Java WEB: Criando uma tela de Login com JPA, JSF, PrimeFaces e MySQL

Extraido de Devmedia (<http://www.devmedia.com.br/java-web-criando-uma-tela-de-login-com-jpa-jsf-primefaces-e-mysql/32456>)

Esse artigo mostrará como criar uma tela de Login de uma aplicação web com JSF, PrimeFaces e JPA.

O Java Server Faces é uma especificação para o desenvolvimento de aplicações Web, seguindo o padrão Model View Controler (MVC) em Java, essa especificação surgiu como uma alternativa ao Struts, que na época era o principal framework para implementar aplicações nesse padrão de projeto. Sua primeira versão foi disponibilizada em 2004, e em 2013 foi lançada a versão 2.2.

Atualmente, existem diversos frameworks para construção de interfaces ricas para o JSF, como o RichFaces, o IceFaces, e principalmente o PrimeFaces, que se destaca por disponibilizar uma grande variedade de componentes. Atualmente esse framework está na versão 5.1. e contém um grande número de componentes de formulários, listagem, menus, entre outros, que facilitam muito o desenvolvimento de interfaces.

O JPA é uma especificação para a persistência de dados em Java utilizando o Mapeamento Objeto-Relacional (ORM – Object Relational Mapping). Uma das principais implementações do JPA é a do framework Hibernate. JPA pode ser utilizado com qualquer banco de dados que tenha um driver para a comunicação com aplicações Java, entre eles, o PostgreSQL, o Oracle, o SQL Server e o MySQL.

Este artigo mostrara como configurar uma aplicação web com o PrimeFaces e o Hibernate, e também como implementar uma tela de login com essas tecnologias. O servidor de banco de dados utilizado no desenvolvimento da aplicação será o MySQL.

**Configurando o projeto**

Para facilitar a configuração do projeto, será utilizado o Maven. Para a criação desse projeto serão necessárias as dependências das bibliotecas do PrimeFaces, do JSF, do Hibenate e do driver do MySQL. Para a biblioteca do PrimeFaces também é necessário adicionar as informações do repositório desse framework. A **Listagem 1**mostra o arquivo pom.xml do projeto criado.

**Listagem 1**. Configurando o projeto com o PrimeFaces e JPA

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.devmedia</groupId>

<artifactId>primefaces</artifactId>

<version>0.0.1</version>

<packaging>war</packaging>

<dependencies>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>4.2.0.Final</version>

</dependency>

<dependency>

<groupId>org.hibernate.common</groupId>

<artifactId>hibernate-commons-annotations</artifactId>

<version>4.0.1.Final</version>

<classifier>tests</classifier>

</dependency>

<dependency>

<groupId>org.hibernate.javax.persistence</groupId>

<artifactId>hibernate-jpa-2.0-api</artifactId>

<version>1.0.1.Final</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>4.0.1.Final</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.6</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>org.primefaces</groupId>

<artifactId>primefaces</artifactId>

<version>5.1</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>org.glassfish</groupId>

<artifactId>javax.faces</artifactId>

<version>2.1.13</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.4</version>

</dependency>

</dependencies>

<repositories>

<repository>

<id>prime-repo</id>

<name>PrimeFaces Maven Repository</name>

<url>http://repository.primefaces.org</url>

<layout>default</layout>

</repository>

</repositories>

</project>

Depois de configurar as dependências do projeto com o Maven, é necessário configurar a aplicação para usar o framework JSF, como o projeto é de uma aplicação Web, necessitamos criar o arquivo web.xml. O passo mais importante, é configurar o Faces Servlet, que utiliza a classe *javax.faces.webapp.FacesServlet*, e configurar o mapeamento das URL’s que serão tratadas pelo JSF, no caso, todos os endereços que terminarem com o padrão \*.xhtml. Também foi configurado que caso acontece algum erro na aplicação, a requisição seja redirecionada para a página inicial da aplicação. A **Listagem 2**mostra o código do arquivo web.xml.

**Listagem 2**. Configuração do arquivo web.xml do projeto.

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd">

<display-name>com.devmedia.primefaces</display-name>

<servlet>

<servlet-name>Faces Servlet</servlet-name>

<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>Faces Servlet</servlet-name>

<url-pattern>\*.xhtml</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>/index.xhtml</welcome-file>

</welcome-file-list>

<error-page>

<exception-type>javax.faces.application.ViewExpiredException</exception-type>

<location>/index.xhtml</location>

</error-page>

</web-app>

Também é necessário configurar o arquivo persistence.xml, que é o arquivo que configura o acesso ao banco de dados com JPA. A **Listagem 3** mostra esse arquivo. Algumas configurações importantes desse arquivo são, na tag <persistence-unit> deve ser definido o nome da unidade de persistência, na tag <provider> deve ser definido a implementação utilizada do JPA, no caso o Hibernate, a tag <class> define as classes dos objetos que serão persistidos, nesse exemplo, apenas a classes Usuario será usada. As outras propriedades definem as opções de acesso ao MySQL. A propriedade **hibernate.hbm2ddl.**auto define que o banco de dados será criado automaticamente caso ele ainda não exista, o que diminui um pouco o trabalho de configuração.

**Listagem 3**. Arquivo persistence.xml que configura o acesso ao banco de dados com JPA.

<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="usuarios">

<!-- provedor/implementacao do JPA -->

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

<!-- entidade mapeada -->

<class>com.devmedia.model.Usuario</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/user" />

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

<property name="javax.persistence.jdbc.password" value="eduardo73" />

<!-- propriedades do hibernate -->

<property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5InnoDBDialect" />

<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>

Para projetos configurados com o maven, O arquivo persistence.xml deve obrigatoriamente ficar no diretório /src/main/resources/META-INF do projeto.

Para executar os exemplos criados neste artigo será necessário utilizar um container Web. O PrimeFaces e a JPA podem ser executados em qualquer um que seja compatível com o Java Web, entre eles o Jetty e o Tomcat. No desenvolvimento do artigo foi utilizado o Apache Tomcat, versão 8.0.12.

**Desenvolvendo a tela de Login**

O primeiro passo para o desenvolvimento da tela de Login, é desenvolver a classe que implementa os objetos que serão persistidos no banco de dados. Como usaremos o JPA, é necessário utilizar as anotações dessa especificação, para que seja feito o mapeamento Objeto-Relacional dos atributos da classe.

A **Listagem 4** mostra o código da classe Usuario, que é o objeto que será persistido no banco de dados. A classe tem quatro atributos, que são o Id, o nome de usuário, a senha do usuário e a data do último acesso ao sistema. As principais anotações são @Id, que define que o atributo é o identificador único da classe, a anotação @Column que define diversas opções para a coluna, como o nome da coluna no banco de dados, se ela é uma coluna com valores únicos e se ela pode receber valores nulos e a anotação @Temporal que deve ser utilizada para atributos com o tipo data.

Os campos id e nomeUsuario são únicos, por isso o atributo *unique*nesses dois atributos tem o valor *true*, a senha e a data de ultimo acessa não precisam ser únicos, já que dois usuários diferentes podem acessar a aplicação ao mesmo tempo, e também podem ter senhas iguais. Todos os dados do usuário são obrigatórios, menos o ultimoAcesso, os dados obrigatórios devem ter o atributo *nullable* como *false*, e os não obrigatórios *true*.

**Listagem 4**. Classe Usuario que representa o objeto que será persistido

package com.devmedia.model;

import java.util.Date;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Temporal;

import javax.persistence.TemporalType;

@Entity

public class Usuario {

@Id

@Column(name="id", nullable=false, unique=true)

private int id;

@Column(name="userName", nullable=false, unique=true)

private String nomeUsuario;

@Column(name="password", nullable=false, unique=false)

private String senha;

@Column(name="lastAccess", unique=true)

@Temporal(TemporalType.DATE)

private Date ultimoAcesso;

public String getNomeUsuario() {

return nomeUsuario;

}

public void setNomeUsuario(String nomeUsuario) {

this.nomeUsuario = nomeUsuario;

}

public String getSenha() {

return senha;

}

public void setSenha(String senha) {

this.senha = senha;

}

public Date getUltimoAcesso() {

return ultimoAcesso;

}

public void setUltimoAcesso(Date ultimoAcesso) {

this.ultimoAcesso = ultimoAcesso;

}

}

Depois de criar a classe do objeto que será persistido, podemos definir a classe que faz as operações no banco de dados, como incluir, excluir e recuperar dados. Como vamos implementar apenas o login, precisaremos apenas do método que recupera os dados do usuário, caso ele exista. A **Listagem 5** mostra o código da classe UsuarioDAO que tem o método *getUsuario* recebendo os parâmetros nomeUsuario e senha, nesse método é feita uma busca no banco, e caso esse usuário exista, ele é retornado, caso contrário, é retornado *null*. Quando uma *query*não retorna resultados, é lançada uma exceção do tipo NoResultException, por isso foi utilizado um try/catch, e caso essa exceção seja lançada, o método retorna null, indicando que não existe um usuário com o nome de usuário e senha passados como parâmetro.

**Listagem 5**. Classe UsuarioDAO que faz o acesso aos dados

package com.devmedia.db;

import javax.persistence.EntityManager;

import javax.persistence.EntityManagerFactory;

import javax.persistence.NoResultException;

import javax.persistence.Persistence;

import com.devmedia.model.Usuario;

public class UsuarioDAO {

private EntityManagerFactory factory = Persistence

.createEntityManagerFactory("usuarios");

private EntityManager em = factory.createEntityManager();

public Usuario getUsuario(String nomeUsuario, String senha) {

try {

Usuario usuario = (Usuario) em

.createQuery(

"SELECT u from Usuario u where u.nomeUsuario = :name and u.senha = :senha")

.setParameter("name", nomeUsuario)

.setParameter("senha", senha).getSingleResult();

return usuario;

} catch (NoResultException e) {

return null;

}

}

public boolean inserirUsuario(Usuario usuario) {

try {

em.persist(usuario);

return true;

} catch (Exception e) {

e.printStackTrace();

return false;

}

}

public boolean deletarUsuario(Usuario usuario) {

try {

em.remove(usuario);

return true;

} catch (Exception e) {

e.printStackTrace();

return false;

}

}

}

Para o exemplo desenvolvido no artigo, os métodos inserir e excluir não serão utilizados, mas caso seja necessário utilizados, basta utiliza-los como o método getUsuario é chamado.

Com o acesso aos dados já implementado, é possível desenvolver o ManagedBean do JSF, que é a classe que recebe os dados que o usuário envia pela tela da aplicação, para receber os dados, é utilizado o atributo usuário, com esses dados, o método *envia*chama o método*getUsuario*da classe*UsuarioDAO, caso esse método retorne null,*é enviada uma tela de erro para o usuário, e ele continua na mesma página, caso contrário, o usuário é redirecionado para a tela main.xhtml. A **Listagem 6** mostra o código dessa classe.

**Listagem 6**. Classe ManagedBean LoginManagedBean

package com.devmedia.managedbeans;

import javax.faces.application.FacesMessage;

import javax.faces.bean.ManagedBean;

import javax.faces.bean.ViewScoped;

import javax.faces.context.FacesContext;

import com.devmedia.db.UsuarioDAO;

import com.devmedia.model.Usuario;

@ManagedBean(name = "LoginMB")

@ViewScoped

public class LoginManagedBean {

private UsuarioDAO usuarioDAO = new UsuarioDAO();

private Usuario usuario = new Usuario();

public String envia() {

usuario = usuarioDAO.getUsuario(usuario.getNomeUsuario(), usuario.getSenha());

if (usuario == null) {

usuario = new Usuario();

FacesContext.getCurrentInstance().addMessage(

null,

new FacesMessage(FacesMessage.SEVERITY\_ERROR, "Usuário não encontrado!",

"Erro no Login!"));

return null;

} else {

return "/main";

}

}

public Usuario getUsuario() {

return usuario;

}

public void setUsuario(Usuario usuario) {

this.usuario = usuario;

}

}

Agora, é possível implementar a tela de login, a tela tem dois componentes de entrada, um <p:inputText>, para o usuário digitar o nome de usuário, e um <p:password>, que funciona como um inputText, mas mostra os dados com o \* para o usuário digitar sua senha. A **Listagem 7** mostra o código do HTML desenvolvido. O componente <p:messages> é exibido caso ocorra algum erro no login, como por exemplo, se o usuário não existir.

**Listagem 7**. XHTML da tela de Login

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:h="http://java.sun.com/jsf/html"

xmlns:f="http://java.sun.com/jsf/core"

xmlns:p="http://primefaces.org/ui">

<h:head>

</h:head>

<h:body>

<h:form>

<p:messages id="messages" />

<p:panelGrid columns="2">

<p:outputLabel for="nomeUsuario" value="Nome Usuário:" />

<p:inputText id="nomeUsuario" value="#{LoginMB.usuario.nomeUsuario}" />

<p:outputLabel for="senha" value="Senha:" />

<p:password id="senha" value="#{LoginMB.usuario.senha}" />

<p:commandButton value="Enviar" icon="ui-icon-star" action="#{LoginMB.envia}" ajax="false">

</p:commandButton>

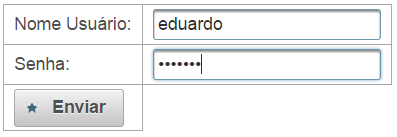
</p:panelGrid>

</h:form>

</h:body>

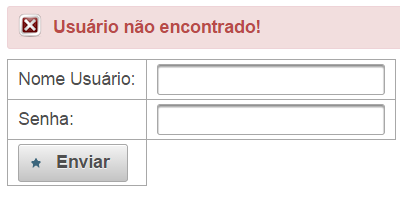
</html>

A **Figura 1** mostra a tela de login implementada, como é possível observar, existem os dois campos para o usuário digitar seu nome de usuário e a sua senha. Depois, ele pode clicar no botão Enviar, caso o login falhe, será exibida uma mensagem de erro, caso contrário, ele é redirecionado para a tela principal do sistema.



**Figura 1**. Tela de login implementada.

A **Figura 2** mostra a tela de login no caso de um usuário ter digitado ou o nome de usuário ou a senha incorretamente, caso isso ocorra, será mostrada uma mensagem de erro.



**Figura 2**. Tela de login com mensagem de erro.

A **Listagem 8** mostra o código do XHTML da página de entrada do sistema, como o objetivo desse artigo era apenas a tela de login, essa tela exibe apenas uma mensagem dizendo “LOgin efetuado com sucesso! Bem-vindo ao sistema!”, mas seria possível implementar qualquer coisa nessa página, e o usuário só a conseguiria acessa-la via a tela de login.

**Listagem 8**. XHTML da tela principal do sistema para caso de sucesso no login.

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:h="http://java.sun.com/jsf/html"

xmlns:f="http://java.sun.com/jsf/core"

xmlns:p="http://primefaces.org/ui">

<h:head>

</h:head>

<h:body>

<h:form>

<p:outputLabel value="Login efetuado com sucesso! Bem vindo ao sistema!"></p:outputLabel>

</h:form>

</h:body>

</html>

Este artigo mostrou como configurar e implementar uma tela de login combinando os frameworks PrimeFaces, o JSF e Hibernate. A configuração feita nesse artigo serve para a criação de qualquer aplicação Web com essas tecnologias. Outro requisito que seria interessante implementar é o controle de acesso as páginas, não permitindo que usuários não logados acessem algumas páginas, mas isso é assunto para outro artigo.

Espero que este artigo seja útil. Até a próxima!

**Links**

**ShowCase do PrimeFaces**http://www.primefaces.org/showcase/

**Página oficial do Hibernate**http://hibernate.org/

**Página da oracle sobre o JSF**http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html