Tutorial JAVA Web com JSF 2.0, Facelts, Hibernate 3.5 com JPA 2.0, Spring 3.0 e PrimeFaces 2.2.1

Autor: Diego Carlos Rezende

Graduando em Sistemas de Informação

Universidade Estadual de Goiás - UEG UnUCET

Anápolis

Outubro, 2011

Sumário

Figuras

Tabelas

Apresentação

Com este tutorial pretendo explicar o funcionamento de um CRUD básico para cadastro de Reino e Filo com apenas o nome dos mesmos em um programa para a Web envolvendo o JSF 2.0 para a camada de visão, Facelets para organização dos templates e das páginas, Hibernate para a camada de persistência juntamente com o JPA, Spring para a camada de controle (injeção de dependências) e o PrimeFaces como interface rica para a camada de visão. Contém também um validador usando o Java Reflection.

Para explicação de Spring e um pouco do Hibernate usarei tutoriais de outros autores que julgo muito bons, sendo que citarei quando necessário e onde deverá ser feito a leitura dos mesmos.

Na construção da aplicação, além dos frameworks citadas acima, também foram usadas as seguintes tecnologias: Eclipse Helios para a IDE (Helios ou Indigo) para desenvolvimento Java EE; Apache Tomcat 7 como servidor; SQL Server 2005 como banco de dados.

O intuito deste tutorial é a disseminação do conhecimento, apenas peço para manter os créditos dos autores, tantos deste tutorial, como dos citados aqui também.

Para um melhor aproveitamento, tente não copiar o código, use as ajudas do Eclipse, as dicas contidas no tutorial e o perfeito CTRL + "espaço" que a IDE nos fornece. Nos arquivos XML aconselho o CTRL + C, CTRL + V mesmo.

A explicação do Spring e Hibernate encontra-se no blog do Felipe Saab, que explica muito bem sobre os dois, e para a compreensão deste tutorial faz-se necessário a leitura do artigo dele, de Spring à Hibernate (parte 1 ao 4),

Configuração do Banco de Dados

No seu SQL Server já instalado, abra-o e crie a database para o sistema:

```
01. CREATE DATABASE portal_virtual;
```

Tabela 01 - Criando banco de dados

Agora é hora de criar a tabelas onde iremos salvar os dados:

```
01. USE portal_virtual;
02.
03. CREATE TABLE reino (
04.     id_reino INT IDENTITY NOT NULL,
05.     nome VARCHAR(50) NOT NULL,
06.     PRIMARY KEY (id_reino)
07. )
08.
09. CREATE TABLE filo (
10.     id_filo INT IDENTITY NOT NULL,
11.     nome VARCHAR(50) NOT NULL,
12.     id_reino INT NOT NULL,
13.     PRIMARY KEY (id_filo),
14.     FOREIGN KEY (id_reino) REFERENCES reino(id_reino)
15. );
```

Tabela 02 - Criando tabelas

Pronto, basta apenas criar o Usuário de acesso. No projeto defini o usuário para o banco de dados "portal_virtual" com o login: ueg e senha: portueg; você pode, então, criar o mesmo usuário com login e senha no seu SQL Server, ou mudar o usuário e a senha do software no arquivo properties do JDBC para o usuário do seu banco de dados, explicarei como fazer essa ultima parte mais adiante.

Banco de dados criado, vamos para o JAVA.

Configurando o TomCat

Baixe o Eclipse no site citado na Apresentação e o descompacte onde desejar, depois baixe o Apache TomCat em zip ou instalador, descompacte-o em uma pasta a sua escolha. Para abrir o Eclipse é necessário ter o Java JDK instalado e configurado as variáveis de ambientes.

Vamos criar o Server para rodar a aplicação WEB: no Eclipse, clique com o botão direito e vá em: New>Other...e procure por Server, depois clique em "Next":

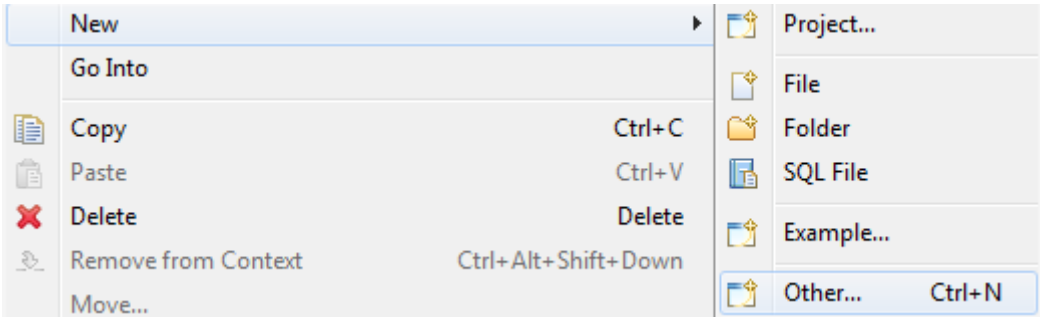


Figura 01 - New Others

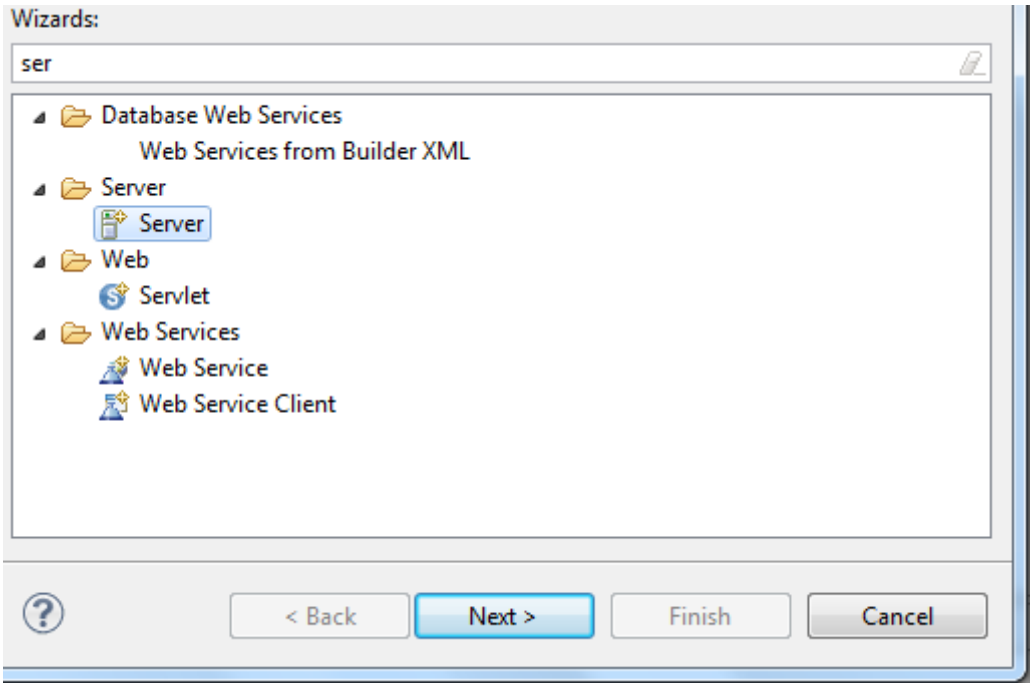


Figura 02 - Criando Server

Na próxima aba selecione o Apache/Tomcat v7.0 Server:

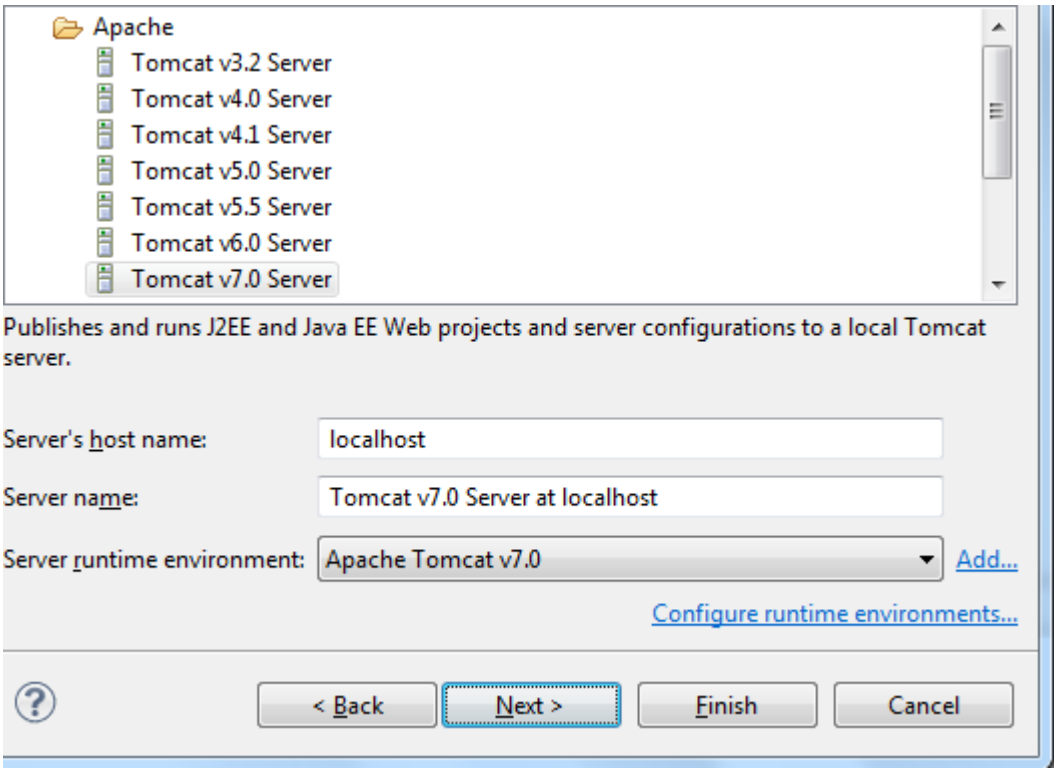


Figura 03 - Selecionando Server

Clique em "Next" e o "Wizard" pedirá a pasta onde o TomCat foi instalado: Clique em "Browse..." e selecione a pasta onde descompactou o arquivo:

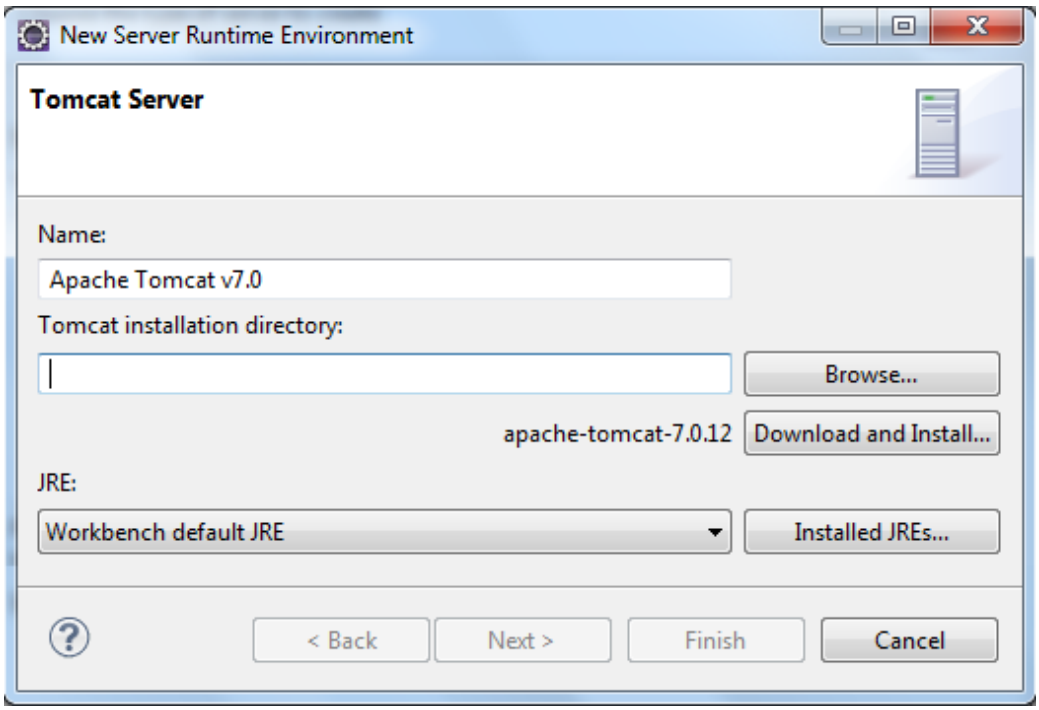


Figura 04 - Escolha do diretório do TomCat

Cliquem em "Finish" e agora você tem seu servidor de aplicação para rodar seu projeto localmente. Vamos definir então onde ficará os arquivos temporários do projeto para teste: Na aba de "Server" clique duas vezes no seu novo Servidor - "Tomcat v7.0 Server at localhost" - e em "Server Locations" clique na segunda opção - "Use tomcat installation" - salve e feche a aba:

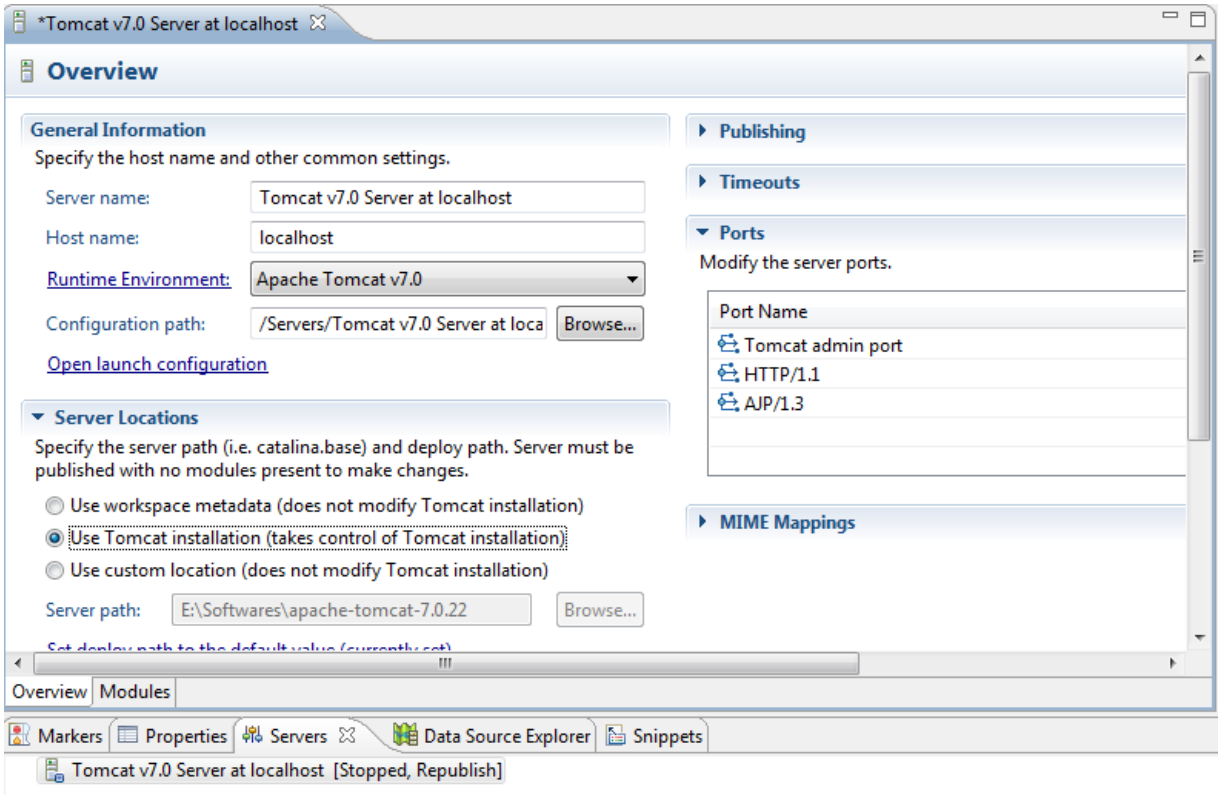


Figura 05 - Configuração do Server Location

Seu Tomcat já está funcionando, se quiser testá-lo, de Start (botão "Play" um pouco acima do nome do Tomcat v7.0 da figura) e tente abrir a página localhost:8080, se abrir uma página do Tomcat, está tudo ok.

Criando o projeto

Em seu Eclipse com o Tomcat rodando, crie seu projeto web: botão direito > New > Other... (Figura 01) e procure por "Dynamic Web Project":

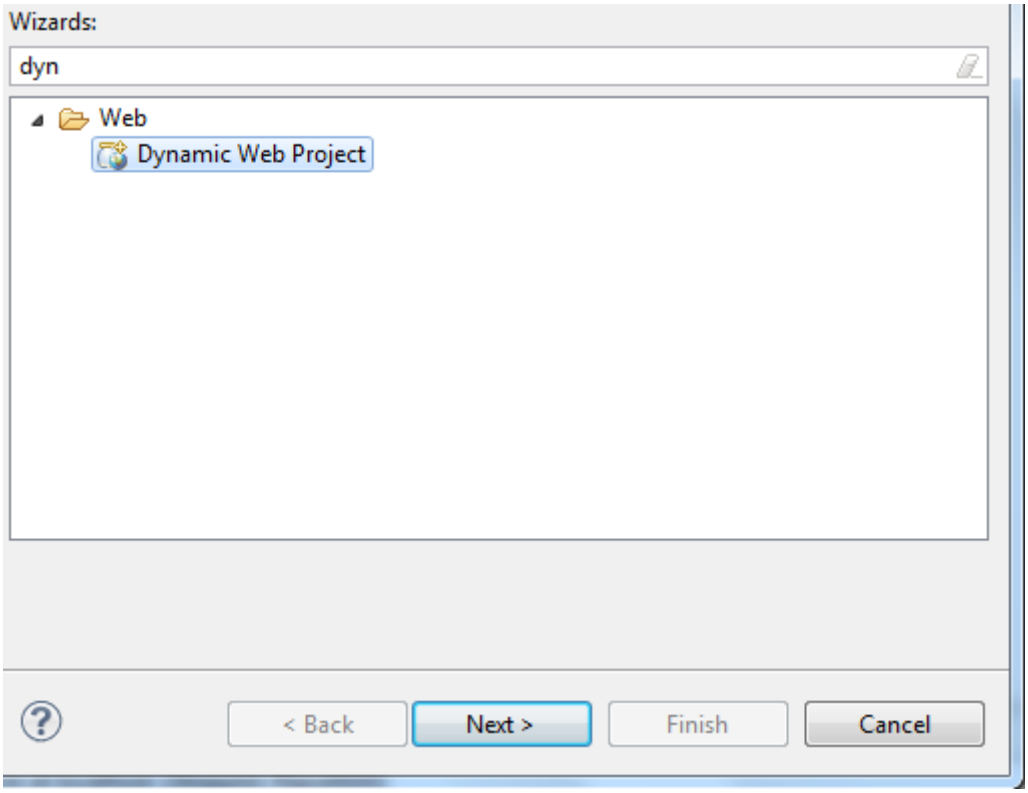


Figura 06 - Criando um Dynamic Web Project

Clique em "Next" e será pedido o nome do projeto e algumas informações da configuração do mesmo. Usaremos o "Wizard" do Eclipse para nos ajudar a fornecer algumas coisas prontas, então para isso, digite o nome projeto e selecione em "Configuration" a opção "JavaServer Faces v2.0 Project", senão aparecer a opção verifique a versão do módulo se está em 3.0. Com essa simples mudança o Eclipse criará o arquivo faces-config.xml automaticamente que é necessário para fazer algumas configurações do JSF. Só para informação, na versão do JSF 1.2 e anteriores, todas as configurações eram feitas nesse arquivo XML, a partir da versão 2.0 começou-se a usar annotation, não sendo necessário mais o uso do faces-config, entretando, ainda tem uma configuração que não conseguir encontrar a annotation da mesma, o que me força a usar este arquivo ainda.

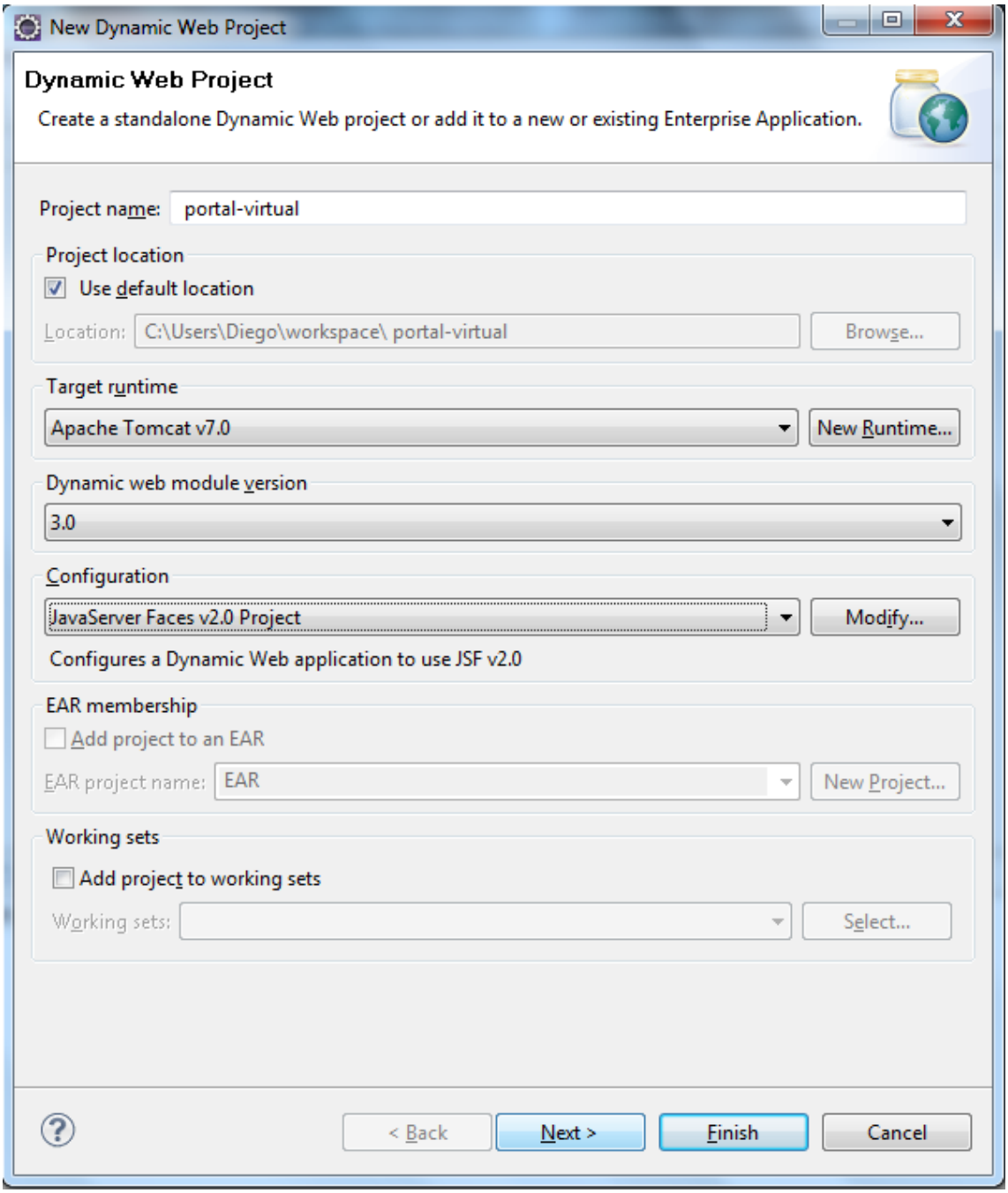


Figura 07 - Configuração do projeto

Defini o nome do meu projeto como "portal-virtual" e essa será a forma a qual irei me referir durante todo o tutorial, para comandos de acesso e configurações de pacotes e arquivos mude para o nome que você escolheu, caso não seja o mesmo.

Clique em "Next", na próxima aba será perguntando onde será salvo os arquivos Java e onde será salvo o .class no Tomcat, simplesmente mantenha a configuração clicando em "Next". Na outra aba será perguntando a pasta de deploy, onde ficará todas as páginas webs, pastas

criadas e .class para rodar a aplicação. Nesta parte também mantenha as configurações (mudanças no Context root irá mudar a forma de acessar a página web), apenas peça para o "Wizard" gerar o deployment do web.xml automaticamente, assim o Eclipse irá gerar a lista de arquivos para inicialização sozinho:

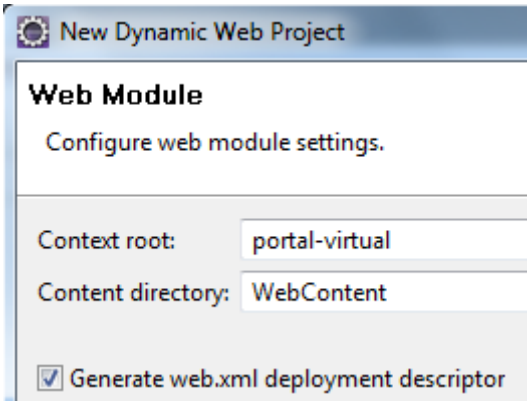


Figura 08 - Gerar web.xml

Vá para a próxima aba clicando em "Next". Nesta parte o "Wizard" te ajuda a configurar o JSF 2.0 já que selecionou que existirá o framework, para isso ele pede as bibliotecas do JavaServer Faces e como será feito o acesso e chamada da servlet. Você pode escolher baixar as bibliotecas do JSF pelo Eclipse ou que o Wizard não configure esta etapa, para isto selecione: "Disable Library Configuration".

Como as bibliotecas se encontram no CODE desabilite a opção de configuração da biblioteca. Logo abaixo tem a configuração de mapeamento da servlet do JSF, por padrão o acesso a qualquer página JSF vem configurado, pelo Eclipse, como /faces/*.

Vamos com um pouco de explicações... JSF é uma tecnologia para a construção da interface do usuário, não passa de uma servlet de visão, que traz HTML e um pouco de JSP embutidos, o que você faria em algumas linhas de código tendo que saber todas as tecnologias citadas, você faz em uma no JSF. Mas para uma página contendo o código do framework funcionar, você deve adicionar a biblioteca do JSF para o projeto e dizer para o servidor que a página precisa passar pela servlet do JSF para uma "renderização", ou seja, ela terá que ser reescrita de forma que o navegador a reconheça (voltar as várias linhas de HTML). E como fazer isso?

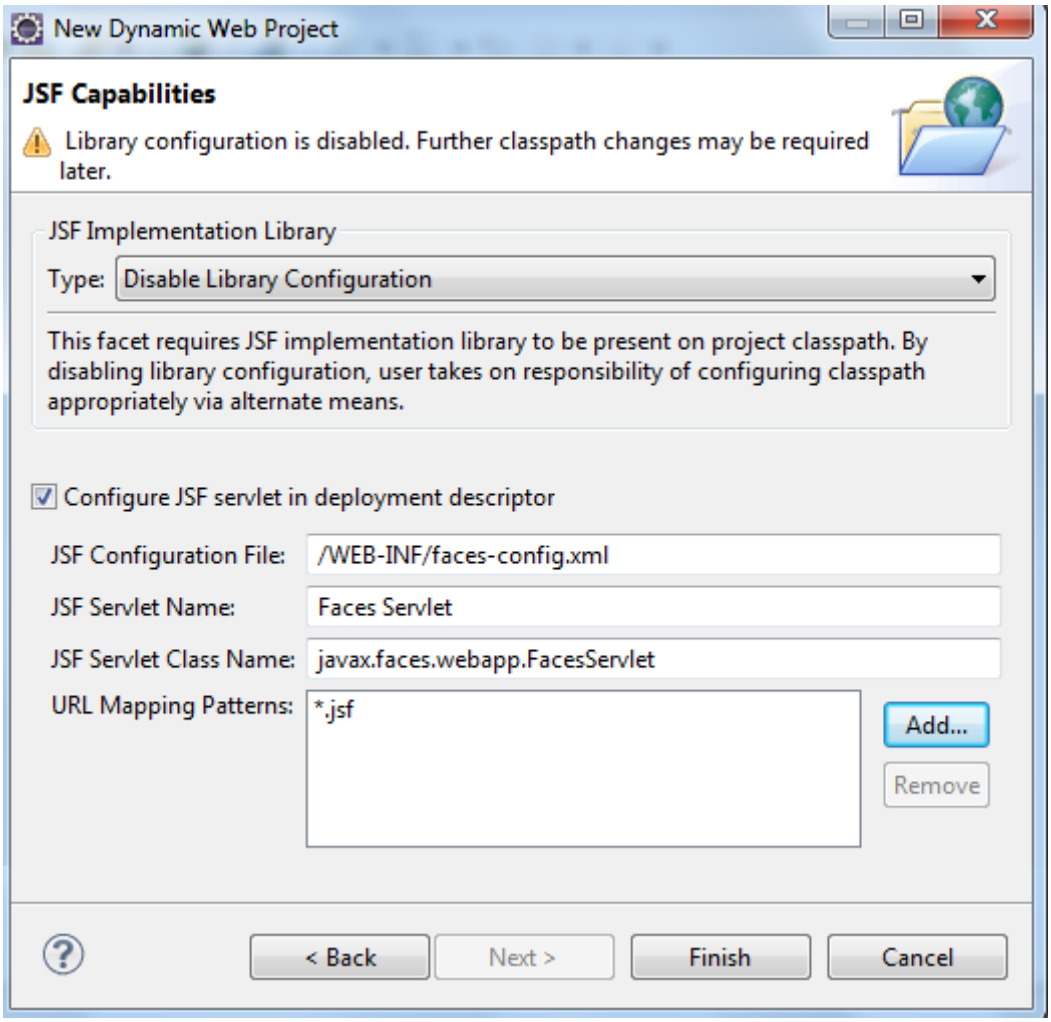


Figura 09 - Configuração do JSF no web.xml

Essa configuração da imagem fica no web.xml, mas o Eclipse nos ajuda, antes mesmo de criar o projeto, a fazê-la. Como informei acima, precisarmos de uma forma para avisar que uma página usa o JSF, e isso é feita na hora de abrir a página pela URL, a configuração que eu fiz diz que toda vez que eu digitar uma página na URL e ela terminar com .jsf, essa página deve passar pela servlet do JSF e renderizá-la. Exemplo, se tenho uma página na aplicação que o nome dela é HelloWorld.xhtml, mas para acessar ela digito: www.testejava.com.br/HelloWorld.jsf, e no web.xml tem a configuração citada, o servidor sabe que a página contém JSF e irá abri-la de forma a reconhecer todas linhas contidas no código. Se não entender bem o que foi dito agora, mais pra frente isso ficará mais claro.

Clique em "Finish" e teremos nosso projeto criado (Caso abra uma mensagem pedindo para muda de perspectiva para JAVA Web, clique em Yes). Vamos agora adicionar as bibliotecas para o projeto funcionar.

Configurando o projeto

Em seu projeto, vá em WebContent/WEB-INF/lib, copie e cole todos .jar que se encontrar na pasta libs do CODE, há uma divisão de pasta nos sites, copie todas as bibliotecas de todas as pasta para a pasta /lib do seu projeto sem criar nova pasta. Ficará mais ou menos assim:

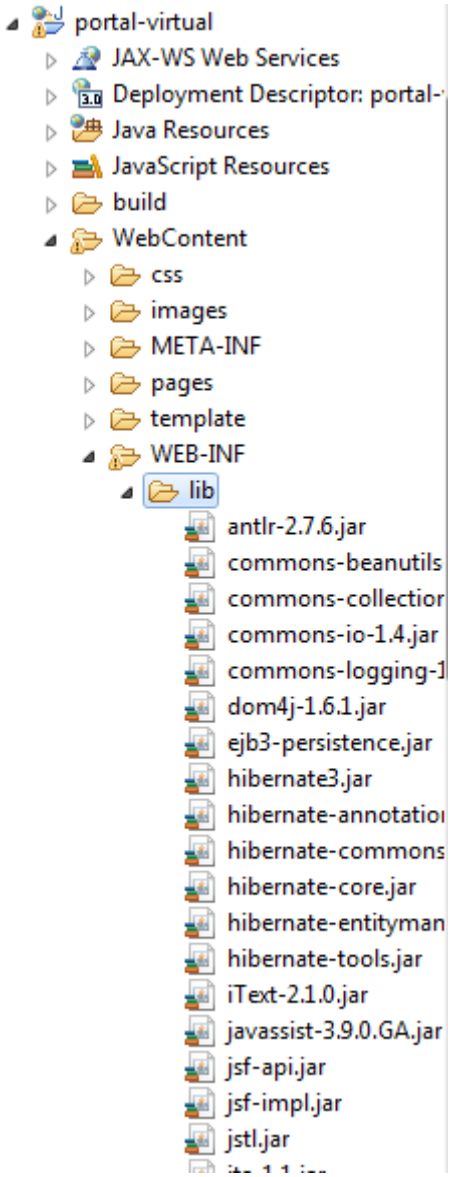


Figura 10 - Bibliotecas

Conforme figura ao lado também, crie as pastas "css", "images", "pages" e "template" dentro do diretório /WebContent (para isso clique com botão direito, New->Folder ou New->Other...->Folder).

Bibliotecas criadas, vamos habilitar os frameworks e visualizar no que o Eclipse já nos ajudou. Dentro do diretório /WEB-INF temos o arquivo web.xml, abra-o. Se for a primeira vez que abre um arquivo XML, ele deve abrir no modo Design, clique no modo "Source" para visualizar o código-fonte (fica acima da barra de Markers/Servers/Console, bem no rodapé do web.xml aberto no Eclipse). Segue abaixo como o web.xml deve ficar, logo depois explico o código:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID"
version="3.0">
  <display-name>portal-virtual</display-name>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
```

```
  <!-- Configuração da servlet do JSF -->
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>

  <!-- Configuração da servlet do primeface -->
  <servlet>
      <servlet-name>Resource Servlet</servlet-name>
      <servlet-class>org.primefaces.resource.ResourceServlet</servlet-
class>
  </servlet>
  <servlet-mapping>
      <servlet-name>Resource Servlet</servlet-name>
      <url-pattern>/primefaces_resource/*</url-pattern>
  </servlet-mapping>
  <!-- Configuração do template ou skin do primefaces -->
  <context-param>
      <param-name>primefaces.THEME</param-name>
      <param-value>redmond</param-value>
  </context-param>

</web-app>
```

Tabela 03 - web.xml

Arquivos XML se iniciam com a tag <?xml /> para a definição da versão do xml e o tipo de codificação (UTF-8), depois vem a tag falando que aquilo é um arquivo do tipo web.xml <web-app>, esse arquivo, principalmente o que já citei, é mais CTRL C + CTRL V, não há necessidade de decorar isso, apenas entender como funciona.

O display-name é apenas o nome do projeto, não faz interferência na aplicação. O wellcome-file-list (gerado porque foi marcado a opção generate deployment citado na criação do projeto) é outro elemento opcional, inicialmente ele vem uma lista de index.algumaExtensão, que irá buscar no /WebContext os arquivos em ordem de listagem e tentará executá-los quando você tentar acessar o site, exemplificando: o projeto executará localmente, então para acessar ele você digita no browser: localhost:8080/portal-virtual, tendo o arquivo do wellcome-file-list no /WebContext o browser tentará abrir a página, não tendo nenhum da lista citado, dará erro dizendo que não existe a aplicação, mesmo que tenha um arquivo (por exemplo, teste.xhtml) dentro da pasta, ele não será aberto se não foi citado na lista de entrada, para abrir a página citada não estando no

wellcome-file-list terá que ser feita do seguinte modo: localhost:8080/portal-virtual/teste.xhtml. Nesse wellcome-file-list ainda há um detalhe, não se pode usar nenhuma página que necessite de biblioteca para abrir, aceita apenas extensões XHTML, HTML, JSP, ou seja, você não pode colocar uma página .jsf como inicialização do projeto, já que é necessário usar bibliotecas para exibir a página, logo explico como solucionar o problema. Coloquei, na lista de inicialização, o arquivo index.html, pois tenho ele na pasta /WebContent que é a página inicial do projeto.

O comando servlet serve para definir as servlets existentes no projeto, dentro dela temos que definir, obrigatoriamente, seu nome e a classe onde deve ser buscada para execução da mesma. A primeira que temos é a Faces Servlet, sua classe de compilação fica na biblioteca do JSF que adicionamos ao projeto no caminho citado na tabela. Logo depois vem o mapeamento da servlet, ou seja, onde iremos usá-la, primeiro dizemos qual a servlet que queremos mapear, no nosso caso a Faces Servlet que acabamos de criar, e depois dizemos a url que ela deve ser chamada, no caso *.jsf. Volto na explicação que disse acima sobre JSF, quando você pedir no browser alguma página da aplicação e você diz que sua extensão é .jsf a página irá passar pela classe javax.faces.webapp.FacesServlet e irá renderizar o que aparecerá para o usuário, exemplificando: tenho um arquivo chamado teste.xhtml, dentro dele tenho comandos do JSF, quando eu tentar acessá-la, se apenas dizer que ela é uma página simples: localhost:8080/portal-virtual/teste.xhtml, provavelmente irá abrir uma página aparecendo apenas os comandos HTML utilizados e o resto em branco, ou dará um erro, já se for dito na URL que tem que passar pela servlet do JSF: localhost:8080/portal-virtual/teste.jsf, mesmo que não exista esse arquivo .jsf, a aplicação sabe (já que foi mapeado) que isso é um comando para chamar a Faces Servlet, e irá procurar alguma página com nome de teste (independente se é .html, .xhtml) e fará sua execução pela servlet, assim irá exibir a página com todos comandos sem problemas.

A próxima servlet é a interface rica do Primefaces, dizemos onde é sua classe, definimos seu nome, no mapeamento dizemos onde pegar os recursos, e depois definimos o parâmetro do tema do mesmo (verifique que

há a biblioteca do tema no diretório /lib), a servlet não é um recurso acessível por URL como o JSF.

Com esse arquivo web.xml a aplicação irá funcionar com JSF e Primefaces. O Spring pode ser inicializado com o projeto criando sua servlet no web.xml, mas pelas pesquisas que fiz caiu um pouco em desuso esse método, no exemplo citado fiz uma inicialização diferente que dará para entender bem o Spring, a forma dele ser inicializado será apenas um detalhe que não fará diferença para o aprendizado.

Vamos criar a página de inicialização do projeto, o index.html citado no wellcome-file-list:

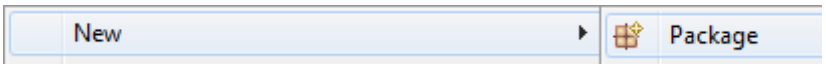
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<head>
<meta http-equiv="refresh" content=" 0 ;url=template/template.jsf" />
</head>
```

Tabela 04 - index.html

A página contém apenas um cabeçalho mandando recarregar a página e ir para a URL template/template.jsf, com isso resolvemos o problema do wellcome-file-list não poder abrir uma página JSF direto. Somente com esse cabeçalho, ao tentar acessar localhost:8080/portal-virtual, o browser fará o redirecionamento para localhost:8080/portal-virtual/template/template.jsf, como a página não existe, dará erro. Vamos agora para os códigos JAVA.

Estruturando o projeto...

Vamos criar os pacotes inicialmente e explicarei como será dividida a nossa arquitetura, clique com o botão direito sobre o nome do projeto,



New->Package ou New->Other->Package.

Figura 11 - Package

Em web, se padroniza que os nomes dos pacotes devem vir com o site ao contrário e depois o nome do pacote desejado, exemplo, quero criar o pacote Teste para meu website www.tutoriais.com.br, então meu pacote ficara: br.com.tutoriais.teste, ou se usa o nome do projeto, usando o mesmo exemplo: no meu site tutoriais tenho o projeto artigos, então meu pacote pode ser também: br.com.artigos.teste.

Vamos criar os seguintes pacotes: annotation, bundle, control, model, persistence, sets, e view. Dentro de view ainda teremos o control e o managed. Usei a base do nome de pacotes como br.ueg.portalVirtual.

Criados os pacotes, ficará mais ou menos assim (não se importem com a cor do package):

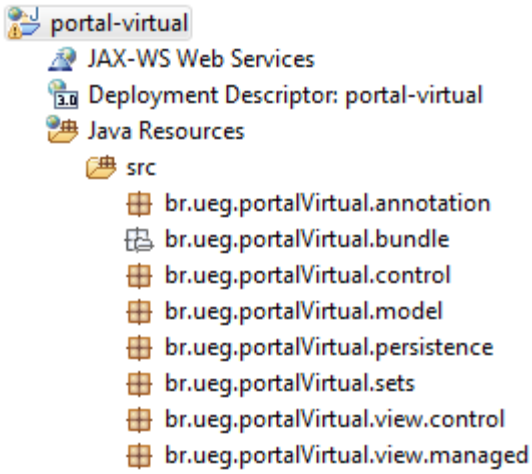


Figura 12 - Pacotes do projeto

O primeiro pacote (annotation) ficará as anotações que criaremos; no bundle terá apenas arquivos .properties para internacionalização e manutenção do JDBC ou qualquer outro que seja necessário; já em control

ficará nossas classes responsáveis pelo controle do projeto/validações, ou seja, é a camada intermediária entre a persistência e a visão; em persistence é a nossa persistência que fará a comunicação com o banco de dados; em sets ficam as configurações, tanto do Spring como de banco de dados; em view fica tudo designado para a visão, sendo que dentro de view temos control e managed, o primeiro são controladores que a visão obriga você a criar e no managed são os gerenciadores da visão, como usamos o JSF nesse pacote que ficam as classes de ManagedBean.

Programando - Modelo e suas dependências

Pronto nossa estrutura, vamos começar a programar, nossa camada mais baixa é o modelo, mas antes de criá-la, devemos criar nossas annotations que serão utilizadas pela camada citada. No Java, annotation são chamadas com o @nomeDaAnnotation, elas são marcações que podem ficar na classe, no método ou no atributo e podem conter atributos para serem preenchidos, no JSF 2.0 trabalhamos com annotation em vez de XML, a única parte XML que fizemos até agora, e não passa disso, foi no web.xml. Para criar uma annotation devemos criar uma interface, só que na verdade criamos uma @interface, isso define que aquela classe é uma annotation. Clique com o botão direito no projeto e vá em New>Annotation:

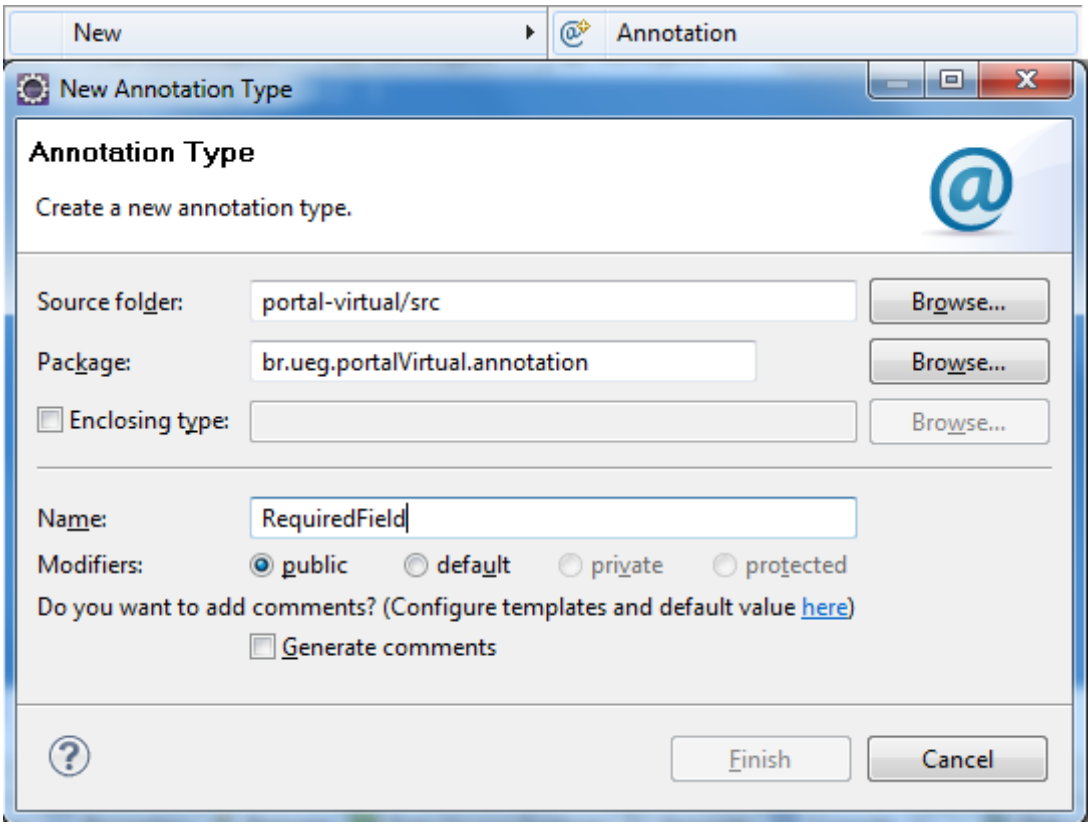


Figura 13 - New>Annotation

Defina o nome da Annotation como RequiredField e a salve no pacote específico para ela: br.ueg.portalVirtual.annotation, a responsabilidade dessa classe é marcar atributos que são obrigatório o preenchimento.

```
package br.ueg.portalVirtual.annotation;

import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface RequiredField {

}
```

Tabela 05 - RequiredField.java

Veja que a classe não tem corpo, usei-a apenas como marcador de campos obrigatórios. É necessário definir o tipo de elemento onde será usado a annotation, no nosso caso é para atributos (elementType.FIELD), e temos que definir a política de retenção que será em execução (RetentionPolicy.RUNTIME), sem a definição da "política" a annotation não irá funcionar bem.

Vamos agora para a camada mais baixa: o modelo. Dentro do pacote /model crie a classe Entity (New->Class), marque-a como abstract:

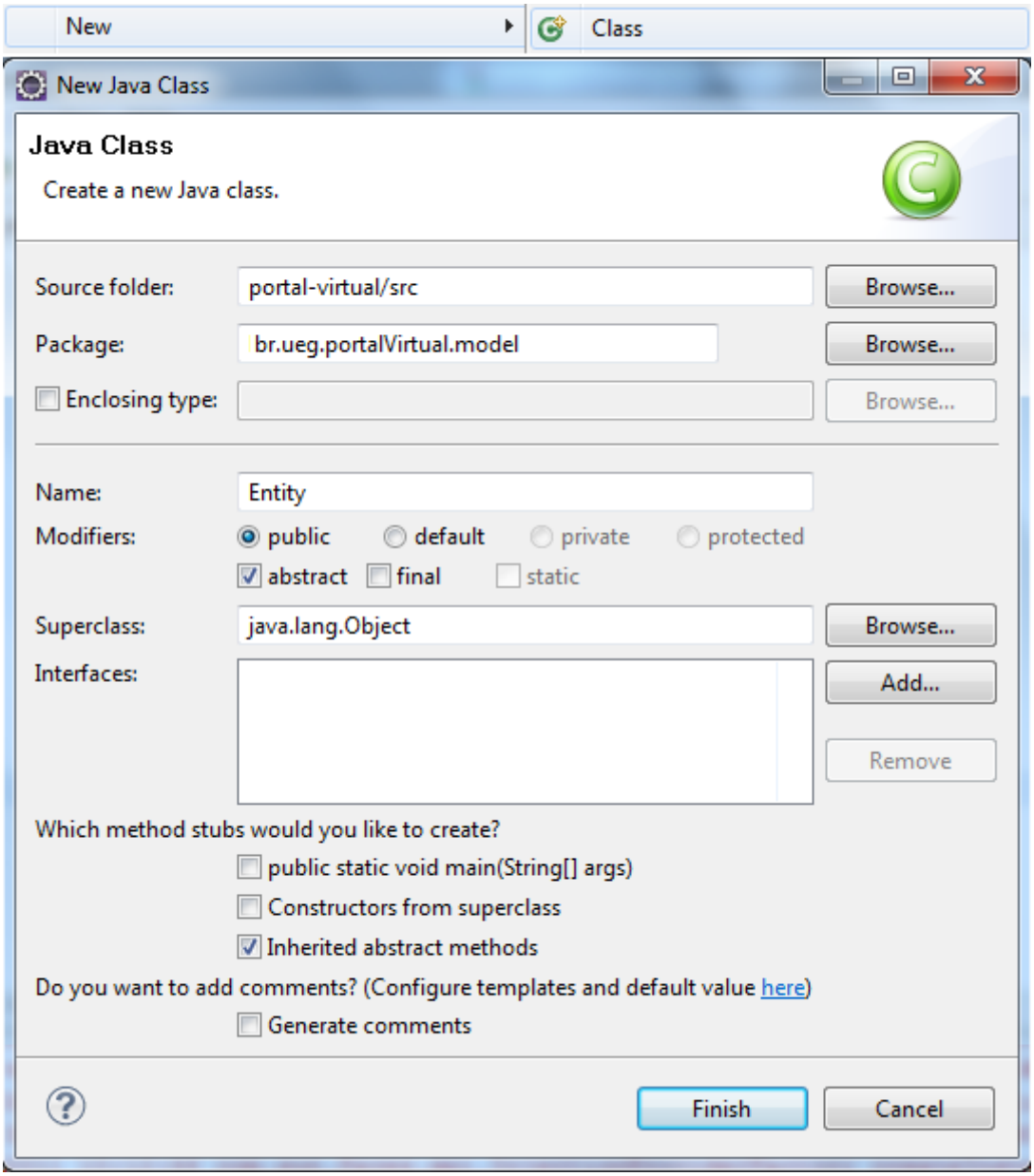


Figura 14 - New>Class

Depois explore os recursos do "Wizard" como o Superclass (extends) e o Interfaces (implements).

```
package br.ueg.portalVirtual.model;

import java.io.Serializable;

@SuppressWarnings("serial")
public abstract class Entity implements Serializable{

	public abstract long getId();

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
```

```
            result = prime * result + Long.signum(getId() ^ (getId()
>>> 32));
            return result;
      }

      @Override
      public boolean equals(Object obj) {
            if (this == obj)
                  return true;
            if (obj == null)
                  return false;
            if (getClass() != obj.getClass())
                  return false;
            Entity other = (Entity) obj;
            if (getId() != other.getId())
                  return false;
            return true;
      }

      public abstract String toString();
}
```

Tabela 06 - Entity.java

Antes da declaração da classe temos uma annotation do Eclipse de advertência, não há necessidade dela, serve apenas para não deixar a classe marcada com o sinal de atenção (@SuppressWarning é apenas uma anotação para que o Eclipse não faça algumas validações e deixe de avisar algumas coisas, nesse caso a SuppressWarning pede para não avisar sobre o serial ID que toda classe que implementa o Serializable deve ter, mas não faz interferência na aplicação, por isso é apenas um pedido de atenção que eu mandei ignorá-lo).

Essa classe abstrata é apenas um marcador, ela obriga a implementar o método de getId retornando um long, que será a chave primária do objeto, implementa Serializable porque as classes que a herdarem serão persistidas no banco de dados e o Serializable serve exatamente como marcador para tal objetivo. Obrigo também quem herda essa classe a sobrescrever o método toString. Repare que os métodos hashCode e equals está marcado como Override, isso quer dizer que eles foram sobrescritos, e usei o Eclipse para fazer isso automaticamente para mim (clique como botão direito sobre o código da classe, vá em Source>Generate hashCode() and equals()...), como essa classe não tem nenhum atributo, não é possível pedir para o Eclipse gerar o hashCode e equals, então crie o atributo id (private long id;) e peça para gerar automaticamente, depois de gerado, apague o atributo e substitua a variável id pelo método getId, repare também que no lugar do

resultado da Id no hashCode tem o método de conversão Long.signum(long i) que converte um Long em Integer.

Todas classes no pacote /model devem herdar a Entity.

No banco de dados temos as tabelas reino e filo, devemos entrar criar as classes para fazer a correspondência no projeto:

```
package br.ueg.portalVirtual.model;

import javax.persistence.Column;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import br.ueg.portalVirtual.annotation.RequiredField;

@SuppressWarnings("serial")
@javax.persistence.Entity
@Table(name = "reino")
public class Reino extends Entity {

	@Id
	@GeneratedValue
	@OneToMany(targetEntity = Filo.class, mappedBy = "reino")
	@Column(name = "id_reino")
	private long id;
	@Column(name = "nome")
	@RequiredField
	private String reino;

	public Reino() {
	}

	public Reino(long id, String reino) {
		this.id = id;
		this.reino = reino;
	}

	public long getId() {
		return id;
	}

	public void setId(long id) {
		this.id = id;
	}

	public String getReino() {
		return reino;
	}

	public void setReino(String reino) {
		this.reino = reino;
	}

	@Override
	public String toString() {
```

```
            return "Reino";
        }

}
```

Tabela 07 - Reino.java

Aqui já temos o uso do JPA verifique que acima da declaração da classe Reino há 2 annotation: @Entity e @Table, essas mesmas annotation existe no Hibernate, mas utilizo a do JPA para haver maior flexibilidade de tecnologias.

O Hibernate é um framework de persistência que pode ou não utilizar as especificações do JPA, já que o Hibernate tem suas próprias especificações (nada mais que CTRL + C, CTRL + V no JPA). Vamos ver então o Hibernate como um container e o JPA como a ferramenta para fazer o container funcionar. Se utilizarmos o Hibernate, ele por si só tem suas ferramentas para entrar em funcionamento, mas e se não quisermos aquele container? Se quisermos trocar por outro que atenda nossas necessidades no momento, teremos que adicionar o novo container e mudar tudo que usa o container para as novas ferramentas. O JPA já é essa ferramenta, se usarmos no projeto as especificações do JPA, podemos mudar o container (Hibernate) por outro com o mínimo de implicações possíveis, por isso uso sempre as especificações do JPA em vez do Hibernate.

A partir de agora, se você não fez a leitura do artigo do Felipe Saab sobre Spring e Hibernate isso o prejudicará numa melhor compreensão, então, leia!

A annotation @Entity é um marcador informando que aquela classe será persistida e tem uma correspondente no banco de dados. A @Table(name) informa qual o nome da tabela correspondente aquele objeto, ou seja, no nosso banco de dados criamos a tabela "reino" e nossa classe Reino faz referencia a essa tabela.

Foi criado então o atributo id, do tipo long, e como annotation para esse atributo temos: @Id, que determina que esse atributo é a primary key do objeto; @GeneratedValue, marcação para que quando a entidade for persistida gere um valor automaticamente, ou seja, não há necessidade de defini-la quando quiser salvar um novo registro; @OneToMany, essa annotation é do tipo relacional, serve para dizer que esse atributo é uma

foreign key de outra tabela, veja que foi definido os atributos targetEntity e mappedBy, o targetEntity é para onde esse valor vai, ou seja, onde ele será a foreign key, o mappedBy quer dizer quem está o mapeando, de onde deve se pegar esse valor (a primary key da tabela reino); e por ultimo temos o @Column, como o próprio nome diz, nessa annotation você define qual o nome da coluna no banco de dados.

Como ultimo atributo temos o reino, do tipo String, e suas annotations são: @Column, para definir a coluna da tabela; e @RequiredField, ai está nossa annotation, marcando o atributo reino como campo obrigatório, depois veremos como vamos utilizá-la.

Crie 2 construtores (polimorfismo de sobrecarga) para instânciar a classe sem fazer nada e outra passando os parâmetros id e reino e associando aos construtores locais, o último construtor é necessário para uma validação no JSF que veremos depois.

Segue os métodos getters and setters dos atributos, para gerá-los automaticamente clique com o botão direito sobre o código da classe, vá em Source -> Generate Getters and Setters... selecione todos os atributos e clique em "Ok", caso você já tenha pedido para o Eclipse adicionar os métodos obrigatórios não implementados, delete o método getId() e deixe o Generate Getters and Setters recriá-lo.

Por último temos o método toString que a classe abstrata Entity nos obriga a implementá-lo(sobrescrevendo, já que o método existe no java.lang.Object), eu mando retornar o nome da classe, nesse caso "Reino", já que se você pedir o toString sem tê-lo sobrescrito irá retornar o pacote + o nome da classe, e mais pra frente precisarei do nome da classe sem saber o pacote onde está.

Próxima entidade que temos no banco de dados é o Filo:

```
package br.ueg.portalVirtual.model;

import javax.persistence.Column;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import br.ueg.portalVirtual.annotation.RequiredField;
```

```
@SuppressWarnings("serial")
@javax.persistence.Entity
@Table(name = "filo")
public class Filo extends Entity {

	@Id
	@GeneratedValue
	@Column(name = "id_filo")
	private long id;
	@Column(name = "nome")
	@RequiredField
	private String filo;
	@ManyToOne(optional = false, targetEntity = Reino.class)
	@JoinColumn(name = "id_reino", referencedColumnName =
"id_reino")
	@RequiredField
	private Reino reino;

	public Filo() {}

	public Filo (long id, String filo, Reino reino) {
		this.id = id;
		this.filo = filo;
		this.reino = reino;
	}

	public long getId() {
		return id;
	}

	public void setId(long id) {
		this.id = id;
	}

	public String getFilo() {
		return filo;
	}

	public void setFilo(String filo) {
		this.filo = filo;
	}

	public Reino getReino() {
		return reino;
	}

	public void setReino(Reino reino) {
		this.reino = reino;
	}

	@Override
	public String toString() {
		return "Filo";
	}

}
```

Tabela 08 - Filo.java

A classe segue o mesmo esquema que a anterior, verifique que o atributo da annotation @Table passou a ser "filo", o nome da tabela que corresponde esse objeto no banco de dados. Temos o atributo id com suas annotations, o atributo filo que é um campo obrigatório (@RequiredField) e temos o atributo reino (também obrigatório), sendo a nossa foreign key que mapeamos na classe Reino, mas temos que fazer as especificações do relacionamento na classe Filo também, para isso temos a annotation @ManyToOne (inverso do que foi declarado no Reino) dizendo que é um atributo obrigatório (false para optional) e que ele vem da classe Reino; logo depois de qual coluna o valor vem pela annotation @JoinColumn, informando o nome da coluna de destino "id_reino", na tabela filo, e o nome da coluna de origem, também "id_reino", na tabela reino (definido pelo targetEntity do @ManyToOne). De uma olhada na SQL do seu banco de dados para ver as comparações.

Depois segue os construtores da classe com seus getters and setters e o método toString sobrescrito agora para retornar "Filo", que é o nome da classe.

Terminamos o nosso modelo, vamos para a persistência do mesmo.

Programando - Persistência

Para a persistência, necessitamos do Hibernate, criaremos uma classe genérica que deve receber algum objeto que herde Entity, ou seja, nosso modelo, e irá fazer qualquer ação no banco de dados a partir disso. Uma classe de persistência básica (CRUD) terá apenas os comandos para salvar, listar, buscar, alterar e deletar, para isso criaremos uma interface contendo as assinaturas desses métodos. Vá então no pacote de persistência, clique com o botão direito sobre ele -> New -> Interface e defina o nome da interface para IGenericDAO:

Figura 15 - New>Interface

```
package br.ueg.portalVirtual.persistence;

import java.util.List;

import br.ueg.portalVirtual.model.Entity;

public interface IGenericDAO<E extends Entity> {

	public long save(E entity);
	public void update(E entity);
	public void delete(E entity);
	public List<E> findByCriteria(E entity, String value);
	public List<E> findByHQL(E entity, String value);
	List<E> getList(E entity);

}
```

Tabela 09 - IGenericDAO.java

Logo após o nome da interface temos o uso de Generics (<E>), essa pratica é utilizada pelo Java no ArrayList, HashMap, entre outros. Serve para não precisarmos definir o tipo do elemento para a classe, generalizamos para um tipo E (Elemento) e quando instânciamos a classe que definimos qual é o tipo dela, exemplo, no ArrayList se usa generics, mas só quando você da um new ArrayList<E> você define o tipo que o ArrayList irá utilizar. O mesmo acontece com nossa interface, ela persiste o tipo E que extende de Entity, não sabemos qual é o tipo, apenas obrigamos que qualquer classe que seja usada no Generics, tenha extendido o Entity. Segue, na interface, as assinaturas dos métodos de salvar, atualizar, deletar, procurar e listar a entidade.

Agora é hora de implementar esses métodos, clique com o botão direito sobre o pacote de persistência e vá em New -> Class e crie a classe GenericDAO:

```
package br.ueg.portalVirtual.persistence;

import java.util.List;

import org.hibernate.criterion.DetachedCriteria;
import org.hibernate.criterion.Restrictions;
import org.springframework.orm.hibernate3.HibernateTemplate;

import br.ueg.portalVirtual.model.Entity;


public class GenericDAO<E extends Entity> implements IGenericDAO<E>{

	private HibernateTemplate hibernateTemplate;


	public void setHibernateTemplate(HibernateTemplate
hibernateTemplate) {
		this.hibernateTemplate = hibernateTemplate;
	}

	@Override
	public long save(E entity) {
		return (Long) hibernateTemplate.save(entity);
	}

	@Override
	public void update(E entity) {
		hibernateTemplate.update(entity);
	}

	@Override
	public void delete(E entity) {
		hibernateTemplate.delete(entity);
	}

	@SuppressWarnings("unchecked")
	@Override
	public List<E> findByCriteria(E entity, String value) {
		DetachedCriteria criteria =
DetachedCriteria.forClass(entity.getClass()).add(Restrictions.like(ent
ity.toString().toLowerCase(), "%"+value+"%"));
		return hibernateTemplate.findByCriteria(criteria);
	}

	@SuppressWarnings("unchecked")
	@Override
	public List<E> findByHQL(E entity, String value) {
		String hql = "from " + entity.toString() + "where nome like
'%"+value+"%'";
		return hibernateTemplate.find(hql);
	}

	@SuppressWarnings("unchecked")
```

```
	@Override
	public List<E> getList(E entity) {
		return (List<E>)
hibernateTemplate.loadAll(entity.getClass());
	}

}
```

Tabela 10 - GenericDAO.java

A classe GenericDAO, assim como sua interface, também usa Generics. Assim que der o implements passe o mouse sobre o erro que aparecerá na classe e peça para "add unimplemented methods", crie o atributo hibernateTemplate, do tipo HibernateTemplate, que é o método que oferece os serviços de persistência do Hibernate, crie um setHibernateTemplate para o Spring gerenciar a instância desse objeto. Se você leu o artigo do Felipe Saab, sabe que o Spring gerencia a instância de uma classe e injeta os objetos necessários para os atributos existente nela (injeção de dependência). A classe GenericDAO será gerenciada pelo Spring, o objeto hibernateTemplate não é e não será inicializado pelo Java, e o nosso framework que ficará responsável por isso, logo veremos que o Spring terá uma instância de um objeto do tipo HibernateTemplate, com todos seus atributos setados, então injetaremos esse objeto no hibernateTemplate da nossa classe de persistência, a partir daí ele poderá fazer ações, resumindo, em vez de no construtor da classe GenericDAO eu passar um new HibernateTemplate() para o objeto e depois setar os atributos obrigatórios, deixo essa parte para o Spring e não preciso me preocupar o tempo que esses objetos ficaram na memória e de que forma, o próprio framework gerencia isso, da melhor forma que ele achar necessário.

Como o Spring tratará da inicialização do hibernateTemplate, tomemos o como inicializado e vamos usá-lo sem preocupação. Para cada método criado na interface de persistência há um correspondente no Hibernate, para o save, o Hibernate tem o mesmo método, que precisa que se passe um objeto com os atributos setados e que tenham annotations de persistência que indiquem qual a tabela e a coluna dos atributos, então o Hibernate fará a persistência do mesmo.

O mesmo se segue para os outros métodos, uma pequena observação para o método de busca, que pode ser feito por Criteria ou por HQL, no

primeiro, você cria um objeto que você define a tabela onde você quer pesquisar através do objeto anotado, de que forma quer pesquisar, no nosso caso usando o LIKE, qual coluna que deverá ser analisada e o objeto a ser comparado com a coluna. Veja que agora fiz o uso do método sobrescrito toString que definir abstrato na classe Entity, lembra que para cada classe eu pedi para esse método retornar o nome da classe em minúscula, isso porque nossas entidades, o atributo principal tem o mesmo nome da classe, verifique que para cadastrar o nome do reino da classe Reino temos que definir o atributo reino, o mesmo acontece com Filo. Então o nosso Criteria fará, para um objeto do tipo Reino, uma pesquisa na tabela Reino, comparando os registros da coluna reino com o valor passado. O Criteria analisa o objeto, e não os registros no banco de dados, então ele vai na classe, verifica suas annotations, depois escreve uma SQL correspondente para fazer a consulta.

No HQL acontece a mesma forma, só que em vez de criarmos um objeto que fará a pesquisa, nos escrevemos a SQL, o que nos da maior liberdade. Só que diferente do Criteria, o HQL já vai direto para o banco de dados, não faz analise nenhuma nas annotations da classe, então devemos escrever já na forma que a SQL deve ficar. Nossa SQL para consulta, na tabela reino, seria: "SELECT * FROM reino WHERE nome LIKE %valorASerConsultado%;". O HQL resume algumas coisas desnecessárias, você não precisa falar que é uma consulta, então omitimos o "SELECT *", já que essa é sua única função, dizemos então qual é a tabela (reino) e a coluna a ser pesquisada (nome, que tanto para reino ou filo, tem a mesma coluna, por isso não criei, na SQL, uma coluna nomeFilo e nomeReino, servindo para as duas entidades) e depois o valor a ser analisado.

Por último temos o listar, que mandar carregar todos os dados de uma tabela (classe anotada no Java).

Temos nossa persistência criada, vamos configurar a conexão com o banco de dados, depois o Spring.

Programando - Conectando com o Banco de dados

Precisamos criar uma conexão com o nosso banco de dados para pode fazer as ações que definimos na classe GenericDAO. Já adicionamos o driver jdbc que fará a comunicação entre o Java e o SQLServer (espero!) agora basta estabelecermos a conexão com o database criado. Para isso, crie uma classe com o nome de Connect no nosso pacote de configurações (br.ueg.portalVirtual.sets), para facilitar nosso trabalho, extenda a classe DriverManagerDataSource do Spring:

```
package br.ueg.portalVirtual.sets;

import java.util.ResourceBundle;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

public class Connect extends DriverManagerDataSource{

	private ResourceBundle jdbc;

	public Connect() {
		jdbc =
ResourceBundle.getBundle("br/ueg/portalVirtual/bundle/jdbc");

	this.setDriverClassName(jdbc.getString("driverClassName"));
		this.setUrl(jdbc.getString("url"));
		this.setUsername(jdbc.getString("username"));
		this.setPassword(jdbc.getString("password"));
	}

}
```

Tabela 11 - Connect.java

A classe tem apenas um atributo chamado jdbc do tipo ResourceBundle. Bundle é um arquivo do tipo .properties que contém apenas texto, sendo que cada linha é composta por: texto, sinal de igualdade e mais texto; o lado esquerdo da igualdade é a chave para se pesquisar e o lado direito da igualdade é o valor que representa a chave (seria a "key" e "value" de um HashMap, por exemplo). Então para se obter o lado direito da igualdade, você deve buscar enviando o texto do lado esquerdo.

No construtor da nossa classe inicializamos nosso ResourceBundle pedindo para pegar o Bundle jdbc(arquivo jdbc.properties) que fica no pacote br.ueg.portalVirtual.bundle. Nossa classe Connect é responsável por estabelecer a conexão com o banco de dados, para isso precisamos definir

alguns atributos no nosso gerenciador do driver para que ele possa "entender" que banco de dados utilizamos, como deverá criar as SQLs e qual database analisar. Como citei anteriormente, devemos extender o DriverManagerDataSource do Spring para facilitar nossas vidas, e nessa super-classe temos os atributos DriverClassName, URL, Username e Password que são necessários "preencher". O primeiro diz qual banco de dados utilizamos, a URL diz o caminho da database, e dois últimos informam o usuário e senha para se logar no banco de dados. Como utilizamos o SQLServer o DriverClassName é com.microsoft.sqlserver.jdbc.SQLServerDriver (isso para o Driver original da Microsoft), nossa URL será o banco de dados instalado localmente na máquina, normalmente especificamos qual database em seguida, no URL mesmo, só que esse driver JDBC do SQLServer não deixa fazer isso (não pesquisei muito se há como fazer isso mesmo), então para dizer que usaremos o servidor local, a URL será: jdbc:sqlserver://127.0.0.1:1433 e depois colocamos o usuário e a senha que setamos no banco de dados. Até agora eu informei o que tem que ser passado para os atributos, mas no código está diferente, veja que eu peço para o Bundle uma String, mandando suas respectivas chaves de busca, ou seja, mandei o lado esquerdo da igualdade de um Bundle. Vamos então construir esse ResourceBundle para que a classe funcione, clique com o botão direito sobre o pacote de bundles e depois vá em New>Other...>File, digite o nome jdbc.properties (é necessário digitar a extensão do arquivo) e "Finish":

```
driverClassName=com.microsoft.sqlserver.jdbc.SQLServerDriver
url=jdbc:sqlserver://127.0.0.1:1433
username=ueg
password=portueg
```

Tabela 12 - jdbc.properties

Na criação do banco de dados citei que você poderia criar o mesmo usuário que uso no projeto ou alterar o arquivo Properties que tem as informações do banco de dados. Esse é o arquivo, se você não criou o mesmo usuário e senha, pode alterar agora nesse arquivo, caso contrário, mantenha as configurações e sua classe Connect funcionará perfeitamente pegando os dados de conexão desse arquivo.

Configurando o Spring

As bibliotecas do Spring já foram adicionadas quando configuramos o projeto (preciso repetir que espero?! =D), então basta configurar a sua inicialização e o que ele deve fazer no projeto. Há duas formas de configuração do Spring, por annotation e por XML, eu gosto de annotation, pra mim XML cai em desuso a cada dia, além de ser mais chato, mas se você usar annotation haverá um problema no controlador de visão do Spring (justamente na annotation @Controler), que ainda não consegui resolvê-lo, e se usarmos annotation seria bom configura a inicialização do Spring junto ao projeto, que citei anteriormente que também está caindo em desuso. Decidi, então, usar XML para a configuração e a não inicialização do Spring junto ao projeto, depois mando outra versão do tutorial explicando como funciona com annotation, apesar de que mesmo usando annotation, você deve criar o XML do Spring dizendo que vai usar annotation... Pois bem, na nossa camada de configurações (br.ueg.portalVirtual.sets) crie uma XML com o nome de spring, New > Other... > XML File:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
         xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

      <!-- Conexão com banco de dados -->
      <bean id="dataSource" class="br.ueg.portalVirtual.sets.Connect"
/>

      <!-- Hibernate -->
    <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSession
FactoryBean">
```

```
      <property name="annotatedClasses">
            <list>
                  <value>br.ueg.portalVirtual.model.Reino</value>
                  <value>br.ueg.portalVirtual.model.Filo</value>
            </list>
      </property>
      <property name="hibernateProperties">
            <props>
                  <prop
key="hibernate.dialect">org.hibernate.dialect.SQLServerDialect</prop>
                  <prop key="hibernate.show_sql">true</prop>
                  <prop key="hibernate.hbm2ddl.auto">update</prop>

            </props>
      </property>
      <property name="dataSource" ref="dataSource" />
    </bean>

      <!-- Injeta uma sessão do hibernate -->
      <bean id="hibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
            <property name="sessionFactory" ref="sessionFactory" />
      </bean>

      <!-- DAOs -->
    <bean id="genericDAO"
class="br.ueg.portalVirtual.persistence.GenericDAO">
      <!-- Injeta esse objeto hibernateTemplate dentro do DAO -->
      <property name="hibernateTemplate" ref="hibernateTemplate" />
    </bean>
</beans>
```

Tabela 13 - spring.xml

Iniciamos nosso XML, depois definimos que ele é um arquivo do Spring com a tag <beans /> dentro dessa tag estão todos os pacotes do Spring que utilizaremos (acha que aquele tanto de biblioteca é muito... não viu nada ainda!). O aop (xmlns:aop) declarado é a programação orientado a aspectos do Spring, se quiser retirá-la, fique a vontade, está ai para alguns testes meus e a possível incorporação da AOP no próximo (se houver) tutorial. Nesse arquivo XML também use o método CTRL + C, CTRL + V para nossa tag inicial.

Depois de definir que o arquivo é um configurador do Spring, iniciamos com a tag <bean /> ela designa os objetos que usaremos, seria a instancia do objeto, e o primeiro objeto declarado foi nossa conexão com o banco de dados, que dei o nome de dataSource (id do bean) e informei para o Spring onde se localiza essa classe para ele fazer sua instancia : br.ueg.portalVirtual.sets.Connect (o único atributo dessa classe nos o instanciamos no construtor da mesma, então não é necessário pedir para o

Spring gerenciá-lo). Definido essa tag, já temos a conexão com o banco de dados criada, e podemos referenciá-la pelo id dado (dataSource).

Na próxima tag temos a definição da sessionFactory, um atributo necessário para inicializar o hibernate, setamos seu nome e a localização da classe nos pacotes do Spring e definimos algumas propriedades (atributos), como fizemos o uso das annotations do JPA para configuração das entidades, devemos definir a propriedade do atributo "annotatedClasses" com isso o Hibernate irá ler as annotations contidas nas classes e não irá procurar um arquivo XML para fazer as análises. Essa propriedade é uma lista de entidades declaradas para o uso no projeto, e por isso devemos definir quais são. Como criamos apenas duas, Reino e Filo, dizemos sua localização no value da lista. Próximo propriedade a ser declarada é o hibernateProperties, sendo que essa propriedade tem propriedades dentro delas a ser definidas (mais atributos), mas não é necessário definir todos, nesse projeto definimos o "dialect" que seria a linguagem utilizada pelo banco de dados, como usamos o SQLServer, o "dialect" é: org.hibernate.dialect.SQLServerDialect, depois definimos para mostrar a SQL (true), isso serve apenas para testes no desenvolvimento, quando for implantar o projeto normalmente desabilita essa opção, já que será apenas algo escrito no console sem uso nenhum. Por ultimo definimos o hbm2ddl.auto, essa propriedade diz como deve tratar o banco de dados, se tem um alto controle: criar tabelas, salvar, atualizar e listar dados, definido por "create", e se pode apenas salvar, atualizar e listar os dados, definido por "update" (que é o nosso caso), tem mais alguns tipos, mas não irei citá-los. Se estiver setado o hbm2ddl como create, você nem precisaria criar as tabelas, apenas criava a database e executava o projeto que ele criaria sozinho, mas prefiro não usar esse recurso.

Como última propriedade do hibernateProperties, precisamos passar a conexão com o banco de dados para o Hibernate poder fazer alguma coisa, como já definimos ela pelo Spring, basta passar o a referência pelo nome que a demos: ref="dataSource" e por aqui termina a definição da sessão do Hibernate.

Lembra que nossa classe GenericDAO temos um atributo do tipo HibernateTemplate, vamos agora setar as propriedades desse objeto e criar uma instancia pelo Spring, e a próxima tag de <bean /> é isso. Definimos o id do HibernateTemplate e a localização da classe, depois definimos suas propriedades, que é apenas uma: sessionFactory, como estamos controlando essa instancia também pelo Spring, basta passar a referência da mesma, assim como já fizemos e pronto, temos um objeto do tipo HibernateTemplate.

Vamos utilizar o objeto que acabamos de criar na nossa classe de acesso aos dados do banco. Crie a tag <bean /> com o id de "genericDAO" e passe onde se localiza a nossa classe GenericDAO, como nós mesmo criamos essa classe, sabemos que há a necessidade de definir apenas uma propriedade, a hibernateTemplate, e como já temos a instancia desse objeto pelo Spring, passe apenas a referência da mesma.

Pronto, nosso Spring já está configurado, gerenciando o Hibernate e fazendo injeção de dependência na classe GenericDAO. O Spring é enorme e pode fazer várias coisas, e com esse tanto de linhas no XML apenas configuramos o que foi citado. Poderíamos gerenciar também outras classes, mas como citei que há um problema no controlador da visão do Spring, preferi não estender essa injeção de dependências.

Spring configurado, mas como iremos instanciar a GenericDAO? Como a aplicação saberá que estamos utilizando o Spring? Para resolver isso, vamos criar uma classe de instanciação dos beans definidos no XML (apenas a GenericDAO).

Programando - Fábrica de Beans

No tutorial do Felipe Saab, ele citou o AplicationContext para retornar as instancias dos objetos controlados pelo Spring, e disse que o BeanFactory é seria outro método de trazer esses objetos, só que bastante simples. Fiz a análise dos dois e decidi criar a minha fábrica de beans, ou a minha extensão do AplicationContext.

Para isso, crie uma classe com o nome de SpringFactory e extenda a classe ClassPathXmlApplicationContext do Spring:

```
package br.ueg.portalVirtual.sets;

import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class SpringFactory extends ClassPathXmlApplicationContext {

	private static ClassPathXmlApplicationContext instance = null;

	public static ClassPathXmlApplicationContext getInstance() {
		if (instance == null) {
			instance = new
ClassPathXmlApplicationContext("br/ueg/portalVirtual/sets/spring.xml")
;
		}
		return instance;
	}

	private SpringFactory() {}

}
```

Tabela 14 - SpringFactory.java

Essa classe usa o Design Pattern (Padrão de Projeto) Singleton. Esse método faz com que haja somente uma instancia da classe que o implemente na aplicação toda. Esse método é bastante usado e soluciona vários problemas, nesse caso não é necessário o uso desse Design Pattern, o usei porque gosto e não ficar gerando a mesma instancia do AplicationContext em diferentes objetos.

Para esse Padrão de Projeto funcionar, você tem que desabilitar a possibilidade de instanciar a classe, fazendo com que o construtor seja privado, e crie um método estático que retorne a instancia do objeto, nesse

método ele irá verificar se há alguma instancia ou não, não havendo ele gera a instancia.

No método estático eu retornei um ClassPathXmlApplicationContext que pode ser convertido em um AplicationContext contendo os métodos de retorno de beans configurados no spring.xml (Veja a semelhança com o método que o Felipe Saab usa).

Agora que temos nossa classe que traz os objetos gerenciados pelo Spring, vamos construir nosso controlador.

Programando - Criando nossos controladores

Na camada de controle, ficarão nossas classes de controle de persistência, de validação e de mensagens, também poderia ficar nesse pacote as regras de negócios, se existissem. Vamos começar com os controladores auxiliares, crie, então, a classe ValidatorControl:

```
package br.ueg.portalVirtual.control;

import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

import br.ueg.portalVirtual.annotation.RequiredField;
import br.ueg.portalVirtual.model.Entity;

public class ValidatorControl {

	private List<String> emptyFields;

	public ValidatorControl() {
		emptyFields = new ArrayList<String>();
	}

	@SuppressWarnings("rawtypes")
	public boolean isEmpty(Entity entity) {
		clean();
		boolean empty = false;
		Class entityClass = entity.getClass();
		Field[] fields = entityClass.getDeclaredFields();
		for (Field currentField:fields) {
			try {

	currentField.getAnnotation(RequiredField.class);
				Object fieldValue = getField(entity,
currentField.getName());
				if (fieldValue == null ||
fieldValue.equals("")) {
					empty = true;
					emptyFields.add(currentField.getName());
				}
			}catch (Exception e) {
				// do nothing yet
			}
		}
		return empty;
	}

	public boolean isEmpty(String value) {
		clean();
		boolean empty = false;
		if (value == null || value.equals("") ) {
			empty = true;
```

```
                  emptyFields.add("Busca");
            }
            return empty;
      }

      @SuppressWarnings({ "unchecked", "rawtypes" })
      private Object getField(Entity entity, String fieldName) throws
SecurityException, NoSuchMethodException, IllegalArgumentException,
IllegalAccessException, InvocationTargetException {
            Class entityClass = entity.getClass();
            Method method = entityClass.getMethod("get" +
fieldName.substring(0, 1).toUpperCase() + fieldName.substring(1));
            return method.invoke(entity);
      }

      public void clean() {
            emptyFields.clear();
      }

      public List<String> getEmptyFields() {
            return emptyFields;
      }

}
```

Tabela 15 - ValidatorControl.java

Devo dizer que essa é umas das minhas classes favoritas, pois usa o Java Reflection para validar qualquer classe que extenda Entity (=D), e é aqui que veremos o uso da nossa annotation @RequiredField.

Para nível de conhecimento, Reflection é uma prática que podemos analisar a classe em tempo de execução, assim descobrimos quais atributos ela tem, seus métodos e podemos até invocá-los. Não é uma prática recomendada para quem está iniciando no Java, pois requer um conhecimento maior da linguagem, mas é errando que se aprende.

Tem algumas características que definem a Reflection, que é a introspecção e a intersecção. No Java tem a introspecção que é a capacidade da classe se auto-examinar, a intersecção é a capacidade de tomar uma decisão em tempo de execução. Essa prática é muito mais do que foi citado, mas somente isso é o bastante para ter conhecimento do que estamos usando. Para mais informações tem umas boas explicações na Oracle e GUJ.

Nossa classe tem como atributo uma lista de String com nome de emptyFields, que usaremos para relacionar todos os campos obrigatórios

que não foram preenchidos. O construtor apenas inicializa essa lista com nada dentro.

Vamos entender os métodos dessa classe de baixo para cima (bottom-up). Como último método temos o getEmptyFields que retorna o atributo de lista de nomes de campos que não foram preenchidos. O método clean() serve para limpar a lista.

No método getField temos alguns throws, esses throws significam que podem acontecer as exceptions citadas e não serão tratadas nesse método, obrigando a quem usá-lo fazer esse tratamento. E agora começa o Java Reflection! Criamos uma variável local do tipo Class que recebe a Class do parâmetro entity, com essa variável que teremos acesso aos métodos internos, atributos o que já citei que a Reflection pode fazer, e na próxima linha já vemos o exemplo disso, pedimos a entityClass para nos dar o método get*NomeDoCampo*, isso foi feito porque nossa Entity nada mais é que um POJO, então todos atributos são privates e para verificar seus valores preenchidos, devemos pegar seu respectivo método "get". Então invocamos o método "get" do atributo e os retornamos.

O próximo método é um polimorfismo de sobrecarga de método do primeiro. Em vez de receber uma Entity, recebe uma String, esse método serve para verificar se a String não está vazia, valida então se o campo de busca foi preenchido. Veja que ele limpa a lista e faz as verificações na String, caso esteja vazia, adiciona a lista o campo que não foi preenchido: Busca, edita o booleano que representa o retorno do método para true e o retorna.

E por último, ou primeiro método, temos o "isEmpty" que recebe um objeto do tipo Entity, verifica quais são os campos obrigatório e retorna se tem algum não preenchido. O método recebe um Entity, mas essa classe é abstrata, não tem como instanciá-la, então será enviado um objeto que a extenda. A primeira coisa que o método faz é chamar o método "clean", criamos uma variável local do tipo boolean que será o retorno desse método e o inicializamos com false (temos como hipótese inicial que a entidade está com os campos preenchidos).

Criamos um objeto do tipo Class, e mandamos o Class do objeto entity que veio como parâmetro do método (entity.getClas()), pegamos os campos dessa Class e a associamos a um vetor do tipo Field, e a nova variável fields tem todos os campos da classe entity. Agora devemos percorrer esses campos e ver se estão preenchidos. Usei o "for" para fazer o laço de repetição, mas veja que a assinatura do "for" é um pouco diferente, esse jeito de utilizá-lo se chama "for each" que cria um laço de repetição para um vetor (ou lista) percorrendo-o todo, seria a mesma coisa que:

```
for (int i = 0; i<fields.length; i++) {
      Field currentField = fields[i];
      //...
}
```

Tabela 16 - For each em modo rótulo

Dentro do nosso "for" criamos um bloco de try/catch que tentará fazer a captura da annotation @RequiredField que criamos e como chamaremos os métodos explicados acima, necessita do try/catch para fazer o tratamento. Se o campo não contém a annotation citada, ele pula para o bloco de catch. Se o campo estiver com a annotation @RequiredField criamos uma variável do tipo Object que recebe o valor associado ao atributo, para isso chamamos o método local "getField", passando a objeto entity que estamos analisando e o nome do campo que queremos verificar se está preenchido.

Com o valor do campo associado ao Object fieldValue fazemos as verificações se ele está preenchido, se estiver vazio, alteramos o valor do booleano de retorno para true e adicionamos o nome do campo obrigatório que não foi preenchido.

O bloco de catch deste método deveria ser melhor desenvolvido, pois há várias exceptions a serem tratadas, e eu assumi que não ocorrerá nenhuma exception que impedirá o funcionamento da aplicação, apenas terá a exception da annotation não encontrada, e não se deve fazer nada mesmo. Mas como já fiz os testes e estou controlando muito bem esse controlador, não ocorre nenhuma exception que irá "travar" a aplicação (se desenvolvido como no tutorial xD).

Vamos agora para o nosso controlador de mensagens. Crie uma classe abstrata com o nome de MessagesControl:

```
package br.ueg.portalVirtual.control;

import java.util.ResourceBundle;

public abstract class MessagesControl {

	private static ResourceBundle messages;

	public MessagesControl() {
		messages =
ResourceBundle.getBundle("br/ueg/portalVirtual/bundle/messages");
	}

	protected String getMessage(String typeMessage) {
		return messages.getString("erro_" + typeMessage);
	}

	public abstract void addMessage(String message);

}
```

Tabela 17 - MessagesControl.java

No nosso controlador abstrato de mensagens temos o ResourceBundle como atributo, um construtor que inicializa o nosso Bundle de mensagens (ainda será criado) e um método protected getMessage que recebe como parâmetro o tipo da mensagem, e retornamos a String da chave enviada pelo Bundle messages. Logo depois temos um método abstrato addMessage(). Tive alguns problemas em extender essa classe, por causa do FacesContext do JSF, e tive que tirar a inicialização do bundle do construtor. Essa classe tem como responsabilidade pegar mensagens do bundle e obrigar quem a implemente criar um método para adicionar a mensagem e exibi-la, independente se é Java Desktop, Web ou Mobile, quem a implemente que fará a distinção.

Agora sim podemos criar nosso controlador de persistência, e como é para CRUD, criaremos uma interface para ter uma melhor organização:

```
package br.ueg.portalVirtual.control;

import java.util.List;

public interface IControl<E> {

	public List<E> getListAll(E entity);
	public Long save(E entity);
	public void update (E entity);
	public List<E> findByCriteria(E entity, String value);
	public List<E> findByHQL(E entity, String value);
	public void delete (E entity);
```

```
}
```

Tabela 18 - IControl.java

A nossa interface contém apenas as assinaturas para fazer a persistência, é idêntica a IGenericDAO, nem há a necessidade dessa interface, a classe Control pode implementar a IGenericDAO mesmo, mas a criei para fazer uma diferenciação, que pode haver.

Nosso controlador de persistência (Control) assim como a DAO é genérico, qualquer classe que extenda Entity poderá ser controlada e persistida pelo objeto. Também uso o Generics do Java como já expliquei anteriormente.

Vamos então criar a classe que implementa nossa interface, veja que o Control trabalha parecido como uma "fachada" para a persistência, ele apenas iria chamar os métodos do GenericDAO, mas antes, faz algumas validações o que torna um controlador pré-persitência:

```
package br.ueg.portalVirtual.control;

import java.util.List;

import br.ueg.portalVirtual.model.Entity;
import br.ueg.portalVirtual.persistence.GenericDAO;
import br.ueg.portalVirtual.sets.SpringFactory;
import br.ueg.portalVirtual.view.control.MessagesWeb;

public class Control<E extends Entity> implements IControl<E> {

	private GenericDAO<E> persistence;
	private ValidatorControl validator;
	private MessagesControl messages;

	@SuppressWarnings("unchecked")
	public Control() {
		persistence = (GenericDAO<E>)
SpringFactory.getInstance().getBean("genericDAO", GenericDAO.class);
		validator = new ValidatorControl();
		messages = new MessagesWeb();
	}

	@Override
	public List<E> getListAll(E entity) {
		return persistence.getList(entity);
	}

	@Override
	public Long save(E entity) {
		if (!validator.isEmpty(entity)) {
			return persistence.save(entity);
		} else {
			List<String> emptyFields =
```

```
validator.getEmptyFields();
                  for (int i = 0; i < emptyFields.size(); i++) {
                        messages.addMessage(emptyFields.get(i));
                  }
            }
            return (long) 0;
      }

      @Override
      public void update(E entity) {
            if (!validator.isEmpty(entity)) {
                  persistence.update(entity);
            } else {
                  List<String> emptyFields =
validator.getEmptyFields();
                  for (int i = 0; i< emptyFields.size(); i++) {
                        messages.addMessage(emptyFields.get(i));
                  }
            }

      }

      @Override
      public List<E> findByCriteria(E entity, String value) {
            if (!validator.isEmpty(value)) {
                  return persistence.findByCriteria(entity, value);
            } else {

      messages.addMessage(validator.getEmptyFields().get(0));
            }
            return null;
      }

      @Override
      public List<E> findByHQL(E entity, String value) {
            if (!validator.isEmpty(value)) {
                  return persistence.findByHQL(entity, value);
            } else {

      messages.addMessage(validator.getEmptyFields().get(0));
            }
            return null;
      }

      @Override
      public void delete(E entity) {
            persistence.delete(entity);
      }

}
```

Tabela 19 - Control.java

No Control temos como atributos a persistence, do tipo GenericDAO, o validator e o messagesControl, ou seja, temos a camada inferior de persistência (GenericDAO) e os controladores secundários (validator e messages). No construtor inicializamos os atributos, veja que a inicialização

do GenericDAO usa nossa fábrica de beans, instanciando o objeto via Spring, e não com o new como os outros embaixo dele. Verifique também que o MessagesControl recebe a instancia do MessagesWeb, essa última classe extende MessagesControl, e iremos implementá-la na camada de visão. Como o MessagesControl é abstract não se pode instanciá-lo.

Como primeiro método, temos o de listagem, como não é necessário fazer nenhuma validação, chamamos a camada inferior e pedimos para trazer a listagem do item. Depois segue os métodos de save e update, os dois seguem a mesma lógica, então vou explicá-los juntos. Primeiro é verificado se todos os campos foram preenchidos através do ValidatorControl mandando a entidade que queremos persistir para a análise. Se não houver campos obrigatórios vazios, chamamos o GenericDAO e salvamos/atualizamos, caso contrário, é chamado o MessagesControl e pedimos para adicionar a mensagem de erro (campo obrigatório não preenchido).

Os próximos métodos findByCriteria e findByHQL trabalham de forma semelhante, primeiro analisam se o campo de busca foi preenchido chamando o ValidatorControl, se sim, pedem para o GenericDAO fazer a busca, caso contrário pede para o MessagesControl criar mensagem de erro.

Por último temos o delete, também sem validações, apenas peço para excluir o item selecionado através da camada inferior.

Terminamos nossa camada intermediária, vamos a última camada, a visão.

Programando - Controlando a visão

Como estávamos falando de controlador, vamos começar criando nosso controlador de mensagens da visão. Crie a classe MessagesWeb no pacote de controle da visão: br.ueg.portalVirtual.view.control, e extenda o MessagesControl:

```
package br.ueg.portalVirtual.view.control;

import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;


import br.ueg.portalVirtual.control.MessagesControl;

public class MessagesWeb extends MessagesControl {

	public MessagesWeb() {
		super();
	}

	@Override
	public void addMessage(String message) {
		FacesMessage msg = new
FacesMessage(FacesMessage.SEVERITY_WARN, "Erro", getMessage(message));
		FacesContext.getCurrentInstance().addMessage(null, msg);
	}

}
```

Tabela 20 - MessagesWeb.java

Essa classe é bem simples, no seu construtor apenas chamamos o construtor da classe superior (método super()). Implementamos o método addMessage que o MessagesControl nos obriga a fazê-lo. Como se trata de um controlador de mensagens para a Web, criamos uma variável local do tipo FacesMessage, essa classe faz parte do JSF e guarda mensagens de erros e alerta, basta adicionarmos algum item e a página que tiver um componente de mensagens irá exibi-la (simples não?). Instanciamos o FacesMessage passando um tipo de mensagem para exibir o ícone corresponde ao mesmo (SEVERITY_WARN, mensagem de atenção apenas) e a mensagem que desejamos exibir. Na próxima linha adicionamos a mensagem a instancia do JSF.

Já temos um controlador de mensagens funcionando, e nem é preciso chamar métodos para exibir mensagem, basta criar um componente que exiba as mensagens salvas no JSF.

Na visão temos o componente ComboBox (ou SelectOne) que mostra uma lista de opções para selecionarmos e escolhermos uma. Iremos criar a ComboBox para o Reino, que possui os atributos id e reino. Será exibido apenas o nome do reino (reino), mas precisaremos enviar o id e o reino do item selecionado para o gerenciador da visão, e o JSF não faz isso sozinho. Ele apenas pega o item que deseja exibir (Reino), transforma em String e você escolhe o que tem que aparecer (itemLabel), depois de selecionado o item se mantém em String e você escolhe o que deve ir para o gerenciador (itemValue), quando você quer mandar a entidade toda (Reino, em vez de só id ou só reino) dá um erro, pois não dá para passar a String para o objeto desejado, para solucionar esse problema criamos um conversor.

Vamos criar nosso conversor do Reino e tentarmos entende-lo. No pacote de controladores da visão, crie a classe Converter e implemente a classe Converter do faces.converter:

```
package br.ueg.portalVirtual.view.control;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.FacesConverter;

import br.ueg.portalVirtual.model.Reino;

@FacesConverter
public class Converter implements javax.faces.convert.Converter{

	@Override
	public Object getAsObject(FacesContext context, UIComponent
component,
			String value) {

		int index = value.indexOf(':');

		if (index != -1) {
			return new Reino(Long.parseLong(value.substring(0,
index)),
					value.substring(index + 1));
		}
		return value;
	}

	@Override
	public String getAsString(FacesContext context, UIComponent
component,
```

```
                    Object value) {

            try {
                  Reino optionItem = (Reino) value;
                  return optionItem.getId() + ":" +
optionItem.getReino();
            } catch (Exception e) { }
            return (String) value;
      }

}
```

Tabela 21 - Converter.java

A classe possui a annotation @FacesConverter que diz para o JSF que essa classe se trata de um conversor de entidades. Quando implementamos o Converter do faces.converter, ele nos obriga a criar os métodos getAsObjet e getAsString, o primeiro transforma String em Object, e o outro transforma Object em String.

Essa classe é CTRL + C, CTRL + V, alterando para a sua necessidade, sendo que estamos criando agora o conversor para a entidade Reino, e poderíamos definir isso dentro da annotation @FacesConverter definindo os atributos delas dessa forma: @FacesConverter(forClass=Reino.class); mas como uso somente um conversor, não fiz essa declaração. Vamos agora entender a classe.

O método getAsObject pega a String do ComboBox (que veio do getAsString), procura onde fica o ":" pelo indexOf, se houver os ":" é que a String passou pelo método getAsString e podemos fazer a conversão sem nenhum problema, para isso repartimos a String na quantidade de atributos da classe (Reino tem dois atributos, id e reino) e instanciamos o objeto. Veja que temos um new Reino, convertendo a primeira String até o ":" em long (id) e depois passamos o valor pra frente do ":" sem conversão (reino) e retornamos o objeto do tipo Reino.

O método getAsString faz o inverso, ele recebe um objeto e tenta converte-lo no tipo Reino (Reino optionItem = (Reino) value;) se a conversão for possível, então retornamos todos os atributos do tipo como se fossem uma String, separando-os com ":", caso haja algum problema apenas há uma tentativa de converter o objeto em String direto.

Esses são os controladores da visão, antes de ir para os gerenciadores da visão, ou ManagedBeans, vamos criar as mensagens que usaremos nas telas, ou seja, mais Bundle (=D).

Programando - Internacionalização por Bundle

Já criamos o Bundle do JDBC, que sua única função é reunir, em seu arquivo, todos os dados para configurar e criar uma conexão com o banco de dados. Agora usaremos o Bundle para outra causa, além de reunirmos todas as mensagens na tela em um arquivo .properties, também o usaremos para internacionalizar o nosso projeto, ou seja, iremos facilitar a possível alteração de línguas no website.

No pacote de Bundle, crie o arquivo "messages_pt_BR.properties" (New>Other... >File):

```
######################## Mensagens de Erro ########################
erro_reino = O campo Reino é obrigatório!
erro_filo = O campo Filo é obrigatório!
erro_Busca = É necessário preencher o campo de busca para efetuar a
ação!

######################## Comuns ########################
button_save = Salvar
button_list = Listar
button_cancel = Cancelar
button_search = Buscar
column_edit = Editar
column_delete = Excluir

######################## Mensagens utilizadas no caso de uso Manter
Reino ########################
reino_title = Manter Reino
reino_add = Adicionar Reino
reino_header = Cadastrar Reino
reino_headerEdit = Editar Reino
reino_reino = Reino:
reino_reinoColumn = Reino
reino_empty = Não há reinos registrados.

######################## Mensagens utilizadas no caso de uso Manter
Filo ########################
filo_title = Manter Filo
filo_add = Adicionar Filo
filo_header = Cadastrar Filo
filo_headerEdit = Editar Filo
filo_filo = Filo:
filo_filoColumn = Filo
filo_empty = Não há filos registrados.
filo_select = Selecione um

######################## Mensagens utilizadas no template
########################
template_register = Cadastrar
```

Tabela 22 - messages_pt_BR.properties

Veja que separei as mensagens em: mensagens de erro (lembra do MessagesControl?); comuns; caso de uso Manter Reino; caso de uso Manter Filo; e da tela template. Esse Bundle é feito durante a confecção das telas, você não lembra de todas as mensagens e botões que uma tela vai ter antes.

Não há o que explicar nesse Bundle, temos a chave de pesquisa a esquerda e o valor a direita, utilizamos esse Bundle na confecção de mensagens de erro e na tela. Antes de configurar a internacionalização, vamos criar as mensagens em inglês, usei o pouco conhecimento que tenho e o Google Tradutor e criei o messages_en_US.properties:

```
####################### Mensagens de Erro ######################
erro_reino = The field Kingdom is required!
erro_filo = The field Phylum is required!
erro_Busca = Please fill the field of search to perform the action!

####################### Comuns ######################
button_save = Save
button_list = List
button_cancel = Cancel
button_search = Search
column_edit = Edit
column_delete = Delete

####################### Mensagens utilizadas no caso de uso Manter
Reino ######################
reino_title = Keep Kingdom
reino_add = Add Kingdom
reino_header = Register Kingdom
reino_headerEdit = Edit Kingdom
reino_reino = Kingdom:
reino_reinoColumn = Kingdom
reino_empty = There are not registered kingdom.

####################### Mensagens utilizadas no caso de uso Manter
Filo ######################
filo_title = Keep Phylum
filo_add = Add Phylum
filo_header = Register Phylum
filo_headerEdit = Edit Phylum
filo_filo = Phylum:
filo_filoColumn = Phylum
filo_empty = There are not registered phylum.
filo_select = Select one

####################### Mensagens utilizadas no template
######################
template_register = Register
```

Tabela 23 - messages_en_US.properties

Essa são as mensagens traduzidas, espero que esteja certo, vamos então configurar o Bundle para funcionar nas telas que criaremos, você já viu como configurar um Bundle no código do Java, usando o ResourceBundle, mas para ele ser acessível por qualquer tela devemos configurar o nosso arquivo no faces-config.xml que o Eclipse, provavelmente, já criou. O arquivo fica no diretório /WebContent/WEB-INF, perto do web.xml, ao abri-lo, assim como todo XML, irá abrir na forma não-código - Introduction, para poder visualizar o código e alterá-lo clique em "Source":

```
<?xml version="1.0" encoding="UTF-8"?>

<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">

    <application>
        <resource-bundle>
            <base-name>br.ueg.portalVirtual.bundle.messages</base-
name>
            <var>msg</var>
        </resource-bundle>
        <locale-config>
            <default-locale>pt_BR</default-locale>
            <supported-locale>pt_BR</supported-locale>
            <supported-locale>en_US</supported-locale>
        </locale-config>
        <message-bundle>
            br.ueg.portalVirtual.bundle.messages
        </message-bundle>
    </application>

</faces-config>
```

Tabela 24 - faces-config.xml

Se o Eclipse criou esse arquivo, ele vem com a tag <faces-config /> preenchida, essa tag apenas define que é um arquivo do JSF e sua versão, caso você tenha criado, use o CTRL + C e CTRL + V. Caso não estivéssemos querendo configurar por annotation o JSF, é aqui que configuraríamos as páginas, regras de navegações e todo o resto. Como vamos utilizar annotation apenas a configuração do Bundle ficará nesse arquivo, já que não encontrei como configurar por annotation.

Crie a tag <application /> (não use CTRL + C; CTRL + V, use o CTRL + ESPAÇO), dentro crie a <resource-bundle> nela temos que definir onde está o arquivo de internacionalização e o seu nome base

(br.ueg.portalVitual.bundle e o nome sem a definição do idioma: messages) e o nome da variável que receberá a instancia do Bundle (var). O ResourceBundle já está configurado, agora fazemos a internacionalização através do <locale-config /> dentro dela definimos o <default-locale> que é a linguagem padrão do sistema, e depois o <supported-locale>, que são as linguagens disponíveis. E como ultima tag do <application /> temos que criar a <message-bundle /> que é onde está o arquivo .properties, sem os locais de internacionalização (pt_BR ou en_US).

Agora se seu navegador for por padrão usar o português (pt_BR), irá exibir todas as mensagens em português, se estiver selecionado o inglês (en_US) irá exibir as mensagens em inglês, e caso a linguagem padrão não seja nenhuma dessas duas, irá exibir a linguagem padrão (pt_BR).

Vamos agora gerenciar a visão e criar as páginas.

Programando - Gerenciando a visão

Agora sim vamos mexer com JSF, e suas annotation. No pacote de gerenciador da visão crie a classe abstrata MB:

```
package br.ueg.portalVirtual.view.managed;

import java.util.List;

import javax.faces.event.ActionEvent;

import br.ueg.portalVirtual.control.Control;
import br.ueg.portalVirtual.model.Entity;

public abstract class MB<E extends Entity> {

	private Control<E> control;
	private boolean listing = false;
	private E entity;
	private List<E> listEntity;
	private String busca;

	public MB() {
		control = new Control<E>();
		initializeEntity();
	}

	protected Control<E> getControl() {
		return control;
	}

	protected void setControl(Control<E> control) {
		this.control = control;
	}

	public boolean isListing() {
		return listing;
	}

	public void setListing(boolean listing) {
		this.listing = listing;
	}

	public E getEntity() {
		return entity;
	}

	public void setEntity(E entity) {
		this.entity = entity;
	}

	public List<E> getListEntity() {
		return listEntity;
	}

	public void setListEntity(List<E> listEntity) {
```

```
			this.listEntity = listEntity;
		}

		public String getBusca() {
			return busca;
		}

		public void setBusca(String busca) {
			this.busca = busca;
		}

		public void list () {
			setListing(true);
			listEntity = control.getListAll(getEntity());
		}

		public void saveOrUpdate() {
			if (entity.getId() == 0) {
				control.save(entity);
			} else {
				control.update(entity);
			}
			initializeEntity();
			verifyListing();
		}

		public void delete() {
			control.delete(entity);
			verifyListing();
		}

		public void find() {
			setListing(true);
			listEntity = control.findByCriteria(getEntity(), busca);
		}

		private void verifyListing() {
			if (listing) {
				listEntity = control.getListAll(getEntity());
			}
		}

		public void cancel(ActionEvent event) {
			initializeEntity();
		}

		protected abstract void initializeEntity();

}
```

Tabela 25 - MB.java

Classe linda não? Essa é uma classe genérica com todos atributos e métodos comuns aos objetos ManagedBeans e também utiliza Generics.

Temos como atributos da classe o Control, um booleano que representa se estamos listando (listing), a Entity, uma listEntity e um String de busca.

O construtor da classe inicializa o Control e chama o método abstrato initializeEntity, e segue os Getters and setters dos atributos citados.

Temos como primeiro método comum da classe o list(), que seta o booleano de listagem (listing) como true e chama o control para carregar todos registros da entidade. O próximo é o saveOrUpdate() que verifica se estamos adicionando novo item (id = 0) ou se estamos apenas atualizando (id ≠ 0), assim que adicionamos ou atualizamos, é chamado o método initializeEntity e verifyListing, o primeiro é chamado pois depois de persistir o objeto, é necessário instanciar um novo para que possamos fazer mais adições, e o segundo é que se já estivermos listando os objetos, temos que atualizar a lista.

Como próximo método temos o delete(), que chama a classe inferior (control) e tenta deletar a entidade selecionada, verifica se está listando para atualizar a lista (verifyListing). Depois temos o find(), que seta o booleano listing como true e chama o Control para fazer a busca usando Criteria e enviando o objeto de entidade e o que foi digitado no campo de busca.

Temos, então, o método privado verifyListing(), ele simplesmente verifica se o booleano listing é true, se sim, chama o controlador para listar tudo novamente (caso haja alguma modificação, sempre chame o método para verificar se é necessário atualizar a lista).

O método cancel() tem como parâmetro um ActionEvent, que faz parte do faces.event, temos esse parâmetro para usarmos o ActionListenner do PrimeFaces (veremos melhor seu uso na criação das páginas). O método simplesmente chama o initializeEntity, isso porque quando pedimos para editar algum item e clicarmos em cancelar, o item selecionado para a edição ficará associado a entidade, e não terá como adicionarmos um novo, sempre ficará editando o último item que selecionarmos (depois de pronto o projeto, comente a chamada do initializeEntity e veja o que acontece).

Por ultimo obrigamos quem extenda a classe MB a implementar o método initializeEntity.

Vamos criar as classes que extendam MB. Será criada uma página por caso de uso, temos dois casos de uso: Manter Reino e Manter Filo. Então

teremos dois gerenciadores das entidades, sendo eles: ReinoMB e FiloMB. Crie a classe ReinoMB e extenda o MB<Reino>:

```
package br.ueg.portalVirtual.view.managed;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

import br.ueg.portalVirtual.model.Reino;

@ManagedBean
@SessionScoped
public class ReinoMB extends MB<Reino>{

	public ReinoMB() {
		super();
	}

	@Override
	protected void initializeEntity() {
		setEntity(new Reino());
	}

}
```

Tabela 26 - ReinoMB.java

De inicio temos duas annotation que definem o JSF, @ManagedBean e @SessionScoped, a primeira diz que a classe tem métodos que serão utilizados nas páginas, podemos até dizer que cada método é um comando de um botão da tela. Lembra que disse que o Spring pode gerenciar o JSF e a visão também? Se quiséssemos isso, em vez de @ManagedBean, usaríamos o @Controller, e teríamos que declarar a classe no XML, mas como já comentei, isso traz alguns problemas, por isso prefiro que o próprio JSF gerencie sua instancia. A annotation @SessionScoped define o escopo que usaremos a classe, os principais são: Session, Request, Application e View. Session mantém a classe sem perca dos parâmetros e dados durante toda a sessão (trabalharia como um Singleton de sessão, enquanto a sessão estiver aberta, tem apenas uma instancia da classe); o Request instancia a classe toda vez que a página é renderizada, ou seja, cada vez que você acessa a página tem uma nova instancia; Application segura a instancia durante a execução da aplicação, mas não recomendo o seu uso (falta de testes meu, e não vi muita gente usar =D); no escopo de visão trabalha semelhante ao Session.

Como setamos a annotation @ManagedBean, podemos usar essa classe nas páginas com o nome de reinoMB, veja que apenas a primeira letra da classe que é alterada pra minúscula, o resto continua normal, isso porque não definimos o "name" do ManagedBean, caso queira usar um outro adicione o atributo citado na annotation, exemplo @ManagedBean(name = "primeiroBean")

No construtor da classe, chamamos o construtor da classe pai apenas. E como extendemos o MB, devemos escrever a classe initializeEntity, e apenas inicializamos a entidade nesse método, mandando uma nova instancia de Reino para o objeto entity. Simples não? Todos os métodos (quase botões) estão implementados já no MB, basta apenas setar as diferenças: inicialização da entidade.

Vamos agora construir o gerenciado do caso de uso Manter Filo, crie, então, a classe FiloMB e extenda MB<Filo>:

```
package br.ueg.portalVirtual.view.managed;

import java.util.List;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.event.ActionEvent;

import br.ueg.portalVirtual.control.Control;
import br.ueg.portalVirtual.model.Filo;
import br.ueg.portalVirtual.model.Reino;

@ManagedBean
@SessionScoped
public class FiloMB extends MB<Filo> {

	private List<Reino> reinos;
	private Control<Reino> localControl;

	public FiloMB() {
		super();
		localControl = new Control<Reino>();
	}

	@Override
	protected void initializeEntity() {
		setEntity(new Filo());
	}

	public List<Reino> getReinos() {
		return reinos;
	}

	public void setReinos(List<Reino> reinos) {
```

```
            this.reinos = reinos;
      }

      public void listReino (ActionEvent event) {
            reinos = localControl.getListAll(new Reino());
      }

}
```

Tabela 27 - FiloMB.java

FiloMB difere do ReinoMB pela entidade Filo ter chave estrangeira de Reino, ou seja, quando formos cadastrar um Filo, teremos que escolhe um Reino, através de uma ComboBox que listará os reinos cadastrados. Devemos então ter uma lista de reinos, por isso existe o atributo reinos, do tipo lista de Reino e o localControl que pedirá a lista para o controle.

O construtor da super no construtor da classe pai e inicializa o controlador local (ou controlador do reino), depois escrevemos o método initializeEntity enviando uma nova instancia de Filo. Segue então o getter and setter de reinos.

Como último método temos o listReino que tem como parâmetro um ActionEvent, sendo, esse método, responsável por buscar a lista de reinos pelo controlador local e associá-la ao atributo reinos.

Temos os dois ManagedBeans funcionando com sua ações implementadas, devemos agora escrever as páginas.

Programando - Criando nosso layout

As páginas e todo o resto que tem haver com o desenvolvimento de sites ficam na pasta /WebContent (veja na figura 08 que definimos esse como o nome do diretório). Como pedi, já deve ter a estruturação de pastas "css", "images", "pages" e "templates", dentro de /images coloque as figuras de edição e exclusão: editar.png e lixeira.png. Essas imagens se encontram no CODE, mas se quiserem escolher outras fiquem à vontade, só não extrapolem o tamanho de 18 x 18 de dimensão para que a tabela de listagem não fique grande e nem alterem o nome delas.

Para definir os nossos layouts será utilizado o CSS, que formatar a página, altera fonte, cor, fundo entre outros, vamos então criar o nosso arquivo de configuração. Dentro da pasta /css crie o arquivo CSS template.css, New > Other... > CSS File:

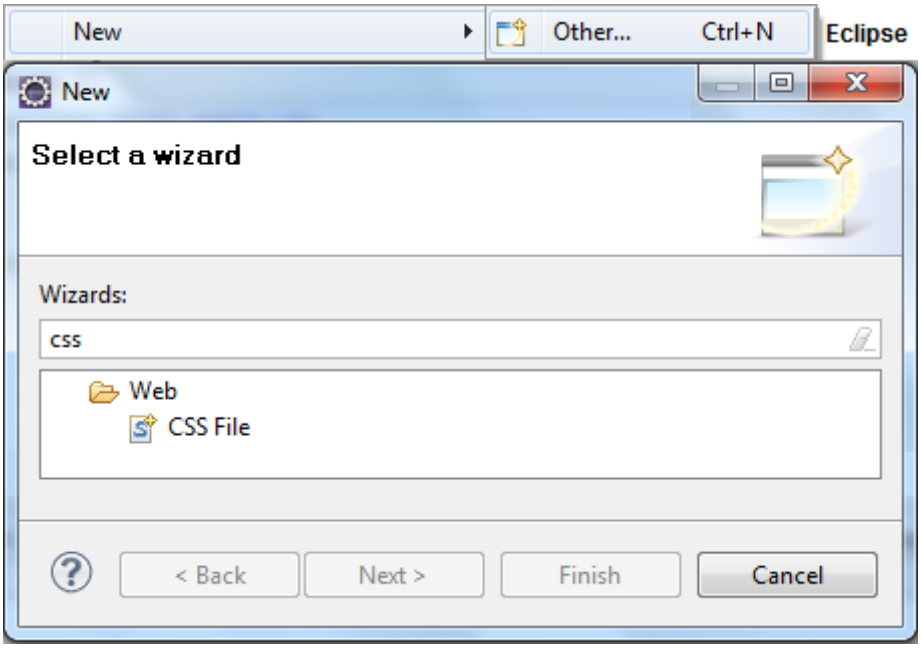


Figura 16 - New>CSS File

```
@CHARSET "ISO-8859-1";
html {
	height: 100%;
}
body  {
	margin: 1px;
	height: 100%;
	background: #ddd;
	margin-left: 5%;
	margin-right: 5%;
```

```
	}
.barColumn {
	width: 30%;
}
.contextColumn {
	width: 70%;
}
```

Tabela 28 - template.css

o CHARSET define a codificação de caracteres, o importante saber é que não usa UTF-8, sendo que o UTF-8 aceita "ç" e acentos, caso a página use ISSO-8859-1 e tiver algum acento, ele não será mostrado, mas como nosso arquivo CSS não usa acento e nem é visível não precisamos de tal codificação. Primeiro é definido o bloco <html />, com a altura de 100%, para definirmos que o html é o corpo total e dele se deriva os outros. Depois é modificado o <body> dando um espaço entre todas as margens de 1 pixel, utilizando 100% da altura, cor de fundo cinza (#ddd) e espaçamento de margem esquerda e direita de 5%, ou seja, será inutilizado 5% de cada lado da página.

Como html e body já são do próprio HTML, eles foram chamados sem nenhum caractere antes deles, os próximos tem um ".". Quando se tem um "." antes do nome, definimos um seletor tipo "class" e pode ser usado em qualquer elemento do HTML, além desse há também o seletor tipo id que antes vem um "#", esse seletor só pode ser usado uma única vez.

Os próximos elementos são barColumn e contextColumn, são apenas divisores, dando a um 30% da página e a outro 70% da página, veremos seu uso no template.xhtml.

Programando - Facelets e template

Facelets é um subprojeto do JSF mantido pela Sun, que tem por finalidade criar templates e componentes reutilizáveis. O uso de Facelets deixa a aplicação de 30 a 50% mais rápido. Ainda adiciono uma característica ao Facelets por minha conta e risco: com o uso desse "framework" trabalhamos a orientação a objetos na criação de páginas.

Podemos fazer várias coisas com Facelets, por isso ele foi integrado ao JSF 2.0, no 1.2 teríamos que adicionar alguns comandos no web.xml e faces-config.xml, mas com o 2.0 basta apenas criar um projeto e pedir uma nova página XHTML que ela já vem com Facelets.

Usaremos o Facelets para a criação do template, assim fixaremos o que não muda e as partes da páginas que são alteráveis deixaremos apenas esses "pedaço" para edição em uma nova página, para isso, crie na pasta /template o arquivo template.xhtml, clique em New ->HTML File e digite o nome template.xhtml (é necessário digitar a extensão também, senão será criado um arquivo HTML, se quiser que o "Wizard" pré-configure a página, clique em "Next" em vez de "Finish" e peça "New Facelet Template"):

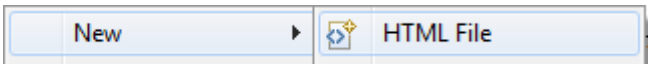


Figura 17 - New>HTML File

```
<!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:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.prime.com.tr/ui">
<h:head>
      <link rel="stylesheet" type="text/css"
href="../css/template.css" />
      <title>
            <ui:insert name="title">Teste</ui:insert>
      </title>
</h:head>

<h:body>

      <h:panelGrid >
            <ui:insert name="top">TOP</ui:insert>
            <h:panelGrid columns="2" columnClasses="barColumn,
contextColumn">
```

```
                    <h:column >
                          <ui:insert name="menuBar">
                                <p:menu>
                                      <p:submenu
label="#{msg.template_register}">
                                                  <p:menuitem
url="#{pageContext.servletContext.contextPath}/pages/CadReino.jsf"
value="${msg.reino_reinoColumn}"/>
                                                  <p:menuitem
url="#{pageContext.servletContext.contextPath}/pages/CadFilo.jsf"
value="#{msg.filo_filoColumn}"/>
                                      </p:submenu>
                                </p:menu>
                          </ui:insert>
                    </h:column>
                    <h:column>
                          <ui:insert name="context">CONTEXT</ui:insert>
                    </h:column>
              </h:panelGrid>
        </h:panelGrid>

</h:body>

</html>
```

Tabela 29 - template.xhtml

Como já explicado no início é definido o tipo de documento: XHTML 1.0. Na tag <html /> definimos as bibliotecas que usaremos nessa página "ui" é o Facelets, "h" é o HTML do JSF, "f" é o Faces ou JSF, e "p" o PrimeFaces. Definido os prefixos das tags, podemos utilizá-las a vontade, faça bom uso do CTRL + ESPAÇO.

Em HTML normal utilizaríamos o <head /> para definir o cabeçalho e o <body /> para definir o corpo, mas o JSF reimplementou essas tags dentro do "h", e toda página que contenha conteúdo JSF é preciso utilizá-la. No JSF 1.2 como não tinha ainda essa solução, usava-se o padrão <head /> <body /> e onde tinha JSF, era obrigatório ser colocado dentro da tag <f:view />.

Dentro do <h:head /> chamamos nosso layout pela tag <link />, dizendo que se trata de um arquivo CSS. Depois definimos um título para a página na tag <title />, sendo que dentro desta tag chamamos o Facelets para criar uma inserção de componentes, e dentro dele escrevemos "Teste". No caso, esses <ui:insert /> podem ser substituídos em outras páginas, e caso tente acessar a página sem substituí-los (o próprio template) aparecerá o que está dentro deles, ou seja, se você pedir para acessar o template.jsf o título será "Teste", se for outra página que tenha substituído o insert, será o novo texto.

Finalizamos o <h:head /> e iniciamos o <h:body />, e dentro do corpo criamos um painel do JSF que servirá como uma tabela para organização e estruturação da nossa página, para as pessoas que gostam de CSS, podem utilizar em vez do <h:panelGrid /> o no lugar. Veja que no panelGrid utilizamos o columnClasses e dentro dele colocamos o nome dos seletor tipo "class" criado no CSS, ou seja, teremos 2 colunas, a primeira com 30% de uso e a segunda com 70%.

Na primeira coluna criamos o menuBar, esse menu é fixo, por isso já o criei no template.xhtml, mas caso queira diferenciá-lo em alguma página, deixei o comando de <ui:insert /> antes dele. Dentro da coluna uso o PrimeFaces para criar um menu, qualquer comando dentro do JSF ou PrimeFaces e afins podemos utilizar String ou #{comando}, quando se usa #{} estamos chamando o Java, lembra que definimos no faces-config.xml o messages.properties e definimos um nome para a variável de "msg", então toda vez que utilizar #{msg} estarei abrindo a arquivo messages.properties.

Definimos o label (título) do <p:menu /> como "Cadastrar" em português, ou "Register" em inglês, que são os valores da chave template_register do arquivo messages.properties. Dentro desta tag criamos dois menuitens. O primeiro chamando a página de Manter Reino e o segundo a de Manter Filo (ainda as criaremos). Veja que é chamado os comandos do Java para buscar o contexto da aplicação com #{pageContext.servletContext.contextPath} (no Java podemos usar esse mesmo método: instanciando o PageContext, e depois teremos os métodos getServletContext().getContextPath()), ele retorna a URL ou onde está instalado a aplicação. Então passamos a localização das páginas: /pages/... Exemplo: Supondo que hospedamos o site no domínio www.site.com.br, esses "links" ficarão: www.site.com.br/pages/CadReino.jsf e www.site.com.br/pages/CadFilo.jsf. Depois definimos o nome que aparecerá para o item com o "value" também chamando o Bundle.

Menu criado, vamos para a próxima coluna, onde ficará o contexto, apenas coloquei o Facelets para inserir um componente na coluna. Temos então nosso template pronto.

Programando - Criando as páginas

Agora criaremos as páginas de CRUD, as duas são bastante parecidas, tendo mínimas diferenças, sendo a página do Filo um pouco mais complexa devido ao relacionamento com Reino, então mostrarei as duas, e explicarei só a última, para isso crie os HTML File CadReino.xhtml e CadFilo.xhtml dentro de /pages:

```
<!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:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.prime.com.tr/ui">

<ui:composition template="../template/template.xhtml">
      <ui:define name="title"> <h:outputText
value="#{msg.reino_title}"/> </ui:define>

      <ui:define name="context">
          <h:form>
                  <h:panelGrid columns="2">
                        <p:commandButton value="#{msg.reino_add}"
oncomplete="dlg.show();" update="formCad"/>
                  <p:commandButton value="#{msg.button_list}"
ajax="false" action="#{reinoMB.list}" update="listPanel"/>
                  <p:inputText value="#{reinoMB.busca}" />
                  <p:commandButton value="#{msg.button_search}"
ajax="false" action="#{reinoMB.find}"/>
            </h:panelGrid>
            <p:dataTable rendered="#{reinoMB.listing}"
value="#{reinoMB.listEntity}" var="reinos" rows="5"
emptyMessage="#{msg.reino_empty}" paginator="true">
                  <p:column headerText="#{msg.reino_reinoColumn}">
                        <h:outputText value="#{reinos.reino}"/>
                  </p:column>
                  <p:column headerText="#{msg.column_edit}">
                        <p:commandLink update="formCad"
oncomplete="dlg.show();">
                                          <f:setPropertyActionListener
target="#{reinoMB.entity}" value="#{reinos}"/>
                                    <p:graphicImage
value="#{pageContext.servletContext.contextPath}/images/editar.png"

      title="#{msg.column_edit}"  />
                        </p:commandLink>
                  </p:column>
                  <p:column headerText="#{msg.column_delete}">
                        <p:commandLink action="#{reinoMB.delete}"
ajax="false">
                                    <p:graphicImage
value="#{pageContext.servletContext.contextPath}/images/lixeira.png"
```

```
      title="#{msg.column_delete}"  />
                                    <f:setPropertyActionListener
target="#{reinoMB.entity}" value="#{reinos}" />
                        </p:commandLink>
                  </p:column>
            </p:dataTable>

            <p:messages showDetail="true" showSummary="false"/>

          </h:form>

          <p:dialog widgetVar="dlg" modal="true" closable="true"
header="#{msg.reino_header}" >
            <h:form id="formCad">
                  <h:panelGrid columns="2">
                        <h:outputText value="#{msg.reino_reino}"/>
                        <p:inputText value="#{reinoMB.entity.reino}"/>
                  </h:panelGrid>
                  <p:column>
                        <p:commandButton value="#{msg.button_save}"
action="#{reinoMB.saveOrUpdate}" ajax="false"/>
                        <p:commandButton value="#{msg.button_cancel}"
onclick="dlg.hide();" actionListener="#{reinoMB.cancel}"/>
                  </p:column>
            </h:form>
          </p:dialog>

      </ui:define>
</ui:composition>

</html>
```

Tabela 30 - CadReino.xhtml

```
<!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:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:p="http://primefaces.prime.com.tr/ui">

<ui:composition template="../template/template.xhtml">
      <ui:define name="title"> <h:outputText
value="#{msg.filo_title}"/> </ui:define>

      <ui:define name="context">
          <h:form>
                  <h:panelGrid columns="2">
                        <p:commandButton value="#{msg.filo_add}"
oncomplete="dlg.show();" actionListener="#{filoMB.listReino}"
update="formCad"/>
                  <p:commandButton value="#{msg.button_list}"
ajax="false" action="#{filoMB.list}" update="listPanel"/>
                  <p:inputText value="#{filoMB.busca}" />
                  <p:commandButton value="#{msg.button_search}"
ajax="false" action="#{filoMB.find}"/>
            </h:panelGrid>
```

```
            <p:dataTable rendered="#{filoMB.listing}"
value="#{filoMB.listEntity}" var="filos" rows="5"
emptyMessage="#{msg.filo_empty}" paginator="true">
                  <p:column headerText="#{msg.filo_filoColumn}">
                        <h:outputText value="#{filos.filo}"/>
                  </p:column>
                  <p:column headerText="#{msg.reino_reinoColumn}">
                        <h:outputText value="#{filos.reino.reino}"/>
                  </p:column>
                  <p:column headerText="#{msg.column_edit}">
                        <p:commandLink
actionListener="#{filoMB.listReino}" update="formCad"
oncomplete="dlg.show();">
                                    <f:setPropertyActionListener
target="#{filoMB.entity}" value="#{filos}"/>
                              <p:graphicImage
value="#{pageContext.servletContext.contextPath}/images/editar.png"

      title="#{msg.column_edit}"  />
                        </p:commandLink>
                  </p:column>
                  <p:column headerText="#{msg.column_delete}">
                        <p:commandLink action="#{filoMB.delete}"
ajax="false">
                              <p:graphicImage
value="#{pageContext.servletContext.contextPath}/images/lixeira.png"

      title="#{msg.column_delete}"  />
                                    <f:setPropertyActionListener
target="#{filoMB.entity}" value="#{filos}" />
                        </p:commandLink>
                  </p:column>
            </p:dataTable>

            <p:messages showDetail="true" showSummary="false"/>

          </h:form>

          <p:dialog widgetVar="dlg" modal="true" closable="true"
header="#{msg.filo_header}" >
            <h:form id="formCad">
                  <h:panelGrid columns="2">
                        <h:outputText value="#{msg.filo_filo}"/>
                        <p:inputText value="#{filoMB.entity.filo}"/>
                        <h:outputText value="#{msg.reino_reino}"/>
                        <h:selectOneListbox
value="#{filoMB.entity.reino}" id="selectReino" size="1"
converter="#{converter}">
                              <f:selectItem
itemLabel="#{msg.filo_select}" itemValue="" />
                              <f:selectItems value="#{filoMB.reinos}"
var="reinos" itemLabel="#{reinos.reino}" itemValue="#{reinos}"/>
                        </h:selectOneListbox>
                  </h:panelGrid>
                  <p:column>
                        <p:commandButton value="#{msg.button_save}"
action="#{filoMB.saveOrUpdate}" ajax="false"/>
                        <p:commandButton value="#{msg.button_cancel}"
onclick="dlg.hide();" actionListener="#{filoMB.cancel}"/>
                  </p:column>
```

```
            </h:form>
          </p:dialog>

      </ui:define>
</ui:composition>

</html>
```

Tabela 31 - CadFilo.xhtml

Começamos a página como todas as outras e definindo as bibliotecas utilizadas com a tag <html /> que são as mesmas utilizadas no template. Só que essas páginas não são mais páginas HTML comuns, e sim componentes do template, ou seja, elas derivam do template.xhtml, não necessitando de <h:head /> e <h:body /> que já foram setados no arquivo citado.

Para reutilizar o template.xhtml (olha a "orientação a objetos" que comentei) utilizamos a tag do Facelets <ui:composition /> passando o local onde está nosso template, veja que é utilizado "../" antes do local do arquivo, esses ".." seria um "comando" para voltar a página, no caso estamos dentro da aplicação, no diretório /pages/, e o template.xhtml está dentro da aplicação, mas no diretório /template/, então teremos que sair do /pages/ voltar para a pasta da aplicação e acessar /template, o comando "../template/" faz exatamente isso.

Primeiro definimos o título da página que era o <ui:insert name="title" /> e fazemos a substituição utilizando o <ui:define />, depois definimos o contexto, onde ficará o conteúdo.

Criamos um formulário para o conteúdo <h:form /> e um panelGrid com duas colunas, dentro desse painel adicionamos o botões "adicionar" e "listar" <p:commandButton /> o campo de busca <p:inputText /> e o botão de "buscar". No inputText apenas definimos a que variável será associado o texto digitado no campo, já o commandButton definimos o nome que aparecerá "value" e a ação sem Ajax "action" por isso devo desativar o Ajax para renderizar a página e chamar o método com sucesso "ajax=false" e se quisermos atualizar algum componente utilizamos o "update". Se não for colocado o "ajax=false" a página não será recarregada, e nossos métodos precisam atualizar os dados na tela sem Ajax, pois não fiz nenhum tratamento já que não há necessidade (mas pode ser feito). Usar

action não obriga a desativar o Ajax, nem usar o actionListenner é obrigatório estar com Ajax ativo, apenas faço isso dependendo da necessidade da ação recarregar a página ou não.

Quando queremos usar o Ajax, não recarregar a página e utilizar ações no Java, é bom atualizar o formulário onde serão alterados os dados, deixar o Ajax ativo e se possível usar o actionListenner. Veja o botão "adicionar", ele usa um actionListenner para popular uma lista com o reinos cadastrados, ao completar a ação (oncomplete) chama por JavaScript para mostrar o componente "dlg" e manda atualizar o formulário onde está o componente (formCad).

Criamos abaixo a tabela de listagem: dataTable, o parâmetro "rendered" pode receber "true" ou "false", e passo para ele o booleano listing do MB que será alterado de acordo com as ações para listar durante a execução. O "value" recebe a lista de entidade, e criamos uma variável para ela "var=filos", informamos a quantidade de linhas da tabela, qual será sua mensagem se não houver item na lista (utilizando o Bundle) e pedimos para exibir os paginadores. Dentro da tabela criamos colunas para exibir as características da entidade, no caso em análise, o nome do filo e a que reino pertence. Dentro do <p:column /> definimos também qual o cabeçalho da coluna que será exibido. As próximas colunas são os ícones de editar e deletar. O editar chama o mesmo componente para adicionar uma entidade, atualiza o mesmo formulário e também pede para carregar os reinos cadastrados, só que diferente do adicionar que a entidade é nova, devemos associar o item que queremos editar a variável entity do MB, além de utilizar o commandLink em vez do commandButton. Fazemos isso com o <f:setPropertyActionListenner /> que passa o item selecionado da lista para a variável entity. Dentro ainda do commandLink definimos a figura que irá aparecer (editar.png), o "title" do componente será a caixa de mensagem que irá aparecer caso fique com o mouse parado sobre a imagem.

O deletar funciona semelhante, mas ele não usa Ajax, ele seta a entidade selecionada no MB e chama a ação responsável por deletar, depois recarrega a página.

Depois da tabela criamos o componente <p:messages /> ele é o componente do JSF responsável por exibir as mensagem, lembra que no MessagesWeb mandamos adicionar no FacesContext algumas mensagens, se houver, o <p:messages /> será o encarregado de exibi-las, defini que as mensagens apareceram apenas o "detail" e não o "sumary". Com isso finalizamos o nosso formulário.

A próxima tag é o <p:dialog /> que seria uma "janela" bastante estilizada. O parâmetro "widgetVar" seria o nome por qual chamaremos esse componente por JavaScript (lembre do oncomplete=dlg.show()), colocando o modal como true, quer dizer que a janela ficará visível e a página ficará escurecida, dando foco ao p:dialog. Definimos que terá um botão para fechar (closable=true) e o texto que ficará como título da janela.

Criamos então outro formulário, o "formCad" que é atualizado quando clicamos em adicionar ou editar. Temos o panelGrid para criar a tabela, com duas colunas. Então definimos que o lado esquerdo ficará a descrição do campo, e no lado direito o campo. No primeiro inputText definimos que ele será associado a entity do MB no campo "filo", e depois vem o SelectOne.

Nesse SelectOne associamos o atributo que será selecionado ao campo reino do Filo, informamos que serão visualizado só um item, e que utilizaremos o converter que criamos para fazer a conversão para String e depois voltar para o Object (sem o converter vai haver problema ao associar o item selecionado). Como primeiro item que já virá selecionado, criamos o texto: "Selecione um" com a tag <f:selectItem /> e depois criamos a lista de itens com o <f:selectItens /> utilizando uma variável para a lista (var=reinos) para definir o que será exibido (itemLabel) e o que será enviado para o Java (itemValue).

Terminamos o panelGrid e criamos o botões para salvar e cancelar, cada um chamando sua respectiva ação. E agora finalizamos o projeto, vamos botar ele para executar.

Rodando a aplicação

Agora que nossa aplicação está finalizada, você pode aperta o "Play" do Eclipse, seguir os passos do "Wizard" e assim visualizará sua página, entretanto vou ensinar outro método, que enquanto seu servidor estiver iniciado, a aplicação estará rodando.

Para isso, la na aba "Server" clique com o botão direito sobre o nome do Tomcat: "Tomcat v7.0 Server at localhost [...]", e vá em "Add and Remove..":

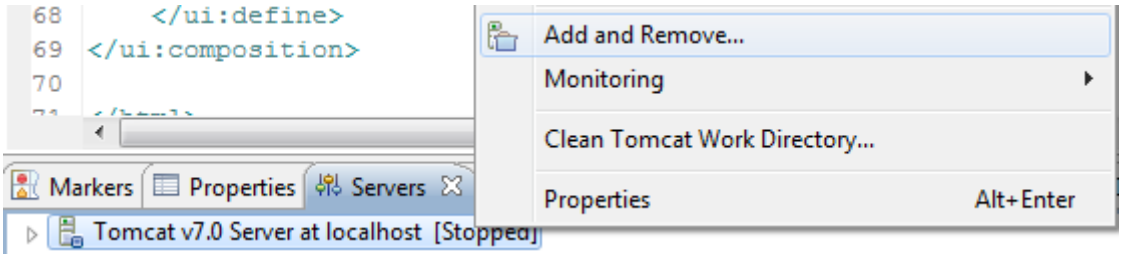


Figura 18 - Add and Remove...

Jogue o projeto do portal-virtual para o lado direito do "PickList" e clique em "Finish":

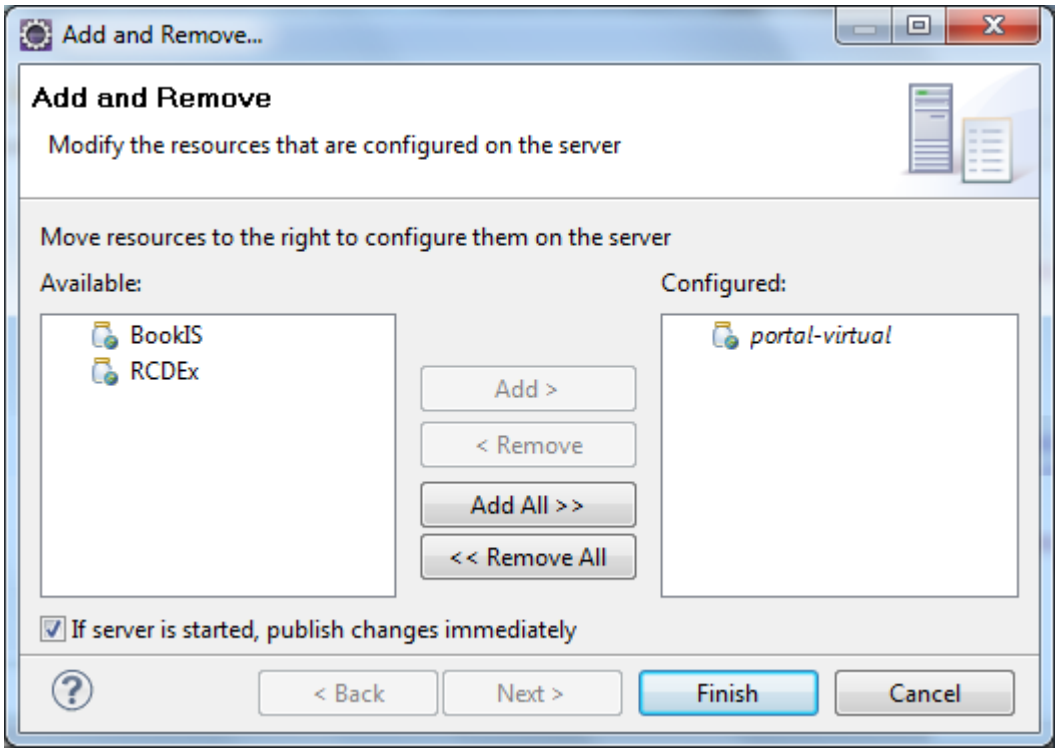


Figura 19 - PickList de projetos

Pronto, agora clique com o botão direito sobre o nome do Tomcat novamente e vá em “Start” ou clique no ícone de “Play” no canto direito, um pouco acima do nome do servidor, após alguns segundo deve estar no nome do Apache Tomcat: Tomcat v7.0 Server at localhost [Started].

Agora vá no seu navegador (menos o Internet Explorer, isso não é navegador) e digite: localhost:8080/portal-virtual. Você acessará o seu projeto em Java Web com JSF 2.0 com Spring, Hibernate com JPA, PrimeFaces e Facelets.

Exercício

Implemente o Caso de Uso Manter Classe, com essa arquitetura será bem fácil, faça alguns testes e novas validações, implemente uma classe de regra de negócio e a instancie no Control. Brinque para ver a diferença e importância dos detalhes.

Agradecimentos

Espero que façam bom proveito do Tutorial, que sirva como base de estudos para novos projetos e pesquisa, e que com ele possam desenvolver ótimas aplicações. Obrigado por ler o tutorial e ótimos estudos.

Agradeço a todos que colaboraram para a criação desse tutorial, tanto emprestando notebook quando o meu estragou quanto em incentivo para terminar o mesmo. E agradeço também todos aqueles que citei no tutorial como base de estudo.

Dúvidas, sugestões, críticas, correções, chingamentos ou qualquer coisa, entre em contato no e-mail: diego160291@msn.com, estarei à disposição.

Diego Carlos Rezende

Graduando em Sistemas de Informação

Universidade Estadual de Goiás - UnUCET

Bolsista de Iniciação Tecnológica Industrial do CNPq - Nível A