Disciplina: Desenvolvimento de Software

Aula 6: Java para Web

Apresentação

O ambiente Java Web é considerado de grande robustez, sendo utilizado por grandes empresas, principalmente em sistemas de missão crítica. Nesse ambiente temos um Web Server padrão, denominado Tomcat, o qual responde ao protocolo HTTP e fornece um container para execução de Servlets e JSPs.

Apesar de ser o suficiente para aplicações simples, em ambientes corporativos devemos adotar componentes do tipo Enterprise Java Beans, providos por Application Servers, como GlassFish. A utilização de frameworks, como JSF, é um fator que trará grande produtividade para o programador.

Objetivos

- Identificar as características do Java para Web;
- Descrever as tecnologias Servlet, JSP e JSF;
- Empregar operações básicas do framework JSF.

Ambiente servidor

Quando enviamos os dados para alguma tecnologia servidora, temos uma **requisição** HTTP, a qual irá iniciar algum processamento ao nível do servidor e que, ao final, retornará uma **resposta** HTTP, normalmente com conteúdo **HTML** ou **XML**.

Utilizaremos, no ambiente Java, um objeto da classe **HttpServletRequest**, normalmente denominado request, para capturar os dados enviados pela requisição HTTP, efetuando qualquer tipo de processamento com esses dados, como a inserção em banco de dados ou validações de segurança.

Os dados de retorno para o usuário são controlados por um objeto da classe **HttpServletResponse**, que costuma ser denominado response, o qual irá permitir escrever o conteúdo HTML ou XML de saída, além de outros elementos, como cookies.

Algo que temos de ter em mente, com relação ao modelo Web, é a grande diferença do modelo desktop na gestão de estado de objetos.

No **modelo desktop** existe um programa que guarda os objetos em um espaço local, permitindo que tais objetos existam durante toda a execução, mesmo que ocorra a troca de janelas, o que viabiliza a manutenção do estado dos objetos.

No **modelo Web**, a cada requisição e resposta, todos os objetos antigos deixam de existir, e o estado não pode ser mantido.

A solução para manter esses estados é a utilização de sessões, as quais correspondem a objetos alocados ao nível do servidor, fazendo referência a uma dada conexão. Enquanto o usuário se mantiver no site, esses dados serão mantidos, sendo eliminados na perda de conexão.

Atenção

No Java, o objeto session, da classe HttpSession, permite a gerência de sessões HTTP, mas é sempre importante lembrar que as sessões consomem memória do servidor, devendo ser utilizadas para fins bem específicos.

Tomcat e GlassFish

O servidor Tomcat é um projeto da Apache Software Foundation para a implementação com código aberto de tecnologias como Java Server Pages (JSP), Java Servlet, Java WebSocket e Java Expression Language.

Esse conjunto de tecnologias oferecerá um ambiente consistente para a resposta às chamadas HTTP, além de prover suporte a Servlets e JSPs.

Por ser de código aberto, acabou se tornando o padrão para hospedagem de aplicativos Java para Web, podendo atuar como servidor independente ou como módulo de servidores, como JBoss e GlassFish.

Os principais diretórios e arquivos do Tomcat podem ser observados no quadro a seguir.

Diretório	Conteúdo
bin	Binários do servidor, incluindo o executável do mesmo.
conf	Arquivos de configuração, como server.xml, que guarda as configurações gerais do Tomcat.
lib	Bibliotecas Java de inicialização do servidor no formato jar, as quais também ficam disponíveis para todos os aplicativos do ambiente.
logs	Arquivos de log para identificação de erros na execução do Tomcat.
webapps	Aplicativos Java para Web.

É possível alterar várias características do Tomcat, editando o arquivo server.xml.

Atenção

O servidor Tomcat executa, por padrão, na porta 8080, mas podemos modificá-la procurando e alterando a ocorrência desse valor no arquivo.

Os aplicativos que criaremos deverão obedecer à estrutura exigida pelo Tomcat, composta por:

- um diretório de base, com páginas JSP, HTML, XML, e outros formatos que não são compilados;
- um subdiretório WEB-INF, onde fica o arquivo de configuração web.xml;
- um diretório classes, para o armazenamento das classes Java;
- um diretório lib, com as bibliotecas nos formatos jar e zip;
- um subdiretório denominado META-INF, onde podemos encontrar o arquivo context.xml, com informações gerais de contexto, como pool de conexões com banco, por exemplo.

Toda essa estrutura pode ser compactada em um arquivo com extensão war (Web Archived), que ao ser copiado para o diretório webapps, ou deploy, de acordo com a distribuição, é automaticamente expandido, ficando o aplicativo disponível para os usuários.

Essa é uma característica do Tomcat, denominada hot deployment, fornecendo uma forma muito prática de disponibilizar o aplicativo Web, pois resume o upload de dezenas de arquivos a apenas um.

Enquanto o Tomcat suporta, de forma nativa, apenas Servlets e JSPs, atuando como um Web Server, o GlassFish vai além, oferecendo suporte às tecnologias Java de Objetos Distribuídos, no caso os Enterprise Java Beans (EJBs), sendo classificado como Application Server.

Atenção

Algo interessante a ser mencionado é que o Tomcat é utilizado pelo GlassFish como módulo interno, delegando a ele toda a parte de comunicação HTTP e o tratamento de Servlets e JSPs, enquanto os demais elementos da robusta arquitetura do GlassFish tratam das diversas tecnologias do Java Enterprise Edition (JEE).

Com o uso do GlassFish, seremos capazes de criar sistemas mais complexos, com uso de EJBs e transações distribuídas, além de obtermos ferramentas para gerenciamento de componentes corporativos, como mensagerias, e ambiente de testes simplificado para Web Services.

Criação do aplicativo web

Uma grande vantagem do uso de NetBeans, na versão completa, é a de que já traz todo o suporte ao desenvolvimento de aplicativos para Web e com uso de Enterprise Java Beans, além de dar suporte a alguns frameworks e trazer os servidores Tomcat e GlassFish incorporados ao ambiente de desenvolvimento.

Exemplo 1

Para criar um projeto Web, devemos seguir os seguintes passos:

1. No menu principal do NetBeans escolha a opção **Arquivo..Novo Projeto**, ou **Ctrl+Shift+N**;
2. Na janela que se abrirá escolha o tipo de projeto como **Java Web..Aplicação Web** e clique em **Próximo**;
3. Dê um **nome** para o projeto (WebTeste01) e **diretório** para armazenar seus arquivos (C:\MeusTestes), e clique em **Próximo**;
4. Escolha o servidor **(GlassFish)** e versão do JEE **(Java EE7)** e clique em **Finalizar**.

Esta sequência de passos pode ser observada na seguinte sequência:

Ao término desses passos, o projeto gerado será apresentado no **Painel de Controle**, guia **Projetos**, como pode ser observado a seguir.

Precisamos observar que o arquivo **web.xml** não é apresentado nessa estrutura, e isso se deve ao fato de termos utilizado o **JEE versão 7**, onde as configurações são efetuadas, em sua grande maioria, com o uso de anotações.

O projeto é dividido em:

Páginas Web

Onde se encontram elementos JSP, HTML, CSS e outros.

Pacotes de Códigos-Fonte

Onde iremos colocar nossas classes e pacotes Java.

Bibliotecas

Para adicionar bibliotecas Java para uso pelo sistema.

Arquivos de Configuração

Incluindo elementos como MANIFEST e web.xml.

Criado o aplicativo, podemos começar a criar Servlets, JSPs e demais elementos constituintes de nosso sistema, e, ao clicar no botão de execução, será iniciado o servidor (caso ainda não esteja ativo), gerado o arquivo **war**, copiado esse arquivo para o diretório correto do servidor, efetuado o deploy e aberto o navegador no endereço correto para apresentação do **index**.

Servlets e JSPs

Conceitualmente, podemos dizer que Servlet é uma classe Java voltada para a criação de serviços remotos que estendem a funcionalidade de algum servidor específico.

A tecnologia Servlet foi criada com o intuito de se tornar uma solução genérica para a criação de aplicativos hospedados em servidores, quaisquer que fossem os protocolos utilizados, mas foi a especialização para o HTTP que se tornou popular.

A classe HttpServlet, descendente de Servlet, integra-se com o ambiente onde é executada, o qual é denominado container Web, tirando proveito de diversas tecnologias Java presentes ali, como a segurança com Realm e o uso de pools de conexão, e fornece métodos para resposta às chamadas GET e POST do HTTP.

Tudo que precisamos fazer é criar um descendente da classe **HttpServlet**, herdando toda a integração com o ambiente já existente, e alterar os métodos **doGet** e **doPost** para, através do polimorfismo, personalizar as respostas às necessidades de nosso aplicativo.

Para adicionar um Servlet ao nosso projeto, devemos escolher o menu **Arquivo..Novo Arquivo**, ou pressionar ***CTRL+N***, e seguir os seguintes passos na janela que se abrirá:

Exemplo 2

1. Escolha o tipo de arquivo como **Web..Servlet** e clique em **Próximo**;

2. Dê um **nome** para o Servlet (ServCalc) e para o **pacote** onde será gerado (servlets), e clique em **Próximo**;

3. Observe as configurações de mapeamento do Servlet e clique em **Finalizar**.

Esta sequência de passos pode ser observada na sequência a seguir:

Teremos o código do Servlet gerado no editor de código do NetBeans, e devemos observar alguns detalhes, como o código oculto (editor-fold), que trata de um trecho de código oculto através de comentários anteriores e posteriores, com a visualização alternada através do clique sobre o sinal de "+" presente na margem esquerda do editor.

Comentário

O código oculto, nesse caso, engloba os métodos doGet e doPost, responsáveis pela recepção de chamadas HTTP dos tipos **GET e POST**, respectivamente. No código gerado, ambos os métodos redirecionam para **processRequest**, o que fará com que GET e POST sejam tratados da mesma forma, algo que nem sempre é interessante, pois temos situações, como no processo de login, em que não devemos aceitar chamadas GET.

O próximo passo é alterar o método processRequest, para que o mesmo corresponda ao processamento necessário em nosso sistema:

```
protected void processRequest(HttpServletRequest request,
   HttpServletResponse response) throws ServletException,
   IOException {
     response.setContentType("text/html;charset=UTF-8");
     try (PrintWriter out = response.getWriter()) {
       int a = new Integer(request.getParameter("a"));
       int b = new Integer(request.getParameter("b"));
       out.println("‹html›‹body›");
       out.println("A soma de "+a+" e "+b+" será "+(a+b));
       out.println("›/body›‹/html›");
     }
}
```

A assinatura de processRequest, assim como em doGet e doPost, traz dois parâmetros:

O primeiro do tipo **HttpServletRequest**, encapsulando a requisição HTTP.

O segundo do tipo **HttpServletResponse**, responsável pela resposta HTTP.

Inicialmente é definido o tipo de saída que será utilizado, através do objeto response, e o fluxo da resposta é apontado pelo objeto out.

```
response.setContentType("text/html;charset=UTF-8");
try (PrintWriter out = response.getWriter()) {
```

Em seguida, capturamos os parâmetros enviados pela requisição HTTP, através do objeto request, os quais chegam no formato texto, e os transformamos para o tipo inteiro.

```
int a = new Integer(request.getParameter("a"));
int b = new Integer(request.getParameter("b"));
```

Finalmente, construímos a saída HTML desejada, nesse caso apenas exibindo a soma de dois números enviados a partir da chamada HTTP.

```
out.println("‹html›‹body›");
out.println("A soma de "+a+" e "+b+" será "+(a+b));
out.println("‹/body›‹/html›");
```

Observe também a presença de uma anotação na definição do Servlet.

```
@WebServlet(name = "ServCalc", urlPatterns = {"/ServCalc"})
public class ServCalc extends HttpServlet {
```

Essa anotação efetua o mapeamento do Servlet, indicando o nome utilizado pelo mesmo e a URL para invocá-lo.

Embora tradicionalmente sejam utilizados textos correspondentes ao nome da classe em si, não é uma regra, podendo inclusive ser adotadas URLs dinâmicas através de curingas, como “*.jsf”.

Com o nosso Servlet construído, podemos criar o formulário necessário para efetuar sua chamada, e faremos isso com uma pequena inclusão no trecho **‹body›** da página **index**.

Exemplo 4

```
<!DOCTYPE html>
   <html>
    <body>
     <form method="GET" action="ServCalc">
     <input type="text" name="a"/>
     <input type="text" name="b"/>
     <input type="submit" value="somar"/>
     </form>
   </body>
</html>
```

Finalizando todas essas modificações, precisamos apenas executar o projeto, sendo iniciado o servidor, efetuado o deploy e aberto o navegador na página index.

Preenchendo os dois números desejados e clicando em somar, o Servlet será acionado e a resposta montada, sendo exibida no navegador do cliente.

A criação de Servlets seria suficiente para prover as necessidades de todo e qualquer aplicativo Web, porém a construção de páginas através de código direto pode se tornar desconfortável para a maioria dos designers.

Uma solução para isso foi a definição de um novo modelo de programação, onde códigos Java são escritos dentro do conteúdo HTML ou XML, o que foi chamado de Java Server Pages (JSP).

O processo para a criação de uma página JSP é basicamente o mesmo que o de um Servlet, sendo que devemos escolher arquivo do tipo **Web..JSP**. O arquivo gerado ficará na seção **Páginas Web** do projeto, ou em um subdiretório.

Vamos observar a estrutura de uma página JSP simples.

Exemplo 5

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
   <html>
    <head>
     <meta http-equiv="Content-Type"
      content="text/html; charset=UTF-8">
     <title>JSP Page</title>
   </head>
   </body>
      <ul>
      <%
      <String[] cores = {"vermelho","verde","azul"};
      for(String x: cores) {
      for(String x: cores) {
   out.println("<li>"+x+"</li>");
    }
   %>
  </ul>
 </body>
</html>
```

A primeira linha desse código é uma diretiva, no caso indicando o tipo de conteúdo e a página de acentuação utilizada.

Diretivas também são utilizadas para importar bibliotecas, indicar herança, importar taglibs, definir a página de erro, entre diversas outras opções.

```
‹%@page contentType="text/html" pageEncoding="UTF-8"%›
```

As linhas seguintes são código HTML padrão, e que não serão executados ao nível do servidor, mas simplesmente irão compor a página de saída.

No meio das tags ‹ul› temos um Scriptlet, iniciado com ‹% e terminado com %›, tratando de código Java que será executado no Servidor.

Nesse código, temos a definição de um vetor de cores, as quais serão impressas no conteúdo da página através do objeto out, implícito nas páginas JSP.

```
‹%
    String[] cores = {"vermelho","verde","azul"};
    for(String x: cores) {
      out.println("‹li›"+x+"‹/li›");
    }
%>
```

Para executar diretamente o JSP, basta clicar com o botão direito sobre ele na guia de **Projetos** e mandar executar arquivo. Podemos ver a página gerada a seguir.

Os objetos **request** e **response** também são implícitos para as páginas JSP, podendo ser utilizados da mesma forma que o foram nos Servlets.

Atenção

Toda página JSP é convertida em Servlet pelo container Web no primeiro acesso. Logo, o que muda é basicamente a forma de programar, não a funcionalidade original.

Gerência de sessões

As sessões são de grande utilidade no ambiente Web, provendo uma forma de manutenção de estados na troca de páginas.

Ao contrário dos sistemas desktop, a cada nova página temos outro conjunto de variáveis na memória, desconsiderando-se todas aquelas existentes antes da requisição ser efetuada.

Podemos controlar sessões de forma muito simples, com o uso da classe **HttpSession**, e um exemplo típico de utilização é no controle de login.

Atenção

Nas páginas JSP, o controle de sessão é feito com o uso do objeto implícito session, da classe HttpSession.

Vamos, inicialmente, criar uma página JSP protegida, denominada “Segura.jsp”, como pode ser observado no código seguinte.

```
‹%@page contentType="text/html" pageEncoding="UTF-8"%›
‹%
   if(session.getAttribute("usuario")==null)
    response.sendRedirect("Login.jsp");
     else {
     %›
     ‹!DOCTYPE html›
   ‹html›
   ‹body›
      ‹h1›Esta é uma página protegida!‹/h1›
      &O usuário ‹%=session.getAttribute("usuario")%›
      está logado.‹br/›
      Para desconectar clique
      ‹a href="ServletLogin?acao=desconectar"›aqui‹/a›
 ‹/body›
‹/html›
‹% } %›
```

Enquanto os **parâmetros** da requisição assumem apenas valores do tipo texto, os **atributos** de uma sessão permitem guardar qualquer tipo de objeto, inclusive texto.

Na primeira parte do arquivo JSP, temos o teste para a existência do atributo “usuario” na sessão. Se não existir, isso significa que não foi efetuado o processo de login, devendo ocorrer o redirecionamento para a página de login através de **sendRedirect**.

```
if(session.getAttribute("usuario")==null)
response.sendRedirect("Login.jsp");
else {
```

Atenção

Há dois tipos de redirecionamento no ambiente Java Web:

- O modo **sendRedirect** envia um sinal de redirecionamento ao navegador, para que efetue uma nova requisição ao servidor;
- Através de **forward**, efetuamos um redirecionamento interno no servidor, e as informações da requisição original são mantidas no envio de um componente para outro.

Ainda observando o código, podemos notar que a instrução else é aberta antes do início do código **HTML** e fechada apenas no final, logo após a tag de finalização **‹/html›**.

Isso significa que todo esse bloco será processado apenas se ocorre o atributo usuário na sessão, ou seja, se existe alguém logado no sistema.

Existindo um usuário logado, o bloco é executado e a página de resposta é montada, contando inclusive com a identificação do login atual.

```
O usuário ‹%=session.getAttribute("usuario")%› está logado.‹br/›
```

Agora precisamos criar a página de login, denominada **“Login.jsp”**, e o Servlet responsável pelo controle das ações referentes aos processos de conexão e desconexão, o qual será chamado de **ServletLogin**.

Podemos observar o código da página de login a seguir.

Exemplo 7

```
‹%@page contentType="text/html" pageEncoding="UTF-8"%›
‹!DOCTYPE html›
   ‹html›
    ‹body›
     ‹h1›Acesso ao Sistema‹/h1›
     ‹form action="ServletLogin" method="post"›
     ‹input type="hidden" name="acao" value="conectar"/›
   Login: ‹input type="text" name="login"/›
   Senha: ‹input type="password" name="senha"/›
   Senha: ‹input type="submit" value="login"/›
      ‹/form›
      ‹%
      if(request.getAttribute("erro")!=null) {
      %›
      ‹hr/›Ocorreu um erro:‹%=request.getAttribute("erro")%›
     ‹%
        }
     %›
 ‹/body›
‹/html›
```

Na primeira parte desse JSP temos um formulário HTML bastante simples, contendo as informações que deverão ser enviadas ao Servlet para verificação.

Na segunda parte, apresentamos mensagens de erro enviadas pelo Servlet, como “Dados inválidos”.

Atenção

Esse trecho será apresentado apenas se o atributo de “erro” estiver presente na chamada ao JSP.

```
  ‹%
     if(request.getAttribute("erro")!=null) {
 %›
    ‹hr/›Ocorreu um erro: ‹%=request.getAttribute("erro")%›
‹%
}
%›
```

Agora precisamos criar o ServletLogin e implementar o código apresentado a seguir.

```
@WebServlet(name = "ServletLogin",
urlPatterns = {"/ServletLogin"}) public class ServletLogin extends HttpServlet {

@Override
protected void doPost(HttpServletRequest request,
     HttpServletResponse response)
     throws ServletException, IOException
     {
     String acao = request.getParameter("acao");
     if(acao==null)
       throw new ServletException("Parâmetro Requerido");
     HttpSession session = request.getSession();
     switch(acao){
     case "conectar":
     if(request.getParameter("login").equals("admin")&&
          request.getParameter("senha").equals("123")){
          session.setAttribute("usuario", "Administrador");
          response.sendRedirect("index.html");
     } else {
          request.setAttribute("erro","Dados inválidos.");
          RequestDispatcher rd =
               request.getRequestDispatcher("Login.jsp");
          rd.forward(request,response);
      }
      break;
  case "desconectar":
      session.invalidate();
      response.sendRedirect("index.html");
      break;
  default:
      throw new ServletException("Parâmetro incorreto");
     }
   }
}
```

Atenção

Por se tratar de um processo de login, apenas o método doPost poderá ser utilizado, e o parâmetro acao indicando solicitação de conexão ou desconexão é obrigatório, gerando exceção caso não seja fornecido.

```
if(acao==null)
   throw new ServletException("Parâmetro Requerido");
```

Para responder à ação **“conectar”**, um teste é realizado, e apenas o login “admin” e senha “123” em conjunto permitirão o login, com acréscimo à sessão do atributo **“usuario”**, com valor “Administrador” e redirecionamento para a página index.

```
if(request.getParameter("login").equals("admin")&&
  request.getParameter("senha").equals("123")){
    session.setAttribute("usuario", "Administrador");
    response.sendRedirect("index.html");
}
```

Caso não sejam fornecidos os dados com os valores corretos, a página de login é acionada, passando o atributo de erro para a mesma através de um atributo de requisição.

```
request.setAttribute("erro","Dados inválidos.");
RequestDispatcher rd = request.getRequestDispatcher("Login.jsp");
rd.forward(request,response);
```

Para a desconexão, basta chamar o método invalidate de session e redirecionar para o index. Todos os atributos associados à conexão do usuário são removidos da memória do servidor.

Agora basta adicionar uma chamada à página **“Segura.jsp”** em **index.html**.

```
‹a href="Segura.jsp" ›Página Segura ‹/a›
```

Podemos observar, nas figuras seguintes, a tentativa de acesso a partir de index, redirecionando para a tela de login, inicialmente passando dados incorretos para o Servlet e depois passando os dados corretos, o que liberará a página protegida na segunda tentativa de acesso.

Exemplo 9

Embora seja apenas um exemplo simples de processo de autenticação de usuário, com valores pré-fixados, você pode alterá-lo facilmente para a utilização de uma base de dados com senhas criptografadas.

Java Server Faces

Um framework é um conjunto de ferramentas que busca solucionar problemas comuns de determinada área do desenvolvimento de sistemas, o que parece a funcionalidade de uma biblioteca comum.

Os frameworks determinam o fluxo de execução que será utilizado, o que é chamado de inversão de controle.

Existem diversos frameworks para o ambiente Java Web, como Hibernate, Struts, Spring, e Java Server Faces (JSF), cada um com uma finalidade específica.

O framework JSF permite a utilização de uma metodologia orientada a eventos, como nos sistemas desktop, ao mesmo tempo que garante uma arquitetura robusta, no modelo MVC, com padrão Front Control.

Para acrescentar o JSF ao projeto Web, basta que você selecione a opção no momento da criação, logo após escolher o servidor, clicando em “Próximo”, no lugar de “Finalizar”, o que levará para a janela apresenta a seguir.

Exemplo 10

Com a utilização de JSF, teremos a substituição da sintaxe JSP por uma sintaxe XHTML denominada facelets para a criação de templates. Nas versões mais antigas do JSF, os templates eram criados com o uso de JSP.

Podemos observar o primeiro uso de facelets na página **“index.xhtml”**, gerada na criação do projeto e replicada a seguir.

Exemplo 11

```
‹?xml version='1.0' encoding='UTF-8' ?›
‹!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "//www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"›
‹html xmlns="//www.w3.org/1999/xhtml"
xmlns:h="//xmlns.jcp.org/jsf/html"›
   ‹h:head›
   ‹title›Facelet Title‹/title›
  ‹/h:head›
  ‹h:body›
 Hello from Facelets
 ‹/h:body›
‹/html›
```

O primeiro passo para a utilização de facelets é a definição de um namespace apontando para o endereço da biblioteca de suporte aos prefixos com finalidades específicas — no caso a construção dos elementos HTML.

```
xmlns:h="//xmlns.jcp.org/jsf/html"
```

Em seguida, podemos utilizar as instruções dessa biblioteca para a criação de elementos HTML na página, como o uso de ‹h:head› e ‹h:body›.

Podemos observar outros prefixos na tabela seguinte, cada qual configurando seu próprio conjunto de comandos.

Prefixo	URI	Exemplos
h	//xmlns.jcp.org/jsf/html	h:inputText
f	//xmlns.jcp.org/jsf/core	f:facet f:actionListener
c	//xmlns.jcp.org/jsp/jstl/core	c:forEach c:if
fn	//xmlns.jcp.org/jsp/jstl/functions	fn:toUpperCase
ui	//xmlns.jcp.org/jsf/facelets	ui:component

O que existe de especial nesse framework, além da utilização de diversos prefixos facilitadores para a construção de páginas, é a integração com o Java através do uso de componentes que são chamados de **Managed Beans**.

Para acrescentar um componente desse tipo basta acrescentar um novo arquivo ao projeto e seguir os seguintes passos na janela que se abrirá, conforme exemplo:

1 - Selecionar o tipo **Java Server Faces..Bean Gerenciado**
JSF.

2 - Preencher o nome da classe e do pacote, escopo **(session)**
e nome do bean.

Exemplo 12

Será criada uma classe denominada Cores, a qual iremos alterar para o código seguinte.

```
@Named(value = "cores")
@SessionScoped
public class Cores implements Serializable {
private final List‹String› lista = new ArrayList‹›();
private String atual;
public Cores(){ }
public void addCorAtual(){
    lista.add(atual);
    atual = "";
  } public List‹String› getListaCores(){
    return lista;
  }
public void setAtual(String atual){
    this.atual = atual;
  }
public String getAtual(){
    return atual;
  }
}
```

A classe criada é definida pelas duas anotações como Managed Bean, bem como o uso do escopo de sessão.

```
@Named(value = "cores")
@SessionScoped
```

Um escopo define o tempo de vida de uma variável. Além do escopo de sessão, onde os valores são mantidos entre chamadas sucessivas, existem escopos como View e Request, o primeiro restrito à página em si, o segundo para toda a requisição corrente. Os métodos públicos poderão ser utilizados a partir dos facelets, como podemos observar na modificação efetuada na página index.

```
‹?xml version='1.0' encoding='UTF-8' ?›
‹!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "//www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"›
‹html xmlns="//www.w3.org/1999/xhtml"
   xmlns:h="//xmlns.jcp.org/jsf/html"›
‹h:head›
   ‹title›Facelet Title‹/title›
‹/h:head›
‹h:body›
     ‹h:form›
      ‹h:inputText value="#{cores.atual}"/›
    ‹h:commandButton action="#{cores.addCorAtual()}"
        value="Adicionar"/›
    ‹/h:form›
      ‹h:dataTable value="#{cores.listaCores}" var="c"›
     ‹h:column›#{c}‹/h:column›
‹/h:dataTable›
‹/h:body›
‹/html›
```

Agora temos um componente do tipo DataTable, responsável por exibir toda a lista de cores existente no bean. O parâmetro value deve ser associado à lista, e o parâmetro var define um nome para o objeto que irá percorrer a lista e fornecer valores para as colunas.

```
‹h:dataTable value="#{cores.listaCores}" var="c"›
‹h:column›#{c}‹/h:column›
‹/h:dataTable›
```

Além disso, temos um componente do tipo InputText relacionado ao campo atual do bean, e o que for digitado nesse componente irá alterar o valor do atributo desse bean, pois foi utilizado o escopo de sessão.

Também podemos observar um CommandButton fazendo chamada ao método addCorAtual do bean, o que fará adicionar o valor digitado no InputText à lista interna do bean, sendo refletido automaticamente pelo DataTable.

```
‹h:inputText value="#{cores.atual}"/›
‹h:commandButton action="#{cores.addCorAtual()}"
value="Adicionar"/›
```

Note como essa programação acaba fornecendo um meio de trabalhar no modelo orientado a eventos, tipicamente utilizado nos sistemas desktop.

Como resultado final teremos a janela que pode ser observada a seguir.

Exemplo 13

Com o uso de JSF, simplificamos muito a forma de lidar com a interface, e esse framework serve de base para a construção de outros bem mais impactantes em termos visuais, como Prime Faces e Rich Faces, os quais trabalham com elementos JQuery e fornecem um design muito aprimorado.

Atividade

1. Qual a diferença entre a utilização do Tomcat e do GlassFish?

2. Qual classe do Java permite a manutenção de valores armazenados ao nível do servidor mesmo que entre chamadas sucessivas, enquanto o usuário mantiver a conexão?

a) HttpRequest
b) HttpServlet
c) HttpResponse
d) HttpSession
e) HttpListener

3. Considerando um Managed Bean chamado calculadora, que tem os atributos inteiros a e b, além dos métodos somar e subtrair, retornando, respectivamente, à soma e à subtração dos atributos anteriores, implemente uma página JSF para receber esses atributos e exibir o resultado das operações.

Referências
Notas

CASSATI, J. P. **Programação servidor em sistemas web.** Rio de Janeiro: Estácio, 2016.

DEITEL, P; DEITEL, H. **Ajax, rich internet applications e desenvolvimento web para programadores.** São Paulo: Pearson Education, 2009.

Java, como programar 8. ed. São Paulo: Pearson, 2010.

MARINHO, A.L. **Desenvolvimento de aplicações para internet.** 1. ed. São Paulo: Pearson, 2016.

SANTOS, F. **Tecnologias para internet II.** 1. ed. Rio de Janeiro: Estácio, 2017.

Próxima aula

- As características de bancos relacionais e sintaxe SQL.
- O conceito de Middleware e características do JDBC.
- O JDBC e padrão DAO para acesso ao banco de dados.

Explore mais

Leia os textos:

- Manual do GlassFish; <https://javaee.github.io/glassfish/doc/5.0/reference-manual.pdf >
- Tutorial de Servlet e JSP; <https://www.journaldev.com/2114/servlet-jsp-tutorial >
- Tutorial de JSF. <https://netbeans.org/kb/docs/web/jsf20-intro_pt_BR.html >

Assista ao vídeo:

- Configuração do Tomcat no Eclipse; <https://www.youtube.com/watch?v=9Z4OKoh-Omw >