



Java EE: Aplicações Web

Prof. Emilio Celso de Souza



Sumário

Apresentação	5
Módulo 1 - Elementos de uma aplicação Java Web	6
Aula 1 – A Internet, o protocolo HTTP, Servidores e Aplicações Web	6
O nascimento da WWW e da Internet	6
O Protocolo HTTP	6
Servidores Web	9
Aplicações Web	11
Aula 2 – Criando um projeto Java Web no Eclipse	18
Configurando o servidor Apache Tomcat	18
Criando o projeto no Eclipse	20
Criando e executando um Servlet	25
Exercícios do Módulo 1	31
Módulo 2 - Servlets	32
Aula 3 – Conceitos básicos sobre Servlets	32
Definição de Servlets	32
Localização das classes de um servlet	33
Criação de Servlets – Até a versão 2.5 (Java EE 5.0)	33
Estrutura do Servlet	34
Aula 4 - Ciclo de vida do Servlet	35
Métodos do ciclo de vida de um Servlet	35
Principais métodos da requisição	37
Transferência de requisições	38
Principais métodos da resposta	39
Redirecionamento de Servlets	40
Aula 5 - Criação de Servlets – versão 3.0	41
Anotações em Servlets	41
Considerações sobre mapeamento do servlet	44
ServletConfig	44
ServletContext	45
Aula 6 - Gerenciamento de Sessão	46
Definição de gerenciamento de sessão	46
Modelo do controle de sessão	47
A classe HttpSession	48
Campos ocultos	49
Reescrita de URL	50
Escopo de armazenamento de dados na memória do servidor	50
Exercícios do Módulo 2	52
Módulo 3 - Java Server Pages (JSP)	56
Aula 7 – Fundamentos de JSP	56
Definição de JSP	56
Arquitetura JSP	57
Diretivas	57
Formulários	58
Aula 8 - Objetos implícitos no JSP	60
Definição de objetos implícitos	60

· Declarações de variáveis	61
· Expressões ou Scriptlets	61
· Ações	62
· Passando parâmetros para uma página JSP via URL	63
Aula 9 - JavaBeans	67
· Bean	67
· Atribuindo valores para as propriedades	68
· Obtendo valores das propriedades	69
Aula 10 - JSTL	69
· Definição de JSTL	69
· Expression Language (EL)	69
· Bibliotecas de tags	71
· Biblioteca Core	71
· Biblioteca Sql	77
· Biblioteca Format	80
· Biblioteca XML	83
Exercícios do Módulo 3	85
Módulo 4 - Conceitos do Framework JSF 2.0	92
Aula 11 – JavaServer Faces	92
· Definição de JSP e Framework	92
· A arquitetura MVC (Model–View–Controller)	93
· O MVC de JSF	94
· Exemplo de uma aplicação JSF	95
Aula 12 - Visão Geral das Tags JSF	99
· Tags JSF Fundamentais	99
· Principais Tags HTML JSF	100
· Anotações em beans gerenciáveis	101
· Escopos de beans gerenciáveis	102
· Navegação condicional	103
· Elementos do JSF	103
Aula 13 - Ciclo de vida do JSF	104
· Fases do ciclo de vida do JSF	104
Aula 14 – Eventos JSF	106
· ValueChange Event	106
· Action Event	108
· Phase Event	108
Aula 15 - Usando conversores padrão	110
· Conversão de números, datas e textos em JSF	110
· Uso de propriedades e o arquivo de propriedades	112
Exercícios do Módulo 4	114
Módulo 5 – Web services	119
Aula 16 – Conhecendo os Web services	119
· Definição de Web services	119
Aula 17 - SOAP	121
· Definição de SOAP	121

Estrutura de um arquivo SOAP	121
Atributos SOAP	123
Aula 18 - WSDL	124
Definição de WSDL	124
Estrutura WSDL	124
WSDL ports	125
Bindings SOAP	125
Aula 19 - Conceitos de Web services JAX-WS	125
Definição de JAX-WS	125
Vinculando WSDL a Java com JAXB	125
Processamento Síncrono	133
Processamento Assíncrono	134
Considerações Finais	138
Respostas Comentadas dos Exercícios	139
Exercícios do Módulo 1	139
Exercício1: Calculadora	139
Exercícios do Módulo 2	143
Exercício 1: Formulário para validação de usuário	143
Exercício 2: Definição do servlet para receber os dados do formulário	145
Exercício 3: Armazenamento de dados na sessão	149
Exercícios do Módulo 3	151
Exercício 1: Definição do banco de dados	151
Exercício 2: Camada de acesso a dados	151
Exercício 3: Definição da aplicação	155
Exercícios do Módulo 4	169
Exercício 1: Para desenvolver este exercício, siga estes passos:	169
Exercício 2	172
Referências Bibliográficas	179

Apresentação

Caros alunos, nesta disciplina investigaremos outra vertente importante do mundo Java: os conceitos de Java Web. Para tanto, torna-se imprescindível um bom conhecimento e domínio do que foi estudado até a disciplina que trata do Java SE Avançado.

Estudaremos os conceitos de desenvolvimento Web em Java, incluindo o mecanismo de requisição e resposta, os componentes que são executados no servidor, como a resposta é entregue ao cliente, além dos detalhes de como escrever códigos para os dois lados: cliente e servidor.

Aqui, cliente é o browser e servidor é o local em que a aplicação está hospedada. Por exemplo, suponha que você deseja acessar ao site o banco no qual você mantém uma conta corrente. O acesso é realizado a partir de um navegador através de uma interface gráfica desenhada, na sua maior parte, por HTML. Essas informações são enviadas a um servidor, que as recebe e as processa, gerando uma resposta que pode ser um extrato, um comprovante de pagamento, dentre outras.

Em muitos casos, os sistemas necessitam de informações de outro sistema, geralmente desenvolvido em outra linguagem. Para esse tipo de acesso são necessários os Web services, ou seja, elementos que trocam informações com outros sistemas externos sem se importarem em qual plataforma esses sistemas foram desenvolvidos.

Além dos conceitos da linguagem Java, o mecanismo de acesso ao banco de dados é fundamental nesse contexto Web. O banco de dados pode fazer parte do mesmo servidor em que a aplicação web está hospedada, ou em outro servidor, exclusivo para os dados.

Teremos, nesta disciplina, uma grande quantidade de códigos nos exercícios, pois não se trata mais de simples programas escritos em uma única classe, mas de diversas classes e códigos HTML, interagindo entre si.

Desejo um excelente aproveitamento, e como sempre, estamos à disposição para auxiliá-los no que for preciso. Mas não se esqueçam: dedicação é fundamental, pois o mercado está carente de bons profissionais e está de portas abertas, só aguardando seu ingresso!

Prof. Emilio

Módulo 1 - Elementos de uma aplicação Java Web

Aula 1 – A Internet, o protocolo HTTP, Servidores e Aplicações Web

O nascimento da WWW e da Internet

O conceito de *Internet* surgiu nos anos 1960 e início da década de 1970 quando se vivia o auge da Guerra Fria e os laboratórios militares americanos sentiam a necessidade de compartilhar de forma segura informações sigilosas, armazenadas em computadores espalhados pelo país. Foi criada, então, uma rede de comunicação militar interligando esses computadores. Para evitar que um ataque nuclear soviético interrompesse essa comunicação, desenvolveram um esquema para a transmissão em que as informações seriam divididas em pacotes. Esses pacotes tomariam caminhos diferentes para chegar ao mesmo local. Caso um trecho de comunicação fosse destruído, os pacotes pegariam outro caminho e chegariam ao mesmo destino. Foi assim que surgiu a ARPANET, o antecessor da *Internet*.

Do âmbito militar o projeto passou para o campo acadêmico com o objetivo de:

[...] coligar universidades para que fosse possível uma transmissão de dados de forma mais eficaz, rápida e segura. No Brasil, a *internet* iniciou no final da década de 1980 quando no Laboratório Nacional de Computação Científica (LNCC), localizado no Rio de Janeiro, conseguiu acesso à Bitnet, através de uma conexão de 9 600 bits por segundo, estabelecida com a Universidade de Maryland (TOTALIZE, 2014).

Com o surgimento da WWW (World Wide Web) na década de 1990, a *Internet* avançou rapidamente alcançando quase todos os países do planeta, disseminando e interligando informações e pessoas por todo o mundo (MULLER, 2011).

O Protocolo HTTP

Podemos definir “http” da seguinte forma:

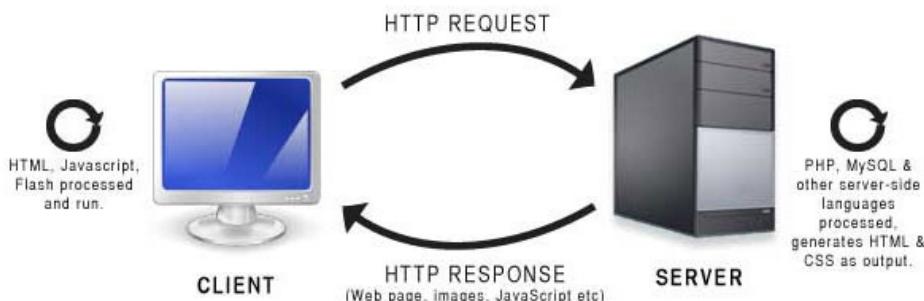
HTTP - *HyperText Transfer Protocol* é um protocolo de comunicação utilizado para transferência de páginas HTML do computador para a WWW na Internet. Por isso, os endereços dos websites (URLs) utilizam no início a expressão «`http://`», definindo o protocolo usado. Esta informação é necessária para estabelecer a comunicação entre a URL e o servidor Web que armazena os dados, enviando então a página HTML solicitada pelo usuário.

Fonte: <<http://www.significados.com.br/http/>>. Acesso em: 28 ago. 2014.

O funcionamento do protocolo HTTP pode ser visto na Figura 1.2a, e possui as características:

- Baseado em Request-Response;
- Cliente abre conexão e envia *request* para o servidor;
- Cliente recebe *response* e fecha conexão;
- Comunicação sem estado (stateless);
- Para o servidor cada *request* é o primeiro *request* enviado pelo cliente;
- Cada recurso é identificado por único e exclusivo Uniform Resource Identifier (URI);
- URL (Uniform Resource Locator) são URIs que especificam protocolos de *internet* (HTTP, FTP, mail-to);

Figura 1.2a: Mecanismo de requisição e resposta



Fonte: <<http://demosthenes.info/blog/137/The-ClientServer-Model>>. Acesso em: 28 ago. 2014.

As partes de uma mensagem HTTP são:

- Linha inicial: Especifica o propósito da mensagem ou *response*;
- Seção Header: Especifica meta-informações, tais como tamanho, tipo e *encoding*, sobre o conteúdo da mensagem;
- Linha em branco ;
- Corpo da mensagem;
- O conteúdo principal do *request* ou *reponse*.

A primeira linha de um HTTP *request* tem três partes, separadas por espaço:

- Nome do método;
- O *path* do recurso solicitado (URI);
- A versão do HTTP sendo utilizado.

```
GET /hello/HelloApp?userid=Joao HTTP/1.1
```

Os métodos do protocolo HTTP são:

- **Método GET:** Obtém informações do servidor – usualmente um documento – especificando qual documento e eventuais parâmetros através de um URI (*Universal Resource Identifier*). O método GET é usado para requisitar informações do servidor, mas pode ser empregado no envio de um conjunto dos dados do formulário junto com a URL (endereço e nome da aplicação especificada pelo

atributo *action*). Os dados são separados da URL pelo caracter "?" e são agrupados por nome e valor, separados entre si pelo caracter "&".

- **Método HEAD:** Obtém informações sobre um determinado documento no servidor, mas não necessariamente o documento em si.
- **Método POST:** Envia informações para o servidor processar ou armazenar. O método POST envia o conjunto de dados do formulário para a aplicação que irá fazer o tratamento como um bloco de dados separado da requisição. Esse método é utilizado quando o bloco de dados ultrapassa o tamanho máximo da URL ou quando não é seguro anexar os dados no final da URL.
- **Método OPTIONS:** Por meio desse método o cliente obtém as propriedades do servidor.
- **Método DELETE:** Informa por meio do URL o objeto a ser deletado.
- **Método TRACE:** Para enviar mensagem do tipo *loopback* para teste.
- **Método PUT:** Aceita criar ou modificar algum objeto do servidor.
- **Método CONNECT:** Comunicar com servidores Proxy.

Os dois principais métodos do *request* são **GET** e **POST**.

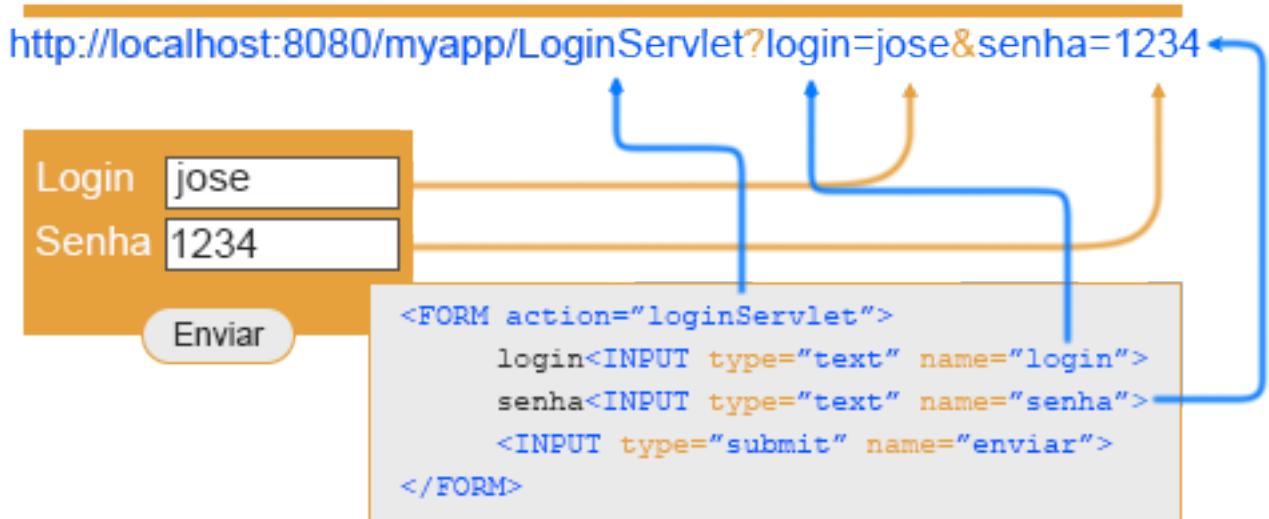
A passagem de parâmetros é realizada como mostra o Quadro 1, e é ilustrado na Figura 1.2b.

Quadro 1: Requisição a um recurso web via URL

<http://localhost:1234/hello/HelloServlet?userid=10&nome=Ze>

- Cada parâmetro tem um identificador, seguido pelo sinal de igual e valor;
- Cada par parâmetro/valor é separado pelo caracter &;
- Todos os parâmetros são separados da URL através do caracter ? (somente quando utilizado GET);
- Toda URL é *case sensitive*.

Figura 1.2b: Passagem de parâmetros via método GET



O método GET é ilustrado na Figura 1.2c e:

- Envia dados através da URI;
- É menos seguro;
- Pode navegar dados de um *site* para outro.

Figura 1.2c: Cabeçalho do método GET

```

Header { GET /hello/HelloServlet?userid=Joao HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/x-shockwave-flash, /*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Host: 127.0.0.1
Connection: Keep-Alive
Linha em branco }
  
```

O método POST é ilustrado na Figura 1.2d e:

- Envia dados através do fluxo da requisição;
- É mais seguro;
- Dados navegam somente no mesmo contexto.

Figura 1.2d: Cabeçalho do método POST

```

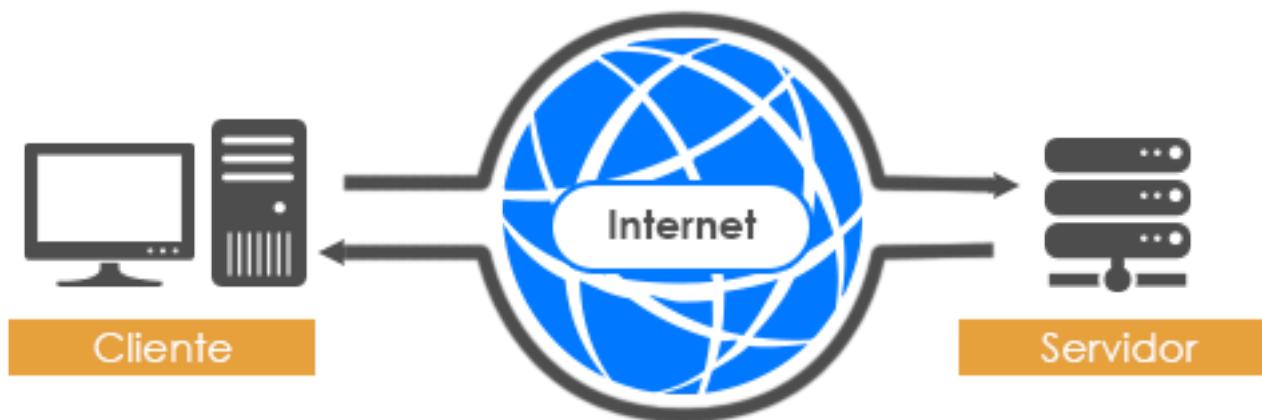
Header { POST /hello/HelloServlet HTTP/1.1
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/x-shockwave-flash, /*
Referer: http://localhost:1234/hello/hello.html
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Host: 127.0.0.1
Content-Length: 12
Connection: Keep-Alive
Cache-Control: no-cache
Linha em branco }
  
```

Dados { userid=Joao }

Servidores Web

Servidores Web são softwares que ficam em execução em uma máquina permanentemente conectada à Internet, preparados para se comunicar usando o protocolo TCP/IP. Ficam monitorando uma porta (usualmente a porta 80) em um ou mais endereços IP, aguardando conexões. Ao receber uma conexão, o servidor analisa os dados recebidos através da conexão, em busca de uma requisição HTTP, e então processa a requisição e, por meio da mesma conexão, retorna o resultado.

O servidor (no nosso caso, o Tomcat ou o GlassFish) recebe informações do browser, processa as informações e retorna uma resposta no formato HTML para o cliente (*browser*), como mostra a Figura 1.3a.

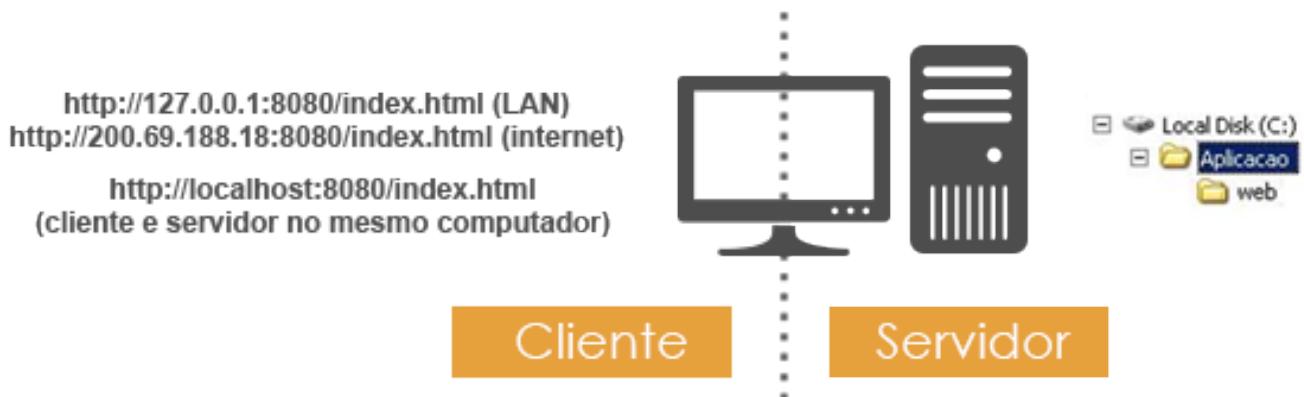
Figura 1.3a: Requisição e resposta

No servidor existe uma estrutura de diretórios devidamente configurada. O acesso externo (cliente) aos arquivos nas pastas desse diretório é realizado através de uma URL, definida para apontar para aquela estrutura de pastas. A Figura 1.3b ilustra este processo.

Figura 1.3b: Requisição e resposta – cliente e servidor em diferentes computadores

É comum, para fins de testes ou de desenvolvimento, executar um programa no servidor, sendo o servidor o próprio computador cliente. Ou seja, não é necessário ter uma infraestrutura complexa para testar uma aplicação, como mostra a Figura 1.3c.

Figura 1.3c: Requisição e resposta – cliente e servidor no mesmo computador



Aplicações Web

Há dois tipos de aplicações web:

- **Orientadas à apresentação:** Páginas interativas contendo linguagens tais como HTML, XML, DHTML, XHTML etc .
- **Orientadas a serviços:** Acesso aos serviços externos, como os *Web services*.

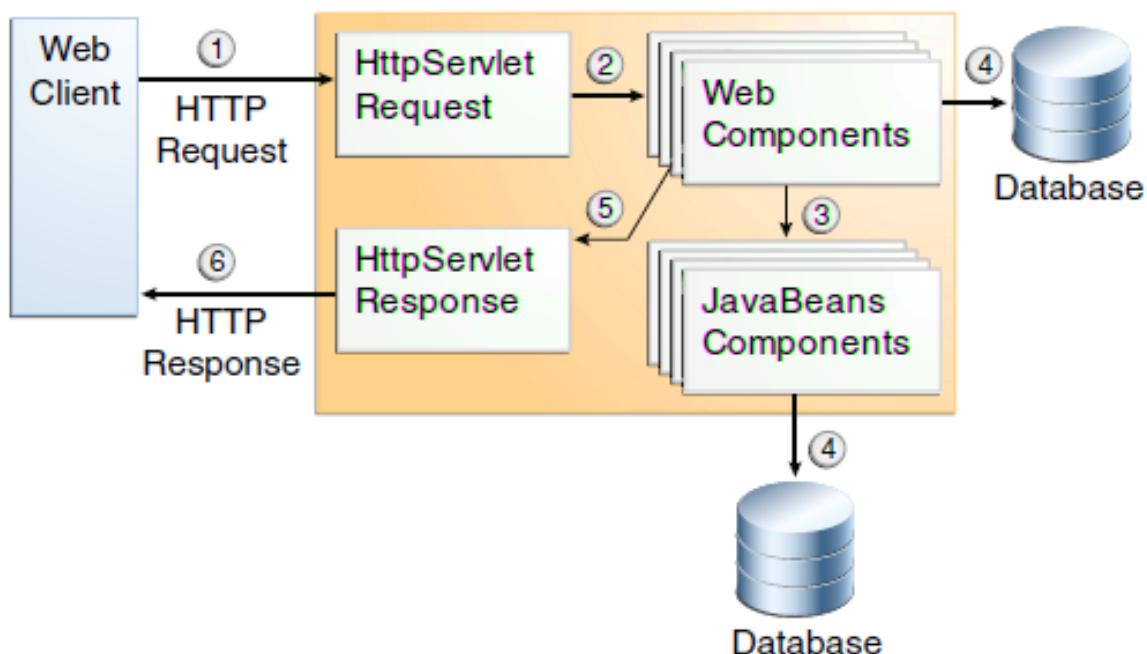
Na plataforma Java, componentes *web* fornecem funcionalidades dinâmicas para um servidor *web*. Os componentes *web* são:

- Java *Servlets*;
- Páginas JSP;
- Web *services*.

A *iteração* entre o cliente e o servidor ocorre de acordo com os passos descritos a seguir:

1. O cliente envia uma requisição HTTP para o servidor.
2. O servidor converte a requisição para um objeto **HttpServletRequest**.
3. O componente *web* recebe o **HttpServletRequest** e interage com componentes **JavaBeans**.
4. Os componentes **JavaBeans** interagem com um banco de dados para gerar o conteúdo dinâmico.
5. O conteúdo dinâmico é processado e um **HttpServletResponse** é gerado.
6. Servidor *web* converte **HttpServletResponse** para uma resposta HTTP e a retorna para o cliente *web*.

Esses passos estão ilustrados na Figura 1.4a.

Figura 1.4a: Requisições em aplicações Web

Fonte: <<http://docs.oracle.com/javaee/6/tutorial/doc/geysj.html>>. Acesso em: 28 ago. 2014.

As aplicações Web essencialmente são constituídas por *Servlets* e páginas JSP:

- **Servlets:** São classes Java que processam requisições dinamicamente e constroem a resposta. Recomendados para controlarem a camada de apresentação (**controller**) de aplicações orientadas à apresentação e para implementar serviços em aplicações orientadas aos serviços (*web services* são normalmente implementados como *servlets*).
- **Páginas JSP:** Documentos texto que são executados como *servlets*. Recomendadas para páginas baseadas em HTML, WML, DHTML, *Web Designers*.

Mesmo alguns *frameworks*, como **JavaServer Faces** ou **Struts**, são constituídos por *Servlets*.

Todos os componentes Web devem estar dentro de um *container Web*. Um *container Web* fornece serviços tais como:

- Encaminhamento de requests (request dispatching);
- Segurança;
- Acesso concorrente;
- Gerenciamento do ciclo de vida dos componentes web;
- Controle de transações;
- *E-mail*;
- etc.

Componentes *Web* (*Servlets*, *JSPs*) e componentes estáticos (imagens, arquivos) são chamados de recursos *web* que devem ser empacotados dentro de um módulo *web*. Um módulo *web* é a menor unidade que pode ser instalada (deployed) em um servidor *Web*. Um módulo *web* corresponde a uma aplicação *web*.

Módulo Web

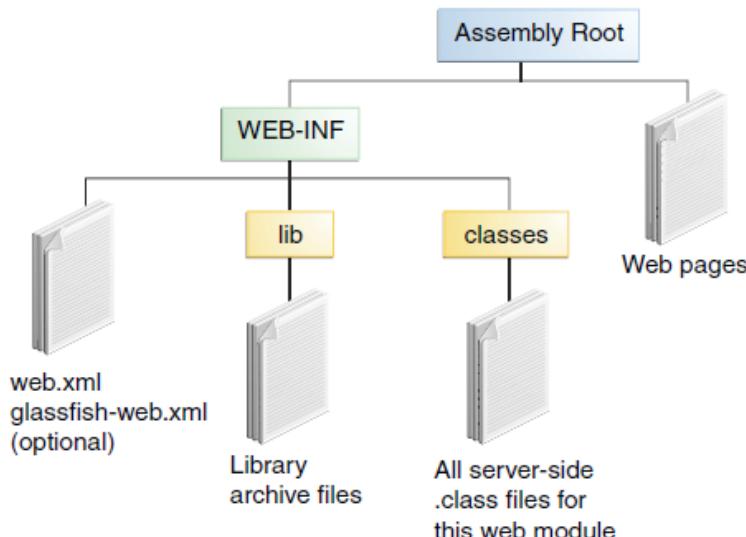
Um módulo *Web* possui uma estrutura específica. O diretório inicial é a raiz da aplicação, em que *JSPs*, *applets*, classes utilitárias e recursos estáticos (*html*, imagens) são armazenados.

O módulo *Web* possui um subdiretório chamado WEB-INF, no qual são armazenados:

- **web.xml** – Descritor da aplicação;
- **tld** – Tag library descriptor;
- **classes servidoras** – *servlets*, *javabeans* etc.;
- **lib** – Diretório que contém jar's de bibliotecas.

A Figura 1.4.1 ilustra a estrutura de um módulo *Web*.

Figura 1.4.1 Estrutura do Módulo *Web*



Fonte: <<http://docs.oracle.com/javaee/6/tutorial/doc/bnadx.html>>. Acesso em: 28 ago. 2014.

Se a sua aplicação não possui nenhum *servlet*, filtro ou *listener*, então não precisa de um descritor *web.xml*. Um módulo *web* pode ser instalado (deployed) com um pacote WAR ou como uma estrutura de diretórios e arquivos desempacotados.

Contexto de uma aplicação e o arquivo WEB.XML

Cada aplicação possui seu próprio contexto, que consiste em um espaço em memória e em disco reservado para cada aplicação e uma área exclusiva para cada aplicação.

- No disco, o contexto recebe o nome da própria aplicação, o pacote **.war** ou o nome da pasta inicial (root) da aplicação.
- Cada aplicação possui acesso somente ao seu próprio contexto.
- Dentro do contexto podem ser armazenados parâmetros e atributos visíveis por todos os componentes, independentemente de sessão de usuário.
- Um contexto é implementado pelo *container* através da interface **javax.servlet.ServletContext**

A Figura 1.4.2 mostra um exemplo de um contexto.

Figura 1.4.2: Estrutura do módulo Web - diretórios



Quando criamos um novo projeto *web*, de uma forma geral o nome do projeto representa o nome do contexto, embora seja possível alterar esse nome.

O contexto é representado por um arquivo XML presente em uma pasta específica do servidor, com as informações de localização da aplicação. Os passos para definição do contexto são:

- Definir a estrutura de pastas no servidor;
- Escolher um nome adequado para o contexto;
- Escrever um arquivo xml, com o mesmo nome do contexto;
- Salvar o arquivo na pasta **[TOMCAT_HOME]/conf/Catalina/localhost**.

Exemplo:

Estrutura de pastas: **C:\Aplicacao\web**

Nome do contexto: **app**

Nome do arquivo: **app.xml**

Conteúdo do arquivo app.xml:

O Quadro 2 apresenta a configuração de um contexto.

Quadro 2: Configuração de um contexto no servidor

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<Context path="\app" docBase="C:/Aplicacao/web" />
  
```

O acesso a um arquivo (HTML, JSP etc.) localizado no nosso novo contexto é acessado, então, da seguinte forma:

Arquivo: **ínicio.jsp**

Pasta: **C:\Aplicacao\web**

Contexto: **app**

O Quadro 3 apresenta o acesso ao contexto.

Quadro 3: Acesso a um recurso web usando o contexto configurado

<http://localhost:8080/app/ínicio.jsp>

Existe um arquivo web.xml para cada contexto definido para as aplicações. Este arquivo é chamado de descritor de implantação do aplicativo Web e deve estar na pasta WEB-INF.



No exemplo da estrutura de pasta anterior, o arquivo servirá para as páginas JSP usadas na aplicação e para as classes referentes aos servlets.

Atenção: Caso contenha algum erro de descrição do XML a aplicação não será iniciada.

Esse arquivo era obrigatório para uma aplicação Web até a versão 2.5 do Servlet. A partir do Servlet 3.0 (JEE 6.0), a maior parte das configurações dispensaram o uso desse arquivo.

As partes do arquivo são descritas no que se segue.

O Quadro 4 configura parâmetros para o contexto da aplicação.

Quadro 4: Configuração de um parâmetro de contexto

```
<context-param>
    <param-name>qtd</param-name>
    <param-value>1</param-value>
</context-param>
```

O Quadro 5 configura um *servlet*, criando também um parâmetro de inicialização (Servlet 2.5 e anterior).

Quadro 5: Configuração de um parâmetro de *servlet*

```
<servlet>
    <servlet-name>Cadastrar</servlet-name>
    <servlet-class>com.fiap.servlet.Cadastrar</servlet-class>
    <init-param>
        <param-name>tipo</param-name>
        <param-value>2</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Quadro 6: Configuração do mapeamento de um *servlet* (Servlet 2.5 e anterior)

```
<servlet-mapping>
  < servlet-name>Cadastrar</servlet-name>
  < url-pattern>/Cadastrar</url-pattern>
</servlet-mapping>
```

Quadro 7: Configuração do tempo de vida de um *se[rv]let* (timeout de sessão do usuário)

```
<session-config>
  < session-timeout>60</session-timeout >
</session-config>
```

Quadro 8: Configuração de página de erro padrão para uma aplicação

```
<error-page>
  <error-code>404</error-code>
  <location>/erro.jsp</ location>
</error-page >
```

Quadro 9: Configuração da página inicial de uma aplicação (as páginas se-
rão chamadas automaticamente ao chamar o contexto).

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

A Tabela 1.4.2a apresenta uma breve descrição das principais *tags* usadas no arquivo *web.xml*.

Tabela 1.4.2a: Breve descrição das principais *tags* utilizadas no arquivo *WEB.XML*

Elemento	Descrição
web-app	Raiz do descritor de implantação.
display-name	Nome curto do aplicativo
servlet-name	O nome oficial do <i>servlet</i> . Deve ser exclusivo.
servlet-class	Nome da classe totalmente qualificado da <i>servlet</i> .
init-param	Parâmetros de início disponíveis para o <i>servlet</i> . Isso é seguido de pares de parâmetros nome/valor.
param-name	Nome do parâmetro.
param-value	Valor do parâmetro.

servlet-mapping	Usado para mapear um <i>servlet</i> em uma URL.
session-config	Define o comportamento de interrupção para sessões.
error-page	Usado para mapear um código de <i>status</i> de erro de http em um recurso da WEB , como uma página HTML, que será apresentada em lugar das páginas de erro padrão do navegador.
error-code	Especifica um código de erro para a página definida em <error-page>.
welcome-file-list	Usado para apresentar uma lista de arquivos que serão usados como página inicial de uma aplicação. Normalmente seu nome começa com index .
welcome-file	Especifica o nome do arquivo inicial da aplicação. Alguns exemplos são: index.htm, index.html, index.jsp etc.

Os principais códigos de erro usados como conteúdo de <error-code> (elemento filho de <error-page>) estão summarizados na Tabela 1.4.2b.

Tabela 1.4.2b: Principais códigos de erro do arquivo WEB.XML

Código	Erro	Descrição
400	Bad Request	Requisição inválida. O servidor detectou um erro de sintaxe na requisição.
401	Unauthorized	Não autorizada. A requisição não tinha autorização correta.
403	Forbidden	Proibida. A requisição foi negada, motivo desconhecido
404	Not found	O documento não foi encontrado.
500	Internal Server Error	Erro interno do servidor. Normalmente, indica que parte do servidor (provavelmente seu <i>servlet</i>) está corrompida.
501	Not Implemented	O servidor não pode executar a ação requisitada por esta não estar implementada.

Um exemplo completo do descriptor de aplicativo está mostrado no Quadro 10.

Quadro 10: Exemplo de um arquivo web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
  version="2.5">

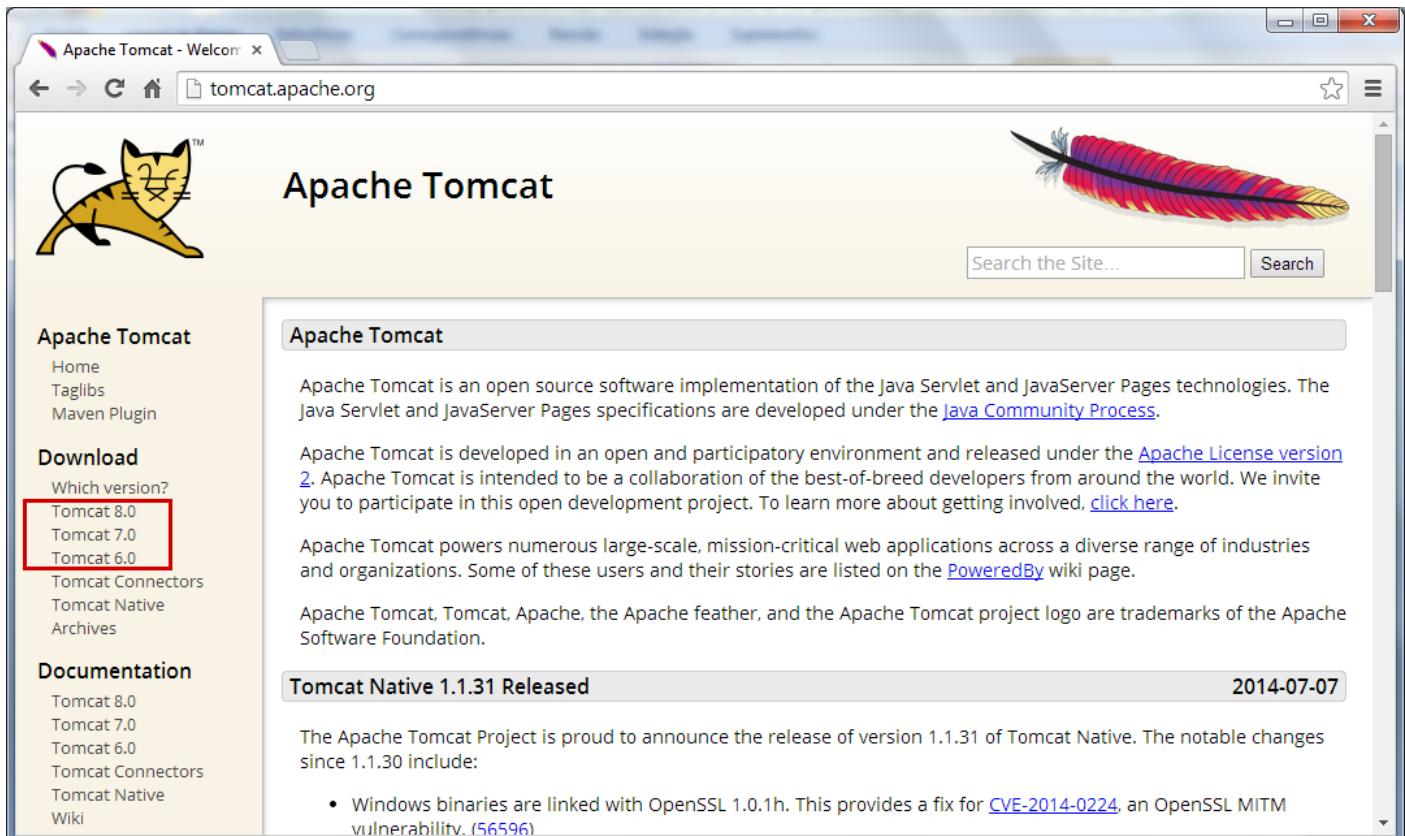
<servlet>
  <servlet-name>Servlet01</servlet-name>
    <servlet-class>fiap.si.Servlet01</servlet-class>
    <init-param>
      <param-name>titulo</param-name>
      <param-value>JSP</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>Servlet01</servlet-name>
    <url-pattern>/portalfiap</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
  <error-page>
    <error-code>404</error-code>
    <location>/erro404.jsp</location>
  </error-page >
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>

```

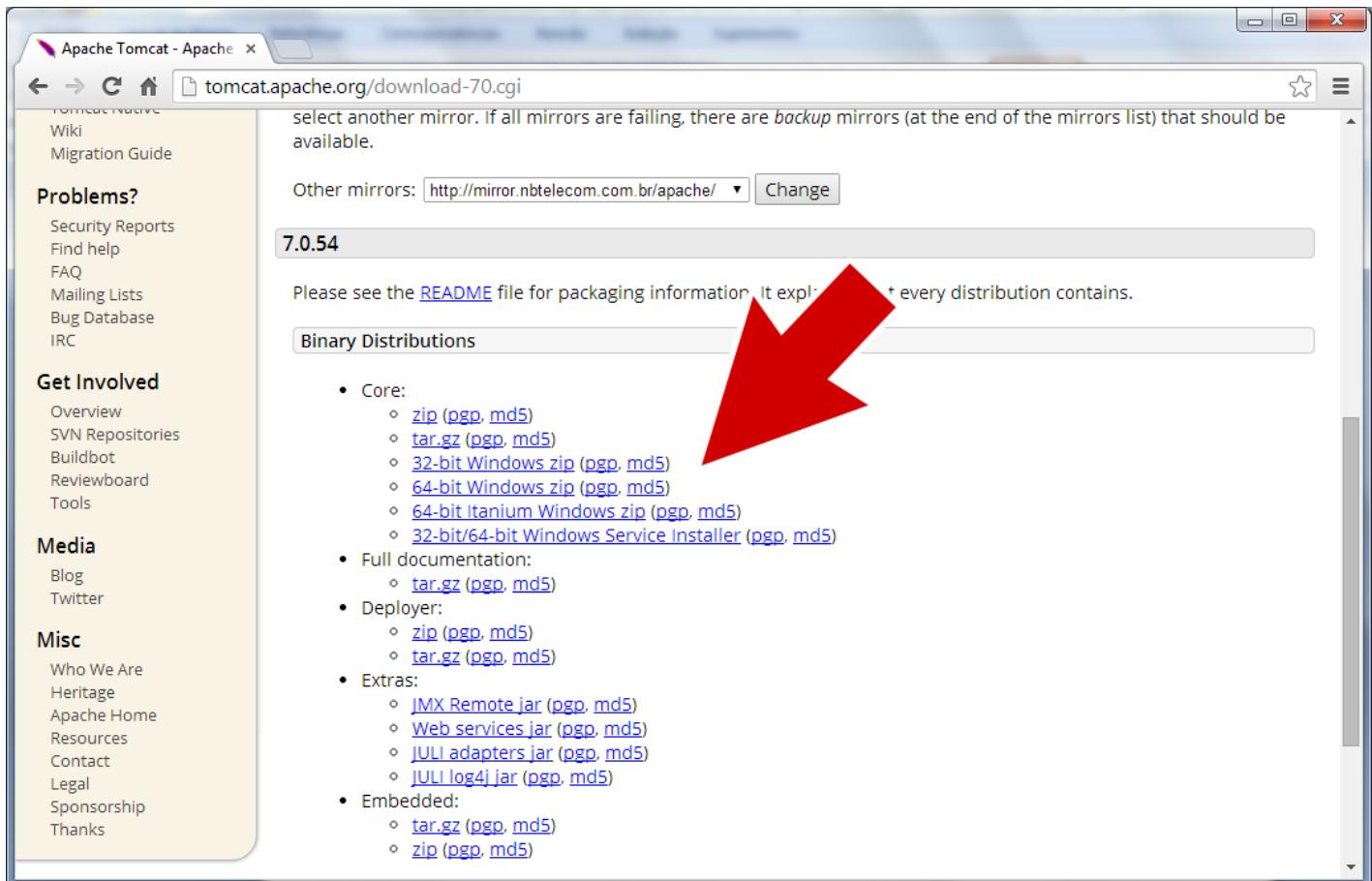
Aula 2 – Criando um projeto Java Web no Eclipse

Configurando o servidor Apache Tomcat

Caros alunos, aqui aprenderemos a criar um projeto *web* no Eclipse. Precisaremos de um servidor de aplicações para hospedar nossas aplicações e escolheremos o servidor Tomcat por ser suficiente para nossos propósitos, além de ser bastante leve. Esse servidor deve ser obtido por *download* no site <<http://tomcat.apache.org/>>. A Figura 2.1a mostra a página na qual o *download* deve ser realizado. Nesta página, você deve selecionar Tomcat 7.0 no menu do lado esquerdo.

Figura 2.1a: Portal do Apache Tomcat

Clicando na opção Tomcat 7.0 chegamos às opções de *download*, conforme mostrado na Figura 2.1b. Escolher a opção “zip” indicada.

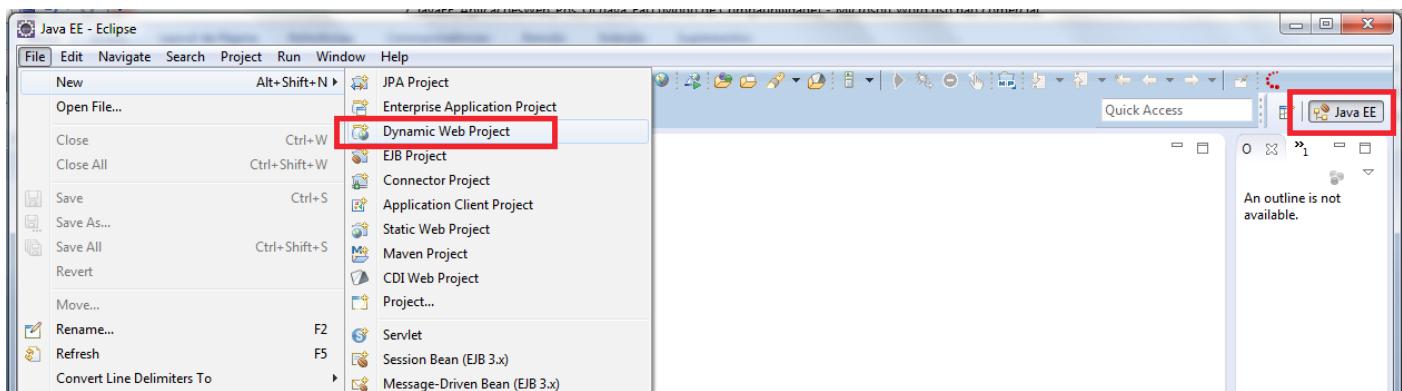
Figura 2.1b: Opções de download do Tomcat

Fonte: <<http://tomcat.apache.org/download-70.cgi>>. Acesso em: 28 ago. 2014.

Esse arquivo deve ser descompactado em uma pasta de sua preferência. Nesta disciplina usaremos a pasta **C:\Servidor** como local para o servidor. Vamos descompactar o arquivo **apache-tomcat-7.0.42.zip** nesta pasta, resultando em **C:\Servidor\apache-tomcat-7.0.42**.

Criando o projeto no Eclipse

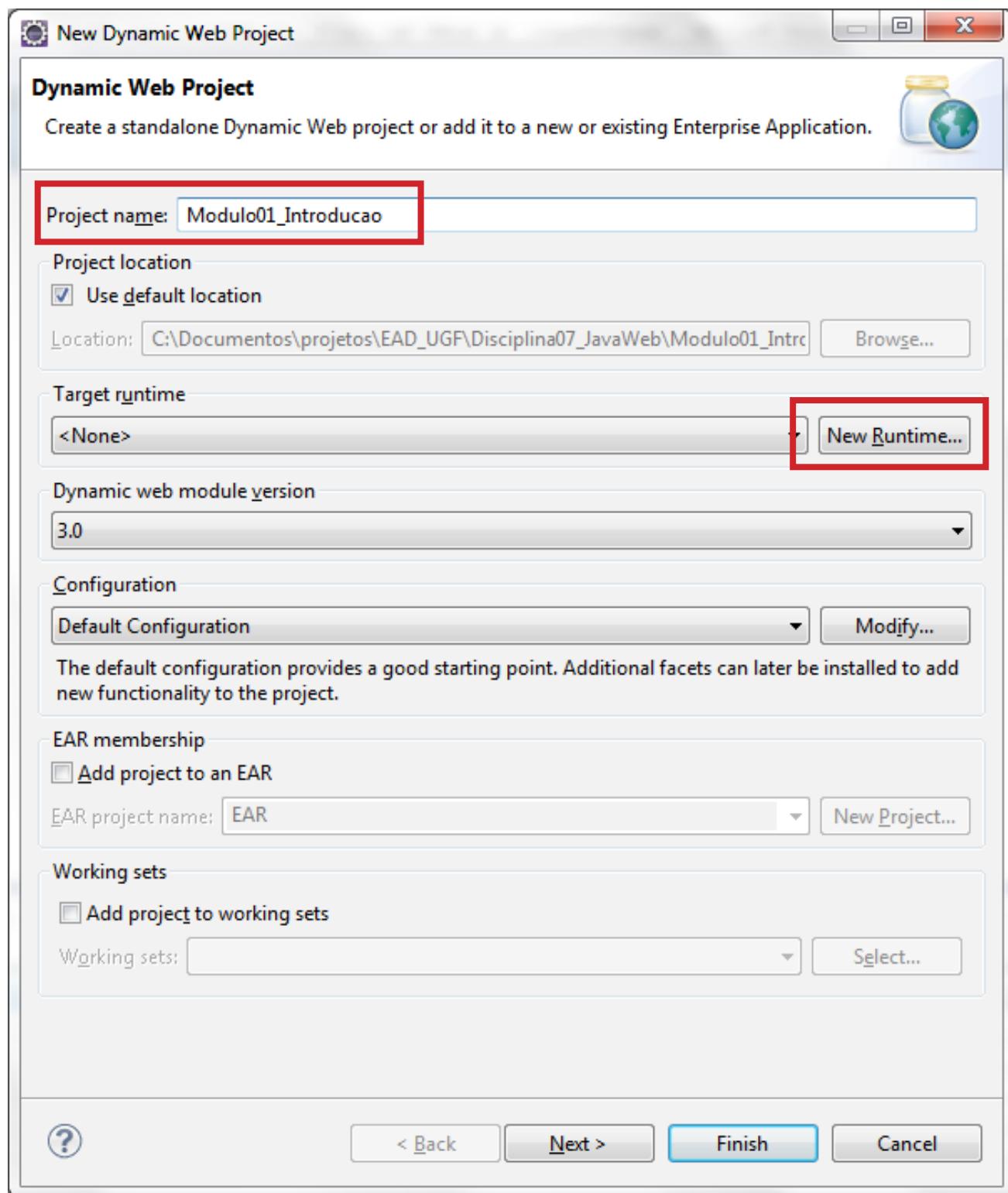
Para criarmos um projeto web no Eclipse temos de manter a perspectiva Java EE. Com essa perspectiva criamos um projeto "Dynamic Web Project". A Figura 2.2a ilustra esse procedimento.

Figura 2.2a: Procedimento para criação de um projeto Java Web

Ao selecionar esta opção, seguimos os passos descritos a seguir. É importante ficar atento aos passos a fim de evitar falhas no projeto.

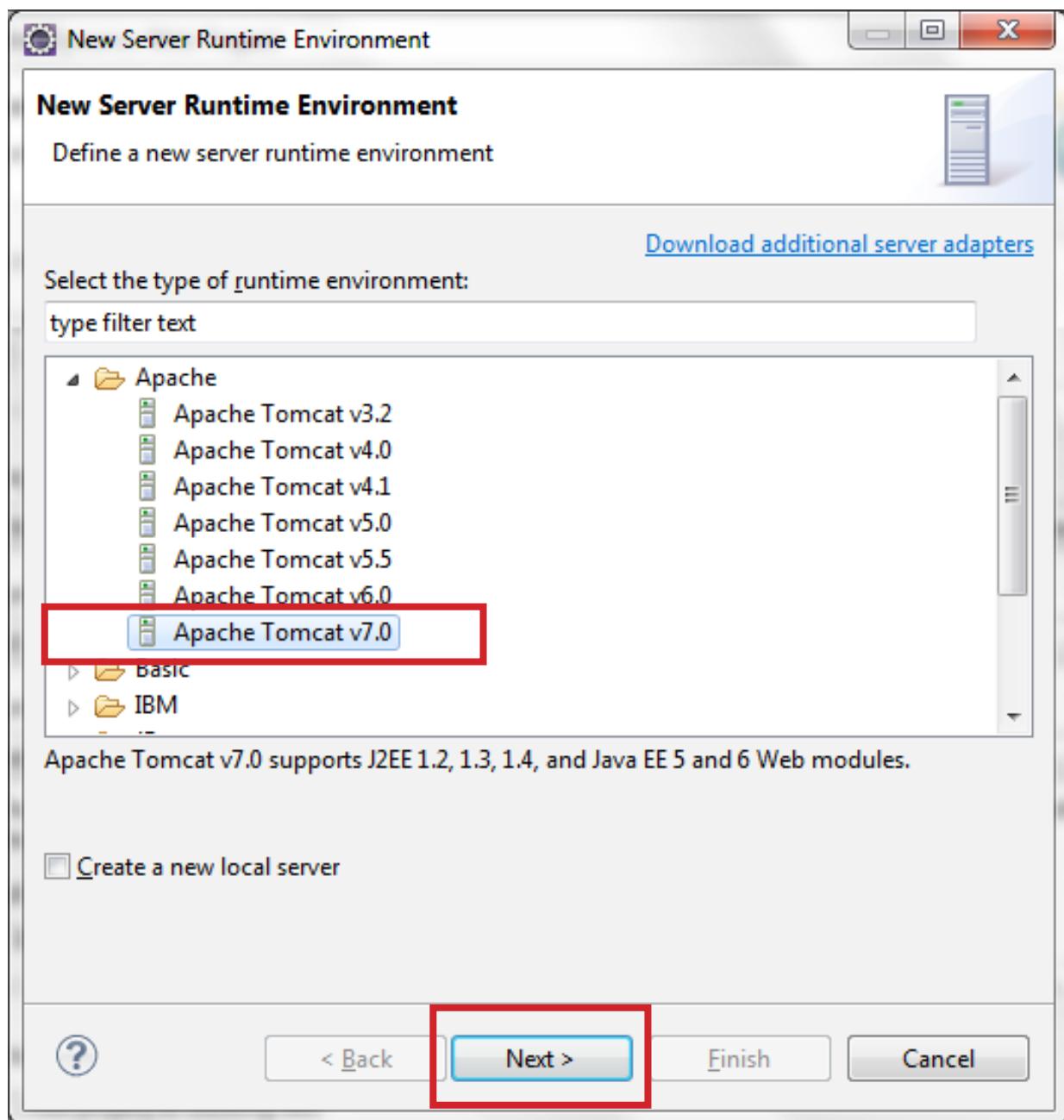
Na Figura 2.2b temos as opções do projeto, em que informamos um nome para ele e, em seguida, selecionamos o servidor de aplicações clicando no botão “**New Runtime...**”.

Figura 2.2b: Definição do projeto

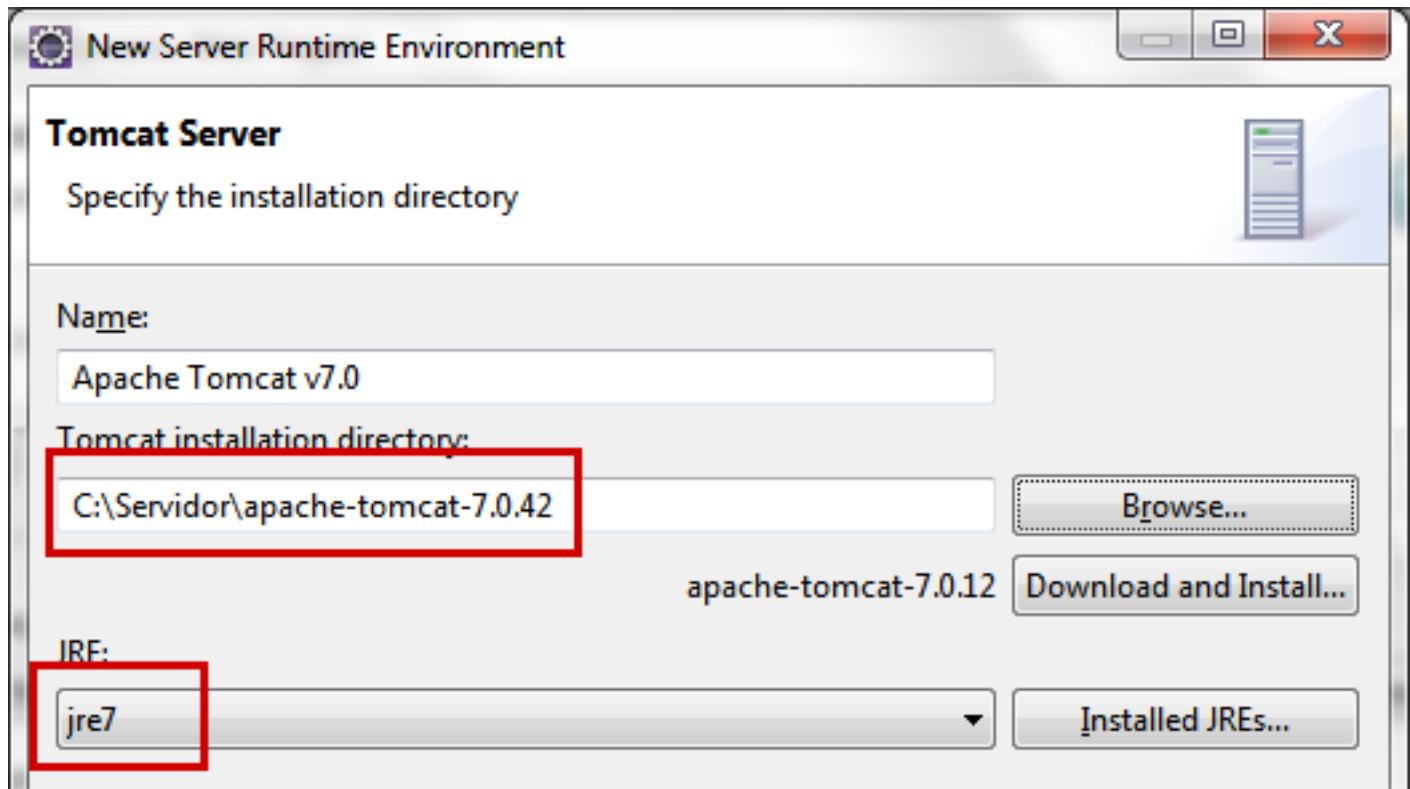


Clicando no botão “New Runtime...” abrimos a janela para seleção do servidor, conforme mostrado na Figura 2.2c. Como estamos trabalhando com a versão 7.0 do Tomcat, selecionamos a opção indicada na Figura 2.2c para que o Eclipse reúna as configurações adequadas para essa versão.

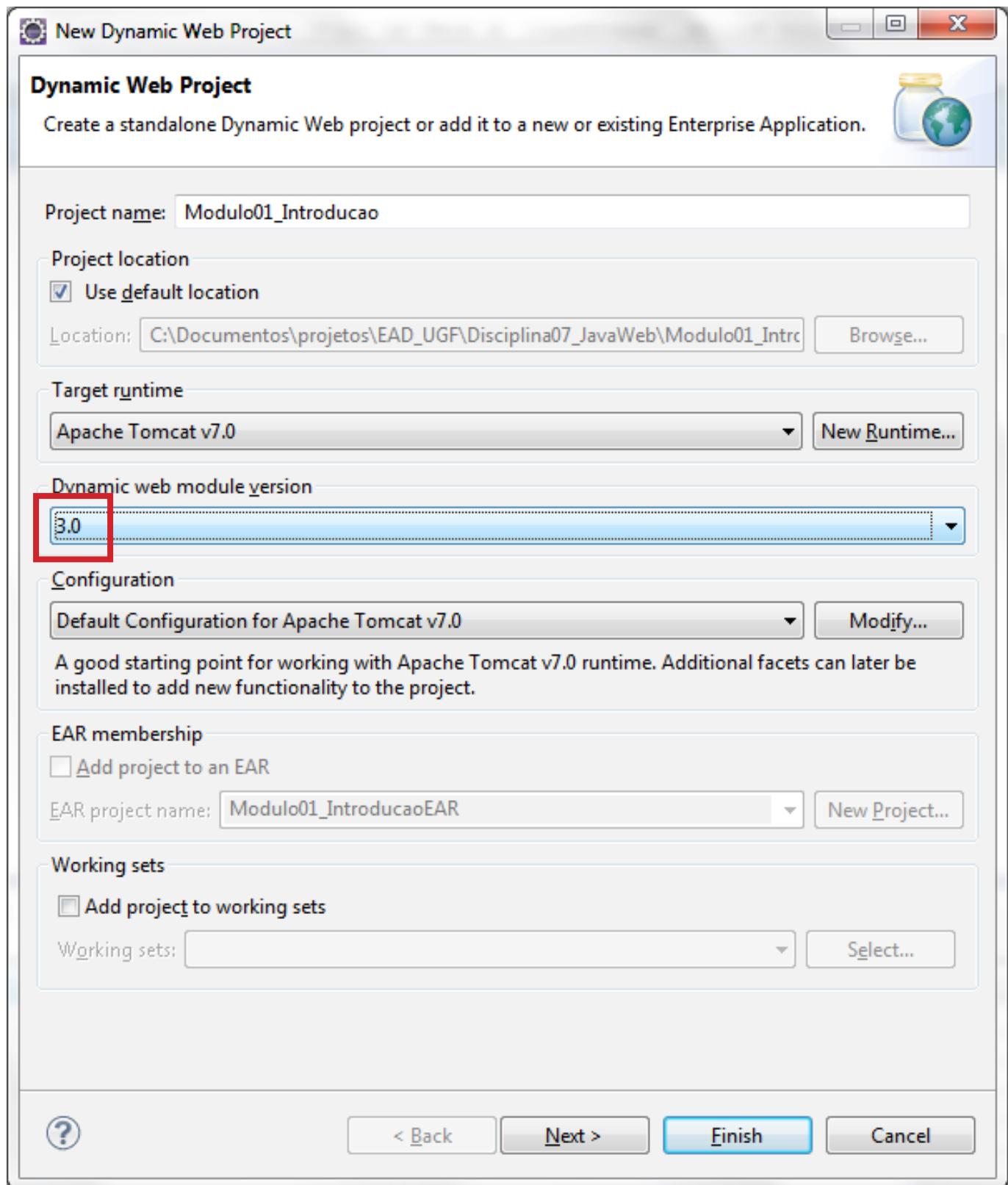
Figura 2.2c: Seleção do servidor de aplicações



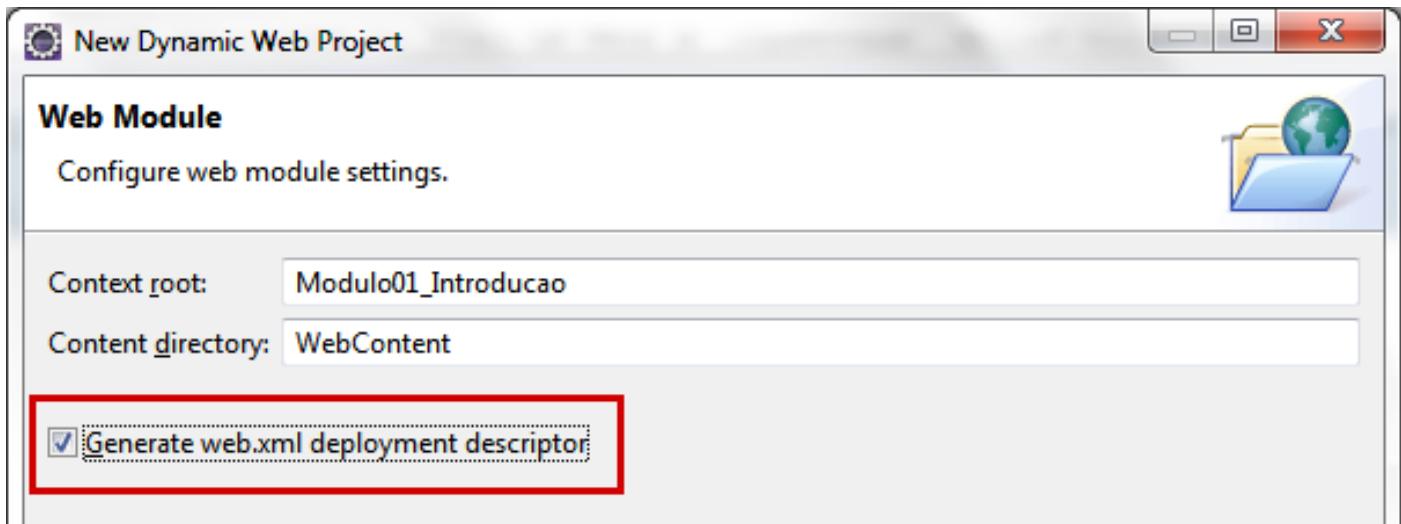
Clicando em “Next” obtemos uma nova janela na qual indicamos a pasta de instalação do Tomcat e a versão do JRE. A Figura 2.2d mostra essas informações devidamente configuradas.

Figura 2.2d: Configuração do servidor

Ao finalizar esta configuração devemos selecionar a versão do módulo *web*. Módulo *web* está relacionado à versão do *servlet* a ser utilizada no projeto. Esse tema será estudado no Módulo 2 desta disciplina, mas aqui podemos adiantar que a versão 3.0 dispensa o uso do arquivo de configurações *web.xml* para o *servlet*. Para os componentes que necessitam de configurações fora do *Servlet*, este arquivo ainda se faz necessário. Por isso, vamos mantê-lo no projeto. A Figura 2.2e mostra a seleção da versão, e para este exemplo manteremos a versão 3.0.

Figura 2.2e: Seleção da versão do módulo web

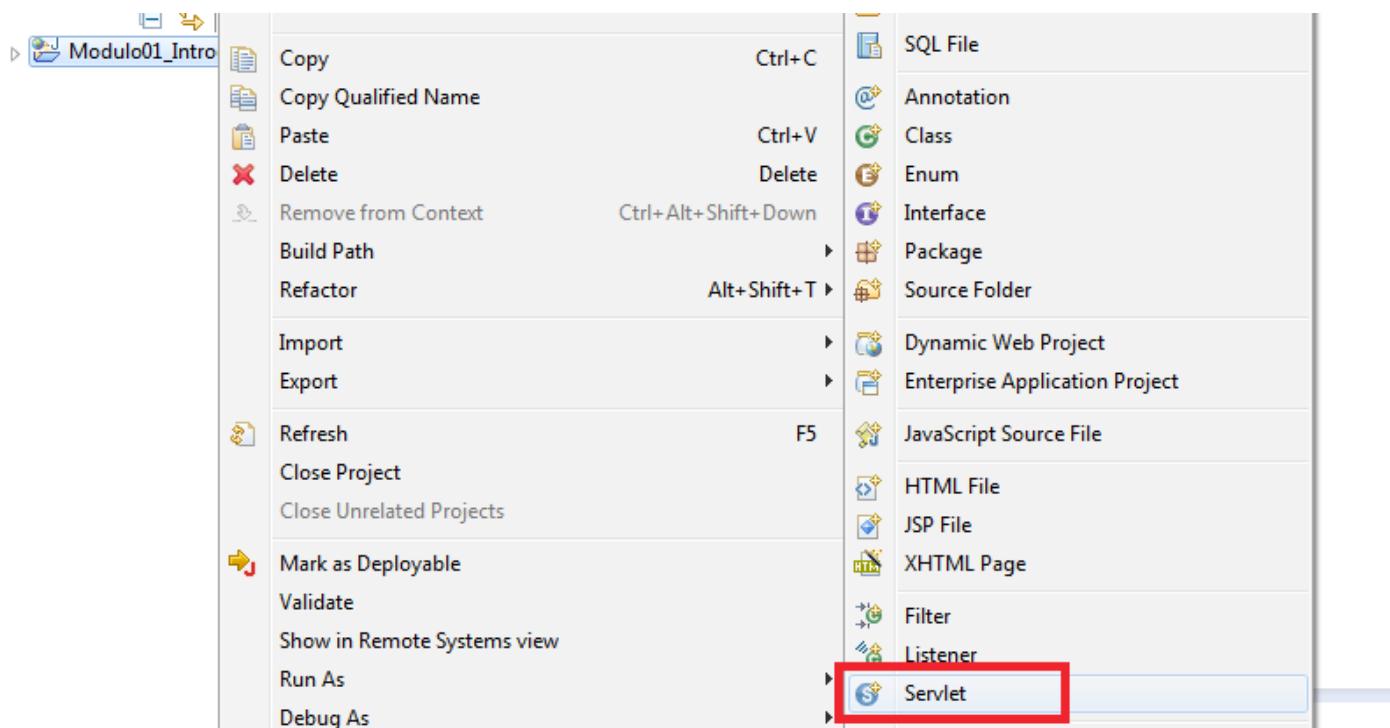
Clicamos em **Next** até chegarmos à configuração mostrada na Figura 2.2f. Nesta tela marcamos a opção **"Generate web.xml deployment descriptor"**. Depois clique em **Finish** e nosso projeto está pronto!

Figura 2.2f: Procedimento para criação de um projeto Java Web

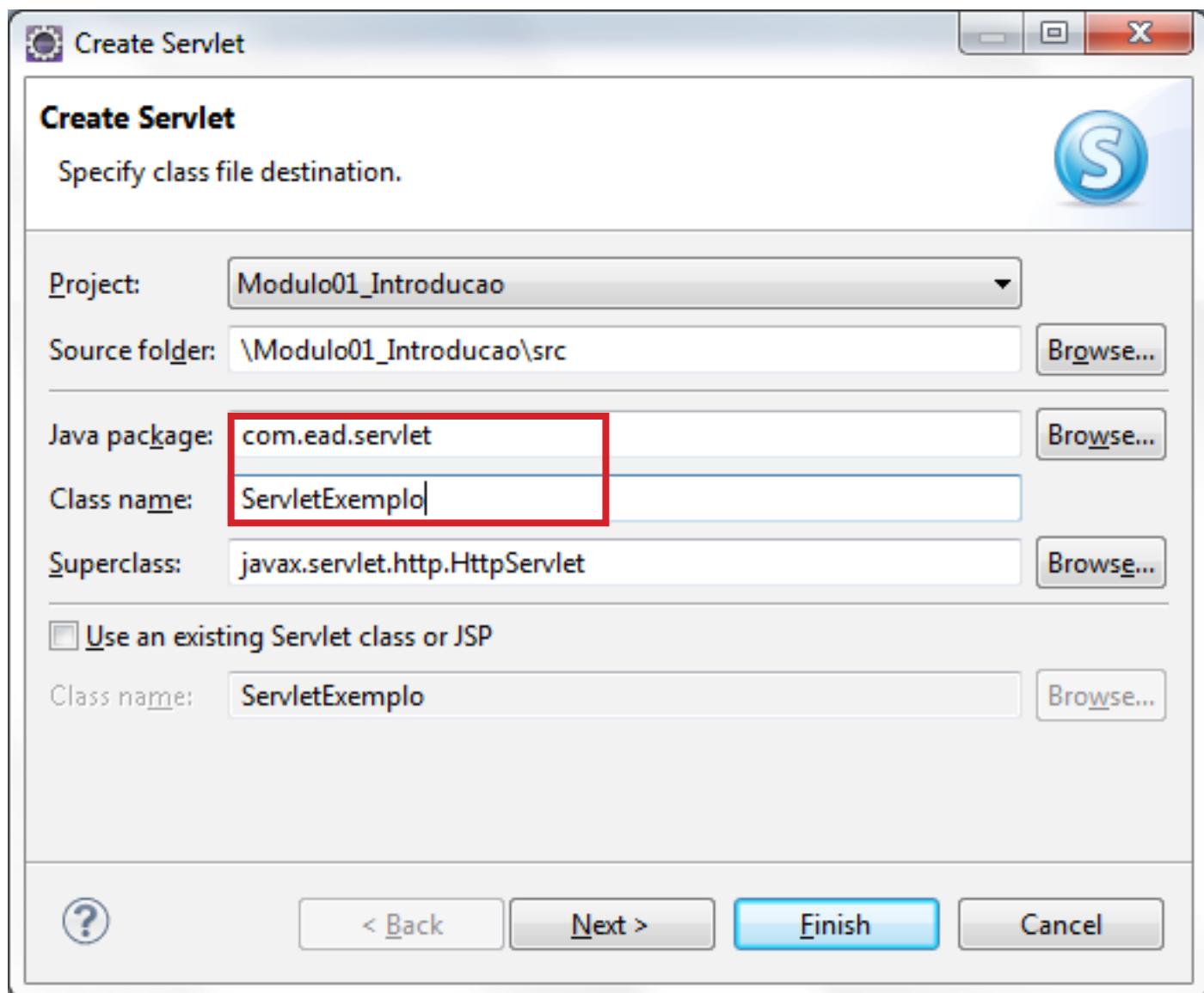
Criando e executando um Servlet

Para ilustrar a criação do nosso primeiro componente *Web*, iniciaremos com o *Servlet*. Vamos seguir os passos indicados:

1. Clique com o botão direito do mouse sobre o projeto, e selecione *Servlet* (Figura 2.3a):

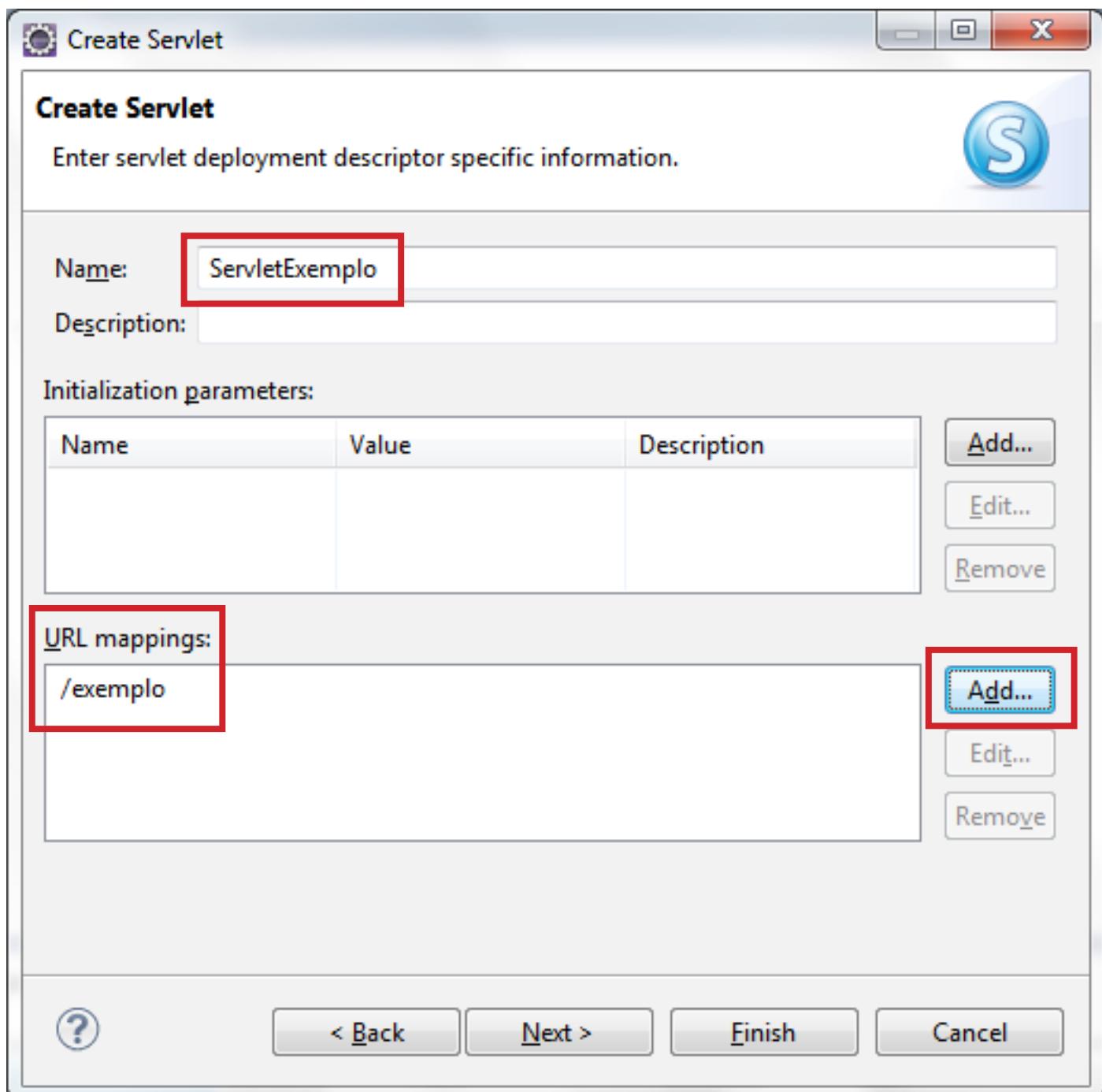
Figura 2.3a: Procedimento para inclusão de um Servlet

2. Forneça o pacote e o nome da classe conforme indicado na Figura 2.3b:;

Figura 2.3b: Configuração do Servlet (Classe e pacote)

3. No próximo passo defina o nome para o *Servlet* (não precisa ser o mesmo nome que a classe). Defina o mapeamento (url pattern) como “/exemplo”. Para alterar o nome do mapeamento, selecione-o na janela e clique em “Edit...” (vide Figura 2.3c):

Figura 2.3c: Configuração do Servlet (Nome e mapeamento)



4. Clique em "Finish". Observe que o Eclipse cria uma nova classe chamada ServletExemplo. Nesta classe existem diversos comentários. Para tornar o código mais limpo, vamos remover estes comentários, deixando a classe da forma mostrada na Figura 2.3d:

Figura 2.3d: Classe ServletExemplo, representativa do Servlet

```

1 package com.ead.servlet;
2
3 import java.io.IOException;
4 import javax.servlet.ServletException;
5 import javax.servlet.annotation.WebServlet;
6 import javax.servlet.http.HttpServlet;
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletResponse;
9
10 @WebServlet("/exemplo")
11 public class ServletExemplo extends HttpServlet {
12     private static final long serialVersionUID = 1L;
13
14
15 public ServletExemplo() {
16     super();
17     // TODO Auto-generated constructor stub
18 }
19
20
21 protected void doGet(HttpServletRequest request, HttpServletResponse response)
22     throws ServletException, IOException {
23     // TODO Auto-generated method stub
24 }
25
26
27 protected void doPost(HttpServletRequest request, HttpServletResponse response)
28     throws ServletException, IOException {
29     // TODO Auto-generated method stub
30 }
31
32 }
--
```

5. Observe que a classe foi criada com dois métodos: **doGet()** e **doPost()**. Escreva o código indicado na Figura 2.3e no método doGet() (os detalhes serão apresentados em módulos posteriores):

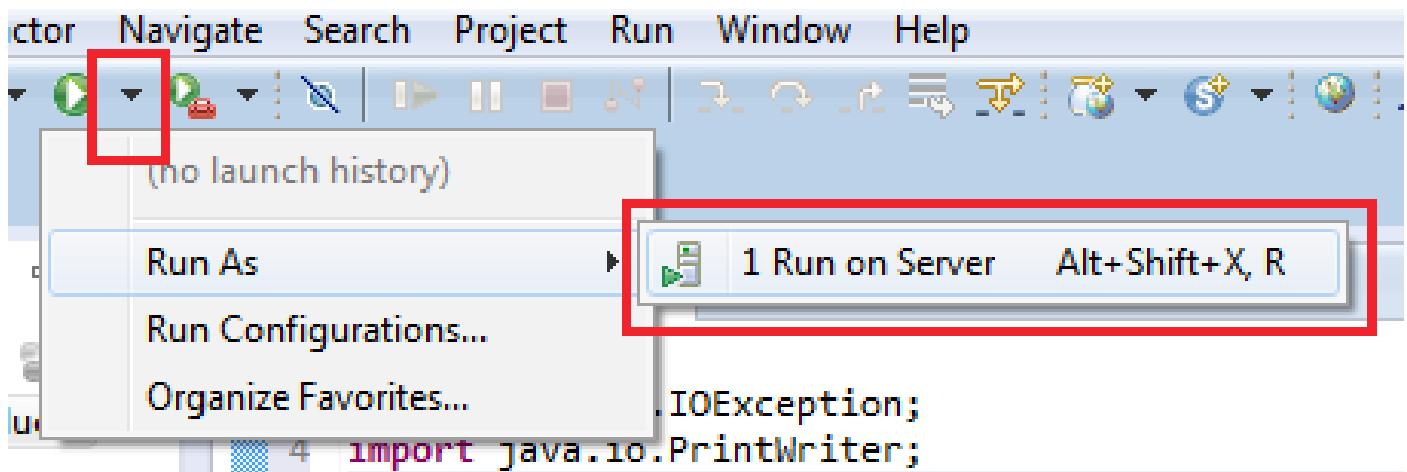
Figura 2.3e: Método doGet() da classe ServletExemplo

```

23
24     protected void doGet(HttpServletRequest request, HttpServletResponse response)
25         throws ServletException, IOException {
26
27         PrintWriter out = response.getWriter();
28         response.setContentType("text/html");
29         out.print("<h2>Benvindo ao estudo de Java Web</h2>");
30     }
31
32 }
```

6. Para visualizar a execução, selecione o ícone de execução, clique na seta à direita e, em seguida, clique na opção “Run on Server”, conforme Figura 2.3f:

Figura 2.3f: Execução da aplicação



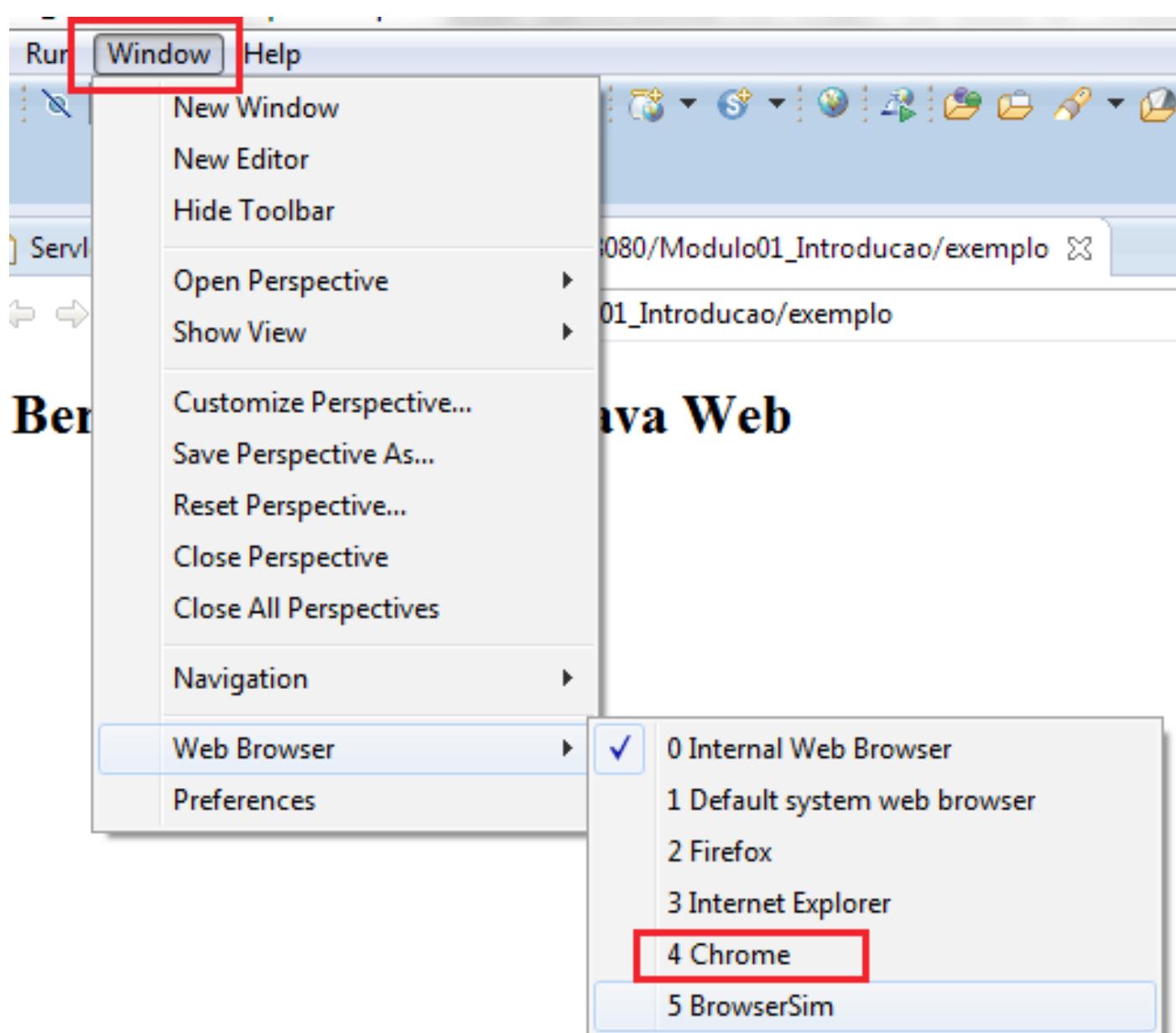
7. Aceite as opções seguintes. Você verá o resultado da aplicação no *browser* embutido no Eclipse (Figura 2.3g):

Figura 2.3g: Browser interno do Eclipse com o resultado



Para visualizar o resultado no seu *browser* preferido, altere a opção no menu Window, conforme indicado na Figura 2.3h:

Figura 2.3h: Configuração do *Browser* no Eclipse



Exercícios do Módulo 1

Exercício 1: Calculadora

Neste exercício criaremos uma calculadora web. Os operandos e a operação são informados como parâmetro na URL. Siga os passos para criar a aplicação:

1. Criar um projeto no Eclipse do tipo “Dynamic Web Project” chamado Modulo01_Exercicio01. Seguir os passos descritos neste módulo para criar o projeto e associar o servidor Tomcat ao projeto.
2. Neste projeto, incluir um novo servlet, com as seguintes configurações:
 - a. Classe: ServletCalculadora
 - b. Nome: ServletCalculadora
 - c. url mapping: /calculadora
3. No método doGet() do servlet, incluir o seguinte código:

```
PrintWriter out = response.getWriter();
response.setContentType("text/html");

try {
    double op1 = Double.parseDouble(request.getParameter("op1"));
    double op2 = Double.parseDouble(request.getParameter("op2"));
    int operacao = Integer.parseInt(request.getParameter("op"));
    double resultado;

    switch(operacao){
        case 1: resultado = op1 + op2;break;
        case 2: resultado = op1 - op2;break;
        case 3: resultado = op1 * op2;break;
        case 4: resultado = op1 / op2;break;
        default: resultado = 0;
    }
    out.print("Resultado: " + resultado);
} catch (Exception e) {
    out.print(e.getMessage());
}
```

4. Executar a aplicação no Eclipse. Você verá que na URL aparecerá um erro no browser. Escrever, então na URL o seguinte código: http://localhost:8080/Modulo01_Exercicio01/calculadora?op1=10&op2=20&op=2
5. Alterar os valores dos parâmetros op1, op2 e op, analisando o resultado.

Módulo 2 - Servlets

Aula 3 – Conceitos básicos sobre Servlets

Definição de Servlets

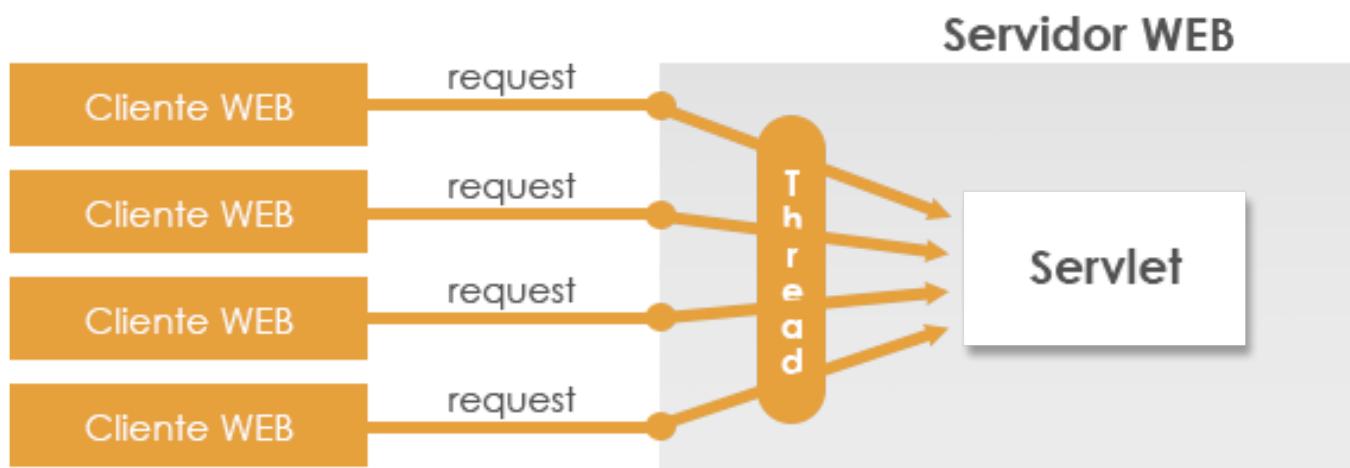
Servlets são classes Java derivadas da classe HttpServlet. Sua instanciação ocorre pelo *container* Web, e no *container* existe apenas uma instância. Havendo mais de uma requisição, o *container* cria múltiplas *threads* para as requisições que ele recebe através do protocolo HTTP.

Servlets não possuem interface gráfica e suas instâncias são executadas dentro de um ambiente Java denominado Container. Isso significa que elas evitam todos os problemas de portabilidade associados às diferentes interfaces de exibição.

O container gerencia as instâncias dos Servlets e provê os serviços de rede necessários para as requisições e respostas. O container atua em associação com servidores Web recebendo as requisições reencaminhada por eles.

Tipicamente existe apenas uma instância de cada Servlet, no entanto, o *container* pode criar vários servlets (múltiplas linhas de execução), como mostra a Figura 3.1a.

Figura 3.1a: Requisições simultâneas a um Servlet



Depois de executado, um *servlet* em geral residirá na memória do servidor. Isso evita a sobrecarga da construção de um novo processo de *servlet* para cada acesso, economiza memória e torna o acesso à página eficiente. Como os Servlets permanecem na memória, eles podem manter referências a outros objetos Java.

Por exemplo, se seu servidor de banco de dados contém licenças de conexão simultânea suficientes, uma conexão a banco de dados pode ser compartilhada entre as linhas de execução, reduzindo assim a sobre-

carga associada ao estabelecimento e manutenção da conexão. O esquema de execução de um *Servlet* está ilustrado na Figura 3.1b:

Figura 3.1b: Esquema de requisição a um *Servlet*



Localização das classes de um *Servlet*

Conforme descrito no capítulo anterior, as classes correspondentes a um *Servlet* devem ser colocadas na pasta **classes** ou na pasta **lib**, conforme esteja ou não definida em um arquivo **.jar**, ambas abaixo da pasta **WEB-INF**.

Criação de Servlets – Até a versão 2.5 (Java EE 5.0)

Quando o *Servlet* estava na versão 2.5, correspondendo ao JEE 5, o *Servlet* usava o arquivo de configurações **web.xml** para definir suas configurações. O Quadro 11 apresenta a classe que define um *Servlet*.

Quadro 11: Classe `ServletExemplo01`, representando um *Servlet*

```

package com.servlet;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletExemplo01 extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    }
}

```

O Quadro 12 apresenta o Deployment Descriptor (*web.xml*).

Quadro 12: Arquivo *web.xml* para o *servlet* *ServletExemplo01*

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
        <servlet-name>ServletExemplo01</servlet-name>
        <servlet-class>com.servlet.ServletExemplo01</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServletExemplo01</servlet-name>
        <url-pattern>/exemplo01</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>

```

Estrutura do *Servlet*

Os pacotes `javax.servlet` e `javax.servlet.http` definem as interfaces e classes para implementação de **Servlets**. O mais usual é criar uma classe que estenda à classe `javax.servlet.http.HttpServlet`.

A classe `javax.servlet.http.HttpServlet` fornece métodos, tais como: `doGet()` e `doPost()` para tratar serviços HTTP.

Para cada método HTTP há um método correspondente na classe `HttpServlet` do tipo:

```
public void doXXX(HttpServletRequest, HttpServletResponse) throws ServletException,
IOException {}
```

`doXXX()` depende do método HTTP, conforme o esquema:

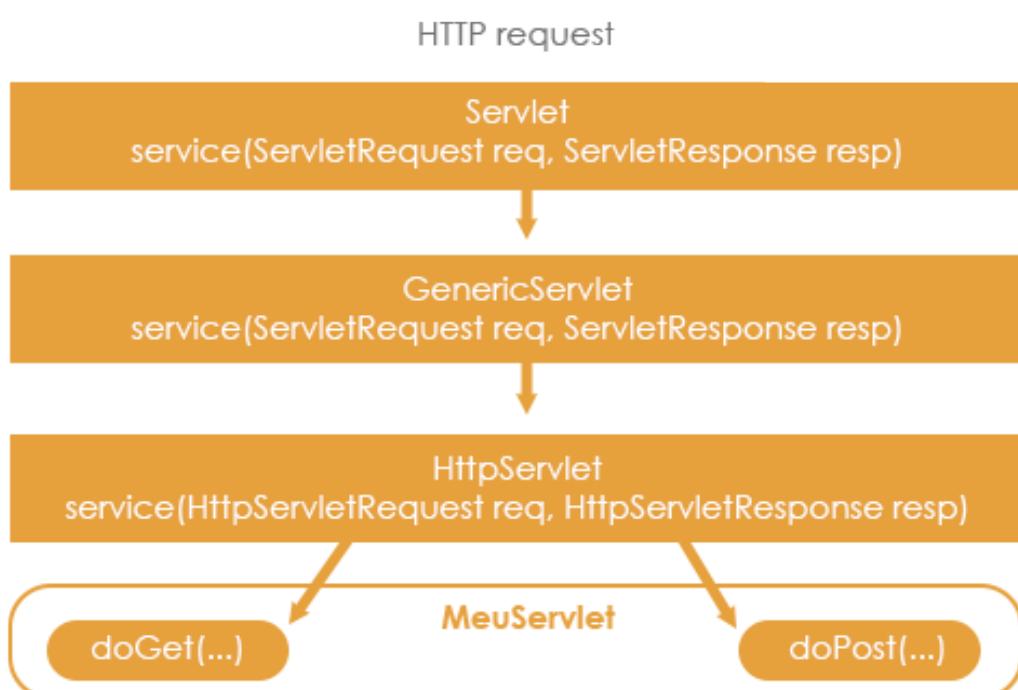
Método HTTP	Método HttpServlet
GET	<code>doGet()</code>
POST	<code>doPost()</code>

Aula 4 - Ciclo de vida do *Servlet*

Métodos do ciclo de vida de um *Servlet*

A hierarquia de classes ilustrada na Figura 4.1a mostra a sequência de eventos `HttpServlet`. Observe que o *Servlet* que criamos possui os métodos `doGet()` e `doPost()` sobrescritos, e estes são chamados pelo método `service()`. O método `service()` é responsável pelo ciclo de vida do *Servlet*. Ele reconhece a requisição e decide por chamar `doGet()` ou `doPost()`.

Figura 4.1a: Hierarquia de classes para um *Servlet*



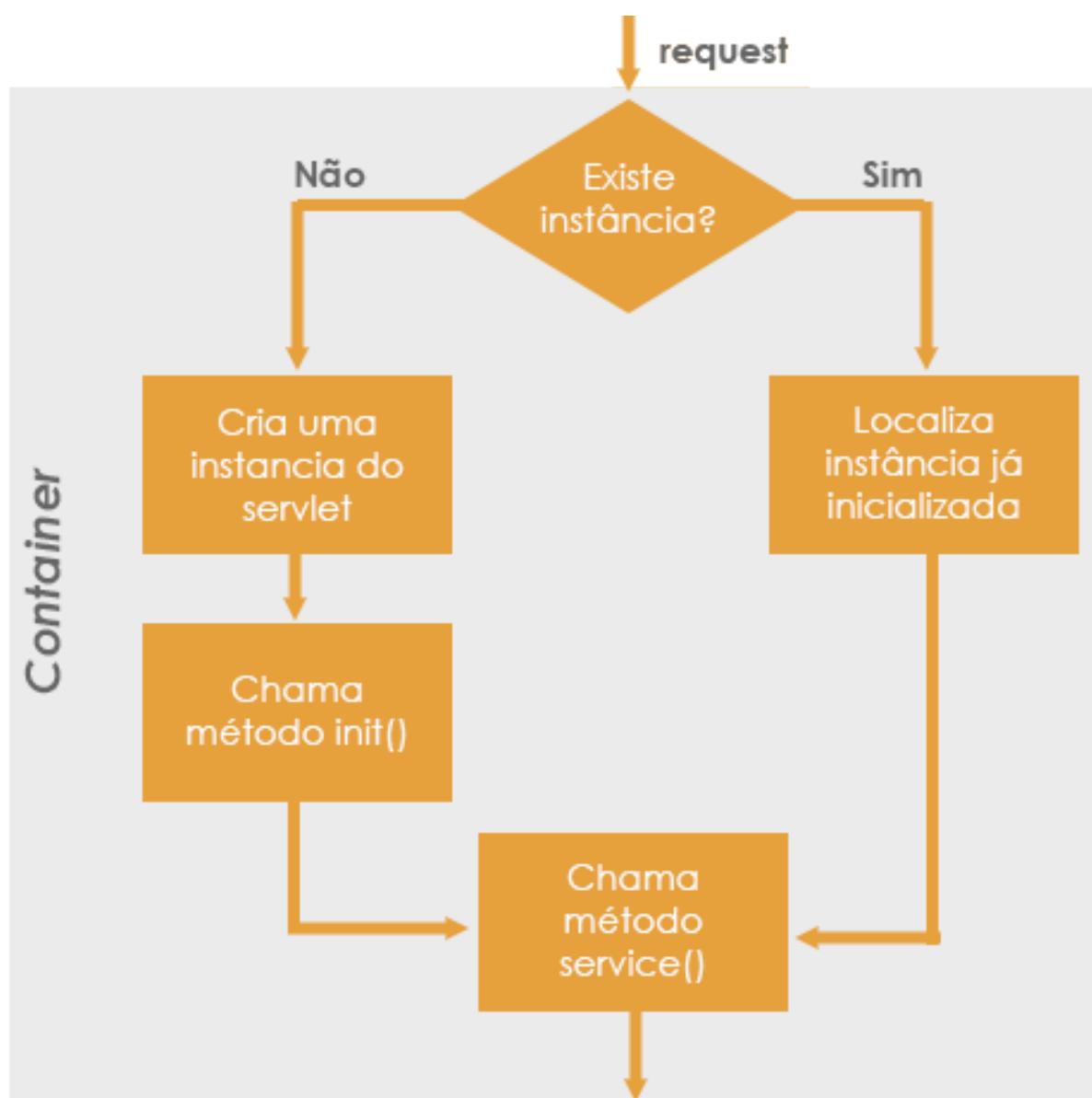
O ciclo de vida de um *Servlet* é controlado pelo *container*. A classe HttpServlet fornece métodos para controle do ciclo de vida do *Servlet* pelo *container*, como:

- `init()`
- `destroy()`
- `service()`

Se o método `service()` for sobreescrito, os métodos `doXXX()` não serão chamados.

A Figura 4.1b ilustra o ciclo de vida do *servlet*.

Figura 4.1b: Ciclo de vida de um *Servlet*



Todos os métodos **doXXX()** recebem dois parâmetros, ambos criados pelo *container* e passados para o *servlet* como argumentos:

- **HttpServletRequest**: Define um objeto que fornece informações do *request* HTTP para o *servlet*. Provê dados, tais como parâmetros e valores enviados pelo *browser*, atributos e *streams* de entrada, informações sobre o protocolo, endereço IP do cliente etc. Ele estende a interface **ServletRequest**.
- **HttpServletResponse**: Define um objeto que permite ao *servlet* enviar resposta ao cliente *web*. Permite definir atributos no header HTTP e criar *cookies*. Ele estende a interface **ServletResponse**.

Principais métodos da requisição

Na Tabela 4.2 é apresentado os principais métodos da requisição.

Tabela 4.2: Métodos da Requisição

Método	Definido em	Descrição
<code>void setAttribute(String key, Object value)</code>	<code>ServletRequest</code>	Associa um valor de atributo com um nome.
<code>Enumeration getAttributeNames()</code>	<code>ServletRequest</code>	Recupera os nomes de todos os atributos associados com o objeto.
<code>Object getAttribute(String key)</code>	<code>ServletRequest</code>	Recupera o valor de atributo associado com a chave.
<code>void removeAttribute(String key)</code>	<code>ServletRequest</code>	Remove o valor de atributo associado com a chave.
<code>Enumeration getParameterNames()</code>	<code>ServletRequest</code>	Retorna os nomes de todos os parâmetros de solicitação.
<code>String getParameter(String name)</code>	<code>ServletRequest</code>	Retorna os primeiros valores (principal) de um único parâmetro de solicitação.
<code>String[] getParameterValues(String name)</code>	<code>ServletRequest</code>	Recupera todos os valores para um único parâmetro de solicitação.
<code>Enumeration getHeaderNames()</code>	<code>HttpServletRequest</code>	Recupera os nomes de todos os cabeçalhos associados com a solicitação.
<code>String getHeader(String name)</code>	<code>HttpServletRequest</code>	Retorna o valor de um único cabeçalho de solicitação, como uma cadeia.
<code>Enumeration getHeaders(String name)</code>	<code>HttpServletRequest</code>	Retorna todos os valores para um único cabeçalho de solicitação.
<code>int getIntHeader(String name)</code>	<code>HttpServletRequest</code>	Retorna o valor de um único cabeçalho de solicitação, com um número inteiro.

long getDateHeader(String name)	HttpServletRequest	Retorna o valor de um único cabeçalho de solicitação, como uma data.
Cookies[] getCookies()	HttpServletRequest	Recupera todos os cookies associados com a solicitação.
String getMethod()	HttpServletRequest	Retorna o método de HTTP (POST, GET, etc) para a solicitação.
String getRequestURI()	HttpServletRequest	Retorna o URL de solicitação (não inclui a cadeia de consulta).
String getQueryString()	HttpServletRequest	Retorna a cadeia de consulta que segue o URL de solicitação, se houver algum.
HttpSession getSession()	HttpServletRequest	Recupera os dados da sessão para a solicitação (i.e, o objeto implícito session).
HttpSession getSession(boolean flag)	HttpServletRequest	Recupera os dados da sessão para a solicitação (i.e, o objeto implícito session), opcionalmente criando-o se ele ainda não existir.
RequestDispatcher getRequestDispatcher(String path)	ServletRequest	Cria um dispatcher de solicitação para o URL local indicado.
String getRemoteHost()	ServletRequest	Retorna o nome totalmente qualificado do host que enviou a solicitação.
String getRemoteAddr()	ServletRequest	Retorna o endereço de rede (IP) do host que enviou a solicitação.
String getRemoteUser()	HttpServletRequest	Retorna o nome do usuário que enviou a solicitação, se conhecido.

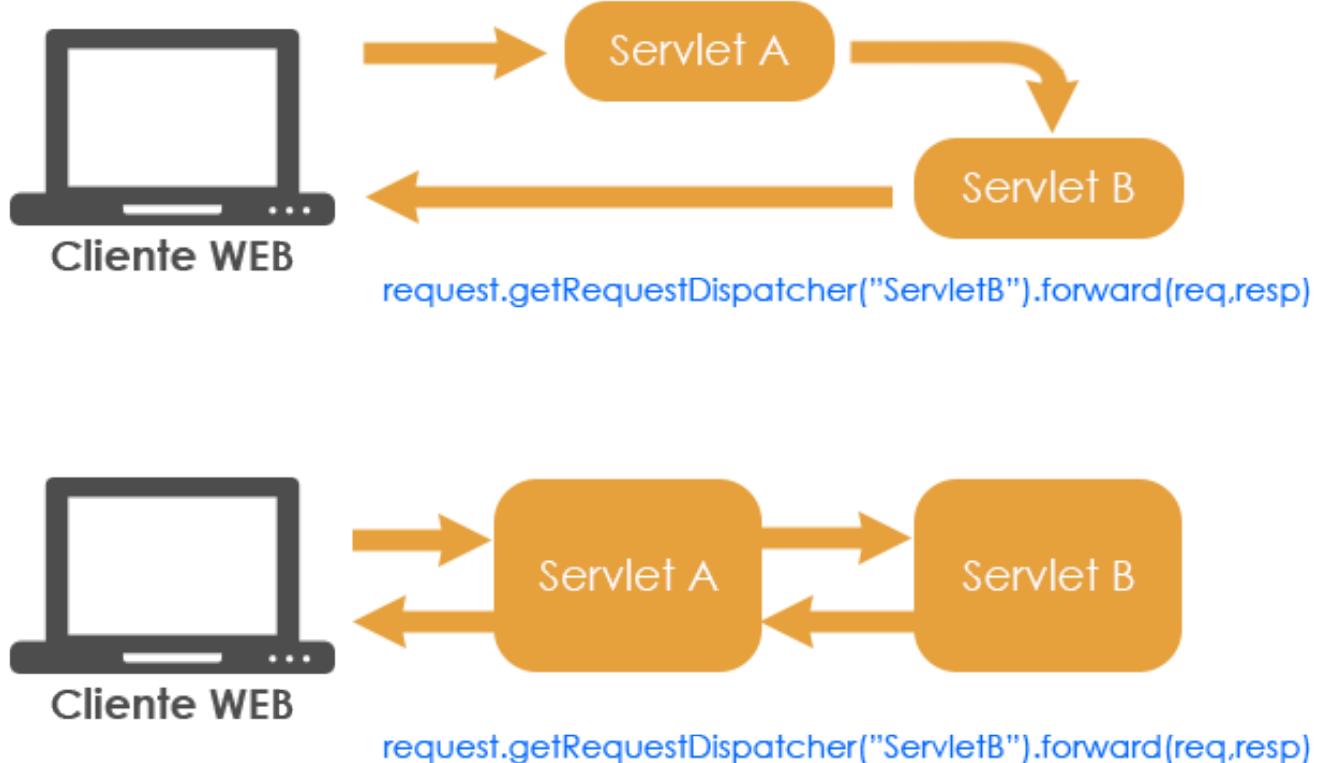
Transferência de requisições

O *Servlet* pode receber uma requisição e transferir seu conteúdo para outro componente, mantendo a execução (o controle) na *servlet* atual. Este processo é usado normalmente quando se deseja incluir e/ou remover atributos na requisição e transferi-la para outro recurso utilizá-la. O mecanismo de transferência pode ocorrer de duas formas:

- enviando a requisição para um recurso externo (forward);
- recebendo a requisição de um recurso externo (include).

A Figura 4.3 ilustra esse procedimento.

Figura 4.3: Transferência de requisições



Principais métodos da resposta

A Tabela 4.4 apresenta os principais métodos da resposta.

Tabela 4.4: Métodos da Requisição

Método	Definido em	Descrição
<code>void addCookies(Cookie cookie)</code>	<code>HttpServletResponse</code>	Adiciona o <i>cookie</i> especificado.
<code>boolean containsHeader(String name)</code>	<code>HttpServletResponse</code>	Verifica se a resposta inclui o cabeçalho.
<code>void setHeader(String name, String value)</code>	<code>HttpServletResponse</code>	Atribui o valor definido pela variável "value" ao cabeçalho especificado por "name"
<code>void setIntHeader(String name, int value)</code>	<code>HttpServletResponse</code>	Atribui o valor de número inteiro especificado por "value" ao cabeçalho especificado por "name"

void setDateHeader(String name, long date)	HttpServletResponse	Atribui o valor de data especificado por "value" ao cabeçalho especificado por "name"
void addHeader(String name, String value)	HttpServletResponse	Adiciona o valor definido por "value" ao cabeçalho especificado por "name"
void addIntHeader(String name, int value)	HttpServletResponse	Adiciona o valor de número inteiro especificado por "value" ao cabeçalho especificado por "name"
void addDateHeader(String name, long date)	HttpServletResponse	Adiciona o valor de data especificado por "value" ao cabeçalho especificado por "name"
void setStatus(int code)	HttpServletResponse	Define o código de <i>status</i> para a resposta (para circunstâncias sem erro)
void sendError(int status, String msg)	HttpServletResponse	Define o código de <i>status</i> e mensagem de erro para a resposta.
void sendRedirect(String url)	HttpServletResponse	Envia uma resposta para o navegador indicando que ele deveria solicitar um URL alternativo (absoluto)
String encodeRedirectURL(String url)	HttpServletResponse	Codifica um URL para uso com o método <i>sendRedirect()</i> para incluir informações de sessão.
String encodeURL(String url)	HttpServletResponse	Codifica um URL usado em um link para incluir informações de sessão.
void setContentType(String type)	ServletResponse	Define o tipo MIME e, opcionalmente, a codificação de caracteres do conteúdo da resposta.
String getCharacterEncoding()	ServletResponse	Retorna o conjunto de estilos de codificação de caracteres para o conteúdo da resposta.

Redirecionamento de *Servlets*

Ao contrário da transferência de requisições, o redirecionamento abandona o recurso atual e executa, via método GET, o novo recurso. Esse processo equivale a realizar uma chamada ao novo recurso através da URL. Exemplo:

```
response.sendRedirect("cadastro");
```

Neste exemplo, "cadastro" representa o mapeamento do *servlet* destino.

Aula 5 - Criação de *Servlets* – versão 3.0

Anotações em *Servlets*

A versão 3.0 do *Servlet* corresponde ao Java EE 6.0 e incluiu algumas funcionalidades, além de dispensar o uso do arquivo de configurações *web.xml*.

Para ilustrar, considere um *servlet* chamado *ServletLogin*, contendo como parâmetros de inicialização: **usuario** e **senha**, conforme mostra o Quadro 13.

Quadro 13: Classe *ServletLogin* representando um *servlet* na versão 3.0.

```
package com.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="ServletLogin", urlPatterns = {" /login"}, initParams = {
    @WebInitParam(name = "usuario", value = "admin"),
    @WebInitParam(name = "senha", value = "admin")})
public class ServletLogin extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    }
}
```

Observe as anotações na classe:

```
@WebServlet(name="ServletLogin", urlPatterns = {" /login"}, initParams = {
    @WebInitParam(name = "usuario", value = "admin"),
    @WebInitParam(name = "senha", value = "admin")})
```

Isso equivale à configuração mostrada no Quadro 14.

Quadro 14: Configuração equivalente na versão 2.5 do *Servlet*

```
< servlet >
    < servlet-name > ServletLogin < /servlet-name >
    < servlet-class > com.servlet.ServletLogin < /servlet-class >
    < init-param >
        < param-name > usuario < /param-name >
        < param-value > admin < /param-value >
    < /init-param >
    < init-param >
        < param-name > senha < /param-name >
        < param-value > admin < /param-value >
    < /init-param >
< /servlet >
< servlet-mapping >
    < servlet-name > ServletLogin < /servlet-name >
    < url-pattern > /login < /url-pattern >
< /servlet-mapping >
```

Atenção: Se houver tanto anotações na classe quanto a presença do arquivo *web.xml*, a prioridade é das informações contidas no arquivo *web.xml*. O objetivo é manter a compatibilidade com versões anteriores.

A configuração do *Servlet* 3.0 se dá através de anotações, como já vimos. Estas anotações estão no pacote **javax.servlet.annotation**, e são elas:

- **@WebServlet** – usada nas classes representativas de *Servlets*.
- **@WebInitParam** – usada para definir parâmetros de inicialização do *Servlet*.
- **@WebFilter** – usada em Filtros (assunto a ser abordado posteriormente).
- **@WebListener** - usada para definir Listeners (assunto abordado posteriormente).
- **@MultipartConfig** – usada para definir que um *Servlet* receberá uma requisição do tipo mime/multipart.

A tabela 5.1a mostra os atributos de **@WebServlet**.

Tabela 5.1a: Atributos da anotação **@WebServlet**

Atributo	Descrição
asyncSupported	Se operações assíncronas são suportadas pelo <i>servlet</i> , deve-se atribuir valor <i>true</i> ; do contrário, <i>false</i> . O valor default é <i>false</i> .
description	Descrição do <i>Servlet</i> .
displayName	Nome visível do <i>Servlet</i> .
initParams	Parâmetros de inicialização do <i>Servlet</i> .
largeIcon	Contém a String URL como <i>large icon</i> .
loadOnStartup	Usado para definir a prioridade do <i>Servlet</i> .

name	Usado para definir o nome do <i>Servlet</i> .
smallIcon	Contém a String URL como <i>small icon</i>
urlPatterns	Descreve as <i>URL patterns</i> do <i>Servlet</i> .
value	Também usado para definir as <i>URL patterns</i> do <i>Servlet</i> . Um deles deve ser implementado: <i>urlPatterns</i> ou <i>value</i>

A tabela 5.1b mostra os atributos de `@WebInitParam`.

Tabela 5.1b: Atributos da anotação `@WebInitParam`

Atributo	Descrição
name	Contém o nome do parâmetro de inicialização como string.
value	Contém o valor do parâmetro de inicialização como string.
description	Contém a descrição do parâmetro de inicialização.

A tabela 5.1c mostra os atributos de `@WebFilter`.

Tabela 5.1c: Atributos da anotação `@WebFilter`

Atributo	Descrição
asyncSupported	Se operações assíncronas são suportadas pelo Filtro, deve-se atribuir valor <i>true</i> ; do contrário, <i>false</i> . O valor default é <i>false</i>
description	Descrição do Filtro.
dispatcherTypes	Contém o tipo dos dispatches aplicados ao Filtro.
displayName	Nome visível do Filtro.
filterName	Nome do Filtro.
initParams	Parâmetros de inicialização do Filtro
largeIcon	Contém a String URL como <i>large icon</i>
 servletNames	Nome dos servvlets sobre os quais o Filtro é aplicado
smallIcon	Contém a String URL como <i>small icon</i>
urlPatterns	Descreve as <i>URL patterns</i> dos Servlets sobre os quais o Filtro é aplicado

Com a anotação `@WebListener` os seguintes *listeners* podem ser registrados:

- **Context Listener** (`javax.servlet.ServletContextListener`)
- **Context Attribute Listener** (`javax.servlet.ServletContextAttributeListener`)
- **Servlet Request Listener** (`javax.servlet.ServletRequestListener`)
- **Servlet Request Attribute Listener** (`javax.servlet.ServletRequestAttributeListener`)
- **Http Session Listener** (`javax.servlet.http.HttpSessionListener`)
- **Http Session Attribute Listener** (`javax.servlet.http.HttpSessionAttributeListener`)

Considerações sobre mapeamento do servlet

O exemplo do Quadro 14 definiu um *servlet* acessível através do mapeamento/**login**. Esse tipo de mapeamento permite o acesso ao *servlet* apenas desta forma, mas é possível que vários mapeamentos sejam estabelecidos para o mesmo *servlet*, além da utilização de um curinga.

Por exemplo, considere o mapeamento:

```
@WebServlet(name = "ServletLogin", urlPatterns = {"*.do"})
```

Ele permite a execução do *servlet* **ServletLogin** através de diversos mapeamentos, como:

```
programa.do  
cadastro.do  
login.do  
etc.do
```

Ou seja, qualquer expressão pode ser usada, desde que tenha o termo “.do” anexado ao final. Esse mecanismo é útil para *servlets* que centralizam requisições e geram respostas adequadas para cada uma destas requisições. Frameworks (JSF, por exemplo) utilizam esta abordagem.

ServletConfig

No ciclo de vida do *Servlet*, o método `init()` recebe como parâmetro uma referência `ServletConfig`, fornecida pelo *container*. É a partir desta referência que são obtidos os valores de configuração do *Servlet*, incluindo os parâmetros de inicialização.

O Quadro 15 mostra um exemplo obtendo os valores dos parâmetros de inicialização do *servlet* `ServletLogin`.

Quadro 15: Método doGet() obtendo parâmetros de inicialização

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    String usuario = getServletConfig().getInitParameter("usuario");
    String senha = getServletConfig().getInitParameter("senha");

    PrintWriter out = response.getWriter();
    out.print(usuario);
    out.print(senha);
}
```

O método `getServletConfig()` retorna uma referência ao `ServletConfig` informado pelo *container*. A partir desta referência podemos executar:

- `getInitParameter(String)`: acesso aos parâmetros de inicialização
- `getInitParameterNames()`: acesso aos nomes dos parâmetros de inicialização
- Uma estratégia interessante é carregar os parâmetros no método `init()` do *Servlet*.

ServletContext

Permite o acesso às configurações do contexto, ou seja, da aplicação. Ele é criado no momento em que o servidor de aplicações é iniciado. Alguns métodos de `ServletContext` são:

- `getInitParameter(String)`: permite acesso aos parâmetros de inicialização do contexto;
- `getInitParameterNames()`: permite acesso aos nomes dos parâmetros de inicialização do contexto;
- `setAttribute(String, Object)`: grava um objeto no contexto, ou seja adiciona um item no contexto;
- `getAttribute(String)`: obtém um objeto do contexto;
- `removeAttribute(String)`: remove um objeto do contexto.

Para obter um parâmetro de inicialização de um contexto, definir os nós no arquivo **web.xml**, como mostra o Quadro 16.

Quadro 16: Configuração de parâmetro de inicialização de um contexto

```
<context-param>
    <param-name>aplicacao</param-name>
    <param-value>Contas a pagar</param-value>
</context-param>
```

Os parâmetros de inicialização do *servlet* são muito parecidos. É importante verificar que os parâmetros de inicialização do contexto são definidos FORA do elemento `<servlet>`, enquanto que os parâmetros de inicialização do *servlet* são definidos, obviamente, DENTRO do elemento `<servlet>`.

O acesso a esses parâmetros é realizado através de uma referência a ServletContext, como mostra o Quadro 17.

Quadro 17: Método doGet() obtendo os parâmetros de inicialização do contexto

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    String app = getServletContext().getInitParameter("aplicacao");
    //restante do código
}
```

Geralmente os parâmetros de inicialização do contexto são usados para definir configurações para toda a aplicação.

Aula 6 - Gerenciamento de Sessão

Definição de gerenciamento de sessão

Quando uma requisição é realizada, ocorrem quatro etapas mostradas na Figura 6.1.

Figura 6.1: Mecanismo de requisição e resposta



Se uma nova requisição é realizada, esta não possui nenhuma informação a respeito da requisição anterior. Sendo assim, dizemos que uma requisição HTTP é por padrão **Stateless** (não mantém o estado entre requisições). No entanto, o servidor gera um número de identificação para cada requisição realizada, e essa identificação é única para cada cliente.

Aplicações *Web*, por natureza, diferentemente de aplicações *Desktop*, não possuem uma “instância”, portanto não possuem um espaço de memória onde se podem guardar variáveis que serão usadas durante a execução do processo – no caso de uma visita a uma aplicação *web* durante uma sessão.

Uma forma de preservar informações, levando em consideração a natureza desconectada de solicitação ou resposta da *web*, seria utilizar formulários e páginas dinâmicas: por exemplo, o usuário seleciona um

produto e, ao postar para a página de outra categoria de produtos, o código do produto selecionado deveria ser postado junto, preservado na página como um campo oculto, ao longo de várias requisições.

Muitas vezes é necessário preservar em memória uma informação sensível que não poderia ser postada e uma reposta sem oferecer riscos, como o login e senha do usuário. Estes casos requerem que a informação seja preservada durante toda a visita do usuário ao site, entre as chamadas a diversas páginas dele.

O Container Java oferece um sistema robusto e sofisticado de controle de sessão, que permite armazenar praticamente qualquer tipo de objeto na memória do servidor, preservando o seu valor durante um limite de tempo que é renovado a cada requisição.

A cada requisição de página emitida pelo mesmo usuário através da mesma conexão e do mesmo navegador (situação esperada na navegação normal em um site), o tempo de vida da sessão é renovado e os valores armazenados são mantidos. Após o limite de tempo ter se esgotado, o servidor descarta as informações de sessão, para preservar seus recursos.

Modelo do controle de sessão

A sessão é gerada no servidor, como ilustra a Figura 6.2a e armazenada no cliente em forma de cookie, em um campo oculto ou na URL, como mostra a Figura 6.2b.

Figura 6.2a: Geração de sessão no servidor

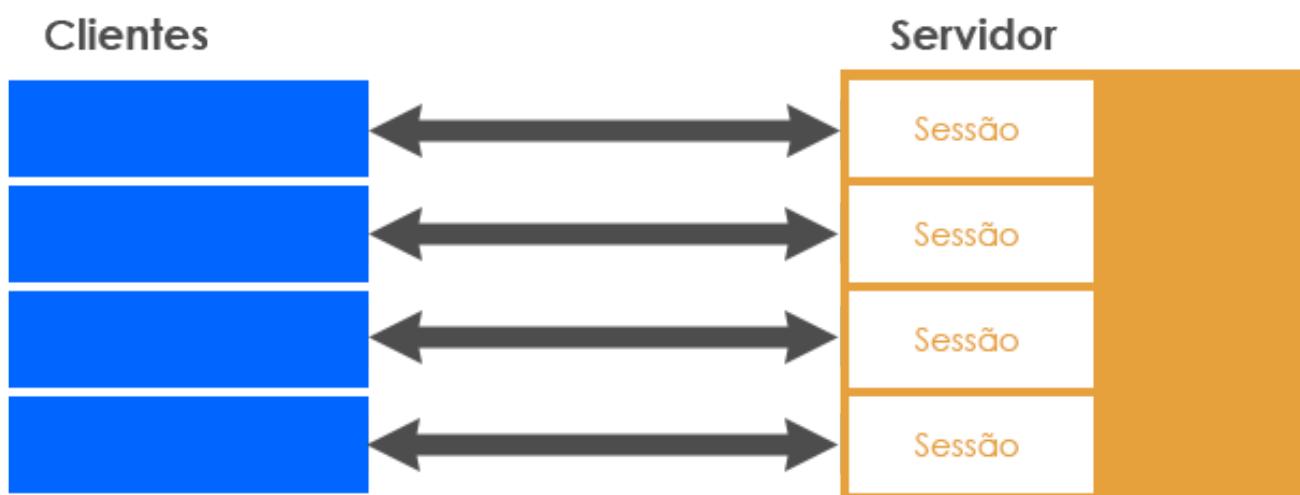
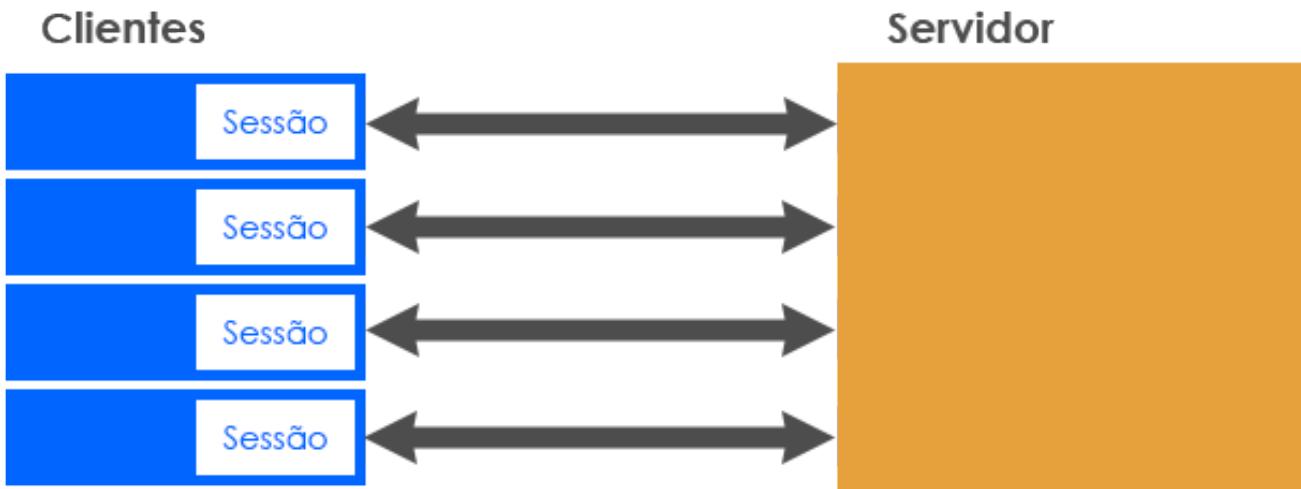


Figura 6.2b: Transferência da sessão para o cliente

O servidor fornece um identificador para a sessão (**ID da sessão**) que é enviada para o cliente. Na próxima vez que o mesmo cliente realizar outra requisição, o ID é lido e enviado juntamente com a requisição, de forma que o servidor reconhece o usuário. Assim, a cada requisição, a informação desejada é transferida para o servidor e, então, é enviada de volta para o cliente.

A classe HttpSession

A sessão é implementada no *container Web* através da interface:

`javax.servlet.http.HttpSession`.

O *container* cria um objeto **HttpSession** para um usuário quando ele inicia a aplicação. Há um **HttpSession** para cada usuário ativo.

Se o usuário não executar nenhuma ação por um determinado período de tempo, o servidor assume que o usuário está inativo e invalida a sessão por *timeout*. O tempo de permanência da sessão é configurável, tanto no *deployment descriptor* como de forma programática.

A tabela 6.3 mostra os principais métodos da classe **HttpSession**.

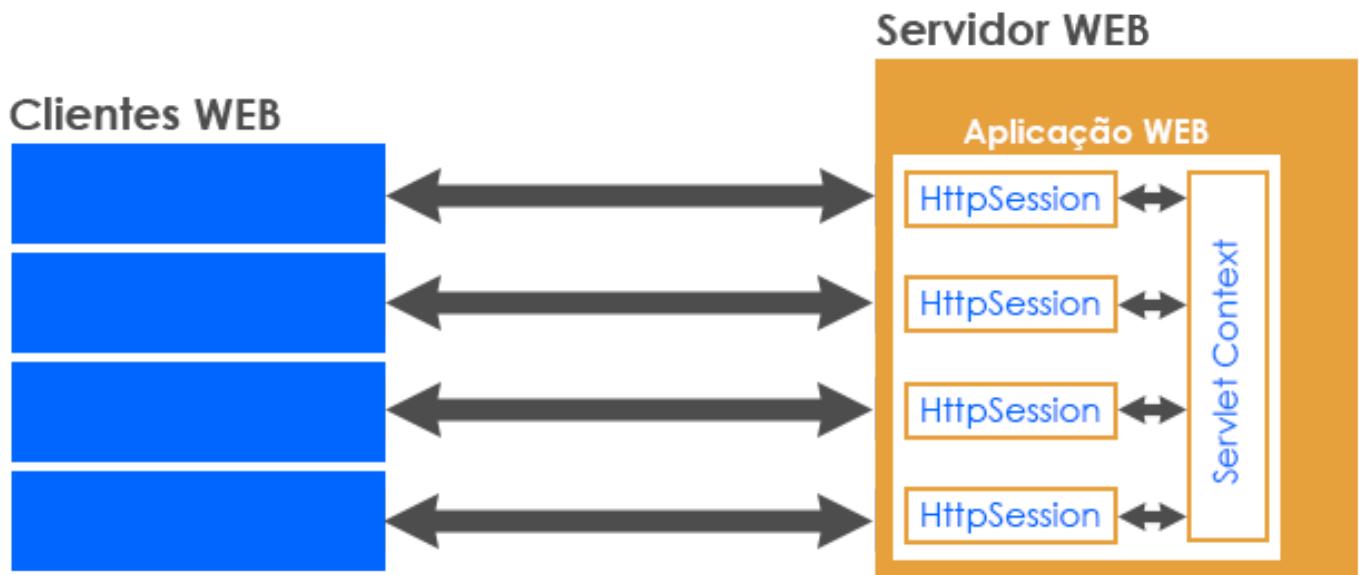
Tabela 6.3: Principais métodos da classe **HttpSession**

Método	Descrição
<code>void setAttribute(String key, Object value)</code>	Associa um valor de atributo com um nome.
<code>Object getAttribute(String key)</code>	Recupera o valor do atributo associado com a chave.

Enumeration getAttributeNames()	Recupera os nomes de todos os atributos associados com o objeto.
String getId()	Retorna o Id da sessão.
long getCreationTime()	Retorna a hora na qual a sessão foi criada.
long getLastAccessedTime()	Retorna a última vez que uma solicitação associada com a sessão foi recebida.
int getMaxInactiveInterval()	Retorna o tempo máximo (em segundos) entre solicitações pelo qual a sessão será mantida.
void setMaxInactiveInterval(int time)	Define o tempo máximo (em segundos) entre solicitações pelo qual a sessão será mantida.
boolean invalidate()	Descarta a sessão, liberando quaisquer objetos armazenados como atributos.

Dentro da sessão podem ser armazenados parâmetros e atributos que são visíveis somente dentro de uma sessão de usuário, como pode ser visto na Figura 6.3.

Figura 6.3: – Escopos do contexto da Sessão e do Contexto



Por padrão, a sessão é armazenada no cliente através de *cookies*. É possível que os *cookies* estejam desabilitados no cliente. Neste caso, a sessão pode ser armazenada em campos ocultos ou através do mecanismo de reescrita de URL.

Campos ocultos

Campos ocultos são acessados pelo formulário, como outros elementos. Para o servidor não existe diferença entre campos visíveis ou ocultos, daí sua possibilidade de utilização. Um exemplo de utilização é:

```
<input type="hidden" name="cmpHidden" value="true" >
```

A desvantagem é que ele só funciona para uma sequência de formulários gerados dinamicamente, e a técnica falha se houver um erro antes que os dados sejam permanentemente armazenados em algum lugar.

Reescrita de URL

Nem todos os navegadores suportam *cookies* e, mesmo que suportem, podem estar configurados para não armazenarem-nos. Para contornar este problema, o *servlet* pode ser configurado para usar reescrita de URL, como alternativa de controlar sessão.

Com esse procedimento, o servidor adiciona informações extras dinamicamente na URL. Normalmente esta informação é a ID de sessão (uma ID exclusiva associada a um objeto **HttpSession** quando ele é criado).

Podemos usar o método **HttpServletResponse.encodeURL()** para codificar a ID de sessão dentro de uma resposta, como um parâmetro adicional.

Por exemplo:

```
out.println("<form action=' " + response.encodeURL("/JSPPage" + " '>");
```

adiciona a ID de sessão na URL de ação do formulário, como segue:

```
http://localhost:8080/execicio02/JSPPage;jsessionid=30F76DE281982B57B4BEFC2434B6F5E
```

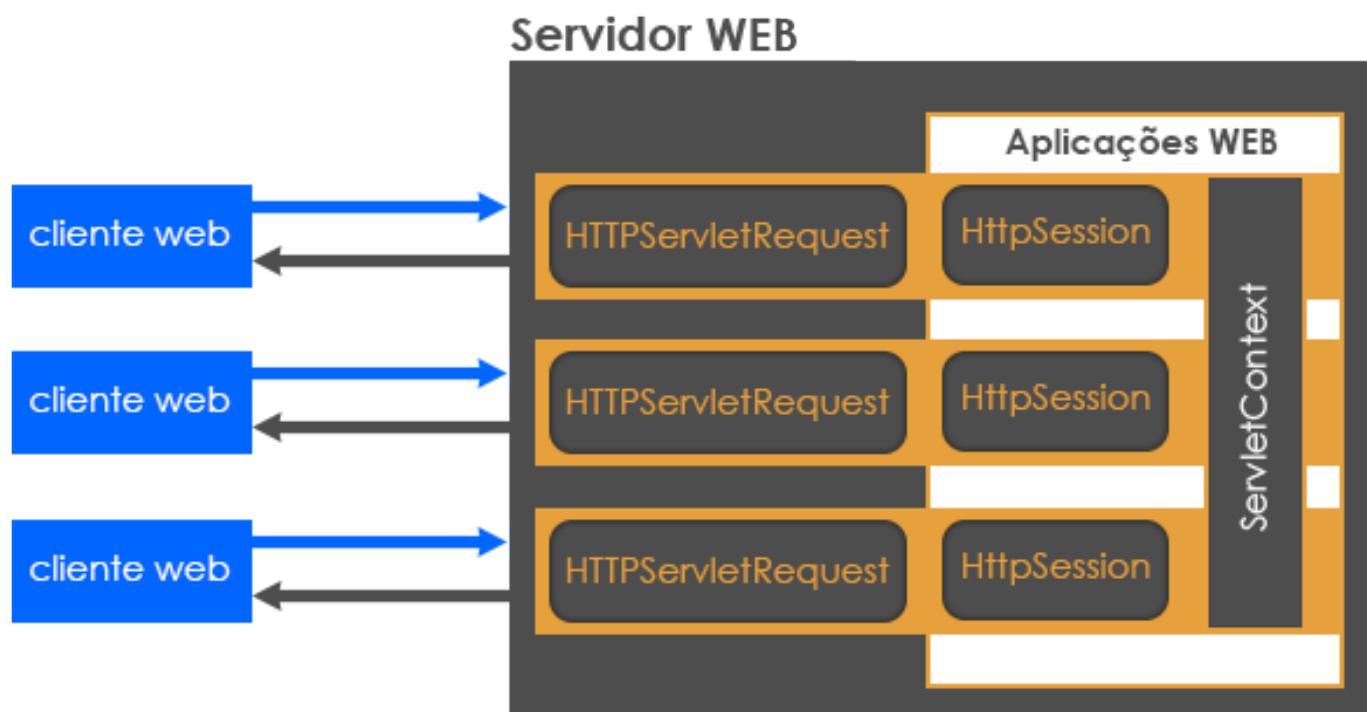
Escopo de armazenamento de dados na memória do servidor

É possível manipularmos atributos em:

- requisição (request);
- sessão (session);
- contexto (application).

Cada um tendo seus benefícios, aplicações e vantagens. A Figura 6.6 ilustra estes três componentes no servidor.

Figura 6.6: Container Web com os objetos Request, Session e Context



A Tabela 6.6 apresenta o tempo de vida para cada situação.

Tabela 6.6: Escopo de requisições

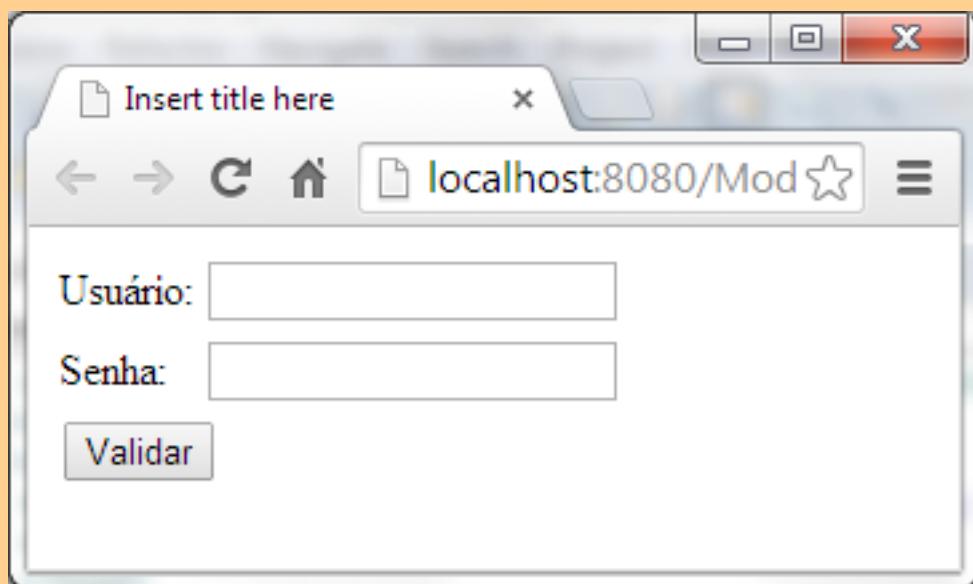
Escopo	Tempo de vida	Quando é criado	Quando é destruído
<i>ServletContext</i>	<i>Enquanto a aplicação estiver no ar</i>	<i>Quando a aplicação ou o servidor é inicializado</i>	<i>Quando a aplicação ou o servidor é desativado</i>
<i>HttpSession</i>	<i>Enquanto o usuário estiver logado na aplicação</i>	<i>Na primeira requisição efetuada pelo usuário</i>	<i>Quando invalidado por timeout ou programaticamente</i>
<i>HttpServletRequest</i>	<i>Durante o processamento de uma requisição</i>	<i>A cada nova requisição</i>	<i>Após a resposta ser retornada</i>

Exercícios do Módulo 2

Exercício 1: Formulário para validação de usuário

Iniciaremos nossa aplicação através de uma série de exercícios. Neste exercício criaremos um formulário para fornecer os dados do login de um usuário. Para tanto, siga os passos a seguir:

1. Criar um novo projeto no Eclipse chamado Modulo02_Exercicio01
2. Neste projeto, incluir um novo arquivo HTML chamado login.html (clique com o botão direito do mouse no projeto, e selecione New > HTML file).
3. O formulário login.html deve ser semelhante ao mostrado nesta ilustração:



Para este formulário, escrever o código a seguir:

```

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<form action="login" method="post">
  <table>
    <tr>
      <td>Usuário:</td>
      <td><input type="text" name="usuario" size="20"/></td>
    </tr>
    <tr>
      <td>Senha:</td>
      <td><input type="password" name="senha" size="20"/></td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="submit" value="Validar"/>
      </td>
    </tr>
  </table>
</form>
</body>
</html>

```

4. Executar a aplicação para visualizar o formulário no browser.

Exercício 2: Definição do servlet para receber os dados do formulário

Neste exercício definiremos o servlet que receberá os dados do formulário desenvolvido no Exercício 1.

1. Criar um servlet com as configurações:

- a. Classe: ServletLogin
- b. Nome: ServletLogin
- c. url mapping: /login

2. Ao criar o servlet, definir dois parâmetros de inicialização, com os valores:

	nome	valor
parâmetro 1	user	admin
parâmetro 2	pwd	admin

3. No método doGet() escrever a seguinte instrução:

response.sendRedirect("login.html");

4. Quando as informações do formulário são enviadas para o servlet, o método doPost() as recebe, pois na definição do formulário temos o atributo method= "post". Então, no método doPost() escrever o código para receber estes dados e compará-las com valores definidos como parâmetro de inicialização do servlet:

```

PrintWriter out = response.getWriter();
response.setContentType("text/html");

//parâmetros do formulário
String usuario = request.getParameter("usuario");
String senha = request.getParameter("senha");

//parâmetros de inicialização do servlet
String user = this.getServletConfig().getInitParameter("user");
String pwd = this.getServletConfig().getInitParameter("pwd");

if(usuario.equals(user) && senha.equals(pwd)){
    out.print("Usuário validado!");
}
else {
    out.print("Usuário ou senha inválidos");
}

```

5. Executar a aplicação a partir do formulário, fornecer valores para o usuário e para a senha, e verificar o resultado.

Exercício 3: Armazenamento de dados na sessão

Neste exercício os dados do formulário serão armazenados em sessão se estiverem corretos, ou seja, se forem iguais aos parâmetros de inicialização.

6. Alterar o código do método doPost() do servlet ServletLogin de forma a armazenar as informações na sessão:

```

PrintWriter out = response.getWriter();
response.setContentType("text/html");

//parâmetros do formulário
String usuario = request.getParameter("usuario");
String senha = request.getParameter("senha");

//parâmetros de inicialização do servlet
String user = this.getServletConfig().getInitParameter("user");
String pwd = this.getServletConfig().getInitParameter("pwd");

if(usuario.equals(user) && senha.equals(pwd)){

    HttpSession sessao = request.getSession();
    sessao.setAttribute("jusuario", usuario);

    out.print("Usuário validado!");
}
else {
    out.print("Usuário ou senha inválidos");
}

```

7. Alterar o código do método doGet() de forma a realizar a seguinte tarefa:

- Se houver um usuário armazenado na sessão, apresentar uma mensagem de boas-vindas, incluindo o nome do usuário.
- Se não houver um usuário validado, apresentar uma mensagem informando que não existe usuário autenticado e um link para retornar para a tela de login:

```
PrintWriter out = response.getWriter();
response.setContentType("text/html");
HttpSession sessao = request.getSession();

String usuario = (String)sessao.getAttribute("jusuario");
if(usuario == null){
    out.print("Nenhum usuário autenticado!<br/>");
    out.print("<a href='login.html'>Voltar para login</a>");
}
else {
    out.print("Seja benvindo(a), " + usuario);
}
```

8. Executar a aplicação a partir do formulário, fornecer valores para o usuário e para a senha, e verificar o resultado.

Módulo 3 - Java Server Pages (JSP)

Aula 7 – Fundamentos de JSP

Definição de JSP



A tecnologia Java Server Pages (JSP) é uma extensão da tecnologia de servlets. Ambas são tecnologias Web JEE que utilizam o conceito de *container*, responsável por gerar conteúdo dinâmico via protocolo HTTP e fornecer o *runtime* para aplicações, entre outras tarefas.

Assim sendo, a arquitetura geral das páginas JSP tem muito em comum com os servlets, tendo em vista que a especificação JSP é definida como uma extensão da API *Servlet*. Além das classes e interfaces para programar servlets (pacotes `javax.servlet` e `javax.servlet.http`), as classes e interfaces específicas à programação de Java Server Pages encontram-se nos pacotes `javax.servlet.jsp`.

Atuando na camada de apresentação, as páginas JSP utilizam a tecnologia Java no formato de servlets no lado do servidor para a criação de conteúdo dinâmico, junto às *tags* HTML para o conteúdo estático.

Quando um servidor compatível com JSP recebe a primeira solicitação para uma JSP, o *container* de JSP traduz essa JSP em um *servlet* Java que trata a solicitação atual e as solicitações futuras para a JSP. Se houver algum erro na compilação do novo *servlet*, esses erros resultam em erros em tempo de tradução.

Alguns containers de JSP traduzem as JSPs para servlets em tempo de instalação. Isso elimina o *overhead* de tradução para o primeiro cliente que solicita cada JSP

Podemos observar as seguintes características do JSP:

- Separação do conteúdo estático do dinâmico: como vimos, a lógica de geração de conteúdo dinâmico é mantida separada das *tags* HTML responsáveis pela interface para o usuário. A parte lógica é encapsulada em componentes Java do tipo JavaBeans, que são utilizados pelas páginas JSP através de *scriptlets* ou *tags* especiais, chamadas *taglibs*.
- “Escreva uma vez, rode em qualquer lugar”: como a tecnologia JSP é uma extensão da plataforma Java, as páginas JSP têm a vantagem da independência de plataforma.
- Diversos formatos: possibilidade de implementação de linguagens como HTML, XML, DHTML, WML, etc.
- Utilização de código Java: é possível utilizar os chamados scriplets, que são trechos de código Java puro inseridos dentro da página JSP, tendo assim as vantagens tanto da Orientação a Objetos quanto de Java.

Arquitetura JSP

Há componentes-chaves para JSPs, que são diretivas, ações e *scriptlets*, assim descritos:

- As **diretivas** são mensagens para o *container* de JSP que permite ao programador especificar as configurações de uma página, incluir conteúdo a partir de outros recursos e especificar bibliotecas de marcas personalizadas para utilização em uma JSP.
- As **ações** encapsulam funcionalidades em marcas predefinidas que os programadores podem incorporar a uma JSP. As ações são frequentemente realizadas com base nas informações enviadas ao servidor como parte de uma solicitação feita pelo cliente. Eles também podem criar objetos Java para a utilização em *scriptlets* JSP.
- Os **scriptlets** permitem inserir código Java que interaja com os componentes em uma JSP (e, possivelmente, outros componentes de aplicativos Web) para realizar processamento de solicitações.

Durante o processamento de uma requisição, várias páginas JSP podem estar envolvidas. Diversos objetos podem ser criados, tanto usando diretivas JSP, como por ações definidas na página JSP ou ainda por trechos de código Java embutidos na página JSP.

Qualquer objeto criado pode ser associado com um atributo de escopo, que define onde existe uma referência ao objeto e quando a referência é removida. Assim sendo, temos seguintes atributos:

- **page**: objetos acessíveis somente nas páginas em que foram criados;
- **request**: objetos acessíveis nas páginas que processam a mesma requisição;
- **session**: objetos acessíveis nas páginas que pertençam à mesma sessão em que os objetos foram criados;
- **application**: objetos acessíveis nas páginas que pertençam a mesma aplicação.

Diretivas

As diretivas são elementos da página que não dependem de quaisquer solicitações, e funcionam como um pré-processamento para a página. A sintaxe básica de uma diretiva é:

```
<%@ diretiva %>
```

As diretivas são:

- **page** – associada ao comportamento da página atual;
- **include** – usada para incluir arquivos texto ou jsp no documento atual;
- **taglib** – usada para definir uma biblioteca de tags personalizadas.

Diretiva page

Considerada a mais extensa das diretivas. Os principais atributos são descritos na tabela 7.3.1.

Tabela 7.3.1: Atributos da diretiva *page*

Atributo	Descrição
language	Especifica a linguagem a ser usada na página JSP. Como default, a linguagem utilizada é Java. Pode ser especificada uma outra, desde que compatível com Java.
extends	Empregado para especificar uma superclasse para a página JSP.
import	Importa para a página atual um pacote de classes necessário ao seu funcionamento.
session	Determina se a página fará parte da sessão atual. Seus valores são booleanos.
buffer	Especifica o tamanho do buffer que armazenará os dados enviados pelo objeto <i>response</i> . O valor padrão é 8 KB.
autoFlush	Define se os dados devem ser enviados automaticamente para o cliente quando o buffer estiver cheio.
isThreadSafe	Opera com valores booleanos, e o valor <i>true</i> informa ao <i>container</i> JSP que a página pode lidar com solicitações múltiplas simultaneamente. O valor <i>false</i> informa ao <i>container</i> JSP que as solicitações devem ser todas respondidas, mas apenas uma depois da outra e na ordem em que foram recebidas.
info	Empregado para adicionar informações diversas à página.
isErrorPage	Valor usado na página para indicar se esta deve ser apresentada caso algum erro tenha ocorrido em uma outra página da aplicação.
errorPage	Define um endereço para uma página de erro.
contentType	Define o tipo MIME para o conteúdo a ser enviado na resposta. O valor padrão é "text/html".
pageEncoding	Define o conjunto codificador de caracteres para a página JSP em questão. O valor padrão é "ISO-8859-1".

Exemplos:

```
<%@ page import="java.sql.*" %>
<%@ page import="java.sql.*,java.io.*" %>
<%@ page errorPage="erroGeral.jsp" %>
```

Formulários

Um formulário consiste em um ou mais campos de entrada de dados, descritos pela tag **<input>**, em que o atributo **type** permite a criação de campos de texto, senhas, caixa e lista de seleção, menus, botão de opção, entre outros. A forma geral é:

```
<input type=... name=... value=... size=... maxlength=...>
```

Formulários são usados para **enviar dados pela web**. Sozinhos, formulários são inúteis, eles devem estar vinculados a um programa que irá processar seus dados. Esses programas podem ser elaborados em diversas linguagens, incluindo o ASP.Net

Os elementos de um formulário são:

- **form**: define o formulário. Dentro desse elemento existe um atributo obrigatório, o **action**, que diz o endereço do programa para onde os dados do formulário serão enviados.
- O atributo opcional **method** diz a forma em que o formulário será enviado, e pode ter o valor **get** (que é o *default*) ou **post**. Frequentemente usa-se **post** que esconde a informação (**get** manda a informação através da URL).

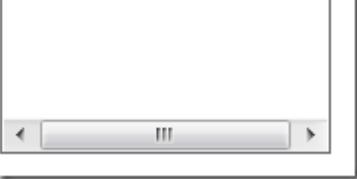
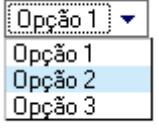
Um elemento form vai se parecer com:

```
<form action="script.jsp" method="post">
    elementos
</form>
```

Geralmente os elementos de um formulário são inseridos com a tag **<input .. >** e o tipo do elemento. A tabela 7.4 apresenta uma lista com o conjunto de elementos usados em um formulário.

Tabela 7.4: Elementos de um formulário

Elemento	Descrição	Visualização
<input type="text"/>	É a caixa de texto padrão.	Campo: <input type="text"/>
<input type="password"/>	Similar à caixa de texto , mas o que for digitado pelo usuário não vai ser visível.	Senha: <input type="password"/>
<input type="checkbox"/>	é uma checkbox , que pode ser marcada e desmarcada pelo usuário.	Checkbox: <input type="checkbox"/>
<input type="radio"/>	é similar a um checkbox , mas o usuário só pode selecionar um radio button em um grupo.	Radio 1: <input checked="" type="radio"/> Radio 2: <input type="radio"/>
<input type="file"/>	é um campo que permite procurar e escolher um arquivo em seu computador para permitir ao usuário fazer <i>upload</i> de arquivos.	Arquivo: <input type="file"/> Browse...

<code><input type="submit"/></code> <code><input type="button"/></code> <code><input type="image"/></code>	submit é um botão que quando clicado envia o formulário. O button necessita de código Javascript. O image utiliza uma imagem ao invés de texto.	
<code><input type="reset"/></code>	é um botão que quando clicado vai limpar os campos do formulários para seus valores default.	
<code><input type="hidden"/></code>	é um campo que não aparece na tela e é usado para passar informação como o nome da página em que o usuário está, o endereço de e-mail para o qual o post vai ser enviado ou qualquer outro dado que não seja inserido pelo usuário.	
<code><input type="textare</code></code>	Uma textarea é, basicamente, uma textbox grande.	Textarea: 
<code><select><option></select></code>	A tag select trabalha junto com a tag option pra criar caixas de seleção drop-down.	

Aula 8 - Objetos implícitos no JSP

Definição de objetos implícitos

É possível criar, dentro de *scriptlets* na página JSP, instâncias de uma classe Java e manipulá-las a fim de produzir um conteúdo dinâmico. Por exemplo, podemos criar um objeto de uma classe que acessa uma base de dados e então usar métodos desse objeto para exibir na página uma consulta ou transação com a base de dados. Ou seja, através da manipulação desse objeto, quer seja acessando seus métodos e suas variáveis, podemos gerar conteúdo dinâmico para a página JSP.

Além de objetos como esses, que estão completamente sob o controle do programador, o *container JSP* se encarrega de instanciar automaticamente, durante a execução de uma página JSP, alguns objetos. Tais objetos podem ser usados dentro da página JSP e são conhecidos como **Objetos Implícitos**.

Assim como todo objeto em Java, cada objeto implícito é uma instância de uma classe ou interface e segue uma API correspondente. A Tabela 8.1 apresenta um resumo dos objetos implícitos disponíveis em JSP, suas respectivas classes ou interfaces e uma pequena descrição do objeto.

Tabela 8.1: Objetos Implícitos JSP

Objeto	Classe ou Interface	Descrição
page	<code>javax.servlet.jsp.HttpJspPage</code>	Instância de <i>servlet</i> da página.
config	<code>javax.servlet.ServletConfig</code>	Dados de configuração de <i>servlet</i> .
request	<code>javax.servlet.http.HttpServletRequest</code>	Dados de solicitação, incluindo parâmetros.
response	<code>javax.servlet.http.HttpServletResponse</code>	Dados de resposta.
out	<code>javax.servlet.jsp.JspWriter</code>	Fluxo de saída para conteúdo da página.
session	<code>javax.servlet.http.HttpSession</code>	Dados de sessão específicos de usuário.
application	<code>javax.servlet.ServletContext</code>	Dados compartilhados por todas as páginas de aplicação.
pageContext	<code>javax.servlet.jsp.PageContext</code>	Dados de contexto para execução da página.
exception	<code>javax.lang.Throwable</code>	Erros não capturados ou exceção.

Declarações de variáveis

Um conjunto de variáveis pode ser declarado separadamente em uma página, desde que estejam contidas nas tags:

```
<%! variáveis %>
```

Exemplo:

```
<%!
    int idade;
    double salario;
    String endereço;
%>
```

Expressões ou Scriptlets

São mais largamente usadas em páginas JSP. Todo código JSP deve ser escrito dentro das tags.

```
<% expressões %>
```

Exemplos:

```
<%
    String nome = request.getParameter("txtNome");
    int codigo = request.getParameter("txtCodigo");
%>
<%
    if (idade < 18) {
        out.println("<b>Você é menor de idade</b>");
    }
%>
```

```
<%
    if (idade < 18) {
        <b>Você é menor de idade</b>
    }
%>
```

Ações

São informações empregadas na fase de execução. Elas podem ser separadas em dois grupos:

[1] Ações padrões

[2] Ações personalizadas

As ações padrão são definidas pela especificação JSP e, as personalizadas, em tags personalizadas. Ambas empregam a sintaxe XML.

As principais ações padrão são mostradas na tabela 8.4.

Tabela 8.4: Ações padrão

Ação	Sub elemento de	Descrição
<jsp:useBean>		Define uma instância de uma classe chamada de JavaBean.
<jsp:setProperty>	<jsp:useBean>	Atribui valores para as propriedades de um JavaBean.
<jsp:getProperty>	<jsp:useBean>	Obtém valores das propriedades de um JavaBean.
<jsp:include>		Insere o conteúdo de uma nova página na página atual.
<jsp:forward>		Envia o conteúdo e o controle da página atual para outra página
<jsp:param>	<jsp:params>, <jsp:forward>, <jsp:include>	Atribui parâmetros a serem manipulados pela ação pai.

<jsp:plugin>		Define uma instância de um componente externo. O caso mais comum é o applet.
<jsp:params>	<jsp:plugin>	Relaciona os parâmetros a serem enviados para uma ação plugin.
<jsp:fallback>	<jsp:plugin>	Executada quando um plugin não pode ser iniciado.

Exemplo:

Suponha que se queira incluir na página atual, a página chamada **DadosPessoais.jsp** com o conteúdo:

```
<%
    String código = request.getParameter("txtCodigo");
    String nome = request.getParameter("txtNome");
    String email = request.getParameter("txtEmail");
%>
```

Na página atual há:

```
<jsp:include page="DadosPessoais.jsp">
    <jsp:param name="txtCodigo" value = "100" />
    <jsp:param name="txtNome" value = "João" />
    <jsp:param name="txtEmail" value = "joão@ninguem.com" />
</jsp:include>
```

Observe o fechamento das tags. No caso da tag **include**, houve uma linha para fechamento, com um conteúdo entre a abertura e o fechamento. Neste caso a tag é chamada de tag com conteúdo não vazio ou de corpo não vazio.

No caso da tag **jsp:param**, o fechamento ocorreu na mesma linha. Esta tag é chamada de **tag de corpo vazio**, ou sem conteúdo.

Passando parâmetros para uma página JSP via URL

Normalmente uma página JSP recebe os dados de um formulário. É possível executar uma página JSP sem a necessidade de um formulário.

Suponha uma página chamada **Cadastro.jsp** com as expressões abaixo:

```
<%
    String código = request.getParameter("txtCodigo");
    String nome = request.getParameter("txtNome");
    String email = request.getParameter("txtEmail");
%>
```

Normalmente seria necessário um formulário com os componentes **txtCodigo**, **txtNome** e **txtEmail**. Para chamá-la no *browser* diretamente, deve-se inserir a informação abaixo:

<http://localhost:8080/Cadastro.jsp?txtCodigo=10&txtNome=João&txtEmail=joao@ninguem.com.br>

No que se segue, alguns exemplos são apresentados.

- **Exemplo 1:** envio de dados usando o método GET.

Assumindo que o programa a seguir se chame arquivo.jsp e que faça parte de um contexto chamado arquivos, seu acesso é realizado através da seguinte url:

<http://localhost:8080/arquivos/arquivo.jsp?txtNome=Carlos&txtIdade=10&txtSalario=1000>

```
<%@ page import="java.io.* , java.util.*" %>
<%
    String nome = request.getParameter("txtNome");
    String id = request.getParameter("txtIdade");
    String sal = request.getParameter("txtSalario");
    int idade = Integer.parseInt(id);
    double salario = Double.parseDouble(sal);
    String linha = nome + ";" + id + ";" + sal + "\r\n";
    try {
        FileWriter f = new FileWriter("c:/arquivo.txt",true);
        f.write(linha);
        f.close();
    }
    <center><h2>Dados gravados com sucesso !</h2></center>
%>
%}
catch( Exception e) {
%>
<h1>OCORREU UM ERRO AO TENTAR GRAVAR O ARQUIVO</h1>
<%
}>
```

- **Exemplo 2:** No exemplo a seguir, não é necessário informar nenhum parâmetro: o programa, chamado **abreArquivo.jsp**, se encarrega de abrir o arquivo e exibir seus dados no **browser**:

```

<%@ page import="java.io.*,java.util.*" %>
<%
    try {
        FileReader f = new FileReader("c:/arquivo.txt");
        BufferedReader b = new BufferedReader(f);
        do {
            String l = b.readLine();
            if (l == null) break;
            String [] s1 = l.split(";");
        } while(true);
        b.close();
        f.close();
    }
    catch(Exception e) {
%>
<h1>ERRO AO LER ARQUIVO !</h1>
<%
    }
%>

```

- **Exemplo 3:** envio de dados usando o método POST.

Neste caso é conveniente ter um formulário HTML para realizar o envio, como:

O formulário contém os seguintes campos:

- Nome: campo de texto
- Idade: campo de texto
- Salário: campo de texto
- Botão Enviar
- Botão Limpar

O código para o formulário é o seguinte:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=iso-8859-1" />
    <title>Untitled Document</title>
</head>
<body bgcolor="#FFCC99">
<form method="post" action="arquivo.jsp">
    <table width="425" border="0">
        <tr>
            <td width="83">Nome:</td>
            <td width="332">
                <input name="txtNome" type="text" id="txtNome" size=40/>
            </td>
        </tr>
        <tr>
            <td>Idade:</td>
            <td>
                <input name="txtIdade" type="text" id="txtIdade" />
            </td>
        </tr>
        <tr>
            <td>Salário:</td>
            <td>
                <input name="txtSalario" type="text" id="txtSalario" />
            </td>
        </tr>
        <tr>
            <td colspan="2">
                <input name="btnEnviar" type="submit" id="btnEnviar" value="Enviar" />
                <input name="btnLimpar" type="reset" id="btnLimpar" value="Limpar" />
            </td>
        </tr>
    </table>
</form>
</body>
</html>
```

Aula 9 - JavaBeans

Bean

JavaBeans consiste em uma tecnologia que permite escrever componentes Java acessíveis como propriedades. São necessárias classes Java para que funcionem corretamente e estas classes devem:

- possuir um construtor padrão
- implementar a interface Serializable (se não implementar, funcionará normalmente, mas não constitui formalmente um JavaBeans)

JavaBeans são, portanto, classes Java que encapsulam informações a serem executadas em páginas JSP.

Um objeto instanciado a partir destas classes é chamado de **bean**. Veja o exemplo a seguir.

Classe Java:

```
package com.javabeans;

import java.io.Serializable;

public class Pessoa implements Serializable{
    private String nome;
    private int idade;
    private char sexo;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }

    public char getSexo() {
        return sexo;
    }

    public void setSexo(char sexo) {
        this.sexo = sexo;
    }
}
```

Nesta classe, cada par setter / getter representa no JSP, uma propriedade, cujo nome é obtido removendo-se o prefixo set e/ou get, e passando a primeira letra para minúsculo.

Exemplo:

Os métodos

```
public String getName() {
    return nome;
}

public void setName(String nome) {
    this.nome = nome;
}
```

representam a propriedade **nome** na página JSP.

A utilização desta propriedade requer que uma instância da classe seja criada, mas a nomenclatura JavaBeans não utiliza código Java. Os elementos usados na página JSP que manipulam beans são:

```
<jsp:useBean >
<jsp:setProperty >
<jsp:getProperty >
```

O elemento **<jsp:useBean >** é usado para criar o objeto:

```
<jsp:useBean id="pessoa" class="com.javabeans.Pessoa" scope="request"/>
```

Nesta instrução, temos:

- **id** – declara o bean, ou seja, o objeto
- **class** – classe a partir da qual o bean é criado
- **scope** (opcional) – indica o escopo do bean na aplicação. Os possíveis valores são: request, page, session e application.

Atribuindo valores para as propriedades

As instruções a seguir mostram como atribuir valores para as propriedades, usando o objeto **pessoa** criado na instrução anterior.

```
<jsp:setProperty name="pessoa" property="nome" value="Jose"/>
<jsp:setProperty name="pessoa" property="idade" value="25"/>
<jsp:setProperty name="pessoa" property="sexo" value="m"/>
```

- **name** – indica o bean, representado pelo atributo id no elemento **<jsp:useBean>**.
- **property** – nome da propriedade. Implicitamente, este elemento executa o setter correspondente a esta propriedade.

- **value** – valor a ser atribuído à propriedade. Pode ser fixo ou proveniente de um parâmetro (formulário ou URL).

É possível atribuir valores variáveis para as propriedades, de duas formas. A primeira é:

```
<jsp:setProperty name=>pessoa</> property=>nome</> value=><%= request.getParameter("nome")%></>
```

A segunda forma é usando o objeto implícito param através de “**Expression Language (EL)**”:

```
<jsp:setProperty name=>pessoa</> property=>nome</> value=>${param.nome}</>
```

Obtendo valores das propriedades

A leitura das propriedades é realizada através do elemento `<jsp:getProperty>`. Este elemento encapsula a funcionalidade da instrução `out.print()`.

```
Nome: <jsp:getProperty name=>pessoa</> property=>nome</>
Idade: <jsp:getProperty name=>pessoa</> property=>idade</>
Sexo: <jsp:getProperty name=>pessoa</> property=>sexo</>
```

Aula 10 - JSTL

Definição de JSTL

JSTL é uma API Java que contém um conjunto de tags agrupadas de acordo com sua aplicabilidade. O objetivo é simplificar o desenvolvimento de páginas JSP, eliminando boa parte de código escrito em *scriptlets*.

A JSTL encapsulou, em tags simples, toda a funcionalidade que diversas páginas web precisam, como controle de laços (for), comandos de decisão (if .. else), manipulação de dados xml, internacionalização da aplicação, além de acesso e manipulação de bancos de dados.

Enfim, JSTL foi a forma encontrada de padronizar o trabalho de programação em JSP.

Na data da elaboração deste material, a API JSTL foi obtida em:

```
https://jstl.java.net/download.html
```

Expression Language (EL)

JSTL faz uso extensivo de **Expression Language (EL)**. EL representa um conjunto de operadores e objetos implícitos, semelhantes àqueles definidos no JSP, mas com o objetivo de tornar possível o uso do JSTL.

Por padrão `${algumaCoisa}` procura por uma variável com o nome **algumaCoisa** em algum dos escopos, porém podemos deixar explícito o escopo usando um nome de variável que representa cada um dos escopos.

Por exemplo, `${sessionScope.usuarioLogado}` ou `${pageScope.pessoas}`.

sessionScope e **pageScope** são variáveis implícitas do contexto EL.

A tabela 10.2a exibe os contextos implícitos disponíveis em EL.

Tabela 10.2a: Contextos disponíveis em EL

Objeto	Descrição
pageScope	Define as variáveis que estão no escopo da página.
requestScope	Define as variáveis que estão no escopo da requisição.
sessionScope	Define as variáveis que estão no escopo da sessão.
applicationScope	Define as variáveis que estão no escopo da aplicação.
param	Mapa que contém o valor dos parâmetros da requisição.
paramValues	Mapa que contém o valor dos parâmetros da requisição como <i>array</i> de <i>strings</i> .
header	Mapa que contém o nome e o valor dos <i>headers</i> da requisição.
headerValues	Mapa que contém o nome e o <i>array</i> de valores dos <i>headers</i> da requisição.
cookie	Mapa do nome dos <i>cookies</i> e o <i>cookie</i> .
initParam	Mapa com o nome dos parâmetros iniciais e seus valores.

Cada uma dessas variáveis representa um objeto Java. EL também possui operadores aritméticos, relacionais e lógicos. A Tabela 10.2b mostra quais são estes operadores e como usá-los.

Tabela 10.2b: Operadores EL

Operador	Operador EL	Descrição
+		Adição

-		Subtração
*		Multiplicação
/	div	Divisão
%	mod	Módulo (resto)
==	eq	Igual (equal)
!=	ne	Diferente (not equal)
<	lt	Menor que (lower than)
>	gt	Maior que (greater than)
<=	le	Menor ou igual (lower or qual)
>=	ge	Maior ou igual (greater ore qual)
&&	and	Verdadeiro se ambos operandos são verdadeiros; Falso caso contrário.
	or	Verdadeiro se um ou ambos operandos forem verdadeiros; Falso caso contrário.
!	not	Verdadeiro se operando for verdadeiro; Falso caso contrário.
	empty	Verdadeiro se o operando for nulo, uma string vazia, um mapa vazio ou uma lista vazia; Falso caso contrário.

Bibliotecas de tags

As bibliotecas de tags se dividem nas categorias:

- Core *manipulação de dados*
- SQL *manipulação de banco de dados*
- Format *formatação de dados*
- XML *configuração de arquivos XML*

Biblioteca Core

Propósitos:

1. Manipulação de variáveis de escopo e tratamentos de erros;
2. Facilidade no processamento condicional em páginas JSP;
3. Iteração entre coleções e dados;
4. Criação de links, importação e redirecionamento para URLs.

As principais tags contidas na biblioteca Core são:

`<c:out>`

```
<c:set>
<c:remove>
<c:catch>
<c:if>
<c:choose>
<c:forEach>
<c:forTokens>
<c:param>
<c:url>
<c:import>
<c:redirect>
<c:when>
```

<c:out>

Atributos:

value	valor a ser avaliado
default	valor (opcional) retornado caso value seja vazio ou nulo
escapeXml	flag (opcional) indicando se caracteres especiais devem ser convertidos

Utilização:

Para utilizar tags da biblioteca Core nas páginas JSP é necessário inserir no cabeçalho:

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

Exemplos de aplicação:

Especificando o prefixo “c”:

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

Apresentando uma mensagem no *browser*:

```
<c:out value="Iniciando o estudo do JSTL" />
```

<c:set>

Atributos:

var	nome da variável
value	valor a ser atribuído a variável
scope	escopo da variável (page, request, session ou application)
target e property	objeto e propriedade (ao invés de var)

Obtendo o valor de uma variável:

```
<c:set var="nome" value="José Maria" scope="page"/>
```

Apresentando o conteúdo da variável:

```
<c:out value="${nome}" />
```

Observe que o acesso a variável é realizado através do símbolo “\${ }”:

É possível receber valores de parâmetros obtidos através da URL:

```
<c:set var=>nome</var> value=>${param.nome}</value> />
```

O que é equivalente à expressão:

```
String nome = request.getParameter("nome");
```

<c:remove>

Observe as instruções a seguir:

```
<c:set var="idade" value="45"/>
<c:out value="${idade}"/>
<c:remove var="idade"/>
<c:out value="${idade}" escapeXml="false">
    A variável <b>idade</b> foi removida do escopo!
</c:out>
```

A variável **idade**, primeiro foi criada, apresentada, e depois, removida do escopo através da tag **<c:remove>**. O texto no corpo de **<c:out> ... </c:out>** é apresentado porque a variável não tem valor *default* nem existe mais.

<c:catch>

Atributos:

var	nome da variável
-----	------------------

Esse atributo é opcional e, se não colocado, a exceção será ignorada.

Define a variável que irá capturar a exceção:

```
<c:catch var="erro">
<c:set target="${usuario}" property="SeuNome" value="Zé"/>
</c:catch>
```

Exceção gerada:

```
<c:out value="${erro}">Nenhuma</c:out>
```

<c:if>

Atributos:

test	expressão a ser testada
var	variável (opcional) a receber o valor da expressão
scope	escopo (opcional) da variável
target e property	objeto e propriedade (ao invés de var)
Corpo da tag	bloco a ser executado se a expressão for verdadeira

A expressão a seguir verifica o nome do usuário:

```
<c:if test="${usuario == 'Vivaneide Rocha'}">
    Usuário cadastrado!
</c:if>
```

Ou:

```
<c:if test="${usuario == 'Vivaneide Rocha'}" var="acesso">
    Usuário cadastrado!
</c:if>
```

A variável acesso apresentará o resultado da pesquisa: true ou false.

<c:choose>, <c:when>, <c:otherwise>

Usado para condições de múltipla escolha.

Atributos:

test	condição a ser testada (<c:when>)
-------------	-----------------------------------

- <c:choose>:** tags <c:when> e <c:otherwise>
- <c:when> :** executado quando teste for verdadeiro
- <c:otherwise>:** executado caso nenhum <c:when> seja aplicado

```
<c:choose>
<c:when test="${usuario == 'Vivaneide Rocha'}">
    Usuário cadastrado!
</c:when>
<c:otherwise>
    Usuário não cadastrado!
</c:otherwise>
</c:choose>
```

Equivalente a um comando **if ... else**

<c:forEach>

Executa o corpo da tag várias vezes.

Atributos:

itens	coleção (opcional) a ser iterada.
var	variável a receber cada valor da iteração.
varStatus	variável a receber status de cada iteração.
begin e end	índice do primeiro e último passos da iteração.
step	frequência das iterações.

Apresenta o resultado de uma estrutura de repetição:

```
<c:forEach var="i" begin="1" end="10">
    <c:out value="${i}" /><br>
</c:forEach>
```

Apresenta o resultado de um formulário:

```
<c:forEach itens="${param.txtNome}" var="p">
    <c:out value="${p}"/>
</c:forEach>
```

A expressão **param.txtNome** é equivalente à **request.getParameter("txtNome")**;

<c:forTokens>

Interage com os símbolos de uma string:

Atributos:

itens	coleção (opcional) a ser iterada
delims	variável a receber cada valor da iteração
varStatus	variável a receber status de cada iteração
begin,end,varStatus e step	mesmo significado que em <c:forEach>

Apresenta o resultado de uma estrutura de repetição:

```
c:forTokens itens="nome;idade" delims=";" var="tok" varStatus="st">
    <c:out value="${st.count}"/><br>
    <c:out value="${tok}"/><br>
</c:forTokens>
```

O resultado dessa execução será:

```
1
nome
2
idade
```

<c:url>

Cria uma string representando uma URL.

Atributos:

value	valor da url
var	variável (opcional – caso omitido, a URL será impressa na página) que receberá a url

scope	(opcional) escopo da variável
context	(opcional) contexto da página

Apresenta o resultado de uma estrutura de repetição:

```
<c:url value="resultado.jsp" var="url">
    <c:param name="txtNome" value="Felipe" />
    <c:param name="txtEmail" value="felipe@felipe" />
</c:url>
<c:out value="${url}" />
```

O resultado dessa execução será:

```
resultado.jsp?txtNome=Felipe&txtEmail=felipe%40felipe
```

Aqui convém relembrar que os dados para um formulário podem ser enviados através da barra de endereços do *browser*. Por exemplo, seja o arquivo chamado **recebe.jsp**, que contém os campos nome e idade a serem preenchidos:

```
<%
    String nome = request.getParameter("txtNome");
    String email = request.getParameter("txtEmail");
    ...
%>
```

Os valores podem ser chamados assim:

```
recebe.jsp?txtNome=Felipe&txtEmail=felipe%40felipe
```

Biblioteca Sql

Propósitos:

1. Trabalhar com DataSources/Connections;
2. Realizar consultas;
3. Realizar inclusão, alteração ou exclusão.

As principais tags contidas na biblioteca Sql são:

```
<sql:query>
<sql:update>
<sql:transaction>
<sql:param>
<sql:setDataSource>
```

<sql:setDataSource>

Define um DataSource e o exporta como variável de escopo ou DataSource padrão

Atributos:

dataSource	origem do banco de dados (conexão)
driver/url/user/password	valores usados pelo DriverManager para obter a conexão
var	variável que conterá o DataSource
scope	(opcional) escopo da variável (request, page, session, application)

<sql:update>

Executa instruções SQL do tipo INSERT, UPDATE e DELETE.

Atributos:

sql	instrução a ser executada
dataSource	origem do banco de dados (conexão)
var	variável (opcional) que conterá o nº de linhas afetadas
scope	(opcional) escopo da variável (request, page, session, application)

Exemplo:

```
<sql:setDataSource
    var="dataSource" driver="sun.jdbc.odbc.JdbcOdbcDriver"
    url="jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};dbq=C:/jstl.mdb"
    scope="session" />
<sql:update dataSource="${dataSource}">
    INSERT INTO CLIENTES VALUES (5,'USER')
</sql:update>
```

<sql:query>

Executa uma instrução SELECT.

Atributos:

sql	instrução a ser executada
dataSource	origem do banco de dados (conexão)

maxRows	(opcional) nº máx. de linhas retornadas
startRow	(opcional) primeira linha retornada
var	variável (opcional) que conterá o nº de linhas afetadas
scope	(opcional) escopo da variável (request, page, session, application)

Exemplo:

```
<sql:setDataSource
    var="dataSource" driver="sun.jdbc.odbc.JdbcOdbcDriver"
    url="jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};dbq=C:/jst1.mdb"
    scope="session" />
<sql:query var = "users" dataSource="${dataSource}">
    select ID, NOME from CLIENTES
</sql:query>
<table border=1>
    <c:forEach var="row" itens="${users.rows}">
        <tr>
            <td><c:out value="${row.ID}" /></td>
            <td><c:out value="${row.NOME}" /></td>
        </tr>
    </c:forEach>
</table>
```

<sql:param>

Atribui o valor para um parâmetro "?" numa query

Atributos:

value	valor do parâmetro
--------------	--------------------

Exemplo:

```
<sql:update dataSource="${dataSource}">
    UPDATE CLIENTES SET NOME='CARLOS' WHERE ID=?
    <sql:param value="${param.cod}" />
</sql:update>
```

<sql:transaction>

Define um contexto transacional dentro do qual <> e <> serão utilizados.

Atributos:

dataSource	String de conexão para o banco de dados
-------------------	---

isolationLevel	nível de isolamento da transação
-----------------------	----------------------------------

Exemplo:

```
<sql:transaction>
    <sql:update>UPDATE CLIENTES set NOME=? where ID=?
    <sql:param value="${vnome}" />
    <sql:param value="${vID}" />
    </sql:update>
</sql:transaction>
```

Biblioteca Format

Propósitos:

1. Formatação de datas.
2. Formatação de números.
3. Formatação de horas.
4. Formatos customizáveis.

As principais tags contidas na biblioteca Format são:

```
<fmt:formatNumber>
<fmt:parseNumber>
<fmt:formatDate>
<fmt:timeZone>
<fmt:parseDate>
```

<fmt:formatNumber>

Atributos:

value	valor a ser formatado
type	(opcional) number, currency ou percentage
pattern	(opcional) padrão de formatação
currencyCode	(opcional) código ISO da moeda
currencySymbol	(opcional) símbolo da moeda
maxIntegerDigits/ minIntegerDigits	(opcional) dígitos máximos/mínimos da parte inteira da saída
maxFractionalDigits/ minFractionalDigits	(opcional) dígitos máximos/mínimos da parte fracionária da saída

var	(opcional) variável que conterá a saída
scope	(opcional) escopo da variável (request, page, session, application)

Exemplos:

```
<fmt:formatNumber value="9876543,21" type="currency"/>
```

Saída: R\$ 9.876.543,21

```
<fmt:formatNumber value="12,3" pattern=".000"/>
```

Saída: 12,300

```
<fmt:formatNumber value="123456,7891" pattern="#,#00.0#"/>
```

Saída: 123.456,79

```
<fmt:formatNumber value="123456789" type="currency" var="cur"/>
```

O resultado é armazenado em cur: **R\$ 123.456.789,00**

<fmt:parseNumber>

Atributos:

value	valor a ser formatado
type	number, currency ou percentage
parseLocale	padrão de formatação, de acordo com o local

Exemplo:

```
<c:set var="reais" value="R$ 5,00" />
<fmt:parseNumber value="${reais}" type="currency" parseLocale="pt_BR" />
```

Saída: 5

<fmt:formatDate>

Atributos:

value	valor a ser formatado
type	date, time, ou both
dateStyle	(opcional) padrão de formatação da data
timeStyle	(opcional) padrão de formatação da hora
pattern	(opcional) padrão de formatação customizado
timeZone	(opcional) TimeZone da data/hora a ser formatada
var	(opcional) Variável que conterá a saída
scope	(opcional) escopo da variável (request, page, session, application)

Exemplo:

```
<jsp:useBean id="now" class="java.util.Date" />
<fmt:formatDate value="${now}" timeStyle="long" dateStyle="long"/>
<fmt:formatDate value="${now}" pattern="dd.MM.yy"/>
```

Saída:

18 de Agosto de 2006

18.08.06

<fmt:parseDate>Atributos:

value	valor a ser convertido
type	date, time, ou both
dateStyle	(opcional) padrão de formatação da data
timeStyle	(opcional) padrão de formatação da hora
pattern	(opcional) padrão de formatação customizado
timeZone	(opcional) TimeZone da data/hora a ser formatada
parseLocale	(opcional) Local a ser usado como base de conversão
var	(opcional) variável que conterá a saída

scope	(opcional) escopo da variável (request, page, session, application)
--------------	---

Exemplo:

```
<fmt:parseDate value="13:15" pattern="HH:mm" />
```

Saída:

Thu Jan 01 13:15:00 GMT-03:00 1970

Biblioteca XML

Propósitos:

1. *Parsing* de documentos XML.
2. Manipulação de elementos XML.
3. Fluxo condicional.
4. Iterações.
5. Transformações.

As principais *tags* contidas na biblioteca XML são:

```
<x:parse>
<x:out>
<x:set>
<x:if>
```

<x:parse>

Faz o *parsing* de um documento XML

Atributos:

xml	documento a ser “parseado”
var e scope	nome e escopo da variável que receberá o documento parseado
systemId	URI identificando o documento

Exemplo:

```
<c:import url="http://localhost:8080/doc.xml" varReader="xmlReader" />
<x:parse xml="${xmlReader}" var="documento" />
```

<x:out>

Avalia uma expressão XPath e imprime o resultado na página.

Atributos:

select	expressão XPath
escapeXml	flag (opcional) indicando se os caracteres especiais devem ser substituídos

Exemplo:

```
<c:import url="http://localhost:8080/doc.xml" varReader="xmlReader" />
<x:parse xml="${xmlReader}" var="documento" />
<x:out select="${documento/titulo}" />
<x:out select="${documento/autor}" />
```

<x:set>

Avalia uma expressão XPath e atribui o resultado a uma variável de escopo.

Atributos:

select	expressão XPath
escapeXml	flag (opcional) indicando se caracteres especiais devem ser substituídos
var	nome da variável que receberá o valor da expressão
scope	escopo da variável

Exemplo:

```
<c:import url="http://localhost:8080/doc.xml" varReader="xmlReader" />
<x:parse xml="${xmlReader}" var="documento" />
<x:set select="${documento/titulo}" var="tit" scope="page"/>
<x:set select="${documento/autor}" var="aut" scope="page"/>
```

<x:if>

Executa o corpo da tag caso uma expressão XPath seja avaliada em verdadeiro.

Atributos:

select	expressão XPath
---------------	-----------------

var	nome da variável que receberá o valor da expressão de teste
scope	escopo da variável

Exemplo:

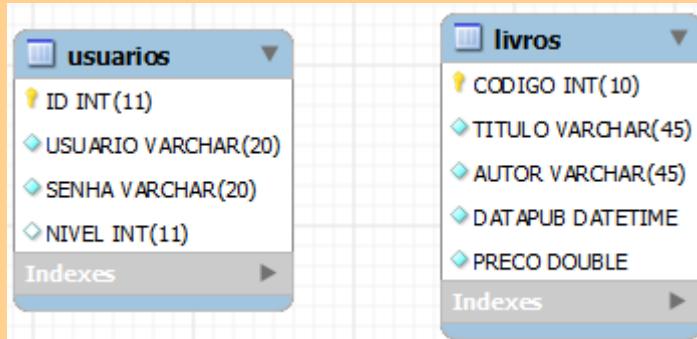
```
<c:import url="http://localhost:8080/doc.xml" varReader="xmlReader" />
<x:parse xml="${xmlReader}" var="documento" />
<x:if select="${documento/[titulo=='JSTL']}" />
    Introdução a bibliotecas JSTL
<x:if />
```

Exercícios do Módulo 3

Neste módulo o aluno desenvolverá uma aplicação para cadastro e consulta de livros. A aplicação será desenvolvida em etapas, e cada etapa será representada por um exercício. Comecemos então pelo Exercício 1.

Exercício 1: Definição do banco de dados

Desenvolver um banco de dados chamado **servletjsp** contendo as tabelas:



Exercício 2: Camada de acesso a dados

Vamos criar as classes para o acesso a dados. As classes a serem criadas são:

1. Livros
2. Usuários
3. Dao
4. DaoLivros
5. DaoUsuarios
- Classe Livros:

```

package com.ead.model;

public class Livros {

    private String codigo, titulo, autor;
    private java.util.Date datapub;
    private double preco;

    //getters e setters
}

```

- Classe Usuários:

```

package com.ead.model;

public class Usuarios {
    private String usuario, senha;
    private int nivel;

    //getters e setters
}

```

- Classe Dao:

```

package com.ead.dao;

import java.sql.*;
public class Dao {
    private String url="jdbc:mysql://localhost:3306/servetjsp";
    protected Connection cn;
    protected PreparedStatement st;
    protected ResultSet rs;

    protected boolean abreConexao() throws Exception{
        try{
            Class.forName("com.mysql.jdbc.Driver");
            cn = DriverManager.getConnection(url,"root","root");
            return true;
        }
        catch(Exception ex){
            throw ex;
        }
    }

    protected void fechaConexao() throws Exception{
        cn.close();
    }
}

```

- Classe DaoUsuarios:

```
package com.ead.dao;

import com.ead.modelUsuarios;

public class DaoUsuarios extends Dao{

    public boolean validaUsuario(Usuarios usuario) throws Exception{
        boolean b = false;
        try{
            abreConexao();
            st = cn.prepareStatement("SELECT * FROM USUARIOS WHERE USUARIO=? AND SENHA=?");
            st.setString(1, usuario.getUsuario());
            st.setString(2, usuario.getSenha());
            rs = st.executeQuery();
            if(rs.next()){
                usuario.setNivel(rs.getInt("NIVEL"));
                b = true;
            }
        }
        catch(Exception ex){
            throw ex;
        }
        finally{
            fechaConexao();
        }
        return b;
    }
}
```

- Classe DaoLivros

```
package com.ead.dao;

import java.util.ArrayList;
import java.util.List;
import com.ead.model.Livros;

public class DaoLivros extends Dao{

    public String cadastraLivro(Livros livro) throws Exception{
        String msg="";
        try{
            abreConexao();
            st = cn.prepareStatement("INSERT INTO LIVROS (CODIGO,TITULO,AUTOR,DATAPUB,PRECO) VALUES (?,?,?,?,?)");
            st.setString(1, livro.getCodigo());
            st.setString(2, livro.getTitulo());
            st.setString(3, livro.getAutor());
            st.setDate(4, new java.sql.Date(livro.getDataPub().getTi-
me()));
            st.setDouble(5, livro.getPreco());
        }
    }
}
```

```

        int cont = st.executeUpdate();
        if(cont == 0){
            msg = "Nenhum livro foi inserido!";
        }
        else{
            msg = "Livro inserido com sucesso!";
        }
    }
    catch(Exception ex){
        throw ex;
    }
    finally{
        fechaConexao();
    }

    return msg;
}

public List<Livros> listaLivros() throws Exception{
    List<Livros> lista = new ArrayList<Livros>();
    try{
        abreConexao();
        st = cn.prepareStatement("SELECT * FROM LIVROS");
        rs = st.executeQuery();

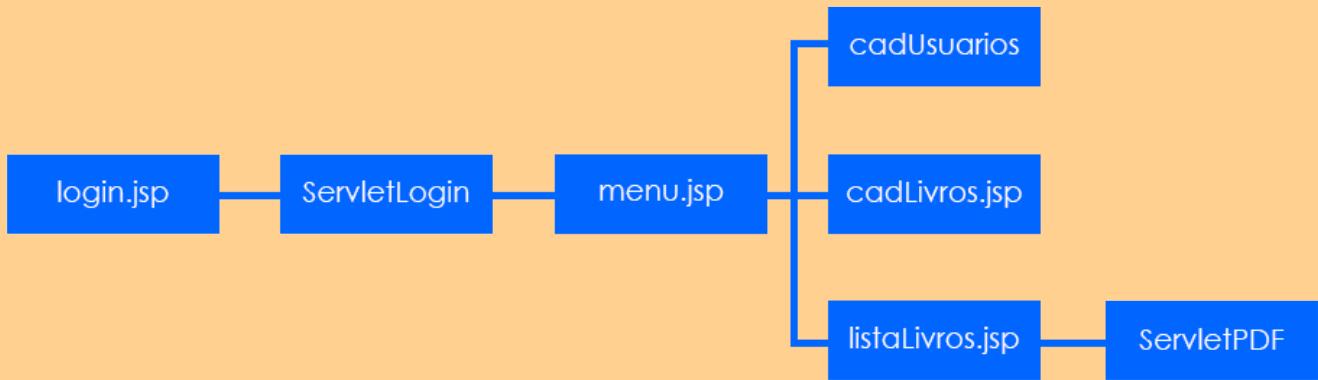
        while(rs.next()){
            Livros livro = new Livros();
            livro.setCodigo(rs.getString("CODIGO"));
            livro.setTitulo(rs.getString("TITULO"));
            livro.setAutor(rs.getString("AUTOR"));
            livro.setDatapub(rs.getDate("DATAPUB"));
            livro.setPreco(rs.getDouble("PRECO"));

            lista.add(livro);
        }
    }
    catch(Exception ex){
        throw ex;
    }
    finally{
        fechaConexao();
    }
    return lista;
}
}

```

Exercício 3: Definição da aplicação

A aplicação consistirá em páginas JSP, um *Servlet* para validação do usuário e um *Srvlet* para exibir o sumário de um livro. O diagrama de navegação para a aplicação é:



Siga os passos descritos a seguir para elaborar a aplicação:

login.jsp: conteúdo HTML para entrada do usuário e senha tradicionais.

Incluir um filtro para verificar se os dados estão preenchidos, antes de enviá-los para o Servlet.

ServletLogin: recebe os dados do formulário em **login.jsp**, autentica o usuário (valida e armazena em sessão) e direciona para **menu.jsp**. Nota: armazenar na sessão o objeto **Usuarios**, pois seus dados serão recuperados nas páginas seguintes.

O método **doGet()** deste formulário deve transferir a requisição para **login.jsp** no sentido de incluir seu conteúdo.

Se houver erro na validação, transferir a requisição para uma página chamada **erro.jsp** (não indicada no esquema). Essa página deverá apresentar a mensagem de erro e ter um link para **login**.

Quando os dados do usuário forem armazenados em sessão, notificar a aplicação através de um Listener (HttpSessionListener** e **HttpSessionAttributeListener**).**

menu.jsp: Nessa página deverá ter apenas três links para as páginas seguintes, conforme especificado no modelo.

cadUsuarios.jsp: Uma página que permite realizar o cadastro de novos usuários. Esta página deve ser elaborada com conteúdo HTML e scriptlet.

cadLivros.jsp: Formulário com os dados dos livros, com elementos HTML e componentes JavaBeans.

listaLivros.jsp: Apresenta os dados dos livros em forma tabular. O código deve estar envolvido por um link que, quando selecionado pelo usuário, apresenta um documento PDF contendo um resumo do livro. Esta página deverá ser escrita em JSTL.

ServletPDF: recebe um parâmetro representando o código do livro, e busca por um arquivo PDF com o mesmo nome do código. Se não for encontrado, apresentar um documento padrão.

Código para ler o PDF

Este código supõe que os arquivos PDF estejam na pasta **D:\arquivos**.

```

protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    ServletOutputStream stream = null;
    BufferedInputStream buffer = null;

    String arquivo = request.getParameter("cod");

    try {
        stream = response.getOutputStream();
        File pdf = new File("D:/arquivos/" + arquivo + ".pdf");
        if(!pdf.exists()){
            pdf = new File("D:/arquivos/geral.pdf");
        }

        response.setContentType("application/pdf");
        FileInputStream input = new FileInputStream(pdf);
        buffer = new BufferedInputStream(input);
        int bytes = 0;

        while((bytes = buffer.read()) != -1){
            stream.write(bytes);
        }
    } catch (Exception e) {
        throw new ServletException(e.getMessage());
    } finally {
        if(stream != null){ stream.close(); }
        if(buffer != null){ buffer.close(); }
    }
}

```

Incluir todas as páginas a partir de **menu.jsp** em uma pasta chamada **admin**. Este procedimento facilitará a manipulação do filtro

Módulo 4 - Conceitos do Framework JSF 2.0

Aula 11 – JavaServer Faces

Definição de JSP e Framework

A tecnologia JavaServer Faces (JSF) é constituída de um *framework* para desenvolvimento de aplicações *Web*, executada em um servidor Java e que retorna ao cliente uma interface gráfica.



Os principais componentes do JSF são:

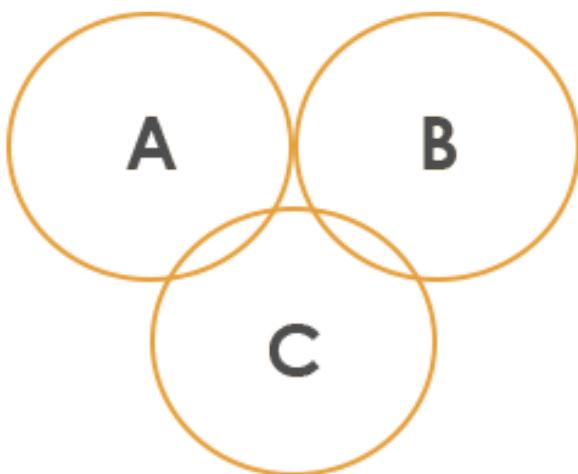
- Uma API para representação de componentes de interface gráfica, manipulação de eventos, validação de dados, conversão de dados, internacionalização, definição de regras de navegação.
- Uma biblioteca de tags personalizadas, semelhante à JSTL, para representar os componentes JSF dentro da página.
- Um modelo de componentes que permite a desenvolvedores independentes fornecerem componentes adicionais.

A JSF contém todos os códigos necessários para a manipulação de eventos e organização de componentes. Sendo assim, existem ambientes de desenvolvimento (IDE) para a construção desses componentes. O NetBeans (utilizado neste curso) já contém a API JSF.

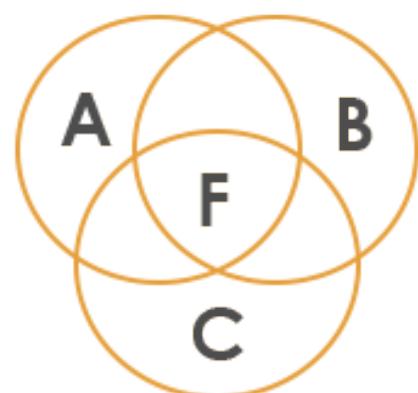
Um *framework* é uma aplicação “quase” completa, com partes faltantes. Essas partes são justamente as implementações peculiares de cada problema. Apresenta as características:

- Captura a funcionalidade comum a várias aplicações;
- Providencia uma solução para um conjunto de problemas;
- Utiliza um conjunto de classes e interfaces que mostram como decompor o problema;
- Define como os objetos dessas classes colaboram para cumprir suas responsabilidades.

A Figura 11.1 mostra dois cenários que ajudam a entender o conceito de *framework*.

Figura 11.1: Conceito de Framework

Impossível criar um framework
A,B e C = Implementações Específicas



F = Domínio do Framework

A API JSF providencia justamente a parte comum às aplicações, ou seja, o domínio do *framework*. O desenvolvedor é o responsável pelas partes complementares ou aplicações específicas.

O foco da JSF é oferecer um *framework* que suporte componentes gráficos, eventos e gerenciamento de estado.

A arquitetura MVC (Model–View–Controller)

O Model-view-controller (MVC) é um padrão de arquitetura de software. Em aplicações complexas, que enviam uma série de dados para o usuário, o desenvolvedor frequentemente necessita separar os dados (**Model**) da interface (**View**). Desta forma, alterações feitas na interface não afetarão a manipulação dos dados e estes poderão ser reorganizados sem alterar a interface do usuário. O MVC resolve esse problema através da separação das tarefas de acesso aos dados e lógica do negócio da apresentação e *iteração* com o usuário, introduzindo um componente entre os dois: o **Controller**.

MVC é usado em padrões de projeto de software, mas o MVC abrange mais da arquitetura de uma aplicação do que é típico para um padrão de projeto. Cada um dos elementos da arquitetura MVC é descrito a seguir:

- **Model:** É a representação do elemento que contém a informação ou as informações de interesse para o sistema. Por exemplo, um sistema de cadastro de alunos e cursos possui como modelos a entidade Aluno, a entidade Curso e a relação existente entre eles.
- **View:** Refere-se à parte visual da aplicação como definida pela camada Model. É a parte do sistema que interage com o usuário.
- **Controller:** Responsável pelo processamento das informações, como a manipulação de eventos e execução de propriedades.

No MVC destacam-se dois modelos essenciais: Arquitetura Page-Centric e Arquitetura Servelet-Centric, mostradas nas Figuras 11.2a e 11.2b.

Figura 11.2a: Arquitetura MVC Model 1

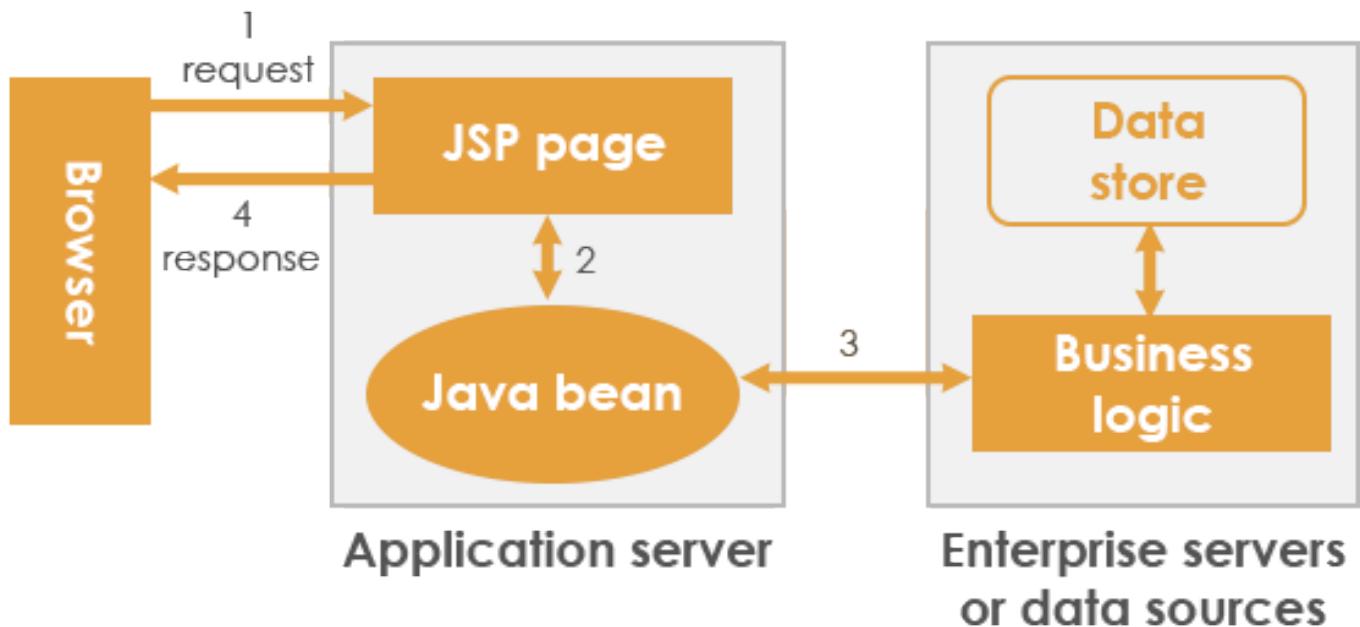
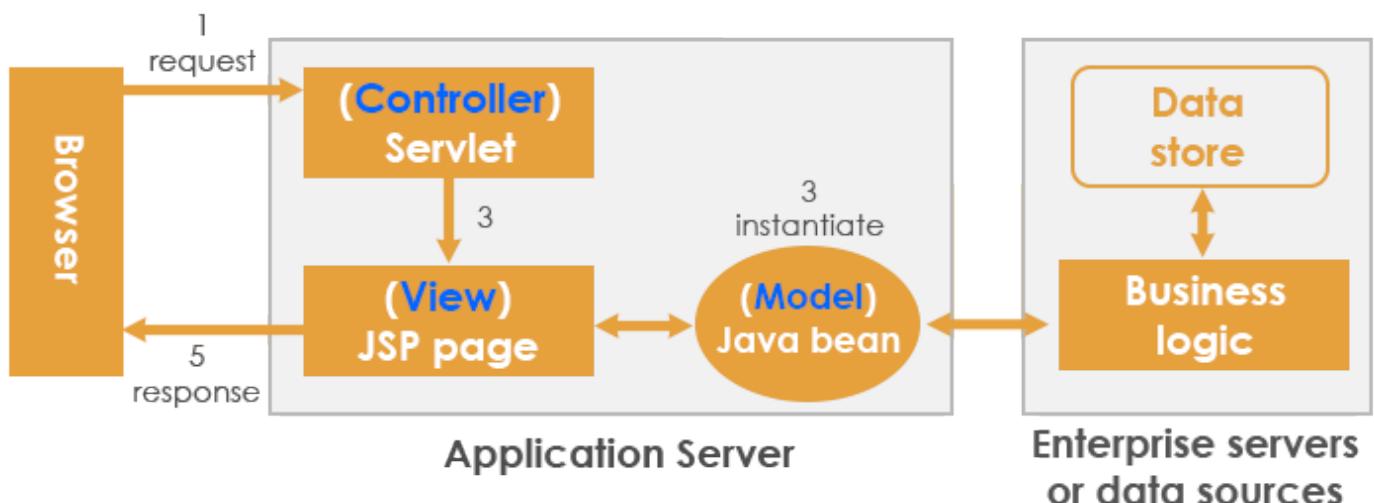
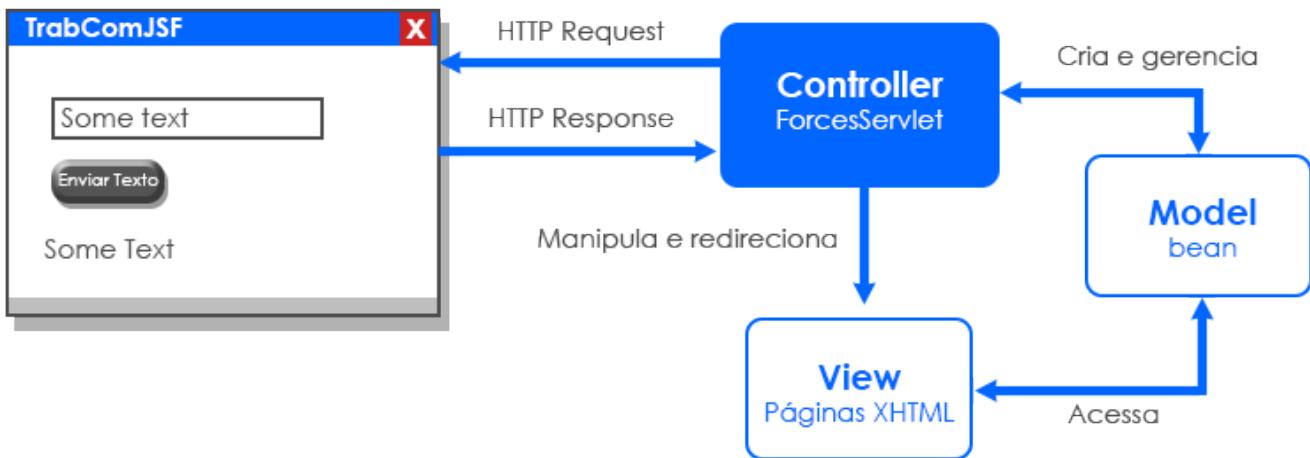


Figura 11.2b: Arquitetura MVC Model 2



O MVC de JSF

O funcionamento do JSF consiste no registro do `servlet FacesServlet`. Com esse registro, as requisições são controladas pela classe `javax.faces.webapp.FacesServlet` (o Controller do MVC), uma implementação de `javax.servlet.HttpServlet`, roteando o tráfego e administrando o ciclo de vida dos beans e componentes de Interface do Usuário ou User Interface (UI). A Figura 11.3 ilustra este esquema.

Figura 11.3: Mecanismo de requisições JSF 2.0

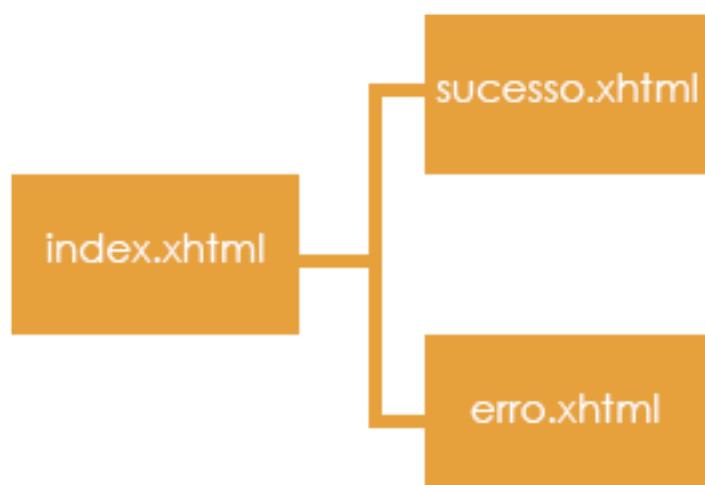
Os componentes UI são organizados em uma estrutura de árvore em que cada componente pode ser associado com os métodos e atributos de um bean. Cada componente também pode ser associado com uma função de validação ou classe.

Exemplo de uma aplicação JSF

Para melhor compreender o uso do JSF, caro aluno, vamos mostrar uma aplicação de exemplo com o objetivo de ilustrar o uso das tags JSF, das classes que representam os beans e da regra de navegação.

O JSF 2.x utiliza Facelets desenvolvidos em XHTML, diferentemente das versões anteriores do JSF que adotavam JSP como suas páginas que renderizam a visualização.

Neste exemplo vamos considerar a aplicação representada no diagrama da Figura 11.4.

Figura 11.4: Diagrama base da aplicação JSF

Esse exemplo representa uma simulação de login. A página `index.xhtml` é a página em que o usuário fornece seus dados de usuário e senha. Se estiver correto, ele é direcionado para `sucesso.xhtml` e, caso contrário, direcionado para `erro.xhtml`.

O arquivo **index.xhtml** possui o código mostrado no Quadro 18.

Quadro 18 – arquivo index.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>Login</title>
</h:head>
<h:body>
    <f:view>
        <h:form>
            <h3>Por favor, informar seu usuário e sua
                senha.</h3>
            <table>
                <tr>
                    <td>Nome:</td>
                    <td>
                        <h:inputText
                            value="#{usuarios.nome}">
                    </td>
                </tr>
                <tr>
                    <td>Senha:</td>
                    <td>
                        <h:inputSecret
                            value="#{usuarios.senha}">
                    </td>
                </tr>
            </table>
            <p>
                <h:commandButton value="Login"
action="#{usuarios.validarUsuario}">
            </p>
        </h:form>
    </f:view>
</h:body>
</html>
```

A maior parte da página é semelhante ao formato HTML, com as diferenças:

- Todas as tags JSF estão contidas dentro de uma tag `<f:view>`.
- Ao invés de usar uma tag HTML `form`, os componentes foram incluídos dentro de uma tag `<h:form>`.
- Ao invés de usar tags `input` para os componentes da interface, foram utilizados os elementos `<h:inputText>`, `<h:inputSecret>` e `<h:commandButton>`. Esses componentes estão definidos no *framework*.

Os valores dos campos de entrada são ligados às propriedades do bean de nome **usuarios**.

O bean usuários é o que em JSF chamamos de **Bean Gerenciado (Managed Bean)**, e é definido na classe mostrada no Quadro 19.

Quadro 19 – classe UsuariosBean

```
package com.jsf2;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "usuarios")
@SessionScoped
public class UsuariosBean {

    private String nome, senha;

    public void setNome(String nome) {
        this.nome = nome;
    }

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

    public String getNome() {
        return nome;
    }

    public String getSenha() {
        return senha;
    }

    public String validarUsuario(){
        if(nome.equals("user") && senha.equals("123")){
            return "/sucesso";
        }
        else {
            return "/erro";
        }
    }
}
```

Observe as anotações no início da classe: `@ManagedBean(name = "usuarios")` e `@SessionScoped`. Elas definem a classe como sendo um bean gerenciado, acessível através do nome “usuários” e armazenado na sessão, quando acessado.

A anotação `@ManagedBean` possui, além do atributo `name`, o atributo `eager`, de tipo boolean, que somente é considerado quando o escopo do bean é gerenciável para `application`. Caso o valor desse atributo seja `true` e o escopo `application`, o bean gerenciável será criado e colocado nesse escopo quando a aplicação iniciar.

O componente `<h:inputText value="#{usuarios.nome}">` renderiza uma caixa de textos, aguardando um valor de entrada. O valor informado nesta caixa de textos é atribuído à propriedade **nome** através do bean **usuarios**. Por ser um componente de entrada, implicitamente o valor da caixa de textos é passado como parâmetro para o *setter* correspondente a esta propriedade. Elementos de saída usam o *getter* correspondente.

Quando o formulário é preenchido, ele é submetido através do botão:

```
<h:commandButton value="Login" action="#{usuarios.validarUsuario}">
```

É importante observar que esse botão não acessa uma propriedade do bean, mas um método. Este método representa um *action* a ser executado, ou seja, um evento de ação padrão. O conteúdo deste método executa a tarefa de validação do usuário e retorna a navegação entre as páginas, como mostra o Quadro 20.

```
<h:commandButton value="Login" action="#{usuarios.validarUsuario}">
```

Quadro 20 – método validarUsuario

```
public String validarUsuario(){
    if(nome.equals("user") && senha.equals("123")){
        return "/sucesso";
    }
    else {
        return "/erro";
    }
}
```

O retorno “/sucesso” indica que o usuário será transferido para `sucesso.xhtml`, e o retorno “/erro”, para a página `erro.xhtml`. A transferência ocorre através de um objeto `RequestDispatcher`.

As páginas `sucesso.xhtml` e `erro.xhtml` são mostradas nos Quadros 21 e 22.

Quadro 21 – Página sucesso.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
    <h:head>
        <title>Sucesso</title>
    </h:head>
    <h:body>
        <f:view>
            <h:form>
                <h3>Seja benvindo, <h:outputText value="#{usuarios.nome}" /></h3>
            </h:form>
        </f:view>
    </h:body>
</html>
```

Quadro 22 – Página erro.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
    <h:head>
        <title>Erro!!</title>
    </h:head>
    <h:body>
        <f:view>
            <h:form>
                <h3>Usuário ou senha inválidos!!</h3>
            </h:form>
        </f:view>
    </h:body>
</html>
```

Aula 12 - Visão Geral das Tags JSF

Tags JSF Fundamentais

As tags fundamentais representam objetos que podemos adicionar a componentes. A biblioteca fundamental também contém tags para definir visões e subvisões, carregar recursos “bundle” e adicionar texto arbitrário a uma página. Essas tags são definidas pelo namespace “<http://java.sun.com/jsf/core>”, indicado no início da página xhtml. A Tabela 12.1 mostra estas tags.

Tabela 12.1: Tags JSF fundamentais

Tag	Descrição
-----	-----------

view	Cria a visão de nível mais alto
subview	Cria uma subvisão de uma visão
facet	Adiciona uma faceta a um componente
attribute	Adiciona um atributo (key/value) a um componente
param	Adiciona um parâmetro a um componente
actionListener	Adiciona um listener de ação a um componente
ValueChangeListener	Adiciona um listener de mudança de valor a um componente
converter	Adiciona um conversor arbitrário a um componente
convertDateTime	Adiciona um conversor de data e hora a um componente
convertNumber	Adiciona um conversor numérico a um componente
validator	Adiciona um validador a um componente
validateDoubleRange	Valida uma faixa dupla para um valor de um componente
validadeLength	Valida a extensão de um valor de um componente
validadeLongRange	Valida uma faixa longa para um valor de um componente
loadBundle	Carrega um recurso e armazena propriedade como um Mapa
selectItems	Especifica itens para um componente select one ou select many
selectItem	Especifica um item para um componente select one ou select many
verbatim	Adiciona código markup a uma página JSF

Principais Tags HTML JSF

Essa categoria inclui formulários, mensagens e componentes que geram *layout* para outros componentes. Está definida no namespace “<http://java.sun.com/jsf/html>”. A Tabela 12.2 mostra estas tags.

Tabela 12.2: Tags HTML JSF principais

Tag	Descrição
form	Formulário HTML
inputText	Controle de entrada de texto em uma única linha
inputTextarea	Controle de entrada de texto para várias linhas
inputSecret	Controle de entrada de senha

inputHidden	Campo oculto
outputLabel	Rotulação de outro componente para melhor acessibilidade
outputLink	Âncora HTML
outputFormat	Formatação de mensagens compostas
outputText	Saída de texto em uma única linha
commandButton	Botões: <i>submit</i> , <i>reset</i> ou <i>pushbutton</i>
commandLink	<i>Link</i> que age como um <i>pushbutton</i>
message	Exibição de mensagem mais recente de um componente
messages	Exibição de todas as mensagens
graphicImage	Exibição de uma imagem
selectOneListbox	Caixa de listagem para seleção de um único item
selectOneMenu	Menu para seleção de um único item
selectOneRadio	Conjunto de botões de rádio
selectBooleanCheckbox	Caixa de verificação
selectManyCheckbox	Conjunto de caixas de verificação
selectManyListbox	Caixa de listagem para seleção de vários itens
selectManyMenu	Menu para seleção de vários itens
panelGrid	Tabela HTML
panelGroup	Dois ou mais componentes que são dispostos côo um só
dataTable	Um controle de tabela com muitas funções
column	Coluna de uma <i>Table</i>

Anotações em *beans* gerenciáveis

Ao registrarmos no JSF uma classe criada em nosso projeto *web*, ela se torna um *Managed Bean* (ou bean gerenciável). Quando isso acontece, esta classe fica gerenciável pelo *framework*, ou seja, a partir de qualquer página JSF é possível acessar seus métodos públicos. Daí o nome *Managed Bean*.

No JSF ele corresponde ao Modelo da arquitetura *Model-View-Controller* (MVC), para os componentes da UI.

Na versão 1.2 do JSF, para que uma classe se tornasse um bean gerenciável, era necessário registrá-la através do *faces-config.xml*. Isso trazia um inconveniente, pois na medida que crescia o número de

beans gerenciáveis, o arquivo de configuração também crescia na mesma proporção. E com um número grande desses *beans*, se tornava complicado gerenciar todas as alterações em três lugares distintos, porém relacionados: o *faces-config.xml*, a página JSF e o próprio bean gerenciável.

No exemplo apresentado neste módulo, os *beans* gerenciados e a regra de navegação poderiam ter sido configurados no arquivo *faces-config.xml* da forma mostrada no Quadro 23.

Quadro 23 - Arquivo faces-config.xml

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd" version="1.2">
<navigation-rule>
    <from-view-id>/index.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>sucesso</from-outcome>
        <to-view-id>/sucesso.xhtml</to-view-id>
    </navigation-case>
<navigation-case>
    <from-outcome>erro</from-outcome>
    <to-view-id>/erro.xhtml</to-view-id>
</navigation-case>
</navigation-rule>

<managed-bean>
    <managed-bean-name>usuarios</managed-bean-name>
    <managed-bean-class>com.jsf2.UsuariosBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>
```

No JSF 2.0 não é necessário registrar um bean no arquivo de configuração. Podem-se utilizar anotações para isso. Dessa forma, o *bean* e o seu registro ficam no mesmo lugar, na mesma classe Java. Isso facilita bastante o controle dos *beans* gerenciáveis, como também deixa mais limpo o *faces-config.xml*. Porém, em alguns casos, você será obrigado a registrar os *beans* no arquivo de configuração, por falta de alternativa via anotações.

Escopos de *beans* gerenciáveis

No JSF 2.0 existem as seguintes anotações para a definição de escopo de *beans* gerenciáveis:

- **@NoneScoped:** os *beans* gerenciáveis de escopo none não são instanciados nem salvos em nenhum escopo. Eles são instanciados, sob demanda, por outros *beans* gerenciáveis. Um bean gerenciável de escopo none somente pode instanciar outros *beans* gerenciáveis de escopo none.
- **@RequestScoped:** os *beans* gerenciáveis de escopo request são instanciados e permanecem disponíveis durante uma mesma requisição HTTP. Eles podem instanciar outros *beans* gerenciáveis de escopo: none, request, view, session e application.

- **@ViewScoped:** os *beans* gerenciáveis de escopo view permanecem disponíveis enquanto o usuário permanecer em uma mesma página de uma aplicação. Eles podem instanciar *beans* de escopo: none, view, session e application.
- **@SessionScoped:** os *beans* gerenciáveis de escopo session são salvos na sessão HTTP de um usuário. Podem instanciar *beans* de escopo: none, session e application.
- **@ApplicationScoped:** os *beans* gerenciáveis de escopo application permanecem disponíveis enquanto a aplicação estiver no ar e podem ser acessados por todos os usuários da aplicação. Podem instanciar outros *beans* de escopo: none e application.
- **@CustomScoped:** os *beans* gerenciáveis de escopo custom são *beans* que possuem um tempo de vida personalizado. Por exemplo, você pode definir um escopo de conversação, como existe no JBoss Application Server, no qual um bean permanece disponível para um conjunto de páginas.

Navegação condicional

Diferentemente da navegação implícita, a navegação condicional é definida juntamente com as regras de navegação da aplicação, no arquivo `faces-config.xml`.

Conforme as versões anteriores do JSF, as regras de navegação do JSF 2.0 no `faces-config.xml` são definidas através das tags `<navigation-rule>` e `<navigation-case>`, sendo que dentro de um elemento `<navigation-rule>`, os itens `<navigation-case>` são processados na ordem em que aparecem.

A primeira condição satisfeita fará com que a próxima página a ser mostrada no fluxo de navegação seja aquela definida na tag `<to-view-id>`. Para que um `<navigation-case>` seja satisfeito, basta que o resultado de um método do bean gerenciável, ou da página de origem, seja igual ao valor da tag `<from-outcome>`. A navegação condicional possibilita definir uma verificação a mais dentro de um `<navigation-case>`, com o uso da nova tag `<if>`. O esquema do Quadro 24 mostra um exemplo.

Quadro 24 – Exemplo de regra de navegação

```
<navigation-rule>
    <from-view-id>/confirmacao.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>index</from-outcome>
        <if>#{clienteBean.novoCadastro}</if>
        <to-view-id>/cadastro.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>index</from-outcome>
        <to-view-id>/index.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Elementos do JSF

Os elementos do JSF são:

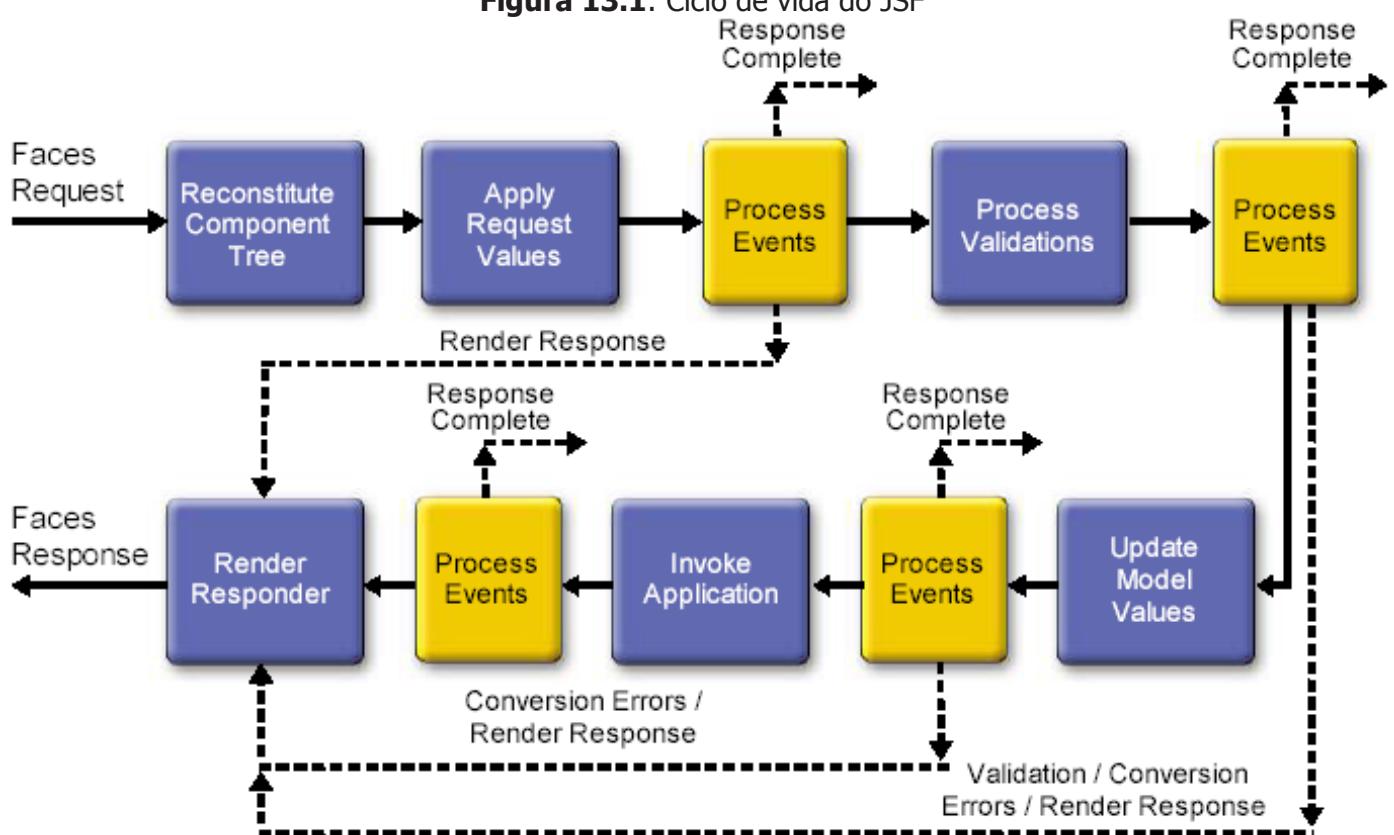
- **Estado dos componentes:** Mantém automaticamente o estado dos componentes Web entre uma requisição e outra.
- **Renderização de componentes:** Endereça várias plataformas clientes Web.
- **Processamento de formulários:** Provê mecanismos para o processamento de telas com um ou vários formulários (sub-views).
- **Validação:** Provê mecanismos para validação de campos e mensagens de erro.
- **Internacionalização:** Suporte nativo para internacionalização.
- **Componentes Visuais:** Provê componentes básicos padrões que podem ser estendidos. Permite adicionar bibliotecas de terceiros.
- **Árvore de componentes:** Realiza o alinhamento de um componente dentro de outro para formar uma interface gráfica. Cada componente possui acesso ao seu pai e filho.
- **Facets:** Permite adicionar componentes independentes da ordem da árvore. Define uma relação especial com o componente externo. Muito utilizado em tabelas para definição de ‘cabecalho’ e ‘rodapé’.
- **Eventos:** Permite acionamento de “Listeners” sob determinados eventos, exemplo: alteração do valor de um campo.

Aula 13 - Ciclo de vida do JSF

Fases do ciclo de vida do JSF

Por trás dos bastidores, **FacesServlet** aceita requisições de entrada e controla o ciclo de vida da aplicação através da classe **javax.faces.lifecycle.Lifecycle**. Usando uma “fábrica”, ela cria um objeto do tipo **javax.faces.context.FacesContext**, que contém todos os processos.

O objeto Lifecycle usa o objeto FacesContext em seis fases que compõem o ciclo de vida de uma aplicação JSF. Cada fase tem uma ação importante em sua aplicação e tem papel importante no desenvolvimento da aplicação, como mostra a Figura 13.1.

Figura 13.1: Ciclo de vida do JSF

Fonte: <http://www.java-forums.org/attachments/java-tutorial/189d1319002863t-javaserver-faces-basics-lifecycle.jpg>. Acesso em: 03 set. 2014.

As seis fases são executadas na seguinte ordem:

[1] Restaurar Apresentação: Esta fase inicia o processamento da requisição do ciclo de vida por meio da construção da árvore de componentes do JSF. Cada árvore de componentes possui um identificador único durante todo o aplicativo. O JSF constrói a apresentação da página e salva na instância **FacesContext** para processamento das fases seguintes.

[2] Aplicar Valores Requisitados: Quaisquer novos valores inseridos são extraídos e armazenados por seus apropriados componentes. Se o valor do componente não for uma String, então ele é convertido para o seu determinado tipo. Se a conversão falhar, ocorrem diversas situações:

- Uma mensagem de erro é gerada e associada com o componente;
- Uma mensagem de erro é armazenada no **FacesContext** e depois mostrada posteriormente na fase de Renderizar a Resposta;
- O ciclo de vida pula a fase de Renderizar a Resposta quando esta se completou.

[3] Processar Validações: Depois do valor de cada componente ser atualizado, na fase de processo de validações, os componentes serão validados naqueles valores, se necessário. Um componente que necessita de validação deve fornecer a implementação da lógica de validação. Por exemplo, em um carrinho de compras, podemos determinar a quantidade mínima e máxima a ser digitada. O valor requisitado é um inteiro (verificado na fase anterior) e, como passou pela fase 2, nessa fase pode ser barrado por estar além do determinado (com uma quantidade mínima ou máxima diferente da estipulada).

[4] Atualizar Valores do Modelo: Alcança-se essa fase após todos os componentes serem validados.

Nesta fase são atualizados os dados do modelo do aplicativo. Na página em que foi criada para enviar um texto, a informação digitada foi armazenada no Managed Bean durante esta fase. Por ter passado pelo processo de validação, temos garantias que o valor armazenado será garantido nessa fase. Entretanto, os dados podem violar a lógica de negócios, cuja validação ocorre na fase seguinte.

[5] Invocar Aplicação: Durante esta fase, a implementação JSF manipula quaisquer eventos do aplicativo, tal como enviar um formulário ou ir a outra página através de um *link*. Esses eventos são ações que retornam geralmente uma string que está associada a uma navegação a qual se encarrega de chamar a página determinada.

[6] Renderizar Resposta: Esta é a fase final, na qual é renderizada a página. Se este é um pedido inicial para esta página, os componentes são acrescentados à apresentação neste momento. Se este é um *postback*, os componentes já foram acrescidos à apresentação. Se há mensagens de conversão ou erros de validação e a página contém um ou mais componentes <message /> ou um componente <messages />, estes serão exibidos. Reciprocamente, se a página não contém um componente de mensagem, nenhuma informação aparecerá.

Aula 14 – Eventos JSF

JSF suporta três tipos de eventos: ValueChange, Action e Phase, descritos no que se segue.

ValueChange Event

Gerado por componentes UIInput. Um componente tradicional que utiliza esse evento é o controle <h:selectOneMenu>.

O componente <h:selectOneMenu> é muitas vezes usado para atualizar uma página, atualizar informações com base no item selecionado ou mesmo para acessar outra página.

A codificação desse componente no JSF 2.0 envolve a utilização de um evento chamado valueChangeListener. Veja um exemplo:

```
<f:view>
    <h:head>
        <title>ValueChangeListener</title>
    </h:head>

    <h:form>
        <table border="1">
            <tr>
                <td>
                    <h:selectOneMenu value="#{aluno.curso}"
                        onchange="submit()"
                        valueChangeListener="#{aluno.
valueChangeCurso}">
                        <f:selectItems value="#{aluno.cursos}" />
                    </h:selectOneMenu>
                </td>
                <td>
                    <h:outputText value="#{aluno.curso}" />
                </td>
            </tr>
        </table>
    </h:form>
</f:view>
```

No **ManagedBean** deve haver o seguinte método:

```
public void valueChangeCurso(ValueChangeEvent evt){
    aluno.setCurso(evt.getNewValue().toString());
}
```

A atualização da página e, consequentemente, das informações, está condicionada à submissão da página.

Se for desejável uma execução **Ajax**, a estrutura deve ser como mostrada a seguir:

```
<f:view>
    <h:head>
        <title>ValueChangeListener</title>
    </h:head>

    <h:form>
        <a4j:region>
            <table border="1">
                <tr>
                    <td>
                        <h:selectOneMenu value="#{aluno.curso}" >
                            <f:selectItems value="#{aluno.cursos}" />
                        <f:ajax listener="#{aluno.valueChangeCurso2}" render="resultado"/>
                    </h:selectOneMenu>
                    </td>
                    <td>
                        <h:outputText id="resultado" value="#{aluno.curso}" />
                    </td>
                </tr>
            </table>
        </a4j:region>
    </h:form>
</f:view>
```

E no **ManagedBean**, o seguinte método:

```
public void valueChangeCurso2(AjaxBehaviorEvent evt){
    String s = evt.getComponent().getAttributes().get("value").toString();
    aluno.setCurso(s);
}
```

A execução de uma tarefa assíncrona (baseada em Ajax), consiste na **API Ajax For JSF** (a4j) disponível com algumas bibliotecas que manipulam Ajax. As mais populares são **RichFaces** e **PrimeFaces**.

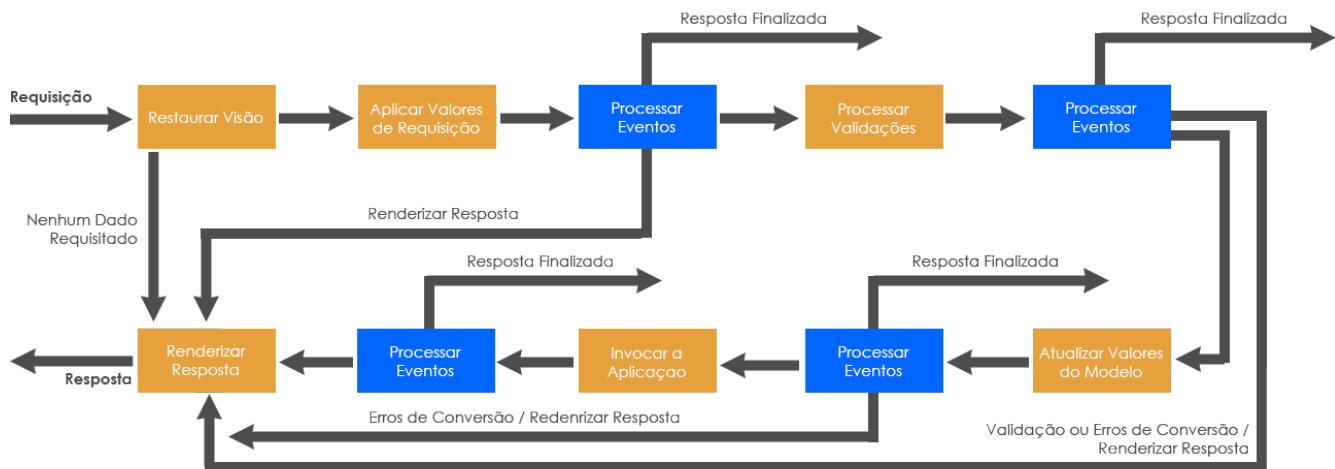
O atributo **render**, do elemento **<f:ajax>** permite transferir o resultado para o controle cujo **id** é seu valor. No nosso exemplo, “resultado” é o id da caixa de textos na qual o resultado será exibido assincronamente.

Action Event

Gerado por componentes **UICommand**, esse evento é usado em controles que produzem ação, como é o caso do **commandButton** e do **commandLink**.

Phase Event

Disparado pelas mudanças de fase do ciclo de vida do JSF. A Figura 14.3 ilustra os eventos nas diferentes fases.

Figura 14.3: Eventos no JSF

Para confirmar como todas as fases do ciclo de vida são usadas e quais são, basta criar um simples **PhaseListener** e mandá-lo gerar a saída antes e depois de cada fase.

Criando uma classe Phase Listener

Criar uma classe que implemente a **PhaseListener** exige dois passos:

- Criar uma classe que implemente PhaseListener;
- Registrar essa classe no arquivo de configurações do JSF.

A classe **PhaseListener** deve implementar a classe **javax.faces.event.PhaseListener**. O código do Quadro 25 mostra como criar essa classe.

Quadro 25 – Classe TestPhaseListener, usada para eventos de mudança de fase

```

package br.jsf2.listener;

public class TestPhaseListener implements javax.faces.event.PhaseListener {
    public void afterPhase(PhaseEvent event) {
        event.getFacesContext().getExternalContext().
            log("AFTER: "+event.getPhaseId());
    }

    public void beforePhase(PhaseEvent event) {
        event.getFacesContext().getExternalContext().
            log("BEFORE: "+event.getPhaseId());
    }

    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }
}
  
```

Na classe do Quadro 25, três métodos são implementados: afterPhase(), beforePhase() e getPhaseId():

- Os dois primeiros, afterPhase() e beforePhase(), definem o que ocorre ANTES e DEPOIS de cada fase. Nesses métodos foram incluídos a capacidade de imprimir na saída console o que ocorreu.
- O terceiro método, getPhaseId(), especifica em qual das fases é invocado este ouvinte (Listener) e PhaseId.ANY_PHASE menciona que será para todas as fases.

Com a classe criada, o próximo passo é registrar esta classe TestPhaseListener no arquivo de configurações do JSF (faces-config.xml).

Para fazer isso, no arquivo faces-config.xml, adicionaremos os seguintes elementos do trecho a seguir:

```
...
<lifecycle>
    <phase-listener>
        br.jsf2.listener.TestPhaseListener
    </phase-listener>
</lifecycle>
...
```

Aula 15 - Usando conversores padrão

Conversão de números, datas e textos em JSF

Apesar do grande número de dados, na web apenas strings são tratados. Por exemplo, um objeto Date que devia ser analisado, é manipulado como String e, ao ser enviado ao servidor, convertido para Date. O mesmo comportamento se aplica aos números.

Para exemplificar, suponha uma caixa de textos que receba números representando um dado monetário. Devemos vincular um conversor ao campo atual com pelo menos dois dígitos após o ponto decimal, veja:

```
<h:inputText value="#{user.amount}">
    <f:convertNumber minFractionDigits="2"/>
</h:inputText>
```

Para testar, caro aluno, digite um valor, acrescentando vírgula para separar os centavos e verifique o resultado. O **JavaServer Faces** atribuiu 2 casas decimais, no mínimo, como a quantidade de dígitos decimais.

A manipulação de datas obedece a um padrão específico. Por exemplo, se desejar formatar uma data para o padrão brasileiro, considere os símbolos: **d** – dia, **M** – mês e **y** – ano. Assim, o padrão para formatação de datas segue o exemplo a seguir:

```
<h:outputText id="date" value="#{user.date}">
    <f:convertDateTime pattern="dd/MM/yyyy"/>
</h:outputText>
```

Analogamente, para formatar um texto, podemos usar:

```
<h:inputText id="cartao" value="#{user.cartao}">
    <f:validateLength minimum="4" />
</h:inputText>
```

Apresentamos as Tabelas 15.1a, 15.1b e 15.1c com os principais atributos para cada formatação.

Tabela 15.1a: Atributos da tag f:convertNumber

Atributo	Tipo	Valor
type	String	number (default), currency ou percent
pattern	String	Padrão de formatação, conforme definido em java.text. DecimalFormat
maxFractionDigits	int	Máximo de dígitos na parte fracionária
minFractionDigits	int	Mínimo de dígitos na parte fracionária
maxIntegerDigits	int	Máximo de dígitos na parte inteira
minIntegerDigits	int	Mínimo de dígitos na parte inteira
integerOnly	boolean	Verdadeiro se somente a parte inteira for extraída (default: false)
groupingUsed	boolean	Verdadeiro se separadores de grupos forem usados (padrão: true)
locale	java.util.Locale	Localização para parsing e formatação
currencyCode	String	Código monetário para valores de moedas
currencySymbol	String	Símbolo monetário para valores de moedas

Tabela 15.1b: Atributos da tag f:convertDateTime

Atributo	Tipo	Valor
type	String	date (default), time ou both
dateStyle	String	default, short, médium, long ou full
timeStyle	String	default, short, médium, long ou full
pattern	String	Padrão de formatação, conforme definido em java.text. DateFormat
locale	java.util.Locale	Localização para parsing e formatação

timeZone	java.util.TimeZone	Fuso horário a ser usado para <i>parsing</i> e formatação
-----------------	--------------------	---

Tabela 15.1c: Validação de strings e faixas numéricas

Tag JSP	Classe do Validador	Atributos	Valida
f:validateDoubleRange	DoubleRangeValidator	minimum, maximum	Um valor double dentro de uma faixa opcional
f:validateLongRange	LongRangeValidator	minimum, maximum	Um valor long dentro de uma faixa opcional
f:validateLength	LengthValidator	minimum, maximum	Uma String com um número mínimo e máximo de caracteres

Uso de propriedades e o arquivo de propriedades

Propriedades representam pares chave-valor, alguns definidos pelo programador e outras pré-definidas pela API.

As propriedades a serem usadas em um aplicativo são definidas em um arquivo de propriedades específico, com a extensão **.properties**, por exemplo, **messages.properties**. Geralmente o arquivo é localizado na mesma estrutura de pastas que contém as classes de usuários na aplicação.

Apresentaremos aqui, caro aluno, os passos para criar e manipular propriedades para uma interface JSF.

Supondo que esse arquivo exista, o próximo passo é definir um grupo de propriedades apropriadas ao seu aplicativo. Por exemplo, suponha que se deseja criar a interface mostrada na Figura 15.2.

Figura 15.2: Formulário JSF

Please enter the payment information

Amount	<input type="text" value="0.00"/>
Credit Card	<input type="text"/>
Expiration date (Month/Year)	<input type="text" value="09/2008"/>
Process	<input type="button"/>

O código **Fonte** é mostrado no Quadro 26.

Quadro 26: implementação do formulário da Figura 15.2

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <f:view>
    <h:head>
      <f:loadBundle basename="com.jsf2.messages" var="msgs" />
      <title><h:outputText value="#{msgs.title}" /></title>
    </h:head>
    <h:body>
      <h:form>
        <h1><h:outputText value="#{msgs.enterPayment}" /></h1>
        <h:panelGrid columns="3">
          <h:outputText value="#{msgs.amount}" />
          <h:inputText id="amount" label="#{msgs.amount}"
                       value="#{payment.amount}" />
          <f:convertNumber minFractionDigits="2"/>
        </h:inputText>
        <h:message for="amount" />
        <h:outputText value="#{msgs.creditCard}" />
        <h:inputText id="card" label="#{msgs.creditCard}"
                     value="#{payment.card}" />
        <h:panelGroup/>
        <h:outputText value="#{msgs.expirationDate}" />
        <h:inputText id="date"
                     label="#{msgs.expirationDate}"
                     value="#{payment.date}" />
        <f:convertDateTime pattern="MM/yyyy"/>
      </h:inputText>
      <h:message for="date" />
    </h:panelGrid>
    <h:commandButton value="#{msgs.process}" />
  <action="process"/>
  </h:form>
</h:body>
</f:view>
</html>

```

Analisando o código, podemos perceber que todo o texto da interface não está definido no código, mas são representados por propriedades de um objeto chamado msgs. Esse objeto foi definido na linha:

```
<f:loadBundle basename="com.corejsf.messages" var="msgs" />
```

Em outras palavras, o arquivo de propriedades é semelhante a uma classe e sua referência envolve a pasta em que está localizado. O elemento **f:loadBundle** permite definir um objeto msgs para acessar as propriedades. Observe, agora, o Quadro 27.

Quadro 27 - conteúdo do arquivo de propriedades (messages.properties)

```
title=An Application to Test Data Conversion
enterPayment=Please enter the payment information
amount=Amount
creditCard=Credit Card
expirationDate=Expiration date (Month/Year)
process=Process
paymentInformation=Payment information
```

Uma vez definido o arquivo e o objeto referenciado na página jsp, é necessário realizar esta alteração no arquivo faces-config.xml:

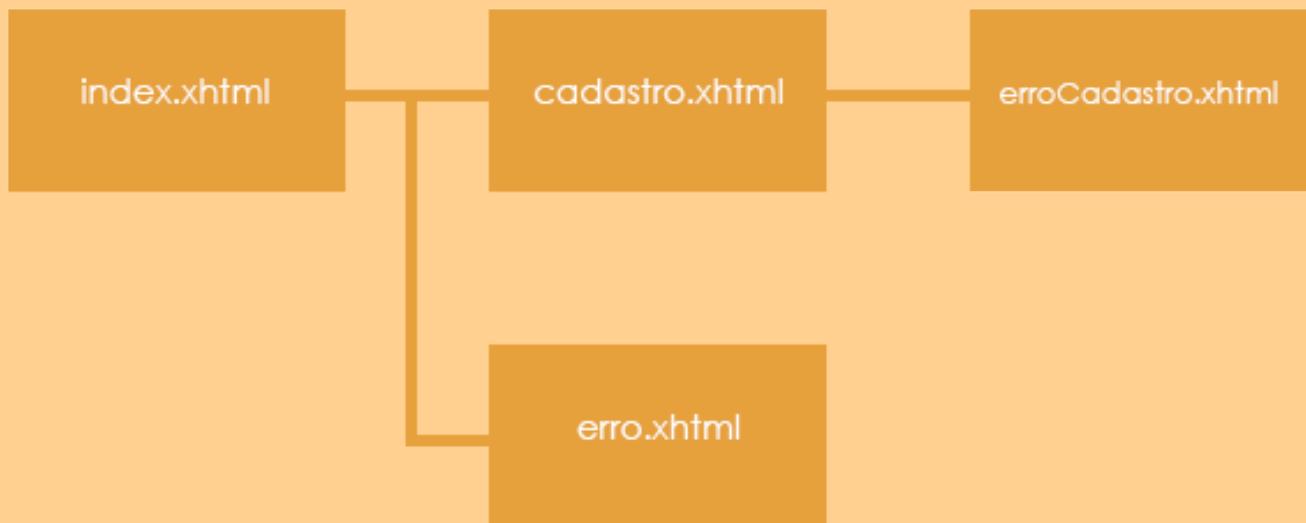
```
<application>
    <message-bundle>com.corejsf.messages</message-bundle>
</application>
```

Uma das várias utilidades do arquivo de propriedades é a definição de diferentes idiomas, de acordo com o idioma do usuário. Chamamos este conceito de **internacionalização**.

Existem diversas variáveis pré-definidas na API JSF, especialmente no que diz respeito às mensagens de erro para o usuário. Consulte a documentação para maiores detalhes.

Exercícios do Módulo 4

Este exercício trata da validação de usuários e cadastro de alunos, usando **JavaServer Faces 2.0**. O diagrama de navegação é dado abaixo:

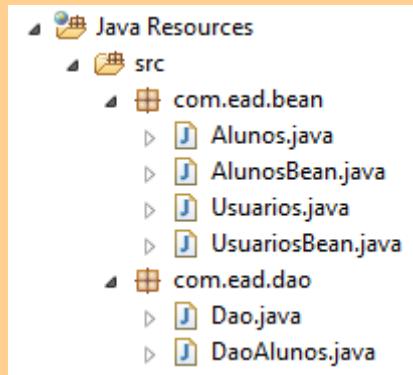


Exercício 1: Para desenvolver este exercício, siga os seguintes passos:

1. Defina um novo projeto web chamado **Modulo04_Exercicio**.
2. Neste projeto será necessário incluir a API do JSF. O arquivo deve ser colocado na pasta lib, abaixo de WEB-INF no projeto. Após incluir o arquivo, clicar com o direito do mouse e selecionar a opção: "**Add to Build Path**". O arquivo se chama javax.faces.X.jar, em que X é a versão. Ele pode ser obtido no link: <http://mvnrepository.com/artifact/javax.faces/>

javax.faces-api/2.1

3. Criar as classes para o projeto. As classes são mostradas na estrutura da seguinte Figura:



Classe Alunos:

```
package com.ead.bean;

import java.util.Date;

public class Alunos {
    private String curso, nome, email;
    private int rm;
    private Date dataNascimento;

    //getters e setters
}
```

Classe AlunosBean – Esta classe será usada como bean nas páginas xhtml:

```

package com.ead.bean;

import java.util.ArrayList;
import java.util.List;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.model.SelectItem;

import com.ead.dao.DaoAlunos;

@ManagedBean(name="aluno")
@RequestScoped
public class AlunosBean {
    private String mensagem;

    public String getMensagem() {
        return mensagem;
    }

    public List<SelectItem> getCursos(){
        List<SelectItem> lista = new ArrayList<SelectItem>();
        lista.add(new SelectItem("SCJ","Soluções Corporativas Java"));
        lista.add(new SelectItem("AOJ","Analise Orientada a Objetos Java"));
        lista.add(new SelectItem("MIT","Master in Information Tecnology"));
        return lista;
    }

    public List<Alunos> getListaAlunos() throws Exception {
        return new DaoAlunos().listarAlunos();
    }
    public String cadastrarAluno(Alunos aluno) throws Exception{
        DaoAlunos dao = new DaoAlunos();

        if(dao.cadastrarAluno(aluno)){
            mensagem = "RM " + aluno.getRm() + " inserido com sucesso!";
            return "";
        }
        else {
            mensagem = "Erro";
            return "";
        }
    }

    public String listarTodosAlunos(){
        return "";
    }
}

```

Classe Usuarios

```
package com.ead.bean;

public class Usuarios {
    private String nome, senha;

    //getters e setters
}
```

Classe UsuariosBean – classe usada com bean nas páginas xhtml. O método validarUsuario() deve ser completado pelo aluno.

```
package com.ead.bean;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name="usuariosBean")
@SessionScoped
public class UsuariosBean {
    private Usuarios usuario = new Usuarios();

    public Usuarios getUsuario() {
        return usuario;
    }

    public void setUsuario(Usuarios usuario) {
        this.usuario = usuario;
    }
    public String validarUsuario(){
        if( //realizar aqui a validação do usuário ){

            return "/cadastro";
        }
        else{
            return "/index";
        }
    }
}
```

Classe Dao: Vamos deixar para o aluno desenvolver esta classe, tomando como base a d=classe Dao do Módulo 4.

Classe DaoAlunos: Também deixaremos para o aluno desenvolver esta classe. Observe que a classe AlunosBean faz referência a ela.

Exercício 2: Desenvolver as páginas apresentadas no diagrama de navegação no início do exercício deste módulo. Para tanto, incluir um novo arquivo HTML no eclipse e mudar a extensão do arquivo para .xhtml. A página index.xhtml será dada como modelo:

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

  <f:view>
    <h:form>
      <table>
        <tr>
          <td>Usuário:</td>
          <td><h:inputText value="#{usuariosBean.usuario.nome}" /></td>
        </tr>
        <tr>
          <td>Senha:</td>
          <td><h:inputSecret value="#{usuariosBean.usuario.senha}" /></td>
        </tr>
        <tr>
          <td colspan="2"><h:commandButton value="Enviar"
            action="#{usuariosBean.validarUsuario}" /></td>
        </tr>
      </table>
    </h:form>
  </f:view>
</html>
```

Módulo 5 – Web services

Aula 16 – Conhecendo os Web services

Definição de Web services

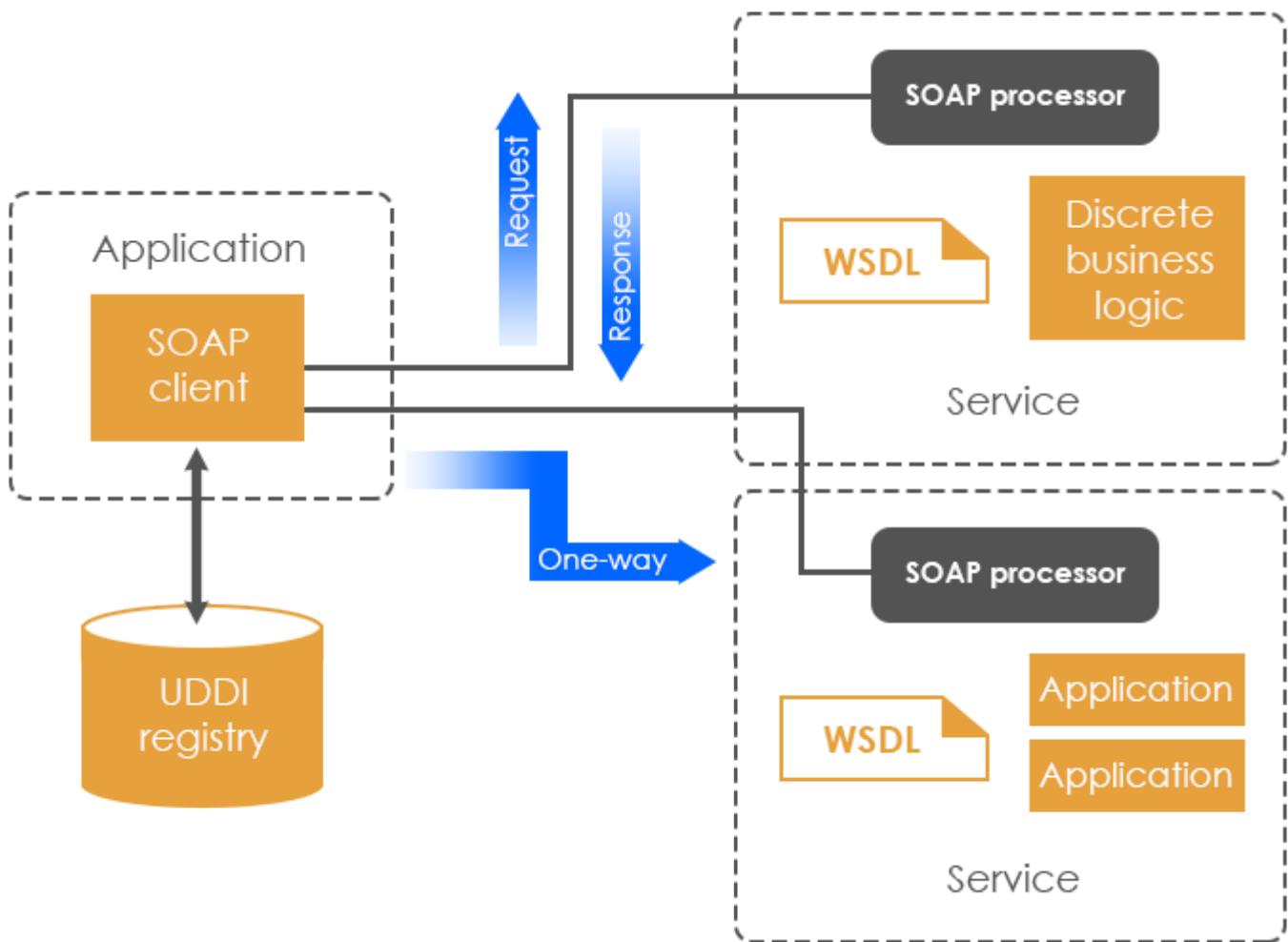
Um requisito básico de qualquer aplicação é prover serviços. Cada empresa oferece serviços para a sua comunicação com outras pessoas, sejam pessoas físicas ou jurídicas, internas ou externas. *Web services* foram criados para construir aplicações que são serviços na *internet*. Porém, não faz parte do conceito de *Web service* a criação de interfaces gráficas para os usuários, deixando esta parte para outras pessoas desenvolverem.

Web services é a tecnologia ideal para comunicação entre sistemas, sendo muito usado em aplicações B2B (Business 2 Business). A comunicação entre os serviços é padronizada, possibilitando a independência de plataforma e de linguagem de programação.

Por exemplo, um sistema de reserva de passagens aéreas feito em Java e rodando em um servidor Linux pode acessar, com transparência, um serviço de reserva de hotel feito em .net rodando em um servidor Microsoft.

Para comunicar com o *Web service*, é necessário uma implementação do protocolo SOAP (Simple Object Access Protocol) definido no W3C. Esse protocolo é o responsável pela independência que o *Web service* precisa.

Na Figura 16.1a encontra-se um diagrama mostrando as mensagens trocadas entre cliente e servidor em uma comunicação SOAP. Existem duas aplicações se comunicando, um Client Wrapper e um Server Wrapper que estão disponibilizando a transparência para as aplicações. Entre eles só trafega XML, seguindo o protocolo SOAP sobre HTTP.

Figura 16.1a: Processamento de um *Web service*

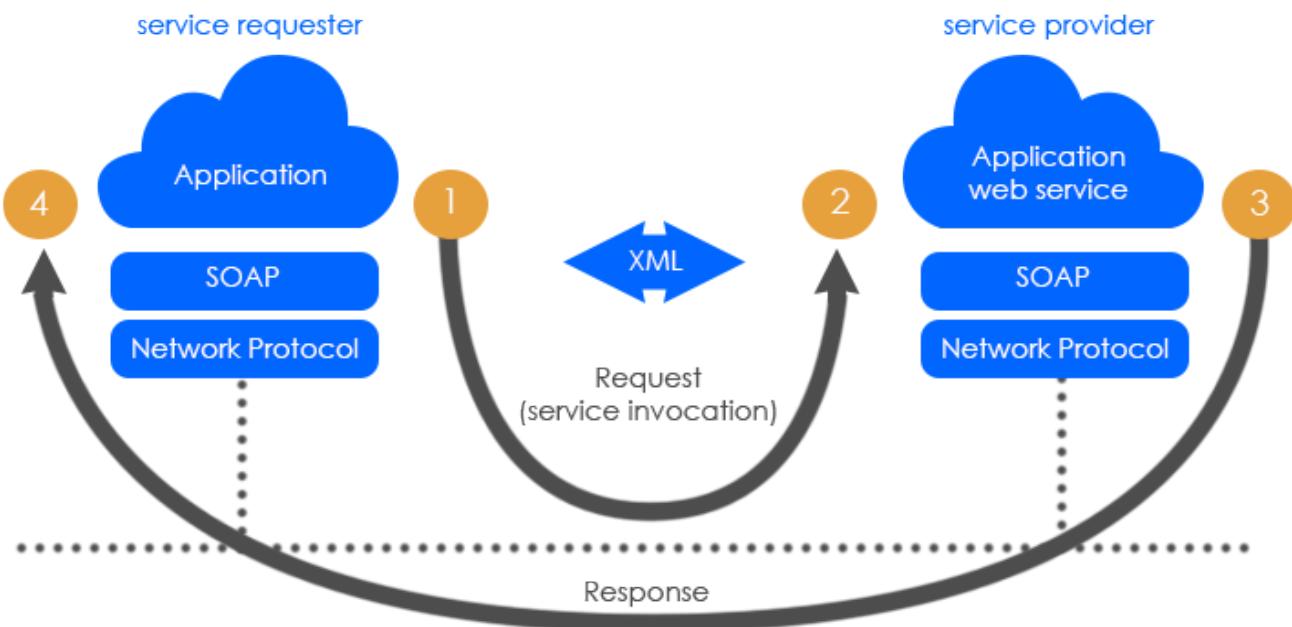
Fonte: (KALIN, 2010).

Um *Web service* será publicado, e para que outras pessoas possam utilizá-lo, é necessário definir como ele é, como deve ser acessado e que valores ele retornará. Estas definições são descritas em um arquivo XML de acordo com a padronização *Web service Description Language* (WSDL). Este arquivo deve ser construído para que os usuários do serviço possam entender o funcionamento do *Web service* e será de acesso público.

Os *Web services* são “escritos” basicamente em SOAP e WSDL, ambas baseadas em XML. Como SOAP é a mensagem e WSDL o descritor desta mensagem, é importante compreender o conceito de XML e XSD, usados posteriormente para interpretar informações no formato XML, através do JAX-WS.

Um *Web service* é uma lógica de aplicativo acessível a programas através de protocolos *web* de um modo independente de plataforma. Um *Web service* expõe a lógica de aplicativo ou de código. Esse código pode realizar diversas tarefas.

O conceito de *Web services* é baseado em um conjunto de protocolos *web*, como HTTP, XML, SOAP, WSDL e UDDI e podem ser implementados em qualquer plataforma. O *Web service* é acessado por uma aplicação, de qualquer natureza, como mostrado na Figura 16.1b.

Figura 16.1b: Requisição / resposta SOAP em um *Web service*

Aula 17 - SOAP

Definição de SOAP

O protocolo SOAP (Simple Object Access Protocol) realiza a comunicação entre o cliente e o servidor através do protocolo HTTP. Por meio do SOAP especificamos o endereço da máquina para a qual se estabelece a comunicação.

SOAP é um protocolo simples e leve baseado em XML que proporciona troca de informações em cima do protocolo HTTP. É um protocolo para acessar *Web services*.

A arquitetura tem sido desenhada para ser independente de qualquer modelo particular de programa e de outras implementações específicas.

Os dois maiores objetivos do SOAP são a simplicidade e extensibilidade, e esse protocolo obedece a esses requisitos, pois trafega em cima do HTTP e o HTTP é suportado por todos os servidores e browsers do mercado, com diferentes tecnologias e linguagens de programação.

Estrutura de um arquivo SOAP

Uma requisição para um servidor em SOAP possui a seguinte estrutura:

```
<SOAP-ENV:envelope>
<SOAP-ENV:header>
<!--Especifica outros dados da mensagem(é opcional)-->
</SOAP-ENV:header>
<SOAP-ENV:body>
<!--O elemento BODY contém a mensagem em si-->
</SOAP-ENV:body>
</SOAP-ENV:envelope>
```

Agora vejamos a estrutura da resposta:

```
<SOAP-ENV:envelope>
<SOAP-ENV:body>
<!--A resposta do servidor-->
</SOAP-ENV:body>
</SOAP-ENV:envelope>
```

Pela requisição e pela resposta, podemos verificar que um arquivo SOAP possui a seguinte estrutura:

- **Elemento Envelope**

Responsável por definir o conteúdo da mensagem

- **Elemento Header (opcional)**

Dados do cabeçalho

- **Elemento Body**

Contém as informações de chamada de resposta ao servidor

- **Elemento Fault**

Possui as informações dos erros ocorridos no envio da mensagem. Esse elemento, obviamente, só aparece nas mensagens de resposta do servidor.

O **namespace** padrão (o namespace define as regras de codificação que o documento deve seguir) para o elemento Envelope possui a estrutura:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/
  soap:encodingstyle="http://schemas.xmlsoap.org/soap/encoding/">
  <!--Dados da mensagem aqui-->
</soap:Envelope>
```

À medida que se avança na estrutura do SOAP, é possível ver que sua codificação é bastante simples.

Atributos SOAP

A estrutura de atributos do SOAP é semelhante à estrutura de atributos da XML, e é formada por:

- **actor**: o atributo actor define a URI (equivalente à URL do HTTP), a qual o HEADER se refere. Lembrando que o elemento HEADER é opcional dentro do SOAP. Veja a sintaxe deste atributo:

```
<soap:Header>
<!--Aqui definimos o namespace como sendo "r" para personalizar nosso documento SOAP, você
pode criar o seu próprio namespace-->
<r:mercado xmlns:r="http://www.mercadao.com.br/valores/">
  soap:actor="http://www.mercadao.com.br/descricao" />
  <r:lingua>port</r:lingua>
  <r:dinheiro>REAL</r:dinheiro>
</r:mercado>
</soap:Header>
```

- **encodingStyle**: esse atributo serve para definir um estilo de codificação do documento. Veja:

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/
  soap:encodingstyle="http://schemas.xmlsoap.org/soap/encoding/">
  Mensagem aqui
</soap:Envelope>
```

- **mustUnderstand**: define qual elemento do HEADER deve aparecer para o receptor da mensagem. O valor "0" deste atributo significa que o elemento não deve aparecer para o receptor e o valor "1" significa que esse elemento deve ser visto pelo receptor da mensagem.

```
<soap:Header>
  <r:mercado xmlns:r="http://www.mercadao.com.br/valores/" />
  <r:lingua soap:mustUnderstand="0">port</r:lingua>
  <r:dinheiro soap:mustUnderstand="1">REAL</r:dinheiro>
</r:mercado>
</soap:Header>
```

Aula 18 - WSDL

Definição de WSDL

WSDL é um formato XML para descrever serviços de rede, contendo um conjunto de operações a serem executadas no sistema remoto, ele é extensível para permitir a descrição do serviço. Suas mensagens não levam em consideração quais formatos de mensagens ou protocolos de rede são usados para a comunicação.

Como protocolos de comunicação e formatos de mensagens são padronizadas na comunidade web, torna-se importante ser capaz de descrever as comunicações em uma forma estruturada. Os endereços WSDL necessitam, por definição, de uma gramática XML para descrever serviços de rede como coleções de *endpoints* capazes de definir mensagens a serem transferidas como serviços.

Um serviço WSDL define documentos como coleções de endpoints de rede, ou portas. No WSDL, a definição abstrata de pontos finais e mensagens são separadas de suas organizações de redes concretas ou formatos de dados de comunicação. Isto permite a reutilização de definições abstratas: mensagens, que são descrições abstratas dos dados sendo trocados, e tipos de portas, que são coleções abstratas de operações.

Estrutura WSDL

A estrutura do WSDL é assim composta:

```
<definitions>
    <types>
        definition of types.....
    </types>
    <message>
        definition of a message....
    </message>
    <portType>
        definition of a port.....
    </portType>
    <binding>
        definition of a binding....
    </binding>
</definitions>
```

Nesta estrutura, podem ser identificados:

- **Types (tipos)** – Um recipiente para definição de tipos de dados usando alguns tipos de sistemas. WSDL usa sintaxe XML para definir tipos de dados.
- **Message (mensagens)** – Um resumo de definições de tipos de dados sendo trafegados, pode conter uma ou mais partes, essas partes podem ser comparadas a parâmetros de uma função.

- **portType (Tipo de porta)** – Um resumo da configuração das operações suportadas por um ou mais endpoints.
- **Binding (Ligaçao)**: Define o formato da mensagem e detalhes de protocolos para cada porta.

WSDL ports

WSDL Ports são o elemento mais importante no WSDL. Este elemento define o local onde o serviço será encontrado. Operation Types (Tipos de operação) – são os modos como os serviços recebem requisições e enviam suas respostas, quais sejam:

- One-Way – A operação pode receber uma mensagem, mas não irá retornar uma resposta.
- Request-response – A operação pode receber uma requisição e retornar uma resposta.
- Solicit-response – A operação pode enviar uma requisição e esperar por uma resposta.

Bindings SOAP

O elemento Binding tem dois atributos, *name* e *type*. O atributo *name* define o nome da ligação e o *type* indica o ponto para a ligação da porta, nesse caso o “glossary terms port”.

O elemento **soap:binding** tem dois atributos, *style* e *transport*. O atributo *style* pode ser o “**rpc**” ou “**documento**”, o **transport** define o protocolo SOAP a ser usado, no caso o http.

Aula 19 - Conceitos de Web services JAX-WS

Definição de JAX-WS

A API Java para Web services XML (JAX-WS), JSR 224, é uma parte importante das plataformas Java EE 5 e EE 6. O JAX-WS simplifica a tarefa de desenvolvimento de *Web services* utilizando a tecnologia Java. Aborda alguns dos problemas em JAX-RPC 1.1, fornecendo suporte a vários protocolos, como SOAP 1.1, SOAP 1.2, XML, e fornecendo um recurso para dar suporte a protocolos adicionais junto com HTTP. O JAX-WS dá suporte a personalizações para controlar interfaces de ponto final de serviço geradas. Com suporte a anotações, o JAX-WS simplifica o desenvolvimento do *Web service* e reduz o tamanho de arquivos JAR do run-time.

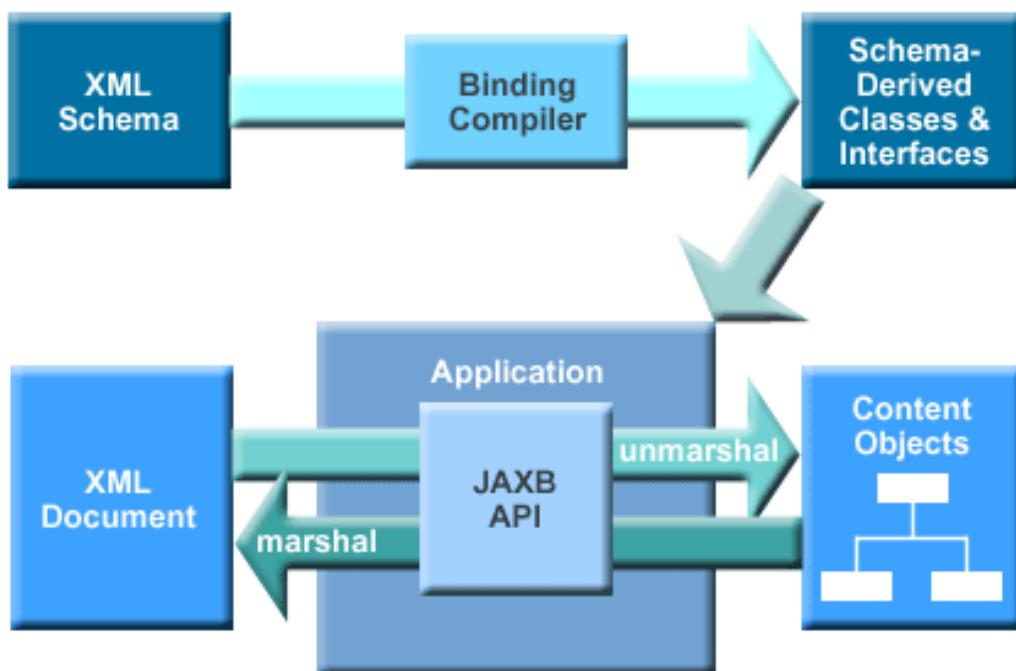
Vinculando WSDL a Java com JAXB

O XML baseado em JAXB facilita o acesso aos documentos XML com base em aplicações escritas na linguagem de programação Java. O JAXB é uma alternativa ao uso de parser SAX ou DOM para acessar os dados em um documento XML. Pode-se usar JAXB para construir um documento XML.

Para construir um documento XML com JAXB, primeiro vincule o esquema do documento XML que deseja construir. Depois, crie uma árvore de conteúdo. Por último, **empacote a árvore de conteúdo** em um documento XML.

O diagrama da Figura 19.2a, mostra os processos para acessar e construir documentos XML com base em aplicações Java.

Figura 19.2a: Arquitetura JAXB



Fonte: <<https://netbeans.org/kb/74/websvc/jaxb.html>>. Acesso em: 04 set. 2014.

A seguir apresentaremos um exemplo de um *Web service* JAX-WS. Após criar o *Web service*, escreveremos um cliente Java *Servlet*.

- **Passo 1: Criar o *Web service***

Vamos criar um *Web service* que contenha duas operações:

- ◊ a primeira recebe como parâmetro uma String e a retorna em ordem invertida.
- ◊ a segunda retorna uma lista de cidades com base em um estado, fornecido como parâmetro.

Nosso *Web service* se chamará OperacoesWS e seu conteúdo será o apresentado no Quadro 28.

Quadro 28 – Web service OperacoesWS

```

package com.webservice;

import java.util.ArrayList;
import java.util.List;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

@WebService(serviceName = "OperacoesWS")
public class OperacoesWS {

    @WebMethod(operationName = "reverterTexto")
    public String reverterTexto(@WebParam(name = "texto")

                                String texto) {
        return new StringBuilder(texto).reverse().toString();
    }

    @WebMethod(operationName = "ListarCidades")
    public List<String> ListarCidades(@WebParam(name = "estado")

                                         String estado) {
        List<String> lista = new ArrayList<String>();
        if(estado.equals("SP")){
            lista.add("São Paulo");
            lista.add("Campinas");
            lista.add("Ribeirão Preto");
        }
        else if(estado.equals("AC")){
            lista.add("Rio Branco");
            lista.add("Sena Madureira");
            lista.add("Cruzeiro do Sul");
        }
        return lista;
    }
}

```

Observe as anotações `@WebService`, `@WebMethod` e `@WebParam`. Elas representam, respectivamente, as definições para um `WebService`, um método de serviço e um parâmetro de serviço.

Os parâmetros para cada anotação são apresentados nas Tabelas 19.2a, 19.2b e 19.2c.

Tabela 19.2a: Parâmetros para a anotação @WebService

Parâmetro	Definição
endpointInterface	O nome completo da interface que representa o endpoint do serviço, ou seja, o contrato <i>Web service</i> .
name	O nome do <i>Web service</i> .
portName	O nome da porta do <i>Web service</i> .
serviceName	O nome do serviço do <i>Web service</i> .
targetNamespace	Se a anotação @WebService.targetName representa uma interface, é usado para o namespace para o wsdl:portType.
wsdlLocation	A localização do WSDL que descreve o serviço.

Tabela 19.2b: Parâmetros para a anotação @WebMethod

Parâmetro	Definição
action	Ação para esta operação.
exclude	Marca um método para não ser exposto como um serviço.
operationName	Nome da operação wsdl:operation combinando com este método.

Tabela 19.2c: Parâmetros para a anotação @WebParam

Parâmetro	Definição
header	Se for true, o parâmetro é buscado do cabeçalho da mensagem, ao invés do corpo da mensagem.
mode	Direção do fluxo do parâmetro (IN, OUT, ou INOUT).
name	Nome do parâmetro.
partName	Nome do wsdl:part representando este parâmetro.
targerNamespace	O namespace XML para o parâmetro.

A execução deste *Web service* produz o resultado mostrado na Figura 19.2b, usando-se o servidor Tomcat:

Figura 19.2b: Resultado da execução do *Web service*

<http://localhost:8084/Operacoes/OperacoesWS>

Web Services

Endpoint	Information
Service Name: {http://webservice.com/}OperacoesWS Port Name: {http://webservice.com/}OperacoesWSPort	Address: http://localhost:8084/Operacoes/OperacoesWS WSDL: http://localhost:8084/Operacoes/OperacoesWS?wsdl Implementation class: com.webservice.OperacoesWS

Um cliente deste serviço é gerado a partir do WSDL produzido através da URL:

<http://localhost:8084/Operacoes/OperacoesWS?wsdl>

Ele permite o download do WSDL para verificar os dados fornecidos pelo cliente. O conteúdo do WSDL é mostrado no Quadro 29.

Quadro 29 – Conteúdo do WSDL

```
<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy" xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://webservice.com/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://webservice.com/" name="OperacoesWS">
<types>
<xsd:schema>
<xsd:import namespace="http://webservice.com/" schemaLocation="http://localhost:8084/Operacoes/OperacoesWS?xsd=1"/>
</xsd:schema>
</types>
<message name="ListarCidades">
<part name="parameters" element="tns>ListarCidades"/>
</message>
<message name="ListarCidadesResponse">
<part name="parameters" element="tns>ListarCidadesResponse"/>
</message>
<message name="reverterTexto">
<part name="parameters" element="tns:reverterTexto"/>
</message>
<message name="reverterTextoResponse">
<part name="parameters" element="tns:reverterTextoResponse"/>
</message>
<portType name="OperacoesWS">
<operation name="ListarCidades">
<input wsam:Action="http://webservice.com/OperacoesWS/ListarCidadesRequest"
       message="tns>ListarCidades"/>
<output wsam:Action="http://webservice.com/OperacoesWS/ListarCidadesResponse"
        message="tns>ListarCidadesResponse"/>
</operation>
<operation name="reverterTexto">
<input wsam:Action="http://webservice.com/OperacoesWS/reverterTextoRequest"
       message="tns:reverterTexto"/>
<output wsam:Action="http://webservice.com/OperacoesWS/reverterTextoResponse"
        message="tns:reverterTextoResponse"/>
```

```
</operation>
</portType>
<binding name="OperacoesWSPortBinding" type="tns:OperacoesWS">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document"/>
<operation name="ListarCidades">
<soap:operation soapAction="" />
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
<operation name="reverterTexto">
<soap:operation soapAction="" />
<input>
<soap:body use="literal"/>
</input>
<output>
<soap:body use="literal"/>
</output>
</operation>
</binding>
<service name="OperacoesWS">
<port name="OperacoesWSPort" binding="tns:OperacoesWSPortBinding">
<soap:address
    location="http://localhost:8084/Operacoes/OperacoesWS"/>
</port>
</service>
</definitions>
```

O cliente, como dito anteriormente, é um *servlet*. Este *servlet* recebe de um formulário um texto e um estado, e apresenta o resultado. O código para o cliente é mostrado no Quadro 30.

Quadro 30 – Código para o cliente

```
package com.webservice.client;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "ClienteWSServlet", urlPatterns = {"/clientews"})
public class ClienteWSServlet extends HttpServlet {
    @Override
    protected void doGet( HttpServletRequest request,
                         HttpServletResponse response)
        throws ServletException, IOException {
    }

    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        String texto = request.getParameter("texto");
        String estado = request.getParameter("estado");

        out.print("Texto fornecido: " + texto);
        out.print("<br>Texto invertido: " +
                 reverterTexto(texto));

        out.print("<br><br>Estado fornecido: " + estado);
    }
}
```

```
    out.print("<br>Cidades: " + listarCidades(estado));
}

private static List< String> listarCidades(String estado) {
    com.webservice.OperacoesWS_Service service = new
        com.webservice.OperacoesWS_Service();
    com.webservice.OperacoesWS port =
        service.getOperacoesWSPort();
    return port.listarCidades(estado);
}

private static String reverterTexto(String texto) {
    com.webservice.OperacoesWS_Service service = new
        com.webservice.OperacoesWS_Service();
    com.webservice.OperacoesWS port =
        service.getOperacoesWSPort();
    return port.reverterTexto(texto);
}
}
```

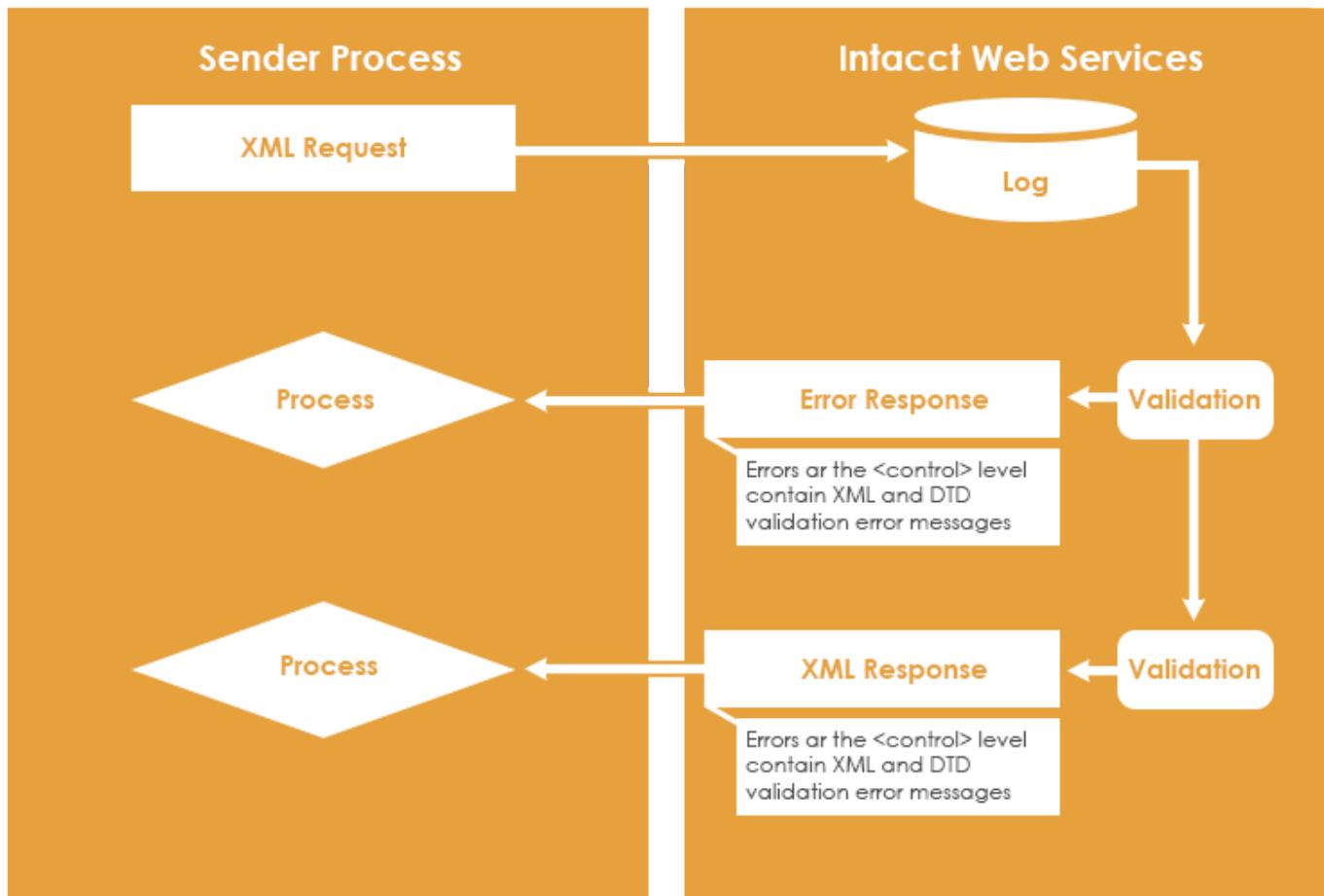
Uma vez que o *Web service* é exposto, o cliente executa seus métodos (remotos) seguindo a forma tradicional.

Processamento Síncrono

Operações síncronas consistem de chamadas que são executadas sem interrupção. Esta chamada bloqueia o processo até que a operação seja completada. O diagrama da Figura 19.3 ilustra o processo síncrono.

Figura 19.3: Processamento Síncrono

Synchronous Request-response Communication



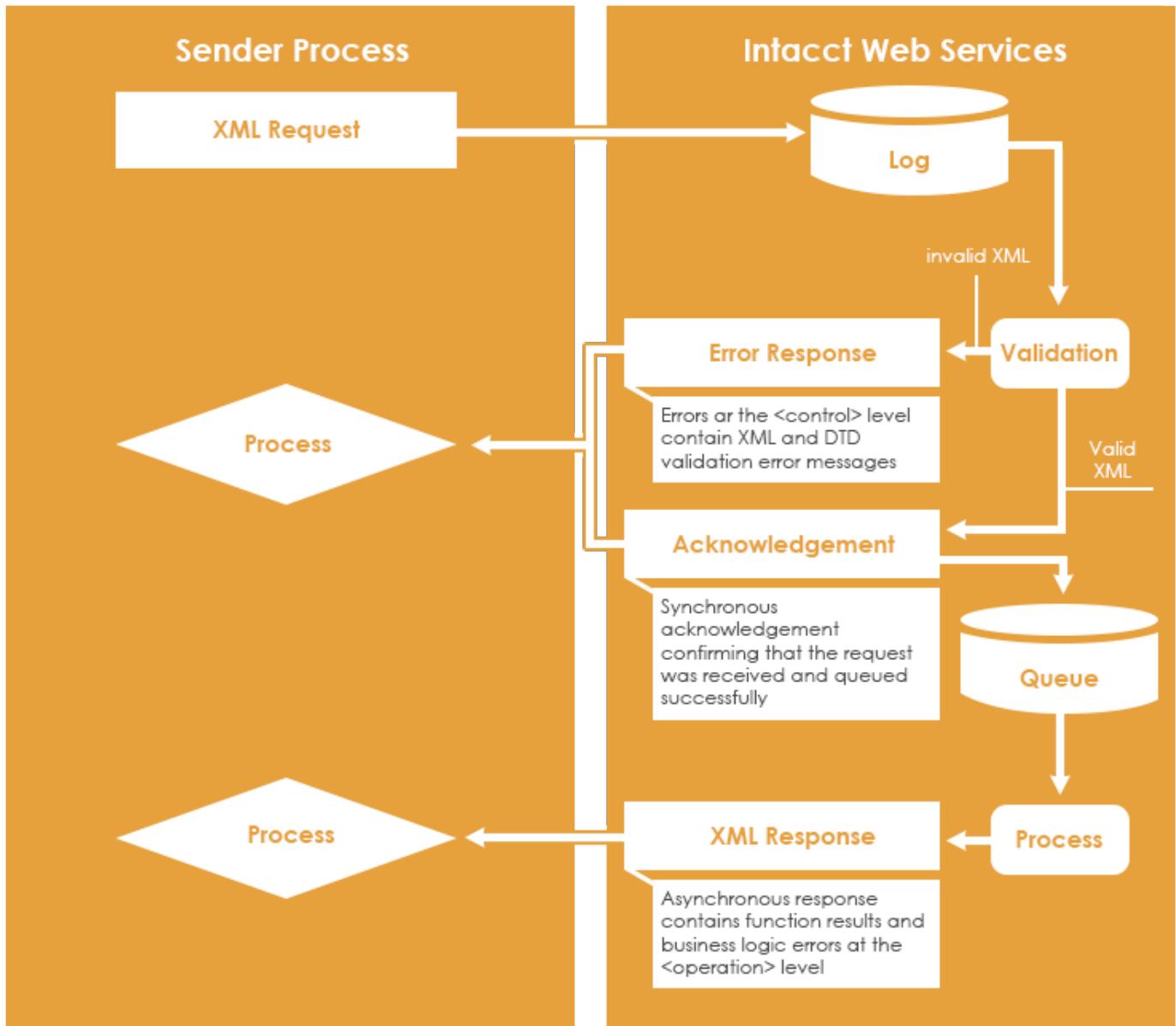
Processamento Assíncrono

Uma operação assíncrona não irá bloquear a chamada ao processo, que simplesmente inicia a operação. Esta operação deverá retornar uma chamada se a aplicação chamadora precisar ser notificada sobre a operação.

O diagrama da Figura 19.4 ilustra o conceito do processo assíncrono.

Figura 19.4: Processamento Assíncrono

Synchronous Request-response Communication



O procedimento de uma chamada assíncrona não é tão trivial como uma chamada síncrona, veja as etapas:

- O cliente obtém um ponteiro para o servidor e chama o método assincronamente.
- O cliente inclui um ponteiro para função callback.
- A chamada retorna imediatamente e a *thread* chamada é livre para executar outras linhas de instrução.
- Quando o processamento do método é finalizado, o servidor notifica o cliente através da função callback.

Exemplo de chamada assíncrona

Chamar um *Web service* envolve fazer uma chamada por uma rede que poderia potencialmente causar atrasos por fatores acima de seu controle (por exemplo, tráfego pesado de rede). Em muitos casos, você poderá querer iniciar uma chamada e continuar a executar sua aplicação enquanto espera o retorno do *Web service*. Isto pode ser feito através da chamada assíncrona do *Web service*.

Por padrão, as chamadas a *Web services* são síncronas.

No nosso exemplo de aplicação, o WSDL deve ser alterado para que operações assíncronas sejam permitidas. Isso é mostrado no Quadro 31.

Na aplicação cliente, uma chamada assíncrona deve ser prevista, como mostra o Quadro 32. No código do Quadro 32, junto com a chamada do *Web service*, observa-se que a resposta do serviço **ReverterTexto** é manipulada por meio de um objeto **AsynchHandler**. Enquanto isso, um objeto **Future** verifica se foi retornado um resultado e inativa o *thread* até que o resultado seja concluído.

Quadro 31 – Alterações no WSDL

```
...
<portType name="OperacoesWS">
<operation name="ListarCidades">
<input wsam:Action="http://webservice.com/OperacoesWS>ListarCidadesRequest"
message="tns>ListarCidades" />
<output wsam:Action="http://webservice.com/OperacoesWS>ListarCidadesResponse" message="tns:
ListarCidadesResponse" />
</operation>
<operation name="reverterTexto">
<input wsam:Action="http://webservice.com/OperacoesWS/reverterTextoRequest"
message="tns:reverterTexto" />
<output wsam:Action="http://webservice.com/OperacoesWS/reverterTextoResponse"
message="tns:reverterTextoResponse" />
</operation>

<jaxws:bindings>
    <jaxws:enableAsyncMapping>
        true
    </jaxws:enableAsyncMapping>
</jaxws:bindings>
</portType>

...

```

Quadro 32 – Chamada assíncrona

```
public void callAsyncCallback(String text){  
    try { // Call Web service Operation(async. callback)  
        com.webservice.OperacoesWS_Service service = new  
        com.webservice.OperacoesWS_Service();  
        com.webservice.OperacoesWS port =  
        getOperacoesWSPort();  
        // TODO initialize WS operation arguments here  
        java.lang.String texto = text;  
        AsyncHandler<com.webservice.  
ReverterTextoResponse>  
        asyncHandler = new  
        AsyncHandler<com.webservice.ReverterTextoResponse>() {  
            public void handleResponse(javax.xml.ws.Response<com.webservice.  
ReverterTextoResponse> response) {  
                try {  
                    // TODO process asynchronous response here  
                    System.out.println("Result = " + response.get());  
                } catch (Exception ex) {  
                    // TODO handle exception  
                }  
            }  
        };  
        java.util.concurrent.Future<? extends java.lang.Object> result = port.  
        reverterTextoAsync(texto, asyncHandler);  
        while (!result.isDone()) {  
            // do something  
            Thread.sleep(100);  
        }  
    } catch (Exception ex) {  
        // TODO handle custom exceptions here  
    }  
}
```

Considerações Finais

Caros alunos, estamos encerrando mais uma etapa de nosso curso!

Pudemos aplicar os conceitos de herança, polimorfismo, interfaces, tratamento de erros, acesso a banco de dados e *threads* na criação de aplicativos para *web*, especialmente no tratamento de *thread*, gerenciamento de sessão e definição de *beans* em um *framework*: o **JavaServer Faces**!

Pudemos constatar que todas as classes que criamos no Java SE puderam ser aproveitadas nas nossas aplicações *web*. Constatamos, também, a importância da orientação a objetos, quando definimos classes capazes de interagir com outros tipos de aplicações além da *web*, como, por exemplo, aplicações *swing*.

Da mesma forma que usamos o mecanismo de acesso a dados baseado em JDBC, podemos também utilizar outros *frameworks*, como o Hibernate, o JPA, o Ibatis, dentre outros, todos usando classes capazes de serem reaproveitadas.

Todas essas aplicações utilizam tudo o que estudamos.

Agora todos estão aptos a desenvolverem aplicações de mercado! Aproveitem ao máximo este novo potencial!

Prof. Emilio

Respostas Comentadas dos Exercícios

Exercícios do Módulo 1

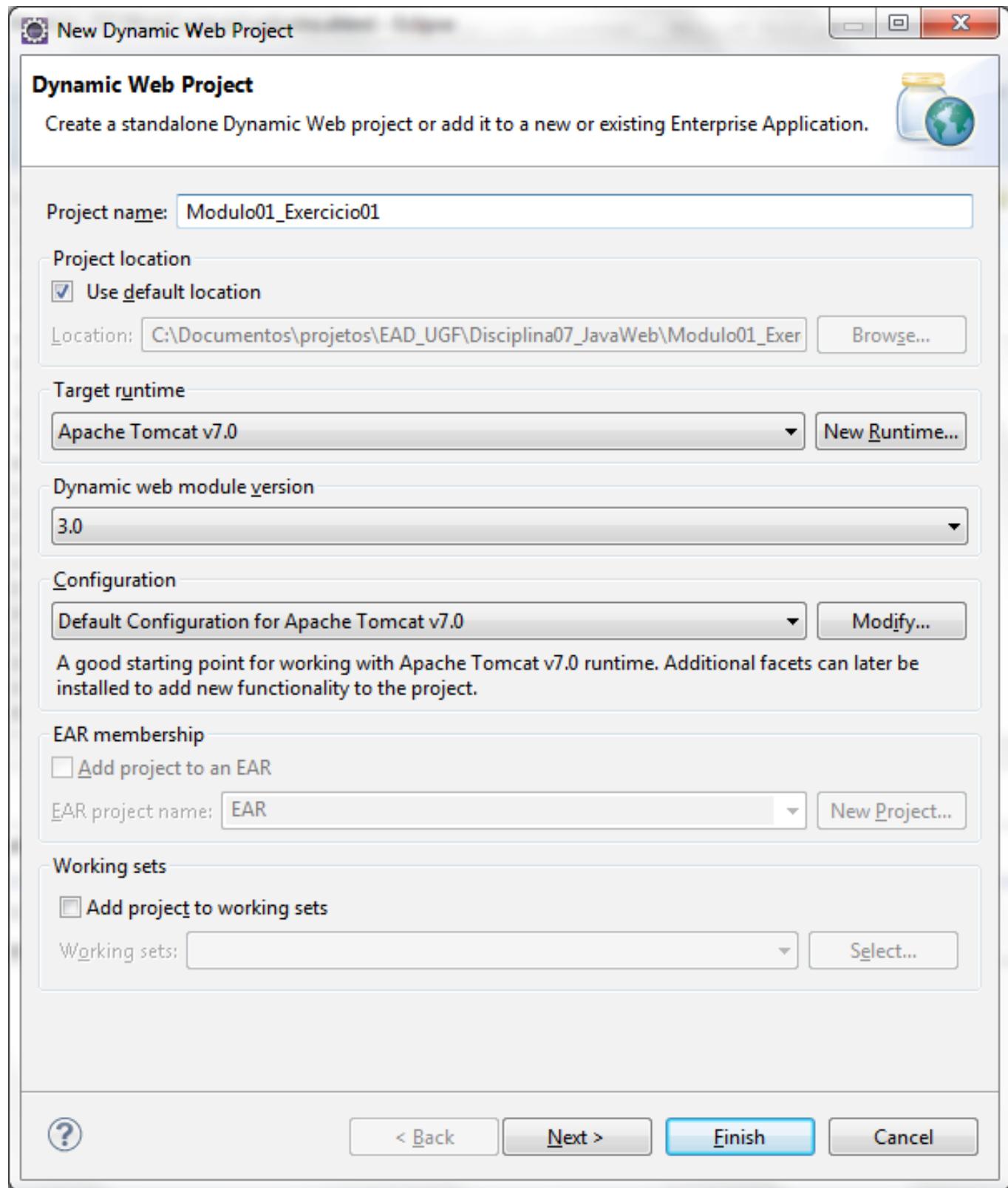
Exercício 1: Calculadora

Neste exercício criaremos uma calculadora web. Os operandos e a operação são informados como parâmetro na URL. Siga os passos para criar a aplicação:

1. Criar um projeto no Eclipse do tipo “Dynamic Web Project” chamado **Modulo01_Exercicio01**.
Seguir os passos descritos neste módulo para criar o projeto e associar o servidor Tomcat ao projeto.

Solução

Para criar o projeto, usamos o Eclipse, com a perspectiva Java EE. É necessário conhecer o local de instalação do servidor Tomcat:

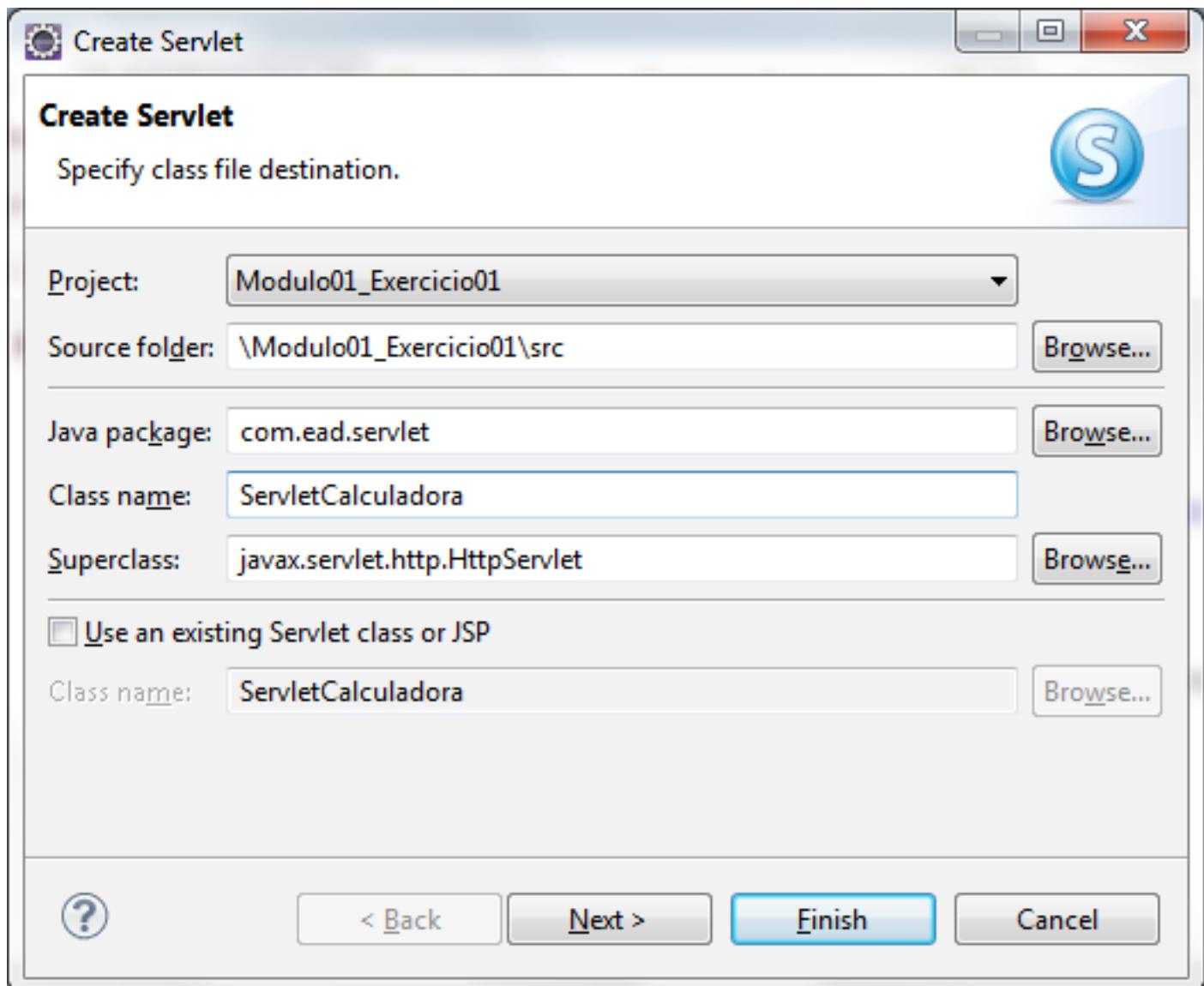


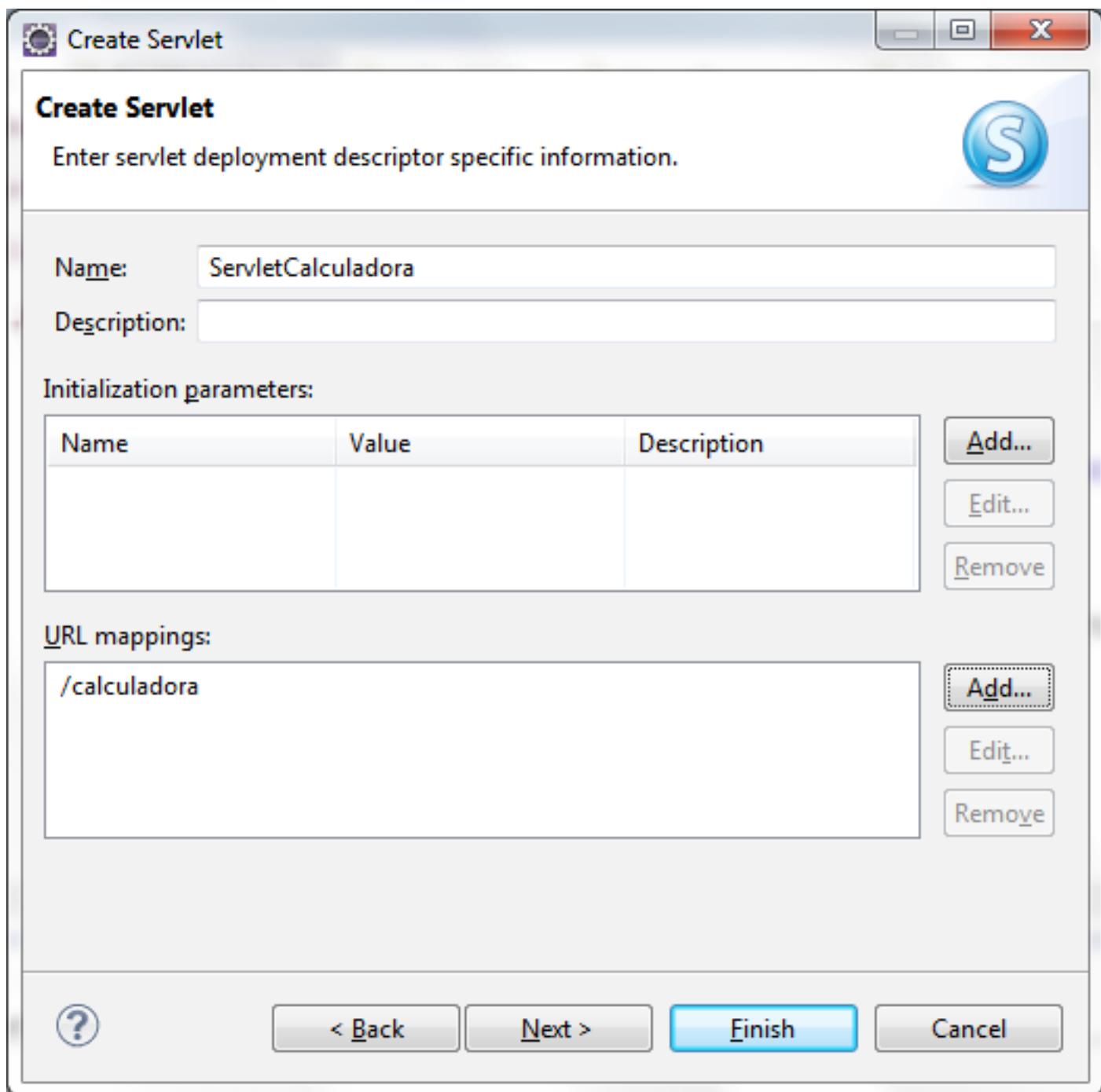
2. Neste projeto, incluir um novo servlet, com as seguintes configurações:

- a. Classe: **ServletCalculadora**
- b. Nome: **ServletCalculadora**
- c. url mapping: **/calculadora**

Solução:

Com o botão direito do mouse sobre o projeto, selecionar a opção Servlet. Incluir as informações a seguir:





1. No método **doGet()** do servlet, incluir o seguinte código:

```

PrintWriter out = response.getWriter();
response.setContentType("text/html");

try {
    double op1 = Double.parseDouble(request.getParameter("op1"));
    double op2 = Double.parseDouble(request.getParameter("op2"));
    int operacao = Integer.parseInt(request.getParameter("op"));
    double resultado;

    switch(operacao){
        case 1: resultado = op1 + op2; break;
    }
}

```

```

        case 2: resultado = op1 - op2; break;
        case 3: resultado = op1 * op2; break;
        case 4: resultado = op1 / op2; break;
    default: resultado = 0;
}
out.print("Resultado: " + resultado);
} catch (Exception e) {
    out.print(e.getMessage());
}

```

Solução:

Neste tópico não há o que acrescentar. Basta copiar o trecho de código no método doGet().

4. Executar a aplicação no Eclipse. Você verá que na URL aparecerá um erro no browser. Escrever, então na URL o seguinte código:
http://localhost:8080/Modulo01_Exercicio01/calculadora?op1=10&op2=20&op=2
5. Alterar os valores dos parâmetros op1, op2 e op, analisando o resultado.

Solução:

Neste exercício o aluno deverá interagir com o browser, fornecendo as informações solicitadas na URL para testar seus resultados.

Exercícios do Módulo 2

Exercício 1: Formulário para validação de usuário

Iniciaremos nossa aplicação através de uma série de exercícios. Neste exercício ciraremos um formulário para fornecer os dados do login de um usuário. Para tanto, siga os passos a seguir:

1. Criar um novo projeto no Eclipse chamado **Modulo02_Exercicio01**

Solução:

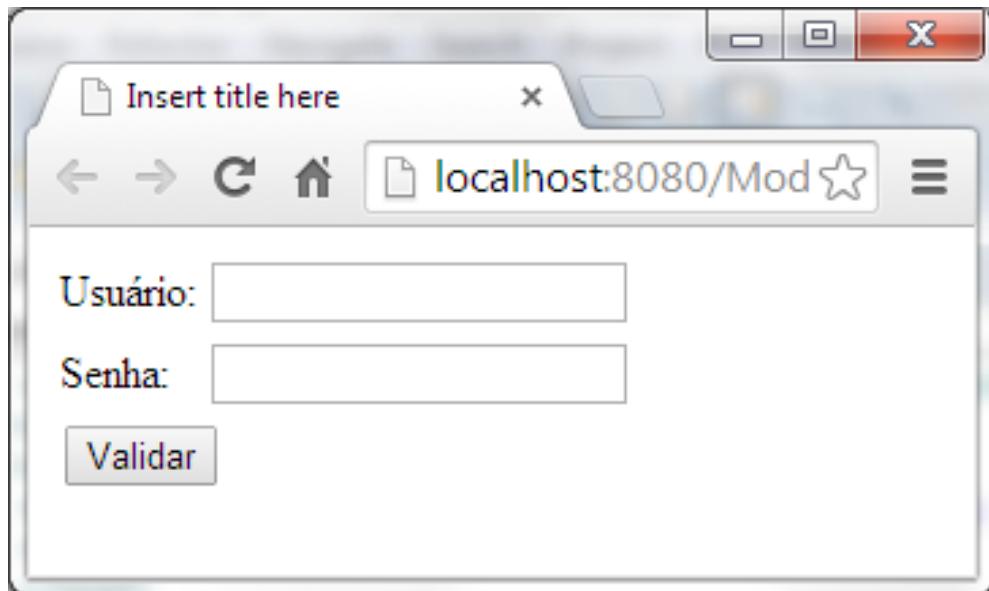
Seguir o mesmo procedimento usado no Exercício 01 do Módulo 01 para criar um novo projeto.

2. Neste projeto, incluir um novo arquivo HTML chamado **login.html** (clique com o botão direito do mouse no projeto, e selecione New > HTML file).

Solução:

Incluir o arquivo seguindo o procedimento descrito. Quando o arquivo surgir, incluir o código descrito no item 3.

O formulário login.html deve ser semelhante ao mostrado nesta ilustração:



Para este formulário, escrever o código a seguir:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<form action="Login" method="post">
    <table>
        <tr>
            <td>Usuário:</td>
            <td><input type="text" name="usuario" size="20"/></td>
        </tr>
        <tr>
            <td>Senha:</td>
            <td><input type="password" name="senha" size="20"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Validar"/>
            </td>
        </tr>
    </table>
</form>
</body>
</html>
```

1. Executar a aplicação para visualizar o formulário no browser.

Solução:

[Ao executar a aplicação você verá no browser o formulário, porém sem nenhuma funcionalidade.](#)

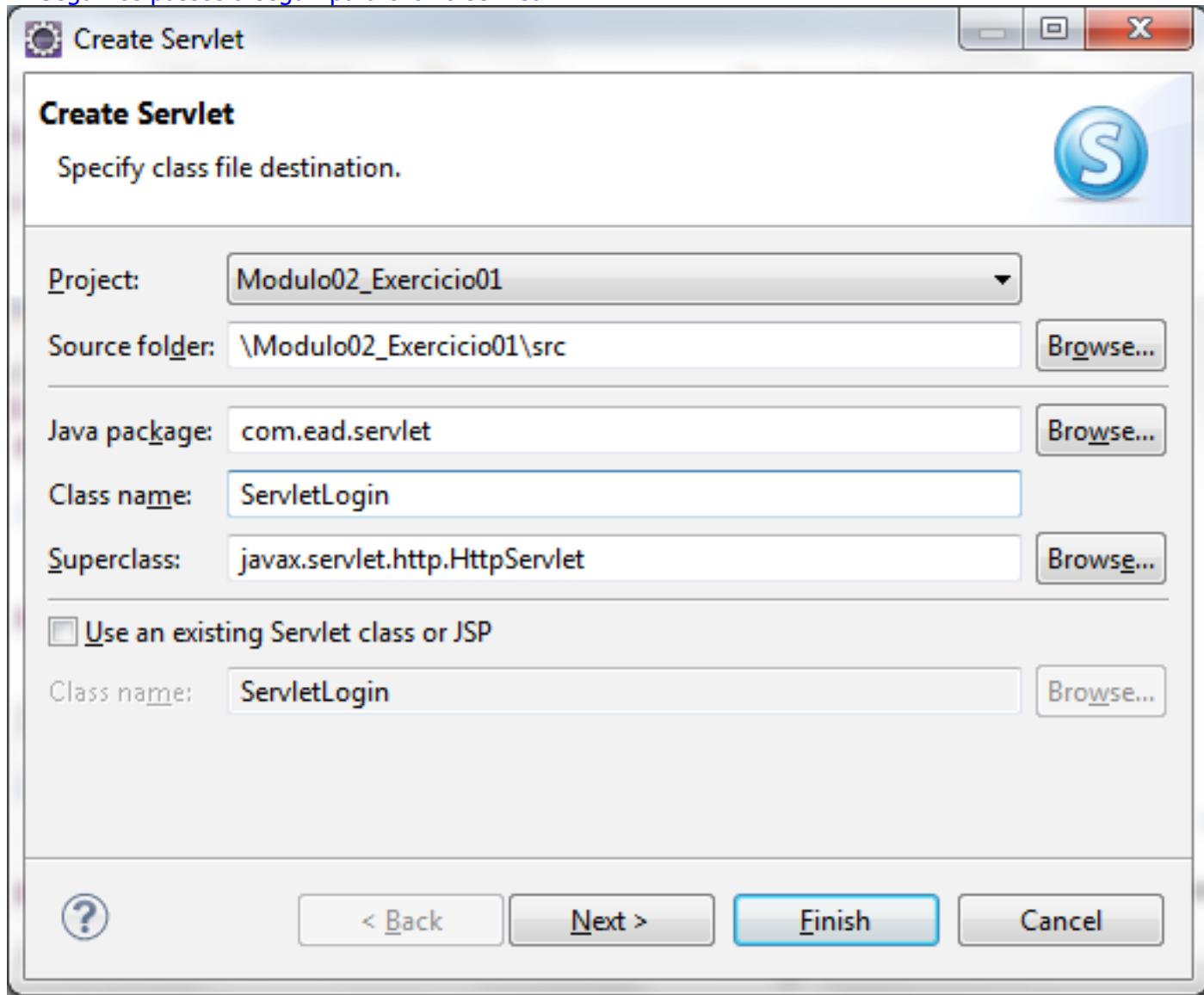
Exercício 2: Definição do servlet para receber os dados do formulário

Neste exercício definiremos o servlet que receberá os dados do formulário desenvolvido no Exercício 1.

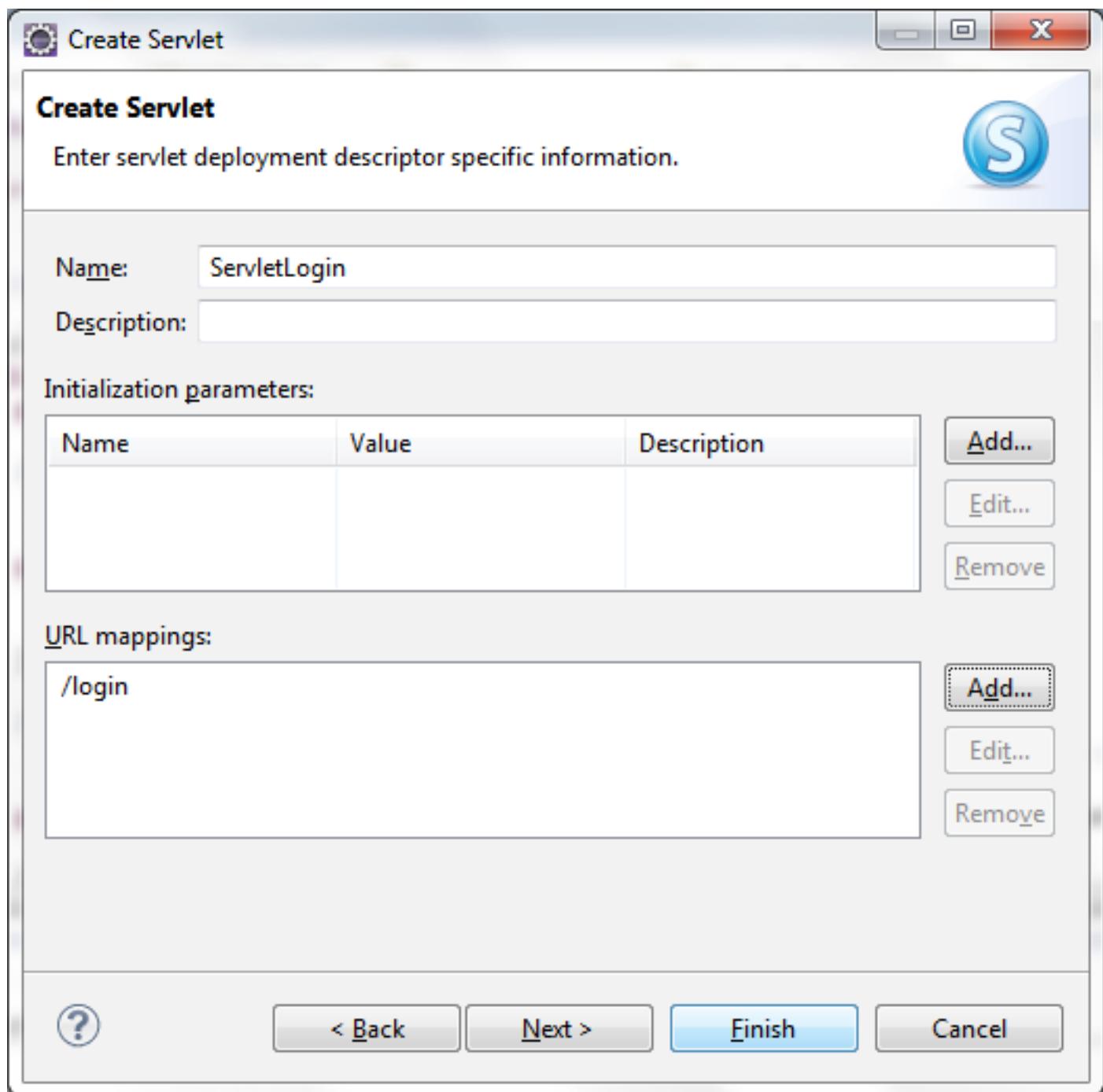
1. Criar um servlet com as configurações:
 - a. Classe: **ServletLogin**
 - b. Nome: **ServletLogin**
 - c. url mapping: **/login**

Solução:

[Seguir os passos a seguir para criar o servlet:](#)



Clicar em Next >. Alterar o mapeamento (url mapping). Não avançar nem finalizar!

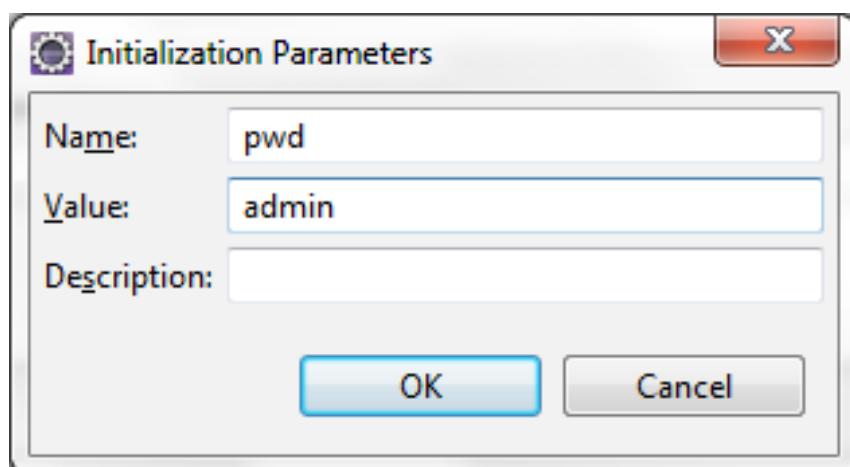
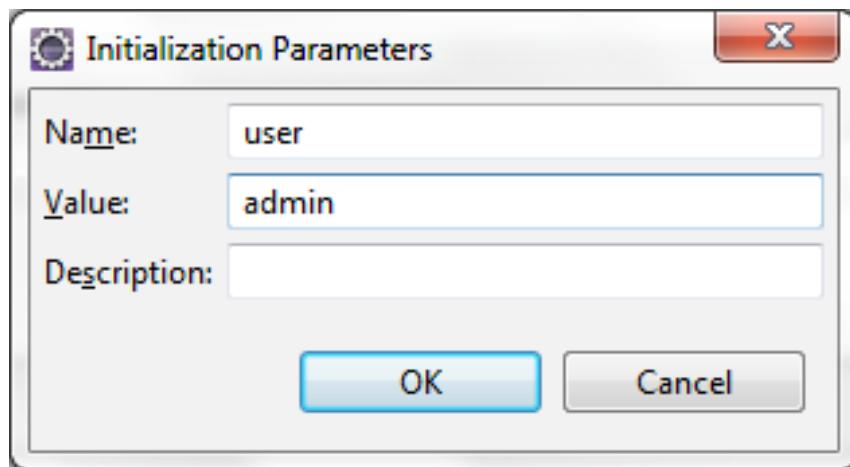


Ao criar o servlet, definir dois parâmetros de inicialização, com os valores:

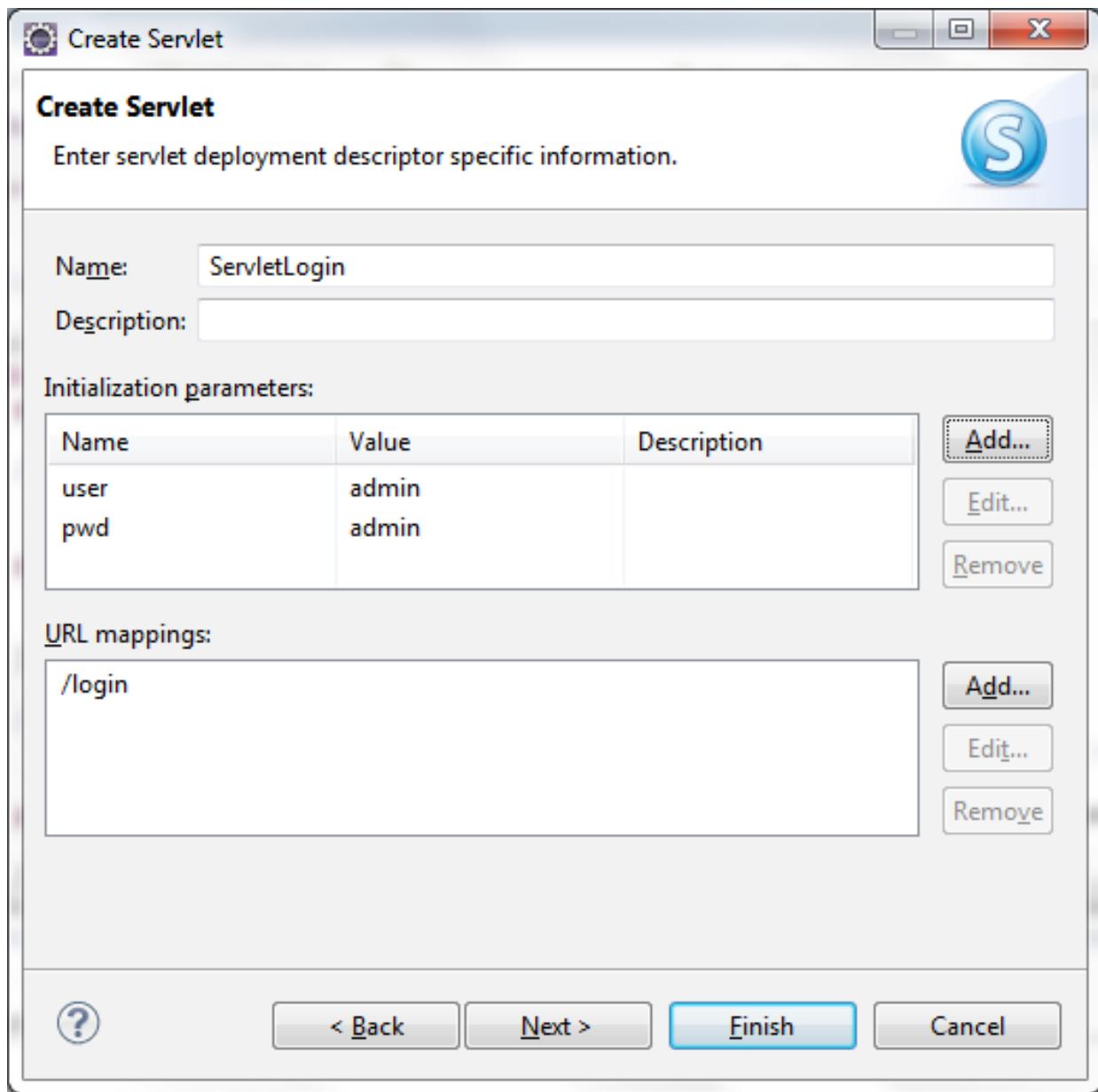
	nome	valor
parâmetro 1	user	admin
parâmetro 2	pwd	admin

Solução:

Na tela anterior, selecionar no quadro superior intitulado Initialization parameters, e clicar no botão Add... Incluir os dois parâmetros indicados no quadro do item 2:



O resultado deve ser o seguinte:



3. No método doGet() escrever a seguinte instrução:

```
response.sendRedirect("login.html");
```

Solução:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.sendRedirect("login.html");
}
```

1. Quando as informações do formulário são enviadas para o servlet, o método doPost() as recebe, pois na definição do formulário temos o atributo **method="post"**. Então, no método doPost()

escrever o código para receber estes dados e compará-las com valores definidos como parâmetro de inicialização do servlet:

```

PrintWriter out = response.getWriter();
response.setContentType("text/html");

//parâmetros do formulário
String usuario = request.getParameter("usuario");
String senha = request.getParameter("senha");

//parâmetros de inicialização do servlet
String user = this.getServletConfig().getInitParameter("user");
String pwd = this.getServletConfig().getInitParameter("pwd");

if(usuario.equals(user) && senha.equals(pwd)){
    out.print("Usuário validado!");
}
else {
    out.print("Usuário ou senha inválidos");
}

```

Solução:

Basta copiar o código anterior no método `doPost()`.

5. Executar a aplicação a partir do formulário, fornecer valores para o usuário e para a senha, e verificar o resultado.

Exercício 3: Armazenamento de dados na sessão

Neste exercício os dados do formulário serão armazenados em sessão se estiverem corretos, ou seja, se forem iguais aos parâmetros de inicialização.

6. Alterar o código do método `doPost()` do servlet `ServletLogin` de forma a armazenar as informações na sessão:

```

PrintWriter out = response.getWriter();
response.setContentType("text/html");

//parâmetros do formulário
String usuario = request.getParameter("usuario");
String senha = request.getParameter("senha");

//parâmetros de inicialização do servlet
String user = this.getServletConfig().getInitParameter("user");

```

```

String pwd = this.getServletConfig().getInitParameter("pwd");

if(usuario.equals(user) && senha.equals(pwd)){
    HttpSession sessao = request.getSession();
    sessao.setAttribute("jusuario", usuario);

    out.print("Usuário validado!");
}
else {
    out.print("Usuário ou senha inválidos");
}

```

Solução:

Alterar o código conforme descrito no código anterior. Observe as linhas em negrito, que representam as alterações.

7. Alterar o código do método doGet() de forma a realizar a seguinte tarefa:

- Se houver um usuário armazenado na sessão, apresentar uma mensagem de boas vindas incluindo o nome do usuário
- Se não houver um usuário validado, apresentar uma mensagem informando que não existe usuário autenticado e um link para retornar para a tela de login:

Solução:

O código a seguir apresenta as alterações que devem ser realizadas.

```

PrintWriter out = response.getWriter();
response.setContentType("text/html");
HttpSession sessao = request.getSession();

String usuario = (String)sessao.getAttribute("jusuario");
if(usuario == null){
    out.print("Nenhum usuário autenticado!<br/>");
    out.print("<a href='login.html'>Voltar para login</a>");
}
else {
    out.print("Seja benvindo(a), " + usuario);
}

```

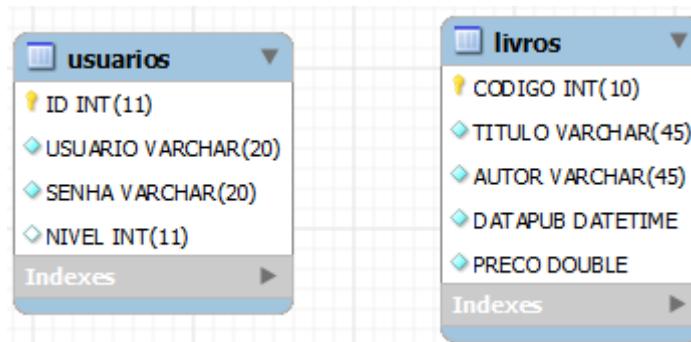
Executar a aplicação a partir do formulário, fornecer valores para o usuário e para a senha, e verificar o resultado.

Exercícios do Módulo 3

Neste módulo o aluno desenvolverá uma aplicação para cadastro e consulta de livros. A aplicação será desenvolvida em etapas, e cada etapa será representada por um exercício. Comecemos então pelo Exercício 1.

Exercício 1: Definição do banco de dados

Desenvolver um banco de dados chamado **servletjsp** contendo as tabelas:



Solução:

Criar este banco de dados no MySQL Workbench. Tomar cuidado para que os campos tenham exatamente estes nomes para combinar com os códigos que surgirão ao longo do exercício.

Exercício 2: Camada de acesso a dados

Vamos criar as classes para o acesso a dados. As classes a serem criadas são:

1. Livros
 2. Usuarios
 3. Dao
 4. DaoLivros
 5. DaoUsuarios
- Classe Livros:

```
package com.ead.model;
```

```
public class Livros {
```

```
    private String codigo, titulo, autor;
    private java.util.Date datapub;
    private double preco;
```

```
//getters e setters
```

- Classe Usuarios:

```
package com.ead.model;
```

```
public class Usuarios {
```

```
    private String usuario,senha;
    private int nivel;
```

```
//getters e setters
```

- Classe Dao:

```
package com.ead.dao;
```

```
import java.sql.*;
```

```
public class Dao {
```

```
    private String url="jdbc:mysql://localhost:3306/servletjsp";
```

```
    protected Connection cn;
```

```
    protected PreparedStatement st;
```

```
    protected ResultSet rs;
```

```
    protected boolean abreConexao() throws Exception{
```

```
        try{
```

```
            Class.forName("com.mysql.jdbc.Driver");
```

```
            cn = DriverManager.getConnection(url,"root","root");
```

```
            return true;
```

```
        }
```

```
        catch(Exception ex){
```

```
            throw ex;
```

```
        }
```

```
}
```

```
    protected void fechaConexao() throws Exception{
```

```
        cn.close();
```

```
}
```

```
}
```

- Classe DaoUsuarios:

```
package com.ead.dao;
```

```
import com.ead.model.Usuarios;
```

```

public class DaoUsuarios extends Dao{

    public boolean validaUsuario(Usuarios usuario) throws Exception{
        boolean b = false;
        try{
            abreConexao();
            st = cn.prepareStatement("SELECT * FROM USUARIOS WHERE USUARIO=? AND SENHA=?");
            st.setString(1, usuario.getUsuario());
            st.setString(2, usuario.getSenha());
            rs = st.executeQuery();
            if(rs.next()){
                usuario.setNivel(rs.getInt("NIVEL"));
                b = true;
            }
        }
        catch(Exception ex){
            throw ex;
        }
        finally{
            fechaConexao();
        }
        return b;
    }
}

```

- Classe DaoLivros

```

package com.ead.dao;

import java.util.ArrayList;
import java.util.List;
import com.ead.model.Livros;

public class DaoLivros extends Dao{

    public String cadastraLivro(Livros livro) throws Exception{
        String msg="";
        try{
            abreConexao();
            st = cn.prepareStatement("INSERT INTO LIVROS (CODIGO,TITULO,AUTOR,DATAPUB,PRECO) VALUES (?,?,?,?,?)");
            st.setString(1, livro.getCodigo());

```

```

        st.setString(2, livro.getTitulo());
        st.setString(3, livro.getAutor());
        st.setDate(4, new java.sql.Date(livro.getDataPub().getTime()));
        st.setDouble(5, livro.getPreco());

        int cont = st.executeUpdate();
        if(cont == 0){
            msg = "Nenhum livro foi inserido!";
        }
        else{
            msg = "Livro inserido com sucesso!";
        }
    }

    catch(Exception ex){
        throw ex;
    }

    finally{
        fechaConexao();
    }

    return msg;
}

public List<Livros> listaLivros() throws Exception{
    List<Livros> lista = new ArrayList<Livros>();
    try{
        abreConexao();
        st = cn.prepareStatement("SELECT * FROM LIVROS");
        rs = st.executeQuery();

        while(rs.next()){
            Livros livro = new Livros();
            livro.setCodigo(rs.getString("CODIGO"));
            livro.setTitulo(rs.getString("TITULO"));
            livro.setAutor(rs.getString("AUTOR"));
            livro.setDataPub(rs.getDate("DATAPUB"));
            livro.setPreco(rs.getDouble("PRECO"));

            lista.add(livro);
        }
    }

    catch(Exception ex){
        throw ex;
    }
}

```

```

        finally{
            fechaConexao();
        }
        return lista;
    }
}

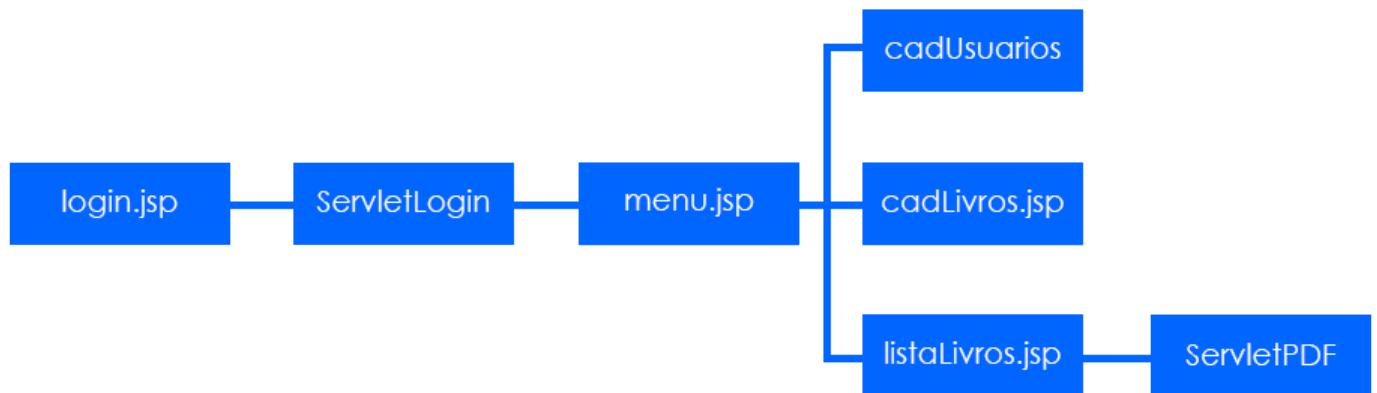
```

Solução:

As classes solicitadas neste exercício já estão definidas. Observar os nomes e os pacotes, e incluí-las no projeto. No comentários que descrevem “getters e setters”, usar o recurso do Eclipse para incluí-los na classe, pois foram omitidos aqui por questões de conveniência.

Exercício 3: Definição da aplicação

A aplicação consistirá de páginas JSP, um Servlet para validação do usuário e um Servlet para exibir o sumário de um livro. O diagrama de navegação para a aplicação é:



Siga os passos descritos a seguir para elaborar a aplicação:

login.jsp: conteúdo HTML para entrada do usuário e senha tradicionais.

Incluir um filtro para verificar se os dados estão preenchidos, antes de enviá-los para o Servlet.

ServletLogin: recebe os dados do formulário em **login.jsp**, autentica o usuário (valida e armazena em sessão) e direciona para **menu.jsp**. Nota: armazenar na sessão o objeto **Usuarios**, pois seus dados serão recuperados nas páginas seguintes.

O método **doGet()** deste formulário deve transferir a requisição para **login.jsp** no sentido de incluir seu conteúdo.

Se houver erro na validação, transferir a requisição para uma página chamada **erro.jsp** (não indicada no esquema). Esta página deverá apresentar a mensagem de erro e ter um link para **login**.

Quando os dados do usuário forem armazenados em sessão, notificar a aplicação através de um Listener (HttpSessionListener e HttpSessionAttributeListener).

menu.jsp: Nesta página deverá ter apenas três links para as páginas seguintes, conforme especificado no modelo.

cadUsuarios.jsp: Uma página que permite realizar o cadastro de novos usuários. Esta página deve ser elaborada com conteúdo HTML e scriptlet.

cadLivros.jsp: Formulário com os dados dos livros, com elementos HTML e componentes JavaBeans.

listaLivros.jsp: Apresenta os dados dos livros em forma tabular. O código deve estar envolvido por um link que, quando selecionado pelo usuário, apresenta um documento PDF contendo um resumo do livro. Esta página deverá ser escrita em JSTL.

ServletPDF: recebe um parâmetro representando o código do livro, e busca por um arquivo PDF com o mesmo nome do código. Se não for encontrado, apresentar um documento padrão.

Código para ler o PDF

Este código supõe que os arquivos PDF estejam na pasta **D:\arquivos**.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    ServletOutputStream stream = null;
    BufferedInputStream buffer = null;

    String arquivo = request.getParameter("cod");

    try {
        stream = response.getOutputStream();
        File pdf = new File("D:/arquivos/" + arquivo + ".pdf");
        if(!pdf.exists()){
            pdf = new File("D:/arquivos/geral.pdf");
        }

        response.setContentType("application/pdf");
        FileInputStream input = new FileInputStream(pdf);
        buffer = new BufferedInputStream(input);
        int bytes = 0;

        while((bytes = buffer.read()) != -1){
            stream.write(bytes);
        }
    } catch (Exception e) {
```

```

        throw new ServletException(e.getMessage());
    } finally {
        if(stream != null){ stream.close(); }
        if(buffer != null){ buffer.close(); }
    }
}

```

Incluir todas as páginas a partir de **menu.jsp** em uma pasta chamada **admin**. Este procedimento facilitará a manipulação do filtro.

Solução:

Comecemos pela página **login.jsp**. Criar um arquivo com este nome e incluir o seguinte conteúdo:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

<form action="Login" method="post">
    <table>
        <tr>
            <td>Usuário:</td>
            <td><input type="text" name="usuario" size="20"/></td>
        </tr>
        <tr>
            <td>Senha:</td>
            <td><input type="password" name="senha" size="20"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Validar"/>
            </td>
        </tr>
    </table>
</form>

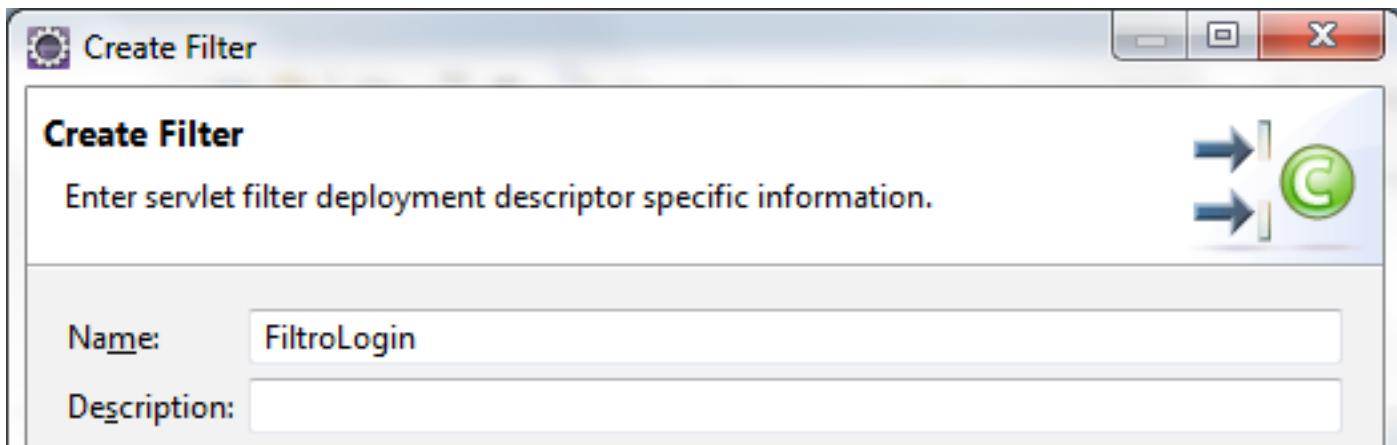
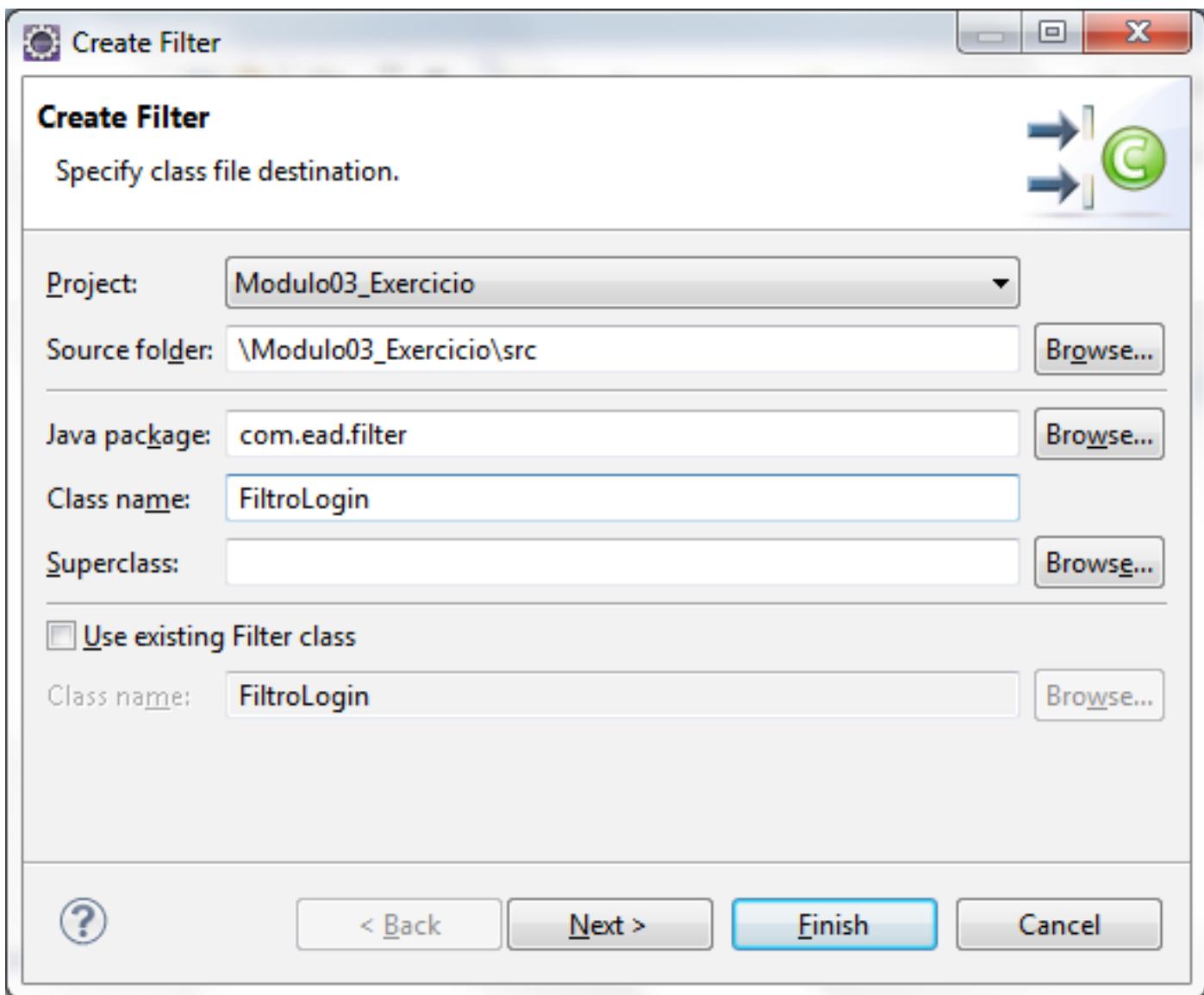
</body>

```

```
</html>
```

Na observação dada em azul, é solicitado que seja criado um filtro para cerofocar se os dados estão preenchidos antes de enviá-los para o servlet. Torna-se necessário, então, criarmos um filtro no projeto, de modo semelhante a incluir um servlet: botão direito do mouse, selecionar Filter, e fornecer as informações como nome do filtro e pacote, além de informar o que o filtro irá verificar. Neste caso, para que haja acesso às páginas a partir de menu.jsp, é necessário que o usuário esteja autenticado. Vamos criar o filtro:

- Inserir um novo filtro:



Ao aceitar estas opções, teremos a classe a seguir (aqui eu já removi os comentários desnecessários, gerados pelo Eclipse), já com o código para verificar a existência de usuário na sessão.

```
package com.ead.filter;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@WebFilter("/admin/*")
public class FiltroLogin implements Filter {

    public FiltroLogin() {
        // TODO Auto-generated constructor stub
    }

    public void destroy() {
        // TODO Auto-generated method stub
    }

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest)request;
        HttpSession session = req.getSession();
        if(session.getAttribute("jUser") == null){
            ((HttpServletResponse)response).sendRedirect("/Modulo03_Exercicio/login.jsp");
        }
        else {
            chain.doFilter(request, response);
        }
    }

    public void init(FilterConfig fConfig) throws ServletException {
        // TODO Auto-generated method stub
    }
}
```

```

    }
}

}

```

Agora, vamos criar o servlet ServletLogin. Os passos para criar o servlet já foram descritos em exercícios anteriores, e por isso não serão repetidos aqui. O método doPost() do servlet resultante é:

```

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    Usuarios user = new Usuarios();
    DaoUsuarios dao = new DaoUsuarios();
    try {
        user.setUsuario(request.getParameter("usuario"));
        user.setSenha(request.getParameter("senha"));

        dao.setUsuario(user);
        if(dao.validaUsuario()){
            HttpSession session = request.getSession();
            session.setAttribute("jUser", user);
            response.sendRedirect("paginas/menu.jsp");
        }
        else{
            out.print("Usuário ou senha inválidos!<br/>");
            out.print("<a href='login.jsp'>Voltar para Login</a>");
        }
    } catch (Exception e) {
        out.print(e.getMessage());
    }
}

```

e o método doGet():

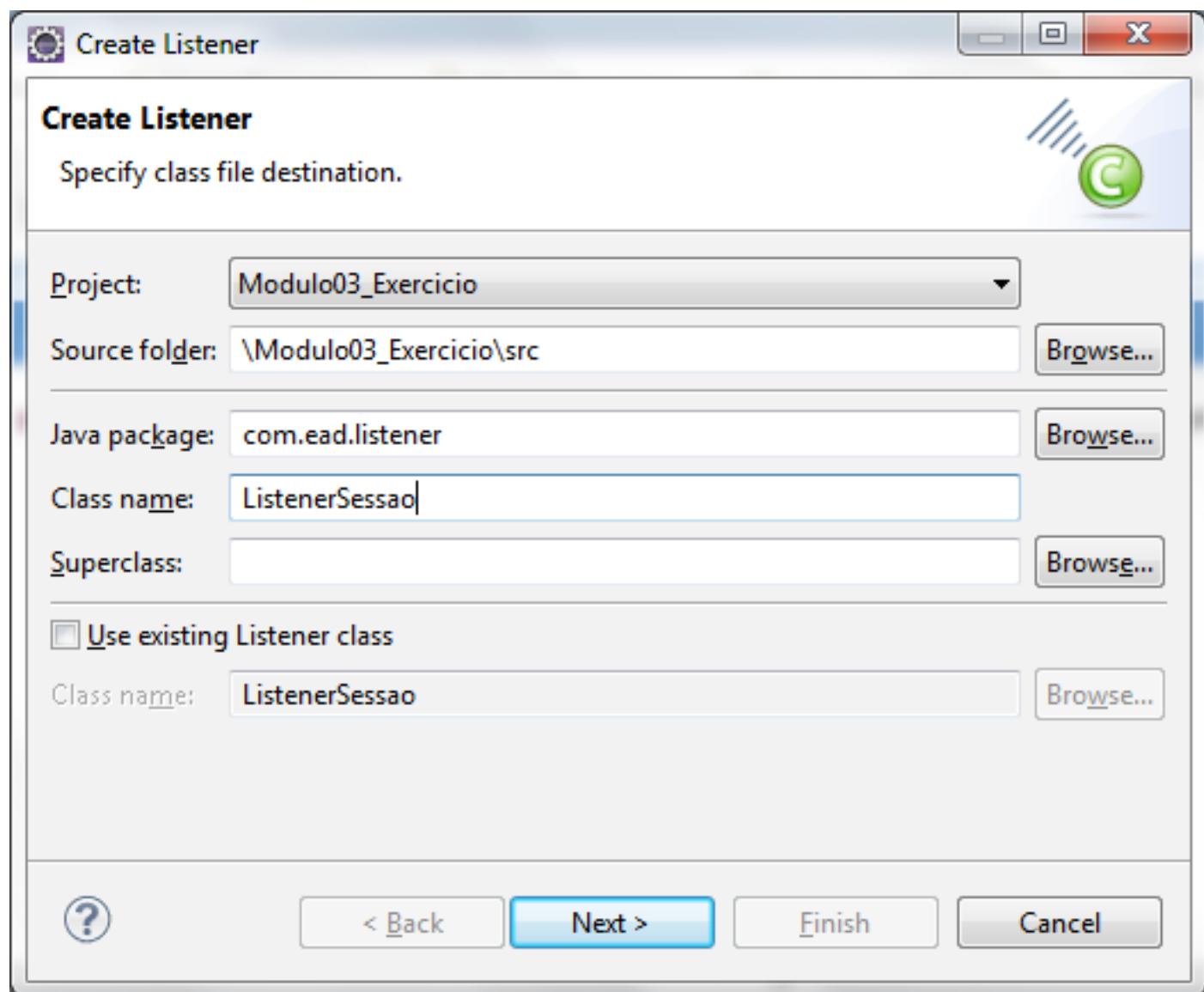
```

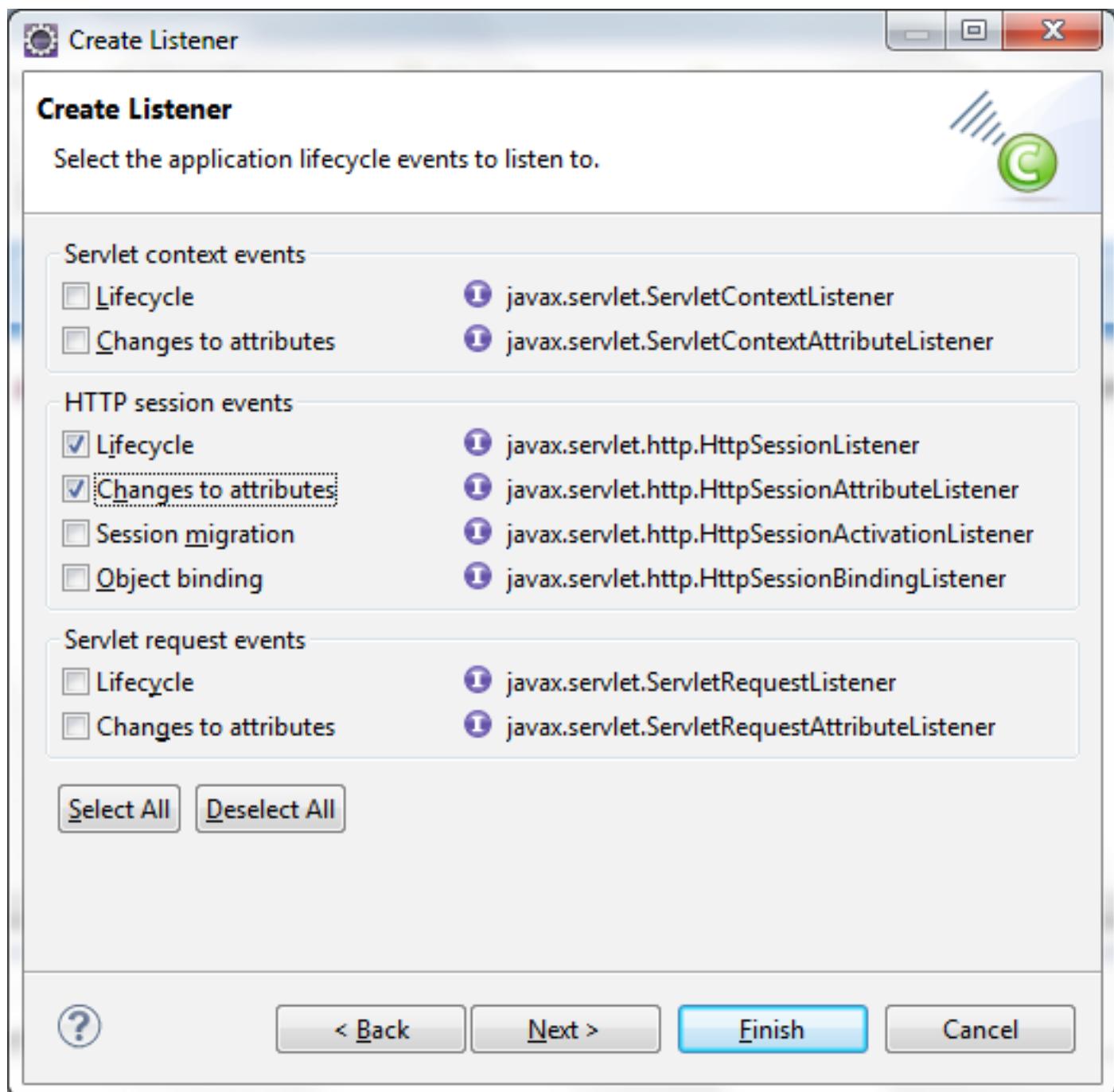
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.sendRedirect("login.jsp");
}

```

A seguir mostraremos como criar o Listener para que haja notificação conforme solicitado (HttpSessionListener e HttpSessionAttributeListener). Para isso, clicamos com o botão direito do mouse sobre

o projeto e selecionamos a opção Listener. Seguimos as etapas mostradas a seguir para criar o Listener solicitado:





O eclipse criará a classe para este listener. O código a ser incluído já está apresentado na classe:

```
package com.ead.listener;

import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

@WebListener
public class ListenerSessao implements HttpSessionListener, HttpSessionAttributeListener {

    public ListenerSessao() {
        // TODO Auto-generated constructor stub
    }

    public void attributeRemoved(HttpSessionBindingEvent arg0) {
        System.out.println("Atributo removido");
    }

    public void attributeAdded(HttpSessionBindingEvent arg0) {
        System.out.println("Atributo adicionado à sessão");
    }

    public void attributeReplaced(HttpSessionBindingEvent arg0) {
        System.out.println("Atributo substituído");
    }

    public void sessionCreated(HttpSessionEvent arg0) {
        System.out.println("Sessão criada!");
    }

    public void sessionDestroyed(HttpSessionEvent arg0) {
        System.out.println("Sessão removida");
    }
}
```

A partir deste ponto, as páginas deverão ser incluídas em uma pasta chamada **admin**. Para tanto, criar esta pasta no projeto.

Conteúdo da pasta menu.jsp:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <h1>Menu de Opções</h1>
    <ul>
        <li><a href="cadUsuarios.jsp">Cadastrar Usuários</a></li>
        <li><a href="cadLivros.jsp">Cadastrar Livros</a></li>
        <li><a href="ListaLivros.jsp">Listar Livros</a></li>
        <li><a href="ListaLivrosJSTL.jsp">Listar Livros - JSTL</a></li>
    </ul>
</body>
</html>
```

conteúdo de cadUsuarios.jsp:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Cadastro de Usuários</title>
</head>
<body>
    <form action="cadUsuariosJSTL.jsp" method="post">
        <table>
            <tr>
                <td>Usuário:</td>
                <td><input type="text" name="usuario" size="20"></td>
            </tr>
            <tr>
                <td>Senha:</td>
                <td><input type="password" name="senha" size="20"></td>
            </tr>
            <tr>
```



```

<td>Nível:</td>
<td>
    <select name="nivel">
        <option value="1">Administrador</option>
        <option value="2">Cliente</option>
    </select>
</td>
</tr>
<tr>
    <td>
        <input type="submit" value="Enviar">
    </td>
</tr>
</table>
</form>
</body>
</html>

```

Observe que os dados deste formulário são enviados para uma página chamada cadUsuariosJSTL.jsp, que realizará a persistência. Vamos criar este JSP:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>

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

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>

<c:catch var="erro">

    <jsp:useBean id="usuario" class="com.ead.model.U
suarios" />

    <c:set var="user" value="${param.usuario}" />
    <c:set var="pwd" value="${param.senha}" />
    <c:set var="nivel" value="${param.nivel}" />

    <jsp:setProperty property="usuario" name="usuario" value="${user}" />

```

```

<jsp:setProperty property="senha" name="usuario" value="${pwd}"/>
<jsp:setProperty property="nivel" name="usuario" value="${nivel}"/>

<jsp:getProperty property="cadastro" name="usuario"/>
</c:catch>

<c:out value="${erro}" /><br/>
<a href="menu.jsp">Retornar ao menu</a>
</body>
</html>

```

Conteúdo de cadLivros.jsp:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ include file="valida.jsp" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<form action="cadLivrosJB.jsp" method="post">
    <table>
        <tr>
            <td>Código:</td>
            <td><input type="text" name="codigo" size="20"> </td>
        </tr>
        <tr>
            <td>Título:</td>
            <td><input type="text" name="titulo" size="20"> </td>
        </tr>
        <tr>
            <td>Autor:</td>
            <td><input type="text" name="autor" size="20"> </td>
        </tr>
        <tr>
            <td>Data Publicação:</td>
            <td><input type="text" name="data" size="20"> </td>
        </tr>
        <tr>
            <td>Preço:</td>

```

```

        <td><input type="text" name="preco" size="20"> </td>
    </tr>
    <tr>
        <td colspan="2">
            <input type="submit" value="Enviar">
        </td>
    </tr>
</table>
</form>
</body>
</html>
```

Este formulário envia os dados para cadLivrosJB.jsp. Vamos criar este arquivo, então:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ page import="br.com.fiap.javabeans.Livros" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="Livro" class="com.ead.model.Livros" />
<jsp:setProperty property="codigo" name="Livro" value="${param.codigo}" />
<jsp:setProperty property="titulo" name="Livro" value="${param.titulo}" />
<jsp:setProperty property="autor" name="Livro" value="${param.autor}" />
<jsp:setProperty property="data" name="Livro" value="${param.data}" />
<jsp:setProperty property="preco" name="Livro" value="${param.preco}" />

<jsp:getProperty property="cadastro" name="Livro"/>

</body>
</html>
```

Conteúdo de listaLivros.jsp:

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

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

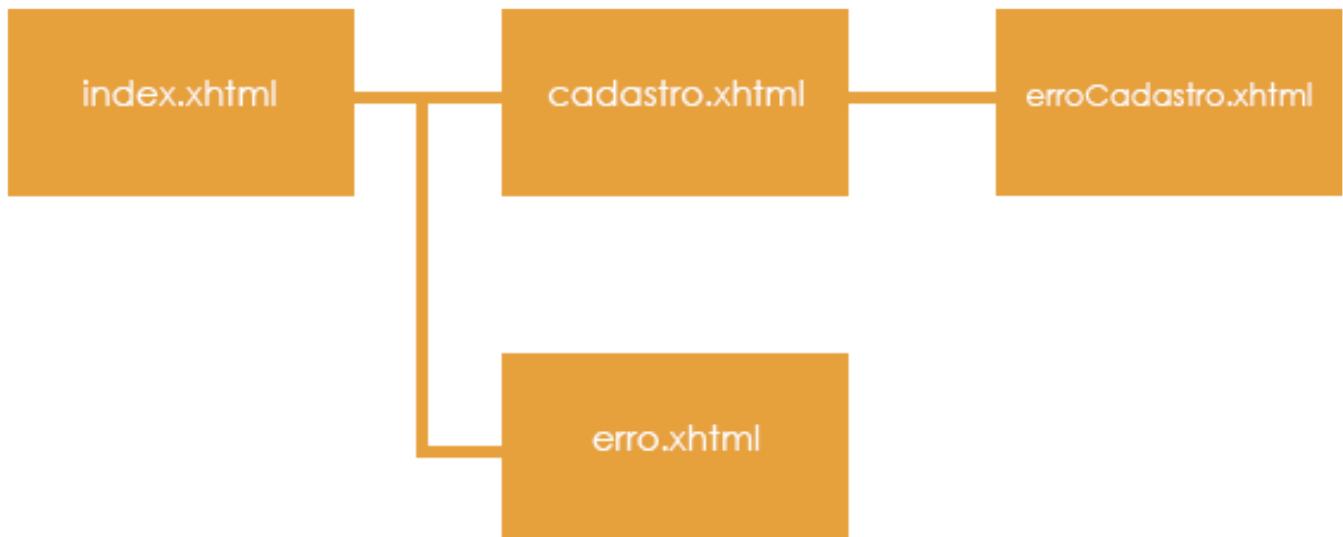
```
<%@ include file="valida.jsp" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/
html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Lista de Livros Cadastrados</title>
</head>
<body>
    <h1>Lista de Livros Cadastrados (JSTL)</h1>
    <jsp:useBean id="Livros" class="br.com.fiap.javabeans.Livros" />

    <table border="1">
        <tr>
            <td>CÓDIGO</td>
            <td>TITULO</td>
        </tr>

        <c:forEach var="item" items="${livros.listaLivros}">
            <tr>
                <td><a href='pdf?codigo=${item.codigo}'>${item.codigo}</a></
td>
                <td>${item.titulo}</td>
            </tr>
        </c:forEach>
    </table>
</body>
</html>
```

Esta página contém um link para um servlet chamado mapeado como pdf, conforme indicado no código.

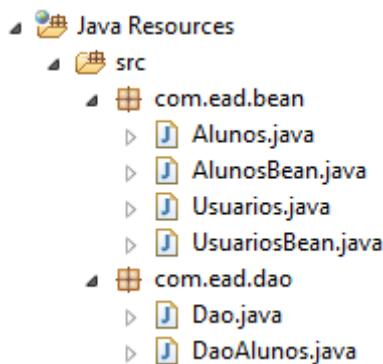
Exercícios do Módulo 4



Este exercício trata da validação de usuários e cadastro de alunos, usando **JavaServer Faces 2.0**. O diagrama de navegação é dado abaixo:

Exercício 1: Para desenvolver este exercício, siga estes passos:

1. Defina um novo projeto web chamado **Modulo04_Exercicio**.
2. Neste projeto será necessário incluir a API do JSF. O arquivo deve ser colocado na pasta lib, abaixo de WEB-INF no projeto. Após incluir o arquivo, clicar com o direito do mouse e selecionar a opção: **"Add to Build Path"**. O arquivo se chama javax.faces.X.jar, onde X é a versão. Ele pode ser obtido no link: <http://mvnrepository.com/artifact/javax.faces/javax.faces-api/2.1>
3. Criar as classes para o projeto. As classes são mostradas na estrutura da seguinte figura:



Classe Alunos:

```
package com.ead.bean;
```

```
import java.util.Date;
```

```

public class Alunos {
    private String curso, nome, email;
    private int rm;
    private Date dataNascimento;

        //getters e setters
}

```

Classe AlunosBean – Esta classe será usada como bean nas páginas.xhtml:

```

package com.ead.bean;

import java.util.ArrayList;
import java.util.List;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.model.SelectItem;

import com.ead.dao.DaoAlunos;

@ManagedBean(name="aluno")
@RequestScoped
public class AlunosBean {
    private String mensagem;

    public String getMensagem() {
        return mensagem;
    }

    public List<SelectItem> getCursos(){
        List<SelectItem> lista = new ArrayList<SelectItem>();
        lista.add(new SelectItem("SCJ","Soluções Corporativas Java"));
        lista.add(new SelectItem("AOJ","Analise Orientada a Objetos Java"));
        lista.add(new SelectItem("MIT","Master in Information Tecnology"));
        return lista;
    }

    public List<Alunos> getListaAlunos() throws Exception {
        return new DaoAlunos().listarAlunos();
    }

    public String cadastrarAluno(Alunos aluno) throws Exception{

```

```

DaoAlunos dao = new DaoAlunos();

if(dao.cadastrarAluno(aluno)){
    mensagem = "RM " + aluno.getRm() + " inserido com sucesso!";
    return "";
}
else {
    mensagem = "Erro";
    return "";
}
}

public String listarTodosAlunos(){
    return "";
}
}

```

Classe Usuarios

```

package com.ead.bean;

public class Usuarios {
    private String nome, senha;

    //getters e setters
}

```

Classe UsuariosBean – classe usada com bean nas páginas xhtml. O método validarUsuario() deve ser completado pelo aluno.

```

package com.ead.bean;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name="usuariosBean")
@SessionScoped
public class UsuariosBean {
    private Usuarios usuario = new Usuarios();

    public Usuarios getUsuario() {
        return usuario;
    }
}

```

```

}

public void setUsuario(Usuarios usuario) {
    this.usuario = usuario;
}

public String validarUsuario(){
    if( //realizar aqui a validação do usuário ){

        return "/cadastro";
    }
    else{
        return "/index";
    }
}
}

```

Classe Dao: Vamos deixar para o aluno desenvolver esta classe, tomando como base a classe Dao do Módulo 4.

Classe DaoAlunos: Também deixaremos par ao aluno desenvolver esta classe. Observe que a classe AlunosBean faz referência a ela.

Exercício 2

Desenvolver as páginas apresentadas no diagrama de navegação no início do exercício deste módulo. Para tanto, incluir um novo arquivo HTML no eclipse, e mudar a extensão do arquivo para .xhtml. A página index.xhtml será dada como modelo:

```

<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>Validação do Usuario</title>
</h:head>

<f:view>
    <h:form>
        <table>
            <tr>

```

```

        <td>Usuário:</td>
        <td><h:inputText value="#{usuariosBean.usuario.nome}"/></td>
    </tr>
    <tr>
        <td>Senha:</td>
        <td><h:inputSecret value="#{usuariosBean.usuario.senha}"/></td>
    </tr>
    <tr>
        <td colspan="2"><h:commandButton value="Enviar"
            action="#{usuariosBean.validarUsuario}"/></td>
    </tr>
</table>
</h:form>
</f:view>
</html>
```

Solução:

Assim como nos exercícios anteriores, vamos destacar os itens que devem ser implementados. Definiremos as classes Dao e DaoAlunos:

Classe Dao:

```

package com.ead.dao;
import java.sql.*;

public class Dao {
    private String url="jdbc:mysql://localhost:3306/aula_jsf";
    protected Connection cn;
    protected PreparedStatement st;
    protected ResultSet rs;

    protected boolean abreConexao() throws Exception{
        try{
            Class.forName("com.mysql.jdbc.Driver");
            cn = DriverManager.getConnection(url,"root","root");
            return true;
        }
        catch(Exception ex){
            throw ex;
        }
    }

    protected void fechaConexao() throws Exception{
```

```

        cn.close();
    }
}

```

Classe DaoAlunos:

```

package com.ead.dao;

import java.util.ArrayList;
import java.util.List;

import com.ead.bean.Alunos;

public class DaoAlunos extends Dao{

    public boolean cadastrarAluno(Alunos aluno) throws Exception{
        boolean b = false;

        if(abreConexao()){
            try {
                String sql = "INSERT INTO ALUNOS (RM,NOME,EMAIL,DATANASC,CURSO)
VALUES (?,?,?,?,?)";
                st = cn.prepareStatement(sql);
                st.setInt(1, aluno.getRm());
                st.setString(2, aluno.getNome());
                st.setString(3, aluno.getEmail());
                st.setDate(4, new java.sql.Date(aluno.getDataNascimento().
getTime()));
                st.setString(5, aluno.getCurso());
                st.executeUpdate();
                b = true;
            } catch (Exception ex) {
                throw ex;
            }
        finally{
            fechaConexao();
        }
    }
    return b;
}

public List<Alunos> listarAlunos() throws Exception{
    List<Alunos> lista = new ArrayList<Alunos>();
    if(abreConexao()){

```

```

try {
    st = cn.prepareStatement("SELECT * FROM ALUNOS");
    rs = st.executeQuery();

    while(rs.next()){
        Alunos aluno = new Alunos();
        aluno.setRm(rs.getInt("RM"));
        aluno.setNome(rs.getString("NOME"));
        aluno.setEmail(rs.getString("EMAIL"));
        aluno.setDataNascimento(rs.getDate("DATANASC"));
        aluno.setCurso(rs.getString("CURSO"));
        lista.add(aluno);
    }
} catch (Exception e) {
    throw e;
}
finally{
    fechaConexao();
}
}

return lista;
}
}
}

```

Conteúdo da página cadastro.xhtml:

```

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

<h:head>
    <title>Cadastro de Alunos</title>
</h:head>

<f:view>
    <h:body>
        <h:form>
            <h:panel header="Cadastro de Alunos" id="paineL">
                <br />
                <table id="tab">

```

```

<tr>
    <td>Rm:</td>
    <td><h:inputText value="#{aluno.aluno.rm}" /></td>
</tr>
<tr>
    <td>Nome:</td>
    <td><h:inputText value="#{aluno.aluno.nome}" /></td>
</tr>
<tr>
    <td>Email:</td>
    <td><h:inputText value="#{aluno.aluno.email}" /></td>
</tr>
<tr>
    <td>Data Nascimento:</td>
    <td><h:inputText value="#{aluno.aluno.dataNascimento}">
        <f:convertDateTime pattern="dd/MM/yyyy">
    </h:inputText></td>
</tr>
<tr>
    <td>Curso:</td>
    <td><h:selectOneMenu value="#{aluno.aluno.curso}">
        <f:selectItems value="#{aluno.cursos}">
    </h:selectOneMenu></td>
</tr>
</table>

<h:commandButton value="Cadastrar" action="#{aluno.cadastrarAluno}" />

<h:commandButton value="Listar Alunos" action="#{aluno.ListarTodosAlunos}" />

<h:dataTable id="tabela" var="us" value="#{aluno.ListaAlunos}">
    <h:column>
        <f:facet name="header">
            <h:outputText value="RM" />
        </f:facet>
            <h:outputText value="#{us.rm}" />
    </h:column>

```

```

<h:column>
    <f:facet name="header">
        <h:outputText value="NOME" />
    </f:facet>
    <h:outputText value="#{us.nome}" />
</h:column>

<h:column>
    <f:facet name="header">
        <h:outputText value="EMAIL" />
    </f:facet>
    <h:outputText value="#{us.email}" />
</h:column>

<h:column>
    <f:facet name="header">
        <h:outputText value="DATA NASC." />
    </f:facet>
    <h:outputText value="#{us.dataNascimento}">
        <f:convertDateTime pattern="dd/MM/yyyy" />
    </h:outputText>
</h:column>

<h:column>
    <f:facet name="header">
        <h:outputText value="CURSO" />
    </f:facet>
    <h:outputText value="#{us.curso}" />
</h:column>
</h:dataTable>

</h:panel>
<h:commandLink value="Voltar para Login" action="/index" />
</h:form>
</h:body>
</f:view>

</html>

```

Conteúdo de erro.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>Validação do Usuário</title>
</h:head>

<f:view>
    <h:form>
        <h:outputText value="Ocorreu um erro na validação!" />
    </h:form>
</f:view>
</html>
```

Conteúdo de erroCadastro.xhtml

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

<f:view>
    <h:form>
        <h:outputText value="Ocorreu um erro no cadastro!" />
    </h:form>
</f:view>
</html>
```

Referências Bibliográficas

- DELISLE, P. **Java Server Pages Standard Tag Library**: (Version 1.2). Sun Microsystems, 2006.
- GONÇALVES, A. **Beginning Java EE 6 Platform with GlassFish 3**. Disponível em: <http://digitus.itk.ppke.hu/~farba/BEGINNING_JAVA_EE_6_with_GlassFish_3.pdf>. Acesso em: 05 set. 2014.
- GONÇALVES, E. **Ajax com JSF 2.0**. 2011. Disponível em: <<http://www.edsongoncalves.com.br/tag/jsf-2-0/>>. Acesso em: 25 jul. 2014.
- IBM KNOWLEDGE CENTER. **JAXB**. Disponível em: <<http://pic.dhe.ibm.com/infocenter/radhelp/v9/index.jsp?topic=%2Fcom.ibm.webservice.doc%2Ftopics%2Fcore%2Fcjaxb.html>>. Acesso em: 12 set. 2014.
- IBM . **Struts framework and model-view-controller design pattern**. Disponível em: <<http://publib.boulder.ibm.com/infocenter/rtnlhelp/v6r0m0/index.jsp?topic=%2Fcom.ibm.etools.struts.doc%2Ftopics%2Fcstrdoc001.html>>. Acesso em: 28 jun. 2014.
- INTACCT. **Responses from the gateway**. Disponível em: <<http://developer.intacct.com/wiki/responses-gateway>>. Acesso em: 28 jun. 2014.
- JAVA FORUM. **JavaServer Faces**: Basics. 2008. Disponível em: <<http://www.java-forums.org/java-tutorial/9637-javaserver-faces-basics.html>>. Acesso em: 28 jun. 2014.
- JUNIOR, E. P. **Web services**: uma solução para aplicações distribuídas na internet. Disponível em: <<http://www.apostilando.com/download.php?cod=415&categoria=Internet>>. Acesso em: 14 Ago. 2014.
- KALIN, M. **Java Web services**: Implementando. Porto Alegre: Alta Books Editora, 2010.
- MULLER, N. **Internet, intranet e extranet o que são, e quais as diferenças?**. Disponível em: <http://www.oficinadanet.com.br/artigo/1276/internet_intranet_e_extranet_o_que_sao_e_quais_as_diferencias>. Acesso em: 07 Ago. 2014.
- NETBEANS. **Binding WSDL to Java with JAXB**. Disponível em: <<https://netbeans.org/kb/74/websvc/jaxb.html>>. Acesso em: 28 jun. 2014.
- ORACLE. **The Java EE 6 Tutorial: Web Applications**. Disponível em: <<http://docs.oracle.com/javaee/6/tutorial/doc/geysj.html>>. Acesso em: 28 jun. 2014.
- REDDY, L. **Creating SOAP Web services with JAX-WS**. Disponível em: <<http://www.developer.com/java/creating-soap-web-services-with-jax-ws.html>>. Acesso em: 28 jun. 2014.

SIGNIFICADOS.COM.BR. **Significado de HTTP:** O que é HTTP. Disponível em: <<http://www.significados.com.br/http/>>. Acesso em: 28 ago. 2014.