Artigo – http://mballem.wordpress.com/

Utilizando Swing com JPA/Hibernate

Autor

Marcio Ballem: é formado em Sistemas de Informação e possui certificação Oracle Certified Professional, Java SE 6 Programmer. Trabalhou profissionalmente com desenvolvimento em Delphi 7 e Java JEE.

Introdução

Este tutorial poderia ser uma segunda parte do tutorial anterior *Utilizando Swing com Banco de Dados* (http://mballem.wordpress.com/2011/02/21/utilizando-swing-com-banco-de-dados/), a classe de interface com usuário é 99% a mesma, a classe *controller* idem.

O que está diferente aqui é que utilizaremos o *framework Hibernate*, para a persistência com banco de dados, com JPA (*Java Persistence API*).

Utilizaremos desta vez o banco de dados *HSQLDB* no modo *standalone*, mas quem preferir pode continuar utilizando o *MySql* como no tutorial anterior, é só utilizar as mesmas configurações de url, drive, usuário e senha, quando for necessário utilizar.

O padrão MVC também será seguido como no tutorial anterior, do link acima, então caso não tenha lido o tutorial anterior, seria importante ler a respeito do que é o padrão MVC.

1. Arquivos necessários

Para o desenvolvimento deste tutorial serão necessárias algumas bibliotecas referentes ao Hibernate e a JPA, e mais algumas que necessitam ser utilizadas como algumas dependências. E também é claro do Driver JDBC do HSQLDB ou do MySql.

Veja as bibliotecas necessárias na figura 1.

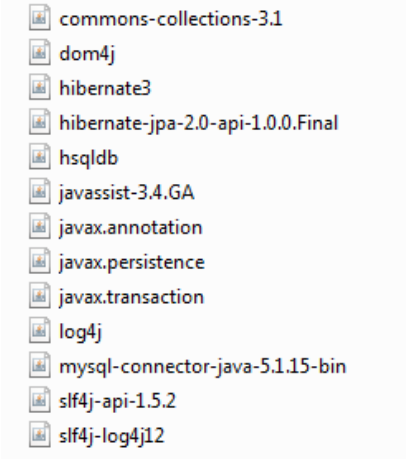

Figura 1 - Bibliotecas 1

Montei um pacote com todas as bibliotecas e as disponibilizei neste para download neste link (http://www.megaupload.com/?d=UHR7G6WI)

2. Java Persistence API e Hibernate

JPA é uma especificação padrão do Java para persistência de dados. A JPA define um meio de mapeamento objeto-relacional para objetos Java, os chamados *POJOs*, denominados *beans* de entidade.

Diversos *frameworks* de mapeamento objeto/relacional, como o *Hibernate*, implementam a JPA gerenciando o desenvolvimento de entidades do Modelo Relacional usando a plataforma nativa Java SE e Java EE.

O *Hibernate* é um *framework* para mapeamento objeto-relacional, totalmente escrito em Java. O principal objetivo do *Hibernate* é diminuir a complexidade entre os programas escritos em Java, baseando-se no modelo orientado objeto.

Quando utilizamos o *HIbernate*, deixamos de trabalhar com tabelas e colunas, e passamos a trabalhar com objetos. Quem já desenvolveu em Java com banco de dados e utilizando *ResultSet* e *PreparedStatement* para criar consultas e outras ações com o banco de dados, trabalhou diretamente com tabelas e colunas. Utilizando o *Hibernate*, as tabelas e colunas são mapeadas por mapeamentos XML ou anotações, e a partir daí trabalhamos apenas com objetos, o que facilita em muito o desenvolvimento.

3. Arquivo persistence.xml

Quando utilizamos JPA, precisamos utilizar um arquivo que contem algumas configurações importantes para a conexão com o banco de dados através do *Hibernate*. Este arquivo deve ser chamado de ***persistence.xml*** e estar dentro de um diretório chamado ***META-INF*** no *classpath* da aplicação, ou seja, no mesmo nível do pacote br, o primeiro pacote dos fontes, veja na figura 2.

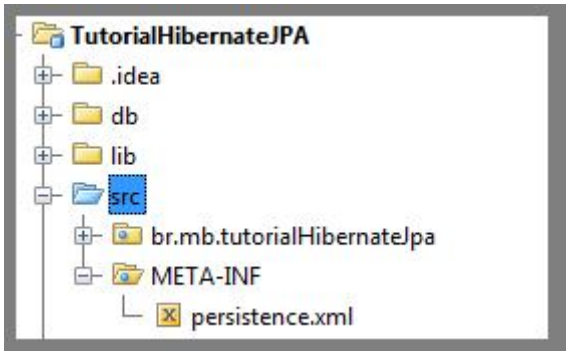

Figura 2 - Projeto

Listagem 1. Arquivo persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">

    <persistence-unit name="agenda" transaction-type="RESOURCE_LOCAL">
        <class>br.mb.tutorialHibernateJpa.model.Contato</class>
        <properties>
            <property name="hibernate.connection.url"
value="jdbc:hsqldb:file:./db/agenda"/>
```

```
            <property name="hibernate.connection.driver_class"
value="org.hsqldb.jdbcDriver"/>
            <property name="hibernate.connection.username" value="SA"/>
            <property name="hibernate.connection.password" value=""/>
            <property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.hbm2ddl.auto" value="update"/>
        </properties>
    </persistence-unit>

</persistence>
```

A tag ***persistence-unit*** possui dois elementos importantes, o ***name*** que recebe um nome para identificar as configurações e o ***transaction-type*** que indica se aplicação será com ou sem container (servidor web).

Precisamos também indicar neste arquivo onde se encontra o mapeamento, para isso, usamos a tag ***class*** e nela indicamos a classe que contém as anotações referentes a tabela Contatos. As demais tags, filhas de ***properties***, são referentes as configurações do *Hibernate*, como as configurações para conexão e para exibir alguns *logs* no console dos SQLs executados.

4. Tabela contatos

Desta vez não iremos precisar de script para criar a tabela Contatos no banco de dados, a JPA nos fornece uma maneira simples para isso. Basta a criação de um método que através dos mapeamentos as tabelas são geradas. Para criar o banco de dados basta executar a classe principal *GeraBanco*, conforme a listagem 2.

Listagem 2. Gerando o banco de dados.

```
package br.mb.tutorialHibernateJpa.agenda;

import org.hibernate.cfg.Configuration;
import org.hibernate.ejb.Ejb3Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class GeraBanco {
    public static void main(String[] args) {
        Ejb3Configuration cfg = new Ejb3Configuration();

        //agenda eh o nome do persistence-unit no persistence.xml.
        cfg.configure("agenda", null);

        Configuration hbmcfg = cfg.getHibernateConfiguration();

        SchemaExport schemaExport = new SchemaExport(hbmcfg);
        schemaExport.create(true, true);
    }
}
```

Ao gerar o banco de dados *HSQLDB*, será criada uma pasta chamada “db” e dentro dela estarão os arquivos do banco de dados. Esta pasta será gerada um nível acima do pacote dos arquivos fontes da aplicação, ou seja, no mesmo nível em que você deve ter o pacote “lib”, veja na figura 2.

Quem utilizar um banco de dados no modo servidor, como o *Mysql* ou outros, algumas vezes será necessário criar o banco de dados manualmente no gerenciador, para então rodar a classe *GeraBanco*, que ela irá criar as tabelas.

5. Classe de Conexão

Trabalhando com JPA/Hibernate, teremos uma classe de conexão diferente do que temos quando criamos uma conexão direta por JDBC. No caso da JPA devemos criar uma classe conforme a listagem 3. Essa classe vai ler as configurações do arquivo ***persistence.xml*** e configurar uma "fabrica de conexões" para o sistema. Sempre que for necessária uma conexão, faremos uma chamada ao método estático ***getEntityManager()***.

Listagem 3. Classe EntityManagerUtil

```
package br.mb.tutorialHibernateJpa.dao;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class EntityManagerUtil {
  private static EntityManagerFactory emf;

        public static EntityManager getEntityManager() {
                if (emf == null){
                        emf = Persistence.createEntityManagerFactory("agenda");
                }
                return emf.createEntityManager();
        }
}
```

6. Classe GenericDao

Vamos criar uma classe genérica para os métodos *insert*, *update* e *delete,* e algumas consultas que podem ser padrão para várias entidades, veja na listagem 4.

Listagem 4. Classe GenericDao

```
package br.mb.tutorialHibernateJpa.dao;

import org.hibernate.Session;
import org.hibernate.criterion.Restrictions;

import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;
import javax.persistence.PersistenceContext;
import java.io.Serializable;
import java.lang.reflect.ParameterizedType;
import java.util.List;

public class GenericDao<T extends Serializable> {

    @PersistenceContext(unitName = "agenda")
    private final EntityManager entityManager;
    private final Class<T> persistentClass;
```

```
    public GenericDao() {
        this.entityManager = EntityManagerUtil.getEntityManager();
        this.persistentClass = (Class<T>) ((ParameterizedType)
getClass().getGenericSuperclass()).getActualTypeArguments()[0];
    }

    public EntityManager getEntityManager() {
        return entityManager;
    }

    protected void save(T entity) {
        EntityTransaction tx = getEntityManager().getTransaction();

        try {
            tx.begin();
            getEntityManager().persist(entity);
            tx.commit();
        } catch (Throwable t) {
            t.printStackTrace();
            tx.rollback();
        } finally {
            close();
        }
    }

    protected void update(T entity) {
        EntityTransaction tx = getEntityManager().getTransaction();

        try {
            tx.begin();
            getEntityManager().merge(entity);
            tx.commit();
        } catch (Throwable t) {
            t.printStackTrace();
            tx.rollback();
        } finally {
            close();
        }

    }

    protected void delete(T entity) {
        EntityTransaction tx = getEntityManager().getTransaction();

        try {
            tx.begin();
            getEntityManager().remove(entity);
            tx.commit();
        } catch (Throwable t) {
            t.printStackTrace();
            tx.rollback();
        } finally {
            close();
        }
    }

    public List<T> findAll() throws Exception {
        Session session = (Session) getEntityManager().getDelegate();
        return session.createCriteria(persistentClass).list();
```

```
    }

    public T findByName(String nome) {
        Session session = (Session) getEntityManager().getDelegate();
        return (T)
session.createCriteria(persistentClass).add(Restrictions.eq("nome",
nome).ignoreCase()).uniqueResult();
    }

    public T findById(long id) {
        Session session = (Session) getEntityManager().getDelegate();
        return (T)
session.createCriteria(persistentClass).add(Restrictions.eq("id",
id)).uniqueResult();
    }

    private void close() {
        if (getEntityManager().isOpen()) {
            getEntityManager().close();
        }
        shutdown();
    }

    private void shutdown() {
        EntityManager em = EntityManagerUtil.getEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        em.createNativeQuery("SHUTDOWN").executeUpdate();
        em.close();
    }
}
```

A anotação ***@PersistenteContex()*** indica qual classe será responsável pela persistência dos dados. A variável ***persistentClass*** é utilizada para termos acesso a entidade que está sendo executada no momento. Utilizamos o tipo genérico na classe, e é através do tipo genérico que obtemos a entidade para utilizá-la nas consultas, veremos mais a frente.

No construtor da classe criamos um objeto ***EntityManager***, que nos dará acesso ao métodos necessários, e obtemos a entidade que está utilizando a classe no momento.

Nos métodos *save*, *update* e *delete*, precisamos criar uma transação, abrir está transação e então executar o que queremos fazer. No caso do método *save()*, utilizamos o *persist()*, que recebe como parâmetro um objeto e o *framework* executa um *insert* através dele. A grande vantagem é que não precisamos nos preocupar com o SQL de *insert*, *update* ou *delete*, só passamos o objeto e o *framework* faz o resto.

Depois da execução, se tudo ocorrer bem é executado um *commit* na transação do banco de dados e então um close é chamado para fechar a comunicação com o banco de dados.

Uma observação para quem não está utilizando o HSQLDB, o método ***shutdown()*** não deve ser utilizado, este método é utilizado apenas no HSQLDB. No artigo *JDBC com Banco de Dados Standalone* (http://mballem.wordpress.com/2011/02/02/jdbc-com-banco-de-dados-standalone/) eu falo sobre isso.

Nas consultas foi utilizada a API *Criteria*, uma forma diferente de fazer o "*select*" quando utilizamos orientação objetos e não objetos relacionais. Outra forma que pode ser utilizada é o HQL, um dialeto SQL para o *Hibernate*, uma consulta muito parecida com o SQL, porém se trabalha com objetos e seus atributos e não com tabelas e colunas.

Quando utilizamos *Criteria*, precisamos indicar qual a entidade que fará a consulta, por isso, utilizamos a variável ***persistentClass***, para passarmos para o método qual entidade está sendo executada no momento da consulta. Seria como se ele estivesse passando por parâmetro algo como isto: *Contato.class*

7. Classe Contato

Nossa classe contato terá como atributos os campos da tabela Contatos, mapeados em forma de anotações. Através das anotações podemos passar todas as informações que uma coluna teria, como o tipo de dado, tamanho, nome, entre outros.

Listagem 5. Classe Contato

```
package br.mb.tutorialHibernateJpa.model;

import java.sql.Date;
import javax.persistence.*;
import java.io.Serializable;

@Entity
@Table(name = "CONTATOS")
public class Contato implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID_CONTATO")
    private Long id;
    @Column(name = "NOME")
    private String nome;
    @Column(name = "APELIDO")
    private String apelido;
    @Temporal(TemporalType.DATE)
    @Column(name = "DATA_NASCIMENTO")
    private Date dtNascimento;
    //gere os métodos getters and setters
}
```

8. Classe ContatoDao

Criamos a classe *GenericDao* para ser herdadas pelos demais dao's, assim, vamos agora criar a classe *ContatoDao*, onde estarão os métodos mais específicos da classe. Veja na listagem 6.

Listagem 6. Classe ContatoDao

```
package br.mb.tutorialHibernateJpa.dao;

import br.mb.tutorialHibernateJpa.model.Contato;

public class ContatoDao extends GenericDao<Contato> {

    public void salvar(Contato contato) {
        save(contato);
    }

    public void alterar(Contato contato) {
        update(contato);
```

```
    }

    public void excluir(long id) {
        Contato c = findById(id);
        delete(c);
    }

}
```

Veja que utilizamos herança nesta classe, herdando os métodos criados na classe *GenericDao*. Nossos métodos *salvar()*, *alterar()* e *excluir()* não possuem mais a criação do SQL que será executado no banco de dados, como no artigo anterior, e sim uma chamada ao método da classe *GenericDao* e lá os métodos do *EntityManager* farão o trabalho.

9. Classe ContatoController

Pouquíssimas alterações serão feitas na classe controller em relação a classe controller do artigo anterior. Apenas foram alterados o nome do pacote do projeto, as exceções que antes eram *SQLException* para *Exception* e no método *listaContatos()* o retorno foi alterado de *dao.findContatos()* para *dao.findAll()*, já que agora temos um método genérico e não exclusivo para contatos.

Essa é uma das vantagens do padrão MVC, nós podemos modificar parte do projeto sem que influencie drasticamente em outros pontos dele.

Listagem 7. Classe ContatoController

```
package br.mb.tutorialHibernateJpa.controller;

import br.mb.tutorialHibernateJpa.dao.ContatoDao;
import br.mb.tutorialHibernateJpa.model.Contato;

import javax.swing.*;
import java.sql.Date;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.List;

public class ContatoController {

    private Date formatarData(String data) throws ParseException {
        DateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
        return new Date( formatter.parse(data).getTime() );
    }

    public void salvar(String nome, String apelido, String dtNascimento) throws
Exception {
        Contato contato = new Contato();
        contato.setNome(nome);
        contato.setApelido(apelido);
        contato.setDtNascimento(formatarData(dtNascimento));

        new ContatoDao().salvar(contato);
    }

    public void alterar(long id, String nome, String apelido, String dtNascimento)
throws Exception {
```

```
        Contato contato = new Contato();
        contato.setId(id);
        contato.setNome(nome);
        contato.setApelido(apelido);
        contato.setDtNascimento(formatarData(dtNascimento));

        new ContatoDao().alterar(contato);
    }

    public List<Contato> listaContatos() {
        ContatoDao dao = new ContatoDao();
        try {
            return dao.findAll();
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, "Problemas ao localizar contato\n"
+ e.getLocalizedMessage());
        }
        return null;
    }

    public void excluir(long id) throws Exception {
        new ContatoDao().excluir(id);
    }

    public Contato buscaContatoPorNome(String nome) throws Exception {
        ContatoDao dao = new ContatoDao();
        return dao.findByName(nome);
    }
}
```

9. Classe ContatoFrame

A interface continua a mesma, uma interface como a da figura 3.

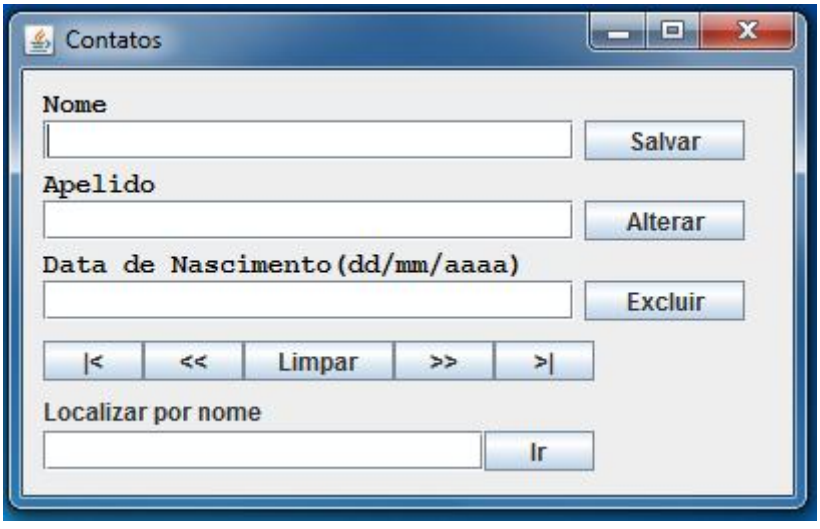

Figura 3 - Interface Contatos

A classe *ContatoFrame* sofreu alterações mínimas, que foram apenas, alterar as exceções *SQLException* por exceções do tipo *Exception*. Faça essa alteração e nos casos dos métodos que tenham dois tipos de exceções, deixe a *Exception* no último *catch{}*.

Essa classe não foi postada no artigo, mas você pode pegá-la no tutorial anterior, acessando *Utilizando Swing com Banco de Dados* (http://mballem.wordpress.com/2011/02/21/utilizando-swing-com-banco-de-dados/), e é claro, não se esqueça de alterar o nome do pacote e dos *imports* de *ContatoController* e *Contato*.

Conclusão

Este tutorial demonstrou como configurar e criar um projeto utilizando JPA com *Hibernate*. Fazendo uma comparação com o tutorial "*Utilizando Swing com Banco de Dados"*, onde utilizamos conexão direta com JDBC e utilizávamos *PreparedStatement* e *ResultSet*, e os SQLs para consultas e demais métodos com banco de dados, o *framework Hibernate* nos facilita muito nesse ponto.

Também podemos comparar como é válido separar as camadas da aplicação utilizando o padrão MVC, assim, facilita muito a manutenção dos códigos.

Saiba mais

- Criteria http://download.oracle.com/javaee/6/tutorial/doc/gjitv.html
- Genéricos em Java http://en.wikipedia.org/wiki/Generics_in_Java
- HQL http://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html
- Criteria com hibernate http://docs.jboss.org/hibernate/core/3.3/reference/en/html/querycriteria.html
- JPA http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html
- Hibernate Annotations http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html/entity.html
- Graphical User Interface http://download.oracle.com/javase/tutorial/ui/index.html