

# JavaServer Pages

# Agenda

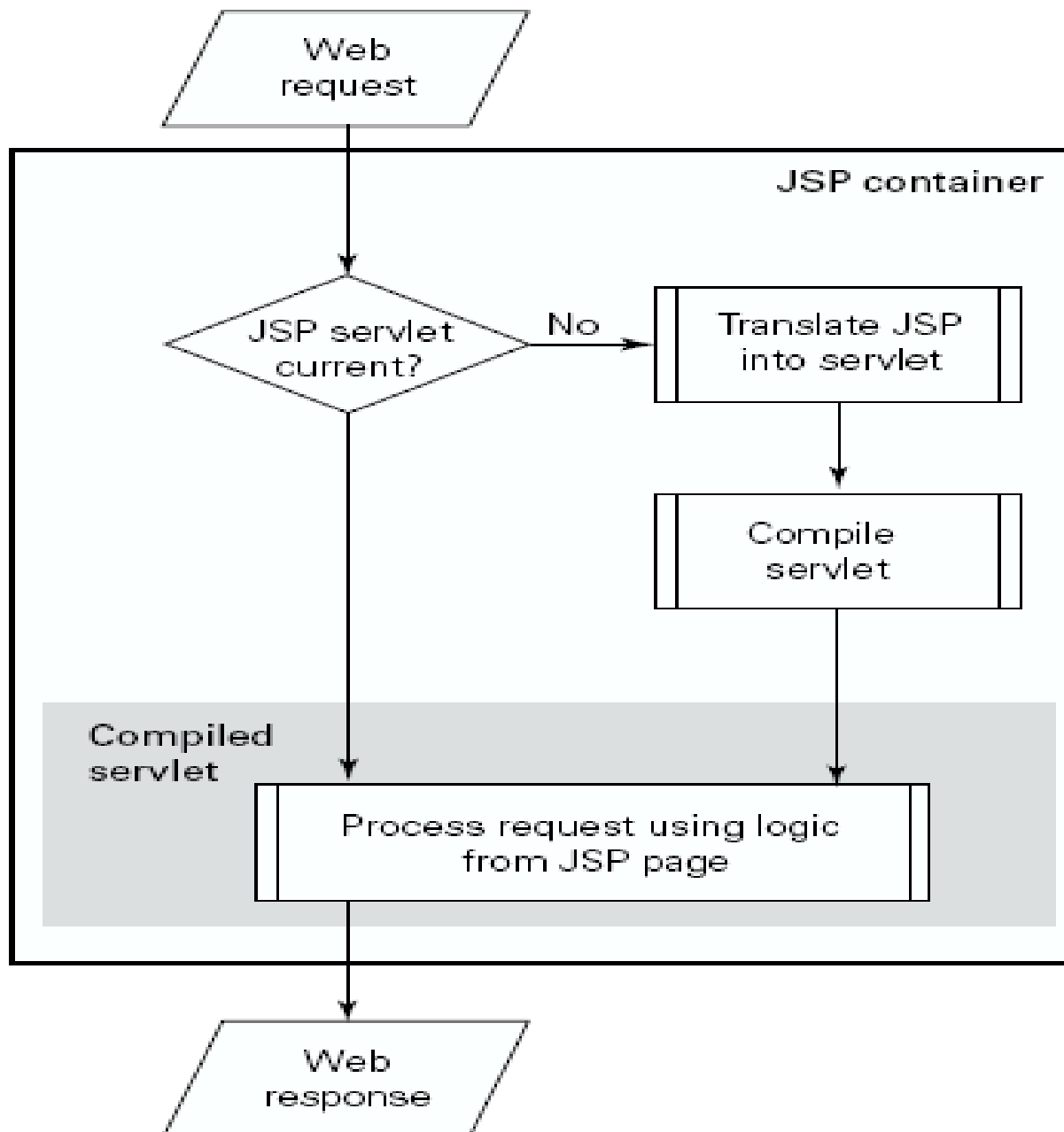
- O que é JSP;
- Suas vantagens e desvantagens;
- Escrever arquivos JSP com scriptlets;
- Usar Expression Language;
- O que são taglibs.

# Java Server Pages (JSP)

---

- ▶ Servlets → Toda vez que você precisa fazer uma mudança no HTML, o código Java deve ser mudado, re-compilado e re-executado. Problemas na manutenção das nossas páginas.
- ▶ JSP foi introduzido para resolver este problema.
  - ▶ Um mecanismo de template onde a lógica baseada em Java pode ser incorporada em páginas HTML. Podemos escrever código Java.
  - ▶ Detecção e compilação automática toda vez que a JSP é mudada
- ▶ Nas JSPs a página web está primeiro e o código está no segundo lugar.
- ▶ Eles são usualmente compilados em um Servlets.



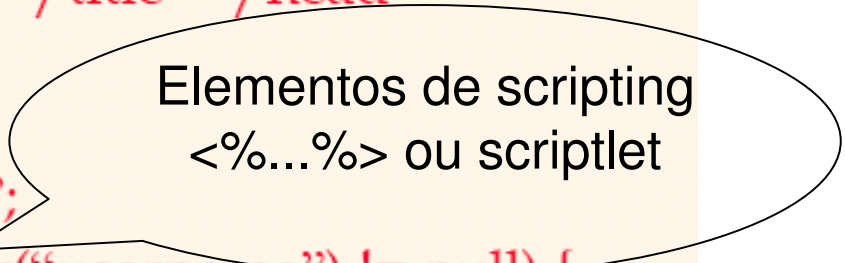


**Figure 4.2 The typical process for translating and running JSP pages**

# Exemplo de JSP

---

```
<html>
<head><title>Welcome to B&N</title></head>
<body>
  <h1>Welcome back!</h1>
  <% String name="NewUser";
    if (request.getParameter("username") != null) {
      name=request.getParameter("username");
    }
  %>
  You are logged on as user <%=name%>
  <p>
</body>
</html>
```



Elementos de scripting  
<%...%> ou scriptlet




- 
- ▶ Podemos usar variáveis já implícitas no JSP: todo arquivo JSP já possui uma variável chamada *out* (do tipo `JspWriter`) que permite imprimir para o response através do método `println`: `<% out.println(nome); %>`.
  - ▶ Funcionalidade semelhante ao `out` do Servlets mas sem precisarmos declará-lo antes.
  - ▶ Vejamos um exemplo: Crie um arquivo `WebContent/benvindo.jsp` com:



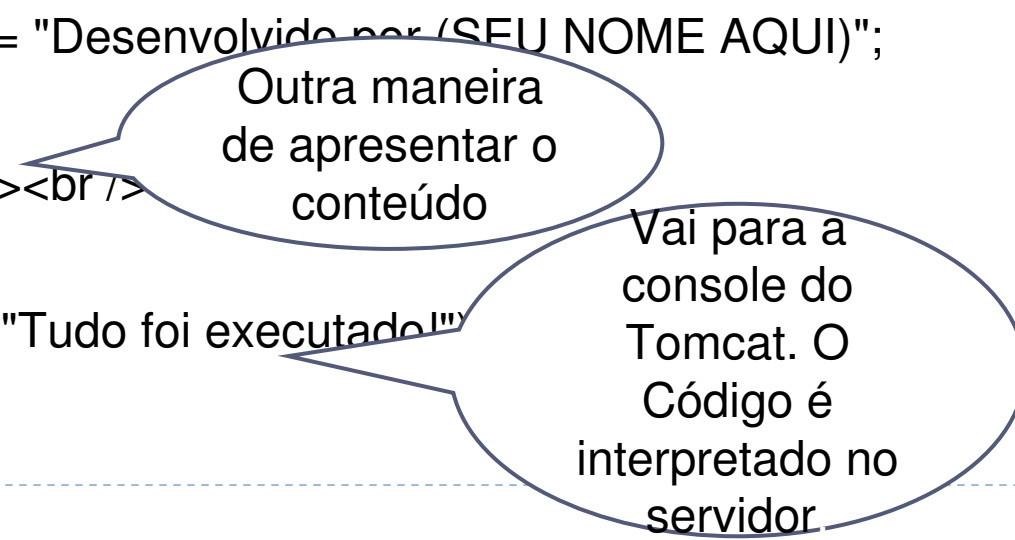
---

```
<html>
  <body>
    <%-- comentário em JSP aqui: nossa primeira página JSP --%>
    <%
      String mensagem = "Bem vindo ao sistema de agenda do FJ-21!";
    %>
    <% out.println(mensagem); %><br />
    <%
      String desenvolvido = "Desenvolvido por (SEU NOME AQUI)";
    %>
    <%= desenvolvido %><br />
    <%
      System.out.println("Tudo foi executado!");
    %>
  </body>
</html>
```

---



```
<html>
  <body>
    <%-- comentário em JSP aqui: nossa primeira página JSP --%>
    <%
      String mensagem = "Bem vindo ao sistema de agenda do FJ-21!";
    %>
    <% out.println(mensagem); %><br />
    <%
      String desenvolvido = "Desenvolvido por (SEU NOME AQUI)";
    %>
    <%= desenvolvido %><br />
    <%
      System.out.println("Tudo foi executado!")
    %>
  </body>
</html>
```



Outra maneira de apresentar o conteúdo

Vai para a console do Tomcat. O Código é interpretado no servidor



# Listando os contatos com Scriptlet

---

- ▶ Já que podemos escrever qualquer código Java no scriptlet, não fica difícil criar uma listagem de todos os contatos do BD.

```
<%  
ContatoDAO dao = new ContatoDAO();  
List<Contato> contatos = dao.getList();  
for (Contato contato : contatos ) {  
%>  
    <li><%=contato.getNome()%>, <%=contato.getEmail()%>:  
        <%=contato.getEndereco()%></li>  
%>  
}  
%>
```

---



- 
- ▶ No código acima ainda falta importar as classes dos pacotes corretos. Para isso, utilizamos diretiva com a seguinte:

```
<%@ page import="br.com.caelum.agenda.dao.ContatoDAO %>
```

- ▶ Parece um scriptlet, mas com um @. Diretiva de página
- ▶ O atributo *import* permite que seja especificado qual o pacote a ser importado. Este atributo pode aparecer várias vezes.



# Exercícios: Lista de contatos com scriptlet

## 1. Crie o arquivo **WebContent/lista-contatos-scriptlet.jsp**:

### ▶ **Importe os pacotes necessários:**

```
<%@ page import="java.util.*, br.com.caelum.agenda.dao.*, br.com.caelum.agenda.modelo.*" %>
```

```
<html>
```

```
<body>
```

```
<table>
```

```
<%
```

```
ContatoDAO dao = new ContatoDAO();
```

```
List<Contato> contatos = dao.getLista();
```

```
for (Contato contato : contatos ) {
```

```
%>
```

```
<tr>
```

```
<td><%=contato.getNome() %></td>
```

```
<td><%=contato.getEmail() %></td>
```

```
<td><%=contato.getEndereco() %></td>
```

```
<td><%=contato.getDataNascimento().getTime() %></td>
```

```
</tr>
```

```
<%
```

```
}
```

```
%>
```

```
</table>
```

```
</body>
```

```
▶ </html>
```

- 
2. Teste a url <http://localhost:8080/fj21-agenda/lista-contatos-scriptlet.jsp>
  3. Repare a data. Tente mostrá-la formatada utilizando a classe SimpleDateFormat
  4. Coloque cabeçalhos para as colunas da tabela.

Definição da página padrão de um site.

```
<welcome-file-list>
```

```
<welcome-file>bemvindo.jsp</welcome-file>
```

```
</welcome-file-list>
```

Reinicie o tomcat e acesse a URL: <http://localhost:8080/fj21-agenda/>



- 
2. Teste a url <http://localhost:8080/fj21-agenda/lista-contatos-scriptlet.jsp>
  3. Repare a data. Tente mostrá-la formatada utilizando a classe SimpleDateFormat
  4. Coloque cabeçalhos para as colunas

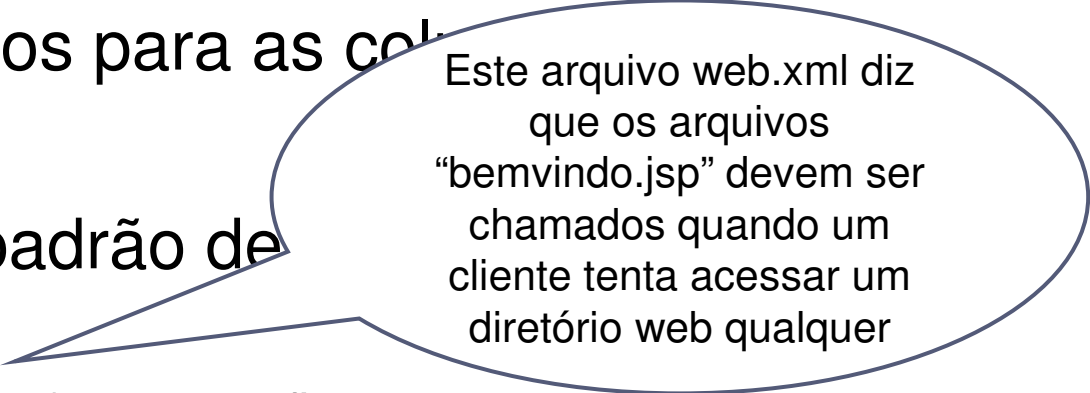
Definição da página padrão de

```
<welcome-file-list>
```

```
<welcome-file>bemvindo.jsp</welcome-file>
```

```
</welcome-file-list>
```

Reinicie o tomcat e acesse a URL: <http://localhost:8080/fj21-agenda/>



Este arquivo web.xml diz que os arquivos “bemvindo.jsp” devem ser chamados quando um cliente tenta acessar um diretório web qualquer



# EL: Expression Language

---

- ▶ A mistura ainda não é boa.
- ▶ Para remover um pouco do código Java que fica na página JSP, foi desenvolvida uma linguagem EL → interpretada pelo servlet container.
- ▶ Por exemplo, se o cliente chama a página `testaparam.jsp?idade=24`, o programa deve mostrar a mensagem que o cliente tem 24 anos.
- ▶ A seguir Exercícios com parâmetros →



- 
- ▶ Crie uma página chamada WebContent/digidade.jsp com o conteúdo:

```
<html>
  <body>
    Digite sua idade e pressione o botão:<br/>
    <form action="mostra-idade.jsp">
      Idade: <input name="idade"/> <input type="submit"/>
    </form>
  </body>
</html>
```

- ▶ Crie um arquivo chamado WebContent/mostra-idade.jsp
- 



---

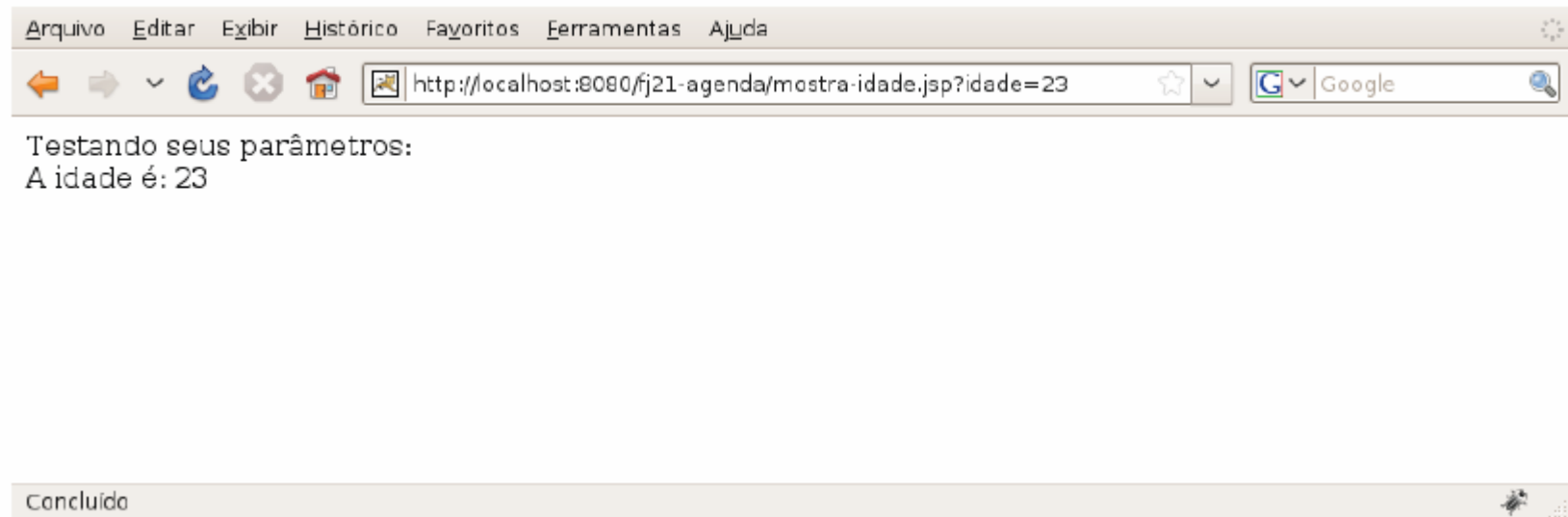
<html>

Testando seus parametros:<br/>

A idade é \${param.idade}.

</html>

- ▶ Teste o sistema acessando a página: <http://localhost:8080/fj21-agenda/digita-idade.jsp>.





- 
- ▶ Os arquivos JSPs não são compilados dentro do Eclipse → não precisamos das classes do driver.
  - ▶ Os JSPs são transformados em um servlet por um compilador JSP (embutido no Tomcat)



# Usando Taglibs

---

- ▶ Melhorar nossos problemas com relação à mistura de código Java com HTML → EL?
- ▶ Sozinha não pode nos ajudar muito. Não dá para instanciar objetos, fazer verificações condicionais (if else), iterações como for e assim por diante.
- ▶ Para que possamos ter esse comportamento sem impactar na legibilidade → escrever como HTML, código baseado em Tags.
- ▶ Resultado → conjunto de tags padrão – biblioteca de tags (taglib). Possui, entre outras tags, a funcionalidade de instanciar objetos através do construtor sem argumentos.



# Instanciando POJOs

---

- ▶ Os Javabeans devem possuir o construtor público sem argumentos, getters e setters (um típico Plain Old Java Object).
- ▶ Para instanciá-los na página JSP → utilizarmos a tag correspondente para essa função  
`<jsp:useBean>`
- ▶ Para utilizá-la, basta indicarmos qual a classe queremos instanciar e como se chamará a variável que será atribuída essa nova instância.

```
<jsp:useBean id="contato"  
  class="agenda.modelo.Contato"/>
```

- ▶ Agora, podemos imprimir o nome do contato:

---

```
▶ ${contato.nome}
```

# Instanciando POJOs

---

- ▶ Os Javabeans devem possuir o construtor público sem argumentos, getters e setters (um típico Plain Old Java Object).
- ▶ Para instanciá-los na página JSP → utilizarmos a tag correspondente para essa função  
`<jsp:useBean>`
- ▶ Para utilizá-la, basta indicarmos qual a classe queremos instanciar e como se chamará a variável que será atribuída essa nova instância.

```
<jsp:useBean id="contato"  
  class="agenda.modelo.Contato"/>
```

- ▶ Agora, podemos imprimir o nome

▶ `${contato.nome}`

**`${contato.Nome}`  
não funciona**

- 
- ▶ Mas, onde está o `getnome()`? A EL é capaz de perceber sozinha a necessidade de chamar um método do tipo `getter`, por isso o padrão `getter/setter` do POJO é tão importante.
  - ▶ Solução JSTL (JavaServer Pages Standard Tag Library)  
→ Padrão da SUN.
  - ▶ JSTL → API que encapsulou em tags simples toda a funcionalidade que diversas páginas Web precisam, como controle de laços (`for`), controle de fluxo (`if else`), manipulação de dados XML e a internacionalização da aplicação.
  - ▶ Existem ainda outras partes da JSTL, por exemplo aquela que acessa a BD e permite escrever códigos SQL na nossa página → só em casos especiais.

- 
- ▶ Muitas páginas JSP ainda possuem grandes pedaços de scriptlets.
  - ▶ Recomendação → usar JSP com JSTL para evitar o código incompreensível que pode ser gerado com scriptless.
  - ▶ Para instalar e implementar basta abaixar no <https://jstl.dev.java.net/>
  - ▶ Ao usar o JSTL em alguma página precisamos primeiro definir o cabeçalho. Existem 4 APIs básicas e iremos aprender primeiro a utilizar a biblioteca chamada **core**.



## Cabeçalho para o JSTL core

---

- ▶ Sempre que vamos utilizar uma taglib devemos primeiro escrever um cabeçalho através de uma tag JSP define qual taglib iremos utilizar e um nome, chamado prefixo. NO caso do JSTL → **c**

`<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`

- ▶ A URI não implica em uma requisição pelo protocolo http e sim uma busca entre os arquivos .jar no diretório lib.
  - ▶ Precisamos instanciar e declarar nosso DAO.
  - ▶ Já vimos que a tag **jsp:useBean** é capaz de instanciar determinada classe através do construtor sem argumentos e dar um nome id para essa
- 
- ▶ variável

---

```
<jsp:useBean id="dao"  
  class="br.com.caelum.agenda.dao.ContatoDAO"/>
```

- ▶ Tendo a variável dao, desejamos chamar o método getLista e podemos fazer isso através da EL:

```
${dao.lista}
```

- ▶ Desejamos executar um laço para cada contato dentro da coleção retornada por esse método (dao.lista). Em scriptlets era:

```
<%  
// ...  
List<Contato> contatos = dao.getLista();  
for (Contato contato : contatos ) {  
%>  
    <%=contato.getNome()%>, <%=contato.getEmail()%>,  
    <%=contato.getEndereco()%>, <%=contato.getDataNascimento() %>  
  
    <%  
    }  
%>
```

---

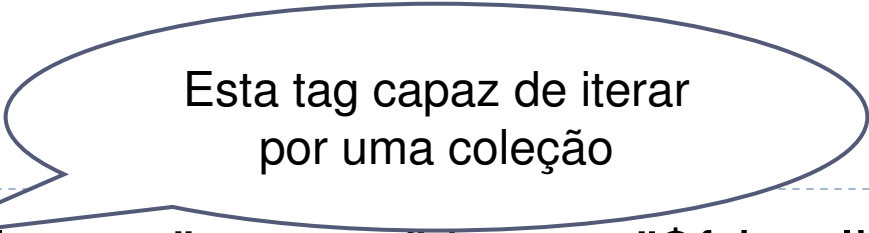


---

```
<c:forEach var="contato" items="${dao.lista}">  
    ${contato.nome}, ${contato.email}, ${contato.endereco},  
    ${contato.dataNascimento}  
</c:forEach>
```

- ▶ Precisamos indicar a coleção na qual vamos iterar, através do atributo `items` e também como chamará o objeto que será atribuído para cada iteração; através do atributo `var`.
- ▶ Qual código é mais elegante?





Esta tag capaz de iterar  
por uma coleção

```
<c:forEach var="contato" items="${dao.lista}">  
    ${contato.nome}, ${contato.email}, ${contato.endereco},  
    ${contato.dataNascimento}  
</c:forEach>
```

- ▶ Precisamos indicar a coleção na qual vamos iterar, através do atributo `items` e também como chamará o objeto que será atribuído para cada iteração; através do atributo `var`.
- ▶ Qual código é mais elegante?



## forEach e varStatus

---

- ▶ É possível criar um contador do tipo int dentro do seu laço forEach. Para isso basta definir o atributo chamado varStatus para a variável desejada e utilizar a propriedade count dessa variável.

```
<table border="1">  
  <c:forEach var="contato" items="${dao.lista}"  
    varStatus="id">  
    <tr bgcolor="#${id.count % 2 == 0 ? 'aaee88' : 'ffffff' }" >  
      <td>${id.count}</td><td>${contato.nome}</td>  
    </tr>  
  </c:forEach>  
</table>
```



# Exercício

---

1. Colocar os JARs da JSTL na aplicação
  - ▶ Copiar dois jars, `jstl-impl.jar` e `jstl-api.jar` em `workspace/fj21_agenda/WebContent/WEB-INF/lib`
  - ▶ No eclipse dê um F5 no seu projeto
2. Liste os contatos de ContatoDAO usando `jsp:useBeans` e JSTL
  - ▶ Crie o arquivo `WebContent/lista_contatos_elegante.jsp` com o conteúdo (seg. slide)
  - ▶ Acesse [http://localhost:8080/fj21\\_agenda/lista\\_contatos\\_elegante.jsp](http://localhost:8080/fj21_agenda/lista_contatos_elegante.jsp) - - >Veja Resultado



```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
  <body>
    <!-- cria a lista -->
    <jsp:useBean id="dao"
class="br.com.caelum.agenda.dao.ContatoDAO"/>
    <table>
      <!-- for -->
      <c:forEach var="contato" items="{dao.lista}">
        <tr>
          <td>${contato.nome}</td>
          <td>${contato.email}</td>
          <td>${contato.endereco}</td>
          <td>${contato.dataNascimento.time}</td>
        </tr>
      </c:forEach>
    </table>
  </body>
```

---

Arquivo Editar Exibir Histórico Favoritos Ferramentas Ajuda

← → ▾ ↺ × 🏠 📌 http://localhost:8080/fj21-agenda/lista-contatos-elegante.jsp ☆ ▾ G Google 🔍

João Carlos joao@caelum.com.br Rua Vergueiro, 3185 cj 87 Tue Feb 11 00:00:00 BRST 1986  
Luiz Felipe luiz@caelum.com.br Rua Vergueiro, 3185 cj 87 Sun Sep 23 00:00:00 BRT 1984  
Ricardo Luis ricardo@caelum.com.br Rua Vergueiro, 3185 cj 87 Fri Jan 07 00:00:00 BRT 1977

Concluído



## Exercício opcional

---

1. Coloque um cabeçalho nas colunas da tabela com um título dizendo à que se refere a coluna.
2. Utilize uma variável de status no seu `c:forEach` para colocar duas cores diferentes em linhas pares e ímpares. (Utilize o box imediatamente antes do exercício como auxílio)



## Evoluindo nossa listagem

---

- ▶ Suponhamos que ser quer uma listagem onde caso o usuário tenha e-mail cadastrado, coloquemos um link no e-mail que quando clicado abra o software de e-mail do computador do usuário.

```
<c:if test="${not empty contato.email}">
```

```
<a href="mailto:${contato.email}">${contato.email}</a>
```

```
</c:if>
```

- ▶ Tag c:if indica um teste lógico sobre o atributo a seguir “test”.
  - ▶ Agora é possível colocar a mensagem “e-mail não informado” caso não tenha sido preenchido. Porém não existe tag else na JSTL. Mas existe a tag c:choose.
- 





---

```
<c:choose>
```

```
  <c:when test="\${not empty contato.email}">
```

```
    <a
```

```
    href="mailto:\${contato.email}">\${contato.email}</a>
```


```
  </c:when>
```

```
  <c:otherwise>
```

```
    E-mail não informado
```

```
  </c:otherwise>
```

```
</c:choose>
```

- ▶ É possível fazer com duas tags c:if, a segunda com a lógica invertida, mas a solução não muito elegante.
- ▶ O código completo seria 



```
<c:forEach var="contato" items="${dao.lista}">
```

```
<tr>
```

```
<td>${contato.nome}</td>
```

```
<td>
```

```
<c:choose>
```

```
<c:when test="${not empty contato.email}">
```

```
<a
```

```
href="mailto:${contato.email}">${contato.email}</a>
```

```
</c:when>
```

```
<c:otherwise>
```

E-mail não informado

```
</c:otherwise>
```

```
</c:choose>
```

```
</td>
```

```
<td>${contato.endereco}</td>
```

```
<td>${contato.dataNascimento.time}</td>
```

```
</tr>
```

Exercício: testar  
este código

# Importando páginas

---

- ▶ Um requisito comum que temos nas aplicações web hoje em dia é colocar cabeçalhos e rodapé nas páginas do nosso sistema. Esses cabeçalhos e rodapés podem ter informações da empresa, do sistema e assim por diante. O problema é que deve ser colocado em todas as páginas.
- ▶ Criar um arquivo cabeçalho.jsp e todas as páginas da nossa aplicação, apenas dizem que precisam importar essa página → tag c:import
- ▶ Seja cabeçalho.jsp:

```
<html>
```

```
  <body>
```

```
     Nome da  
empresa
```

- 
- ▶ E uma página de rodapé → rodape.jsp

Copyright 2010 - Todos os direitos reservados

```
</body>
```

```
</html>
```



---

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<c:import url="cabecalho.jsp" />
<jsp:useBean id="dao" class="br.com.caelum.agenda.dao.ContatoDAO"/>
<table>
  <!-- for -->
  <c:forEach var="contato" items="${dao.lista}">
    <tr>
      <td>${contato.nome}</td>
      <td>${contato.email}</td>
      <td>${contato.endereco}</td>
      <td>${contato.dataNascimento.time}</td>
    </tr>
  </c:forEach>
</table>
<c:import url="rodape.jsp" />
```

---



# Exercícios

---

1. Criar um cabeçalho, utilizando algum logotipo. Crie a página cabeçalho.jsp com o seguinte conteúdo:

```
<html>
  <body>
    
    <h2>Agenda de Contatos do(a) (Seu nome aqui)</h2>
    <hr />
```

Criar a página rodape.jsp:

```
<hr />
  Copyright 2010 - Todos os direitos reservados
</body>
</html>
```

Importemos as duas páginas dentro da lista-

---

▶ contatos-elegante.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

```
<c:import url="cabecalho.jsp" />
```

```
<!-- cria a lista -->
```

```
<jsp:useBean id="dao" class="br.com.caelum.agenda.dao.ContatoDAO"/>
```

```
<table>
```

```
  <!-- for -->
```

```
  <c:forEach var="contato" items="${dao.lista}">
```

```
    <tr>
```

```
      <td>${contato.nome}</td>
```

```
      <c:if test="${not empty contato.email}">
```

```
        <a href="mailto:${contato.email}">${contato.email}</a>
```

```
      </c:if>
```

```
      <c:if test="${empty contato.email}">
```

```
        E-mail não informado
```

```
      </c:if>
```

```
      <td>${contato.endereco}</td>
```

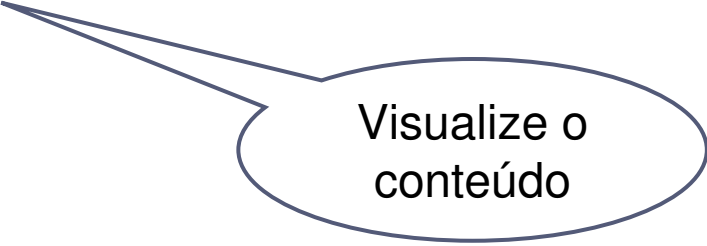
```
      <td>${contato.dataNascimento.time}</td>
```

```
    </tr>
```

```
  </c:forEach>
```

```
</table>
```

```
<c:import url="rodape.jsp" />
```



Visualize o  
conteúdo

# Trabalhando com links

---

- ▶ Muitas vezes trabalhar com links em nossa aplicação é complicado. Ex: no arquivo cabeçalho.jsp foi incluída uma imagem chamada exemplo.png que está na pasta imagens.  
`` indica um caminho relativo.
- ▶ Se cabeçalho.jsp estiver na raiz do projeto, o arquivo exemplo.png deverá estar em uma pasta imagens dentro da raiz do projeto → perigoso mudança do arquivo jsp implicará na não exibição da imagem.
- ▶ Para resolver este problema utilizamos a tag `<c:url>` da JSTL e no cabeçalho.jsp



---

```
<c:url value="/imagens/exemplo.gif" var="imagem"/>
```

```

```

Ou, mais simples:

```
" />
```



# Formatando as datas

---

- ▶ A listagem tem problemas com a formatação das datas. No lugar de fazer a formatação através da classe Java SimpleDateFormat (evitar código Java no JSP). Usaremos outra Taglib da JSTL → fmt. Precisamos importá-la.
- ▶ `<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>`
- ▶ Dentro da taglib fmt existe a tag formatDate, que faz com que um objeto do tipo java.util.Date seja formatado para um dado pattern.
- ▶ `<fmt:formatDate value="${contato.dataNascimento.time}" pattern="dd/MM/yyyy" />`
- ▶ Podemos colocar no atributo pattern outras informações:
  - ▶ m – Minutos; s – Segundos; H – Horas (0 -23); D – Dia no ano, ex: 230.



# Outras Tags

---

- ▶ JSTL possui várias tags:
  - ▶ c:import - import
  - ▶ c:otherwise - default do switch
  - ▶ c:out - saída
  - ▶ c:param - parâmetro
  - ▶ c:redirect - redirecionamento
  - ▶ c:remove - remoção de variável
  - ▶ c:set - criação de variável
  - ▶ c:url - veja adiante
  - ▶ c:when - teste para o switch



## Indo além da JSTL

---

- ▶ É muito comum, no desenvolvimento de JSPs, ter muita repetição de código. Por exemplo, sempre que for criada uma caixa de texto →
- ▶ `<label for="nomeContato">Nome</label> <input type="text" id="nomeContato" name="nome" />`
- ▶ Não seria mais simples:
- ▶ `<campoTexto id="nomeContato" name="nome" label="Nome:" />`
- ▶ Outro exemplo, utilizando componentes Javascript: um campo que mostra um calendário. Componentes visuais do JQuery, uma biblioteca com alguns componentes Javascript. Um desses é o campo com calendário → datepicker. Para usá-lo precisamos criar um input de texto seimples e associá-lo com
- ▶ uma função Javascript.

---

```
<script type="text/javascript">
    $(function() {
        $("#dataNascimento").datepicker();
    });
</script>
<input id="dataNascimento" type="text">
```

- Imagine se toda vez que precisemos um campo data, tivéssemos que escrever esse código. Maiores chances de erro e perda de tempo escrevendo código repetitivo. Não seria melhor:

```
<campoData id="dataNascimento" />
```

---



## Criando as suas próprias tags.

---

- ▶ Para poder ter a tag <campoData> precisamos criá-la. Uma das formas que temos de criar nossas próprias taglibs é criando um arquivo contendo o código que nossa Taglib gerará → tagfiles.
- ▶ Tagfiles: pedaços de JSP com a extensão “.tag” contendo o código que queremos que a nossa tag gere.
- ▶ Seja campoData.tag:

```
<script type="text/javascript">  
    $(function() {  
        $("#dataNascimento").datepicker();  
    });  
</script>
```

---

```
<input id="dataNascimento" name="dataNascimento" type="text">
```

- 
- ▶ Tag está totalmente inflexível, pois o id e o nome do campo estão dentro da tag. Se tivéssemos dois calendários dentro do mesmo formulário?
  - ▶ Tag com parâmetros. Adicionar a diretiva `<%@attribute` com parâmetros `name` representando o nome do atributo e o parâmetro `required` (`true` ou `false`) indicando se o parâmetro é obrigatório ou não.  
`<%@attribute name="id" required="true" %>`
  - ▶ Agora basta usarmos esses parâmetros nos lugares adequados através de `expression language`.



---

```
<%@attribute name="id" required="true" %>
```

```
<script type="text/javascript">
```

```
    $(function() {  
        $("#${id}").datepicker();  
    });
```

```
</script>
```

```
<input id="${id}" name="${id}" type="text">
```

- ▶ Os JSPs que vão utilizar esta tag precisam importar os tagfiles. Para isso precisamos usar a diretiva taglib que recebe o caminho das tags e um prefixo:

```
<%@taglib tagdir="/WEB-INF/tags" prefix="javappl" %>
```

- ▶ E o tag pode ser usado:

```
< javappl :campoData id="dataNascimento" />
```





---

```
<%@attribute name="id" required="true" %>
```

```
<script type="text/javascript">
```

```
    $(function() {  
        $("#${id}").datepicker();  
    });
```

```
</script>
```

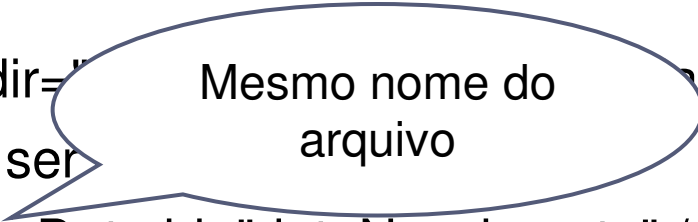
```
<input id="${id}" name="${id}" type="text">
```

- ▶ Os JSPs que vão utilizar esta tag precisam importar os tagfiles. Para isso precisamos usar a diretiva taglib que recebe o caminho das tags e um prefixo:

```
<%@taglib tagdir="..." prefix="javappl" %>
```

- ▶ E o tag pode ser

```
< javappl :campoData id="dataNascimento" />
```



Mesmo nome do  
arquivo



# Utilizando bibliotecas Javascript e estilos em CSS

---

- ▶ Para podermos utilizar bibliotecas Javascript precisamos importar o arquivo “.js” que contem a biblioteca. Para fazermos essa importação:

```
<head>  
<script type="text/javascript" src="js/arquivo.js"></script>  
</head>
```

- ▶ Para poder construir uma interface agradável → usar CSS (Cascading Stylesheet). Arquivo contendo as definições visuais para sua página – extensão “.css”. Para fazer a importação:

```
<head>  
<link type="text/css" href="css/meuArquivo.css" rel="stylesheet" />  
</head>
```



# Exercícios

---

1. Criação da tag para o campo de calendário com datepicker → biblioteca javascript JQuery
  - a. Copie os diretórios js e css dentro de WebContent no seu projeto. Fazer download desses arquivos em <http://jqueryui.com/download>
  - b. Precisamos importar o JQuery agora nos nossos arquivos. No cabeçalho.jsp, adicionar dentro a tag html:

```
<html>
  <head>
    <link type="text/css" href="css/jquery.css" rel="stylesheet" />
    <script type="text/javascript" src="js/jquery.js"></script>
    <script type="text/javascript" src="js/jquery-ui.js"></script>
  </head>
  <body>
    <!-- Restante do cabeçalho aqui -->
```

---



- 
2. Agora temos que importar nosso cabeçalho.jsp na página aciona-contatos.html. Que tem que trocar a extensão para “.jsp”. Remova a declaração <html> e <body> e inclua:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

```
<c:import url="cabecalho.jsp" />
```

```
<!-- Aqui continua o formulário com os campos -->
```

```
<c:import url="rodape.jsp" />
```

Acesse agora <http://localhost:8080/fj21-agenda/adiciona-contato.jsp> e verifique o resultado



---

### 3. Cria a nossa tag calendário.

- a. Dentro de WEB-INF crie u diretório tags.
- b. Crie um arquivo campoData.tag com o seguinte conteúdo:

```
<%@ attribute name="id" required="true" %>
<script type="text/javascript">
    $(function() {
        $("#${id}").datepicker({dateFormat: 'dd/mm/yy'});
    });
</script>
<input type="text" id="${id}" name="${id}" />
```

- c. Utilizar a tag modificando adiciona- contato.jsp colocando junto à declaração da Taglib core:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@taglib tagdir="/WEB-INF/tags" prefix="javappl" %>
```

---

- 
- d. Vamos trocar o input de data de nascimento pela nossa nova tag:

```
<form action="adicionaContato">
```

```
  Nome: <input type="text" name="nome" /><br />
```

```
  E-mail: <input type="text" name="email" /><br />
```

```
  Endereço: <input type="text" name="endereco" /><br />
```

```
  Data Nascimento: <caelum:campoData id="dataNascimento" /><br />
```

```
  <input type="submit" value="Gravar" />
```

```
</form>
```

- e. Recarregue a página adiciona-contato.jsp e clique no campo da data de nascimento. O calendário deve ser aberto, escolha uma data e faça a gravação do contato.
-

## Outras taglibs no mercado

---

- ▶ Existem muitas Taglibs disponíveis. Antes de criar uma, verifique se já não existe:
  - ▶ **Displaytag:** É uma Taglib para geração fácil de tabelas, e pode ser encontrada em <http://displaytag.sf.net>
  - ▶ **Cewolf:** É uma Taglib para geração de gráficos nas suas páginas e também pode ser encontrada em: <http://cewolf.sourceforge.net/new/>
  - ▶ **Waffle Taglib:** Tags para auxiliar na criação de formulários HTML que é encontrada em: <http://waffle.codehaus.org/taglib.html>



# MVC – Model View Controller

---

- ▶ Colocar HTML dentro de uma Servlet não é a melhor idéia → dificulta mudar o design da página; designer em geral não conhece java
  - ▶ Imagine agora usar apenas JSP → problema o uso de muito scriptlet, difícil de dar manutenção.
  - ▶ Idéia → usar o que é bom de cada um dos dois.
  - ▶ O JSP foi feito para apresentar o resultado, e ele não deveria fazer acesso a BD e nem fazer a instanciação de objetos. Papel do servlets.
  - ▶ O ideal então é que a Servlet faça o trabalho da lógica de negócios, e o JSP apenas apresente visualmente os resultados gerados pela Servlet →
- 
- ▶ lógica de apresentação.



- 
- ▶ Seja o código do método da servlet AdicionaContatoServlet feito antes:

```
protected void service(  
    HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
    // log  
    System.out.println("Tentando criar um novo contato...");  
    // acessa o bean  
    Contato contato = new Contato();  
    // chama os setters  
    ...  
    // adiciona ao banco de dados  
    ContatoDAO dao = new ContatoDAO();  
    dao.adiciona(contato);  
    // ok.... visualização  
    out.println("<html>");  
    out.println("<body>");  
    out.println("Contato " + contato.getNome() + " adicionado com sucesso");  
    out.println("</body>");  
    out.println("</html>");  
}
```

---

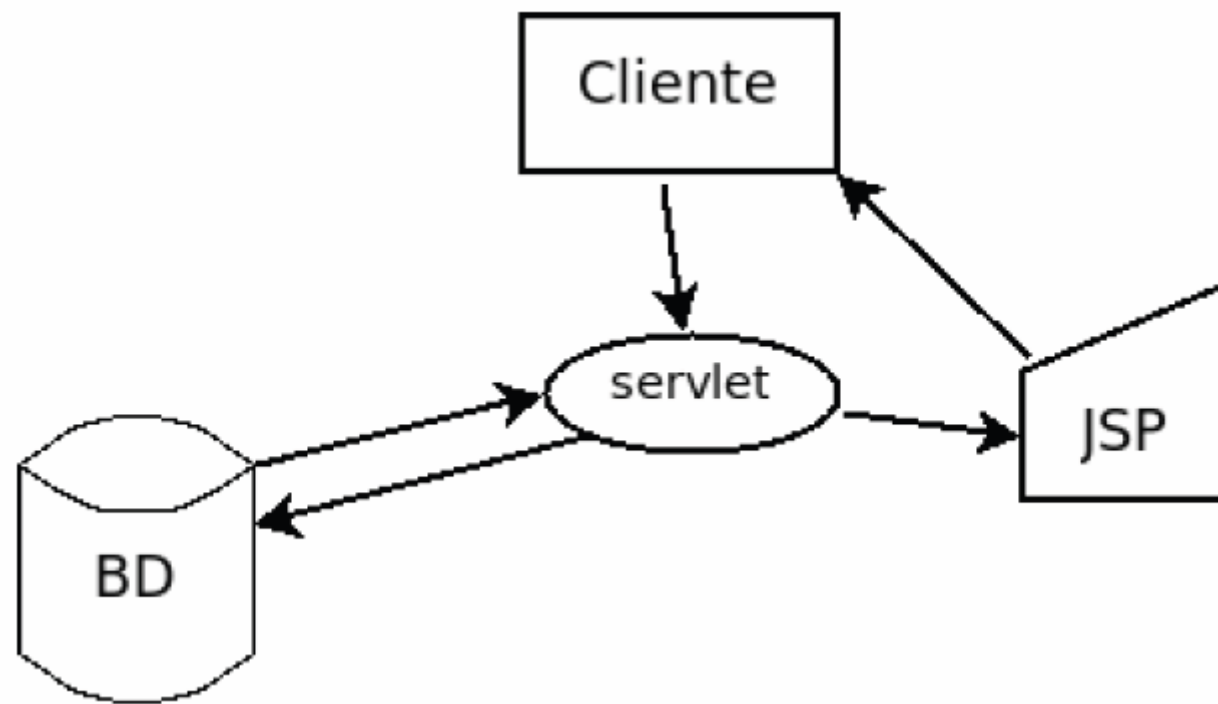


- 
- ▶ O código mistura HTML com Java. Portanto, seria bom criar um arquivo JSP `contato-adicionado.jsp` apenas com o HTML:

```
<html>
  <body>
    Contato ${param.nome} adicionado com sucesso
  </body>
</html>
```

- ▶ Precisamos redirecionar as requisições. Encaminhar a requisição para um outro recurso: Servlet → JSP. Dispatch de requisições.





# Request Dispatcher

---

- ▶ A API, como já foi visto permite separar a apresentação da lógica. Basta conhecermos a URL e podermos usar um objeto RequestDispatcher para acessar outro recurso Web (JSP ou outra Servlet).

```
RequestDispatcher rd = request.getRequestDispatcher("/contato-  
adicionado.jsp");
```

```
rd.forward(request,response);
```

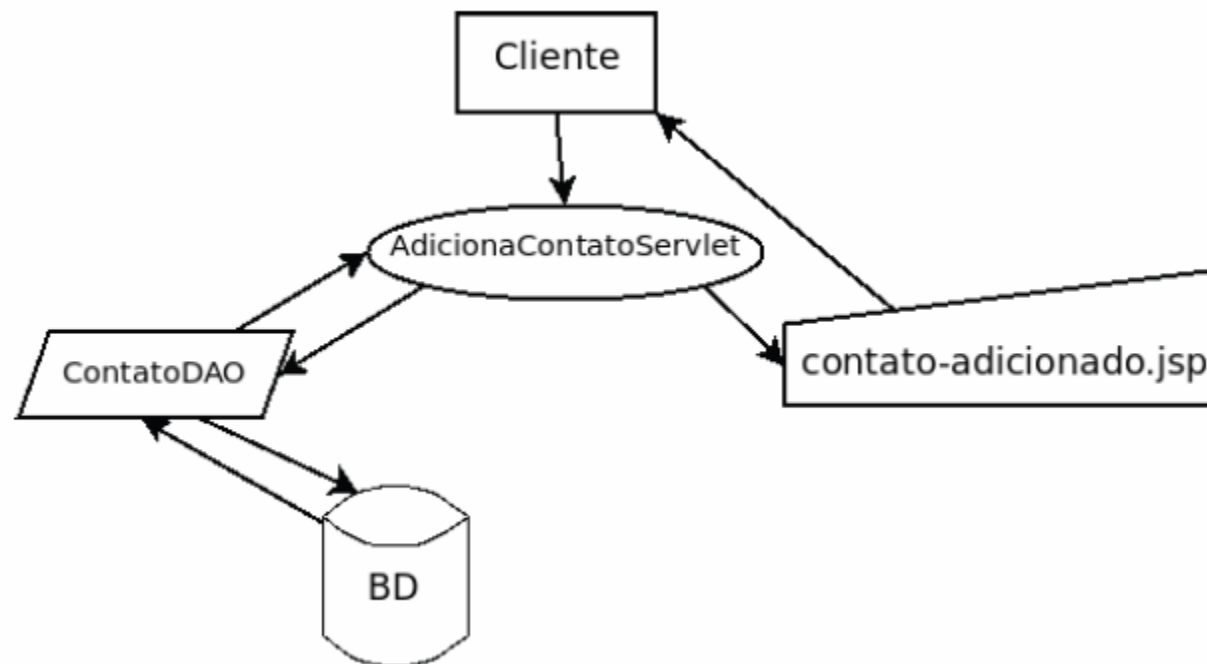
- ▶ Podemos agora executar a lógica na Servlet e então redirecionar o resultado para a JSP.
- ▶ O método forward só pode ser chamado quando nada fora escrito para a saída.
- ▶ Existe outro método da classe RequestDispatcher que representa a inclusão e não o redirecionamento → include
- ▶ Acrescenta ao resultado de uma página os dados de outra.



## Melhorando o processo

---

- ▶ Temos várias servlets acessando o BD, trabalhando com os DAOs e pedindo para que o JSP apresente esses dados. Veja o diagrama a seguir:



---

▶ Como centralizar todos os servlets?

- 
- ▶ Para cada lógica de negócio → servlet diferente. Isso implica 8 linhas de código no web.xml. Imagine 10 classes de modelo, 5 lógicas diferentes → 400 linhas de configuração.
  - ▶ Podemos colocar tudo em uma Servlet (Veja Figura).
  - ▶ Poderíamos acessar como:  
<http://localhost:8080/contexto/sistema?logica=AdicionaContato>



```
// Todas as lógicas dentro de uma Servlet
public class SistemaTodoServlet extends HttpServlet {

    protected void service(HttpServletRequest request, HttpServletResponse
response) {
        String acao = request.getParameter("logica");
        ContatoDAO dao = new ContatoDAO();
        if (acao.equals("AdicionaContato")) {
            Contato contato = new Contato();
            contato.setNome(request.getParameter("nome"));
            contato.setEndereco(request.getParameter("endereco"));
            contato.setEmail(request.getParameter("email"));
            dao.adiciona(contato);
            RequestDispatcher rd =
                request.getRequestDispatcher("/contato-adicionado.jsp");
            rd.forward(request, response);
        } else if (acao.equals("ListaContatos")) {
            // busca a lista no DAO
            // despacha para um jsp
        } else if (acao.equals("RemoveContato")) {
            // faz a remoção e redireciona para a lista
        }
    }
}
```

---

- 
- ▶ Para cada ação teríamos um if/else, tornando o Servlet muito grande.
  - ▶ Porque não colocar cada regra de negócio (como inserir contato, remover contato, fazer relatório, etc.) em uma classe separada. Cada ação (regra de negócio) em nossa aplicação estaria em uma classe. O código poderia ser:

```
if (acao.equals("AdicionaContato")) {  
    new AdicionaContato().executa(request,response);  
} else if (acao.equals( "ListaContato")) {  
    new ListaContatos().executa(request,response);  
}
```

- ▶ Executa faz a lógica de negócios apropriada.
- ▶ A cada lógica nova, removida, alteração, etc., temos que alterar o Servlet → trabalhoso e propenso a erros





- 
- ▶ O código acima possui o mesmo comportamento de switch → pode ser substituído por polimorfismo quase sempre.
  - ▶ Tentemos generalizar:  

```
String nomeDaClasse = request.getParameter("logica");  
new nomeDaClasse().executa(request,response);
```
  - ▶ O código acima não é válido. nomeDaClasse é o nome de uma variável.
  - ▶ Mas a partir do nome da classe podemos recuperar um objeto que representará as informações contidas dentro daquela classe: atributos, métodos e construtores. Exemplo →



---

```
String nomeDaClasse = "nomePacote." +  
    request.getParameter("logica");  
Class classe = Class.forName(nomeDaClasse);
```

- ▶ Agora podemos ter uma representação das classes, mas como instanciar essas classes?
- ▶ Uma das informações contidas pelo objeto do tipo Class é o construtor, portanto podemos invocá-lo para instanciar a classe através do método `newInstance`:

```
Object objeto = classe.newInstance();
```

- ▶ E agora, um primeira alternativa, seria fazer novamente if/else para sabermos qual é a lógica invocada:



---

```
String nomeDaClasse = "nomePacote.mvc." +  
    request.getParameter("logica");  
Class classe = Class.forName(nomeDaClasse);  
Object objeto = classe.newInstance();  
if (nomeDaClasse.equals("nomePacote.mvc.AdicionaContato")) {  
    ((AdicionaContato) objeto).executa(request, response);  
} else if (nomeDaClasse.equals("nomePacote.mvc.ListaContatos")) {  
    ((ListaContatos) objeto).executa(request, response);  
} //e assim por diante
```

- ▶ Mas, de novo if/else. newInstance() retorna um tipo Object que pode ser qualquer uma das lógicas. Podemos criar uma Interface Lógica que declara o método executa:

```
public interface Logica {  
    void executa(HttpServletRequest req, HttpServletResponse res) throws  
    Exception;
```

---

```
}
```

- 
- ▶ Podemos simplificar nossa Servlet para executar a lógica de forma polimórfica e, tudo aquilo que fazíamos em 8 linhas de código, agora podemos fazer em apenas 2:

```
Logica logica = (Logica) classe.newInstance();  
logica.executa(request, response);
```

- ▶ Uma lógica simples para logar algo no console poderia ser:

```
public class PrimeiraLogica implements Logica {  
    public void executa(HttpServletRequest req, HttpServletResponse res)  
        throws Exception {  
        System.out.println("Executando a logica e redirecionando...");  
        RequestDispatcher rd = req.getRequestDispatcher("/primeira-logica.jsp");  
        rd.forward(req, res);  
    }  
}
```

---

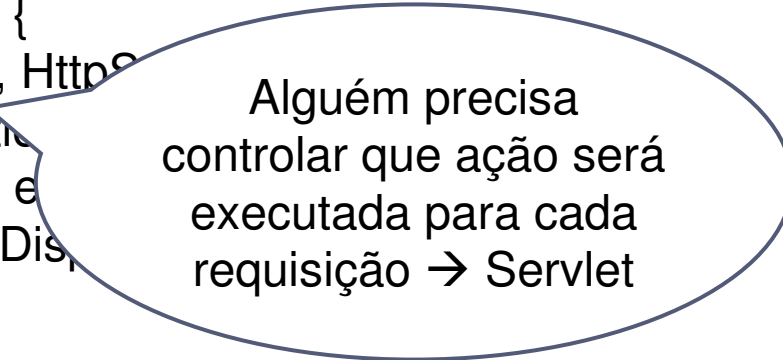


- 
- ▶ Podemos simplificar nossa Servlet para executar a lógica de forma polimórfica e, tudo aquilo que fazíamos em 8 linhas de código, agora podemos fazer em apenas 2:

```
Logica logica = (Logica) classe.newInstance();  
logica.executa(request, response);
```

- ▶ Uma lógica simples para logar algo no console poderia ser:

```
public class PrimeiraLogica implements Logica {  
    public void executa(HttpServletRequest req, HttpServletResponse res)  
        throws Exception {  
        System.out.println("Executando a logica e  
RequestDispatcher rd = req.getRequestDispatcher(  
rd.forward(req, res);  
    }  
}
```



Alguém precisa  
controlar que ação será  
executada para cada  
requisição → Servlet

# A configuração do web.xml

---

- ▶ A única coisa que falta para que tudo o que fizemos funcione é declarar a nossa única Servlet.

```
<web-app>
    <servlet>
        <servlet-name>controlador</servlet-name>
        <servlet-class>br.com.caelum.mvc.ControllerServlet</servlet-
class>
    </servlet>

    <servlet-mapping>
        <servlet-name>controlador</servlet-name>
        <url-pattern>/mvc</url-pattern>
    </servlet-mapping>
</web-app>
```



# Exercícios: Criando as lógicas e servlet de controle

---

1. Crie a sua interface no pacote nomePacote.mvc.logica:

```
public interface Logica {  
    void executa(HttpServletRequest req, HttpServletResponse res)  
        throws Exception;  
}
```

2. Crie uma implementação da interface Logica, classe PrimeiraLogica, no pacote acima.

```
1 public class PrimeiraLogica implements Logica {  
2     public void executa(HttpServletRequest req, HttpServletResponse res)  
        throws Exception {  
3         System.out.println("Executando a logica e redirecionando...");  
4  
5         RequestDispatcher rd = req.getRequestDispatcher("/primeira-  
            logica.jsp");  
6         rd.forward(req, res);  
7     }  
8 }
```

---

- 
3. Faça um arquivo jsp chamado primeira-logica.jsp dentro do diretório WebContent:

```
<html>  
  <body>  
    <h1> Página da nossa primeira lógica </h1>  
  </body>  
</html>
```

4. Escreva a Servlet que coordenará o fluxo da aplicação. Coloque-a nome Pacote.mvc.servlet






---

```
public class ControllerServlet extends HttpServlet {  
    protected void service(HttpServletRequest request, HttpServletResponse  
        response)  
        throws ServletException, IOException {  
        String parametro = request.getParameter("logica");  
        String nomeDaClasse = "br.com.caelum.mvc.logica." + parametro;  
  
        try {  
            Class classe = Class.forName(nomeDaClasse);  
            Logica logica = (Logica) classe.newInstance();  
            logica.executa(request, response);  
        } catch (Exception e) {  
            throw new ServletException("A lógica de negócios causou uma  
                exceção", e);  
        }  
    }  
}
```

---



---

5. Mapeie a URL /mvc para essa servlet no arquivo web.xml

```
<servlet>
    <servlet-name>controlador</servlet-name>
    <servlet-class>br.com.caelum.mvc.servlet.ControllerServlet</servlet-
class>
</servlet>

<servlet-mapping>
    <servlet-name>controlador</servlet-name>
    <url-pattern>/mvc</url-pattern>
</servlet-mapping>
```

6. Teste a url:

<http://localhost:8080/fj21-agenda/mvc?logica=PrimeiraLogica>



# Exercícios: Lógica para alterar contatos

1. Crie uma nova classe chamada AlteraContatoLogic no pacote meuPacote.mvc.logica.

```
public class AlteraContatoLogic implements Logica {  
    public void executa(HttpServletRequest request, HttpServletResponse response)  
        throws Exception {  
        Contato contato = new Contato();  
        long id = Long.parseLong(request.getParameter("id"));  
        contato.setId(id);  
        contato.setNome(request.getParameter("nome"));  
        contato.setEndereco(request.getParameter("endereco"));  
        contato.setEmail(request.getParameter("email"));  
        //Converte a data de String para Calendar  
        String dataEmTexto = request.getParameter("dataNascimento");  
        Date date = new SimpleDateFormat("dd/MM/yyyy").parse(dataEmTexto);  
        Calendar dataNascimento = Calendar.getInstance();  
        dataNascimento.setTime(date);  
        contato.setDataNascimento(dataNascimento);  
        ContatoDAO dao = new ContatoDAO();  
        dao.atualiza(contato);  
        RequestDispatcher rd = request.getRequestDispatcher("/lista-contatos-elegante.jsp");  
        rd.forward(request, response);  
        System.out.println("Alterando contato ..." + contato.getNome());  
    }  
}
```

}



2. Crie uma nova página altera-contato.jsp que através do método POST que chama a lógica criada acima.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib tagdir="/WEB-INF/tags" prefix="caelum" %>
<c:import url="cabecalho.jsp" />
Formulário para alteração de contatos:<br/>
<form action="mvc" method="POST">
    Id: <input type="text" name="id"/><br/>
    Nome: <input type="text" name="nome"/><br/>
    E-mail: <input type="text" name="email"/><br/>
    Endereço: <input type="text" name="endereco"/><br/>
    Data de Nascimento: <caelum:campoData id="dataNascimento" />
        <input type="hidden" name="logica"
value="AlteraContatoLogic"/>
        <input type="submit" value="Enviar"/>
</form>
<c:import url="rodape.jsp" />
```



# Model View Controller

---

- ▶ Responsável por apresentar os resultados na página web é chamado de Apresentação (view).
- ▶ A servlet (e auxiliares) que faz os dispatchs para quem deve executar determinada tarefa é chamada de Controladora (Controller)
- ▶ As classes que representam suas entidades e as que ajudam a armazenar e buscar dados são chamadas de Modelo (Model).
- ▶ MVC garante a separação de tarefas, facilitando assim a reescrita de alguma parte, e a manutenção do código.
- ▶ O famoso Struts ajuda a implementar MVC, pois tem uma controladora já pronta, com uma série de ferramentas para se auxiliar. O Hibernate pode ser usado como Model. E como View você não precisa usar só JSP, pode usar a ferramenta Velocity, por exemplo.

# Lista de tecnologias: camada de controle

---

- 1) **Struts Action** - o controlador mais famoso do mercado Java, é utilizado principalmente por ser o mais divulgado e com tutoriais mais acessíveis. Possui vantagens características do MVC e desvantagens que na época ainda não eram percebidas. É o controlador pedido na maior parte das vagas em Java hoje em dia.
- 2) **JSF** - JSF é uma especificação da Sun para frameworks MVC. Ele é baseado em componentes e possui várias facilidades para desenvolver a interface gráfica. Devido ao fato de ser um padrão da Sun ele é bastante adotado.
- 3) **JBoss Seam** - segue as idéias do Stripes na área de anotações e é desenvolvido pelo pessoal que trabalhou também no Hibernate. Trabalha muito bem com o Java Server Faces e EJB 3.
- 4) **Spring MVC** - é uma parte do Spring Framework focado em implementar um controlador MVC. É fácil de usar em suas últimas versões e tem a vantagem de se integrar a toda a estrutura do Spring.
- 5) **Stripes** - um dos frameworks criados em 2005, abusa das anotações para facilitar a configuração.



# Lista de tecnologias: Camada de visualização

---

1. **JSP** - como já vimos, o JavaServer Pages, temos uma boa idéia do que ele é, suas vantagens e desvantagens. O uso de taglibs (a JSTL por exemplo) e expression language é muito importante se você escolher JSP para o seu projeto. É a escolha do mercado hoje em dia.
  2. **Velocity** - um projeto antigo, no qual a EL do JSP se baseou, capaz de fazer tudo o que você precisa para a sua página de uma maneira extremamente compacta.
  3. **Freemarker** - similar ao Velocity e com idéias do JSP - como suporte a taglibs - o freemarker vem sendo cada vez mais utilizado, ele possui diversas ferramentas na hora de formatar seu texto que facilitam muito o trabalho do designer.
  4. **Sitemesh** - não é uma alternativa para as ferramentas anteriores mas sim uma maneira de criar templates para seu site, com uma idéia muito parecida com o struts tiles, porém genérica: funciona inclusive com outras linguagens como PHP etc.
- 



## Recursos importantes: Filtros

---

- ▶ Criar classes que filtram a requisição e a resposta
- ▶ Guardar objetos na requisição





# Reduzindo o acoplamento com filtros

---

- ▶ Qualquer aplicação tem requisitos que não são diretamente relacionados com a regra de negócios. Requisitos não funcionais: auditoria (Logging), autorização, etc.
- ▶ Como implementar estas facilidades? Colocar o código diretamente na classe que possui a lógica. Exemplo → Próximo Slide.
- ▶ Podemos ver que além da lógica é preciso implementar os outros requisitos, mas não só apenas em uma funcionalidade (altera o contato) senão em todas as outras (adiciona, remove ou faz listagem de contatos). Esta solução cria um acoplamento muito forte entre a lógica e a implementação dos requisitos não funcionais → Veja Figura.

---

```
public class AlteraContatoLogic implements Logica {  
    public void executa(HttpServletRequest request, HttpServletResponse  
        response)  
        throws Exception {  
        //auditoria  
        Logger.info("acessando altera contato logic");  
        //autorização  
        if(!usuario.ehCliente()) {  
            request.getRequestDispatcher("/acessoNegado.jsp").  
                forward(request,response);  
        }  
        //toda lógica para alterar o contato aqui  
        //...  
    }  
}
```

---



Requisições

AlteraContatoLogic

RemoveContatoLogic

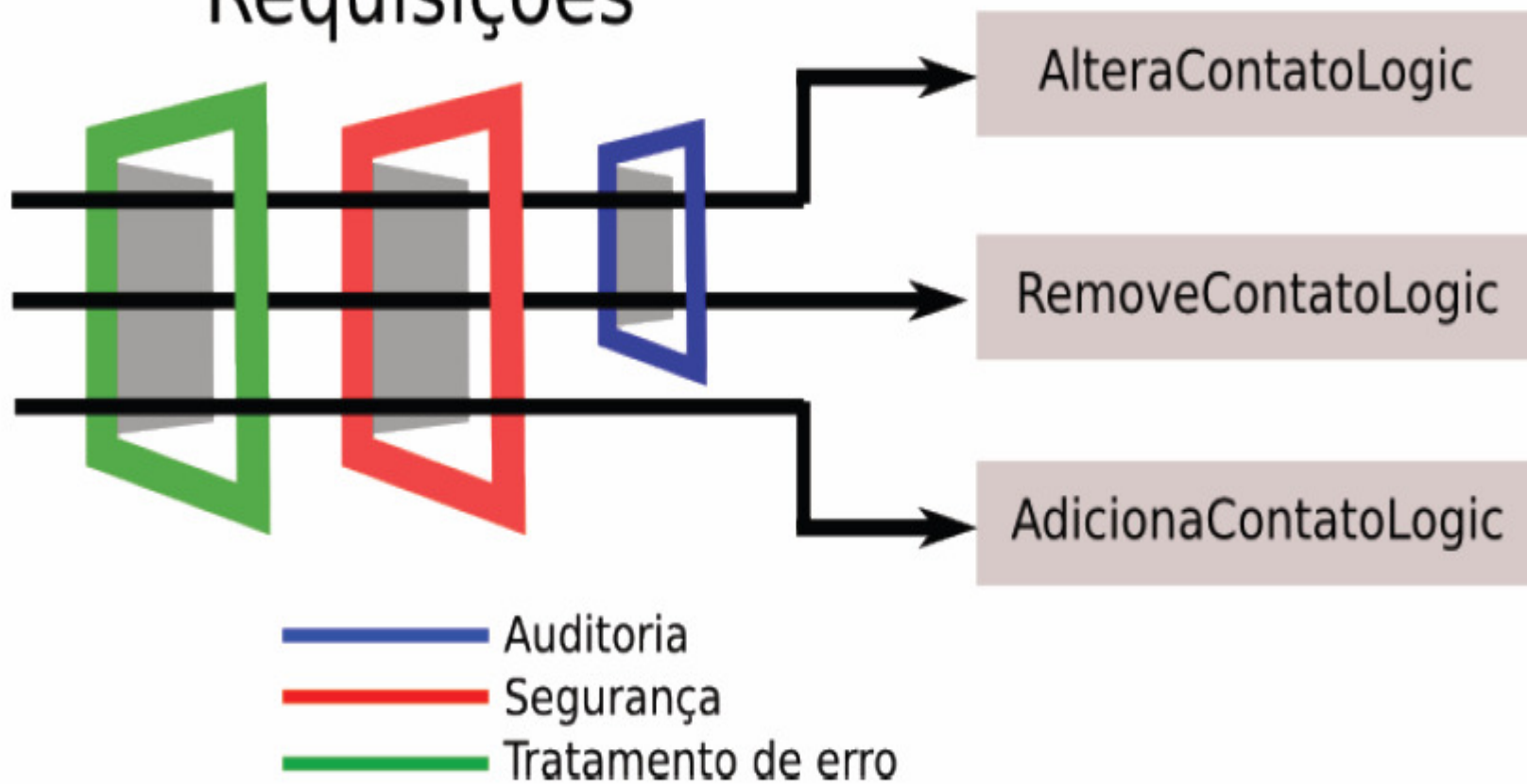
AdicionaContatoLogic

Auditoria  
Segurança  
Tratamento de erro

- 
- ▶ A API de Servlets nos provê um mecanismo para tirar esse acoplamento e isolar esse comportamento, que são os Filtros. Filtros são classes que permitem que executemos código antes da requisição e também depois que a resposta foi gerada.
  - ▶ Uma analogia é pensar que as lógicas são quartos em uma casa. Os filtros seriam portas. Cada filtro encapsula apenas uma responsabilidade, um para fazer auditoria, outro para fazer a segurança, etc. → Veja Fig.



## Requisições



- 
- ▶ A grande vantagem é que cada requisito fica em um lugar só, conseguimos tirar o acoplamento dentro das lógicas.
  - ▶ Para criarmos um filtro, basta criarmos uma classe que implemente a interface `javax.servlet.Filter`. Ao implementar essa interface, temos que implementar 3 métodos: `init`, `destroy` e `doFilter`.
  - ▶ Os métodos `init` e `destroy` possuem a mesma função dos métodos de um `Servlet`. O método que fará todo o processamento que queremos executar é o `doFilter` → recebe os parâmetros `ServletRequest`, `ServletResponse` e `FilterChain`.



---

```
public class FiltroConexao implements Filter {  
    // implementação do init e destroy  
    public void doFilter(ServletRequest request,    ServletResponse  
        response, FilterChain chain)  
        throws IOException, ServletException {  
        //todo o processamento vai aqui  
    }  
}
```

- ▶ Um filtro processa requests como os servlets, mas de maneira mais genérica para vários tipos de requests. Mais um filtro vai além de um servlet, com um filtro podemos também fechar “a porta” → FilterChain – Permite indicar ao container que o request deve prosseguir seu processamento.
- 



---

```
public void doFilter(ServletRequest request, ServletResponse response,  
                    FilterChain chain)  
    throws IOException, ServletException {  
    //passa pela porta  
    chain.doFilter(request, response);  
}
```

- ▶ Um filtro intercepta vários requests semelhantes, executar alguma coisa, mas depois permitir que o processamento normal do request prossiga através das Servlets e JSPs normais.
- ▶ Qualquer código antes da chamada chain.doFilter será executado na ida, qualquer código depois na volta. Com isso podemos fazer uma verificação de acesso antes da lógica, ou abrir um recurso (conexão ou transação) antes e na volta fechar o mesmo → fazer tratamento de erro ou medir o tempo de execução.





- 
- ▶ Precisamos, para que funcione, registrá-lo, assim o container saberá que ele precisa ser executado → web.xml

```
<filter>  
  <filter-name>FiltroTempoDeExcecucacao</filter-name>  
  <filter-class>br.com.caelum.agenda.filtro.FiltroTempoDeExcecucacao</filter-class>  
</filter>
```

```
<filter-mapping>  
  <filter-name>FiltroTempoDeExcecucacao</filter-name>  
  <url-pattern>/*</url-pattern>  
</filter-mapping>
```

---

- Precisamos, para que funcione, registrá-lo, assim o container saberá que ele precisa ser executado → web.xml

```
<filter>
  <filter-name>FiltroTempoDeExcecucacao</filter-name>
  <filter-
class>br.com.caelum.agenda.filtro.FiltroTempoDeExcecucacao</filter-
class>
</filter>
<filter-mapping>
  <filter-name>FiltroTempoDeExcecucacao</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Todas as requisições serão filtradas. Será aplicado em cada requisição

\*.jsp mais específico. Só requisições para páginas JSPs

# Exercício: Filtro para medir o tempo de execução

---

1. Fazer filtro para medir o tempo de execução de uma requisição.
  - a. Criar uma classe chamada FiltroTempoDeExecução no pacote meuPacote.agenda.filtro e faça ela implementar a interface javax.servlet.Filter.
  - b. Deixar os métodos init e destroy vazios e implemente o doFilter → Veja Slide.
  - c. Declarar o filtro no web.xml, para fazer com que todas as requisições passem por ele → Veja Slide
  - d. Reiniciar o servidor e acesse: [http://localhost:8080/fj21\\_agenda/altera\\_contato.jsp](http://localhost:8080/fj21_agenda/altera_contato.jsp). Procurar a saída no console.



---

```
public void doFilter(ServletRequest request, ServletResponse response,
                    FilterChain chain)
    throws IOException, ServletException {
    long tempoInicial = System.currentTimeMillis();
    chain.doFilter(request, response);
    long tempoFinal = System.currentTimeMillis();
    System.out.println("Tempo da requisição em millis: " + (tempoFinal - tempoInicial));
}
```

```
<filter>
    <filter-name>FiltroTempoDeExcecucacao</filter-name>
    <filter-class>br.com.caelum.agenda.filtro.FiltroTempoDeExcecucacao</filter-class>
</filter>
<filter-mapping>
    <filter-name>FiltroTempoDeExcecucacao</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

---



# Problemas na criação de conexões

---

- ▶ A aplicação de agenda precisa em vários momentos, de uma conexão com o BD. O nosso DAO invoca em seu construtor a `ConnectionFactory` pedindo para a mesma uma nova conexão. Mas e o fechamento da conexão?

```
public class ContatoDAO {  
    private Connection connection;  
    public ContatoDAO() {  
        this.connection = new ConnectionFactory().getConnection();  
    }  
    // métodos adiciona, remove, getLista etc  
    // onde fechamos a conexão?  
}
```

- 
- ▶ Péssima prática deixar conexões abertas.

## Tentando outras estratégias

---

- ▶ Não é uma boa idéia colocarmos a criação da conexão no construtor dos nossos DAOs. Mas qual é o lugar ideal?
- ▶ Precisamos, de alguma forma, criar a conexão e fazer com que essa mesma conexão possa ser usada por todos os DAOs em uma determinada requisição.

```
public class ContatoDAO {  
    private Connection connection;  
    public ContatoDAO(Connection connection) {  
        this.connection = connection;  
    }  
    // métodos adiciona, remove, getLista etc
```

---

```
}
```

---

```
public class AdicionaContatoLogic implements Logica {  
    public void execute(HttpServletRequest request,  
        HttpServletResponse response) {  
        Contato contato = //contato montado com os dados do request  
        Connection connection = new  
        ConnectionFactory().getConnection();  
        // passa conexao pro construtor  
        ContatoDAO dao = new ContatoDAO(connection);  
        dao.adiciona(contato);  
        connection.close();  
    }  
}
```

- ▶ Não é uma solução muito boa. Acoplamos com a ConnectionFactory todas as nossas lógicas que precisam utilizar DAOs



# Reduzindo o acoplamento com Filtros

---

- ▶ Para diminuirmos esse acoplamento, queremos que, sempre que chegar uma requisição para a nossa aplicação, uma conexão seja aberta e, depois que essa requisição for processada, ela seja fechada. Precisamos então interceptar toda requisição para executar esses procedimentos.
- ▶ Os Filtros seria ideais → Veja código
- ▶ Com a conexão aberta, precisamos então fazer com que a requisição saia do nosso filtro e vá para o próximo passo, seja ele um outro filtro, ou um Servlet ou um JSP.
- ▶ Conseguimos abrir uma conexão no começo dos requests, prosseguir o processamento do request e fechar a conexão após a execução. Mas nossas lógicas vão executar manipulações de Contatos e vão precisar da conexão aberta no filtro. Como acessá-la? Como, dentro de uma Servlet pegar um objeto criado dentro de um filtro, uma outra classe?



---

```
public class FiltroConexao implements Filter {  
    // implementação do init e destroy, se necessário  
    public void doFilter(ServletRequest request, ServletResponse  
response,                               FilterChain chain)  
                                throws IOException, ServletException {  
        //abre uma conexão  
        Connection connection = new  
ConnectionFactory().getConnection();  
        // indica que o processamento do request deve prosseguir  
        chain.doFilter(request, response);  
        //fecha conexão  
        connection.close();  
    }  
}
```

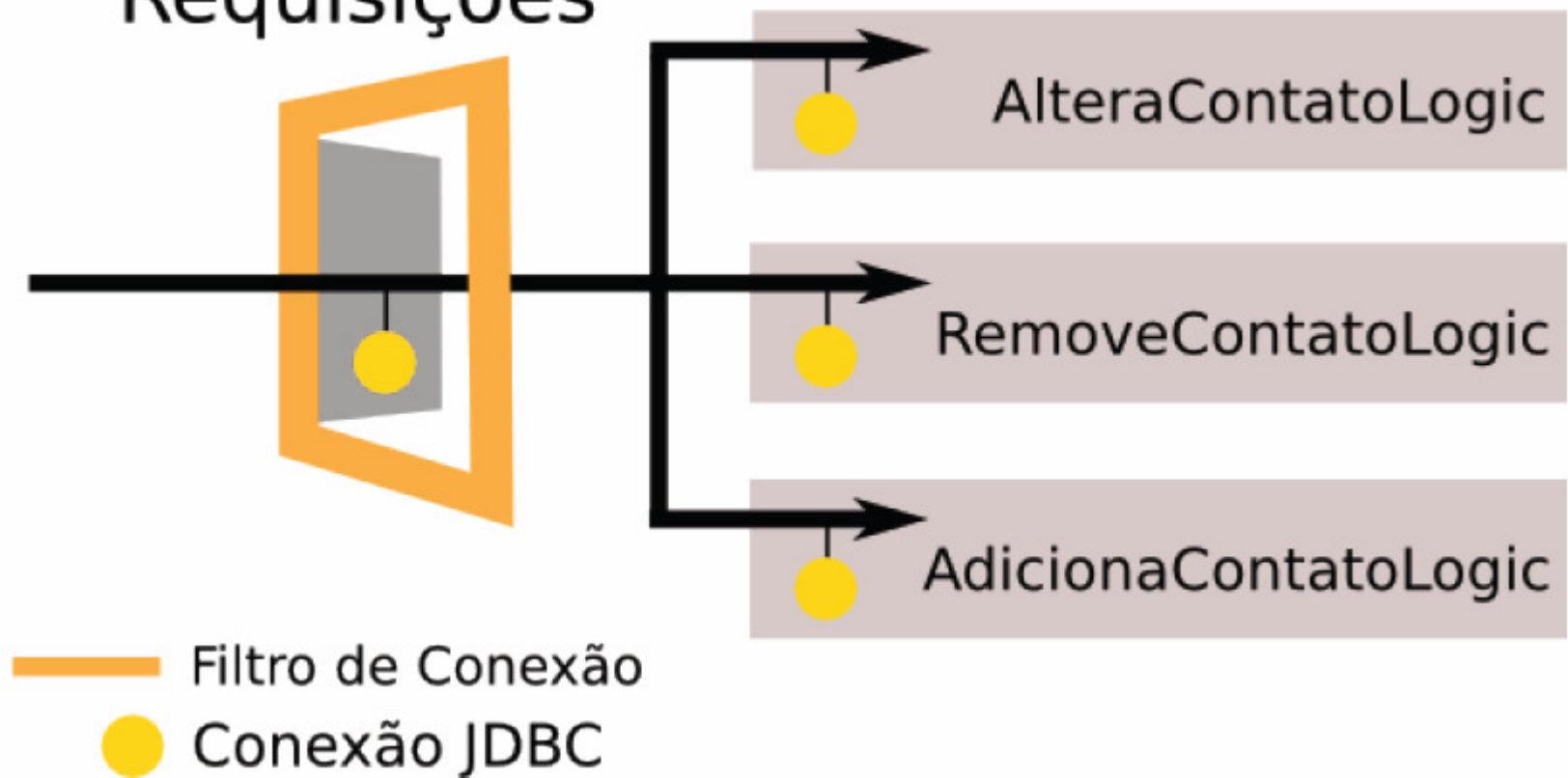
---



- 
- ▶ A idéia é associar (pindurar) de alguma forma a conexão criada ao request atual. Tanto o filtro quanto o servlet estão no mesmo request. Veja Figura.
  - ▶ Para guardarmos algo na requisição, precisamos invocar o método `setAttribute` no request. Passamos para esse método uma identificação para o objeto que estamos guardando na requisição e também passamos o próprio objeto para ser guardado no request → Veja código.



## Requisições



---

```
public void doFilter(ServletRequest request, ServletResponse
    response, FilterChain chain) {
    Connection connection = new ConnectionFactory().getConnection();
    // "pendura um objeto no Request"
    request.setAttribute("connection", connection);
    chain.doFilter(request, response);
    connection.close();
}
```

- ▶ Ao invocarmos o doFilter, a requisição seguirá o seu fluxo normal levando o objeto connection junto.
- ▶ Portanto, o Filtro é o único ponto da nossa aplicação que criará conexões. Agora é necessário registrá-lo



---

<filter>

<filter-name>FiltroConexao</filter-name>

<filter-class>br.com.caelum.agenda.filtro.FiltroConexao</filter-class>

</filter>

<filter-mapping>

<filter-name>FiltroConexao</filter-name>

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

</filter-mapping>

- ▶ Agora só falta saber como pegar a conexão que guardamos no request. Para isso basta invocarmos o método `getAttribute` no request.



---

```
public class AdicionaContatoLogic implements Logica {  
    public void execute(HttpServletRequest request, HttpServletResponse  
        response)  
        throws Exception {  
        Contato contato = // contato montado com os dados do request  
        // buscando a conexao do request  
        Connection connection = (Connection)  
request.getAttribute("connection");  
        ContatoDAO dao = new ContatoDAO(connection);  
        dao.adiciona(contato);  
        // faz o dispatch para o JSP como de costume  
    }  
}
```

---



## Exercícios: Filtros

---

1. Vamos criar o nosso filtro para abrir e fechar a conexão com o BD.
  - a. Crie uma nova classe chamada FiltroConexão no pacote meuPacote.agenda.Filtro e faça ela implementar a interface javax.servlet.Filter
  - b. Deixe os métodos init e destroy vazios e implemente o doFilter → Veja próxima Transparência
  - c. Declare o filtro no web.xml, para fazer com que todas as requisições passem por ele. Isso foi feito acima.
2. Crie um construtor no seu ContatoDAO que receba Connection e armazene-a no atributo → Veja transparência.



---

```
public void doFilter(ServletRequest request, ServletResponse
    response,
                    FilterChain chain)
    throws IOException, ServletException {
    try {
        Connection connection = new
        ConnectionFactory().getConnection();
        request.setAttribute("connection", connection);
        chain.doFilter(request, response);
        connection.close();
    } catch (SQLException e) {
        throw new ServletException(e);
    }
}
```

---





---

```
public class ContatoDAO {  
    private Connection connection;  
    public ContatoDAO(Connection connection) {  
        this.connection = connection;  
    }  
    //outro construtor e métodos do DAO  
}
```


3. Agora na sua conta AlteraContatoLogica criada acima, busque a conexão no request, e repasse-a para o DAO  
→ Veja próxima transparência.
4. Altere um contato já existente na sua aplicação e verifique que tudo continua funcionando normalmente.



---

```
public class AlteraContatoLogic implements Logica {  
    public void execute(HttpServletRequest request, HttpServletResponse  
        response)  
        throws Exception {  
        // buscando a conexao do request  
        Connection connection = (Connection)  
request.getAttribute("connection");  
        // busca dados do request  
        // faz conversão e monta objeto contato  
        Contato contato = // ....  
        ContatoDAO dao = new ContatoDAO(connection);  
        dao.atualiza(contato);  
        // Faz o dispatch para o JSP  
    }  
}
```

---



# Struts 2

---

- ▶ Porque utilizar frameworks
- ▶ Como funciona o framework Struts 2
- ▶ As diferentes formas de se trabalhar com Struts 2.



# Porque precisamos de frameworks MVC?

---

- ▶ Quando estamos desenvolvendo aplicações, em qualquer linguagem queremos nos preocupar com infraestrutura o mínimo possível.
- ▶ Quando trabalhamos com containers e a API de Servlets existe muito trabalho repetitivo que precisamos fazer para que possamos desenvolver a nossa lógica.
- ▶ Frameworks MVC, surgem para diminuir o impacto da API de Servlets em nosso trabalho e fazer com que passemos nos preocupar exclusivamente com a lógica de negócios → valor para a aplicação.



## Um pouco de história

---

- ▶ Logo que percebeu que o trabalho com Servlets e JSPs puros não era tão produtivo e organizado. Sun começou fomentar o uso do padrão MVC e de patterns com Front Controller.
- ▶ Empresas implementaram suas soluções baseadas em mini-frameworks caseiros → muito retrabalho de projeto para projeto.
- ▶ O Struts foi um dos primeiros frameworks MVC com a idéia de se criar um controlador reutilizável entre projetos. Lançado no 2000.



## Um pouco de história

---

- ▶ Entretanto, hoje em dia ele demanda muito trabalho, por ter sido criado quando não existiam muitas das facilidades da linguagem Java.
- ▶ Para resolver isso, a comunidade do Struts uniu-se com outro framework → WebWork. Foi desenvolvido Struts 2 → versão mais simples de se trabalhar do que Struts 1, e com ainda mais recursos e funcionalidades.
- ▶ Permite uma migração suave de Struts 1.



## Configurando o Struts 2

---

- ▶ Para que possamos aprender o Struts 2, vamos criar um sistema de lista de tarefas. O primeiro que devemos fazer é ter o Struts 2 → <http://struts.apache.org/2.x/>
- ▶ Uma vez que adicionarmos os JARs do Struts 2 em nosso projeto dentro WEB-INF/lib, precisamos declarar o Filtro, que fará o papel de Front Controller da nossa aplicação, recebendo as requisições e as enviando às lógicas corretas. Para declararmos o Filtro do Struts 2, basta adicionarmos no web.xml, o seguinte →



---

<filter>

    <filter-name>struts2</filter-name>

    <filter-class>

        org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter

    </filter-class>

</filter>

<filter-mapping>

    <filter-name>struts2</filter-name>

    <url-pattern>/\*</url-pattern>

</filter-mapping>

- 
- ▶ Repare como é uma configuração normal de filtro, como as que
  - ▶ fizemos antes.



# Criando as lógicas

---

- ▶ Existem diversas abordagens seguidas pelos frameworks para possibilitar a criação de lógicas. Declaração em um arquivo XML, e, a partir do Java 5, surgiram as anotações, que são uma forma de introduzir metadados dentro de nossas classes.
  - ▶ Struts 2 permite criar lógicas de 3 formas:
    - ▶ Declaração em um arquivo XML → como Struts 1
    - ▶ Utilizando convenções: não é necessário herdar nem escrever nenhum XML, desde que a sua lógica seja criada em um pacote pré definido pelo framework
    - ▶ Utilizando anotações: foi uma forma introduzida pelo WebWork. Conseguido através do uso do plugin de convenções.
- 



# A lógica Olá Mundo!

---

- ▶ Nossa primeira lógica → `OlaMundoAction`. Criar uma classe com esse nome. O sufixo `Action` é convenção em Struts. Como utilizaremos o plugin de convenções, é obrigatório que a sua `Action` esteja dentro de um subpacote com um dos nomes: `struts2`, `action` ou `actions`. No nosso caso, criaremos a classe dentro do pacote: `meuPacote.tarefas.action`.
- ▶ Dentro dessa nossa nova classe, vamos criar um método que imprimirá algo no console e, em seguida, irá redirecionar para um JSP com a mensagem “Olá Mundo!”. Podemos criar também um método de qualquer nome dentro dessa classe, desde que ele esteja com a anotação `@Action`.



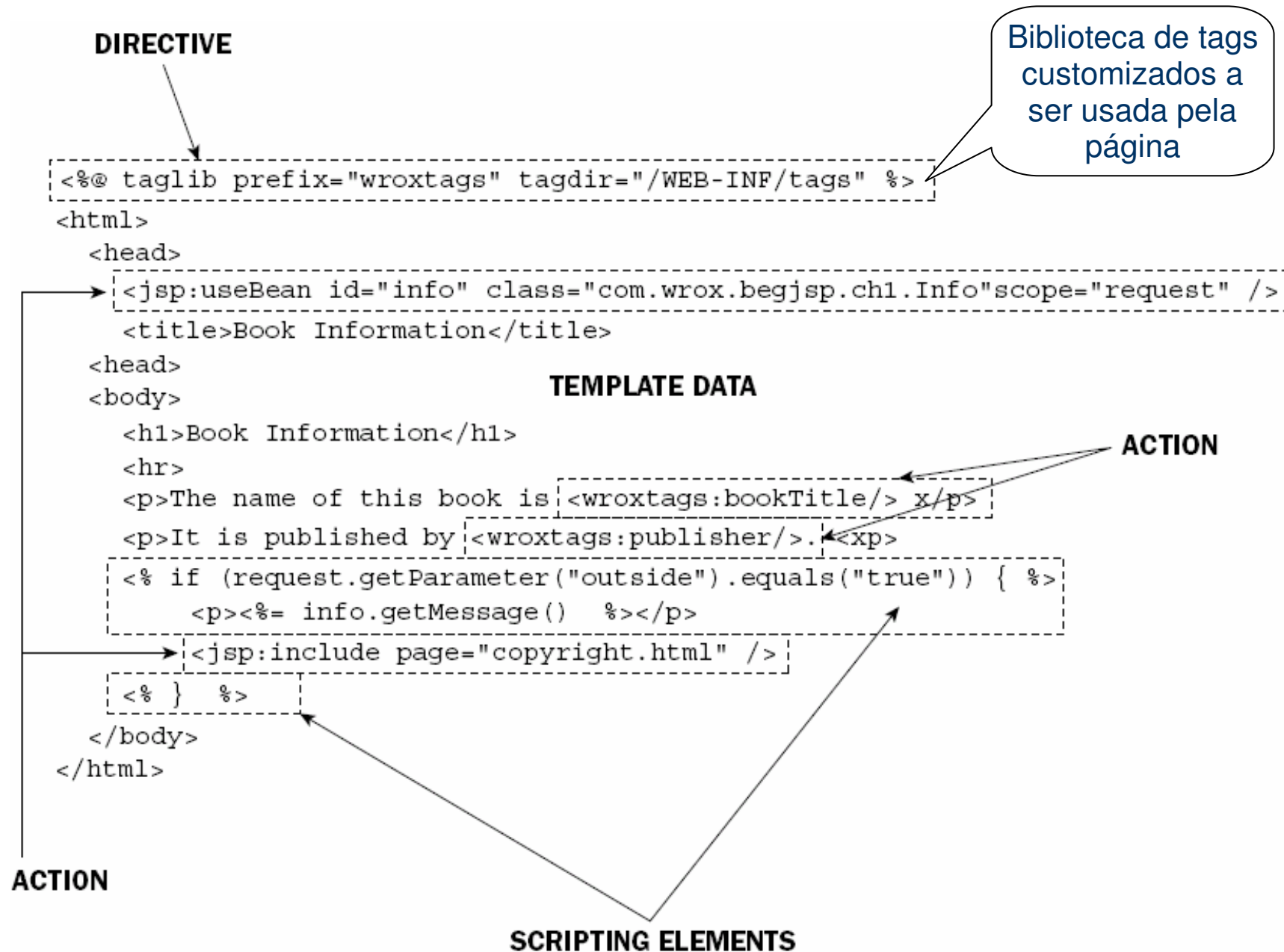
## A lógica Olá Mundo!

---

- ▶ A anotação `@Action` recebe um atributo chamado `value` que indica qual será a URL utilizada para invocar a Action.
- ▶ Se colocarmos o valor `olaMundoStruts` acessaremos nossa Action pela URL <http://localhost:8080/fj21-tarefas/olaMundoStruts>.
- ▶ Uma outra convenção que é criar um método chamado `execute`. Esse método deve retornar uma `String` que indica qual JSP deve ser executado após a lógica.



# Anatomia de uma página JSP



# Anatomia de uma página JSP

---

- ▶ Diretivas: diferencia-se dos outros elementos porque não gera saídas diretamente. Elas são usadas para controlar algumas características da página. Diretivas permitidas:
  - ▶ Diretivas de página
  - ▶ Diretivas taglib
  - ▶ Diretivas include

`<%@ taglib .... %>` não compatível com XML →  
`<jsp>directive.taglib .... />`
- ▶ Elas fornecem instruções ao contenedor sobre como processar JSP.



# Anatomia de uma página JSP

---

- ▶ Dados de template: texto estático. Não processado pelo contenedor JSP. HTML estático.
- ▶ Ação: elementos envolvidos no processamento da solicitação. Facilitam acessar dados e manipular ou transformar dados na geração dinâmica da saída. Classificados em: estándares e customizados.
- ▶ Elementos de script: incorporação de código numa linguagem de programação (Java). Três elementos diferentes:
  - ▶ Declarações: declaração de variáveis e métodos em java
  - ▶ Scriptlets: segmentos arbitrários de código java
  - ▶ Expressões: expressões java que conduzem a um valor resultante.



*Declarations* are Java code that is used to declare variables and methods. They appear as follows:

```
<%! ... Java declaration goes here... %>
```

The XML-compatible syntax for declarations is:

```
<jsp:declaration> ... Java declaration goes here ... </jsp:declaration>
```

*Scriptlets* are arbitrary Java code segments. They appear as follows:

```
<% ... Java code goes here ... %>
```

The XML-compatible syntax for scriptlets is:

```
<jsp:scriptlet> ... Java code goes here ... </jsp:scriptlet>
```

*Expressions*:

```
<%= ... Java expression goes here ... %>
```

The XML-compatible syntax for expressions is:

```
<jsp:expression> ... Java expression goes here ... </jsp:expression>
```

## Um outro exemplo de JSP

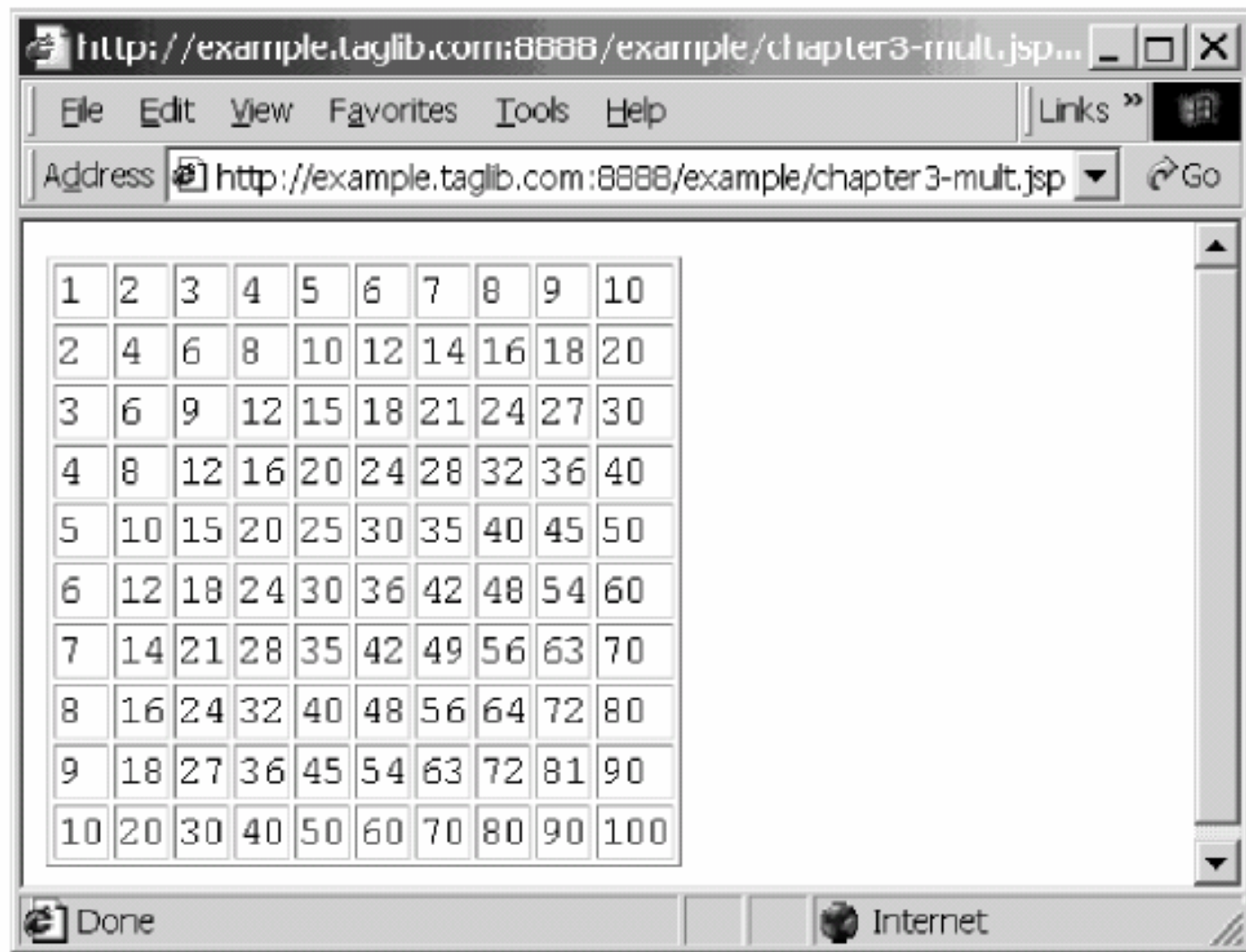
---

```
<table border="1">
<% for (int row = 1; row < 11; row++) { %>
  <tr>
    <% for (int column = 1; column < 11; column++) { %>
      <td><tt><%= row * column %></tt></td>
    <% } %>
  </tr>
<% } %>
</table>
```





# Resultado



The image shows a screenshot of a web browser window. The title bar indicates the URL is `http://example.taglib.com:8888/example/chapter3-mult.jsp...`. The address bar shows the same URL. The main content area displays a 10x10 multiplication table. The status bar at the bottom shows 'Done' and 'Internet'.

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

**Figure 3.2** A multiplication table printed in a web browser

# Um exemplo misturando Servlets

---

```
<% String name = request.getParameter("name"); %>
<html>
<body>
<p>
<% if (name != null) { %>
    Hello, <%= name %>.
<% } else { %>
    Welcome, anonymous user.
<% } %>
You're accessing this servlet
from <%= request.getRemoteAddr() %>.
</p>
</body>
</html>
```



# Manutenção do Estado

---

- ▶ HTTP é stateless
- ▶ Vantagens
  - ▶ Fácil de usar: não precisa nada
  - ▶ Bom para aplicações com informação estática
  - ▶ Não precisa de espaço em memória adicional
- ▶ Desvantagens
  - ▶ Nenhum registro de solicitações anteriores
    - ▶ Impossível carrinho de compras
    - ▶ Impossível logins de usuários
    - ▶ Nenhum conteúdo dinâmico ou customizado
    - ▶ Segurança é mais difícil de implementar



# Estado da aplicação

---

- ▶ Estado do lado do servidor
  - ▶ Informação é armazenada num BD, ou na memória local da camada de aplicação
- ▶ Estado do lado do cliente
  - ▶ Informação é armazenada no computador do cliente na forma de uma cookie
- ▶ Estado escondido
  - ▶ Informação é escondida dentro dentro das páginas web criadas dinamicamente



---

# Estado do lado do servidor

## Várias formas

### 1. Armazenar informação num BD

- ▶ Dados estarão seguros no BD
- ▶ MAS: precisa de acessos ao BD para consultar e atualizar a informação

### 2. Usar a memória local da camada de aplicação

- Pode mapear o endereço IP do usuário para algum estado
- MAS: a informação é volátil e pode exigir muita memória principal do servidor

---

5 milhões de IPs = 20 MB



---

## Estado do lado do servidor

1. Deveria usar manutenção do estado do lado do servidor para informação que precisa persistir
  - ▶ Pedidos de clientes antigos
  - ▶ Seqüências de click de um usuário se movimentando num “site”
  - ▶ Escolhas permanentes que um usuário faz



---

## Estado no lado do cliente: Cookies

- ▶ Armazenando texto sobre o cliente que será passado à aplicação com toda a solicitação HTTP.
  - ▶ Pode ser desabilitado pelo cliente.
  - ▶ São erradamente percebidos como perigosos, e vários usuários evitam visitar sites que usem estas facilidades
  - ▶ São um coleção de pares (Nome, Valor)



---

# Estado do lado do cliente: Cookies

- ▶ **Vantagens**

- ▶ Fácil de usar em Java Servlets/ JSP
- ▶ Oferece uma forma simples para que persistam dados não essenciais no cliente mesmo quando o cliente (browser) tem fechado

- ▶ **Desvantagens**

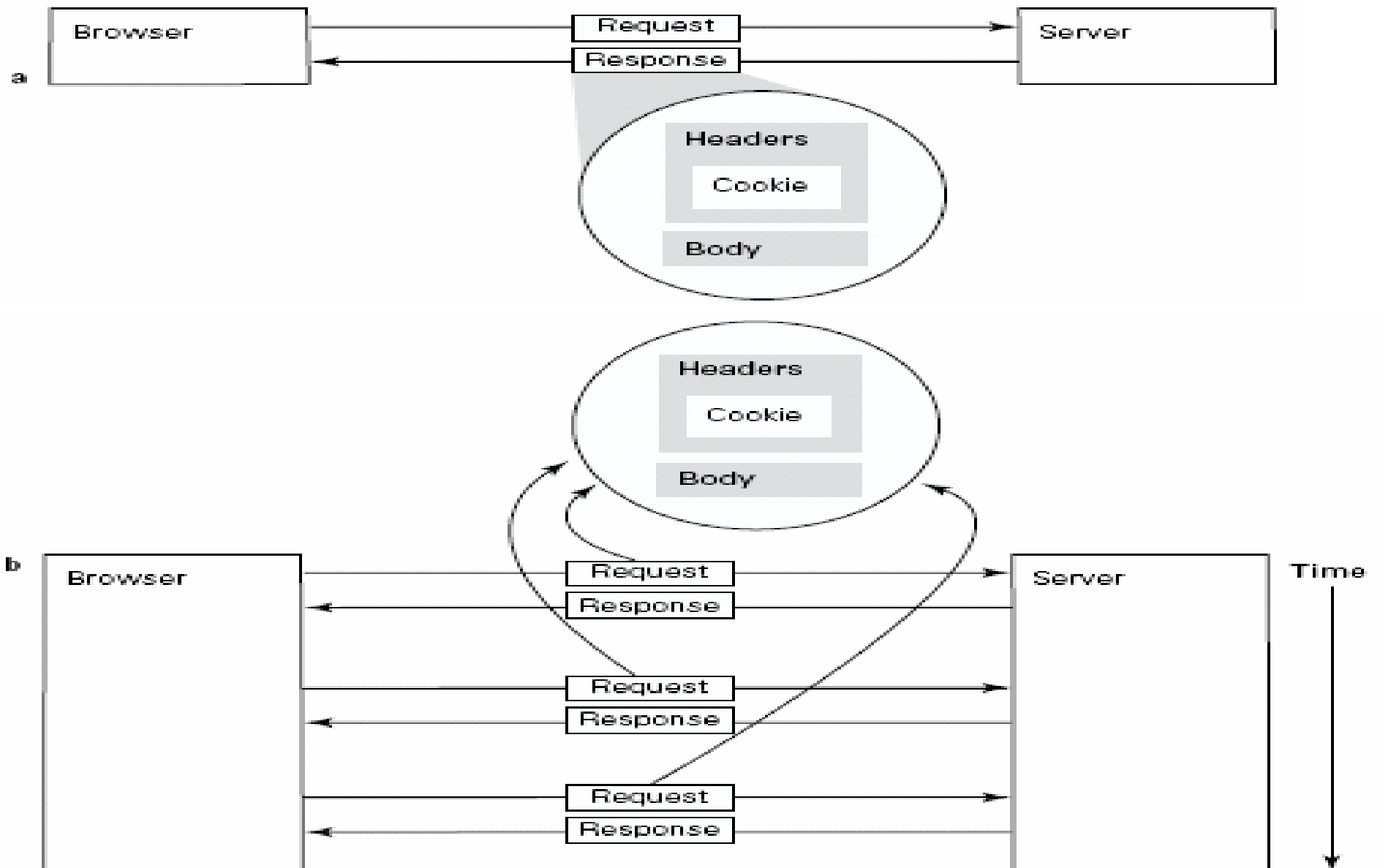
- ▶ Limitado a 4Kbytes de informação
- ▶ Usuários podem (e freqüentemente o fazem) desabilitar eles

- ▶ **Os cookies deveriam ser usados para armazenar o estado interativo**

- ▶ A informação de login do usuário atual
- ▶ O carrinho de compras atual
- ▶ Qualquer escolha não permanente que o usuário tenha feito





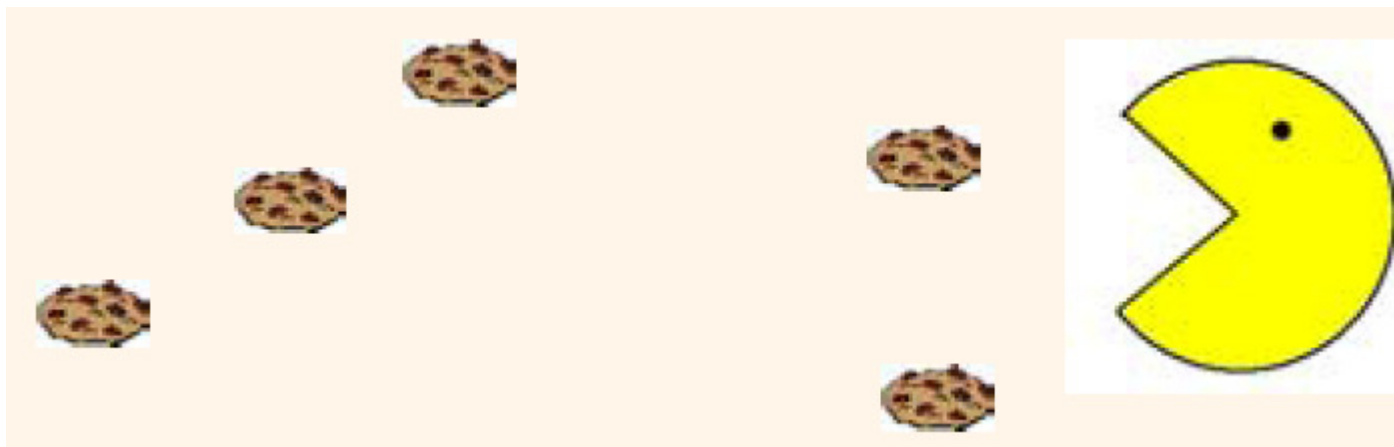


**Figure 4.4** Setting and using an HTTP cookie. Step 1, a cookie is sent to the browser as part of the response headers (a). Step 2, once the cookie is sent, it is sent back by the browser automatically for all requests in the cookie's scope. The server does not subsequently request the cookie; it gets it automatically (b).

# Criando um Cookie

```
Cookie myCookie =  
    new Cookie("username", "jeffd");  
response.addCookie(userCookie);
```

Você pode criar cookies em qualquer momento.



---

## Acessando um Cookie

```
Cookie[] cookies = request.getCookies();
String theUser;
for(int i=0; i<cookies.length; i++) {
    Cookie cookie = cookies[i];
    if(cookie.getName().equals("username")) theUser =
        cookie.getValue();
}
// at this point theUser == "username"
```



---

## Características dos Cookies

- ▶ Os cookies podem ter
  - ▶ Uma duração (expira ou persistem mesmo depois que o browser tem fechado)
  - ▶ Filtros para os quais as trajetórias de domínio/diretórios o cookie é enviado
- ▶ Ver os tutorias de APIS de Java Servlets e Servlets.



---

## Estados ocultos

- ▶ Com freqüência os usuários desabilitam os cookies.
- ▶ Você pode “ocultar” dados em dois lugares:
  - ▶ Campos ocultos dentro de um formulário
  - ▶ Usando a informação do caminho
- ▶ Não precisa de armazenamento de informação porque a informação do estado é enviado dentro de cada página web



---

## Estados ocultos: Campos ocultos

- ▶ Declarar campos ocultos dentro de um formulário:
  - ▶ `<input Type='hidden' name='user' value='username' />`
- ▶ Usuários não enxergarão esta informação (a menos que eles vejam o fonte HTML)
- ▶ Se for usado prolificamente, ele deteriora o desempenho, já que TODA página deve ser contida dentro de um formulário.



---

## Estado oculto: informação do caminho

- ▶ A informação do caminho é armazenado na solicitação de URL:  
`http://server.com/index.htm?user=jeffd`
- ▶ Pode separar 'campos' com caracter &:  
`index.htm?user=jeffd&preference=pepsi`
- ▶ Existem mecanismos para analisar estes campos em java. Verifique o método  
`javax.servlet.http.HttpUtilsparserQueryString()`



---

# Métodos de múltiplos estados

- ▶ Tipicamente todos os métodos de manutenção de estado são usados:
  - ▶ O usuário se “loga” e esta informação é armazenada em um cookie
  - ▶ O usuário emite uma consulta a qual é armazenada na informação do caminho
  - ▶ O usuário coloca um item em um cookie carrinho de compras
  - ▶ Compras de um usuário e informação do cartão de crédito é armazenada/recuperada de um BD.
  - ▶ O usuário deixa uma “seqüência de clicks” a qual é mantida no log do servidor web (o qual é analisado mais tarde)

