

Conteúdo

Introdução a Java Web.....	1
O Protocolo HTTP.....	2
A Resposta HTTP.....	3
Noções de HTML.....	4
Parágrafos.....	5
Títulos.....	5
Listas.....	6
Lista não ordenada.....	6
Listas ordenadas.....	6
Hyperlinks e Âncoras.....	7
Tabelas.....	8
Formulários.....	9
Principais atributos do marcador <FORM>.....	9
Marcadores relativos aos campos de formulário.....	9
Páginas JSP – Java Server Pages.....	12
Preciso Compilar uma Página Java Server Pages?.....	12
Java Server Pages são Servlets?.....	12
Como o Servlet Container Saberá que Alterei o Arquivo?.....	12
A Configuração do Arquivo web.xml.....	13
A Estrutura do Java Server Pages.....	13
Scriptlet ou Scripting.....	14
Expression Language.....	15
Classes em JSP.....	15
Parâmetros.....	15
Ações-Padrão JSP.....	16
Diretivas JSP.....	16
Necessidade do JavaScript.....	17

Como colocar Funções de JavaScript em HTML.....	18
Validação de formulários	18
Extra- Validando por JSP (JavaScript vs. Java).....	19
Extra- Validação completa de Email com Javascript e Expressão Regular.....	22
Extra- Validar CPF via JavaScript	22
Exercícios Complementares de fixação:.....	23
Exercícios de Pesquisa:.....	25
Servlets	26
O Ciclo de Vida de um Servlet	28
Exemplo de processamento de formulários usando um servlets	29
A Classe HttpServlet	31
Criando um Servlet que Trabalha com o Método POST.....	31
Trabalhando com o Método GET	32
Recuperando Strings de Consulta com Servlets.....	33
Outros Métodos Muito Uteis da Interface HttpServletRequest (Extra):.....	34
Praticando os Métodos	34
Varrendo um Formulário	36
Cookies, Sessões e Java Beans	39
Formas de modificar o escopo de um parâmetro.	39
Cookies.....	43
Criando Cookies.....	43
Recuperando um Cookie	44
Sessões.....	45
Terminando uma Sessão	48
Gerenciando uma Sessão	49
Descobrimo a Criação e o Último Acesso.....	50
JavaBeans – um estudo inicial.....	52
TagLibs, o JSTL.....	54
Exercícios Complementares de fixação:.....	56

Exercícios de Pesquisa:.....	56
Banco de dados	58
JDBC (Java Data Base Connectivity)	58
Exemplo Passo a Passo: CRUD Básico de Produtos	59
Padrão DAO.....	65
Projeto Loja ITTraining	73
Configurando páginas de erro com Servlets	83
Validando os dados e prevendo erros.....	87
Exercício do projeto	88
E quando as tabelas estão relacionadas?	89
Exercícios Complementares de Fixação:	90
Persistência de Objetos com JPA, EJB 3.0 e Hibernate	92
Projeto Escola – utilizando JPA, EJB e Hibernate para um desenvolvimento produtivo e veloz	93
Extra - Relacionamento n:n, um exemplo	115
Exercícios Complementares de Fixação:	117
Exercícios de Pesquisa:.....	117
MVC (Model - View – Controller)	119
Framework Baseado em Componentes - JSF – Java Server Faces.....	121
Trabalhando com JavaServer Faces no NetBeans	121
Regras de navegação:	123
Componentes básicos.....	124
Managed Bean.....	128
Detalhando uma EL	130
Value binding.....	130
Method binding.....	130
Mensagens.....	131
Converters.....	132
Validators.....	132

Eventos e Listeners.....	133
Value-change events.....	133
Action events	133
Crud com Java Server Faces	134
Exercícios Complementares de Fixação:	145
Exercícios de Pesquisa.....	145
RIA – Rich Internet Applications – Aplicações ricas para Internet.	147
Frameworks RIA para JSF 2.0 – PrimeFaces.....	149
Botões e mensagens	150
Formulários	152
Menus.....	154
Painéis e Efeitos.....	159
Tabelas.....	162
Temas	163
Considerações finais.....	164
Exercícios Complementares de Fixação	164
Exercícios de Pesquisa:.....	164
Anexo 2 – Configurando o acesso ao MYSQL no NetBeans.....	166
Anexo3 – Noções de SQL.....	167
Criando Tabelas:	167
Inserir registros:.....	167
Atualizar registros:	167
Deletar registros:	167
Visualizar registros:.....	167

Introdução a Java Web

Sistemas com clientes baseados em navegadores Internet (browsers) exibem muitas vantagens sobre as aplicações C/S (Cliente/Server) tradicionais, tais como o número virtualmente ilimitado de usuários, procedimentos de distribuição simplificada (instalação e configuração apenas no Servidor), operação em múltiplas plataformas e possibilidade de gerenciamento remoto. Mas tais vantagens requerem a geração dinâmica de conteúdo para web.

Atualmente podemos encontrar aplicações web em todos os lugares, em todas as empresas, de vários tipos diferentes como, por exemplo: blogs, fotoblogs, sites de vendas de todo e qualquer tipo de produtos, pedidos/reservas de passagens de ônibus, avião, hotel e até aplicativos de alto risco e desempenho como aplicações corporativas, bancárias, leilões etc. e etc.

Entrando na parte técnica da coisa, aplicativos web são por natureza APLICAÇÕES DISTRIBUÍDAS. Ou seja, são pedaços de um mesmo programa que de forma combinada executam em diferentes lugares em máquinas separadas denominados clientes e servidores, interconectados através de uma rede comum.

JSP é uma tecnologia JavaEE (java Enterprise Edition) destinada à construção de aplicações para geração de conteúdo dinâmico para web, tais como HTML, XML, e outros, oferecendo facilidades de desenvolvimento. Em nossas aulas iremos aprender a construir páginas JSP para processar dados de formulários HTML, também iremos abordar um pouco sobre Servlets e como construir páginas JSP utilizando eles; Utilizar uma linguagem dinâmica chamada JavaScript, para validar e deixar nossas aplicações mais robustas e confiáveis; Escopo de parâmetro (até onde eles são visíveis para as aplicações); Enterprise JavaBeans e TagLibs; Conectar nossas páginas JSP com o Banco de Dados MySQL através do JDBC e do padrão de projetos DAO; JavaServerFaces, a tecnologia do JavaEE 6.

Esta apostila foca em utilizar recursos para desenvolvimento Web na tecnologia Java, muitas pessoas acabam se deparando com um absurdo de frameworks e na hora de escolher é uma dificuldade, temos basicamente dois tipos de frameworks web, aqueles que se baseiam em Ações, como o Struts 1 e 2, WebWork, VRaptor, dentre outros e temos os que se baseiam em componentes como o Java Server Faces, Click, Wicket e mais alguns que estão surgindo a cada dia.

Por onde o desenvolvedor iniciante deve começar? Qual é o melhor?

Os Action Based ainda são muito encontrados nas empresas, no caso o mais famoso deles o Struts, ainda existem inúmeros sistemas legados que utilizam este framework, a maior dificuldade do iniciante é descobrir como desenvolver em cima do framework.

Por exemplo:

Trabalhando com o Struts, o desenvolvedor deve saber que tem que criar um Bean, um Form, uma página para exibição dos dados. Deve entender também que o Form é que faz a comunicação entre suas regras de negócio e a página, deve conhecer todas as burocracias do Struts para navegação, comunicação e afins, isto é muito difícil, ainda mais para alguém que acabou de conhecer a tecnologia.

Os frameworks baseados em componentes, vieram para simplificar isto, já que o desenvolvedor tem em mente algo como Desktop. Com eventos simples e fáceis de tratar, temos a maior preocupação na regra de negócio do cliente e não em validadores de tela e preocupações para manter estado dos objetos do formulário que se perdem. A curva de aprendizado é muito maior, simples e produtiva com os frameworks baseados em componentes, tornando o desenvolvedor livre de preocupações com telas.

O Protocolo HTTP

Existe um conjunto de protocolos que auxiliam o mapeamento de ações dos usuários do lado cliente no uso de aplicações Web .

A Web é uma aplicação cliente/servidor em grande escala, onde o cliente (um navegador Web ou um programa FTP) se conecta ao servidor usando um protocolo. O mais comum desses protocolos é o HTTP (Hypertext Transfer Protocol), onde em uma requisição do browser é devolvido pelo servidor textos e imagens. Esse protocolo trabalha com pedidos e respostas

O protocolo HTTP começa com uma solicitação, que por sua vez o devolve com uma resposta. A seguir você tem as solicitações desse protocolo:

- **GET** - Solicita ao servidor um recurso chamado de solicitação
- **URI** - Os parâmetros da solicitação devem ser codificados nessa solicitação, para que o mesmo os entenda. Este é o método mais usado, pois é a forma como o browser chama o servidor quando você digita um URL para que ele o recupere.
- **POST** - Embora similar ao GET, o POST contém um corpo nos quais seus parâmetros de solicitação já são codificados. O mais frequente uso desse método é na submissão de formulários (X)HTML.
- **HEAD** - Similar ao método GET, o servidor apenas retoma a linha de resposta e os cabeçalhos de resposta.
- **PUT** - Esse método permite o envio de arquivos para o servidor Web.
- **DELETE** - Permite a exclusão de documentos dentro do servidor Web.
- **OPTIONS** - É possível fazer uma consulta de quais comandos estão disponíveis para um determinado usuário.
- **TRACE** - Permite depurar as requisições, devolvendo o cabeçalho de um documento

A Resposta HTTP

A resposta HTTP tem uma linha de status (como a de solicitação) e cabeçalhos de resposta, assim como o corpo de resposta opcional. No caso de você entrar em um site e ele estiver disponível, você teria a seguinte resposta:

```
HTTP/1.1 200 OK
Date: Sat, 15 Apr 2006 18:21:25 GMT
Content-Type: text/html; charset=ISO-8859-1
Content-Length: 2541
<html><head>
<title> Integrator Technology and Design </title>
</head>
<body>...
```

FTP - Sigla para File Transfer Protocol, onde é muito usado para transmissão de arquivos para um servidor. Fornece os principais comandos para troca de arquivos.

SMTP - Sigla para Simple Message Transfer Protocol, e fornece os comandos necessários para envio de mensagens a um servidor de e-mail.

POP - Sigla para Post Office Protocol, onde permite que um cliente acesse e manipule mensagens de correio eletrônico disponíveis em um servidor.

IMAP - Sigla para Internet Message Access Protocol e permite que um cliente acesse e manipule mensagens de correio eletrônico disponíveis em um servidor, assim como ocorre no protocolo POP.

Noções de HTML

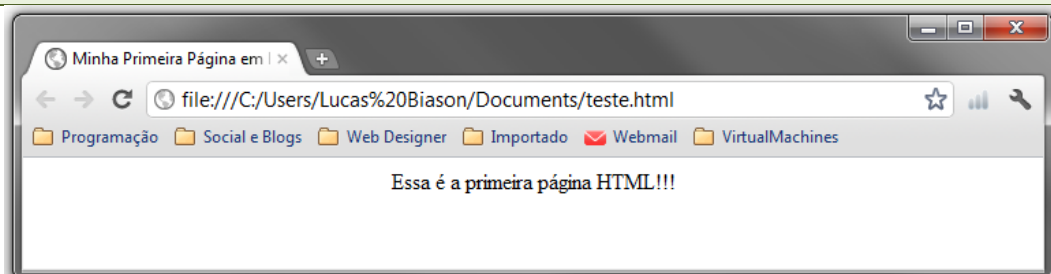
HTML (Hyper Text Markup Language - Linguagem de Marcação de Hipertexto) é uma linguagem para descrever páginas da internet. É a uma linguagem considerada a base de todas as outras linguagens de desenvolvimento de projetos para WEB. Com ela você pode compartilhar fotos, vídeos, músicas, textos e fazer muitas coisas. HTML não é uma linguagem de programação, é uma linguagem de marcação, composta por um conjunto de "tags" que são palavras-chaves cercadas pelos símbolos < >, como <html>. Normalmente são usadas em pares, como (indica o início de um bloco pertencente a tag) e (indica o fim desse bloco).

Tags essenciais para nosso uso:

```
<html> - indica que é uma página html
  <body> - indica o começo do corpo da página
    <h1> ...texto... </h1> - define um título
    <p> ...texto... </p> - define um parágrafo.
    <a href="http://www.google.com.br">um link</a>
    <br /> - quebra de linha
    <hr /> - cria uma linha na página
  </body> - indica o fim do corpo da página
</html> - indica o fim de uma página html.
```

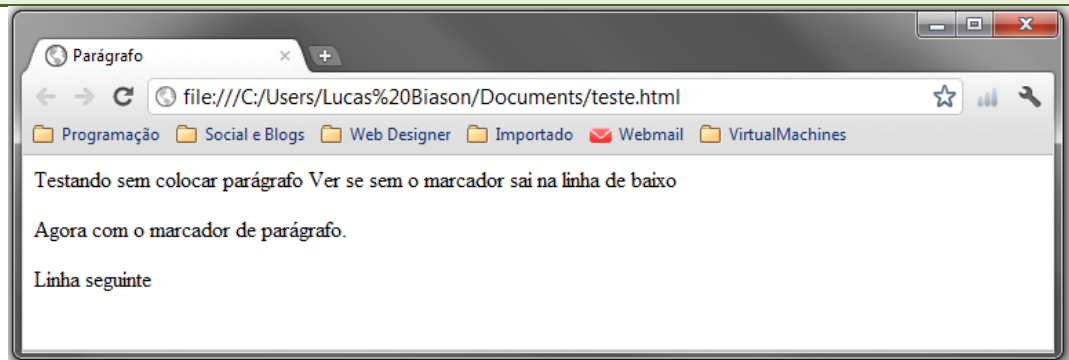
Exemplo:

```
<html>
  <head>
    <title>Minha Primeira Página em HTML</title>
  </head>
  <body>
    <p align="center">Essa é a primeira página HTML!!</p>
  </body>
</html>
```



Parágrafos

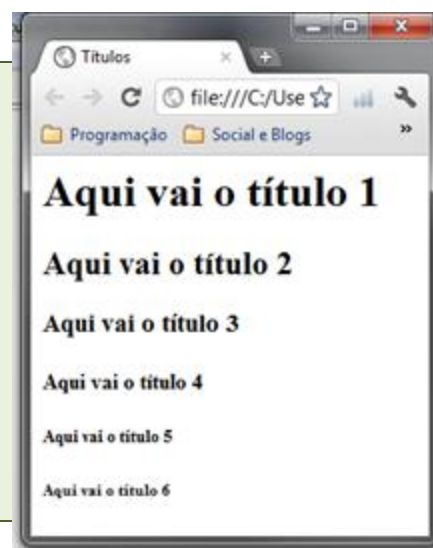
```
<html>
  <head>
    <title>Parágrafos</title>
  </head>
  <body>
    Testando sem parágrafo
    Ver como fica sem o marcador para linha de baixo.
    <p>
      Agora com o marcador de parágrafos.
    </p>
    Linha seguinte.
  </body>
</html>
```



Títulos

Identifica títulos, usados para dividir seções do texto. Existem 6 níveis de títulos. Numerados de H1 a H6, que são exibidos em fonte maior que a fonte normal. Os marcadores de título podem ser alinhados.

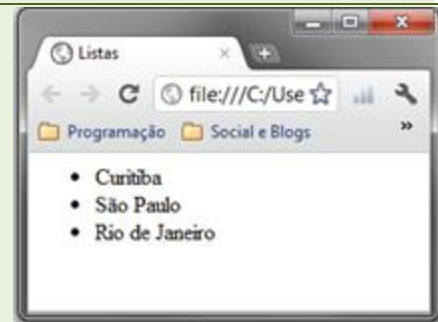
```
<html>
  <head>
    <title>Títulos</title>
  </head>
  <body>
    <h1>Aqui vai o título 1</h1>
    <h2>Aqui vai o título 2</h2>
    <h3>Aqui vai o título 3</h3>
    <h4>Aqui vai o título 4</h4>
    <h5>Aqui vai o título 5</h5>
    <h6>Aqui vai o título 6</h6>
  </body>
</html>
```



Listas

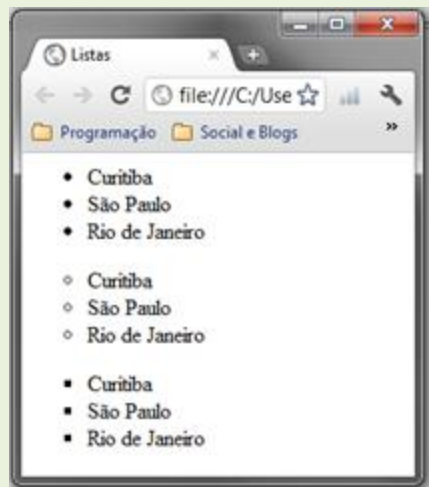
Lista não ordenada

```
<html>
  <head>
    <title>Listas</title>
  </head>
  <body>
    <ul>
      <li>Curitiba</li>
      <li>São Paulo</li>
      <li>Rio de Janeiro</li>
    </ul>
  </body>
</html>
```



Atributos de UL: Type: Indica qual será o símbolo usado para demarcar cada elemento da lista. Pode ser: “disc”, “circle” ou “square”. (“Disc” é padrão, caso não especifique o type).

```
<html>
  <head>
    <title>Listas</title>
  </head>
  <body>
    <ul type="disc">
      <li>Curitiba</li>
      <li>São Paulo</li>
      <li>Rio de Janeiro</li>
    </ul>
    <ul type="circle">
      <li>Curitiba</li>
      <li>São Paulo</li>
      <li>Rio de Janeiro</li>
    </ul>
    <ul type="square">
      <li>Curitiba</li>
      <li>São Paulo</li>
      <li>Rio de Janeiro</li>
    </ul>
  </body>
</html>
```



Listas ordenadas

```
<html>
  <head>
    <title>Listas</title>
  </head>
  <body>
```



```
<ol>
  <li> Curitiba </li>
  <li> São Paulo </li>
  <li> Rio de Janeiro </li>
</ol>
</body>
</html>
```

Hyperlinks e Âncoras

Uma referência de hipertexto é algo muito simples. Consiste em uma âncora e um endereço, ou URL. A âncora é o texto ou a imagem sobre o qual o usuário dá um clique para ir até outro lugar. O endereço indica a localização do documento que o navegador irá carregar quando o usuário der um clique sobre a âncora.

Em HTML, uma âncora pode ser um texto ou um gráfico. De modo geral, as âncoras de texto aparecem em estilo sublinhado e em cor diferente do texto normal nos navegadores.

Como Criar Âncoras? Qualquer texto pode ser uma âncora em HTML, independente do seu tamanho ou da sua formatação. Uma âncora pode consistir em algumas letras, palavras, ou mesmo linhas de texto.

O formato de um par âncora-endereço é simples.

```
<A HREF="URL">Texto da âncora</A>
```

A letra A na tag <A HREF> significa "âncora" (anchor) e HREF quer dizer "referência de hipertexto" (hypertext reference). Tudo que se encontra entre as tags <A HREF> e representa o texto da âncora, que aparece sublinhado ou em negrito, dependendo do navegador. Observe o exemplo a seguir:

```
<A HREF="comprar.htm">Clique aqui para comprar!</A>
```

Podem ser utilizados outros códigos de formatação em conjunto com as âncoras de hipertexto. Por exemplo, para fazer uma âncora de hipertexto aparecer no estilo de título nível 4, você escreveria:

```
<A HREF="URL"><H4>Texto da âncora</H4></A>
```

A ordem dos pares de tags não é relevante. Também seria possível escrever:

```
<H4><A HREF="URL">Texto da âncora</A></H4>
```

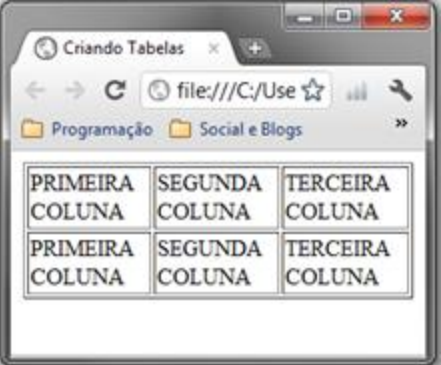
Com hyperlinks e âncoras, também é possível direcionar o visitante para posições específicas do próprio documento em questão ou de um outro documento ainda não visitado. Isso é muito comum em páginas que contêm índices. Por exemplo, você está criando uma página que irá conter dicas de culinária e informática. Não é preciso dizer que os assuntos são totalmente distintos, logo, quem curte informática, provavelmente não está interessado em ler sobre culinária, então é possível criar um índice para que se o visitante clicar em “informática”, o navegador exibe imediatamente as dicas de informática, mesmo que as dicas de culinária venham antes do que as de informática, na página de dicas.

Tabelas

Tabelas são definidas usando a tag <table>. Uma tabela é dividida em colunas e linhas, sendo uma célula a intersecção entre uma linha e uma coluna. As linhas são definidas pela tag <tr> (table row). Cada linha é dividida em células definidas pela tag <td> (table data). Em cada célula é possível representar: textos, links, imagens, listas, formulários, tabelas, etc. Exemplo:

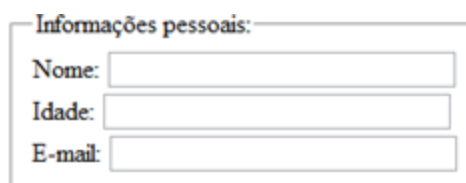
```
<html>
  <head>
    <title>Listas</title>
  </head>

  <body>
    <table border=1>
      <!-- Inicia a tabela e coloca uma borda de espessura igual a 1 -->
      <tr>
        <!-- Cria a primeira linha -->
        <td>Primeira Coluna</td>
        <td>Segunda Coluna</td>
        <td>Terceira Coluna</td>
      </tr>
      <!-- Fecha a primeira linha -->
      <tr>
        <!-- Cria a segunda linha -->
        <td>Primeira Coluna</td>
        <td>Segunda Coluna</td>
        <td>Terceira Coluna</td>
      </tr>
      <!-- Fecha a segunda linha -->
    </table>
    <!-- Encerra a tabela -->
  </body>
</html>
```



Formulários

Formulários HTML são usados para passar informações do cliente para o servidor. Um formulário pode conter elementos de entrada como: text fields, checkboxes, radio-buttons, submit buttons, select lists, textarea, fieldset, legend e labels.



Informações pessoais:

Nome:

Idade:

E-mail:

Principais atributos do marcador <FORM>

```
<FORM NAME="nome do formulário" METHOD="valor" ACTION="tratador do formulário">elementos do formulários</FORM>
```

NAME: Especifica o nome do formulário. Útil em casos de validação dos campos, por exemplo, quando se utiliza uma linguagem de scripts como JavaScript.

METHOD: Método que define como os dados serão transmitidos para o programa que irá processá-los. Devem ter valores GET ou POST, sendo que a diferença entre estes dois valores está no modo como os dados são empacotados. Normalmente o programa que será utilizado para processar o formulário já especifica o valor para o atributo METHOD.

ACTION: Indica o endereço do programa que receberá os dados do formulário.

Marcadores relativos aos campos de formulário

Utilizado para indicar um novo campo de formulário é diferenciado pelo seu tipo:

```
<INPUT TYPE="tipo" NAME="nome" VALUE="valor" SIZE="tamanho em pixels" />
```

MAXLENGTH: "tamanho máximo em pixel">

INPUT: Especifica um campo de entrada de dados.

TYPE: Atributo mais importante do marcador por definir o tipo de elemento a ser inserido no formulário.

NAME: Nome do elemento, útil em casos de validação de campos, por exemplo, quando se usa uma linguagem de scripts como JavaScript.

VALUE: Valor que pode ser predefinido para o campo.

MAXLENGTH: Comprimento máximo do campo.

CHECKED: Em casos de caixas de checagem, predefine como checada.

Text Fields

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

Exemplo:

Nome:
 Sobrenome:

Password Fields

Senha:
`<input type="password" name="senha" />`

Exemplo:

Senha:

Radio Buttons

`<input type="radio" name="sexo" value="masculino"/>Masculino
`
`<input type="radio" name="sexo" value="feminino"/>Feminino`

Exemplo:

☐ Masculino
☐ Feminino

Checkboxes

`<p>Eu tenho:</p>`
`<input type="checkbox" name="veiculo" value="Bicicleta"/>Bicicleta
`
`<input type="checkbox" name="veiculo" value="Carro"/>Carro`

Exemplo:

Eu tenho:
☐ Bicicleta
☐ Carro

Submit Button

`<form name="input" action="algumapagina.html" method="post">`
 Usuario:
`<input type="text" name="usuario" />`
`<input type="submit" value="Enviar" />`
`</form>`

Ex:

Usuario:

Select list

`<select name="carros">`
`<option value="volvo">`
Volvo`</option>`
`<option value="fiat">`
Fiat`</option>`
`<option value="audi">`
Audi`</option>`
`</select>`

Ex:

Textarea

`<textarea rows="10" cols="30">`
 Linha 1
 Linha 2
 Linha 3
`</textarea>`

Ex:



Módulo 01:

Páginas JSP

Páginas JSP – Java Server Pages

Java Server Pages são páginas Java embebidas em HTML. Dessa forma a página dinâmica é gerada pelo código JSP. A primeira vez que uma página JSP é carregada pelo container JSP, o código Java é compilado gerando um Servlet que é executado. As chamadas subsequentes são enviadas diretamente ao Servlet, não havendo mais a recompilação do código Java.

Preciso Compilar uma Página Java Server Pages?

Uma das grandes vantagens de desenvolver em JavaServer Pages é que você não precisa compilar seu código. Você cria a página e a coloca pra rodar no Servlet Container. Caso precise alterar, altere e pronto.

Java Server Pages são Servlets?

A página JSP é um arquivo de script interpretado inicialmente e depois compilado em um Servlet (Visto mais a frente). O arquivo é pré-compilado numa classe Java quando acontecer o primeiro chamado desse arquivo. Você pode visualizar o Java criado, assim como o compilado no Tomcat em:

`$CATALINA_HOME/work/Catalina/localhost/SiteJSP/org/apache/jsp`

Como o Servlet Container Saberá que Alterei o Arquivo?

O compilador verifica a data de alteração do arquivo que contém a página JSP e caso essa data se modifique, o processo de compilação é executado novamente para garantir que as alterações feitas na página sejam visíveis para os usuários da aplicação.

Devido a todo esse processo de compilação / recompilação, após a modificação, assim como na criação, de uma página JSP, é sempre mais lento que os acessos seguintes.

Por trás de uma página JSP existe um Servlet especial, chamado Page Compiler, que intercepta requisições direcionadas a recursos com extensão .jsp.

A Configuração do Arquivo web.xml

No caso da criação de arquivos JSP, você adiciona novos elementos no arquivo web.xml para ter a chamada de um arquivo inicial, quando sua aplicação for chamada:

```
<?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">
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

Note a adição do elemento **<welcome-file-list/>** que contém um sub-elemento **<welcome-file/>**, indicando um ou mais nomes de arquivos que deverão ser carregados automaticamente caso você não digite o nome do arquivo na chamada da aplicação no navegador. Como você já deve ter notado, o primeiro da lista é o chamado para ser a página default, caso não seja encontrada, a segunda da lista é o arquivo inicial e assim por diante até o fim.

Ao executar essa aplicação no Servlet Container, você notará que a chamada inicial é lenta, isso graças a sua compilação. Depois disso, na segunda chamada em diante fica consideravelmente mais rápido, como já dito anteriormente.

A Estrutura do Java Server Pages

Em páginas dinâmicas escritas em JSP você tem as tags de abertura **<%** e fechamento **%>**, como já dito anteriormente, para se adicionar o comando desejado.

O JSP é composto de cinco elementos:

Scripting: possibilitam o uso de código Java capaz de interagir com outros elementos do JSP e do conteúdo do documento, por exemplo, Scriptlet.

```
<% ...algum comando Java ....%>
<%= alguma expressão em Java que retorne algum valor%>
```

Ações: tags pré-definidas incorporadas em páginas JSP, cujas ações em geral se realizam com base nas informações do pedido enviado ao servidor.

```
<jsp:include page="url_da_pagina"> - Ação de Inclusão  
<jsp:forward page="url_da_pagina"> - Ação de encaminhamento
```

Diretivas: mensagens enviadas ao web container para especificar configurações de página, inclusão de recursos e bibliotecas.

```
<%@ page import="Java.io.*" %> - diretiva de importação  
<%@ page isErrorPage="true|false" ou errorPage="url" %> -  
direcionamento de erros  
<%@ include ... - inclusão de arquivos  
<%@ taglibs .... - inclusão de taglibs
```

Taglibs: bibliotecas de tags que podem ser criadas e usadas pelos programadores para personalizar ações e evitar a repetição de código.

```
<c:forEach: ...> </c:forEach> , <c:if ...>...</c:if>  
Taglibs...
```

Conteúdo Fixo: marcações fixas HTML ou XML que determinam uma parcela do conteúdo do documento.

```
<html>  
  <head>...</head>  
  <body>  
    Conteudo fixo  
  </body>  
</html>
```

Scriptlet ou Scripting

Um scriptlet é um pedaço de código Java embutido em um código JSP semelhante a um código HTML. O scriptlet sempre está dentro de tags <% %> ou <%= %>. Eles definem uma expressão/código Java dentro de uma Página HTML. A diferença de cada um é:

<% %>: usado para instruções java, por exemplo: <% String str = "oi"; %>

<%= %>: usado para expressões que retomam um valor a ser inserido na página HTML, por exemplo: <%= str %> (esse scriptlet irá mostrar o conteúdo de "str" onde for colocado na página html).

O uso de páginas JSP contendo scriptlets simplifica sua construção, mas duas questões a considerar:

1) a maioria dos web designers não domina programação Java e, portanto, os trechos de código são pouco amigáveis e não podem ser modificados sem auxílio de um programador;

2) muitas das ferramentas HTML não sabem lidar com código Java nem com as marcações JSP.

Para resolver esses problemas, é conveniente separar o código Java das páginas JSP, o que se pode fazer com o auxílio dos JavaBeans e das Taglibs (vistos mais a frente).

Expression Language

Similares aos scriptlets `<%= %>`. São usadas para expressões que retornem valores, JavaBeans (vistos mais a frente) e parâmetros. Sua sintaxe é: `${...}`. Facilita o acesso às informações presentes no servidor, sua sintaxe é simples, não requer associação com código Java.

Uma EL simples pode ser construída de um valor literal de tipos lógicos, inteiro, em ponto flutuante, string ou nulo. Também se pode utilizar os valores dos campos de objetos existentes nos escopos visíveis, desde que existam métodos públicos `getAlgo()` para sua obtenção. Exemplo:

```
${75}, ${"oi"}, ${cliente.idade}
```

Ela oferece uma série de vantagens sobre as scriptlets tradicionais: sintaxe mais simples e concisa, tratamento implícito de null pointers, coerção automática e o uso indireto da API Java.

Contudo, é menos flexível e realiza um conjunto limitado de ações, em geral restritas aos métodos padronizados pela especificação dos JavaBeans.

Classes em JSP

Os scriptlet nos dão poder de usar qualquer código Java numa página HTML. Desse modo, iremos trabalhar com classes e afins do mesmo modo que numa aplicação desktop (vista no módulo Java Básico). Exemplo:

```
<%@ page import="java.sql.*, travel.*" %>
<html>
  <body>
    <% String str = "Oi Mundo!"; %>
    <%= str %>
  </body>
</html>
```

Parâmetros

As páginas de uma aplicação WEB precisam trocar informações entre si. Para isso elas utilizam muitas vezes formulários. Quando se usa um formulário em HTML, se

coloca seu nome, o endereço da página receptora e o método a ser enviados os dados (GET, menos seguro, com os parâmetros visíveis e POST, mais seguro. Com os parâmetros ocultos).

```
<form name="nome" action="pagina" method="método">
    ...
</form>
```

Cada objeto do formulário – text, textbox, button, submit...entre outros – deve vir com um nome. Na página receptora, esse nome será muito importante. Nela usaremos um comando chamado “getParameter()” onde ele precisa do nome do parâmetro para retornar seu valor. É importante ressaltar que o parâmetro retornado é sempre uma String, e caso originalmente ele fossem um número, ele deve ser convertido. Veja abaixo o exemplo:

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

Uma vez capturado o parâmetro da página anterior, e atribuído seu valor a variável, seu valor já pode ser utilizado para a páginas.

Ações-Padrão JSP

As ações são elementos padronizados que permitem realizar tarefas sem a necessidade de adicionar código Java.

Ação de inclusão: permite incluir um arquivo na página onde se executa o comando.

```
<JSP:include page="url_arquivo" flush="false|true">
```

URL relativa do
arquivo

Descarregamento próprio

Ação de encaminhamento: a ação de encaminhamento <JSP:forward> é semelhante á ação include, mas em vez de induzir uma página, dirige a execução para URL indicada.

```
<jsp:forward page="url_pagina"/>
```

Diretivas JSP

As diretivas JSP destinam-se a passar informações para o web container que hospeda a página JSP. O que possibilita configurá-las e definir a linguagem de scripting, além de incluir arquivos e taglibs. Todas as diretivas são delimitadas por <%@ %> e processadas em tempo de tradução.

Configuração de página: a diretiva de configuração de página `<%@ page ...%>` serve para especificar as condições de uso da página em si, tais como seu buffer, uso de threads e comportamento em situações de erro.

Importação de classes: o atributo `import` da diretiva `<%@ page%>` permite relacionar classes e pacotes usados pela página.

```
< %@page import="java.io.*" %>
```

Direcionamento de erros: o atributo `isErrorPage` define as páginas que devem ser tratadas como páginas de erro da aplicação:

```
<%@page isErrorPage="true" %>
```

Para direcionar eventuais erros a esse tipo de página, emprega-se o atributo `errorPage` da diretiva `<%@page %>` nas demais páginas, a seguir:

```
<%@page errorPage="nomePagina.jsp" %>
```

Inclusão de arquivos: a diretiva `<%@include %>` permite incluir um documento externo na página, tal como cabeçalho, rodapé ou fragmento. Assim, diferentes páginas podem compartilhar elementos comuns, cuja alteração centraliza-se no respectivo documento incluído.

```
<%@include file="url_arquivo" %>
```

Inclusão de Taglibs: a diretiva `<%@ taglib %>` indica quais bibliotecas de tags (taglibs) serão utilizadas pela página JSP.

```
<%@ taglib prefix="prefixo" uri="uri_taglib" %>
```

Necessidade do JavaScript

Muitas vezes o usuário informa dados inválidos ou deixam campos obrigatórios vazios antes de pressionarem o botão Enviar. Isso pode causar problemas ao processar os dados.

Para assegurar que os dados sejam válidos e que todos os campos obrigatórios foram informados antes de o botão Enviar ser pressionado, você poderá utilizar JavaScript nas páginas HTML.

O JavaScript não faz parte da “família” Java, ele é um tópico pleno em si e separado. É outra linguagem de programação, muito boa, que torna as páginas mais

dinâmicas. Ela não necessita de um servidor próprio como o Java, o php, o asp entre outros. Por isso veremos JavaScript apenas como uma forma de validar os campos dos formulário. E para dar mais dinamismo as aplicações. Não há necessidade de um compilador, ou kit para o JavaScript, ele é tão nativo quanto o HTML.

Como colocar Funções de JavaScript em HTML

Antes ou dentro das etiquetas <head>...</head> de maneira interna a pagina ou em um arquivo próprio separado, de maneira externa a página, mas colocando um link para ligar a HTML e a JavaScript. Maneira Interna, usando as etiquetas SCRIPT:

```
<script language="JAVASCRIPT">
...
</script>
```

Maneira externa, ao criar um arquivo .js, faça o link para a página.

```
<script language="JavaScript" src="Lugar onde está e nome">
...</script>
```

Ex:

```
<script language="JavaScript" src="js/validacoes.js">
...</script>
```

Validação de formulários

Para tornar o código de validação acessível, deve-se criar uma função usando o comando "function". Dentro dessa função criada é onde o código irá ser colocado. Para acessar o campo do formulário é preciso chegar ate ele pela hierarquia de objetos do JavaScript:

```
Document.forms.<nome do formulário>.<nome do campo>
```

Ex:

```
Document.forms.cadastro.nome
```

Para acessar o valor do campo, usa-se: "value". Exemplo de um código de validação:

```
Function validar(){
    var nome = Document.forms.cadastro.nome;
    if(<condição de erro>){
        alert("mensagem");
    }
}
```

```
        nome.focus();  
        return;  
    }  
    document.forms.principal.submit();  
}
```

Seguindo sempre esse modelo: criamos um objeto que representa o campo do formulário (no exemplo, nome). Assim poderemos acessar o campo mais facilmente, deixando o código mais limpo. Depois disso, é realizada a verificação necessária. Caso ela atenda ao erro, damos um alerta usando o comando “alert”, focamos a atenção para o objeto, e por fim, terminamos o código. Caso ele não esteja com problema e passe sem problemas pelo “if”, o formulário poderá ser enviado usando o comando submit().

```
Function validar(){  
    var nome = Document.forms.cadastro.nome;  
    if(nome.value == ""){  
        alert("O campo nome não foi preenchido");  
        nome.focus();  
        return;  
    }  
    document.forms.principal.submit();  
}
```

Várias outras funções podem ser utilizada, desde simples verificações de campos vazios, até mesmo expressões regulares complexas. Para números, podemos usar : isNaN() , que verifica se o conteúdo do campo não é um número – is Not a Number. E também podemos utiliza length para verificar se a quantidade de letras no campo obedece o máximo ou o mínimo desejado.

Extra- Validando por JSP (JavaScript vs. Java)

Crie um novo Projeto. Na página index digite os códigos para criar um formulário pequeno onde se tem nome e idade:

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>  
<% //verificar se é a primeira vista a página ou se é o retorno  
do erro.  
    String msg = request.getParameter("msg");  
    String msgOut = "";  
    if (msg != null && (msg.equals("1") || msg.equals("2"))) {  
        msgOut = (msg.equals("1")) ? "Forneça um nome!" : "Forneça  
idade válida!";  
    }  
}
```

```
%>
<html>
  <body>
    <jsp:include page="topo.html"/>
    <form action="form4proc.jsp" method="post">
      <table border="0">
        <tr><th colspan="2"><b>Formulário</b></th></tr>
        <tr bgcolor="#dfdfdf">
          <td colspan="2">
            <font color="red"><%= msgOut%></font>
          </td>
        </tr>
        <tr bgcolor="#dfdfef">
          <td>Nome:</td>
          <td><input type="text" name="nome"
value="${param.nome}"></td>
        </tr>
        <tr bgcolor="#efdfdf">
          <td>Idade:</td>
          <td><input type="text" name="idade"
value="${param.idade}"></td>
        </tr>
        <tr>
          <td colspan="2">
            <input type="submit" value="OK">
            <input type="reset" value="Limpar">
          </td>
        </tr>
      </table>
    </form>
    <jsp:include page="footer.html"/>
  </body>
</html>
```

Crie uma nova página JSP para receber esses parâmetros e validar.

Form4proc.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<% //capturo os parâmetros
    int erro = 0, idade = -1;
    String nome = request.getParameter("nome");
%>
<% //defino o erro
    if (nome == null || nome.length() == 0) erro = 1;
    else {
```



```

        try {idade = Integer
            .parseInt(request.getParameter("idade"));}
        catch (NumberFormatException exc) { erro = 2; }
        if (idade < 0) erro = 2;
    }
%>
<% //redireciono para para a página correta utilizando diretiva
forward
    if (erro > 0) { %>
<jsp:forward page="index.jsp">
    <jsp:param name="msg" value="<%= erro%>"/>
</jsp:forward>
    }
    String pagina = (idade < 18) ? "form4menor.jsp" :
                                "form4maior.jsp";%>
<jsp:forward page="<%= pagina %>">
    <jsp:param name="nome" value="${param.nome}"/>
    <jsp:param name="idade" value="${param.idade}"/>
</jsp:forward>

```

Ao clicar em OK, o usuário é redirecionado para a página de validação, caso gere um erro, ele verifica qual o erro foi gerado e volta a página index. A página SEMPRE irá verificar se existem os parâmetros de erro.

Caso eles existam, ele mostra uma mensagem de erro, caso contrário, nada acontece. No caso de um erro, ele retorna e mostra um erro acima do formulário.

Formulário 4

Nome:	<input type="text"/>
Idade:	<input type="text"/>
<input type="button" value="OK"/> <input type="button" value="Limpar"/>	

Formulário 4

Forneça um nome!

Nome:	<input type="text"/>
Idade:	<input type="text"/>
<input type="button" value="OK"/> <input type="button" value="Limpar"/>	

Formulário 4

Forneça idade válida!

Nome:	<input type="text" value="Nome"/>
Idade:	<input type="text"/>
<input type="button" value="OK"/> <input type="button" value="Limpar"/>	

Caso não ocorra erro, ele usa a diretiva forward para redirecionar o usuário para a determinada página, passando os parâmetros nome e idade usando a diretiva param.

Usando o JavaScript é bem mais fácil de se fazer isso, por que o JavaScript “reconhece” os elementos do html, eu consigo ler, modificar, excluir, limpar um valor de um campo ou outro objeto na página sem a necessidade de ficar passando parâmetros. Parâmetros, isso por que o JavaScript é uma linguagem dinâmica. Muitos Programadores usam outras linguagens dinâmicas como o Python e o Ruby para fazer

validações entre outras coisas e deixam a parte mais pesada e a comunicação com o banco de dados com o Java.

Extra- Validação completa de Email com Javascript e Expressão Regular

Com esta função em Javascript, é possível realizar uma validação completa do formato do email utilizando Expressão Regular. A utilização é bem simples, basta passar o email por parâmetro para a função **IsEmail** e função retornará **True** caso o email seja válido e **False** caso o formato esteja inválido. Simples assim.

```
function IsEmail(email){
    var exclude=/^[^\-\.\\w]|^[_@\\-\\.]|^[\-\\.]{2}|^[@\\-]{2}|^([^\-\.\\w]*\\1)/;
    var check=/@([\\w\\-]+\\.)/;
    var checkend=/\\.[a-zA-Z]{2,3}$/;
    if(((email.search(exclude) != -1) || (email.search(check)) == -1 )
    || ( email.search( checkend ) == -1 ) ) {return false;}
    else {return true;}
}
```

Extra- Validar CPF via JavaScript

```
function VerificaCPF () {
    if (vercpf(document.frmcpf.cpf.value))
        {document.frmcpf.submit();}
    else
        {errors="1";if (errors) alert('CPF NÃO VÁLIDO');
        document.retorno = (errors == '1');}
}

function vercpf (cpf){
    if (cpf.length != 11 || cpf == "000000000000" ||
        cpf == "111111111111" || cpf == "222222222222" ||
        cpf == "333333333333" || cpf == "444444444444" ||
        cpf == "555555555555" || cpf == "666666666666" ||
        cpf == "777777777777" || cpf == "888888888888" ||
        cpf == "999999999999")
        return false;
    add = 0;
    for (i=0; i < 9; i ++){
        add += parseInt(cpf.charAt(i)) * (10 - i);
    }
    rev = 11 - (add % 11);
    if (rev == 10 || rev == 11) rev = 0;
    if (rev != parseInt(cpf.charAt(9))) return false;
    add = 0;
    for (i = 0; i < 10; i ++){
        add += parseInt(cpf.charAt(i)) * (11 - i);
    }
    rev = 11 - (add % 11);
    if (rev == 10 || rev == 11) rev = 0;
    if (rev != parseInt(cpf.charAt(10))) return false;
    alert('O CPF INFORMADO É VÁLIDO.');
```

Exercícios Complementares de fixação:

1) Em tempos de grande concorrência, o Hotel São Patrício quer ampliar a sua competitividade, por isso encomendou um sistema para calcular as contas de seus clientes. A promoção funciona da seguinte forma: A 1ª noite custa R\$100,00, a segunda R\$50,00 ($100/2$), a n -ésima noite custa $100/n$. Calcule e informe o valor a ser cobrado de um cliente após n noites de hospedagem no hotel.

2) Crie uma página para gerar as notas finais de um aluno. Crie um formulário nela para que receba o Código do aluno, seu nome e suas três notas. Calcule a média ponderada do aluno usando os pesos: prova1 (25%), prova2 (25%) e prova3 (50%). Mostre na tela o código e nome do aluno assim como suas notas e média. Caso a média seja menor que 5, imprima a abaixo dos dados : “reprovado” , caso contrario “aprovado”. Mostre a mensagem e as médias nas cores vermelham para reprovado e azul para reprovado.

3) A Equipe de desenvolvimento ITTraining SA foi contratada para desenvolver um sistema para a área de recursos humanos da Companhia Brasília NET SA para cálculo do salário líquido em que três valores devem ser informados pelo usuário: o salário bruto, o valor do salário-hora e o número de dependentes. O sistema deverá ser construído de acordo com as seguintes regras de negócio:

- a. Salário bruto
o $\text{Horas trabalhadas} * \text{salário hora} + (50 * \text{número de dependentes})$
- b. Desconto INSS
o Se $\text{salário bruto} \leq 1000$ $\text{INSS} = \text{salário bruto} * 8.5/100$
o Se $\text{salário bruto} > 1000$ $\text{INSS} = \text{salário bruto} * 9/100$
- c. Desconto IR
o Se $\text{salário bruto} \leq 500$ $\text{IR} = 0$
o Se $\text{salário bruto} > 500$ e ≤ 1000 $\text{IR} = \text{salário bruto} * 5/100$
o Se $\text{salário bruto} > 1000$ $\text{IR} = \text{salário bruto} * 7/100$
- d. Salário líquido
o $\text{Salário bruto} - \text{INSS} - \text{IR}$

4) Uma loja de acessórios automotivos está liquidando seus preços. Os descontos variam de acordo com a cor da etiqueta fixada nas peças. Construa a aplicação solicitada onde o usuário deverá informar a cor da etiqueta e o valor normal do produto e o sistema deverá informar o preço com desconto. Os descontos seguem a seguinte tabela:

Etiqueta	Desconto
Azul	10%
Rosa	20%
Amarelo	30%
Branco	40%

5) A Concessionária VM Automóveis SA está solicitando um sistema para calcular os valores das prestações e o valor final pago por seus clientes que financiam veículos em suas lojas. Os valores são calculados com base nas seguintes informações. Dependendo da negociação a taxa de retorno irá variar entre 3% e 10%, essa taxa é calculada uma única vez sobre o valor principal do veículo. A modalidade de juros cobrados será o modelo composto, ou seja, juros sobre juros e as taxas cobradas serão as seguintes:

- a. 12 Meses => Juros de 1% ao mês;
- b. 24 Meses => Juros de 1,5% ao mês;
- c. 36 Meses => Juros de 2% ao mês.;
- d. 48 Meses => Juros de 2,5% ao mês.

6) Crie um formulário que cadastre os tipos de objetos indicados abaixo. Não é preciso cadastrar, apenas faça a estrutura do formulário HTML, envie para uma página de resposta e mostre os dados dos objetos “cadastrados”. Faça esse exercício utilizando uma primeira versão sem o uso de classes e uma segunda versão com o uso de classes.

a. Aluno

Nome: String
RA: Integer
Curso: String
Ano: Integer
Período: String
Endereço: String
Cidade: String
Telefone: String

b. Produto

Codigo: Integer
Descrição: String
Quantidade em estoque: Integer
Valor: Float
Quantidade mínima em estoque: Integer
Quantidade máxima para compra: Integer
Fornecedor: String

c. Escola

Cnpj: Integer
Nome: String
Endereço: String
Cidade: String
Estado: String
Telefone: String
E-mail: String

c. Fornecedor

Cnpj: Integer
Nome: String
Endereço: String
Cidade: String
Estado: String
Telefone: String
E-mail: String

d. Cliente

Nome: String
RG: Integer
CPF: Integer
Idade : Integer
Endereço: String
Cidade: String
Estado: String
Telefone: String
E-mail: String

e. Curso

Codigo: Integer
Nome: String
Descrição: String
Coordenador: String
CargaHorária: Integer

- 7) Validar usando JavaScript TODOS os formulários dos exercícios anteriores. Verifique se os campos foram preenchidos, caso haja necessidade, verifique se os campos são números (para valores numéricos).
- 8) Idem ao exercício 4, mas validando via JSP.

Exercícios de Pesquisa:

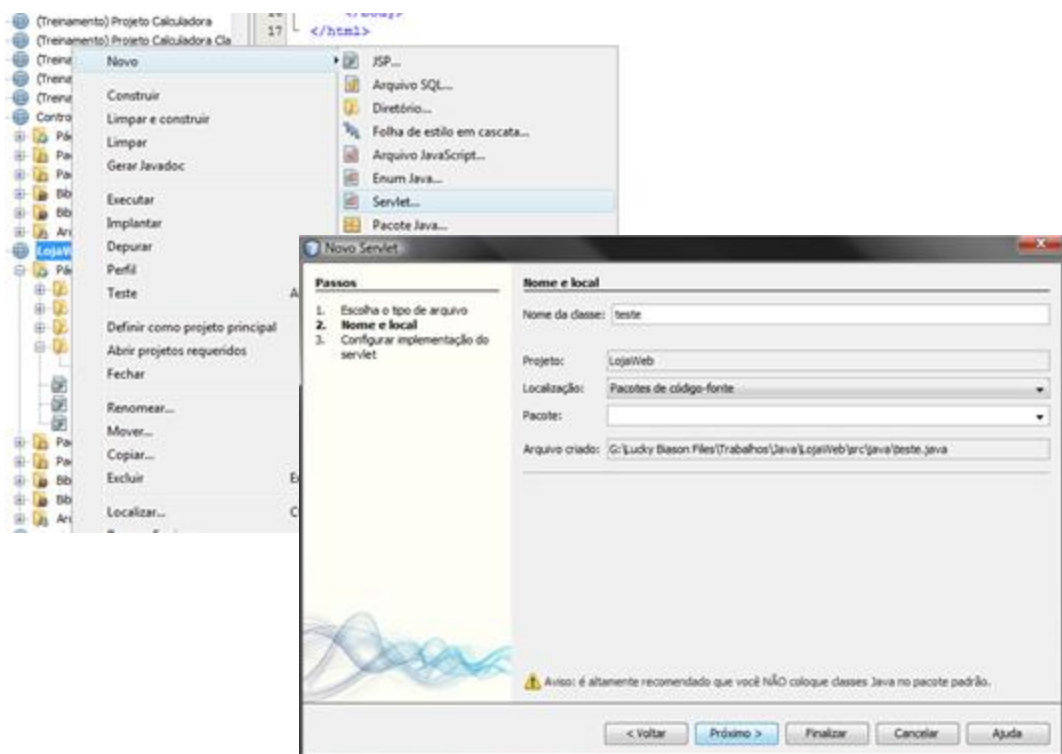
- 1) Defina: JSP, Java e TomCat.
- 2) Para que surgiu a JSP?
- 3) Qual(is) linguagem pode ser utilizada para se criar páginas JSP?
- 4) Antes de surgir a JSP era possível um desenvolvedor Java criar alguma aplicação para a Internet? Justifique.
- 5) O que é necessário para o desenvolvedor trabalhar com a JSP?
- 6) Defina e cite exemplos, que não sejam os da apostila, sobre : ações, diretivas, Scripting, taglibs.
- 7) O que é J2SE? Porque precisamos dele para trabalhar com a JSP?
- 8) Explique o funcionamento da JSP.
- 9) NA JSP, qual comando deveu utilizar para recebermos parâmetros passados através de tags de formulários?
- 10) Monte uma página JSP com uma tabela contendo os principais operadores condicionais, matemáticos e lógicos Java

Servlets

Formulários HTML podem receber entradas via caixa de texto, mas eles não conseguem processar ou usar essas informações para criar uma resposta dinâmica. Uma forma de processar informação de entrada é ter um **servlets** – trecho especial de código em Java que pode extrair informação de uma solicitação e enviar a resposta desejada de volta ao cliente.

SERVLETS SÃO CLASSES JAVA, desenvolvidas de acordo com uma estrutura bem definida, e que, quando instaladas junto a um Servidor que implemente um Servlet Container (um servidor que permita a execução de Servlets, muitas vezes chamado de Servidor de Aplicações Java), podem tratar requisições recebidas de clientes. Na verdade, Servlets são à base do desenvolvimento de qualquer aplicação escrita em Java para a Web . O container Tomcat que você está usando é um Container Servlet.

Os dois mais importantes métodos de uma classe **servlets** são o **doPost()** e **doGet()**, que são chamados sempre que houver uma solicitação do tipo POST/GET para servlets, respectivamente. Para criar um servlet, vá a Novo, Servlet. Defina um nome para ele, e na próxima tela (A MAIS IMPORTANTE!!), a configuração do servlets. Com isso ele será criado no XML principal para a aplicação WEB, e com isso ele poderá ser acessado facilmente.



Ele terá uma cara assim:

```
package servlet;

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

public class NewServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            /* TODO output your page here
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet NewServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet NewServlet at " +
request.getContextPath () + "</h1>");
            out.println("</body>");
            out.println("</html>");
            */
        } finally {
            out.close();
        }
    }

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

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

```
@Override
public String getServletInfo() {
    return "Short description";
} // </editor-fold>
}
```

Esse é toda a parte dele que você precisa se preocupar no começo. O resto cria métodos para receber as informações dos formulários, via POST ou GET, e envia para esse método.

Caso haja necessidade de um tratamento diferenciado no envio de informações, isto é, precise que o tratamento para caso recebe POST seja um e GET outro, daí a próxima parte se torna muito importante.

Um exemplo simples seria fazer o servlets não aceitar o envio de informações via um dos dois, e retornar para a página anterior, ou outro exemplo seria, tratar com mais segurança o método GET, por ser mais vulnerável que o POST.

Os métodos doPost e doGet estão descritos a seguir:

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

Dentro desses métodos você pode programar de forma diferenciada.

O Ciclo de Vida de um Servlet

Todo Servlet segue um ciclo de vida composto de 3 fases: inicialização, atendimento às requisições e finalização.



A inicialização ocorre quando o Servlet Container carrega o Servlet: se o parâmetro `<load-on-startup/>` estiver presente e contiver um inteiro positivo, essa carga ocorre quando o próprio servidor é iniciado; caso contrário, essa carga ocorre quando é recebida a primeira requisição a ser mapeada para a aplicação que contém o Servlet.

A inicialização ocorre quando o Servlet Container carrega o Servlet: se o parâmetro `<load-on-startup/>` estiver presente e contiver um inteiro positivo, essa carga ocorre quando o próprio servidor é iniciado; caso contrário, essa carga ocorre quando é recebida a primeira requisição a ser mapeada para a aplicação que contém o Servlet.

Após a inicialização, o Servlet pode atender requisições. Assim, enquanto o servidor estiver ativo, e a aplicação que contém o Servlet estiver carregada, este permanecerá na fase 2 de seu ciclo.

Uma vantagem da tecnologia de Servlets e páginas JSP com relação a outras tecnologias é que o fato do Servlet permanecer carregado permitindo assim com que dados armazenados em variáveis de classe persistam ao longo dos diversos pedidos recebidos. Assim, é possível manter um pool de conexões ao banco de dados, por exemplo, de maneira que não haja necessidade de iniciar e estabelecer uma nova conexão ao banco de dados a cada novo pedido recebido.

Finalmente, quando o servidor é finalizado, ou quando a aplicação é tornada inativa pelo Servlet Container, o Servlet é finalizado.

Exemplo de processamento de formulários usando um servlets

1º) Vamos criar uma nova JSP chamada Acesso e criar o conteúdo da página. Um pequeno formulário com o Primeiro e Ultimo Nome de uma Pessoa.

acesso.jsp

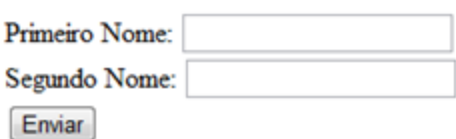
```

<%@page contentType="text/html" pageEncoding="ISO-8859-1"%>
<html>
  <head>
    
```

```

<title>Testando Servlets</title>
</head>
<body>
  <form name="principal" action="Teste" method="post">
    Primeiro Nome: <input type="text"
name="primeironome"><br/>
    Segundo Nome: <input type="text"
name="segundonome"><br/>
    <input type="submit" value="Enviar">
  </form>
</body>
</html>

```



3º) Vamos criar um servlet para receber os parâmetros do formulário.

Teste.java

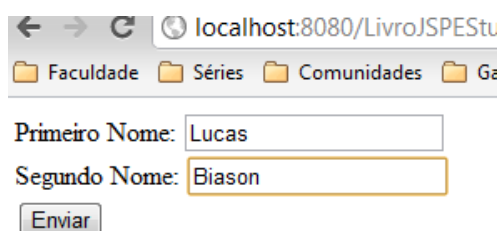
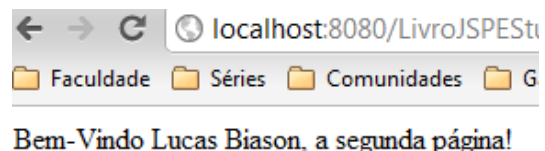
```

public class Teste extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=ISO-8859-9");

        String nome = request.getParameter("primeironome");
        String snome = request.getParameter("segundonome");

        PrintWriter out = response.getWriter();
        try {
            out.println("<html><head>");
            out.println("<title>Servlet Teste</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Bem vindo "+nome+" "+
                        snome+", a segunda página</h1>");
            out.println("</body></html>");
        } finally {out.close();}
    }
}

```

A Classe HttpServlet

A classe `HTTPSRVLET` sobreescreve o método `SERVICE` para distinguir entre as solicitações típicas recebidas de um navegador Web cliente. Os dois métodos mais comuns e usados de solicitação HTTP são `GET` e `POST`. As utilizações dos dois métodos são muito comuns, uma vez que o método `GET` pode tanto obter informações, onde você pode requisitar um arquivo ou uma imagem, como também pode enviar dados, que neste caso temos o limite do cabeçalho HTTP. O método `POST` não requisita informações, e sim as envia (posta), dados para o servidor. As utilizações mais comuns de solicitações `POST` consistem em enviar ao servidor informações de um formulário HTML em que o cliente insere dados ou envia informações ao servidor para que esse possa pesquisar em um banco de dados e etc.

A classe `HttpServlet` define os métodos `doGet` e `doPost` para responder as solicitações `GET` e `POST` vindas de um cliente. Os dois métodos recebem como argumentos um objeto `HttpServletRequest` e um objeto `HttpServletResponse` que permitem interação entre o cliente e o servidor

Criando um Servlet que Trabalha com o Método POST

A seguir você tem um Servlet que trabalha com o método `POST`:

TrabComPost.java

```
public class TrabComPost extends HttpServlet {

    public void destroy(){super.destroy();}

    protected void doPost(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=ISO-8859-9");

        String usuario = request.getParameter("usuario");
        String senha = request.getParameter("senha");

        PrintWriter out = response.getWriter();
        try {
            out.println("<html><head>");
            out.println("<title>Servlet Teste</title>");
            out.println("</head><body>");

            if(usuario.equals("lucas") && senha.equals("123"))
                out.println("<h1>Bem vindo "+usuario+"</h1>");
        }
    }
}
```

```
        else
            out.println("Usuario ou senha inválidos");

            out.println("</body></html>");
            response.setContentType("text/html");
        } finally {out.close();}
    }
    public void init() throws ServletException{
        super.init();
    }
}
```

Utilizando o método `doPost()` O você recupera valores vindos pelo método POST. Quando uma requisição HTTP é recebida por uma classe que estende `HttpServlet`, seu método `service()` é chamado, sendo que a implementação padrão desse método irá chamar a função correspondente ao método da requisição recebida. Ou seja, caso um envio seja feito pelo método POST, como no exemplo, o método `doPost()` implementado por você será chamado.

A interface `HttpServletRequest` trabalha com alguns métodos e no caso, você conheceu o método `getParameter(String n)`. Esse método retoma o valor associado com um parâmetro enviado para o Servlet como parte de uma associação GET ou POST. O argumento `n` representa o nome do parâmetro. No caso do seu servlet foi usuário e a senha, no qual vinham das tags `<input />` do xhtml de mesmo nome.

Através de uma condicional, você verifica se foram passados valores como usuário e senha iguais ao valor verificado pelo método `equals(Strings)`.

A interface `HttpServletResponse` contém a resposta ao cliente. Uma grande número de métodos são fornecidos para permitir ao Servlet formular uma resposta. No seu caso, o método `setContentType(String tipo)` define o tipo MIME da resposta ao navegador. O tipo MIME permite ao navegador determinar como exibir os dados. No caso o tipo MIME de resposta foi `"text/html"`, que indica que a resposta é um documento HTML. Para dar uma resposta ao cliente, você pode usar a `OutputStream` ou o `PrintWriter` que é retomado do objeto `response`.

Trabalhando com o Método GET

O método GET trabalha com informações enviadas via URL. Esse método pode ser usado via query string de um link ou via formulário com o atributo `method` em GET. Uma string de consulta é parte do URL que aparece depois de um ponto de interrogação. Por exemplo, o URL a seguir contém uma string de consulta:

<http://integrator.com.br/buscar/?p=Hypertext+Preprocessor>

Nesse exemplo, a string de consulta contém uma variável denominada `p` cujo valor é "Hypertext Preprocessor".

As strings de consulta são usadas para transmitir informações do navegador para o servidor. Normalmente, você não digita a string de consulta diretamente na barra de endereços do navegador. Ao contrário, cria um link em uma página que contém a string de consulta.

<http://integrator.com.br/TrabComGetServlet?empresa=Integrator>

Você pode transmitir diversas variáveis de consulta em uma única string. Para fazer isso, basta separá-las com o caractere `&` ("e" comercial).

<http://integrator.com.br/TrabComGetServlet?nome=Edson&empresa=Integrator>

Recuperando Strings de Consulta com Servlets

Para recuperar uma string de consulta com um Servlet, crie o seguinte arquivo:

TrabComGet.java

```
public class TrabComGet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=ISO-8859-9");

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

        PrintWriter out = response.getWriter();
        try {
            out.println("<html><head>");
            out.println("<title>Trabalhando com GET em
Servlet</title>");
            out.println("</head><body>" +
                "Nome: <string>"+nome+"</string><br/>" +
                "Empresa:
<string>"+empresa+"</string><br/>");
            out.println("</body></html>");
        } finally {out.close();}
    }
}
```

O método `getParameter(String s)` além de retornar o valor informado, também retorna null caso não recupere o valor indicado entre os parênteses.

Outros Métodos Muito Úteis da Interface `HttpServletRequest` (Extra):

Logicamente você deve estar querendo não só recuperar cada parâmetro separado, mas varrer esses valores em um loop. Essa interface, além de oferecer os métodos da superclasse `ServletRequest`, oferece também suporte a cookies e sessões. Além do método `getParameter()` você também conta com os seguintes métodos:

`String[] getParameterValues(String s)` - Retorna um array de Strings caso o parâmetro tenha múltiplos valores.

`Enumeration getParameterNames()` - Retorna uma enumeração com os nomes de todos os parâmetros enviados.

`String getHeader(String s)` - Retorna o valor do cabeçalho enviado.

`Enumeration getHeaders(String s)` - Retorna uma enumeração com os valores do cabeçalho.

`Enumeration getHeaderNames()` - Retorna uma enumeração com os nomes de todos os cabeçalhos.

Como são métodos encontrados com a classe `ServletRequest` você pode usá-los em outros protocolos que não sejam HTTP.

Praticando os Métodos

Escolha as músicas

☐ Rock
☐ POP
☐ Dance
☐ MPB
☐ Sertanejo

Considere a seguinte situação :

Você tem um formulário com opções musicais e gostaria de varrê-lo para saber quais foram escolhidas pelo usuário.

Enviar

Musica.jsp

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
    <title>Trabalhando com outros métodos de
    HttpServletRequest</title>
  </head>
  <body>
    <jsp:include page="topo.html"/>
```

```
<form action= "TrabOutrosMetComServlet" method="post">
  <h2>Escolha as músicas</h2>
  <p>
    <input name="musica" type="checkbox"
value="ROCK"/> Rock<br/>
    <input name="musica" type="checkbox" value="POP"/>
    POP<br/>
    <input name="musica" type="checkbox"
value="DANCE"/> Dance<br/>
    <input name="musica" type="checkbox" value="MPB"/>
    MPB<br/>
    <input name=" musica" type="checkbox"
value="SERTANEJO"/> Sertanejo
  </p>
  <br/>
  <input type="submit" name="btEnviar" value="Enviar"/>
  <input type='button' value='Voltar'
onClick='history.go(-1)'/>
</form>
<jsp:include page="footer.html"/>
</body>
</html>
```

O formulário demonstra várias caixas de checagem com o mesmo nome. A idéia é o usuário selecionar uma ou mais músicas, mas capturar apenas as selecionadas.

O Servlet a seguir demonstrará como você varre os valores para obter esse resultado, usando o método `getParameterValues()`:

TrabOutrosMetComServlet.java

```
public class TrabOutrosMetComServlet extends HttpServlet {

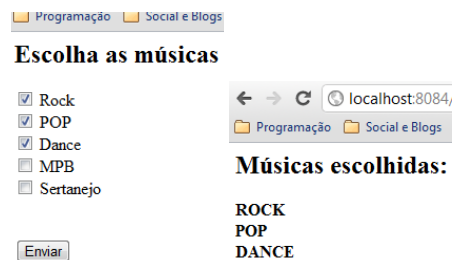
    protected void processRequest(HttpServletRequest request,
                                   HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            String[] e = request.getParameterValues("musica");
            String html = "<html><head>"
                + "<title>Trabalhando com Outros métodos em"
                + "Servlet</title><head><body>"
                + "<h2>Músicas escolhidas: </h2>";
```

```

        for (int i = 0; i < e.length; i++) {
            html += "<strong>" + e[i] + "</strong><br I>";
        }
        html += "<input type='button' value='Voltar'
onClick='history.go(-1)'/>"
            + "</body></html>";
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writer.print(html);
        writer.close();
    } finally {
        out.close();
    }
}

```

Através do método `getParameterValues(String s)` você chama o campo música, selecionados pelo usuário, e o atribui a um array (isso porque é um array de informações). Com o loop `for()` você distribui os valores encontrados na variável `e[]` gravando-os no html que será impresso na tela do usuário.



Varrendo um Formulário

Cadastre-se aqui:

Nome:

E-mail:

Site:

Agora se você preferir, pode fazer também a varredura em todos os objetos enviados em um formulário, por exemplo:

O formulário criado no documento é simples com campos de texto somente.

Varrendo.jsp

```

<%@page contentType="text/html" pageEncoding="ISO-8859-1"%>
<html>
<head>

```



```
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1" />
<title>Trabalhando com outros métodos de
HttpServletRequest</title>
</head>

<body>
<jsp:include page="topo.html"/>
<form action= "VarrendoTodosObjetos" method="post" >
  <h2>Cadastre-se aqui:</h2>
  Nome:<input name="nome" type="text"/><br/>
  E-mail:<input name="email" type="text"/><br/>
  Site:<input name="site" type="text"/> <br/>
  <br/>
  <input type="submit" name="Enviar" value="Enviar"/>
  <input type='button' value='Voltar'
onClick='history.go(-1)'/>
</form>
<jsp:include page="footer.html"/>
</body>
</html>
```

A seguir você tem o Servlet que faz a varredura nesse formulário:

VarrendoTodosObjetos.java

```
public class VarrendoTodosObjetos extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            Enumeration e = request.getParameterNames();
            String html = "<html><head>"
                + "<title>Trabalhando com getParameterNames ()
</title>"
                + "</head>"
                + "<body>"
                + "<h2>Dados Cadastrados: </h2>";
            while (e.hasMoreElements()) {
                String param = (String) e.nextElement();
                html += "<strong>" + param + ":</strong>"
                    + request.getParameter(param) + "<br/>";
            }
        }
```

```
html += "<input type='button' value='Voltar'
onClick='history.go(-1)'/>"
      + "</body></html>";

response.setContentType("text/html");
PrintWriter writer = response.getWriter();
writer.print(html);
writer.close();
} finally {
    out.close();
}
}
```



Dados Cadastrados:

nome:Lucas

site:oi

email:biasonlucky@hotmail.com

Enviar:Enviar

Como já foi dito anteriormente, mas não custa reforçar, o método `getParameterNames()` retorna uma enumeração com os nomes de todos os parâmetros enviados.

Com o método `hasMoreElements()` você testa se a enumeração contém mais elementos, onde o loop `while` faz com que todos os elementos sejam vistos.

A chamada ao método `nextElement()`, de `Enumeration` retorna os sucessivos elementos, onde ocorre uma coerção de tipo (`Type Casting`) para `String`. A variável `param` nesse momento contém como valor o nome dos campos passados pelo formulário.

Nesse momento você transmite a `String html` o nome do campo enviado pelo formulário, através da variável `param` e também os valores enviados por esse campo, trazidos pelo método `getParameter()` que recebe como parâmetro a variável `param` (pois é nela que se encontra o nome do campo, alterado a cada nova passada do loop `while`).

Cookies, Sessões e Java Beans

Os cookies, usados até os dias de hoje para armazenar no computador do usuário suas preferências na navegação de um site, comum em sites de comércio eletrônico e as sessões, utilizadas para armazenar temporariamente informações de um usuário em uma determinada área de um site, como por exemplo, à área administrativa ou então uma caixa de e-mails.

Formas de modificar o escopo de um parâmetro.

Cada vez que você envia uma solicitação para o servidor, ele considera que você é um novo usuário. Ele não sabe que você é a mesma pessoa que acabou de solicitar algum outro URL alguns momentos atrás. É assim que o protocolo HTTP funciona e isto pode ser um problema em certas situações.

Por exemplo: digamos que você esteja conectado e comprando um item em um sitio de compras on-line. Quando você clicar em “Comprar” e seguir para examinar algum outro item, é fundamental que o servidor reconheça que você é a mesma pessoa que quer continuar comprando e receber uma conta global ao terminar. Um processo de compras normalmente se estende por umas poucas páginas o que significa que os valores informados pelo usuário na primeira página não estarão disponíveis na terceira página nem mais além. Da mesma forma, valores informados na segunda página não estarão disponíveis na quarta página nem mais além.

Para resolver esses problemas, é conveniente separar o código Java das páginas JSP, o que se pode fazer com o auxílio dos JavaBeans e da taglibs. Outro uso possível é o armazenamento de informações nos vários escopos da aplicação. Exemplo:

```
- application.setAttribute("nome","Júlio");  
- session.setAttribute("nome","Júlio");  
- request.setAttribute("nome","Júlio");  
- page.setAttribute("nome","Júlio");
```

Escopo de objetos JSP:

Application: Define objetos que persistem durante toda a aplicação onde foram criados, até sua remoção do web container.

Session: Define objetos válidos durante a sessão de navegação do cliente.

Request: Define objetos acessíveis em todas as paginas envolvidas no processamento de uma requisição, até a finalização da resposta.

Page: Define objetos acessíveis apenas nas páginas onde foram criados.

Exemplo: Vamos criar uma jsp chamada inicio. A qual terá dois campos a ser preenchidos: nome e telefone. Ambos os campos, em vez de serem validados por JavaScript, serão por Java mesmo. Para isso é necessário criar uma página jsp vazia, que ira apenas receber os parâmetros e verificar se estão preenchidos. Caso não estejam, ela irá retornar a pagina anterior. Caso estejam, ela irá para a próxima página.

index.jsp

```
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<html>
  <head>
    <title>Testando Escopo</title>
  </head>
  <body>
    <jsp:include page="topo.html"/>
    <form name="principal" action="validar.jsp" method="post">
      Nome: <input type="text" name="nome"><br/>
      Telefone: <input type="text" name="telefone"><br/>
      <input type="submit" value="Enviar">
    </form>
    <jsp:include page="footer.html"/>
  </body>
</html>
```

Validar.jsp

```
<%
String nome = request.getParameter("nome");
String telefone = request.getParameter("telefone");

if (nome == null || nome.equals("")) {
  response.sendRedirect("index.jsp");
} else {
  response.sendRedirect("ProcessarAcesso.jsp");
}
%>
```

ProcessarAcesso.jsp

```
<html>
  <head>
```

```

<title>Resultado</title>
</head>
<body>
    <jsp:include page="topo.html"/>

    Bem-vindo, <%= request.getParameter("nome")%>, a segunda
    página.<br/>
    Seu telefone é : <%= request.getParameter("telefone")%>

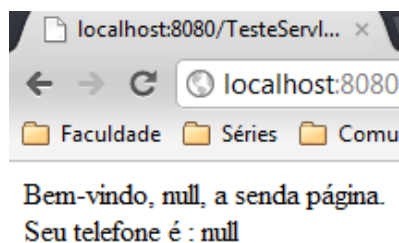
    <br/>
    <input type='button' value='Voltar' onClick='history.go(-
1)'/>
    <jsp:include page="footer.html"/>
</body>
</html>

```

O Iniciamos o projetos normalmente, e inserimos os valores no campo:

Nome:

Telefone:



E o erro já aparece de cara:

Em vez de colocar o nome e o telefone digitados, ele coloca null. Isso por que os parâmetros nome e telefone, vivem apenas entre Acesso e Validar, pois estão no request. Como explicado acima, temos algumas opções:

- Colocar campo "hidden" – ocultos em um formulário e enviar para a próxima página ou enviar os parâmetros via url. (deixa o sistema mais lento, ruim por que via url os parâmetros ficam visíveis. Existe muita redundância de informações.).
- Usando servlets (visto mais a frente) (dificulta a programação e criação da pagina.jsp).
- Usar JavaBean (uma boa opção, mas o Javabeans é mais complicado que mexer em certos casos, nesse nem tanto)
- Armazenar os parâmetros recebidos no objeto Session ou Application para que possam ser vistos por toda a aplicação. (vamos escolher esse por ser o mais fácil e também um método muito bom).

Validar.jsp

```
<%
    String nome = request.getParameter("nome");
    String telefone = request.getParameter("telefone");

    if(nome == null || nome.equals("") || telefone == null ||
    telefone.equals("")){
        response.sendRedirect("index.jsp");
    }else{
        application.setAttribute("nome",nome);
        application.setAttribute("telefone",telefone);
        response.sendRedirect("ProcessarAcesso.jsp");
    }
%>
```

ProcessarAcesso.jsp

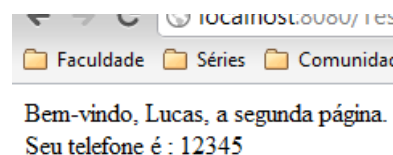
```
<%@page contentType="text/html" pageEncoding="ISO-8859-1"%>

<html>
    <head>
        <title>Resultado</title>
    </head>
    <body>
        <jsp:include page="topo.html"/>

        Bem-vindo, <%= application.getAttribute("nome")%>, a
        segunda página.<br/>
        Seu telefone é : <%= application.getAttribute("telefone")%>

        <br/>
        <input type='button' value='Voltar' onClick='history.go(-
        1)'/>
        <jsp:include page="footer.html"/>
    </body>
</html>
```

Com isso o problema foi resolvido:



Cookies

Você pode usar um cookie para armazenar informações no computador de um cliente quando ele visitar seu site da Web.

Essas informações podem ser usadas para identificar o cliente quando ele retornar ao seu site.

A capacidade de identificar os clientes e personalizar conteúdo é importante porque pode ser usada para aumentar as vendas. Um exemplo simples : talvez você queira exibir anúncios distintos para clientes diferentes, de acordo com seus interesses. Se você registrou o fato de que um determinado cliente gosta de visitar as páginas de seu site da Web que mostram livros de informática, pode mostrar automaticamente a esse cliente mais anúncios relacionados a livros de informática quando ele retornar a visitar o site. É assim que o comércio eletrônico Amazon.com ficou famoso.

Há dois tipos de cookies: cookies de sessão e cookies persistentes.

Os cookies de sessão são armazenados na memória. Permanecem no computador do cliente somente enquanto ele está visitando o seu site da Web.

O cookie persistente, por outro lado, podem durar meses ou até anos. Os cookies persistentes são armazenados em um arquivo de texto no computador do cliente. Esse arquivo de texto é denominado arquivo Cookie nos computadores com sistema operacional Windows e arquivo Magic Cookie nos computadores Macintosh.

Criando Cookies

Para criar um cookie, você precisa instanciar a classe `javax.servlet.http.Cookie`. Essa classe fornece apenas um tipo de construtor que recebe duas variáveis do tipo `String`, que representam o nome e o valor do cookie. O servlet a seguir mostra como criar um cookie:

usecookie.java

```
public class usecookie extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();

        Cookie meucookie = new Cookie("nome", "Lucas");
        meucookie.setMaxAge(60);
        meucookie.setSecure(false);
        meucookie.setComment("meu nome");
        response.addCookie(meucookie);
    }
}
```

```
String html = "<html><body>"
    + "<h2> Seu cookie foi criado com sucesso!</h2>"
    + "<a href=\"VisualizaCookie\">"
    + "Clique aqui para ver o cookie criado"
    + "</a>"
    + "</body></html>";
out.print(html);
out.close();
}
```

A classe `Cookie(String s, String s)` é chamada com a passagem de dois parâmetros, um o nome do cookie e o outro o valor.

O método `setMaxAge(int i)` define o tempo (em segundos) para que o cookie expire. No caso de você ter um cookie por dois dias na máquina, poderia ser definido da seguinte forma: `seucookie.setMaxAge(2*24*60*60)`.

O método `setSecure(boolean b)` indica se o cookie deve ser transferido pelo protocolo HTTP padrão.

O método `setComment(String s)` cria um comentário para o cookie criado.

O método `addCookie(Cookie c)` grava o cookie na máquina do usuário.

O resultado dessa página será apenas uma saída HTML comum, contendo um texto e um link. Esse link o levará direto para um Servlet que irá receber o cookie.

Recuperando um Cookie

Para recuperar um cookie você usa o método `getCookies()` do objeto implícito `request`. O Servlet a seguir recupera o cookie criado anteriormente:

usecookie.java

```
public class VisualizaCookie extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=UTF-8");
        PrintWriter out = response.getWriter();

        Cookie listaCookies[] = request.getCookies();
        Cookie nome = null;
        for (int i = 0; i < listaCookies.length; i++) {
            if (listaCookies[i].getName().equals("nome")) {
                nome = listaCookies[i];
                break;
            }
        }
    }
}
```

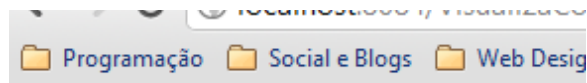


```
    }  
  }  
  String html = "<html><body>"  
    + "O cookie chamado <strong>nome</strong> tem o  
valor:"  
    + "<strong>" + nome.getValue() + "</strong>"  
    + "</body></html>";  
  
  out.print(html);  
  out.close();  
}
```

O método `getCookies()` recupera os cookies encontrados.

O loop `for` varre os cookies e com o `if` você verifica se o cookie é o que se chama `nome`. Caso seja, o valor é atribuído a variável `nome` e o `break` é chamado para finalizar o loop.

O método `getValue()` é o responsável por retornar o valor encontrado no cookie. No caso seria `Edson` o valor encontrado no cookie criado no Servlet anterior.



O cookie chamado **nome** tem o valor: **Lucas**

Sessões

Largamente usada em aplicações Web administrativas e também em comércios eletrônicos, as sessões carregam geralmente informações de uma página para outra, usando ou não cookies.

A API de Servlets disponibiliza um módulo extremamente útil no controle de informações associadas ao usuário que acessa uma área restrita ou que necessita de informações que sejam transmitidas de uma página para outra, conhecido como módulo de gerenciamento de sessões de usuários.

Esse módulo funciona basicamente criando um identificador de sessão na primeira vez que o usuário acessa a aplicação. A interface que representa a sessão de usuário é a `javax.servlet.http.HttpSession`.

A partir dessa definição, o servidor procura fazer com que todas as requisições vindas daquele usuário carreguem esse identificador de sessão, seja através

de cookies, ou de URLs (com informações adicionais de caminho) para que incorporem essa informação.

Dentro do servidor é estabelecido um objeto de sessão único e que somente pode ser acessado pelo cliente que o chamou. Assim sendo, esses objetos de sessão não são compartilhados entre cada usuário da aplicação.

A seguir você vai fazer dois Servlets e uma página index.jsp que representarão a criação e a utilização da Sessão:

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!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=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <a href="CriarSessao">Criar Sessão</a>
  </body>
</html>
```

CriarSessao.java

```
public class CriarSessao extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();

        HttpSession sessao = request.getSession(true);
        sessao.setAttribute("nome", "lucas");
        sessao.setMaxInactiveInterval(50);

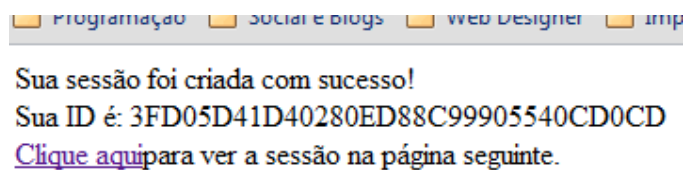
        SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yy
- HH:mm:ss");

        String html = "<html><body>"
```

```

        + "Sua sessão foi criada com sucesso!<br/>"
        + "Sua ID é: <strong>" + sessao.getId() +
"</strong> <br/> "
        + "<a href=\"VisualizarSessao\">Clique aqui</a>"
        + " para ver a sessão na página seguinte . "
        + "</body></html>";
out.print(html);
out.close();
}

```



O método `getSession(boolean b)` retorna um objeto `HttpSession` associado com a atual sessão de navegação do cliente. Esse método pode criar um objeto `HttpSession` (argumento `true`) se ainda não existir um para o cliente.

O armazenamento de um objeto de sessão é feito pelo método `setAttribute(String s, Object obj)`. No caso, o objeto de sessão chamado `nome` contém um valor chamado `Edson`.

O método `getId()` captura o identificador dessa sessão. Quando você clicar para ir ao segundo Servlet criado, essa sessão será mantida, para que o nome criado nessa sessão seja recuperado.

VisualizarSessao.java

```

public class VisualizarSessao extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=UTF-8");
        PrintWriter out = response.getWriter();

        HttpSession sessao = request.getSession(true);
        String nome = (String) sessao.getAttribute("nome");
        String html = "<html><body>";
        if(nome!=null){
            html= html + "Sua ID é
<strong>" + sessao.getId() + "</strong><br/>"
            + "E seu nome é: <strong>" +
            nome + "</strong><br/>"

```

```
        + "<a href='FecharSessao'>Clique aqui</a>"
        + "para fechar a sessão.";
    }else{
        html= html+ "Sua sessão não foi criada. <br/>"
        + "<a href='CriarSessao'>Clique aqui</a>"
        + "para criar a sua sessão.";
    }
    html=html + "</body></html>";
    out.print(html);
    out.close();
}
```

Programação Social e Blogs Web Designer Importa

Sua ID é 3FD05D41D40280ED88C99905540CD0CD
E seu nome é: **lucas**
[Clique aqui](#) para fechar a sessão.

Para recuperar um objeto de sessão, você deve usar o método `getAttribute(String s)`. Esse método retorna um `Object` e que, para você ter em formato `String`, uma coerção de tipo deve ser feita. Caso não encontre o objeto de sessão procurado, esse método retorna um valor `null`;

Terminando uma Sessão

Uma outra característica importante dos objetos de sessão é que ele permite que sessões criadas automaticamente expirem após um determinado tempo de inatividade. Obviamente, o desenvolvedor também pode expirar uma sessão explicitamente através de sua programação.

FecharSessao.java

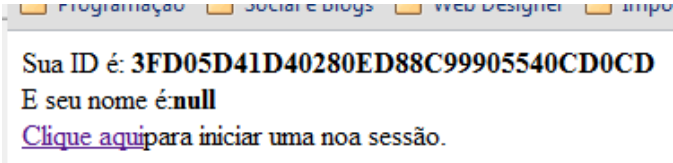
```
public class FecharSessao extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=UTF-8");
        PrintWriter out = response.getWriter();

        HttpSession sessao = request.getSession(true);
        sessao.removeAttribute("nome");
        String nome = (String) sessao.getAttribute("nome");

        String html = "<html><body>"
            + "Sua ID é: <strong>" + sessao.getId()
            + "</strong><br/>"

```

```
+ "E seu nome é:<strong>" + nome + "</strong><br/>"
+ "<a href='CriarSessao'>Clique aqui</a>"
+ "para iniciar uma noa sessão."
+ "</body></html>";
out.print(html);
out.close();
}
```



Sua ID é: 3FD05D41D40280ED88C99905540CD0CD
E seu nome é: null
[Clique aqui](#) para iniciar uma noa sessão.

Gerenciando uma Sessão

Você pode controlar a expiração de uma sessão, obtendo um maior controle de sua aplicação Web.

Usando método `setMaxInactiveInterval(int i)` você pode definir o tempo máximo de inatividade na aplicação, em segundos. Com o método `getMaxInactiveInterval()` você captura o tempo dado como máximo de inatividade, em segundos. Usando o método `invalidate()` você permite que a sessão seja expirada explicitamente pela aplicação.

CriarSessao.java

```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();

    HttpSession sessao = request.getSession(true);
    sessao.setAttribute("nome", "lucas");
    sessao.setMaxInactiveInterval(50);

    SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yy
- HH:mm:ss");

    String html = "<html><body>"
        + "Sua sessão foi criada com sucesso!<br/>"
        + "Sua ID é: <strong>" + sessao.getId()
        + "</strong> <br/> "
        + "O tempo máximo de inatividade é:<strong>"
        + sessao.getMaxInactiveInterval()
```

```
        + "</strong> segundo(s)<br/>"
        + "<a href=\"VisualizarSessao\">Clique aqui</a>"
        + " para ver a sessão na página seguinte . "
        + "</body></html>";
    out.print(html);
    out.close();
}
```

Nesse ponto você determinou, através do método `setMaxInactiveInterval()` que o tempo de inatividade do acesso ao seu Servlet seria de apenas um segundo. Com o método `getMaxInactiveInterval()` você pode colocar na página o tempo configurado para a inatividade da aplicação.

```
Sua sessão foi criada com sucesso!
Sua ID é: 3FD05D41D40280ED88C99905540CD0CD
O tempo máximo de inatividade é: 1 segundo(s)
Clique aqui para ver a sessão na página seguinte .
```

Caso você venha a querer ir para o Servlet seguinte, será necessário ir antes de um segundo, pois a sessão será expirada. É uma ótima forma de configurar áreas que exijam segurança.

Descobrimos a Criação e o Último Acesso

Além de controlar a expiração de uma sessão, você também pode descobrir quando foi criada e o último acesso. Para fazer isso basta chamar o método `getCreationTime()` para a data de criação e `getLastAccessedTime()`.

Caso queira usá-los, faça como no exemplo a seguir:

```
new Date (sessao.getCreationTime ( )) ;
new Date (sessao . getLastAccessedTime ( )) ;
```

Foi utilizado `Date` do pacote `java.util` para converter em formato de data o resultado obtido pelos métodos usados. Perceba que nesta simples data você tem diversas informações como dia da semana (Sun), mês (Set) dia, hora e outras informações como o `timezone`. Caso queira converter em nosso sistema idiomático, pode fazê-lo de forma simples, como mostrado a seguir:

CriarSessao.java

```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html; charset=UTF-8");
```

```
PrintWriter out = response.getWriter();

HttpSession sessao = request.getSession(true);
sessao.setAttribute("nome", "lucas");
sessao.setMaxInactiveInterval(50);

SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yy
- HH:mm:ss");

String html = "<html><body>"
+ "Sua sessão foi criada com sucesso!<br/>"
+ "Sua ID é: <strong>" + sessao.getId()
+ "</strong> <br/> "
+ "O tempo máximo de inatividade é:<strong>"
+ sessao.getMaxInactiveInterval()
+ "</strong> segundo(s)<br/>"
+ "Sessão criada em: <strong>"
+ formato.format(
    new Date(sessao.getCreationTime()))
+ "</strong><br/>"
+ "Último acesso em : <strong>"
+ formato.format(
    new Date(sessao.getLastAccessedTime()))
+ "</strong><br/>"
+ "<a href=\"VisualizarSessao\">Clique aqui</a>"
+ " para ver a sessão na página seguinte . "
+ "</body></html>";
out.print(html);
out.close();
}
```

```
Sua sessão foi criada com sucesso!
Sua ID é: 893E2514014F62BE55C637B180F26EB3
O tempo máximo de inatividade é: 1 segundo(s)
Sessão criada em: 12/11/11 - 22:40:06
Último acesso em : 12/11/11 - 22:40:06< / strong>
Clique aqui para ver a sessão na página seguinte .
```

Primeiro, e muito importante, não esqueça de importar os pacotes referentes a classe Date e a classe SimpleDateFormat. A classe SimpleDateFormat fornece um conjunto de caracteres padrão para formatação do objeto Date. Para utilizar, chame o método format e , trazendo assim o formato de data que criou na chamada da classe destruindo uma Sessão.

Você viu que é possível remover um objeto de sessão, mas que isso não destruiu a sessão. Destruir uma sessão é removê-la, e que, se for chamar novamente uma página contendo sessões, um novo objeto de sessão é criado.

JavaBeans – um estudo inicial

O desenvolvimento de sistemas com JSP apresenta como problema principal a mistura de código e XHTML.

Em caso de alterações tanto programadores quanto web-designers devem ser envolvidos. A melhor solução é separar a lógica em classes designadas por Java Beans. Estas classes podem ser acedidas diretamente da página JSP através de uso de propriedades. Como não há programação, a tarefa pode ser realizada pelo web designer diminuindo o impacto tanto da alteração de código quanto ao do layout.

Java Beans são classes Java que obedecem determinadas regras:

- Deve existir um construtor público e sem parâmetros
- Nenhum atributo pode ser público
- Os atributos são acedidos através de métodos públicos `setXxx`, `getXxx` e `isXxx`.

Estas regras determinam um padrão que possibilita o uso de Beans como componentes em ferramentas de desenvolvimento. Estes componentes minimizam a necessidade de programação pois são utilizados através de suas propriedades.

Para utilizar Java Beans em uma aplicação comum deve-se criar um objeto e aceder aos seus métodos; Em JSP, existem Marcas especiais para criação e recuperação de propriedades que não exigem conhecimento de programação:

<jsp:useBean>: Cria objetos Java ou seleciona um objeto que já existe para que seja possível utilizá-lo numa JSP

- Exemplo:

```
<jsp:useBean id = "aluno" class = "Aluno"/>
```

- Esta tag é semelhante a:

```
Aluno aluno = new Aluno();
```

Para ler uma propriedade de um Bean usa-se o atributo **getProperty**

- Exemplo:

```
<jsp:getProperty name = "aluno" property = "nome" />
```

Esta Marca retorna no local em que estiver o valor da propriedade recuperada.

<jsp:setProperty>: Para alterar uma propriedade.

• Exemplo:

```
<jsp:setProperty name = "aluno" property = "nome" value =  
"Maria"/>
```

Inicializar Beans: Caso seja necessário inicializar um Beans usa-se a sintaxe:

```
<jsp:useBean id = "aluno" class = "Aluno">  
    <!-- Inicialização do Bean -->  
</jsp:useBean>
```

O código é executado apenas se o Bean for criado

Propriedades Indexadas: Não existem Marcas específicas para o acesso a propriedades indexadas. Para aceder tais propriedades deve-se usar scriptlets e expressões.

• Exemplo:

```
<%for(int i=0; i<10; i++) { %>  
    <%=aluno.getPropriedade(i)%> <br>  
<% } %>
```

Propriedades e Parâmetros: Os parâmetros (getParameter) podem ser inseridos diretamente em propriedades de Java Beans. Basta usar o nome do parâmetro no atributo param:

```
<jsp:setProperty name="aluno" property="nome" param =  
"nome"/>
```

Para propriedades e parâmetros com o mesmo nome é possível fazer a associação total com o uso de "*"

Exemplo:

```
<jsp:setProperty name="aluno" property="*" />
```

A comparação dos nomes é sensível a maiúsculas e minúsculas!!!!

TagLibs, o JSTL

A possibilidade de criação de tags personalizadas levou a uma proliferação de TagLibs. Embora destinada a objetivos distintos, essa multiplicidade de bibliotecas trouxe duas situações inconvenientes: a duplicação de funcionalidades e a incompatibilidade entre TagLibs diferentes.

Para prover algum grau de padronização e evitar o dispêndio de esforço com tarefas repetitivas, propôs-se a JSTL (JSP Standard Tag Library), uma biblioteca padronizada de tags para atividades comuns em aplicações web, tais como execução de laços, controle de decisão, formatação de texto, acesso a bancos de dados, etc. O objetivo da JSTL é permitir a criação de páginas JSP sem uso de scriptlet (isto é, sem a presença de código Java), empregando exclusivamente a estrutura de tags.

Vantagens: melhora a legibilidade do código, facilita a separação entre a lógica de negócios e a apresentação, provê reuso e possibilita maior grau de automação por parte das ferramentas de autoria.

Desvantagem: são menos flexíveis que os scriptlet e seu uso torna os servlets equivalentemente maiores e mais complexos.

A JSTL prove um conjunto de tags destinadas a realiza tarefas comuns, tais como: repetição, tomada de decisão, seleção, acesso a banco de dados, internacionalização e processamento de documentos XML. Ela se divide em quatro bibliotecas:

Core: tarefas comuns (saída, repetição, tomadas de decisão e seleção), sua URI é <http://java.sun.com/jsp/jstl/core>.

Database Access: acesso aos bancos de dados. A URI para seu uso é <http://java.sun.com/jsp/jstl/sql>.

Formating & I18N: contém tags destinadas a internacionalização e formatação de datas, moedas, valores e outros dados. Sua URI é: <http://java.sun.com/jsp/jstl/fmt>.

XML Processing: processamento de documentos XML. Sua URI é <http://java.sun.com/jsp/jstl/x>.

Existem duas versões de cada uma dessas taglibs: uma aproveita o suporte oferecido pela EL (Expression Language), enquanto a outra usa o mecanismo de expressões comuns. Caso seja necessário, para acessar esse outro conjunto de taglib, basta adicionar o sufixo -rt às URIs, tal como: <http://java.sun.com/jsp/jstl/x-rt>.

Core taglibs

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

Saída básica: Permite a saída de mensagens e expressões e tem a seguinte sintaxe:

```
<c:out value="conteúdo da mensagem"
      default="conteúdo alternativo"
      escapeXML="<true|false>" />
```

*escapeXML indica o uso dos caracteres especiais.

Variáveis: definir ou ajustar o valor de uma variável no escopo indicado.

```
<c:set var="nomeVar"
      target="nomeTarget"
      property="nomeProperty"
      value="exprEL"
      scope="<request|page|session|application>" />
```

*var - nome da variável para armazenar valor.

*target - nome da variável para modificar valor (requer uso de property).

*value - expressão que dá valor a variável.

*scope - indica o escopo da variável var (default=page).

Decisões: avaliação condicional.

```
<c:if Test="condição" Var="nome"
      scope="<request|page|session|application>"
      <!-- código executado quando expressão EL resulta true --%>
</c:if>
```

Repetições: repete a execução de um bloco de código conforme estabelecido por sua variável. Repetição comum:

```
<c:forEach Var="nome"
          Begin="valor de inicio"
          End="valor de final"
          Step="passo">
<!-- corpo a ser repetido conforme controle --%>
</c:forEach>
```

repetição sobre conjuntos:

```
<c:forEach Var="nome"
          Items="<array|Collection|Map|Iterator|Enumeration>"
          varStatus="nome">
<!-- corpo a ser repetido conforme controle --%>
</c:forEach>
```

Exercícios Complementares de fixação:

- 1) Refaça os exercícios feitos em sala.
- 2) Estude o Projeto Produtos (será dado em aula).
- 3) Refaça os exercícios complementares anteriores. Use servlets para receber os parâmetros, e envie para uma página de resposta.
- 4) Refaça os exercícios complementares anteriores. Use sessões e valide os campos utilizando JSP. (conforme foi visto do exemplo sobre sessões).
- 5) Refaça os exercícios complementares anteriores. Use JavaBeans para passar o objeto para uma terceira página e mostre os valores do objeto na página.
- 6) Refaça os exercícios complementares anteriores. Misture o uso de Sessões, JavaBeans e Taglibs.
- 7) Troque as estruturas de controle (if, for, while...) nos exercícios anteriores por taglibs.

Exercícios de Pesquisa:

- 1) Defina Servlets, Javabeans e tagLibs.
- 2) Pesquisa sobre as 4 grandes bibliotecas do JSTL.
- 3) Pesquise sobre Custom Tags.
- 4) Pesquise mais sobre Sessões e sobre Cookies.



Módulo 02:

JDBC- Java Com Banco de Dados

Banco de dados

O banco de dados é um repositório de dados, é onde os dados são armazenados ou persistidos em nosso computador. Para que um programa desenvolvido em uma determinada linguagem possa se comunicar com o SGBD – Sistema de Gerenciamento de Banco de Dados (MySQL, MS Server, Oracle,...) é preciso uma plataforma ou framework de conexão. No caso do Java temos o JDBC, assim como no C#, o ADO.NET e assim por diante. Utilizaremos o JDBC do java e o MySQL como banco de dados.

JDBC (Java Data Base Connectivity)

Java Database Connectivity ou JDBC é um conjunto de classes e interfaces (API) escritas em Java que fazem o envio de instruções SQL para qualquer banco de dados relacional; Api de baixo nível e base para api's de alto nível; Amplia o que você pode fazer com Java; Possibilita o uso de bancos de dados já instalados; Para cada banco de dados há um driver JDBC que pode cair em quatro categorias.

(Wikipedia)

Antes de qualquer coisa, o JDBC precisa conhecer o caminho até o banco de dados. Para isso ele precisa de duas coisas: String de conexão e Driver de conexão. Classe base:

ConnectionFactory.java

```
class ConnectionFactory {  
  
    protected static Connection getConnection() {  
        try {  
            Class.forName("com.mysql.jdbc.Driver");  
            return (Connection) DriverManager.getConnection(  
                "jdbc:mysql://localhost:3306/projetojsp",  
                "usuario", "senha");  
        } catch (ClassNotFoundException ex) {return null;}  
        catch (SQLException s) {return null;}  
    }  
}
```

Desenhada segundo o Padrão de Projeto Factory. Essa classe cria uma conexão com o banco de dados. O método getConnection() retorna uma conexão válida ao banco de dados utilizado.

Na linha : Class.forName("com.mysql.jdbc.Driver"), é carregado o driver do banco de dados utilizado. No caso, o MySQL (lembrando que pode ser utilizado qualquer

banco de dados desde que esse tenha o driver de conexão JDBC adicionado nas bibliotecas do seu projeto.

Repare do código a seguir,

```
return (Connection) DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/nomBanco",
    "usuario",
    "senha");
```

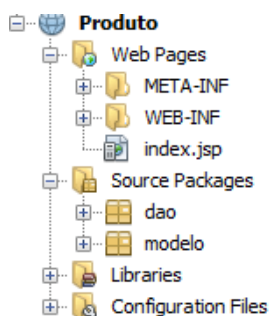
O gerenciador do driver do banco estabelece a conexão uma vez que é informado o caminho/ endereço do banco junto com seu nome, usuário e senha. Uma vez conectado, estamos prontos para a manipulação do banco, isto é, inserir comandos SQL.

Exemplo Passo a Passo: CRUD Básico de Produtos

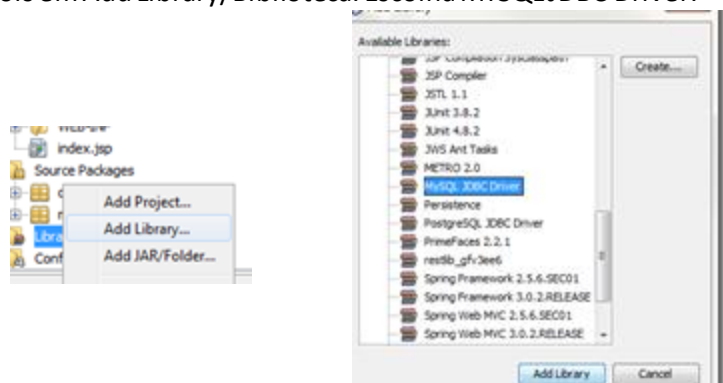
Primeiro, vamos criar nosso banco de dados. O código para criação é o seguinte:

```
create table produtos(
  codigo int auto_increment,
  nome varchar(45) not null,
  valor decimal(4,2),
  primary key(codigo)
);
```

Criamos um novo projeto Netbeans. Em seguida, criamos os pacotes modelo e dao.



Não podemos esquecer de adicionar a biblioteca do Driver MySQL. Clique com o botão direito na pasta Libraries/bibliotecas e depois em Add Library/Biblioteca. Escolha MYSQLJDBC Driver:



Coloque a classe `ConnectionFactory` no pacote `dao`. Não esqueça de configurar a conexão. O banco de dados para nossa aula é ou um servidor local, acreditando que você irá instalar o MySQL localmente. (vá para o site: <http://dev.mysql.com/downloads/mysql/>, instale o MySQL Server Community e caso queira uma interface amigável para lidar com o Mysql, instale o <http://dev.mysql.com/downloads/workbench/5.2.html>). O Usuário padrão é o root e a nossa senha é em branco.

No modelo, criaremos a classe que representa nossa tabela no banco de dados: A classe `Produto`.

Produto.java

```
public class Produto {  
  
    private int codigo;  
    private String nome;  
    private float valor;  
  
    //métodos get e set, hash code e equals...  
  
    @Override  
    public String toString() {  
        return nome + ", R$ " + valor;  
    }  
}
```

Criaremos uma classe para manipular os dados dos produtos. Chamada `ProdutosDao` (mais a frente iremos ver sobre o padrão `dao` e iremos mostrar uma maneira muito mais fácil de fazer essas classes. Você irá gostar!), ela ficará no pacote `dao`.

1º - vamos criar a conexão. Para isso, o construtor de `ProdutoDao` irá se comunicar com a `ConnectionFactory` e pedir uma conexão. (repare que não há necessidade de saber como isso será feito).

ProdutoDao.java

```
public class ProdutoDao {  
    private Connection con; // a conexão com o banco de dados  
    public ProdutoDao() {  
        this.con = new ConnectionFactory().getConnection();  
    }  
}
```

2º - vamos criar o método para inserir os dados.


```
public void adiciona(Produto produto) {
    try {
        // prepared statement para inserção
        PreparedStatement stmt = con.prepareStatement(
            "insert into contatos set "
            + " nome=?, valor=?");
        // seta os valores
        stmt.setString(1, produto.getNome());
        stmt.setFloat(2, produto.getValor());
        // executa
        stmt.execute();
        stmt.close();
    } catch (SQLException e){throw new RuntimeException(e);}
}
```

PreparedStatement é uma container para suas instruções(Statements), colocamos os "?" onde seria os valores a ser passados na instrução SQL (mais sobre no anexo: Noções de SQL).

Após armazenar a instrução SQL, iremos setar os "?" com os valores:

```
// seta os valores
stmt.setInt(1, produto.getCodigo());
stmt.setString(2, produto.getNome());
stmt.setFloat(3, produto.getValor());
```

Agora no falta mandar executar e, é claro, fechar o recurso PreparedStatement:

```
// executa
stmt.execute();
stmt.close();
```

3º Vamos criar os métodos para alterar os dados e remover um dados.

```
public void altera(Produto produto) {
    try {
        PreparedStatement stmt = con.prepareStatement(
            "update produtos set "
            + " nome=?, valor=? where codigoo=?");
        stmt.setString(1, produto.getNome());
        stmt.setFloat(2, produto.getValor());
        stmt.setInt(3, produto.getCodigo());
        stmt.execute();
        stmt.close();
    } catch (SQLException e) {throw new RuntimeException(e); }
}
```

Repare que não muda muita coisa , na verdade só que dessa vez setamos o código do produto a ser alterado (é necessário para evitar alterar a tabela inteira). A próxima é remove, repare que sobrecarregamos o método para suportar duas opções, passar o objeto produto ou somente o valor de seu código.

```
public void remove(Produto produto) {
    remove(produto.getCodigo());
}
public void remove(int id) {
    try {
        PreparedStatement stmt =
            con.prepareStatement(
                "delete from produtos where codigo=?");
        stmt.setInt(3, id);
        stmt.execute();
        stmt.close();
    } catch (SQLException e){throw new RuntimeException(e);}
}
```

4º Precisamos pesquisar os nossos produtos no banco. Podemos fazer de duas formas: apenas um, e uma lista. Também podemos adicionar filtros. Esse é o último tópico do CRUD básico (Create – criar/insert, Read – ler/pesquisar/select, Update – atualizar/alterar/update, Delete – deletar/remover/delete).

Primeiro uma lista: criamos o método getList():

```
public List<Produto> getList() {
    try {
        PreparedStatement stmt =
            this.con.prepareStatement(
                "select * from produtos");
        //o execute query retorna um ResultSet
        ResultSet rs = stmt.executeQuery();

        //lista onde ficarão nossos produtos
        List<Produto> produtos = new ArrayList<Produto>();

        Produto produto;
        //enquanto houver registros no ResultSet
        while (rs.next()) {
            produto = new Produto();
            produto.setCodigo(rs.getInt("codigo"));
            produto.setNome(rs.getString("nome"));
            produto.setValor(rs.getFloat("valor"));
            // adicionando o objeto à lista
        }
    }
}
```

```
        produtos.add(produto);
    }
    rs.close();
    stmt.close();
    return produtos;
} catch (SQLException e) {
    return new ArrayList<Produto>();
}
}
```

O executeQuery irá retornar um ResultSet, isso é, um conjunto de resultados em forma de tabela. O laço While percorre por todo esse conjunto até que não haja mais elementos.

Para cada elemento pegamos seus valores e colocamos em nosso objeto, depois adicionamos na lista.

```
produto.setCodigo(rs.getInt("codigo"));
produto.setNome(rs.getString("nome"));
produto.setValor(rs.getFloat("valor"));
```

para capturar apenas um produto, é necessário receber seu código e colocar na instrução SQL. Repare que é um pouco mais simples. Apenas perguntamos se exista algum elemento no ResultSet, ou seja, se o produto pesquisado existe ou não.

```
public Produto get(int codigo) {
    try {
        PreparedStatement stmt =
            this.con.prepareStatement(
                "select * from produtos where codigo=?");
        stmt.setInt(1, codigo);

        ResultSet rs = stmt.executeQuery();

        Produto produto;
        if(rs.next()){
            produto = new Produto();
            produto.setCodigo(rs.getInt("codigo"));
            produto.setNome(rs.getString("nome"));
            produto.setValor(rs.getFloat("valor"));
        }else{
            produto = new Produto();
        }
        rs.close();
        stmt.close();
    }
```

```
        return produto;
    } catch (SQLException e){return new Produto();}
}
```

Com isso nossa classe ProdutoDao já pode ser utilizado não só pelas páginas JSP, como páginas JSF, ou interface GUI Swing do java, ou até mesmo um Programa simples de console.

Na página index.jsp, podemos criar um pequeno exemplo:

```
<%
ProdutoDao dao = new ProdutoDao();

Produto p = new Produto();
p.setNome("bolacha");
p.setValor(2.34f);
dao.adiciona(p);

Produto p1 = new Produto();
p1.setNome("bola");
p1.setValor(3.50f);
dao.adiciona(p1);

out.println("</br>Lista de produtos</br>");
for(Produto atual : dao.getList())
    out.println(atual+"<br/>");

out.println("</br>--Produto 1--</br>");
Produto p1 = dao.get(1);
out.println(p1);
out.println("</br>--Produto 2--</br>");
Produto p2 = dao.get(2);
out.println(p2);
%>
```

---Lista de produtos---
bolacha, R\$ 2.34
bola, R\$ 3.5

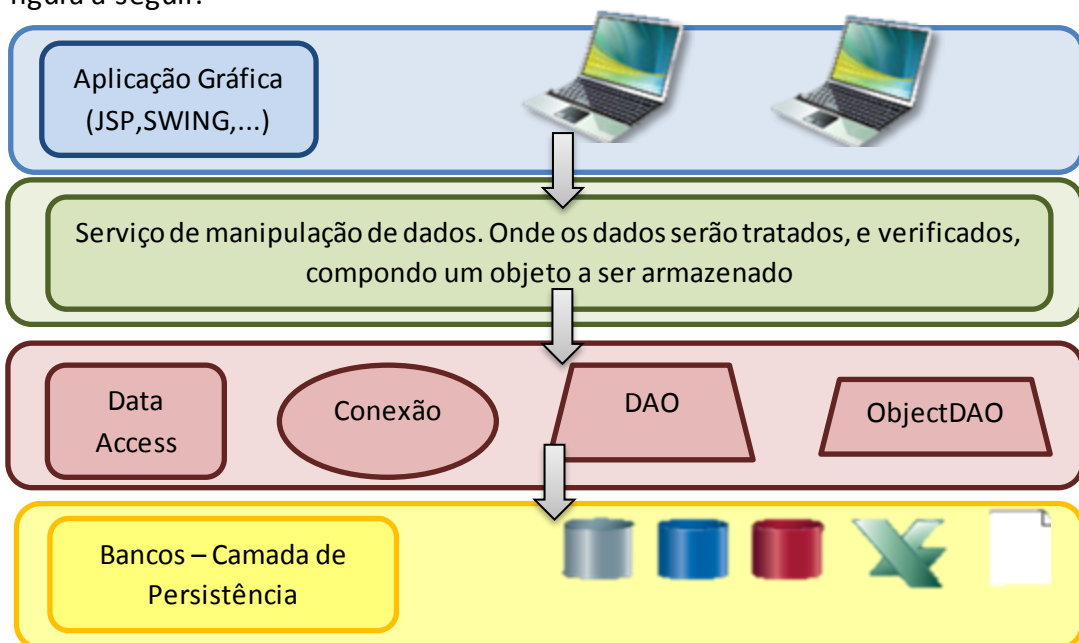
---Produto 1---
bolacha, R\$ 2.34
---Produto 2---
bola, R\$ 3.5

Tudo certo, porém temos alguns problemas... Essa classe é muito grande e complexa. A única responsabilidade do ProdutoDao deveria ser preparar o código SQL. Será que não existe um meio menos trabalhoso de executar os códigos, umas vez q esses mesmos códigos serão utilizados em várias outras classes? E ainda não vimos o pior: classes relacionadas entre si.

Para isso, a próxima seção foi criada para nos dar a luz que precisamos: A classe Dao. Uma classe complexa, porem 100% reutilizável, e o melhor, não precisa alterar nada. Nessa próxima seção iremos explicar melhor o Padrão Dao.

Padrão DAO

O JDBC oferece uma interface com vários comandos para serem utilizados para manipular SQL e o banco. Mas utiliza-los no meio da aplicação pode ser cansativo, e difícil que detectar onde estão os erros. Fora que caso precise mudar algo será extremamente difícil reparar. Para facilitar esse nosso trabalho, foi criado um padrão de projetos adotado por toda a comunidade Java: O DAO – Data Access Object, ou Objeto de Acesso a Dados. Ele visa criar uma “película” entre a aplicação Java e a conexão com o banco de dados. Entre outras palavras, a aplicação Java só precisa mandar os dados a serem manipulados e dizer o que ela quer com ele (inserir, atualizar, consultar, deletar, ou seja, as 4 operações básicas de um sistema CRUD – Create, Read, Update e Delete). Como mostra a figura a seguir.



Vamos agora criar uma classe chamada DAO, que programa esse padrão. Nela, será concentrado todo o código JDBC para manipulação de dados, assim como o acesso a Classe Conectar.

Dao.java

```
import java.sql.*;

public class Dao {
    private Connection conn;
    private PreparedStatement ps;
    private ResultSet rs;
```

```
Dao(){ try {conn = getConnection();}
      catch (SQLException ex) { ex.printStackTrace();}    }

private Connection getConnection() throws SQLException {
    if (null == conn || conn.isClosed())
        conn = ConnectionFactory.getConnection();
    return conn;
}

private Statement getStatement() throws SQLException{
    return getConnection().createStatement();
}

private PreparedStatement getStatement(String statement)
throws SQLException {
    return getConnection().prepareStatement(statement);
}

private void makeStatement(String query, Object... params)
throws SQLException {
    ps = getStatement(query);
    for (int i = 0; i < params.length; i++) {
        ps.setObject(i + 1, params[i]);
    }
}

protected final int executeCommand(String query, Object...
params) throws SQLException {
    makeStatement(query, params);
    int result = ps.executeUpdate();
    ps.close();
    return result;
}

protected final ResultSet executeQuery(String query, Object...
params) throws SQLException {
    makeStatement(query, params);
    rs = ps.executeQuery();
    return rs;
}

protected final boolean contain(String tableName,String
field,Object value) throws SQLException {
    return contain(tableName, new Tupla(field, value));
}
```

```
protected final boolean contain(String tableName, Tupla...
tuplas) throws SQLException {
    String query = "select * from " + tableName + " where ";
    int length = tuplas.length;
    Object[] values = new Object[length];
    for (int i = 0; i < length; i++) {
        query += tuplas[i].field + "=?";
        values[i] = tuplas[i].value;
        if (i < length - 1) query += " and ";
    }
    rs = executeQuery(query, values);
    boolean result = rs.next();
    rs.close();
    return result;
}

protected final int getIDFromLastInsert(String table) {
    try {
        int ultimoId = -1;
        ResultSet rs = executeQuery("select max(id) as idMax
from " + table);
        if (rs.next()) ultimoId = rs.getInt("idMax");
        rs.close();
        return ultimoId;
    } catch (SQLException ex) {return -1;}
}

public final int getNextId(String tableName) throws
SQLException {
    rs = executeQuery("select max(id) from " + tableName);
    rs.next();
    Object o = rs.getObject(1);
    Integer toreturn = (o == null) ? 1 : ((Integer) o) + 1;
    rs.close();
    return toreturn;
}

public class Tupla {
    private String field;
    private Object value;
    public Tupla(String field, Object value) {
        this.field = field;
        this.value = value;
    }
}
}
```

Como pode ver, ela é uma classe, mais complexa, que executa qualquer código SQL que for passada a ela. Agora vamos ver mais a fundo.

Esse método cria uma conexão com a classe Conecta. Ele verifica, pelo padrão de projeto 'Singleton', se a conexão já existe, caso exista, ele não criará outra. Isso faz com que cada Java Virtual Machine tenha apenas uma única conexão, evitando estourar o limite do banco de dados.

```
private Connection conn;
private PreparedStatement ps;
private ResultSet rs;

Dao(){ try {conn = getConnection();}
      catch (SQLException ex) { ex.printStackTrace();}    }

private Connection getConnection() throws SQLException {
    if (null == conn || conn.isClosed())
        conn = ConnectionFactory.getConnection();
    return conn;
}
```

Ambos os métodos a seguir criam Statements, ou instruções semi prontas, para rodar SQL. O mais utilizado é o `getStatement(String statement)`, ele recebe uma String que representa o código SQL e a vincula na conexão

```
private Statement getStatement() throws SQLException{
    return getConnection().createStatement();
}
private PreparedStatement getStatement(String statement)
throws SQLException {
    return getConnection().prepareStatement(statement);
}
```

Esses dois métodos a seguir recebem a String SQL desejada e podem receber vários ou nenhum parâmetro do tipo Object, ou seja, elas podem receber parâmetros que identificam valores a serem colocadas na SQL, como os dados de uma inserção, ou atualização, ou remoção e parâmetros de pesquisa.

```
private void makeStatement(String query,
                           Object... params) throws SQLException {
    ps = getStatement(query);
    for (int i = 0; i < params.length; i++) {
        ps.setObject(i + 1, params[i]);
    }
}

protected final int executeCommand(String query,
```



```
        Object... params) throws SQLException {
    makeStatement(query, params);
    int result = ps.executeUpdate();
    ps.close();
    return result;
}

protected final ResultSet executeQuery(String query,
        Object... params) throws SQLException {
    makeStatement(query, params);
    rs = ps.executeQuery();
    return rs;
}
```

MakeStatment é especial, ele vai produzir um Statement e setar seus valores, é o único ponto em comum dos métodos executeQuery e executeCommand.

O executeQuery é um método próprio para consultas, e retorna um resultSet, ou seja, um conjunto de dados, ou mais abruptamente, a tabela do banco. Usada com o comando **select**, pode receber parâmetros para uma pesquisa filtrada, ou não é essa a função dos ‘...’ depois de Object, eles indicam que se trata de um parâmetro variável, ou seja, posso passar, um, dois, três, mil, ou nenhum.

Esse parâmetro é tratado como um vetor, um conjunto de dados. Por isso usamos o for para passear pelo vetor, e setarmos as posições dos pparâmetros. Como veremos mais a frente, onde forem colocados parâmetros na SQL, podemos simplesmente coloca um “?”, e depois passarmos o objeto cujo valor representa o parâmetro. Por exemplo:

```
Select * from alunos where código=?
```

A passamos o parâmetro aluno.codigo, para recebermos uma tabela que contem o aluno cujo código é igual ao passado.

O executeCommand faz a mesma coisa que o executeQuery, mas não retorna nenhuma tabela de dados.

```
executeCommand( "insert into aluno (nome,telefone) values (?,?)",
aluno.nome, aluno.telefone );
```

Repare que a ordem dos parâmetros deve ser a mesma que a ordem das suas “?” (posições).

Os próximos são muito utilizados para capturar chaves primárias.

```
protected final int getIDFromLastInsert(String table,String
keyField) {
    try {
        int ultimoId = -1;
        ResultSet rs = executeQuery("select max(+"keyField" +)
as idMax from " + table);
        if (rs.next()) ultimoId = rs.getInt("idMax");
        rs.close();
        return ultimoId;
    } catch (SQLException ex) { return -1;}
}

public final int getNextId(String tableName) throws
SQLException {
    rs = executeQuery("select max(id) from " + tableName);
    rs.next();
    Object o = rs.getObject(1);
    Integer toreturn = (o == null) ? 1 : ((Integer) o) + 1;
    rs.close();
    return toreturn;
}
```

getIDFromLastInsert, recupera a chave do registro que acabou de ser colocado no banco, deve ser utilizada para classes relacionadas, como veremos na Classe Venda e Item do Projeto a seguir. É utilizado quando a chave é gerada automaticamente (auto_increment).

getNextID captura a próxima chave disponível para ser utilizada, é utilizado quando a chave primária não é gerada automaticamente e há necessidade de passar seu valor na instrução insert into.

O contain, um pouco complexo. É uma pesquisa para saber se o registro já existe no banco. Sua primeira versão feita por mim. Ele pedia um campo, um valor e o nome da tabela, e apenas podia verificar se existia um registro cujo aquele campo tinha aquele valor. Surgia, as vezes, a necessidade de verificar mais campos e valores. Então criei uma classe interna chamada Tupla com os atributos field e value. Ou seja, agora é possível enviar uma lista de Tuplas pela método (da mesma forma que o executeQuery/Command recebem um a lista de objetos) e verificar mais de um campo.

```
protected final boolean contain(String tableName,
    Tupla... tuplas) throws SQLException {
    //criação da SQL
    String query = "select * from " +tableName+ " where ";
    int length = tuplas.length;
    Object[] values = new Object[length];
    // esse for lê as tuplas, coloca o nome do campo na
    instrução e o valor da tupla um vetor object
    for (int i = 0; i < length; i++) {
        query += tuplas[i].field + "=?";
        values[i] = tuplas[i].value;
        if (i < length - 1) query += " and ";
    }
    //o executeQuery receb a instrução e o vetor de valores
    rs = executeQuery(query, values);
    boolean result = rs.next();
    rs.close();
    return result;
}
```

A classe Dao que fiz, contém ambas as implementações.

```
protected final boolean contain(String tableName,
    String field, Object value) throws SQLException{
    ResultSet rs = executeQuery(
        "select * from "+tableName
        +" where "+field+"=?",value);
    boolean result = rs.next();
    rs.close();
    return result;
}
```

A classe tupla é coloca dentro da classe Dao.

```
public class Tupla {
    private String field;
    private Object value;
    public Tupla(String field, Object value) {
        this.field = field;
        this.value = value;
    }
}
```

Essa classe pode ser usada, para facilitar o trabalho de programar todas as conexões, abrir e fechar sessão, setar objetos nas SQL, prepara statements, entre outras.

Para utilizar essa classe, basta criar classes que estendem a ela, usando herança. Assim, criamos uma classe, que pode utilizar os métodos de Dao como seus, sem se preocupar com como estão programados.

Vamos modificar o nosso projeto Produto utilizando a Dao. Aproveitaremos para iniciar a construções de um mini projeto, reaproveitando o banco e a tabela produtos, e as classes já criadas.

Projeto Loja ITTraining

Começaremos alterando a classe ProdutoDao da seção retrasada, utilizando os conceitos aprendidos na seção passada.

Serão os mesmos 4 métodos, porem as implementações serão diferentes e bem mais “Orientadas a Objetos”.

Para que os métodos da classe Dao fiquem disponíveis, podemos utilizar herança ou composição. Escolhemos Herança, pois como os métodos da Classe Dao estão protegidos contra sobrescrita (modificador **final** e **private**), a herança não nos trás desvantagens.

ProdutoDao.java

```
public class ProdutoDao extends Dao{
```

1º Adicionar:

```
public void adiciona(Produto produto) {  
    try {  
        executeCommand("insert into produtos set "  
            + "nome=?, valor=?",  
            produto.getNome(), produto.getValor());  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Repare que só pe preciso chamar o método executeCommand, passar a instrução e os valores na ordem exata e pronto. Dao já nos faz o serviço “sujo”.

2º Alterar e Remover

```
public void altera(Produto produto) {  
    try {  
        executeCommand("update produtos set "  
            + " nome=?, valor=? where codigo=?",  
            produto.getNome(), produto.getValor(),  
            produto.getCodigo());  
    } catch (SQLException e){throw new RuntimeException(e);}  
}  
public void remove(Produto produto){remove(produto.getCodigo());}  
public void remove(int id) {
```

```
try {
    executeCommand("delete from produtos where
codigo=?",id);
} catch (SQLException e) {throw new RuntimeException(e);}
}
```

3º get() e getLista()

Repare que ficou um pouco menor. Ainda necessitamos do ResultSet.

```
public List<Produto> getLista() throws SQLException {
    ResultSet rs = executeQuery("select * from produtos");
    List<Produto> produtos = new ArrayList<Produto>();
    Produto produto;
    while (rs.next()) {
        produto = new Produto();
        produto.setCodigo(rs.getInt("codigo"));
        produto.setNome(rs.getString("nome"));
        produto.setValor(rs.getFloat("valor"));
        produtos.add(produto);
    }
    rs.close();
    return produtos;
}

public Produto get(int codigo) throws SQLException {
    ResultSet rs = executeQuery(
        "select * from produtos where codigo=?",
        codigo);
    Produto produto;
    if(rs.next()){
        produto = new Produto();
        produto.setCodigo(rs.getInt("codigo"));
        produto.setNome(rs.getString("nome"));
        produto.setValor(rs.getFloat("valor"));
    }else{produto = new Produto();}
    rs.close();
    return produto;
}
```

Podemos melhorar um pouco ambos os métodos criando um método chamado populate(resultSet rs):

```
private Produto p;
public Produto populate(ResultSet rs) throws SQLException {
    p = new Produto();
    p.setCodigo(rs.getInt("codigo"));
}
```

```
p.setNome(rs.getString("nome"));
p.setValor(rs.getFloat("valor"));
return p;
}
```

Agora vamos modificar os get() e getLista():

```
public List<Produto> getLista() throws SQLException {
    ResultSet rs = executeQuery("select * from produtos");
    List<Produto> produtos = new ArrayList<Produto>();
    while (rs.next()) produtos.add(populate(rs));
    rs.close();
    return produtos;
}

public Produto get(int codigo) throws SQLException {
    ResultSet rs = executeQuery(
        "select * from produtos where codigo=?",codigo);
    Produto produto = rs.next()?
        populate(rs) : new Produto();
    rs.close();
    return produto;
}
```

Bem melhor, não? Agora vamos criar nossas Páginas. A index será nosso menu inicial. Dela partiram as outras páginas.

Index.jsp

```
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
        <title>Loja ITTraining</title>
    </head>
    <body>
        <jsp:include page="topo.html"/>

        <input type="button" value="Produtos >>"
            onClick="location.href='produtos'"/>

        <jsp:include page="footer.html"/>
    </body>
</html>
```

Vamos criar uma página chamada **produtos.jsp**. Nela vamos criar uma tabela com os produtos cadastrados no banco.

produtos.jsp

```
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
    charset=ISO-8859-9">
    <title> Loja ITTraining </title>
  </head>
  <body>
    <h3>Lista de Produtos Cadastrados</h3>
    <table>

      <tr>
        <td>Código</td>
        <td>Nome</td>
        <td>Valor (R$)</td>
      </tr>

      <tr>...</tr>

    </table>
  </body>
</html>
```

O resultado ainda não é o esperado. Vamos consultar o banco e montar uma tabela com os produtos cadastrados.

```
<%@page import="dao.ProdutoDao"%>
<%@page import="modelo.Produto"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
    charset=ISO-8859-9">
    <title> Loja ITTraining</title>
  </head>
  <body>
    <jsp:include page="topo.html"/>

    <h3>Lista de Produtos Cadastrados</h3>
```



```
<table>

  <tr>
    <td>Código</td>
    <td>Nome</td>
    <td>Valor (R$)</td>
  </tr>

  <% for(Produto atual : new ProdutoDao().getLista()){ %>
  <tr>
    <td><%= atual.getCodigo() %></td>
    <td><%= atual.getNome() %></td>
    <td><%= atual.getValor() %></td>
  </tr>
  <% } %>
</table>

<jsp:include page="footer.html"/>
</body>
</html>
```

Lista de Produtos Cadastrados

Código	Nome	Valor (R\$)
1	bolacha	2.34
2	bola	3.5

Próximo Passo, vamos criar uma página para inserir os dados. Crie a página **novoProduto.jsp**.

novoProduto.jsp

```
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
  charset=ISO-8859-9">
    <title> Loja ITTraining </title>
  </head>
  <body>
    <jsp:include page="topo.html"/>

    <h3>Cadastrando Novo Produto</h3>
```

```
<form name="novoProduto" method="post"
action="controlProduto">
    <input type="hidden" name="operacao" value="novo"/>
    Nome: <input type="text" name="nome"/><br/>
    Valor: <input type="text" name="valor"/><br/>
    <input type="submit" value="Enviar >>"/>
</form>

<jsp:include page="footer.html"/>
</body>
</html>
```

Adicione seguinte código na página **produtos.jsp** para que possa acessar essa página:

```
<input type="button" value="Novo >>"
onClick="location.href='novoProduto.jsp'"/>
```

Nessa página, passamos o conteúdo do formulário para um servlet chamado `controlProduto` e no campo operação, colocamos o valor de **novo**. Indicando um novo produto. Iremos criar esse servlet de controle. Ele terá 3 métodos: novo, altera, deleta. Fora os métodos normais visto nas seções anteriores.

controlProduto.java

```
public class controlProduto extends HttpServlet {

    private Produto p;
    private ProdutoDao dao = new ProdutoDao();

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

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

        try {
            String operacao = request.getParameter("operacao");
            if ("novo".equalsIgnoreCase(operacao)) {
                novo(request, response);
            } else if ("altera".equalsIgnoreCase(operacao)) {
                altera(request, response);
            } else if ("deleta".equalsIgnoreCase(operacao)) {
                deleta(request, response);
            }
        }
    }
}
```

```
        response.sendRedirect("produtos.jsp");
    } finally {out.close();}
}

private void novo(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {

    p = new Produto();
    p.setNome(request.getParameter("nome"));
    p.setValor(
        Float.parseFloat(request.getParameter("valor")));
    dao.adiciona(p);
}

private void altera(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {

    p = new Produto();
    p.setCodigo(
        Integer.parseInt(request.getParameter("cod")));
    p.setNome(request.getParameter("nome"));
    p.setValor(
        Float.parseFloat(request.getParameter("valor")));
    dao.altera(p);
}

private void deleta(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {

    int cod = Integer.parseInt(
        request.getParameter("cod"));
    dao.remove(cod);
}
```

O servlet encaminha os dados para os métodos respectivos de acordo com o parâmetro operação.

upProduto.jsp

```
<%@page import="dao.ProdutoDao"%>
<%@page import="modelo.Produto"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
```

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
    <title> Loja ITTraining </title>
  </head>
  <body>
    <jsp:include page="topo.html"/>
    <%
      int id= Integer.parseInt(request.getParameter("cod"));
      Produto p = new ProdutoDao().get(id);
    %>
    <h3>Alterando o Produto</h3>
    <form name="produto" method="post" action="controlProduto">
      <input type="hidden" name="operacao" value="altera"/>
      Código: <input type="hidden" name="cod"
        value="<%= p.getCodigo() %>"/><br/>
      Nome: <input type="text" name="nome"
        value="<%= p.getNome() %>"/><br/>
      Valor: <input type="text" name="valor"
        value="<%= p.getValor() %>"/><br/>
      <input type="submit" value="Enviar >>"/>
    </form>
    <jsp:include page="footer.html"/>
  </body>
</html>
```

Teremos que passar um parâmetro com o código do produto a ser alterado para que a página faça uma pesquisa e pegue o produto. Uma vez feito isso, os dados do produto serão colocados no formulário para poder serem alterados, com exceção do código que deverá ficar escondido(hidden) para não ser alterado indevidamente.

delProduto.jsp

```
<%@page import="dao.ProdutoDao"%>
<%@page import="modeLo.Produto"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
    <title> Loja ITTraining </title>
  </head>
  <body>
    <jsp:include page="topo.html"/>
```

```

<%
    int id= Integer.parseInt(request.getParameter("cod"));
    Produto p = new ProdutoDao().get(id);
%>
<h3>Deseja realmente deletar o produto a seguir?</h3>
<form name="produto" method="post" action="controlProduto">
    <input type="hidden" name="operacao" value="deleta"/>
    Código: <input type="hidden" name="cod"
        value="<%= p.getCodigo() %>"/><br/>
    Nome:&nbsp;<%= p.getNome() %><br/>
    Valor:&nbsp;<%= p.getValor() %><br/>
    <input type="button" value="Voltar"
        onclick="history.go(-1)" /> |
    <input type="submit" value="Deletar >>"/>
</form>
<jsp:include page="footer.html"/>
</body>
</html>

```

Para podermos passar o código devido, vamos alterar a tabela da página **produtos.jsp**, acrescentando uma linha nela, que terá as opções de alterar e deletar.

produtos.jsp

```

<%@page import="dao.ProdutoDao"%>
<%@page import="modelo.Produto"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
        charset=ISO-8859-9">
        <title>Loja ITTraining</title>
    </head>
    <body>
        <jsp:include page="topo.html"/>
        <input type="button" value="Novo >>"
            onClick="location.href='novoProduto.jsp'"/>
        <h3>Lista de Produtos Cadastrados</h3>
        <table>
            <tr>
                <td>Código</td>
                <td>Nome</td>
                <td>Valor (R$)</td>
                <td></td>
            </tr>

```

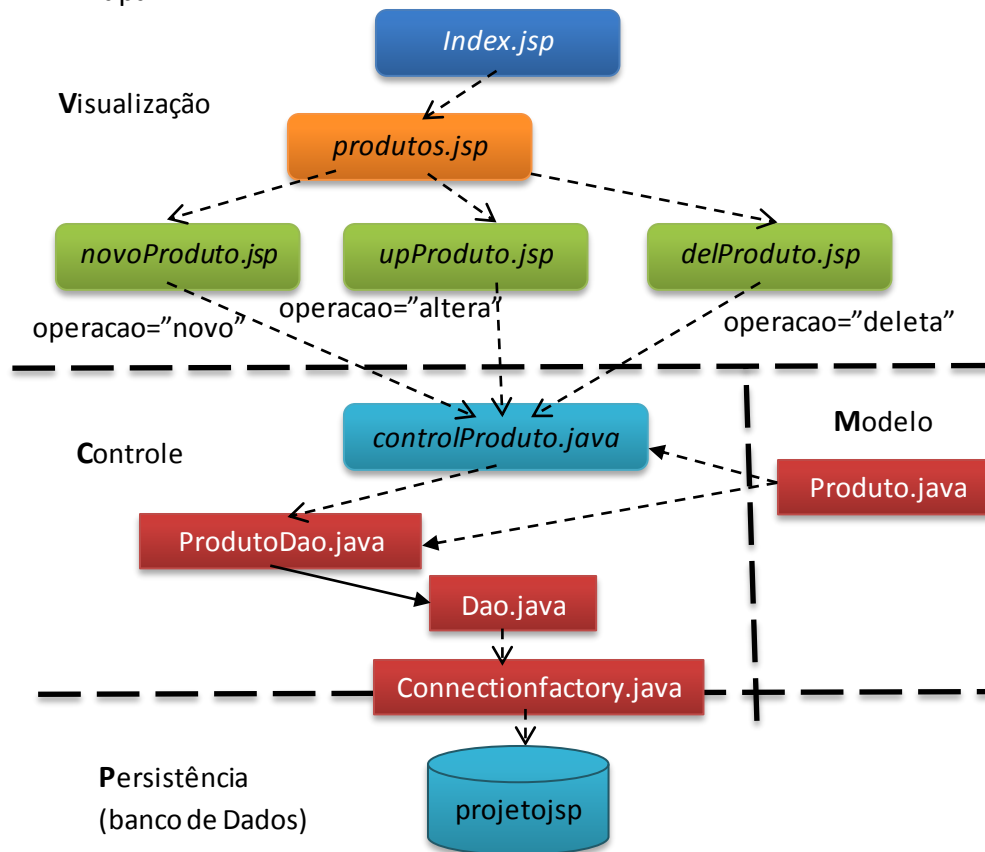
```
<% for(Produto atual : new ProdutoDao().getLista()){ %>
<tr>
<td><%= atual.getCodigo() %></td>
<td><%= atual.getNome() %></td>
<td><%= atual.getValor() %></td>
<td>
<input type="button" value="Alterar ">
onClick="location.href='upProduto.jsp?cod=
<%=atual.getCodigo()%>'"/>
<input type="button" value="Deletar ">
onClick="location.href='delProduto.jsp?cod=
<%=atual.getCodigo()%>'"/>
</td>
</tr>
<% } %>
</table>
<jsp:include page="footer.html"/>
</body>
</html>
```

Lista de Produtos Cadastrados

Código	Nome	Valor (R\$)		
1	bolacha	2.34	<input type="button" value="Alterar >"/>	<input type="button" value="Deletar >"/>
2	bola	3.5	<input type="button" value="Alterar >"/>	<input type="button" value="Deletar >"/>
5	coxinha	2.34	<input type="button" value="Alterar >"/>	<input type="button" value="Deletar >"/>
6	velas	2.34	<input type="button" value="Alterar >"/>	<input type="button" value="Deletar >"/>

A seguir, temos um mapa das páginas do projeto e o caminho que o usuário irá percorrer (sem saber):

Mapa:



Esse é o “desenho” ou “forma” (se preferir) do nosso projeto. Repare que separamos as responsabilidades de cada membro. Cada um faz uma parte das ações nesse “ecossistema”.

Configurando páginas de erro com Servlets

Eventualmente podem ocorrer acessos a páginas que não existam e também exceções. Vamos configurar para caso aconteça, o servlet container (Tomcat) redirecione para páginas própria, que irão avisar ao usuário do problema.

Crie duas páginas JSP: **404.jsp** e **error.jsp**.

404.jsp

```

<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>

```

```
<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
  <title>Relatório de Erro</title>
</head>
<body>
  <jsp:include page="topo.html"/>

  <h2>A página que você acessou não existe.</h2>
  <input type="button" value="Voltar"
    onClick="history.go(-1)"/>

  <jsp:include page="footer.html"/>
</body>
</html>
```

error.jsp

```
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
    <title>Relatório de Erro</title>
  </head>
  <body>
    <jsp:include page="topo.html"/>

    <h2>Um erro foi encontrado.<br/>
      A operação não pôde ser concluída.</h2>
    <input type="button" value="Voltar"
      onClick="history.go(-1)"/>

    <jsp:include page="footer.html"/>
  </body>
</html>
```

Não há necessidade de explicar as páginas que acabamos de criar, seu conteúdo fala por si. Vamos tratar erros de página inexistente ou exceções ocorridas nas operações.

Para isso vamos ao **web.xml**. Lembra dele? A xml de configurações no projeto. Nela se encontram as definições de todas as bibliotecas e servlets, assim como filtros dentre outros. Ela fica na pasta **WEB-INF**.

Web.xml

```
<?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>controlProduto</servlet-name>
        <servlet-class>servlets.controlProduto</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>controlProduto</servlet-name>
        <url-pattern>/controlProduto</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

Como podemos ver, quando um servlet é criado, o Netbeans cria uma definição. Uma para declarar o servlet:

```
<servlet>
    <servlet-name>controlProduto</servlet-name>
    <servlet-class>servlets.controlProduto</servlet-class>
</servlet>
```

E outra para declarar o mapeamento dele, isto é, qual será o nome acessível dele por todas as páginas e outros servlets no sistema.

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

Vamos definir dois conjuntos de tags, uma para 404 e outro para exceções:

```
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/error.jsp</location>
</error-page>
<error-page>
    <error-code>404</error-code>
    <location>/404.jsp</location>
</error-page>
```

Agora, quando nosso sistema lançar alguma exceção ou o usuário tentar acessar uma página inexistente dentro do site, essas páginas irão aparecer.

Tente modificar uma url , por exemplo, tente acessar produtoses.jsp...



Projeto Loja

A página que você acessou não existe.

[Voltar](#)

ITTraining - Curso Java WEB

Tente gerar alguma exceção: exemplo, da página upProduto remova o parâmetro cod na url.



Projeto Loja

**Um erro foi encontrado.
A operação não pôde ser concluída.**

[Voltar](#)

ITTraining - Curso Java WEB

Validando os dados e prevendo erros

Você deve ter reparado que deixamos os formulário de alteração e adição de produtos com o mesmo nome. Pois ambos são relativamente iguais e vamos usar um código Java Script para prevenir erros.

Altere os botões dos formulário: no lugar de “submit” coloque “button” e adicione o evento onClick():

```
<input type="button" value="Enviar">>" onclick=
    "validarProduto()"/>
```

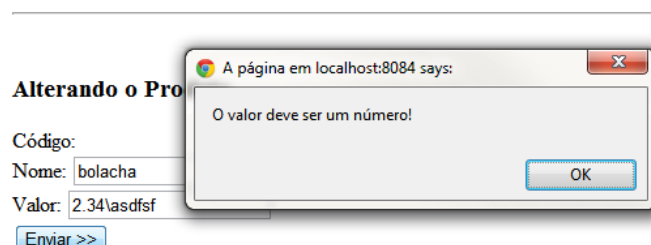
Crie o arquivo JavaScript: **validação.js**

```
function validarProduto(){
    var nome = document.forms.produto.nome;
    if(nome.value==""){
        alert("Nome não pode estar em branco!");
        nome.focus();
    }
    var valor = document.forms.produto.valor;
    if(valor.value==""){
        alert("O valor não pode estar em branco!");
        valor.focus();
    }
    if(isNaN(valor.value)){
        alert("O valor deve ser um número!");
        valor.focus();
    }
    document.forms.produto.submit();
}
```

Adicione o vinculo com a página javaScript:

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

Projeto Loja

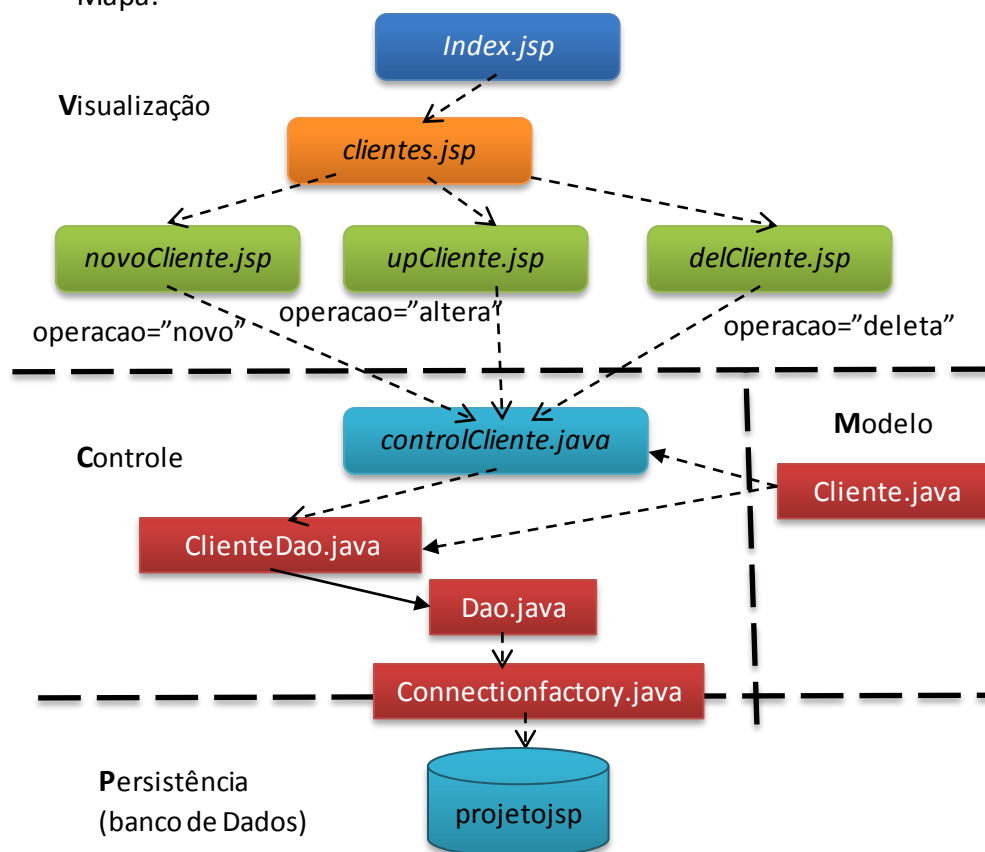


Exercício do projeto

Crie agora a parte dos clientes. O código SQL é o seguinte:

```
create table clientes(
codigo int auto_increment,
cpf varchar(15) not null,
nome varchar(45) not null,
telefone varchar(45) not null,
endereço varchar(150),
primary key(codigo)
);
```

Mapa:

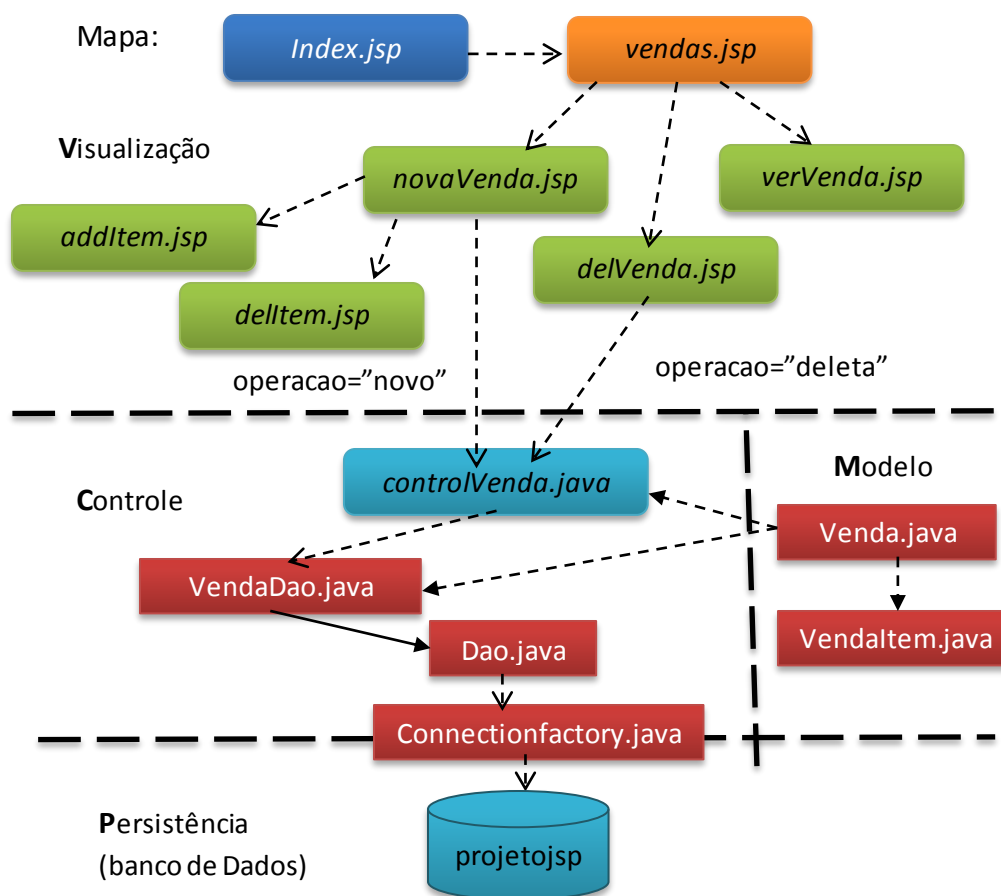


! A correção desse exercício está na pasta Projeto Loja juntamente com todos os códigos fontes desse projeto. Não esqueça de lembrar seu instrutor de lhe entregar as fontes de todos os projetos visto durante o curso. !

E quando as tabelas estão relacionadas?

Tente fazer a parte das vendas. Os códigos SQL para as tabelas do banco são esses a seguir:

```
create table vendas(
codigo int auto_increment,
cliente int not null,
data_venda date,
total decimal(5,2),
primary key(codigo)
);
create table itens(
produto int not null,
venda int not null,
qtde int,
primary key (produto,venda)
);
```



Exercícios Complementares de Fixação:

1) Fornecedores (cadastrar fornecedores e alterar a classe produto para que guarde um fornecedor do produto). Um fornecedor tem: código, nome, cnpj, email, endereço. Altere a tabela produto para que guarde o código do fornecedor que o fornece. Para cadastrar esse código, no formulário de adição e alteração de produto, insira um combobox com os fornecedores cadastrados. Dica: o valor das opções da caixa serão os códigos dos fornecedores e a palavra que irá aparecer na seleção será o nome do fornecedor.

2) (Fazer o desafio proposto) Crie um sistema para controle dos produtos/fornecedores. Listando todos eles e indicando em vermelho os produtos com estoque baixo (menor que 5).

Dica: Primeiramente crie o banco de dados “estoque”. Com as seguintes tabelas:

Fornecedor:

Codigo: Integer

Nome: String

Telefone: String

Produto:

Codigo: Integer

Nome: String

Quantidade: String

Valor: String

Fornecedor_id: Integer

Após isso, crie uma página onde será listado (em forma de tabela) todos os produtos com seus respectivos fornecedores.

Deverá ser Listado:

Nome do Produto | Quantidade | Valor | Nome do Fornecedor | Telefone do Fornecedor

3) Refaça o exercício 6 na primeira leva de exercícios complementares, dessa vez, crie um banco de dados e faça um sistema JSP com Servlets para realizar as seguintes operações para cada uma das tabelas: Inserir novo, Atualizar, Remover, Criar uma página inicial com uma tabela contendo os dados do banco. Faça um Sistema para cada uma das tabelas segundo a lista a seguir:

Sistema de venda:

a. Produto

b. Fornecedor

c. Cliente

Sistema de Escola:

- a. Escola
- b. Curso
- c. Aluno

4) Controle de Alunos: Crie um banco de dados “escola” com a tabela:

Aluno:

- a. Código : String
- b. Nome: String
- c. Idade: String

Crie uma página para listar todos os alunos.

Crie um sistema para Inserir, Atualizar e Deletar.

Crie uma outra página que mostre apenas os alunos de uma determinada idade recebida pelo usuário na página anterior.

Crie uma outra página que separe os alunos por série e mostre várias tabelas de série com seus alunos, de acordo com a relação a seguir:

- série 1 - 6 a 10 anos;
- série 2 – 11 a 15 anos;
- série 3 – 16 a 18 anos;
- série 4 – a partir de 19 anos;

6) Crie um sistema para gerar as notas finais dos alunos. Dica:

Crie um banco de dados com as tabelas:

Alunos :

Código: Integer;

Nome: String;

Notas:

Cod_aluno: Integer;

P1, p2, p3: Float;

Existirá uma tela para a entrada das notas dos alunos e seus respectivos pesos.

Crie uma Página que mostre uma tabela com os alunos e suas notas e sua média final. Caso a média seja menor que 5, escreva a média e o nome do aluno , na tabela, em vermelho, caso contrário normal (em preto);

Persistência de Objetos com JPA, EJB 3.0 e Hibernate

Com a popularização do Java em ambientes corporativos, logo se percebeu que grande parte do tempo do desenvolvedor era gasto na codificação de queries SQL e no respectivo código JDBC responsável por trabalhar com elas.

Além de um problema de produtividade, algumas outras preocupações aparecem: onde devemos deixar nossas queries? Dentro de um arquivo .java? Dentro do DAO? O ideal é criar um arquivo onde essas queries sejam externalizadas.

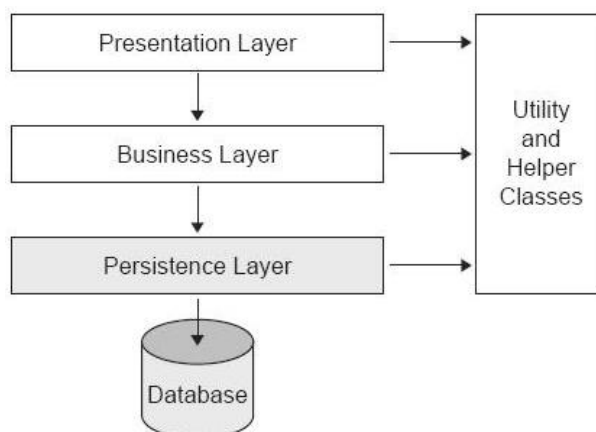
Há ainda a mudança do paradigma. A programação orientada a objetos difere muito do esquema entidade relacional e precisamos pensar das duas maneiras para fazer um único sistema. Esse buraco entre esses dois paradigmas gera bastante trabalho: a todo momento devemos “transformar” objetos em linhas e linhas em objetos, sendo que essa relação não é um-para-um.

Ferramentas para auxiliar nesta tarefa tornaram-se popular entre os desenvolvedores Java e são conhecidas como ferramentas de mapeamento objeto-relacional (ORM).

Java Persistence API, chamada apenas de JPA, é uma API padrão do Java para persistência que deve ser implementada por frameworks que queiram seguir o padrão.

A JPA define um meio de mapeamento objeto-relacional para objetos Java simples e comuns (POJOs), denominados *beans de entidade*. Diversos frameworks de mapeamento objeto/relacional como o Hibernate implementam a JPA. Também gerencia o desenvolvimento de entidades do Modelo Relacional usando a plataforma nativa Java SE e Java EE.

O JPA é um framework utilizado na camada de persistência para o desenvolvedor ter uma maior produtividade, com impacto principal num modo para controlarmos a persistência dentro de Java. Pela primeira vez, nós, desenvolvedores temos um modo "padrão" para mapear nossos objetos para os do Banco de Dados. Persistência é uma abstração de alto-nível sobre JDBC.



Para utilizar JPA, deve-se escolher um provedor JPA, ou seja, uma implementação da API. A JPA é uma API para frameworks, tendo-se como implementação de referência o *Oracle TopLink Essentials*. Existem outros provedores JPA no mercado, como o *Hibernate Entity Manager*, *Bea Kodo* eo *Apache JPA*.

Para desenvolver um sistema em JPA é necessário:

- Mapear classes de negócio em entidades de um banco;
- Definir uma unidade de persistência;
- Definir um *managed datasource* (conexão com um banco de dados);

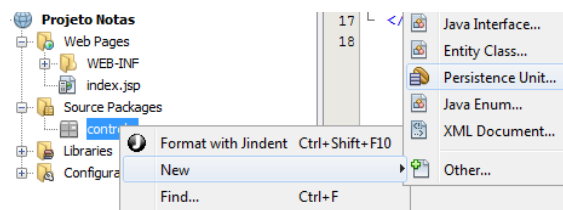
Toda classe do modelo que será armazenada no banco deve ser um Pojo: JavaBean, com métodos gets e sets para atributos e construtor sem argumentos. Isso porque Hibernate faz reflexão de código.

Há várias formas de mapear classes de negócio a entidades de um banco de dados;

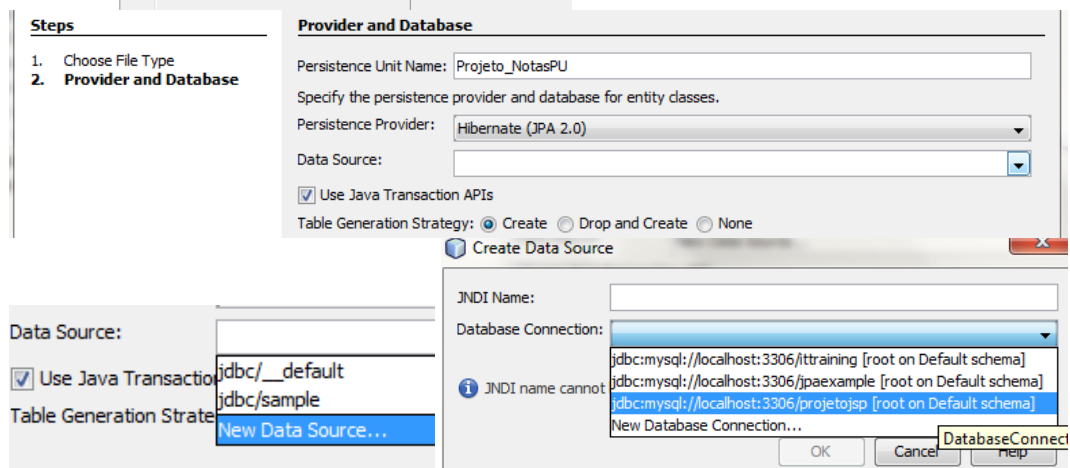
- É possível fazer com xml;
- Porém, é mais prático usar anotações;

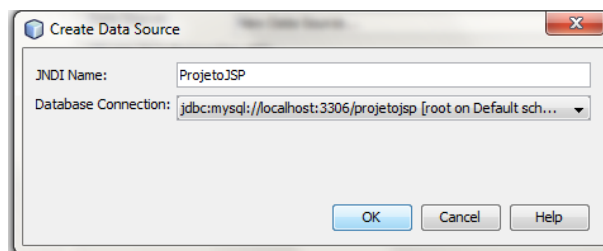
Projeto Escola – utilizando JPA, EJB e Hibernate para um desenvolvimento produtivo e veloz

Primeiramente, vamos criar um novo projeto chamado Projeto Notas. Nele, crie uma nova Persistence Unit (Unidade de Persistência).

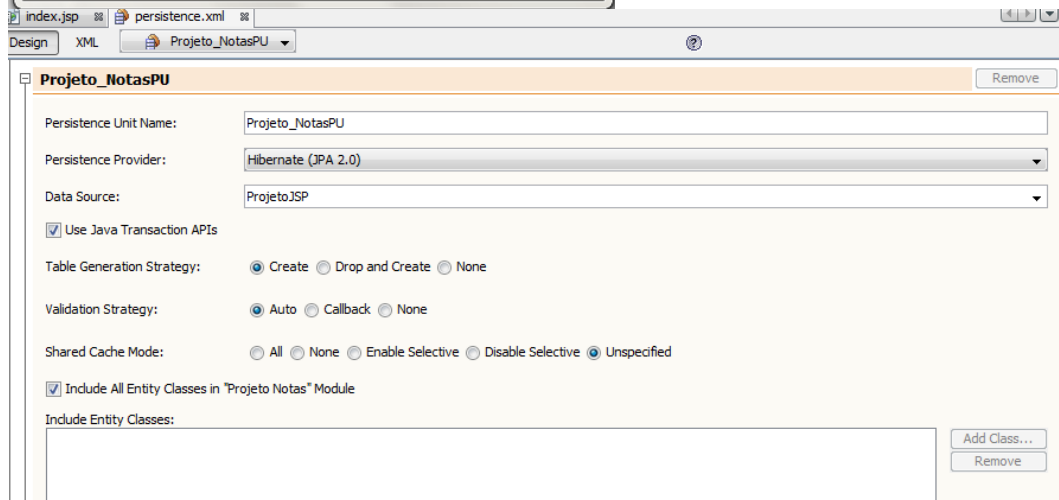


Vamos escolher o nosso ORM : Hibernate (JPA 1.0). e a conexão com o banco de dados será com o projetojsp feito em aula.

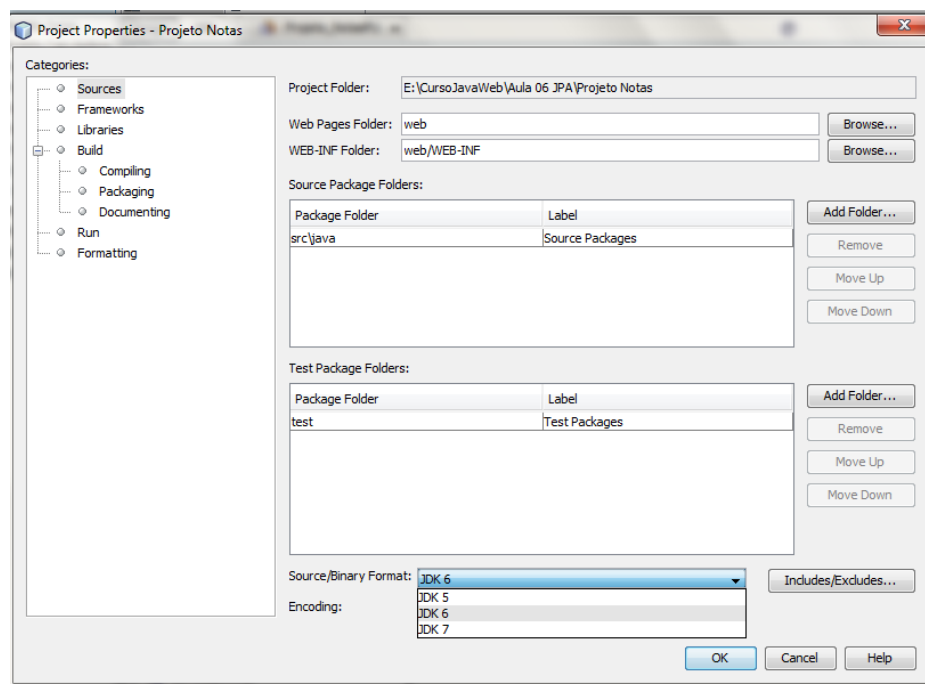




Uma vez definida a conexão com o banco, é criado um documento xml. Onde ficarão todas as configurações de nossa unidade de persistência.



Geralmente, o editor , por padrão coloca nosso projeto Java Web utilizando o JDK5, vamos trocar por JDK6, pois é mais recomendado ao se trabalhar com frameworks JEE. Clique com o botão direito no projeto e depois em propriedades.



Por padrão, o NetBeans cria uma documentação própria para a JPA, algumas vezes incompleta, então é bom verificar se a sua foi criada com quase a mesma cara da documentação a seguir.

Persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0"
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="ProjetoNotasPU" transaction-
type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <non-jta-data-source>ProjetoJSP</non-jta-data-source>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      <property name="hibernate.connection.username"
value="root"/>
      <property name="hibernate.connection.driver_class"
value="com.mysql.jdbc.Driver"/>
      <property name="hibernate.connection.password"
value="<<SENHA>>"/>
      <property name="hibernate.connection.url"
value="jdbc:mysql://localhost:3306/<<NOME BANCO>>"/>
      <property name="hibernate.cache.provider_class"
value="org.hibernate.cache.NoCacheProvider"/>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

Nessa configuração, definimos as propriedades do ORM Hibernate, como o nome do usuário, senha, driver e caminho do banco, exatamente como no JDBC. Será usado cache. O tipo de mapeamento: **top down** ou **bottom up**. Se você quiser que apareça o código SQL criado pelo Hibernate no console, muito usado durante o desenvolvimento do projeto e se você quer o SQL formatado.

Top down: A partir de um modelo e de classes desenvolvidas em Java, gera-se o esquema do banco: usa-se a ferramenta *hbm2ddl*.

Bottom up: A partir de um modelo de dados e esquema de banco de dados pré-existente, deve-se usar engenharia reversa para se ter o esquema em hibernate xml ou classes Java anotadas. Usa-se, para esse último caso, hbm2java

Vamos criar o nosso banco de dados contendo a tabela curso.

```
CREATE TABLE `curso` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `nome` varchar(45) NOT NULL,  
  PRIMARY KEY (`id`)  
)
```

Agora vamos criamos o pacote **modelo** e dentro dele as nossas classes que serão transformadas em entidades:

```
public class Curso {  
  
    private int id;  
    private String nome;  
  
    //getters e setters  
}
```

O classe curso é um POJO, eu seja, uma classe simples em java, também definida no javaBeans. Ela contém atributos que representam as colunas do banco bem encapsulados e métodos get e set.

Do jeito que está, a classe não poderá ser persistida (a menos que se use o JDBC, como usado até agora). Ara garantir que ela seja mapeada pela nossa unidade de persistência, vamos usar **annotations**.

Curso.java

```
import java.io.Serializable;  
import javax.persistence.*;  
  
@Entity(name="curso")  
public class Curso implements Serializable {  
  
    @Id  
    @GeneratedValue(strategy= GenerationType.AUTO)  
    @Column(name="id")  
    private int id;  
  
    @Column(name="nome")  
    private String nome;  
  
    //getters e setters  
}
```

Explicações: **@Entity** é usado para indicar que uma classe será persistida, ela transforma a classe em uma entidade. O atributo **name** da anotação define qual é o nome da tabela no banco de dados a qual essa entidade representa.

```
@Entity
@Table(name="curso")
public class Curso implements Serializable {
```

Repare que essa classe deve implementar `Serializable`, isso indica que ela pode ser serializada, ou persistida, armazenada, guardada, como queira chamar.

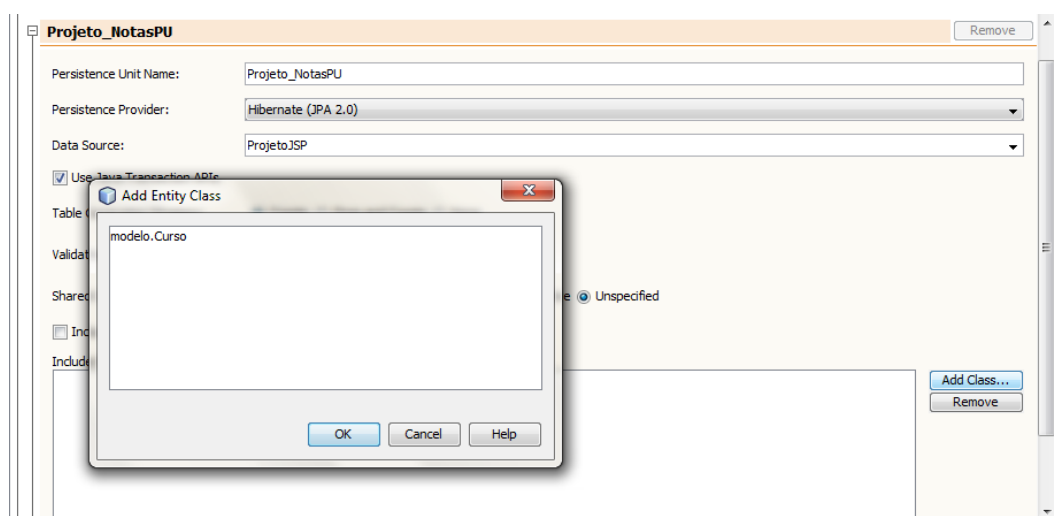
Toda Entity deve ter um atributo marcado como chave primária, um e somente um (caso uma tabela tenha mais de uma chave primária, é utilizada em chave Embebed).

```
@Id
@GeneratedValue(strategy= GenerationType.AUTO)
@Column(name="id")
private int id;
```

`GeneratedValue` indica que a chave primária é **auto_increment**, `Column` significa que ela é uma coluna da tabela. Todo atributo que representar uma coluna deverá ter a anotação `Column`.

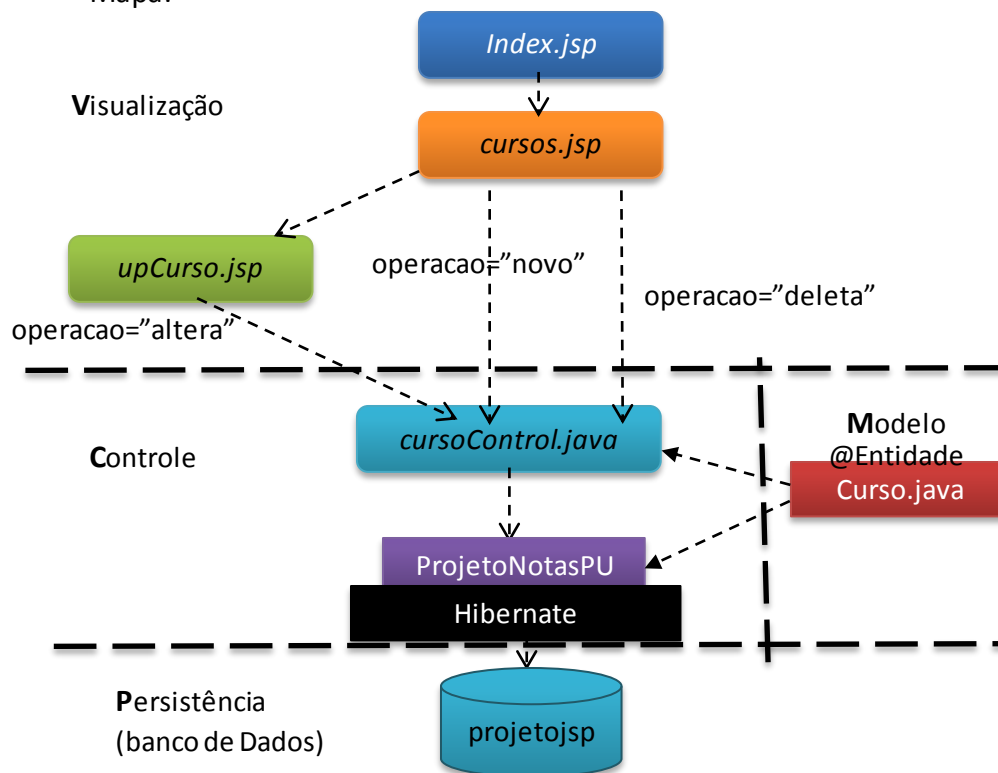
```
@Column(name="nome")
private String nome;
```

Uma vez anotada, nossa unidade de persistência deverá saber que a classe deverá ser mapeada. Vamos a ate ela e em `Add Class`, ou `Adicionar Classe`, escolhemos a classe a ser mapeada.



Nos iremos adotar quase o mesmo modelo do JDBC, mas repare que não haverá necessidade de Criarmos um DAO. (a menos que você queira, aqui para facilitar, não será necessário), tanto que essa é uma discussão: JPA ira tornar o padrão DAO obsoleto? Você já irá entender o porque.

Mapa:



Uma vez anotada e mapeada, agora vamos criar o controle. Todo controle precisara criar um objeto chamado EntityManager, que nada mais é que um gerenciador de entidades. Para cria-lo vamos aproveitar o padrão Factory já criado: **EntityManagerFactory**. Recebe a configuração da unidade de persistência e cria uma transação segura.

CursoControl.java

```
private EntityManagerFactory factory =
    Persistence.
        createEntityManagerFactory("ProjetoNotasPU");
private EntityManager em = factory.createEntityManager();
```

Muito cuidado, o nome colocado no CreateEntityManagerFactory deve ser O MESMO DA UNIDADE DE PERSISTENCIA, MUITO CUIDADO.

Vamos criar o nosso controle com o mesmo jeito do projeto anterior, um servlet que receberá o parâmetro **operação** afim de saber qual tarefa será realizada.

```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {

        String operacao = request.getParameter("operacao");
        if("novo".equalsIgnoreCase(operacao))
            salvar(request);
        else if("altera".equalsIgnoreCase(operacao))
            alterar(request);
        else if("deleta".equalsIgnoreCase(operacao))
            excluir( Integer.parseInt( request
                .getParameter("id")));
        response.sendRedirect("cursos.jsp");
    } finally {
        out.close();
    }
}

public Curso populate(HttpServletRequest request){
    Curso c= new Curso();
    if(null!= request.getParameter("id"))
        c.setId(Integer.parseInt(request.getParameter("id")));
    c.setNome(request.getParameter("nome"));
    return c;
}
```

A diferença está aqui...sem SQL, isso mesmo...**SEM SQL**. Você abre uma transação, faz o que tem que fazer, e depois confirma caso de certo, ou anula caso algum erro ocorra.

```
public void salvar(HttpServletRequest request) {
    try{
        em.getTransaction().begin();
        em.persist(populate(request));
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }
}
```

```
public void alterar(HttpServletRequest request) {
    try{
        em.getTransaction().begin();
        em.merge(populate(request));
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }
}

public void excluir(int id) {
    try{
        em.getTransaction().begin();
        em.remove(get(id));
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }
}

public Curso get(int id){
    return em.find(Curso.class,id);
}

public List<Curso> getProdutos(){
    List l = em.createQuery(
        "from curso").getResultList();
    LinkedList<Curso> ll = new LinkedList<Curso>();
    for(Object o : l) ll.add((Curso) o);
    return ll;
}
```

Perceba que só é preciso cinco palavras, cinco métodos. É fácil e rápido. Ao contrário do JDBC, não há necessidade de sql.

Vamos criar mais duas tabelas, aluno e notas.

```
CREATE TABLE `aluno` (
  `ra` int(11) NOT NULL AUTO_INCREMENT,
  `nome` varchar(45) NOT NULL,
  `email` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`ra`)
)

CREATE TABLE `notas` (
  `ra` int(11) NOT NULL DEFAULT '0',
  `codCurso` int(11) NOT NULL DEFAULT '0',
  `nota_1` double DEFAULT NULL,
  `nota_2` double DEFAULT NULL,
  `nota_3` double DEFAULT NULL,
```



```
`nota_4` double DEFAULT NULL,
PRIMARY KEY (`ra`,`codCurso`)
)
```

Começaremos pela classe Aluno:

Aluno.java

```
@Entity(name="aluno")
public class Aluno implements Serializable{

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    @Column(name="ra")
    private int ra;

    @Column(name="nome")
    private String nome;

    @Column(name="email")
    private String email;

    //getters e setters
}
```

Uma vez feita, não se esqueça de adicionar na persistenceUnit, a gora, o seu controle:

AlunoControl.java

```
public class alunoControl extends HttpServlet {

    private EntityManagerFactory factory =
        Persistence
            .createEntityManagerFactory("ProjetoNotasPU");
    private EntityManager em = factory
        .createEntityManager();

    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            String operacao = request.getParameter("operacao");
            if("novo".equalsIgnoreCase(operacao))
                salvar(request);
        }
    }
}
```

```
        else if("altera".equalsIgnoreCase(operacao))
            alterar(request);
        else if("deleta".equalsIgnoreCase(operacao))
            excluir(Integer.parseInt(request.getParameter("ra")));
        response.sendRedirect("alunos.jsp");
    } finally {
        out.close();
    }
}

public Aluno populate(HttpServletRequest request){
    Aluno c= new Aluno();
    if(null!= request.getParameter("ra"))
        c.setRa(Integer.parseInt
            (request.getParameter("ra")));
    c.setNome(request.getParameter("nome"));
    c.setEmail(request.getParameter("email"));
    return c;
}

public void salvar(HttpServletRequest request) {
    try{
        em.getTransaction().begin();
        em.persist(populate(request));
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }
}

public void alterar(HttpServletRequest request) {
    try{
        em.getTransaction().begin();
        em.merge(populate(request));
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }
}

public void excluir(int id) {
    try{
        em.getTransaction().begin();
        em.remove(get(id));
        em.remove(a);
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }
}
```

```
}

public Aluno get(int id){
    return em.find(Aluno.class,id);
}
public List<Aluno> getList(){
    List l = em.createQuery("from aluno").getResultList();
    LinkedList<Aluno> ll = new LinkedList<Aluno>();
    for(Object o : l) ll.add((Aluno) o);
    return ll;
}
```

O mapa das páginas será do mesmo jeito dos cursos.

Agora vamos ter cuidado ao adicionar a nossa nova classe Notas.

Notas.java

```
@Entity(name="notas")
public class Notas implements Serializable {

    @Id
    @OneToOne
    @JoinColumn(name="ra",nullable=false)
    private Aluno aluno;

    @Id
    @OneToOne
    @JoinColumn(name="codCurso",nullable=false)
    private Curso curso;

    @Column(name="nota_1")
    private float nota_1;
    @Column(name="nota_2")
    private float nota_2;
    @Column(name="nota_3")
    private float nota_3;
    @Column(name="nota_4")
    private float nota_4;

    //getters e setters
}
```

1º A tabela notas contém uma chave composta. **JPA não tolera mais de um Id por entidade.** Como vamos resolver? Criando uma classe que represente essa **chave composta**.

NotasPK.java

```
@Embeddable
public class NotasPk implements Serializable{

    @OneToOne
    @JoinColumn(name="ra",nullable=false)
    private Aluno aluno;

    @OneToOne
    @JoinColumn(name="codCurso",nullable=false)
    private Curso curso;

    public NotasPk() {
    }

    public NotasPk(Aluno aluno, Curso curso) {
        this.aluno = aluno;
        this.curso = curso;
    }
}
```

A anotação **Embeddable** faz com que essa classe possa ser inserida em uma entidade como sendo seu Id ou seu complemento. Para finalizar o processo, devemos criar um objeto dessa classe dentro da entidade desejada e marcar como sendo **EmbeddedId**.

Notas.java

```
@Entity(name="notas")
public class Notas implements Serializable {

    @EmbeddedId
    private NotasPk chave;

    @Column(name="nota_1")
    private float nota_1;
    @Column(name="nota_2")
    private float nota_2;
    @Column(name="nota_3")
    private float nota_3;
    @Column(name="nota_4")
    private float nota_4;

    //getters e setters
}
```

2º Devemos garantir que Aluno e curso tenha uma lista de notas que pertence a eles. Por exemplo: um aluno tem várias notas e um curso tem várias notas de vários alunos.

Aluno e Curso agora terão que ter uma coleção de notas. Vamos marcar essa coleção com a anotação **OneToMany**. (um para muitos – um aluno tem várias notas, uma nota pertence a um aluno ou um curso tem várias notas, uma npta pertence a um curso).

Vamos também definir a **JoinColumn**, a coluna na tabela notas que faz a junção das tabelas. Veja primeiro na classe **Aluno**:

```
@OneToMany
@JoinColumn(name="ra",updatable=false)
private List<Notas> notas = new LinkedList<Notas>();
```

Agora na classe Curso:

```
@OneToMany()
@JoinColumn(name="codCurso",updatable=false)
private List<Notas> notas = new LinkedList<Notas>();
```

3º Ao deletarmos um curso ou um aluno (última decisão a ser feita, apenas em casos extremo) devemos garantir que as notas referentes a eles seja apagadas.

Nos controles, antes de removermos os objetos, vamos criar uma Query que deleta na tabela notas onde a chave do objeto for igual a desse objeto em questão. Veja na Classe alunoControl, vamos criar a query: "**DELETE FROM notas where ra=?**", delete na notas onde o ra pertencer a esse aluno e na curso a mesma coisa: "**DELETE FROM notas where codCurso=?**".

Veja na **cursoControl**:

```
public void excluir(int id) {
    try{
        em.getTransaction().begin();
        Curso c = get(id);
        Query q = em.createQuery(
            "DELETE FROM notas where codCurso=?")
            .setParameter(1,c.getId());
        q.executeUpdate ();
        em.remove(c);
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }
}
```

```
}
```

Veja na `alunoControl`:

```
public void excluir(int id) {
    try{
        em.getTransaction().begin();
        Aluno a = get(id);
        Query q = em.createQuery (
            "DELETE FROM notas where ra=?")
            .setParameter(1,a.getRa());
        q.executeUpdate ();
        em.remove(a);
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }
}
```

Repare que mesmo usando o JPA e Hibernate, podemos utilizar Querys SQL como no JDBC, porem é mais simples que no JDBC. Muito cuidado com o uso delas!! O JPA tenta garantir as referencias do banco de dados, é muito importante isso. Pois uma referencia a um registro inexistente pode para seu sistema ate que você a localiza e a destrua.

Agora vamos terminar de criar nossas páginas e nosso CRUD.

cursos.jsp

[Voltar](#)

Adicionando novo curso:

Nome: [Adicionar >>](#)

Lista de Cursos Cadastrados

Código Nome

1	Java Orientado a Objetos	Deletar	Alterar	Visualizar Notas
2	Java Desenvolvimento Web	Deletar	Alterar	Visualizar Notas
5	PHP	Deletar	Alterar	Visualizar Notas

```
<%@page import="controle.cursoControl"%>
<%@page import="modelo.Curso"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
    <title>JSP Page</title>
  </head>
  <body>
    <jsp:include page="topo.html"/>
    <input type="button" value="Voltar"
      onClick="location.href='index.jsp'"/>
    <h4>Adicionando novo curso:</h4>
    <form name="curso" action="cursoControl" method="post">
      <input type="hidden" name="operacao" value="novo"/>
      Nome:<input type="text" value="" name="nome"/>
      <input type="button" value="Adicionar >>"
        onClick="document.forms.curso.submit()"/>
    </form>
    <hr>
    <h3>Lista de Cursos Cadastrados</h3>
    <table>
      <tr>
        <td>Código</td>
        <td>Nome</td>
      </tr>
      <% for (Curso c:new cursoControl().getLista()) {%>
      <tr>
        <td><%= c.getId()%></td>
        <td><%= c.getNome()%></td>
        <td><input type="button" value="Deletar"
          onClick=
"location.href='cursoControl?operacao=deleta&id=<%=c.getId()%>'" /
></td>
        <td><input type="button" value="Alterar"
          onClick=
"location.href='upCurso.jsp?id=<%=c.getId()%>'" /></td>
        <td><input type="button"
          value="Visualizar Notas"
          onClick="location.href='viewCurso.jsp?id=
<%=c.getId()%>'" /></td>
      </tr>
      <% }%>
    </table>

    <jsp:include page="footer.html"/>
  </body>
```

</html>

upCursos.jsp

Alterando o curso:

Nome:

```
<%@page import="modelo.Curso"%>
<%@page import="controle.cursoControl"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
    <title>JSP Page</title>
  </head>
  <body>
    <jsp:include page="topo.html"/>
    <% Curso c = new cursoControl()
      .get(Integer.parseInt(request.getParameter("id")));%>
    <input type="button" value="Voltar"
      onClick="history.go(-1)"/>
    <h4>Alterando o curso:</h4>
    <form name="curso" action="cursoControl" method="post">
      <input type="hidden" name="operacao" value="altera"/>
      <input type="hidden" name="id" value="<%= c.getId()%>"/>
      Nome:<input type="text" value="<%= c.getNome()%>"
name="nome"/>
      <input type="button" value="Atualizar >>"
        onClick="document.forms.curso.submit()"/>
    </form>

    <jsp:include page="footer.html"/>
  </body>
</html>
```

viewCurso.jsp

[Voltar](#)

Notas de alunos

Curso:Java Orientado a Objetos

Lucas : 9.0 | 9.0 | 9.0 | 9.0

```
<%@page import="modelo.Notas"%>
<%@page import="modelo.Aluno"%>
<%@page import="controle.alunoControl"%>
<%@page import="controle.cursoControl"%>
<%@page import="modeloCurso"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
    charset=ISO-8859-9">
    <title>JSP Page</title>
  </head>
  <body>
    <jsp:include page="topo.html"/>
    <input type="button" value="Voltar"
      onClick="history.go(-1)"/>
    <h4>Notas de alunos</h4>
    <% Curso c = new
cursoControl().get(Integer.parseInt(request.getParameter("id")));
    %>
    Curso:<%= c.getNome()%><br/>
    <% for (Notas n : c.getNotas()) {%>
    <% Aluno a = n.getChave().getAluno();%>
    <b><%= a.getNome()%></b> :
    <%= n.getNota_1()%>
    | <%= n.getNota_1()%>
    | <%= n.getNota_1()%>
    | <%= n.getNota_1()%><br/>
    <% }%>
    <jsp:include page="footer.html"/>
  </body>
</html>
```

alunos.jsp

Adicionando novo aluno:

Nome: E-mail: **Lista de Alunos Cadastrados**

Ra Nome E-mail

1 Lucas biamonlucky@hotmail.com

```

<%@page import="modelo.Aluno"%>
<%@page import="controle.alunoControl"%>
<%@page import="controle.cursoControl"%>
<%@page import="modelo.Curso"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
    <title>JSP Page</title>
  </head>
  <body>
    <jsp:include page="topo.html"/>
    <input type="button" value="Voltar"
      onClick="location.href='index.jsp'"/>
    <h4>Adicionando novo aluno:</h4>
    <form name="aluno" action="alunoControl" method="post">
      <input type="hidden" name="operacao" value="novo"/>
      Nome:<input type="text" value="" name="nome"/><br/>
      E-mail:<input type="text" value="" name="email"/>
      <input type="button" value="Adicionar >>"
        onClick="document.forms.aluno.submit()"/>
    </form>
    <hr>
    <h3>Lista de Alunos Cadastrados</h3>
    <table>
      <tr>
        <td>Ra</td>
        <td>Nome</td>
        <td>E-mail</td>
      </tr>

```

```

    <% for (Aluno a : new alunoControl().getList()) {%>
    <tr>
        <td><%=a.getRa()%></td>
        <td><%=a.getNome()%></td>
        <td><%=a.getEmail()%></td>
        <td><input type="button" value="Deletar"

onClick="location.href='alunoControl?operacao=deleta&ra=<%=a.getR
a()%>'"/></td>
        <td><input type="button" value="Alterar"

onClick="location.href='upAluno.jsp?ra=<%=a.getRa()%>'"/></td>
        <td><input type="button" value="Visualizar Notas"

onClick="location.href='viewAluno.jsp?ra=<%=a.getRa()%>'"/></td>
    </tr>
    <% }%>
</table>

<jsp:include page="footer.html"/>
</body>
</html>

```

upAluno.jsp

Alterando o aluno:

Nome: E-mail:


```

<%@page import="modelo.Aluno"%>
<%@page import="controle.alunoControl"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
        <title>JSP Page</title>
    </head>
    <body>

```

```

<jsp:include page="topo.html"/>
<% Aluno a = new alunoControl()
    .get(Integer.parseInt(request.getParameter("ra")));%>
<input type="button" value="Voltar"
    onClick="history.go(-1)"/>
<h4>Alterando o aluno:</h4>
<form name="aluno" action="alunoControl" method="post">
    <input type="hidden" name="operacao" value="altera"/>
    <input type="hidden" name="ra" value="<%=a.getRa()%>"/>
    Nome:<input type="text" value="<%=a.getNome()%>"
name="nome"/><br/>
    E-mail:<input type="text" value="<%=a.getEmail()%>"
name="email"/>
    <input type="button" value="Atualizar >>"
    onClick="document.forms.aluno.submit()"/>

</form>

<jsp:include page="footer.html"/>
</body>
</html>

```

viewAluno.jsp

Notas do aluno:

Aluno: Lucas
 E-mail: biasonlucky@hotmail.com

Java Orientado a Objetos

--Notas:--
 | 9.0 | 9.0 | 9.0 | 9.0

Java Desenvolvimento Web

--Notas:--
 | 3.0 | 3.0 | 3.0 | 3.0

```

<%@page import="modelo.Notas"%>
<%@page import="modelo.Aluno"%>
<%@page import="controle.alunoControl"%>
<%@page import="controle.cursoControl"%>
<%@page import="modelo.Curso"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
<head>

```

```

<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-9">
<title>JSP Page</title>
</head>
<body>
<jsp:include page="topo.html"/>
<input type="button" value="Voltar"
onClick="history.go(-1)"/>
<h4>Notas do aluno:</h4>
<% Aluno a = new alunoControl()
.get(Integer.parseInt(request.getParameter("ra")));%>
Aluno:<%= a.getNome()%><br/>
E-mail:<%= a.getEmail()%><br/>
<% for (Notas n : a.getNotas()) {%>
<% Curso c = n.getChave().getCurso();%>
<p><b><%= c.getNome()%></b><br/>
--Notas:--<br/>
| <%= n.getNota_1()%>
| <%= n.getNota_1()%>
| <%= n.getNota_1()%>
| <%= n.getNota_1()%><br/></p>
<hr>
<% }%>
<jsp:include page="footer.html"/>
</body>
</html>

```

notas.jsp

Adicionando novo aluno:

Aluno:

Curso:

--Notas:--

| 01=

| 02=

| 03=

| 04=

Notas do curso : Java Orientado a Objetos

Aluno: Lucas | 9.0 | 10.0 | 10.0 | 9.8 |

Notas do curso : Java Desenvolvimento Web

Aluno: Lucas | 3.0 | 3.0 | 3.0 | 3.0 | [Deletar >>](#)

Notas do curso : PHP

```
<%@page import="modelo.Notas"%>
<%@page import="modelo.Aluno"%>
<%@page import="controle.alunoControl"%>
<%@page import="controle.cursoControl"%>
<%@page import="modelo.Curso"%>
<%@page contentType="text/html" pageEncoding="ISO-8859-9"%>
<!DOCTYPE html>
<html>
  <head><meta http-equiv="Content-Type" content="text/html;
  charset=ISO-8859-9">
    <title>JSP Page</title>
  </head>
  <body><jsp:include page="topo.html"/>
    <input type="button" value="Voltar"
      onClick="location.href='index.jsp'"/>
    <h4>Adicionando novo aluno:</h4>
    <form name="nota" action="notasControl" method="post">
      <input type="hidden" name="operacao" value="novo"/>
      Aluno:
      <select name="aluno">
        <option value=""></option>
        <% for (Aluno a : new alunoControl().getList()) {%>
          <option value="<%= a.getRa()%>"><%=
a.getNome()%></option>
        <%}%>
      </select><br/>
      Curso:
      <select name="curso">
        <option value=""></option>
        <% for (Curso c : new cursoControl().getLista()) {%>
          <option value="<%= c.getId()%>"><%=
c.getNome()%></option>
        <%}%>
      </select><br/>
      --Notas:--<br/>
      | 01= <input type="text" value="" name="nota1"/><br/>
      | 02= <input type="text" value="" name="nota2"/><br/>
      | 03= <input type="text" value="" name="nota3"/><br/>
      | 04= <input type="text" value="" name="nota4"/><br/>
      <input type="button" value="Adicionar >>"
```

```

        onClick="document.forms.nota.submit()"/>
    </form>
    <hr/>
    <% for (Curso c : new cursoControl().getLista()) {%>
    <h3>Notas do curso : <b><%= c.getNome()%></b></h3>
    <p>
        <% for (Notas n : c.getNotas()) {%>
        Aluno: <b><%= n.getChave().getAluno().getNome()%></b> |
        <%= n.getNota_1()%> | <%= n.getNota_2()%> | <%=
n.getNota_3()%> | <%= n.getNota_4()%> |
        <input type="button"
onClick="Location.href='notasControl?operacao=deleta&aluno=<%=
n.getChave().getAluno().getRa()%>&curso=<%=
n.getChave().getCurso().getId()%>' value="Deletar >>"><br/>
        <% }%>
    </p>
    <%}%>
    <jsp:include page="footer.html"/>
</body>
</html>

```

Extra - Relacionamento n:n, um exemplo

Vamos supor o caso: uma pessoa pertence a vários grupos, e um grupo tem várias pessoas..um relacionamento n:n. Para fazer isso, no banco de dados criamos uma tabela que contem as chaves primárias de ambas as tabelas, "linkando-as".

O jpa garante que o programador nem se preocupe com essa tabela. Para isso criamos uma lista de objetos de uma classe na outra e mapeamos a especificação, tabela de junção e as colunas de junção. Veja a seguir.

Na classe People(Pessoa) colocamos uma lista de grupos e a mapeamos com a anotação **ManyToMany**, especificamos que a configuração será ministrada pela propriedade peoples (a lista de pessoas) da classe Group (Grupos) e no final indentificamos a tabela de junção (people_groups).

People.java

```

@Entity(name="people")
public class People implements Serializable{
    @Id
    @Column(name="id")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;
    @Column(nullable=false,name="name")
    private String name;

```

```
@Column(nullable=true,name="age")
private Integer age;

@OneToOne(cascade= CascadeType.ALL)
@JoinColumn(name="id_people")
private PeopleInfo info;

@ManyToMany(cascade= CascadeType.ALL,
            targetEntity=Group.class,
            mappedBy="peoples")
@JoinTable(name="people_groups")
private List<Group> groups;
```

Na classe Group, fazemos quase a mesma coisa. Porém na JoinTable, fornecemos não somente a tabela mais as colunas que farão o relacionamento. No ManyToMany de ambas aplicamos o targetEntity, para definir qual é a outra entidade alvo do relacionamento.

Group.java

```
@Entity(name="groups")
public class Group implements Serializable{
    @Id
    @Column(nullable=false,name="id")
    private Integer id;
    @Column(nullable=false,name="name")
    private String name;

    @ManyToMany(targetEntity=People.class)
    @JoinTable(name="people_groups",
              joinColumns=@JoinColumn(name="id_people"),
              inverseJoinColumns=@JoinColumn(name="id_group"))
    private List<People> peoples;
```

As propriedades joinColumns e inverseJoinColumns indicam a coluna de junção do caminho de ida (de Groups para People: id_people) e o de volta (de People para Group: id_group).

```
joinColumns=@JoinColumn(name="id_people"),
inverseJoinColumns=@JoinColumn(name="id_group"))
```

Na entidade People essa configuração seria trocada. (id_group para JoinColumns e id_people para inverseJoinColumns) mas como usamos a propriedade mappedBy no atributo grupos da entidade People, não há necessidade dessa configuração.

```
mappedBy="peoples"
```


Exercícios Complementares de Fixação:

- 1) Note que não fizemos um `upNotas`. Faça-os. Não esqueça de garantir que apenas as notas serão alteradas.
- 2) Refaça os exercícios complementares anteriores utilizando os conceitos aprendidos nessa seção.
- 3) Você será capaz de refazer o projeto Loja usando JPA? Tente fazer esse desafio.

Exercícios de Pesquisa:

1. Pesquise sobre JPA 2.0, EJB 3.0, Hibernate, TopLink EclipseLink, ORM.
2. Estude as especificações do JPA e do hibernate:
 - a. http://docs.jboss.org/hibernate/core/3.6/quickstart/en-US/html_single/
 - b. <http://docs.jboss.org/hibernate/core/3.6/reference/pt-BR/html/>
 - c. <http://docs.jboss.org/hibernate/entitymanager/3.6/reference/en/html/>
 - d. <http://docs.jboss.org/hibernate/core/3.6/quickstart/en-US/html/>



Módulo 03:

Mvc e Frameworks

Web para Java

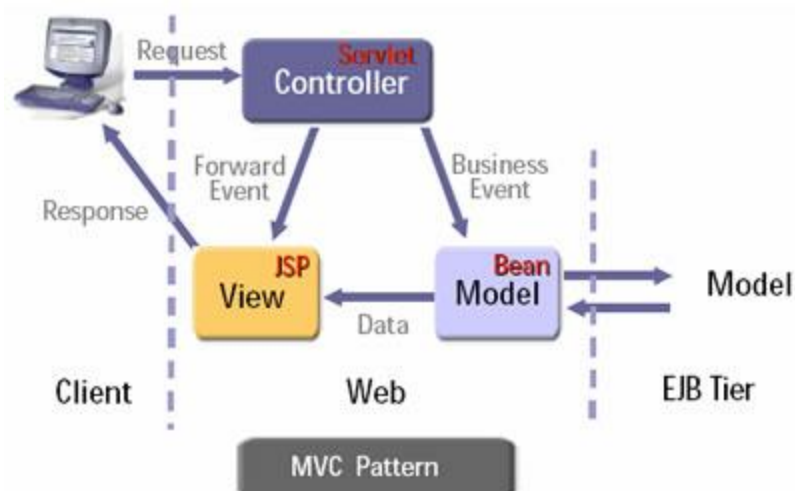
MVC (Model - View - Controller)

O padrão MVC surgiu para melhorar a forma de desenvolvimento inicialmente Desktop e atualmente para qualquer plataforma, MVC consiste em dividir em 3 camadas a aplicação.

M (Model) – Representa o “domínio”, é a especificação da informação na qual a aplicação opera, exemplo: secretária, advogado e cliente fazem parte do domínio de sistema de advocacia, também é comum ter funções que não fazem acessos a camada de persistência descritas nessa camada, por exemplo: calculo de datas, ou alguma informação que o usuário calculará ou precisará ter calculado sem a necessidade de uma persistência.

V (View) – Representa a renderização do model, ou seja uma interface para o usuário.

C (Controller) – É a ponte entre o Model e o View, processa e responde aos eventos de ações, normalmente essas ações são feitas pelo usuário e pode invocar alterações no Model.



Há diversas opções para a camada de controle no mercado. Veja um pouco sobre algumas delas:

1) **Struts Action** - o controlador mais famoso do mercado Java, é utilizado principalmente por ser o mais divulgado e com tutoriais mais acessíveis. Possui vantagens características do MVC e desvantagens que na época ainda não eram percebidas. É o controlador pedido na maior parte das vagas em Java hoje em dia.

É um projeto que não terá grandes atualizações pois a equipe dele se juntou com o Webwork para fazer o Struts 2, nova versão do Struts incompatível com a primeira e totalmente baseada no Webwork.

2) **VRaptor 3** - desenvolvido inicialmente por profissionais da Caelum e baseado em diversas ideias dos controladores mencionados acima, o Vraptor 3 usa o conceito de favorecer Convenções em vez de Configurações para minimizar o uso de XML e anotações em sua aplicação Web.

3) **JSF** - JSF é uma especificação da Sun para frameworks MVC. Ele é baseado em componentes e possui várias facilidades para desenvolver a interface gráfica. Devido ao fato de ser um padrão da Sun ele é bastante adotado.

4) **JBoss Seam** - segue as idéias do Stripes na área de anotações e é desenvolvido pelo pessoal que trabalhou também no Hibernate. Trabalha muito bem com o Java Server Faces e EJB 3.

5) **Spring MVC** - é uma parte do Spring Framework focado em implementar um controlador MVC. É fácil de usar em suas últimas versões e tem a vantagem de se integrar a toda a estrutura do Spring com várias tecnologias disponíveis.

6) **Stripes** - um dos frameworks criados em 2005, abusa das anotações para facilitar a configuração.

7) **Wicket** - controlador baseado na idéia de que todas as suas telas deveriam ser criadas através de código Java. Essa linha de pensamento é famosa e existem diversos projetos similares, sendo que é comum ver código onde instanciamos um formulário, adicionamos botões, etc como se o trabalho estivesse sendo feito em uma aplicação Swing, mas na verdade é uma página html.

Temos também diversas opções para a camada de visualização. Um pouco sobre algumas delas:

4) **JSP** - como já vimos, o JavaServer Pages, temos uma boa idéia do que ele é, suas vantagens e desvantagens. O uso de taglibs (a JSTL por exemplo) e expression language é muito importante se você escolher JSP para o seu projeto. É a escolha do mercado hoje **em dia**.

5) **Velocity** - um projeto antigo, no qual a EL do JSP se baseou, capaz de fazer tudo o que você precisa para a sua página de uma maneira extremamente compacta.

6) **Freemarker** - similar ao Velocity e com idéias do JSP - como suporte a taglibs - o freemarker vem sendo cada vez mais utilizado, ele possui diversas ferramentas na hora de formatar seu texto que facilitam muito o trabalho do designer.

7) **Sitemesh** - não é uma alternativa para as ferramentas anteriores mas sim uma maneira de criar templates para seu site, com uma idéia muito parecida com o struts tiles, porém genérica: funciona inclusive com outras linguagens como PHP etc.

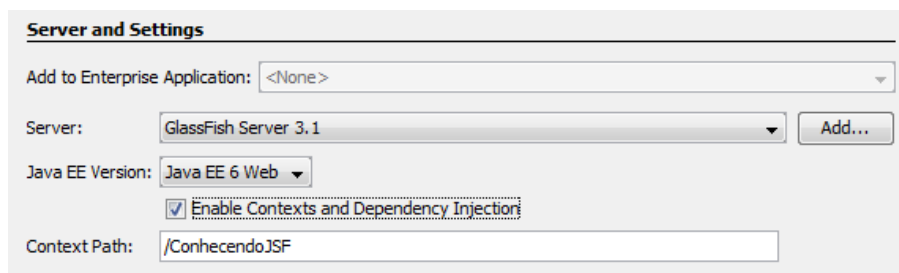
Framework Baseado em Componentes - JSF – Java Server Faces.

JSF, ou Java Server Faces são páginas com a extensão “.xHTML” que recebem tags especiais para utilização do código Java. Isso tudo com o auxílio do JavaBeans. Uma vantagem sobre as JSPs é que as JSF tem a sintaxe mais “natural” ao HTML, sendo colocado muito pouco código Java nelas (apenas algumas palavras), assim facilitando a divisão de tarefas entre web designers e programadores.

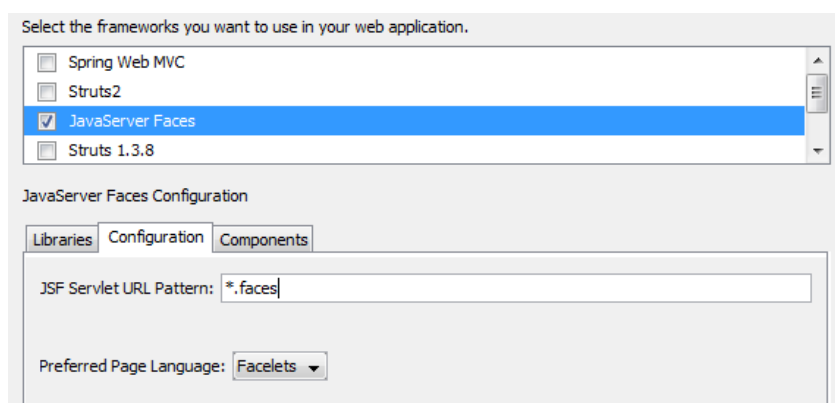
Trabalhando com JavaServer Faces no NetBeans

Para criar o JavaServer Faces no NetBeans, basta iniciar a criação de um projeto. Selecione em Categories o item Web e em Projects selecione Web Application.

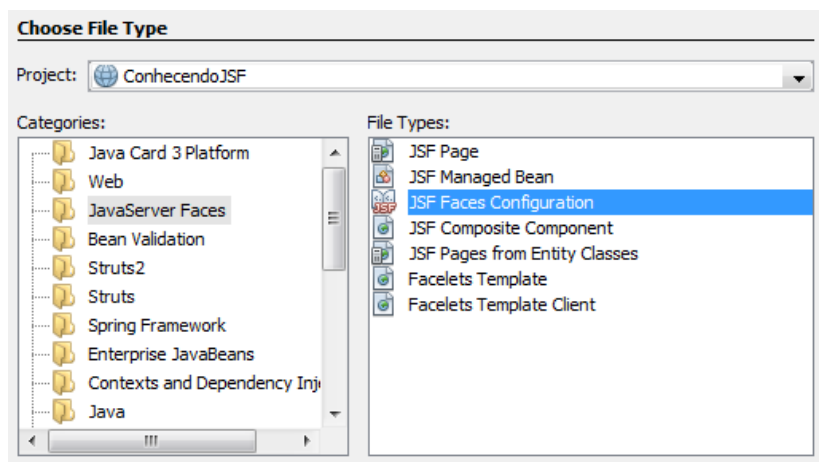
Na segunda etapa você colocará o nome do projeto e, clicando no botão Next, na terceira etapa, escolhemos o servidor e a versão do Java EE



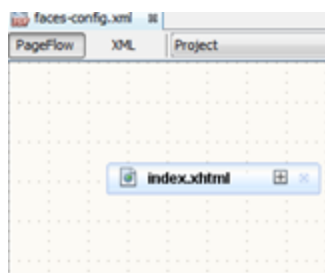
Clique em next, marque a opção JavaServer Faces em *Select the frameworks you want to use in your web application*. E configure exatamente como no mostrado.



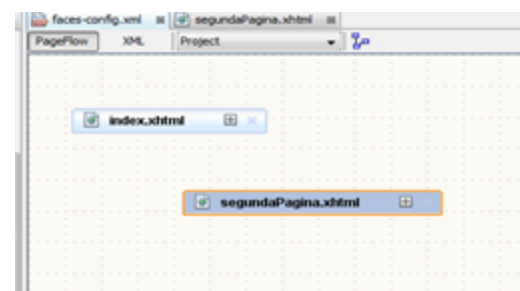
Podemos criar uma configuração JSF para controlar visualmente o fluxo das páginas e suas regras. Crie um JSF Faces Configuration;



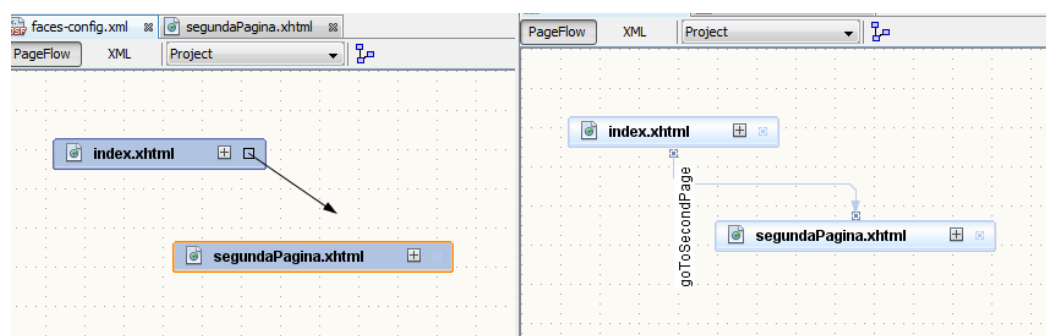
Nele, termos uma visão gráfica das paginas e seus fluxo.



Vamos criar uma segunda página JSF para criarmos uma regra de fluxo.



Segure com o mouse um pequeno quadrado na pagina index...e arrasta a seta que ira surgir a segunda página. Ele criará uma regra, mude seu nome pra **goToSecondPage**.



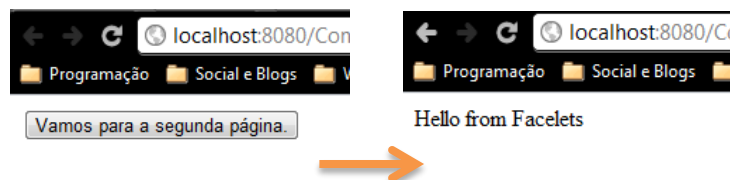
Na index.xhtml, vamos criar um botão pra ir até a segunda página utilizando a regra criada. Utilizamos a tag **commandButton**, no atributo action colocamos o nome de nossa regra.

Index.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:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Facelet Title</title>
  </h:head>
  <h:body>
    <h:form>
      <h:commandButton action="goToSecondPage"
                      value="Vamos para a segunda página."/>
    </h:form>
  </h:body>
</html>
```

!Lembrando que todo component de ação deve estar dentro das tags form.!



Regras de navegação:

É o mecanismo que permite unir algum tipo de processamento com sequência de páginas a serem mostradas.

Quando um botão ou um link é clicado, o componente associado gera um `ActionEvent` que é lançado para os seus listeners.

O receptor desse evento é o que chamamos de Backbean (Bean comum), um action method nessa classe vai ser executada e o resultado disso vai ser uma `String` que deve se encaixar na regra de navegação.

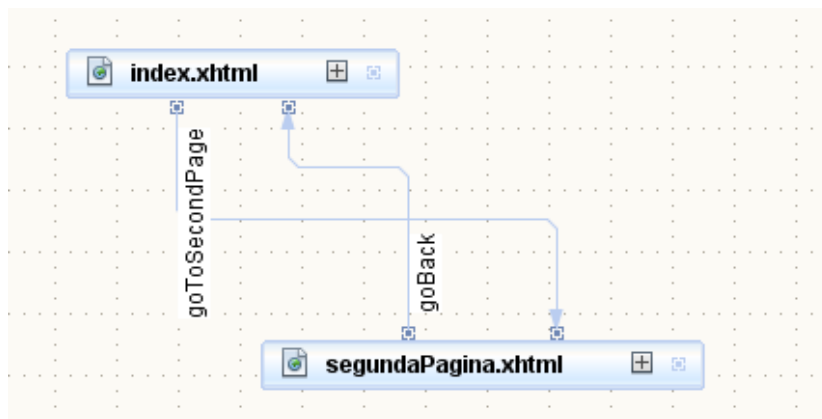
Vendo o XML de nosso `faces-config.xml` teremos:

faces-config.xml

```
<faces-config version="2.0"
  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_2_0.xsd">
  <navigation-rule>
    <from-view-id>/index.xhtml</from-view-id>
```

```
<navigation-case>
  <from-outcome>goToSecondPage</from-outcome>
  <to-view-id>/segundaPagina.xhtml</to-view-id>
</navigation-case>
</navigation-rule>
</faces-config>
```

Vamos definir uma regra de volta.



segundaPagina.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:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Facelet Title</title>
  </h:head>
  <h:body>
    <h:form>
      <h:commandButton action="goback"
                      value="Vamos voltar."/>
    </h:form>
  </h:body>
</html>
```

Componentes básicos

Para começar crie um projeto com o nome de ComponentesBasicos, dentro de pacotes de código-fonte crie um pacote com o nome de controle. Nossa base será um TesteBean.java que deverá ser criando dentro do pacote controle. Segue o código:

TesteBean.java

```
package controle;

import java.util.*;
import javax.inject.Named;

@Named
@javax.enterprise.context.RequestScoped
public class TesteBean {

    private boolean confirma;
    private String tipoPessoa;
    private List<String> cores;
    private List<String> coresSelecionadasCheckbox;
    private List<String> coresSelecionadasManyMenu;
    private String corSelecionadaOneMenu;

    public TesteBean() {
        cores = new ArrayList<String>();
        cores.add("Amarelo");
        cores.add("Azul");
        cores.add("Preto");
        cores.add("Vermelho");
    }
    //Gerar getters e setters
}
```

outputText: `<h:outputText value="OI...Eu sou um texto!"/>`

selectBooleanCheckbox: Como o nome já diz, ele retorna um valor booleano (true ou false).

```
<h:selectBooleanCheckbox value="#{testeBean.confirma}" />
<h:outputText value="Deseja receber mais informações desse
blog?"/>
```

☐ Deseja receber mais informações desse blog?

SelectOneRadio: Quanto a esse componente, entre as várias opções, apenas uma pode ser selecionada

```
<h:selectOneRadio value="#{testeBean.tipoPessoa}">
    <f:selectItem itemLabel="Fisica" itemValue="F" />
    <f:selectItem itemLabel="Juridica" itemValue="J" />
```

```
</h:selectOneRadio>
```

☐ Física ☐ Juridica

SelectOneMenu: Este componente gera algo mais conhecido como um combobox. Mais detalhes...

```
<h:selectOneMenu value="#{testeBean.corSelecionadaOneMenu}" >
    <f:selectItems value="#{testeBean.cores}" />
</h:selectOneMenu>
```

Amarelo ▼

SelectManyCheckbox: Componente que permite selecionar vários objetos de uma vez só, marcando os checkbox.

```
<h:selectManyCheckbox
    value="#{testeBean.coresSelecionadasCheckbox}"
    layout="pageDirection" >
    <f:selectItems value="#{testeBean.cores}" />
</h:selectManyCheckbox>
```

☐ Amarelo
☐ Azul
☐ Preto
☐ Vermelho

SelectManyMenu: Componente parecido com o SelectOneMenu, com a diferença que este permite selecionar vários objetos de uma vez, basta usar o CTRL e clicar nos objetos que deseja selecionar.

```
<h:selectManyMenu value="#{testeBean.coresSelecionadasManyMenu}"
    style="height: 100px" >
    <f:selectItems value="#{testeBean.cores}" />
</h:selectManyMenu>
```

Amarelo ▲
Azul
Preto
Vermelho ▼

Diferença entre usar **f:selectItem** e **f:selectItems**:

f:selectItem - esta tag serve para definir valores estaticamente, ou seja diretamente na página, ele tem as propriedades itemLabel e itemValue, o primeiro é o que deve mostrar na página e o segundo é o valor que deve ser salvo em algum atributo do bean.

f:selectItems - este é mais claro, como está no plural já dá a entender que receberá uma lista de objetos.

Exemplo:

```
private String carro;
private String[] carros = {
    "Honda Accord", "Toyota 4Runner", "NissanZ350"};
```

```
<h:selectOneRadio value="#{testeBean.carro}">
  <f:selectItems value="#{testeBean.carros}" />
</h:selectOneRadio>
```

☐ Honda Accord ☐ Toyota 4Runner ☐ NissanZ350

Tabelas:

```
<h:panelGrid columns="4" footerClass="subtitle"
  headerClass="subtitlebig" styleClass="medium"
  columnClasses="subtitle,medium">
  <f:facet name="header">
    <h:outputText value="Table with numbers"/>
  </f:facet>
  <h:outputText value="1" />
  <h:outputText value="2" />
  <h:outputText value="3" />
  <h:outputText value="4" />
  <h:outputText value="5" />
  <h:outputText value="6" />
  <h:outputText value="7" />
  <f:facet name="footer">
    <h:panelGroup>
      <h:outputText value="one row" />
      <h:outputText value=" " />
      <h:outputText value="grouped with panelGroup" />
    </h:panelGroup>
  </f:facet>
</h:panelGrid>
```

Tabela de dados:

```
public class Livro {  
    private String titulo;  
    private String Assunto;  
    private float preco;  
    //getters and setters
```

```
private List<Livro> livros;  
public TesteBean() {  
    livros = new ArrayList<Livro>();  
    livros.add(new Livro("JSF For Dummies", "JSF", 25.0f));  
    livros.add(new Livro("Struts For Dummies", "Struts", 22.95f));  
    ...
```

```
<h:dataTable id="books" value="#{testeBean.livros}" var="store">  
    <h:column>  
        <f:facet name="header">Title</f:facet>  
        <h:outputText value="#{store.titulo}"/>  
    </h:column>  
    <h:column>  
        <f:facet name="header">Subject</f:facet>  
        <h:outputText value="#{store.assunto}"/>  
    </h:column>  
    <h:column>  
        <f:facet name="header">Preço (R$)</f:facet>  
        <h:outputText value="#{store.preco}"/>  
    </h:column>  
</h:dataTable>
```

Title	Subject	Preço (R\$)
JSF For Dummies	JSF	25.0
Struts For Dummies	Struts	22.95

Managed Bean

POJO's (Plain Old java Object) que funcionam como representação da view. Eles linkam a view ao modelo. São acessadas via Expression Language (EL) na JSP. É a classe que criamos para o nosso exemplo dos componentes.

É necessário registramos um managed bean para que o framework (JSF) saiba que ele existe. Um managed bean é registrado no arquivo de configurações do JSF (faces-config.xml) ou no próprio managed bean. A configuração de um managed bean no **faces-config.xml** se resume basicamente ao código abaixo:

```
<managed-bean>
  <managed-bean-name>testeBean</managed-bean-name>
  <managed-bean-class>controle.TesteBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Ou no próprio bean:

```
@Named
@javax.enterprise.context.RequestScoped
```

Um managed bean vive de acordo com seu escopo definido pela tag xml **managed-bean-scope** ou a segundo anotação vista no TesteBean. Como ligar nosso managed bean aos componenetes do nosso formulário? Exemplo de um beans de Login:

```
public class LoginBean {
    private String login;
    private String senha;

    // gets e sets

    /**
     * Efetua login no sistema
     */
    public String logar() {
        // processa
        return "index";
    }
}

LoginBean.java]

<h:form>
  <h:panelGrid columns="2" border="1">
    <h:outputLabel value="Login" for="login"/>
    <h:inputText value="" id="login"/>
    <h:outputLabel value="Senha" for="senha"/>
    <h:inputSecret value="" id="senha"/>
  </h:panelGrid>
  <h:commandButton value="Logar" />
</h:form>
```

[/pages/login.jsp]

Usamos a Expression Language:

```
<h:form>
  <h:panelGrid columns="2" border="1">
    <h:outputLabel value="Login" for="login"/>
    <h:inputText value="#{loginBean.login}" id="login"/>
    <h:outputLabel value="Senha" for="senha"/>
    <h:inputSecret value="#{loginBean.senha}" id="senha"/>
  </h:panelGrid>
  <h:commandButton value="Logar" action="#{loginBean.logar}" />
</h:form>
```

Detalhando uma EL

`#{testeBean.senha}`

Nome do
nosso
managed bean

Atributo que existe em nosso bean.
É necessário criar os métodos get e
set para o atributo

Semelhante ao JSTL visto anteriormente, porém usa-se `#{}` no lugar de `${}`. Pode-se executar métodos no modelo através de expressões.

Em JSF nós temos 3 tipos de bindings (ligações):

Value binding

Ligamos a propriedade de um bean como valor do componente:

```
<h:inputText value="#{meuBean.login}">
public String getLogin(){
    return login;
}
public void setLogin(String login ){
    this.login=login;
}
```

Method binding

Ligamos o método de um bean como uma “ação” do componente:

```
<h:commandButton value="Logar"
    action="#{loginBean.logar}"/>
public String logar(){
    return "index";
}
```

Component binding

Ligamos a propriedade de um bean como representação do componente na view.

```
<h:inputtext value="#{loginBean.login}"
             binding="#{loginBean.uiLogin}" id="login"/>

public UIInput getUiLogin(){
    return uiLogin;
}
public void setUiLogin(UIInput uiLogin){
    this.uiLogin=uiLogin;
}
```

Mensagens

São as mensagens de informação ou erro exibidas ao usuário da aplicação. Qualquer parte da aplicação (managed beans, converters, validators etc) poderá gerar as mensagens quando necessário.

Basicamente existem dois tipos de mensagens de erro.

Aplicação(lógica de negócios, banco de dados, conexão etc...). Exemplo:

```
private int idade;
public int getIdade(){return idade;}
public void setIdade(int idade){
    FacesContext ctx = FacesContext.getCurrentInstance();
    if(idade>=18)
        ctx.addMessage(null,new FacesMessage("Usuário maior de
idade"));
    else
        ctx.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_ERROR,
                "Usuário menor de idade",""));
}
```

```
<h:messages globalOnly="true"
             errorStyle="color:red;"
             infoStyle="color:green;"/>
Idade:<h:inputText value="#{testeBean.idade}"/>
```

• Usuário maior de idade

• Usuário menor de idade

Idade: 18

Idade: 6

Inputs (conversão, validação de campos etc...). Exemplo:

```
<h:inputText id="input" required="true"
              requiredMessage="Este campo é obrigatório"/>
<h:message for="input" errorStyle="color:red"/>
```

Este campo é obrigatório

Converters

Tem por finalidade converter e/ou formatar objetos do tipo java para String e vice-versa. JSF já nos fornece converters padrões para date/time, inteiros, números e moeda e nos permite implementar nosso próprio converter.

```
<h:inputText value="#{testeBean.data}"
              id="datanascimento"
              maxLength="10"
              required="true"
              converterMessage="Data inválida, formato
correto é dd/MM/yyyy">
    <f:convertDateTime pattern="dd/MM/yyyy"/>
</h:inputText>
```

8909/34/34

- Data inválida, formato correto é dd/MM/yyyy

! Implementando seu próprio converter: 1.: Devemos implementar a interface `javax.faces.convert.Converter`; 2.: Registrar nosso converter no `faces-config.xml`; 3.: Informar ao componente qual converter utilizar. !

Validators

Tem a responsabilidade de garantir que o valor entrado pelo usuário seja válido. Você pode validar tanto objetos como strings. JSF já nos fornece validators padrões como `required`, `length` e `ranges` e nos permite implementar nosso próprio validator.

```
<h:inputText id="input2"
              required="true"
              validatorMessage="Tamanho mínimo de 2 e máximo de 6">
    <f:validateLength minimum="2" maximum="6"/>
</h:inputText>
```


- Tamanho mínimo de 2 e máximo de 6

! Implementando seu próprio validator: 1.: Devemos implementar a interface `javax.faces.validator.Validator`; 2.: Registrar nosso validator no `faces-config.xml`; 3.: Informar ao componente qual validator utilizar. !

Eventos e Listeners

JSF usa o modelo JavaBeans event/listener (também utilizado no Swing). Componentes UI (e outros objetos) geram eventos, e listeners podem ser registrados para manipular esses eventos.

Os listeners mais comuns do JSF são:

Value-change events

```
<h:selectOneRadio id="dependentes"
                 onchange="submit();"

valueChangeListener="#{eventBean.exibirNumeroDeDependentes}">
    <f:selectItem itemValue="S" itemLabel="Sim"/>
    <f:selectItem itemValue="N" itemLabel="Não"/>
</h:selectOneRadio>
```

```
public void exibirNumeroDeDependentes(ValueChangeEvent ev){
    String opcao = (String) ev.getNewValue();
    if("S".equalsIgnoreCase(opcao))
        this.panelNumDep.setRendered(true);
    else
        this.panelNumDep.setRendered(true);
}
```

Action events

```
<h:commandLink id="teste" value="Clique"
               actionListener="#{testeBean.enviar}"
               action="login"/>
```

Crud com Java Server Faces

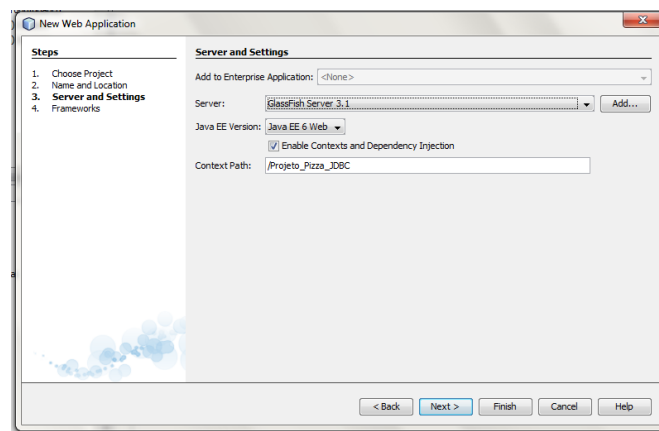
Criaremos um novo mini projeto:Projeto Pizzaria. Iremos aprender com ele o JSF na prática. (Atenção, faça a configuração inicial exatamente como for feito aqui, isso será uma receita de bolo para próximos projetos)

Inicialmente criaremos o banco de dados:

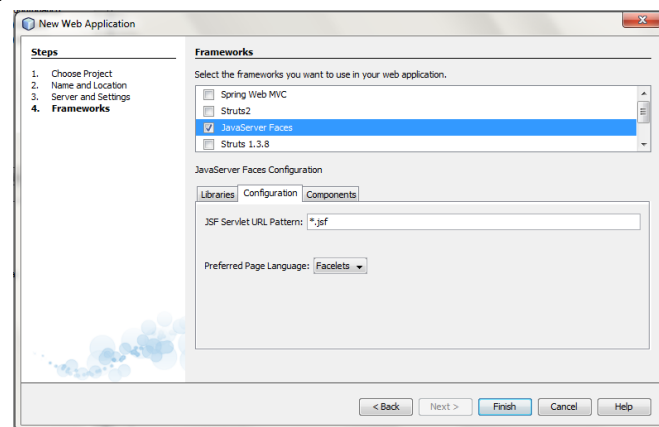
```
CREATE TABLE clientes (  
    codigo INTEGER(11) NOT NULL AUTO_INCREMENT,  
    nome VARCHAR(255) NOT NULL,  
    telefone VARCHAR(30) NOT NULL,  
    endereco VARCHAR(255) NOT NULL,  
    PRIMARY KEY (codigo)  
);  
CREATE TABLE pizza(  
    codigo INTEGER(11) NOT NULL AUTO_INCREMENT,  
    nome varchar(45) NOT NULL,  
    descricao VARCHAR(255) NOT NULL,  
    preco REAL NOT NULL,  
    n_pedacos INTEGER,  
    PRIMARY KEY(codigo)  
);  
CREATE TABLE pedido(  
    codigo INTEGER(11) NOT NULL AUTO_INCREMENT,  
    cliente_id INTEGER NOT NULL,  
    data_pedido INTEGER NOT NULL,  
    PRIMARY KEY(codigo)  
);  
CREATE TABLE iten_pedido(  
    pedido_codigo INTEGER NOT NULL,  
    pizza_codigo INTEGER NOT NULL,  
    PRIMARY KEY(pedido_codigo,pizza_codigo)  
);
```

Primeiro, vamos criar um novo projeto chamado Projeto PizzaJDBC:

Na parte dos Servidores e Configurações escolha o GlassFish Server 3, que vem junto com o NetBeans e a versão JavaEE6. Marque a caixa “Habilitar injeção de contextos e dependências”.



Na parte de Frameworks escolha o JavaServer Faces. Não se preocupe com a aba Biblioteca. Vamos para a aba configuração. No campo da URL digite *.jsf, e escolha a opção Facelets.



Agora criaremos a nossa PersistenceUnit chamada **ProjetoPizzaPU** com Hibernate, como visto no módulo anterior.

Criaremos Primeiramente o nossos Clientes:

Cliente.java

```
package modelo;

import java.io.Serializable;
import java.util.List;
import javax.persistence.*;

@Entity(name="clientes")
public class Cliente implements Serializable{

    @Id
```

```
@Column(name="codigo")
private int codigo;
@Column(name="nome")
private String nome;
@Column(name="telefone")
private String telefone;
@Column(name="endereco")
private String endereco;

@OneToMany(cascade= CascadeType.ALL)
@JoinColumn(name="cliente_id")
private List<Pedido> pedidos;
```

Criaremos uma classe que cuidará da persistência do Cliente, a **ClienteControl**:

ClienteControl.java

```
public class ClienteControl {

    private EntityManagerFactory factory =
Persistence.createEntityManagerFactory("ProjetoPizzaPU");
    private EntityManager em = factory.createEntityManager();

    public boolean salvar(Cliente c) {
        try{
            em.getTransaction().begin();
            em.persist(c);
            em.getTransaction().commit();
            return true;
        }catch(Exception e){
            em.getTransaction().rollback();
            return false;
        }
    }

    public boolean alterar(Cliente c) {
        try{
            em.getTransaction().begin();
            em.merge(c);
            em.getTransaction().commit();
            return true;
        }catch(Exception e){
            em.getTransaction().rollback();
            return false;
        }
    }
}
```

```
public boolean excluir(Cliente c) {
    try{
        em.getTransaction().begin();
        em.remove(c);
        em.getTransaction().commit();
        return true;
    }catch(Exception e){
        em.getTransaction().rollback();
        return false;
    }
}

public Cliente get(int codigo){
    return em.find(Cliente.class,codigo);
}

public List<Cliente> getList(){
    List l = em.createQuery("from clientes").getResultList();
    LinkedList<Cliente> ll = new LinkedList<Cliente>();
    for(Object o : l) ll.add((Cliente) o);
    return ll;
}
}
```

A ultima classe necessária é a ClienteBean. As classes Beans obedecem algumas regras. Sempre procure construir suas Beans nos moldes dados. Pois internamente essa classe será vinculada com a página xHTML. Essa classe é obrigatória para o JSF.

ClienteBean.java

```
import java.io.Serializable;
import java.util.List;
import javax.enterprise.context.SessionScoped;
import javax.faces.model.*;
import javax.inject.Named;
import modelo.Cliente;

@Named
@SessionScoped
public class ClienteBean implements Serializable{

    private ClienteControl controle = new ClienteControl();
    private Cliente cliente = new Cliente();
    private DataModel<Cliente> clientes;
```

```
public void novo() {
    cliente = new Cliente();
}
public String inserir() {
    return (controle.salvar(cliente)) ? "clientes" :
    "falhou";
}
public void selecionar() {
    cliente = clientes.getRowData();
}
public String alterar() {
    return (controle.alterar(cliente))? "clientes" :
    "falhou";
}
public String remover() {
    return (controle.excluir(cliente))? "clientes" :
    "falhou";
}
public Cliente getCliente() {
    return cliente;
}
public void setCliente(Cliente cliente) {
    this.cliente = cliente;
}
public DataModel<Cliente> getClientes() {
    List<Cliente> clienteList = controle.getList();
    clientes = new ListDataModel<Cliente>(clienteList);
    return clientes;
}
public void setClientes(DataModel<Cliente> clientes) {
    this.clientes = clientes;
}
}
```

Nos métodos inserir, alterar e remover repare que uma String é passada SEMPRE como valor de retorno. Essa String representa o nome de uma pagina para a qual, após o código ser executado, o usuário deverá ser direcionado.

O método getClientes, recebe do ClienteDao, uma List contendo os clientes no banco. E deve retornar uma ListDataModel, para ser usado com uma tag especial do Jsf que constrói tabelas dinamicamente.

Por ultimo, as páginas:

Index.xhtml, vemos aqui uma tag chamada `commandButton`, ela cria um botão para uma determinada página, assim como os `type="button"`.

Index.xhtml

```
<?xml version='1.0' encoding='ISO-8859-9' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>Pizzaria ITTraining</title>
  </h:head>
  <h:body>
    <br/>
    <h1>Projeto Pizza</h1>
    <hr/>
    <h:form>
      <h:commandButton action="clientes" value="Clientes >>"/>
      | <h:commandButton action="pizzas" value="Pizzas >>"/>
      | <h:commandButton action="pedidos" value="Pedidos >>"/>
    </h:form>
    <br/><hr/>ITTraining - Curso Java WEB
  </h:body>
</html>
```

Em `Clientes.xhtml`, vemos muitos novos comandos.

O principal é o `dataTable`. Ele recebe um `ListDataModel` do método `getClientes` e cria uma tabela dinamicamente sem a necessidade do uso de um comando `For` ou criação da tag `<table>`.

A cada coluna da tabela usamos o comando `column`, dentro dele, definimos o que é cabeçalho, usando o comando `facet` junto com o parâmetro `name=header` e o que será o espaço para os dados, usando outro comando chamado `outputtext`, que serve para mostrar texto na página. Repare que o uso de “código Java” se aplica as expressões `javabeans : #{...}` e apenas isso.

As regras para o uso dos métodos é a seguinte: aqueles que forem métodos `get` e `set`, não necessitam das palavras `get` e `set`; o uso de parênteses é vetado, ou seja, métodos que precisem de parâmetros não podem ser chamados. Para entrada de dados o objeto cliente criado na classe beans deve ser vinculado com os campos `inputtext` com o auxílio do parâmetro `value`, o que

clientes.xhtml

```
<?xml version='1.0' encoding='ISO-8859-9' ?>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>Pizzaria ITTraining</title>
  </h:head>
  <h:body>
    <br/>
    <h1>Projeto Pizza</h1>
    <hr/>
    <h:form>
      <h:commandButton action="index" value="Voltar"/>
      | <h:commandButton action="novoCliente"
actionListener="#{clienteBean.novo}"
value="Novo"/>
    <hr/>
    <h3>Lista de Clientes Cadastrados</h3>
    <h:dataTable value="#{clienteBean.clientes}" var="c"
border="1" cellpadding="0">

      <h:column>
        <f:facet name="header">
          <h:outputText value="Codigo"/>
        </f:facet>
        <h:outputText value="#{c.codigo}"/>
      </h:column>

      <h:column>
        <f:facet name="header">
          <h:outputText value="Nome"/>
        </f:facet>
        <h:outputText value="#{c.nome}"/>
      </h:column>

      <h:column>
        <f:facet name="header">
          <h:outputText value="Telefone"/>
        </f:facet>
        <h:outputText value="#{c.telefone}"/>
      </h:column>

      <h:column>
        <f:facet name="header">
          <h:outputText value="Endereco"/>
        </f:facet>
        <h:outputText value="#{c.endereco}"/>
      </h:column>
    </h:dataTable>
  </h:body>
</html>
```



```

        </f:facet>
        <h:outputText value="#{c.endereco}"/>
    </h:column>

    <h:column>
        <f:facet name="header">
            <h:outputText value="Ações"/>
        </f:facet>
        <h:commandButton action="alterarCliente"
            actionListener="#{clienteBean.selecionar}"
            value="Alterar"/>
        <h:commandButton action="removerCliente"
            actionListener="#{clienteBean.selecionar}"
            value="Remover"/>
    </h:column>

</h:dataTable>
</h:form>
<br/><hr/>ITTraining - Curso Java WEB
</h:body>
</html>

```

novoCliente.xhtml

```

<?xml version='1.0' encoding='ISO-8859-9' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Pizzaria ITTraining</title>
    </h:head>
    <h:body>
        <br/>
        <h1>Projeto Pizza</h1>
        <hr/>
        <h:form>
            <h4>Cadastrando Novo Cliente</h4>
            <h:panelGrid columns="2">
                <h:outputText value="Nome"/>
                <h:inputText value="#{clienteBean.cliente.nome}"/>
                <h:outputText value="Telefone"/>
                <h:inputText
value="#{clienteBean.cliente.telefone}"/>
                <h:outputText value="Endereço"/>
            </h:panelGrid>
        </h:form>
    </h:body>
</html>

```

```

        <h:inputText
value="#{clienteBean.cliente.endereco}"/>
        <h:commandButton action="#{clienteBean.inserir}"
value="Inserir >>"/>
        <h:commandButton action="clientes" immediate="True"
value="Cancelar"/>
    </h:panelGrid>
</h:form>
<br/><hr/>ITTraining - Curso Java WEB
</h:body>
</html>

```

alterarCliente.xhtml

```

<?xml version='1.0' encoding='ISO-8859-9' ?>
<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>Pizzaria ITTraining</title>
    </h:head>
    <h:body>
        <br/>
        <h1>Projeto Pizza</h1>
        <hr/>
        <h:form>
            <h4>Alterando o Cliente
            #{clienteBean.cliente.codigo}</h4>
            <h:panelGrid columns="2">
                <h:outputText value="Nome"/>
                <h:inputText value="#{clienteBean.cliente.nome}"/>
                <h:outputText value="Telefone"/>
                <h:inputText
value="#{clienteBean.cliente.telefone}"/>
                <h:outputText value="Endereço"/>
                <h:inputText
value="#{clienteBean.cliente.endereco}"/>
                <h:commandButton action="#{clienteBean.alterar}"
value="Alterar"/>
                <h:commandButton action="clientes" immediate="True"
value="Cancelar"/>
            </h:panelGrid>
        </h:form>
        <br/><hr/>ITTraining - Curso Java WEB
    </h:body>

```

```
</html>
```

removerCliente.xhtml

```
<?xml version='1.0' encoding='ISO-8859-9' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
  <h:head>
    <title>Pizzaria ITTraining</title>
  </h:head>
  <h:body>
    <br/>
    <h1>Projeto Pizza</h1>
    <hr/>
    <h:form>
      <h:outputText
        value="Deseja remover o cliente
#{clienteBean.cliente.nome}?"/>
      <h:panelGrid columns="2">
        <h:commandButton action="#{clienteBean.remover}"
value="Remover >>"/>
        <h:commandButton action="clientes" immediate="True"
value="Cancelar"/>
      </h:panelGrid>
    </h:form>
    <br/><hr/>ITTraining - Curso Java WEB
  </h:body>
</html>
```

falhou.xhtml

Falhou.xhtml, será utilizada caso ocorra algum erro com os comandos SQL

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

```
<br/>
<h1>Projeto Pizza</h1>
<hr/>
<h:form>
    <h:outputText value="Ocorreu um erro, tente novamente"/>
    <h:commandButton action="index" value="Tentar
novamente"/>
</h:form>
<br/><hr/>ITTraining - Curso Java WEB
</h:body>
</html>
```

Veja:

Projeto Pizza

Lista de Clientes Cadastrados

Codigo	Nome	Telefone	Endereco	Ações
1	Lucas	49722637	Rua ABC	<input type="button" value="Alterar"/> <input type="button" value="Remover"/>

ITTraining - Curso Java WEB

Cadastrando Novo Cliente

Nome

Telefone

Endereço

ITTraining - Curso Java WEB

Alterando o Cliente 1

Nome

Telefone

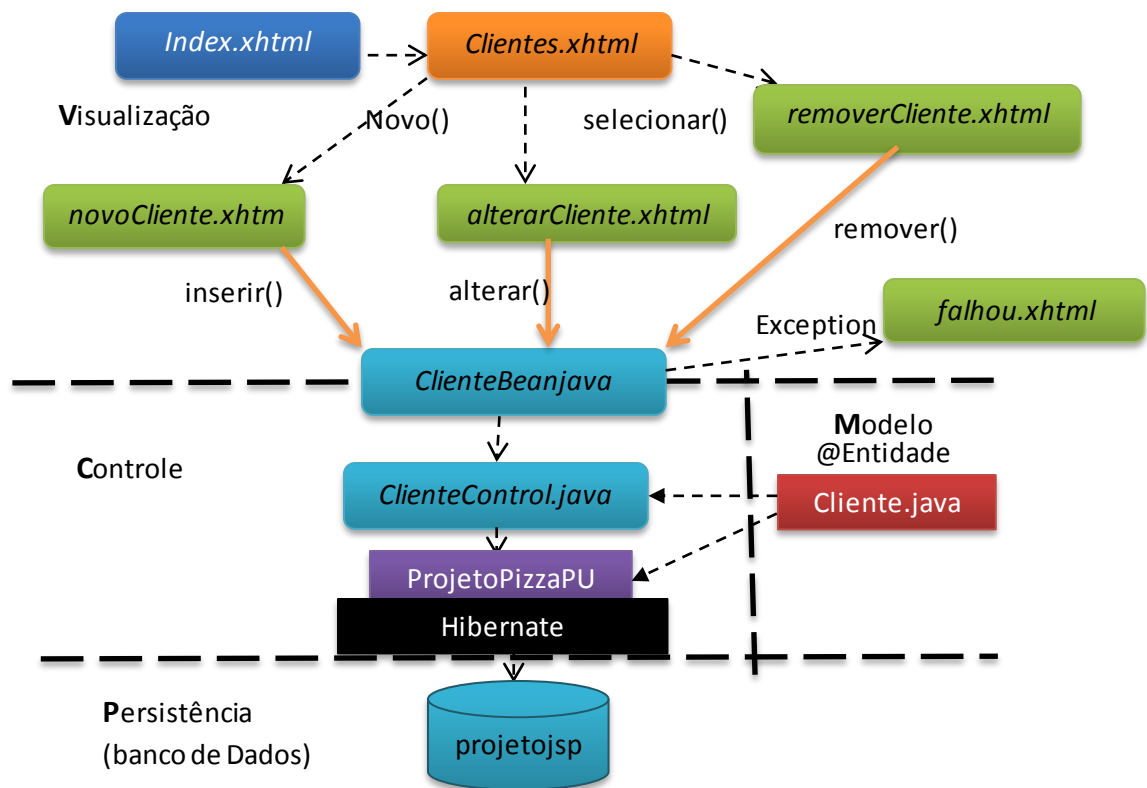
Endereço

ITTraining - Curso Java WEB

Deseja remover o cliente Fabio?

ITTraining - Curso Java WEB

Nosso mapa ficou:

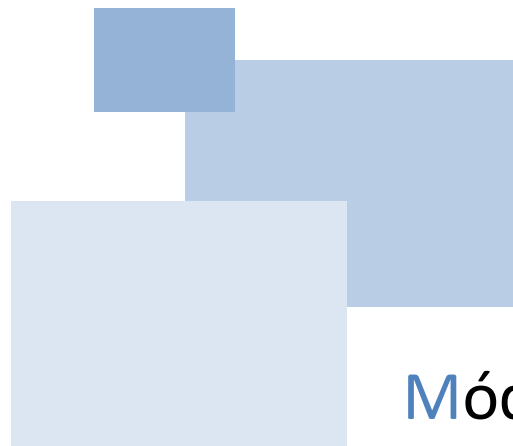


Exercícios Complementares de Fixação:

- 1) Complete o Projeto Pizza com a entidade Pizza e os Pedidos.
- 2) Crie (ou use os existentes) Validators para validar os campos do projeto.
- 3) Refaça TODOS os exercícios feitos durante o curso utilizando agora o JSF.

Exercícios de Pesquisa

- 1) Explique sobre JSF.
- 2) Faça uma tabela comparativa sobre os recursos do JSP e JSF. Quais as vantagens e desvantagens de cada tecnologia.
- 3) Pesquise sobre JavaBeans e Beans Validator para validar os formulários em JSF.
- 4) Pesquise sobre os frameworks JSF (PrimeFaces, RichFaces e ICEFaces).



Módulo 04:

RIA – Aplicações

Ricas para Web

RIA – Rich Internet Applications – Aplicações ricas para Internet.

Aplicações de Internet Rica são Aplicações Web que tem características e funcionalidades de softwares tradicionais do tipo Desktop. RIA típicos transferem todo o processamento da interface para o navegador da internet, porém mantém a maior parte dos dados (como por exemplo, o estado do programa, dados do banco) no servidor de aplicação.

Benefícios:

- **Riqueza:** É possível oferecer à interface do usuário características que não podem ser obtidos utilizando apenas o HTML disponível no navegador para aplicações Web padrão. Esta capacidade de poder induzir qualquer coisa no lado do cliente, incluindo o arraste e solte, utilizar uma barra para alterar dados, cálculos efetuados apenas pelo cliente e que não precisam ser enviados volta para o servidor, como por exemplo, uma calculadora básica de quatro operações.
- **Melhor resposta:** A interface é mais reativa a ações do usuário do que em aplicações Web padrão que necessitam de uma constante interação com um servidor remoto. Com isto os RIAs levam o usuário a ter a sensação de estarem utilizando uma aplicação desktop.
- **Equilíbrio entre Cliente/Servidor:** A carga de processamento entre o Cliente e Servidor torna-se mais equilibrada, visto que o servidor web não necessita realizar todo o processamento e enviar para o cliente, permitindo que o mesmo servidor possa lidar com mais sessões de clientes concomitantemente.
- **Comunicação assíncrona:** O *client engine* pode interagir com o servidor de forma *assíncrona* -- desta forma, uma ação na interface realizada pelo usuário como o clique em um botão ou link não necessite esperar por uma resposta do servidor, fazendo com que o usuário espere pelo processamento. Talvez o mais comum é que estas aplicações, a partir de uma solicitação, antecipe uma futura necessidade de alguns dados, e estes são carregados no cliente antes de o usuário solicite-os, de modo a acelerar uma posterior resposta. O site Google Maps utiliza esta técnica para, quando o usuário move o mapa, os segmentos adjacentes são carregados no cliente antes mesmo que o usuário mova a tela para estas áreas.
- **Otimização da rede:** O fluxo de dados na rede também pode ser significativamente reduzida, porque um *client engine* pode ter uma inteligência imbutida maior do que um navegador da Web padrão quando decidir quais os dados que precisam ser trocados com os servidores. Isto pode acelerar a solicitações individuais ou reduzir as respostas, porque muitos dos dados só são transferidos quando é realmente necessário, e a carga global da rede é reduzida. Entretanto, o uso destas técnicas podem neutralizar, ou mesmo reverter o potencial desse benefício. Isto porque o código não pode prever exatamente o que cada usuário irá fazer em seguida, sendo comum que tais técnicas baixar dados extras, para muitos ou todos os clientes, cause um tráfego desnecessário.

As deficiências e restrições associadas aos RIAs são:

- *Sandbox*: Os RIAs são executado dentro de um *Sandbox*, que restringe o acesso a recursos do sistema. Se as configurações de acesso aos recursos estiverem incorretas, os RIAs podem falhar ou não funcionar corretamente.
- *Scripts desabilitados*: JavaScripts ou outros scripts são muitas vezes utilizados. Se o usuário desativar a execução de scripts em seu navegador, o RIA poderá não funcionar corretamente, na maior parte das vezes.
- *Velocidade de processamento no cliente*: Para que as aplicações do lado do cliente tenha independência de plataforma, o lado do cliente muitas vezes são escritos em linguagens interpretadas, como JavaScript, que provocam uma sensível redução de desempenho. Isto não é problema para linguagens como Java, que tem seu desempenho comparado a linguagens compiladas tradicionais, ou com o Flash, em que a maior parte das operações são executadas pelo código nativo do próprio Flash.
- *Tempo de carregamento da aplicação*: Embora as aplicações não necessitem de serem *instaladas*, toda a inteligência do lado cliente (ou *client engine*) deve ser baixada do servidor para o cliente. Se estiver utilizando um web cache, esta carga deve ser realizada pelo menos uma vez. Dependendo do tamanho ou do tipo de solicitação, o carregamento do script pode ser demasiado longo. Desenvolvedores RIA podem reduzir este impacto através de uma compactação dos scripts, e fazer um carregamento progressivo das páginas, conforme elas forem sendo necessárias.
- *Perda de Integridade*: Se a aplicação-base é X/HTML, surgem conflitos entre o objetivo de uma aplicação (que naturalmente deseja estar no controle de toda a aplicação) e os objetivos do X/HTML (que naturalmente não mantém o estado da aplicação). A interface DOM torna possível a criação de RIAs, mas ao fazê-lo torna impossível garantir o seu funcionamento de forma correta. Isto porque um cliente RIA pode modificar a estrutura básica, sobrescrevendo-a, o que leva a uma modificação do comportamento da aplicação, causando uma falha irreversível ou *crash* no lado do cliente. Eventualmente, este problema pode ser resolvido através de mecanismos que garantam uma aplicação do lado cliente com restrições e limitar o acesso do usuário para somente os recursos que façam parte do escopo da aplicação. (Programas que executam de forma nativa não tem este problema, porque, por definição, automaticamente possuem todos os direitos de todos os recursos alocados).
- *Perda de visibilidade por Sites de Busca*: Sites de busca podem não ser capazes de indexar os textos de um RIA.
- *Dependência de uma conexão com a Internet*: Enquanto numa aplicação desktop ideal permite que os seus usuários fiquem *ocasionalmente conectados*, passando de uma rede para outra, hoje (em 2007), um típico RIA requer que a aplicação fique permanentemente conectada à rede.

(Fonte: Wikipédia)

Frameworks RIA para JSF 2.0 – PrimeFaces

PrimeFaces é um dos melhores frameworks RIA para JSF. Todos os seus componentes, quando renderizados, são estruturas HTML, definidas esteticamente por CSS, com controle de eventos e troca de mensagens em JavaScript. Ou seja, encapsulam a complexidade de usar CSS com HTML e JavaScript para construir certos componentes. Além de prover o reuso dos mesmos.

Para utilizar o Primefaces é preciso baixar a biblioteca “[primefaces-2.2.1.jar](#)” e criar um projeto NetBeans de acordo com a aula 05.

Antes de começar, vamos criar um beans que utilizaremos a seguir:

```
@Named
@RequestScoped
public class TableBean{

    private String nome;
    private String descricao;
    private String telefone;
    private String cpf;
    private String observacao;
    private Date dataCadastro;
    private String senha;

    public void testar(){
        FacesContext context = FacesContext.getCurrentInstance();
        context.addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_INFO, "Aviso", "Testado com sucesso!"));
    }

    public String getCpf() {...}
    public void setCpf(String cpf) {...}
    public Date getDataCadastro() {...}
    public void setDataCadastro(Date dataCadastro) {...}
    public String getDescricao() {...}
    public void setDescricao(String descricao) {...}
    public String getNome() {...}
    public void setNome(String nome) {...}
    public String getObservacao() {...}
    public void setObservacao(String observacao) {...}
    public String getSenha() {...}
    public void setSenha(String senha) {...}
    public String getTelefone() {...}
    public void setTelefone(String telefone) {...}
}
```

O método testar será utilizado para criar mensagens de erro ou confirmação como veremos a seguir.

Não podemos esquecer-nos de adicionar a biblioteca na página.jsf e de colocar uma tag chamada view, que corrige alguns problemas de compatibilidade, principalmente no Google Chrome. Veja como deverá ficar sua página index.

```
<?xml version='1.0' encoding='UTF-8' ?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.prime.com.tr/ui"
      xmlns:f="http://java.sun.com/jsf/core">
  <f:view contentType="text/html">
    <h:head>
      <title>Facelet Title</title>
    </h:head>
    <h:body>

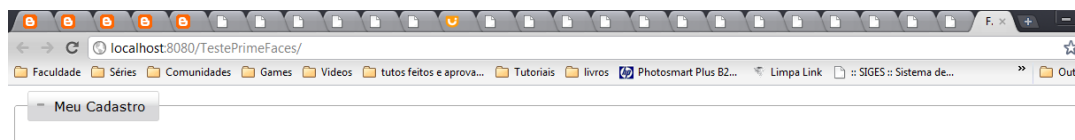
    </h:body>
  </f:view>
</html>
```

Botões e mensagens

Começaremos mostrando quatro componentes interessantes. O primeiro é o FieldSet, uma borda que circula formulário e deixa o visual interessante.

```
<p:fieldset legend="Meu Cadastro" toggleable="true">
</p:fieldset>
```

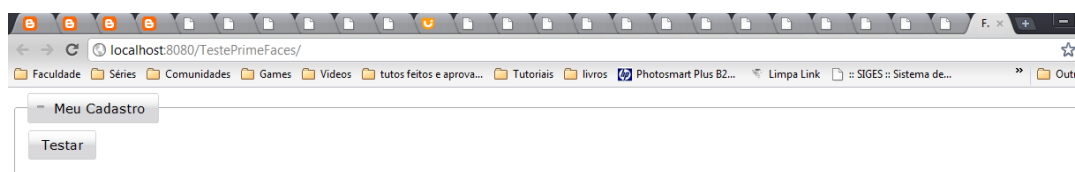
Legend é o título e toggleable indica se o usuário poderá esconder o conteúdo dentro do fieldset ou não. Veja o resultado:



O segundo componente é muito importante, trata-se de um botão commandButton.

```
<p:commandButton value="Testar" actionListener="#{testeBean.testar}" />
```

Veja o resultado:



O atributo actionListener indica ao botão qual método/comando no beans ele deverá fazer quando pressionado. Você pode utilizar action para se

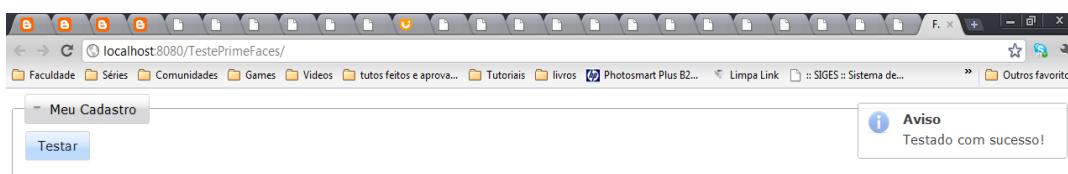
referir as páginas jsf. Mas não esqueça de sempre utilizar o atributo: `ajax="false"` quando o fizer, por exemplo:

```
<p:commandButton value="Painéis" action="testepainels.jsf"
ajax="false"/>
<p:commandButton value="Dock" action="testedock.jsf"
ajax="false"/>
```

Agora vamos fazer o botão mostrar uma mensagem quando for clicado. O PrimeFaces oferece dois tipos de mensagens, embutidas na páginas, e uma que aparece numa pequena janelinha e desaparece com o tempo. Essa ultima é usada da seguinte forma:

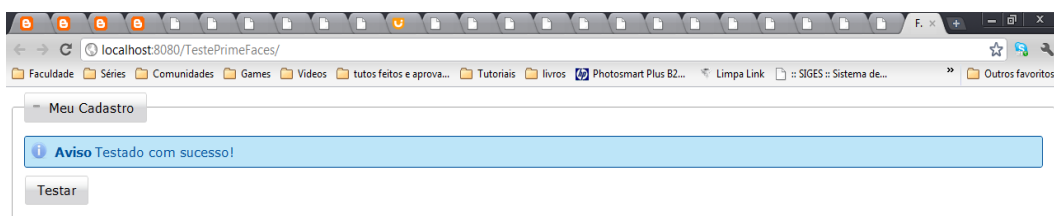
```
<p:growl id="avisos" showDetail="true" life="3000" />
```

O atributo `life` é para controlar, em milissegundos o tempo que a mensagem será mostrada. Veja o resultado:



Outra forma é usando:

```
<p:messages id="mensagens" showDetail="true" />
```



Para que o botão execute ação é preciso adicionar o seguinte atributo:

```
update="avisos,mensagens"
```

Ambos os métodos não precisam ser utilizados ao mesmo tempo, mas podem caso queira. Nossa página ficou assim no final:



Os botões Dock e Testar nos levarão para as páginas que serão criadas nos próximos tópicos.

Formulários

Formulários estão presentes em praticamente todos os sites. São compostos com campos de texto, senhas, datas, entre outros. O exemplo a seguir mostra um exemplo de uso de formulários.

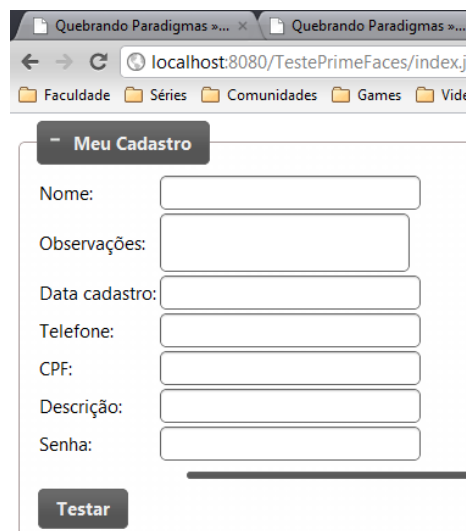
```
<p:fieldset legend="Meu Cadastro" toggleable="true">
  <h:form>
    <h:panelGrid columns="2">
      <h:outputText value="Nome:" />
      <p:inputText id="nome" value="#{tableBean.nome}" />
      <h:outputText value="Observações:" />
      <p:inputTextarea value="#{tableBean.observacao}" />
      <h:outputText value="Data cadastro:" />
      <p:calendar value="#{tableBean.dataCadastro}" />
      <h:outputText value="Telefone:" />
      <p:inputMask value="#{tableBean.telefone}" />
      <h:outputText value="CPF:" />
```

```

        <p:inputMask mask="999.999.999-99" value="#{tableBean.cpf}"
/>

        <h:outputText value="Descrição:" />
        <p:keyboard                                layout="qwertyBasic"
value="#{tableBean.descricao}" />
        <h:outputText value="Senha:" />
        <p:keyboard password="true" keypadOnly="true"
value="#{tableBean.descricao}" />

    </h:panelGrid>
    <p:separator style="width: 80%; height: 5px" />
</h:form>
</p:fieldset>
    
```



```

<h:outputText value="Data cadastro:" />
    <p:calendar value="#{tableBean.dataCadastro}" />
    
```



```

<h:outputText value="Telefone:" />
    
```

```
<p:inputMask mask="(999)9999-9999"
value="#{tableBean.telefone}" />
```

Telefone: () -

```
<h:outputText value="CPF:" />
<p:inputMask mask="999.999.999-99" value="#{tableBean.cpf}"
/>
```

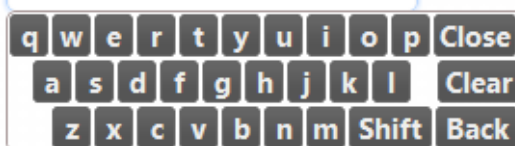
CPF: . -

```
<h:outputText value="Descrição:" />
<p:keyboard layout="qwertyBasic"
value="#{tableBean.descricao}" />
```

Descrição:

Senha:

Testar



```
<h:outputText value="Senha:" />
<p:keyboard password="true" keypadOnly="true"
value="#{tableBean.descricao}" />
```

Senha:

Testar



Menus

Menus são muito utilizados nas aplicações Desktop e Web, servem para facilitar a navegação pelo site. Na mesma página, criamos um novo fieldset e colocamos o componente toolbar, ele cria uma barra de ferramentas que serve como um menu. Dentro dele estipulamos os toolbarGroup que são o conjunto de elementos (botões) dispostos na barra de ferramentas. Podemos utilizar o componente divider, para separar os botões. Veja o código e seu seu resultado:

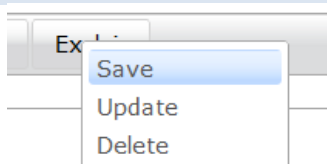


```
<p:fieldset legend="Menus" toggleable="true">
<h:form>
```

```
<p:toolbar>
  <p:toolbarGroup align="left">
    <p:commandButton value="Buscar"/>
    <p:divider/>
    <p:commandButton value="Novo"/>
    <p:commandButton value="Salvar"/>
    <p:commandButton value="Excluir"/>
  </p:toolbarGroup>
  <p:toolbarGroup align="right">
    <p:commandButton value="Sair"/>
  </p:toolbarGroup>
</p:toolbar>
</h:form>
</p:fieldset>
```

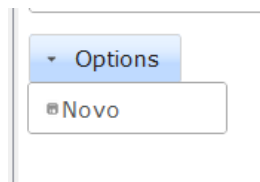
O componente `contextMenu` serve para criar menus que aparecem ao clicar no botão direito do mouse.

```
<h:form>
  <p:contextMenu>
    <p:menuitem value="Save" actionListener="..."/>
    <p:menuitem value="Update" actionListener="..."/>
    <p:menuitem value="Delete" actionListener="..."/>
  </p:contextMenu>
</h:form>
```



`MenuButton` é um componente que cria um botão-menu, ao ser clicado, ele estende um menu criado. No exemplo a seguir, percebe-se que podemos colocar uma pequena imagem no menu.

```
<h:form>
  <p:menuButton value="Options">
    <p:menuitem value="Novo" action="novoCadastro.jsf" ajax="false"
    icon="ui-icon ui-icon-disk"/></p:menuitem>
  </p:menuButton>
</h:form>
```



Temos outros menus oferecidos pelo prime-faces.

```
<h:form>
    <p:growl id="messages"/>
    <h3>Tiered Menu</h3>
    <p:menu type="tiered">
        <p:submenu label="Ajax Menuitems" icon="ui-icon ui-icon-
refresh">
            <p:menuitem value="Save" actionListener="#{buttonBean.save}"
update="messages" icon="ui-icon ui-icon-disk" />
            <p:menuitem value="Update"
actionListener="#{buttonBean.update}" update="messages" icon="ui-icon ui-
icon-arrowrefresh-1-w" />
        </p:submenu>
        <p:submenu label="Non-Ajax Menuitem" icon="ui-icon ui-icon-
newwin">
            <p:menuitem value="Delete"
actionListener="#{buttonBean.delete}" update="messages" ajax="false"
icon="ui-icon ui-icon-close"/>
        </p:submenu>
        <p:submenu label="Navigations" icon="ui-icon ui-icon-extlink">
            <p:submenu label="Prime Links">
                <p:menuitem value="Prime" url="http://www.prime.com.tr" />
                <p:menuitem value="PrimeFaces"
url="http://www.primefaces.org" />
            </p:submenu>
            <p:menuitem value="TouchFaces"
url="#{request.contextPath}/touch" />
        </p:submenu>
    </p:menu>
    <h3>Sliding Menu</h3>
    <p:menu type="sliding">
        <p:submenu label="Ajax Menuitems"
icon="ui-icon ui-icon-refresh">
```

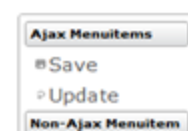
Tiered Menu



Sliding Menu



Regular Menu




```

        <p:menuitem value="Save"
actionListener="#{buttonBean.save}"
update="messages" icon="ui-icon ui-icon-disk" />
        <p:menuitem value="Update"
actionListener="#{buttonBean.update}"
update="messages" icon="ui-icon ui-icon-arrowrefresh-
1-w"/>
    </p:submenu>
    <p:submenu label="Non-Ajax Menuitem"
icon="ui-icon ui-icon-newwin">
        <p:menuitem value="Delete"
actionListener="#{buttonBean.delete}"
update="messages" ajax="false" icon="ui-icon ui-icon-
close"/>
    </p:submenu>
    <p:submenu label="Navigations" icon="ui-icon ui-icon-extlink">
        <p:submenu label="Prime Links">
            <p:menuitem value="Prime" url="http://www.prime.com.tr" />
            <p:menuitem value="PrimeFaces"
url="http://www.primefaces.org" />
        </p:submenu>
        <p:menuitem value="TouchFaces"
url="#{request.contextPath}/touch" />
    </p:submenu>
</p:menu>
<h3>Regular Menu</h3>
<p:menu>
    <p:submenu label="Ajax Menuitems">
        <p:menuitem value="Save" actionListener="#{buttonBean.save}"
update="messages" icon="ui-icon ui-icon-disk"/>
        <p:menuitem value="Update"
actionListener="#{buttonBean.update}" update="messages" icon="ui-icon ui-
icon-arrowrefresh-1-w"/>
    </p:submenu>
    <p:submenu label="Non-Ajax Menuitem">
        <p:menuitem value="Delete"
actionListener="#{buttonBean.delete}" update="messages" ajax="false"
icon="ui-icon ui-icon-close"/>
    </p:submenu>
    <p:submenu label="Navigations">

```

```

        <p:menuitem value="Home" url="http://www.primefaces.org"
icon="ui-icon ui-icon-home"/>
        <p:menuitem value="TouchFaces"
url="#{request.contextPath}/touch" icon="ui-icon ui-icon-star"/>
    </p:submenu>
</p:menu>
<h3>Dynamic Position</h3>
<p:commandButton id="dynaButton" value="Show" type="button"/>
<p:menu position="dynamic" trigger="dynaButton" my="left top"
at="left bottom">
    <p:submenu label="Ajax Menuitems">
        <p:menuitem value="Save" actionListener="#{buttonBean.save}"
update="messages" icon="ui-icon ui-icon-disk"/>
        <p:menuitem value="Update"
actionListener="#{buttonBean.update}" update="messages" icon="ui-icon ui-
icon-arrowrefresh-1-w"/>
    </p:submenu>
    <p:submenu label="Non-Ajax Menuitem">
        <p:menuitem value="Delete"
actionListener="#{buttonBean.delete}" update="messages" ajax="false"
icon="ui-icon ui-icon-close"/>
    </p:submenu>
    <p:submenu label="Navigations">
        <p:menuitem value="Home" url="http://www.primefaces.org"
icon="ui-icon ui-icon-home"/>
        <p:menuitem value="TouchFaces"
url="#{request.contextPath}/touch" icon="ui-icon ui-icon-star"/>
    </p:submenu>
</p:menu>
</h:form>

```

Dock Menu - é um menu especial que imita o menu dos computadores Mac.



```

<p:dock position="top">
    <p:menuitem value="Home" icon="/images/dock/home.png" url="#" />
    <p:menuitem value="Music" icon="/images/dock/music.png" url="#" />

```

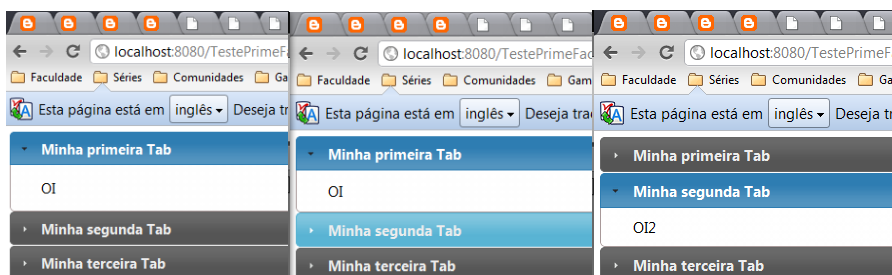
```
<p:menuitem value="Video" icon="/images/dock/video.png" url="#" />
<p:menuitem value="Email" icon="/images/dock/email.png" url="#" />
<p:menuitem value="Portfolio" icon="/images/dock/portfolio.png"
url="#" />
<p:menuitem value="Link" icon="/images/dock/link.png" url="#" />
<p:menuitem value="RSS" icon="/images/dock/rss.png" url="#" />
<p:menuitem value="History" icon="/images/dock/history.png"
url="#" />
</p:dock>
```

Painéis e Efeitos

Nessa parte iremos ver sobre painéis e efeitos de transição. Para isso criaremos uma nova página chamada testepainels. Onde terá nossos painéis.

AccordionPanel é um painel especial que contem uma lista de Titulos, ao clicar no titulo, abre-se o painel com um conteúdo. Podendo ter qualquer coisa dentro.

```
<p:accordionPanel>
  <p:tab title="Minha primeira Tab">
    <h:outputText value="OI" />
  </p:tab>
  <p:tab title="Minha segunda Tab">
    <h:outputText value="OI2" />
  </p:tab>
  <p:tab title="Minha terceira Tab">
    <h:outputText value="OI3" />
  </p:tab>
</p:accordionPanel>
```



Efeitos em painéis – os painéis podem ter efeitos diversos. Use o código a seguir e confira.

Blind click	Clip click	Drop click	Explode click
Fold doubleclick	Puff doubleclick	Slide doubleclick	Scale doubleclick
Bounce click	Pulsate click	Shake click	Size click

```

<p:fieldset legend="Meu teste!!!">
  <h:panelGrid columns="4">

    <p:panel header="Blind">
      <h:outputText value="click" />
      <p:effect type="blind" event="click">
        <f:param name="direction" value="horizontal" />
      </p:effect>
    </p:panel>

    <p:panel header="Clip">
      <h:outputText value="click" />
      <p:effect type="clip" event="click" />
    </p:panel>

    <p:panel header="Drop">
      <h:outputText value="click" />
      <p:effect type="drop" event="click" />
    </p:panel>

    <p:panel header="Explode">
      <h:outputText value="click" />
      <p:effect type="explode" event="click" />
    </p:panel>

    <p:panel header="Fold">
      <h:outputText value="doubleclick" />
      <p:effect type="fold" event="dblclick" />
    </p:panel>

    <p:panel header="Puff">
      <h:outputText value="doubleclick" />
      <p:effect type="puff" event="dblclick" />

```

```
</p:panel>

<p:panel header="Slide">
  <h:outputText value="doubleclick" />
  <p:effect type="slide" event="dblclick" />
</p:panel>

<p:panel header="Scale">
  <h:outputText value="doubleclick" />
  <p:effect type="scale" event="dblclick">
    <f:param name="percent" value="90" />
  </p:effect>
</p:panel>

<p:panel header="Bounce">
  <h:outputText value="click" />
  <p:effect type="bounce" event="click" />
</p:panel>

<p:panel header="Pulsate">
  <h:outputText value="click" />
  <p:effect type="pulsate" event="click" />
</p:panel>

<p:panel header="Shake">
  <h:outputText value="click" />
  <p:effect type="shake" event="click" />
</p:panel>

<p:panel header="Size">
  <h:outputText value="click" />
  <p:effect type="size" event="click">
    <f:param name="to" value="{width: 200,height: 60}" />
  </p:effect>
</p:panel>

</h:panelGrid>
</p:fieldset>
```

Tabelas

Vimos no decorrer do curso que podemos criar tabelas para mostrar dados do banco. O primeFaces cria mais rapidamente essas tabelas, e com controles e funções diversas. Vamos modificar nosso beans para criar uma lista de animais e criar uma tabela com base nela.

```
private List<String> animais;

public List<String> getAnimais() {
    animais = new ArrayList<String>();
    animais.add("Girafa");
    animais.add("Pato");
    animais.add("Leopardo");
    animais.add("Elefante");
    animais.add("Zebra");
    return animais;
}

public void setAnimais(List<String> animais) {
    this.animais = animais;
}
```

Iremos criar um componente dataTable, que pode, inclusive, ser página e muitas outras funções.

```
<h:form>
    <p:dataTable value="#{tableBean.animais}" var="a">
        <p:column headerText="Nome dos animais">
            <h:outputText value="#{a}" />
        </p:column>
        <p:column headerText="Nome dos animais 2">
            <h:outputText value="#{a}" />
        </p:column>
    </p:dataTable>
</h:form>
```

Nome dos animais	Nome dos animais 2
Girafa	Girafa
Pato	Pato
Leopardo	Leopardo
Elefante	Elefante
Zebra	Zebra
Girafa	Girafa
Pato	Pato
Leopardo	Leopardo
Elefante	Elefante
Zebra	Zebra
Girafa	Girafa
Pato	Pato
Leopardo	Leopardo
Elefante	Elefante
Zebra	Zebra

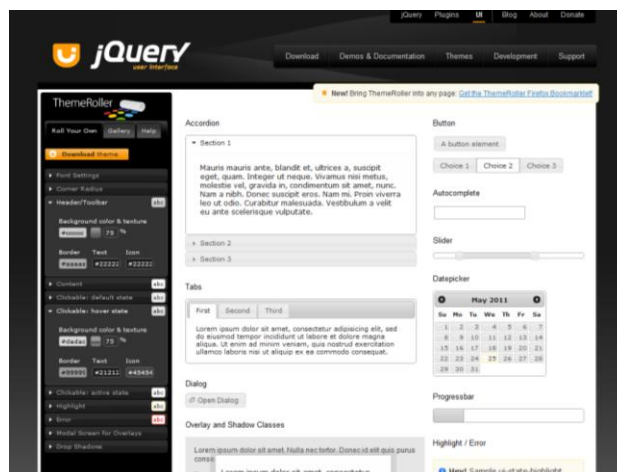
Adicionando uma paginação com até 4 linhas:
`paginator="true" rows="4"`

Temos:

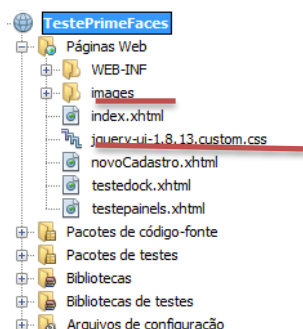
Nome dos animais	Nome dos animais 2
Girafa	Girafa
Pato	Pato
Leopardo	Leopardo
Elefante	Elefante

Temas

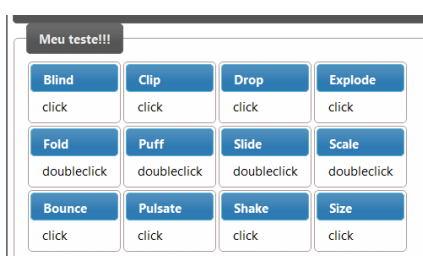
O primeFaces tem diversos temas prontos, e a possibilidade de personalizar e criar novos temas. O site <http://jqueryui.com/themeroller/> oferece todas as ferramentas necessárias para criação de temas para primeFaces basta utilizar o menu do lado esquerdo.



Para usar os temas criados, você deverá baixar o tema pronto do site e colocar a pasta de imagens e o arquivo .css da pasta gerada em seu projeto.



Veja um exemplo que eu crie (a pagina de painéis usada nesse tutorial):



Considerações finais

A lista de componentes do PrimeFaces é enorme, tem muitos componentes, variações e atributos. Explore a biblioteca visual que esta a disposição no site:

<http://www.primefaces.org/showcase/ui/home.jsf>

Exercícios Complementares de Fixação

- 1) Refaça os exercícios feitos em aula.
- 2) Refaça os exercícios complementares das outras aulas aplicando componentes PrimeFaces. Crie seus temas e desafie a imaginação.

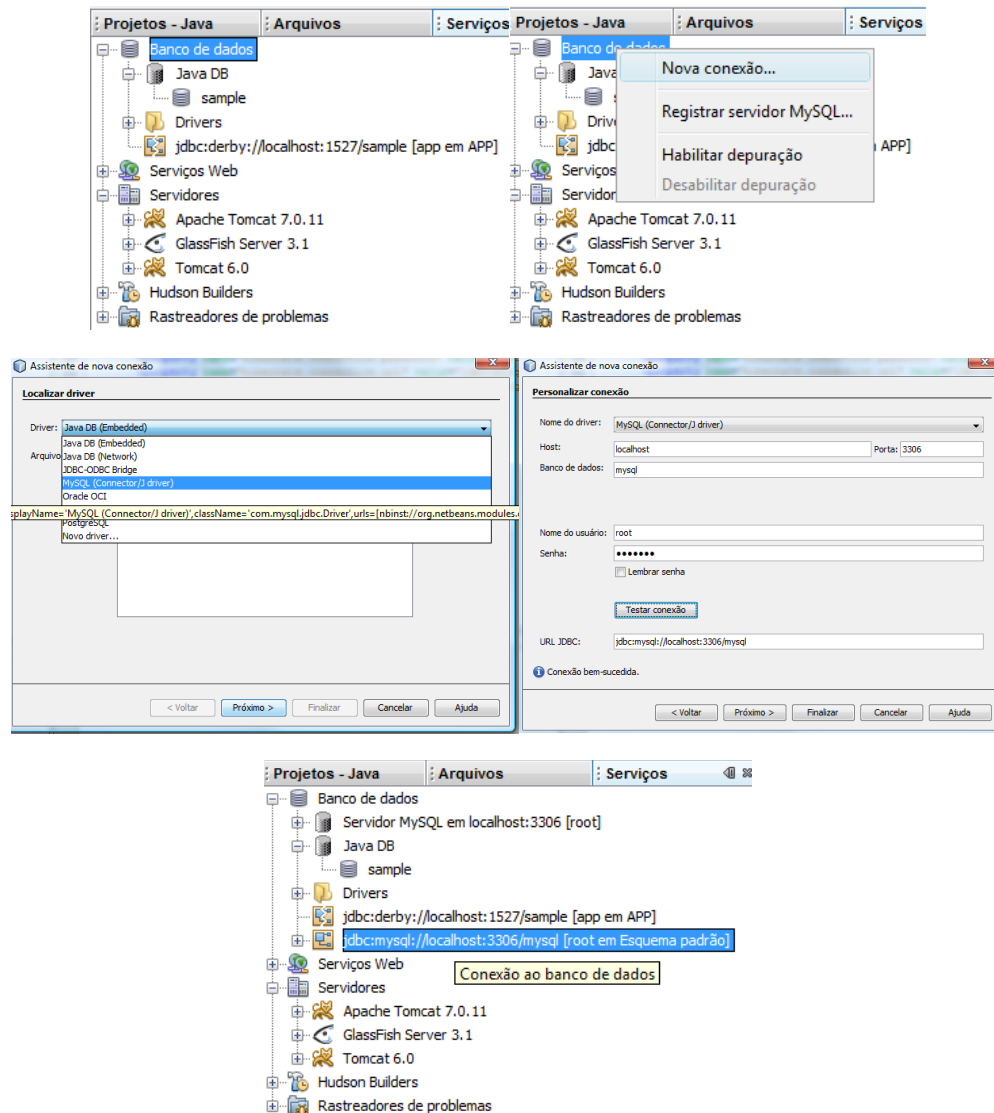
Exercícios de Pesquisa:

- 1) Pesquise sobre RIA e sua importância.
- 2) Pesquise sobre outros frameworks RIA.
- 3) Pesquise sobre ferramentas RIA que podem ser utilizadas com Java como o Flex da Adobe.



Apêndices

Anexo 2 – Configurando o acesso ao MYSQL no NetBeans



The image shows a sequence of steps in NetBeans to configure MySQL access:

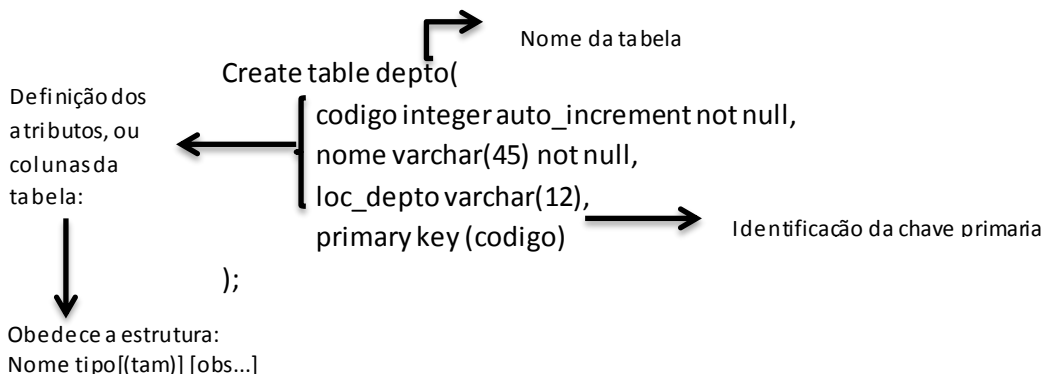
- Top Panel:** The 'Serviços' (Services) tab is active. A context menu is open over the 'Banco de dados' (Database) node, showing options: 'Nova conexão...' (New connection...), 'Registrar servidor MySQL...' (Register MySQL server...), 'Habilitar depuração' (Enable debugging), and 'Desabilitar depuração' (Disable debugging).
- Assistente de nova conexão (New Connection Wizard):**
 - Localizar driver (Locate driver):** The 'Driver' dropdown is set to 'MySQL (Connector/J driver)'. The 'Arquivo' (File) list shows the selected driver file.
 - Personalizar conexão (Customize connection):**
 - Nome do driver: MySQL (Connector/J driver)
 - Host: localhost
 - Porta: 3306
 - Banco de dados: mysql
 - Nome do usuário: root
 - Senha: (masked with dots)
 - URL JDBC: jdbc:mysql://localhost:3306/mysql
- Final Result:** The 'Serviços' tab now shows a new entry: 'Servidor MySQL em localhost:3306 [root]'. The 'Banco de dados' node is expanded, showing 'Servidor MySQL em localhost:3306 [root]' and 'Java DB'. A tooltip 'Conexão ao banco de dados' (Connect to database) is visible over the new MySQL server entry.

Anexo3 – Noções de SQL

Criando Tabelas:

Exemplo:

Criação de uma tabela departamento com os campos: código de departamento, nome do departamento, localização do departamento.



Inserir registros:

```
Insert into <nome da tabela> [(coluna1,coluna2,...)]
[value/values] (valor_coluna1, valor_coluna2,...);
```

Exs: insert into depto values ('1','marketing','13º.Andar');

Insert into depot (nome,loc_depto) values ('financeiro','12o.andar');

Atualizar registros:

```
Update <nome da tabela> set coluna1=valor1,... [where
<condição>];
```

Ex: Update depto set loc_depto='1º.Andar' where nome='financeiro';

Deletar registros:

```
Delete from <nome da tabela> [where <condição>];
```

Ex: delete from depto where nome='marketing';

Visualizar registros:

```
Select * | coluna1,coluna2,... from <nome da tabela> [where
<condição>];
```

Ex: select * from depto; - seleciona todos os registros

Select * from depto where codigo=1; - seleciona o registro 1

Select nome from depto where código=2; - seleciona apenas o nome do registro 2

Aqui, mostramos o básico necessário para desenvolver aplicações CRUD, o restante (que não entra no escopo do curso) pode ser visto em materiais na internet ou outros cursos.