

UMA INTRODUÇÃO PRÁTICA AO JPA COM HIBERNATE

"É uma experiência eterna de que todos os homens com poder são tentados a abusar." -- Baron de Montesquieu

Neste capítulo, você aprenderá a:

- Entender a diferença entre a JPA e o Hibernate;
- Usar a ferramenta de ORM JPA com Hibernate;
- Gerar as tabelas em um banco de dados qualquer a partir de suas classes de modelo;
- Inserir e carregar objetos pelo JPA no banco;
- Buscar vários objetos pelo JPA;

Mapeamento Objeto Relacional

Com a popularização do Java em ambientes corporativos, logo se percebeu que grande parte do tempo do desenvolvedor era gasto na codificação de queries SQL e no respectivo código JDBC responsável por trabalhar com elas.

Além de um problema de produtividade, algumas outras preocupações aparecem: SQL que, apesar de ter um padrão ANSI, apresenta diferenças significativas dependendo do fabricante. Não é simples trocar um banco de dados pelo outro.

Há ainda a mudança do paradigma. A programação orientada a objetos difere muito do esquema entidade relacional e precisamos pensar das duas maneiras para fazer um único sistema. Para representarmos as informações no banco, utilizamos tabelas e colunas. As tabelas

geralmente possuem chave primária (PK) e podem ser relacionadas por meio da criação de chaves estrangeiras (FK) em outras tabelas.

Quando trabalhamos com uma aplicação Java, seguimos o paradigma orientado a objetos, onde representamos nossas informações por meio de classes e atributos. Além disso, podemos utilizar também herança, composição para relacionar atributos, polimorfismo, enumerações, entre outros. Esse buraco entre esses dois paradigmas gera bastante trabalho: a todo momento devemos "transformar" objetos em registros e registros em objetos.

Java Persistence API e Frameworks ORM

Ferramentas para auxiliar nesta tarefa tornaram-se populares entre os desenvolvedores Java e são conhecidas como ferramentas de **mapeamento objeto-relacional** (ORM). O Hibernate é uma ferramenta ORM open source e é a líder de mercado, sendo a inspiração para a especificação **Java Persistence API** (JPA). O Hibernate nasceu sem JPA mas hoje em dia é comum acessar o Hibernate pela especificação JPA. Como toda especificação, ela deve possuir implementações. Entre as implementações mais comuns, podemos citar: Hibernate da Red Hat, EclipseLink da Eclipse Foundation e o OpenJPA da Apache. Apesar do Hibernate ter originado a JPA, o EclipseLink é a implementação referencial.

O Hibernate abstrai o seu código SQL, toda a camada JDBC e o SQL será gerado em tempo de execução. Mais que isso, ele vai gerar o SQL que serve para um determinado banco de dados, já que cada banco fala um "dialetto" diferente dessa linguagem. Assim há também a possibilidade de trocar de banco de dados sem ter de alterar código Java, já que isso fica como responsabilidade da ferramenta.

Como usaremos JPA abstraímos mais ainda, podemos desenvolver sem conhecer detalhes sobre o Hibernate e até trocar o Hibernate com uma outra implementação como OpenJPA:



SEUS LIVROS DE TECNOLOGIA PARECEM DO SÉCULO PASSADO?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil. Conheça os títulos e a nova proposta, você vai gostar.

Casa do Código, livros para o programador

Bibliotecas do Hibernate e JPA

Vamos usar o JPA com Hibernate, ou seja, precisamos baixar os JARs no site do Hibernate. O site oficial do Hibernate é o **www.hibernate.org**, onde você baixa a última versão na seção ORM e Download.

Com o ZIP baixado em mãos, vamos descompactar o arquivo. Dessa pasta vamos usar todos os JARs obrigatórios (`required`). Não podemos esquecer o JAR da especificação JPA que se encontra na pasta `jpa`.

Para usar o Hibernate e JPA no seu projeto é necessário colocar todos esses JARs no *classpath*.

O Hibernate vai gerar o código SQL para qualquer banco de dados. Continuaremos utilizando o banco MySQL, portanto também precisamos o arquivo `.jar` correspondente ao driver JDBC.

Mapeando uma classe Tarefa para nosso Banco de Dados

Para este capítulo, continuaremos utilizando a classe que representa uma tarefa:

```
package br.com.caelum.tarefas.modelo;

public class Tarefa {
    private Long id;
    private String descricao;
    private boolean finalizado;
    private Calendar dataFinalizacao;
}
```

Criamos os getters e setters para manipular o objeto, mas fique atento que só devemos usar esses métodos se realmente houver necessidade.

Essa é uma classe como qualquer outra que aprendemos a escrever em Java. Precisamos configurar o Hibernate para que ele saiba da existência dessa classe e, desta forma, saiba que deve inserir uma linha na tabela `Tarefa` toda vez que for requisitado que um objeto desse tipo seja salvo. Em vez de usarmos o termo "configurar", falamos em **mapear** uma classe a tabela.

Para mapear a classe `Tarefa`, basta adicionar algumas poucas **anotações** em nosso código. Anotação é um recurso do Java que permite inserir **metadados** em relação a nossa classe, atributos e métodos. Essas anotações depois poderão ser lidas por frameworks e bibliotecas, para que eles tomem decisões baseadas nessas pequenas configurações.

Para essa nossa classe em particular, precisamos de apenas quatro anotações:

```
@Entity
public class Tarefa {

    @Id
    @GeneratedValue
    private Long id;

    private String descricao;
    private boolean finalizado;

    @Temporal(TemporalType.DATE)
```

```
private Calendar dataFinalizacao;  
  
// métodos...  
}
```

`@Entity` indica que objetos dessa classe se tornem "persistível" no banco de dados. `@Id` indica que o atributo `id` é nossa chave primária (você precisa ter uma chave primária em toda entidade) e `@GeneratedValue` diz que queremos que esta chave seja populada pelo banco (isto é, que seja usado um `auto increment` ou `sequence`, dependendo do banco de dados). Com `@Temporal` configuramos como mapear um `Calendar` para o banco, aqui usamos apenas a data (sem hora), mas poderíamos ter usado apenas a hora (`TemporalType.TIME`) ou `timestamp` (`TemporalType.TIMESTAMP`). Essas anotações precisam dos devidos `imports`, e pertencem ao pacote `javax.persistence`.

Mas em que tabela essa classe será gravada? Em quais colunas? Que tipo de coluna? Na ausência de configurações mais específicas, o Hibernate vai usar convenções: a classe `Tarefa` será gravada na tabela de nome também `Tarefa`, e o atributo `descricao` em uma coluna de nome `descricao` também!

Se quisermos configurações diferentes das convenções, basta usarmos outras anotações, que são completamente opcionais. Por exemplo, para mapear o atributo `dataFinalizacao` numa coluna chamada `data_finalizado` faríamos:

```
@Column(name = "data_finalizado", nullable = true)  
private Calendar dataFinalizacao;
```

Para usar uma tabela com o nome `tarefas`:

```
@Entity  
@Table(name="tarefas")  
public class Tarefa {
```

Repare que nas entidades há todas as informações sobre as tabelas. Baseado nelas podemos até pedir gerar as tabelas no banco, mas para isso é preciso configurar o JPA.

Configurando o JPA com as propriedades do banco

Em qual banco de dados vamos gravar nossas `Tarefas`? Qual é o login? Qual é a senha? O JPA necessita dessas configurações, e para isso criaremos o arquivo `persistence.xml`.

Alguns dados que vão nesse arquivo são específicos do Hibernate e podem ser bem avançados, sobre controle de cache, transações, connection pool etc, tópicos que são abordados no curso FJ-25.

Para nosso sistema, precisamos de quatro linhas com configurações que já conhecemos do JDBC: string de conexão com o banco, o driver, o usuário e senha. Além dessas quatro configurações, precisamos dizer qual dialeto de SQL deverá ser usado no momento que as queries são geradas; no nosso caso, MySQL. Vamos também mostrar o SQL no console para acompanhar o trabalho do Hibernate.

Vamos também configurar a criação das tabelas baseado nas entidades.

Segue uma configuração completa que define uma unidade de persistência (*persistence-unit*) com o nome *tarefas*, seguidos pela definição do provedor, entidades e properties:

```
<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_2_0.xsd"
  version="2.0">

  <persistence-unit name="tarefas">

    <!-- provedor/implementacao do JPA -->
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <!-- entidade mapeada -->
    <class>br.com.caelum.tarefas.modelo.Tarefa</class>

    <properties>
      <!-- dados da conexao -->
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.jdbc.Driver" />
```

```
<property name="javax.persistence.jdbc.url"
  value="jdbc:mysql://localhost/fj21" />
<property name="javax.persistence.jdbc.user"
  value="root" />
<property name="javax.persistence.jdbc.password"
  value="<SENHA DO BANCO AQUI>" />

<!-- propriedades do hibernate -->
<property name="hibernate.dialect"
  value="org.hibernate.dialect.MySQL8Dialect" />
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="true" />

<!-- atualiza o banco, gera as tabelas se for preciso -->
<property name="hibernate.hbm2ddl.auto" value="update" />
</properties>
</persistence-unit>
</persistence>
```

É importante saber que o arquivo `persistence.xml` deve ficar na pasta **META-INF** do seu *classpath*.

alura

AGORA É A MELHOR HORA DE APRENDER ALGO NOVO

Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

Conheça a Alura

Usando o JPA

Para usar o JPA no nosso código Java, precisamos utilizar sua API. Ela é bastante simples e direta e rapidamente você estará habituado com suas principais classes.

Nosso primeiro passo é fazer com que o JPA leia a nossa configuração: tanto o nosso arquivo `persistence.xml` quanto as anotações que colocamos na nossa entidade `Tarefa`. Para tal, usaremos a classe com o mesmo nome do arquivo XML: `Persistence`. Ela é responsável de carregar o XML e inicializar as configurações. Resultado dessa configuração é uma `EntityManagerFactory`:

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("tarefas");
```

Estamos prontos para usar o JPA. Antes de gravar uma `Tarefa`, precisamos que exista a tabela correspondente no nosso banco de dados. Em vez de criarmos o script que define o *schema* (ou DDL de um banco, *data definition language*) do nosso banco (os famosos `CREATE TABLE . . .`) podemos deixar isto a cargo do próprio Hibernate. Ao inicializar a `EntityManagerFactory` também já será gerada uma tabela `Tarefas` pois configuramos que o banco deve ser atualizada pela propriedade do Hibernate: `hbm2ddl.auto`.

Para saber mais: Configurando o JPA com Hibernate em casa

Caso você esteja fazendo esse passo de casa. É preciso baixar o ZIP do Hibernate ORM 5.x (<http://hibernate.org>), extraí-lo, e copiar todos os JARs das pastas `lib/required` e `lib/jpa`.

Para essa aplicação usaremos as seguintes versões:

- `antlr-2.7.x.jar`
- `byte-buddy-1.9.1x.jar`
- `dom4j-2.1.x.jar`
- `hibernate-commons-annotations-5.1.x.Final.jar`
- `hibernate-core-5.4.x.Final.jar`
- `javassist-3.24.x-GA.jar`
- `javax.persistence-api-2.x.jar`
- `jaxb-runtime-2.3.x.jar`
- `jboss-transaction-api_1.2_spec-1.0.x.Final.jar`

Exercícios: Configurando o JPA e gerando o schema do banco

1. Vamos preparar nosso projeto `fj21-tarefas` com as dependências do Hibernate.

Copie os JARs do Hibernate para a pasta **WebContent/WEB-INF/lib** do seu projeto dessa forma:

- Vá até o diretório **21/projeto-tarefas/jpa hibernate**;
- Selecione todos os JARs, clique com o botão direito e escolha *Copy* (ou `CTRL+C`);
- Cole todos os JARs na pasta **WebContent/WEB-INF/lib** do projeto `fj21-tarefas` (`CTRL+V`)

2. Anote a classe `Tarefa` como uma entidade de banco de dados. Lembre-se que essa é uma anotação do pacote `javax.persistence`.

Obs: Não apague nenhuma anotação, apenas adicione as anotações do JPA.

```
@Entity
public class Tarefa {

}
```

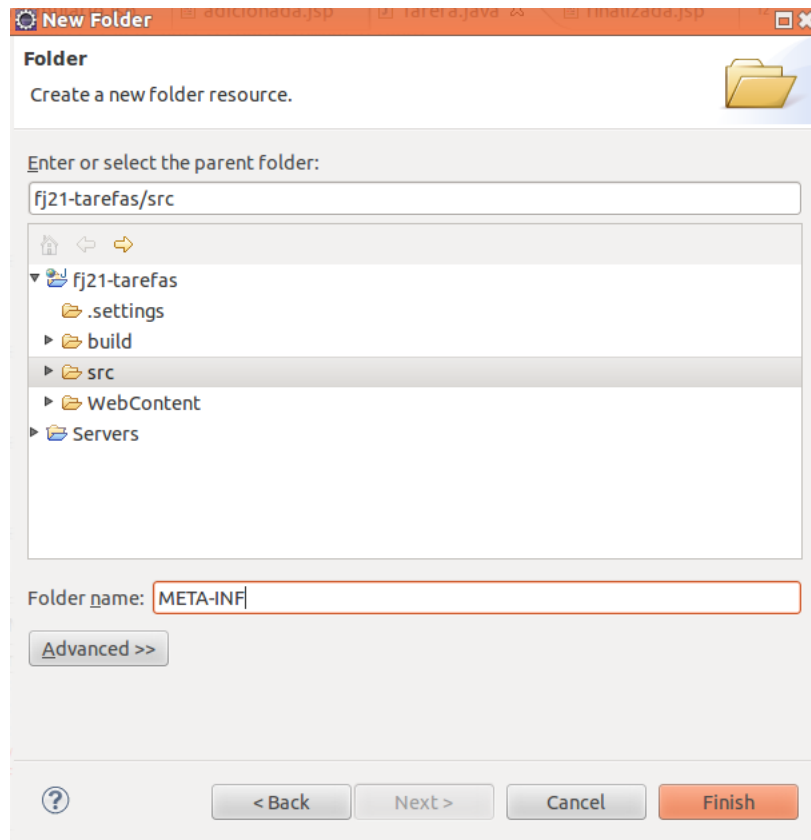
Na mesma classe anote seu atributo `id` como chave primária e como campo de geração automática:

```
@Id
@GeneratedValue
private Long id;
```

Agora é preciso mapear o atributo `dataFinalizacao` para gravar apenas a data (sem hora) no banco:

```
@Temporal(TemporalType.DATE)
private Calendar dataFinalizacao;
```

1. Clique com o botão direito na pasta `src` do projeto do `fj21-tarefas` e escolha *New -> Folder*. Escolha **META-INF** como nome dessa pasta.



- Vá ao diretório **21/projeto-tarefas/jpa hibernate** e copie o arquivo `persistence.xml` para a pasta `META-INF`.
- O arquivo é apenas um esboço, falta configurar a unidade de persistência com o provedor, entidades e propriedades. Adicione dentro do elemento `persistence`:

```
<persistence-unit name="tarefas">
```

```
<provider>org.hibernate.ejb.HibernatePersistence</provider>
```

```
<class>br.com.caelum.tarefas.modelo.Tarefa</class>
```

```
<properties>
```

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

```
<property name="javax.persistence.jdbc.url"  
  value="jdbc:mysql://localhost/fj21" />
```

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

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

```
<property name="hibernate.dialect"  
  value="org.hibernate.dialect.MySQL8Dialect" />
```

```
<property name="hibernate.show_sql" value="true" />
<property name="hibernate.format_sql" value="true" />
<property name="hibernate.hbm2ddl.auto" value="update" />
</properties>
</persistence-unit>
```

As duas propriedades `show_sql` e `format_sql` fazem com que todo SQL gerado pelo Hibernate apareça no console.

2. Crie a classe `GeraTabelas` no pacote `br.com.caelum.tarefas.jpa`.

```
package br.com.caelum.tarefas.jpa;

// imports omitidos

public class GeraTabelas {

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

        factory.close();
    }
}
```

3. Crie sua tabela executando a classe `GeraTabelas`. Para isso, clique da direita no código e vá em *Run As -> Java Application*.

Não é preciso rodar o Tomcat para esse exercício.

4. Agora, abra uma nova aba no **Terminal** e digite `mysql -u root` se não houver senha do banco, ou `mysql -u root -p`, se houver senha, informando a senha correta em seguida. Após isto, digite `use fj21`; e em seguida, `show tables`; Se seu banco de dados já existia, e não foi preciso criá-lo no passo anterior, você vai notar a presença de uma tabela chamada `tarefas`. Não é esta a tabela que queremos. Procuramos pela tabela **Tarefa**, com **T**, em maiúsculo (igual ao nome da classe `Tarefa`).

```
mysql> select * from Tarefa;
+-----+-----+-----+-----+
| id | dataFinalizacao | descricao | finalizado |
+-----+-----+-----+-----+
| 1 | 2013-02-25 | Estudar JPA e Hibernate | 1 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

5. (opcional) Caso algum erro tenha ocorrido, é possível que o Hibernate tenha logado uma mensagem, mas você não a viu dado que o Log4J não está configurado. Mesmo que tudo tenha ocorrido de maneira correta, é muito importante ter o Log4J configurado.

Para isso, adicione no arquivo `log4j.properties` dentro da pasta `src` para que todo o log do nível `info` ou acima seja enviado para o console appender do `System.out` (default do console):

```
log4j.logger.org.hibernate=info
```



Casa do
Código

NOVA EDITORA CASA DO CÓDIGO COM LIVROS DE UMA FORMA DIFERENTE

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook

Trabalhando com os objetos: o EntityManager

Para se comunicar com o JPA, precisamos de uma instância de um objeto do tipo `EntityManager`. Adquirimos uma `EntityManager` através da fábrica já conhecida: `EntityManagerFactory`.

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("tarefas");

EntityManager manager = factory.createEntityManager();

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

Persistindo novos objetos

Através de um objeto do tipo `EntityManager`, é possível gravar novos objetos no banco. Para isto, basta utilizar o método `persist` dentro de uma transação:

```
Tarefa tarefa = new Tarefa();
tarefa.setDescricao("Estudar JPA");
tarefa.setFinalizado(true);
tarefa.setDataFinalizacao(Calendar.getInstance());

EntityManagerFactory factory = Persistence.createEntityManagerFactory("tarefas");
EntityManager manager = factory.createEntityManager();

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

System.out.println("ID da tarefa: " + tarefa.getId());

manager.close();
```

Cuidados ao usar o JPA

Ao usar o JPA profissionalmente, fique atento com alguns cuidados que são simples, mas não dada a devida atenção podem gerar gargalos de performance. Abrir `EntityManagers` indiscriminadamente é um desses problemas. Esse post da Caelum indica outros:

<http://bit.ly/bRc5g>

Esses detalhes do dia a dia assim como JPA com Hibernate avançado são vistos no curso FJ-25 da Caelum, **de Hibernate e JPA avançados**.

Carregar um objeto

Para buscar um objeto dada sua chave primária, no caso o seu `id`, utilizamos o método `find`, conforme o exemplo a seguir:

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("tarefas");
EntityManager manager = factory.createEntityManager();

Tarefa encontrada = manager.find(Tarefa.class, 1L);

System.out.println(encontrada.getDescricao());
```

Exercícios: Gravando e Carregando objetos

1. Crie uma classe chamada `AdicionaTarefa` no pacote `br.com.caelum.tarefas.jpa`, ela vai criar um objeto e adicioná-lo ao banco:

```
package br.com.caelum.tarefas.jpa;

// imports omitidos

public class AdicionaTarefa {

    public static void main(String[] args) {

        Tarefa tarefa = new Tarefa();
        tarefa.setDescricao("Estudar JPA e Hibernate");
        tarefa.setFinalizado(true);
        tarefa.setDataFinalizacao(Calendar.getInstance());

        EntityManagerFactory factory = Persistence.
            createEntityManagerFactory("tarefas");
        EntityManager manager = factory.createEntityManager();

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

        System.out.println("ID da tarefa: " + tarefa.getId());

        manager.close();
    }
}
```

2. Rode a classe `AdicionaTarefa` e adicione algumas tarefas no banco. Saída possível:



3. Verifique os registros no banco, se conecte com o cliente do mysql:

```
mysql -u root
use fj21;
select * from Tarefa;
```

Se houver senha no banco, troque o primeiro comando por `mysql -u root -p`, usando a senha correta pro banco. Rode novamente a classe `AdicionaTarefa` e verifique o banco.

1. Vamos carregar tarefas pela chave primária.

Crie uma classe chamada `CarregaTarefa` no pacote `br.com.caelum.tarefas.jpa`:

```
package br.com.caelum.tarefas.jpa;

// imports omitidos

public class CarregaTarefa {

    public static void main(String[] args) {

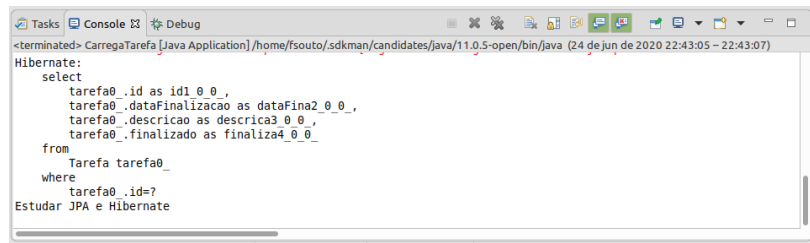
        EntityManagerFactory factory = Persistence.
            createEntityManagerFactory("tarefas");
        EntityManager manager = factory.createEntityManager();

        Tarefa encontrada = manager.find(Tarefa.class, 1L);
        System.out.println(encontrada.getDescricao());

        manager.close();
    }
}
```

2. Rode a classe `CarregaTarefa` e verifique a saída.

Caso recebeu uma exception, verifique o id da tarefa. Ele deve existir no banco.



Removendo e atualizando objeto

Remover e atualizar os objetos com JPA também é muito simples.

O `EntityManager` possui métodos para cada operação. Para remover é preciso carregar a entidade antes e depois usar o método `remove`:

```
EntityManager manager = //abrir um EntityManager
Tarefa encontrada = manager.find(Tarefa.class, 1L);

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

Para atualizar um entidade que já possui um id existe o método `merge`, por exemplo:

```
Tarefa tarefa = new Tarefa();
tarefa.setId(2); //esse id já existe no banco
tarefa.setDescricao("Estudar JPA e Hibernate");
tarefa.setFinalizado(false);
tarefa.setDataFinalizacao(null);

EntityManager manager = //abrir um EntityManager

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

alura

JÁ CONHECE OS CURSOS ONLINE ALURA?

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um

curso nas áreas de Java, Front-end, Ruby, Web, Mobile, .NET, PHP e outros, com um plano que dá acesso a todos os cursos.

Conheça os cursos online Alura

Buscando com uma cláusula where

O JPA possui uma linguagem própria de queries para facilitar a busca de objetos chamada de **JPQL**. O código a seguir mostra uma pesquisa que retorna todas as tarefas não finalizadas:

```
EntityManager manager = //abrir um EntityManager
List<Tarefa> lista = manager
    .createQuery("select t from Tarefa as t where t.finalizado = false")
    .getResultList();

for (Tarefa tarefa : lista) {
    System.out.println(tarefa.getDescricao());
}
```

Também podemos passar um parâmetro para a pesquisa. Para isso, é preciso trabalhar com um objeto que representa a pesquisa (`javax.persistence.Query`):

```
EntityManager manager = //abrir um EntityManager

Query query = manager
    .createQuery("select t from Tarefa as t "+
        "where t.finalizado = :paramFinalizado");
query.setParameter("paramFinalizado", false);

List<Tarefa> lista = query.getResultList();
```

Este é apenas o começo. O JPA é muito poderoso e, no curso FJ-25, veremos como fazer queries complexas, com joins, agrupamentos, projeções, de maneira muito mais simples do que se tivéssemos de escrever as queries. Além disso, o JPA com Hibernate é muito customizável: podemos configurá-lo para que gere as queries de acordo com dicas nossas, dessa forma otimizando casos particulares em que as queries que ele gera por padrão não são desejáveis.

Uma confusão que pode ser feita a primeira vista é pensar que o JPA com Hibernate é lento, pois, ele precisa gerar as nossas queries, ler objetos e suas anotações e assim por diante. Na verdade, o Hibernate faz uma série de otimizações internamente que fazem com que o impacto dessas tarefas seja próximo a nada. Portanto, o Hibernate é, sim, performático, e hoje em dia pode ser utilizado em qualquer projeto que se trabalha com banco de dados.

Exercícios: Buscando com JPQL

1. Crie uma classe chamada `BuscaTarefas` no pacote `br.com.caelum.tarefas.jpa`:

Cuidado, a classe `Query` vem do pacote `javax.persistence`:

```
package br.com.caelum.tarefas.jpa;

import javax.persistence.Query;
// outros imports omitidos

public class BuscaTarefas {

    public static void main(String[] args) {

        EntityManagerFactory factory = Persistence.
            createEntityManagerFactory("tarefas");
        EntityManager manager = factory.createEntityManager();

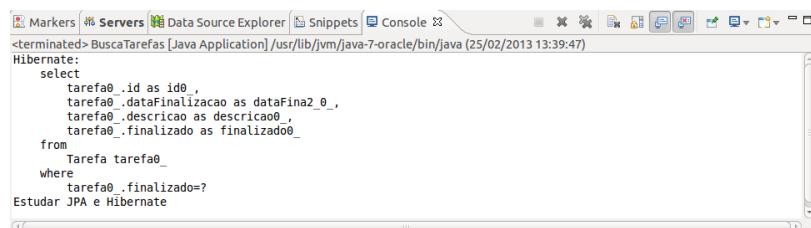
        //cuidado, use o import javax.persistence.Query
        Query query = manager
            .createQuery("select t from Tarefa as t "+
                "where t.finalizado = :paramFinalizado");
        query.setParameter("paramFinalizado", true);

        List<Tarefa> lista = query.getResultList();

        for (Tarefa t : lista) {
            System.out.println(t.getDescricao());
        }

        manager.close();
    }
}
```

2. Rode a classe `BuscaTarefas` e verifique a saída.



The screenshot shows an IDE window with a console tab. The console displays a message indicating a terminated Java application, followed by a Hibernate SQL query. The query is a SELECT statement that joins the 'tarefa' table with itself. It selects columns from both instances of the table, including 'id', 'dataFinalizacao', 'descricao', and 'finalizado'. The query is filtered by a condition on the 'finalizado' column of the second instance of the table.

```
<terminated> BuscaTarefas [Java Application] /usr/lib/jvm/java-7-oracle/bin/java (25/02/2013 13:39:47)
Hibernate:
select
  tarefa_.id as id0_,
  tarefa_.dataFinalizacao as dataFina2_0_,
  tarefa_.descricao as descricao0_,
  tarefa_.finalizado as finalizado0_
from
  Tarefa tarefa_
where
  tarefa_.finalizado=?
Estudar JPA e Hibernate
```