

Conteúdo

1- Introdução a Java Web	1
Um pouco de história.....	3
Servidor Tomcat e NetBeans – Instalação	4
2-Os tipos de Aplicações	7
3-Noções de HTML.....	8
Tabelas	8
Formulários	9
4-Páginas JSP – Java Server Pages.....	11
Scriptlet ou Scripting.....	12
Expression Language.....	12
Classes em JSP.....	12
Parâmetros.....	12
Necessidade do JavaScript	13
Como colocar Funções de JavaScript em HTML.....	14
Validação de formulários	14
Extra- Validando por JSP (Descobrimos o porque o JavaScript é melhor e mais fácil)	15
Exercícios Complementares de fixação:	18
Exercícios de Pesquisa:	20
5-Servlets	21
Exemplo de processamento de formulários usando um servlets.....	23
6-Sessão e Java Beans.....	25
Duas formas de modificar o escopo de um parâmetro.	25
JavaBeans – um estudo.....	28
7-TagLibs, o JSTL.....	31
Exercícios Complementares de fixação:	33
Exercícios de Pesquisa:	33
8-Banco de dados.....	34

JDBC (Java Data Base Connectivity)	34
10-MVC (Model - View – Controller)	36
11-Padrão DAO	37
Exercícios Complementares de Fixação:.....	42
12-JavaEE6 – Conhecendo o JSF – Java Server Faces.....	44
Exercícios Complementares de Fixação:.....	53
Exercícios de Pesquisa	53
13-RIA – Rich Internet Applications – Aplicações ricas para Internet.	54
14-Frameworks RIA para JSF 2.0 – PrimeFaces.....	57
Botões e mensagens	58
Formulários	60
Menus	63
Painéis e Efeitos	68
Tabelas	71
Temas.....	72
Considerações finais	73
Exercícios Complementares de Fixação	74
Exercícios de Pesquisa:	74
Anexo 1 – Hibernate e JPA (no Eclipse)	75
Hibernate	75
O JPA (Java Persistence API (Application Programming Interface).....	76
Hibernate Conceitos	80
Anexo 2 – Configurando o acesso ao MYSQL no NetBeans	86
Anexo3 – Noções de SQL	87
Criando Tabelas:.....	87
Inserir registros:	87
Atualizar registros:	87
Deletar registros:	87
Visualizar registros:.....	87

1- 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.

Atualmente as famílias de produtos disponíveis no atual JEE se resumem assim:

Web Application Technologies

JavaServer Faces 1.2

JavaServer Pages 2.1

JavaServer Pages Standard Tag Library

Java Servlet 2.5

Enterprise Application Technologies

Common Annotations for the Java Platform

Enterprise JavaBeans 3.0

J2EE Connector Architecture 1.5

JavaBeans Activation Framework (JAF) 1.1

JavaMail

Java Data Objects (JDO)

Java Message Service API

Java Persistence API

Java Transaction API (JTA)

Web Services Technologies

Implementing Enterprise Web Services

Java API for XML-Based Web Services (JAX-WS) 2.0

Java API for XML-Based RPC (JAX-RPC) 1.1

Java Architecture for XML Binding (JAXB) 2.0

SOAP with Attachments API for Java (SAAJ)

Streaming API for XML

Web Service Metadata for the Java Platform

Management and Security Technologies

J2EE Application Deployment

J2EE Management

Java Authorization Contract for Containers

Cada um destes pode possuir subdivisões e tecnologias internas, constituindo uma extensa lista de opções de produtos e ferramentas. Mas a boa notícia é que ninguém tem que saber tudo ou dificilmente vai ter que deter o conhecimento de tudo ou utilizar tudo ao mesmo tempo para a construção de uma aplicação web. Provavelmente serão usadas de duas a cinco tecnologias combinadas para construir uma grande, boa, robusta e confiável solução.

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; Java ServerFaces, a tecnologia do JavaEE 6.

Caso possua alguma colaboração para com nosso material como correções de erros, sugestões e novos exemplos, entre em contato conosco. Conforme as dúvidas irão surgindo é comum o aluno procurar listas de discussões e fóruns a respeito. Nós, da It Training, aconselhamos o GUJ (Grupo de Usuários Java – www.guj.com.br). É possível tirar dúvidas por e-mail com os nossos instrutores.

Um pouco de história

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 famosodeles 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.

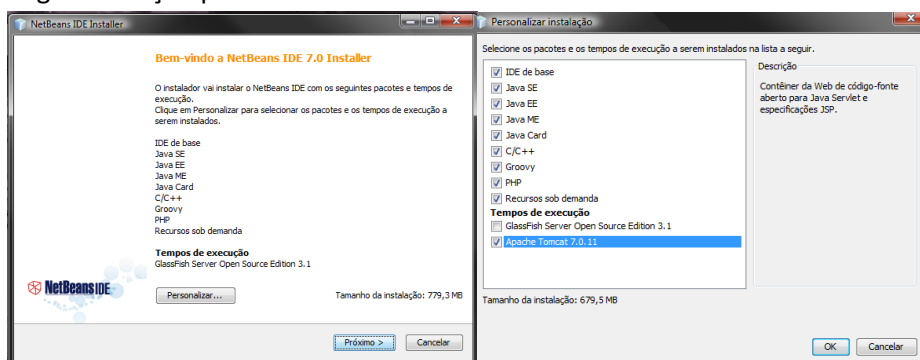
Servidor Tomcat e NetBeans – Instalação

O JSP necessita rodar sobre um servidor, mais precisamente ele é instalado sobre um servidor e assim, a aplicação JSP pode ser aberta por qualquer navegador de internet. Iremos agora mostrar passo a passo a instalação do Tomcat e do Netbeans.

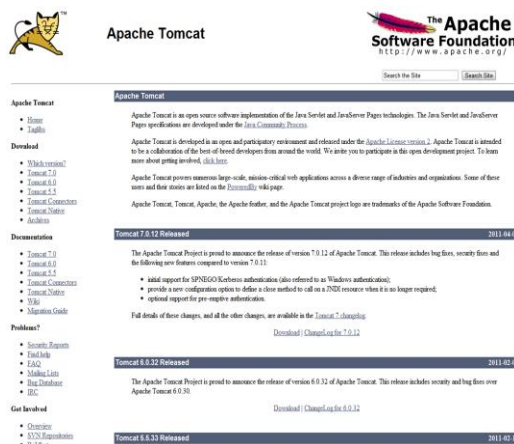
- 1- Vá para o site : <http://netbeans.org/>
- 2- Clique em Download FREE e no botão download na coluna JAVA na página seguinte e Seelcione a versão JavaEE



3- Após baixar o Netbeans, siga as instruções de instalação normalmente. O Apache Tomcat 6.0.26 está incluído nas opções de download "Java" e "Tudo", mas não é instalado por padrão nessas opções. Para instalar o Apache Tomcat da opção de download Java ou "Tudo", inicie o instalador e selecione Apache Tomcat 6.0.26 na caixa de diálogo Instalação personalizada.

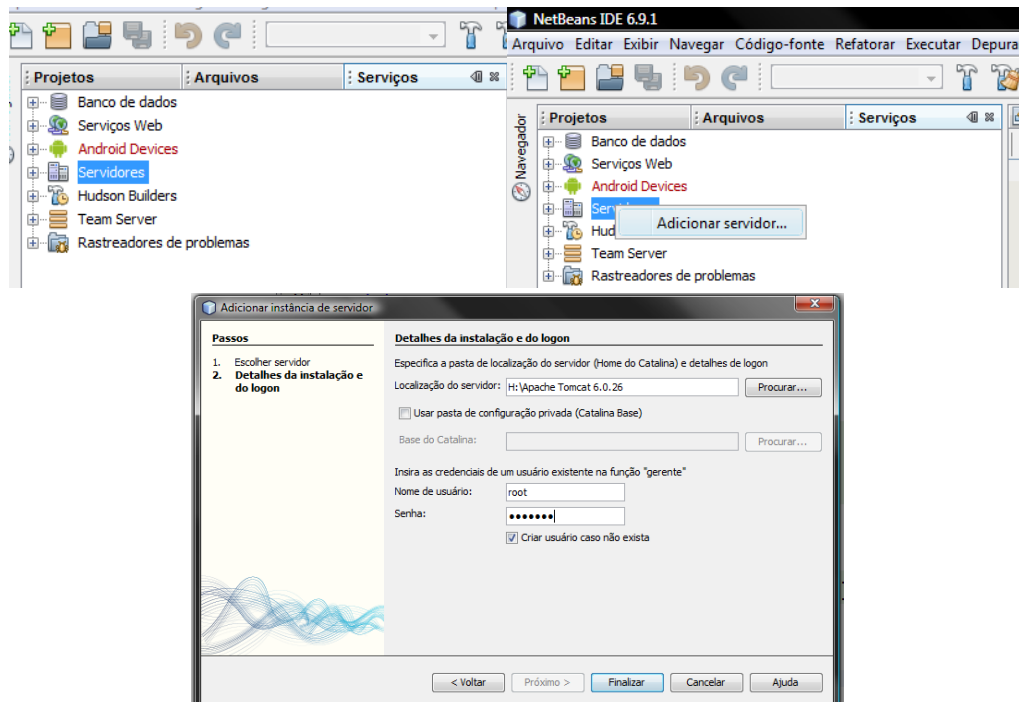


Atenção: selecione também o GlassFish, pois esse será utilizado como JSF.



4- Ou você pode ir até o site da Apache: <http://tomcat.apache.org/>, baixar o tomcat 6.0 (não baixe o 7.0 pois esse só é compatível com a versão 7.0 do Netbeans e estamos utilizando a 6.9.1). Descompacte a pasta zip baixada e coloque no diretório C: ou em outro lugar de sua preferência. Após isso, entre na interface Netbeans para criar a conexão com esse servidor.

5- Uma vez aberto o Netbeans, vá para a aba “Serviços”. Clique com o botão direito em “Adicionar Servidor”. Escolha Tomcat 6.0 , clique em “Próximo” e na próxima janela, procure pela pasta descompactada, crie um usuário chamado root e crie uma senha própria.



O Tomcat e Netbeans já estão configurados e prontos para o uso.

2-Os tipos de Aplicações

One-Tier (Uma Camada)

No final dos anos 70 e início dos anos 80, ocorreu a mudança de computadores centrais para computadores pessoais (PCs), no qual moveu controle de poucas pessoas para muitas, na época as aplicações eram poderosas mas projetadas e desenvolvidas para tarefas de um único usuário, existia carência de colaboração de dados comuns, não existiam banco de dados centrais ou emails, logo este tipo de aplicação é conhecida como monolíticas ou seja One-Tier, ou seja, aplicações instaladas na máquina e isoladas.

Two-Tier (Cliente Servidor)

A próxima fase do desenvolvimento de softwares eram distribuir e compartilhar dados, interação diretamente com usuário final, as lógicas de apresentação e negócios são armazenadas no cliente e os dados no servidor, aplicações como: clientes de email como Outlook ou Thunderbird.

N-Tier ou Multitier (Aplicações Web)

Com o advento da internet, essa nova vantagem de aplicação permite aos desenvolvedores focalizem em áreas de domínio específicos, dividindo em três camadas – MVC (Model, modelo – acesso a dados; View, visão – camada de apresentação; Controller, controlador – lógica), essa regra foi introduzida na linguagem SmallTalk, espalhando pela comunidade nos anos 80 e utilizado até hoje.

3-Noções de HTML

HTML (Hyper Text Markup Language - Linguagem de Marcação de Hipertexto) é uma linguagem para descrever páginas da internet. 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 e .

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.
 ****um link****
 **
** - 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

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.

Ex:

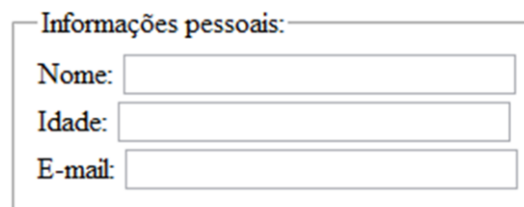
```
<table border="1">  
  <tr>  
    <td>L1, C1</td>  
    <td>L1, C2</td>  
  </tr>  
  <tr>  
    <td>L2, C1</td>  
    <td>L2, C2</td>  
  </tr>  
</table>
```

L1, C1	L1, C2
L2, C1	L2, C2

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.

Ex:



Text Fields

```
<form>
```

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

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

```
</form>
```

Ex:



Password Fields

```
<form>
```

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

```
</form>
```

Ex:



Radio Buttons

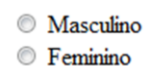
```
<form>
```

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

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

```
</form>
```

Ex:



Checkboxes

```
<form>
```

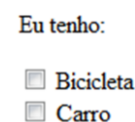
```
<p>Eu tenho:</p>
```

```
<input type="checkbox" name="veiculo" value="Bicicleta" /> Bicicleta<br />
```

```
<input type="checkbox" name="veiculo" value="Carro" /> Carro
```

```
</form>
```

Ex:



Submit Button

```
<form name="input" action="http://www.algumapagina.com/recebendo" method="post">
```

```
Usuario: <input type="text" name="usuario" />
```

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

```
</form>
```

Ex:



Select list

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

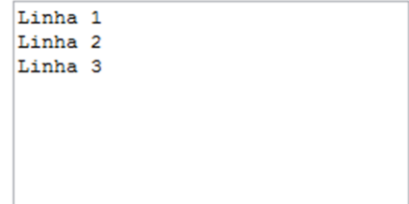
Ex:



Textarea

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

Ex:



4-Páginas JSP – Java Server Pages

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_página"> - 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 retornam 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).

Expression Language

Similares aos scriptlets `<%= %>`. São usadas para expressões que retornem valores, JavaBeans (vistos mas a frente) e parâmetros. Sua sintaxe é: `${...}`

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 resultar 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.

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 validos 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 (Descobrimos o porque o JavaScript é melhor e mais fácil)

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

```
<%@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>
    <form action="form4proc.jsp" method="post">
      <table border="0">
        <tr><td><b>Formulário 4</b></td></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></td>
          <td>
            <input type="submit" value="OK">
            <input type="reset" value="Limpar">
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>

```

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

```

<% //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 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>
```

Veja:

Formulário 4

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

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 ira verificar se existem os parâmetros de erro. Caso eles existam, ele mostra uma mensagem de acordo com o erro, senão nada acontece. No caso de um erro, ele retorna e mostra um erro acima do formulário.

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 pagina, 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 pagina 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.

Exercícios Complementares de fixação:

- 1- Refaçam em casa os exercícios feitos em aula.
- 2- 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
 - i. Nome: String
 - ii. RA: Integer
 - iii. Curso: String
 - iv. Ano: Integer
 - v. Período: String
 - vi. Endereço: String
 - vii. Cidade: String
 - viii.
 - ix. Estado: String
 - x. Telefone: String
 - b. Produto
 - i. Codigo: Integer
 - ii. Descrição: String
 - iii. Quantidade em estoque: Integer
 - iv. Valor: Float
 - v. Quantidade mínima em estoque: Integer
 - vi. Quantidade máxima para compra: Integer
 - vii. Fornecedor: String
 - c. Fornecedor
 - i. Cnpj: Integer
 - ii. Nome: String
 - iii. Endereço: String
 - iv. Cidade: String
 - v. Estado: String
 - vi. Telefone: String
 - vii. E-mail: String
 - d. Cliente
 - i. Nome: String
 - ii. RG: Integer

- iii. CPF: Integer
 - iv. Idade : Integer
 - v. Endereço: String
 - vi. Cidade: String
 - vii. Estado: String
 - viii. Telefone: String
 - ix. E-mail: String
 - e. Escola
 - i. Cnpj: Integer
 - ii. Nome: String
 - iii. Endereço: String
 - iv. Cidade: String
 - v. Estado: String
 - vi. Telefone: String
 - vii. E-mail: String
 - f. Curso
 - i. Código: Integer
 - ii. Nome: String
 - iii. Descrição: String
 - iv. Coordenador: String
 - v. CargaHorária: Integer
- 3- 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.
- 4- 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).
- 5- 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

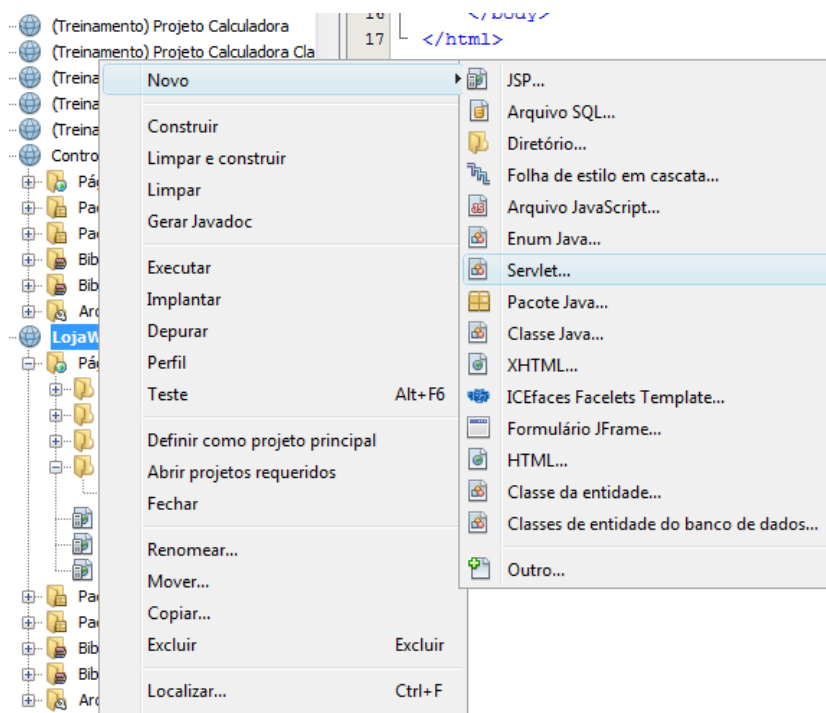
5-Servlets

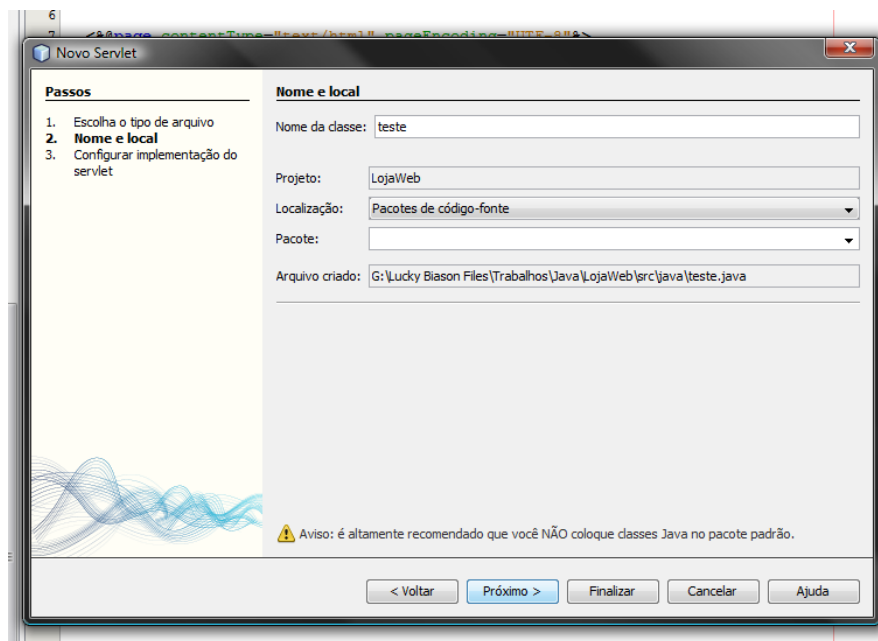
Formulários HTML podem receber entradas via caixa de texto, mas eles não conseguem processar ou usar essas informações para criar uma resposta dinâmica. Uma forma de processar informação de entrada é ter um **servlets** – trecho especial de código em Java que pode extrair informação de uma solicitação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

Um servlets é simplesmente um tipo especial de classe Java que tem uma estrutura predefinida, atributos e métodos que são chamados em uma sequência predefinida e específica, toda vez que houver uma solicitação para o servlets.

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:

```

3
4 import java.io.IOException;
5 import java.io.PrintWriter;
6 import javax.servlet.ServletException;
7 import javax.servlet.http.HttpServlet;
8 import javax.servlet.http.HttpServletRequest;
9 import javax.servlet.http.HttpServletResponse;
10
11 /**
12  *
13  * @author Lucas Biason
14  */
15 public class teste extends HttpServlet {
16
17
18     protected void processRequest(HttpServletRequest request, HttpServletResponse response)
19     throws ServletException, IOException {
20         response.setContentType("text/html;charset=UTF-8");
21         PrintWriter out = response.getWriter();
22         try {
23             /* TODO output your page here
24              */
25             out.println("<html>");
26             out.println("<head>");
27             out.println("<title>Servlet teste</title>");
28             out.println("</head>");
29             out.println("<body>");
30             out.println("<h1>Servlet teste at " + request.getContextPath () + "</h1>");
31             out.println("</body>");
32             out.println("</html>");
33             /*
34              */
35         } finally {
36             out.close();
37         }
38     }
39
40     /*
41      *
42      */
43 }

```

HttpServlet methods. Click on the + sign on the left to edit the code.

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:

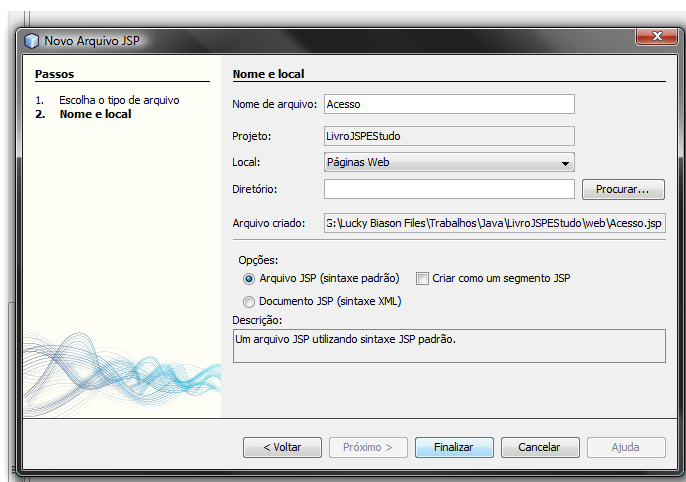
```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

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

Dentro desses métodos você pode programar de forma diferenciada.

Exemplo de processamento de formulários usando um servlets

1º) Vamos criar uma nova JSP chamada Acesso.



2º) Vamos criar o conteúdo da página. Um pequeno formulário com o Primeiro e Ultimo Nome de uma Pessoa.

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

<html>

<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>
```

Primeiro Nome:

Segundo Nome:

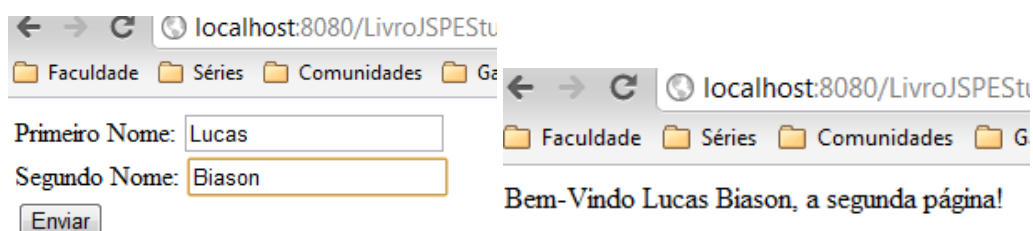
3º) Vamos criar um servlets para receber os parâmetros do formulário.

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

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

    PrintWriter out = response.getWriter();
    out.print("<html><body>"
        + "Bem-Vindo "
        + nome + " "+segundonome
        + ", a segunda página!"
        + "</body></html>");
    out.close();
}
```

Testando:



The screenshot shows two browser windows. The left window displays the web form with 'Lucas' entered in the 'Primeiro Nome' field and 'Biason' in the 'Segundo Nome' field. The right window shows the result of the form submission: 'Bem-Vindo Lucas Biason, a segunda página!'.

6-Sessão e Java Beans

Duas 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.

Inicio.jsp:

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

<html>
  <head>
  </head>
  <body>
    <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>
  </body>
</html>
```

Validar.jsp:

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

<%
    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:

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

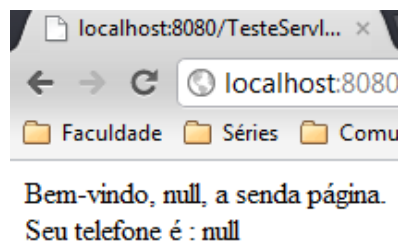
<html>
  <body>
    Bem-vindo, <%= request.getParameter("nome") %>, a segunda página.<br/>
    Seu telefone é : <%= request.getParameter("telefone") %>
  </body>
</html>
```

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:

- a) 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.).
- b) Usando servlets (visto mais a frente) (dificulta a programação e criação da pagina jsp).
- c) Usar JavaBean (uma boa opção, mas o Javabeans é mais complicado que mexer em certos casos, nesse nem tanto)
- d) 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).

A página Validar ficará:

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

A página ProcessarAcesso ficara:

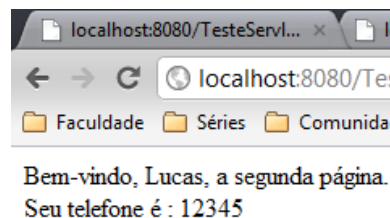
```
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<html>
<body>

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

</body>
</html>
```

Com isso o problema foi resolvido:



JavaBeans – um estudo

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 utiliza-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!!!!

7-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>.

Core taglibs (apenas veremos parte dessa biblioteca. O restante estará disponível no site do Java):

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 o exercício 2 complementar da aula 01. Use servlets para receber os parâmetros, e envie para uma página de resposta.
- 4- Refaça o exercício 2 complementar da aula 01. Use sessões e valide os campos utilizando JSP. (conforme foi visto do exemplo sobre sessões).
- 5- Refaça o exercício 2 complementar da aula 01. Use JavaBeans para passar o objeto para uma terceira página e mostre os valores do objeto na página.
- 6- Refaça o exercício 2 complementar da aula 01. Misture o uso de Sessões, JavaBeans e Taglibs.
- 7- Troque as estruturas de controle (if, for, while...) nos exercícios anteriores por taglibs.
- 8- Refaça os exercícios 3 e 4 na aula 01 nas mesmas condições propostas nos exercícios de 3 a 7 dessa aula.

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.

8-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.

(Wikipédia)

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:

```
| import com.mysql.jdbc.Connection;  
· import java.sql.*;  
  
public class Conexao {  
  
    public static Connection getConn(){  
        try{  
            Class.forName("com.mysql.jdbc.Driver");  
            return (Connection)  
                DriverManager.getConnection("jdbc:mysql://localhost:3306/banco", "usuario", "senha.");  
        } catch (ClassNotFoundException ex) {  
            System.out.println(ex);  
            return null;  
        } catch (SQLException s) {  
            System.out.println(s);  
            return null;  
        }  
    }  
}
```

Na linha:

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

Deve ser colocado o nome do banco de dados, o nome do usuário, que muitas vezes vem por padrão “root” e a senha do SGBD.

Explicando o código acima:

```
//criando uma classe de conexao,
//chamada conexao para ser chamada sempre que for preciso abrir uma conexao com o banco de dados.
public class Conexao {

    //criando um método que realize e retorne essa conexão
    public static Connection getConn(){
        try{
            //aqui vamos mostrar qual é o driver que iremos utilizar,
            //o driver mostra como o JDBC deve se comportar
            //e comunicar com o SGBD, isso é,
            //ele indica qual é o SGBD que iremos nos conectar
            //e como será realizada essa conexão.
            //aqui utilizaremos o JDBC para o MySQL, mas existem outros, um para cada SGBD.
            Class.forName("com.mysql.jdbc.Driver");
            //aqui nós vamos retornar a conexão pronta.
            return (Connection)
                DriverManager.getConnection("jdbc:mysql://localhost:3306/banco", "usuario", "senha.");
            //exceções são criadas para verificar se a classe do driver foi encontrado ou não,
            //e se ocorreu um erro de conexão com o banco de dados.
        } catch (ClassNotFoundException ex) {
            System.out.println(ex);
            return null;
        } catch (SQLException s) {
            System.out.println(s);
            return null;
        }
    }
}
```

Uma vez conectado, estamos prontos para a manipulação do banco , isto é , inserir comandos SQL.

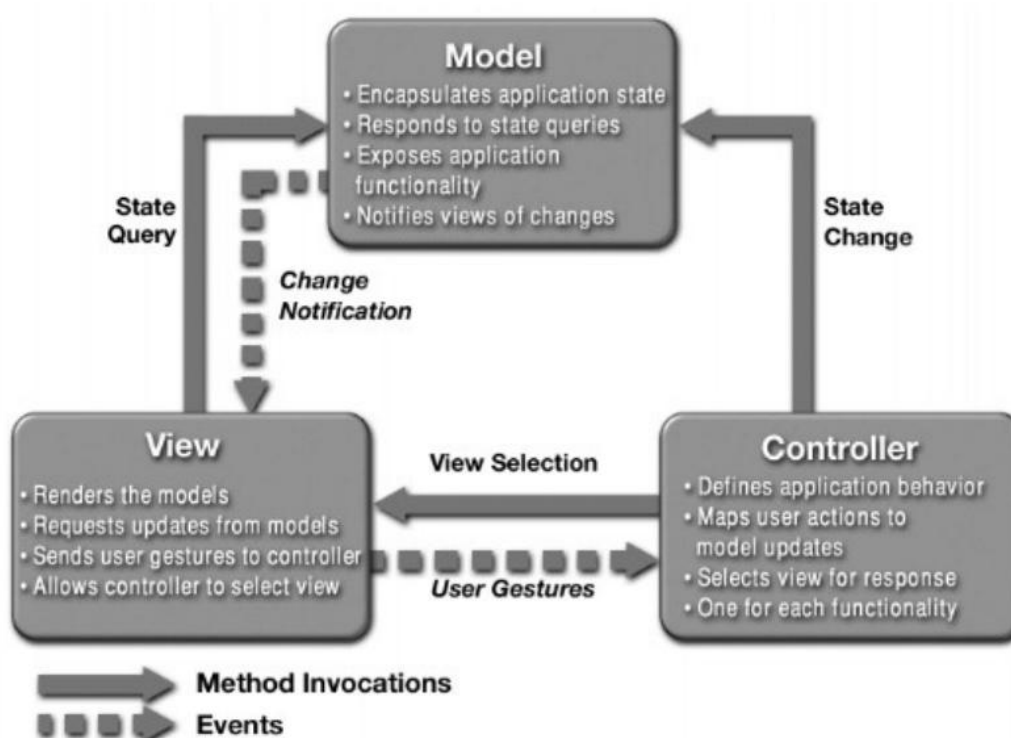
10-MVC (Model - View – Controller)

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.

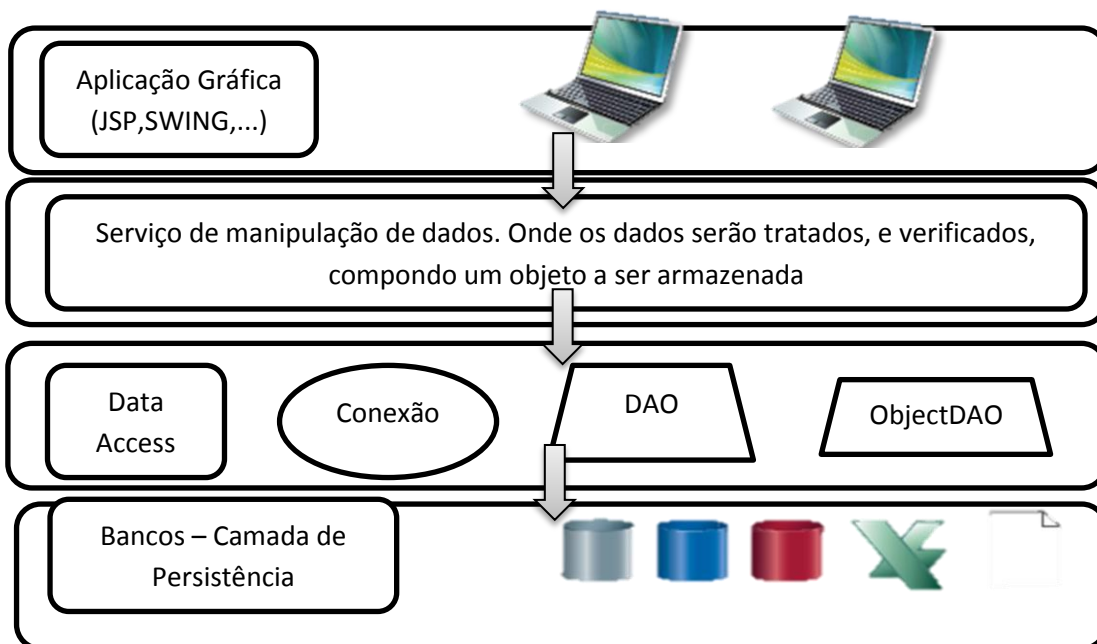
V (View) – Representa a renderização do model, ou seja uma interface para o usuário.

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.



11-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.

```
import java.sql.*;

public abstract class Dao {

    public Connection getConnection(){
        return Conexao.getConn();
    }

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

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

    public ResultSet executeQuery(String query, Object ... params) throws SQLException{
        PreparedStatement ps = getStatement(query);
        for(int i = 0; i< params.length; i++)
            ps.setObject(i+1, params[i]);
        return ps.executeQuery();
    }

    public int executeCommand(String query, Object... params) throws SQLException{
        PreparedStatement ps = getStatement(query);
        for(int i = 0; i< params.length; i++)
            ps.setObject(i+1, params[i]);
        int result = ps.executeUpdate();
        ps.close();
        return result;
    }
}
```

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.

```
import java.sql.*;

public abstract class Dao {

    //esse método cria uma conexão com com a classe Conexao,
    //fazendo com que ela se conecte ao banco. Abrindo assim, uma nova sessão.
    public Connection getConnection(){
        return Conexao.getConn();
    }

    //ambos os métodos a seguir criam Statements, ou instruções semi prontas,
    //para rodar SQL. Elas já estão semi-preparadas.
    //O método mais usado é o getStatement(Stringstatement),
    //onde se passa uma String contendo uma instrução SQL.
    public Statement getStatement() throws SQLException{
        return getConnection().createStatement();
    }

    public 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 colocados nas SQL,
// como os dados de uma inserção, ou atualização, ou remoção e parâmetros de pesquisa.

// 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.
// Ele parâmetro é tratado como um vetor, um conjunto de dados.
// por isso usamos o for para passear pelo vetor, e setarmos as posições dos parâmetros.
// Como veremos mais a frente, onde forem colocados parâmetros nas SQL,
// podemos simplesmente colocar um "?", e depois passarmos o objeto cujo valor
// representa o parâmetro. Por exemplo:
// "select * from aluno where código=?"
// a passamos o parametro aluno.codigo, para recebermos uma tabela que contem o aluno
// cujo código pe igual ao passado.
// outro exemplo, usando o executeCommand, que 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 qua a ordem das suas "?" (posições)
public ResultSet executeQuery(String query, Object ... params) throws SQLException{
    PreparedStatement ps = getStatement(query);
    for(int i = 0; i< params.length; i++)
        ps.setObject(i+1, params[i]);
    return ps.executeQuery();
}

}

public int executeCommand(String query, Object... params) throws SQLException{
    PreparedStatement ps = getStatement(query);
    for(int i = 0; i< params.length; i++)
        ps.setObject(i+1, params[i]);
    int result = ps.executeUpdate();
    ps.close();
    return result;
}

}
```

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. Exemplo:

A Classe Alunos com todos os seus atributos

```
public class Produto {
    private String nome;
    private float valor;
    public Produto() { }
```

```
public Produto(String nome, float valor) {
    this.nome = nome;
    this.valor = valor;
}
public String getNome() {
    return nome;
}
public void setNome(String nome) {
    this.nome = nome;
}
public float getValor() {
    return valor;
}
public void setValor(float valor) {
    this.valor = valor;
}
public String toString(){
    return nome + ", R$" + valor;
}
}
```

Agora, para manipular a persistência.

```
public class ListaProdutos extends Dao{
    public ListaProdutos(){

    }

    public String addProduto(String nome, float valor){
        try {
            executeCommand("insert into produto (nome,valor) values (?,?)", nome, valor);
            return "sucesso";
        } catch (SQLException ex) {
            return "falhou";
        }
    }

    public String upProduto(String nome, float valor){
        try {
            executeCommand("update produto set valor=? where nome=?", valor, nome);
            return "sucesso";
        } catch (SQLException ex) {
            return "falhou";
        }
    }

    public String delProduto(String nome){
        try {
            executeCommand("delete from produto where nome=?", nome);
            return "sucesso";
        } catch (SQLException ex) {
            return "falhou";
        }
    }

    public Produto getProduto(String nome){
        try {
```

```
        ResultSet rs = executeQuery("select * from produto where nome=?",nome);
        rs.next();
        Produto p = new Produto();
        p.setNome(rs.getString("nome"));
        p.setValor(rs.getFloat("valor"));
        return p;
    } catch (SQLException ex) {
        return null;
    }
}

public List<Produto> getProdutos(String filtro){
    try {
        ResultSet rs = executeQuery("select * from produto where nome like ?", "%"+filtro+"%");
        List<Produto> lista = new LinkedList<Produto>();
        Produto p;
        while(rs.next()){
            p = new Produto(rs.getString("nome"),rs.getFloat("valor"));
            lista.add(p);
        }
        return lista;
    } catch (SQLException ex) {
        return null;
    }
}
}
```

Exercícios Complementares de Fixação:

1) Refazer os projetos feitos em aula.

2) 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.

3) (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 pagina 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

4) Refaça o exercício 2 da primeira mini apostila, 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

5) 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 mostra 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);

!!!Faça todos os exercícios uma vez com e outra vez sem servlets, o intuito é ver a diferença de se programar usando JSP sem e com servlets.!!!

12-JavaEE6 – Conhecendo o JSF – Java Server Faces.

Como vimos anteriormente, a versão JavaEE5, é baseada no uso de servlets e JSP. Para a versão do JavaEE6, é mais utilizado o conceito de JSF, ou Java Server faces. Elas 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.

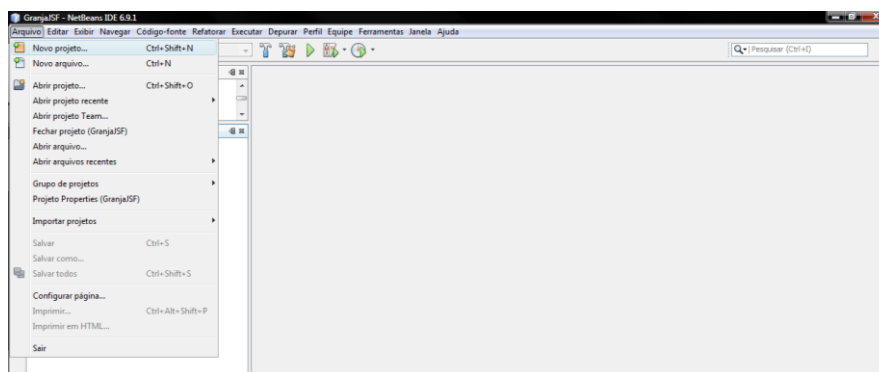
Vamos criar um exemplo baseado no CRUD em JSF feito no site “javasemcafe.blogspot.com” para entendermos como funciona o JSF.

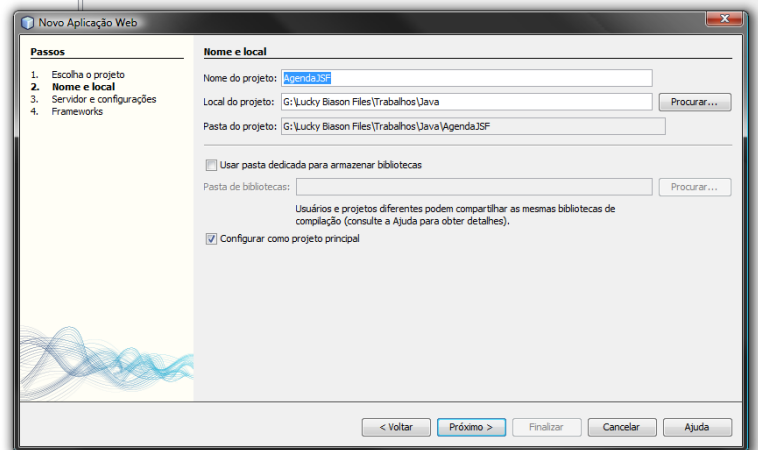
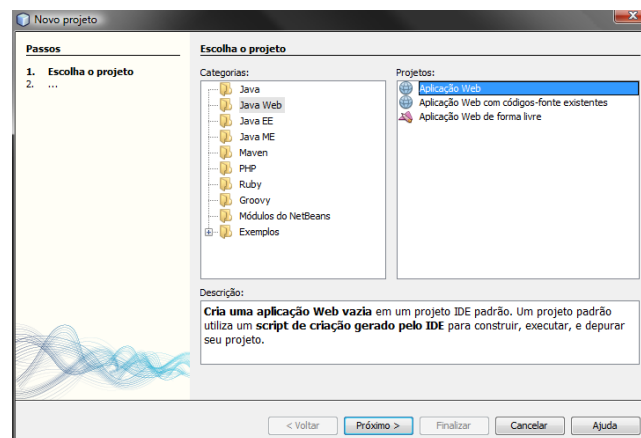
Inicialmente criaremos o banco de dados:

```
CREATE TABLE projetojsp.cliente (  
codigo INTEGER(11) NOT NULL AUTO_INCREMENT,  
nome VARCHAR(255) NOT NULL,  
telefone VARCHAR(30) ,  
PRIMARY KEY (codigo)  
) ENGINE = InnoDB;
```

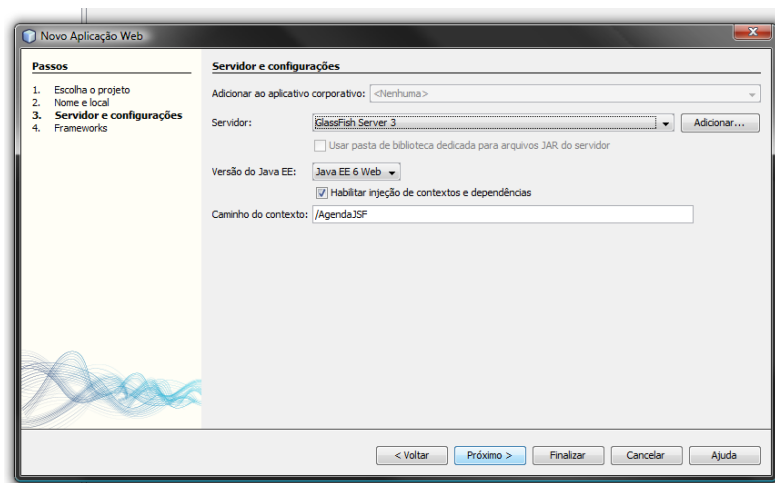
Criaremos uma mini Agenda de Contatos. Agora criaremos um novo projeto com nos moldes do jsf. (Atenção, faça EXATAMENTE como a “receita” a seguir).

Primeiro, vamos criar um novo projeto chamado AgendaJSF:

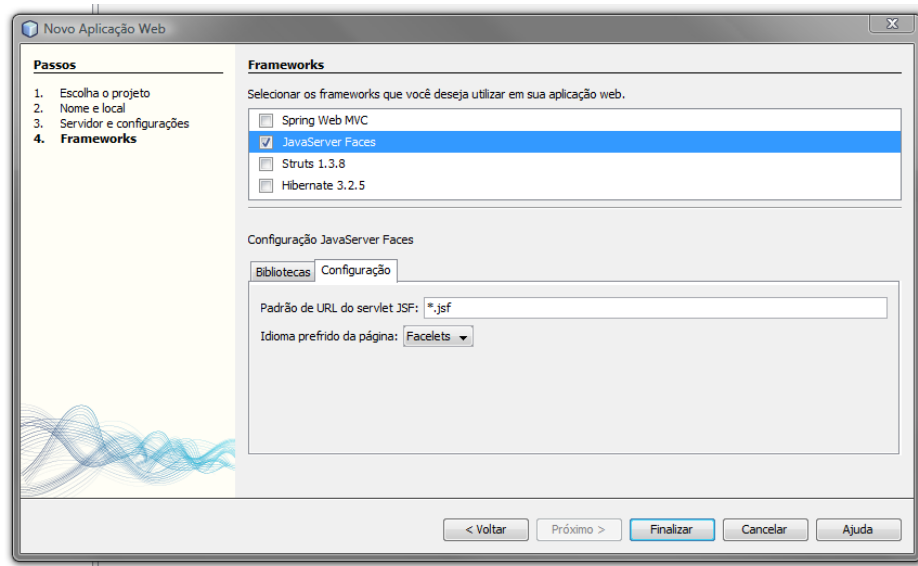




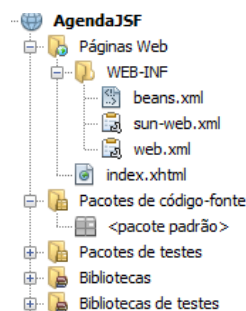
Na parte dos Servidores e Configurações escolha o GlassFish Server 3, que vem junto com o NetBeans 6.9.1 e a versão JavaEE6. Marque a caixa “Habilitar injeção de contextos e dependências”.



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.



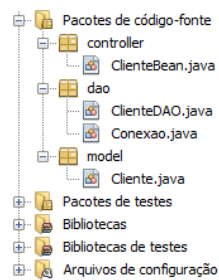
Finalize. A estrutura deve ficar assim:



Na página index.xhtml verificamos o uso na sintaxe do jsf.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
<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>
  Hello from Facelets
</h:body>
</html>
```

Vamos criar três pacotes e algumas classes de acordo com a figura e os códigos a seguir.



Pacote dao, classe Conexao e classe Dao:

```
package dao;

import java.sql.*;

public class Conexao {

    public static Connection getConn(){
        try{
            Class.forName("com.mysql.jdbc.Driver");
            return (Connection)
                DriverManager.getConnection(
                    "jdbc:mysql://localhost:3307/projetojsp", "root", "senha");
        } catch (ClassNotFoundException ex) {
            System.out.println(ex);
            return null;
        } catch (SQLException s) {
            System.out.println(s);
            return null;
        }
    }
}
```

```
package dao;

import java.sql.*;

public class Dao {

    public Connection getConnection(){
        return Conexao.getConn();
    }

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

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

    public ResultSet executeQuery(String query, Object ... params) throws SQLException{
        PreparedStatement ps = getStatement(query);
        for(int i = 0; i< params.length; i++){
            ps.setObject(i+1, params[i]);
        }
        return ps.executeQuery();
    }
}
```

```

public int executeCommand(String query, Object... params) throws SQLException{
    PreparedStatement ps = getStatement(query);
    for(int i = 0; i< params.length; i++){
        ps.setObject(i+1, params[i]);
    }
    int result = ps.executeUpdate();
    ps.close();
    return result;
}

```

Pacote model, a classe que representa a tabela cliente, Cliente.java.
 Pacote Dao, vamos criar a manipulação da tabela cliente.

```

package model;

public class Cliente {

    private int codigo;
    private String nome;
    private String telefone;

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public String getNome() {
        return nome;
    }

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

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }
}

package dao;

import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;
import model.Cliente;

public class ClienteDAO extends Dao{

    public boolean inserir(Cliente cliente) {
        try {
            executeCommand("INSERT INTO cliente (nome, telefone) VALUES (?,?)",
                cliente.getNome(),cliente.getTelefone());
            return true;
        } catch (Exception e) {return false;}
    }

    public boolean alterar(Cliente cliente) {
        try {
            executeCommand("UPDATE cliente SET nome = ?, telefone = ? WHERE codigo = ?",
                cliente.getNome(),cliente.getTelefone(),cliente.getCodigo());
            return true;
        } catch (Exception e) {return false;}
    }

    public boolean remover(Cliente cliente) {
        try {
            executeCommand("DELETE FROM cliente WHERE codigo = ?",cliente.getCodigo());
            return true;
        } catch (Exception e) {return false;}
    }

    public List<Cliente> listar() {
        List<Cliente> clientes = new ArrayList<Cliente>();
        try {
            ResultSet rs = executeQuery("SELECT * FROM cliente ORDER BY nome");
            while(rs.next()){
                Cliente cliente = new Cliente(rs.getInt("codigo"),rs.getString("nome"),rs.getString("telefone"));
                clientes.add(cliente);
            }
        } catch (Exception e) {}
        return clientes;
    }
}

```

```

        while (rs.next()) {
            cliente = new Cliente();
            cliente.setCodigo(rs.getInt("codigo"));
            cliente.setNome(rs.getString("nome"));
            cliente.setTelefone(rs.getString("telefone"));
            clientes.add(cliente);
        }
        return clientes;
    } catch (Exception e) {return null;}
}

```

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 vincada com a página xHTML. Essa classe é obrigatória para o JSF.

```

import dao.ClienteDAO;
import java.io.Serializable;
import java.util.List;
import javax.enterprise.context.SessionScoped;
import javax.faces.model.DataModel;
import javax.faces.model.ListDataModel;
import javax.inject.Named;
import model.Cliente;

@Named
@SessionScoped
public class ClienteBean implements Serializable {

    private ClienteDAO clienteDAO = new ClienteDAO();
    private Cliente cliente = new Cliente();
    private DataModel<Cliente> clientes;

    public void novo() {
        cliente = new Cliente();
    }
    public String inserir() {
        return (clienteDAO.inserir(cliente)) ? "clientes" : "falhou";
    }
    public void selecionar() {
        cliente = clientes.getRowData();
    }
    public String alterar() {
        return (clienteDAO.alterar(cliente)) ? "clientes" : "falhou";
    }
    public String remover() {
        return (clienteDAO.remover(cliente)) ? "clientes" : "falhou";
    }
    public Cliente getCliente() {
        return cliente;
    }
    public void setCliente(Cliente cliente) {
        this.cliente = cliente;
    }
    public DataModel<Cliente> getClientes() {
        List<Cliente> clienteList = new ClienteDAO().listar();
        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"`.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http
<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="clientes" value="Clientes"/>
    </h:form>
  </h:body>
</html>
```

Cientes.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 às 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 é visto na página `novo.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.
<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>Facelet Title</title>
  </h:head>
  <h:body>
    <h:form>
      <h:commandButton action="novo" actionListener="#{clienteBean.novo}" value="Novo"/>
      <h:dataTable value="#{clienteBean.clientes}" var="c">
        <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="Ações"/></f:facet>
          <h:commandButton action="alterar" actionListener="#{clienteBean.selecionar}" value="Alterar"/>
          <h:commandButton action="remover" actionListener="#{clienteBean.selecionar}" value="Remover"/>
        </h:column>
      </h:dataTable>
      <h:commandButton action="index" value="Voltar"/>
    </h:form>
  </h:body>
</html>
```

Novo.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1
<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:panelGrid columns="2">
        <h:outputText value="Nome"/>
        <h:inputText value="#{clienteBean.cliente.nome}"/>
        <h:outputText value="Telefone"/>
        <h:inputText value="#{clienteBean.cliente.telefone}"/>
        <h:commandButton action="#{clienteBean.inserir}" value="Inserir"/>
        <h:commandButton action="clientes" immediate="True" value="Cancelar"/>
      </h:panelGrid>
    </h:form>
  </h:body>
</html>
```

Alterar.xhtml, quase não muda em relação a página anterior.

```
<?xml version='1.0' encoding='UTF-8' ?>
<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>Facelet Title</title>
  </h:head>
  <h:body>
    <h:form>
      <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:commandButton action="#{clienteBean.alterar}" value="Alterar"/>
        <h:commandButton action="Clientes" immediate="True" value="Cancelar"/>
      </h:panelGrid>
    </h:form>
  </h:body>
</html>
```

Remover.xHTML

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/">
<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>Facelet Title</title>
  </h:head>
  <h:body>
    <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>
  </h:body>
</html>
```

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/">
<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:outputText value="Ocorreu um erro, tente novamente"/>
      <h:commandButton action="index" value="Tentar novamente"/>
    </h:form>
  </h:body>
</html>
```

Veja um exemplo de utilização:

Novo

Nome Telefone Ações

Nome:

Telefone:

→

Novo

Nome Telefone Ações

Lucas 12345

→

Exercícios Complementares de Fixação:

- 1) Refaça TODOS os exercícios feitos durante o curso.

Exercícios de Pesquisa

- 1) Explique sobre JSF e JavaEE6.
- 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).

13-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 incluir 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 ela 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 possui 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)

14-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>

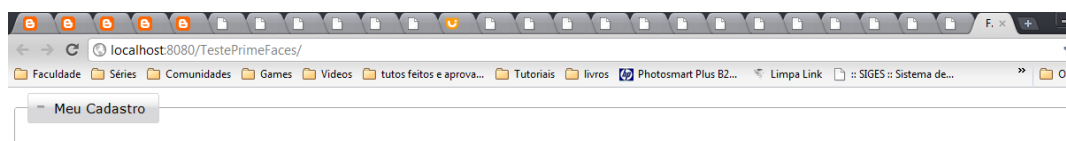
    </h:body>
  </f:view>
</html>
```

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>
```

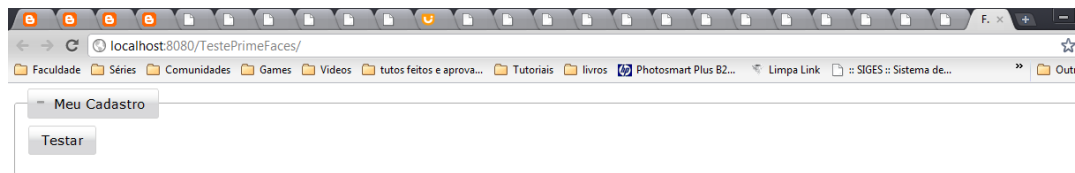
Legend é o título e toggleable indica se o usuário poderá esconder o conteúdo dentro do fieldset ou não. Veja o resultado:



O segundo componente é muito importante, trata-se de um botão commandButton.

```
<p:commandButton value="Testar" actionListener="#{testeBean.testar}" />
```

Veja o resultado:



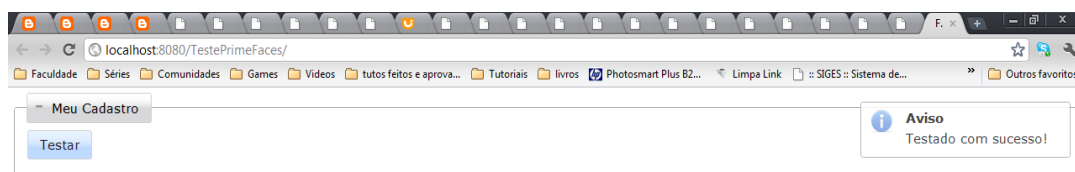
O atributo `actionListener` indica ao botão qual método/comando no beans ele deverá fazer quando pressionado. Você pode utilizar `action` para se referir as páginas jsf. Mas não esqueça de sempre utilizar o atributo: `ajax="false"` quando o fizer, por exemplo:

```
<p:commandButton value="Painéis" action="testepainéis.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:

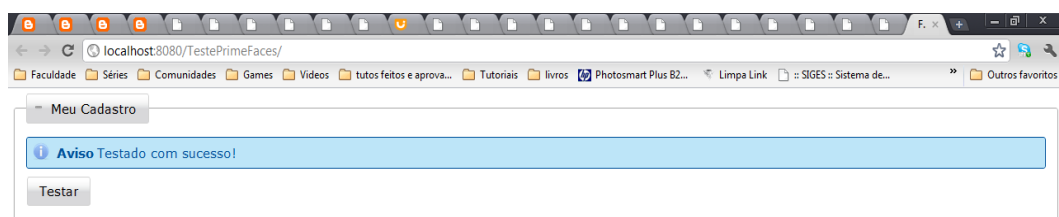
```
<p:growl id="avisos" showDetail="true" life="3000" />
```

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:

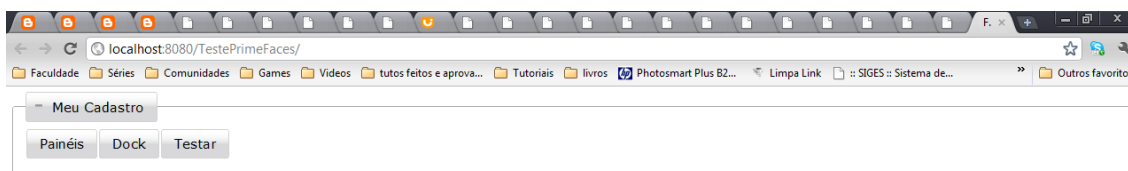
```
<?xml version='1.0' encoding='UTF-8' ?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.prime.com.tr/ui"
      xmlns:f="http://java.sun.com/jsf/core">
  <f:view contentType="text/html">
    <h:head>
      <title>Facelet Title</title>
    </h:head>
    <h:body>
      <p:fieldset legend="Meu Cadastro" toggleable="true">
        <h:form>

          <p:growl id="avisos" showDetail="true" life="3000" />
          <p:messages id="mensagens" showDetail="true" />

          <p:commandButton value="Painéis" action="testepainels.jsf" ajax="false" />
          <p:commandButton value="Dock" action="testedock.jsf" ajax="false" />

          <p:commandButton value="Testar" actionListener="#{tableBean.testar}"
                           update="avisos,mensagens" />

        </h:form>
      </p:fieldset>
    </h:body>
  </f:view>
</html>
```



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:" />
```

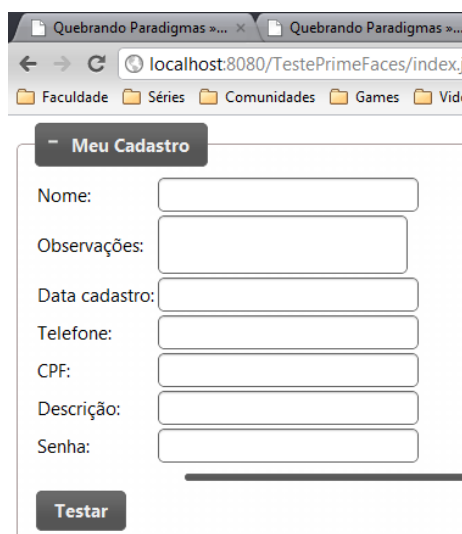
```

        <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                                mask="(999)9999-9999"
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>

```



The screenshot shows a web browser window with two tabs titled 'Quebrando Paradigmas'. The address bar shows 'localhost:8080/TestePrimeFaces/index.jspx'. Below the browser window, a registration form titled 'Meu Cadastro' is displayed. The form contains the following fields: 'Nome:', 'Observações:', 'Data cadastro:', 'Telefone:', 'CPF:', 'Descrição:', and 'Senha:'. Each field has a corresponding input box. At the bottom of the form, there is a 'Testar' button.

```
<h:outputText value="Data cadastro:" />
<p:calendar value="#{tableBean.dataCadastro}" />
```



Data cadastro:

Telefone:

CPF:

Descrição:

Senha:

Testar

```
<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 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>
```

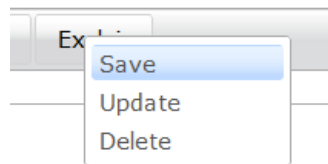
O componente contextMenu serve para criar menus que aparecem ao clicar no botão direito do mouse.

```
<h:form>
  <p:contextMenu>
    <p:menuitem value="Save" actionListener="..." />
    <p:menuitem value="Update" actionListener="..." />
    <p:menuitem value="Delete" actionListener="..." />
  </p:contextMenu>
</h:form>
```

```

</p:contextMenu>
</h:form>

```

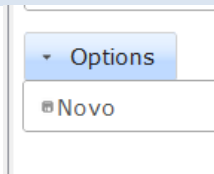


MenuBar é um componente que cria um botão-menu, ao ser clicado, ele estende um menu criado. No exemplo a seguir, percebe-se que podemos colocar uma pequena imagem no menu.

```

<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.

```

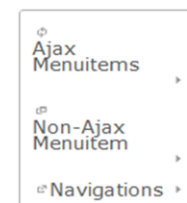
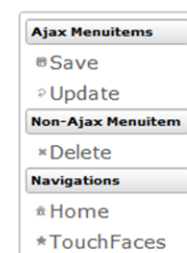
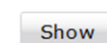
<h:form>
    <p:growl id="messages"/>
    <h3>Tiered Menu</h3>
    <p:menu type="tiered">
        <p:submenu label="Ajax Menuitems" icon="ui-icon ui-icon-
refresh">
            <p:menuitem value="Save" actionListener="#{buttonBean.save}"
update="messages" icon="ui-icon ui-icon-disk" />
            <p:menuitem value="Update"
actionListener="#{buttonBean.update}" update="messages" icon="ui-icon ui-
icon-arrowrefresh-1-w" />
        </p:submenu>
    </p:menu>
</h:form>

```

```

        <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">
        <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>

```

Tiered Menu**Sliding Menu****Regular Menu****Dynamic Position**

```

        <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="#"/>

```

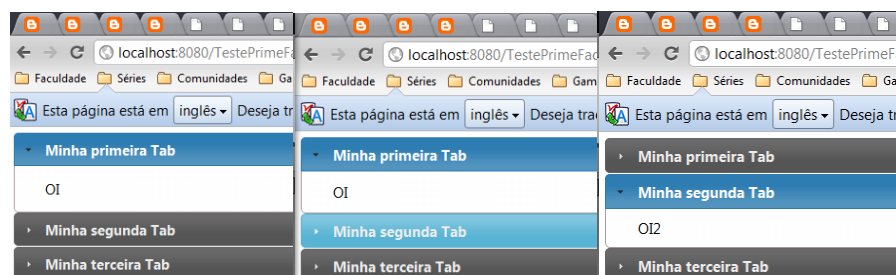
```
<p:menuitem value="RSS" icon="/images/dock/rss.png" url="#" />
<p:menuitem value="History" icon="/images/dock/history.png"
url="#" />
</p:dock>
```

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>
  </h:panelGrid>
</p:fieldset>
```

```
</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:dataTable>
</h:form>
```

```

</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:

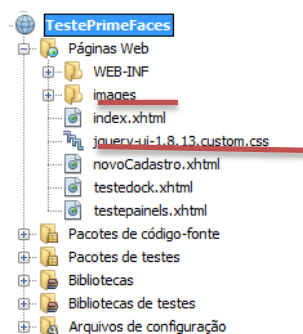
<div> <div>1 2 3 4 5 6 7 8</div> </div>	
Nome dos animais	Nome dos animais 2
Girafa	Girafa
Pato	Pato
Leopardo	Leopardo
Elefante	Elefante
<div> <div>1 2 3 4 5 6 7 8</div> </div>	

Temas

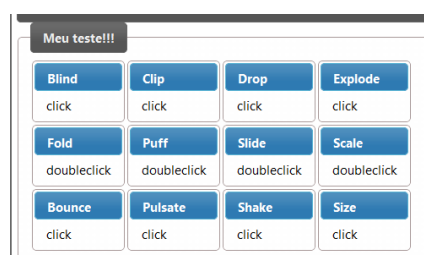
O primeFaces tem diversos temas prontos, e a possibilidade de personalizar e criar novos temas. O site <http://iqueryui.com/themeroller/> oferece todas as ferramentas necessárias para criação de temas para primeFaces basta utilizar o menu do lado esquerdo.



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.

Anexo 1 – Hibernate e JPA (no Eclipse)

Hibernate

É um framework de mapeamento objeto relacional mais utilizado no universo Java, na versão do Hibernate 2, era muito trabalhoso criar seus mapeamentos, um XML enorme, mas com a vinda do Java 5 e as anotações, tudo foi simplificado, na versão do Hibernate 3, já utiliza estes recursos, e não é compatível com sua versão anterior, assim se tornando mais poderoso, pois, mantendo compatibilidade, o framework não poderia crescer muito sem "quebrar" elementos de sua versão anterior.

Basicamente permite fazer a persistência de objetos em banco de dados relacionais de modo transparente e para qualquer aplicação Java.

Sendo assim, ao invés de ter uma perda de tempo em escrever e mesclar códigos SQL em seu código e mapear resultados de suas consultas para objetos, o desenvolvedor só precisa se preocupar com seus objetos.

Com o Hibernate podemos:

- Criar a tabela de sua base de dados;
- Criar o objeto em que seu estado irá ser persistido;
- Relacionar por anotações os relacionamentos e propriedades do objeto aos campos de sua tabela;
- Criar a classe DAO que persistirá seu objeto
- Criar um simples arquivo para que o Hibernate conecte a seu banco de dados.

O JPA (Java Persistence API (Application Programming Interface))

O JPA é um framework mantido pelo JCP, sendo um de seus criadores o mesmo da especificação do Hibernate, este framework é utilizado na camada de persistência, é uma “casca”, logo abaixo deste framework pode se encontrar seu framework de persistência preferido no nosso caso o Hibernate, mas poderia ser o Top Link por exemplo.

A figura 2 representa as camadas, sendo o JPA, logo abaixo o Hibernate, encapsulado o JDBC e finalmente a Base de Dados.

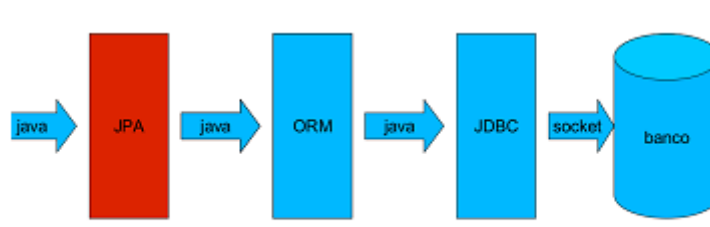


Figura 2 – www.coelum.com.br

JPA nasceu no JCP para ser criado uma especificação de persistência já que o JDP não foi um projeto de grande sucesso e os Entity Beans do EJB eram extremamente difíceis de lidar, foi especificado na JSR 220, standalone ou gerenciado pelo servidor, a não ser que o desenvolvedor opte o JPA não necessita de XML para criar os mapeamentos.

Gerenciador de estados:

Transiente, no qual o objeto não precisa ser persistido;

Detached, no qual o objeto deverá ter o estado de merged para persistir;

Managed, no qual o objeto sempre será persistido.

Bom utilizaremos na apostila os dois frameworks juntos já que JPA é apenas uma especificação e terá que ter uma implementação no caso imagine o JPA como uma casca, onde por baixo rode o Hibernate, OracleTypes, ou o framework ORM que desejar.

Bom para início vamos configurar nosso projeto no NetBeans, então crie um novo projeto, New-> Project → Java Project.

Nesse momento começaremos a utilizar a camada de persistência que incluirá todos nossos projetos web, lembre-se existe a Camada de Persistência (DAO que pode ser JDBC puro ou utilizando o Hibernate sozinho ou com JPA), a parte Web com os frameworks MVC são independente desta camada, não é necessário ter um banco de dados para um sistema Web, mas nunca ou quase nunca fazemos um sistema que armazenará dados em Listas (Arrays) e sim dentro de um repositório, no nosso caso o banco de dados. Vamos configurar nossos JavaBeans que serão as entidades.

```
public class Cliente {  
    private long id;  
    private String nome;  
    private String cpf;  
    private String endereco;  
    private String telefone;  
    private String email;  
  
    //Métodos Getters e Setters  
  
}
```

Opa perai já criei esta classe, o que ela tem de entidade? Veja bem, esta classe é o Java Bean, porém vamos fazer as anotações dizendo que é uma entidade:

Então ela ficará assim:

```
@Entity  
public class Cliente {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private long id;  
    private String nome;  
    private String cpf;  
    private String endereco;
```

```
private String telefone;  
private String email;  
//Métodos Getters e Setters  
}
```

@Entity: aqui estou dizendo, olha sou uma entidade do Banco, represento uma tabela.

@Id: o id é OBRIGATÓRIO quando você diz que uma classe é uma Entity.

@GeneratedValue(strategy = GenerationType.IDENTITY): esta anotação é para dizer que o campo Id é auto-numeração do seu Banco de Dados, também podemos colocar dentro desta anotação opções para não deixar um campo nulo como nullable=false e também dizer o tamanho do campo, length=50, no final ficaria:

```
@GeneratedValue(strategy = GenerationType.IDENTITY, nullable=false, length=50).
```

Até aqui pura JPA. Mas onde está o Hibernate?

Bom vamos configurar as propriedades do Banco em Hibernate, poderíamos fazê-lo em JPA, mas utilizaremos o Hibernate, também o veremos quando utilizarmos HQL.

hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate  
Configuration DTD 3.0//EN"  
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">  
<hibernate-configuration>  
  <session-factory>  
    <property name="hibernate.dialect">  
      org.hibernate.dialect.MySQLDialect  
    </property>  
    <property name="hibernate.connection.url">  
      jdbc:mysql://localhost/loja  
    </property>  
    <property name="hibernate.connection.driver_class">
```



```
com.mysql.jdbc.Driver
</property>
<property name="hibernate.connection.username">root</property>
<property name="hibernate.connection.password">root</property>
<property name="hibernate.show_sql">true</property>
<property name="hibernate.format_sql">true</property>
<!-- usar C3P0 Pool de conexões-->
<property name="hibernate.c3p0.min_size">1</property>
<property name="hibernate.c3p0.max_size">10</property>
<property name="hibernate.c3p0.maxIdleTime">200</property>
<property name="hibernate.c3p0.timeout">180</property>
<property name="hibernate.c3p0.idle_test_period">100</property>
<property name="hibernate.c3p0.max-statements">50</property>
<!-- Cache 2 nível-->
<property name="hibernate.cache.provider_class">
org.hibernate.cache.EhCacheProvider
</property>
<mapping class="br.com.loja.bean.Cliente"/>
</session-factory>
</hibernate-configuration>
```

Basicamente ninguém cria do zero este arquivo, normalmente todo projeto tem esse padrão, como vemos, o NetBeans cria tudo isso automaticamente.

Iremos detalhar um pouco do que cada linha faz para ficar bem claro.

A propriedade `Dialect` serve para informar qual é o banco de dados;

Connection URL informa o caminho para o banco;

Username e password informam qual é o usuário e senha para acesso ao banco. Exatamente como fizemos na classe com o JDBC.

As propriedades do C3P0 servem para meu Pool de conexões, ou seja, para delimitar alguns limites estou usando o C3P0 que já está nos JARs do Hibernate.

Estou usando Cache level 2 para manter alguns objetos em cache (não era preciso, somente é necessário em casos específicos), é algo bem legal que o Hibernate possui.

E por fim, mapeamos as classes Java Beans que serão entidades no banco de dados com a propriedade mapping.

Possíveis valores de dialetos:

```
DB2 - org.hibernate.dialect.DB2Dialect
HypersonicSQL - org.hibernate.dialect.HSQLDialect
Informix - org.hibernate.dialect.InformixDialect
Ingres - org.hibernate.dialect.IngresDialect
Interbase - org.hibernate.dialect.InterbaseDialect
Pointbase - org.hibernate.dialect.PointbaseDialect
PostgreSQL - org.hibernate.dialect.PostgreSQLDialect
Mckoi SQL - org.hibernate.dialect.MckoiDialect
Microsoft SQL Server - org.hibernate.dialect.SQLServerDialect
MySQL - org.hibernate.dialect.MySQLDialect
Oracle (any version) - org.hibernate.dialect.OracleDialect
Oracle 9 - org.hibernate.dialect.Oracle9Dialect
Progress - org.hibernate.dialect.ProgressDialect
FrontBase - org.hibernate.dialect.FrontbaseDialect
SAP DB - org.hibernate.dialect.SAPDBDialect
Sybase - org.hibernate.dialect.SybaseDialect
Sybase Anywhere - org.hibernate.dialect.SybaseAnywhereDialect
```

Hibernate Conceitos

Session (org.hibernate.Session)

O objeto Session é aquele que possibilita a comunicação entre a aplicação e persistência, através de uma conexão JDBC. É um objeto leve de ser criado, não deve ter tempo de vida por toda a aplicação e não é threadsafe. Um objeto Session possui um cache local de objetos recuperados na sessão. Com ele é possível criar, remover, atualizar e recuperar objetos persistentes.

SessionFactory (org.hibernate.SessionFactory)

O objeto `SessionFactory` é aquele que mantém o mapeamento objeto relacional em memória. Permite a criação de objetos `Session`, a partir dos quais os dados são acessados, também denominado como fábrica de objetos `Sessions`. Um objeto `SessionFactory` é `threadsafe`, porém deve existir apenas uma instância dele na aplicação, pois é um objeto muito pesado para ser criado várias vezes.

Configuration (`org.hibernate.Configuration`)

Um objeto `Configuration` é utilizado para realizar as configurações de inicialização do Hibernate. Com ele, define-se diversas configurações do

Hibernate, como por exemplo: o driver do banco de dados a ser utilizado, o dialeto, o usuário e senha do banco, entre outras. É a partir de uma instância desse objeto que se indica como os mapeamentos entre classes e tabelas de banco de dados devem ser feitos.

Transaction (`org.hibernate.Transaction`)

A interface `Transaction` é utilizada para representar uma unidade indivisível de uma operação de manipulação de dados. O uso dessa interface em aplicações que usam Hibernate é opcional. Essa interface abstrai a aplicação dos detalhes das transações JDBC, JTA ou CORBA.

Interfaces Criteria e Query

As interfaces `Criteria` e `Query` são utilizadas para realizar consultas ao banco de dados.

A seguir, vamos criar a Classe `HibernateUtil` que nos auxiliará para a configuração do Hibernate e será nossa fábrica de conexão.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateUtil {
    private static ThreadLocal<Session> sessions;
    private static SessionFactory sessionFactory;
```

```
static {
    sessionFactory = new AnnotationConfiguration().configure()
        .buildSessionFactory();
    sessions= new ThreadLocal<Session>();
}
public static Session openSession() {
    return sessionFactory.openSession();
}
public static Session getCurrentSession() {
    if (sessions.get() == null) {
        sessions.set(openSession());
    }
    return sessions.get();
}
public static void closeCurrentSession() {
    getCurrentSession().close();
    sessions.set(null);
}
}
```

O bloco estático tem a função de configurar o Hibernate e pegar a SessionFactory, este bloco é executado automaticamente no Class Loader e somente nesse momento, o método openSession devolve uma Session, conseguida através da SessionFactory.

Relacionamentos

É possível fazer relacionamentos no Hibernate dentro de nossas entidades que serão geradas depois nas tabelas, basta utilizar as anotações:

@ManytoOne

@ManyToMany

@OneToOne

Também podemos utilizar anotações para quem precisa utilizar os famosos Joins Table com a anotação

@JoinColumn (name="chave_estrangeira").

Quando declaramos atributos como Date ou Calendar devemos utilizar a anotação `@Temporal(TemporalType.DATE)`

Existem muitas anotações e podem ser encontradas nas especificações de EJB3 e no site do Hibernate.

Criando nossa classe que Gera o Banco de Dados:

```
public class GeraBanco {  
    public static void main(String[] args) {  
        Configuration configuration = new AnnotationConfiguration();  
        configuration.configure();  
        SchemaExport schemaExport = new SchemaExport(configuration);  
        SchemaExport.create(true, true);  
    }  
}
```

O SchemaExport vai tratar de gerar suas tabelas baseados na configuração feita acima.

Criando nossa classe DAO:

```
public class ClienteDAO {  
    private Session session;  
    public ClienteDAO() {  
        this.session = HibernateUtil.getCurrentSession();  
    }  
    /**  
    * salva novos dados no sistema  
    */  
    public void salvaDados(Cliente cliente) {  
        Transaction transaction = session.beginTransaction();  
        session.saveOrUpdate(cliente);  
        transaction.commit();  
    }  
    /**  
    * exclui os dados do sistema  
    */  
}
```

```
public void excluiDados(Cliente cliente) {  
    Transaction transaction = session.beginTransaction();  
    session.delete(cliente);  
    transaction.commit();  
}  
/**  
 * busca os dados  
 */  
public Cliente selecionaDados(Cliente cliente) {  
    return (Cliente) session.load(Cliente.class, cliente.getId());  
}  
}
```

Observações:

- save(Object) Inclui um objeto em uma tabela do banco de dados.
- saveOrUpdate(Object) Inclui um objeto na tabela caso ele ainda não exista ou atualiza o objeto caso ele já exista.
- delete(Object) Apaga um objeto da tabela no banco de dados.
- get(Class, Serializable id) Retorna um objeto a partir de sua chave primária. A classe do objeto é passada como primeiro argumento e o seu id como segundo argumento.

Criamos um construtor que receberá como parâmetro a sessão criada anteriormente. Como dito anteriormente o DAO que irá ter nossos métodos CRUD.

Vamos testar todo o procedimento acima, crie a classe InsereDados:

```
public class InsereDados {  
    public static void main(String[] args) {  
        ClienteDAO dao = new ClienteDAO();  
        Cliente cliente = new Cliente();  
        cliente.setNome("Joao");  
        cliente.setCpf("123.222.636-98");  
        cliente.setEndereco("Rua Adm, 130");  
    }  
}
```

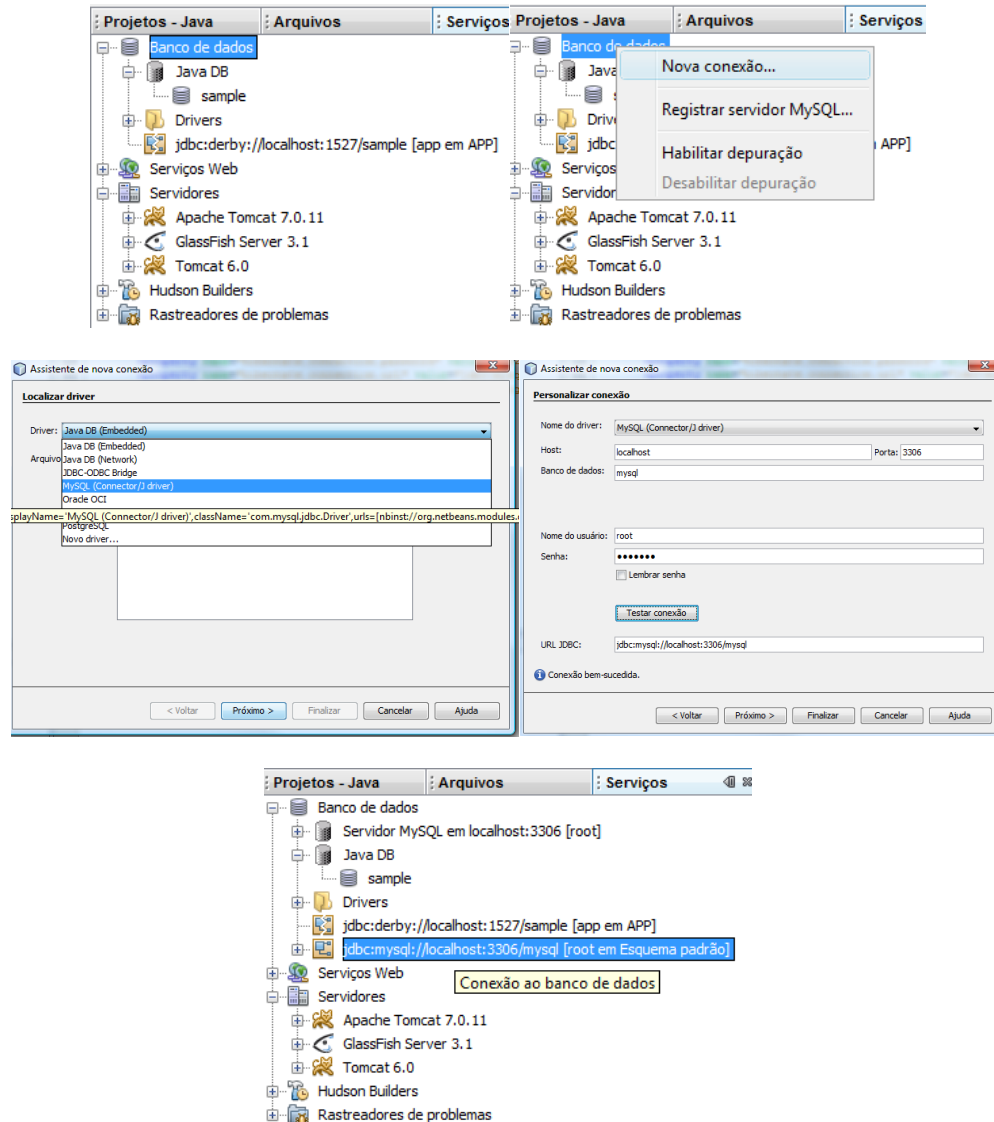
```
        cliente.setTelefone("4221-99.85");  
        cliente.setEmail("email@gmail.com");  
        dao.salvaDados(cliente);  
        HibernateUtil.closeCurrentSession();  
    }  
}
```

Entre no MySQL e digite o seguinte Select:

```
Select * from cliente;
```

Veja se ele inseriu os dados.

Anexo 2 – Configurando o acesso ao MYSQL no NetBeans

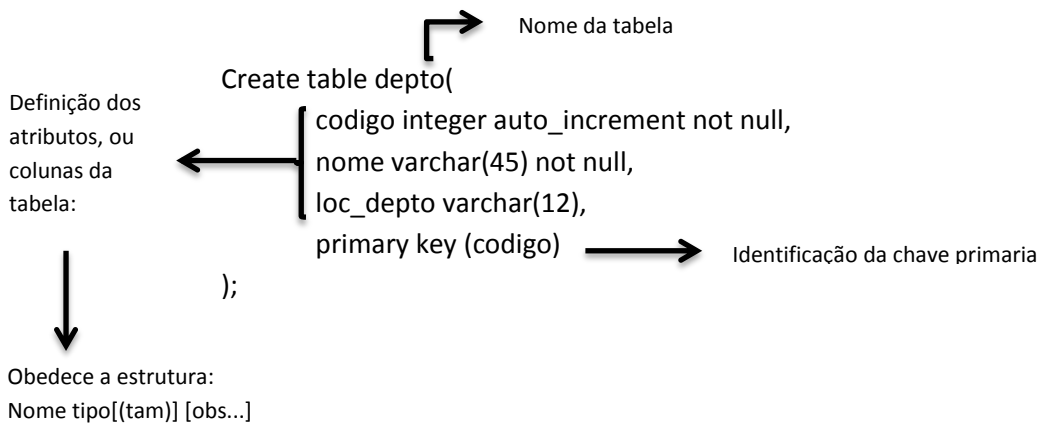


Anexo3 – Noções de SQL

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.