03

Autenticação do usuário

Começando daqui? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/JSF/livraria-capitulo11.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

No último capítulo, vimos algumas novidades do JSF 2.2, as *viewActions* e *pass through atributes*. Neste capítulo, falaremos sobre a **autenticação do usuário**.

Criando a página de login

Autenticação do usuário significa, basicamente, que é preciso fazer um login. Se nós precisamos fazer login, nós precisamos de um formulário para isso, isso quer dizer que precisamos criar mais um `xhtml`, o `login.xhtml`. Nesse `xhtml` devemos ter dois `input`s, um para o e-mail do usuário e outro para a senha:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

    <ui:composition template="_template.xhtml">

        <ui:define name="titulo">
                Login
        </ui:define>

        <ui:define name="conteudo">
            <h:form id="login">
                <fieldset>
                    <legend>Login</legend>
                    <h:panelGrid columns="3">

                        <h:outputLabel value="Email:" for="email" />
                        <h:inputText id="email" value="#{loginBean.usuario.email}"
                            required="true">
                            <f:passThroughAttribute name="type" value="email" />
                        </h:inputText>

                        <h:message for="email" id="messageEmail" />

                        <h:outputLabel value="Senha:" for="senha" />
                        <h:inputText id="senha" value="#{loginBean.usuario.senha}"
                            required="true">
                            <f:passThroughAttribute name="type" value="password" />
                        </h:inputText>

                        <h:message for="senha" id="messageSenha" />

                        <h:commandButton value="Efetuar Login"
                            action="#{loginBean.efetuaLogin}" />
```

```
                    </h:panelGrid>
                </fieldset>
            </h:form>
        </ui:define>
    </ui:composition>
</html>
```

Repare que, para esse formulário funcionar, precisamos de um modelo, o `Usuario` e de um novo *bean*, o `LoginBean`.

O modelo Usuario

O modelo, a classe `Usuario` será muito semelhante à classe `Autor`. A diferença é que, ao invés de nome, um usuário terá um e-mail:

```
@Entity
public class Usuario implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue
    private Integer id;
    private String email;
    private String senha;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getSenha() {
        return senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }

}
```

Criado o modelo, é preciso mapeá-lo no arquivo `persistence.xhtml`, abaixo do mapeamento da classe `Autor`:

```
<class>br.com.caelum.livraria.modelo.Usuario</class>
```

O intermediário do nosso modelo com a tela, o LoginBean

O último passo agora é criar o `LoginBean`, para fazer a intermediação do modelo `Usuario` com a tela `login.xhtml`. Essa classe terá o usuário que vêm do formulário e o seu *getter*:

```
@ManagedBean
@ViewScoped
public class LoginBean {

    private Usuario usuario = new Usuario();

    public Usuario getUsuario() {
        return usuario;
    }
}
```

Mas ainda falta o método que é chamado no botão "Efetuar Login", o método `efetuaLogin`:

```
public void efetuaLogin() {
    System.out.println("Fazendo login do usuário "
            + this.usuario.getEmail());
}
```

Por enquanto o método só imprime o e-mail do usuário que está se logando, mas após esse login, o que deve acontecer? Devemos verificar se o usuário e a senha existem, e se eles existirem, o usuário deve ser redirecionado para a página principal da nossa aplicação, a página `livro.xhtml`. Então dentro deste método, vamos fazer um redirecionamento para a página `livro.xhtml`, e isso não é novidade, nós já fizemos isso antes, quando um autor é cadastrado.

Logo após de cadastrar um autor, o usuário é redirecionado para a página `livro.xhtml`, então vamos fazer a mesma coisa que foi feita anteriormente:

```
public String efetuaLogin() {
    System.out.println("Fazendo login do usuário "
            + this.usuario.getEmail());

    return "livro?faces-redirect=true";
}
```

Com o formulário, modelo e *bean* criados, vamos testar a nossa aplicação. Repare que após clicarmos no botão "Efetuar Login", somos redirecionados para `livro.xhtml`!

Mas agora temos que fazer essa validação, temos que verificar se o usuário existe no banco, para aí sim ele ser redirecionado.

Verificando a existência no usuário

Se iremos verificar se o usuário existe no banco de dados, a primeira coisa que devemos fazer é ter um usuário no banco de dados! Então iremos abrir o **MySQL** e iremos fazer um `insert` para adicionar um usuário válido. Por exemplo:

```
mysql> use livrariadb;
mysql> insert into Usuario(email, senha) values ('nico.steppat@caelum.com.br', '12345');
Query OK, 1 row affected (0.05 sec)

mysql> select * from Usuario;
+----+---------------------------+-------+
| id | email                     | senha |
+----+---------------------------+-------+
|  1 | nico.steppat@caelum.com.br | 12345 |
+----+---------------------------+-------+
1 row in set (0.00 sec)
```

Agora que temos um usuário criado no banco, podemos testá-lo. Isso significa que se o usuário existe, ele é redirecionado para a página `livro.xhtml` , se não existe, não acontece nada, o método retorna `null` :

```
public String efetuaLogin() {
    System.out.println("Fazendo login do usuário "
            + this.usuario.getEmail());

    if (existe) { // Mas o que significa "existe"?
        return "livro?faces-redirect=true";
    }

    return null;
}
```

O `existe` terá que ser um booleano, ele guardará o retorno do método `existe` , que recebe o usuário do formulário como parâmetro e faz um `select` no banco de dados para verificar se esse usuário existe ou não. Mas vamos separar essa responsabilidade na classe `UsuarioDao` que iremos criar:

```
public String efetuaLogin() {
    System.out.println("Fazendo login do usuário "
            + this.usuario.getEmail());

    boolean existe = new UsuarioDao().existe(this.usuario);

    if (existe) {
        return "livro?faces-redirect=true";
    }

    return null;
}
```

Com o método `efetuaLogin` pronto, falta implementar a classe `UsuarioDao` , juntamente com o método `existe` . Esse método inicializará o `EntityManager` e criará uma **query tipada**(*typed query*), do tipo `Usuario` , pois a query retornará o usuário do banco de dados onde seu e-mail e senha for igual ao e-mail e senha do usuário passado por parâmetro.

```
public class UsuarioDao {

    public boolean existe(Usuario usuario) {

        EntityManager em = new JPAUtil().getEntityManager();
        TypedQuery<Usuario> query = em
                .createQuery(
                        "select u from Usuario u where u.email = :pEmail and u.senha = :pSenha",
                        Usuario.class);

        return false;
    }
}
```

Com a query criada, precisamos configurar os parâmetros `pEmail` e `pSenha` da query, que são o e-mail e senha do usuário passado por parâmetro, faremos isso através do método `setParameter` e depois iremos executar a query, através do método `getSingleResult`, pois a nossa query só retornará um único resultado, caso retorne algo. Para finalizar, o método retornará `true` se o resultado for diferente de `null`:

```
public boolean existe(Usuario usuario) {

    EntityManager em = new JPAUtil().getEntityManager();
    TypedQuery<Usuario> query = em
            .createQuery(
                    "select u from Usuario u where u.email = :pEmail and u.senha = :pSenha",
                    Usuario.class);

    query.setParameter("pEmail", usuario.getEmail());
    query.setParameter("pSenha", usuario.getSenha());

    Usuario resultado = query.getSingleResult();

    em.close();

    return resultado != null;
}
```

Testando com um login cadastrado no banco, vemos que está funcionando! Vamos agora testar com um login e senha que não existe... Deu uma exceção, a `NoResultException`.

Para resolver isso iremos realizar um `try...catch` na exceção. Caso essa exceção ocorra, ou seja, caso a query não retorne nenhum resultado, o método irá retornar `false`, para nada acontecer. Se não houver exceção, o método retorna `true`:

```
public boolean existe(Usuario usuario) {

    EntityManager em = new JPAUtil().getEntityManager();
    TypedQuery<Usuario> query = em
            .createQuery(
                    "select u from Usuario u where u.email = :pEmail and u.senha = :pSenha",
                    Usuario.class);
```

```
        query.setParameter("pEmail", usuario.getEmail());
        query.setParameter("pSenha", usuario.getSenha());

        try {
            Usuario resultado = query.getSingleResult();
        } catch (NoResultException ex) {
            return false;
        }

        em.close();

        return true;
    }
```

Podemos testar agora. Com um usuário válido, continuamos sendo redirecionados para a página `livro.xhtml` e com um usuário inválido, nada acontece! Ótimo, a autenticação do usuário está funcionando corretamente. Mas ainda nada impede que acessemos a página `livro.xhtml` sem realizarmos login, através da url `http://localhost:8080/livraria/livro.xhtml` . Então o próximo passo que devemos implementar é a **autorização do usuário**, ou seja, só devemos conseguir acessar as páginas com o login realizado, e é isso que veremos no próximo capítulo!

O que aprendemos nesse capítulo:

- Criamos um novo modelo (`Usuário`) e uma nova Bean(`LoginBean`);
- TypedQuery para usuário;
- Lidamos com erro de `NoResultException` ;