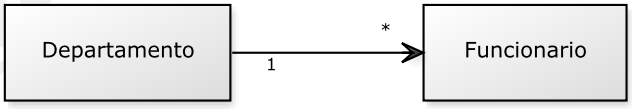
2.10 Relacionamentos

Os relacionamentos entre as entidades de um domínio devem ser expressos na modelagem através de vínculos entre classes. Podemos definir quatro tipos de relacionamentos de acordo com a cardinalidade.

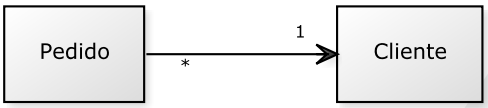
One to One (Um para Um) : Por exemplo, um estado é governado por apenas um governador e um governador governa apenas um estado.



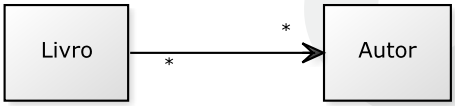
One to Many (Um para Muitos) : Por exemplo, um departamento possui muitos funcionários e um funcionário trabalha em apenas em um departamento.



Many to One (Muitos para Um) : Por exemplo, um pedido pertence a apenas um cliente e um cliente faz muitos pedidos.



Many to Many (Muitos para Muitos) : Por exemplo, um livro possui muitos autores e um autor possui muitos livros.



2.10.1 One to One

Suponha que no nosso domínio há duas entidades: Estado e Governador. Devemos criar uma classe para cada entidade e aplicar nelas as anotações básicas de mapeamento.

```
1 @Entity
2 class Estado {
3     @Id
4     @GeneratedValue
5     private Long id;
6 }
```

```
1 @Entity
2 class Governador {
3     @Id
4     @GeneratedValue
5     private Long id;
6 }
```

Como existe um relacionamento entre estados e governadores devemos expressar esse vínculo através de um atributo que pode ser inserido na classe ESTADO.

```
1 @Entity
2 class Estado {
3     @Id
4     @GeneratedValue
5     private Long id;
6
7     private Governador governador;
8 }
```

Além disso, devemos informar ao provedor do JPA que o relacionamento que existe entre um estado e um governador é do tipo One to One. Fazemos isso, aplicando a anotação **@OneToOne** no atributo que expressa o relacionamento.

```
@Entity
class Estado {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    private Governador governador;
}
```

No banco de dados, a tabela referente a classe ESTADO possuíra uma coluna de relacionamento também chamada de **join column**. Em geral, essa coluna será definida como uma chave estrangeira associada à tabela referente à classe GOVERNADOR.

Por padrão, o nome da coluna de relacionamento é a concatenação com "_" da entidade alvo do relacionamento com o nome da chave primária também da entidade alvo. No exemplo de estados e governadores, a join column teria o nome **governador_id**. Podemos alterar o nome padrão das join columns aplicando a anotação **@JoinColumn**.

```
@Entity
class Estado {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    @JoinColumn(name="gov_id")
    private Governador governador;
}
```

2.10.2 One to Many

Suponha que no nosso domínio há duas entidades: Departamento e Funcionário. Criaríamos duas classes com as anotações básicas de mapeamento.

```
@Entity
class Departamento {
    @Id
    @GeneratedValue
    private Long id;
}
```

```
@Entity
class Funcionario {
    @Id
    @GeneratedValue
    private Long id;
}
```

Como existe um relacionamento entre departamentos e funcionários devemos expressar esse vínculo através de um atributo que pode ser inserido na classe DEPARTAMENTO. Supondo que um departamento pode possuir muitos funcionários, devemos utilizar uma coleção para expressar esse relacionamento.

```
1  @Entity
2  class Departamento {
3      @Id
4      @GeneratedValue
5      private Long id;
6
7      private Collection<Funcionario> funcionarios;
8  }
```

Para informar a cardinalidade do relacionamento entre departamentos e funcionários, devemos utilizar a anotação **@OneToMany** na coleção.

```
1  @Entity
2  class Departamento {
3      @Id
4      @GeneratedValue
5      private Long id;
6
7      @OneToMany
8      private Collection<Funcionario> funcionarios;
9  }
```

No banco de dados, além das duas tabelas correspondentes às classes DEPARTAMENTO e FUNCIONARIO, uma terceira tabela será criada para relacionar os registros dos departamentos com os registros dos funcionários. Essa terceira tabela é chamada de tabela de relacionamento ou **join table**.

Por padrão, o nome da join table é a concatenação com "_" dos nomes das duas entidades. No exemplo de departamentos e funcionários, o nome do join table seria **Departamento_Funcionario**. Essa tabela possuíra duas colunas vinculadas às entidades que formam o relacionamento. No exemplo, a join table DEPARTAMENTO_FUNCIONARIO possuíra uma coluna chamada **Departamento_id** e outra chamada **funcionarios_id**.

Para personalizar os nomes das colunas da join table e dá própria join table, podemos aplicar a anotação **@JoinTable** no atributo que define o relacionamento.

```
1  @Entity
2  class Departamento {
3      @Id
4      @GeneratedValue
5      private Long id;
6
7      @OneToMany
8      @JoinTable(name="DEP_FUNC",
9          joinColumns=@JoinColumn(name="DEP_ID"),
10         inverseJoinColumns=@JoinColumn(name="FUNC_ID"))
11     private Collection<Funcionario> funcionarios;
12 }
```

2.10.3 Many to One

Suponha que no nosso domínio há duas entidades: Pedido e Cliente. As duas classes que modelariam essas entidades seriam anotadas com as anotações principais de mapeamento.

```
@Entity
class Pedido {
    @Id
    @GeneratedValue
    private Long id;
}
```

```
@Entity
class Cliente {
    @Id
    @GeneratedValue
    private Long id;
}
```

Como existe um relacionamento entre pedidos e clientes devemos expressar esse vínculo através de um atributo que pode ser inserido na classe PEDIDO. Supondo que um pedido pertence a um único cliente, devemos utilizar um atributo simples para expressar esse relacionamento.

```
@Entity
class Pedido {
    @Id
    @GeneratedValue
    private Long id;

    private Cliente cliente;
}
```

Para informar a cardinalidade do relacionamento entre pedidos e clientes, devemos utilizar a anotação **@ManyToOne** no atributo.

```
@Entity
class Pedido {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    private Cliente cliente;
}
```

No banco de dados, a tabela referente a classe PEDIDO possuíra uma **join column** vinculada à tabela da classe CLIENTE. Por padrão, o nome da join column é a concatenação com "_" da entidade alvo do relacionamento com o nome da chave primária também da entidade alvo. No exemplo de pedidos e clientes, o nome da join column seria **cliente_id**. Podemos alterar o nome padrão das join columns aplicando a anotação **@JoinColumn**.

```
@Entity
class Pedido {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToOne
    @JoinColumn(name="cli_id")
    private Cliente cliente;
}
```

2.10.4 Many to Many

Suponha que no nosso domínio há duas entidades: livros e autores. As classes e com as anotações básicas de mapeamento seriam mais ou menos assim:

```
@Entity
class Livro {
    @Id
    @GeneratedValue
    private Long id;
}
```

```
@Entity
class Autor {
    @Id
    @GeneratedValue
    private Long id;
}
```

Como existe um relacionamento entre livros e autores devemos expressar esse vínculo através de um atributo que pode ser inserido na classe LIVRO. Supondo que um livro pode ser escrito por muitos autores, devemos utilizar uma coleção para expressar esse relacionamento.

```
@Entity
class Livro {
    @Id
    @GeneratedValue
    private Long id;

    private Collection<Autor> autores;
}
```

Para informar a cardinalidade do relacionamento entre livros e autores, devemos utilizar a anotação **@ManyToMany** na coleção.

```
@Entity
class Livro {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToMany
    private Collection<Autor> autores;
}
```

No banco de dados, além das duas tabelas correspondentes às classes LIVRO e AUTOR, uma join table é criada para relacionar os registros dos livros com os registros dos autores. Por padrão, o nome da join table é a concatenação com "_" dos nomes das duas entidades. No exemplo de livros e autores, o nome do join table seria **Livro_Autor**. Essa tabela possuíra duas colunas vinculadas às entidades que formam o relacionamento. No exemplo, a join table LIVRO_AUTOR possuíra uma coluna chamada **Livro_id** e outra chamada **autores_id**.

Para personalizar os nomes das colunas da join table e dá própria join table, podemos aplicar a anotação **@JoinTable** no atributo que define o relacionamento.

```
@Entity
class Livro {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToMany
    @JoinTable(name="Liv_Aut",
        joinColumns=@JoinColumn(name="Liv_ID"),
        inverseJoinColumns=@JoinColumn(name="Aut_ID"))
    private Collection<Autor> autores;
}
```

2.11 Relacionamentos Bidirecionais

Quando expressamos um relacionamento entre entidades colocando um atributo em uma das entidades, podemos acessar a outra entidade a partir da primeira. Por exemplo, suponha o relacionamento entre governadores e estados.

```
@Entity
class Estado {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    private Governador governador;

    // GETTERS E SETTERS
}
```

Como o relacionamento está definido na classe ESTADO podemos acessar o governador a partir de um estado.

```
Estado e = manager.find(Estado.class, 1L);
Governador g = e.getGovernador();
```

Podemos expressar o relacionamento na classe GOVERNADOR também. Dessa forma, poderíamos acessar um estado a partir de um governador.

```
@Entity
class Governador {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    private Estado estado;

    // GETTERS E SETTERS
}
```

```
Governador g = manager.find(Governador.class, 1L);
Estado e = g.getEstado();
```

Porém, devemos indicar em uma das classes que esse relacionamento bidirecional é a junção de dois relacionamentos unidirecionais. Caso contrário, o provedor do JPA irá considerar dois relacionamentos distintos mapeando-os duas vezes. Em uma das classes devemos adicionar o atributo **mappedBy** na anotação @ONETOONE. O valor do MAPPEDBY é o nome do atributo que expressa o mesmo relacionamento na outra entidade.

```
@Entity
class Governador {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne(mappedBy="governador")
    private Estado estado;

    // GETTERS E SETTERS
}
```

2.12 Objetos Embutidos

Suponha que no nosso domínio existe uma entidade chamada **Pessoa**. Toda pessoa possui um endereço que é formado por **país**, **estado**, **cidade**, **logradouro**, **número**, **complemento** e **CEP**. Para melhorar a organização da nossa aplicação podemos decidir criar duas classes: **Pessoa** e **Endereco**

```
@Entity
class Pessoa {
    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    private Calendar nascimento;

    private Endereco endereco;
}
```

```
class Endereco {
    private String pais;

    private String estado;

    private String cidade;

    private String logradouro;

    private int numero;

    private String complemento;

    private int cep;
}
```

Muitas vezes não queremos mapear a classe ENDERECO para uma outra tabela. Normalmente, queremos que os dados dos endereços sejam guardados na mesma tabela que guarda os dados das pessoas. Nesse caso não devemos aplicar a anotação @ENTITY na classe ENDERECO. No lugar da anotação @ENTITY devemos aplicar a anotação **@Embeddable**. Além disso, não devemos definir uma chave para a classe ENDERECO pois ela não define uma entidade.

```
@Embeddable
class Endereco {
```

2.13 Exercícios

7. Implemente duas entidades no pacote **modelo** do projeto **Mapeamento**: ESTADO e GOVERNADOR.

```
@Entity
public class Governador {
    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    // GETTERS AND SETTERS
}
```

```
@Entity
public class Estado {
    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    @OneToOne
    private Governador governador;

    // GETTERS AND SETTERS
}
```

8. Adicione um governador e um estado através do seguinte teste.

```
public class AdicionaEstadoGovernador {
    public static void main(String[] args) {
        EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("K21_mapeamento");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        Governador g = new Governador();
        g.setNome("Rafael Cosentino");

        Estado e = new Estado();
        e.setNome("São Paulo");
        e.setGovernador(g);

        manager.persist(g);
        manager.persist(e);

        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

9. Abra o MySQL Query Browser e observe as propriedades da tabela **Estado** e da **Governador** da base de dados **K21_mapeamento**.

10. Implemente duas entidades no pacote **modelo** do projeto **Mapeamento**: DEPARTAMENTO e FUNCIONARIO.

```
@Entity
public class Funcionario {
    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    // GETTERS AND SETTERS
}
```

```
@Entity
public class Departamento {
    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    @OneToMany
    private Collection<Funcionario> funcionarios = new ArrayList<Funcionairo>();

    // GETTERS AND SETTERS
}
```

11. Adicione um departamento e um funcionairo através do seguinte teste.

```
public class AdicionaDepartamentoFuncionario {
    public static void main(String[] args) {
        EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("K21_mapeamento");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        Funcionario f = new Funcionario();
        f.setNome("Rafael Cosentino");

        Departamento d = new Departamento();
        d.setNome("Financeiro");
        d.getFuncionarios().add(f);

        manager.persist(f);
        manager.persist(d);

        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

12. Abra o MySQL Query Browser e observe as propriedades da tabela **Departamento**, da **Funcionario** e da **Departamento_Funcionario** da base de dados **K21_mapeamento**.

13. Implemente duas entidades no pacote **modelo** do projeto **Mapeamento**: PEDIDO e CLIENTE.

```
@Entity
public class Cliente {
    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    // GETTERS AND SETTERS
}
```

```
@Entity
public class Pedido {
    @Id
    @GeneratedValue
    private Long id;

    @Temporal(TemporalType.DATE)
    private Calendar data;

    @ManyToOne
    private Cliente cliente;

    // GETTERS AND SETTERS
}
```

14. Adicione um cliente e um departamento através do seguinte teste.

```
public class AdicionaPedidoCliente {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_mapeamento");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        Cliente c = new Cliente();
        c.setNome("Rafael Cosentino");

        Pedido p = new Pedido();
        p.setData(Calendar.getInstance());
        p.setCliente(c);

        manager.persist(c);
        manager.persist(p);

        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

15. Abra o MySQL Query Browser e observe as propriedades da tabela **Cliente** da **Pedido** da base de dados **K21_mapeamento**.

16. Implemente duas entidades no pacote **modelo** do projeto **Mapeamento**: LIVRO e AUTOR.

```
@Entity
public class Autor {
    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    // GETTERS AND SETTERS
}
```

```
@Entity
public class Livro {
    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    @ManyToMany
    private Collection<Autor> autores = new ArrayList<Autor>();

    // GETTERS AND SETTERS
}
```

17. Adicione um livro e um autor através do seguinte teste.

```
public class AdicionaLivroAutor {
    public static void main(String[] args) {
        EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("K21_mapeamento");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        Autor a = new Autor();
        a.setNome("Rafael Cosentino");

        Livro l = new Livro();
        l.setNome("JPA2");
        l.getAutores().add(a);

        manager.persist(a);
        manager.persist(l);

        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

18. Abra o MySQL Query Browser e observe as propriedades da tabela **Livro**, da **Autor** e da **Livro_Autor** da base de dados **K21_mapeamento**.

19. (Opcional) Crie uma classe para modelar endereços no pacote **modelo**.

```
public class Endereco {

    private String estado;

    private String cidade;

    private String logradouro;

    private int numero;

    // GETTERS AND SETTERS
}
```

20. (Opcional) Adicione um atributo na classe FUNCIONARIO do tipo ENDERECO.

```
@Entity
public class Funcionario {
    @Id
    @GeneratedValue
    private Long id;

    private String nome;

    @Embedded
    private Endereco endereco;

    // GETTERS AND SETTERS
}
```

21. (Opcional) Altere a classe ADICIONADEPARTAMENTOFUNCIONARIO e depois execute-a.

```
public class AdicionaDepartamentoFuncionario {
    public static void main(String[] args) {
        EntityManagerFactory factory =
            Persistence.createEntityManagerFactory("K21_mapeamento");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        Endereco e = new Endereco();
        e.setEstado("São Paulo");
        e.setCidade("São Paulo");
        e.setLogradouro("Av. Bigadeiro Faria Lima");
        e.setNumero(1571);

        Funcionario f = new Funcionario();
        f.setNome("Rafael Cosentino");
        f.setEndereco(e);

        Departamento d = new Departamento();
        d.setNome("Financeiro");
        d.getFuncionarios().add(f);

        manager.persist(f);
        manager.persist(d);

        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

2.14 Herança

O mapeamento objeto relacional descreve como os conceitos de orientação a objetos são definidos no banco de dados através dos conceitos do modelo entidade relacionamento. De todos os conceitos de orientação a objetos, um dos mais complexos de se mapear é o de Herança.

A especificação JPA define três estratégia para realizar o mapeamento de herança.

- Single Table
- Joined
- Table Per Class

2.14.1 Single Table

A estratégia Single Table é a mais comum e a que possibilita melhor desempenho em relação a velocidade das consultas. Nessa estratégia, a super classe deve ser anotada com @INHERITANCE(STRATEGY=INHERITANCETYPE.SINGLE_TABLE).

O provedor JPA criará apenas uma tabela com o nome da super classe para armazenar os dados dos objetos criados a partir da super classe ou das sub classes. Todos os atributos da super classe e os das sub classes serão mapeados para colunas dessa tabela. Além disso, uma coluna especial chamada **DTYPE** será utilizada para identificar a classe do objeto correspondente ao registro.

```
1  @Entity
2  @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
3  public class Pessoa {
4
5      @Id @GeneratedValue
6      private Long id;
7
8      private String nome;
9
10 }
```

```
1  @Entity
2  public class PessoaJuridica extends Pessoa{
3      private String cnpj;
4  }
```

```
1  @Entity
2  public class PessoaFisica extends Pessoa{
3      private String cpf;
4  }
```

A desvantagem da Single Table é consumo desnecessário de espaço já que nem todos os campos são utilizados para todos os registros. Por exemplo, se uma pessoa jurídica for cadastrada o campo CPF não seria utilizado. Da mesma forma, que se uma pessoa física for cadastrada o campo CNPJ não seria utilizado.

2.14.2 Joined

Nessa estratégia, uma tabela para cada classe da hierarquia é criada. Em cada tabela, apenas os campos referentes aos atributos da classe correspondente são criados. Para relacionar os registros das diversas tabelas e remontar os objetos quando uma consulta for realizada, as tabelas

correspondentes às sub classes possuem chaves estrangeiras vinculadas à tabela correspondente à super classe.

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

    @Id @GeneratedValue
    private Long id;

    private String nome;

}
```

```
@Entity
public class PessoaJuridica extends Pessoa{
    private String cnpj;
}
```

```
@Entity
public class PessoaFisica extends Pessoa{
    private String cpf;
}
```

O consumo de espaço utilizando a estratégia **Joined** é menor do que utilizando a estratégia **Single Table**. Contudo, as consultas são mais lentas pois é necessário realizar operações de **join** para recuperar os dados dos objetos.

2.14.3 Table Per Class

Nessa estratégia, uma tabela para cada classe concreta da hierarquia é criada. Contudo, os dados de um objeto não são colocados em tabelas diferentes. Dessa forma, para remontar um objeto não é necessário realizar operações de **join**. A desvantagem desse modo é que não existe um vínculo explícito no banco de dados entres as tabelas correspondentes as classes da hierarquia.

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Pessoa {
    @Id
    private Long id;

    private String nome;

}
```

```
@Entity
public class PessoaJuridica extends Pessoa{
    private String cnpj;
}
```

```
@Entity
public class PessoaFisica extends Pessoa{
    private String cpf;
}
```

Na estratégia **Table Per Class**, não podemos utilizar a geração automática de chave primárias simples e numéricas.

2.15 Exercícios

22. Adicione uma classe chamada **Pessoa** no pacote **modelo**.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class Pessoa {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    //GETTERS AND SETTERS
}
```

23. Faça a classe **PessoaJuridica** no pacote **modelo**.

```
@Entity
public class PessoaJuridica extends Pessoa{
    private String cnpj;

    //GETTERS AND SETTERS
}
```

24. Faça a classe **PessoaFisica** no pacote **modelo**.

```
@Entity
public class PessoaFisica extends Pessoa{
    private String cpf;

    //GETTERS AND SETTERS
}
```

25. Crie um teste para adicionar pessoas.

```
1  public class AdicionaPessoa {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence.createEntityManagerFactory("↩
               K21_mapeamento");
4          EntityManager manager = factory.createEntityManager();
5
6          manager.getTransaction().begin();
7
8          Pessoa p1 = new Pessoa();
9          p1.setNome("Marcelo");
10
11         PessoaFisica p2 = new PessoaFisica();
12         p2.setNome("Rafael");
13         p2.setCpf("1234");
14
15         PessoaJuridica p3 = new PessoaJuridica();
16         p3.setNome("K19");
17         p3.setCnpj("567788");
18
19         manager.persist(p1);
20         manager.persist(p2);
21         manager.persist(p3);
22
23         manager.getTransaction().commit();
24
25         manager.close();
26         factory.close();
27     }
28 }
```

Capítulo 3

Entity Manager

As instâncias das entidades do JPA são administradas pelos Entity Managers. As duas principais responsabilidades dos Entity Managers são: gerenciar o estado dos objetos e sincronizar os dados da aplicação e do banco de dados.

3.1 Estados

É necessário conhecer o ciclo de vida das entidades para saber como os objetos são administrados pelos Entity Managers. Uma instância de uma entidade pode passar pelos seguintes estados:

Novo(New) : Um objeto no estado new não possui uma identidade (chave) e não está associado a um Entity Manager. O conteúdo desse objeto não é sincronizado com o banco de dados. Toda instância de uma entidade que acabou de ser criada com o comando NEW se encontra no estado new do JPA.

Administrado(Managed) : Um objeto no estado managed possui uma identidade e está associado a um Entity Manager que mantém o conteúdo desse objeto sincronizado com o banco de dados.

Desvinculado(Detached) : Um objeto no estado detached possui uma identidade e não está associado a um Entity Manager. Dessa forma, o conteúdo desse objeto não é sincronizado com o banco de dados.

Removido(Removed) : Um objeto no estado removed possui uma identidade e está associado a um Entity Manager. O conteúdo desse objeto será removido do banco de dados.

3.2 Sincronização com o Banco de Dados

As modificações realizadas no estado dos objetos managed são propagadas para o banco de dados. Os registros referentes aos objetos em estado removed são apagados do banco de dados. Essa sincronização é responsabilidade dos provedores de JPA. De acordo com a especificação, uma sincronização só pode ocorrer se uma transação estiver ativa.

```
manager.getTransaction().begin();
// nesse trecho pode acorrer sincronizações
manager.getTransaction().commit();
```

3.2.1 Flush Mode

Há duas políticas adotadas pelos provedores em relação às sincronizações:

FlushModeType.AUTO: As sincronizações ocorrem antes de uma consulta ou em chamadas dos métodos COMMIT() ou FLUSH().

FlushModeType.COMMIT: As sincronizações ocorrem em chamadas dos métodos COMMIT() ou FLUSH().

De acordo com a especificação JPA, no FLUSHMODETYPE.AUTO, os provedores devem garantir que as consultas considerem alterações realizadas nas entidades. Por isso, os provedores realizam uma sincronização antes de uma consulta. Por outro lado, no FLUSHMODETYPE.COMMIT, a especificação não determina a mesma responsabilidade aos provedores. Por isso, nesse modo, eles não realizam uma sincronização antes de uma consulta.

Em relação a performance, o FLUSHMODETYPE.COMMIT é mais eficiente pois realiza menos sincronizações com o banco de dados do que o FLUSHMODETYPE.AUTO. Contudo, muitas vezes, o FLUSHMODETYPE.AUTO é necessário para garantir a informação mais atualizada seja obtida nas consultas.

Podemos configurar o flush mode no nível de um Entity Manager afetando o comportamento em todas as consultas realizadas através desse Entity Manager ou configurar apenas para uma consulta.

```
manager.setFlushMode(FlushModeType.COMMIT);
```

```
query.setFlushMode(FlushModeType.AUTO);
```

OBS: O tipo padrão de flush mode é o AUTO de acordo com a especificação JPA.

3.3 Transições

Uma instância de uma entidade pode mudar de estado. Veremos a seguir as principais transições.

3.3.1 New -> Managed

Um objeto no estado new passa para o estado managed quando utilizamos o método **persist()** dos Entity Managers. O estado desse objeto será armazenado no banco de dados e ele ganhará uma identidade.

```
@Entity
class Pessoa {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    // GETTERS AND SETTERS
}
```

```
manager.getTransaction().begin();

Pessoa p = new Pessoa();
p.setNome("Rafael Cosentino");
manager.persist();

manager.getTransaction().commit();
```

O conteúdo do objeto será armazenado no banco de dados quando o provedor realizar uma sincronização. Veja a seção sobre sincronização.

3.3.2 BD -> Managed

Quando dados são recuperados do banco de dados, o provedor do JPA cria objetos para armazenar as informações e esses objetos são considerados managed.

```
Pessoa p = manager.find(Pessoa.class, 1L);
```

```
Pessoa p = manager.getReference(Pessoa.class, 1L);
```

```
Query query = manager.createQuery("select p from Pessoa p");
List<Pessoa> lista = query.getResultList();
```

3.3.3 Managed -> Detached

Quando não queremos mais que um objeto no estado managed seja administrado, podemos desvinculá-lo do seu Entity Manager tornando-o detached. Dessa forma, o conteúdo desse objeto não será mais sincronizado com o banco de dados.

Para tornar apenas um objeto detached, devemos utilizar o método **detach()**:

```
Pessoa p = manager.find(Pessoa.class, 1L);
manager.detach(p);
```

Para tornar detached todos os objetos administrados por um Entity Manager, devemos utilizar o método **clear()**.

```
1 manager.clear();
```

Outra maneira de tornar detached todos os objetos administrados por um Entity Manager é utilizar o método **close()**.

```
1 manager.close();
```

3.3.4 Detached -> Managed

O estado de um objeto detached pode ser propagado para um objeto managed com a mesma identidade para que os dados sejam sincronizados com o banco de dados. Esse processo é realizado pelo método **merge()**.

```
1 Pessoa pessoaManaged = manager.merge(pessoaDetached);
```

3.3.5 Managed -> Removed

Quando um objeto managed se torna detached, os dados correspondentes a esse objeto não são apagados do banco de dados. Agora, quando utilizamos o método **remove()** marcamos um objeto para ser removido do banco de dados.

```
1 Pessoa p = manager.find(Pessoa.class, 1L);
2 manager.remove(p);
```

O conteúdo do objeto será removido no banco de dados quando o provedor realizar uma sincronização. Veja a seção sobre sincronização.

3.3.6 Managed -> Managed

O conteúdo de um objeto no estado managed pode ficar desatualizado em relação ao banco de dados se alguém ou alguma aplicação alterar os dados na base de dados. Para atualizar um objeto managed com os dados do banco de dados, devemos utilizar o método **refresh()**.

```
1 Pessoa p = manager.find(Pessoa.class, 1L);
2 manager.refresh(p);
```

3.4 Regras de Transições

A especificação determina um grande conjunto de regras burocráticas em relação às transições. Consulte essas regras na seção 3.8 da JSR 317 - Java Persistence API, Version 2.0 http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html.

3.5 Exercícios

1. Crie um projeto no eclipse chamado **EntityManager**. Copie a pasta **lib** do projeto **JPA2-Hibernate** para o projeto **EnityManager**. Depois selecione todos os jar's e os adicione no classpath.

2. Abra o **MySQL Query Browser** e apague a base de dados **K21_entity_manager** se existir. Depois crie a base de dados **K21_entity_manager**.

3. Copie a pasta **META-INF** do projeto **JPA2-Hibernate** para dentro da pasta **src** do projeto **EntityManager**. Altere o arquivo **persistence.xml** do projeto **EntityManager**, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/↩
           ns/persistence/persistence_1_0.xsd"
5      version="1.0">
6
7      <persistence-unit name="K21_entity_manager" transaction-type="RESOURCE_LOCAL">
8          <provider>org.hibernate.ejb.HibernatePersistence</provider>
9          <properties>
10             <property name="hibernate.dialect" value="org.hibernate.dialect.↩
                   MySQL5InnoDBDialect"/>
11             <property name="hibernate.hbm2ddl.auto" value="update"/>
12             <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver↩
                   "/>
13             <property name="javax.persistence.jdbc.user" value="root"/>
14             <property name="javax.persistence.jdbc.password" value="root"/>
15             <property name="javax.persistence.jdbc.url" value="jdbc:mysql://↩
                   localhost:3306/K21_entity_manager"/>
16         </properties>
17     </persistence-unit>
18 </persistence>
```

4. Crie um pacote chamado **modelo** no projeto **EntityManager** e adicione a seguinte classe:

```
1  @Entity
2  public class Pessoa {
3
4      @Id @GeneratedValue
5      private Long id;
6
7      private String nome;
8
9      // GETTERS E SETTERS
10 }
```

5. Teste persistir objetos através de um Entity Manager. Crie uma classe chamada **TestePersist** dentro de um pacote chamado **testes**.

```
1  public class TestePersist {
2      public static void main(String[] args) {
3          EntityManagerFactory factory =
4                  Persistence.createEntityManagerFactory("K21_entity_manager");
5          EntityManager manager = factory.createEntityManager();
6
7          // ABRINDO A TRASACAO
8          manager.getTransaction().begin();
9
10         // OBJETO NO ESTADO NEW
11         Pessoa p = new Pessoa();
12         p.setNome("Rafael Cosentino");
13
14         // OBJETO NO ESTADO MANAGED
15         manager.persist(p);
16
17         // SINCRONIZANDO E CONFIRMANDO A TRANSACAO
18         manager.getTransaction().commit();
19
20         System.out.println("Pessoa id: " + p.getId());
21
22         manager.close();
23         factory.close();
24     }
25 }
```

Execute e consulte o banco de dados!

6. Teste buscar objetos através de um Entity Manager dado a identidade dos objetos. Crie uma classe chamada **TesteFind** dentro de um pacote **testes**.

```
1  public class TesteFind {
2      public static void main(String[] args) {
3          EntityManagerFactory factory =
4                  Persistence.createEntityManagerFactory("K21_entity_manager");
5          EntityManager manager = factory.createEntityManager();
6
7          // OBJETO NO ESTADO MANAGED
8          Pessoa p = manager.find(Pessoa.class, 1L);
9          System.out.println("Id: " + p.getId());
10         System.out.println("Nome: " + p.getNome());
11
12         manager.close();
13         factory.close();
14     }
15 }
```

Execute e consulte o banco de dados!

7. Teste alterar objetos no estado managed e depois faça um sincronização com o banco de dados através de uma chamada ao método COMMIT(). Crie uma classe chamada **TesteManaged** dentro de um pacote **testes**.

```java
public class TesteManaged {
    public static void main(String[] args) {
        EntityManagerFactory factory =
                Persistence.createEntityManagerFactory("K21_entity_manager");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        // OBJETO NO ESTADO MANAGED
        Pessoa p = manager.find(Pessoa.class, 1L);

        // ALTERANDO O CONTEUDO DO OBJETO
        p.setNome("Marcelo Martins");

        // SINCRONIZANDO E CONFIRMANDO A TRANSACAO
        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

Execute e consulte o banco de dados!

8. Teste alterar objetos no estado detached e depois faça um sincronização com o banco de dados através de uma chamada ao método COMMIT(). Crie uma classe chamada **TesteDetached** dentro de um pacote **testes**.

```java
public class TesteDetached {
    public static void main(String[] args) {
        EntityManagerFactory factory =
                Persistence.createEntityManagerFactory("K21_entity_manager");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        // OBJETO NO ESTADO MANAGED
        Pessoa p = manager.find(Pessoa.class, 1L);

        // OBJETO NO ESTADO DETACHED
        manager.detach(p);

        // ALTERANDO O CONTEUDO DO OBJETO
        p.setNome("Marcelo Martins");

        // SINCRONIZANDO E CONFIRMANDO A TRANSACAO
        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

Execute e consulte o banco de dados!

3.6 LAZY e EAGER

Como os Entity Managers administram as instâncias das entidades eles são responsáveis pelo carregamento do estado dos objetos. Há dois modos de carregar os dados de um objeto:

LAZY ou EAGER. No modo LAZY, o provedor posterga ao máximo a busca dos dados no banco de dados. No modo EAGER, o provedor busca imediatamente os dados no banco de dados.

3.6.1 find() VS getReference()

Tanto o método FIND() quanto o método GETREFERENCE() permitem que a aplicação obtenha instâncias das entidades a partir das identidade dos objetos. A diferença entre eles é que o FIND() tem comportamento EAGER e o GETREFERENCE() tem comportamento LAZY.

```
1 Pessoa p = manager.find(Pessoa.class, 1L);
2 // o objeto já está carregado
```

```
1 Pessoa p = manager.getReference(Pessoa.class, 1L);
2 // o objeto não está carregado ainda
3 String nome = p.getNome();
4 // agora o objeto está carregado
```

3.6.2 Fetch Type - Tipos Básicos

A escolha do modo de carregamento EAGER ou LAZY pode ser realizada no nível dos atributos básicos. Por padrão, os atributos básicos são carregados em modo EAGER. Não é possível forçar que o modo de carregamento de um atributo seja LAZY. O que é possível fazer é indicar aos provedores de JPA que seria interessante utilizar o modo LAZY para determinados atributos. Contudo, os provedores podem não aceitar essa indicação e utilizar o modo EAGER.

De qualquer forma, podemos indicar o modo LAZY para atributos básicos através da anotação **@Basic** e da propriedade **fetch**.

```
1 @Basic(fetch=FetchType.LAZY)
2 protected String getNome() {
3     return nome;
4 }
```

Obs: O modo LAZY para atributos básicos só pode ser aceito pelos provedores se o modo de acesso for Property Access.

3.6.3 Fetch Type - Relacionamentos

Podemos definir o modo de carregamento que desejamos utilizar para os relacionamentos das entidades. Por exemplo, suponha um relacionamento unidirecional entre estados e governadores.

```
@Entity
class Estado {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne
    private Governador governador;
}
```

```
@Entity
class Governador {
    @Id
    @GeneratedValue
    private Long id;
}
```

Por padrão, quando os dados de um estado são recuperados do banco de dados, os dados do governador associado a esse estado também é recuperado. Em outras palavras, o modo de carregamento padrão do atributo que estabelece o relacionamento entre estados e governadores é EAGER. Podemos alterar esse comportamento padrão aplicando a propriedade **fetch** na anotação **@OneToOne**.

```
@Entity
class Estado {
    @Id
    @GeneratedValue
    private Long id;

    @OneToOne(fetch=FetchType.LAZY)
    private Governador governador;
}
```

O modo de carregamento dos relacionamentos do tipo **One To One** e **Many To One** é por padrão EAGER. O modo de carregamento dos relacionamentos do tipo **One To Many** e **Many To Many** é por padrão LAZY. Lembrando que o modo de carregamento pode ser definido com a propriedade **fetch**.

```
@OneToOne(fetch=FetchType.LAZY)
@ManyToOne(fetch=FetchType.LAZY)
@OneToMany(fetch=FetchType.EAGER)
@ManyToMany(fetch=FetchType.EAGER)
```

3.7 Lazy Initialization

No modo de carregamento LAZY, o provedor JPA posterga ao máximo a busca dos dados no banco de dados. O carregamento só poderá ser feito posteriormente se o Entity Manager correspondente estiver aberto. Se os dados de um objeto ainda não foram carregados e a aplicação fechar o Entity Manager correspondente, o provedor lançará uma exception.

3.8 Persistence Context ou Cache de Primeiro Nível

Um objeto já carregado por um Entity Manager é mantido no Persistence Context (Cache de Primeiro Nível, na nomenclatura do Hibernate). Cada Entity Manager possui o seu próprio Persistence Context. Se a aplicação buscar um objeto através de um Entity Manager e ele já estiver carregado no Persistence Context correspondente, a busca não será realizado no banco de dados evitando assim uma consulta.

3.9 Exercícios

9. Acrescente duas propriedades no arquivo de configuração do JPA (persistence.xml) para indicar ao Hibernate que desejamos ver as consultas no console.

```
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="true" />
```

10. Teste o comportamento do método de busca **find**(). Crie um classe chamada **TesteFindEager** no pacote **testes**.

```
public class TesteFindEager {
    public static void main(String[] args) {
        EntityManagerFactory factory =
                Persistence.createEntityManagerFactory("K21_entity_manager");
        EntityManager manager = factory.createEntityManager();

        System.out.println("-----------CHAMANDO O FIND------------");
        Pessoa p = manager.find(Pessoa.class, 1L);
        System.out.println("-----------FEZ O SELECT---------------");

        manager.close();
        factory.close();
    }
}
```

Execute e veja a saída!

11. Teste o comportamento do método de busca **getReference**(). Crie um classe chamada **TesteGetReferenceLazy** no pacote **testes**.

```
1  public class TesteGetReferenceLazy {
2      public static void main(String[] args) {
3          EntityManagerFactory factory =
4                  Persistence.createEntityManagerFactory("K21_entity_manager");
5          EntityManager manager = factory.createEntityManager();
6
7          System.out.println("-----------CHAMANDO O GETREFERENCE----");
8          Pessoa p = manager.getReference(Pessoa.class, 1L);
9          System.out.println("-----------NAO FEZ O SELECT-----------");
10
11         manager.close();
12         factory.close();
13     }
14 }
```

Execute e veja a saída!

12. Teste o problema de Lazy Initialization. Crie um classe chamada **TesteLazyInitialization** no pacote **testes**.

```
1  public class TesteLazyInitialization {
2      public static void main(String[] args) {
3          EntityManagerFactory factory =
4                  Persistence.createEntityManagerFactory("K21_entity_manager");
5          EntityManager manager = factory.createEntityManager();
6
7          // OBJETO CARREGADO EM MODO LAZY
8          Pessoa p = manager.getReference(Pessoa.class, 1L);
9
10         manager.close();
11         factory.close();
12
13         System.out.println(p.getNome());
14     }
15 }
```

Execute e veja a saída!

13. Crie duas classes para modelar governadores e estados, estabelecendo um relacionamento One to One entre essas entidades.

```
1  @Entity
2  public class Estado {
3
4      @Id @GeneratedValue
5      private Long id;
6
7      private String nome;
8
9      @OneToOne
10     private Governador governador;
11
12     // GETTERS AND SETTERS
13 }
```

```
@Entity
public class Governador {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    @OneToOne(mappedBy="governador")
    private Estado estado;

    // GETTERS AND SETTERS
}
```

14. Adicione um governador e um estado. Crie uma classe chamada **AdicionaGovernadorEstado** no pacote **testes**.

```
public class AdicionaGovernadorEstado {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_entity_manager");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        Governador governador = new Governador();
        governador.setNome("Rafael Cosentino");

        Estado estado = new Estado();
        estado.setNome("São Paulo");

        governador.setEstado(estado);
        estado.setGovernador(governador);

        manager.persist(estado);
        manager.persist(governador);

        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

15. Teste o carregamento EAGER no relacionamento One to One entre estados e governadores. Crie uma classe chamada **TesteCarregamentoRelacionamento** no pacote **testes**.

```
public class TesteCarregamentoRelacionamento {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence.createEntityManagerFactory("↩
            K21_entity_manager");
        EntityManager manager = factory.createEntityManager();

        Estado estado = manager.find(Estado.class, 1L);
    }
}
```

Observe a saída no console para verificar o carregamento tanto do estado quanto

do governador

16. Altere a política padrão do carregamento do governador adicionando a propriedade FETCH na anotação @ONETOONE na classe ESTADO.

```
@OneToOne(fetch=FetchType.LAZY)
private Governador governador;
```

17. Execute novamente a classe TESTECARREGAMENTORELACIOMENTO e observe a saída do console para verificar que agora somente o estado é carregado.

18. (Opcional) Faça acontecer o problema de Lazy Initialization no exemplo de estados e governadores.

19. Verifique o comportamento dos Entity Managers ao buscar duas vezes o mesmo objeto. Crie uma classe chamada **TestePersistenceContext** no pacote **testes**.

```
public class TestePersistenceContext {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_entity_manager");
        EntityManager manager = factory.createEntityManager();

        System.out.println("------------------PRIMEIRO FIND-----------------");
        Estado estado = manager.find(Estado.class, 1L);
        System.out.println("------------------SEGUNDO FIND------------------");
        estado = manager.find(Estado.class, 1L);
    }
}
```

Execute e observe a saída no console para verificar que o provedor só realiza uma busca

3.10 Cascade

As operações dos Entity Managers são realizadas somente no objeto passado como parâmetro para o método que implementa a operação. Por exemplo, suponha o relacionamento entre estados e governadores.

```
@Entity
public class Estado {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    @OneToOne
    private Governador governador;

    // GETTERS AND SETTERS
}
```

```
@Entity
public class Governador {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    @OneToOne(mappedBy="governador")
    private Estado estado;

    // GETTERS AND SETTERS
}
```

Suponha que um objeto da classe ESTADO e outro da classe GOVERNADOR sejam criados e associados. Se apenas um dos objetos for persistido um erro ocorrerá na sincronização com o banco de dados.

```
Governador governador = new Governador();
governador.setNome("Rafael Cosentino");

Estado estado = new Estado();
estado.setNome("São Paulo");

governador.setEstado(estado);
estado.setGovernador(governador);

manager.getTransaction().begin();
manager.persist(estado);
manager.getTransaction().commit();
```

Para evitar o erro, os dois objetos precisam ser persistidos.

```
manager.getTransaction().begin();
manager.persist(estado);
manager.persist(governador);
manager.getTransaction().commit();
```

Ou então podemos configurar o relacionamento entre estados e governadores com a propriedade **cascade** para a operação PERSIST.

```
@OneToOne(fetch=FetchType.LAZY, cascade=CascadeType.PERSIST)
private Governador governador;
```

A propriedade cascade pode ser utilizada para as outras operações dos Entity Managers.

- CascadeType.PERSIST
- CascadeType.DETACH
- CascadeType.MERGE
- CascadeType.REFRESH
- CascadeType.REMOVE

- CascadeType.ALL

Além disso, a propriedade cascade é unidirecional. Dessa forma, nos relacionamentos bidirecionais para ter o comportamente do cascade nas duas direções é necessário utilizar a propriedade cacasde nas duas entidades.

3.11 Exercícios

20. Tente adicionar um governador e um estado. Crie uma classe chamada **TesteCascade** no pacote **testes**.

```
1  public class TesteCascade {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence.createEntityManagerFactory("↩
               K21_entity_manager");
4          EntityManager manager = factory.createEntityManager();
5
6          Governador governador = new Governador();
7          governador.setNome("Rafael Cosentino");
8
9          Estado estado = new Estado();
10         estado.setNome("São Paulo");
11
12         governador.setEstado(estado);
13         estado.setGovernador(governador);
14
15         manager.getTransaction().begin();
16         manager.persist(estado);
17         manager.getTransaction().commit();
18     }
19 }
```

Execute e observe o erro

21. Modifique a classe **Estado** para configurar a propriedade cascade no relacionamento com governadores.

```
1  @Entity
2  public class Estado {
3
4      @Id @GeneratedValue
5      private Long id;
6
7      private String nome;
8
9      @OneToOne(cascade=CascadeType.PERSIST)
10     private Governador governador;
11
12     // GETTERS AND SETTERS
13 }
```

22. Execute a classe **TesteCascade** e observe que não ocorre o mesmo erro que aconteceu anteriormente.

Capítulo 4

JPQL

A capacidade que os bancos de dados possuem de realizar consultas complexas é um forte argumento para utilizá-los. A definição e os resultados das consultas nos banco de dados são fortemente baseados ao modelo relacional. Por outro lado, é natural que as aplicações baseadas no modelo orientado a objetos desejem que a definição e os resultados das consultas sejam baseados no paradigma orientado a objetos.

Por isso, os provedores de JPA oferecem mecanismos para realizar consultas de uma maneira orientada a objetos. Para ser mais exato, a especificação JPA 2 define dois mecanismos para realizar consultas orientadas a objetos: o primeiro utiliza uma linguagem específica para consultas chamada JPQL (Java Persistence Query Language) e o segundo é basicamente uma biblioteca Java para consultas. Nesse capítulo mostraremos o funcionamento da JPQL.

4.1 Consultas Dinâmicas

Consultas em JPQL podem ser definidas em qualquer classe Java, dentro de um método por exemplo. Para criar uma consulta devemos utilizar o método **createQuery()** passando uma string com o código JPQL. Consultas criadas dessa maneira são chamadas de consultas dinâmicas.

```
1 public void umMetodoQualquer() {
2     String jpql = "SELECT p FROM Pessoa p";
3     Query query = manager.createQuery(jpql);
4 }
```

Apesar da flexibilidade, criar consultas dinâmicas pode prejudicar a performance da aplicação. Por exemplo, se uma consulta dinâmica é criada dentro de um método toda vez que esse método for chamado o código JPQL dessa consulta será processado pelo provedor. Uma alternativa menos flexível porém mais performática às consultas dinâmicas são as **Named Queries**.

4.2 Named Query

Diferentemente de uma consulta dinâmica, uma Named Query é processado apenas no momento da criação da fábrica de Entity Manager. Além disso, os provedores JPA podem mapear

as Named Queries para Stored Procedures precompiladas do banco de dados melhorando a performance das consultas.

As Named Queries são definidas através de anotações nas classes que implementam as entidades. Podemos aplicar a anotação **@NamedQuery** quando queremos definir apenas uma consulta ou a anotação **@NamedQueries** quando queremos definir várias consultas.

```
1 @NamedQuery(name="Pessoa.findAll", query="SELECT p FROM Pessoa p")
2 class Pessoa
```

```
1 @NamedQueries({
2     @NamedQuery(name="Pessoa.findAll", query="SELECT p FROM Pessoa p"),
3     @NamedQuery(name="Pessoa.count", query="SELECT COUNT(p) FROM Pessoa p")
4 })
5 class Pessoa
```

Para executar uma Named Query, devemos utilizar o método **createNamedQuery**().

```
1 public void umMetodoQualquer() {
2     Query query = manager.createNamedQuery("Pessoa.findAll");
3     List<Pessoa> pessoas = query.getResultList();
4 }
```

Os nomes das Named Queries devem ser únicos no contexto da aplicação. Caso contrário, o provedor lançará uma exception na criação da fábrica de Entity Managers. Dessa forma, a convenção é utilizar o nome da classe na qual a Named Query está definida como prefixo do nome da consulta.

4.3 Parâmetros

Para tornar as consultas em JPQL mais genéricas e evitar problemas com SQL Injection, devemos parametrizá-las. Adicionar um parâmetro em uma consulta é simples, basta utilizar caractere ":" seguido do nome do argumento.

```
1 @NamedQuery(name="Pessoa.findByIdade",
2             query="SELECT p FROM Pessoa p WHERE p.idade > :idade")
```

Antes de executar uma consulta com parâmetros, devemos definir os valores dos argumentos.

```
1 public void umMetodoQualquer() {
2     Query query = manager.createNamedQuery("Pessoa.findByIdade");
3     query.setParameter("idade", 18);
4     List<Pessoa> pessoasComMaisDe18 = query.getResultList();
5 }
```

É possível também adicionar parâmetros em uma consulta de maneira ordinal basta utilizar

o caractere “?” seguido de um número.

```
@NamedQuery(name="Pessoa.findByIdade",
            query="SELECT p FROM Pessoa p WHERE p.idade > ?1")
```

```
public void umMetodoQualquer() {
    Query query = manager.createNamedQuery("Pessoa.findByIdade");
    query.setParameter(1, 18);
    List<Pessoa> pessoasComMaisDe18 = query.getResultList();
}
```

4.4 Exercícios

1. Crie um projeto no eclipse chamado **JPQL**. Copie a pasta **lib** do projeto **EntityManager** para o projeto **JPQL**. Depois selecione todos os jar’s e os adicione no classpath.

2. Abra o **MySQL Query Browser** e apague a base de dados **K21_jpql** se existir. Depois crie a base de dados **K21_jpql**.

3. Copie a pasta **META-INF** do projeto **EntityManager** para dentro da pasta **src** do projeto **JPQL**. Altere o arquivo **persistence.xml** do projeto **JPQL**, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <persistence-unit name="K21_jpql"
        transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <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="root" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/K21_jpql" />
        </properties>
    </persistence-unit>
</persistence>
```

4. Crie um pacote chamado **modelo** no projeto **JPQL** e adicione as seguintes classes:

```java
@Entity
public class Autor {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    @ManyToMany
    private Collection<Livro> livros = new ArrayList<Livro>();

    // GETTERS E SETTERS
}
```

```java
@Entity
public class Livro {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    private Double preco;

    // GETTERS E SETTERS
}
```

5. Carregue o banco de dados com as informações de alguns livros e autores. Adicione a seguinte classe em um novo pacote chamado **testes**.

```java
public class PopulaBanco {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_jpql");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        Livro livro1 = new Livro();
        livro1.setNome("The Battle for Your Mind");
        livro1.setPreco(20.6);
        manager.persist(livro1);

        Livro livro2 = new Livro();
        livro2.setNome("Differentiate or Die");
        livro2.setPreco(15.8);
        manager.persist(livro2);

        Livro livro3 = new Livro();
        livro3.setNome("How to Transform Your Ideas");
        livro3.setPreco(32.7);
        manager.persist(livro3);

        Livro livro4 = new Livro();
        livro4.setNome("Digital Fortress");
        livro4.setPreco(12.9);
        manager.persist(livro4);

        Livro livro5 = new Livro();
        livro5.setNome("Marketing in an Era of Competition, Change and Crisis");
        livro5.setPreco(26.8);
        manager.persist(livro5);

```

```
34          Autor autor1 = new Autor();
35          autor1.setNome("Patrick Cullen");
36          autor1.getLivros().add(livro2);
37          autor1.getLivros().add(livro4);
38          manager.persist(autor1);
39
40          Autor autor2 = new Autor();
41          autor2.setNome("Fraser Seitel");
42          autor2.getLivros().add(livro3);
43          manager.persist(autor2);
44
45          Autor autor3 = new Autor();
46          autor3.setNome("Al Ries");
47          autor3.getLivros().add(livro1);
48          manager.persist(autor3);
49
50          Autor autor4 = new Autor();
51          autor4.setNome("Jack Trout");
52          autor4.getLivros().add(livro1);
53          autor4.getLivros().add(livro2);
54          autor4.getLivros().add(livro5);
55          manager.persist(autor4);
56
57          Autor autor5 = new Autor();
58          autor5.setNome("Steve Rivkin");
59          autor5.getLivros().add(livro2);
60          autor5.getLivros().add(livro3);
61          autor5.getLivros().add(livro5);
62          manager.persist(autor5);
63
64          manager.getTransaction().commit();
65
66          manager.close();
67          factory.close();
68      }
69  }
```

6. Teste as consultas dinâmicas com JPQL. Crie a seguinte classe no pacote **testes**.

```
1   public class TesteConsultaDinamicas {
2       public static void main(String[] args) {
3           EntityManagerFactory factory = Persistence
4                   .createEntityManagerFactory("K21_jpql");
5           EntityManager manager = factory.createEntityManager();
6
7           Query query = manager.createQuery("select a from Autor a");
8           List<Autor> autores = query.getResultList();
9
10          for (Autor autor : autores) {
11              System.out.println("Autor: " + autor.getNome());
12              Collection<Livro> livros = autor.getLivros();
13
14              for (Livro livro : livros) {
15                  System.out.println("Livro: " + livro.getNome());
16                  System.out.println("Preço: " + livro.getPreco());
17                  System.out.println();
18              }
19              System.out.println();
20          }
21
22          manager.close();
23          factory.close();
24      }
25  }
```

Execute e observe que as consultas são realizadas aos poucos devido ao carregamento em modo LAZY

7. Para testar as Named Queries, acrescente a anotação @NAMEDQUERY na classe AUTOR.

```
@Entity
@NamedQuery(name="Autor.findAll", query="select a from Autor a")
public class Autor {
    ...
}
```

8. Na sequência crie um teste da Named Query definida no exercício anterior.

```
public class TesteNamedQuery {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_jpql");
        EntityManager manager = factory.createEntityManager();

        Query query = manager.createNamedQuery("Autor.findAll");
        List<Autor> autores = query.getResultList();

        for (Autor autor : autores) {
            System.out.println("Autor: " + autor.getNome());
            Collection<Livro> livros = autor.getLivros();

            for (Livro livro : livros) {
                System.out.println("Livro: " + livro.getNome());
                System.out.println("Preço: " + livro.getPreco());
                System.out.println();
            }
            System.out.println();
        }

        manager.close();
        factory.close();
    }
}
```

Execute e observe que as consultas são realizadas aos poucos devido ao carregamento em modo LAZY

9. Acrescente a anotação @NAMEDQUERY na classe LIVRO para definir uma consulta por preço mínimo utilizando parâmetros.

```
@Entity
@NamedQuery(name="Livro.findByPrecoMinimo",
            query="select livro from Livro livro where livro.preco >= :preco")
public class Livro {
    ...
}
```

10. Na sequência crie um teste da Named Query definida no exercício anterior.

```
1  public class TesteParametros {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_jpql");
5          EntityManager manager = factory.createEntityManager();
6
7          Query query = manager.createNamedQuery("Livro.findByPrecoMinimo");
8          query.setParameter("preco", 20.0);
9          List<Livro> livros = query.getResultList();
10
11         for (Livro livro : livros) {
12             System.out.println("Nome: " + livro.getNome());
13             System.out.println("Preço: " + livro.getPreco());
14         }
15
16         manager.close();
17         factory.close();
18     }
19 }
```

4.5 Tipos de Resultado

4.5.1 Lista de Entidades

Uma consulta em JPQL pode devolver uma lista com os objetos de uma entidade que são compatíveis com os filtros da pesquisa. Por exemplo suponha a seguinte consulta:

```
1 String query = "SELECT p FROM Pessoa p";
```

O resultado dessa pesquisa é uma lista com todas as instâncias da entidade PESSOA que foram persistidas. Esse resultado pode ser obtido através do método **getResultList**().

```
1 String query = "SELECT p FROM Pessoa p";
2 Query query = manager.createQuery(query);
3 List<Pessoa> pessoas = query.getResultList();
```

Nesse caso, os objetos da listagem devolvida pela consulta estão no estado managed, ou seja, alterações realizadas no conteúdo desses objetos é sincronizado com o banco de dados de acordo com as regras de sincronização.

4.5.2 Typed Query

```
1 String query = "SELECT p FROM Pessoa p";
2 Query query = manager.createQuery(query);
3 List<Pessoa> pessoas = query.getResultList();
```

O compilador da linguagem Java não verifica a compatibilidade entre a variável e o resultado da consulta. Na consulta acima, o compilador não sabe se o método GETRESULTLIST() devolverá de fato uma lista de pessoas pois ele não processa a string que define a consulta. Sem

ajuda do compilador há mais chances de erros de execução. Por exemplo, a consulta abaixo poderia causar problemas.

```
1 String query = "SELECT p FROM Pessoa p";
2 Query query = manager.createQuery(query);
3 List<Departamento> departamentos = query.getResultList();
```

Para diminuir a chance de erro, podemos utilizar as **Typed Queries**. Com elas o compilador continua não verificando a string da consulta mas passa a assumir o tipo de resultado e verifica todo o código após a criação da consulta.

```
1 String query = "SELECT p FROM Pessoa p";
2 TypedQuery<Pessoa> query = manager.createQuery(query, Pessoa.class);
3 List<Pessoa> pessoas = query.getResultList();
```

4.5.3 Lista de Objetos Comuns

A consulta acima devolve uma lista de pessoas. Dessa forma, teríamos acesso a todos os dados das pessoas dessa listagem. Contudo, muitas vezes, não desejamos todas as informações. Por exemplo, se a nossa aplicação precisa apresentar uma lista dos nomes das pessoas cadastradas não é necessário recuperar nada além dos nomes.

Quando definimos as consultas, podemos determinar o que elas devem trazer de fato do banco de dados. Por exemplo, a consulta abaixo recupera apenas os nomes das pessoas.

```
1 String query = "SELECT p.nome FROM Pessoa p";
2 TypedQuery<String> query = manager.createQuery(query, String.class);
3 List<String> nomes = query.getResultList();
```

4.5.4 Valores Únicos

Suponha que desejamos saber quantas pessoas possuem mais do que 18 anos. Nesse caso, não é necessário trazer mais do que um número do banco de dados. Em outras palavras, o resultado dessa consulta não deve ser uma lista e sim um valor numérico. Para isso podemos aplicar as funções de agregação:

AVG	Calcula a média de um conjunto de números
COUNT	Contabiliza o número de resultados
MAX	Recupera o maior elemento um conjunto de números
MIN	Recupera o menor elemento um conjunto de números
SUM	Calcula a soma de um conjunto de números

A consulta abaixo devolve a quantidade de pessoas persistidas. Observe que é utilizado o método **getSingleResult()** ao invés do **getResultList()** pois o resultado não é uma lista.

```
String query = "SELECT COUNT(p) FROM Pessoa p";
TypedQuery<Long> query = manager.createQuery(query, Long.class);
Long count = query.getSingleResult();
```

A consulta abaixo devolve a maior idade entre as pessoas persistidas.

```
String query = "SELECT MAX(p.idade) FROM Pessoa p";
TypedQuery<Integer> query = manager.createQuery(query, Integer.class);
Integer maiorIdade = query.getSingleResult();
```

4.5.5 Resultados Especiais

Algumas consultas possuem resultados complexos. Por exemplo, suponha que desejamos obter uma listagem com os nomes dos funcionários e o nome do departamento que o funcionário trabalha.

```
"SELECT f.nome, f.departamento.nome FROM Funcionario f";
```

Nesse caso, o resultado será uma lista de array de Object. Para manipular essa lista devemos lidar com o posicionamento dos dados nos arrays.

```
String query = "SELECT f.nome, f.departamento.nome FROM Funcionario f";
Query query = manager.createQuery(query);
List<Object[]> lista = query.getResultList();

for(Object[] tupla : lista) {
    System.out.println("Funcionário: " + tupla[0]);
    System.out.println("Departamento: " + tupla[1]);
}
```

4.5.6 Operador NEW

Para contornar a dificuldade de lidar com o posicionamento dos dados nos arrays, podemos criar uma classe para modelar o resultado da nossa consulta e aplicar o operador **NEW** no código JPQL.

```
package resultado;

class FuncionarioDepartamento {
    private String funcionarioNome;
    private String departamentoNome;

    public FuncionarioDepartamento(String funcionarioNome, String departamentoNome) {
        this.funcionarioNome = funcionarioNome;
        this.departamentoNome = departamentoNome;
    }

    // GETTERS AND SETTERS
}
```

```
String query = "SELECT NEW resultado.FuncionarioDepartamento(f.nome,
                    f.departamento.nome) FROM Funcionario f";

Query<FuncionarioDepartamento> query = manager.createQuery(query);

List<FuncionarioDepartamento> resultados = query.getResultList();

for(FuncionarioDepartamento resultado : resultados) {
    System.out.println("Funcionário: " + resultado.getFuncionarioNome());
    System.out.println("Departamento: " + resultado.getDepartamentoNome());
}
```

4.6 Exercícios

11. Teste o recurso de Typed Query utilizando a Named Query AUTOR.FINDALL.

```
public class TesteTypedQuery {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_jpql");
        EntityManager manager = factory.createEntityManager();

        TypedQuery<Autor> query = manager.createNamedQuery("Autor.findAll", Autor.class);
        List<Autor> autores = query.getResultList();

        for (Autor autor : autores) {
            System.out.println("Autor: " + autor.getNome());
        }

        manager.close();
        factory.close();
    }
}
```

Observe que não há mais warnings

12. Crie um teste para recuperar somente os nomes dos livros cadastrados no banco de dados. Adicione a seguinte classe no pacote **testes**.

```
public class TesteConsultaObjetosComuns {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_jpql");
        EntityManager manager = factory.createEntityManager();

        TypedQuery<String> query = manager.createQuery("select livro.nome from Livro livro", String.class);
        List<String> nomes = query.getResultList();

        for (String nome : nomes) {
            System.out.println(nome);
        }

        manager.close();
        factory.close();
    }
}
```

13. Crie um teste para recuperar o valor da média dos preços dos livros. Adicione a seguinte classe no pacote **testes**.

```
1  public class TesteConsultaLivroPrecoMedio {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_jpql");
5          EntityManager manager = factory.createEntityManager();
6
7          TypedQuery<Double> query = manager.createQuery("select avg(livro.preco) from ↩
               Livro livro", Double.class);
8          Double precoMedio = query.getSingleResult();
9
10         System.out.println("Preço médio: " + precoMedio);
11
12         manager.close();
13         factory.close();
14     }
15 }
```

14. Crie mais duas entidades no pacote **modelo**.

```
1  @Entity
2  public class Departamento {
3
4      @Id @GeneratedValue
5      private Long id;
6
7      private String nome;
8
9      // GETTERS AND SETTERS
10 }
```

```
1  @Entity
2  public class Funcionario {
3
4      @Id @GeneratedValue
5      private Long id;
6
7      private String nome;
8
9      @ManyToOne
10     private Departamento departamento;
11
12     // GETTERS AND SETTERS
13 }
```

15. Adicione alguns funcionários e departamentos.

```
public class AdicionaFuncionarioDepartamento {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_jpql");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        Departamento d = new Departamento();
        d.setNome("Treinamentos");

        Funcionario f = new Funcionario();
        f.setNome("Rafael Cosentino");

        f.setDepartamento(d);

        manager.persist(f);
        manager.persist(d);

        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

16. Crie um teste para recuperar os nomes dos funcionários e os nomes dos seus respectivos departamentos. Adicione a seguinte classe no pacote **testes**.

```
public class TesteBuscaFuncionarioDepartamento {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_jpql");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        Query query = manager
                .createQuery("select f.nome, f.departamento.nome from Funcionario f");
        List<Object[]> lista = query.getResultList();

        for (Object[] tupla : lista) {
            System.out.println("Funcionário: " + tupla[0]);
            System.out.println("Departamento: " + tupla[1]);
        }

        manager.close();
        factory.close();
    }
}
```

17. Crie no pacote **modelo** uma classe para melhorar a manipulação da consulta dos nomes dos funcionários e nomes dos seus respectivos departamentos.

```
1  public class FuncionarioDepartamento {
2      private String funcionario;
3      private String departamento;
4
5      public FuncionarioDepartamento(String funcionario, String departamento) {
6          this.funcionario = funcionario;
7          this.departamento = departamento;
8      }
9
10     // GETTERS AND SETTERS
11 }
```

18. Altere a classe **TesteBuscaFuncionarioDepartamento** para que ela utilize o operador **NEW** da JPQL.

```
1  public class TesteBuscaFuncionarioDepartamento {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_jpql");
5          EntityManager manager = factory.createEntityManager();
6
7          manager.getTransaction().begin();
8
9          Query query = manager
10                 .createQuery("select new modelo.FuncionarioDepartamento(f.nome, f.↩
                       departamento.nome) from Funcionario f");
11         List<FuncionarioDepartamento> lista = query.getResultList();
12
13         for (FuncionarioDepartamento fd : lista) {
14             System.out.println("Funcionário: " + fd.getFuncionario());
15             System.out.println("Departamento: " + fd.getDepartamento());
16         }
17
18         manager.close();
19         factory.close();
20     }
21 }
```

4.7 Paginação

Supondo que exista uma grande quantidade de livros cadastrados no banco de dados, buscar todos os livros sem nenhum filtro vai sobrecarregar a memória utilizada pela aplicação. Nesses casos, podemos aplicar o conceito de **paginação** para obter os livros aos poucos. A paginação é realizada através dos métodos **setFirstResult()** e **setMaxResults()**.

```
1  TypedQuery<Livro> query = manager.createQuery("select livro from Livro livro", Livro.↩
       class);
2  query.setFirstResult(10);
3  query.setMaxResults(20);
4  List<Livro> livros = query.getResultList();
```

4.8 Exercícios

19. Teste o recurso de paginação das consultas. Adicione a seguinte classe no pacote **testes**.

```
1  public class TesteBuscaPaginada {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_jpql");
5          EntityManager manager = factory.createEntityManager();
6
7          TypedQuery<Livro> query = manager.createQuery("select livro from Livro livro",
8                  Livro.class);
9
10         query.setFirstResult(2);
11         query.setMaxResults(3);
12         List<Livro> livros = query.getResultList();
13
14         for (Livro livro : livros) {
15             System.out.println("Livro: " + livro.getNome());
16         }
17
18         manager.close();
19         factory.close();
20     }
21 }
```

4.9 Operadores

As consultas em JPQL utilizam alguns tipos de operadores.

4.9.1 Condicionais

- Menor(<)

```
1  String query = "SELECT p FROM Pessoa p WHERE p.idade < :idade";
```

- Maior(>)

```
1  String query = "SELECT p FROM Pessoa p WHERE p.idade > :idade";
```

- Menor Igual(<=)

```
1  String query = "SELECT p FROM Pessoa p WHERE p.idade <= :idade";
```

- Maior Igual(>=)

```
String query = "SELECT p FROM Pessoa p WHERE p.idade >= :idade";
```

- Igual(=)

```
String query = "SELECT p FROM Pessoa p WHERE p.idade = :idade";
```

- Diferente(<>)

```
String query = "SELECT p FROM Pessoa p WHERE p.idade <> :idade";
```

- IS NULL

```
String query = "SELECT p FROM Pessoa p WHERE p.nome IS NULL";
```

- IS NOT NULL

```
String query = "SELECT p FROM Pessoa p WHERE p.nome IS NOT NULL";
```

- BETWEEN

```
String query = "SELECT p FROM Pessoa p WHERE p.idade BETWEEN :minimo AND :maximo";
```

- NOT BETWEEN

```
String query = "SELECT p FROM Pessoa p WHERE p.idade NOT BETWEEN :minimo AND :maximo";
```

- AND

```
String query = "SELECT p FROM Pessoa p WHERE p.nome IS NOT NULL AND p.idade >= :idade";
```

- OR

```
String query = "SELECT p FROM Pessoa p WHERE p.nome IS NOT NULL OR p.idade >= :idade";
```

- NOT

```
String query = "SELECT p FROM Pessoa p WHERE NOT (p.idade >= :idade)";
```

- MEMBER OF

```
String query = "SELECT f FROM Funcionario f WHERE f MEMBER OF f.empresa.diretoria";
```

- NOT MEMBER OF

```
String query = "SELECT f FROM Funcionario f WHERE f NOT MEMBER OF f.empresa.diretoria";
```

- IS EMPTY

```
String query = "SELECT a FROM Autor a WHERE a.livros IS EMPTY";
```

- IS NOT EMPTY

```
String query = "SELECT a FROM Autor a WHERE a.livros IS NOT EMPTY";
```

- EXISTS

```
String query = "SELECT d FROM Departamento d WHERE EXISTS (SELECT f FROM FUNCIONARIO f ↩
    WHERE f.departamento = d)";
```

- NOT EXISTS

```
String query = "SELECT d FROM Departamento d WHERE NOT EXISTS (SELECT f FROM ↩
    FUNCIONARIO f WHERE f.departamento = d)";
```

- LIKE

```
String query = "SELECT a FROM Autor a WHERE a.nome LIKE Patrick%";
```

- NOT LIKE

```
String query = "SELECT a FROM Autor a WHERE a.nome NOT LIKE Patrick%";
```

- IN

```
String query = "SELECT a FROM Autor a WHERE a.nome IN ('Patrick Cullen', 'Fraser Seitel')";
```

- NOT IN

```
String query = "SELECT a FROM Autor a WHERE a.nome NOT IN ('Patrick Cullen', 'Fraser Seitel')";
```

4.9.2 Escalares

- ALL

```
String query = "SELECT livro FROM Livro livro WHERE livro.preco >= ALL(SELECT livro.preco FROM Livro livro)";
```

- ANY

```
String query = "SELECT livro FROM Livro livro WHERE livro.preco >= ANY(SELECT livro.preco FROM Livro livro)";
```

- SOME

```
String query = "SELECT livro FROM Livro livro WHERE livro.preco >= ANY(SELECT livro.preco FROM Livro livro)";
```

4.9.3 Agregadores

- AVG

```
String query = "SELECT AVG(livro.preco) FROM Livro livro";
```

- SUM

```
String query = "SELECT SUM(livro.preco) FROM Livro livro";
```

- MIN

```
String query = "SELECT MIN(livro.preco) FROM Livro livro";
```

- MAX

```
String query = "SELECT MAX(livro.preco) FROM Livro livro";
```

- COUNT

```
String query = "SELECT COUNT(livro) FROM Livro livro";
```

4.9.4 Funções

ABS: Calcula o valor absoluto de um número.

CONCAT: Concatena strings.

CURRENT_DATE: Recupera a data atual.

CURRENT_TIME: Recupera o horário atual.

CURRENT_TIMESTAMP: Recupera a data e o horário atuais.

LENGTH: Calcula o número de caracteres de uma string.

LOCATE: Localiza uma string dentro de outra.

LOWER: Deixa as letras de uma string minúsculas.

MOD: Calcula o resto da divisão entre dois números.

SIZE: Calcula o número de elementos de uma coleção.

SQRT: Calcula a raiz quadrada de um número.

SUBSTRING: Recupera um trecho de uma string.

UPPER: Deixa as letras de uma string maiúsculas.

TRIM: Elimina espaços do início e fim de uma string.

4.9.5 ORDER BY

Podemos ordenar o resultado de uma consulta através do operador **ORDER BY**.

```
1 String query = "SELECT livro FROM Livro livro ORDER BY livro.preco ASC";
```

4.10 Exemplos

1. Suponha que seja necessário selecionar os livros mais baratos. Em outras palavras, devemos selecionar os livros tais quais não exista nenhum outro livro com preço menor.

```
1 SELECT livro1
2 FROM Livro livro1
3 WHERE NOT EXISTS (
4     SELECT livro2
5     FROM Livro livro2
6     WHERE livro1.preco > livro2.preco
7 )
```

2. Suponha que seja necessário selecionar os livros mais baratos de um determinado autor.

```
1  SELECT livro1
2  FROM Livro livro1, Autor autor1
3  WHERE autor1.nome = :nome and
4  livro1 MEMBER OF autor1.livros and
5  NOT EXISTS (
6      SELECT livro2
7      FROM Livro livro2, Autor autor2
8      WHERE autor2 = autor1 and
9      livro2 MEMBER OF autor2.livros and
10     livro1.preco > livro2.preco
11 )
```

3. Suponha que seja necessário listar os livros em ordem decrescente em relação aos preços.

```
1 SELECT livro FROM Livro livro ORDER BY livro.preco DESC
```

4. Suponha que seja necessário selecionar os autores com mais livros.

```
1 SELECT autor1
2 FROM Autor autor1
3 WHERE NOT EXISTS (
4     SELECT autor2
5     FROM Autor autor2
6     WHERE SIZE(autor2.livros) > SIZE(autor1.livros)
7 )
```

4.11 Referências

Para conhecer mais sobre a sintaxe do JPQL consulte essas referências:

- http://openjpa.apache.org/builds/latest/docs/manual/jpa_overview_query.html
- http://openjpa.apache.org/builds/latest/docs/manual/jpa_langref.html

Capítulo 5

Criteria

O primeiro mecanismo definido pela especificação JPA 2 para realizar consultas orientadas a objetos utiliza a linguagem JPQL. O segundo mecanismo utiliza um conjunto de classes e interfaces e funciona basicamente como uma biblioteca. O nome dessa segunda abordagem é Criteria API.

5.1 Necessidade

A primeira questão a ser discutida nesse momento é a necessidade de dois mecanismos de definição de consultas. Um mecanismo só não seria suficiente?

Teoricamente, qualquer consulta definida com JPQL pode ser definida com Criteria API e vice versa. Contudo, algumas consultas são mais facilmente definidas em JPQL enquanto outras mais facilmente definidas em Criteria API.

As consultas que não dependem de informações externas (ex. buscar todos os livros cadastrados no banco de dados sem nenhum filtro) são mais facilmente definidas em JPQL.

As consultas que dependem de informações externas fixas (ex. buscar todos os livros cadastrados no banco de dados que possuam preço maior do que um valor definido pelos usuários) também são mais facilmente criadas em JPQL utilizando os parâmetros de consulta.

Agora, as consultas que dependem de informações externas não fixas são mais facilmente definidas em Criteria API. Por exemplo, suponha uma busca avançada de livros na qual há muitos campos opcionais. Essa consulta depende de informações externas não fixas já que nem todo campo fará parte da consulta todas as vezes que ela for executada.

5.2 Estrutura Geral

As duas interfaces principais da Criteria API são: **CriteriaBuilder** e **CriteriaQuery**. As consultas em Criteria API são construídas por um Criteria Builder que é obtido através de um Entity Manager.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
```

A definição de uma consulta em Criteria API começa na chamada do método CREATE-

QUERY() de um Criteria Builder. O parâmetro desse método indica o tipo esperado do resultado da consulta.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
2 CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
```

Para definir o espaço de dados que a consulta deve considerar, devemos utilizar o método **from()** da Criteria Query. Este método devolve uma raiz do espaço de dados considerado pela pesquisa.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
2 CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
3 // DEFININDO O ESPAÇO DE DADOS DA CONSULTA
4 Root<Autor> a = c.from(Autor.class);
```

Para definir o que queremos selecionar do espaço de dados da consulta, devemos utilizar o método **select()** da Criteria Query.

```
1 CriteriaBuilder cb = manager.getCriteriaBuilder();
2 CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
3 // DEFININDO O ESPAÇO DE DADOS DA CONSULTA
4 Root<Autor> a = c.from(Autor.class);
5 // SELECIONANDO UMA RAIZ DO ESPAÇO DE DADOS
6 c.select(a);
```

Com a consulta em Criteria API definida, devemos invocar um Entity Manager para poder executá-la da seguinte forma.

```
1 TypedQuery<Autor> query = manager.createQuery(c);
2 List<Autor> autores = query.getResultList();
```

5.3 Exercícios

1. Crie um projeto no eclipse chamado **Criteria**. Copie a pasta **lib** do projeto **EntityManager** para o projeto **Criteria**. Depois selecione todos os jar's e os adicione no classpath.

2. Abra o **MySQL Query Browser** e apague a base de dados **K21_criteria** se existir. Depois crie a base de dados **K21_criteria**.

3. Copie a pasta **META-INF** do projeto **EntityManager** para dentro da pasta **src** do projeto **Criteria**. Altere o arquivo **persistence.xml** do projeto **Criteria**, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <persistence-unit name="K21_criteria"
        transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <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="root" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/K21_criteria" />
        </properties>
    </persistence-unit>
</persistence>
```

4. Crie um pacote chamado **modelo** no projeto **Criteria** e adicione as seguintes classes:

```
@Entity
public class Autor {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    @ManyToMany
    private Collection<Livro> livros = new ArrayList<Livro>();

    // GETTERS E SETTERS
}
```

```
@Entity
public class Livro {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    private Double preco;

    // GETTERS E SETTERS
}
```

5. Carregue o banco de dados com as informações de alguns livros e autores. Adicione a seguinte classe em um novo pacote chamado **testes**. **Você pode copiar a classe PopulaBanco criada no projeto JPQL e alterar a unidade de persistência**.

```
1  public class PopulaBanco {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_criteria");
5          EntityManager manager = factory.createEntityManager();
6
7          manager.getTransaction().begin();
8
9          Livro livro1 = new Livro();
10         livro1.setNome("The Battle for Your Mind");
11         livro1.setPreco(20.6);
12         manager.persist(livro1);
13
14         Livro livro2 = new Livro();
15         livro2.setNome("Differentiate or Die");
16         livro2.setPreco(15.8);
17         manager.persist(livro2);
18
19         Livro livro3 = new Livro();
20         livro3.setNome("How to Transform Your Ideas");
21         livro3.setPreco(32.7);
22         manager.persist(livro3);
23
24         Livro livro4 = new Livro();
25         livro4.setNome("Digital Fortress");
26         livro4.setPreco(12.9);
27         manager.persist(livro4);
28
29         Livro livro5 = new Livro();
30         livro5.setNome("Marketing in an Era of Competition, Change and Crisis");
31         livro5.setPreco(26.8);
32         manager.persist(livro5);
33
34         Autor autor1 = new Autor();
35         autor1.setNome("Patrick Cullen");
36         autor1.getLivros().add(livro2);
37         autor1.getLivros().add(livro4);
38         manager.persist(autor1);
39
40         Autor autor2 = new Autor();
41         autor2.setNome("Fraser Seitel");
42         autor2.getLivros().add(livro3);
43         manager.persist(autor2);
44
45         Autor autor3 = new Autor();
46         autor3.setNome("Al Ries");
47         autor3.getLivros().add(livro1);
48         manager.persist(autor3);
49
50         Autor autor4 = new Autor();
51         autor4.setNome("Jack Trout");
52         autor4.getLivros().add(livro1);
53         autor4.getLivros().add(livro2);
54         autor4.getLivros().add(livro5);
55         manager.persist(autor4);
56
57         Autor autor5 = new Autor();
58         autor5.setNome("Steve Rivkin");
59         autor5.getLivros().add(livro2);
60         autor5.getLivros().add(livro3);
61         autor5.getLivros().add(livro5);
62         manager.persist(autor5);
63
64         manager.getTransaction().commit();
65
66         manager.close();
67         factory.close();
68     }
69 }
```

6. Teste a Criteria API criando uma consulta para recuperar todos os livros cadastrados no banco de dados. Crie a seguinte classe no pacote **testes**.

```
1  public class TesteCriteria {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_criteria");
5          EntityManager manager = factory.createEntityManager();
6
7
8          CriteriaBuilder cb = manager.getCriteriaBuilder();
9          CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
10         Root<Livro> l = c.from(Livro.class);
11         c.select(l);
12
13         TypedQuery<Livro> query = manager.createQuery(c);
14         List<Livro> livros = query.getResultList();
15
16         for (Livro livro : livros) {
17             System.out.println(livro.getNome());
18         }
19     }
20 }
```

7. (Opcional) Crie uma consulta utilizando a Criteria API para recuperar todos os autores.

5.4 Tipos de Resultados

5.4.1 Lista de Entidades

O resultado de uma consulta em Criteria API pode ser uma lista com os objetos de uma entidade que são compatíveis com os filtros da pesquisa.

```
1 CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
2 Root<Livro> livro = c.from(Livro.class);
3 c.select(livro);
```

O resultado da consulta acima é uma lista com todos os objetos da classe LIVRO.

5.4.2 Lista de Objetos Comuns

Muitas vezes, não queremos selecionar todas as informações dos objetos pesquisados. Por exemplo, suponha que seja necessário gerar uma lista com os nomes dos livros cadastrados no banco de dados. Através do método SELECT() podemos definir o que deve ser recuperado do banco de dados.

```
1 CriteriaQuery<String> c = cb.createQuery(String.class);
2 Root<Livro> livro = c.from(Livro.class);
3 c.select(livro.<String>get("nome"));
```

A consulta acima recupera apenas os nomes dos livros. O resultado dessa pesquisa é uma lista de strings.

5.4.3 Valores Únicos

Algumas consultas possuem como resultado valores únicos. Por exemplo, suponha que queremos obter a média dos preços dos livros cadastrados no banco de dados ou a quantidade de livros de uma determinada editora. Nesse tipo de consulta, devemos apenas trazer do banco de dados um valor único calculado no próprio banco.

```
1 CriteriaQuery<Double> c = cb.createQuery(Double.class);
2 Root<Livro> l = c.from(Livro.class);
3 c.select(cb.avg(l.<Double>get("preco")));
```

A consulta acima devolve a média dos preços dos livros cadastrados no banco de dados. Nessa consulta, foi utilizada uma função de agregação. Veja a lista dessas funções:

avg()	Calcula a média de um conjunto de números
count()	Contabiliza o número de resultados
max() e greatest()	Recupera o maior elemento um conjunto de números
min() e least()	Recupera o menor elemento um conjunto de números
sum(), sumAsLong() e sumAsDouble()	Calcula a soma de um conjunto de números

5.4.4 Resultados Especias

Podemos selecionar múltiplos atributos de uma entidade em uma consulta em Criteria API. Por exemplo, podemos montar uma listagem com os nomes e preços dos livros cadastrados no banco de dados. Para selecionar múltiplos atributos, devemos utilizar o método MULTISELECT().

```
1 CriteriaQuery<Object[]> c = cb.createQuery(Object[].class);
2 Root<Livro> l = c.from(Livro.class);
3 c.multiselect(l.<String>get("nome"), l.<Double>get("preco"));
```

O resultado da consulta acima é uma lista de array de Object. Para manipular esse resultado, temos que utilizar a posição dos dados dos arrays da lista.

```
1 TypedQuery<Object[]> query = manager.createQuery(c);
2 List<Object[]> resultado = query.getResultList();
3
4 for (Object[] registro : resultado) {
5     System.out.println("Livro: " + registro[0]);
6     System.out.println("Preço: " + registro[1]);
7 }
```

Também podemos utilizar a interface TUPLE para não trabalhar com posicionamento de dados em arrays. Nessa abordagem, devemos aplicar "apelidos"para os itens selecionados através do método ALIAS().

```
1  CriteriaQuery<Tuple> c = cb.createQuery(Tuple.class);
2  Root<Livro> l = c.from(Livro.class);
3  c.multiselect(l.<String>get("nome").alias("livro.nome"), l.<Double>get("preco").alias("↩
       livro.preco"));
4
5  TypedQuery<Tuple> query = manager.createQuery(c);
6  List<Tuple> resultado = query.getResultList();
7
8  for (Tuple registro : resultado) {
9      System.out.println("Livro: " + registro.get("livro.nome"));
10     System.out.println("Preço: " + registro.get("livro.preco"));
11 }
```

5.5 Exercícios

8. Recupere uma listagem com os nomes dos livros cadastrados no banco de dados. Adicione a seguinte classe no pacote **testes**.

```
1  public class ListaNomesDosLivros {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_criteria");
5          EntityManager manager = factory.createEntityManager();
6
7          CriteriaBuilder cb = manager.getCriteriaBuilder();
8          CriteriaQuery<String> c = cb.createQuery(String.class);
9          Root<Livro> livro = c.from(Livro.class);
10         c.select(livro.<String>get("nome"));
11
12         TypedQuery<String> query = manager.createQuery(c);
13         List<String> nomes = query.getResultList();
14
15         for (String nome : nomes) {
16             System.out.println("Livro: " + nome);
17         }
18
19         manager.close();
20         factory.close();
21     }
22 }
```

9. Recupere a média dos valores dos livros cadastrados no banco de dados. Adicione a seguinte classe no pacote **testes**.

```
public class CalculaMediaDosPrecosDosLivros {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_criteria");
        EntityManager manager = factory.createEntityManager();

        CriteriaBuilder cb = manager.getCriteriaBuilder();
        CriteriaQuery<Double> c = cb.createQuery(Double.class);
        Root<Livro> l = c.from(Livro.class);
        c.select(cb.avg(l.<Double>get("preco")));

        TypedQuery<Double> query = manager.createQuery(c);
        Double media = query.getSingleResult();

        System.out.println("Média: " + media);

        manager.close();
        factory.close();
    }
}
```

10. Recupere os nomes e os preços dos livros cadastrados no banco de dados. Adicione a seguinte classe no pacote **testes**.

```
public class ConsultaNomePrecoDosLivros {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_criteria");
        EntityManager manager = factory.createEntityManager();

        CriteriaBuilder cb = manager.getCriteriaBuilder();
        CriteriaQuery<Object[]> c = cb.createQuery(Object[].class);
        Root<Livro> l = c.from(Livro.class);
        c.multiselect(l.<String>get("nome"), l.<Double>get("preco"));

        TypedQuery<Object[]> query = manager.createQuery(c);
        List<Object[]> resultado = query.getResultList();

        for (Object[] registro : resultado) {
            System.out.println("Livro: " + registro[0]);
            System.out.println("Preço: " + registro[1]);
        }

        manager.close();
        factory.close();
    }
}
```

11. Altere a classe do exercício anterior para que ela utilize a interface **Tuple**.

```
1  public class ConsultaNomePrecoDosLivros {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_criteria");
5          EntityManager manager = factory.createEntityManager();
6
7          CriteriaBuilder cb = manager.getCriteriaBuilder();
8          CriteriaQuery<Tuple> c = cb.createQuery(Tuple.class);
9          Root<Livro> l = c.from(Livro.class);
10         c.multiselect(l.<String>get("nome").alias("livro.nome"), l.<Double>get("preco")↩
               .alias("livro.preco"));
11
12         TypedQuery<Tuple> query = manager.createQuery(c);
13         List<Tuple> resultado = query.getResultList();
14
15         for (Tuple registro : resultado) {
16             System.out.println("Livro: " + registro.get("livro.nome"));
17             System.out.println("Preço: " + registro.get("livro.preco"));
18         }
19
20         manager.close();
21         factory.close();
22     }
23 }
```

5.6 Filtros e Predicados

Podemos definir filtros para as consultas através da criação de **Predicates** e do método **where()** de uma Criteria Query. Os predicates são condições que devem ser satisfeitas para que uma informação seja adicionada no resultado de uma consulta. E eles são criados por um Criteria Builder.

```
1  CriteriaBuilder cb = manager.getCriteriaBuilder();
2  CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
3  Root<Autor> a = c.from(Autor.class);
4  c.select(a);
5
6  // CRIANDO UM PREDICATE
7  Predicate predicate = cb.equal(a.get("nome"), "Patrick Cullen");
8  // ASSOCIANDO O PREDICATE A CONSULTA
9  c.where(predicate);
```

5.7 Exercícios

12. Teste a Criteria API criando uma consulta para recuperar todos os livros cadastrados no banco de dados. Crie a seguinte classe no pacote **testes**.

```
1  public class TestePredicate {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_criteria");
5          EntityManager manager = factory.createEntityManager();
6
7          CriteriaBuilder cb = manager.getCriteriaBuilder();
8          CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
9          Root<Livro> l = c.from(Livro.class);
10         c.select(l);
11
12         Predicate predicate = cb.equal(l.get("nome"), "The Battle for Your Mind");
13         c.where(predicate);
14
15         TypedQuery<Livro> query = manager.createQuery(c);
16         List<Livro> livros = query.getResultList();
17
18         for (Livro livro : livros) {
19             System.out.println(livro.getId());
20             System.out.println(livro.getNome());
21             System.out.println(livro.getPreco());
22         }
23     }
24 }
```

5.8 Lista de Predicados

- equal()

```
1  cb.equal(livro.get("nome"), "The Battle for Your Mind");
```

- and()

```
1  cb.and(cb.equal(livro.get("nome"), "Noites"), cb.equal(livro.get("editora"), "Saraiva")↩
       );
```

- or()

```
1  cb.or(cb.equal(livro.get("nome"), "Noites"), cb.equal(livro.get("editora"), "Saraiva"))↩
       ;
```

- notEqual()

```
1  cb.notEqual(livro.get("nome"), "The Battle for Your Mind");
```

- not()

```
1 cb.not(cb.equal(livro.get("nome"), "The Battle for Your Mind"));
```

- greaterThan(), gt()

```
1 cb.gt(livro.<Double>get("preco"), 20.0);
```

ou

```
1 cb.greaterThan(livro.<Double>get("preco"), 20.0);
```

- greaterThanOrEqualTo(), ge()

```
1 cb.ge(livro.<Double>get("preco"), 20.0);
```

ou

```
1 cb.greaterThanOrEqualTo(livro.<Double>get("preco"), 20.0);
```

- lessThan(), lt()

```
1 cb.lt(livro.<Double>get("preco"), 20.0);
```

ou

```
1 cb.lessThan(livro.<Double>get("preco"), 20.0);
```

- lessThanOrEqualTo(), le()

```
1 cb.le(livro.<Double>get("preco"), 20.0);
```

ou

```
1 cb.lessThanOrEqualTo(livro.<Double>get("preco"), 20.0);
```

- between()

```
1 cb.between(livro.<Double>get("preco"), 20.0, 30.0);
```

- isNull()

```
1 cb.isNull(livro.get("nome"));
```

- isNotNull()

```
1 cb.isNotNull(livro.get("nome"));
```

- isEmpty()

```
1 cb.isEmpty(autor.<Collection<Livro>>get("livros"));
```

- isNotEmpty()

```
1 cb.isNotEmpty(autor.<Collection<Livro>>get("livros"));
```

- isMember()

```
1 cb.isMember(livro, livro.<Editora>get("editora").<Collection<Livro>>get("maisVendidos")↩
    );
```

- isNotMember()

```
1 cb.isNotMember(livro, livro.<Editora>get("editora").<Collection<Livro>>get("↩
    maisVendidos"));
```

- like()

```
1 cb.like(livro.<String>get("nome"), "%Battle%");
```

- notLike()

```
1 cb.notLike(livro.<String>get("nome"), "%Battle%");
```

- in()

```
1 cb.in(livro.<String>get("editora")).value("Saraiva").value("Moderna");
```

ou

```
1 livro.<String>get("editora").in("Saraiva","Moderna");
```

5.9 Funções

- abs()

```
1 cb.abs(livro.<Double>get("preco"));
```

- concat()

```
1 cb.concat(livro1.<String>get("nome"), livro2.<String>get("nome"));
```

- currentDate()

```
1 cb.currentDate();
```

- currentTime()

```
1 cb.currentTime();
```

- currentTimestamp()

```
1 cb.currentTimestamp();
```

- length()

```
cb.length(livro.<String>get("nome"));
```

- locate()

```
cb.locate(livro1.<String>get("nome"), livro2.<String>get("nome"));
```

- lower()

```
cb.lower(livro.<String>get("nome"));
```

- mod()

```
cb.mod(autor1.<Integer>get("idade"), autor2.<Integer>get("idade"));
```

- size()

```
cb.size(autor.<Collection<Livro>>get("livros"));
```

- sqrt()

```
cb.sqrt(autor.<Integer>get("idade"));
```

- substring()

```
cb.substring(livro.<String>get("nome"), 3, 5);
```

- upper()

```
cb.upper(livro.<String>get("nome"));
```

- trim()

```
1 cb.trim(livro.<String>get("nome"));
```

5.10 Ordenação

As consultas em Criteria API também podem ser ordenadas, basta utilizarmos o método ORDERBY()

```
1 CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
2 Root<Livro> livro = c.from(Livro.class);
3 c.select(livro);
4
5 c.orderBy(cb.desc(livro.<Double>get("preco")));
```

5.11 Subqueries

Algumas consultas necessitam de consultas auxiliares. Por exemplo, para encontrar o livro mais caro cadastrado no banco de dados devemos criar uma consulta auxiliar para poder comparar, dois a dois, os preços dos livros.

Uma consulta auxiliar ou uma **subquery** é criada através do método SUBQUERY(). No exemplo, abaixo criamos uma consulta e uma subconsulta para encotrar o livro mais caro.

```
1  CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
2  Root<Livro> livro1 = c.from(Livro.class);
3  c.select(livro1);
4
5  Subquery<Livro> subquery = c.subquery(Livro.class);
6  Root<Livro> livro2 = subquery.from(Livro.class);
7  subquery.select(a2);
8
9  Predicate predicate = cb.greaterThan(livro2.<Double>get("preco"), livro1.<Double>get("↩
       preco"));
10 subquery.where(predicate);
11
12 Predicate predicate2 = cb.not(cb.exists(subquery));
13 c.where(predicate2);
```

5.12 Exemplos

1. Suponha que seja necessário selecionar os livros mais baratos. Em outras palavras, devemos selecionar os livros tais quais não exista nenhum outro livro com preço menor.

```
1  CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
2  Root<Livro> livro1 = c.from(Livro.class);
3  c.select(livro1);
4
5  Subquery<Livro> subquery = c.subquery(Livro.class);
6  Root<Livro> livro2 = subquery.from(Livro.class);
7  subquery.select(livro2);
8  Predicate gt = cb.gt(livro1.<Double>get("preco"), livro2.<Double>get("preco"));
9  subquery.where(gt);
10
11 Predicate notExists = cb.not(cb.exists(subquery));
12 c.where(notExists);
```

2. Suponha que seja necessário selecionar os livros mais baratos de um determinado autor.

```
1  CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
2  Root<Livro> livro1 = c.from(Livro.class);
3  Root<Autor> autor1 = c.from(Autor.class);
4  c.select(livro1);
5
6  Subquery<Livro> subquery = c.subquery(Livro.class);
7  Root<Livro> livro2 = subquery.from(Livro.class);
8  Root<Autor> autor2 = subquery.from(Autor.class);
9  subquery.select(livro2);
10 Predicate isMember1 = cb.isMember(livro2, autor2.<Collection<Livro>>get("livros"));
11 Predicate equal1 = cb.equal(autor2, autor1);
12 Predicate gt = cb.gt(livro1.<Double>get("preco"), livro2.<Double>get("preco"));
13 Predicate predicate = cb.and(isMember1,equal1,gt);
14 subquery.where(predicate);
15
16 Predicate notExists = cb.not(cb.exists(subquery));
17 Predicate equal2 = cb.equal(autor1.<String>get("nome"), "Jack Trout");
18 Predicate isMember2 = cb.isMember(livro1, autor1.<Collection<Livro>>get("livros"));
19 Predicate predicate2 = cb.and(isMember2,equal2,notExists);
20 c.where(predicate2);
```

3. Suponha que seja necessário listar os livros em ordem decrescente em relação aos preços.

```
1  CriteriaQuery<Livro> c = cb.createQuery(Livro.class);
2  Root<Livro> livro = c.from(Livro.class);
3  c.select(livro);
4  c.orderBy(cb.desc(livro.<Double>get("preco")));
```

4. Suponha que seja necessário selecionar os autores com mais livros.

```
1  CriteriaQuery<Autor> c = cb.createQuery(Autor.class);
2  Root<Autor> autor1 = c.from(Autor.class);
3  c.select(autor1);
4
5  Subquery<Autor> subquery = c.subquery(Autor.class);
6  Root<Autor> autor2 = subquery.from(Autor.class);
7  subquery.select(autor2);
8  Predicate gt = cb.gt(cb.size(autor2.<Collection<Livro>>get("livros")), cb.size(autor1.<↩
       Collection<Livro>>get("livros")));
9  subquery.where(gt);
10
11 Predicate notExists = cb.not(cb.exists(subquery));
12 c.where(notExists);
```

Capítulo 6

Tópicos Avançados

6.1 Operações em Lote - Bulk Operations

Para atualizar ou remover registros das tabelas do banco de dados, a abordagem mais comum é realizar uma consulta trazendo para a memória os objetos referentes aos registros que devem ser modificados ou removidos.

```
1 Pessoa p = manager.find(Pessoa.class, 1L);
2 p.setNome("Rafael Cosentino");
```

```
1 Pessoa p = manager.find(Pessoa.class, 1L);
2 manager.remove(p);
```

Em alguns casos essa abordagem não é a mais eficiente. Por exemplo, suponha que uma aplicação que controla os produtos de uma loja virtual necessite atualizar os preços de todos os produtos com uma taxa fixa.

```
1 TypedQuery<Produto> query = manager.createNamedQuery("Produto.findAll");
2 List<Produto> produtos = query.getResultList();
3 for(Produto p : produtos) {
4     double preco = p.getPreco();
5     p.setPreco(preco * 1.1); // 10% de aumento
6 }
```

Todos os produtos seriam carregados na aplicação sobrecarregando a rede pois uma grande quantidade de dados seria transferida do banco de dados para a aplicação e a memória pois muitos objetos seriam criados pelo provedor JPA.

A abordagem mais eficiente nesse caso é realizar uma operação em lote (bulk operation). Uma operação em lote é executada no banco de dados sem transferir os dados dos registros para a memória da aplicação.

```
1 Query query = manager.createQuery("UPDATE Produto p SET p.preco = p.preco * 1.1");
2 query.executeUpdate();
```

A mesma estratégia pode ser adotada quando diversos registros devem ser removidos. Não é necessário carregar os objetos na memória para removê-los, basta realizar uma operação em lote.

```
1 Query query = manager.createQuery("DELETE Produto p WHERE p.preco < 50");
2 query.executeUpdate();
```

6.2 Exercícios

1. Crie um projeto no eclipse chamado **TopicosAvancados**. Copie a pasta **lib** do projeto **EntityManager** para o projeto **TopicosAvancados**. Depois selecione todos os jar's e os adicione no classpath.

2. Abra o **MySQL Query Browser** e apague a base de dados **K21_topicos_avancados** se existir. Depois crie a base de dados **K21_topicos_avancados**.

3. Copie a pasta **META-INF** do projeto **EntityManager** para dentro da pasta **src** do projeto **TopicosAvancados**. Altere o arquivo **persistence.xml** do projeto **TopicosAvancados**, modificando o nome da unidade de persistência e a base da dados. Veja como o código deve ficar:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence xmlns="http://java.sun.com/xml/ns/persistence"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/↩
           ns/persistence/persistence_1_0.xsd"
5      version="1.0">
6
7      <persistence-unit name="K21_topicos_avancados"
8          transaction-type="RESOURCE_LOCAL">
9          <provider>org.hibernate.ejb.HibernatePersistence</provider>
10         <properties>
11             <property name="hibernate.dialect" value="org.hibernate.dialect.↩
                   MySQL5InnoDBDialect" />
12             <property name="hibernate.hbm2ddl.auto" value="update" />
13             <property name="hibernate.show_sql" value="true" />
14             <property name="hibernate.format_sql" value="true" />
15             <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver↩
                   " />
16             <property name="javax.persistence.jdbc.user" value="root" />
17             <property name="javax.persistence.jdbc.password" value="root" />
18             <property name="javax.persistence.jdbc.url" value="jdbc:mysql://↩
                   localhost:3306/K21_topicos_avancados" />
19         </properties>
20     </persistence-unit>
21 </persistence>
```

4. Crie um pacote chamado **modelo** no projeto **TopicosAvancados** e adicione a seguinte classe:

```
@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    private Double preco;

    // GETTERS E SETTERS
}
```

5. Adicione alguns produtos no banco de dados. Crie a seguinte classe em um pacote chamado **testes** no projeto **TopicosAvancados**.

```
public class PopulaBanco {
    public static void main(String[] args) {
        EntityManagerFactory factory = Persistence
                .createEntityManagerFactory("K21_topicos_avancados");
        EntityManager manager = factory.createEntityManager();

        manager.getTransaction().begin();

        for (int i = 0; i < 100; i++) {
            Produto p = new Produto();
            p.setNome("produto " + i);
            p.setPreco(i * 10.0);
            manager.persist(p);
        }

        manager.getTransaction().commit();

        manager.close();
        factory.close();
    }
}
```

Execute e verifique a tabela produto na base de dados K21_topicos_avancados

6. Faça uma operação em lote para atualizar o preço de todos os produtos de acordo com uma taxa fixa.

```
1  public class AumentaPreco {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_topicos_avancados");
5          EntityManager manager = factory.createEntityManager();
6
7          manager.getTransaction().begin();
8
9          Query query = manager
10                 .createQuery("UPDATE Produto p SET p.preco = p.preco * 1.1");
11         query.executeUpdate();
12
13         manager.getTransaction().commit();
14
15         manager.close();
16         factory.close();
17     }
18 }
```

Execute e verifique a tabela produto na base de dados K21_topicos_avancados

Observe também o console do eclipse. Nenhum select é realizado

7. Faça uma operação em lote para remover todos os produtos com preço menor do que um valor fixo.

```
1  public class RemoveProdutos {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_topicos_avancados");
5          EntityManager manager = factory.createEntityManager();
6
7          manager.getTransaction().begin();
8
9          Query query = manager
10                 .createQuery("DELETE Produto p WHERE p.preco < 50");
11         query.executeUpdate();
12
13         manager.getTransaction().commit();
14
15         manager.close();
16         factory.close();
17     }
18 }
```

Execute e verifique a tabela produto na base de dados K21_topicos_avancados

Observe também o console do eclipse. Nenhum select é realizado

6.3 Concorrência

Quando dois Entity Managers manipulam objetos da mesma entidade e com o mesmo identificador, um resultado incorreto pode ser obtido. Por exemplo, suponha que os seguintes trechos de código sejam executados em paralelo.

```
manager1.getTransaction().begin();

Conta x = manager1.find(Conta.class, 1L);

x.setSaldo(x.getSaldo() + 500);

manager1.getTransaction().commit();
```

```
manager2.getTransaction().begin();

Conta y = manager2.find(Conta.class, 1L);

y.setSaldo(y.getSaldo() - 500);

manager2.getTransaction().commit();
```

O primeiro trecho acrescenta 500 reias no saldo da conta com identificador 1. O segundo trecho retira 500 reais da mesma conta. Dessa forma, o saldo dessa conta deve possuir o mesmo valor antes e depois desses dois trechos de código executarem. Contudo, dependendo da ordem na qual as linhas dos dois trechos são executadas o resultado pode ser outro.

Por exemplo, suponha que o valor inicial do saldo da conta com identificador 1 seja 2000 reais e as linhas dos dois trechos são executadas na seguinte ordem:

```
manager1.getTransaction().begin();
manager2.getTransaction().begin();

Conta x = manager1.find(Conta.class, 1L); // x: saldo = 2000

x.setSaldo(x.getSaldo() + 500); // x: saldo = 2500

Conta y = manager2.find(Conta.class, 1L); // y: saldo = 2000

y.setSaldo(y.getSaldo() - 500); // y: saldo = 1500

manager1.getTransaction().commit(); // Conta 1: saldo = 2500
manager2.getTransaction().commit(); // Conta 1: saldo = 1500
```

Nesse caso, o saldo final seria 1500 reais.

6.4 Exercícios

8. Acrescente no pacote **modelo** uma classe para definir contas bancárias.

```
@Entity
public class Conta {

    @Id
    @GeneratedValue
    private Long id;

    private double saldo;

    // GETTERS AND SETTERS
}
```

9. Adicione uma classe no pacote **testes** para cadastrar uma conta no banco de dados.

```
1  public class AdicionaConta {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_topicos_avancados");
5          EntityManager manager = factory.createEntityManager();
6
7          manager.getTransaction().begin();
8
9          Conta c = new Conta();
10         c.setSaldo(2000);
11         manager.persist(c);
12
13         manager.getTransaction().commit();
14
15         manager.close();
16         factory.close();
17     }
18 }
```

Execute e verifique a tabela Conta

10. Simule o problema de concorrência entre Entity Managers adicionando a seguinte classe no pacote **testes**.

```
1  public class TestaAcessoConcorrente {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_topicos_avancados");
5
6          EntityManager manager1 = factory.createEntityManager();
7          EntityManager manager2 = factory.createEntityManager();
8
9          manager1.getTransaction().begin();
10         manager2.getTransaction().begin();
11
12         Conta conta1 = manager1.find(Conta.class, 1L);
13
14         conta1.setSaldo(conta1.getSaldo() + 500);
15
16         Conta conta2 = manager2.find(Conta.class, 1L);
17
18         conta2.setSaldo(conta2.getSaldo() - 500);
19
20         manager1.getTransaction().commit();
21         manager2.getTransaction().commit();
22
23         manager1.close();
24         factory.close();
25     }
26 }
```

Execute e verifique que o saldo da conta com identificador 1 termina com 1500 sendo que o correto seria 2000

6.5 Locking Otimista

Para solucionar o problema da concorrência entre Entity Managers, podemos aplicar a idéia de **Locking Otimista**. Nessa abordagem, um atributo para determinar a versão dos registros é acrescentado na entidade. Esse atributo deve ser anotado com @VERSION.

```
1  @Entity
2  public class Conta {
3
4      @Id
5      @GeneratedValue
6      private Long id;
7
8      private double saldo;
9
10     @Version
11     private Long versao;
12
13     // GETTERS AND SETTERS
14 }
```

Toda vez que um Entity Manager modifica um registro da tabela correspondente à classe CONTA, o campo referente ao atributo anotado com @VERSION é incrementado.

Agora, antes de modificar um registro da tabela referente à classe CONTA, os Entity Managers comparam a versão do registro no banco de dados com a do objeto que eles possuem.

Se as versões forem a mesma significa que nenhum outro Entity Manager modificou o registro, então as modificações podem ser executadas sem problemas. Caso contrário, se as versões forem diferentes significa que algum outro Entity Manager modificou o registro, então as modificações são abortadas e uma exception é lançada. Em geral, as aplicações devem capturar essa exception e tentar refazer a operação.

6.6 Exercícios

11. Acrescente um atributo na classe CONTA anotado com @VERSION.

```
1  @Entity
2  public class Conta {
3
4      @Id
5      @GeneratedValue
6      private Long id;
7
8      private double saldo;
9
10     @Version
11     private Long versao;
12
13     // GETTERS AND SETTERS
14 }
```

12. Apague a tabela **Conta** através do **MySQL Query Browser**.

13. Execute a classe **AdicionaConta** e verifique a tabela **Conta** através do **MySQL Query Browser**.

14. Execute a classe **TestaAcessoConcorrente** e observe a exception gerada pelo segundo Entity Manager.

6.7 Locking Pessimista

Outra abordagem para lidar com o problema da concorrência entre Entity Managers é o **Locking Pessimista**. Nessa abordagem, um Entity Manager pode "travar"os registros fazendo com que os outros Entity Manager que desejem manipular os mesmos registros tenham que aguardar.

Há várias maneiras de utilizar o locking pessismista. Uma delas é passar mais um parâmetro quando um objeto é buscado através do método FIND().

```
1 Conta x = manager.find(Conta.class, 1L, LockModeType.PESSIMISTIC_WRITE);
```

Uma grande dificuldade em utilizar locking pessismista é que podemos gerar um **deadlock**. Suponha que dois Entity Manager busquem o mesmo objeto na mesma thread utilizando o locking pessismista como mostra o código a seguir.

```
1 Conta x = manager1.find(Conta.class, 1L, LockModeType.PESSIMISTIC_WRITE);
2 Conta y = manager2.find(Conta.class, 1L, LockModeType.PESSIMISTIC_WRITE);
3 manager1.commit();// NUNCA VAI EXECUTAR ESSA LINHA
```

Na linha 1, o primeiro Entity Manager "trava"a conta com identificador 1 e esse objeto só será liberado na linha 3. Na linha 2, o segundo Entity Manager vai esperar o primeiro liberar o objeto impedindo que a linha 3 seja executada. Dessa forma, a linha 3 nunca será executada. Depois, de um certo tempo esperando na linha 2, o segundo Entity Manager lança uma exception.

6.8 Exercícios

15. Teste o problema de deadlock quando é utilizado o locking pessimista. Adicione a seguinte classe no pacote **testes**.

```
1  public class TestaDeadLock {
2      public static void main(String[] args) {
3          EntityManagerFactory factory = Persistence
4                  .createEntityManagerFactory("K21_topicos_avancados");
5
6          EntityManager manager1 = factory.createEntityManager();
7          EntityManager manager2 = factory.createEntityManager();
8
9          manager1.getTransaction().begin();
10         manager2.getTransaction().begin();
11
12         manager1.find(Produto.class, 100L, LockModeType.PESSIMISTIC_WRITE);
13         manager2.find(Produto.class, 100L, LockModeType.PESSIMISTIC_WRITE);
14
15         manager1.getTransaction().commit();
16         manager2.getTransaction().commit();
17
18         manager1.close();
19         manager2.close();
20
21         factory.close();
22     }
23 }
```

Execute e aguarde até ocorrer uma exception

6.9 Callbacks

Podemos monitorar o ciclo de vida dos objetos das entidades da nossa aplicação. Certos eventos podem ser capturados e é possível associar um determinado método (callback) a esses eventos. Veja os eventos na listagem abaixo:

PrePersist e PostPersist: Antes e Depois de um objeto ser persistido.

PreRemove e PostRemove: Antes e Depois de um objeto ser removido.

PreUpdate and PostUpdate: Antes e Depois de um objeto ser atualizado no banco de dados.

PostLoad: Depois de um objeto ser carregado do banco de dados.

Os métodos de callback, ou seja, os métodos que serão chamados nos eventos acima listados são definidos nas classes das entidades da nossa aplicação e anotados com @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostPersist e @PostLoad.

```
@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    private Double preco;


    @PrePersist
    public void prePersist(){
        System.out.println("Persistindo....");
    }

    @PostPersist
    public void postPersist(){
        System.out.println("Já persistiu....");
    }


    // GETTERS E SETTERS
}
```

6.10 Exercícios

16. Acrescente dois métodos de callback na classe PRODUTO.

```
@Entity
public class Produto {

    @Id @GeneratedValue
    private Long id;

    private String nome;

    private Double preco;


    @PrePersist
    public void prePersist(){
        System.out.println("Persistindo....");
    }

    @PostPersist
    public void postPersist(){
        System.out.println("Já persistiu....");
    }


    // GETTERS E SETTERS
}
```

17. Execute a classe POPULABANCO novamente e observe o console.

18. (Opcional) Teste os outros callbacks.

6.11 Consultas Nativas

Os provedores JPA também devem oferecer o suporte à consultas nativas, ou seja, consultas definidas em SQL. Contudo, devemos lembrar que consultas definidas em SQL são específicas de um determinado banco de dados e eventualmente podem não funcionar em bancos de dados diferentes.

```
1 String sql = "SELECT * FROM Procuto";
2 Query nativeQuery = manager.createNativeQuery(sql, Produto.class);
3 List<Produto> produtos = nativeQuery.getResultList();
```

6.12 Exercícios

19. Testes as consultas nativas. Acrescente a seguinte classe no pacote **testes**.

```
1  public class TesteConsultaNativas {
2      public static void main(String[] args) {
3          String sql = "SELECT * FROM Produto";
4          Query nativeQuery = manager.createNativeQuery(sql, Produto.class);
5          List<Produto> produtos = nativeQuery.getResultList();
6
7          for (Produto p : produtos) {
8              System.out.println(p.getNome());
9          }
10     }
11 }
```

Capítulo 7

Arquitetura

7.1 Inicialização do JPA

O serviço de persistência definido pela especificação JPA deve ser inicializado antes de ser utilizado. O processo de inicialização pode ser realizado pela própria aplicação ou por Containers Java EE.

Quando a aplicação é standalone (Java SE), ela própria deve inicializar o serviço de persistência. Quando a aplicação não é standalone (Java EE), a inicialização pode ser feita pela própria aplicação ou pelo Container Java EE.

7.1.1 Aplicações Java SE

As aplicações Java SE (standalone) devem inicializar o serviço de persistência antes de utilizá-lo. A especificação JPA define a classe PERSISTENCE que possui um método estático de inicialização (bootstrap). Esse método deve ser chamado pelas aplicações Java para que processo de inicialização do serviço de persistência seja executado.

```
1 EntityManagerFactory factory = Persistence.createEntityManagerFactory("unidade");
```

A execução do método CREATEENTITYMANAGERFACTORY() é custosa pois ele realiza um processo complexo de inicialização. Esse método deve ser chamado apenas uma vez a cada execução da aplicação. Executá-lo duas ou mais vezes implica em desperdicio dos recursos computacionais o que prejudica o desempenho da aplicação.

Para evitar que o método CREATEENTITYMANAGERFACTORY() seja executa mais do que uma vez a cada execução da aplicação, podemos realizar a chamada desse método dentro de um bloco static. O códido de um bloco static é executado quando a classe na qual ele está definido é carregada na memória pela máquina virtual. O carregamento de uma classe é realizado no máximo uma vez a não ser que a própria aplicação explicitamente faça um novo carregamento.

```
class JPAUtil {

    // bloco static
    static {
        Persistence.createEntityManagerFactory("unidade");
    }
}
```

Além disso, para disponibilizar globalmente para a aplicação a factory criada pelo método CREATEENTITYMANAGERFACTORY(), podemos armazenar a referência dela em um atributo static e criar um método de acesso.

```
class JPAUtil {

    private static EntityManagerFactory factory;

    // bloco static
    static {
        JPAUtil.factory = Persistence.createEntityManagerFactory("unidade");
    }

    public EntityManagerFactory getFactory(){
        return JPAUtil.factory;
    }
}
```

7.1.2 Aplicações Java EE

As aplicações Java EE podem gerenciar a inicialização do serviço de persistência ou deixar essa tarefa para o Container Java EE no qual ela está implantada. Para exemplificar, suponha uma aplicação Java Web implantada em um Container Java EE.

Geralmente, quando uma aplicação Java Web gerencia a inicialização do serviço de persistência, ela utiliza um filtro para realizar esse gerenciamento.

```
public class JPAFilter implements Filter {

    private EntityManagerFactory factory;

    public void init(FilterConfig filterConfig) throws ServletException {
        this.factory = Persistence.createEntityManagerFactory("unidade");
    }

    public void destroy() {
        this.factory.close();
    }

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        // implementação
    }
}
```

Os métodos INIT() e DESTROY() são chamados apenas uma vez a cada execução da aplicação. O INIT() e chamado no começo da execução e o DESTROY() no final. O filtro acima inicializa o serviço de persistência no método INIT() e o encerra no método DESTROY().

Além disso, a aplicação deve gerenciar diretamente as transações e portanto configurar o arquivo **persistence.xml** aplicando o valor **RESOURCE_LOCAL** para a propriedade **transaction-type** da unidade de persitência.

```
1 <persistence-unit name="copadomundo" transaction-type="RESOURCE_LOCAL">
```

Por outro lado, quando uma aplicação Java Web deixa o Container Java EE no qual ela está implantada gerenciar a inicialização do serviço de persistência, ela deve configurar o arquivo PERSISTENCE.XML de maneira diferente.

A unidade de persistência deve utilizar transações do tipo **JTA**. Esse tipo de transação é o padrão para aplicações Java EE. Dessa forma, podemos definir explicitamente esse tipo de transação ou simplesmente não definir nenhum tipo.

```
1 <persistence-unit name="copadomundo" transaction-type="JTA">
```

```
1 <persistence-unit name="copadomundo">
```

Além disso, um **data source** previamente configurado no Container Java EE deve ser configurado na unidade de persistência.

```
1 <jta-data-source>jdbc/MySQL</jta-data-source>
```

7.2 Repositórios

Todo o controle do ciclo de vida das instâncias das entidades e todas as consultas podem ser realizadas através dos métodos da interface dos Entity Managers. Dessa forma, é possível desenvolver uma aplicação utilizando a seguinte abordagem: Quando alguma classe necessita acessar ou modificar os dados das instâncias das entidades ela deve obter um Entity Manager e interagir diretamente com ele. Essa abordagem é válida e para muitos casos não prejudicará o desenvolvimento e manutenção da aplicação, ou seja, não há problemas em adotá-la. Contudo, algumas condições podem tornar essa abordagem problemática.

Por exemplo, determinada consulta específica da nossa aplicação pode ser complexa demais ou até mesmo impossível de ser implementada apenas com os mecanismos de pesquisa do JPA (JPQL e Criteria). Nesses casos, a alternativa imediata é fazer uma ou mais consultas auxiliares através de um Entity Manager e depois realizar um pós processamento nos dados obtidos para gerar o resultado da consulta principal.

Supondo que número de consultas que exigam um pós processamento é considerável, faz sentido pensar em centralizá-las para evitar repetição de código, facilitar a manutenção e viabilizar os testes. Daí surge um padrão de projeto chamado **Repository**. Os repositórios são objetos que oferecem acesso às instâncias das entidades para toda a aplicação e portanto encapsulam a lógica do controle do ciclo de vida das instâncias das entidades e das consultas. De fato, o padrão Repository é mais vantagoso quando essa lógica é mais complexa.

Contudo, podemos padronizar as nossas aplicações e sempre utilizar o padrão Repository para controlar o ciclo de vida das instâncias da entidades e realizar as consultas mesmo quando a lógica para essas tarefas é simples. Dessa forma, os repositórios sempre funcionariam como uma "casca" envolvendo os Entity Managers do JPA.

Resumidamente, temos três abordagens para manipular as instâncias das entidades e realizar as consultas:

1. Utilizar diretamente os Entity Managers.

2. Utilizar os Entity Managers para os casos mais "simples" e os Repositórios para os mais "complexos".

3. Utilizar sempre os Repositórios como "casca" dos Entity Managers.

7.3 Controle de Transações

Há diversas abordagens para controlar transações. A primeira abordagem seria criar uma transação por operação.

```
1 manager.getTransaction().begin();
2 manager.persist(livro);
3 manager.getTransaction().commit();
```

Adicionando o código necessário para **rollback**.

```
1  EntityTransaction transaction = null;
2  try {
3      transaction = manager.getTransaction();
4      transaction.begin();
5      manager.persist(livro);
6      transaction.commit();
7  } catch(Exception ex) {
8      if(transaction != null && transaction.isActive()) {
9          transaction.rollback();
10     }
11 } finally {
12     manager.close();
13 }
```

O custo para abrir, manter, confirmar e desfazer uma transação é relativamente alto. Portanto, a abordagem de criar uma transação por operação pode prejudicar a performance da aplicação.

Além disso, o código para um controle de transações adequado é complexo e não é prático nem seguro replicá-lo por toda a aplicação.

Podemos melhorar o controle de transações centralizando o processo em uma classe capaz de executar qualquer trecho de código dentro de uma transação.

```
1  public abstract class TarefaTransacional {
2
3      private EntityManager manager;
4
5      public TarefaTransacional(EntityManager manager) {
6          this.manager = manager;
7      }
8
9      public void executa() {
10         EntityTransaction transaction = null;
11         try {
12             transaction = this.manager.getTransaction();
13             transaction.begin();
14
15             this.blocoTransacional();
16
17             transaction.commit();
18         } catch(Exception ex) {
19             if(transaction != null && transaction.isActive()) {
20                 transaction.rollback();
21             }
22         } finally {
23             this.manager.close();
24         }
25     }
26
27     public abstract void blocoTransacional();
28 }
```

Podemos utilizar a classe TAREFATRANSACIONAL toda vez que é necessário executar um bloco de código dentro de uma transação.

```
1  TarefaTransacional tarefa = new TarefaTransacional(manager) {
2      public void blocoTransacional(){
3          this.manager.persist(livro);
4          this.manager.persist(editora);
5      }
6  }
7  tarefa.executa();
```

7.3.1 Open Session in View

Um padrão muitas vezes adotado para realizar o controle de transações é o **Open Session in View**. Aplicando esse padrão, obtemos um controle centralizado e um único ponto da aplicação e transparente para todo o resto do código.

Em aplicações Java Web esse padrão pode ser aplicado através de um filtro.

```
1  public class JPAFilter implements Filter {
2
3      private EntityManagerFactory factory;
4
5      public void init(FilterConfig filterConfig) throws ServletException {
6          this.factory = Persistence.createEntityManagerFactory("unidade");
7      }
8
9      public void destroy() {
10         this.factory.close();
11     }
12
13     public void doFilter(ServletRequest request, ServletResponse response,
14             FilterChain chain) throws IOException, ServletException {
15         EntityTransaction transaction = null;
16         try {
17             EntityManager manager = this.factory.createEntityManager();
18             request.setAttribute("em", entityManager);
19             transaction = manager.getTransaction();
20             transaction.begin();
21
22             chain.doFilter(request, response);
23
24             transaction.commit();
25         } catch(Exception ex) {
26             if(transaction != null && transaction.isActive()) {
27                 transaction.rollback();
28             }
29         } finally {
30             manager.close();
31         }
32     }
33 }
```

Nessa abordagem, todas as requisições são processadas dentro de uma única transação.