

CAPÍTULO 7

Introdução ao JSF e Primefaces

"Eu não temo computadores, eu temo é a falta deles"

— Isaac Asimov

Durante muitos anos, os usuários se habituaram com aplicações Desktop. Este tipo de aplicação é instalada no computador local e acessa diretamente um banco de dados ou gerenciador de arquivos. As tecnologias típicas para criar uma aplicação Desktop são Delphi, VB (Visual Basic) ou, no mundo Java, Swing.

Para o desenvolvedor, a aplicação Desktop é construída com uma série de componentes que a plataforma de desenvolvimento oferece para cada sistema operacional. Esses componentes ricos e muitas vezes sofisticados estão associados a eventos ou procedimentos que executam lógicas de negócio.

Problemas de validação de dados são indicados na própria tela sem que qualquer informação do formulário seja perdida. De uma forma natural, esses componentes lembram-se dos dados do usuário, inclusive entre telas e ações diferentes.

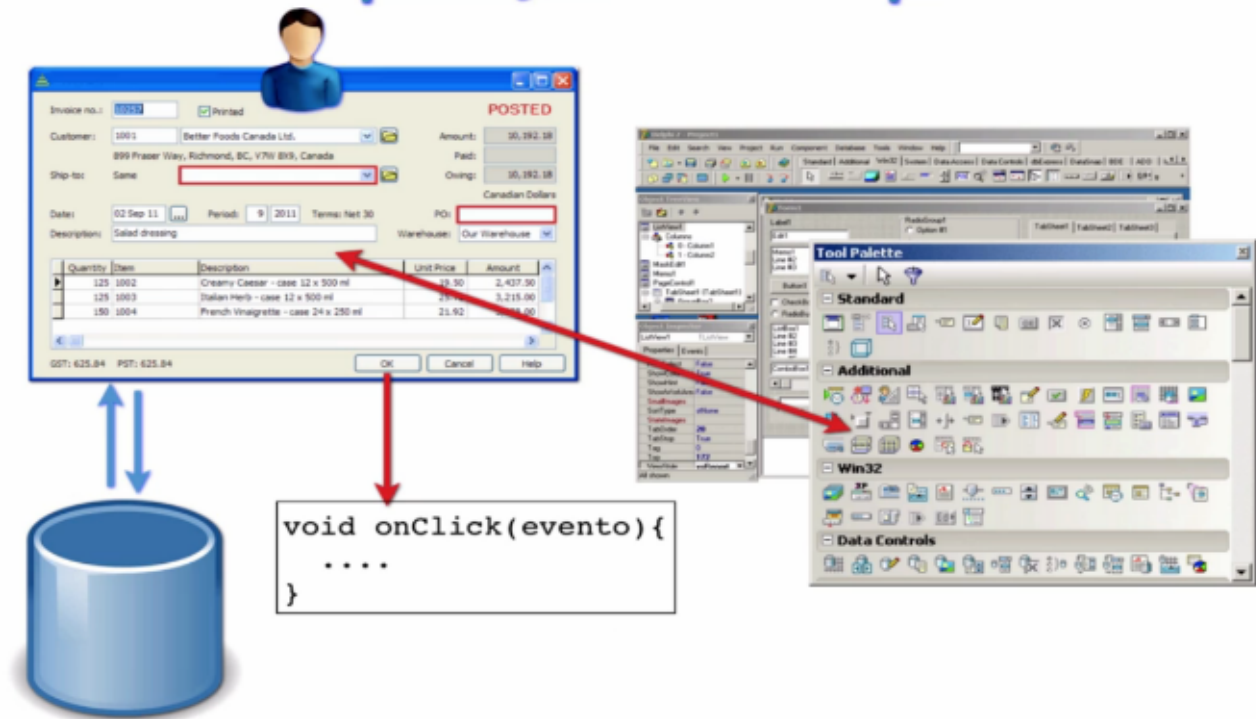
Nesse tipo de desenvolvimento são utilizados diversos **componentes ricos**, como por exemplo, calendários, menus diversos ou componentes *drag and drop* (arrastar e soltar). Eles ficam associados a eventos, ou ações, e guardam automaticamente seu estado, já que mantêm os valores digitados pelo usuário.

Aplicação Desktop?



Esses componentes não estão, contudo, associados exclusivamente ao desenvolvimento de aplicações Desktop. Podemos criar a mesma sensação confortável para o cliente em uma aplicação web, também usando componentes ricos e reaproveitáveis.

Aplicação Desktop?



7.1 – DESENVOLVIMENTO DESKTOP OU WEB?

Existem algumas desvantagens no desenvolvimento desktop. Como cada usuário tem uma cópia integral da aplicação, qualquer alteração precisaria ser propagada para todas as outras máquinas. Estamos usando um *cliente gordo*, isto é, com muita responsabilidade no lado do cliente.

Note que, aqui, estamos chamando de **cliente** a aplicação que está rodando na máquina do usuário.

Para piorar, as regras de negócio rodam no computador do usuário. Isso faz com que seja muito mais difícil depurar a aplicação, já que não costumamos ter acesso tão fácil à máquina onde a aplicação está instalada. Em geral, enfrentamos **problemas de manutenção e gerenciabilidade**.

O desenvolvimento Web e o protocolo HTTP

Para resolver problemas como esse, surgiram as aplicações baseadas na web. Nessa abordagem há um servidor central onde a aplicação é executada e processada e todos os usuários podem acessá-la através de um cliente simples e do protocolo HTTP.

Um navegador web, como Firefox ou Chrome, que fará o papel da aplicação

cliente, interpretando HTML, CSS e JavaScript -- que são as tecnologias que ele entende.

Enquanto o usuário usa o sistema, o navegador envia requisições (*requests*) para o lado do servidor (*server side*), que responde para o computador do cliente (*client side*). Em nenhum momento a aplicação está salva no cliente: todas as regras da aplicação estão no lado do servidor. Por isso, essa abordagem também foi chamada de **cliente magro** (*thin client*).



Isso facilita bastante a manutenção e a gerenciabilidade, pois temos um lugar central e acessível onde a aplicação é executada. Contudo, note que será preciso conhecer HTML, CSS e JavaScript, para fazer a interface com o usuário, e o protocolo **HTTP** para entender a comunicação pela web. E, mais importante ainda, não há mais eventos, mas sim um modelo bem diferente **orientado a requisições e respostas**. Toda essa base precisará ser conhecida pelo desenvolvedor.

Comparando as duas abordagens, podemos ver vantagens e desvantagens em ambas. No lado da aplicação puramente Desktop, temos um estilo de desenvolvimento orientado a eventos, usando componentes ricos, porém com problemas de manutenção e gerenciamento. Do outro lado, as aplicações web são mais fáceis de gerenciar e manter, mas precisamos lidar com HTML, conhecer o

protocolo HTTP e seguir o modelo requisição/resposta.

Mesclando desenvolvimento Desktop e Web

Em vez de desenvolver puramente para desktop, é uma tendência mesclar os dois estilos, aproveitando as vantagens de cada um. Seria um desenvolvimento Desktop para a web, tanto central quanto com componentes ricos, aproveitando o melhor dos dois mundos e abstraindo o protocolo de comunicação. Essa é justamente a ideia dos **frameworks web baseados em componentes**.

No mundo Java há algumas opções como **JavaServer Faces (JSF)**, Apache Wicket, Vaadin, Tapestry ou GWT da Google. Todos eles são *frameworks* web baseados em componentes.

7.2 – CARACTERÍSTICAS DO JSF

JSF é uma tecnologia que nos permite criar aplicações Java para Web utilizando componentes visuais pré-prontos, de forma que o desenvolvedor não se preocupe com Javascript e HTML. Basta adicionarmos os componentes (calendários, tabelas, formulários) e eles serão renderizados e exibidos em formato html.

Guarda o estado dos componentes

Além disso o estado dos componentes é sempre guardado automaticamente (como veremos mais à frente), criando a característica Stateful. Isso nos permite, por exemplo, criar formulários de várias páginas e navegar nos vários passos dele com o estado das telas sendo mantidos.

Separa as camadas

Outra característica marcante na arquitetura do JSF é a separação que fazemos entre as camadas de apresentação e de aplicação. Pensando no modelo MVC, o JSF possui uma camada de visualização bem separada do conjunto de classes de modelo.

Especificação: várias implementações

O JSF ainda tem a vantagem de ser uma especificação do Java EE, isto é, todo servidor de aplicações Java tem que vir com uma implementação dela e há diversas outras disponíveis.

A implementação mais famosa do JSF e também a implementação de referência, é a Oracle Mojarra disponível em <http://javaserverfaces.java.net/>. Outra implementação famosa é a MyFaces da *Apache Software Foundation* em <http://myfaces.apache.org/>.



Primeiros passos com JSF

Nosso projeto utilizará a implementação Mojarra do JSF. Ela já define o modelo de desenvolvimento e oferece alguns componentes bem básicos. Nada além de inputs, botões e ComboBoxes simples.

The screenshot shows a web form with several JSF components: a text input field containing "JSF", a password input field with five dots, a text area containing "JSF é component-based.", a dropdown menu currently showing "JSF", a group of radio buttons for selecting a framework (JSF is selected), a list box containing "JSF", "Tapestry", "vaadin", "Wicket", and "GWT", and two buttons labeled "salva".

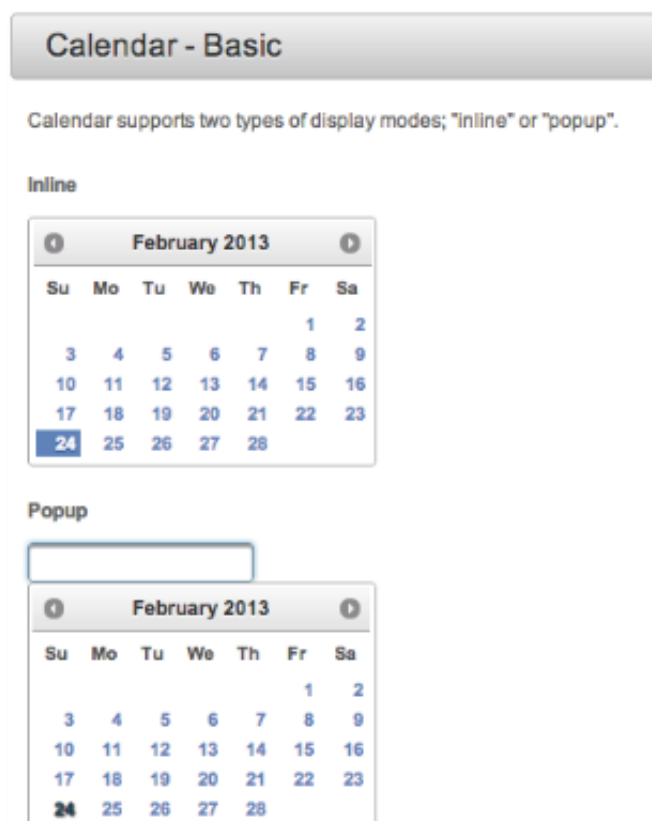
Não há componentes sofisticados dentro da especificação e isso é proposital: uma especificação tem que ser estável e as possibilidades das interfaces com o usuário crescem muito rapidamente. A especificação trata do que é fundamental, mas outros projetos suprem o que falta.

Para atender a demanda dos desenvolvedores por componentes mais sofisticados, há várias extensões do JSF que seguem o mesmo ciclo e modelo da especificação. Exemplos dessas bibliotecas são **PrimeFaces**, **RichFaces** e **IceFaces**. Todas elas definem componentes JSF que vão muito além da especificação.



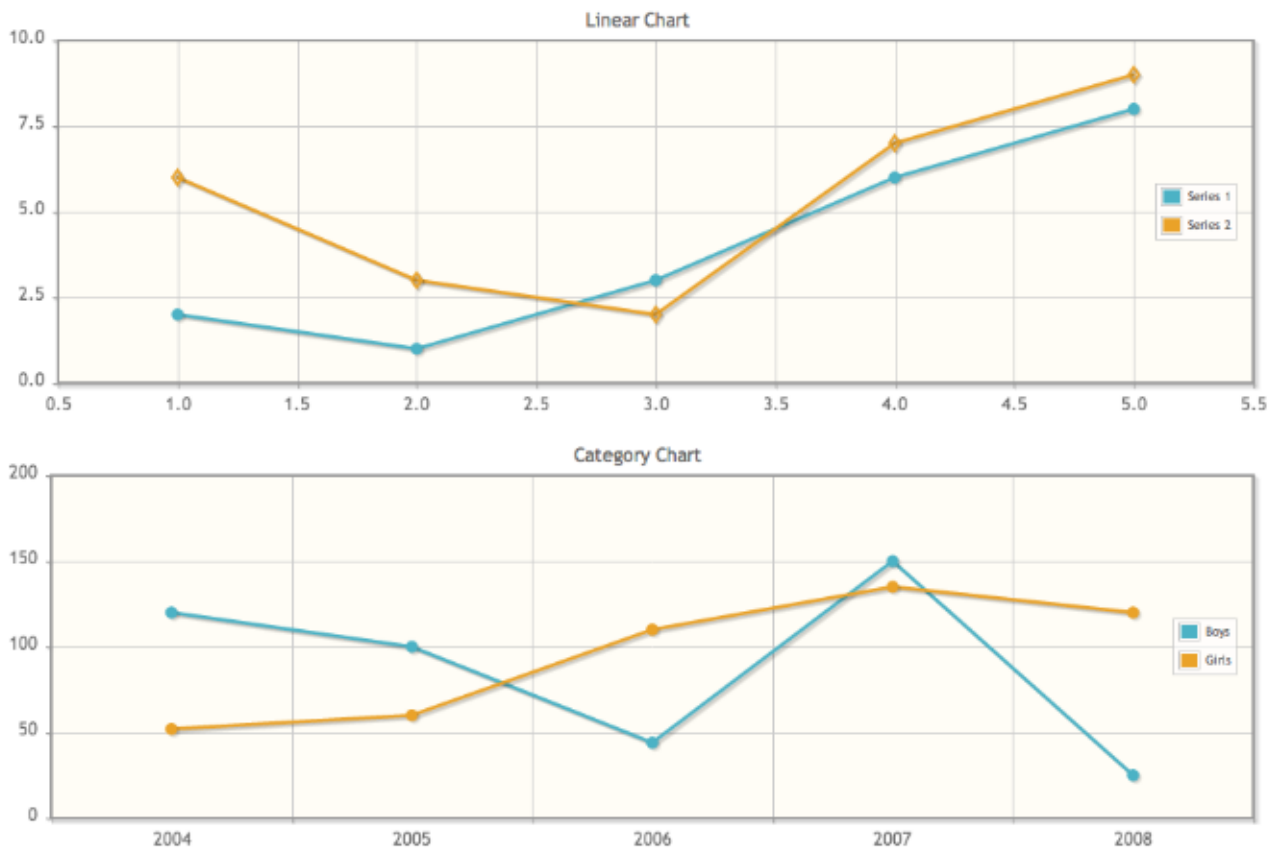
Cada biblioteca oferece *ShowCases* na web para mostrar seus componentes e suas funcionalidades. Você pode ver o *showcase* do **PrimeFaces** no endereço <http://www.primefaces.org>.

Na sua *demo online*, podemos ver uma lista de componentes disponíveis, como inputs, painéis, botões diversos, menus, gráficos e componentes *drag & drop*, que vão muito além das especificações, ainda mantendo a facilidade de uso:



Charts - Line

LineChart is created with a CartesianChartModel.

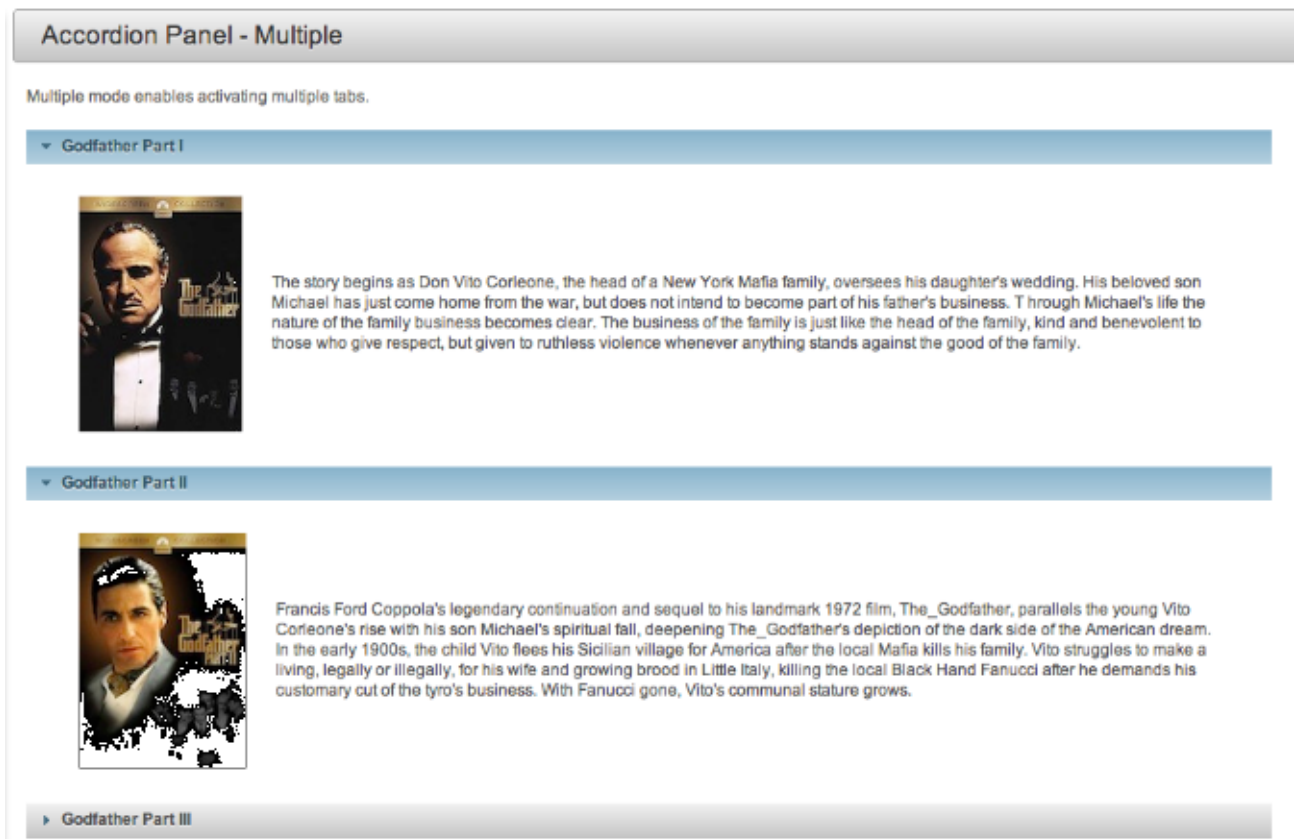


Menubar

Menubar brings the desktop application menubars to JSF. Using menuitems, it is very easy to execute ajax, non-ajax and navigations.

Default Menubar





Para a definição da interface do projeto *Argentum* usaremos **Oracle Mojarra** com **PrimeFaces**, uma combinação muito comum no mercado.

Preparação do ambiente

Nossa aplicação *Argentum* precisa de uma interface web. Para isso vamos preparar uma aplicação web comum que roda dentro de um *Servlet Container*. Qualquer implementação de servlet container seria válida e, no curso, usaremos o *Apache Tomcat 7*. Uma outra boa opção seria o *Jetty*.

Configuração do controlador do JSF

O JSF segue o padrão arquitetural MVC (*Model-View-Controller*) e faz o papel do *Controller* da aplicação. Para começar a usá-lo, é preciso configurar a servlet do JSF no `web.xml` da aplicação. Esse Servlet é responsável por receber as requisições e delegá-las ao JSF. Para configurá-lo basta adicionar as seguintes configurações no `web.xml`:

```
<servlet>
  <servlet-name>FacesServlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>FacesServlet</servlet-name>
  <url-pattern>*.xhtml</url-pattern>
```

</servlet-mapping>

Ao usar o Eclipse com suporte a JSF 2 essa configuração no web.xml já é feita automaticamente durante a criação de um projeto.

Faces-config: o arquivo de configuração do mundo JSF

Além disso, há um segundo XML que é o arquivo de configuração relacionado com o mundo JSF, o `faces-config.xml`.

Como o JSF na versão dois encoraja o uso de anotações em vez de configurações no XML, este arquivo torna-se pouco usado. Ele era muito mais importante na primeira versão do JSF. Neste treinamento, deixaremos ele vazio:

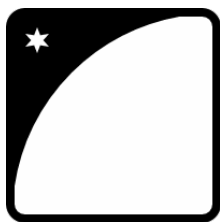
<faces-config

```
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
version="2.0">
```

</faces-config>

Agora já temos as informações necessárias para criar nosso primeiro projeto utilizando JSF.

Você pode também fazer o curso FJ-22 dessa apostila na Caelum



Querendo aprender ainda mais sobre boas práticas de Java, JSF, Web Services, testes e design patterns? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o **curso FJ-22** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

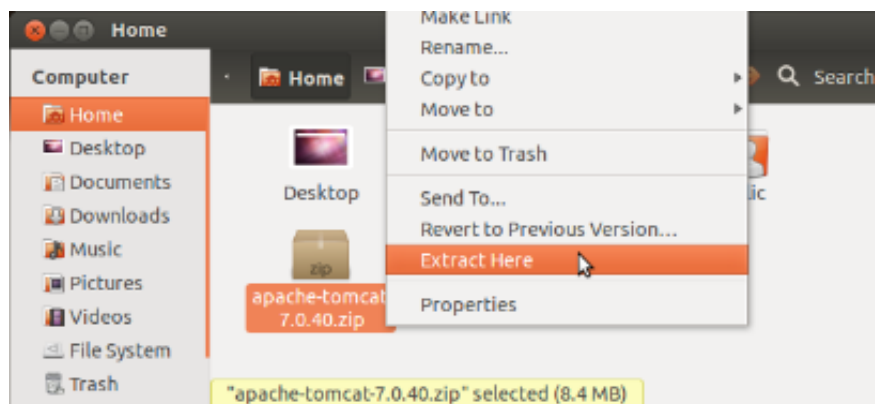
[Consulte as vantagens do curso *Lab. Java com Testes, JSF e Design Patterns*.](https://www.caelum.com.br/apostila-java-testes-jsf-web-services-design-patterns/introducao-ao-jsf-e-primefaces/)

7.3 – EXERCÍCIOS: INSTALANDO O TOMCAT E CRIANDO O PROJETO

1. Primeiramente, precisamos instalar o Tomcat. Usaremos a versão 7.x:

a. Vá no Desktop e entre na pasta *Caelum* e em seguida na pasta 22.

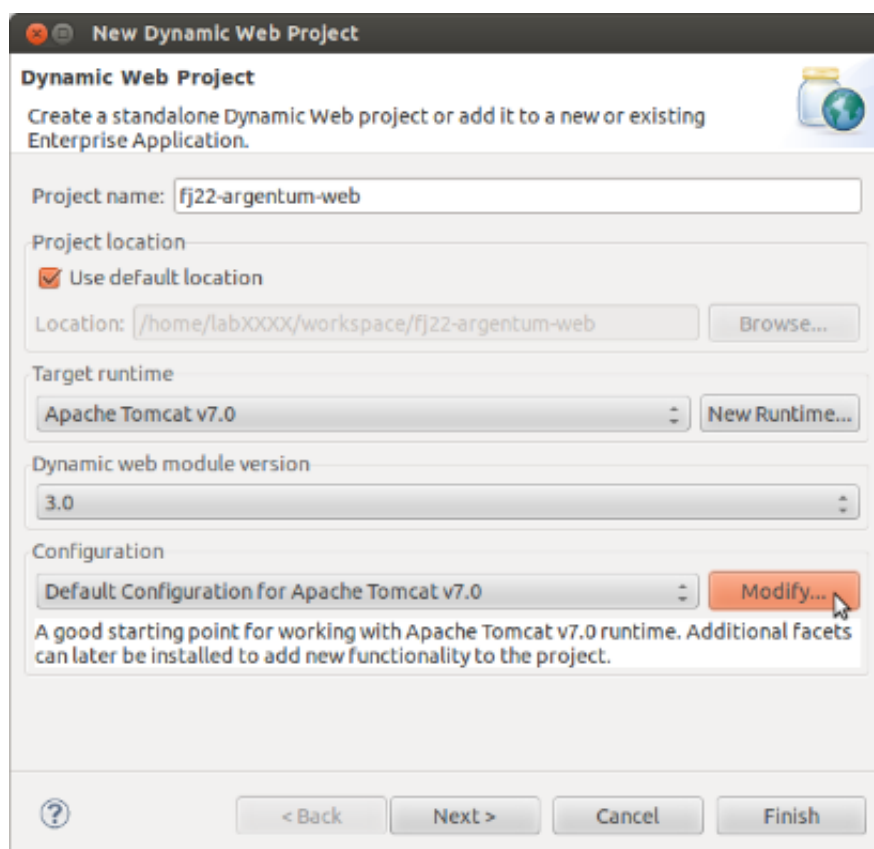
- b. Copie o arquivo zip do TomCat e cole ele na sua pasta Home.
- c. Clique com o botão direito e escolha *Extract here*.



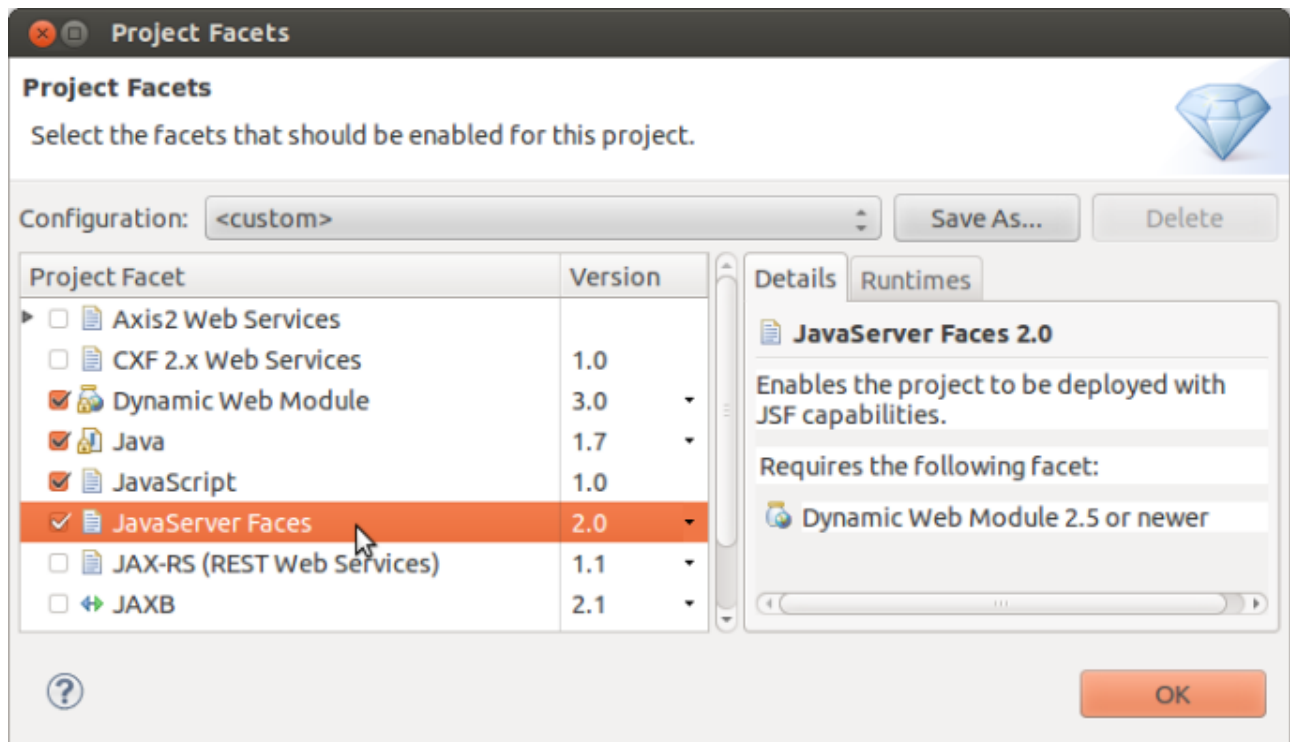
- d. O *Tomcat* já está pronto para o uso!

2. O próximo passo é criar o nosso projeto no Eclipse.

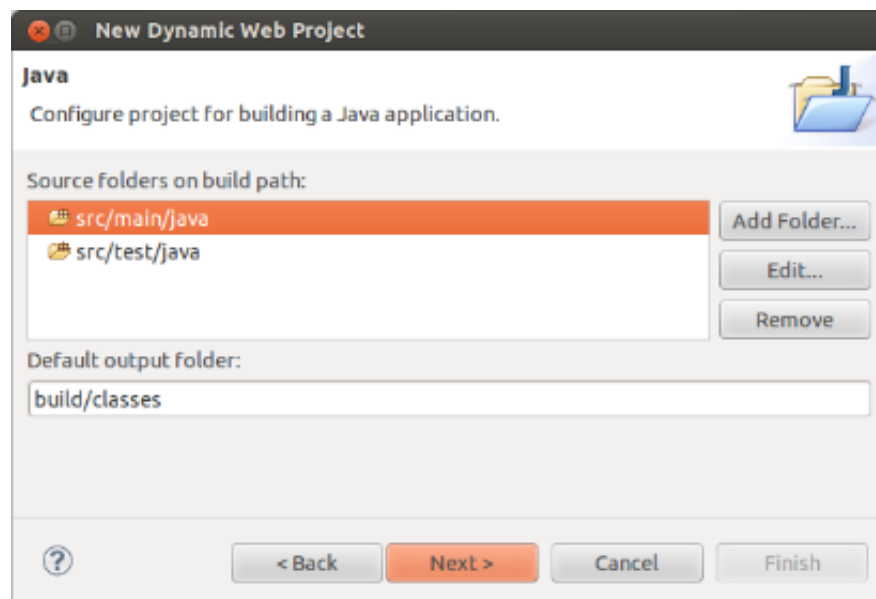
- a. Crie um novo projeto web usando o **ctrl + 3** *Dynamic Web Project*.
- b. Em *Project name* coloque `fj22-argentum-web`.
- c. Na seção *Configuration* clique em *Modify* para acrescentarmos suporte ao JSF.



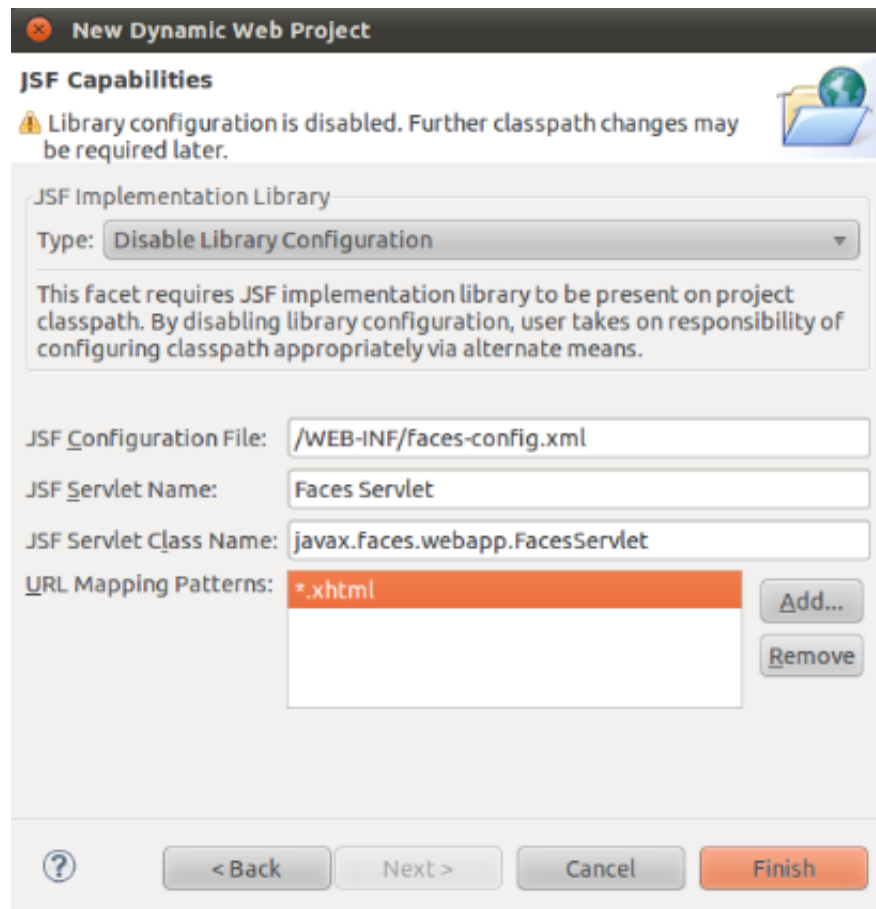
- d. Na tela que abre, marque o checkbox com **JavaServer Faces 2.0** e clique dê ok:



e. De volta à tela de criação do projeto, clique em **Next**. Nessa tela, faremos como no início do curso: removeremos a *source folder* padrão (*src*) e adicione as source folders *src/main/java* e *src/test/java*.



f. Dê *Next* mais duas vezes até chegar à tela de **JSF Capabilities**. Nessa tela, escolha a opção **Disable Library Configuration** para indicarmos para o Eclipse que nós mesmos copiaremos os JARs do JSF. Ainda nessa tela, na parte *URL Mapping Patterns*, **remova** o mapeamento */faces/** e **adicione** um novo mapeamento como ***.xhtml**

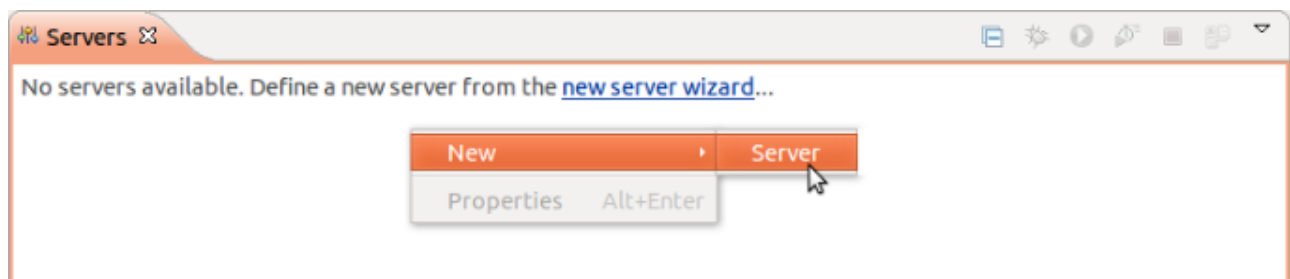


g. Clique em *Finish* e o projeto está criado.

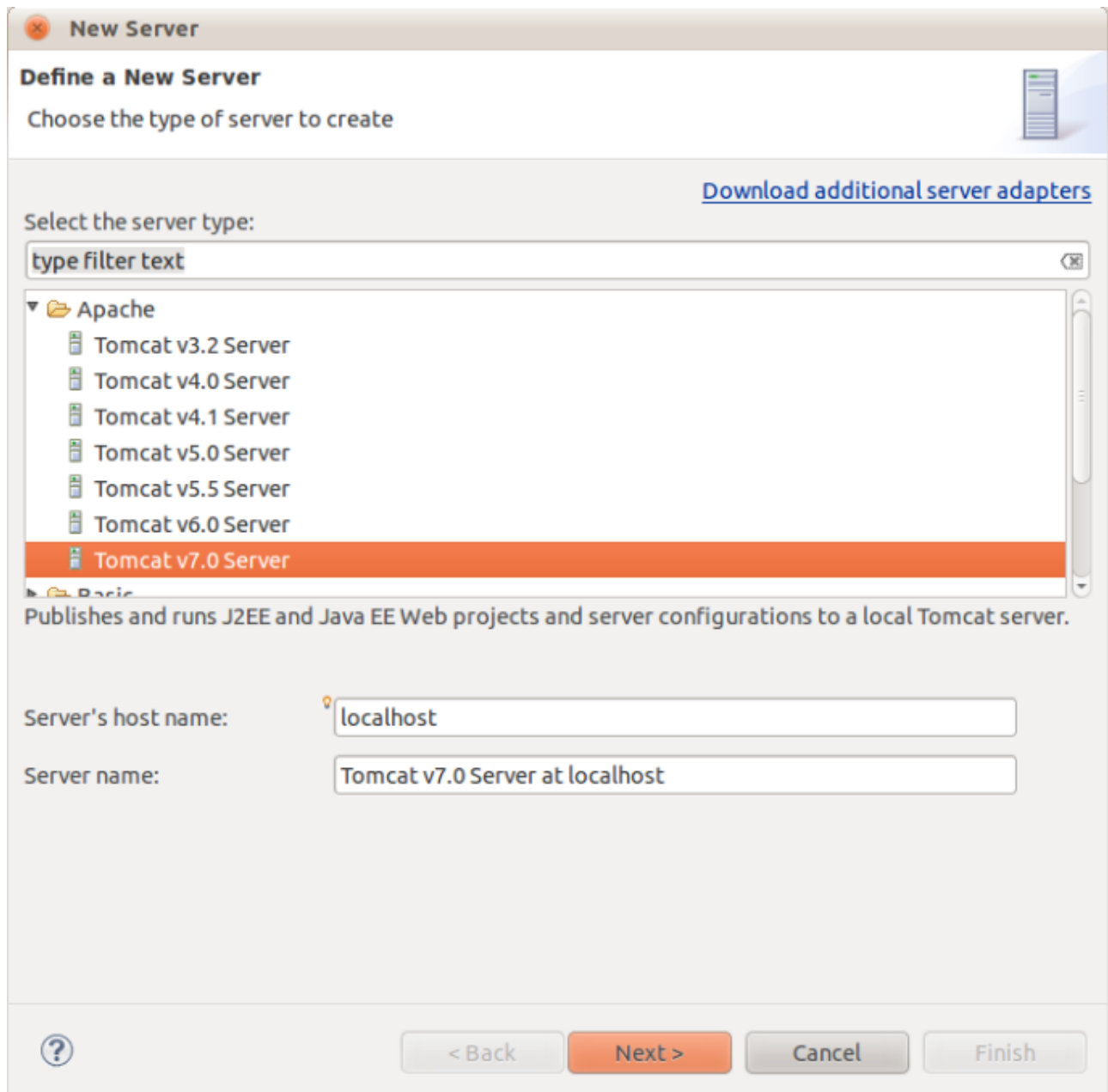
3. O próximo passo é configurar o Tomcat no Eclipse, para que possamos controlá-lo mais facilmente.

a. Dentro do Eclipse, abra a view *Servers*. Para isso, pressione **ctrl + 3**, digite *Servers* e escolha a view. Ela será aberta na parte inferior do seu Eclipse.

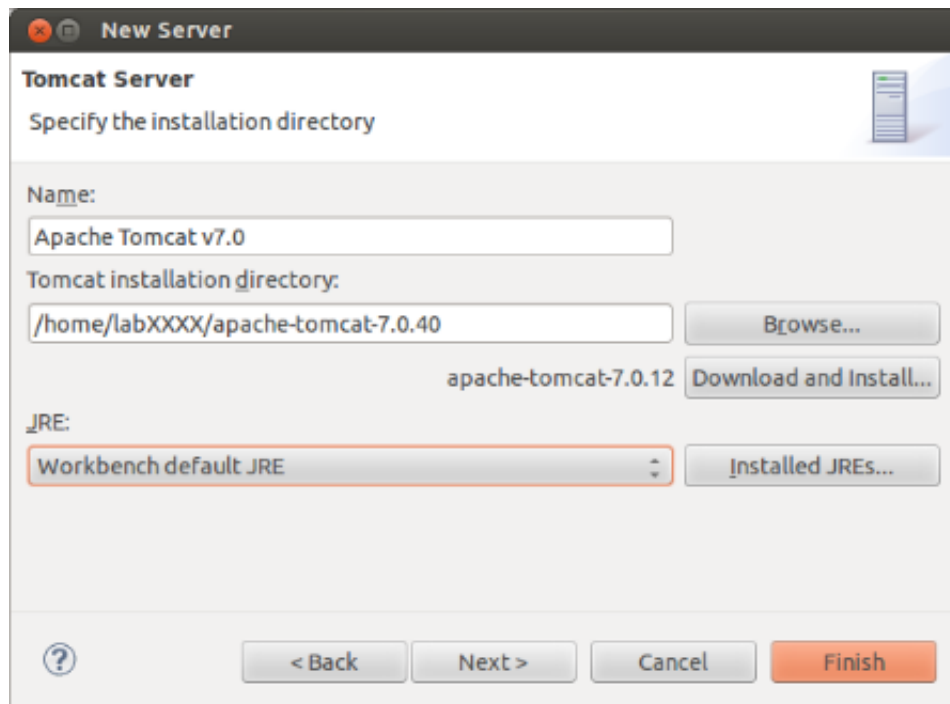
b. Dentro da aba *Servers* clique com o botão direito do mouse e escolha *New -> Server*. Se não quiser usar o mouse, você pode fazer **ctrl+3** *New server*.



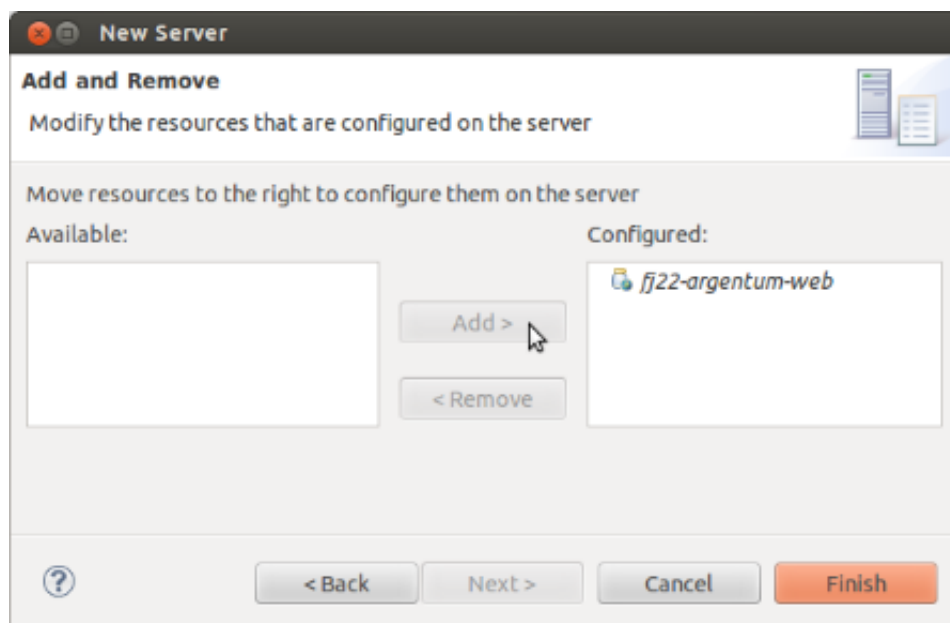
c. Dentro da Janela *New Server* escolha *Apache Tomcat v7.0 Server* e clique em *Next*.



d. O próximo passo é dizermos ao Eclipse em qual diretório instalamos o Tomcat. Clique no botão *Browse...* e escolha a pasta na qual você descompactou o *Tomcat*.



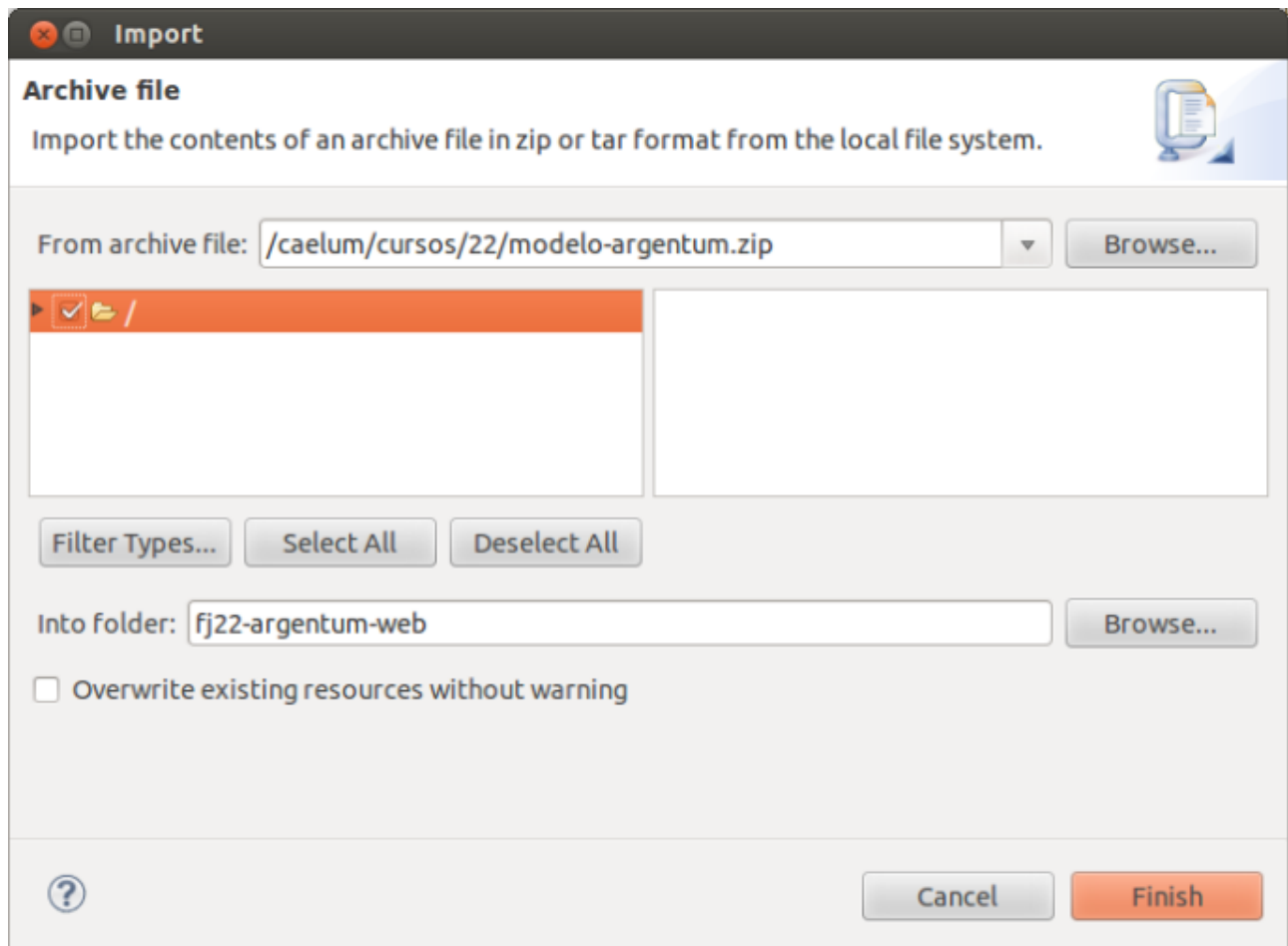
e. Clique em *Next* e, na próxima tela, selecione o projeto *fj22-argentum-web* no box *Available* (da esquerda), pressione o botão *Add >* (moverá para o box *Configured* da direita) e depois *Finish*.



f. Clique em *Finish*.

4. Por fim, precisamos importar do projeto anterior as classes como *Negociacao* ou *Candlestick*. Já o temos pronto na pasta */caelum/cursos/22/*, com o nome de *modelo-argentum.zip*. Precisamos apenas importá-lo:

- Para importá-lo, use **ctrl + 3** *Archive File* e escolha a opção **Import (Archive file)**.
- Em *Browse...*, selecione o nosso arquivo **modelo-argentum.zip** e finalize-o.



- Note que, com esse import, trouxemos também os jars da implementação Mojarra do JSF e do Primefaces, que usaremos daqui pra frente.

Para casa...

Se você está fazendo esse exercício em casa, certifique-se que seu projeto anterior está funcionando corretamente e simplesmente copie os pacotes dele para o novo.

Não esqueça de copiar também o jar do XStream para a pasta WebContent/WEB-INF/lib/.

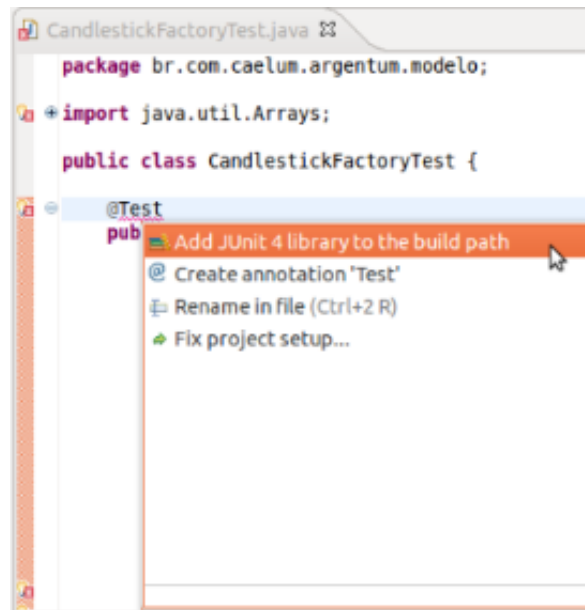
Além disso, no zip da aula ainda há os jars do JSF e do PrimeFaces, que usaremos a seguir. Nesta versão da apostila estamos usando as versões 2.x.x e 3.5.x, respectivamente. Links para o download:

- JSF: <https://javaserverfaces.java.net/download.html>
- Primefaces: <http://primefaces.org/downloads.html>

5. Nossas classes de teste (*src/test/java*) ainda apresentam problemas

relacionados ao JUnit. Falta adicioná-lo ao *Build Path*.

Abra a classe **CandlestickFactoryTest** e dê **ctrl + 1** na anotação `@Test`. Escolha a opção *Add JUnit 4 library to the build path*.



6. Finalmente, para evitar confusões mais para a frente, feche o projeto que fizemos nos outros dias de curso. Clique com o botão direito no **fj22-argentum-base** e escolha a opção **Close project**

7.4 - A PRIMEIRA PÁGINA COM JSF

Como configuramos, na criação do projeto, que o JSF será responsável por responder às requisições com extensão `.xhtml`. Dessa forma, tabalharemos com arquivos `xhtml` no restante do curso.

Vale relembrar uma diferença fundamental entre as duas formas de desenvolvimento para a web. A abordagem *action based*, como no SpringMVC e no VRaptor, focam seu funcionamento nas classes que contêm as lógicas. A view é meramente uma camada de apresentação do que foi processado no modelo.

Enquanto isso, o pensamento *component based* adotado pelo JSF leva a view como a peça mais importante -- é a partir das necessidades apontadas pelos componentes da view que o modelo é chamado e populado com dados.

As tags que representam os componentes do JSF estão em duas *taglibs* principais (bibliotecas de tags): a **core** e a **html**.

A taglib *html* contém os componentes necessários para montarmos nossa tela

gerando o HTML adequado. Já a *core* possui diversos componentes não visuais, como tratadores de eventos ou validadores. Por ora, usaremos apenas os componentes da *h:html*

Importando as tags em nossa página

Diferente da forma importação de taglibs em JSPs que vimos no curso de Java para a web (FJ-21), para importar as tags no JSF basta declararmos seus namespaces no arquivo `.xhtml`. Dessa forma, teremos:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

    <!-- aqui usaremos as tags do JSF -->

</html>
```

Definindo a interface da aplicação

Como qualquer outro aprendizado de tecnologia, vamos começar a explorar o JSF criando nossa primeira tela com uma mensagem de boas vindas para o usuário.

Como todo arquivo HTML, todo o cabeçalho deve estar dentro da tag `head` e o que será renderizado no navegador deve ficar dentro da tag `body`. Uma página padrão para nós seria algo como:

```
<html ...>
  <head>
    <!-- cabeçalho aqui -->
  </head>
  <body>
    <!-- informações a serem mostradas -->
  </body>
</html>
```

Quando estamos lidando com o JSF, no entanto, precisamos nos lembrar de utilizar preferencialmente as tags do próprio framework, já que, à medida que utilizarmos componentes mais avançados, o JSF precisará gerenciar os próprios *body* e *head* para, por exemplo, adicionar CSS e javascript que um componente requisitar.

Assim, usando JSF preferiremos utilizar as tags estruturais do HTML que vêm da taglib <http://java.sun.com/jsf/html>, nosso html vai ficar mais parecido com esse:

```
<html ...>
  <h:head>
    <!-- cabeçalho aqui -->
  </h:head>
  <h:body>
    <!-- informações a serem mostradas -->
  </h:body>
</html>
```

Mostrando informações com h:outputText

Como queremos mostrar uma saudação para o visitante da nossa página, podemos usar a tag `h:outputText`. É através do seu atributo `value` que definimos o texto que será apresentado na página.

Juntando tudo, nosso primeiro exemplo é uma tela simples com um texto:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

  <h:head>
    <title>Argentum Web</title>
  </h:head>
  <h:body>
    <h:outputText value = "Olá JSF!" />
  </h:body>
</html>
```

7.5 – INTERAGINDO COM O MODELO: MANAGED BEANS

O `h:outputText` é uma tag com um propósito aparentemente muito bobo e, no exemplo acima, é exatamente equivalente a simplesmente escrevermos "Olá JSF!" diretamente. E, de fato, para textos fixos, não há problema em escrevê-lo diretamente!

Contudo, se um pedaço de texto tiver que interagir com o modelo, uma lógica ou mesmo com outros componentes visuais, será necessário que ele também esteja guardado em um componente.

Exemplos dessas interações, no caso do `h:outputText`: mostrar informações vindas de um banco de dados, informações do sistema, horário de acesso, etc.

Para mostrar tais informações, precisaremos executar um código Java e certamente não faremos isso na camada de visualização: esse código ficará separado da *view*, em uma classe de modelo. Essas classes de modelo que

interagem com os componentes do JSF são os **Managed Beans**.

Estes, são apenas classezinhas simples que com as quais o JSF consegue interagir através do acesso a seus métodos. Nada mais são do que POJOs anotados com `@ManagedBean`.

POJO (Plain Old Java Object)

POJO é um termo criado por Martin Fowler, Rebecca Parsons e Josh Mackenzie que serve para definir um objeto simples. Segundo eles, o termo foi criado pois ninguém usaria objetos simples nos seus projetos pois não existia um nome extravagante para ele.

Se quisermos, por exemplo, mostrar quando foi o acesso do usuário a essa página, podemos criar a seguinte classe:

```
@ManagedBean
public class OlaMundoBean {

    public String getHorario() {
        SimpleDateFormat sdf = new SimpleDateFormat("hh:mm:ss");
        return "Atualizado em " + sdf.format(new Date());
    }
}
```

E, bem semelhantemente à forma padrão nas JSPs vistas no treinamento de Java para a Web, acessaremos o *getter* através da *Expression Language*. Existe apenas uma pequena diferença: para chamar os métodos no JSF, em vez do cifrão (\$), usaremos a cerquilha (#).

```
<h:outputText value="#{olaMundoBean.horario}" />
```

Ao fazer colocar o código acima, estamos dizendo que há uma classe gerenciada pelo JSF chamada **OlaMundoBean** que tem um método `getHorario` -- e que o retorno desse método será mostrado na página. É uma forma extremamente simples e elegante de ligar a *view* a métodos do *model*.

Tire suas dúvidas no novo G.U.J Respostas

O G.U.J é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do G.U.J é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais



de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

[Faça sua pergunta.](#)

7.6 – RECEBENDO INFORMAÇÕES DO USUÁRIO

Agora que já sabemos conectar a página à camada de modelo, fica fácil obter dados do usuário! Por nossa vivência com aplicações web, até mesmo como usuários, sabemos que a forma mais comum de trazer tais dados para dentro da aplicação é através de formulários.

A boa notícia é que no JSF não será muito diferente! Se para mostrar dados na página usamos a tag `h:outputText`, para trazer dados do usuário para dentro da aplicação, usaremos a tag `h:inputText`. Ela fará a ligação entre o atributo do seu bean e o valor digitado no campo.

Note que a ideia é a mesma de antes: como o JSF precisará interagir com os dados desse componente, não podemos usar a tag HTML que faria o mesmo trabalho. Em vez disso, usaremos a taglib de HTML provida pelo próprio JSF, indicando como a informação digitada será guardada no bean.

```
<h:outputLabel value="Digite seu nome:"/>
<h:inputText value="#{olaMundoBean.nome}"/>
```

Apenas com esse código, já podemos ver o texto *Digite seu nome* e o campo de texto onde o usuário digitará. Sabemos, no entanto, que não faz sentido ter apenas um campo de texto! É preciso ter também um botão para o usuário confirmar que acabou de digitar o nome e um formulário para agrupar todas essas tags.

Botão e o formulário em JSF

Esse é um pequeno ponto de divergência entre o HTML puro e o JSF. Em um simples formulário HTML, configuramos a *action* dele na própria tag `form` e o papel do botão é apenas o de mandar executar a ação já configurada.

Para formulários extremamente simples, isso é o bastante. Mas quando queremos colocar dois botões com ações diferentes dentro de um mesmo formulário, temos que recorrer a um JavaScript que fará a chamada correta.

Como dito antes, no entanto, o JSF tem a proposta de abstrair todo o protocolo

HTTP, o JavaScript e o CSS. Para ter uma estrutura em que o formulário é marcado apenas como um agregador de campos e cada um dos botões internos pode ter funções diferentes, a estratégia do JSF foi a de deixar seu form como uma tag simples e adicionar a configuração da ação ao próprio botão.

```
<h:form>
  <h:outputLabel for="nome" value="Digite seu nome:"/>
  <h:inputText id="nome" value="#{olaMundoBean.nome}"/>
  <h:commandButton value="Ok" action="#{olaMundoBean.digaOi}"/>
</h:form>
```

Quando o usuário clica no botão *Ok*, o JSF chama o setter do atributo nome do *OlaMundoBean* e, logo em seguida, chama o método *digaOi*. Repare que esta ordem é importante: o método provavelmente dependerá dos dados inseridos pelo usuário.

Note, também, que teremos um novo método no *managed bean* chamado *digaOi*. Os botões sempre estão atrelados a métodos porque, na maior parte dos casos, realmente queremos executar alguma ação além da chamada do setter. Essa ação pode ser a de disparar um processo interno, salvar no banco ou qualquer outra necessidade.

O que fazer enquanto não ainda houver informação?

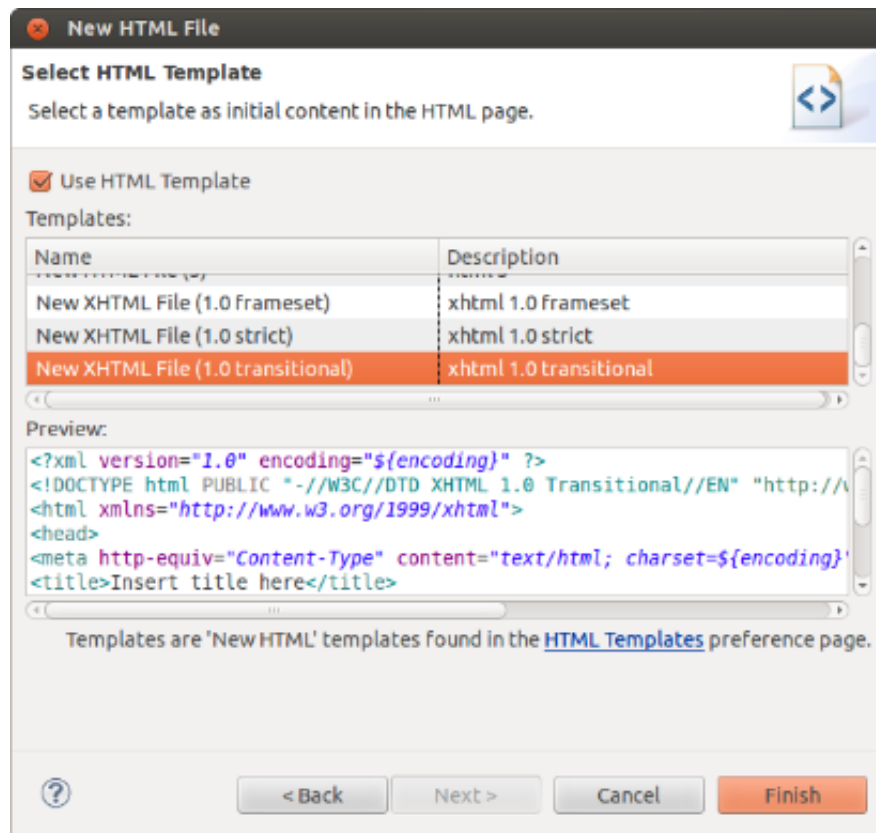
Sabendo que, antes de chamar o método correspondente à ação do botão, o JSF preenche os atributos através dos setters, sabemos que teremos a informação a ser mostrada para o usuário.

No entanto, muitas vezes não gostaríamos de mostrar um campo enquanto ele não estiver preenchido e, felizmente, o JSF tem uma forma bastante simples de só mostrar um *h:outputText* na tela apenas se a informação estiver preenchida! Basta usar o atributo *rendered*:

```
<h:outputText value="Oi #{olaMundoBean.nome}"
  rendered="#{not empty olaMundoBean.nome}"/>
```

7.7 – EXERCÍCIOS: OS PRIMEIROS COMPONENTES JSF

1. Use **ctrl + N HTML** para criar o arquivo *olaMundo.xhtml* na pasta *WebContent* da sua aplicação. Escolha *Next* e, na próxima tela, escolha o template *xhtml 1.0 transitional*, usualmente a última opção da lista:



Selecione a mesma opção da imagem acima e pressione *Finish*.

Implemente nosso primeiro código JSF com apenas uma saída de texto:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Argentum</title>
    </h:head>

    <h:body>
        <h:outputText value="Ola Mundo" />
    </h:body>
</html>
```

2. Inicie o Tomcat e acesse a URL: <http://localhost:8080/fj22-argentum-web/olaMundo.xhtml>

3. Verifique o código fonte gerado pela página. Repare que ele não é nada mais que simples HTML. Para isso, na maior parte dos navegadores, use **ctrl + U**.

Repare no uso das tags `<h:head>`, `<h:body>` e `<h:outputText>`: elas não aparecem no html gerado! Sua função é apenas indicar para o JSF como gerar o código HTML necessário para o exemplo funcionar.

4. Além de usar mensagens fixas, poderíamos fazer com que a mensagem seja

devolvida de uma classe responsável por prover objetos para uma view: um dos chamados `ManagedBeans`. Vamos começar criando essa classe contendo apenas a mensagem inicial.

Crie uma classe chamada `OlaMundoBean`, com apenas o atributo `mensagem` já inicializada, seu getter e não esqueça de **anotar a classe** com `@ManagedBean`

```
@ManagedBean
public class OlaMundoBean {

    private String mensagem = "Quem é você?";

    public String getMensagem() {
        return mensagem;
    }
}
```

5. Alteremos o arquivo `xhtml`, então, para que ele use a mensagem *Quem é você?* que escrevemos *hard-coded* na classe `OlaMundoBean`. Usaremos a *Expression Language* específica do JSF para isso, que é capaz de pegar informações de qualquer classe configurada como um `ManagedBean`.

Basta alterar o *value* da tag `h:outputText`:

```
...
<h:body>
    <h:outputText value="#{olaMundoBean.mensagem}" />
</h:body>
```

6. Agora, se quisermos pegar a resposta do usuário e cumprimentá-lo propriamente, podemos adicionar à nossa página um campo de texto para que o usuário digite seu nome. Então, trocaremos a mensagem cumprimentando ele. Começamos pelas alterações no `olaMundo.xhtml`, adicionando um `h:inputText` e um botão para o usuário enviar seu nome, como abaixo.

Atenção! Não esqueça da tag `h:form` em volta do formulário. Lembre-se que, sem ela, os botões não funcionam.

```
...
<h:body>
    <h:form>
        <h:outputText value="#{olaMundoBean.mensagem}" /><br />
        <h:inputText value="#{olaMundoBean.nome}" />
        <h:commandButton action="#{olaMundoBean.nomeFoiDigitado}"
            value="Ok" />
    </h:form>
</h:body>
```

7. Essa alteração, no entanto, não é suficiente. Se você rodar o servidor agora,

notará que a página, que antes funcionava, agora lança uma `ServletException` informando que *Property 'nome' not found on type br.com.caelum.argentum.bean.OlaMundoBean*.

Isto é, falta adicionarmos o atributo `nome` e seu `getter` à página, como fizemos com a mensagem, no outro exercício. Adicione à classe `OlaMundoBean` o atributo e seu `getter`.

```
@ManagedBean
public class OlaMundoBean {
    ...
    private String nome;

    public String getNome() {
        return nome;
    }
    ...
}
```

8. Agora sim podemos ver a mensagem, o campo de texto e o botão. Contudo, ao apertar o botão, levamos uma `javax.el.PropertyNotFoundException` informando que `nome` é um atributo não alterável.

Faltou adicionarmos o `setter` do atributo à `OlaMundoBean`, para que o JSF possa preenchê-lo! Além disso, o botão chamará o método `nomeFoiDigitado`, que também não existe ainda.

Complete a classe com o `setter` faltante e o método `nomeFoiDigitado`, reinicie o servidor e teste!

```
@ManagedBean
public class OlaMundoBean {
    // ...tudo o que já existia aqui

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

    public void nomeFoiDigitado() {
        System.out.println("\nChamou o botão");
    }
}
```

9. (Opcional) Para entender melhor o ciclo de execução de cada chamada ao JSF, adicione `System.out.println("nome do método")` a cada um dos métodos da sua aplicação e veja a ordem das chamadas pelo console do Eclipse.

7.8 – A LISTA DE NEGOCIAÇÕES

Agora que já aprendemos o básico do JSF, nosso objetivo é listar em uma página as negociações do web service que o Argentum consome. Nessa listagem, queremos mostrar as informações das negociações carregadas -- isto é, queremos uma forma de mostrar preço, quantidade e data de cada negociação. E a forma mais natural de apresentar dados desse tipo é, certamente, uma tabela.

Até poderíamos usar a tabela que vem na taglib padrão do JSF, mas ela é bastante limitada e não tem pré-definições de estilo. Isto é, usando a taglib padrão, teremos sim uma tabela no HTML, mas ela será mostrada da forma mais feia e simples possível.

Já falamos, contudo, que a proposta do JSF é abstrair toda a complexidade relativa à web -- e isso inclui CSS, formatações, JavaScript e tudo o mais. Então, em apoio às tags básicas, algumas bibliotecas mais sofisticadas surgiram. As mais conhecidas delas são PrimeFaces, RichFaces e IceFaces.

Taglibs como essas oferecem um visual mais bacana já pré-pronto e, também, diversas outras facilidades. Por exemplo, uma tabela que utilize as tags do Primefaces já vem com um estilo bonito, possibilidade de colocar cabeçalhos nas colunas e até recursos mais avançados como paginação dos registros.

O componente responsável por produzir uma tabela baseada em um modelo se chama `dataTable`. Ele funciona de forma bem semelhante ao `for` do Java 5 ou o `forEach` da JSTL: itera em uma lista de elementos atribuindo cada item na variável definida.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.org/ui">
  <h:head>
    <title>Argentum</title>
  </h:head>
  <h:body>
    <p:dataTable var="negociacao" value="#{argentumBean.negociacoes}">

      </p:dataTable>
    </h:body>
  </html>
```

O código acima chamará o método `getNegociacoes` da classe `ArgentumBean` e iterará pela lista devolvida atribuindo o objeto à variável `negociacao`. Então, para cada coluna que quisermos mostrar, será necessário apenas manipular a

negociação do momento.

E, intuitivamente o bastante, cada coluna da tabela será representada pela tag `p:column`. Para mostrar o valor, você pode usar a tag que já vimos antes, o `h:outputText`. Note que as tags do Primefaces se integram perfeitamente com as básicas do JSF.

```
<p:dataTable var="negociacao" value="#{argentumBean.negociacoes}">
  <p:column headerText="Preço">
    <h:outputText value="#{negociacao.preco}"/>
  </p:column>
  ... outras colunas
</p:dataTable>
```

Falta ainda implementar a classe que cuidará de devolver essa lista de negociações. O código acima sugere que tenhamos uma classe chamada `ArgentumBean`, gerenciada pelo JSF, que tenha um getter de negociações que pode, por exemplo, trazer essa lista direto do `ClienteWebService` que fizemos anteriormente:

```
@ManagedBean
public class ArgentumBean {

    public List<Negociacao> getNegociacoes() {
        return new ClienteWebService().getNegociacoes();
    }
}
```

Da forma acima, o exemplo já funciona e você verá a lista na página. No entanto, nesse exemplo simples o JSF chamará o método `getNegociacoes` duas vezes durante uma mesma requisição. Isso não seria um problema se ele fosse um getter padrão, que devolve uma referência local, mas note como nosso `getNegociacoes` vai buscar a lista diretamente no web service. Isso faz com que, para construir uma simples página, tenhamos que esperar a resposta do serviço... duas vezes!

Esse comportamento não é interessante. Nós gostaríamos que o `Argentum` batesse no serviço em busca dos dados apenas uma vez por requisição, e não a cada vez que o JSF chame o *getter*. Isso significa que o acesso ao serviço não pode estar diretamente no método `getNegociacoes`, que deve apenas devolver a lista pré-carregada.

No JSF, o comportamento padrão diz que um objeto do `ManagedBean` dura por uma requisição. Em outras palavras, o escopo padrão dos *beans* no JSF é o de requisição. Isso significa que um novo `ArgentumBean` será criado a cada vez que um

usuário chamar a página da listagem. E, para cada chamada a essa página, precisamos buscar a lista de negociações no serviço apenas uma vez. A resposta para esse problema, então, é bastante simples e apareceu logo no início do aprendizado do Java orientado a objetos.

Basta colocar a chamada do web service naquele bloco de código que é chamado apenas na criação do objeto, isto é, no construtor. Ao armazenar a listagem em um atributo, o *getter* de negociações passa a simplesmente devolver a referência, evitando as múltiplas chamadas a cada requisição.

```
@ManagedBean
public class ArgentumBean {

    private List<Negociacao> negociacoes;

    public ArgentumBean() {
        ClienteWebService cliente = new ClienteWebService();
        this.negociacoes = cliente.getNegociacoes();
    }

    public List<Negociacao> getNegociacoes() {
        return this.negociacoes;
    }
}
```

Juntando as informações dessa seção, já conseguimos montar a listagem de negociações com os dados vindos do *web service*. E o processo será muito frequentemente o mesmo para as diversas outras telas: criamos a página usando as tags do Primefaces em complemento às básicas do JSF, implementamos a classe que cuidará da lógica por trás da tela e a anotamos com `@ManagedBean`.

Nova editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não conhecem programação para revisar os livros tecnicamente a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](https://www.caelum.com.br/apostila-java-testes-jsf-web-services-design-patterns/introducao-ao-jsf-e-primefaces/)

7.9 – FORMATAÇÃO DE DATA COM JSF

A tabela já é funcional, mas com a data mal formatada. O componente não sabe como gostaríamos de formatar a data e chama por de baixo dos panos o método `toString` da data para receber uma apresentação como `String`.

Preço	Quantidade	Volume	
321.65	18	5789.7	<code>java.util.GregorianCalendar[time=1380758400000,areFieldsSet=true,areAllFie</code>
384.39	22	8456.58	<code>java.util.GregorianCalendar[time=1380758400000,areFieldsSet=true,areAllFie</code>
298.99	17	5082.83	<code>java.util.GregorianCalendar[time=1380758400000,areFieldsSet=true,areAllFie</code>
309.49	18	5570.82	<code>java.util.GregorianCalendar[time=1380758400000,areFieldsSet=true,areAllFie</code>
392.33	23	9023.59	<code>java.util.GregorianCalendar[time=1380844800000,areFieldsSet=true,areAllFie</code>
425.02	25	10625.5	<code>java.util.GregorianCalendar[time=1380844800000,areFieldsSet=true,areAllFie</code>

A forma clássica de resolver esse problema seria através de um getter que traria a data formatada por um `SimpleDateFormat`. Mas, assim como a JSTL vista no curso de Java para a Web, o JSF também tem uma tag para formatar valores, números e, claro, datas. Essas tags e muitas outras, são parte da biblioteca fundamental de tags lógicas do JSF e, para usá-las, será necessário importar tal taglib.

Assim como as bibliotecas de tags de HTML e do Primefaces, para utilizar essas será necessário declará-las no namespace da sua página.

Daí, podemos facilmente mudar a forma padrão de exibição usando o componente de formatação `f:convertDateTime` que define um *pattern* para a data. É importante lembrar que, internamente, o `f:convertDateTime` acaba fazendo uma chamada ao `SimpleDateFormat` e, assim, só podemos formatar objetos do tipo `java.util.Date` com ele. Por essa razão, chamaremos o método *getTime* que devolve a representação em `Date` do `Calendar` em questão. Mais uma vez podemos omitir a palavra "get" com expression language. Segue a tabela completa:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.org/ui">

  <h:body>
    <p:dataTable var="negociacao" value="#{argentumBean.negociacoes}">

      ... outras colunas, e então:
      <p:column headerText="Data">
        <h:outputText value="#{negociacao.data.time}">
          <f:convertDateTime pattern="dd/MM/yyyy"/>
        </h:outputText>
      </p:column>
    </p:dataTable>
  </h:body>
```

```
</html>
```

7.10 – EXERCÍCIOS: P:DATATABLE PARA LISTAR AS NEGOCIAÇÕES DO WEB SERVICE

1. Use **ctrl + N HTML** para criar um novo arquivo na pasta WebContent chamado `index.xhtml`. Como já fizemos antes, clique em *Next* e, na tela seguinte, escolha o template *xhtml 1.0 transitional*.

O Eclipse vai gerar um arquivo com um pouco de informações a mais, mas ainda muito parecido com o seguinte, onde mudamos o title:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Argentum Web</title>
  </head>
  <body>

  </body>
</html>
```

2. Como vamos usar o JSF nesse arquivo e já temos até mesmo o JAR `primefaces-3.x.jar` adicionado ao projeto (veja em `WebContent/WEB-INF/lib`) basta declarar os namespaces das taglibs do JSF e do Primefaces, que usaremos no exercício.

Além disso, para que os componentes consigam incluir seu CSS à nossa página, altere as tags `head` e `body` de forma a usar suas versões gerenciadas pelo JSF:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.org/ui">
  <h:head>
    <title>Argentum Web</title>
  </h:head>
  <h:body>

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

3. Agora, **dentro do `h:body`**, vamos começar a montar nossa tabela de negociações. O componente que usaremos para isso é o `p:datatable`, do Primefaces. Ele precisará da lista de negociações e, assim como um `forEach`, uma variável para que cada coluna seja preenchida.

```
<p:datatable var="negociacao" value="#{argentumBean.negociacoes}">

</p:datatable>
```

4. Esse código acima diz que o componente `dataTable` do Primefaces chamará o método `getNegociacoes()` da classe `ArgentumBean` e, para cada linha da tabela, disponibilizará a negociação da vez na variável `negociacao`.

O problema é que o managed bean `ArgentumBean` ainda não existe e, claro, nem o método `getNegociacoes()` dela. E como cada vez que a página `index.xhtml` for requisitada ela fará algumas chamadas ao `getNegociacoes`, faremos a chamada ao *webservice* no construtor e, a cada chamada ao getter, apenas devolveremos a referência à mesma lista.

- a. Crie a classe `ArgentumBean` com **ctrl + N Class**, no pacote `br.com.caelum.argentum.bean` e anote ela com `@ManagedBean`.

```
@ManagedBean
public class ArgentumBean {

}
```

- b. Adicione o construtor que faça a chamada ao webservice através do `ClienteWebservice`, guarde a lista em um atributo e crie o getter que o componente chamará.

```
@ManagedBean
public class ArgentumBean {

    private List<Negociacao> negociacoes;

    public ArgentumBean() {
        negociacoes = new ClienteWebService().getNegociacoes();
    }

    public List<Negociacao> getNegociacoes() {
        return negociacoes;
    }
}
```

5. Agora, nossa página já não dá erro, mas nada é mostrado na tela, quando a acessamos. Falta indicarmos quais colunas queremos na nossa tabela -- no nosso caso: preço, quantidade, volume e data. Em cada coluna, adicionaremos um título e cada uma delas também mostrará o valor de texto.

Para criar a coluna com o título, usaremos o componente `p:column` e, como já fizemos antes, para mostrar o valor necessário, usaremos a `h:outputText`.

```
<p:dataTable var="negociacao" value="#{argentumBean.negociacoes}">
  <p:column headerText="Preço">
    <h:outputText value="#{negociacao.preco}"/>
  </p:column>
  <p:column headerText="Quantidade">
```



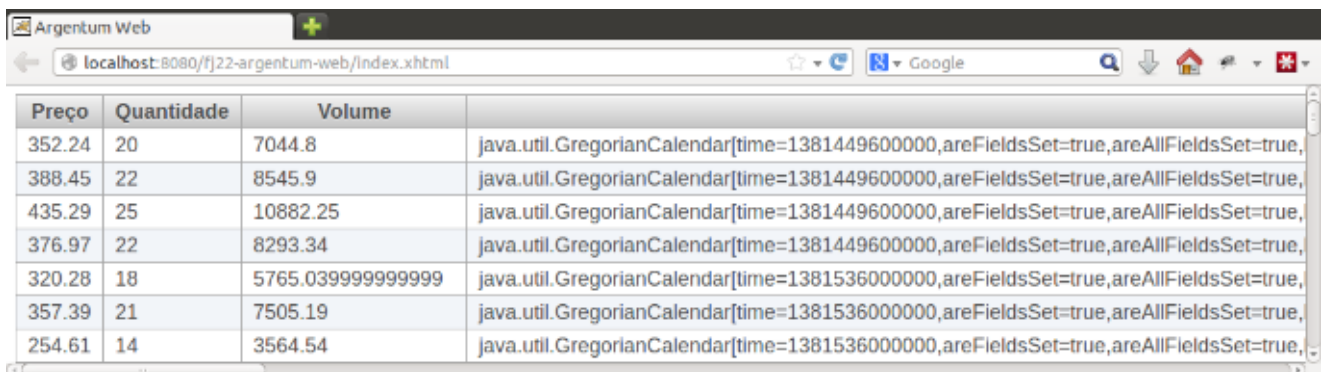
```

    <h:outputText value="#{negociacao.quantidade}"/>
  </p:column>
  <p:column headerText="Volume">
    <h:outputText value="#{negociacao.volume}"/>
  </p:column>
  <p:column headerText="Data">
    <h:outputText value="#{negociacao.data}"/>
  </p:column>
</p:dataTable>

```

6. Reinicie o Tomcat e acesse em seu navegador o endereço

<http://localhost:8080/fj22-argentum-web/index.xhtml>. O resultado deve ser algo parecido com:



Preço	Quantidade	Volume	Data
352.24	20	7044.8	java.util.GregorianCalendar[time=1381449600000,areFieldsSet=true,areAllFieldsSet=true,
388.45	22	8545.9	java.util.GregorianCalendar[time=1381449600000,areFieldsSet=true,areAllFieldsSet=true,
435.29	25	10882.25	java.util.GregorianCalendar[time=1381449600000,areFieldsSet=true,areAllFieldsSet=true,
376.97	22	8293.34	java.util.GregorianCalendar[time=1381449600000,areFieldsSet=true,areAllFieldsSet=true,
320.28	18	5765.039999999999	java.util.GregorianCalendar[time=1381536000000,areFieldsSet=true,areAllFieldsSet=true,
357.39	21	7505.19	java.util.GregorianCalendar[time=1381536000000,areFieldsSet=true,areAllFieldsSet=true,
254.61	14	3564.54	java.util.GregorianCalendar[time=1381536000000,areFieldsSet=true,areAllFieldsSet=true,

7. As informações de preço, quantidade e volume estão legíveis, mas a data das negociações está mostrando um monte de informações que não nos interessam. Na verdade, o que precisamos na coluna data é de informações de dia, mês, ano e, no máximo, horário de cada movimentação.

Adicione a tag `f:convertDateTime` à coluna da data. Essa tag modificará o comportamento da `h:outputText` para mostrá-lo formatado de acordo com o padrão passado. Note que a tag `h:outputText` passará a ser fechada depois da formatação da data:

```

...
<p:column headerText="Data">
  <h:outputText value="#{negociacao.data.time}">
    <f:convertDateTime pattern="dd/MM/yyyy"/>
  </h:outputText>
</p:column>
...

```

7.11 – PARA SABER MAIS: PAGINAÇÃO E ORDENAÇÃO

O componente `p:dataTable` sabe listar itens, mas não pára por aí. Ele já vem com várias outras funcionalidades frequentemente necessárias em tabelas já prontas e fáceis de usar.

Muitos dados

Por exemplo, quando um programa traz uma quantidade muito grande de dados, isso pode causar uma página pesada demais para o usuário que provavelmente nem olhará com atenção todos esses dados.

Uma solução clássica para resultados demais é mostrá-los aos poucos, apenas conforme o usuário indicar que quer ver os próximos resultados. Estamos, é claro, falando da paginação dos resultados e o componente de tabelas do Primefaces já a disponibiliza!

Para habilitar a paginação automática, basta adicionar o atributo `paginator="true"` à sua `p:dataTable` e definir a quantidade de linhas por página pelo atributo `rows`. A definição da tabela de negociações para paginação de 15 em 15 resultados ficará assim:

```
<p:dataTable var="negociacao" value="#{argenteumBean.negociacoes}"
    paginator="true" rows="15">

    <!-- colunas omitidas -->
</p:dataTable>
```

Essa pequena mudança já traz uma visualização mais legal para o usuário, mas estamos causando um problema silencioso no servidor. A cada vez que você chama uma página de resultados, a cada requisição, o `ArgenteumBean` é recriado e perdemos a lista anterior. Assim, na criação da nova instância de `ArgenteumBean`, seu construtor é chamado e acessamos novamente o webservice.

Como recebemos a lista completa do webservice, podíamos aproveitar a mesma lista para todas as páginas de resultado e, felizmente, isso também é bastante simples.

O comportamento padrão de um `ManagedBean` é durar apenas uma requisição. Em outras palavras, o escopo padrão de um `ManagedBean` é de *request*. Com apenas uma anotação podemos alterar essa duração. Os três principais escopos do JSF são:

- **RequestScoped:** é o escopo padrão. A cada requisição um novo objeto do bean será criado;
- **ViewScoped:** escopo da página. Enquanto o usuário estiver na mesma página, o bean é mantido. Ele só é recriado quando acontece uma navegação em si, isto é, um botão abre uma página diferente ou ainda quando acessamos novamente a página atual.

- **SessionScoped:** escopo de sessão. Enquanto a sessão com o servidor não expirar, o mesmo objeto do `ArgentumBean` atenderá o mesmo cliente. Esse escopo é bastante usado, por exemplo, para manter o usuário logado em aplicações.

No nosso caso, o escopo da página resolve plenamente o problema: enquanto o usuário não recarregar a página usaremos a mesma listagem. Para utilizá-lo, basta adicionar ao *bean* a anotação `@ViewScoped`. No exemplo do `Argentum`:

```
@ManagedBean
@ViewScoped
public class ArgentumBean {
    ...
}
```

Sempre que um `ManagedBean` possuir o escopo maior que o escopo de requisição, ele deverá implementar a interface `Serializable`:

```
@ManagedBean
@ViewScoped
public class ArgentumBean implements Serializable {
    ...
}
```

Tirando informações mais facilmente

Outra situação clássica que aparece quando lidamos com diversos dados é precisarmos vê-los de diferentes formas em situações diversas.

Considere um sistema que apresenta uma tabela de contatos. Se quisermos encontrar um contato específico nela, é melhor que ela esteja ordenada pelo nome. Mas caso precisemos pegar os contatos de todas as pessoas de uma região, é melhor que a tabela esteja ordenada, por exemplo, pelo DDD.

Essa ideia de ordenação é extremamente útil e muito presente em aplicações. Como tal, essa funcionalidade também está disponível para tabelas do Primefaces. Apenas, como podemos tornar diversas colunas ordenáveis, essa configuração fica na tag da coluna.

Para tornar uma coluna ordenável, é preciso adicionar um simples atributo `sortBy` à tag `h:column` correspondente. Esse atributo torna o cabeçalho dessa coluna em um elemento clicável e, quando clicarmos nele, chamará a ordenação.

Contudo, exatamente pela presença de elementos clicáveis, será necessário colocar a tabela dentro de uma estrutura que comporte botões em HTML: um formulário. E, como quem configurará o que cada clique vai disparar é o JSF, será necessário usar o formulário da taglib de HTML dele. Resumidamente, precisamos

colocar a tabela inteira dentro do componente `h:form`.

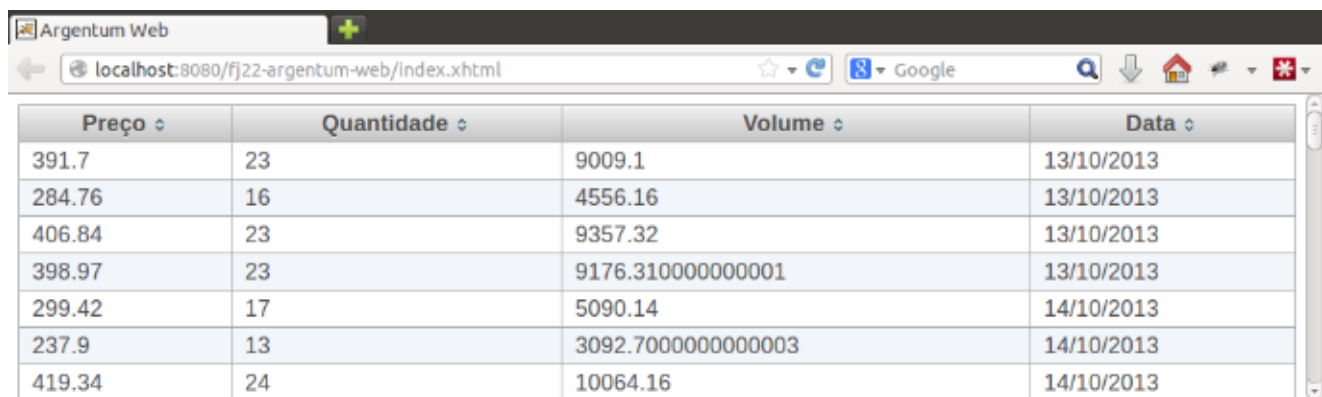
Se quiséssemos tornar ordenáveis as colunas da tabela de negociações, o resultado final seria algo como:

```
<h:form id="listaNegociacao">
  <p:dataTable var="negociacao" value="#{argentumBean.negociacoes}">

    <p:column sortBy="#{negociacao.preco}" headerText="Preço" >
      <h:outputText value="#{negociacao.preco}" />
    </p:column>

    <!-- outras colunas omitidas -->
  </p:dataTable>
</h:form>
```

Se permitirmos ordenar por qualquer coluna do modelo `Negociacao`, teremos um resultado bem atraente:



The screenshot shows a web browser window titled 'Argentum Web' with the address 'localhost:8080/fj22-argentum-web/index.xhtml'. The browser displays a table with four columns: 'Preço', 'Quantidade', 'Volume', and 'Data'. The table contains seven rows of transaction data.

Preço ↕	Quantidade ↕	Volume ↕	Data ↕
391.7	23	9009.1	13/10/2013
284.76	16	4556.16	13/10/2013
406.84	23	9357.32	13/10/2013
398.97	23	9176.310000000001	13/10/2013
299.42	17	5090.14	14/10/2013
237.9	13	3092.7000000000003	14/10/2013
419.34	24	10064.16	14/10/2013

Note que não foi necessário adicionar código algum à classe `ArgentumBean`! Note também que é até possível usar ambas as funcionalidades na mesma tabela. E essas são apenas algumas das muitas facilidades que o `p:dataTable` oferece. Vale a pena verificar o *showcase* e documentação no site do Primefaces.

Já conhece os cursos online Alura?



A **Alura** oferece dezenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Java, Ruby, Web, Mobile, .NET e outros, com uma **assinatura** que dá acesso a todos os cursos.

[Conheça os cursos online Alura.](https://www.caelum.com.br/apostila-java-testes-jsf-web-services-design-patterns/introducao-ao-jsf-e-primefaces/)

7.12 - EXERCÍCIO OPCIONAL: ADICIONE PAGINAÇÃO E ORDENAÇÃO À TABELA

1. Vamos colocar paginação na tabela. Adicione os atributos `paginator="true"` e `rows="15"`. Adicione os atributos `paginator` e `rows`:

```
<p:dataTable var="negociacao" value="#{argentumBean.negociacoes}"
  paginator="true" rows="15">
```

Salve a página, suba o servidor e acesse no seu navegador o endereço

<http://localhost:8080/fj22-argentum-web/index.xhtml>. Agora você já consegue ver resultados paginados de 15 em 15 negociações:

Preço	Quantidade	Volume	Data
360.91	21	7579.110000000001	16/10/2013
230.32	13	2994.16	16/10/2013
353.19	20	7063.8	16/10/2013
354.78	20	7095.599999999999	16/10/2013
332.23	19	6312.370000000001	17/10/2013
353.51	20	7070.2	17/10/2013
288.7	16	4619.2	17/10/2013
331.43	19	6297.17	17/10/2013
286.54	16	4584.64	18/10/2013
277.5	16	4440.0	18/10/2013
280.82	16	4493.12	18/10/2013
337.81	19	6418.39	18/10/2013
413.78	24	9930.72	19/10/2013
425.21	25	10630.25	19/10/2013
286.83	16	4589.28	19/10/2013

2. Para evitar chamar o webservice a cada vez que pedimos os próximos resultados paginados, **adicione** a anotação `@ViewScoped` à classe `ArgentumBean`:

```
@ManagedBean
@ViewScoped
public class ArgentumBean {
    ...
}
```

3. Como estamos utilizando um escopo maior do que o escopo de requisição, nosso `ManagedBean` precisa implementar a interface `Serializable`. Adicione a implementação da interface `Serializable` à classe `ArgentumBean`:

```
@ManagedBean
@ViewScoped
public class ArgentumBean implements Serializable {
    ...
}
```

4. Também será necessário implementar a interface `Serializable` na classe `Negociacao`, pois ela é utilizada pela classe `ArgentumBean`:

```
public class Negociacao implements Serializable {  
    ...  
}
```

5. Deixe as colunas ordenáveis, use o atributo `sortBy` em cada atributo. Por exemplo, para a coluna que mostra o preço da negociação:

```
<p:column sortBy="#{negociacao.preco}" headerText="Preço" >  
    <h:outputText value="#{negociacao.preco}" />  
</p:column>
```

Repare que usamos a *expression language* `#{negociacao.preco}` do JSF dentro do `sortBy` para definir o valor a ordenar.

6. Como estamos permitindo a ordenação das colunas da nossa tabela, temos que colocar nossa `p:dataTable` dentro de um `h:form`:

```
<h:form id="listaNegociacao">  
    <p:dataTable var="negociacao" value="#{argotumBean.negociacoes}">  
  
        <p:column sortBy="#{negociacao.preco}" headerText="Preço" >  
            <h:outputText value="#{negociacao.preco}" />  
        </p:column>  
  
        <!-- outras colunas omitidas -->  
    </p:dataTable>  
</h:form>
```

Salve a página e veja o resultado recarregando a página (F5) no seu navegador.

h:form sempre usará HTTP POST

É importante saber que diferente da tag `form` no HTML, o `h:form` sempre envia uma requisição HTTP do tipo *POST*. Ele nem nos dá a possibilidade de escolher usar requisições *GET*.

Isso ocorre porque o JSF tenta abstrair o mundo HTTP e assim fica mais perto do desenvolvimento Desktop tradicional. Ele esconde do desenvolvedor o fato de que uma URL está sendo chamada. Em vez disso, para o desenvolvedor, é como se botões efetivamente chamassem métodos ou eventos dentro de um *Managed Bean*.

A decisão automática pelo POST foi a forma encontrada para abstrair o HTTP.

CAPÍTULO ANTERIOR:

[Acessando um Web Service](#)

PRÓXIMO CAPÍTULO:

[Refatoração: os Indicadores da bolsa](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter