INTRODUÇÃO A PERSISTÊNCIA DE DADOS COM HIBERNATE E ANNOTATION

Marcus Vinícius de Sousa Lemos
http://www.marvinlemos.net

Teresina/PI

INTRODUÇÃO

Grande parte das aplicações desenvolvidas mantém suas informações gravadas em um banco de dados relacional. O grande problema, é que, atualmente, as melhores linguagens de programação são orientadas a objeto, o que torna complicado a integração entre esses bancos e essas linguagens. Além disso, mesmo em linguagens estruturadas como Pascal e C, trabalhar com banco de dados tornava-se uma tarefa árdua a medida que a aplicação ia crescendo. Um modelo de programação muito usado, mesmo em linguagens tipicamente orientadas a objeto como PHP e Java, é misturar lógica de negócio com código SQL. Se o banco de dados da aplicação mudasse, seria necessário reescrever praticamente toda a aplicação, para dar suporte ao novo banco.

Uma técnica bastante conhecida da orientação a objetos é o encapsulamento, onde podemos esconder nossas regras dentro de objetos e definir alguns métodos nesses objetos que o mundo externo poderá usar para ter acesso ao resultado de nossos códigos. Essa idéia foi adaptada à programação com banco de dados. Os métodos necessários ao acesso e manipulação do banco ficam escondidos dentro de classes básicas. As outras partes da aplicação usam essas classes e seus objetos, ou seja, a aplicação nunca terá acesso diretamente ao nosso banco.

MAPEAMENTO OBJETO/RELACIONAL

As bibliotecas de Mapeamento Objeto/Relacional fazem o mapeamento de tabelas para classes. Se o banco de dados possui uma tabela chamada Pedido, a aplicação possuirá, provavelmente, uma classe denominada Pedido. Essa classe definirá atributos, que serão usados para receber e alterar os dados dos campos das tabelas, além de métodos para calcular os juros, valor total do pedido e etc.

Além disso, as classes que fazem essa interface com as tabelas do banco de dados, provêem um conjunto de métodos de alto-nível que servem para realizar operações básicas nas tabelas, como recuperar um registro através de um id, dentre outros.

O Hibernate é uma framework ORM (*Object-Relational Mapping*) ou Mapeamento Objeto-Relacional para ambientes Java. Segundo a documentação oficial: "o Hibernate pretende retirar do desenvolvedor cerca de 95% das tarefas mais comuns de persistência de dados".

Talvez o Hibernate não seja a solução mais adequada para aplicações que usam stored-procedures para implementar suas lógicas no banco de dados. É mais apropriada para modelos orientado a objetos e lógica de negócios implementados em uma camada de uma aplicação baseado em Java.

O hibernate é a framework de persistência Java mais utilizada e documentada. É mantido pela Jboss sob a licença LGPL. Dentre algumas de suas características, podemos citar:

- Suporta classes desenvolvidas com agregações, herança, polimorfismo, composições e coleções
- Permite a escrita de consultas tanto através de uma linguagem própria (HQL) como também através de SQL
- É um framework não intrusivo, ou seja, não restringe a arquitetura da aplicação
- Implementa a especificação Java Persistente API (JPA)
- Grande a ativa comunidade

HIBERNATE COM ANNOTATION

Como a maioria das ferramentas ORM, o Hibernate precisa de metadados para determinar como deverá ser feita a transformação dos dados entre classes e tabelas. Como opção, podemos utilizar um recurso, que veio com o JDK 5.0, chamado de Annotation ao invés de arquivos XML como eram feitos anteriormente.

É importante que você entenda o que são annotations antes de ler o artigo, uma vez que não pretendo entrar em detalhes sobre eles.

SOBRE O ARTIGO

A proposta desse artigo é de servir como um ponto de partida para quem deseja aprender Hibernate. Estou partindo da premissa que o leitor já conhece bem as principais características da linguagem Java. Utilizaremos a IDE Eclipse, mas nada impede que você use outras IDE's, tais como NetBeans, IntelliJ, dentre outras.

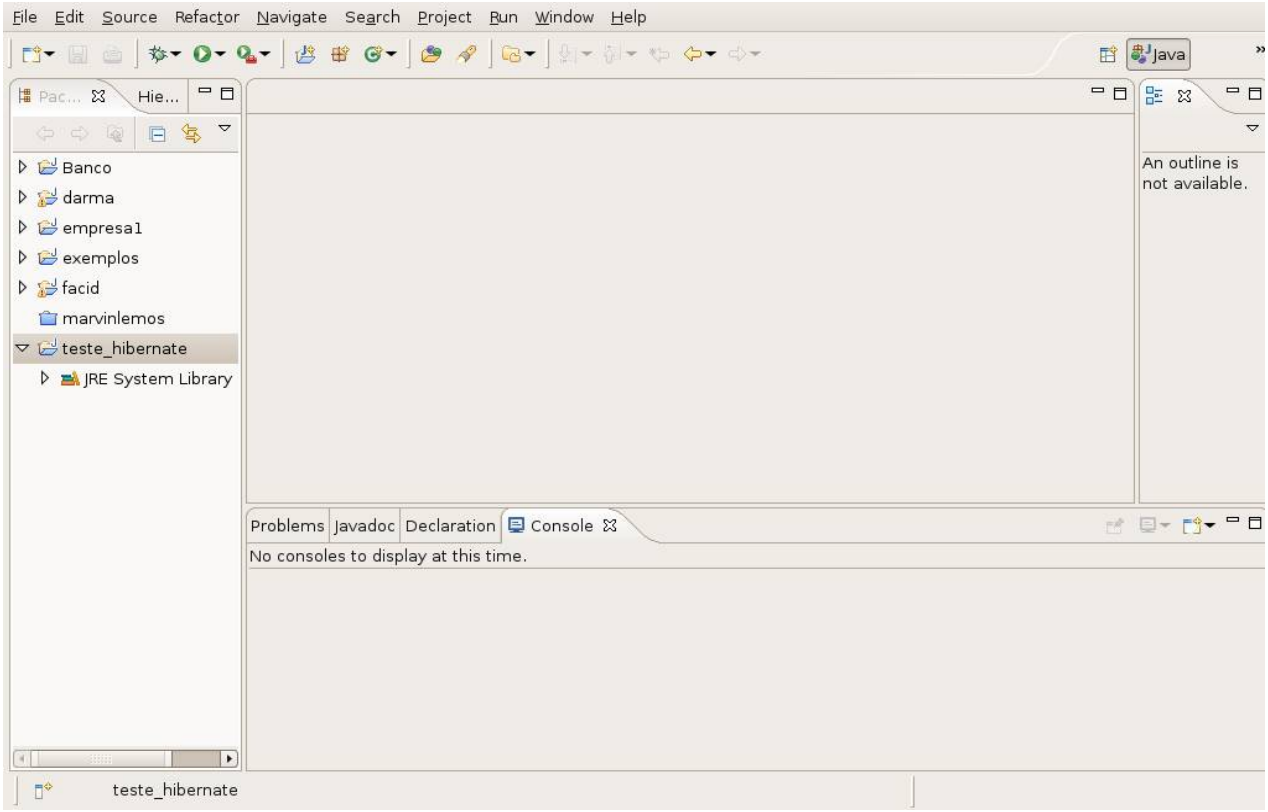
AGRADECIMENTOS

Gostaria de agradecer ao meu amigo Maurílio Lacerda pela revisão final do texto.

PARTE I

1.1 CRIANDO O PROJETO

Para os nossos exemplos, crie um novo projeto Java chamado **teste_hibernate**:



1.2 ADICIONANDO AS BIBLIOTECAS AO PROJETO

Agora vem a parte mais importante, adicionar as bibliotecas do Hibernate ao nosso projeto. O Hibernate pode ser baixado no seguinte endereço **http://www.hibernate.org/6.html**
Recomendo o download de pelo menos três arquivos: o **Hibernate Core**, o **Hibernate Annotation** e o **Hibernate Validator**. O primeiro é o núclo do Hibernate, o segundo as extensões para trabalharmos com Annotation e o último as extensões para validações das entidades.

Extraia os arquivos em um diretório qualquer do seu sistema. Deverão ser criados três novas pastas: a hibernate-3.2 (o final do nome varia de acordo com a versão), a hibernate-annotation-3.3.0.GA e a hibernate-validator-3.0.0.GA.

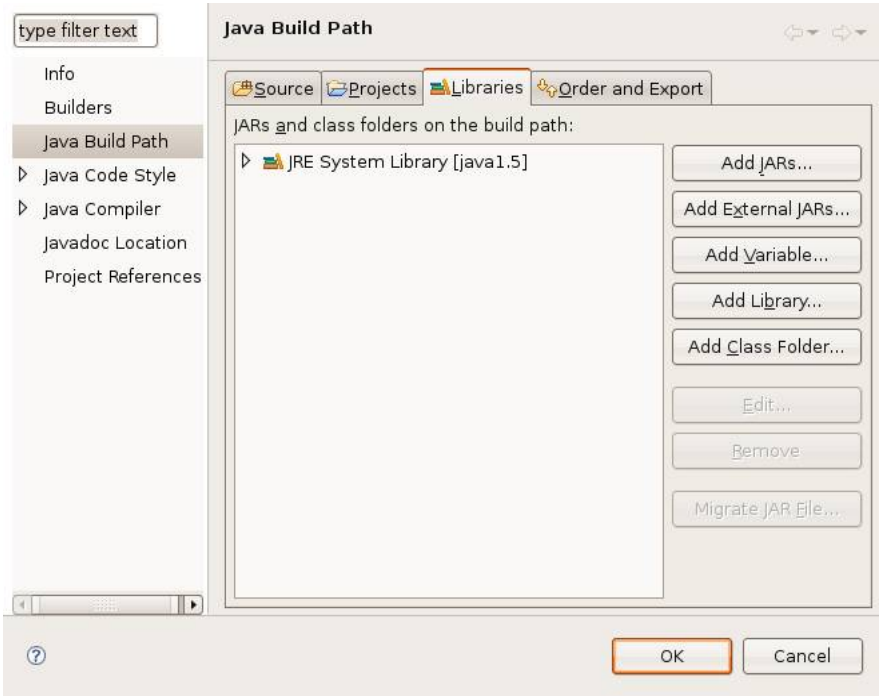
Vamos agora adicionar as bibliotecas ao nosso projeto. Basicamente, os seguintes arquivos:

hibernate-3.2.5\hibernate3.jar
hibernate-3.3.5\lib*
hibernate-annotations-3.3.0.GA\hibernate-annotation.jar
hibernate-annotations-3.3.0.GA\lib*
hibernate-validator-3.0.0.GA\hibernate-validator.jar
hibernate-validator-3.0.0.GA\lib*

Temos duas opções: a primeira seria adicionar esses arquivos ao CLASSPATH do seu sistema ou, a forma que iremos utilizar, adicionar as bibliotecas ao BUILD PATH do projeto:

1) Primeiro, clique com o botão direito em cima do nome do projeto e selecione a opção "**Build Path**" --> "**Configure Build Bath**"

2) Vamos criar uma biblioteca chamada "**Hibernate**". Clique na opção "**Add Library**".

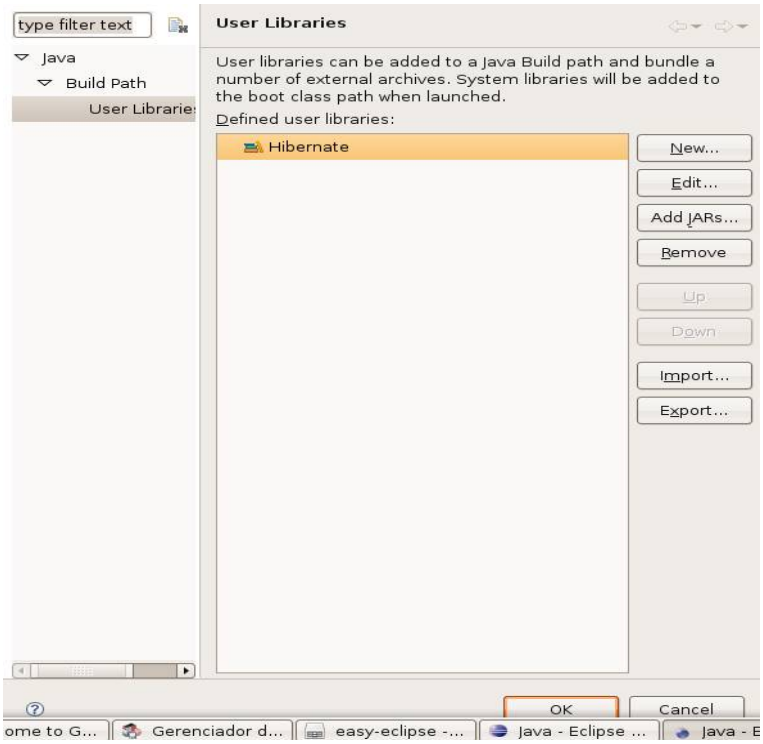


3) Agora, selecione a opção "**User Library**" e clique em **Next** (Próximo).

4) Nessa tela, clique no botão "**User Library**".

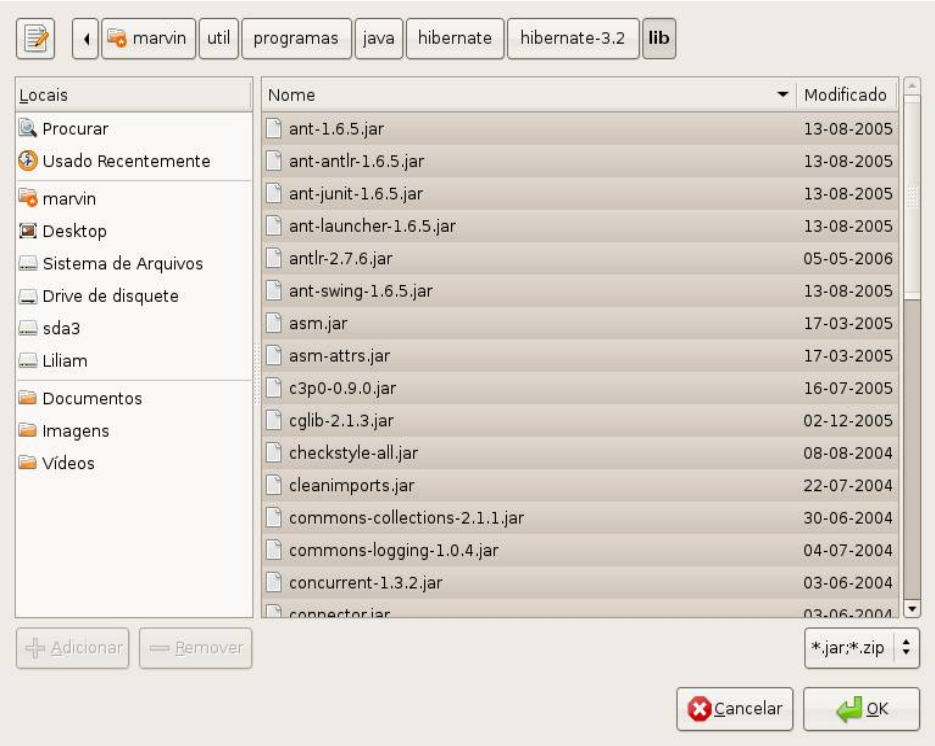
5) Em seguida, vamos criar uma nova Biblioteca. Clique no botão "**New**" (Novo). Será solicitado o nome da biblioteca. Digite "**Hibernate**" e clique em "**OK**".

6) Sua nova biblioteca foi criada.



7) O próximo passo é adicionar os JAR's do Hibernate à nova biblioteca. Com a biblioteca criada marcada, clique no botão "Add JARs". Um nova janela será aberta onde devemos escolher o JAR. Navegue até o diretório "hibernate-3.2.5" e selecione o arquivo "hibernate3.jar".

8) Repita o passo 7 novamente, mas dessa vez adicione todos os JAR's que estão dentro do diretório "hiberbate-3.3.5\lib.



9) Vamos criar nossa segunda biblioteca. Clique no botão "**new**" e digite o nome "**Hibernate-Annotation**".

10) Com a biblioteca criada marcada, clique no botão "**Add JARs**". Um nova janela será aberta onde devemos escolher o JAR. Navegue até o diretório "hibernate-annotations-3.3.0.GA" e selecione o arquivo "hibernate-annotation.jar".

11) Repita o passo 10 novamente, mas dessa vez adicione todos os JAR's que estão dentro do diretório "hibernate-annotations-3.3.0.GA\lib.

12) Repita os procedimentos acima para criar a terceira biblioteca, contendo os arquivos do Hibernate Validator.

Nesse nosso artigo, iremos utilizar o banco de dados **MySQL**, por isso devemos também adicionar o driver JDBC do mesmo. O download por ser feito aqui: http://dev.mysql.com/downloads/connector/j/

Repita o processo acima para adicionar driver JDBC do MySQL ao **Build Path** do projeto.

Nesse mini-tutorial, não irei demonstrar como utilizar o MySQL. Em caso de dúvidas sobre a criação do database e das tabelas, recomendo esse site:
http://dev.mysql.com/doc/refman/4.1/pt/tutorial.html

Para o nossos teste é necessário um *database* chamado **teste** e uma tabela chamada **cliente** de acordo com a seguinte DDL:

```
CREATE TABLE cliente (
        id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
        nome_cliente VARCHAR(100)  NOT NULL,
        idade INTEGER  NOT NULL,
        PRIMARY KEY(id)
);
```

1.3 CONFIGURANDO O HIBERNATE UTIL

Em projetos que usam a framework Hibernate, é bastante comum a criação de um classe estática chamada de "HibernateUtil". A função dessa classe é encapsular a criação e recuperação de sessões(detalhes mais a frente).

```
package teste;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateUtil {
        private static final SessionFactory sessionFactory;
        static {
        try {
          // Create the SessionFactory from hibernate.cfg.xml
          sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory();
          } catch (Throwable ex) {
             // Make sure you log the exception, as it might be swallowed
             System.err.println("Initial SessionFactory creation failed." + ex);
             throw new ExceptionInInitializerError(ex);
          }
        }
        public static SessionFactory getSessionFactory() {
          return sessionFactory;
        }
}
```

Perceba o uso da classe "**AnnotationConfiguration**". Os pacotes e as classes anotadas são declaradas em um arquivo XML regular, geralmente o "**hibernate.cfg.xml**". Além disso, as propriedades de acesso ao banco também foram definidas nesse arquivo. A seguir, temos um exemplo:

```
<?xml version='1.0' encoding='utf-8'?>
        <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
        <hibernate-configuration>
        <session-factory>
                <!-- Database connection settings -->
                <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
                <property name="connection.url">jdbc:mysql://localhost/teste</property>
                <property name="connection.username">root</property>
                <property name="connection.password"></property>
                <!-- JDBC connection pool (use the built-in) -->
                <property name="connection.pool_size">1</property>
                <!-- SQL dialect -->
                <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
```

```
                <!-- Enable Hibernate's automatic session context management -->
                <property name="current_session_context_class">thread</property>
                <!-- Disable the second-level cache -->
                <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
                <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>
        </session-factory>
</hibernate-configuration>
```

Crie esse arquivo dentro do diretório-raiz do seu projeto.

1.4 CRIANDO A PRIMEIRA CLASSE PERSISTENTE

Nossa primeira classe persistente será um **JavaBean** padrão:

```
package teste;

import java.io.Serializable;

public class Cliente implements Serializable {

        private Long id;
        private String nome;
        private Long idade;

        public Cliente(){

        }

        public Long getId() {
                return id;
        }

        private void setId(Long id) {
                this.id = id;
        }
        public Long getIdade() {
                return idade;
        }

        public void setIdade(Long idade) {
                this.idade = idade;
        }

        public String getNome() {
                return nome;
        }

        public void setNome(String nome) {
                this.nome = nome;
        }
}

//Fim da Classe
```

Temos, basicamente, atributos privados e métodos acessores (getters and setters) para cada um dos atributos. O construtor vazio é necessário para instanciar um objeto via reflexão.

A propriedade **id** representa um identificador único para um cliente. Todas as classes de entidade (entity classes) persistentes precisarão de tal atributo se quisermos utilizar todos os recursos do Hibernate. Outro detalhe, como dificilmente iremos manipular o valor do atributo **id**, seu método "**set**" foi definido como privado. Somente o Hibernate irá registrar um valor a esse campo quando um determinado objeto for salvo. Mas lembre-se, isso não é uma regra. Se a sua aplicação precisar manipular o campo **id**, nada impede de você trocar a visibilidade para **public**.

Bom, como nossa classe é uma classe de entidade, devemos "anotá-la" com a propriedade especifica. Para isso, usaremos a anotação **@Entity** antes da definição da classe:

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

@Entity declara a classe como uma classe de entidade (ou uma classe persistente).

Agora, vamos alterar o nosso código adicionando a anotação **@Id** ao atributo id:

```
 @Entity
public class Cliente implements Serializable {

        @Id
        private Long id;
        . . .
}
```

@Id declara qual campo será usado como identificador. Neste caso, definimos nosso atributo "id".

1.5 DEFININDO A TABELA:

A anotação **@Table** será usada para definir qual tabela será usada para persistir os objetos dessa classe. Detalhe: se essa anotação não for usado, o Hibernate irá procurar por uma tabela com o mesmo nome da classe.

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

1.5 MAPEANDO OS ATRIBUTOS

Usaremos a anotação **@Column** para definir as propriedades dos nossos atribtuos:

```
@Column(name = "nome_cliente", nullable = false, length=60)
private String nome;

@Column(name = "idade_cliente", nullable = false, length = 5)
private Long idade;
```

A propriedade **nome** é mapeada para uma coluna chamada "nome_cliente", enquanto o atributo **idade** é mapeado para uma coluna chamada "idade_cliente".
Se o nome da coluna da tabela fosse igual ao nome do atributo, não seria necessário especificar a propriedade "name" da anotação @Column.

Propriedades de @Column:

```
@Column(
        name="columnName";
        // o nome da coluna (padrão é o nome do atributo)

        boolean unique() default false;
        // determina se o valor do campo é único ou não ( padrão falso)

        boolean nullable() default true;
        // determinal se a coluna aceita valores nulos (padrão falso)

        boolean insertable() default true;
        // se a coluna irá fazer parte da setença SQL de inserção

        boolean updatable() default true;
        // se a coluna irá fazer parte da setença SQL de alteração

        String columnDefinition() default "";
        // sobrescreve o fragmento de DDL para esta coluna

        String table() default "";
        // define a tabela alvo (padrão a tabela primária)

        int length() default 255;
        // tamanho da coluna (padrão 255)

        int precision() default 0; // decimal precision
        // precisão da coluna decimal

        int scale() default 0; // decimal scale
        // escala da coluna decimal
)
```

1.6 MAPEANDO ATRIBUTOS IDENTIFICADORES:

A anotação **@Id** permite indicar qual atributo da classe será usado como identificador. Além disso, podemos usar a anotação **@GeneratedValue** para indicar a estratégia de geração dos identificadores. Aqui usaremos apenas a estratégia **AUTO**, o qual fica a cargo do banco determinar como será gerado o valor do atributo. Para mais informações sobre as outras estratégias, recomendo a documentação oficial do Hibernate.

```
@Id  @GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

1.7 CRIANDO O DAO PARA NOSSO BEAN

Antes de testarmos a classe Cliente, vamos criar um **DAO** para ele. Outra vez, estou assumindo que você entenda o padrão **DAO**. Mas, bem resumidamente, corresponde a uma classe responsável pelas operações a uma determinada tabela do banco. No nosso caso a tabela Cliente.

```
package teste.dao;

import org.hibernate.Session;

import teste.Cliente;
import teste.HibernateUtil;

public class ClienteDAO {
	private Session session;

	public ClienteDAO(){
	}

	public void salvar(Cliente cli){
		session = HibernateUtil.getSessionFactory().getCurrentSession();
		session.beginTransaction();
		session.save(cli);
		session.getTransaction().commit();
	}

}
```

Perceba que o método **salvar** recebe um objeto cliente como parâmetro e o mesmo é persistido no banco pelo Hibernate.

Vamos entender o que aconteceu. Primeiramente, recuperamos a **sessão** (*session*) graças à nossa classe **HibernateUtil**. Sem entrar em detalhes nesse momento, uma **sessão** (session) seria mais ou menos como uma unidade de trabalho, ou uma transação com o banco de dados (lembrando, essa definição é "bastante" resumida). Ela nos "protege" dos "detalhes sangrentos" que dizem respeito ao acesso à nossa tabela: sql's, transações, restrições e etc.

Depois, usamos nossa sessão para iniciar uma transação, salvar nosso objeto no banco de dados e, por último, "***commitar***" nossa transação.

A sessão é iniciada sempre que o método **getCurrentSession** é invocado e terminado atráves de um **commit** ou **rollback**.

1.8 ADICIONANDO A CLASSE CLIENTE AO HIBERNATE.CFG.XML

Para que o Hibernate saiba quais as classe ele vai ter que tratar, devemos adicionar ao **hibernate.cfg.xml** a propriedade ***mapping***, indicando o nome da classe.

```
<hibernate-configuration>
   <session-factory>

        ...
        <mapping class="teste.Cliente"/>

   </session-factory>
</hibernate-configuration>
```

1.9 TESTANDO A APLICAÇÃO:

Para testar a aplicação, criaremos uma classe chamada **Principal** com um método ***main:***

```
public class Principal {

        /**
         * @param args
         */
        public static void main(String[] args) {

                Cliente c1 = new Cliente();
                c1.setNome("Raul Seixas");
                c1.setIdade(new Long(12));

                Cliente c2 = new Cliente();
                c2.setNome("Bruce Dickinson");
                c2.setIdade(new Long(13));

                ClienteDAO clienteDAO = new ClienteDAO();
                clienteDAO.salvar(c1);
                clienteDAO.salvar(c2);
        }
}
```

A classe é bastante simples. Criamos dois objetos do tipo Cliente (**c1** e **c2**), “setamos” seus atributos atráves dos métodos **set's** e, em seguida, instanciamos um objteto do Tipo ClienteDAO para persistir os dois objetos clientes criados anteriormente. A saída gerada no console deverá ser algo parecido com:

Hibernate: insert into cliente (id, idade, nome_cliente) values (null, ?, ?)
Hibernate: insert into cliente (id, idade, nome_cliente) values (null, ?, ?)

O Hibernate está configurado para imprimir, no console, os sql's gerados pela aplicação.

1.10 LISTANDO OS CLIENTES

Vamos criar um novo método no DAO para recuperar a lista de Clientes:

```
public List listar(){
        session = HibernateUtil.getSessionFactory().openSession();
        session.beginTransaction();
        List l1 = session.createQuery("from Cliente").list();
        session.getTransaction().commit();
        return l1;
}
```

Usamos um método da sessão chamado **createQuery**, o qual nos permite usar uma **HQL**(Hibernate Query Language) para recuperar registros do banco. HQL consiste em um diatelo SQL para o Hibernate. É uma poderosa ferramenta de consulta que, apesar de se parecer com o SQL, é totalmente orientado a objetos. Para saber mais sobre HQL's, recomendo o seguinte site:

http://www.hibernate.org/hib_docs/reference/en/html/queryhql.html

Na classe Principal, vamos adicionar o seguinte código para imprimir a relação dos clientes cadastrados:

```
List<Cliente>  listaClientes = clienteDAO.listar();

for (Cliente cliente : listaClientes) {
        System.out.println(cliente.getId()+" - "+cliente.getNome());
}
```

O resultado gerado no console deverá ser algo mais ou menos assim:

Hibernate: insert into cliente (id, idade, nome_cliente) values (null, ?, ?)
Hibernate: insert into cliente (id, idade, nome_cliente) values (null, ?, ?)
Hibernate: select cliente0_.id as id0_, cliente0_.idade as idade0_,
cliente0_.nome_cliente as nome3_0_ from cliente cliente0_
1 - Raul Seixas
2 - Bruce Dickinson

PARTE II

2 – MAPEAMENTO DE ASSOCIAÇÕES

Neste tópico iremos aprender como mapear as principais associações entre os beans.

2.1 MANY-TO-ONE

Podemos definir as associações ***many-to-one*** a nível de propriedade com a anotação ***@ManyToOne***. Por exemplo:

```
@Entity
@Table(name = "cliente")
public class Cliente implements Serializable {

. . .
        @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE} )
        @JoinColumn(name="cat_id")
        private Categoria categoria;
. . .
}
```

Neste exemplo, criamos uma associação com a classe Categoria. Um cliente cadastrado deve estar associado a uma categoria existente.

A anotação **@JoinColumn** serve para especificar qual campo da tabela(neste caso a tabela **cliente**) será usado como chave estrangeria. Entretanto **@JoinColum** não é obrigatória, caso seja omitida, será usado o valor padrão. O hibernate buscará por um campo cujo nome seja formado pelo nome da classe "dono" da relação, um "*underscore*" e o nome **id**. Assim, para o exemplo anterior, o nome do campo procurado seria **categoria_id**.

@ManyToOne possui um atributo chamado ***targetEntity*** o qual descreve o nome da entidade alvo. Geralmente não precisamos usá-lo, pois o valor padrão(o tipo da propriedade que armazena a associação) serve para a maioria dos casos. Entretanto, este atributo é útil quando queremos usar interfaces como o tipo de retorno ao invés da entidade regular. Assim:

```
@Entity
@Table(name = "cliente")
public class Cliente implements Serializable {

. . .
        @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        targetEntity = Categoria.class)
        @JoinColumn(name="categoria_id")
        private Categoria categoria;
. . .
}
```

Abaixo, temos o bean Categoria:

```
@Entity
@Table(name="categoria")
public class Categoria {

	@Id @GeneratedValue(strategy=GenerationType.AUTO)
	private Long id;

	@Column(nullable=false)
	private String nome;

	//Getters and Setters
}
```

A DDL para a criação para a tabela **Categoria:**

```
CREATE TABLE categoria (
  id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  nome VARCHAR(20) NOT NULL,
  PRIMARY KEY(id)
);
```

A DDL para inclusão do campo **categoria_id** na tabela **Cliente**:

```
ALTER TABLE `cliente` ADD COLUMN `categoria_id` INTEGER NOT NULL;
```

2.1.1 ALTERANDO A APLICAÇÃO

Antes de usarmos a nova classe, devemos adicioná-la ao ***hibernate.cfg.xml***:

```
<mapping class="br.com.facid.padroes.Cliente"/>
<mapping class="br.com.facid.padroes.Categoria"/>
```

Bem como criar o DAO:

```
public class CategoriaDAO {
	private Session session;
	public void salvar(Categoria ct){
		session = HibernateUtil.getSessionFactory().openSession();
		session.beginTransaction();
		session.save(ct);
		session.getTransaction().commit();
	}
}
```

Agora vamos modificar a classe **Principal:**

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

		Categoria ct1 = new Categoria();
		ct1.setNome("Especial");

		Cliente c1 = new Cliente();
		c1.setNome("Raul Seixas");
		c1.setIdade(new Long(32));
		c1.setFormacao(f1);
		c1.setCategoria(ct1);

		Cliente c2 = new Cliente();
		c2.setNome("Bruce Dickinson");
		c2.setIdade(new Long(45));
		c2.setFormacao(f1);
		c2.setCategoria(ct1);

		CategoriaDAO categoriaDAO = new CategoriaDAO();
		categoriaDAO.salvar(ct1);

		ClienteDAO clienteDAO = new ClienteDAO();
		clienteDAO.salvar(c1);
		clienteDAO.salvar(c2);


		List<Cliente>  listaClientes = clienteDAO.listar();
		for (Cliente cliente : listaClientes) {
			System.out.println(cliente.getId()+" - "+cliente.getNome() + " - " +
cliente.getCategoria().getNome());
		}
	}

}
```

Executando a aplicação, teremos a seguinte saída no console:

Hibernate: insert into categoria (id, nome) values (null, ?)
Hibernate: insert into cliente (id, categoria_id, idade, nome_cliente) values (null, ?, ?, ?)
Hibernate: insert into cliente (id, categoria_id, idade, nome_cliente) values (null, ?, ?, ?)
Hibernate: select cliente0_.id as id0_, cliente0_.categoria_id as categoria4_0_,
cliente0_.idade as idade0_, cliente0_.nome_cliente as nome3_0_ from cliente cliente0_
Hibernate: select categoria0_.id as id2_0_, categoria0_.nome as nome2_0_ from
categoria categoria0_ where categoria0_.id=?
21 - Raul Seixas - Especial
22 - Bruce Dickinson – Especial

3 – MAPEANDO CONSULTAS(QUERYS) HQL

Podemos mapear consultas HQL's usando a anotação ***@NamedQuery***. Sua definição é feita antes da definição da classe. Por exemplo:

```
@Entity
@Table(name = "cliente")
@NamedQuery(name="cliente.igual", query="select c from Cliente c where
c.nome=:nome")
public class Cliente implements Serializable {

...

}
```

Criamos uma *NamedQuery* chamado **cliente.igual** e definimos sua HQL.

Vamos alterar o DAO, adicionando um novo método responsável por executar a *NamedQuery*:

```
public class ClienteDAO {
	...
	public List recuperarClientePorNome(String nomeConsultar){
		session = HibernateUtil.getSessionFactory().openSession();
		Query q = session.getNamedQuery("cliente.igual");
		q.setString("nome", nomeConsultar);
		List resultado = q.list();
		session.getTransaction().commit();
		return resultado;
	}
}
```

E na classe **Princial**, vamos adicionar o seguinte código:

```
List<Cliente> listaCliente = clienteDAO.recuperarClientePorNome("Raul Seixas");

for (Cliente cliente : listaCliente) {
	System.out.println("Nome do Cliente...:" + cliente.getNome());
}
```

CONSIDERAÇÕES FINAIS

Chegamos ao fim do nosso artigo. Como foi dito anteriormente, nossa proposta consistia apenas em um ponta-pé inicial para quem deseja começa a usar Hibernate com Annotations. Com certeza falta muita coisa a ser dita. Quem sabe em artigos futuros?