05

Introdução ao JSF e Primefaces

Introdução ao JSF e Primefaces

Durante muitos anos, a grande maioria das aplicações eram todas as Desktop. Utilizavam-se tecnologias como Delphi ou o Swing do Java para criar estas interfaces, baseando em componentes prontos que as plataformas ofereciam 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.

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.

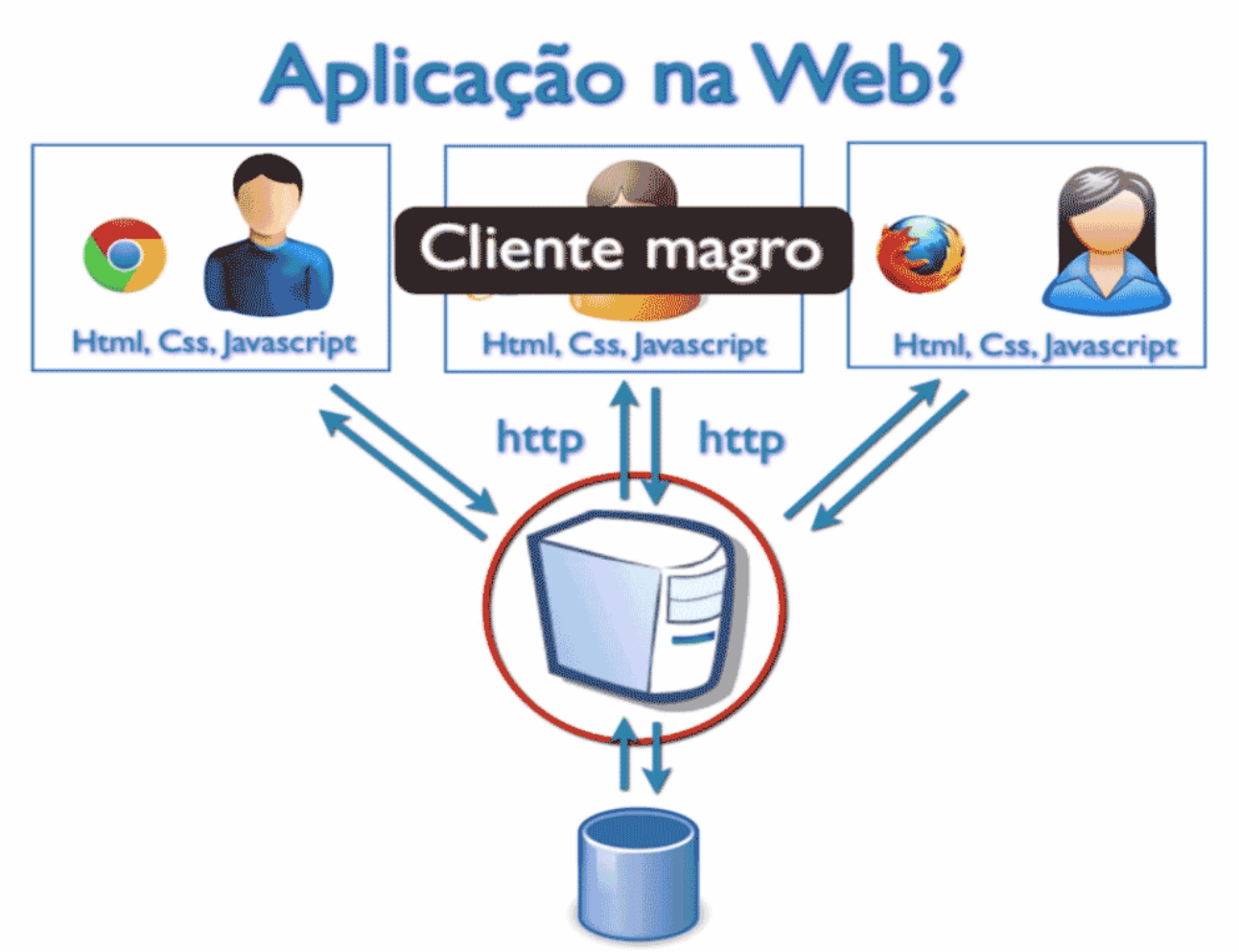
Para piorar, as regras de negócio rodam no computador do usuário. Isso faz com que seja muito mais difícil depurar a aplicação, já que não costumamos ter acesso tão fácil à maquina onde a aplicação está instalada. Em geral, enfrentamos **problemas de manutenção e gerenciabilidade**.

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.

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.

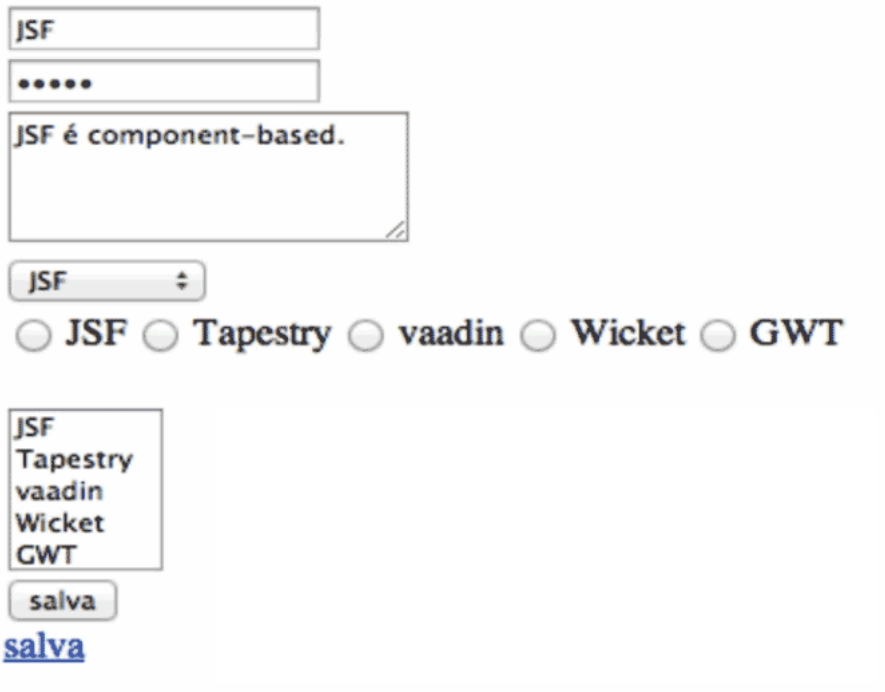
Especificação: várias implementações

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 https://javaserverfaces.java.net/ (https://javaserverfaces.java.net/).

Primeiros passos com JSF

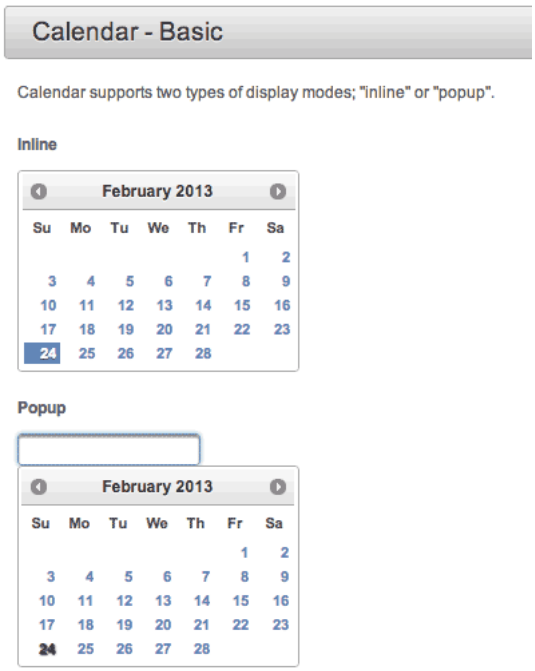
Nosso projeto utilizará a implementação Mojarra do JSF. Ela já define o modelo de desenvolvimento e oferece alguns componentes bem básicos. Nada além de `inputs`, botões e `ComboBoxes` simples.

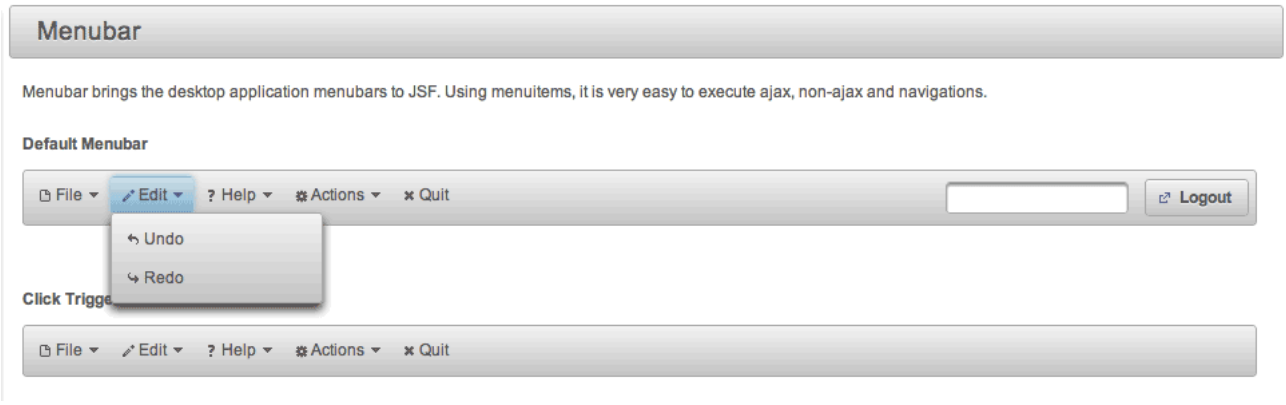


Para atender a demanda dos desenvolvedores por componentes mais sofisticados, há várias extensões do JSF que seguem o mesmo ciclo e modelo da especificação. Exemplos dessas bibliotecas são **PrimeFaces**, **RichFaces** e **IceFaces**. Todas elas definem componentes JSF que vão muito além da especificação.

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 (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:





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

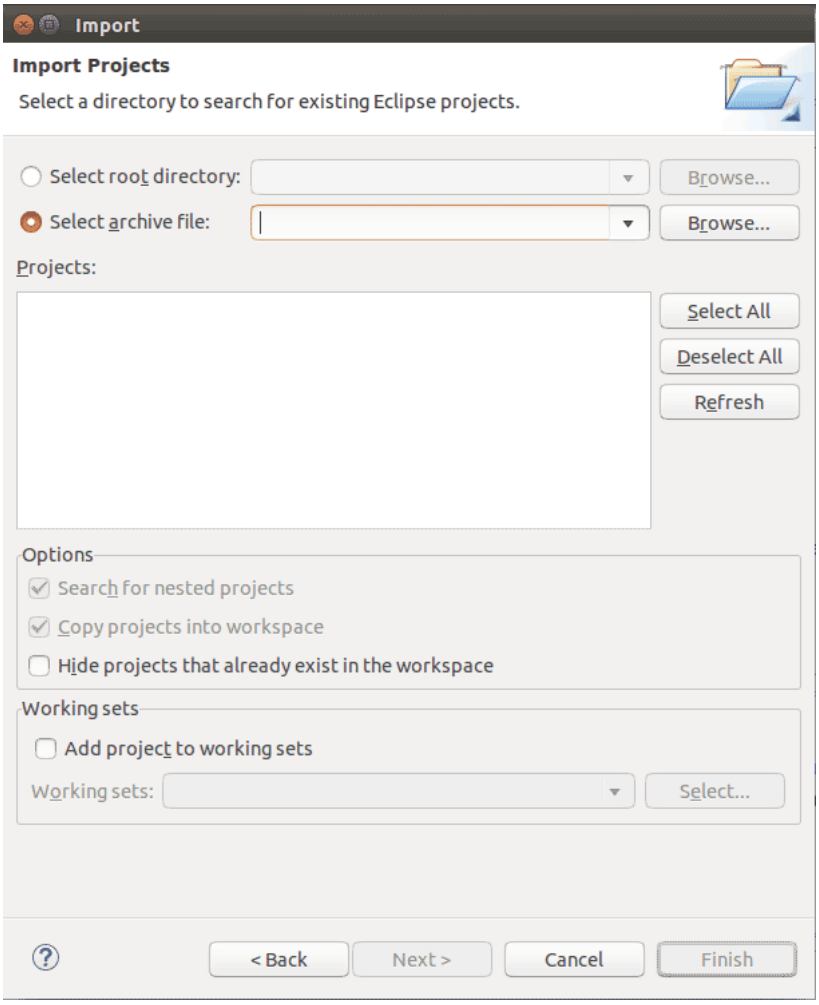
Preparação do ambiente

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 8*.

Adaptando o projeto para Web

O primeiro passo é modificar o nosso projeto do módulo anterior para um projeto Web, visto que ele era um projeto Java tradicional.Se você já tem o projeto argentum **final** do módulo anterior no seu Eclipse pode pular esta etapa, mas caso não tenha vamos importá-lo:

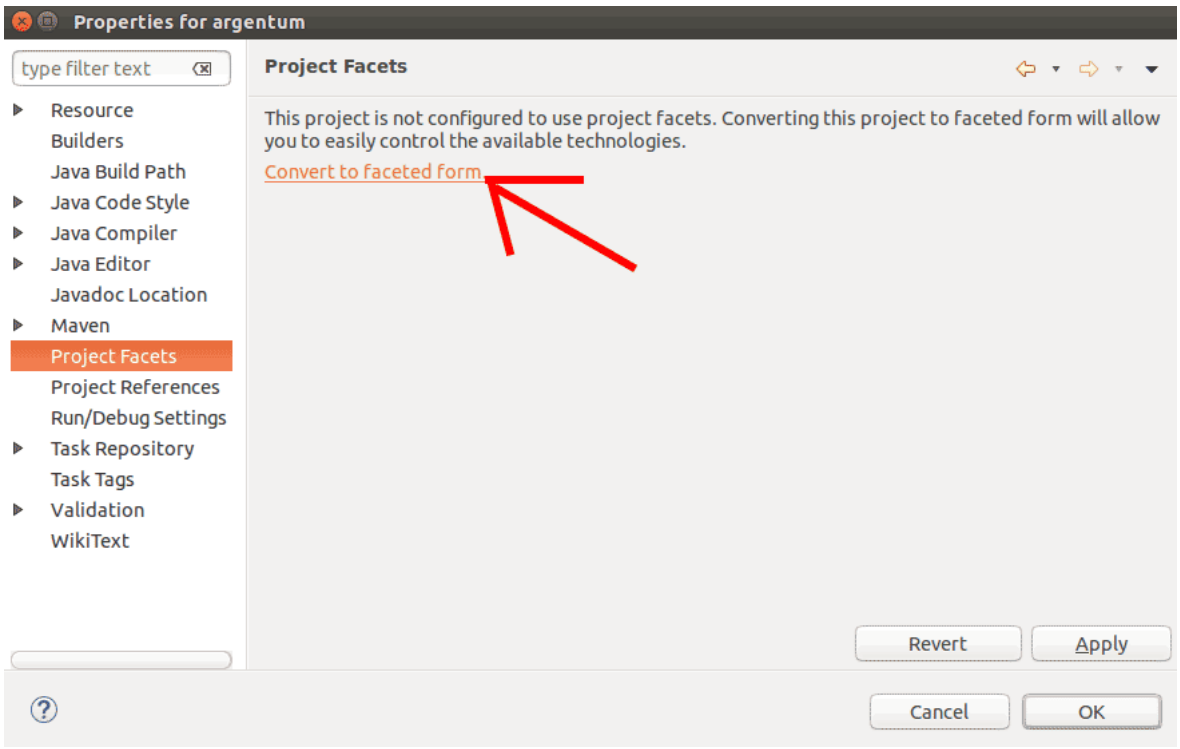
1- Faça o download (https://s3.amazonaws.com/caelum-online-public/laboratorio-java/stages/projeto-final.zip) do projeto **final** da primeira parte do treinamento laboratório java 1. 2- Abra seu ecplise e vá em *File > Import... > Existing Projects into Workspace* .



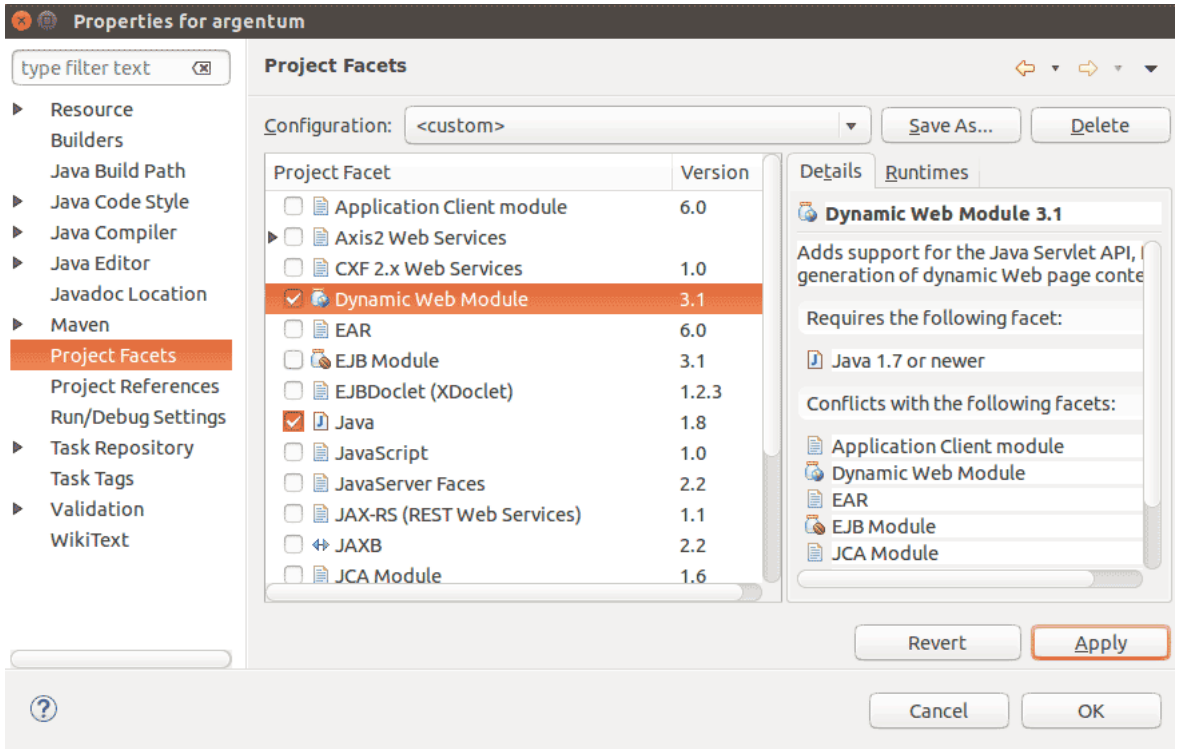
3- Com a opção *Select archive file:* marcada, clique no botão *Browse...* e selecione o **.zip** baixado no passo anterior. 4- Clique em **Finish** para terminar a importação.

Agora para converte-lo em um projeto Web e habilitar o JSF , siga os seguintes passos:

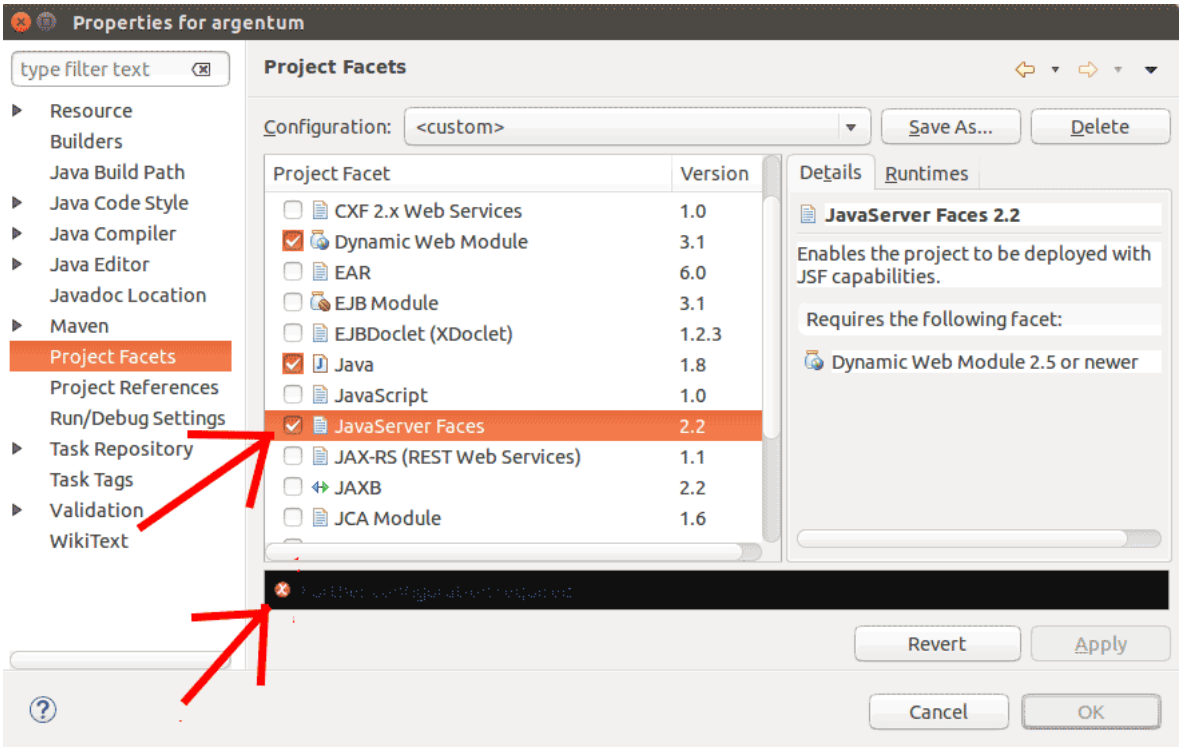
1- Clique com o botão direito em cima do nome do projeto e vá em propriedades, que é normalmente a última opção. 2- Procure no menu lateral pela opção *Java Facets*.3- Clique em *Convert to faceted form....*



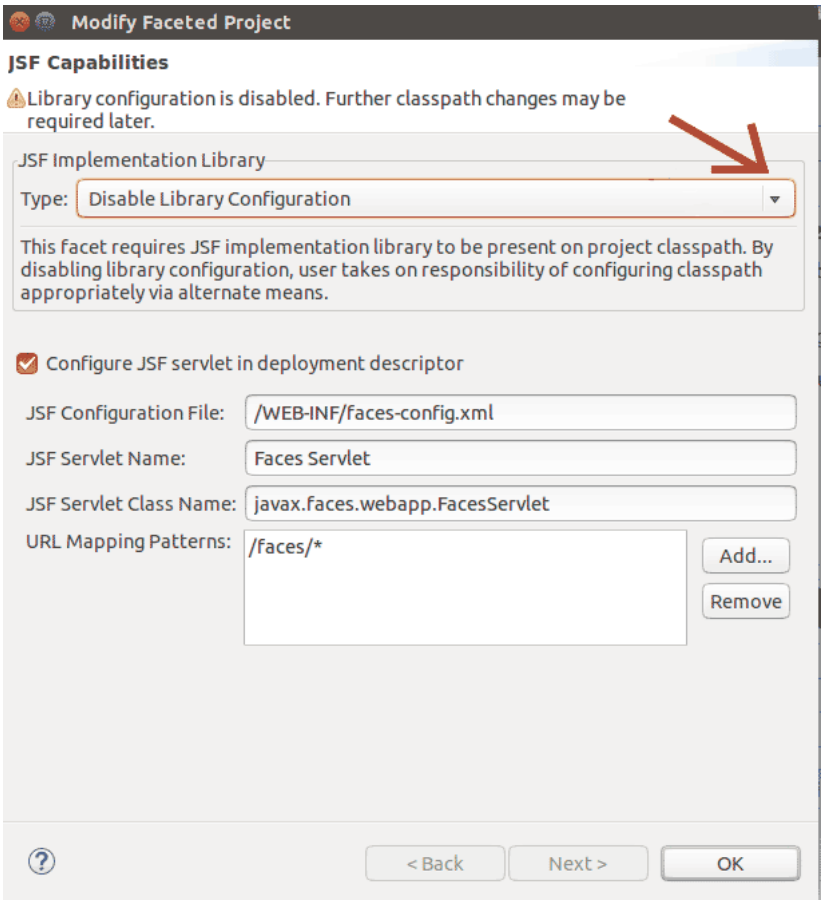
4- Habilite a caixa *Dynamic Web Module* com a versão **3.1**.



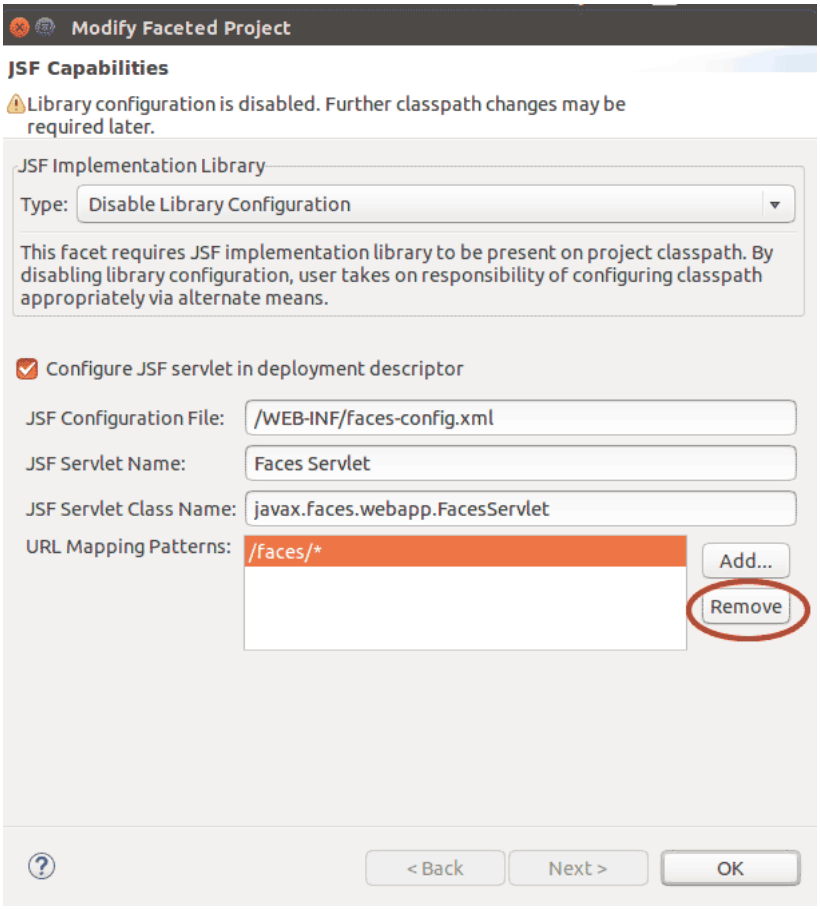
5- Clique no botão **Apply**. 6- Agora habilite a caixa *Java Server Faces* com a versão **2.2**. 7- Ele mostrará um campo preto com a opção *Further configuration required...*, clique nesta opção.



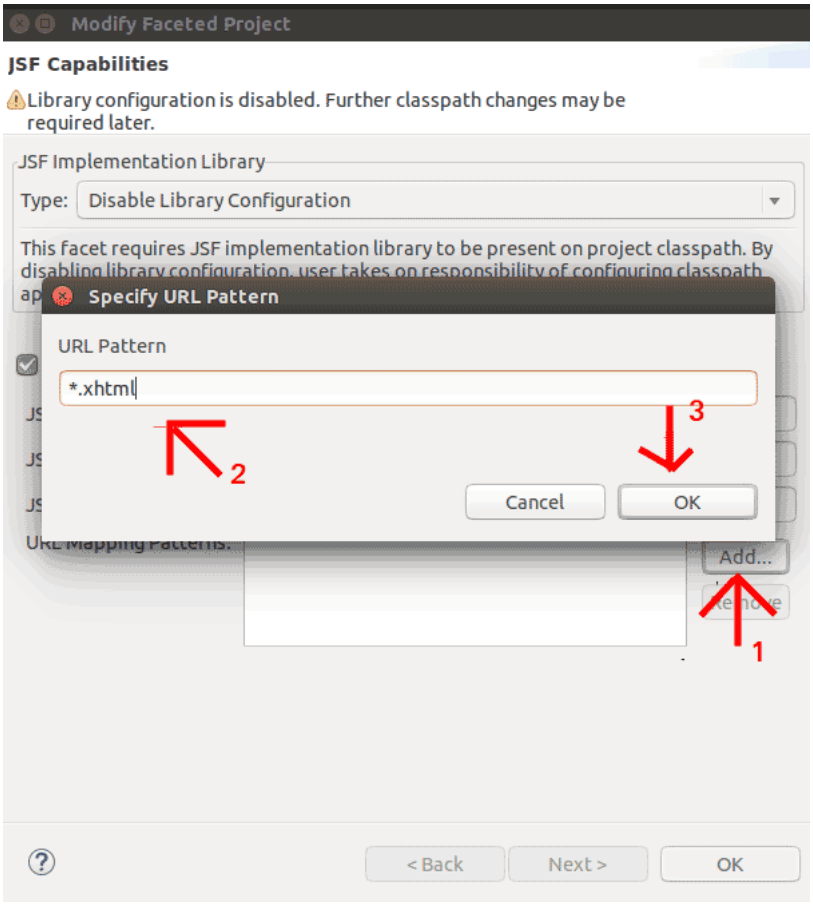
8- Na nova janela que se abriu, faremos algumas configurações. Primeiramente clique na caixa de seleção aonde está escrito *Type: User Library* e troque por Type: *Disable Library Configuration*



9- Agora na parte de baixo desta mesma janela, aonde encontramos a caixa com o nome de *Url Mapping Patterns* , marque a opção */faces/** e clique em *Remove*:



10- Em seguida clique em add, e adicione o mapeanto para : **.xhtml** .



11- De OK para finalizar, e OK na janela de propriedades do projeto.

Instalando o Tomcat

O próximo passo é instalar o Tomcat. Usaremos a versão 8.x:

1- Vá no site do tomcat (http://tomcat.apache.org/download-80.cgi) e faça o download do **.zip** adequado para o **seu sistema operacional**.

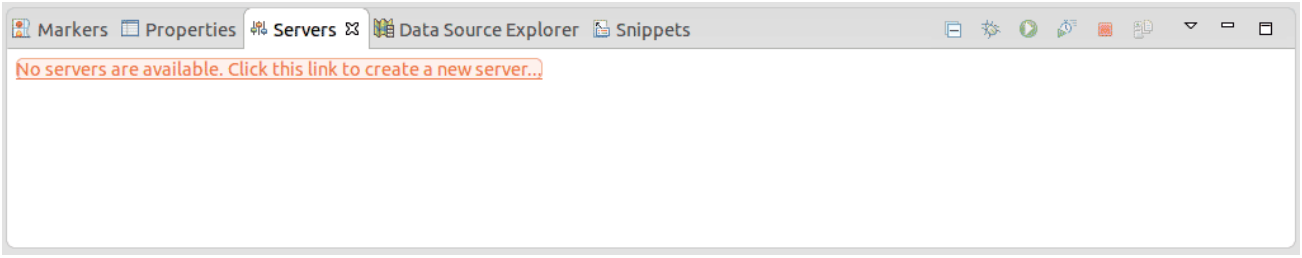
2- Extraia o arquivo zip do TomCat e coloque-o na sua pasta de preferência.

3- 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.

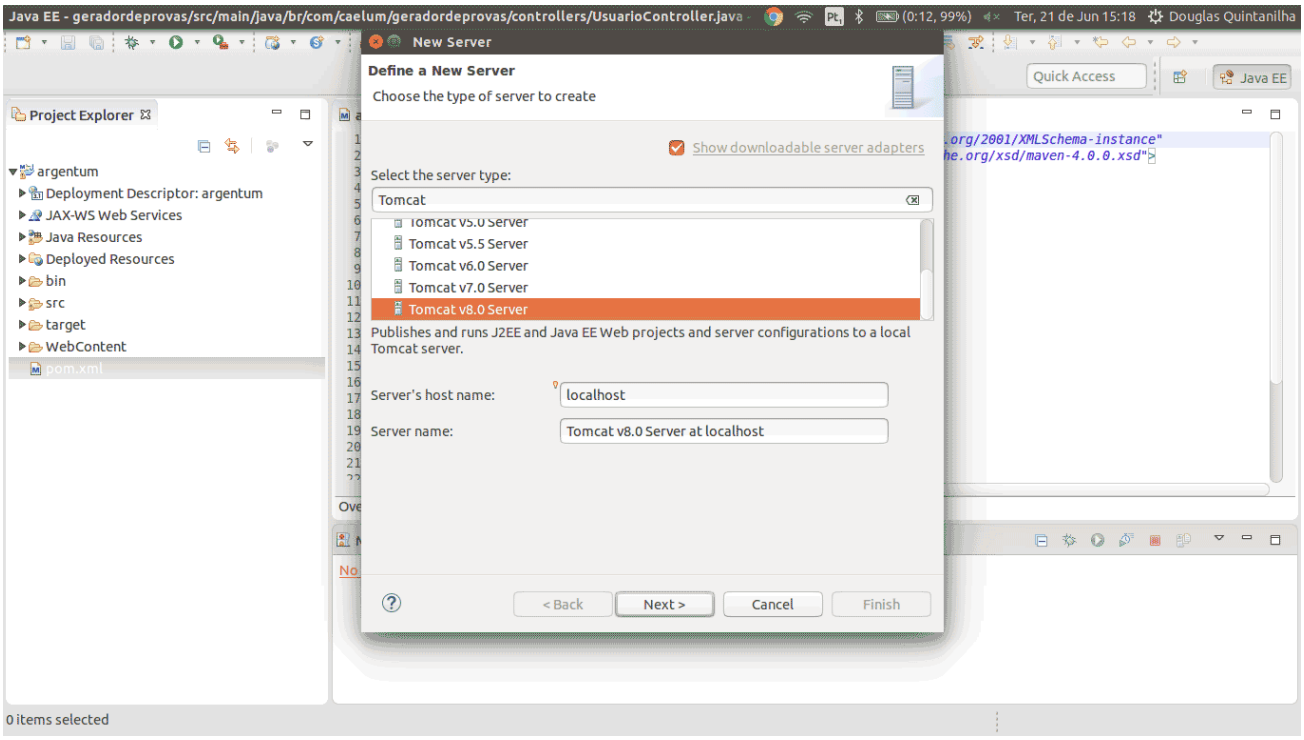
Configurando o Tomcat

Agora precisamos configurar o Tomcat no Eclipse, para que possamos controlá-lo mais facilmente.

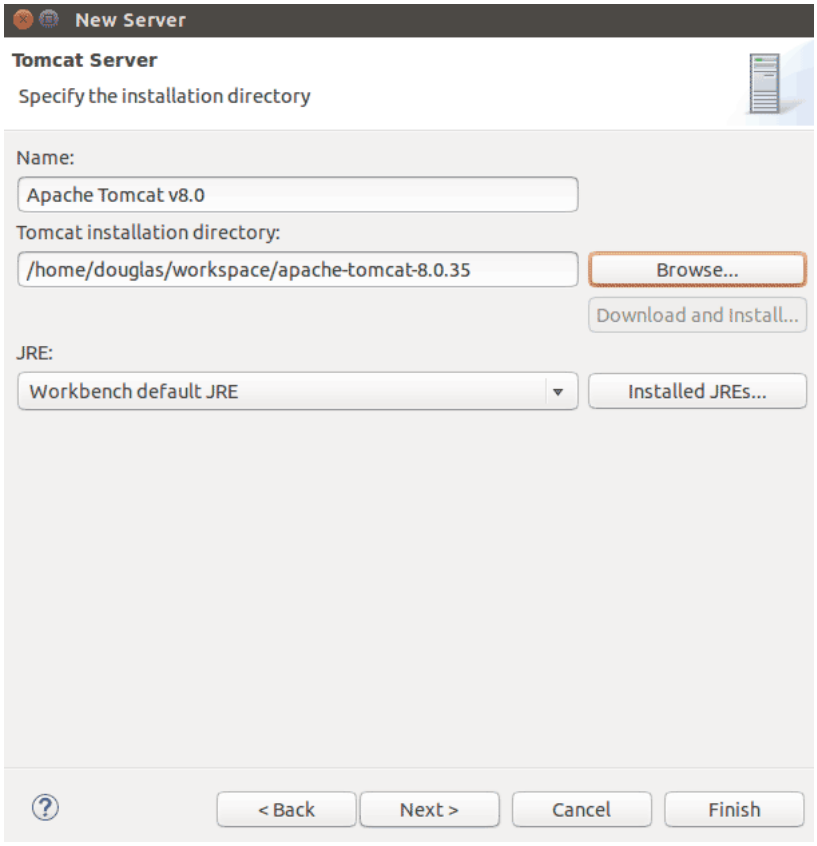
1- Dentro da aba *Servers* clique com o botão direito do mouse e escolha *New -> Server*. Se não quiser usar o mouse, você pode fazer **ctrl+3** *New server*.



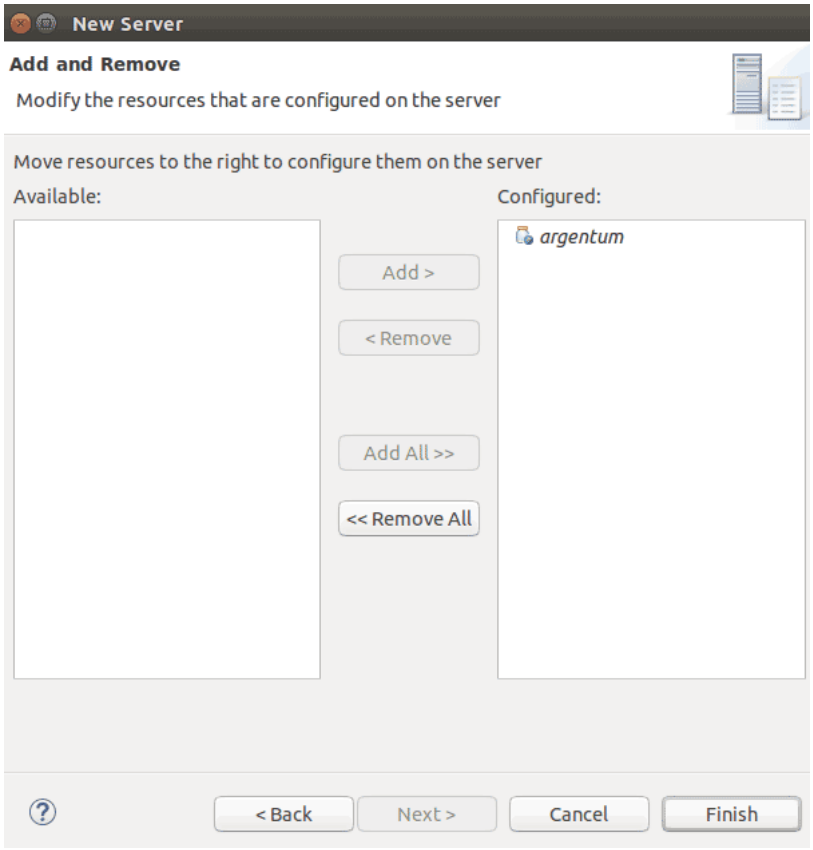
2- Dentro da Janela *New Server* escolha *Apache Tomcat v8.0 Server* e clique em *Next*.



3- 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*.



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



5- Clique em *Finish*.

Adicionando as depêndencias do JSF no Maven

Agora vamos adicionar algumas novas depêndencias ao Maven, para que o nosso JSF e o Primefaces funcionem corretamente. 1- Abra o seu *pom.xml*.

2- Lá, dentro de `<dependecies>` , adicione as seguintes depêndencias:

```
<dependecies>
    <!-- ...outras dependencias -->

    <dependency>
        <groupId>com.sun.faces</groupId>
        <artifactId>jsf-api</artifactId>
        <version>2.2.13</version>
    </dependency>
    <dependency>
        <groupId>com.sun.faces</groupId>
        <artifactId>jsf-impl</artifactId>
        <version>2.2.13</version>
    </dependency>
    <dependency>
        <groupId>org.primefaces</groupId>
        <artifactId>primefaces</artifactId>
        <version>6.0</version>
    </dependency>
<dependecies>
```

3- Salve seu *pom.xml*

Por último, vamos configurar o projeto para que ele use as depêndencias do Maven corretamente.

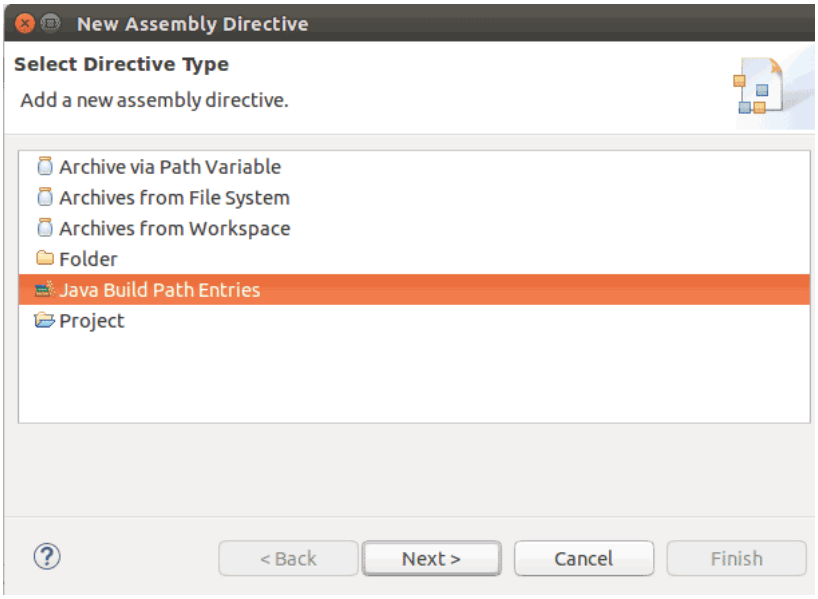
1- Acesse novamente as propriedades do seu projeto, clicando com o botão direito em cima dele e selecionando a opção *Properties*.

2- Vá na opção *Java Build Path*.

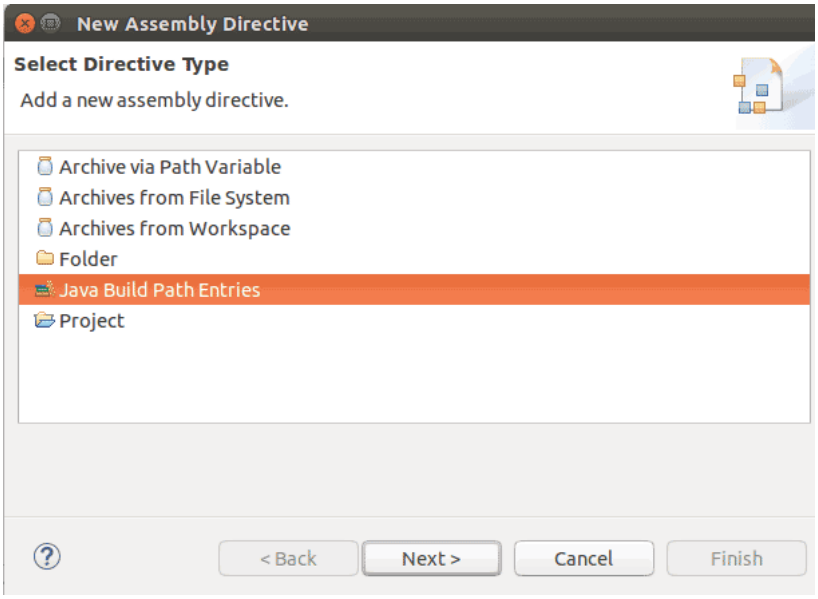
3- Habilite a caixa *Maven Dependecies* e clique em *Apply*.

4- Agora acesse a opção *Deployment Assembly* , e clique em *Add*.

5- Selecione a opção *Java Build Path Entries*



6- Por último, selecione a opção *Maven Dependecies* e clique em *Finish*.



7- Clique em apply e feche as opções do projeto.

Agora nosso projeto está convertido para um projeto Java Web, com o JSF habiltado e com o Maven funcionando corretamente.

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 anteriormente em Java para, 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 (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>
```

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 (*Plain Simple Java Objects*) anotados com `@ManagedBean`.

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() {
        LocalDateTime agora = LocalDateTime.now();
        SimpleDateFormat sdf = new SimpleDateFormat("hh:mm:ss");
        return "Atualizado em " + sdf.format(agora.getDate().getTime());
```

```
    }
}
```

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*.

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.

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`.

Paginação e ordenação

O componente `p:dataTable` sabe listar items, 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="#{argentumBean.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 `ArgentumBean` é recriado e perdemos a lista anterior. Assim, na criação da nova instância de `ArgentumBean`, 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 sí, 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 implementes 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 diversar 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>
```

O que aprendemos neste capítulo:

- A diferença entre o desenvolvimento Web e Desktop.
- As vantagens e desvantagens entre Web e Desktop.
- O que é a especificação JSF.
- O que é um framework component based.
- A adaptar um projeto Java tradicional para um projeto Java Web no Eclipse.
- Como instalar e configurar um servidor no Eclipse.
- A configurar o Maven para que ele funcione corretamente com o Tomcat.
- Criando um hello world com JSF.
- Como importar e utilizar as taglibs do JSF.
- Os componente h:outputText, h:inputText e h:commandButton.
- A expression language do JSF.
- O que é um Managed Bean.
- O componente p:dataTable do Primefaces.
- Como exibir a tabela de negociações
- Como fazer a ordenação e paginação em uma tabela.