

INTEGRATION

FRAMEWORKS

EM JAVA

ALEXANDRE CARLOS DE JESUS



04

LISTA DE FIGURAS

Figura 1 – Código macarrônico	7
Figura 2 – Recuperação do atributo usuário por meio de EL na página JSP	9
Figura 3 – Resultado na página JSP com EL.....	10
Figura 4 – Resultado da operação da EL realizando validação	11
Figura 5 – Resultado da página exemplo1.jsp que envia dados para outra página JSP.....	15
Figura 6 – Resultado da página exemplo2.jsp com os dados que foram enviados da página exemplo1.jsp	16
Figura 7 – Resultado da página exemplo1.jsp com <i>Scriptlets</i>	20
Figura 8 – Resultado da página exemplo1.jsp com <i>TagLibs</i>	22
Figura 9 – Estrutura de pastas Eclipse.....	23
Figura 10 – <i>Jars</i> das <i>TagLibs</i> na pasta lib.....	24
Figura 11 – Resultado da página exemplo1.jsp com a TagLib <i>forEach</i>	28
Figura 12 – Resultado da página de formatação de data.....	34
Figura 13 – Resultado da página de formatação de números.....	37
Figura 14 – Criando um projeto web – Parte 1	38
Figura 15 – Criando um projeto web – Parte 2.....	39
Figura 16 – Criando um projeto web – Parte 3.....	40
Figura 17 – Criando uma página JSP – Parte 1	41
Figura 18 – Criando uma página JSP – Parte 2.....	42
Figura 19 – Execução da página JSP de cadastro de produto.....	44
Figura 20 – Criação da classe <i>Servlet</i> – Parte 1	45
Figura 21 – Criação da classe <i>Servlet</i> – Parte 2	46
Figura 22 – Criação da classe <i>Java Bean</i> Produto – Parte 1	47
Figura 23 – Criação da classe <i>Java Bean</i> Produto – Parte 2.....	48
Figura 24 – Configuração das bibliotecas JSTL.....	52
Figura 25 – Teste do cadastro de produto	54
Figura 26 – Teste da mensagem de sucesso.....	55
Figura 27 – Teste da listagem de produto e menu	58

LISTA DE QUADROS

Quadro 1 – Operadores EL	12
Quadro 2 – Objetos implícitos para EL.....	13
Quadro 3 – Tabela de <i>tags</i> da biblioteca <i>core</i>	26
Quadro 4 – Atributos da <i>tag import</i>	31
Quadro 5 – <i>Tags</i> de <i>formatting</i>	32
Quadro 6 – Quadro com opções para o <i>pattern</i> de data	35

EXEMPLO

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Exemplo de código-fonte para página de <i>script</i> JSP com EL	8
Código-fonte 2 – Exemplo de <i>Servlet</i> realizando <i>request</i> para JSP	8
Código-fonte 3 – JSP com EL recuperando atributo no <i>request</i>	9
Código-fonte 4 – JSP com EL realizando operações	10
Código-fonte 5 – EL executando uma operação de validação	11
Código-fonte 6 – Página exemplo1.jsp com exemplo de envio de dados para a página JSP	14
Código-fonte 7 – Página exemplo2.jsp recebendo os dados da página exemplo1.jsp	16
Código-fonte 8 – Atributo do tipo <i>JavaBean(Object)</i> , que foi criado no <i>request</i>	17
Código-fonte 9 – EL recuperando atributo enviado no <i>request</i> “cli” em página JSP .	18
Código-fonte 10 – Exemplo de <i>bean</i> padrão	19
Código-fonte 11 – EL acessando os atributos nome e idade do objeto “cli” em página JSP	19
Código-fonte 12 – Página JSP exemplo1.jsp com <i>Scriptlets</i>	20
Código-fonte 13 – Página JSP exemplo1.jsp com <i>TagLibs</i>	21
Código-fonte 14 – Diretiva <i>taglib</i> com o <i>import</i> para a biblioteca core	25
Código-fonte 15 – <i>Servlet</i> gerando um <i>ArrayList</i> no <i>request</i> e enviando para a página JSP	27
Código-fonte 16 – Página com o código da <i>TagLib</i> para iteração da lista	28
Código-fonte 17 – Página com o código da <i>TagLib</i> if	29
Código-fonte 18 – Página com o código da <i>TagLib</i> choose	30
Código-fonte 19 – Página com o código da <i>TagLib</i> out.....	30
Código-fonte 20 – Página com o código da <i>TagLib</i> url.....	30
Código-fonte 21 – Página com o resultado da <i>TagLib</i> url	31
Código-fonte 22 – Página com o código da <i>TagLib</i> import.....	31
Código-fonte 23 – Página com o código da diretiva <i>taglib</i> e o <i>prefix</i> para a biblioteca <i>formatting</i>	32
Código-fonte 24 – Página com exemplos da <i>tag</i> <i>formatDate</i>	33
Código-fonte 25 – Página com o código da <i>tag</i> <i>formatNumber</i>	36
Código-fonte 26 – Página de cadastro de produto	43
Código-fonte 27 – <i>Java Bean</i> Produto	49
Código-fonte 28 – Implementação da <i>Servlet</i>	51
Código-fonte 29 – Exibindo a mensagem de sucesso	53
Código-fonte 30 – Ajuste na <i>Servlet</i> para a funcionalidade de listagem	55
Código-fonte 31 – Página JSP de listagem de produtos	56
Código-fonte 32 – Página JSP de listagem de produtos	58

SUMÁRIO

1 FRAMEWORKS EM JAVA.....	6
1.1 Recuperando dados com EL.....	7
1.2 Objetos implícitos.....	12
1.3 EL e <i>JavaBeans</i>	17
1.4 JSTL.....	19
1.5 Utilização.....	22
1.6 TagLibs – Core.....	25
1.6.1 <c:forEach>.....	27
1.6.2 <c:if>.....	29
1.6.3 <c:choose> <c:when> <c:otherwise>.....	29
1.6.4 <c:out>.....	30
1.6.5 <c:url>.....	30
1.6.6 <c:import>.....	31
1.7 TagLibs – Formatting.....	31
1.7.1 <fmt:formatDate>.....	32
1.7.2 <fmt:formatNumber>.....	35
2 PRÁTICA!.....	38
REFERÊNCIAS.....	59
GLOSSÁRIO.....	60

1 FRAMEWORKS EM JAVA

Depois de conhecer as páginas JSP, que dão mais agilidade para desenvolver as páginas web dinâmicas, chegou o momento de utilizar a *Expression Language* e as bibliotecas de *tags*, que tornarão a implementação das nossas aplicações muito mais simples, limpas e produtivas.

Expression Language (EL), como o próprio nome diz, é uma linguagem de expressão utilizada em páginas JSP. A EL começou como parte do projeto JSTL (*Java Standard Tag Libraries*), e foi originalmente chamada SPEL (*Spring Expression Language*), em seguida recebeu o nome *Expression Language* (EL). Era uma linguagem de *scripts* que permitia o acesso a componentes Java (*JavaBeans*) por meio de JSP.

Ao longo dos anos, a EL evoluiu para incluir funcionalidades mais avançadas, surgindo, assim, a especificação JSP 2.0. A criação de *scripts* foi pensada para os designers web, que possuem pouco ou praticamente nenhum conhecimento da linguagem Java. Essa linguagem de *script* fez do JSP uma linguagem de *script* no verdadeiro sentido da palavra.

Antes da EL, o JSP consistia em algumas *tags* especiais, como *Scriptlets*, expressões e outros elementos, que vimos no capítulo anterior, em que o código Java era escrito explicitamente nas páginas JSP. Isso talvez seja ruim, pois misturar código Java com *tags* pode deixar o código confuso, difícil de compreender. Assim, a utilização de JSTL e EL vai tornar o código mais limpo e claro, já que será possível implementar as páginas com código Java por meio de *tags*. Antes da EL, o JSP se parecia com isto, apenas um código macarrônico.



Figura 1 – Código macarrônico
Fonte: Banco de imagens Shutterstock (2017)

1.1 Recuperando dados com EL

A grande vantagem de utilizar EL nas páginas é a facilidade de recuperar os atributos que foram inseridos em algum escopo (*request*, *session*, *application*, *pageContext*). A sintaxe é bem simples, utilizamos o caractere \$ e as chaves \${ }. Dentro das chaves, colocamos o valor da chave em que o atributo foi inserido. Analise o “Exemplo de código-fonte para página de *script* JSP com EL”.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html lang="pt-br">
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<meta name="viewport" content="device-width">
<title>Exemplo EL</title>
</head>
<body>
```

```
<a href="Servlet">Enviar para Servlet</a>
<h1>${atributoDoRequest}</h1>

<body>
</html>
```

Código-fonte 1 – Exemplo de código-fonte para página de *script* JSP com EL
Fonte: Elaborado pelo autor (2017)

Com a EL, só precisamos saber o nome que foi dado ao atributo, não é necessário nenhum código Java. Imagine o cenário, na *Servlet*, criamos um atributo para enviar um texto para a página JSP receber e exibir na tela. Anteriormente, era preciso utilizar *Scriptlets* e, com isso, tínhamos que misturar código Java com *tags*. Agora podemos receber esse atributo por meio da EL de uma forma mais simples e clara, observe o código da *Servlet* que envia um atributo para a página JSP (Código-fonte “Exemplo de *Servlet* realizando *request* para JSP”).

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

    String nome = "Alexandre";
    request.setAttribute("usuario", nome);
    request.getRequestDispatcher("pagina.jsp").forward(request,
    response);

}
```

Código-fonte 2 – Exemplo de *Servlet* realizando *request* para JSP
Fonte: Elaborado pelo autor (2017)

Na *Servlet*, criamos o atributo “usuario” no *request*. Agora, vamos analisar o código para recuperar o atributo que foi enviado para a página utilizando EL (Código-fonte “JSP com EL recuperando atributo no *request*”).

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html lang="pt-br">
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<meta name="viewport" content="device-width">
<title>Exemplo EL</title>
</head>
<body>
```



```
<h1>${usuario}</h1>

<body>
</html>
```

Código-fonte 3 – JSP com EL recuperando atributo no *request*
Fonte: Elaborado pelo autor (2017)

A utilização da EL se dá pela chamada “`${ }`”, no nosso exemplo `${usuario}` dentro da página JSP, o atributo que iremos retornar deve estar entre chaves “`{ }`”. Observe que não é necessário colocar aspas entre o nome do atributo. Podemos conferir o resultado desse processo na Figura “Recuperação do atributo usuário por meio de EL na página JSP”.



Figura 2 – Recuperação do atributo usuário por meio de EL na página JSP
Fonte: Elaborado pelo autor (2017)

A EL pode fazer muito mais do que recuperar valores de atributos, conforme podemos ver no Código-fonte “JSP com EL realizando operações” e “no resultado da execução”, exibido na Figura “Resultado na página JSP com EL”.

```
<%@ page language="java" contentType="text/html;
charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
<title>Exemplo EL</title>
</head>
<body>
    <h1>Operação com EL : ${(((10 * 26)/57)+9)}</h1>

</body>
</html>
```

Código-fonte 4 – JSP com EL realizando operações
Fonte: Elaborado pelo autor (2017)



Figura 3 – Resultado na página JSP com EL
Fonte: Elaborado pelo autor (2017)

Além de operações matemáticas, como a do Código-fonte “JSP com EL realizando operações”, podemos realizar comparações e operações com *Strings*, como concatenação. No Código-fonte “EL executando uma operação de validação” é realizada uma operação ternária.

```
<%@ page language="java" contentType="text/html;
charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width,
initial-scale=1.0">
<title>Exemplo EL</title>
</head>
<body>
```

```
<h1>Operação ternária</h1>
<h2>Você recebeu dois números verifique se a soma
desses
        números é maior ou igual a 20, caso seja realize
a impressão na tela
        da mensagem "O número é maior ou igual a 20.",
caso contrário faça a
        impressão da mensagem "O número é menor do que
20.":</h2>

<p>${(17 > 7) ? 'O número é maior ou igual a 20.' :
'O número é menor do que 20.' }</p>

</body>
</html>
```

Código-fonte 5 – EL executando uma operação de validação
Fonte: Elaborado pelo autor (2017)

A Figura “Resultado da operação da EL realizando validação” apresenta o resultado da execução da página. Observe que o resultado é “O número é menor do que 20”.

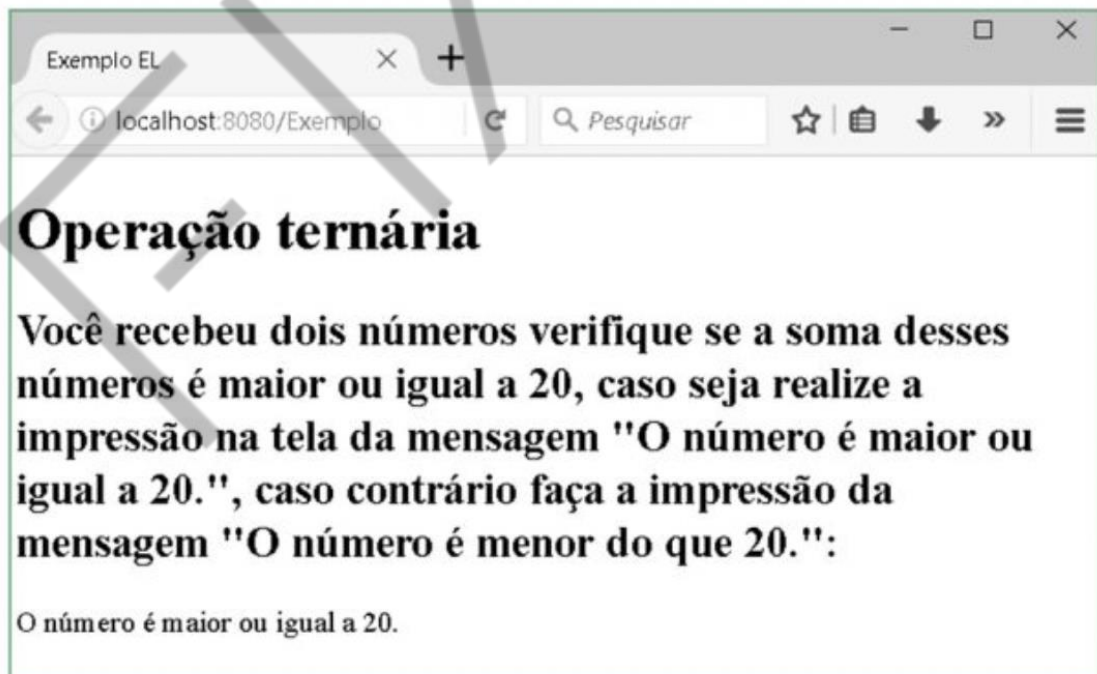


Figura 4 – Resultado da operação da EL realizando validação
Fonte: Elaborado pelo autor (2017)

Praticamente podemos executar qualquer operação básica com EL, como por exemplo comparações, operações matemáticas e acesso a objetos e coleções que estão em algum escopo. No Quadro “Operadores EL”, apresentamos todos os operadores que podemos utilizar com a EL em páginas JSP.

Operator	Description
.	Acessar uma propriedade de bean ou escopo
[]	Acessar um elemento de array ou list
()	Agrupe uma subexpressão para alterar a ordem de avaliação
+	Adição
-	Subtração ou negação de um valor
*	Multiplicação
/ ou div	Divisão
% or mod	Módulo (resto)
== ou eq	Igualdade
!= or ne	Diferente
< or lt	Menor que
> or gt	Maior que
<= or le	Menor ou igual
>= or ge	Maior ou igual
&& ou and	AND lógico
ou or	OR lógico
! or not	Unário, negação
empty	Teste de valores de variáveis vazias

Quadro 1 – Operadores EL
Fonte: Tutorials Point (2016)

1.2 Objetos implícitos

Acabamos de mencionar que podemos acessar os objetos e coleções que estão em algum escopo. Isso acontece porque temos acesso a vários objetos que estão implícitos nas páginas JSP. No Quadro “Objetos implícitos para EL”, podemos ver o nome e a descrição desses objetos implícitos.

Implicit Object	Description
PageScope	Variáveis de escopo do escopo da página
requestScope	Variáveis de escopo do escopo solicitação
sessionScope	Variáveis de escopo do escopo da sessão
ApplicationScope	Variáveis de escopo do escopo da aplicação
param	Solicitar parâmetros como sequências de caracteres
ParamValues	Solicitar parâmetros como coleções de strings
header	Cabeçalhos de solicitação HTTP como sequências de caracteres
headerValues	Cabeçalhos de solicitação HTTP como coleções de strings
initParam	Parâmetros de inicialização de contexto
cookie	Valores de cookie
pageContext	O objeto PageContext JSP para página atual

Quadro 2 – Objetos implícitos para EL
Fonte: Elaborado pelo autor (2017)

Você se lembra de que podemos acessar os atributos que foram adicionados em algum escopo por meio da EL simplesmente referenciando o seu nome? Por exemplo, o código `request.setAttribute("usuario","Teste");` adiciona um atributo com o nome "usuario" e valor "Teste". Na página JSP, podemos recuperar esse valor com a expressão `${usuario}`. Porém, podemos também adicionar atributos em todos os outros escopos, conforme vimos anteriormente. Então, quando não especificamos o escopo, a EL procura pelo atributo em todos. Para procurar em um escopo específico, precisamos utilizar o nome do escopo, junto do nome do atributo: `${requestScope.usuario}`, neste exemplo, a EL procura pelo atributo somente no escopo de **request**.

Vamos desenvolver um exemplo que trabalha com o objeto implícito **param**. Neste exemplo, vamos passar parâmetros para outro JSP e recuperá-los por meio

desse objeto implícito. O Código-fonte “Página exemplo1.jsp com exemplo de envio de dados para a página JSP” apresenta o formulário que envia os parâmetros para uma outra página JSP.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exemplo EL / Objetos Implícitos</title>
  </head>
  <body>
    <div class="container">
      <h1>Envio de parâmetros para outra página JSP.</h1>
      <form action="exemplo2.jsp" method="get">
        <fieldset>
          <legend>Formulário</legend>
          <div>
            <label for="pNome">Nome</label>
            <input type="text" name="nome" id="pNome" placeholder="Digite o seu nome">
          </div>
          <div>
            <label for="pIdade">Idade</label>
            <input type="number" name="idade" id="pIdade" placeholder="Digite a sua idade">
          </div>
          <div>
            <input type="submit" value="Enviar">
          </div>
        </fieldset>
      </form>
    </div>
  </body>
</html>
```

Código-fonte 6 – Página exemplo1.jsp com exemplo de envio de dados para a página JSP
Fonte: Elaborado pelo autor (2017)

A Figura “Resultado da página exemplo1.jsp que envia dados para outra página JSP” exibe o resultado da execução do JSP do Código-fonte “Página exemplo1.jsp com exemplo de envio de dados para a página JSP”.



Figura 5 – Resultado da página exemplo1.jsp que envia dados para outra página JSP
Fonte: Elaborado pelo autor (2017)

O Código-fonte “Página exemplo2.jsp recebendo os dados da página exemplo1.jsp” utiliza a **EL** e o objeto implícito **param** para recuperar o nome e a idade do formulário HTML. Note que **nome** e **idade** são os nomes dos campos do formulário que foram enviados como parâmetro para essa página.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Página do :${param.nome}</title>
</head>
<body>
  <h1>Recuperando os parâmetros do objeto implícito param.</h1>
  <p>Nome :${param.nome}</p>
  <p>Idade :${param.idade}</p>
</body>
</html>
```

Código-fonte 7 – Página exemplo2.jsp recebendo os dados da página exemplo1.jsp
Fonte: Elaborado pelo autor (2017)

Podemos ver a recepção do objeto implícito da mesma forma como trabalhamos com qualquer outro atributo dentro do escopo da página JSP. A Figura “Resultado da página exemplo2.jsp com os dados que foram enviados da página exemplo1.jsp” apresenta o resultado da execução da página, os valores “Alexandre” e 22 foram utilizados no formulário, para teste.



Figura 6 – Resultado da página exemplo2.jsp com os dados que foram enviados da página exemplo1.jsp
Fonte: Elaborado pelo autor (2017)

1.3 EL e *JavaBeans*

A EL facilita a leitura e o acesso aos **JavaBeans**. Por exemplo, imagine que precisamos exibir todos os dados do usuário do sistema Health Track. Esse usuário possui quinze informações que são importantes, como nome, data de nascimento, sexo, endereço etc.

Claro que podemos enviar cada informação como um atributo na *request*. Contudo, será preciso adicionar esses quinze atributos separadamente. Imagine o trabalho! No decorrer do curso, vimos que podemos criar uma classe **JavaBean** para armazenar as informações como atributos dessa classe. Então por que não criar uma classe **Usuario** e depois instanciá-lo, colocar todas as informações em seus atributos e finalmente adicionar esse objeto como um atributo no *request*? Fácil! Um atributo no *request* com todas as informações, que pode ser recuperado com a EL, na página JSP. Vamos a um exemplo. Analise o Código-fonte “Atributo do tipo *JavaBean(Object)*”.

```
protected void doGet(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException {

    Cliente cliente = new Cliente("Alexandre",40);
    request.setAttribute("cli",cliente);
    request.getRequestDispatcher("pagina.jsp").forward(request, response);
}
```

Código-fonte 8 – Atributo do tipo *JavaBean(Object)*, que foi criado no *request*
Fonte: Elaborado pelo autor (2017)

No Código-fonte “Atributo do tipo *JavaBean(Object)*, que foi criado no *request*” podemos ver um método de uma *Servlet* que cria uma instância de um **JavaBean** (Cliente) e define um atributo no **request** chamado “cli”, que será enviado para a página JSP. Agora, veremos na página JSP a recuperação desse atributo com a EL, no Código-fonte a seguir.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html lang="pt-br">
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<meta name="viewport" content="device-width">
<title>Exemplo JavaBean com EL</title>
</head>
<body>

    <h2>${cli.nome}</h2>
    <h2>${cli.idade}</h2>

</body>
</html>
```

Código-fonte 9 – EL recuperando atributo enviado no *request* “cli” em página JSP
Fonte: Elaborado pelo autor (2017)

No Código-fonte “EL recuperando atributo enviado no *request* “cli” em página JSP” podemos ver como recuperar um **JavaBean** por meio da EL na página JSP. Basta chamar o atributo que foi enviado por meio do request, nesse caso, “cli”. Se observarmos os **JavaBeans**, vamos ver que possuem os atributos que estão sendo acessados por meio da EL.

Para ser possível recuperar os atributos de um **JavaBean**, que foi enviado como atributo, devemos implementar a estrutura completa, com os métodos *get* e *set*. O Código-fonte “Exemplo de *bean* padrão” implementa a classe **JavaBean** “Cliente”.

```
package br.com.fiap.ead;

public class Cliente {

    private String nome;
    private int idade;

    public Cliente(){}

    public Cliente(String nome, int idade){
        this.nome = nome;
        this.idade = idade;
    }

    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;  
        }  
    }  
}
```

Código-fonte 10 – Exemplo de *bean* padrão
Fonte: Elaborado pelo autor (2017)

A regra para que tudo funcione é que tenha os métodos *get* e *set*, a EL internamente realiza acesso aos atributos por meio desses métodos.

Portanto, para recuperar a informação o **JavaBean** precisa do método *get*, já para adicionar uma informação, será preciso o método *set*. Entretanto, na EL sempre referenciamos o nome do atributo, já que existe um padrão de nomenclatura para os métodos *get* e *set*. No Código-fonte “EL acessando os atributos nome e idade do objeto “cli” em página JSP” a EL utiliza o método *getNome()* e *getIdade()* para recuperar e exibir os valores na página JSP.

```
<h2>${cli.nome}</h2>  
<h2>${cli.idade}</h2>
```

Código-fonte 11 – EL acessando os atributos nome e idade do objeto “cli” em página JSP
Fonte: Elaborado pelo autor (2017)

Atenção: Você deve utilizar o nome correto dos atributos ou a EL não vai conseguir acessá-los dentro da estrutura do projeto.

1.4 JSTL

JSTL (*JavaServer Pages Standard Tag Library*), Tag Library ou TagLibs é uma coleção de *tags* que possuem funções da linguagem Java, ou seja, podemos utilizar *tags* para adicionar comandos Java na página JSP.

Dessa forma, podemos utilizar um laço **for**, uma estrutura de decisão do tipo **if/else** ou **switch-case** diretamente no JSP por meio de *tags*. Isso mesmo, *tags*! O intuito da criação dessas bibliotecas foi facilitar a vida do desenvolvedor, porque

ninguém aguentava mais o código “macarrônico”, aquele código cheio de *Scriptlets* com código Java no meio do HTML, um verdadeiro caos.

Para exemplificar essa mudança, vamos desenvolver uma página que utiliza código Java e outra que utiliza o JSTL. O Código-fonte “Página JSP exemplo1.jsp com *Scriptlets*” apresenta um JSP que utiliza código Java.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Exemplo JSP / Scriptlets</title>
</head>
<body>

    <%
        String msg = "Alô mundo!";
    %>

    <p>Mensagem :<%=msg%></p>
</body>
</html>
```

Código-fonte 12 – Página JSP exemplo1.jsp com *Scriptlets*
Fonte: Elaborado pelo autor (2017)

A Figura “Resultado da página exemplo1.jsp com *Scriptlets*” exibe o resultado da execução da página.



Figura 7 – Resultado da página exemplo1.jsp com *Scriptlets*
Fonte: Elaborado pelo autor (2017)

Agora vamos ver como a JSTL e a EL praticamente resolveram isso, para não ter nenhuma linha de código nativo Java em uma página JSP. Duvida? Vamos analisar o Código-fonte “Página JSP exemplo1.jsp com *TagLibs*”.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

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

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Exemplo JSP / Scriptlets</title>
</head>
<body>

    <c:set var="msg" value="Alô mundo!" />

    <p>Mensagem : ${msg}</p>

</body>
</html>
```

Código-fonte 13 – Página JSP exemplo1.jsp com *TagLibs*
Fonte: Elaborado pelo autor (2017)

Não se preocupe com essas novas *tags*, veremos com detalhes a seguir. Nesse momento, perceba que o código ficou mais organizado e fácil de ser entendido, até mesmo por uma pessoa que não conhece a linguagem Java, mas conhece HTML. A Figura “Resultado da página exemplo1.jsp com *TagLibs*” exibe o resultado da página no browser. Note que o resultado é o mesmo!



Figura 8 – Resultado da página exemplo1.jsp com *TagLibs*
Fonte: Elaborado pelo autor (2017)

Com esse recurso, o desenvolvimento e a manutenção das páginas em JSP ficou mais fácil.

1.5 Utilização

Veremos como é fácil utilizar as *TagLibs* dentro de nossas páginas JSP. Hoje, estamos utilizando o JSP e *Servlets* para construir nossas aplicações web. Porém, é possível utilizar o JSTL com outros *frameworks* Java web, como, por exemplo, o Spring MVC.

Para começar, precisamos das bibliotecas, dos *jars* das *TagLibs*, assim como utilizamos *jars* externos para conectar no banco de dados oracle (jdbc), precisaremos de outros *jars*, porque as *TagLibs* são um projeto da Apache do grupo Jakarta. Para fazer o download dos *jars*, acesse o seguinte endereço web:

<http://tomcat.apache.org/taglibs/index.html>

Depois de realizar o download dos arquivos, precisamos ir até a pasta lib do seu projeto. Como utilizamos o Eclipse para os exemplos, vamos ver como chegar até essa pasta. Mas caso tenha utilizado outra IDE, a estrutura é igual, só muda a forma

de chegar até a pasta. No Eclipse, vamos até a pasta WebContent e expandiremos toda a estrutura de pastas, conforme a “Figura Estrutura de pastas Eclipse”.

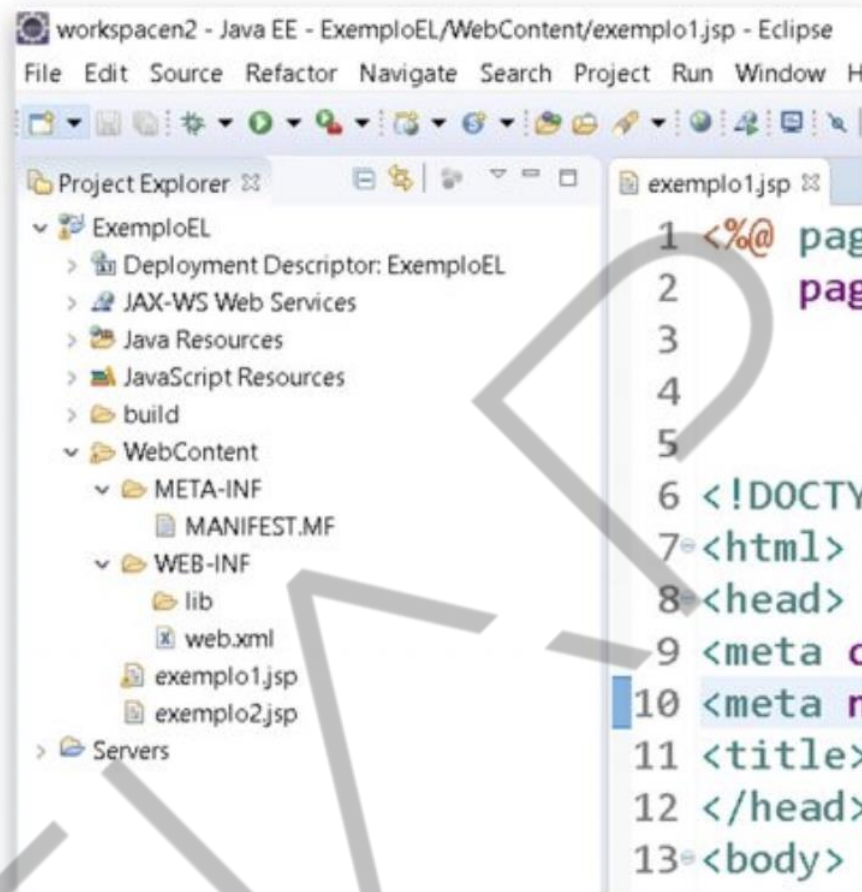


Figura 9 – Estrutura de pastas Eclipse
Fonte: Elaborado pelo autor (2017)

Agora pegue os arquivos e copie na pasta **lib** que está dentro da pasta **WEB-INF**. Diferentemente do projeto de exemplo de conexão com banco de dados, neste projeto web, não será preciso adicionar os *jars* no *build path*, pois como é uma aplicação web, existe essa pasta específica para adicionar as bibliotecas.

Caso copie os *jars* em qualquer outra pasta e adicione no *build path*, o projeto pode até compilar, porém, quando executar, o servidor não será capaz de encontrar as bibliotecas e o erro *ClassNotFoundException* vai acontecer.

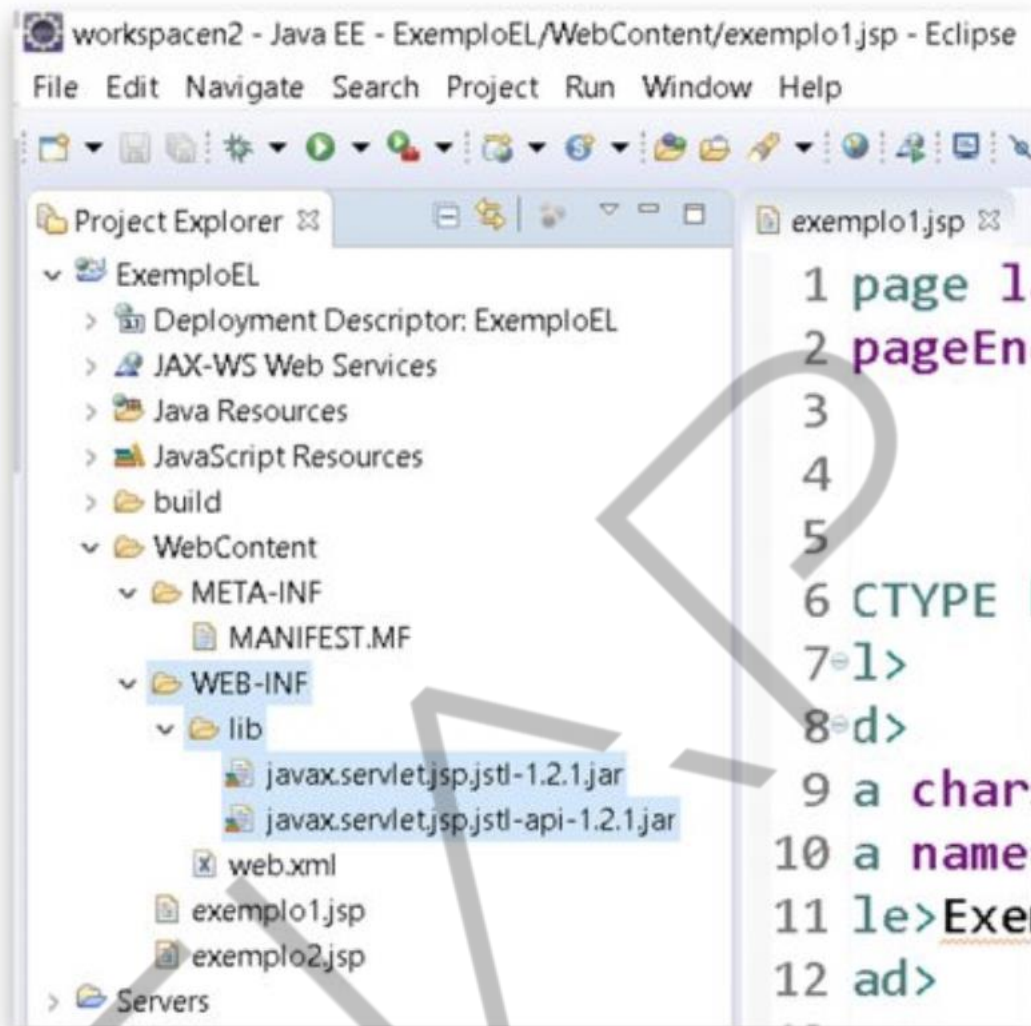


Figura 10 – Jars das *TagLibs* na pasta lib
Fonte: Elaborado pelo autor (2017)

Agora só falta ir para a página JSP e realizar a referência a essas bibliotecas via diretiva **taglib**.

No Código-fonte “Página JSP exemplo1.jsp com *TagLibs*” apresentamos um primeiro exemplo de uso do JSTL. Nesse código, realizamos a referência da **taglib** para a página funcionar corretamente, percebeu? Fique atento ao topo da página e perceberá uma segunda diretiva chamada **taglib**, é essa diretiva que nos permite utilizar as TagLibs em nossas páginas JSP.

No Código-fonte “Diretiva *taglib* com o *import* para a biblioteca core”, apresentamos separadamente a chamada da **taglib**.


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

Código-fonte 14 – Diretiva *taglib* com o *import* para a biblioteca *core*

Fonte: Elaborado pelo autor (2017)

Vamos aos detalhes dessa diretiva:

- **taglib** - É o nome da diretiva.
- **prefix** - É a propriedade que nos permite criar um token (identificador da *tag* na página JSP). Ex: `<c:if test=""></c:if>`. Existe uma convenção que pede o uso da letra “c”, por estar trabalhando com a biblioteca “core”. O exemplo aqui fala da biblioteca *core*, mas temos mais uma que é a *formatting*, para essa é melhor utilizarmos letra *f*! Acertou quem disse não! Em *formatting*, a convenção pede (*fmt*), que é o acrônimo de formatar, logo vamos falar mais dela.
- **uri** - É o caminho relativo da api (biblioteca local). O servidor de aplicação usa esse caminho relativo para carregar o conjunto de *tags* específicas para o token selecionado em prefix.

Existem quatro grupos de bibliotecas nos quais podemos classificar as TagLibs:

- Core tags
- Formatting tags
- SQL tags
- XML tags

Vamos abordar as duas principais, que são mais utilizadas no mercado: *core* e *formatting*.

1.6 TagLibs – Core

A biblioteca **core** da TagLibs resume, em seu interior, as principais instruções da linguagem Java no formato de *tag*. No Quadro Tabela de *tags* da biblioteca *core*, apresentamos todas as *tags* da biblioteca *core*.

TAG	Descrição
<c:out>	<% = ...> nas expressões
<c:set>	Define o resultado de uma avaliação de expressão em um 'escopo', você pode definir uma espécie de variável no escopo em que você estiver trabalhando.
<c:remove>	Remove uma variável de escopo (de um escopo específico se especificando).
<c:catch>	Captura qualquer Throwable (Exception) que ocorrer no body e se você quiser pode optar por expor o resultado é opcional
<c:if>	Tag condicional simples que avalia seu body se a condição fornecida for verdadeira. O mesmo que if mas sem o Else.
<c:choose>	Tag condicional simples que estabelece um contexto para operações condicionais mutuamente exclusivas, marcadas por <when> e <otherwise>. O mesmo que um switch-case, que pode ser utilizado como if/else.
<c:when>	Subtag de <choose> que inclui seu body se sua condição foi alternada para 'true'.
<c:otherwise>	Subtag de <choose> que segue <when>, marca e executa somente se todas as condições anteriores forem avaliadas como 'falso'.
<c:import>	Recuara uma URL absoluta ou relativa e carrega o seu conteúdo para a página onde se está utilizando. Muito utilizado para modularização.
<c:forEach>	A tag de iteração básicas, aceitando muitos tipos de collections e subconjuntos de suporte e outras funcionalidades. O mesmo que o foreach.
<c:forTokens>	Iterates sobre tokens, separados pelos delimitadores fornecidos.
<c:param>	Adiciona um parâmetro a uma URL contendo a tag 'import'
<c:redirect>	Redireciona para um novo URL.
<c:url>	Cria uma URL com parâmetros de consulta opcionais.

Quadro 3 – Tabela de *tags* da biblioteca *core*
 Fonte: Tutorials Point (2016)

Vamos abordar as *tags* principais dessa biblioteca.

1.6.1 <c:forEach>

- Permite iterar uma lista de elementos.
- Ótima para montar tabelas e selects (combobox).

Vamos desenvolver um exemplo que utiliza essa *tag* para exibir uma tabela para o usuário. Para isso, precisamos implementar uma *Servlet* que envia um objeto de *Collections* (Lista) para a página JSP, conforme o Código-fonte “*Servlet* gerando um *ArrayList* no *request* e enviando para a página JSP”.

```
protected void doGet(HttpServletRequest req,
    HttpServletResponse resp)
    throws ServletException, IOException {
    ArrayList<String> nomes = new ArrayList<String>();
    nomes.add("Alexandre");
    nomes.add("Carlos");
    req.setAttribute("lista", nomes);
}
```

Código-fonte 15 – *Servlet* gerando um *ArrayList* no *request* e enviando para a página JSP
Fonte: Elaborado pelo autor (2017)

Observe que o nome do atributo inserido no *request* foi “**lista**”, esse nome será utilizado para recuperar as informações na página JSP. Agora vamos ver na página JSP, ela recupera esse atributo e utiliza a *tag* <c:forEach> para montar a tabela.

```
<%@ page language="java" contentType="text/html; charset=UTF-
8" pageEncoding="UTF-8"%>
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-
scale=1.0">
<title>Exemplo JSP / TagLibs</title>
</head>
<body>
    <table border="1">
        <tr>
            <th>Nome</th>
        </tr>
```

```
<c:forEach var="n" items="${lista}">
    <tr>
        <td>${n}</td>
    </tr>
</c:forEach>
</table>
</body>
</html>
```

Código-fonte 16 – Página com o código da TagLib para iteração da lista
Fonte: Elaborado pelo autor (2017)

A tag `<c:forEach>` possui dois atributos principais, o **var** define o nome da variável que vai receber cada um dos itens da coleção, já o atributo **items** define a coleção que será utilizada para a iteração. É muito parecido com o **foreach** do Java.

Observe também que a primeira linha da tabela não precisa estar dentro do *foreach*, pois ela define a coluna de cabeçalho da tabela. A Figura “Resultado da página exemplo1.jsp com a TagLib forEach” exibe o resultado do processo com a TagLib forEach.

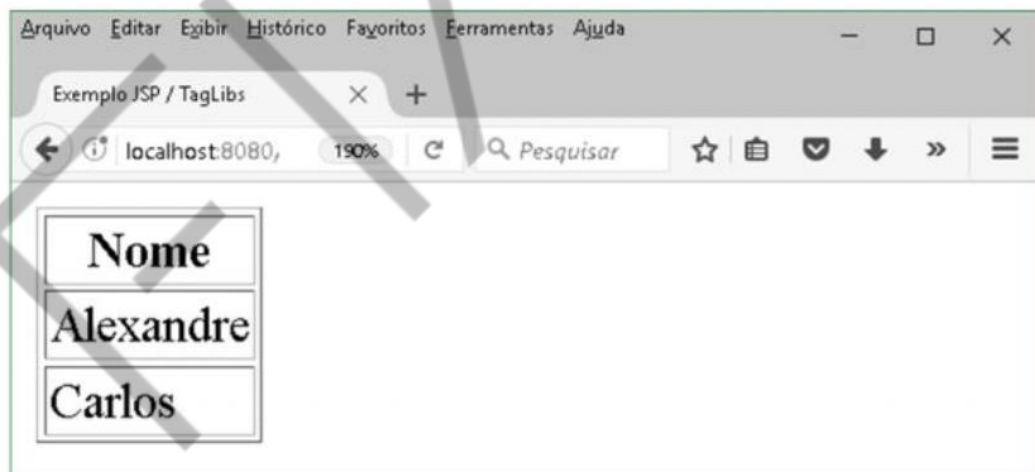


Figura 11 – Resultado da página exemplo1.jsp com a TagLib forEach
Fonte: Elaborado pelo autor (2017)

1.6.2 <c:if>

Tag que realiza uma estrutura de seleção (if). O atributo **test** recebe uma EL que realiza uma comparação e resulta em um *boolean*. Note que não temos a *tag else* do **if**. Alguns exemplos são apresentados no Código-fonte “Página com o código da TagLib if”.

```
<c:if test="${numero > 100 }">
    <p>Valor Maior que 100</p>
</c:if>
<c:if test="${not empty lista }">
    <!-- Tabela -->
</c:if>
<c:if test="${usuario == 'admin' }">
    <p>Ola Administrador!</p>
</c:if>
```

Código-fonte 17 – Página com o código da TagLib if
Fonte: Elaborado pelo autor (2017)

1.6.3 <c:choose> <c:when> <c:otherwise>

Muito parecido com o **switch-case** do Java. Permite testar várias condições, somente um bloco é executado. Como não existe a *tag else*, podemos utilizar essas *tags* para ter esse comportamento, já que é possível realizar o teste com o <c:when> e adicionar a *tag* <c:otherwise> para funcionar como o **else**. Apresentamos um exemplo no Código-fonte “Página com o código da TagLib choose”.

```
<c:choose>
<c:when test="${numero > 100 }">
    <p>Valor Maior que 100</p>
</c:when>
<c:when test="${numero < 50 }">
    <p>Valor Menor que 50</p>
```

```
</c:when>
<c:otherwise>
    <p>Valor entre 50 e 100</p>
</c:otherwise>
</c:choose>
```

Código-fonte 18 – Página com o código da TagLib choose
Fonte: Elaborado pelo autor (2017)

1.6.4 <c:out>

Utilizado para exibir informações na página. Apresentamos um exemplo no Código-fonte “Página com o código da TagLib”.

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

Código-fonte 19 – Página com o código da TagLib out
Fonte: Elaborado pelo autor (2017)

1.6.5 <c:url>

Permite criar links com parâmetros. Dessa forma, o código fica mais fácil de ser entendido, já que não é necessário criar uma *String* com concatenações. Podemos adicionar a quantidade de parâmetros que forem necessários. O Código-fonte “Página com o código da TagLib url” exibe um exemplo.

```
<c:url value="editarCliente" var="link">
    <c:param name="nome" value="\${cli.nome}"/>
</c:url>
<a href="\${link}">Cliente</a>
```

Código-fonte 20 – Página com o código da TagLib url
Fonte: Elaborado pelo autor (2017)

Depois, podemos ver o resultado da execução da página JSP no browser. Apresentamos o resultado no Código-fonte “Página com o resultado da TagLib url”.

```
<a href="editarCliente?nome=Alexandre">Cliente</a>
```

Código-fonte 21 – Página com o resultado da TagLib url
Fonte: Elaborado pelo autor (2017)

1.6.6 <c:import>

Permite importar páginas web do mesmo contexto web (aplicação), de contextos diferentes e até mesmo de máquinas diferentes.

Atributo	Descrição	Requerido?	Padrão
url	URL a ser importada.	Sim	Nenhum
context	"/" seguido do nome da aplicação web local.	Não	Contexto corrente
var	Nome do atributo onde será armazenado o conteúdo da página importada.	Não	Nenhum
scope	Escopo do atributo onde será armazenado o conteúdo da página importada. Pode ser: page, request, session, application.	Não	Page

Quadro 4 – Atributos da *tag import*
Fonte: Elaborado pelo autor (2017)

```
<c:import url="exemplo2.jsp" scope="session"/>
```

Código-fonte 22 – Página com o código da TagLib import
Fonte: Elaborado pelo autor (2017)

1.7 TagLibs – Formatting

As *tags* de formatação JSTL são usadas para formatar e exibir texto, data, hora e números para sites internacionalizados. Antes de vermos algumas *tags*, vamos observar o quadro com a lista de todas as *tags* da biblioteca, para depois abordar as

duas principais *tags* de *formatting* e sua sintaxe. Assim como a *taglib core*, precisamos adicionar a *taglib* de *formatting* nas páginas que irão utilizar a formatação:

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

Código-fonte 23 – Página com o código da diretiva *taglib* e o *prefix* para a biblioteca *formatting*
Fonte: Elaborado pelo autor (2017)

Quadro com as *tags* de formatação da biblioteca *formatting*.

Tag	Descrição
<fmt:formatNumber>	Para renderizar valores numéricos com precisão ou formato específico.
<fmt:parseNumber>	Analisa a representação de sequência de caracteres de um número, moeda ou porcentagem.
<fmt:formatDate>	Formata uma data e/ou uma hora usando os estilos e padrões fornecidos.
<fmt:parseDate>	Analisa a representação de sequência de uma data e/ou tempo.
<fmt:bundle>	Carrega um pacote de recursos para ser usado pelo seu corpo de tag.
<fmt:setLocale>	Armazena a localidade fornecida na variável de configuração de localidade.
<fmt:setBundle>	Carrega um bundle de recursos e o armazena na variável de escopo nomeada ou na variável de configuração do bundle.
<fmt:timeZone>	Especifica o fuso horário para qualquer formatação de tempo ou parsing de ações aninhadas em seu conteúdo.
<fmt:setTimeZone>	Armazena o determinado fuso horário na variável de configuração fuso horário.
<fmt:message>	Para exibir uma mensagem internacionalizada.

Quadro 5 – *Tags* de *formatting*
Fonte: Tutorials Point (2016)

1.7.1 <fmt:formatDate>

Já tentou exibir na página JSP um objeto que representa uma data? Se já tentou, percebeu que não é muito amigável o que é exibido. Por isso, precisamos utilizar um formatador. Só precisamos de atenção, pois deve se passar um objeto de

data (*Date*) e não uma *String* para a *tag*. No Código-fonte “Página com exemplos da *tag* formatDate” apresentamos vários exemplos. Criamos um objeto chamado *data*, que recebe a data atual.

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

<html>
<head>
  <title>JSTL Formatação de datas</title>
</head>
<body>
<h3>Formatação de datas:</h3>
<c:set var="data" value="<%=new java.util.Date()%>" />

<p>Data Formatada (1): <fmt:formatDate type="time"
                        value="${data}" /></p>
<p>Data Formatada (2): <fmt:formatDate type="date"
                        value="${data}" /></p>
<p>Data Formatada (3): <fmt:formatDate type="both"
                        value="${data}" /></p>
<p>Data          Formatada          (4):          <fmt:formatDate
pattern="dd/MM/yyyy"
                        value="${data}" /></p>

</body>
</html>
```

Código-fonte 24 – Página com exemplos da *tag* formatDate
Fonte: Elaborado pelo autor (2017)

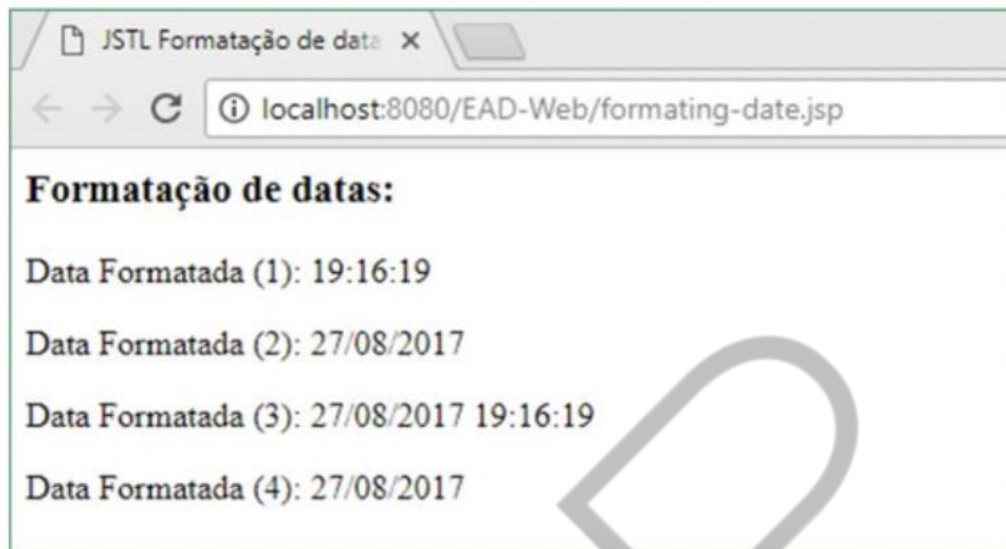


Figura 12 – Resultado da página de formatação de data
Fonte: Elaborado pelo autor (2017)

O atributo **type** determina o tipo de dado que será exibido: a data, hora ou ambos. O atributo **value** recebe o objeto *Date* que representa a data e será formatado. Podemos ainda utilizar o atributo **pattern** para determinar exatamente a forma como a data será exibida. No Quadro “Quadro com opções para o pattern de data” apresentamos os valores possíveis para esse atributo.

Código	Descrição	Exemplo
G	The area designator	AD
Y	The year	2002
M	The month	April & 04
d	The day of the month	20
h	The hour(12-hour time)	12
H	The hour(24-hour time)	0
m	The minute	45
s	The second	52
S	The millisecond	970
E	The day of the week	Tuesday
D	The day of the year	180
F	The day of the week in the month	2(end we in month)
w	The week in the year	27
W	The week in the month	2
a	The a.m./p.m. indicator	PM
k	The hour(12-hour time)	24
K	The hour(24-hour time)	0
z	The time zone	Central Standard Time
'		The escape for text
"		The single quote

Quadro 6 – Quadro com opções para o *pattern* de data
 Fonte: Elaborado pelo autor (2017)

1.7.2 <fmt:formatNumber>

Podemos formatar números no formato de moedas e porcentagem, e utilizar a localização do usuário para mostrar o valor da moeda local. O atributo **value** recebe o número que será formatado, já o atributo **type** recebe o tipo de formatação, como **number** para número, **currency** para valores monetários e **percent** para porcentagens. Também é possível utilizar o atributo **pattern** para definir

detalhadamente o tipo de formatação. O Código-fonte “Página com o código da tag formatNumber” apresenta alguns exemplos da tag de formatação de números.

```
<%@ page language="java" contentType="text/html;
charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt"
uri="http://java.sun.com/jsp/jstl/fmt"%>

<html>
<head>
<title>JSTL Formatação de Números</title>
</head>
<body>
    <h3>Formatar Números:</h3>
    <c:set var="valor" value="999.888" />
    <p>
        Número Formatado (1):
        <fmt:formatNumber value="${valor}"
type="currency" />
    </p>
    <p>
        Número Formatado (2):
        <fmt:formatNumber type="number"
maxIntegerDigits="3" value="${valor}" />
    </p>
    <p>
        Número Formatado (3):
        <fmt:formatNumber type="number"
maxFractionDigits="3" value="${valor}" />
    </p>
    <p>
        Número Formatado (4):
        <fmt:formatNumber type="percent"
value="${valor}" />
    </p>
    <p>
        Número Formatado (5):
        <fmt:formatNumber type="number"
pattern="###.###E0" value="${valor}" />
    </p>
</body>
</html>
```

Código-fonte 25 – Página com o código da tag formatNumber
Fonte: Elaborado pelo autor (2017)

Utilizamos o valor 999.888 para realizar os testes de formatação. Veja que podemos determinar o tipo e a quantidade de dígitos, tanto na parte inteira, quanto na parte fracionária (*maxIntegerDigits* e *maxFactionDigits*). O resultado pode ser observado na Figura “Resultado da página de formatação de números”.



Figura 13 – Resultado da página de formatação de números
Fonte: Elaborado pelo autor (2017)

As *tags formatting* estão intimamente ligadas às configurações do navegador do usuário, principalmente no que diz respeito à língua e à localização.

2 PRÁTICA!

Vamos criar um exemplo que utilizará tudo que vimos até agora: JSP, *Servlets*, JSTL e EL. Neste exemplo, vamos desenvolver as funcionalidades de cadastro e listagem de produtos.

O primeiro passo é criar um novo projeto Java web, como fizemos das outras vezes. No eclipse, utilize a opção: File > New > Dynamic Web Project (Figura “Criando um projeto web – Parte 1”).

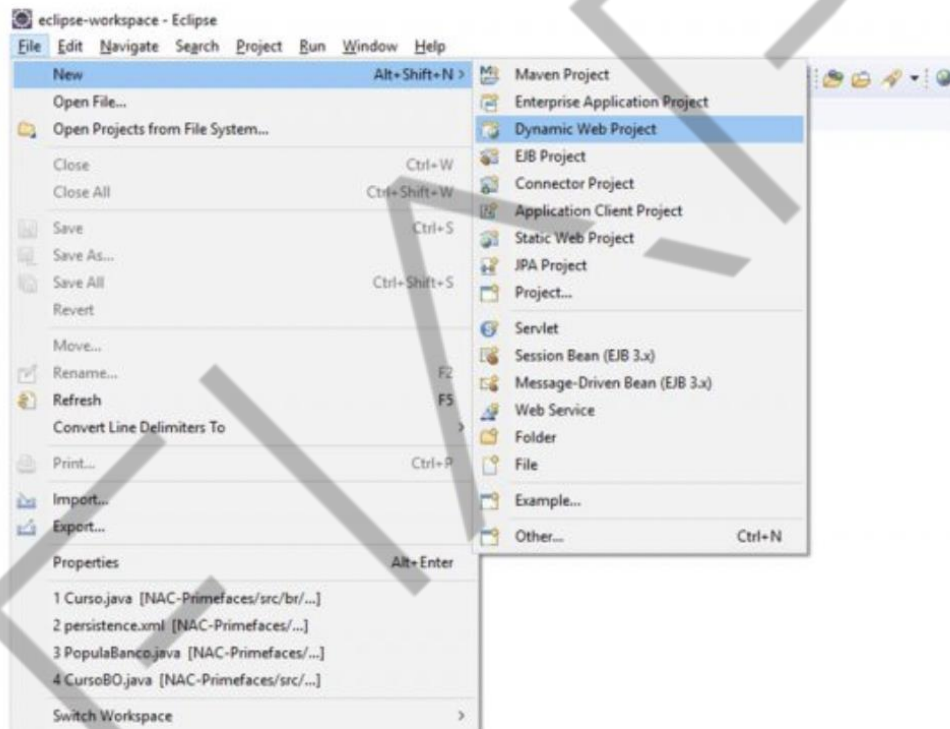


Figura 14 – Criando um projeto web – Parte 1
Fonte: Elaborado pelo autor (2017)

Dê um nome ao projeto e configure o servidor Tomcat 7 (Figura “Criando um projeto web – Parte 2”).

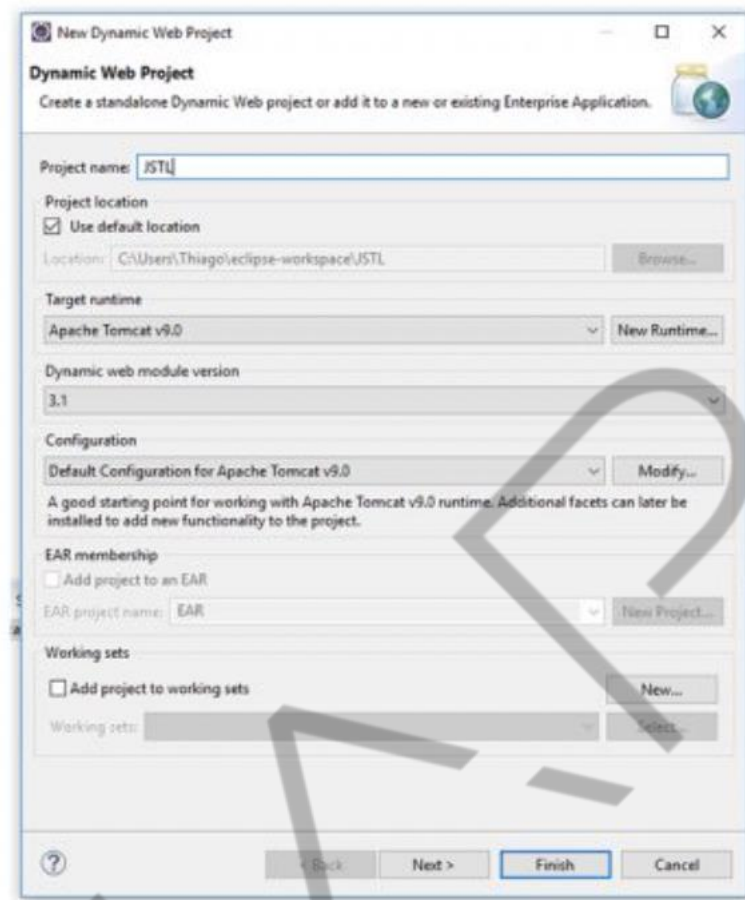


Figura 15 – Criando um projeto web – Parte 2
Fonte: Elaborado pelo autor (2017)

Clique em Next e clique novamente em Next até chegar à última tela, marque para criar o web.xml e finalize o processo (Figura “Criando um projeto web – Parte 3”).

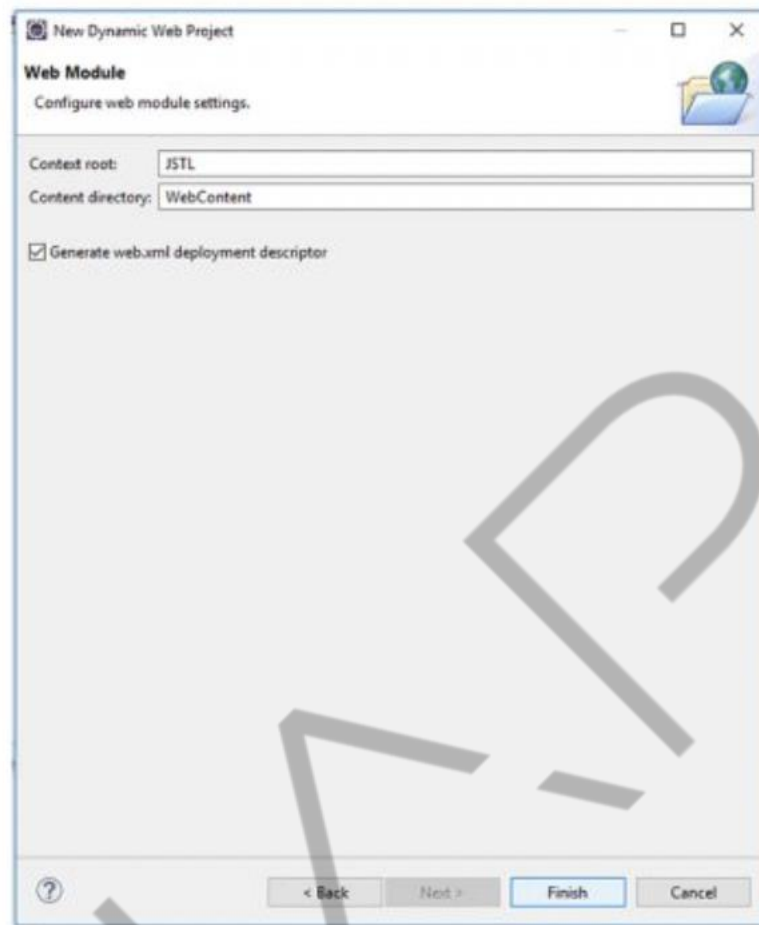


Figura 16 – Criando um projeto web – Parte 3
Fonte: Elaborado pelo autor (2017)

Para deixar nossa aplicação com um layout profissional, vamos utilizar o *bootstrap*. Para isso, configure o projeto como o que fizemos no capítulo anterior. Crie um diretório chamado **resources** dentro de **Web Content**, copie os arquivos de css e javascript do *bootstrap* e *jquery*. Vamos utilizar os arquivos JSPs para realizar o *include* do *header*, *footer* e *menu*.

O próximo passo é criar a tela com um formulário de cadastro de produto. O formulário será parecido com o desenvolvido no capítulo de *Servlets*, só que utilizaremos um arquivo JSP em vez de HTML e vamos utilizar o *bootstrap* para dar estilo ao formulário. Crie uma página JSP conforme a Figura “Criando uma página JSP – Parte 1”.

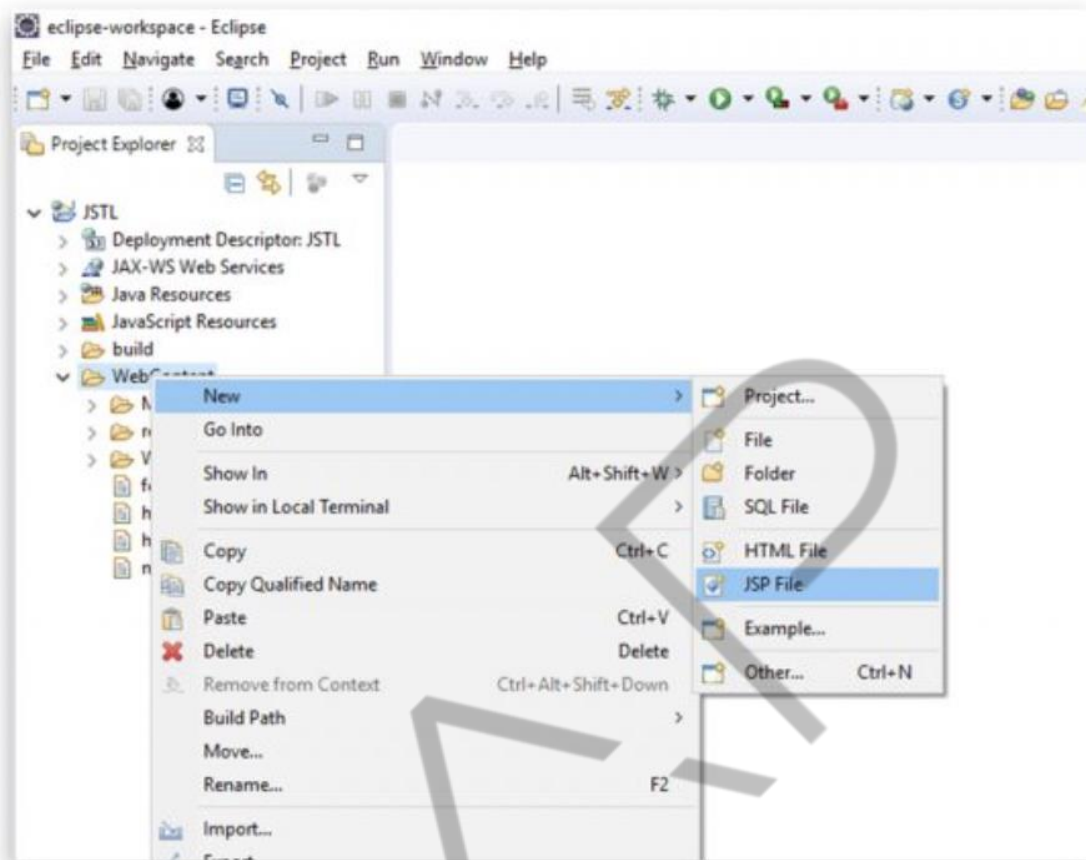


Figura 17 – Criando uma página JSP – Parte 1
Fonte: Elaborado pelo autor (2017)

Vamos utilizar “cadastro-produto.jsp” para o nome da página (Figura “Criando uma página JSP – Parte 2”).

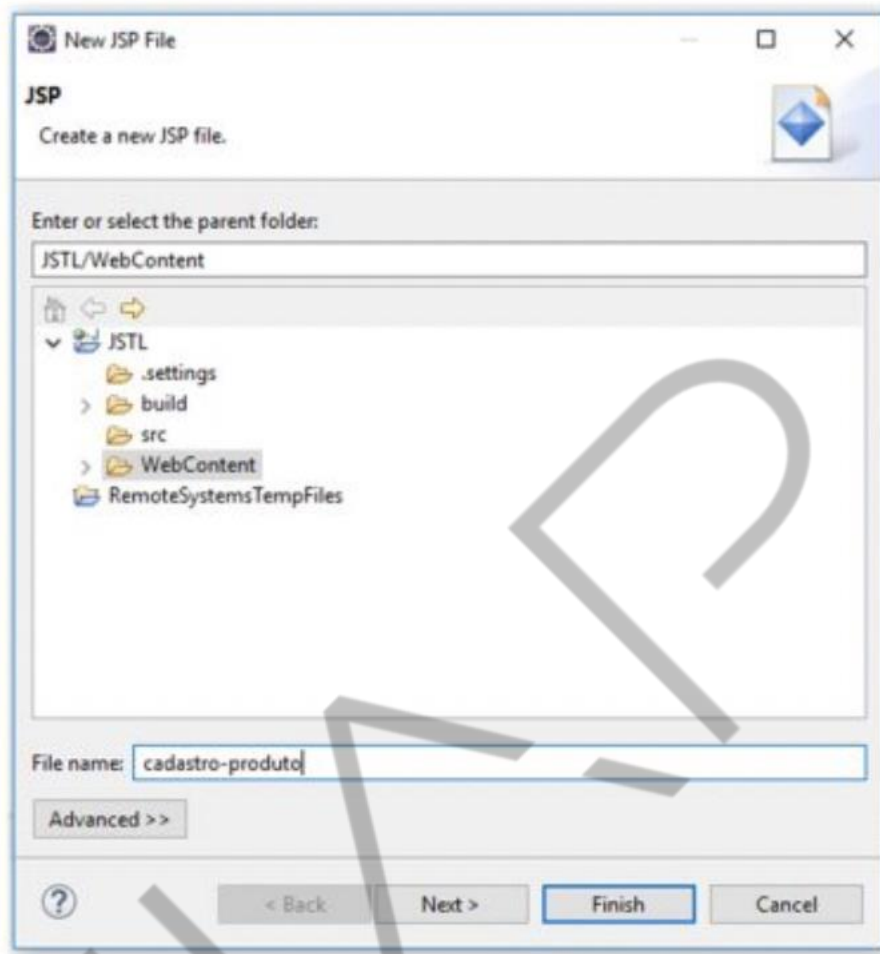


Figura 18 – Criando uma página JSP – Parte 2
Fonte: Elaborado pelo autor (2017)

Agora vamos para a implementação! Desenvolva o formulário de cadastro de produto, com os campos nome, quantidade e valor. O formulário deve ter o *action* para “produto” e o método POST, conforme o Código-fonte “Página de cadastro de produto”.

```
<%@ 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 Produto</title>
```

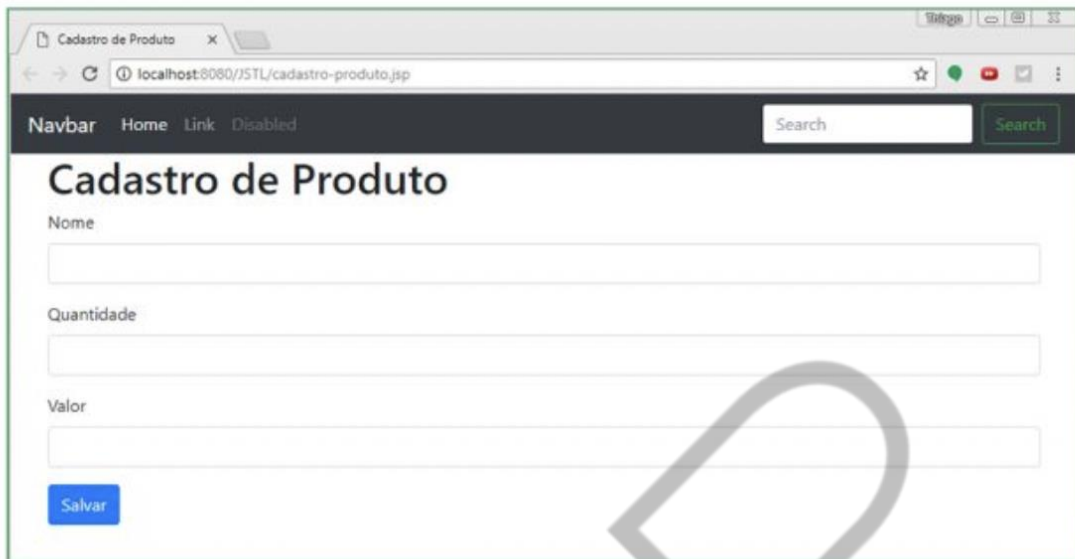
```
<%@ include file="header.jsp" %>
</head>
<body>

<%@ include file="menu.jsp" %>
  <div class="container">
    <h1>Cadastro de Produto</h1>

    <form action="produto" method="post">
      <div class="form-group">
        <label for="id-nome">Nome</label>
        <input type="text" name="nome"
id="id-nome" class="form-control">
      </div>
      <div class="form-group">
        <label for="id-
qtd">Quantidade</label>
        <input type="text" name="quantidade"
id="id-qtd" class="form-control">
      </div>
      <div class="form-group">
        <label for="id-valor">Valor</label>
        <input type="text" name="valor"
id="id-valor" class="form-control">
      </div>
      <input type="submit" value="Salvar"
class="btn btn-primary">
    </form>
  </div>
  <%@ include file="footer.jsp" %>
</body>
</html>
```

Código-fonte 26 – Página de cadastro de produto
Fonte: Elaborado pelo autor (2017)

Observe que utilizamos as classes do *bootstrap* “form-group” e “form-control”, o primeiro para agrupar o *label* e o campo e o segundo para dar um estilo ao campo de *input*. Repare também que incluímos os JSPs: *header*, para fazer o link do css, *footer*, para os *scripts* e *menu* para adicionar a barra de navegação. O resultado da execução pode ser visto na Figura “Execução da página JSP de cadastro de produto”.



The screenshot shows a web browser window with the address bar displaying 'localhost:8080/JSTL/cadastro-produto.jsp'. The page has a dark blue header with a 'Navbar' section containing links for 'Home', 'Link', and 'Disabled'. To the right of the links is a search bar with the text 'Search' and a green 'Search' button. Below the header, the main content area has the title 'Cadastro de Produto' in a large, bold font. Underneath the title are three input fields: 'Nome', 'Quantidade', and 'Valor'. At the bottom left of the form is a blue button labeled 'Salvar'.

Figura 19 – Execução da página JSP de cadastro de produto
Fonte: Elaborado pelo autor (2017)

Só para lembrar, para executar, clique com o botão direito do mouse na página JSP e escolha a opção “Run as” > “Run on Server”, depois escolha o Tomcat 9.

Agora chegou o momento de criar a *Servlet* que receberá os dados do formulário. Clique com o botão direito do mouse na pasta src e escolha New > Servlet (Figura “Criação da classe *Servlet* – Parte 1”).

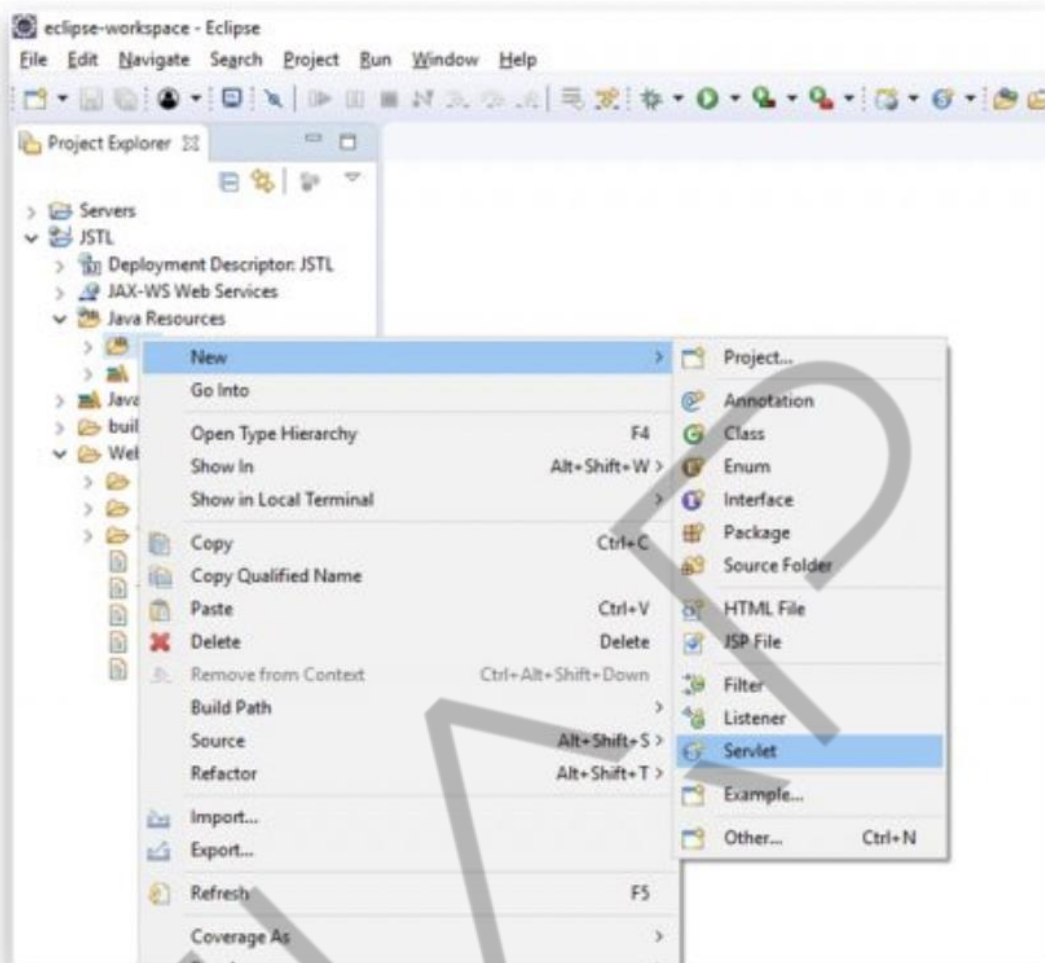


Figura 20 – Criação da classe *Servlet* – Parte 1
Fonte: Elaborado pelo autor (2017)

Configure o pacote e o nome da classe *Servlet*. Como sugestão, utilizamos “br.com.fiap.controller” para o pacote e “ProdutoServlet” para a classe (Figura “Criação da classe *Servlet* – Parte 2”).

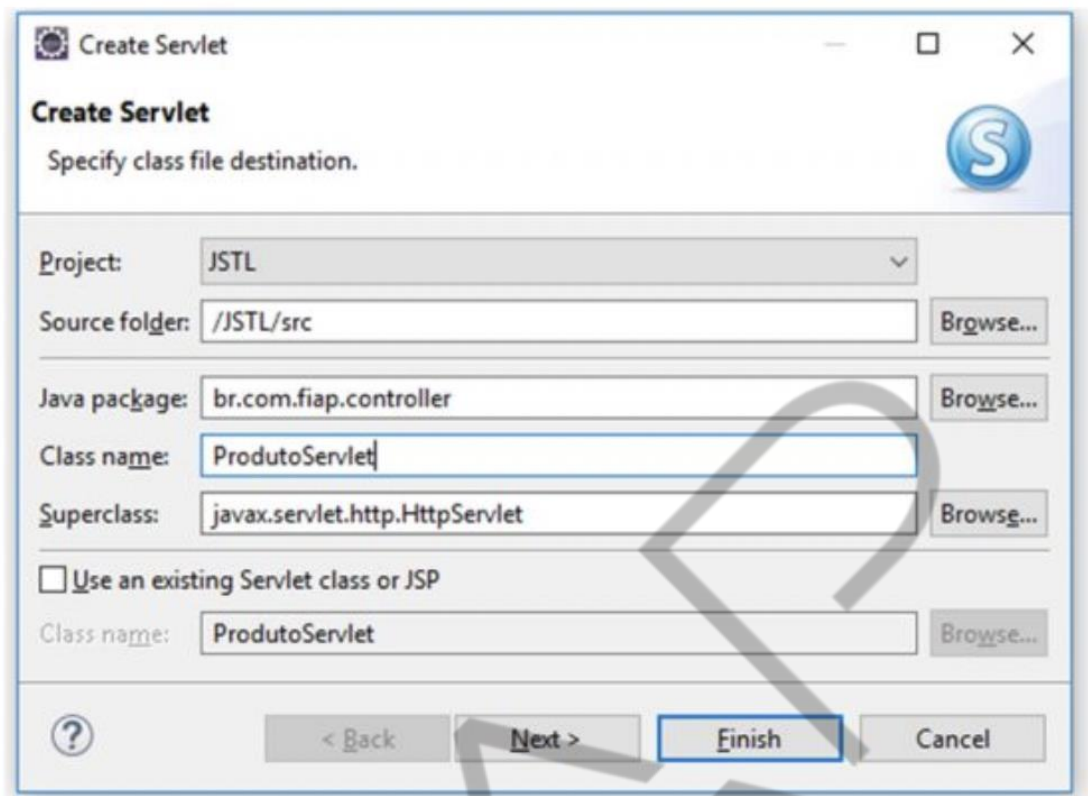


Figura 21 – Criação da classe *Servlet* – Parte 2
Fonte: Elaborado pelo autor (2017)

Para agrupar as informações do produto, precisamos de uma classe *Java Bean*. Dessa forma, vamos criar uma classe “Produto”. Clique com o botão direito do mouse na pasta src e escolha “New” > “Class” (Figura “Criação da classe *Java Bean* Produto – Parte 1”).

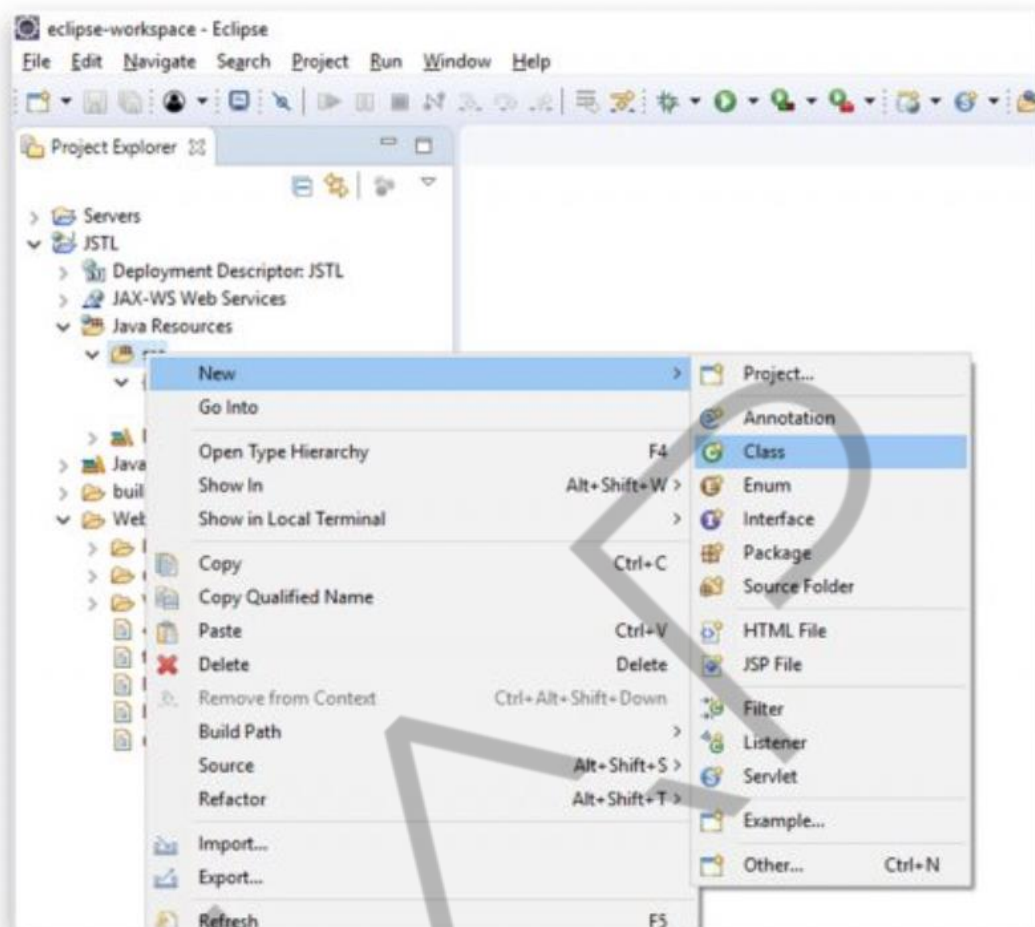


Figura 22 – Criação da classe *Java Bean* Produto – Parte 1
Fonte: Elaborado pelo autor (2017)

Para deixar o projeto bem estruturado, criaremos um novo pacote para a classe Produto, para isso, renomeie o *package* para “br.com.fiap.bean”. Depois de dar o nome à classe, finalize o processo (Figura “Criação da classe *Java Bean* Produto – Parte 2”).

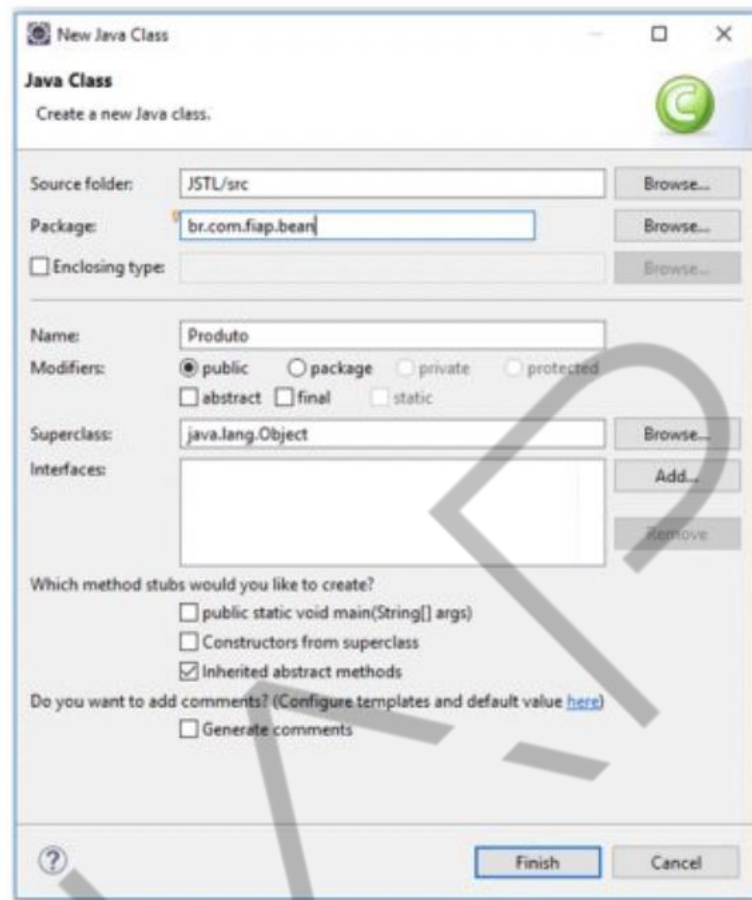


Figura 23 – Criação da classe *Java Bean* Produto – Parte 2
Fonte: Elaborado pelo autor (2017)

A classe Produto deve conter os atributos: nome, quantidade e preço. Iguais aos campos do formulário (Código-fonte “*Java Bean* Produto”).

```
package br.com.fiap.bean;

public class Produto {

    private String nome;

    private int quantidade;

    private double valor;

    public Produto() {
        super();
    }
}
```



```
public Produto(String nome, int quantidade, double
valor) {
    super();
    this.nome = nome;
    this.quantidade = quantidade;
    this.valor = valor;
}

public String getNome() {
    return nome;
}

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

public int getQuantidade() {
    return quantidade;
}

public void setQuantidade(int quantidade) {
    this.quantidade = quantidade;
}

public double getValor() {
    return valor;
}

public void setValor(double valor) {
    this.valor = valor;
}
}
```

Código-fonte 27 – *Java Bean* Produto
Fonte: Elaborado pelo autor (2017)

Para finalizar a classe, crie os *gets* e *sets* para encapsular os atributos. Crie também os construtores, um que receba como parâmetro os valores para todos os atributos e outro que seja um construtor padrão, isto é, um construtor sem parâmetros.

Agora estamos prontos para implementar a *Servlet*. Essa classe tem a responsabilidade de recuperar os dados enviados pelo usuário, enviar esses dados para outra parte da aplicação, responsável pelas regras de negócio ou persistência de dados e, por fim, devolver uma informação para o usuário. Essas são as principais responsabilidades de um controlador (*Controller*), uma camada da aplicação da

arquitetura MVC (*Model, View e Controller*). Isso foi só um *spoiler*, no próximo capítulo abordaremos essa famosa arquitetura com mais profundidade. Analise o Código-fonte “Implementação da *Servlet*”.

```
package br.com.fiap.controller;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import br.com.fiap.bean.Produto;

@WebServlet("/produto")
public class ProdutoServlet extends HttpServlet {

    private static List<Produto> lista = new
ArrayList<Produto>();

    private static final long serialVersionUID = 1L;

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

        //Recupera os parâmetros do formulário
        String nome = request.getParameter("nome");
        int qtd = Integer.parseInt(request.getParameter("quantidade"));
        double valor = Double.parseDouble(request.getParameter("valor"));

        //Cria um objeto do tipo Produto
        Produto produto = new Produto(nome, qtd, valor);

        //Adiciona o produto na lista
        lista.add(produto);

        //Mensagem de sucesso
        request.setAttribute("msg", "Produto
cadastrado!");

        request.getRequestDispatcher("cadastro-
produto.jsp").forward(request, response);
    }
}
```

```
        }  
    }  
}
```

Código-fonte 28 – Implementação da *Servlet*

Fonte: Elaborado pelo autor (2017)

Como configuramos a *action* do formulário com “produto”, precisamos colocar o mesmo valor na anotação `@WebServlet("/produto")`, e devemos, também, implementar o código no método `doPost`, já que o formulário HTML utiliza o método POST, que é mais indicado para cadastros.

Como ainda não vamos utilizar o banco de dados, vamos utilizar uma coleção (*List*) de Produtos para gravar as informações. Essa lista deve ser estática, pois deve pertencer à classe e não à instância dessa classe. Assim, todos os objetos dessa *Servlet* compartilham a mesma lista, ou seja, todos possuem a mesma informação. Isso será feito para simular o banco de dados, mas não se preocupe, logo implementaremos uma aplicação completa.

Dentro do método `doPost`, recuperamos todas as informações do formulário por meio da *request* e o método `getParameter("campo")`, lembre-se de que o valor do parâmetro do método deve ser o mesmo do nome do campo do formulário do qual você quer recuperar o valor.

Depois de recuperar os valores, instanciamos uma classe *Produto*, passando os valores em seu construtor. Com isso, podemos adicionar esse objeto na lista para “salvar no banco de dados”.

Por fim, devemos dar um *feedback* para o usuário. Vamos adicionar uma mensagem como um atributo do *request* e depois encaminhar para o usuário uma página, que nesse caso será a mesma página de cadastro de produto.

O último detalhe é ajustar a página JSP para exibir essa mensagem, mas antes vamos adicionar as bibliotecas JSTL na nossa aplicação. Copie e cole os dois *jars* dentro do diretório Web Content > WEB-INF > lib (Figura “Configuração das bibliotecas JSTL”).

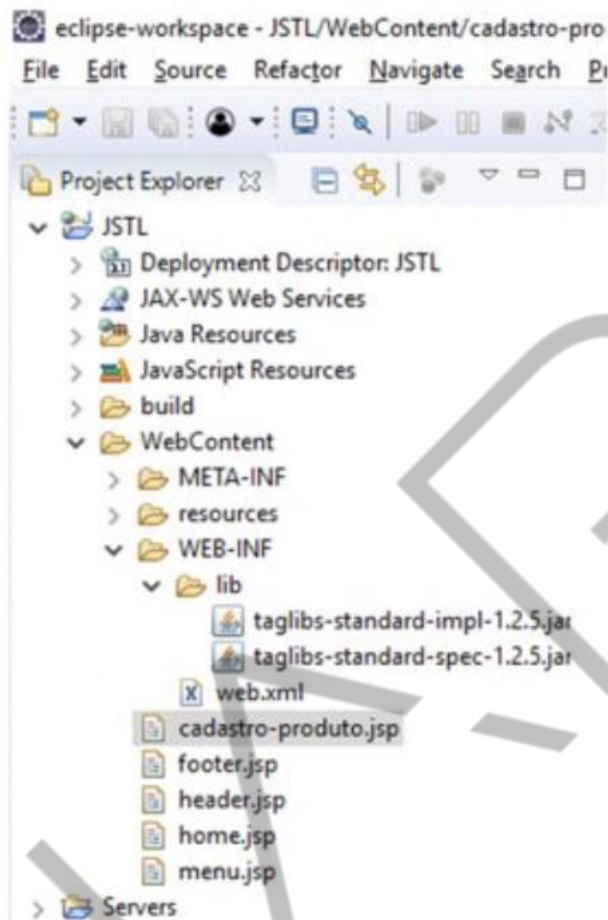


Figura 24 – Configuração das bibliotecas JSTL
Fonte: Elaborado pelo autor (2017)

Agora podemos utilizar a *tag* JSTL `<c:if>` para testar se existe uma mensagem no *request* para exibir a caixa de mensagem (Código-fonte “Exibindo a mensagem de sucesso”).

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core" %>
<!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 Produto</title>
<%@ include file="header.jsp" %>
</head>
<body>

<%@ include file="menu.jsp" %>
<div class="container">
<h1>Cadastro de Produto</h1>

<c:if test="${not empty msg}">
<div class="alert alert-success">
    ${msg}
</div>
</c:if>

<form action="produto" method="post">
<div class="form-group">
<label for="id-nome">Nome</label>
<input type="text" name="nome"
id="id-nome" class="form-control">
</div>
<div class="form-group">
<label for="id-
qtd">Quantidade</label>
<input type="text" name="quantidade"
id="id-qtd" class="form-control">
</div>
<div class="form-group">
<label for="id-valor">Valor</label>
<input type="text" name="valor"
id="id-valor" class="form-control">
</div>
<input type="submit" value="Salvar"
class="btn btn-primary">
</form>
</div>
<%@ include file="footer.jsp" %>

</body>
</html>

```

Código-fonte 29 – Exibindo a mensagem de sucesso
 Fonte: Elaborado pelo autor (2017)

Implementamos o teste para que a caixa não seja sempre visível, já que não queremos mostrar quando não existir uma mensagem. Adicionamos uma classe do

bootstrap na <div> para criar uma caixa de mensagem de sucesso. Não podemos nos esquecer de adicionar a *taglib* na página.

Agora já podemos testar! Execute a página e preencha os campos do formulário (Figura “Teste do cadastro de produto”).

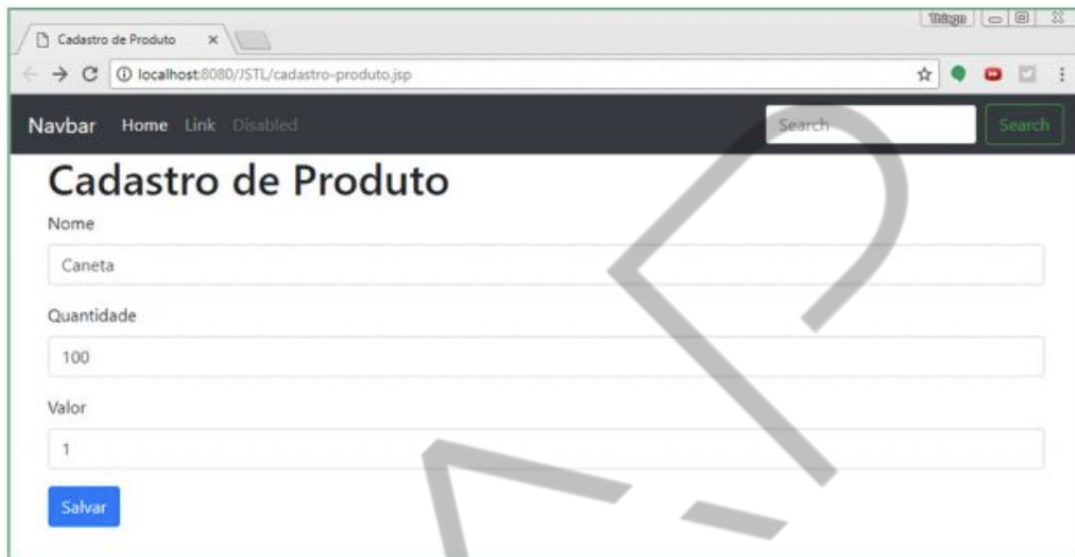


Figura 25 – Teste do cadastro de produto
Fonte: Elaborado pelo autor (2017)

Podemos ver a mensagem de sucesso após enviar os dados para o servidor, clicando no botão Salvar (Figura “Teste da mensagem de sucesso”).

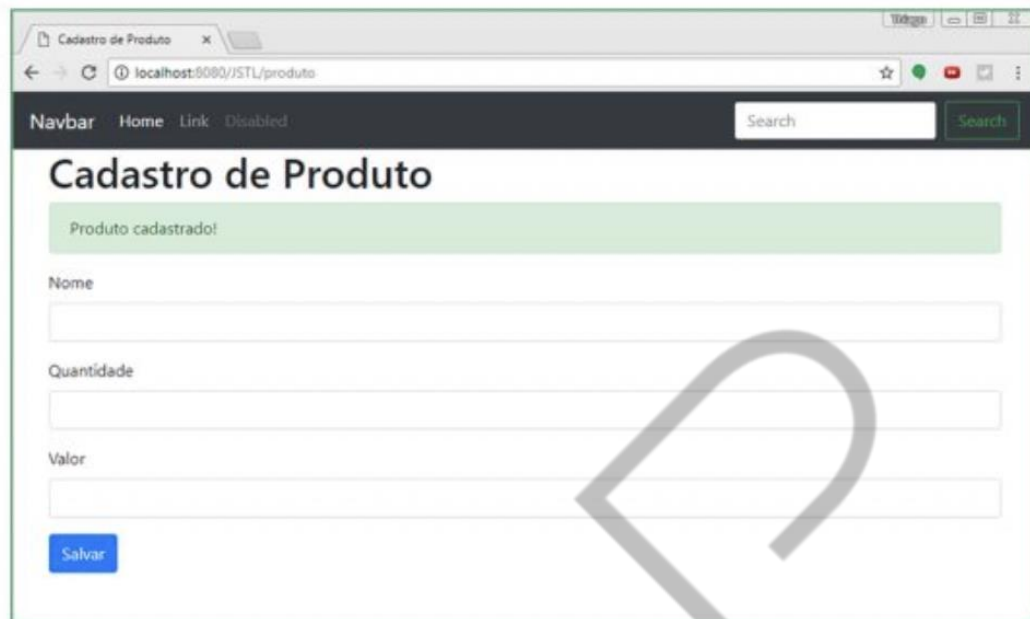


Figura 26 – Teste da mensagem de sucesso
Fonte: Elaborado pelo autor (2017)

Cadastro realizado! Vamos desenvolver a funcionalidade de listagem de produtos. O primeiro passo é ajustar a *Servlet*, vamos utilizar a mesma classe, *ProdutoServlet*. Implementaremos o método *doGet*, pois a *Servlet* será acionada por meio de um link e não de um formulário.

A *Servlet* precisa enviar a lista para a tela JSP e depois encaminhar o usuário para a página de listagem. São duas linhas de código! Observe o Código-fonte “Ajuste na *Servlet* para a funcionalidade de listagem”.

```
@Override
protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    request.setAttribute("produtos", lista);
    request.getRequestDispatcher("lista-
    produto.jsp").forward(request, response);
}
```

Código-fonte 30 – Ajuste na *Servlet* para a funcionalidade de listagem
Fonte: Elaborado pelo autor (2017)

A página JSP deve se chamar “lista-produtos.jsp” e o nome do atributo com a lista é “produtos”. Agora, vamos criar a página JSP dentro do diretório Web Content. O Código-fonte “Página JSP de listagem de produtos” apresenta a implementação da página.

```
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core" %>
<!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 Produto</title>
<%@ include file="header.jsp" %>
</head>
<body>

<%@ include file="menu.jsp" %>
<div class="container">
<h1>Produtos</h1>
<table class="table table-striped">
<tr>
<th>Nome</th>
<th>Quantidade</th>
<th>Valor</th>
</tr>
<c:forEach items="${produtos}" var="p">
<tr>
<td>${p.nome}</td>
<td>${p.quantidade}</td>
<td>${p.valor}</td>
</tr>
</c:forEach>
</table>
</div>
<%@ include file="footer.jsp" %>

</body>
</html>
```

Código-fonte 31 – Página JSP de listagem de produtos
Fonte: Elaborado pelo autor (2017)

Utilizamos o *include* para adicionar o menu, css e *scripts* na página. Criamos uma tabela HTML com uma classe do *bootstrap*, para dar estilo à tabela. Para exibir

todos os itens cadastrados, utilizamos a *taglib* `<c:forEach>`. Para utilizar essa *tag*, não se esqueça de adicionar a *taglib* na página.

A *tag* `<c:forEach>` recebe a lista no atributo **itens**, por isso utilizamos a EL (*Expression Language*) com o nome do atributo que foi adicionado na *Servlet* (produtos). O atributo **var** define o nome da variável que irá receber cada item da lista. Dessa forma, basta utilizar a variável para acessar os atributos do produto.

Para finalizar, não podemos simplesmente executar a página como fizemos com o cadastro. Precisamos primeiro executar a *Servlet*, pois é ela que envia os produtos cadastrados para a página JSP exibir. Assim, podemos testar a funcionalidade de listagem executando o projeto e ajustando a URL com o endereço `http://localhost:8080/JSTL/produto`.

Dessa forma, o browser envia uma requisição com o método GET para a nossa *Servlet*, que devolve a página JSP junto com a lista de produtos. Para que não seja preciso sempre modificar a URL para acessar a página, vamos arrumar os links da barra de navegação. Assim, abra o JSP de menu e ajuste os links, conforme o Código-fonte “Página JSP de listagem de produtos”.

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" href="home.jsp">FIAP</a>
  <button class="navbar-toggler" type="button" data-
toggle="collapse" data-target="#navbarSupportedContent" aria-
controls="navbarSupportedContent" aria-expanded="false" aria-
label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="collapse navbar-collapse"
id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item">
        <a class="nav-link" href="cadastro-
produto.jsp">Cadastro</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="produto">Produtos</a>
      </li>
    </ul>
    <form class="form-inline my-2 my-lg-0">
      <input class="form-control mr-sm-2" type="text"
placeholder="Search" aria-label="Search">
    </form>
  </div>
</nav>
```

```
<button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</button>
</form>
</div>
</nav>
```

Código-fonte 32 – Página JSP de listagem de produtos
Fonte: Elaborado pelo autor (2017)

Observe que o link para a página de cadastro é o próprio nome da página JSP, porém, para a página de listagem, é o valor configurado na *Servlet*, no `@WebServlet("/produto")`.

Agora é só testar! Cadastre alguns produtos e depois teste a listagem! O resultado da listagem é apresentado na Figura “Teste da listagem de produto e menu”.

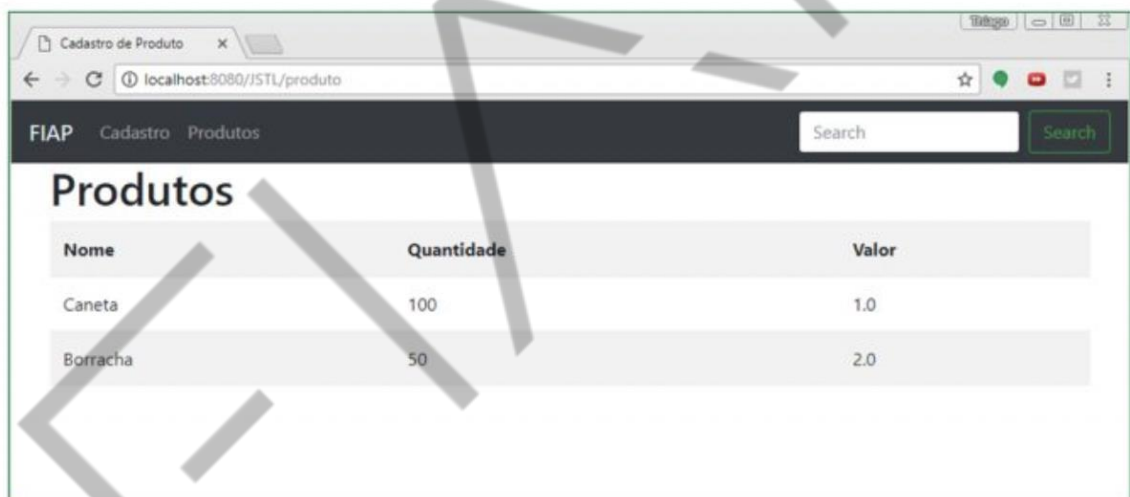


Figura 27 – Teste da listagem de produto e menu
Fonte: Elaborado pelo autor (2017)

Por enquanto, estamos utilizando uma lista estática para simular o banco de dados, isso significa que as informações ficam em memória, e caso o servidor seja atualizado ou reinicializado, as informações serão perdidas.

Mas não se preocupe! No próximo capítulo desenvolveremos uma aplicação completa, com banco de dados e a arquitetura MVC! Então, pratiquem e fiquem preparados para esse novo desafio!

REFERÊNCIAS

TUTORIALS POINT. **JSP – Expression Language (EL)**. [s.d.]. Disponível em: <https://www.tutorialspoint.com/jsp/jsp_expression_language.htm>. Acesso em: 13 set. 2021.

TUTORIALS POINT. **JSP – Standard Tag Library (JSTL) Tutorial**. [s.d.]. Disponível em: <https://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm>. Acesso em: 13 set. 2021.

EXEMPLO

GLOSSÁRIO

JSTL	Acrônimo de <i>JavaServer Pages Standard Tag Library</i> .
TagLibs	O mesmo que <i>JavaServer Pages Standard Tag Library</i> .
EL	<i>Expression Language</i> , linguagem baseada em expressões ativas nas páginas JSP.
Scriptlets	Linguagem utilizada para escrever código Java em páginas JSP através dos marcadores <code><% %></code> .
request	Ação realizada pelo usuário ao submeter informações ao servidor e do servidor ao devolver informações ao usuário.
JavaBeans	Classe Java com atributos e suas operações.
beans	Classe Java com atributos e suas operações.
get/set	Operações da classe Java para interação com os atributos.
tags	Marcações do tipo XHTML/XML/HTML que são traduzidas pelo servidor e transformadas em código Java.
Apache	Nome de uma fundação de desenvolvimento de software, no qual diversos outros grupos pertencem ao primeiro. A fundação é famosa por possuir o desenvolvimento dos melhores servidores web do planeta.

Jakarta	Nome de grupo da Fundação Apache, que é responsável pelo desenvolvimento das JSTL.
<i>jars</i>	Um JAR (Java ARchive) é um formato de arquivo de pacote normalmente usado para agregar muitos arquivos de classe Java e metadados e recursos associados (texto, imagens etc.) em um arquivo para distribuição.
IDE	Um ambiente de desenvolvimento integrado é um aplicativo de software que fornece instalações completas para programadores de computador desenvolverem software. Uma IDE normalmente consiste em um editor de código-fonte, ferramentas de automação de compilação e um depurador.