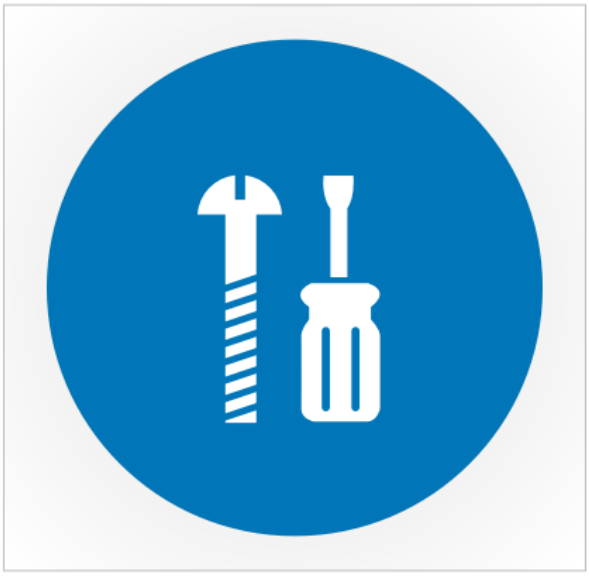
algaworks
softwares e treinamentos



DWJSF

Desenvolvimento Web com JavaServer Faces

www.algaworks.com

ALGAWORKS SOFTWARES E TREINAMENTOS

DWJSF – Desenvolvimento Web com JavaServer Faces

40 Horas/Aula

2ª Edição

Setembro/2010

www.algaworks.com

+55 (34) 3255-9898

treinamentos@algaworks.com

Av. Monsenhor Eduardo, 983, Sala 06 A, Bairro Bom Jesus
Uberlândia-MG, CEP. 38400-748

Sobre a empresa

A AlgaWorks é uma empresa localizada em Uberlândia/MG, que trabalha para fornecer treinamentos em TI de qualidade e softwares que realmente funcionam.

Estamos no mercado desde 2004 trabalhando com treinamentos e consultorias sobre a plataforma Java.

Nossa missão

Capacitar pessoas em tecnologias e metodologias de desenvolvimento de software e fornecer aplicativos na internet de baixo custo que contribuem para a organização da vida de milhares de usuários, micro e pequenas empresas de todo o mundo.

Nossa visão

Ser reconhecida como uma das principais empresas de treinamentos em TI do Brasil e como a principal exportadora de software web de baixo custo para indivíduos e pequenas empresas do mundo.

Nossos valores

- Honestidade
- Transparência
- Respeito
- Excelência
- Simplicidade

Origem de nosso nome

"As algas produzem energia necessária ao seu próprio metabolismo através da fotossíntese ... As algas azuis foram os primeiros serem vivos a aparecerem na Terra, e acredita-se que tenham tido um papel fundamental na formação do oxigênio da atfosmera."

A palavra em inglês “Works”, além de trabalho, pode ser traduzido como algo que funciona, que tem sucesso.

AlgaWorks foi o nome dado para a nossa empresa, e quer dizer que:

- Queremos ser uma empresa independente, que através de nosso próprio esforço conseguimos produzir uma energia suficiente para a nossa sobrevivência;
- Não queremos depender de investidores externos que não agregam conhecimento ao negócio;
- Desejamos criar alianças com empresas, autônomos, estudantes e profissionais e transmitir todo o nosso conhecimento em tecnologia;
- Fornecemos softwares que funcionam e agregam valor ao negócio, sem complicações;
- Tudo isso deve funcionar com sucesso para ajudar nossos clientes, parceiros, fornecedores, colaboradores e acionistas serem mais felizes.

Nossos cursos

Conheça nossos cursos abaixo ou acesse www.algaworks.com/treinamentos para mais informações.

Java Básico e Orientação a Objetos
[24 horas]

Java Avançado
[32 horas]

Migrando para o Java 5
[8 horas]

Desenvolvimento Web com HTML, CSS e JavaScript
[12 horas]

Desenvolvimento Web com Servlets e JSP
[36 horas]

Desenvolvimento da Camada de Persistência com Hibernate
[24 horas]

Desenvolvimento Web com JavaServer Faces
[40 horas]

Desenvolvimento de Relatórios com JasperReports e iReport
[24 horas]

Desenvolvimento Avançado com Java EE
[32 horas]

Preparatório para Certificação de Programador Java
[32 horas]

Modelagem UML
[28 horas]

Gerenciamento Ágil de Projetos com Scrum
[16 horas]

Sobre esta apostila

Esta apostila faz parte do material didático distribuído pela AlgaWorks para ministrar o curso **Desenvolvimento Web com JavaServer Faces**.

Apesar da distribuição desta apostila para fins **não-comerciais** ser permitida, recomendamos que você sempre prefira compartilhar o link da página de download (http://www.algaworks.com/treinamentos/apostilas) ao invés do arquivo, pois atualizamos nosso material didático constantemente para corrigir erros e incluir melhorias.

Para você realmente aproveitar o conteúdo desta apostila e aprender a tecnologia JavaServer Faces, é necessário que você já tenha conhecimento prévio da plataforma Java, Orientação a Objetos, banco de dados e desenvolvimento web com JSP.

Se você quiser se especializar ainda mais em JSF ou outras tecnologias, sugerimos que faça os cursos presenciais na AlgaWorks, pois, além de usarmos este material didático ou outros com qualidade semelhante, você ficará por dentro das últimas novidades e terá a chance de aprender com a experiência de nossos instrutores.

Licença desta obra

Encontrou erros ou gostaria de sugerir melhorias?

Se você encontrar algum erro neste material ou tiver alguma sugestão, crítica ou elogio, por favor, envie um e-mail para treinamentos@algaworks.com.

Sobre o autor

Thiago Faria de Andrade (thiago.faria@algaworks.com) é fundador e diretor da AlgaWorks. Iniciou seu interesse por programação aos 14 anos, quando desenvolveu um software para entretenimento e se tornou um dos mais populares no Brasil e outros países de língua portuguesa.

Graduado em Sistemas de Informação e certificado como programador Java pela Sun/Oracle (SCJP), Thiago trabalhou em consultorias para grandes empresas de Uberlândia/MG, co-fundou a OpenK Tecnologia e trabalhou como diretor de tecnologia e treinamentos nesta empresa por quase 6 anos.

Fundou a AlgaWorks após adquirir 100% das cotas da OpenK Treinamentos, e no ano de 2010, possuía experiência profissional de mais de 10 anos, sendo 7 dedicados à tecnologia Java.

Além de programar, Thiago ministra cursos, palestras e presta consultorias sobre Java para órgãos do governo, universidades, pequenas e grandes empresas e pessoas físicas, tendo acumulado aproximadamente 2.500 horas em salas de aulas e auditórios, até o ano de 2010.

Conteúdo do DVD

O DVD que acompanha o material de estudo do curso **DWJSF** contém arquivos úteis para o aprendizado.

Os seguintes arquivos e programas são disponibilizados através deste DVD:

- **Bibliotecas do Hibernate, JFreeChart, MySQL JDBC, Mojarra (implementação JSF da Sun), JSTL, Facelets, Richfaces, Commons BeanUtils, Commons Digester, Commons Logging, Log4j e SL4J Log4j**
[DVD]:\Bibliotecas

- **Documentação do Hibernate, JFreeChart, JSF, Facelets e Richfaces**
[DVD]:\Documentação

- **Eclipse, NetBeans, JBoss Tools, EMS SQL Manager for MySQL, Toad for MySQL e MySQL GUI Tools**
[DVD]:\Ferramentas

- **Servidor MySQL**
[DVD]:\MySQL Server

- **Kit de desenvolvimento Java**
[DVD]:\JDK

- **Apache Tomcat**
[DVD]:\Tomcat

** DVD distribuído apenas para clientes do curso.*

Índice

1. Objetivo do curso

As constantes evoluções da tecnologia exigem das pessoas um freqüente aperfeiçoamento de seus conhecimentos profissionais. No dia-a-dia, muitas pessoas não conseguem acompanhar as mudanças e novidades nas tecnologias.

Para estes profissionais, a **AlgaWorks** oferece treinamentos personalizados, ministrados por pessoas com larga experiência no mercado de software. O conteúdo programático de nossos cursos se adequam à realidade, abordando os temas com aulas presenciais realizadas com muita prática.

Tudo isso garante um investimento em TI que se traduz num diferencial competitivo para a empresa, pois sua equipe estará totalmente capacitada a extrair todo seu potencial.

Este curso tem o objetivo de apresentar a JavaServer Faces e alguns frameworks, bibliotecas e especificações, como Hibernate, Richfaces, Facelets, JAAS e JFreeChart, utilizados amplamente para desenvolvimento de sistemas em todo o mundo.

Após a conclusão deste curso, os alunos deverão estar aptos a:

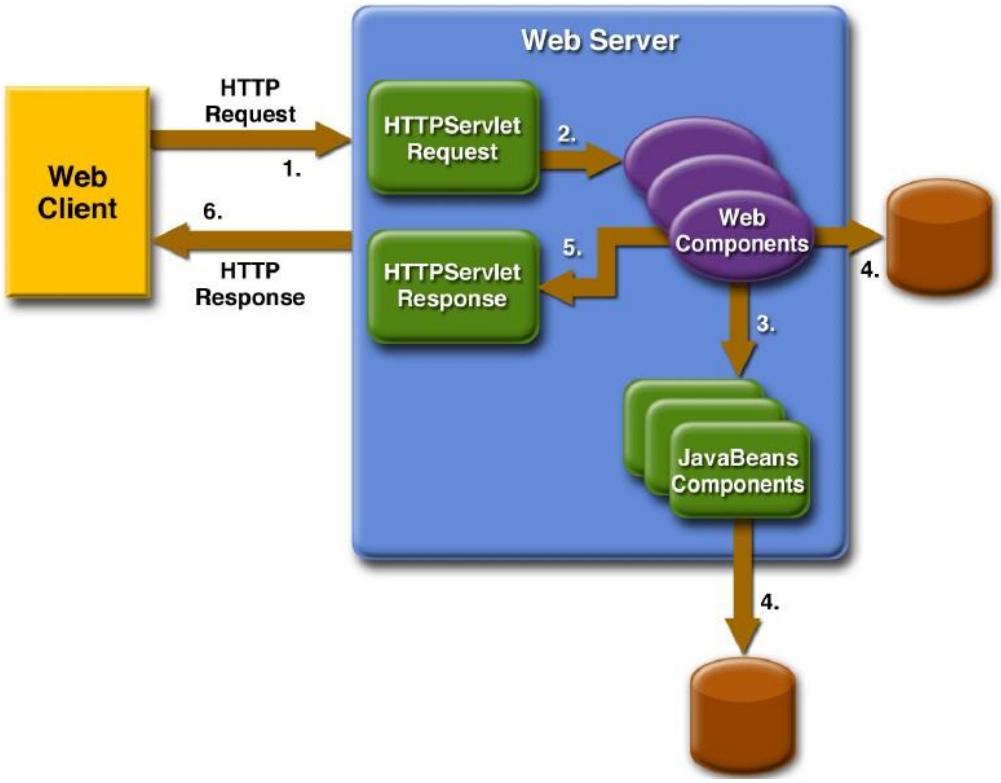
- Entender e criar managed beans;
- Entender o ciclo de vida do JSF;
- Utilizar navegação entre páginas;
- Entender e utilizar os principais componentes;
- Trabalhar com validação de campos;
- Criar páginas com internacionalização;
- Utilizar componentes de projetos open-source, como Richfaces;
- Criar sistemas que utilizem AJAX;
- Utilizar Facelets para templating de páginas JSF;
- Efetuar download e configuração do Hibernate;
- Mapear classes com Hibernate;
- Entender e utilizar o padrão de projeto DAO;
- Desenvolver classe de DAO genérico;
- Recuperar, gravar e consultar objetos com Hibernate;
- Criar gráficos com dados dinâmicos usando JFreeChart;
- Implementar sistemas que suportem autenticação e autorização usando JAAS;
- Desenvolver um projeto completo usando boas práticas de programação, padrões de projetos e frameworks de mercado, com separação em camadas.

2. Introdução

Com o avanço da tecnologia sobre redes de computadores e com o crescimento da internet, as páginas web estão se tornando cada vez mais atraentes e cheias de recursos que aumentam a interatividade com o usuário. São verdadeiras aplicações rodando sobre a web.

Quando falamos em aplicações web, estamos nos referindo a sistemas onde toda a programação fica hospedada em servidores na internet, e o usuário (cliente) normalmente não precisa ter nada instalado em sua máquina para utilizá-los, além de um navegador (browser). O acesso às páginas (telas) desses sistemas é feita utilizando um modelo chamado de *request-response*, ou seja, o cliente solicita que alguma ação seja realizada (request) e o servidor a realiza e responde para o cliente (response). Na plataforma Java, esse modelo foi implementado inicialmente através da API de Servlets e JSP (JavaServer Pages). Um Servlet estende a funcionalidade de um servidor web para servir páginas dinâmicas aos navegadores, utilizando o protocolo HTTP. Já o JSP, utilizando-se de uma sintaxe especial, permite que desenvolvedores web criem páginas que possuam programação Java, semelhante ao PHP e ASP, porém muito mais robusto e padronizado.

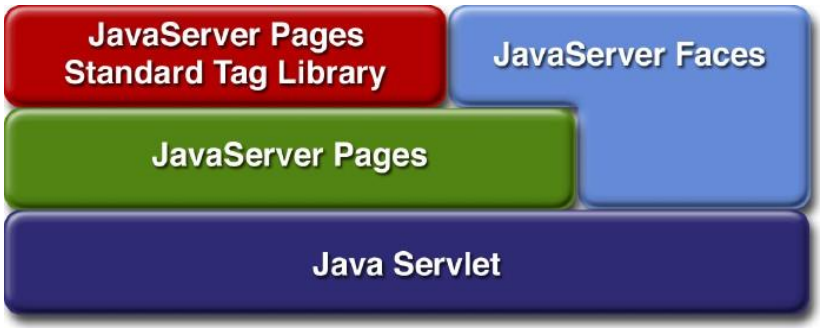
A interação entre o cliente e uma aplicação web é ilustrada no diagrama abaixo.



Uma aplicação web em Java

O cliente envia uma requisição HTTP ao servidor web. No mundo Java os servidores web são chamados de Servlet Container, pois implementam a especificação de Servlet e JSP. O servidor converte a requisição (request) em um objeto do tipo *HttpServletRequest*. Este objeto é então passado aos componentes web, que podem interagir com JavaBeans ou mesmo um banco de dados, para que possa ser gerado um conteúdo dinâmico. Em seguida, o componente web gera um objeto *HttpServletResponse*, que representa a resposta (response) ao cliente. Este objeto é utilizado para que o conteúdo dinâmico gerado seja enviado ao navegador (usuário).

Desde o lançamento de Servlets e JSPs, outras tecnologias Java e frameworks foram surgindo com o objetivo de melhorar a produtividade e recursos no desenvolvimento de aplicações web. Atualmente JavaServer Faces é a tecnologia do momento, requisitada na maioria das oportunidades de emprego para desenvolvedores Java. Esta tecnologia foi baseada em outras já existentes, como pode ver na imagem abaixo.



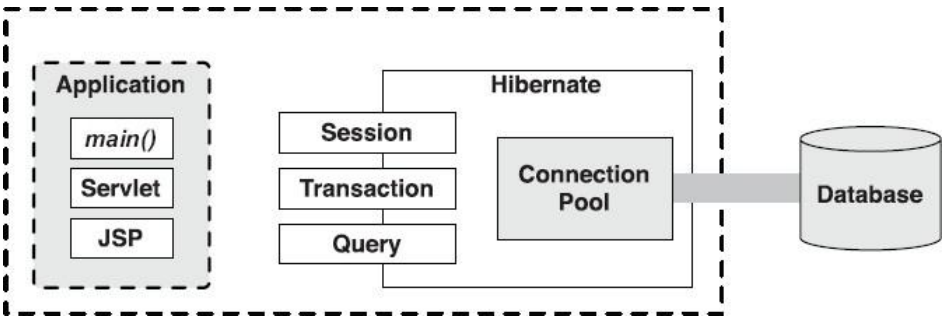
Veja que Servlet é a base de todas as aplicações Java para web. As outras tecnologias criam um nível de abstração sobre Servlets, criando aplicações mais fáceis de manutenção, escaláveis e robustas. Vários outros frameworks baseados em Servlets foram desenvolvidos sobre propriedade de alguma empresa ou open-source, e muitos deles fizeram sucesso por muito tempo, mas JSF revolucionou o desenvolvimento web e agora está em grande expansão.

Você aprenderá como instalar, configurar e desenvolver sistemas utilizando JavaServer Faces, a tecnologia que foi criada para que o desenvolvimento de aplicações web fosse parecido com o de aplicações clientes desktop em termos de produtividade e funcionalidades. O conceito básico deste framework é a existência de classes que representam os componentes (formulários, botões, campos de entrada, tabelas e etc) de sua página JSP. Veremos muitos mais detalhes sobre esta tecnologia adiante.

Em aplicações corporativas, é extremamente importante gravar os dados de cadastros, transações e etc. Os bancos de dados vieram para facilitar a vida dos programadores. Hoje existem diversos sistemas gerenciadores de banco de dados (SGBD) no mercado, uns trazendo facilidades de administração, outros com suporte a grandes quantidades de dados, alguns totalmente gratuitos e outros muito caros! Enfim, o importante é saber que, como programador, você certamente terá que utilizar algum banco de dados.

Como veremos nesta apostila, existem frameworks que facilitam a vida de um programador orientado a objetos (sim, para você estar lendo esta apostila, você deve ser um programador orientado a objetos) para trabalhar com banco de dados.

O Hibernate é um excelente framework que trabalha entre o banco de dados e sua aplicação Java. A figura abaixo ilustra como isso ocorre.



Com o Hibernate, ganhamos muito em agilidade e tornamos nossos produtos muito mais flexíveis, pois como veremos, fica muito fácil trocar o banco de dados inteiro com quase nenhum esforço.

O objetivo desta apostila é lhe ensinar JavaServer Faces, porém ensinaremos também como criar uma aplicação usando JSF e Hibernate. Você já deve conhecer pelo menos o básico de Hibernate. Abordaremos também outros padrões e conceitos importantes no desenvolvimento de software, como por exemplo, o DAO (Data Access Object), utilizado na camada de persistência, o famoso MVC (Model, View e Controller) e outros.

O que nós desejamos é que você tenha um ótimo aprendizado e que entre mais preparado neste mercado de trabalho, que necessita rapidamente de profissionais cada vez mais qualificados.

Sinta-se a vontade em voltar a esta apostila sempre que tiver dúvidas. Nos primeiros dias você poderá achar alguns códigos estranhos, mas ao longo do tempo vai ver que tudo se encaixa perfeitamente!

2.1. O que é JavaServer Faces?

JavaServer Faces, também conhecido como JSF, é uma tecnologia para desenvolvimento web que utiliza um modelo de interfaces gráficas baseado em eventos. Esta tecnologia foi definida pelo JCP (Java Community Process), o que a torna um padrão de desenvolvimento e facilita o trabalho dos fornecedores de ferramentas ao criarem produtos que valorizem a produtividade no desenvolvimento de interfaces visuais, inclusive com recursos de *drag-and-drop*.

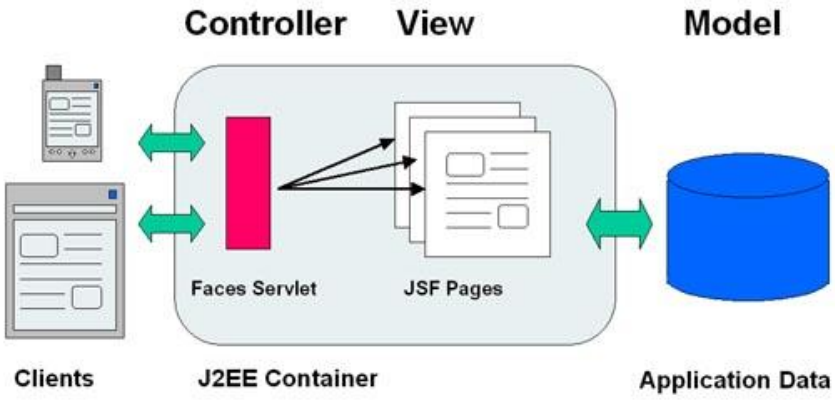
JSF é baseado no padrão de projeto MVC (Model-View-Controller), o que torna o desenvolvimento de sistemas menos complicado, pois a separação entre a visualização e regras de negócio é muito clara.

O padrão MVC separa o sistema em três responsabilidades (modelo, visualização e controle), onde o modelo é responsável por representar os objetos de negócio, manter o estado da aplicação e fornecer ao controlador o acesso aos dados. A visualização é responsável pela interface do usuário. Ela que define a forma como os dados são apresentados e encaminha as ações do usuário para o controlador. O controlador é responsável por ligar o modelo e a visualização, interpretando as solicitações do usuário, traduzindo para uma operação no modelo (onde são realizadas efetivamente as mudanças no sistema) e retornando a visualização adequada à solicitação.

Em JSF, o controle é feito através de um servlet chamado *Faces Servlet*, por arquivos XML de configuração e por vários manipuladores de ações e observadores de eventos. O *Faces Servlet* recebe as requisições dos usuários na web, redireciona para o modelo e envia uma resposta. Os arquivos de configuração possuem informações sobre mapeamentos de ações e regras de navegação. Os manipuladores de eventos são responsáveis por receber os dados da camada de visualização, acessar o modelo e devolver o resultado para o usuário através do *Faces Servlet*.

O modelo é representado por objetos de negócio, que executa uma lógica de negócio ao receber dados oriundos da camada de visualização.

A visualização é composta por uma hierarquia de componentes (*component tree*), o que torna possível unir componentes para construir interfaces mais ricas e complexas.



Arquitetura do JavaServer Faces baseada no MVC

2.2. Principais componentes

O verdadeiro poder de JavaServer Faces está em seu modelo de componentes de interface do usuário, onde aplicações são desenvolvidas com um conjunto de componentes que podem ser renderizados em diversas formas para vários tipos de clientes, como por exemplo para um browser da web, celular, etc.

JSF é similar à tecnologia proprietária ASP.NET, pois o modelo de componentes oferece alta produtividade aos desenvolvedores, permitindo a construção de interfaces para web usando um conjunto de componentes pré-construídos, ao invés de criar interfaces inteiramente do zero. Existem vários componentes de visualização JSF, desde os mais simples, como um *outputLabel*, que apresenta simplesmente um texto, ou um *dataTable,* que representa dados tabulares de uma coleção que pode vir do banco de dados.

A especificação do JSF fornece um conjunto de componentes visuais básicos em sua implementação de referência. Ela inclui duas bibliotecas de componentes, como a “HTML”, que possui componentes que representam diversos elementos HTML e a biblioteca “Core”, que é responsável por tarefas comuns no desenvolvimento de sistemas, como internacionalização, validação e conversão de dados de entrada. Além de fornecer uma biblioteca base de componentes, a API da JSF suporta a extensão e criação de novos componentes, que podem fornecer funcionalidades adicionais.

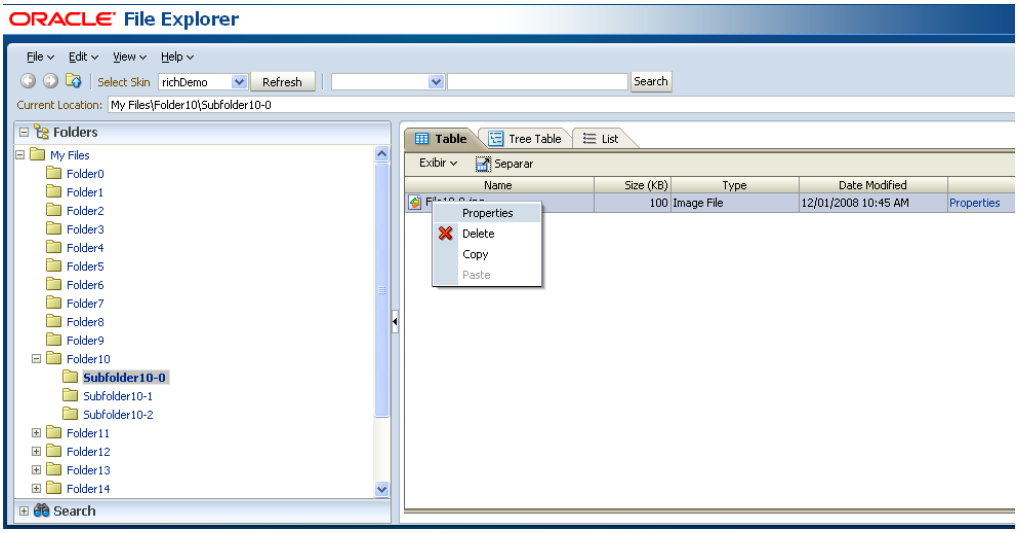
Os principais componentes que a implementação de referência do JSF fornece são:

- Formuário
- Campos de entrada de texto
- Campo de entrada de senha
- Rótulos com textos
- Textos de uma única linha
- Links
- Botões
- Mensagens
- Painéis
- Tabela HTML
- Tabela de dados (grid)
- Coluna de tabela de dados
- Etc

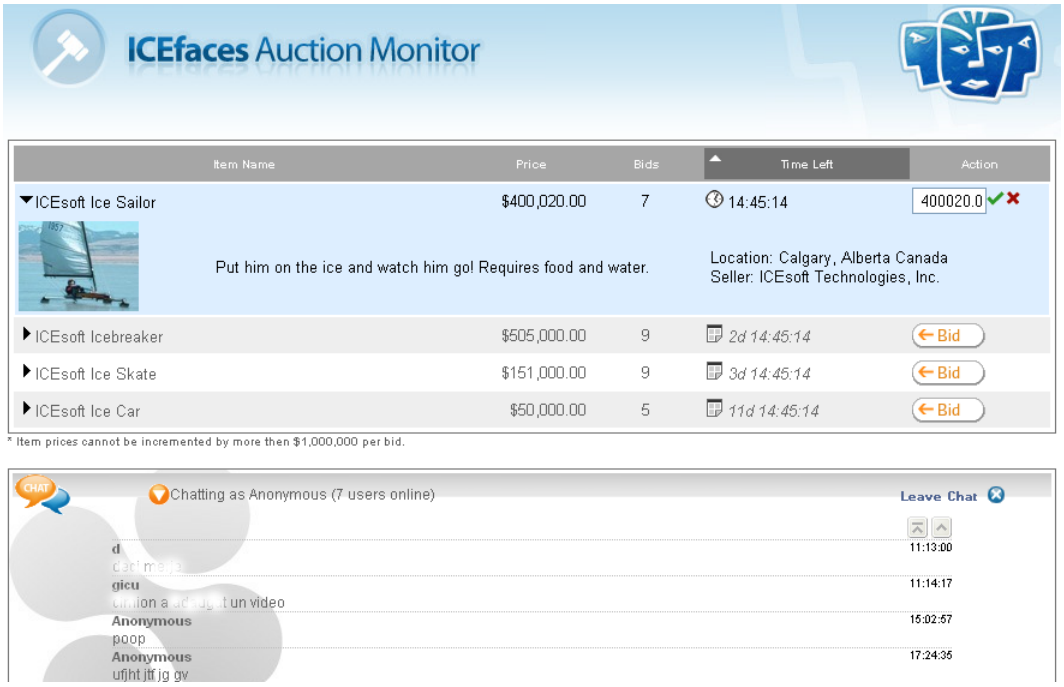
2.3. Componentes adicionais

Atualmente existem diversas empresas que trabalham na criação de componentes personalizados. Como exemplo, podemos citar a Oracle, com o ADF Faces Rich Client, IceSoft, com o IceFaces, JBoss (Redhat), com o Richfaces, e etc. Muitos desses componentes incluem múltiplos renderizadores para diversos tipos de clientes, tabelas avançadas, componentes para captura de datas e cores, menus suspensos, botões, barras de progressão, telas suspensas para seleção de valores e etc.

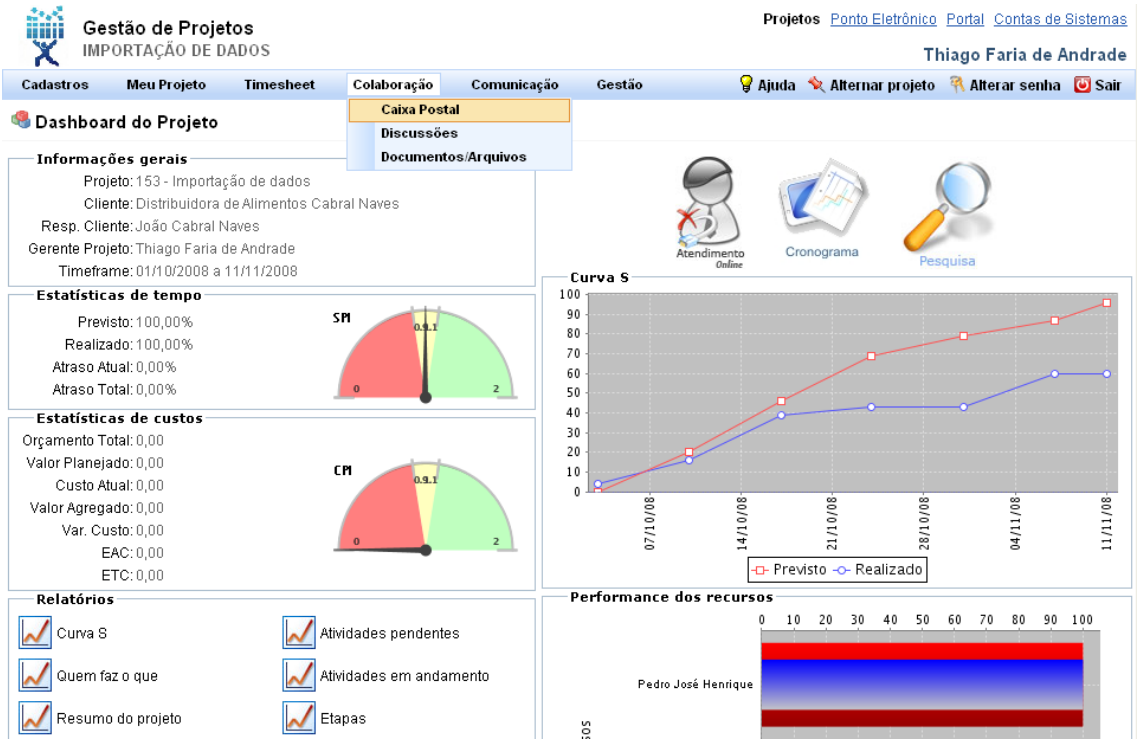
Veja alguns exemplos de sistemas usando diversas bibliotecas de componentes:



Aplicação de exemplo usando ADF Faces Rich Client



Aplicação de exemplo usando IceFaces



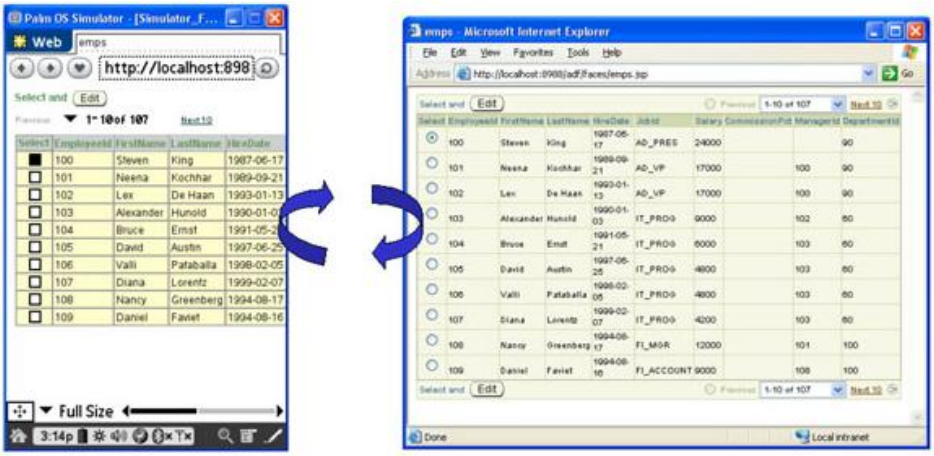
Software de gestão de projetos usando Richfaces

2.4. Renderização de componentes

A tecnologia JSF possui capacidade de utilizar renderizadores de componentes plugáveis. Isso se traduz na capacidade dos componentes se apresentarem diferentes dependendo do tipo do cliente que está sendo utilizado. Por exemplo, um browser como o

Firefox ou Internet Explorer, deve receber código HTML para que possa ser exibido ao usuário, enquanto um dispositivo móvel como um celular poderá ser capaz de exibir apenas conteúdos WML.

Um mesmo componente JSF pode se adequar aos dois tipos de conteúdos, dependendo do que for adequado no momento. Isso é possível graças ao desacomplamento dos componentes visuais do código de renderização, tornando possível criar múltiplos renderizadores para um mesmo componente. Diferentes renderizadores podem então ser associados ao componente e em tempo de execução é decidido qual renderizador será utilizado, baseado no tipo do cliente que enviou a requisição.

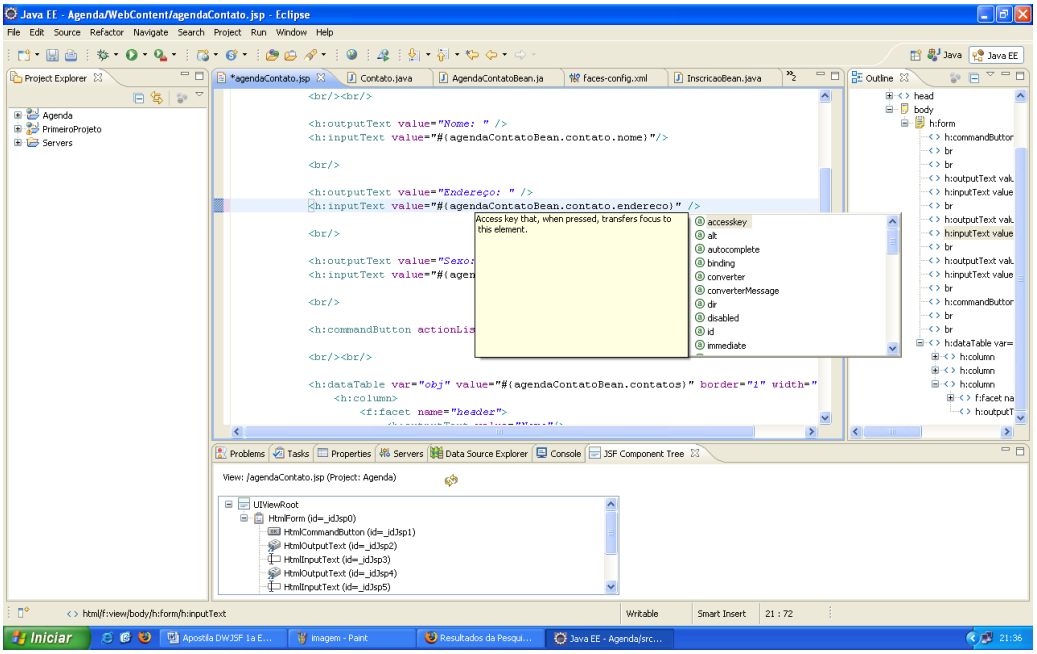


Um componente de tabela de dados sendo renderizado para clientes WML e HTML

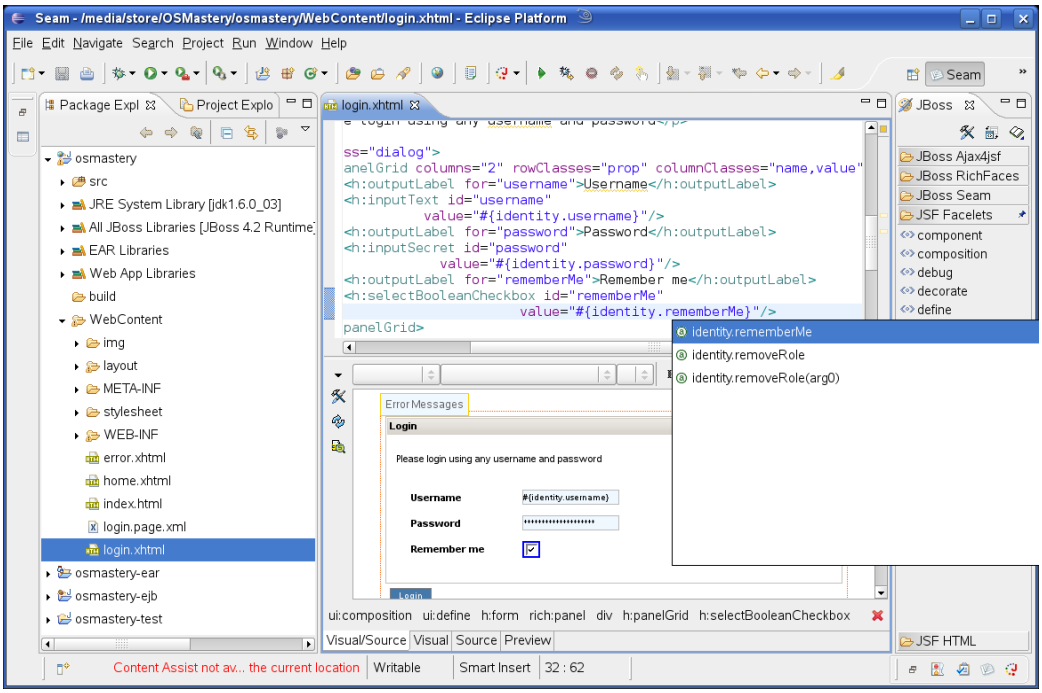
3. Ambiente de desenvolvimento

3.1. Escolhendo a ferramenta de desenvolvimento

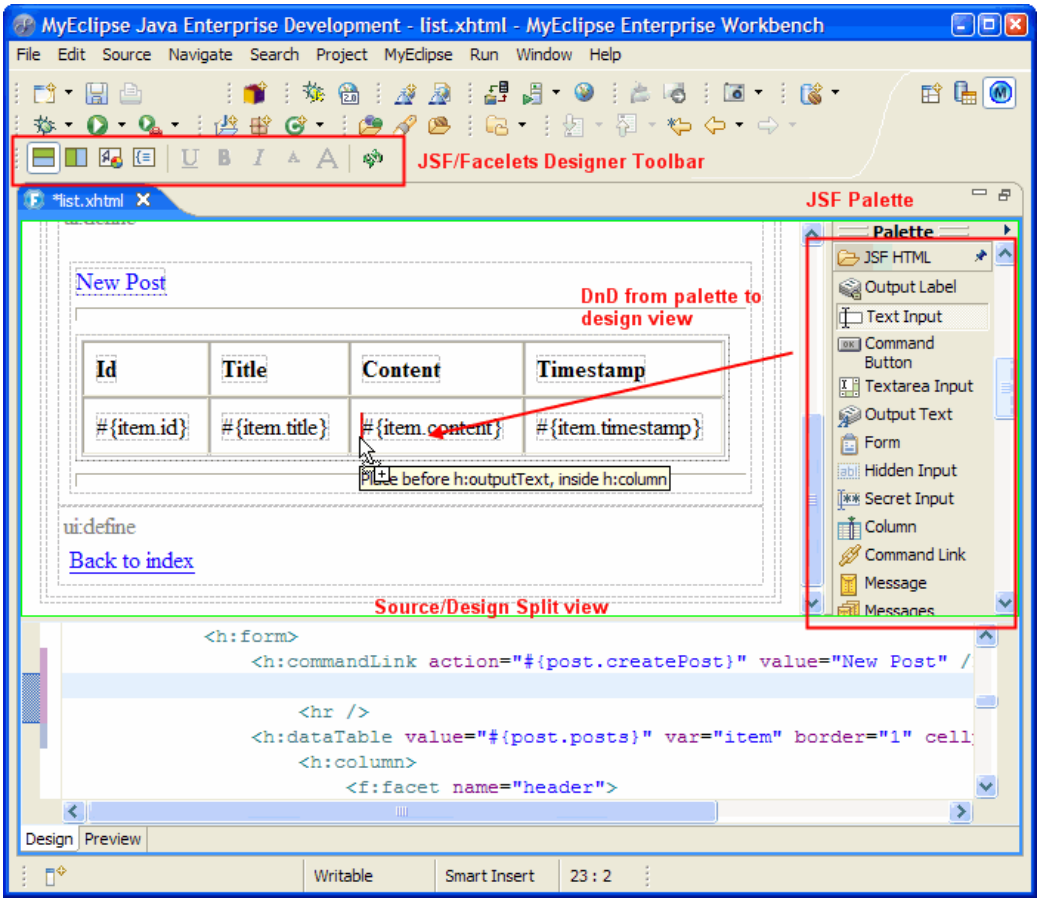
Atualmente existem várias ferramentas que suportam o desenvolvimento com JSF. Algumas delas possuem um ambiente visual que exibe uma representação gráfica da tela em desenvolvimento e permite arrastar e soltar componentes a partir de uma paleta. Outras ferramentas apóiam mais na edição do arquivo de configuração e na escrita de códigos e templates.



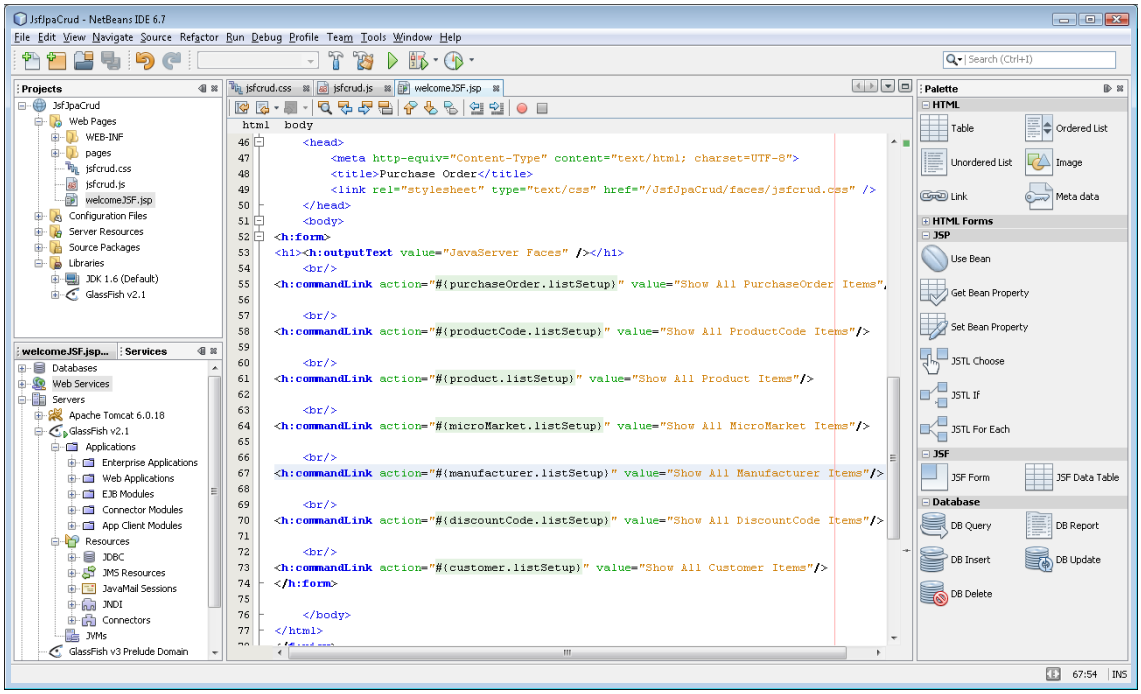
Eclipse



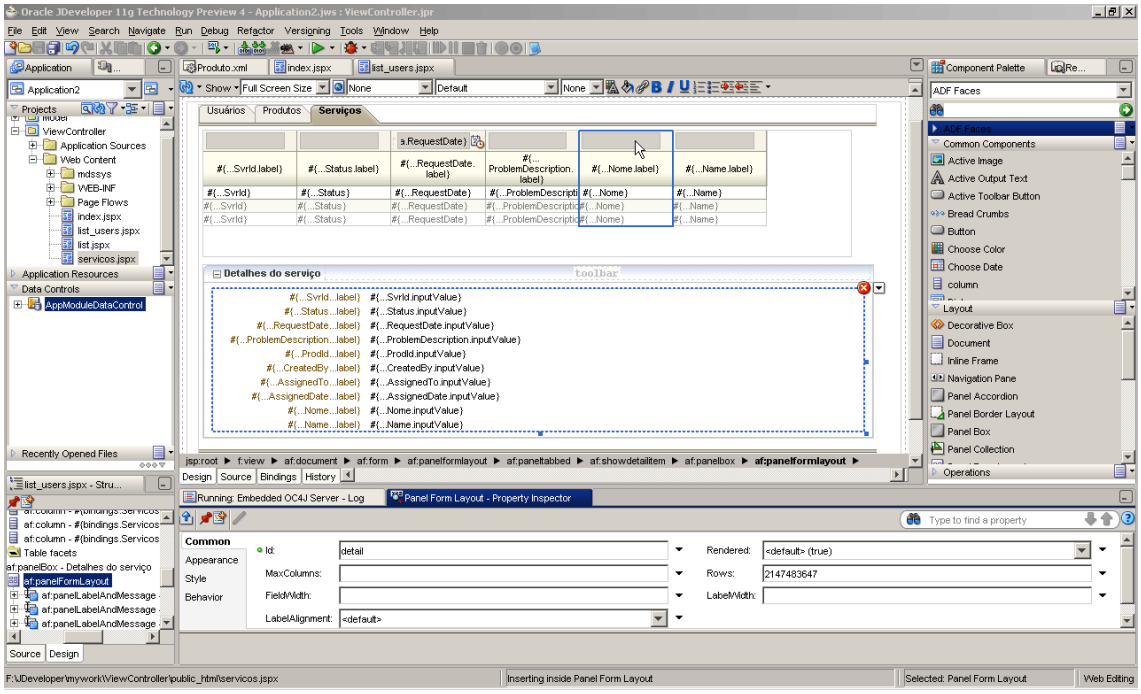
Eclipse com plugins do JBoss Tools



MyEclipse IDE



NetBeans



JDeveloper

Várias ferramentas são baseadas no Eclipse, como o JBoss Developer Studio (que é basicamente o Eclipse + JBoss Tools) e o MyEclipse IDE, pois o Eclipse é considerado uma plataforma muito flexível e atualmente a mais usada em todo o mundo.

Disponibilizamos os arquivos de instalação das IDEs grátis mais importantes na comunidade no DVD que acompanha esta apostila, porém utilizaremos o Eclipse para desenvolver os exemplos e exercícios da apostila.

O Eclipse distribuído para desenvolvedores Java EE não possui recursos visuais interessantes para criar páginas JSF, porém apóia visualmente na configuração da aplicação nesta tecnologia. Nossa escolha pelo Eclipse é pelo fato de ser uma ferramenta rápida e com excelente suporte a edição de código Java e JSP (isso inclui JSF).

Antes de terminar nosso estudo, instalaremos o plugin JBoss Tools para trabalharmos com a biblioteca Richfaces, e aí tiraremos proveito da funcionalidade “arrastar-e-soltar”.

O Eclipse pode ser baixado em http://www.eclipse.org, mas felizmente já possuímos o arquivo de instalação no DVD. A instalação dessa ferramenta será apresentada em um próximo tópico.

3.2. Escolhendo um container web

Para executar aplicações desenvolvidas em JSF, você precisará de um servidor com um container web instalado. Um container web é um programa que recebe requisições HTTP, executa componentes Java (Servlets) e devolve para o usuário (browser) código HTML, além de todos os outros recursos necessários (como imagens, vídeos, folhas de estilo e etc). Na verdade, trata-se de um servidor web (como o Apache HTTPD), porém com outras funcionalidades para executar código Java de acordo com a especificação Java EE.

A JavaServer Faces foi construída baseada na tecnologia de Servlets e JSP, mas você não é obrigado a saber sobre elas para desenvolver em JSF, apesar de ser recomendado.

Existem diversos containers web (e outros containers completos, que também não deixam de ser web) disponíveis para download, e o melhor, sem custo algum para implantar em produção. Apesar de tantas ofertas gratuitas de excelentes produtos, algumas empresas ainda vendem licenças de seus próprios servidores, pois oferecem suporte diferenciado ao cliente e normalmente implementam funcionalidades que os servidores gratuitos talvez não possuam.

Dentre os vários servidores que são containeres web, podemos destacar os mais conhecidos:

- Apache Tomcat
- Jetty
- JBoss AS
- Glassfish
- JOnAS
- Apache Geronimo
- IBM WebSphere Application Server
- Oracle WebLogic Server (antigo BEA WebLogic Server)
- Oracle Application Server

Para desenvolver nossos códigos, usaremos o Apache Tomcat, pois é mais leve, grátis e possui tudo que precisamos para executar nossos exemplos.

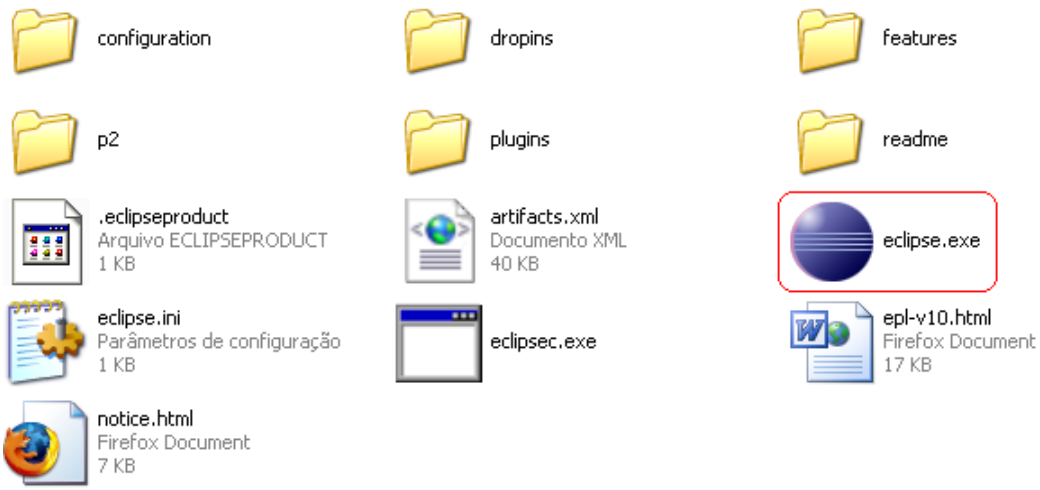
Como estes servidores são baseados nas especificações da tecnologia Java EE, você pode implantar os softwares que desenvolveremos neste curso em qualquer servidor mais atual compatível com a Java EE.

O download do Apache Tomcat pode ser feito em http://tomcat.apache.org. O DVD que acompanha a apostila já possui o arquivo de instalação desse servidor, e aprenderemos a instalá-lo ainda neste capítulo.

3.3. Instalando e configurando o Eclipse

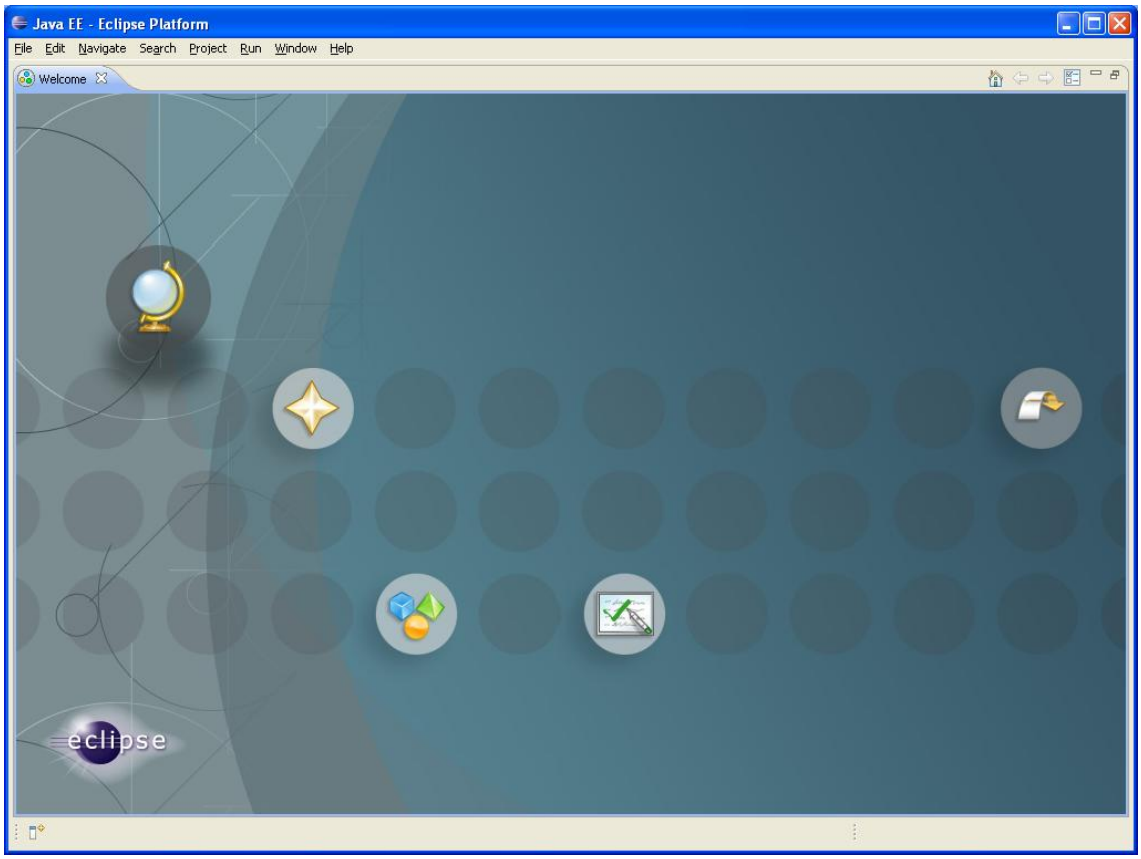
A instalação do Eclipse é simples, basta descompactar o arquivo de instalação (com extensão .zip) no local desejado.

Para iniciar o Eclipse, você já deve ter a Java SDK 5 ou superior instalada em seu computador. Execute o arquivo “eclipse.exe” na pasta “eclipse” que foi descompactada e a ferramenta será iniciada.



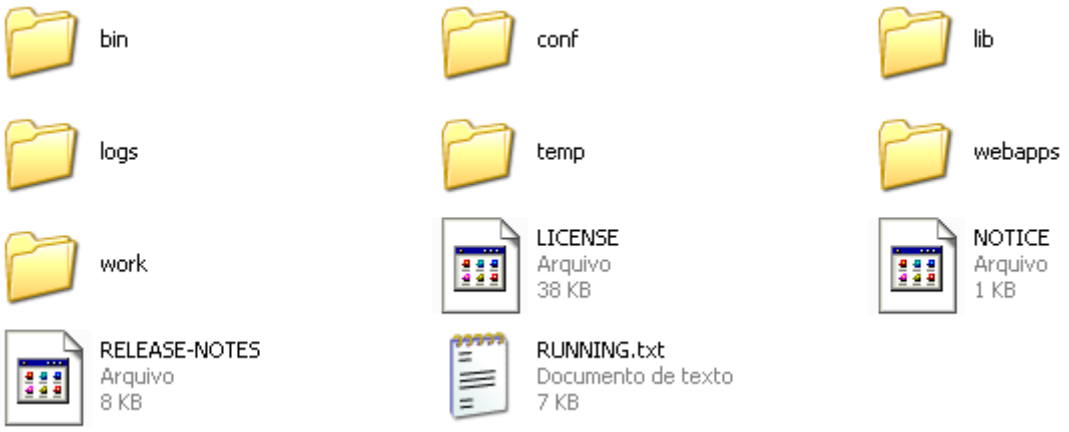
Aparecerá uma tela solicitando um diretório onde o Eclipse gravará seus projetos, que é chamado de *workspace*. Pressione o botão *OK* para usar o diretório sugerido ou selecione outro local.

Aguarde o carregamento e a tela “Welcome” aparecerá, indicando que a instalação ocorreu com sucesso.



3.4. Instalando e configurando o Apache Tomcat

O processo de instalação do Apache Tomcat é simples, basta descompactar o arquivo com extensão ".zip" no local desejado.



Uma vez finalizado, tem-se um servidor de aplicações pronto para produção. De qualquer forma, o site disponibiliza toda a documentação necessária para resolver problemas encontrados e esclarecer dúvidas com relação ao processo de instalação e configuração do servidor.

Para entender um pouco sobre o funcionamento do Tomcat, examine os diretórios criados durante o processo de instalação. Os principais são:

bin Executáveis, incluindo os aplicativos para iniciar e para encerrar a execução do servidor.

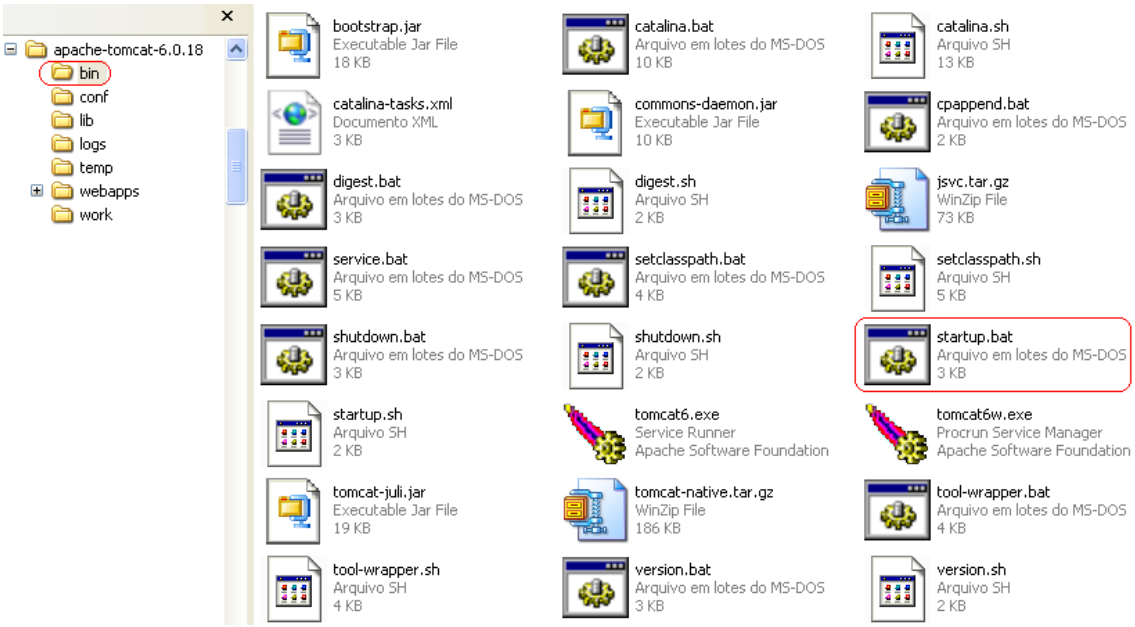
conf Arquivos de configuração do Tomcat. O arquivo "server.xml", em particular, define uma série de parâmetros para a execução do servidor, como por exemplo, a porta onde o servidor irá receber requisições (essa porta é, por *default*, 8080), devendo ser examinado com cuidado e modificado conforme as necessidades.

logs Arquivos de log do servidor. Além de gerar arquivos de log contendo entradas para cada requisição recebida, como qualquer servidor web, o Tomcat também pode gerar arquivos de log com tudo o que as aplicações desenvolvidas enviam para a saída padrão do sistema.

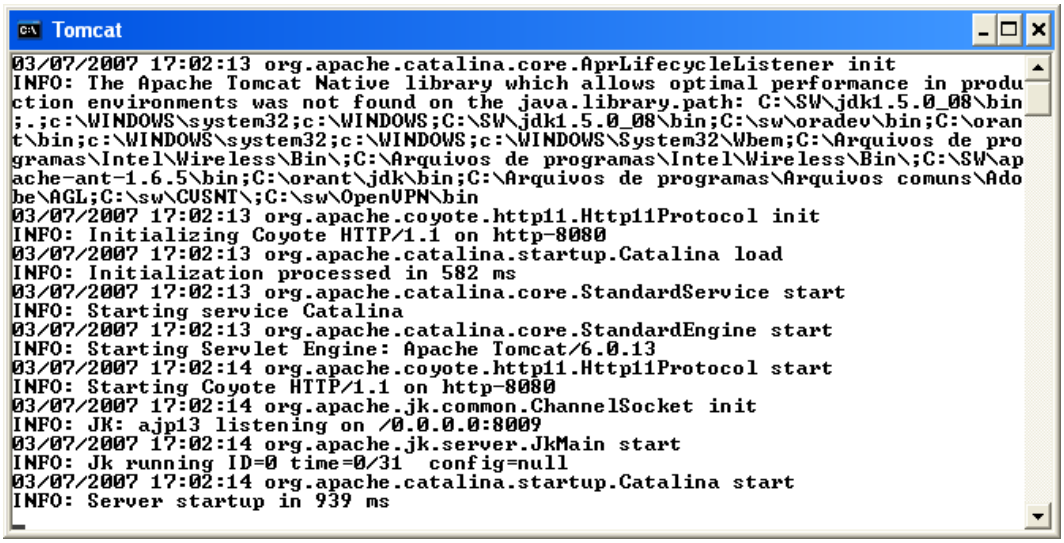
work Diretório temporário do Tomcat. Esse diretório é utilizado, por exemplo, para realizar a recompilação automática de páginas JSP.

webapps Nesse diretório são instaladas as diversas aplicações web desenvolvidas.

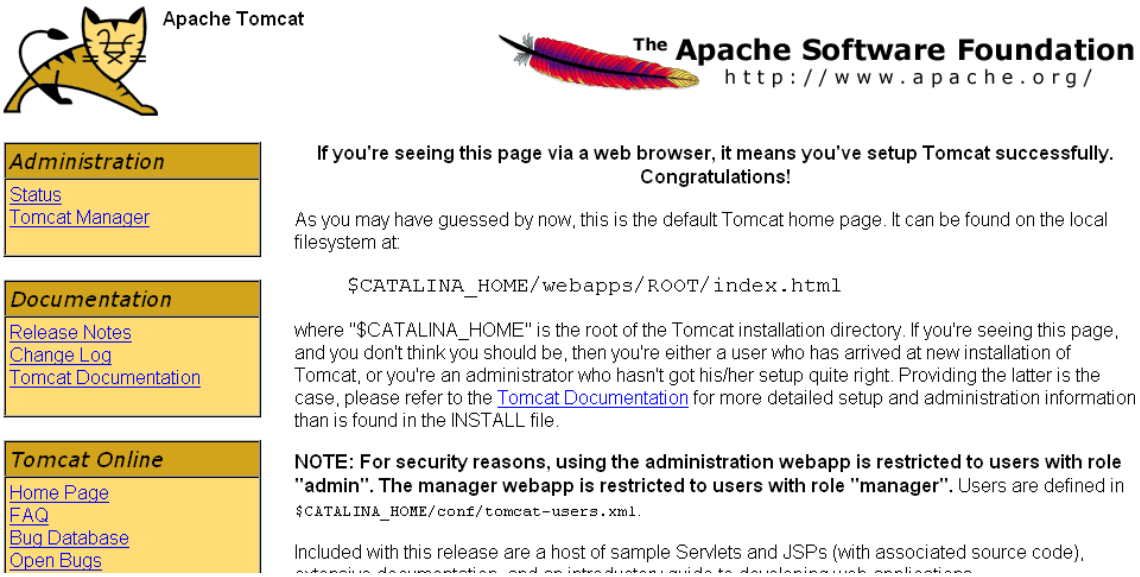
Para verificar se a instalação foi bem sucedida, acesse o diretório "bin" e execute o arquivo "startup.bat" (ou "startup.sh" para Linux) para iniciar o servidor. Você já deve ter a Java SDK 5 ou superior instalada em seu computador para efetuar este passo.



Uma console será exibida e os passos da inicialização do servidor serão impressos nela.



Abra um browser (pode ser o Firefox, que é melhor que o Internet Explorer) e acesse o endereço http://localhost:8080. Se a tela abaixo aparecer para você, parabéns, o Tomcat está instalado e funcionando em seu computador.

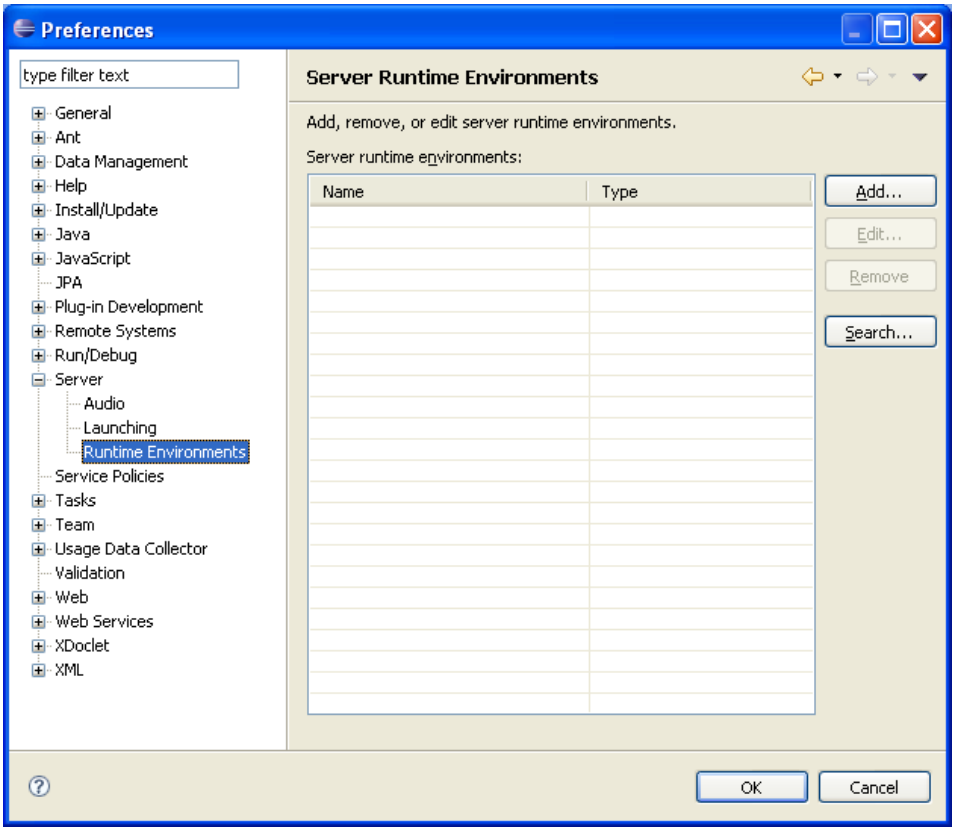


Agora interrompa a execução do Tomcat executando o arquivo “shutdown.bat” (“shuwdown.sh” no Linux), ou na console do servidor, pressione em conjunto as teclas *Ctrl* + *C*.

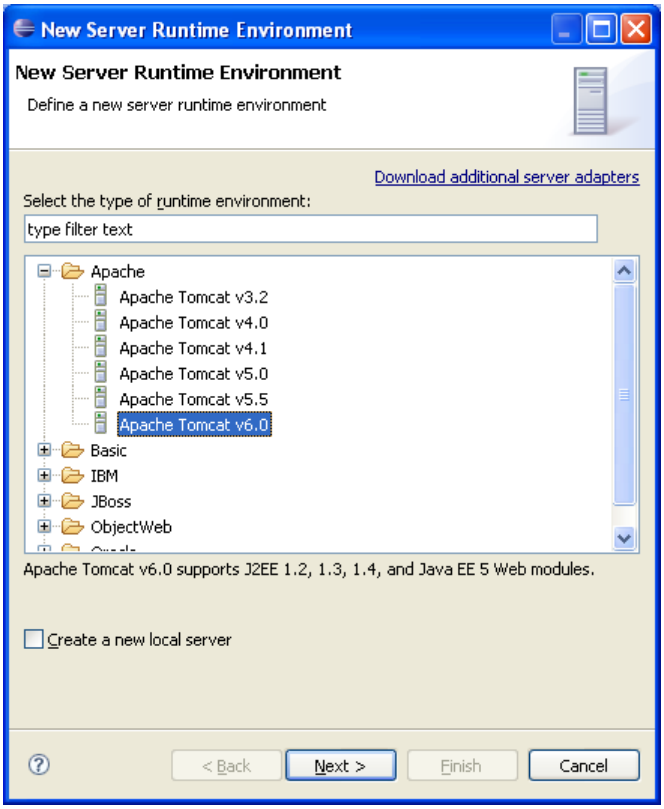
3.5. Integrando o Eclipse com o Apache Tomcat

Para tornar nosso estudo mais prático, iremos integrar o Apache Tomcat ao Eclipse. Desta forma, você poderá iniciar e parar o Tomcat e implantar as aplicações a partir do ambiente de desenvolvimento.

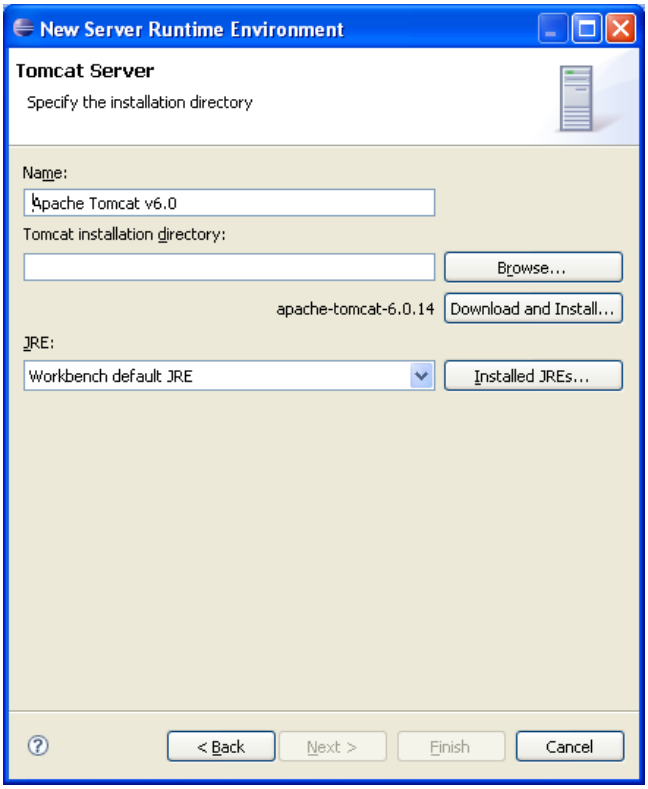
Acesse o menu “Window”, “Preferences”. Na tela que se abre, clique em “Server” e depois em “Runtime Environments”.



Clique sobre o botão “Add...”. A tela “New Server Runtime Environment” será aberta para seleção do servidor.

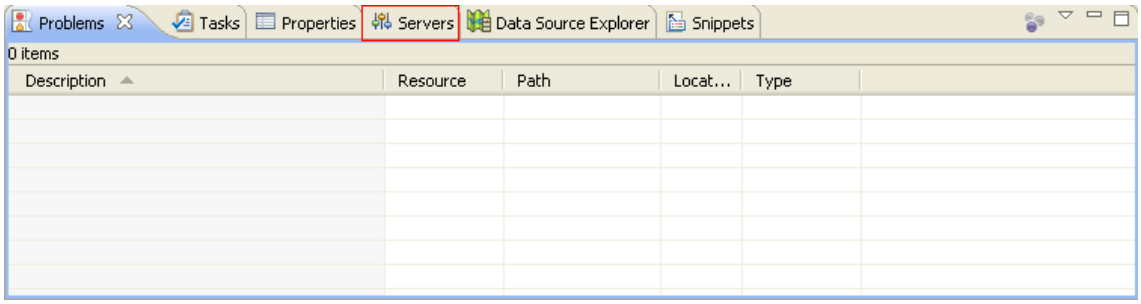


Localize e selecione a versão que mais se aproxima da que usaremos, neste caso, “Apache Tomcat v6.0”. Pressione “Next >”.

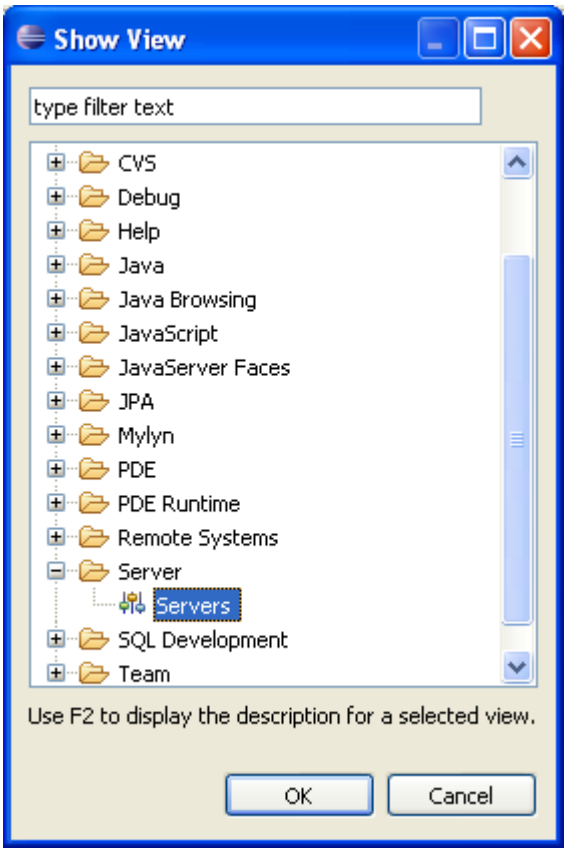


Clique sobre o botão “Browse...” e localize o diretório raiz onde o Apache Tomcat foi instalado, em seguida pressione “Finish” e depois “OK”.

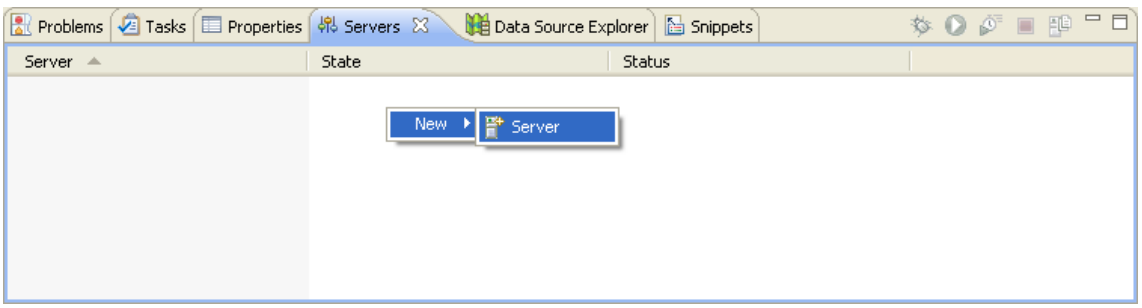
Para testar a integração do Eclipse com o Tomcat, localize e clique sobre aba “Servers”.



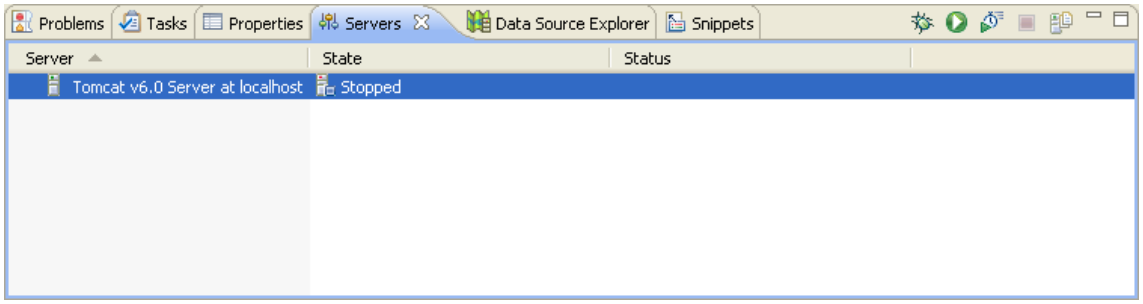
Se não encontrar esta aba ou ela estiver desaparecido, acesse o menu “Window”, “Show view” e “Others”. Localize a opção “Servers” e clique em “OK”.



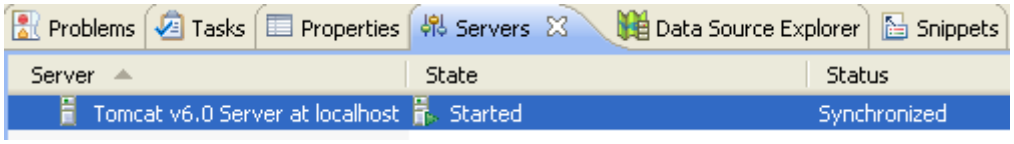
Pronto. Agora clique com o botão direito na área branca, selecione “New” e clique em “Server”.



Selecione a opção “Tomcat 6.0 Server” e clique em “Finish”.



O Tomcat aparecerá na lista de servidores integrados ao Eclipse. Clique com o botão direito sobre este servidor e selecione a opção “Start”.



Neste momento, o Apache Tomcat está funcionando integrado ao Eclipse. Você pode parar, configurar ou publicar aplicações no servidor a partir do ambiente de desenvolvimento.

4. Primeiro projeto JSF

4.1. Introdução

Nossa primeira aplicação em JSF será bem simples. O objetivo é que você consiga preparar o ambiente de um novo projeto e deixe-o funcionando no servidor.

Você poderá ficar com várias dúvidas do tipo “como funciona isso?”, “porque tive que escrever aquilo?”. Não se preocupe! Nos próximos capítulos aprofundaremos em cada detalhe da tecnologia.

4.2. Escolhendo uma implementação de JSF

A JSF foi criada através do *Java Community Process* (JCP), que é uma entidade formada pelas mais importantes empresas de tecnologia do mundo e especialistas em diversos assuntos.

O JCP é composto por vários grupos de trabalho, que são chamados de JSR (*Java Specification Request*). Uma JSR é um projeto de desenho de uma nova tecnologia. O artefato produzido através das JSRs são documentações, interfaces e algumas classes que especificam como deve funcionar um novo produto.

A JSF foi criada e é controlada pelo JCP através de JSRs (uma para cada versão). Quando uma JSR é finalizada, empresas fornecedoras de tecnologia têm a chance de entender a especificação e implementar um produto final compatível com o proposto pela especificação.

No caso da JSF, as implementações mais conhecidas atualmente são a da própria Sun, conhecida como Mojarra, e a da Apache, chamada de MyFaces Core.

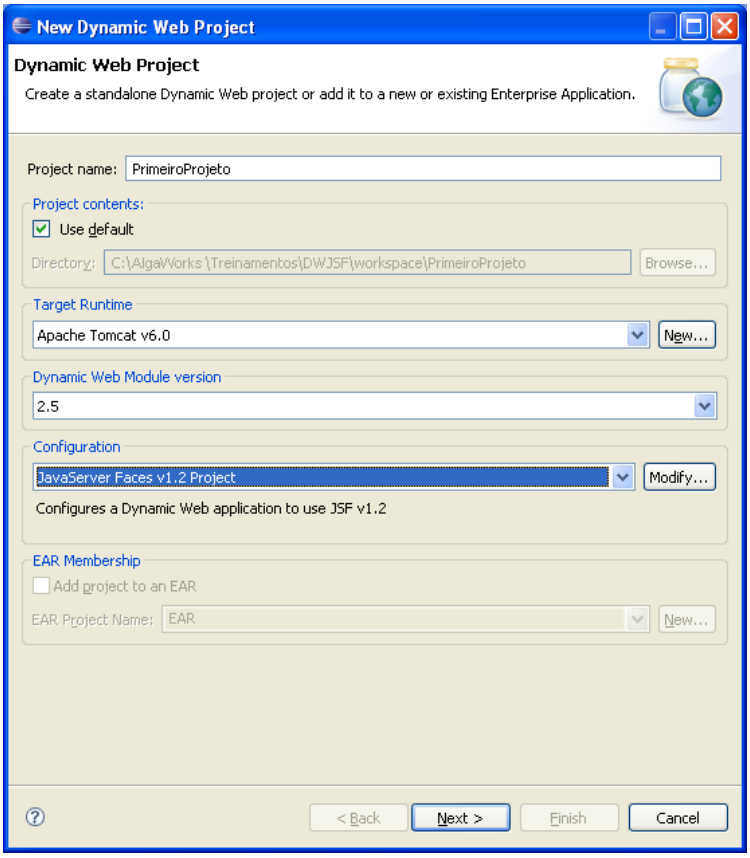
Todas as implementações devem fazer a mesma coisa, mas o que diferencia uma da outra é a qualidade que foi desenvolvido o produto, documentação disponível, suporte oferecido pela empresa e principalmente pela comunidade, etc.

Usaremos neste curso a implementação da Sun, a Mojarra versão 1.2.x. O DVD que acompanha esta apostila possui o arquivo distribuído pela Sun, mas caso você tenha interesse em conhecer o site e até mesmo baixar a última versão disponível, acesse https://javaserverfaces.dev.java.net/.

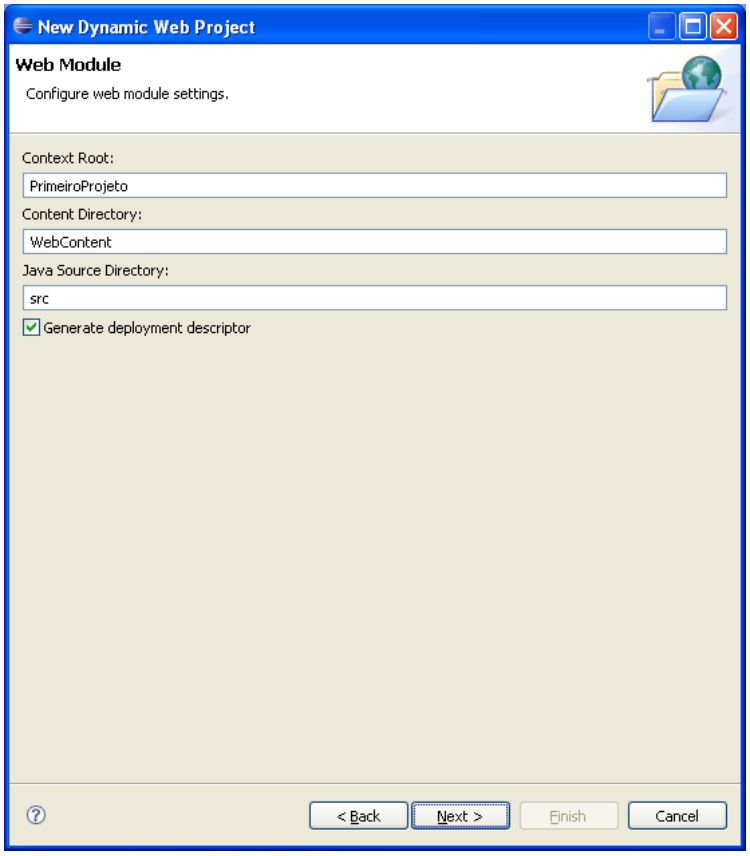
4.3. Criando um novo projeto

Para começar, crie uma nova aplicação web no Eclipse, selecionando a opção “File”, “New” e “Dynamic Web Project” no menu da ferramenta.

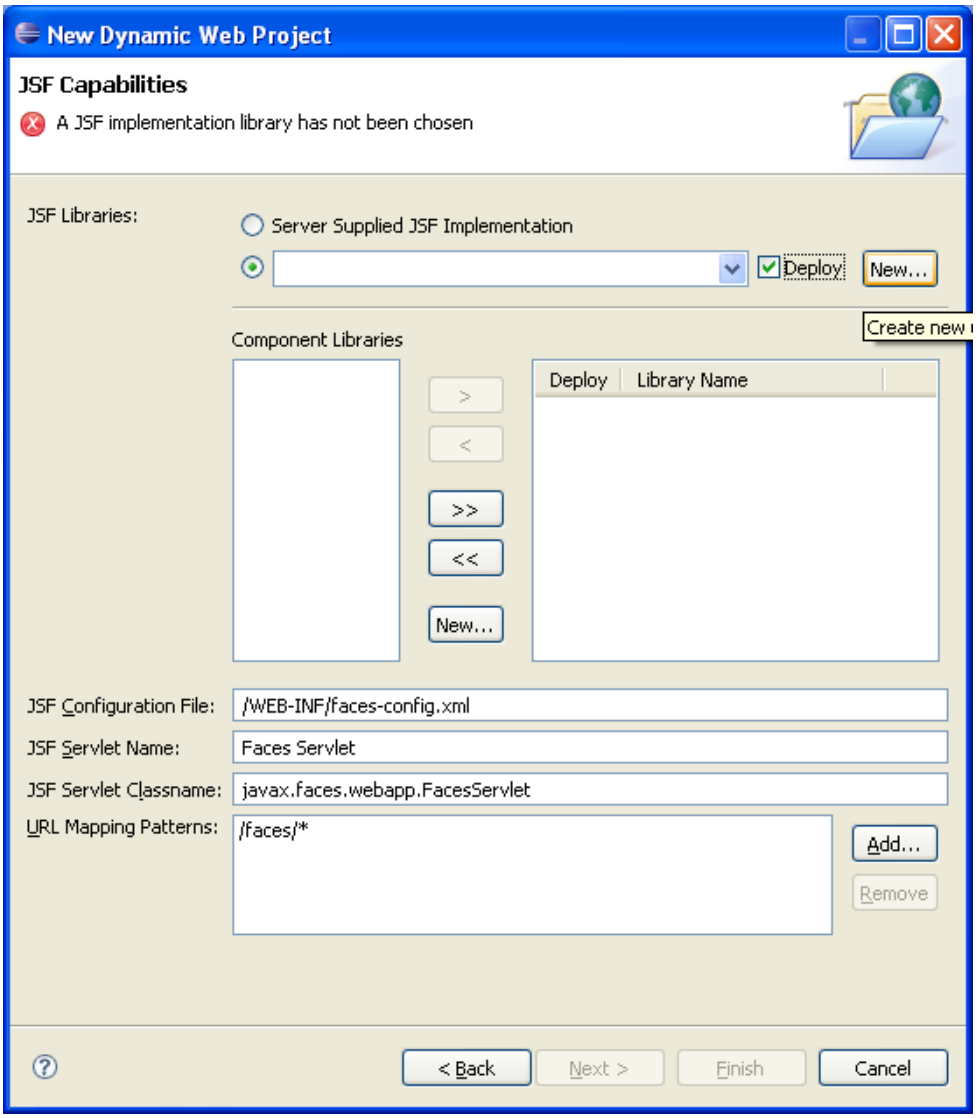
Digite o nome do projeto (por exemplo “PrimeiroProjeto”) e na seção “Configuration” selecione “JavaServer Faces v1.2 Project”. Pressione o botão “Next”.



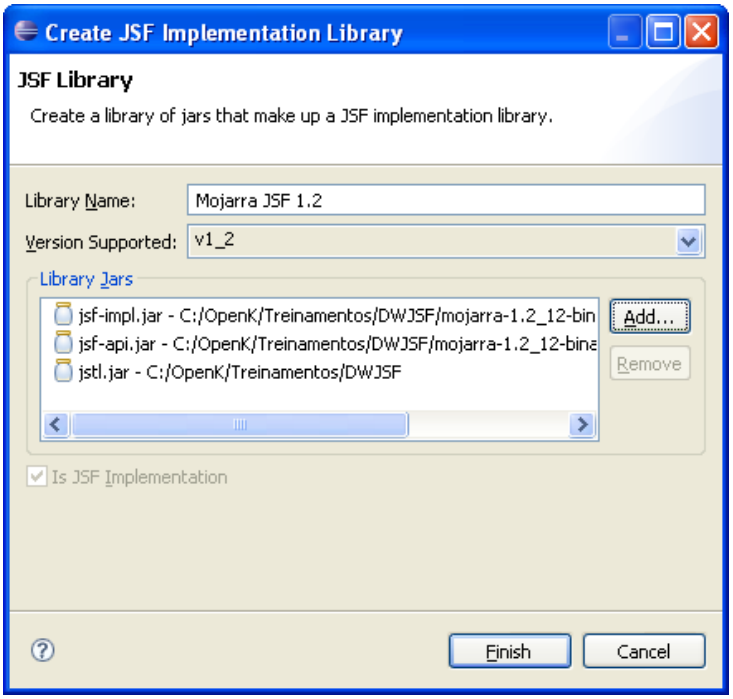
A nova tela que aparece apresenta algumas informações sobre o nome do contexto da aplicação e nomes dos diretórios de conteúdos web e fontes Java. Apenas pressione “Next”.



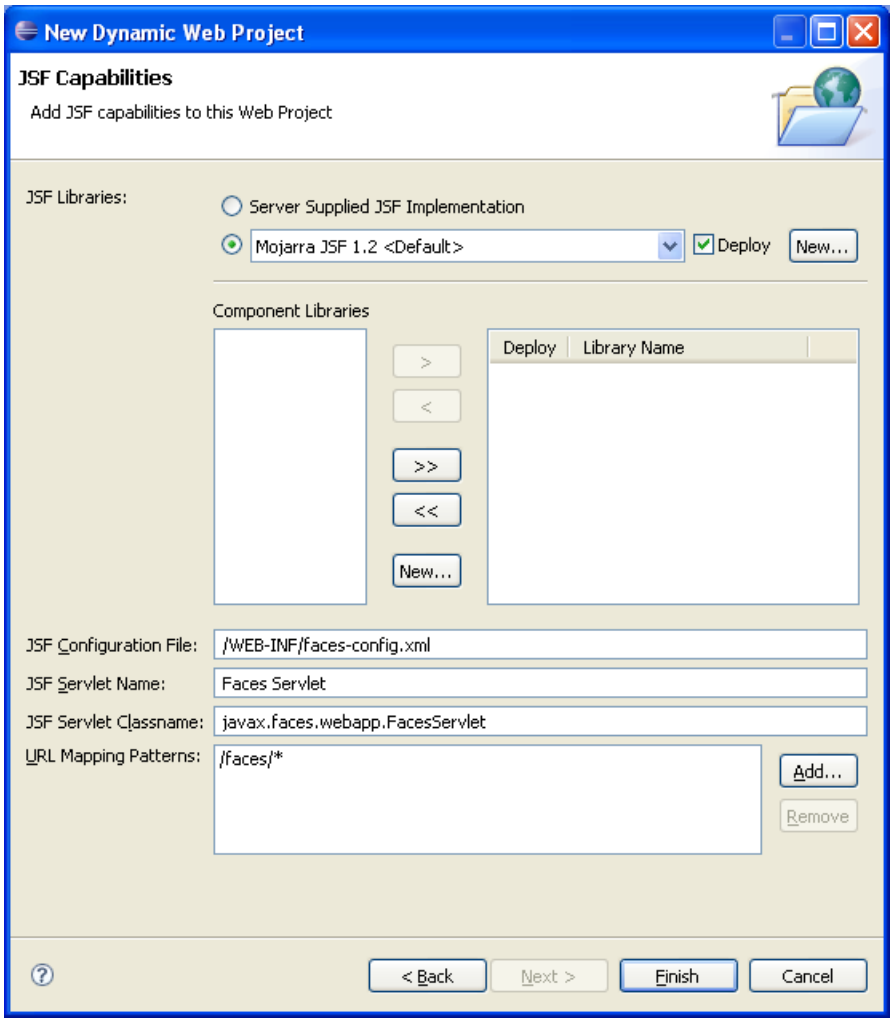
A próxima tela solicita as configurações da JSF para o seu projeto. Em “JSF Libraries”, selecione a segunda alternativa, selecione a opção “Deploy” e clique no botão “New...”. Iremos referenciar os pacotes (JARs) da implementação JSF.



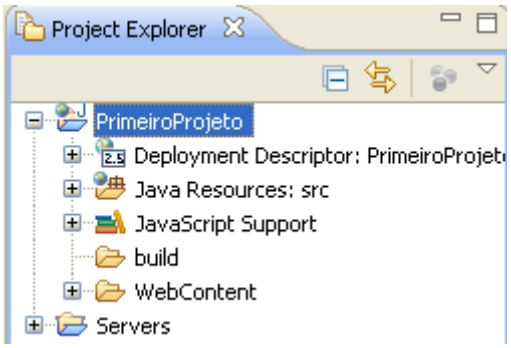
Informe um nome para a biblioteca, como por exemplo, “Mojarra JSF 1.2”, selecione “v1_2” em “Version Supported”, clique sobre o botão “Add...” e selecione os arquivos “jsf-api.jar” e “jsf-impl.jar”, que se encontram dentro do diretório “lib” distribuído no arquivo de instalação da Mojarra. Clique novamente em “Add...” e inclua o arquivo “jstl.jar”, que também está disponível no DVD. Clique no botão “Finish”.



Nosso projeto agora utilizará a biblioteca cadastrada. Clique sobre o botão “Finish”.



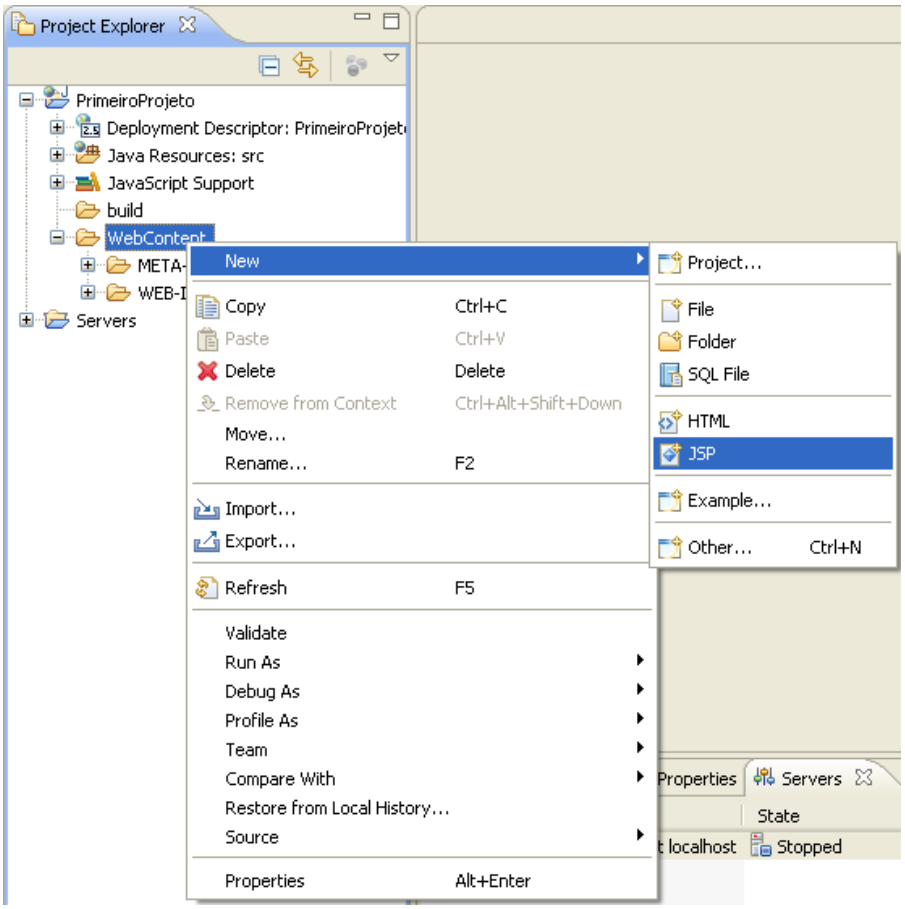
Quando o projeto for criado, você o verá em “Project Explorer”.



4.4. Codificando a primeira aplicação JSF

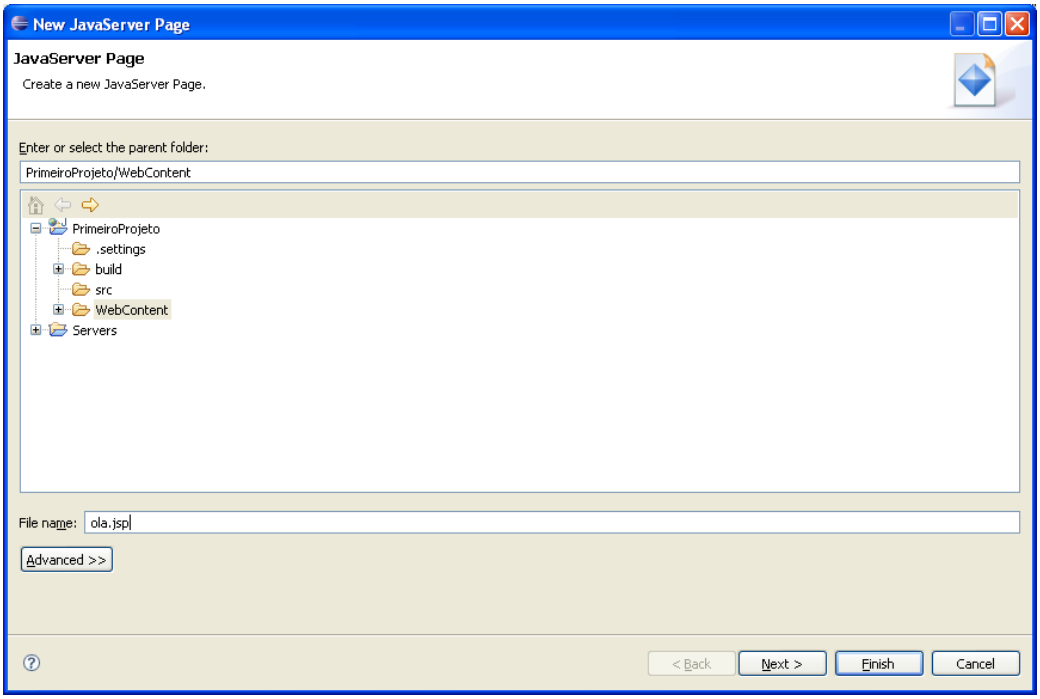
Nossa primeira aplicação será uma tela com um campo de entrada de texto e um botão. O usuário deve informar o nome no campo e pressionar o botão. Uma mensagem de boas vindas será dada ao usuário, transformando o nome digitado em letras maiúsculas.

Clique com o botão direito sobre a pasta “WebContent”, selecione “New” e “JSP”.

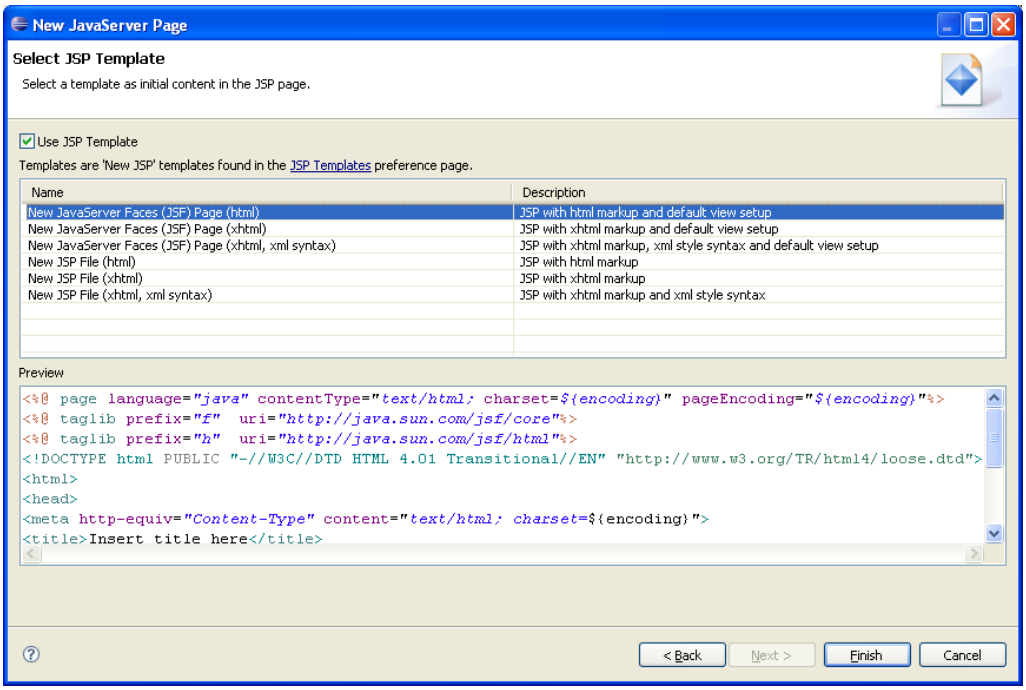


Em “File name”, digite o nome de nossa primeira página JSF, como por exemplo, “ola.jsp”, e clique no botão “Next”.

A extensão do arquivo deve ser “.jsp”, pois a tecnologia JSF foi construída baseada na JSP.



O Eclipse nos fornece alguns templates (modelos de códigos prontos) para facilitar nosso trabalho. Selecione um template JSF com tags HTML, como o “New JavaServer Faces (JSF) Page (html)”. Clique no botão “Finish”.



O Eclipse irá gerar um código JSF com um conteúdo que é comum em praticamente todas as telas que desenvolvemos. Isso facilita para não precisarmos digitar tudo.

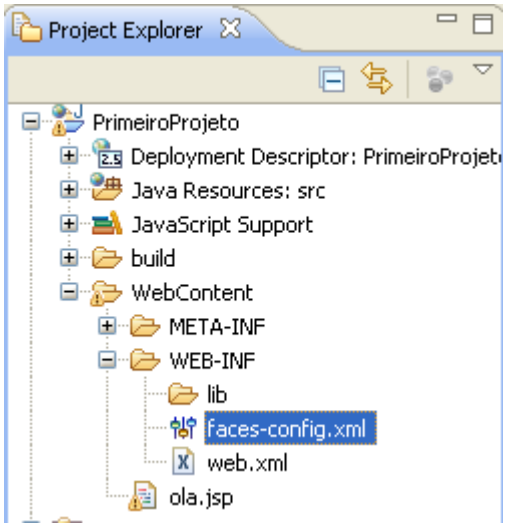
Altere o título da tela na tag HTML `<title>`.

```
*ola.jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Minha primeira aplicação</title>
</head>
<body>
<f:view>
</f:view>
</body>
</html>
```

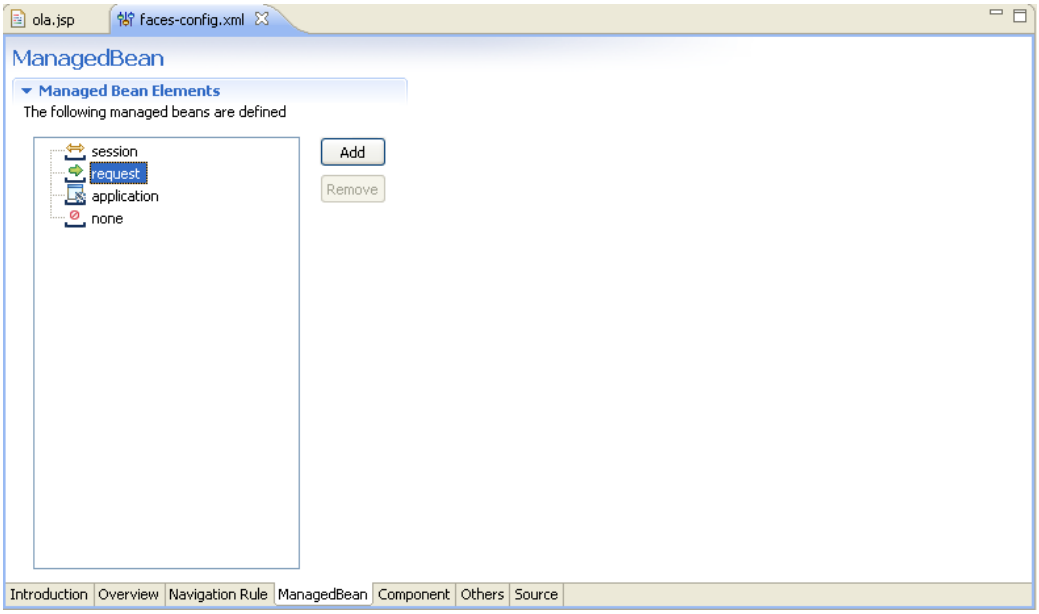
Dentro da tag `<f:view>`, digite o código a seguir:

```
<h:form>
        <h:outputLabel value="Seu nome: "/>
        <h:inputText value="#{usuarioBean.nome}"/>
        <h:commandButton value="Enviar"
            actionListener="#{usuarioBean.enviar}"/>
        <br/>

        <h:outputText value="Bem vindo a primeira aplicação JSF, #{usuarioBean.nome}"
                rendered="#{usuarioBean.nome != null}"/>
</h:form>
```

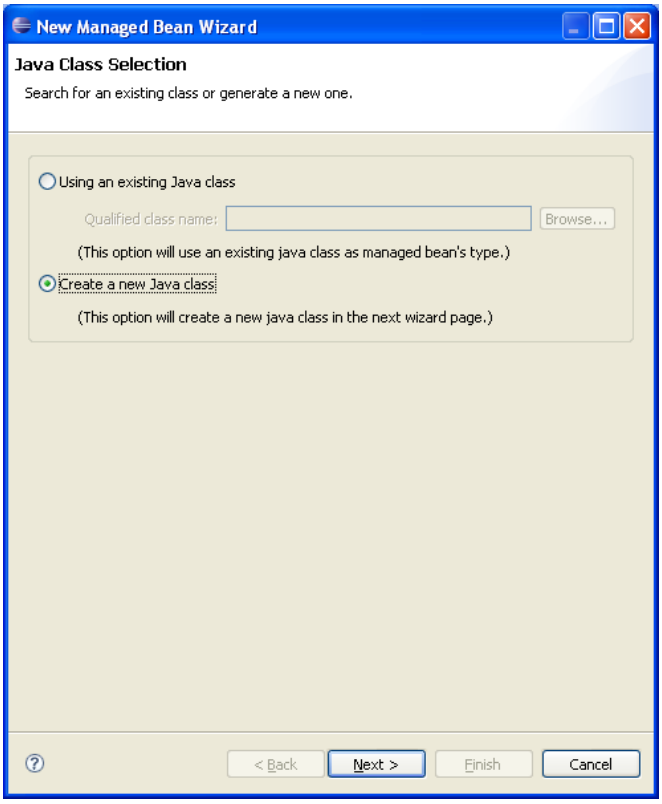
Encontre o arquivo “faces-config.xml” no diretório “WebContent/WEB-INF” e clique duas vezes sobre ele. Um editor visual deste arquivo será apresentado.



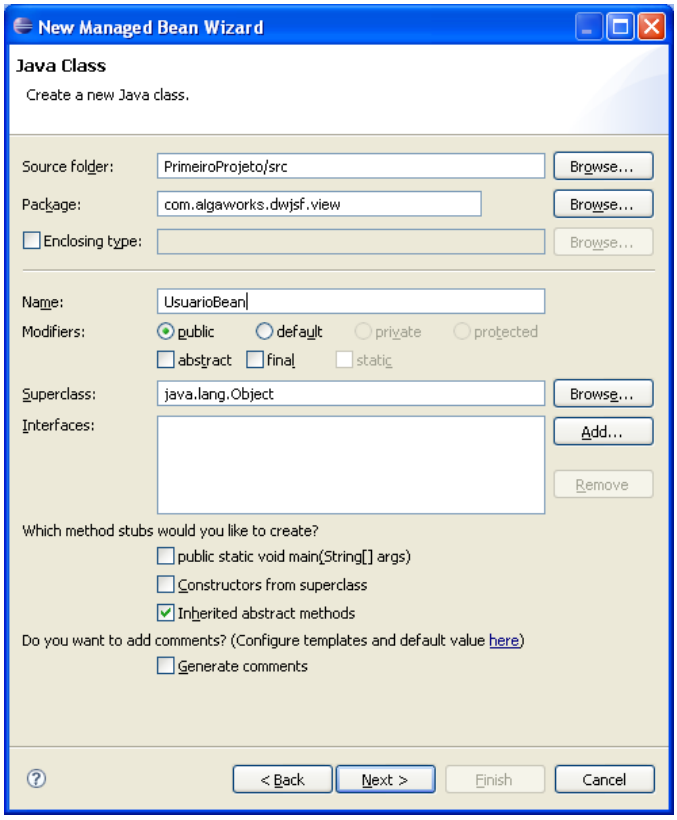
No editor do arquivo “faces-config.xml”, clique sobre a aba “ManagedBean”. Selecione a opção “request” na lista à esquerda e clique sobre o botão “Add”.



Nesta nova tela, selecione a opção “Create a new Java class” e clique em “Next >”.

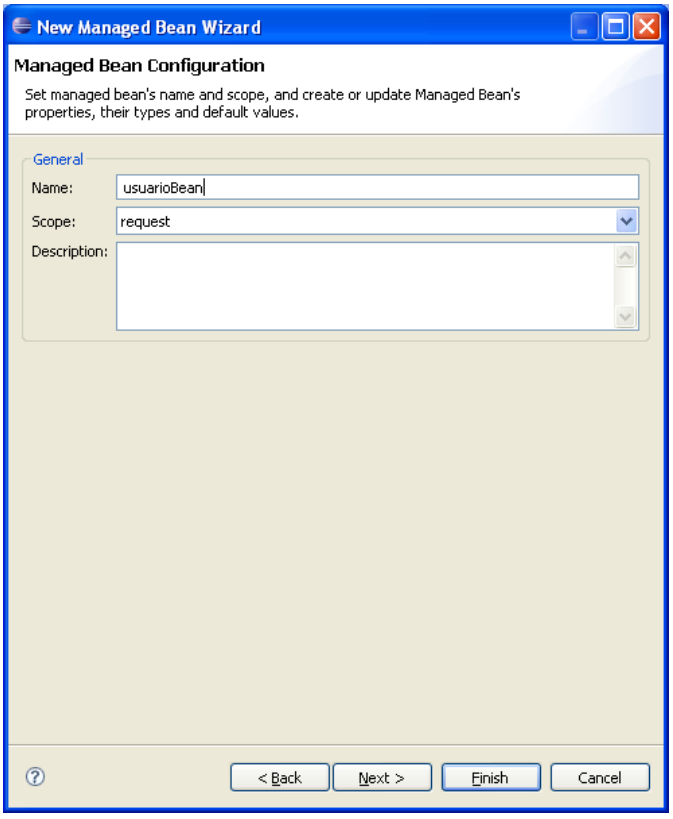


Você criará uma nova classe responsável por controlar os componentes da tela. Digite “com.algaworks.dwjsf.view” para o nome do pacote e “UsuarioBean” para o nome da classe. Clique no botão “Next >”.

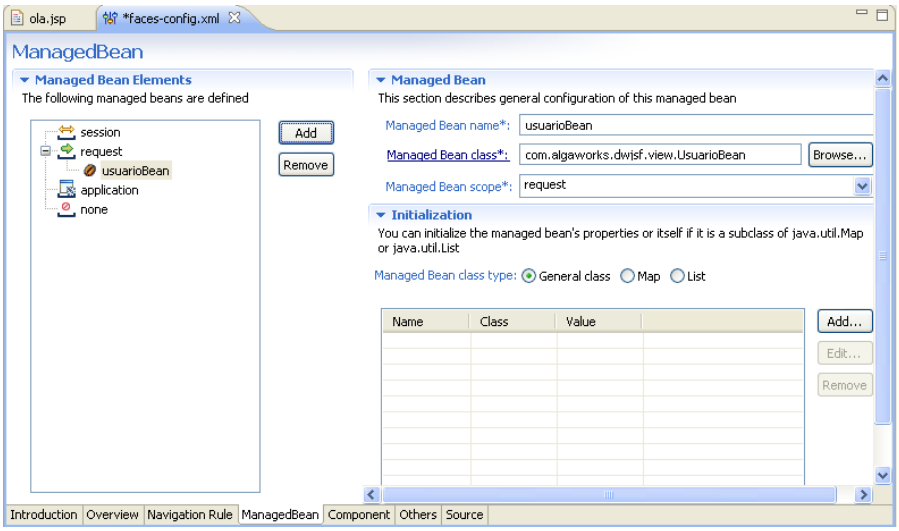


O Eclipse fará o mapeamento necessário para que a classe que estamos criando consiga realmente se "conectar" com a tela JSP. Não se preocupe, aprenderemos a fazer isso sem ajuda do Eclipse. Nosso objetivo agora é apenas deixar funcionando a primeira tela.

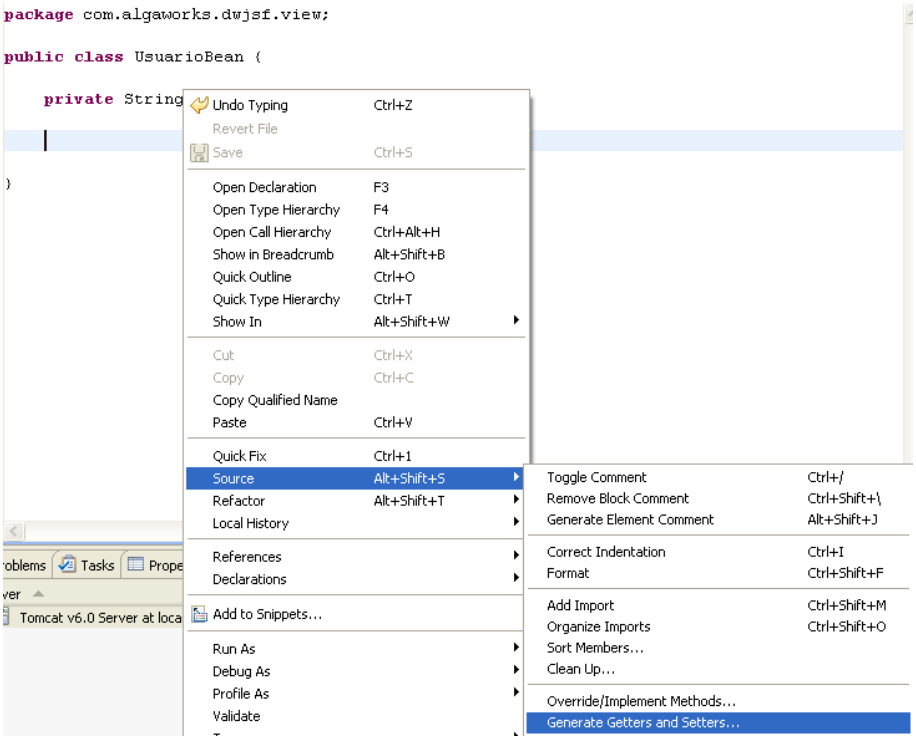
Clique sobre o botão "Finish".



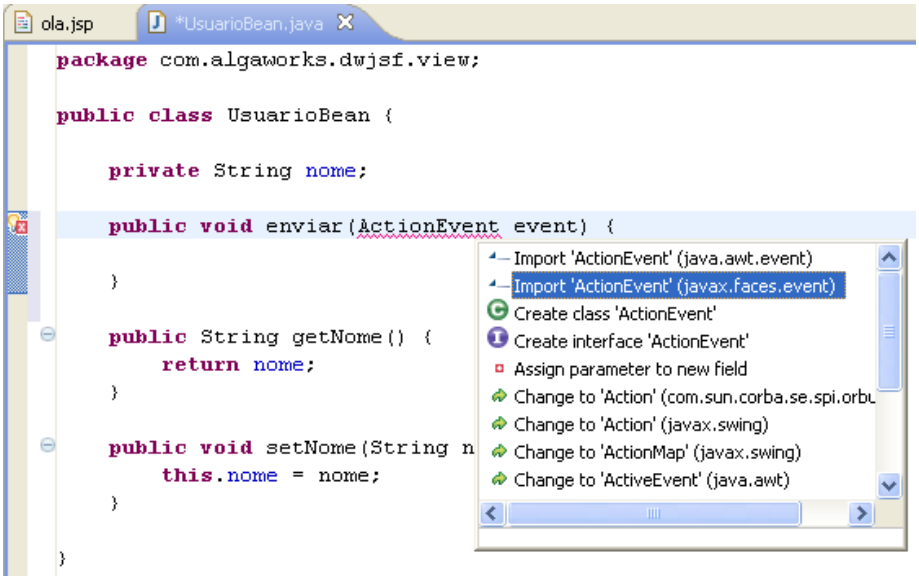
Salve o arquivo “faces-config.xml” pressionando as teclas *Ctrl* + *S*. Se tiver curiosidade, clique sobre a aba “Source” para ver o código que o Eclipse gerou a partir desta ferramenta de configuração.



A classe “UsuarioBean” foi criada pelo Eclipse, mas ainda não tem métodos e atributos. Crie um atributo privado do tipo “String” chamado “nome”. Depois gere os métodos *getters* e *setters* para este atributo. O Eclipse possui um gerador de métodos *getters* e *setters* através da opção “Source”, “Generate Getters and Setters”.



Crie um método público, sem retorno e com o nome “enviar”. Este método deve receber um parâmetro chamado “event” do tipo `ActionEvent`. A importação deste tipo deve ser feita através do pacote `javax.faces.event`.



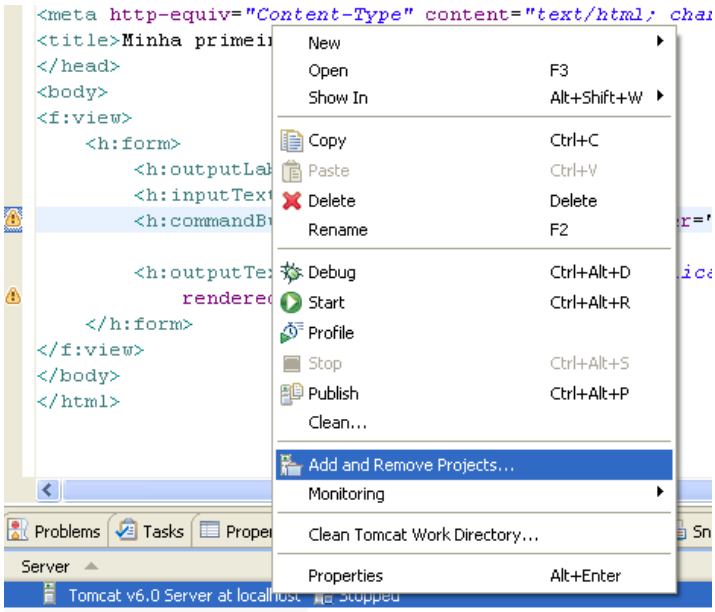
O método enviar será chamado quando o usuário clicar no botão “Enviar”. Precisamos passar o nome digitado para maiúsculo, por isso, você pode deixar este método da seguinte forma:

```
public void enviar(ActionEvent event) {
	this.setNome(this.getNome().toUpperCase());
}
```

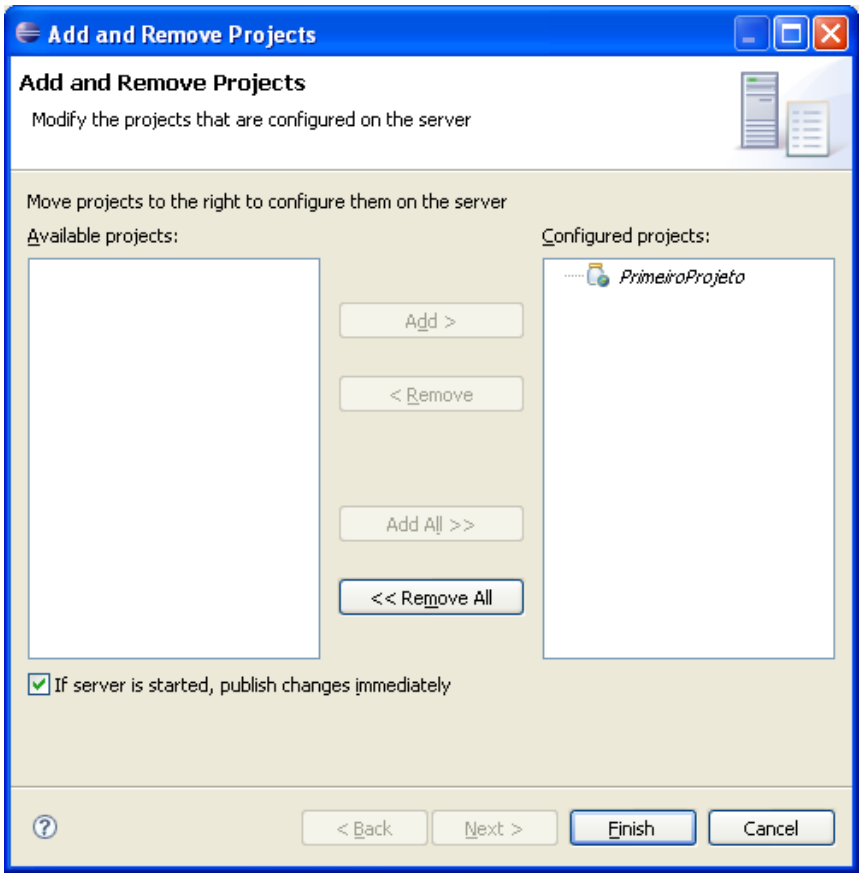
Nossa tela está pronta!

4.5. Implantação e execução da aplicação

Para testar a tela que criamos, precisamos adicionar o projeto ao Tomcat que já está integrado ao Eclipse. Na aba “Servers”, clique com o botão direito sobre o servidor e selecione “Add and Remove Projects...”.

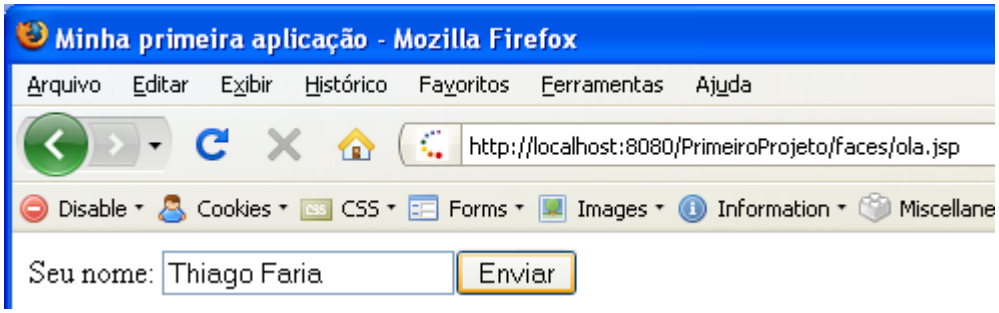


Selecione o projeto “PrimeiroProjeto” e transfira-o para a lista de projetos configurados. Clique no botão “Finish”.

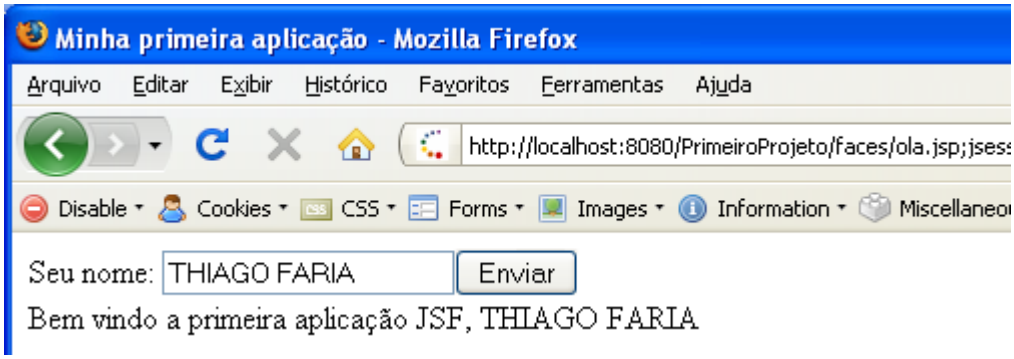


Inicie o Tomcat. O Eclipse implantará automaticamente a aplicação “PrimeiroProjeto”.

Abra um browser e acesse a primeira tela que desenvolvemos através do endereço: http://localhost:8080/PrimeiroProjeto/faces/ola.jsp.



Digite seu nome no campo e clique sobre o botão “Enviar”.



Veja que o nome digitado foi transformado para maiúsculo, e a mensagem de boas vindas foi apresentada.

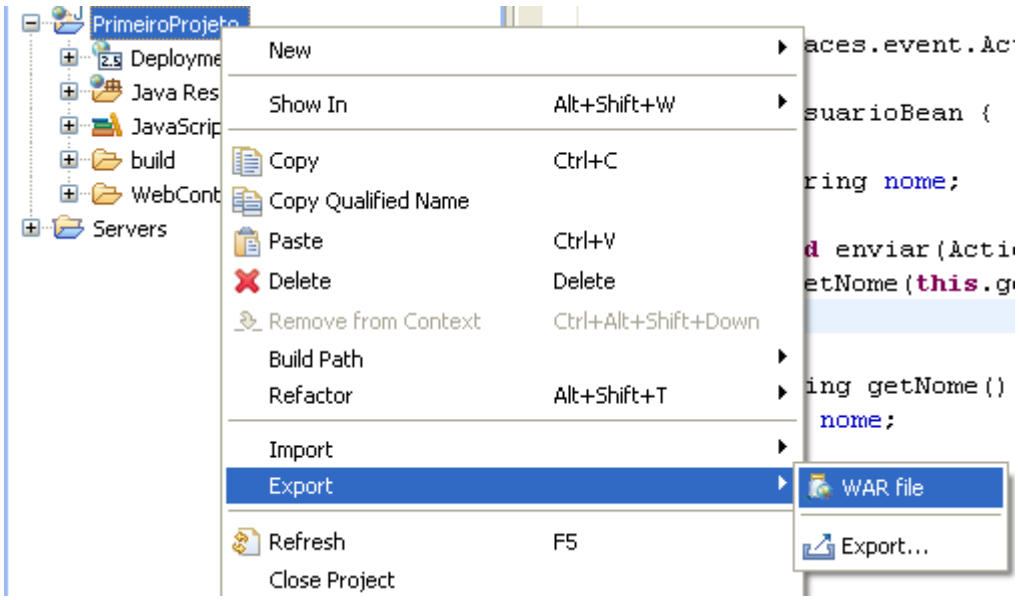
4.6. Gerando um WAR da aplicação

A integração que o Eclipse faz com o Tomcat facilita bastante o desenvolvimento de sistemas, mas você não terá o Eclipse no servidor de produção que executará as aplicações.

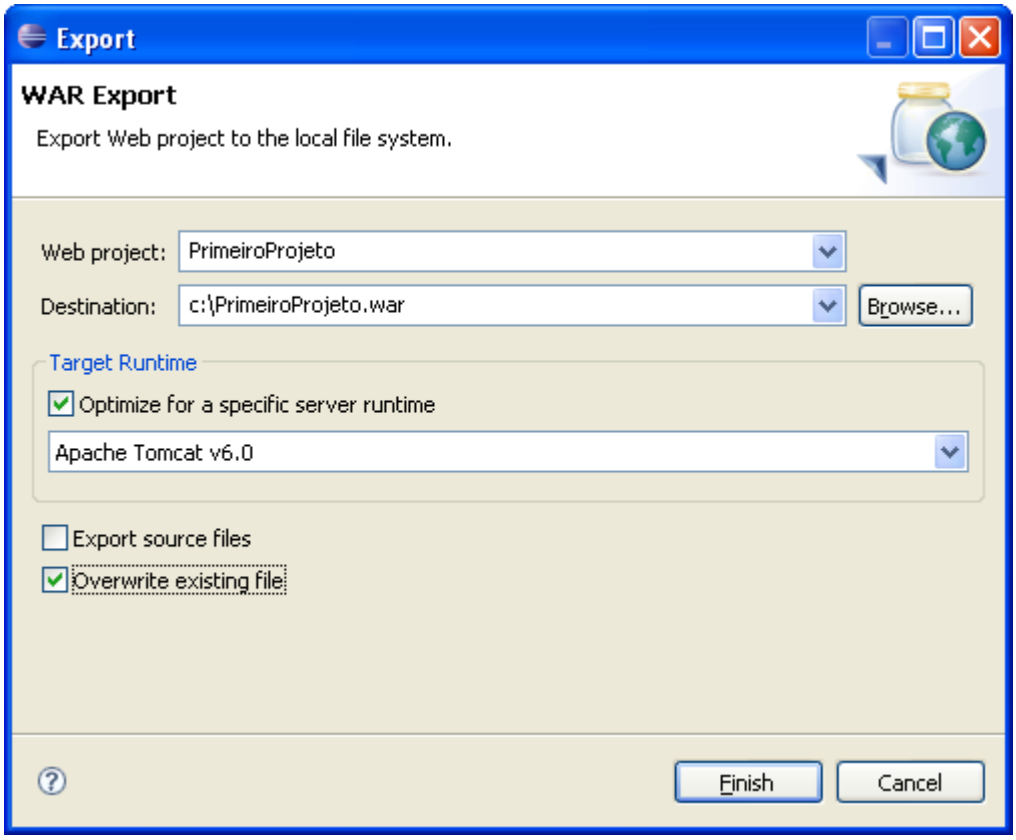
Para você distribuir suas aplicações de forma elegante e de fácil instalação, você deve gerar um pacote único que contém tudo necessário para o funcionamento, sem complicações. Esse pacote é chamado de WAR (Web Application Archive).

Um arquivo WAR nada mais é que um arquivo ".jar" nomeado com a extensão ".war".

O Eclipse possui uma ferramenta para gerar arquivos WAR. Clique sobre o botão direito do mouse no projeto e selecione a opção "Export". Clique sobre "WAR File".



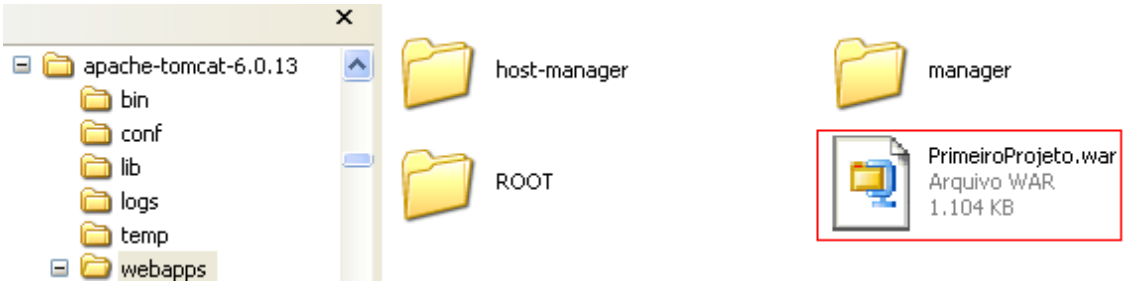
A tela "Export" deve ser exibida. Informe o caminho completo (incluindo o nome do arquivo) que deseja gerar o WAR e clique no botão "Finish".



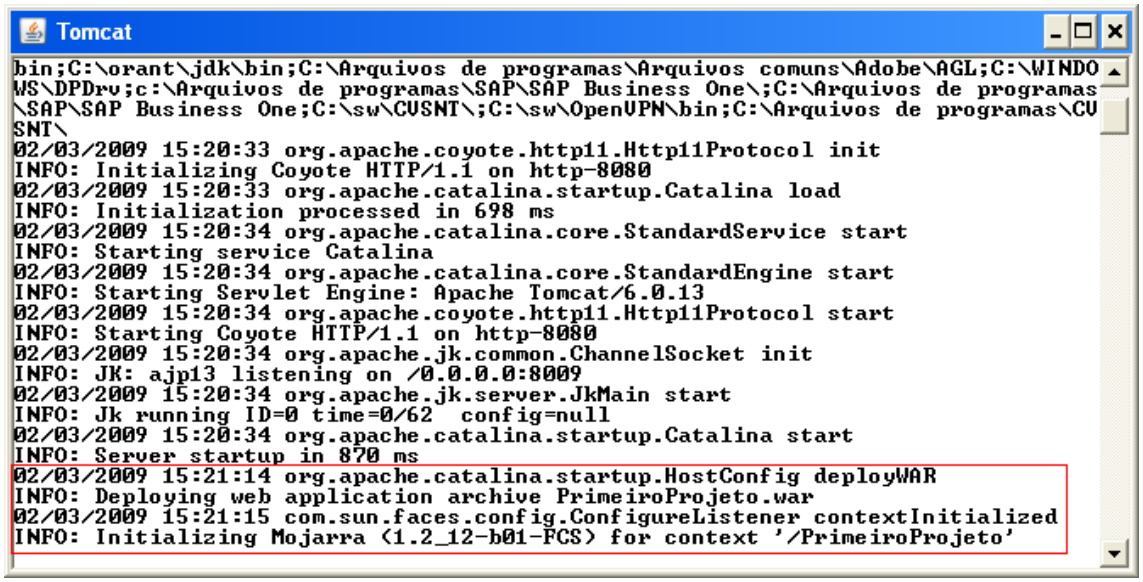
Com um arquivo ".war", podemos instalar nossa aplicação web em qualquer servidor de aplicações. No caso do Tomcat, basta copiar esse arquivo para a pasta *webapps* do servidor.

Para testarmos, inicie o Tomcat por fora do Eclipse. Não copie ainda o arquivo WAR gerado para a pasta *webapps*.

Depois que o Tomcat for iniciado, copie o arquivo "PrimeiroProjeto.war" para dentro da pasta *webapps*.



Veja que o Tomcat fará o *deploy* automático do pacote.



Agora você pode acessar o sistema através do browser para confirmar se deu tudo certo.

5. Desvendando o mistério

5.1. Introdução

Para entender o que acabamos de desenvolver, precisamos conhecer o que o Eclipse gerou de código para nós e também as programações que fizemos.

Estudaremos basicamente o *managed bean* `UsuarioBean`, os arquivos "faces-config.xml", "ola.jsp" e "web.xml", além dos demais conteúdos do diretório "WEB-INF".

5.2. Managed bean UsuarioBean

Antigamente (há não muito tempo atrás), alguns programadores desenvolviam todo o comportamento de uma tela no próprio arquivo de layout, na verdade ainda existem programadores que fazem isso. Em JSF, não fazemos isso! O arquivo JSP deve possuir tags HTML e JSF que definem o layout da página. O comportamento da tela deve ser implementado em código Java, em uma classe caracterizada como um *managed bean*.

Os *managed beans* nada mais é que Java Beans, que servem como canais entre a interface gráfica (a tela) e o *back-end* da aplicação (regras de negócio, acesso ao banco de dados, etc).

Um Java Bean é uma classe Java implementada de acordo com algumas convenções que determinam como devem ser escritas propriedades e os métodos públicos que as acessam. A especificação completa pode ser encontrada em http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html.

Os *managed beans* no JSF podem ser criados e configurados sem programação (através de arquivos XML de configuração, que veremos adiante). Em JSF, usa-se beans para conectar classes Java com páginas web ou arquivos de configuração.

Os beans são muito importantes para os desenvolvedores JSF, portanto, é fundamental entender a especificação JavaBean. Nesta seção veremos as partes relevantes para o desenvolvimento com JSF.

As características mais importantes de um bean são suas propriedades. Uma propriedade tem:

- Um nome;
- Um tipo;
- Métodos para obter (*getter*) e/ou definir (*setter*) o valor da propriedade.

Veja que os JavaBeans são classes Java, portanto, propriedades são os atributos desta classe.

Uma propriedade não deve ser acessada diretamente, mas através dos métodos de acesso (*getters* e *setters*). Algumas linguagens de programação permitem isso (inclusive o Java), porém, na especificação JavaBean isso não é correto.

Uma propriedade de um JavaBean pode ser:

- Leitura/escrita, somente-leitura ou somente-escrita;
- Simples, ou seja, contém apenas um valor, ou indexado, quando representa uma lista de valores.

Para obter o resultado de uma propriedade, o bean precisa ter um método da seguinte forma:

```
public ClasseDaPropriedade getPropriedade() { ... }
```

E para alterar uma propriedade:

```
public setPropriedade(ClasseDaPropriedade propriedade) { ... }
```

A especificação ainda exige que a classe bean tenha pelo menos um construtor padrão, ou seja, sem parâmetros.

O exemplo abaixo ilustra um bean `Pessoa` com as propriedades `nome`, `idade` e `profissoes`.

```
package com.algaworks.dwjsf.dominio.Pessoa;

import java.util.List;

public class Pessoa {

	private String nome;
	private Long idade;
	private List profissoes;

	public Pessoa() {
	}

	public Pessoa(String nome, Long idade, List profissoes) {
		this.nome = nome;
		this.idade = idade;
		this.profissoes = profissoes;
	}

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

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

	public Long getIdade() {
		return this.idade;
	}

	public void setIdade(Long idade) {
		this.idade = idade;
	}

	public List getProfissoes() {
		return this.profissoes;
	}

	public void setProfissoes(List profissoes) {
		this.profissoes = profissoes;
	}

}
```

Uma propriedade possui apenas o método *getter* para somente-leitura, somente o método *setter* para somente-escrita ou ambos os métodos para leitura e escrita.

Os nomes e as assinaturas dos métodos devem coincidir precisamente com o padrão, ou seja, começar com `get` ou `set`, como apresentado no exemplo acima. Um método `get` deve ter um valor de retorno e nenhum parâmetro. Um método `set` deve ter um parâmetro e nenhum valor de retorno.

A regra para a criação do nome do método é colocar `set` ou `get` na frente do nome da propriedade com sua primeira letra maiúscula, ou seja, uma propriedade `nome` teria os métodos com nomes `getNome` e `setNome`. Uma exceção a está regra ocorre quando o nome da propriedade está em maiúscula, por exemplo, uma propriedade `URL` teria os métodos com nomes `getURL` e `setURL`. Repare que a propriedade referenciada não é `uRL`.

Uma propriedade do tipo `boolean` tem uma variância no seu método *getter*. Por exemplo, uma propriedade de nome `ativo` do tipo boleano, poderia ser tanto:

```
boolean isAtivo()
```

Quanto:

```
boolean getAtivo()
```

Nosso bean `UsuarioBean` possui apenas a propriedade `nome`.

```
public class UsuarioBean {

	private String nome;

	...

	public String getNome() {
		return nome;
	}

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

}
```

Uma vantagem de se utilizar os métodos de acesso às propriedades é a possibilidade de manipular seus valores através estes métodos, ou seja, podem-se converter dados, acessar um banco de dados, fazer validações e várias outras coisas.

Voltando à tela que desenvolvemos, o bean `UsuarioBean` possui apenas o atributo `nome` porque é o único campo que incluímos na tela.

O método `enviar(...)` recebe como parâmetro um objeto do tipo `ActionEvent`, que aprenderemos em outro capítulo. Neste momento você só precisa saber que este método será invocado quando o botão “Enviar” for pressionado pelo usuário. Esse método é então um manipulador de eventos, que no nosso exemplo, altera o nome digitado para maiúsculo.

```
public void enviar(ActionEvent event) {
	this.setNome(this.getNome().toUpperCase());
}
```

5.3. Arquivo faces-config.xml

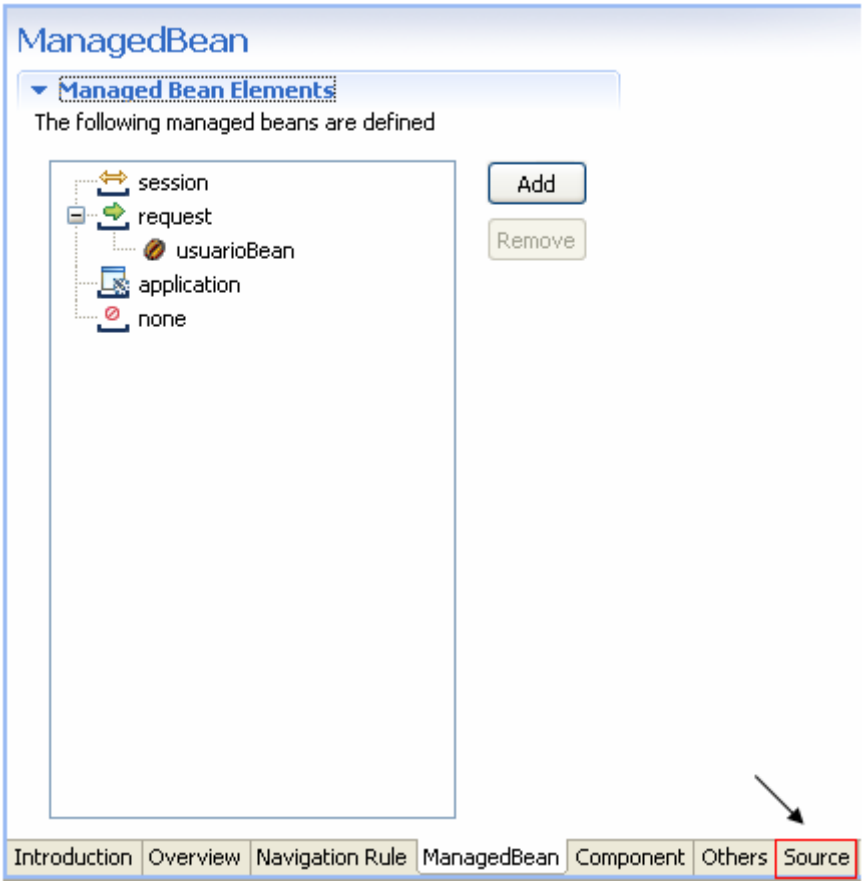
Para instanciar um *managed bean* e determinar seu escopo, utiliza-se um arquivo de configuração XML.

Este arquivo é processado quando a aplicação é iniciada. Quando uma página referencia um bean, a implementação do JSF o inicializa de acordo com as configurações contidas neste arquivo.

O arquivo de configuração mais utilizado é o “WEB-INF/faces-config.xml”, porém é possível mudar este nome e até utilizar vários arquivos em conjunto.

Para configurar um bean é necessário no mínimo o nome, a classe e o escopo do bean (que será visto mais adiante).

No último exemplo, configuramos o bean através de uma ferramenta visual do Eclipse, mas poderíamos ter feito manualmente. Para ver o código gerado pelo Eclipse, acesse a aba "Source" da ferramenta de edição do "faces-config.xml".



O XML de configuração dos beans deve-se parecer com este:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config ...>
   <managed-bean>
      <managed-bean-name>usuarioBean</managed-bean-name>
      <managed-bean-class>
         com.algaworks.dwjsf.view.UsuarioBean
      </managed-bean-class>
      <managed-bean-scope>request</managed-bean-scope>
   </managed-bean>
</faces-config>
```

O XML acima declara que teremos um *managed bean* chamado "usuarioBean", que é referente a classe `com.algaworks.dwjsf.view.UsuarioBean`, ou seja, sempre que usarmos o nome "usuarioBean" em uma página JSF, estamos falando de uma instância da classe `UsuarioBean`.

A propriedade *managed-bean-scope* define o escopo do bean. Neste caso, o escopo é *request*, o que significa que o estado do bean será mantido a cada requisição do cliente. Veremos outros escopos em outro capítulo.

5.4. Página ola.jsp

Toda tela de usuário deve possuir um arquivo JSP com códigos HTML e tags JSF. Normalmente, os arquivos das páginas possuem extensão ".jsp". No exemplo que desenvolvemos, o nome desse arquivo foi "ola.jsp". Veja o código da tela do último exemplo:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Minha primeira aplicação</title>
</head>
<body>
<f:view>
  <h:form>
    <h:outputLabel value="Seu nome: "/>
    <h:inputText value="#{usuarioBean.nome}"/>
    <h:commandButton value="Enviar"
      actionListener="#{usuarioBean.enviar}"/>
    <br/>

    <h:outputText
       value="Bem vindo a primeira aplicação JSF, #{usuarioBean.nome}"
       rendered="#{usuarioBean.nome != null}"/>
  </h:form>
</f:view>
</body>
</html>
```

Observe que grande parte deste arquivo possui tags HTML. Isso prova que você pode misturar códigos HTML com tags JSF na maioria dos casos.

Agora veremos o que cada trecho de código significa para a construção da tela.

Começamos pelas declarações das bibliotecas de tags (taglibs) do JSF:

```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
```

As declarações acima dizem para o framework do JSF que queremos usar as bibliotecas de tags *html* e *core*. A biblioteca *html* contém todas as tags (componentes) para trabalhar com itens específicos da HTML, como formulários, campos de entrada de dados, botões e etc. A biblioteca *core* contém lógicas para validações, conversões e outras coisas específicas da JSF.

Todas as páginas JSF devem possuir uma tag `<f:view>`. Essa tag diz ao JSF que você quer gerenciar os componentes que estiverem em seu corpo, em outras palavras, todos os componentes filhos de `<f:view>` podem ser gerenciados.

```
<f:view>
   ...
</f:view>
```

Se você não colocar a tag `<f:view>`, o JSF não poderá construir a árvore de componentes gerenciáveis da página, e isso impossibilitaria uma futura localização dos componentes criados na tela, ou seja, você deve sempre incluir a tag `<f:view>` englobando todos os seus componentes JSF para que tudo funcione da maneira desejada.

A tag `<h:form>` diz que você quer um formulário da HTML neste local.

```
<f:view>
  <h:form>
      ...
  </h:form>
</f:view>
```

Os componentes de entrada de dados e também os que permitem comandos (ações) dos usuários (como os botões e links) sempre devem estar dentro da tag `<h:form>`, pois assim os valores digitados pelos usuários nos campos e os estímulos aos comandos podem ser enviados para o servidor fazer o processamento.

Usamos o componente `<h:inputText>` para apresentar um campo de entrada de texto e um componente `<h:commandButton>` para um botão.

```
<h:inputText value="#{usuarioBean.nome}"/>
<h:commandButton value="Enviar"
       actionListener="#{usuarioBean.enviar}"/>
```

Os delimitadores `#{...}` serão explicados no próximo capítulo com mais detalhes. Esses delimitadores podem ser chamados de expressões de ligação ou JSF EL (JavaServer Faces Expression Language). O texto "usuarioBean" está referenciando o *managed bean* que configuramos anteriormente, e "nome" faz referência à propriedade com mesmo nome no bean. Esta propriedade é acessada usando o método `getNome` e alterada usando o método `setNome`, por isso é tão importante o padrão Java Beans.

Quando a página é exibida, o método `getNome` é chamado para obter o valor atual da propriedade. Quando a página é submetida através do botão "Enviar", o método `setNome` é chamado para alterar o valor que foi digitado pelo usuário através do campo de entrada.

A tag `<h:commandButton>` possui o atributo `actionListener`, que liga a ação sobre este botão ao método `enviar` da classe `UsuarioBean`, ou seja, quando o usuário clicar sobre o botão "Enviar", o servidor identificará este comando e executará automaticamente o método `enviar` do bean. Este método por sua vez tem a chance de fazer qualquer processamento do lado do servidor. No caso de nosso exemplo, apenas mudamos o nome digitado no campo de entrada para letras maiúsculas.

A tag `<h:outputLabel>` apenas apresenta um texto (rótulo) digitado no atributo `value`. Nada de extraordinário.

```
<h:outputLabel value="Seu nome: "/>
```

Você deve ter percebido que a mensagem de boas vindas só foi apresentada depois que o nome foi informado no campo de entrada e o botão "Enviar" foi pressionado. Isso ocorreu porque incluímos um componente chamado `<h:outputText>` que só é apresentado (renderizado) quando o nome não for nulo.

A tag `<h:outputText>` apresenta textos contidos no atributo `value`. Esse atributo pode conter textos estáticos (fixos), como a mensagem de boas vindas, e também textos dinâmicos, como o nome digitado pelo usúario.

```
<h:outputText
    value="Bem vindo a primeira aplicação JSF, #{usuarioBean.nome}"
    rendered="#{usuarioBean.nome != null}"/>
```

O atributo `rendered` do componente `<h:outputText>` é utilizado incluindo uma expressão boleana (que retorna *true* ou *false*). O componente só é renderizado se essa expressão for verdadeira. Nossa expressão só será verdadeira se o nome for diferente de nulo, ou seja, apenas se o usuário digitar o seu nome no campo de entrada.

5.5. Arquivo web.xml

As aplicações JSF precisam de um arquivo chamado "web.xml" com algumas configurações para funcionarem. Normalmente este arquivo é o mesmo na maioria das aplicações. No exemplo que desenvolvemos, o Eclipse criou e configurou este arquivo de acordo com as informações que passamos para ele.

Para entender um pouco mais, abra o código-fonte do arquivo "web.xml" (dentro da pasta "WEB-INF"). Ele deve se parecer com o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
  <display-name>PrimeiroProjeto</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <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>/faces/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Todas as páginas JSF são processadas por um servlet do próprio framework. Este servlet é implantado no "web.xml" através da tag `<servlet>`.

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

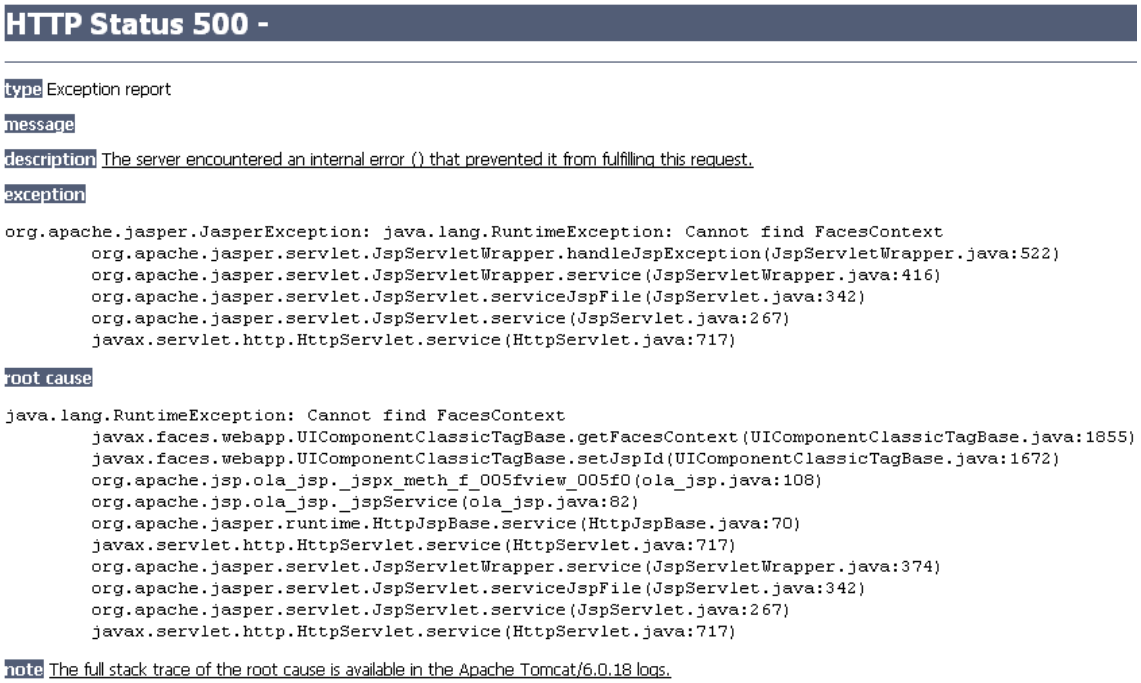
Este servlet é chamado toda vez que o caminho da URL iniciar por `/faces/*`. Esta regra foi definida através da tag `<servlet-mapping>` e, portanto, pode ser alterada para outro padrão que você gostar mais.

```
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Você não pode simplesmente chamar o endereço sem incluir "/faces", como:

http://localhost:8080/PrimeiroProjeto/ola.jsp

Veja o resultado se isso acontecer:



Quando você chama o arquivo JSP diretamente, o *Faces Servlet* não é executado, e por isso o JSF não inicializa seu contexto e o *view root* (os componentes dentro da tag `<f:view>`) antes de exibir a página.

Experimente mudar o mapeamento do Faces Servlet para o padrão `*.jsf`.

```
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.jsf</url-pattern>
</servlet-mapping>
```

Agora podemos chamar nossa página usando a URL:

http://localhost:8080/PrimeiroProjeto/ola.jsf

5.6. Conhecendo mais sobre o diretório WEB-INF

O diretório “WEB-INF” contém arquivos de configuração e outros necessários para o funcionamento da aplicação web. Todas as classes (arquivos *.class*) e bibliotecas (arquivos *.jar*) devem ficar dentro desse diretório. Este diretório é vital para sua aplicação, e ela não funcionará sem isso.

Além dos arquivos “web.xml” e “faces-config.xml”, o diretório “WEB-INF” pode possuir os sub-diretórios “classes” e “lib”. O diretório “classes” contém as classes criadas, tais como os *managed beans*, Java beans, regras de negócio, acesso a banco de dados e etc, e o diretório “lib” possui as bibliotecas de terceiros necessárias para o funcionamento de seu software, como para geração de gráficos e relatórios, driver JDBC, bibliotecas para leitura e escrita de arquivos PDF ou XLS, etc.

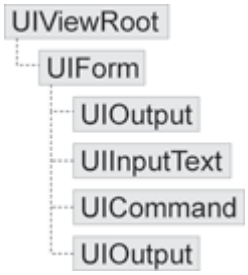
5.7. Como funciona nos bastidores

Você já conhece alguns dos principais conceitos do JSF e sabe como preparar o ambiente para desenvolvimento, inclusive já criou uma tela simples neste framework. Agora vamos estudar o que acontece nos bastidores da execução da nossa aplicação de exemplo.

Quando você digita o endereço da aplicação no *browser* e acessa a tela do sistema pela primeira vez, o servlet da JSF (Faces Servlet) intercepta a requisição, lê o arquivo JSP e efetua o processamento. No nosso caso, o arquivo JSP é o "ola.jsp". Esse arquivo contém tags de componentes, como o formulário, o campo de entrada de texto, o botão e etc.

Cada tag possui uma classe que a representa, chamada de *tag handler*, executada assim que a página é processada. Essas classes trabalham em conjunto e constroem uma hierarquia de componentes na memória do servidor, seguindo o que foi programado no arquivo JSP.

A hierarquia de componentes representa os elementos da interface visual presentes na página. Esses componentes são responsáveis por gerenciar seus estados e comportamentos. Veja como ficou a hierarquia dos componentes do último exemplo:



O componente `UIViewRoot` representa a raiz dos componentes gerenciados pelo JSF, especificado através da tag `<f:view>`, o `UIForm` representa o formulário (`<f:form>`), `UIOutput` significa o *label* de escrita de textos definido como `<h:outputLabel>`, os componentes `UIInputText` são os campos de entrada de texto (`<h:inputText>`) e `UICommand` o botão, especificado pela tag `<h:commandButton>`.

Ainda durante o processamento do arquivo JSP, uma página HTML é renderizada e enviada para o seu *browser*. Os componentes JSF são convertidos para código HTML.

Cada componente JSF possui um renderizador, que é responsável por gerar código HTML refletindo o estado de seu componente. Por exemplo, o componente `<h:inputText>` poderia ser representado no HTML pelo seguinte código, de acordo com seu renderizador, após o usuário digitar o seu nome e submeter ao servidor:

```
<input type="text" name="j_id_jsp_436522555_1:j_id_jsp_436522555_3"
      value="THIAGO FARIA"/>
```

O nome do componente HTML (atributo "name") foi definido com vários caracteres e números estranhos. Esse nome é gerado automaticamente pelo framework JSF, e é chamado de identificador único, ou *unique ID*. Você pode especificar o *unique ID* que deseja utilizar, mas se não fizer isso, o JSF irá gerar um para você, como no último exemplo.

Veja abaixo como ficaria o código HTML gerado pelos renderizadores dos componentes da página "ola.jsp":

Entre a requisição do usuário e a resposta dada pelo servidor, existem várias outras coisas executadas que não abordamos neste tópico. Você aprenderá tudo sobre a sequência de processamento no capítulo sobre “Ciclo de Vida”.

5.8. JSF é apenas isso?

JavaServer Faces é muito mais do que apenas isso! Este framework oferece diversos outros serviços para o desenvolvimento da camada visual de sua aplicação, como:

- **Conversão de dados:** facilita a conversão de textos digitados pelos usuários para tipos específicos, como por exemplo, quando o usuário digita sua data de nascimento, o servidor recebe um texto, que deve ser convertido para um objeto do tipo `Date` representando o que o usuário informou.

- **Validação:** diversas validações de dados básicas podem ser feitas antes de suas regras de negócio ser executadas, como, por exemplo, sobre obrigatoriedade de campos, aceitação de apenas números ou validação de documentos, como CPF e CNPJ. O JSF nos ajuda nesta tarefa, e aprenderemos como utilizar este recurso mais a frente.

- **Componentes customizados:** você pode criar ou baixar da internet componentes customizados para enriquecer o desenvolvimento de suas aplicações. Os componentes podem ser reutilizados em todas as aplicações de sua empresa, e o designer das páginas não precisa conhecer como ele foi feito para utilizá-lo. Por exemplo, você poderia criar um componente para exibir um teclado virtual para digitação de senhas. Veja um exemplo de como o designer de páginas poderia utilizá-lo:

  ```
  <aw:teclado value="#{loginBean.senha}" tipo="alfanumerico"/>
  ```

- **Renderizadores:** o JSF gera código HTML por padrão, porém o framework suporta a inclusão de outros renderizadores plugáveis que podem produzir códigos em outras linguagens para visualizadores de diferentes tipos, como celulares (WML), Telnet, Flash, etc.

- **Internacionalização:** com JSF, é muito fácil você desenvolver aplicações multi-idiomas com o recurso de *i18n* (abreviação para *internacionalization*).

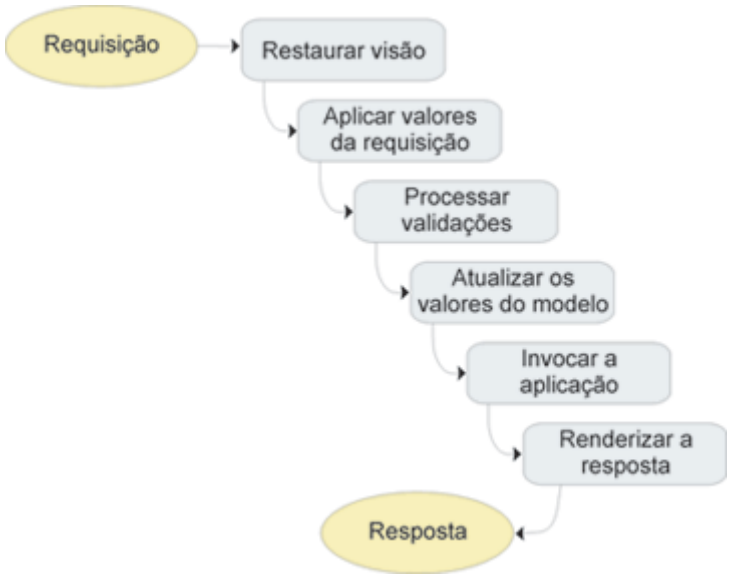
6. Ciclo de vida

Você pode desenvolver uma aplicação completa em JSF sem conhecer todos os detalhes deste framework, porém quanto mais você souber sobre ele, melhor e mais produtivo você se tornará. Por isso, estudaremos agora um assunto que nem todos os desenvolvedores JSF conhecem: o ciclo de vida.

Ao executar uma página construída usando componentes JSF, ela passará por um ciclo de vida de processamento bem definido, constituído por 6 fases, como seguem:

1. Restaurar visão
2. Aplicar valores de requisição
3. Processar validações
4. Atualizar os valores do modelo
5. Invocar a aplicação
6. Renderizar a resposta

Veja abaixo uma ilustração deste ciclo:



Agora vamos estudar o que cada fase do ciclo de vida processa.

1. Restaurar visão

A fase de restauração da visão recupera a hierarquia de componentes para a página solicitada, se ela foi exibida anteriormente, ou constrói uma nova hierarquia de componentes, se for a primeira exibição.

Se a página já tiver sido exibida, todos os componentes são recuperados em seu estado anterior. Isso dá condições dos dados de um formulário submetido ao servidor serem recuperados, caso ocorra algum problema de validação ou restrição de regra de negócio. Por exemplo, se um formulário solicita campos obrigatórios que não são totalmente preenchidos, porém enviados pelo usuário, o mesmo formulário deve aparecer novamente com os campos que não estavam vazios já preenchidos, porém com mensagens de erro indicando os campos requeridos.

2. Aplicar valores de requisição

Nesta fase, cada componente tem a chance de atualizar seu próprio estado com informações que vieram da requisição

3. Processar validações

Nesta fase, os valores submetidos são convertidos em tipos específicos e anexados aos componentes. Quando você programa uma página em JSF, você pode incluir validadores para criticarem os valores recebidos pelos usuários. Neste momento, os validadores entram em ação e, se ocorrerem erros de conversão ou de validação, a fase de renderização de resposta é invocada imediatamente, pulando todas as outras fases, e exibindo a página atual novamente, para que o usuário possa corrigir os erros e submeter os dados mais uma vez.

4. Atualizar os valores do modelo

Durante esta fase, os valores anexados (chamados de valores locais) aos componentes são atualizados nos objetos do modelo de dados e os valores locais são limpos.

5. Invocar a aplicação

Na quinta fase, os eventos que originaram o envio do formulário ao servidor são executados. Por exemplo, ao clicar em um botão para submeter um cadastro, a programação do evento deste botão deve ser executada. Em alguns casos, a execução do evento pode retornar um identificador dizendo qual é a próxima página a ser exibida, ou simplesmente não retornar nada para re-exibir a mesma página.

6. Renderizar a resposta

Por último, a fase de renderização da resposta gera a saída com todos os componentes nos seus estados atuais e envia para o cliente. O ciclo recomeça sempre que o usuário interage com a aplicação e uma requisição é enviada ao servidor.

7. Managed beans

7.1. Introdução

O entendimento de beans no desenvolvimento de aplicações Java de modo geral e principalmente em conjunto com o JSF é essencial para um bom programador. No contexto das páginas JSF, os beans fazem a separação entre a apresentação e a lógica de negócios, ou seja, todo o código de implementação fica no bean e as páginas apenas os referenciam.

Você já aprendeu o que é um Java Bean e também um managed bean em um capítulo anterior. Agora, aprenderá como configurar os *managed beans* através do arquivo XML.

7.2. Configurando beans

Os beans usados para vincular às páginas JSF são chamados de *managed beans*. Geralmente, utiliza-se um arquivo XML para configuração desses beans e seus escopos.

Este arquivo é processado quando a aplicação é iniciada. Quando uma página referencia um bean, a implementação do JSF o inicializa de acordo com as configurações obtidas.

O arquivo de configuração mais utilizado é o "WEB-INF/faces-config.xml", porém para uma maior organização do código, é comum fazer uma separação em vários arquivos, agrupando-os por funcionalidades como navegação, conversões e beans. Para isto, basta acrescentar ao arquivo "WEB-INF/web.xml" um parâmetro de inicialização "javax.faces.CONFIG_FILES" como mostrado abaixo:

```
<web-app>
      <context-param>
            <param-name>javax.faces.CONFIG_FILES</param-name>
            <param-value>
                WEB-INF/navigation.xml, WEB-INF/beans.xml
            </param-value>
      </context-param>
      ...
</web-app>
```

O assunto de navegação e conversões será abordado mais tarde.

Para configurar um bean é necessário no mínimo o nome, a classe e o escopo do bean (que será estudado mais adiante). O código abaixo ilustra um exemplo de configuração:

```
<faces-config>
  <managed-bean>
    <managed-bean-name>meuBean</managed-bean-name>
    <managed-bean-class>com.algaworks.MeuBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>
```

As instruções de configuração acima estão dizendo para o framework JSF construir um objeto da classe `com.algaworks.MeuBean`, dar um nome a ele de `meuBean` e manter este objeto no escopo de sessão do usuário, ou seja, durante toda a navegação do usuário no sistema.

7.3. Expressões de ligação de valor

Depois que o *managed bean* é definido no arquivo XML, ele pode ser acessado pelos componentes das páginas do sistema.

A maioria dos componentes JSF possui propriedades que nos permitem especificar um valor ou uma ligação a um valor que está associado a um bean.

Por exemplo, pode-se especificar um valor definitivo (estático) no componente `<h:outputText/>`:

```
<h:outputText value="Hello, World!"/>
```

Ou também uma ligação de valor (dinâmico):

```
<h:outputText value="#{meuBean.nome}"/>
```

Quando o componente for renderizado, o método *getter* da propriedade `nome` será invocado. Já o método *setter* da propriedade será invocado quando o usuário digitar algo no componente de entrada de texto e submeter a página ao servidor.

7.4. Escopo dos beans

Conforme mostrado no arquivo de configuração "WEB-INF/faces-config.xml", a tag `<managed-bean-scope>` define o escopo no qual o bean é armazenado.

Os quatro escopos permitidos são: `none`, `request`, `session` e `application`. Se o escopo for definido como `none`, então o bean receberá uma nova instância a cada vez que for referenciado.

O escopo de `request` (requisição) tem vida curta. Ele começa quando uma requisição HTTP é submetida e termina quando a resposta é enviada de volta ao cliente.

Para uma aplicação onde as transações com o usuário são importantes, utiliza-se o escopo `session`, por exemplo, para um sistema de comércio eletrônico, que detêm um carrinho de compras virtual.

O escopo `application` persiste por toda a duração da execução da aplicação web. Este escopo é compartilhado por todas as requisições e por todas as sessões de todos os usuários, portanto, tome muito cuidado ao usá-lo.

7.5. Backing beans

Em algumas ocasiões, seu bean pode precisar ter acesso aos componentes da página JSF. O acesso direto aos componentes é diferente do acesso aos valores. Este acesso dá a possibilidade de inspecionar e até modificar propriedades do componente que está sendo renderizado para o usuário, que as vezes pode até não estar disponível como um atributo da tag JSF. Por exemplo, um componente de entrada de texto `<h:inputText/>`, representado como objeto Java do tipo `UIInput`, pode ter a propriedade `disabled` ou `required` modificadas em tempo de execução pelo código Java, através do acesso direto a este objeto.

Para esta ligação entre os componentes da página e propriedades de beans, precisamos criar um *backing bean*. Um bean deste tipo é igual ao *managed bean*. A única diferença é que ele, além de fazer ligações de valor, pode também fazer ligação de componentes, porém a forma de configurá-lo no arquivo "faces-config.xml" é a mesma.

Para um bean ser caracterizado como um *backing bean*, no código-fonte da página é feita uma amarração (melhor chamado de *binding*) em uma tag de um componente JSF para uma propriedade de um *backing bean*. Para conectar os componentes do formulário com as propriedades do bean, usa-se o atributo `binding`:

```
<h:outputText binding="#{meuBean.nomeComponente}"/>
```

No backing bean "meuBean", temos a propriedade e os métodos de acesso:

```
private UIOutput nomeComponente;

public UIOutput getNomeComponente() {
```

```
        return nomeComponente;
    }

    public void setNomeComponente(UIComponent value) {
        this.nomeComponente = value;
    }
```

Neste caso, o componente `<h:outputText/>` é representado como objeto Java do tipo `UIOutput`. Em qualquer método do bean, poderíamos acessar as propriedades do componente, como por exemplo, para ler o valor, chamaríamos o método *getter* da propriedade `value`:

```
this.nomeComponente.getValue();
```

Apesar de poderoso, este recurso deve ser usado com bastante cuidado. O uso excessivo pode deixar o código-fonte grande e difícil de entender, além de que, se seu bean tiver escopo de sessão ou aplicação, seus componentes visuais podem ficar "presos" na memória por bastante tempo e consumir muitos recursos do servidor.

7.6. Definindo valores de propriedades

No momento em que se configura um bean, é possível definir alguns valores iniciais para as suas propriedades, como mostrado no exemplo abaixo:

```
<managed-bean>
    <managed-bean-name>meuBean</managed-bean-name>
    <managed-bean-class>com.algaworks.MeuBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>idade</property-name>
        <property-value>18</property-value>
    </managed-property>
    <managed-property>
        <property-name>sexo</property-name>
        <property-value>masculino</property-value>
    </managed-property>
</managed-bean>
```

Quando o bean for invocado, seu construtor padrão `MeuBean()` é chamado e em seguida os métodos `setIdade` e `setSexo` são executados, passando o número 18 para a idade e "masculino" para o sexo. Para inicializar uma propriedade com valor nulo, utiliza-se o elemento `<null-value/>`. Por exemplo:

```
<managed-property>
    <property-name>nome</property-name>
    <null-value/>
</managed-property>
```

7.7. Inicializando listas e mapas

Freqüentemente, as propriedades dos beans são listas ou mapas, da Collections API. Você pode inicializá-las a partir do arquivo de configuração "faces-config.xml". O exemplo abaixo apresenta a inicialização de uma lista:

```
<managed-bean>
    <managed-bean-name>meuBean</managed-bean-name>
    <managed-bean-class>com.algaworks.MeuBean</managed-bean-class>
```

```
	<managed-bean-scope>session</managed-bean-scope>
	<managed-property>
		<property-name>sexos</property-name>
		<list-entries>
			<value-class>java.lang.String</value-class>
			<value>Masculino</value>
			<value>Feminino</value>
		</list-entries>
	</managed-property>
</managed-bean>
```

A lista pode conter uma mistura de elementos `value` e `null-value`. O `value-class` é opcional, se for omitido uma lista de String é produzida.

Mapas são um pouco mais complexos. Lembre-se que os mapas são formados com chave (*key*) e valor (*value*). Opcionalmente especificam-se os elementos `key-class` e `value-class`. Caso forem omitidos, o valor default String é utilizado. Em seguida, aplica-se uma seqüência de elementos `map-entry` cada um dos quais seguidos de um elemento `key` e um elemento `value` ou `null-value`. Por exemplo:

```
	<map-entries>
		<key-class>java.lang.Integer</key-class>
		<map-entry>
			<key>1</key>
			<value>Pedro</value>
		</map-entry>
		<map-entry>
			<key>2</key>
			<value>Maria</value>
		</map-entry>
	</map-entries>
```

7.8. Usando expressões compostas

Os operadores que podem ser utilizados em uma expressão de ligação de valor são:

- Operadores aritméticos `+ - * / %`. Os dois últimos também podem ser utilizados utilizando as variantes alfabéticas `div` e `mod`.
- Operadores relacionais `< <= > => == !=` e suas variantes alfabéticas `lt le gt ge ne`.
- Operadores lógicos `&& || !` e suas variantes alfabéticas `and or not`.
- O operador `empty`, para verificar se uma variável é nula ou vazia ou uma coleção ou array é de tamanho igual a zero.
- O operador de seleção ternário `condicao ? se_verdadeiro : se_falso`.

A precedência dos operadores segue a mesma regra que em Java.
Não é uma boa prática inserir muitas regras de negócio nas páginas web. Isso violaria a separação entre apresentação e lógica de negócio. Portanto, tome cuidado ao inserir expressões muito complexas para fazer muitos cálculos nas páginas, onde elas deveriam estar nos objetos de negócio.

Uma situação onde é muito comum utilizar os operadores é no momento em que se deseja fazer sumir um campo caso a propriedade de outro seja falso. Por exemplo:

```
	<h:inputText rendered="#{!meuBean.solteiro}"
	     value="#{meuBean.nomeEsposa}"/>
```

Quando a propriedade `solteiro` de `meuBean` for `false`, então a expressão acima não é renderizada.

Para concluir este tópico, você pode concatenar strings com expressões de ligação de valor, simplesmente colocando-as lado a lado, por exemplo:

```
<h:outputText value"#{meuBean.nome}, #{meuBean.sobrenome}"/>
```

7.9. Usando expressões de ligação de método

Alguns componentes também podem chamar um método em um *managed bean*, por exemplo:

```
<h:commandButton action="#{meuBean.efetivarTransacao}"/>
```

Quando o usuário clicar no botão, o método do bean será invocado. Para este caso o método deve ter a seguinte assinatura:

```
public String efetivarTransacao()
```

A String retornada será passada para o *handler* de navegação (discutido em outro capítulo), que definirá para qual tela será redirecionado.

São quatro os atributos de componentes que podem ser usados com uma expressão de ligação de método (todos serão apresentados em outros capítulos):

- `action`
- `validator`
- `actionListener`
- `valueChangeListener`

Os tipos de parâmetros e de retorno do método dependem de qual maneira foi utilizada a ligação do método (qual atributo de componente foi utilizado).

7.10. Aplicação de exemplo

Para demonstrar as funcionalidades descritas até agora, será apresentada uma aplicação de agenda de contatos. O sistema deve contemplar:

- Uma tela de cadastro para o usuário entrar com o nome, endereço, sexo e profissão;
- Os dados digitados serão apresentados em uma grid;
- O sistema não armazenará os cadastros em banco de dados, mas apenas na memória.

Para começar, crie no Eclipse um projeto com o nome “Agenda” e configure o ambiente para a aplicação JSF.

Crie a classe `Contato`, conforme o código abaixo:

```
package com.algaworks.dwjsf.dominio;

public class Contato {

	private String nome;
	private String endereco;
	private String sexo;
```

```
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public String getEndereco() {
		return endereco;
	}
	public void setEndereco(String endereco) {
		this.endereco = endereco;
	}
	public String getSexo() {
		return sexo;
	}
	public void setSexo(String sexo) {
		this.sexo = sexo;
	}

}
```

A classe `Contato` é um JavaBean que representa o modelo de dados de um contato. Isso quer dizer que cada contato incluído na agenda será uma instância dessa classe.

Agora crie a classe do *managed bean*, chamada `AgendaContatoBean`:

```
package com.algaworks.dwjsf.visao;

import java.util.ArrayList;
import java.util.List;

import javax.faces.event.ActionEvent;

import com.algaworks.dwjsf.dominio.Contato;

public class AgendaContatoBean {

	private List<Contato> contatos;
	private Contato contato;

	public AgendaContatoBean() {
		this.setContatos(new ArrayList<Contato>());
		this.setContato(new Contato());
	}

	public void incluirContato(ActionEvent event) {
		this.getContatos().add(this.getContato());
		this.setContato(new Contato());
	}

	public List<Contato> getContatos() {
		return contatos;
	}
	public void setContatos(List<Contato> contatos) {
		this.contatos = contatos;
	}
	public Contato getContato() {
		return contato;
	}
	public void setContato(Contato contato) {
		this.contato = contato;
```

```
        }

    }
```

Configure o *managed bean* da agenda de contatos, incluindo o código abaixo no arquivo "faces-config.xml":

```
    <managed-bean>
        <managed-bean-name>agendaContatoBean</managed-bean-name>
        <managed-bean-class>
            com.algaworks.dwjsf.visao.AgendaContatoBean
        </managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
```

Crie agora o arquivo JSP referente a tela da agenda de contatos. Este arquivo pode se chamar "agendaContato.jsp":

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<html>
<f:view>
   <head>
      <title>Agenda de Contatos</title>
   </head>
   <body>
      <h:form>
         <h:outputText value="Nome: " />
         <h:inputText value="#{agendaContatoBean.contato.nome}"/>

         <br/>

         <h:outputText value="Endereço: " />
         <h:inputText value="#{agendaContatoBean.contato.endereco}"/>

         <br/>

         <h:outputText value="Sexo: " />
         <h:inputText value="#{agendaContatoBean.contato.sexo}" />

         <br/>

         <h:commandButton
            actionListener="#{agendaContatoBean.incluirContato}"
            value="Incluir" />

         <br/><br/>

         <h:dataTable var="obj" value="#{agendaContatoBean.contatos}"
            border="1" width="100%">
            <h:column>
               <f:facet name="header">
                  <h:outputText value="Nome"/>
               </f:facet>
               <h:outputText value="#{obj.nome}" />
            </h:column>
            <h:column>
               <f:facet name="header">
                  <h:outputText value="Endereço"/>
               </f:facet>
```

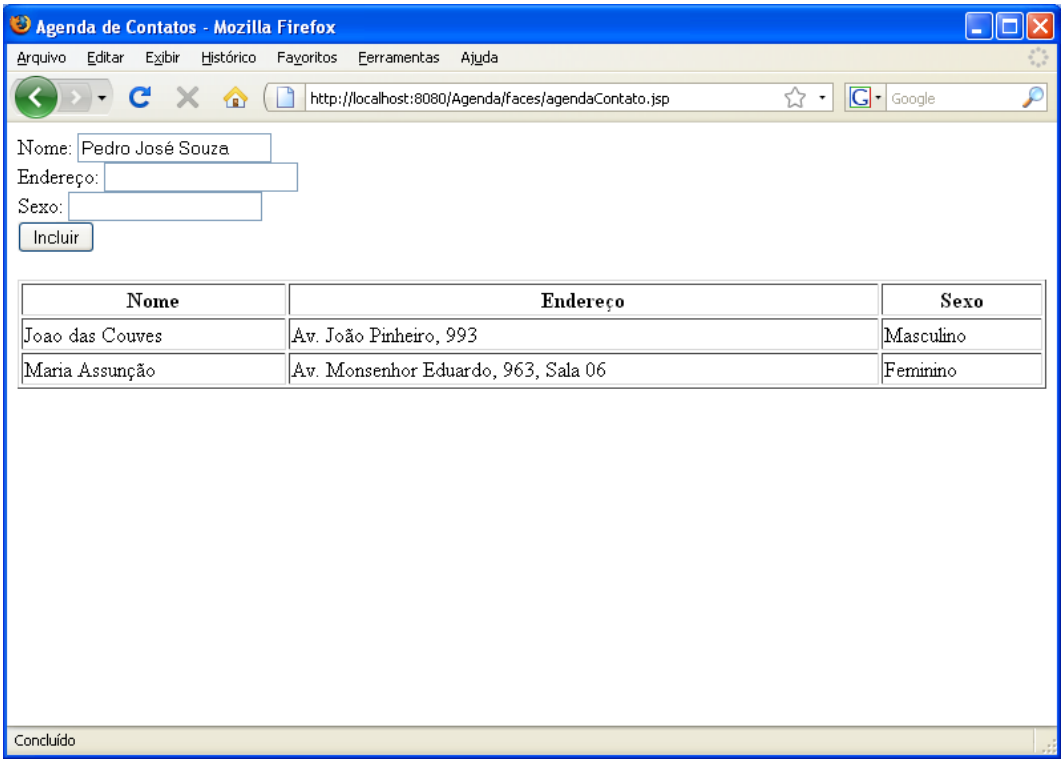
```
                <h:outputText value="#{obj.endereco}" />
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Sexo"/>
                </f:facet>
                <h:outputText value="#{obj.sexo}" />
            </h:column>
        </h:dataTable>
    </h:form>
  </body>
</f:view>
</html>
```

Veja que as ligações de valores dos campos de entrada foram feitas para um atributo da classe `Contato`, através de outro atributo chamado `contato` na classe `AgendaContatoBean`.

Os detalhes de componentes mais avançados, como o `<h:dataTable>`, será visto em capítulos posteriores. Neste momento, basta perceber como são feitas as expressões de ligações de valores com o *managed bean*.

O método `incluirContato(ActionEvent event)` será invocado quando o botão "Incluir" for pressionado, pois foi feita a ligação de método usando o atributo `actionListener` (estudaremos melhor este atributo mais tarde).

Adicione o projeto da agenda no Tomcat, inicie o serviço e teste a inclusão de contatos.



8. Navegação

8.1. Introdução

A JSF possibilita a configuração de regras de navegação, o que permite você definir a conexão entre as páginas. Por exemplo, quando um usuário clica em um botão para se inscrever em um site, qual tela ele deverá ver? Se os dados estiverem incompletos, provavelmente deverá visualizar a mesma tela, com as mensagens de erro apropriadas, porém se tudo estiver correto e a inscrição for efetuada com sucesso, ele poderá ver uma tela de boas vindas ao serviço. Isso é chamado de regras de navegação, e você pode configurá-las no arquivo "faces-config.xml", como veremos neste capítulo.

8.2. Navegação simplificada

Em várias aplicações web, normalmente são necessárias algumas regras de navegações simplificadas (estáticas). Um exemplo desse tipo de navegação é quando o usuário clica sobre um botão ou link para se inscrever em um determinado serviço de seu site, que sempre apresenta a tela de inscrição online. Neste caso, a regra para seleção e exibição da tela é simples, pois não se deseja que seja exibida outra tela, independente do que possa acontecer.

Para configurar a navegação estática, edite o arquivo "faces-config.xml" e inclua uma regra de navegação, como o exemplo abaixo:

```
<navigation-rule>
      <navigation-case>
            <from-outcome>ajuda</from-outcome>
            <to-view-id>/ajudaOnline.jsp</to-view-id>
      </navigation-case>
</navigation-rule>
```

Essa regra define que a ação de nome "ajuda" deve encaminhar para a página "ajudaOnline.jsp". Para usar esta regra, podemos simplesmente adicionar um botão na página com o atributo `action` referenciando-a, conforme o trecho de código abaixo:

```
<h:commandButton value="Preciso de ajuda!" action="ajuda"/>
```

Para testar, crie uma página JSF qualquer com o nome "ajudaOnline.jsp", configure a regra de navegação e inclua o botão em uma outra página. Clique neste botão e veja que você será encaminhado para a página de ajuda.

8.3. Filtro pela visão de origem

No último exemplo, configuramos uma regra para atender toda a aplicação em relação a página de ajuda, ou seja, qualquer página poderia referenciar a ação “ajuda” para ser encaminhado para a página “ajudaOnline.jsp”. Imagine agora se precisássemos criar uma página de ajuda para cada tela do sistema (o que é muito comum em aplicações médias e grandes). Poderíamos resolver essa questão simplesmente configurando várias regras de navegação:

```
<navigation-rule>
	<navigation-case>
		<from-outcome>ajudaAgenda</from-outcome>
		<to-view-id>/ajudaAgenda.jsp</to-view-id>
	</navigation-case>
	<navigation-case>
		<from-outcome>ajudaRelatorios</from-outcome>
		<to-view-id>/ajudaRelatorios.jsp</to-view-id>
	</navigation-case>
	<navigation-case>
		<from-outcome>ajudaConfiguracao</from-outcome>
		<to-view-id>/ajudaConfiguracao.jsp</to-view-id>
	</navigation-case>
</navigation-rule>
```

Na página da agenda, você pode incluir o botão:

```
<h:commandButton value="Ajuda" action="ajudaAgenda"/>
```

Na página de relatórios, o botão:

```
<h:commandButton value="Ajuda" action="ajudaRelatorios"/>
```

E assim por diante.

Essa estratégia funciona, porém pode não ser a melhor. O JSF pode fazer muito mais por você, com o elemento `from-view-id`.

Com este elemento, você pode configurar o seguinte no arquivo “faces-config.xml”:

```
<navigation-rule>
	<from-view-id>/agendaContato.jsp</from-view-id>
	<navigation-case>
		<from-outcome>ajuda</from-outcome>
		<to-view-id>/ajudaAgenda.jsp</to-view-id>
	</navigation-case>
</navigation-rule>
<navigation-rule>
	<from-view-id>/relatorio.jsp</from-view-id>
	<navigation-case>
		<from-outcome>ajuda</from-outcome>
		<to-view-id>/ajudaRelatorios.jsp</to-view-id>
	</navigation-case>
</navigation-rule>
<navigation-rule>
	<from-view-id>/configuracao.jsp</from-view-id>
	<navigation-case>
		<from-outcome>ajuda</from-outcome>
		<to-view-id>/ajudaConfiguracao.jsp</to-view-id>
	</navigation-case>
</navigation-rule>
```

Essas regras dizem que se a ação dos botões for igual a “ajuda”, o usuário será encaminhado para alguma das páginas de ajuda, dependendo de onde ele estiver. Por exemplo, se o usuário estiver visualizando a página da agenda de contato, ele será encaminhado para “ajudaAgenda.jsp”, mas se estiver visualizando a página de relatórios, será encaminhado para “ajudaRelatorios.jsp”.

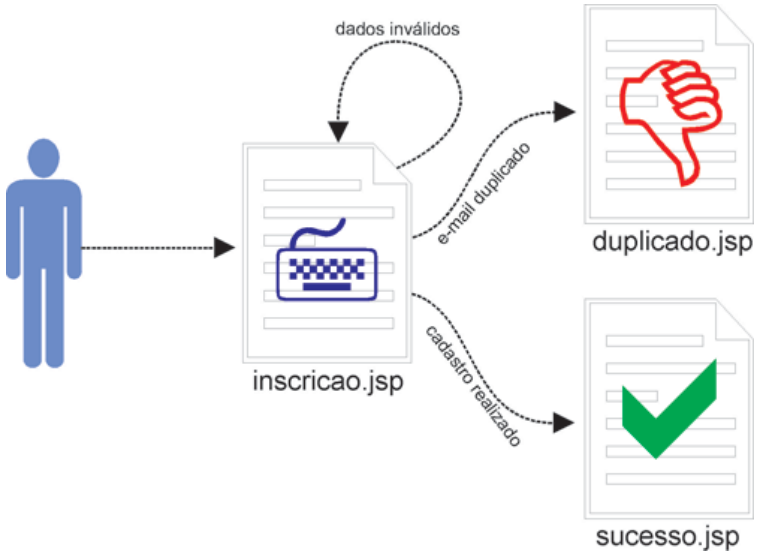
Em todas as páginas do sistema, basta incluir um botão com a ação “ajuda”, e o framework JSF validará qual regra deve ser usada para encaminhar para a página adequada:

```
<h:commandButton value="Ajuda" action="ajuda"/>
```

8.4. Navegação dinâmica

Na maioria dos casos, você precisará que o fluxo das páginas não dependa apenas do botão clicado, mas também das entradas fornecidas pelo usuário e/ou das regras de negócio da aplicação. Quando essa necessidade surgir, você precisará configurar a navegação dinâmica.

Por exemplo, se você estiver desenvolvendo uma tela para seus clientes se inscreverem em uma promoção de sua empresa, fornecendo apenas o nome e e-mail, esse cadastro pode ser bem sucedido ou não, dependendo do que o usuário digitar. Se ele fornecer um e-mail inválido ou um nome muito curto, você deve negar a inscrição e solicitar a correção dos dados, se o e-mail digitado já estiver cadastrado, você não pode permitir duplicatas, e por isso deve avisá-lo sobre o ocorrido, mas se aparentemente os dados estiverem corretos e sem duplicidade, o cadastro deve ser realizado e o usuário deve ser encaminhado para uma página de sucesso. Veja que o encaminhamento do usuário para alguma página é dinâmico, conforme o fluxo abaixo:



Agora vamos ver como ficariam os códigos para criar este fluxo dinâmico. Não trataremos ainda sobre as mensagens de erro do JSF, por isso faremos de uma forma que não é o mais indicado, mas fique tranqüilo, pois abordaremos este assunto em outro capítulo.

No arquivo “faces-config.xml”, configuramos todas as regras de navegação:

```
<navigation-rule>
	<from-view-id>/inscricao.jsp</from-view-id>
	<navigation-case>
		<from-outcome>duplicado</from-outcome>
		<to-view-id>/duplicado.jsp</to-view-id>
	</navigation-case>
	<navigation-case>
		<from-outcome>sucesso</from-outcome>
```

```
                <to-view-id>/sucesso.jsp</to-view-id>
            </navigation-case>
        </navigation-rule>
```

Definimos que, quando o usuário estiver na página de inscrição, existirá uma regra que o encaminhará para a página “duplicado.jsp” e outra que encaminhará para “sucesso.jsp”. Não precisamos criar uma regra para encaminhar para a própria página, e você já descobrirá o motivo.

Criamos a classe do *managed bean*:

```
package com.algaworks.dwjsf.visao;

public class InscricaoBean {

	private String nome;
	private String email;
	private String mensagem;

	public String inscrever() {
		if (nome == null || nome.length() < 10
				|| "".equals(email.trim())) {
			this.setMensagem("Informe corretamente o nome "
					+ "e e-mail!");
			return null;
		}
		//verifica se cadastro é duplicado
		boolean duplicado = false;
		if (duplicado) {
			return "duplicado";
		}

		//armazena inscrição, limpa atributos do formulário
		//e retorna com sucesso
		//...
		return "sucesso";
	}

	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public String getMensagem() {
		return mensagem;
	}
	public void setMensagem(String mensagem) {
		this.mensagem = mensagem;
	}

}
```

O método `inscrever()` retorna uma String como resultado. O retorno do método será usado pelo *handler* de navegação para encontrar a regra adequada para o encaminhamento.

Por exemplo, se o retorno for “duplicado”, o usuário será encaminhado para “duplicado.jsp”, se o retorno for “sucesso”, o usuário será encaminhado para “sucesso.jsp”, mas se o retorno for nulo, a mesma página é re-exibida.

Configuramos também o *managed bean* no arquivo “faces-config.xml”:

```
<managed-bean>
	<managed-bean-name>inscricaoBean</managed-bean-name>
	<managed-bean-class>
		com.algaworks.dwjsf.visao.InscricaoBean
	</managed-bean-class>
	<managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

E criamos o arquivo JSP da página de inscrição:

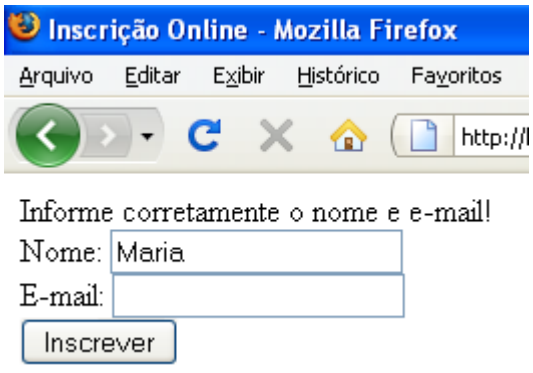
```
<f:view>
	<h:form>
		<h:outputText value="#{inscricaoBean.mensagem}"
			rendered="#{not empty inscricaoBean.mensagem}"/>
		<br/>

		<h:outputText value="Nome: " />
		<h:inputText value="#{inscricaoBean.nome}"/>

		<br/>

		<h:outputText value="E-mail: " />
		<h:inputText value="#{inscricaoBean.email}"/>

		<br/>

		<h:commandButton action="#{inscricaoBean.inscrever}"
			value="Inscrever" />

	</h:form>
</f:view>
```

O componente de saída de texto (`h:outputText`) da mensagem só será renderizado caso a mensagem não seja vazia.

Para fazer referência ao método `inscrever()`, o botão foi criado da seguinte forma:

```
<h:commandButton action="#{inscricaoBean.inscrever}"
	value="Inscrever" />
```

Agora crie os arquivos “duplicado.jsp” e “sucesso.jsp” com algum texto e teste a tela de inscrição, informando entradas corretas e incorretas.



8.5. Filtro pela ação de origem

A definição do elemento `navigation-case` nos dá outras possibilidades mais complexas para configuração das regras. Com o elemento `from-action`, é possível separar dois métodos de *managed beans* que retornam a mesma string de ação.

Imagine que precisamos colocar o cancelamento de inscrição no nosso último exemplo. Podemos criar um método `cancelarInscricao()` no *managed bean* `InscricaoBean`, que retorna a String "sucesso", caso o cancelamento seja efetuado. Como o método `inscrever()` também retorna a String "sucesso", precisamos criar uma regra para dizer que o retorno de sucesso de cada método deve ter um significado diferente, encaminhando o usuário para páginas distintas. No arquivo "faces-config.xml", simplesmente colocamos:

```
<navigation-rule>
	...
	<navigation-case>
		<from-action>#{inscricaoBean.inscrever}</from-action>
		<from-outcome>sucesso</from-outcome>
		<to-view-id>/sucesso.jsp</to-view-id>
	</navigation-case>
	<navigation-case>
		<from-action>
			#{inscricaoBean.cancelarInscricao}
		</from-action>
		<from-outcome>sucesso</from-outcome>
		<to-view-id>/cancelado.jsp</to-view-id>
	</navigation-case>
</navigation-rule>
```

Agora o *handler* de navegação irá comparar o nome do método que está dentro dos delimitadores `#{}` para encontrar a regra a ser usada no encaminhamento do usuário.

8.6. Usando redirecionamentos

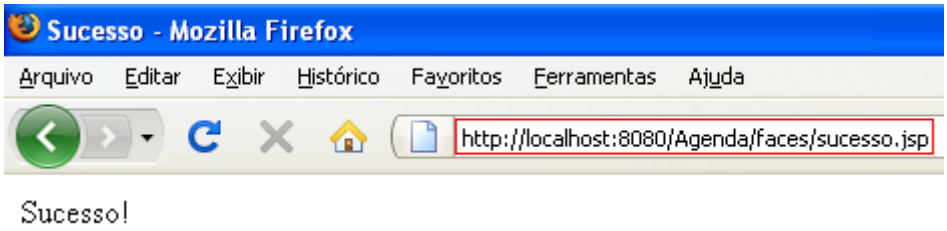
O elemento `<redirect/>`, colocado após `<to-view-id>`, diz ao *handler* de navegação que deve ser feito um redirecionamento, e não um encaminhamento.

O redirecionamento muda a URL no navegador, enquanto o encaminhamento não faz essa atualização. O redirecionamento também é mais lento que o encaminhamento, pois precisa trafegar mais informações entre o servidor e cliente.

Para testar o redirecionamento, altere a regra de navegação para incluir este novo elemento:

```
<navigation-case>
	<from-action>#{inscricaoBean.inscrever}</from-action>
	<from-outcome>sucesso</from-outcome>
	<to-view-id>/sucesso.jsp</to-view-id>
	<redirect/>
</navigation-case>
```

Execute a tela de inscrição, preencha todos os dados e submeta o formulário. Veja que o endereço no browser foi alterado.



Esta técnica é importante para quando você deseja que o usuário possa incluir a URL nos favoritos do browser. Com encaminhamento, isso não se torna possível.

8.7. Usando padrões

Você pode agrupar regras de navegação usando padrões no elemento `<from-view-id>`, como:

```
<navigation-rule>
	<from-view-id>/admin/*</from-view-id>
	<navigation-case>
		...
	</navigation-case>
</navigation-rule>
```

Essa regra diz que os casos de navegação serão aplicados apenas para as páginas que iniciam com “/admin/”.

9. Componentes básicos

9.1. Introdução

Para desenvolver um sistema em JSF completo, você deve conhecer pelo menos os principais componentes básicos. Quando falamos em componentes básicos, queremos dizer os componentes da implementação de referência do JSF, ou seja, aqueles que já fazem parte do framework, sem ter que instalar bibliotecas adicionais. Até o final deste curso, aprenderemos a utilizar bibliotecas de componentes adicionais, porém os componentes básicos são essenciais para o desenvolvimento de qualquer sistema.

As bibliotecas de tags JSF são divididas em HTML e fundamentais (*core*).

A biblioteca HTML possui componentes que geram conteúdo visual, como formulários, campos de entrada, rótulos de saída de textos, botões, links, seleções, painéis, tabela de dados, mensagens e etc. Aprenderemos a maioria destes componentes neste capítulo.

A biblioteca fundamental existe para dar suporte à biblioteca HTML, e não possui componentes visuais, mas apenas auxiliares. Aprenderemos esta biblioteca no decorrer do curso, sempre que for necessário.

Para usar as bibliotecas da JSF, temos que importá-las com diretivas *taglib*, como no exemplo abaixo:

```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
```

Importamos as tags fundamentais e HTML e as nomeamos com o prefixo *f* e *h*, respectivamente. Este nome (prefixo) pode ser modificado para qualquer outro, mas é melhor que use *f* e *h*, que são prefixos convencionados.

9.2. Formulários

O uso de formulários é comum em qualquer sistema web, usando qualquer tecnologia. O framework JSF possui um componente para renderizar formulários HTML chamado `h:form`.

Veja um exemplo abaixo que inclui um formulário:

```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<body>
      <f:view>

            <h:form>

                  ...campos do formulário aqui...

            </h:form>

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

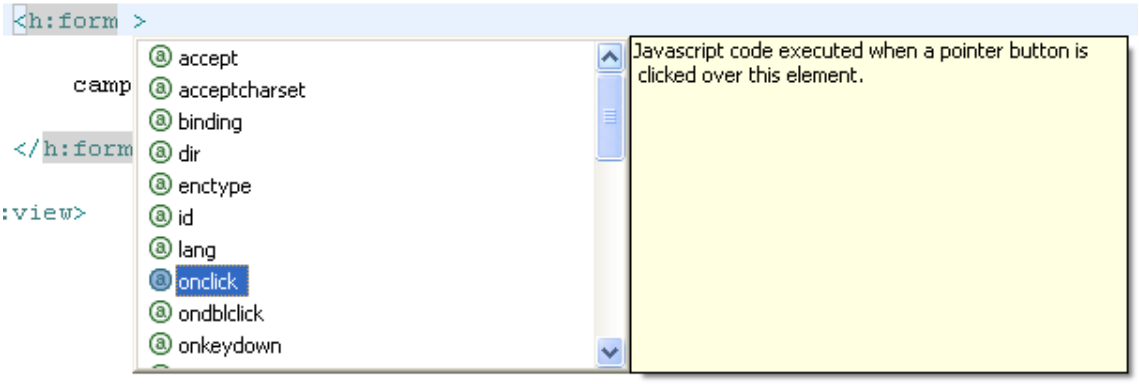
O código-fonte gerado ao executar esta página será:

```
<html>
<body>

        <form id="j_id_jsp_2057641978_1" name="j_id_jsp_2057641978_1"
           method="post"
           action="/Agenda/faces/teste.jsp"
           enctype="application/x-www-form-urlencoded">
```

```
                <input type="hidden" name="j_id_jsp_2057641978_1"
                   value="j_id_jsp_2057641978_1" />

                ...campos do formulário aqui...

                <input type="hidden" name="javax.faces.ViewState"
                   id="javax.faces.ViewState" value="j_id1:j_id2" />
        </form>
</body>
</html>
```

A tag `h:form` gerou um elemento `form` da HTML. Apesar da tag `form` da HTML possuir os atributos `method` e `action`, a tag `f:form` do JSF não possui, pois sempre é considerado o método "post" e a ação igual ao endereço da mesma página, como você pode ver no código gerado.

A tag `h:form` possui vários atributos, sendo que a maioria deles são referentes aos eventos da DHTML. O Eclipse pode te ajudar a descobrir todos os atributos, basta pressionar *Ctrl + Espaço* dentro do código da tag.



Estudaremos no próximo tópico os atributos comuns da maioria dos componentes, que servem inclusive para o `h:form`.

9.3. Um pouco sobre os atributos comuns

As tags dos componentes possuem atributos que dizem como eles devem funcionar. Os atributos podem ser básicos, quando são compartilhados pela maioria das tags, HTML, também conhecido como *pass-through* HTML, quando representam os mesmos atributos dos elementos HTML, e eventos DHTML, quando dão suporte a scripts de eventos, como ao clicar, ao passar o mouse por cima, etc.

O atributo `id`

O atributo `id`, presente em todos os componentes, nos permite identificar os componentes da página para acesso posterior através de classes Java ou outros componentes JSF, além de poder acessar os elementos da HTML através de scripts. Para exemplificar, criaremos uma página com um campo de entrada de texto e um botão. Apenas para simplificar, o botão não será criado usando componentes JSF. Ao clicar no botão, um código JavaScript irá alterar o conteúdo do campo de entrada.

```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<body>
```

```
        <script language="JavaScript">
                function alterarValorCampo() {
                        document.getElementById('meuForm:meuCampo')
                                .value = 'Valor alterado';
                }
        </script>

        <f:view>
                <h:form id="meuForm">
                        <h:inputText id="meuCampo"/>
                        <input type="button" value="Testar"
                                onclick="alterarValorCampo();"/>
                </h:form>
        </f:view>
</body>
</html>
```

Veja que criamos uma função JavaScript chamada `alterarValorCampo()`. Esta função é chamada no evento `onclick` do botão "Testar", e deve alterar o valor do campo para "Valor alterado".

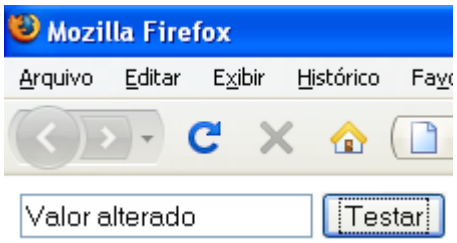
O campo que incluímos é renderizado em HTML da seguinte forma:

```
<input id="meuForm:meuCampo" type="text" name="meuForm:meuCampo" />
```

Veja que o `id` do elemento *INPUT* foi configurado para "meuForm:meuCampo". O `id` do formulário foi juntado com o `id` especificado para o campo de entrada e definido como o identificador do elemento HTML. Com este identificador, usamos a seguinte programação JavaScript para acessá-lo:

```
document.getElementById('meuForm:meuCampo').value = 'Valor alterado';
```

Ao executar a página e clicar no botão "Testar", o valor do campo de entrada é alterado.



O atributo `binding`

O atributo `binding` pode ser especificado com uma expressão ligação que referencia uma propriedade do bean do tipo do componente. Usamos este atributo em um capítulo anterior, quando aprendemos sobre *backing beans*. Este atributo está presente em todos os componentes da biblioteca HTML.

O atributo `rendered`

O atributo `rendered` também está presente na maioria das tags. Este atributo controla a renderização do componente. Se o valor dele for `false`, o componente não será exibido, e se for `true`, será renderizado para o browser do usuário. Você pode usar uma expressão de ligação que referencia uma propriedade do bean para controlar quando o componente será

renderizado. Já usamos este atributo em um exemplo anterior, quando criamos nossa primeira aplicação JSF.

Os atributos `style` e `styleClass`

É possível utilizar estilos CSS (*Cascade Style Sheet*) em suas páginas JSF de modo *inline* ou em classes. Sempre que for possível, prefira criar os estilos em um arquivo separado e usá-los em classes. O exemplo a seguir mostra o uso das duas formas.

Primeiramente, criamos um arquivo CSS de nome “estilo.css” em uma pasta chamada “css”, com o seguinte conteúdo:

```
.campo {

      background-color: #CCCCCC;
      color: white;

}
```

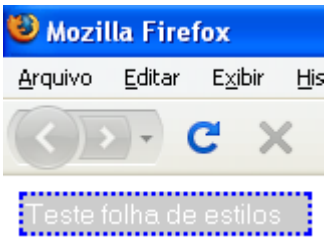
Este arquivo CSS especifica apenas uma classe chamada “campo", que será usada para configurar a cor de fundo e da fonte de campos de entrada.

Em seguida, criamos uma página simples em JSF com apenas um campo para testarmos o uso desta classe e também especificarmos outros estilos adicionais *inline*.

```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<head>
     <link rel="stylesheet" type="text/css" href="../css/estilo.css"/>
</head>
<body>
     <f:view>
         <h:form>
            <h:inputText styleClass="campo"
                  style="border-color: blue; border-style: dotted;" />
         </h:form>
     </f:view>
</body>
</html>
```

Veja que no componente `inputText`, usamos o atributo `styleClass` para referenciar o nome da classe que criamos no arquivo CSS e o atributo `style` para declarar outras características do mesmo componente, como cor e estilo da borda.

Ao executar a página, você verá o campo com fundo cinza, cor de fonte branca e bordas azuis pontilhadas.



A maioria dos componentes da biblioteca HTML possuem os atributos `style` e `styleClass`.

Os atributos da HTML

Os atributos da HTML, também conhecidos como *pass-through* HTML, representam exatamente os atributos de elementos da própria HTML. Não vamos listar todos eles aqui, mas apenas usar alguns como exemplos. Caso você precise usar algum atributo próprio de um elemento da HTML e não souber se existe, tente descobrir usando as teclas *Ctrl + Espaço* dentro da tag, que o Eclipse irá lhe auxiliar.

Em uma página JSF, criamos um campo de entrada com o seguinte código:

```
<h:inputText size="40" maxlength="60"
      title="Informe seu nome neste campo" />
```

Veja como este componente é renderizado para o usuário:

```
<input type="text"
       name="j_id_jsp_2057641978_1:j_id_jsp_2057641978_2"
       maxlength="60" size="40"
       title="Informe seu nome neste campo" />
```

Os atributos `maxlength`, `size` e `title` simplesmente foram repassados para o código de saída, por isso são chamados de *pass-through* HTML.

Os atributos de eventos DHTML

Ao desenvolver sistemas para web, diversas vezes surge a necessidade de criar scripts que rodam no cliente (*client-side*), como validações, criação de elementos da HTML dinamicamente, exibição ou ocultamento de algum elemento, etc. Os atributos que suportam scripts são chamados de atributos de eventos DHTML. Quase todas as tags da biblioteca HTML possuem esses atributos, mas novamente não iremos listar todos, pois basta você solicitar auxílio ao Eclipse para descobri-los.

No campo de entrada do exemplo abaixo, usamos os atributos `onclick`, `onchange`, `onmouseover` e `onmouseout` para incluir códigos JavaScript para tratar eventos do elemento.

```
<h:inputText onclick="this.value = '';"
             onchange="this.value = this.value.toUpperCase();"
             onmouseover="this.style.backgroundColor = 'yellow';"
             onmouseout="this.style.backgroundColor = 'white';"/>
```

Os códigos JavaScript serão renderizados na saída para o cliente. Quando o usuário clicar sobre o campo, seu valor será alterado para vazio, ao alterar o valor do campo, ele será modificado para caixa alta (tudo em maiúsculo), ao passar o mouse por cima do campo, a cor de fundo será alterada para amarelo e ao retirar o mouse de cima do campo, a cor de fundo voltará para branco.

Este exemplo é bem simples, porém mostra como funcionam os eventos da DHTML.

9.4. Entrada de textos

Existem três tipos de componentes que renderizam campos de entrada de texto: `h:inputText`, `h:inputSecret` e `h:inputTextarea`.

O componente `h:inputText`

O componente `h:inputText` renderiza um campo de entrada de texto simples, representado pelo elemento `input` do tipo “text” da HTML. Já usamos este componente anteriormente para apresentar a expressão de ligação e os atributos comuns. No exemplo a seguir, o campo de entrada representa o atributo `nome` do bean `meuBean`.

```
<h:inputText value="#{meuBean.nome}"/>
```

O componente `h:inputSecret`

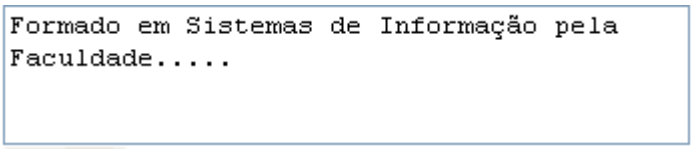
O componente `h:inputSecret` renderiza um campo de entrada de senha, representado pelo elemento `input` do tipo "password" da HTML. O texto digitado é apresentado de forma secreta. O atributo `redisplay`, do tipo boleano, define se o campo deve reter e re-exibir o seu valor quando o formulário for submetido. Por exemplo, em uma tela de login, quando o usuário errar a senha, você quer que ela seje retida no campo ou que apresente o campo vazio para nova digitação? Se escolher reter a senha, informe `true` no atributo `redisplay`.

```
<h:inputSecret value="#{loginBean.senha}" redisplay="true"/>
```

O componente `h:inputTextarea`

O componente h:inputTextarea renderiza um campo de entrada de textos maiores, que podem ter várias colunas e linhas, e é representado pelo elemento textarea na HTML. Os atributos cols e rows definem o tamanho de colunas e linhas da área do texto, respectivamente. O código abaixo cria uma área de texto com 3 linhas e 40 colunas:

```
<h:inputTextarea cols="40" rows="3"
      value="#{cadastroUsuarioBean.resumoCurriculo}"/>
```

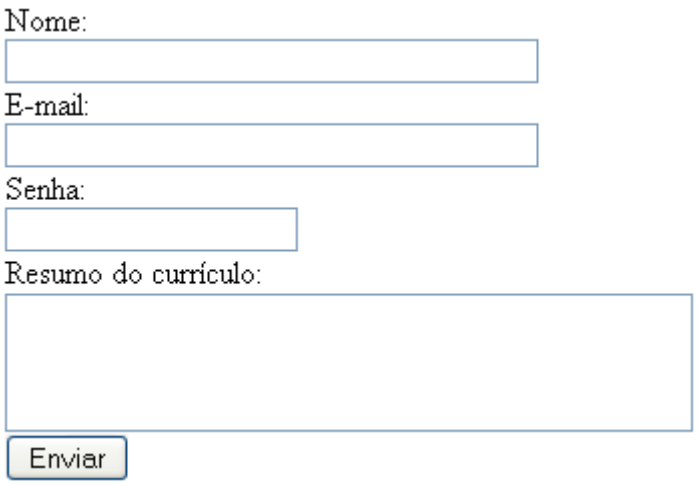


Exemplo usando os três tipos de campos de entrada de texto

Agora vamos ver um único exemplo com o uso dos três tipos de campos de entrada de texto.

```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<body>
<f:view>
      <h:form>
            <h:outputText value="Nome:"/> <br/>
            <h:inputText value="#{cadastroUsuarioBean.nome}" size="40"
                  maxlength="80"/>
            <br/>

            <h:outputText value="E-mail:"/> <br/>
            <h:inputText value="#{cadastroUsuarioBean.email}" size="40"
                  maxlength="250"/>
            <br/>
```

```
            <h:outputText value="Senha:"/> <br/>
            <h:inputSecret value="#{cadastroUsuarioBean.senha}"
                  size="20"/>
            <br/>

            <h:outputText value="Resumo do currículo:"/> <br/>
            <h:inputTextarea cols="40" rows="3"
                  value="#{cadastroUsuarioBean.resumoCurriculo}"/>
            <br/>

            <h:commandButton value="Enviar"/>
      </h:form>
</f:view>
</body>
</html>
```



Para testar este último exemplo, crie um bean com os atributos `nome`, `email`, `senha` e `resumoCurriculo` (todos do tipo String) e registre-o como um *managed bean* no arquivo "faces-config.xml".

9.5. Saída de textos

Existem três tipos de componentes que renderizam saídas de textos: `h:outputText`, `h:outputFormat` e `h:outputLabel`.

O componente `h:outputText`

O componente `h:outputText` renderiza textos simples na página.

```
<h:outputText value="Bem-vindo "/>
<h:outputText value="#{segurancaBean.nomeUsuarioLogado}"/>
```

O exemplo acima renderiza o texto a seguir, sendo que NOME seria substituído pelo nome do usuário logado, caso tivéssemos desenvolvido esta funcionalidade.

```
Bem-vindo NOME
```

Perceba que o texto gerado não está envolto por nenhum elemento da HTML. Quando usamos algum atributo que deve ser refletido no código HTML, como o `id` ou `style`, o texto é gerado dentro da tag `span`. Por exemplo:

```
<h:outputText value="Bem-vindo " id="txtBemVindo"/>
<h:outputText value="#{segurancaBean.nomeUsuarioLogado}"
      style="color: red"/>
```

Os componentes acima são renderizados como:

```
<span id="frm:txtBemVindo">Bem-vindo </span>
<span style="color: red">NOME</span>
```

O componente `h:outputFormat`

O componente `h:outputFormat` renderiza textos parametrizados na página. Os textos parametrizados são compostos por espaços reservados (placeholder) que são substituídos por valores no momento da renderização.

Os valores parametrizados são definidos com números entre chaves, iniciando a partir do número zero. As definições dos valores são feitas através da tag `f:param` da biblioteca fundamental (core).

```
<h:outputFormat value="Oi {0}! Existem {1} tarefas pendentes.">
      <f:param value="#{tarefaBean.nomeUsuario}" />
      <f:param value="#{tarefaBean.qtdeTarefasPendentes}" />
</h:outputFormat>
```

Considerando que o nome do usuário seja “Manoel” e que existem 8 tarefas pendentes, a saída em HTML será:

```
Oi Manoel! Existem 8 tarefas pendentes.
```

O componente `h:outputLabel`

O componente `h:outputLabel` renderiza o elemento `label` da HTML. Os componentes `h:outputLabel` são vinculados com outros através do atributo `for`. O uso deste componente é justificado para rotular principalmente campos de seus formulários.

```
<h:outputLabel value="Nome:" for="nomeCampo" id="nomeLabel"/>
<br/>
<h:inputText id="nomeCampo" value="#{cadastroUsuarioBean.nome}"/>
```

O código acima renderiza a seguinte saída em HTML:

```
<label id="frm:nomeLabel" for="frm:nomeCampo">
Nome:</label> <br/>
<input id="frm:nomeCampo" type="text" name="frm:nomeCampo" />
```

9.6. Imagens

O componente `h:graphicImage` renderiza o elemento `img` da HTML, que exibe uma imagem na página. O endereço da imagem deve ser informado no atributo `value` ou `url`, pois um é atalho para o outro. Este componente lhe permite usar o caminho relativo ao contexto, ou seja, você não precisa informar o nome da aplicação, pois é colocado automaticamente. Veja o exemplo:

```
<h:graphicImage value="/imagens/logo_algaworks.gif" width="81"
      height="35"/>
```

O código acima renderiza a seguinte saída em HTML:

```
<img src="/Agenda/imagens/logo_algaworks.gif" height="35"
 width="81" />
```

9.7. Menus, caixas de listagem e item de seleção

Existem 4 componentes que renderizam elementos de menus e caixas de listagem: `h:selectOneMenu`, `h:selectManyMenu`, `h:selectOneListbox`, `h:selectManyListbox`.

O componente `h:selectOneMenu` e a tag `f:selectItem`

O componente `h:selectOneMenu` renderiza um menu de seleção única, representado pelo elemento `select` da HTML, com o atributo `size` igual a 1 e sem o atributo `multiple`, que permitiria seleções múltiplas.

Todas as tags de seleção, com exceção da `h:selectBooleanCheckbox` (que estudaremos logo mais) devem receber a relação de itens que aparecerão na lista. Para especificar os itens, usamos a tag fundamental `f:selectItem`.

```
<h:outputLabel value="Time de futebol favorito: "
      for="timeFutebol"/>
<h:selectOneMenu value="#{cadastroTorcedorBean.timeFavorito}"
            id="timeFutebol" >
      <f:selectItem itemValue="Corinthians"/>
      <f:selectItem itemValue="Flamengo"/>
      <f:selectItem itemValue="Palmeiras"/>
      <f:selectItem itemValue="Santos"/>
      <f:selectItem itemValue="São Paulo"/>
      <f:selectItem itemValue="Vasco"/>
      <f:selectItem itemValue="Outro"/>
</h:selectOneMenu>
```

O código acima renderiza a seguinte saída em HTML:

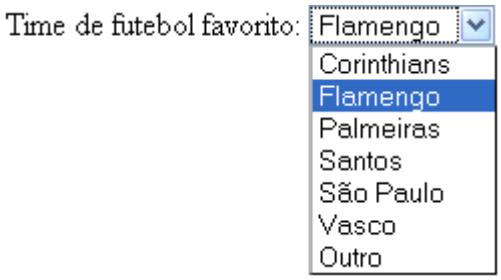
```
<label for="frm:timeFutebol">Time de futebol favorito: </label>
<select id="frm:timeFutebol" name="frm:timeFutebol" size="1">
  <option value="Corinthians">Corinthians</option>
  <option value="Flamengo">Flamengo</option>
  <option value="Palmeiras">Palmeiras</option>
  <option value="Santos">Santos</option>
  <option value="S&atilde;o Paulo">S&atilde;o Paulo</option>
  <option value="Vasco">Vasco</option>
  <option value="Outro">Outro</option>
</select>
```

O valor especificado no atributo `itemValue` da tag `f:selectItem` do item selecionado é passado para o atributo correspondente do *managed bean*, por isso, precisamos criar este atributo com seus *getters* e *setters*.

```
private String timeFavorito;

public String getTimeFavorito() {
      return timeFavorito;
```

```
}
public void setTimeFavorito(String timeFavorito) {
      this.timeFavorito = timeFavorito;
}
```

O resultado no browser é o seguinte:



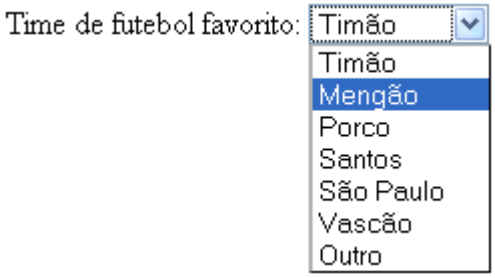
Os valores especificados na tag `f:selectItem` são usados também como rótulos dos itens do menu. Algumas vezes temos necessidade de ter os valores diferentes dos rótulos, e por isso a tag `f:selectItem` possui o atributo `itemLabel`. Veja um exemplo:

```
<h:selectOneMenu value="#{cadastroTorcedorBean.timeFavorito}"
            id="timeFutebol" >
      <f:selectItem itemValue="Corinthians" itemLabel="Timão"/>
      <f:selectItem itemValue="Flamengo" itemLabel="Mengão"/>
      <f:selectItem itemValue="Palmeiras" itemLabel="Porco"/>
      <f:selectItem itemValue="Santos"/>
      <f:selectItem itemValue="São Paulo"/>
      <f:selectItem itemValue="Vasco" itemLabel="Vascão"/>
      <f:selectItem itemValue="Outro"/>
</h:selectOneMenu>
```

Este código será renderizado como:

```
<label for="frm:timeFutebol">Time de futebol favorito:</label>
<select id="frm:timeFutebol" name="frm:timeFutebol" size="1">
      <option value="Corinthians">Tim&atilde;o</option>
      <option value="Flamengo">Meng&atilde;o</option>
      <option value="Palmeiras">Porco</option>
      <option value="Santos">Santos</option>
      <option value="S&atilde;o Paulo">S&atilde;o Paulo</option>
      <option value="Vasco">Vasc&atilde;o</option>
      <option value="Outro">Outro</option>
</select>
```

O resultado será:



O componente `h:selectManyMenu`

O componente `h:selectManyMenu` renderiza um menu de seleção múltipla, representado pelo elemento `select` da HTML, com o atributo `size` igual a 1 e com o atributo `multiple` igual a `true`.

```
<h:outputLabel value="Time de futebol favorito: "
      for="timeFutebol"/>
<h:selectManyMenu value="#{cadastroTorcedorBean.timesFavoritos}"
            id="timeFutebol" >
      <f:selectItem itemValue="Corinthians"/>
      <f:selectItem itemValue="Flamengo"/>
      <f:selectItem itemValue="Palmeiras"/>
      <f:selectItem itemValue="Santos"/>
      <f:selectItem itemValue="São Paulo"/>
      <f:selectItem itemValue="Vasco"/>
      <f:selectItem itemValue="Outro"/>
</h:selectManyMenu>
```

O código acima renderiza a seguinte saída em HTML:

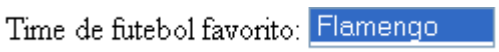
```
<label for="frm:timeFutebol">Time de futebol favorito: </label>
<select id="frm:timeFutebol" name="frm:timeFutebol"
         multiple="multiple" size="1">
      <option value="Corinthians">Corinthians</option>
      <option value="Flamengo">Flamengo</option>
      <option value="Palmeiras">Palmeiras</option>
      <option value="Santos">Santos</option>
      <option value="S&atilde;o Paulo">S&atilde;o Paulo</option>
      <option value="Vasco">Vasco</option>
      <option value="Outro">Outro</option>
</select>
```

Como este componente possibilita a seleção de mais de um item da seleção, o *managed bean* deve possuir um atributo de array para comportar os elementos selecionados. No caso deste exemplo, precisamos criar um atributo do tipo String (array) com o nome `timesFavoritos`.

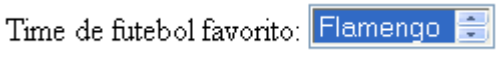
```
private String[] timesFavoritos;

public String[] getTimesFavoritos() {
      return timesFavoritos;
}
public void setTimesFavoritos(String[] timesFavoritos) {
      this.timesFavoritos = timesFavoritos;
}
```

Este componente não é apresentado corretamente por vários navegadores. O Internet Explorer exibe um menu com setas, porém o Firefox não. Apesar de existir, este é um componente que certamente você não deve utilizar.

Componente renderizado no Firefox:



Componente renderizado no Internet Explorer:



O componente `h:selectOneListbox`

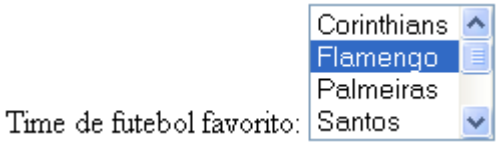
Este componente renderiza um elemento HTML `select` de qualquer tamanho (especificado) e sem o atributo `multiple`, ou seja, é possível selecionar apenas um item da seleção.

```
<h:outputLabel value="Time de futebol favorito: "
      for="timeFutebol"/>
<h:selectOneListbox value="#{cadastroTorcedorBean.timeFavorito}"
            id="timeFutebol" size="4">
      <f:selectItem itemValue="Corinthians"/>
      <f:selectItem itemValue="Flamengo"/>
      <f:selectItem itemValue="Palmeiras"/>
      <f:selectItem itemValue="Santos"/>
      <f:selectItem itemValue="São Paulo"/>
      <f:selectItem itemValue="Vasco"/>
      <f:selectItem itemValue="Outro"/>
</h:selectOneListbox>
```

O código acima gera o seguinte fragmento HTML:

```
<label for="frm:timeFutebol">Time de futebol favorito: </label>
<select id="frm:timeFutebol" name="frm:timeFutebol" size="4">
      <option value="Corinthians">Corinthians</option>
      <option value="Flamengo">Flamengo</option>
      <option value="Palmeiras">Palmeiras</option>
      <option value="Santos">Santos</option>
      <option value="S&atilde;o Paulo">S&atilde;o Paulo</option>
      <option value="Vasco">Vasco</option>
      <option value="Outro">Outro</option>
</select>
```

O valor 4 para o atributo `size` especifica o tamanho da lista, que deve possuir apenas 4 elementos visíveis.



O componente `h:selectManyListbox`

Este componente renderiza um elemento HTML `select` de qualquer tamanho (especificado) e com o atributo `multiple`, ou seja, é possível selecionar vários itens da seleção.

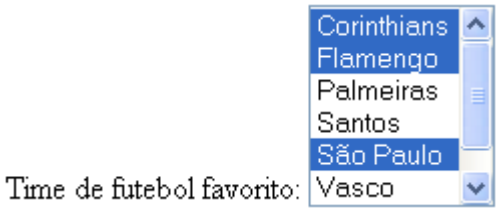
```
<h:outputLabel value="Time de futebol favorito: "
      for="timeFutebol"/>
<h:selectManyListbox id="timeFutebol" size="4"
            value="#{cadastroTorcedorBean.timesFavoritos}">
      <f:selectItem itemValue="Corinthians"/>
      <f:selectItem itemValue="Flamengo"/>
      <f:selectItem itemValue="Palmeiras"/>
      <f:selectItem itemValue="Santos"/>
      <f:selectItem itemValue="São Paulo"/>
      <f:selectItem itemValue="Vasco"/>
      <f:selectItem itemValue="Outro"/>
```

```
</h:selectManyListbox>
```

O código acima gera o seguinte fragmento HTML:

```
<label for="frm:timeFutebol">Time de futebol favorito: </label>
<select id="frm:timeFutebol" name="frm:timeFutebol"
            multiple="multiple" size="6">
      <option value="Corinthians">Corinthians</option>
      <option value="Flamengo">Flamengo</option>
      <option value="Palmeiras">Palmeiras</option>
      <option value="Santos">Santos</option>
      <option value="S&atilde;o Paulo">S&atilde;o Paulo</option>
      <option value="Vasco">Vasco</option>
      <option value="Outro">Outro</option>
</select>
```

O usuário pode selecionar nenhum, um ou vários itens da seleção. Lembre-se de criar o atributo como um array no *managed bean*, pois o componente possibilita seleção múltipla dos itens.



9.8. Campos de checagem e botões rádio

Existem 3 componentes que renderizam elementos de campos de checagem e botões rádio: `h:selectOneRadio`, `h:selectBooleanCheckbox` e `h:selectManyCheckbox`.

O componente `h:selectOneRadio`

Este componente renderiza um elemento HTML `input` do tipo `radio`, usado quando você deseja exibir uma lista de opções que podem ser selecionadas unicamente.

```
<h:selectOneRadio id="sexo">
      <f:selectItem itemValue="M" itemLabel="Masculino"/>
      <f:selectItem itemValue="F" itemLabel="Feminino"/>
</h:selectOneRadio>
```

O seguinte código HTML é renderizado:

```
<table id="frm:sexo">
      <tr>
            <td>
                  <input type="radio" name="frm:sexo"
                        id="frm:sexo:0" value="M" />
                  <label for="frm:sexo:0">Masculino</label>
            </td>
            <td>
                  <input type="radio" name="frm:sexo"
                        id="frm:sexo:1" value="F" />
                  <label for="frm:sexo:1"> Feminino</label>
            </td>
      </tr>
```

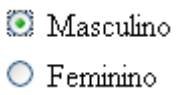
```
</table>
```

Você deve ter percebido que os elementos `input` são gerados dentro de colunas de uma tabela da HTML. Os elementos são exibidos horizontalmente.

É possível alterar o layout no qual os itens são apresentados, especificando o atributo `layout`. Os valores possíveis para este atributo são `pageDirection` e `lineDirection` (padrão). Veja um exemplo com o layout `pageDirection`.

```
<h:selectOneRadio id="sexo" layout="pageDirection">
      <f:selectItem itemValue="M" itemLabel="Masculino"/>
      <f:selectItem itemValue="F" itemLabel="Feminino"/>
</h:selectOneRadio>
```

O layout `pageDirection` exibe os itens na vertical:



O componente `h:selectBooleanCheckbox`

`O componente h:selectBooleanCheckbox renderiza o elemento da HTML input do tipo checkbox`. Use este componente para definir opções boleanas, onde o usuário pode responder alguma pergunta em seu formulário com "sim" ou "não".

```
<h:selectBooleanCheckbox id="aceite"
      value="#{cadastroUsuarioBean.termoAceito}"/>
<h:outputLabel value="Li e aceito os termos e condições"
      for="aceite"/>
```

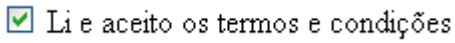
O exemplo acima renderiza o seguinte trecho em HTML:

```
<input id="frm:aceite" type="checkbox" name="frm:aceite"
      checked="checked" />
<label for="frm:aceite">Li e aceito os termos e
condi&ccedil;&otilde;es</label>
```

No *managed bean*, você deve criar um atributo boleano para receber a opção do usuário. Se o campo for selecionado, o atributo receberá o valor `true`, caso contrário, receberá o valor `false`.

```
private boolean termoAceito;

public boolean isTermoAceito() {
      return termoAceito;
}

public void setTermoAceito(boolean termoAceito) {
      this.termoAceito = termoAceito;
}
```

Veja como fica o resultado no browser do usuário.



O componente `h:selectManyCheckbox`

Este componente renderiza uma lista de campos de checagem (elemento `input` do tipo `checkbox`). O usuário pode selecionar nenhum, um ou vários itens.

```
<h:selectManyCheckbox id="assuntos"
        value="#{cadastroUsuarioBean.assuntosInteresse}">
    <f:selectItem itemValue="1" itemLabel="Java"/>
    <f:selectItem itemValue="2" itemLabel=".NET"/>
    <f:selectItem itemValue="3" itemLabel="UML"/>
    <f:selectItem itemValue="4" itemLabel="Arquitetura"/>
    <f:selectItem itemValue="5" itemLabel="Análise"/>
    <f:selectItem itemValue="6" itemLabel="Gestão de Projetos"/>
</h:selectManyCheckbox>
```

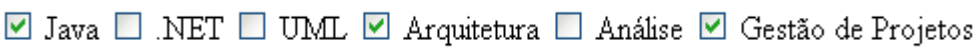
O fragmento de código acima gera o seguinte HTML:

```
<table id="frm:assuntos">
    <tr>
        <td>
            <input name="frm:assuntos" id="frm:assuntos:0"
                value="1" type="checkbox" />
            <label for="frm:assuntos:0"> Java</label>
        </td>
        <td>
            <input name="frm:assuntos" id="frm:assuntos:1"
                value="2" type="checkbox" />
            <label for="frm:assuntos:1"> .NET</label>
        </td>
        <td>
            <input name="frm:assuntos" id="frm:assuntos:2"
                value="3" type="checkbox" />
            <label for="frm:assuntos:2"> UML</label>
        </td>
        <td>
            <input name="frm:assuntos" id="frm:assuntos:3"
                value="4" type="checkbox"/>
            <label for="frm:assuntos:3"> Arquitetura</label>
        </td>
        <td>
            <input name="frm:assuntos" id="frm:assuntos:4"
                value="5" type="checkbox" />
            <label for="frm:assuntos:4"> An&aacute;lise</label>
        </td>
        <td>
            <input name="frm:assuntos" id="frm:assuntos:5"
                value="6" type="checkbox" />
            <label for="frm:assuntos:5"> Gest&atilde;o de
            Projetos</label>
        </td>
    </tr>
</table>
```

Os valores dos itens da seleção são numéricos e o usuário pode selecionar múltiplos elementos da lista, por isso criamos um atributo no *managed bean* que é um array de inteiros.

```
private Integer[] assuntosInteresse;

public Integer[] getAssuntosInteresse() {
	return assuntosInteresse;
}

public void setAssuntosInteresse(Integer[] assuntosInteresse) {
	this.assuntosInteresse = assuntosInteresse;
}
```

Veja agora o resultado no navegador do usuário:



É possível alterar o layout no qual os itens são apresentados, especificando o atributo `layout`. Os valores possíveis para este atributo são `pageDirection` e `lineDirection` (padrão). Veja um exemplo com o layout `pageDirection`.

```
<h:selectManyCheckbox id="assuntos" layout="pageDirection"
		value="#{cadastroUsuarioBean.assuntosInteresse}">
	<f:selectItem itemValue="1" itemLabel="Java"/>
	<f:selectItem itemValue="2" itemLabel=".NET"/>
	<f:selectItem itemValue="3" itemLabel="UML"/>
	<f:selectItem itemValue="4" itemLabel="Arquitetura"/>
	<f:selectItem itemValue="5" itemLabel="Análise"/>
	<f:selectItem itemValue="6" itemLabel="Gestão de Projetos"/>
</h:selectManyCheckbox>
```

O layout `pageDirection` exibe os itens na vertical:

9.9. Itens de seleção

Existem situações que precisamos que uma lista de itens seja obtida dinamicamente, para disponibilizar ao usuário a seleção em qualquer um dos componentes que vimos nos últimos tópicos. Esta lista pode vir de um banco de dados, de um arquivo ou de qualquer outra origem. Quando existe essa necessidade, precisamos da tag fundamental `f:selectItems` (no plural).

Neste exemplo, criaremos uma lista de opções com nomes de cidades. Na página JSF, incluímos o componente `h:selectOneListbox`.

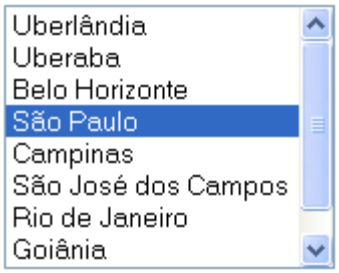
```
<h:selectOneListbox size="8"
	value="#{cadastroUsuarioBean.cidadeNatal}">
	<f:selectItems value="#{cadastroUsuarioBean.cidades}"/>
</h:selectOneListbox>
```

Especificamos o atributo `value` da tag `f:selectItems` referenciando a propriedade `cidades` do *managed bean*. A propriedade `cidades` é do tipo `List<SelectItem>`, pois representa uma lista de itens de seleção. Veja como ficou a declaração dos atributos do *managed bean*, seus *getters* e *setters* e a inicialização da lista de itens no construtor.

```
private Integer cidadeNatal;
private List<SelectItem> cidades;

public CadastroUsuarioBean() {
	this.cidades = new ArrayList<SelectItem>();

	//poderia buscar as cidades do banco de dados
	this.cidades.add(new SelectItem(1, "Uberlândia"));
	this.cidades.add(new SelectItem(2, "Uberaba"));
	this.cidades.add(new SelectItem(3, "Belo Horizonte"));
	this.cidades.add(new SelectItem(4, "São Paulo"));
	this.cidades.add(new SelectItem(5, "Campinas"));
	this.cidades.add(new SelectItem(6, "São José dos Campos"));
	this.cidades.add(new SelectItem(7, "Rio de Janeiro"));
	this.cidades.add(new SelectItem(8, "Goiânia"));
	this.cidades.add(new SelectItem(9, "Porto Alegre"));
}

public Integer getCidadeNatal() {
	return cidadeNatal;
}

public void setCidadeNatal(Integer cidadeNatal) {
	this.cidadeNatal = cidadeNatal;
}

public List<SelectItem> getCidades() {
	return cidades;
}

public void setCidades(List<SelectItem> cidades) {
	this.cidades = cidades;
}
```

O construtor da classe `SelectItem` recebe como parâmetro o valor e o rótulo do item, por isso informamos um código da cidade no primeiro parâmetro e o nome da cidade no segundo.

O resultado no browser é uma listagem com os itens gerados a partir da classe Java que representa o *managed bean*.



9.10. Botões e links

Qualquer aplicação web precisa de botões e links para receber as ações do usuário. Aprenderemos agora como trabalhar com eles em JSF, através dos componentes `h:commandButton`, `h:commandLink` e `h:outputLink`.

O componente `h:commandButton`

Este componente gera um elemento input da HTML do tipo `submit`, `image` ou `reset`. O atributo `type` pode ser especificado com `submit` (padrão) ou `reset`. Para botões representados por imagens, basta informar o caminho no atributo `image`, que o tipo do elemento é alterado automaticamente para `image`.

Nos botões de comando, os atributos `action` ou `actionListener` podem ser especificados para ações de navegação ou referência a métodos de *managed beans*. Já usamos e continuaremos usando estes atributos, porém vamos estudá-los melhor em outro capítulo.

```
<h:commandButton id="cadastrar" value="Cadastrar" type="submit"
      actionListener="#{cadastroUsuarioBean.cadastrar}"/>
<h:commandButton id="limpar" value="Limpar" type="reset"/>
<h:commandButton id="voltar" value="Voltar"
      image="/imagens/voltar.gif" action="cancelar"/>
```

O código acima renderiza o seguinte HTML:

```
<input type="submit" name="frm:cadastrar"
      value="Cadastrar" />
<input type="reset" name="frm:limpar"
      value="Limpar" />
<input type="image" src="/Agenda/imagens/voltar.gif"
      name="frm:voltar" />
```

E o resultado visual é o seguinte:

O componente `h:commandLink`

O componente `h:commandLink` gera uma âncora em HTML (link) que funciona como um botão de submissão de formulário. Veja alguns exemplos de uso.

```
<h:commandLink id="cadastrar" value="Cadastrar"
      actionListener="#{cadastroUsuarioBean.cadastrar}"/>
  
<h:commandLink id="voltar" action="cancelar">
      <h:graphicImage value="/imagens/voltar.gif"
            style="border: none;"/>
      <h:outputText value="Voltar"/>
</h:commandLink>
```

Para os links reagirem como um botão, o framework JSF gera vários códigos em JavaScript. Veja a saída em HTML de nosso último exemplo.

```
<script type="text/javascript" language="Javascript">
```

```
      //vários códigos JavaScript gerados pelo framework JSF
</script>
<a id="frm:cadastrar" href="#" onclick="if(typeof jsfcljs ==
'function'){jsfcljs(document.getElementById('frm'),{'frm:cadastr
ar':'frm:cadastrar'},'');}return false">Cadastrar</a>
  
<a id="frm:voltar" href="#" onclick="if(typeof jsfcljs ==
'function'){jsfcljs(document.getElementById('frm'),{'frm:voltar'
:'frm:voltar'},'');}return false">
<img src="/Agenda/imagens/voltar.gif"
      style="border: none;" />Voltar</a>
```

A execução do último exemplo exibirá:

O componente `h:outputLink`

Os componentes `h:commandButton` e `h:commandLink` submetem requisições ao servidor. Em determinadas situações, você pode precisar simplesmente de um link comum da HTML. Para este caso, existe o componente `h:outputLink`.

```
<h:outputLink value="http://www.algaworks.com" target="_blank">
      <h:graphicImage value="/imagens/logo_algaworks.gif"
            style="border: none;"/>
</h:outputLink>
<br/><br/>
<h:outputLink value="http://twitter.com/algaworks">
      <h:outputText value="Twitter da AlgaWorks"/>
</h:outputLink>
```

O primeiro exemplo é um link em uma imagem para o site da AlgaWorks e o segundo um link em um texto para o Twitter da AlgaWorks. Veja a saída em HTML:

```
<a href="http://www.algaworks.com" target="_blank">
      <img src="/Agenda/imagens/logo_algaworks.gif"
            style="border: none;" /></a>
<br/><br/>
<a href="http://twitter.com/algaworks">Twitter da
AlgaWorks</a>
```

No browser, você verá:

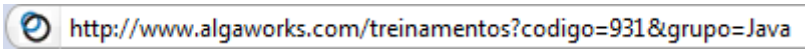
Twitter da AlgaWorks

O componente `h:outputLink` facilita a construção de URLs com parâmetros (querystring). Os parâmetros são especificados usando a tag fundamental `f:param`. Veja um exemplo:

```
<h:outputLink target="_blank"
      value="http://www.algaworks.com/treinamentos">
      <f:param name="codigo" value="931"/>
```

```
        <f:param name="grupo" value="Java"/>
        <h:outputText value="Treinamentos da AlgaWorks"/>
    </h:outputLink>
```

Ao clicar no link gerado pelo código acima, somos redirecionados para o endereço especificado, incluindo os parâmetros “codigo” e “grupo”.

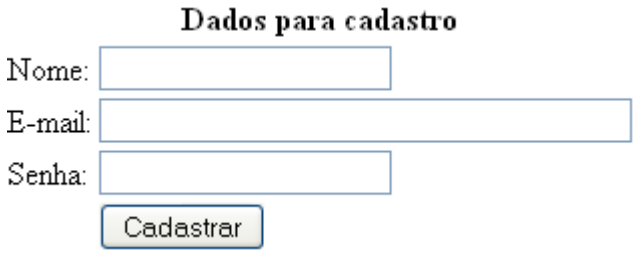


9.11. Painéis

O framework JSF possui componentes que te ajudam a organizar seus layouts. O que você faria para criar um formulário com rótulos e campos, com um bom alinhamento? Sem aprender os componentes JSF para auxílio na construção de layouts, podemos escrever códigos HTML manualmente, como o exemplo abaixo:

```
<table>
      <thead>
            <tr>
                  <th colspan="2">
                     <h:outputText value="Dados para cadastro"/>
                  </th>
            </tr>
      </thead>
      <tr>
            <td><h:outputText value="Nome:"/></td>
            <td>
               <h:inputText value="#{cadastroUsuarioBean.nome}"/>
            </td>
      </tr>
      <tr>
            <td><h:outputText value="E-mail:"/></td>
            <td>
              <h:inputText value="#{cadastroUsuarioBean.email}"
                  size="40" maxlength="250"/>
            </td>
      </tr>
      <tr>
            <td><h:outputText value="Senha:"/></td>
            <td>
              <h:inputSecret value="#{cadastroUsuarioBean.senha}"
                  size="20"/></td>
      </tr>
      <tr>
          <td></td>
          <td colspan="2">
            <h:commandButton id="cadastrar"
              value="Cadastrar
              actionListener="#{cadastroUsuarioBean.cadastrar}"/>
          </td>
      </tr>
</table>
```

Conseguimos fazer o que queríamos. Veja o que o usuário visualizará no navegador.



Apesar de ter funcionado, escrever códigos extensos em HTML é tedioso, por isso aprenderemos sobre o componente `h:panelGrid`, que pode nos auxiliar nisso.

O componente `h:panelGrid`

O componente `h:panelGrid` renderiza uma tabela da HTML. Ele funciona como um organizador de itens no layout da página, e pode acomodar qualquer componente JSF.

O último exemplo poderia ser escrito da seguinte forma usando o componente `h:panelGrid`.

```
<h:panelGrid columns="2">
	<f:facet name="header">
		<h:outputText value="Dados para cadastro"/>
	</f:facet>
	<h:outputText value="Nome:"/>
	<h:inputText value="#{cadastroUsuarioBean.nome}"/>
	<h:outputText value="E-mail:"/>
	<h:inputText value="#{cadastroUsuarioBean.email}"
		size="40" maxlength="250"/>
	<h:outputText value="Senha:"/>
	<h:inputSecret value="#{cadastroUsuarioBean.senha}"
		size="20"/>
	<h:outputText/>
	<h:commandButton id="cadastrar" value="Cadastrar"
		actionListener="#{cadastroUsuarioBean.cadastrar}"/>
</h:panelGrid>
```

O resultado final é o mesmo, porém não precisa nem dizer o quanto é mais fácil usar o componente `h:panelGrid`, certo?

O atributo `columns` é opcional, mas se você não especificar, o número de colunas será igual a 1. O renderizador do `h:panelGrid` organiza os componentes "filhos" em colunas, da esquerda para a direita e de cima para baixo. Cada componente que estiver dentro do `h:panelGrid` será acomodado em uma célula da tabela. Quando precisamos pular uma célula, podemos incluir um componente sem valor algum ou que não gera informações visuais, como foi o caso de `<h:outputText/>`.

É possível alterar a espessura da borda, o espaçamento entre células, cores e várias outras coisas, da mesma forma que você também pode fazer com as tabelas HTML.

Você pode também especificar o cabeçalho e rodapé da tabela usando a tag `f:facet`. Esta tag define um fragmento de código que faz algum sentido para o componente. Por exemplo, para definir o cabeçalho do `h:panelGrid`, o nome do *facet* deve ser "header", e o rodapé "footer". Estes nomes estão programados dentro do componente `h:panelGrid`, e por isso não podem ser mudados.

O componente `h:panelGroup`

O componente h:panelGroup é um container que renderiza seus "filhos". Este componente deve ser usado especialmente em situações onde é desejado a inclusão de vários

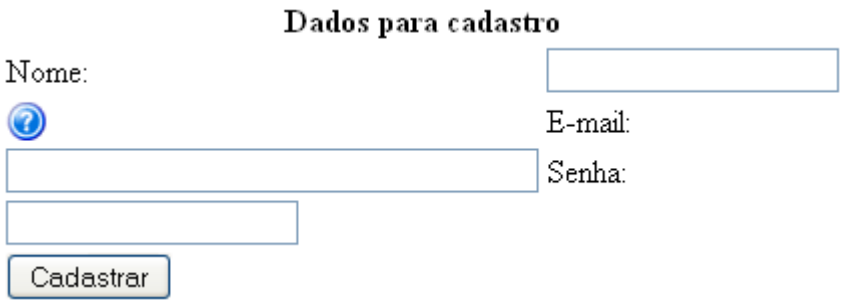
componentes em um lugar onde apenas um é permitido, por exemplo, como uma única célula do h:panelGrid.

Vamos demonstrar como faríamos para incluir uma imagem ao lado de um campo de entrada de texto usando h:panelGrid para organizar os componentes. Antes, vamos tentar fazer isso sem o auxílio do h:panelGroup.

Queremos colocar um ícone de ajuda ao lado do campo de nome do usuário.

```
<h:panelGrid columns="2">
	<f:facet name="header">
		<h:outputText value="Dados para cadastro"/>
	</f:facet>
	<h:outputText value="Nome:"/>
	<h:inputText value="#{cadastroUsuarioBean.nome}"/>
	<h:graphicImage value="/imagens/ajuda.gif"/>
	<h:outputText value="E-mail:"/>
	<h:inputText value="#{cadastroUsuarioBean.email}"
		size="40" maxlength="250"/>
	<h:outputText value="Senha:"/>
	<h:inputSecret value="#{cadastroUsuarioBean.senha}"
		size="20"/>
	<h:outputText/>
	<h:commandButton id="cadastrar" value="Cadastrar"
		actionListener="#{cadastroUsuarioBean.cadastrar}"/>
</h:panelGrid>
```

Além de não funcionar, veja que o formulário ficou totalmente desconfigurado. O ícone de ajuda tomou lugar de uma célula, que “empurrou” todas as outras para frente.

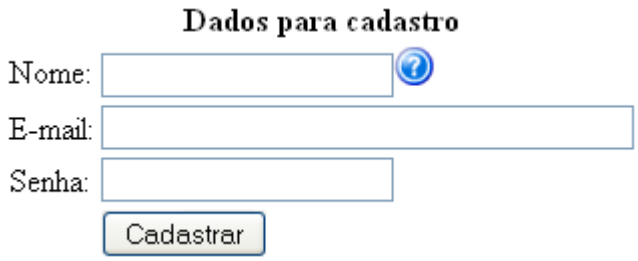


Eis o componente `h:panelGroup`, que agrupa todos os seus filhos em um só lugar.

```
<h:panelGrid columns="2">
	<f:facet name="header">
		<h:outputText value="Dados para cadastro"/>
	</f:facet>
	<h:outputText value="Nome:"/>
	<h:inputText value="#{cadastroUsuarioBean.nome}"/>
	<h:panelGroup>
		<h:inputText value="#{cadastroUsuarioBean.nome}"/>
		<h:graphicImage value="/imagens/ajuda.gif"/>
	</h:panelGroup>
	<h:outputText value="E-mail:"/>
	<h:inputText value="#{cadastroUsuarioBean.email}"
		size="40" maxlength="250"/>
	<h:outputText value="Senha:"/>
	<h:inputSecret value="#{cadastroUsuarioBean.senha}"
		size="20"/>
	<h:outputText/>
	<h:commandButton id="cadastrar" value="Cadastrar"
		actionListener="#{cadastroUsuarioBean.cadastrar}"/>
```

```
</h:panelGrid>
```

Agora o resultado ficou como esperamos, com o campo do nome e o ícone de ajuda lado-a-lado.



9.12. Mensagens

Qualquer aplicação precisa tratar entradas do usuário, processá-las e exibir mensagens de sucesso, erros, advertências e etc. Estudaremos em outro capítulo sobre validação e conversão de dados, mas já podemos trabalhar com todos os tipos mensagens, pois os *managed beans* possuem o "poder" de adicionar mensagens a uma fila, que em algum momento pode ser exibida na tela.

Vamos alterar nosso último exemplo para adicionarmos restrições e mensagens no método `cadastrar()`. Se o usuário informar um nome menor que 10 caracteres, entenderemos que ele não foi informado completo, e por isso adicionaremos uma mensagem de erro solicitando a correção ao usuário. Se o usuário tentar chamar o botão "Cadastrar" no domingo (isso mesmo, se tentar usar o sistema no dia que era para estar descansando), também não aceitaremos. Veja como fica o método `cadastrar()` no *managed bean*.

```
public void cadastrar(ActionEvent event) {
	FacesContext context = FacesContext.getCurrentInstance();
	if (this.getNome() != null
		&& this.getNome().length() < 10) {
		context.addMessage("frm:nome",
			new FacesMessage(FacesMessage.SEVERITY_ERROR,
				"Nome incompleto!",
				"Favor informar seu nome completo."));
	}
	if (Calendar.getInstance().get(Calendar.DAY_OF_WEEK)
		== Calendar.SUNDAY) {
		context.addMessage(null,
			new FacesMessage(FacesMessage.SEVERITY_WARN,
				"Dia da semana inválido!",
				"Você não pode cadastrar usuários no
				domingo."));
	}

	...

}
```

Para incluir mensagens na fila, precisamos antes de tudo pegar uma instância de `FacesContext`. Esta classe é responsável por manter as informações da requisição atual.

```
FacesContext context = FacesContext.getCurrentInstance();
```

Quando a validação do tamanho do nome falhar, precisamos incluir uma mensagem indicando que o nome completo deve ser informado.

```
context.addMessage("frm:nome",
      new FacesMessage(FacesMessage.SEVERITY_ERROR,
            "Nome incompleto!",
            "Favor informar seu nome completo."));
```

Veja acima que chamamos o método `addMessage()` de `FacesContext`. O primeiro parâmetro deve ser informado com o ID gerado para o elemento HTML para o qual queremos anexar esta mensagem. Como neste caso a mensagem é referente ao campo de nome, informamos "frm:nome", que é o ID gerado na saída HTML. Este parâmetro poderia ficar nulo se a mensagem não fosse direcionada a nenhum componente.

O segundo parâmetro recebe uma instância de `FacesMessage`, que representa uma mensagem JSF. O primeiro parâmetro do construtor de `FacesMessage` recebe a severidade da mensagem, que pode ser `SEVERITY_FATAL` (fatal), `SEVERITY_ERROR` (erro), `SEVERITY_WARN` (advertência) e `SEVERITY_INFO` (informação). O segundo parâmetro recebe uma descrição resumida da mensagem e o terceiro uma descrição detalhada.

Fizemos a mesma coisa para caso o dia da semana for domingo, porém não informamos o ID de nenhum componente, pois esta validação não é referente a nenhum campo da tela.

```
context.addMessage(null,
      new FacesMessage(FacesMessage.SEVERITY_WARN,
            "Dia da semana inválido!",
            "Você não pode cadastrar usuários no domingo."));
```

Como programamos a inclusão de mensagens em uma fila do JSF, naturalmente iremos querer exibí-las na tela. Os componentes responsáveis por exibir mensagens são `h:messages` e `h:message`, que iremos estudar agora.

O componente `h:messages` exibe todas as mensagens adicionadas na fila, e `h:message` exibe uma única mensagem para um componente especificado no atributo `for`. Vejamos um exemplo.

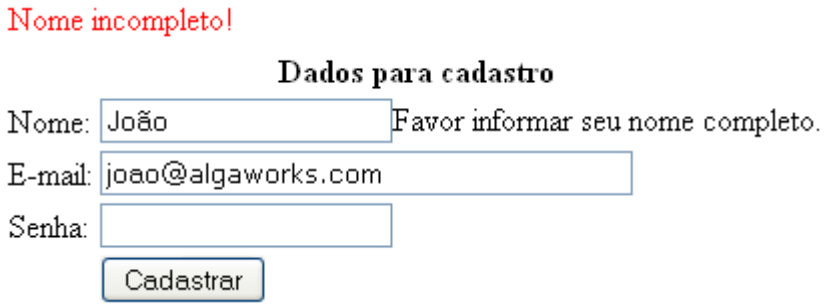
```
<h:messages layout="table" errorStyle="color: red"
      infoStyle="color: green" warnStyle="color: orange"
      fatalStyle="color: gray"/>

<h:panelGrid columns="2" >
      <f:facet name="header">
            <h:outputText value="Dados para cadastro"/>
      </f:facet>
      <h:outputText value="Nome:"/>
      <h:panelGroup>
            <h:inputText id="nome"
                  value="#{cadastroUsuarioBean.nome}"/>
            <h:message for="nome"/>
      </h:panelGroup>
      <h:outputText id="email" value="E-mail:"/>
      <h:panelGroup>
            <h:inputText size="40" maxlength="250"
                  value="#{cadastroUsuarioBean.email}"/>
            <h:message for="email"/>
      </h:panelGroup>
      <h:outputText value="Senha:"/>
      <h:panelGroup>
            <h:inputSecret id="senha" size="20"
                  value="#{cadastroUsuarioBean.senha}"/>
            <h:message for="senha"/>
      </h:panelGroup>
```

```
        <h:outputText/>
        <h:commandButton id="cadastrar" value="Cadastrar"
              actionListener="#{cadastroUsuarioBean.cadastrar}"/>
    </h:panelGrid>
```

Informamos cores de fonte diferentes para cada tipo de mensagem através dos atributos `errorStyle`, `infoStyle`, `warnStyle` e `fatalStyle` do componente `h:messages`. O atributo `layout` pode ser informado com os valores "table" ou "list" (padrão), sendo que o primeiro exibe as mensagens uma de baixo da outra, e o segundo as exibe uma na frente da outra.

O componente `h:message` foi usado sempre após um campo para exibir mensagens de erro referente ao mesmo campo. Veja que o atributo `for` é especificado com o ID do componente de entrada de dados.

Ao executarmos a tela, informarmos um nome incompleto, ao e clicar no botão "Cadastrar", veja o que acontece:

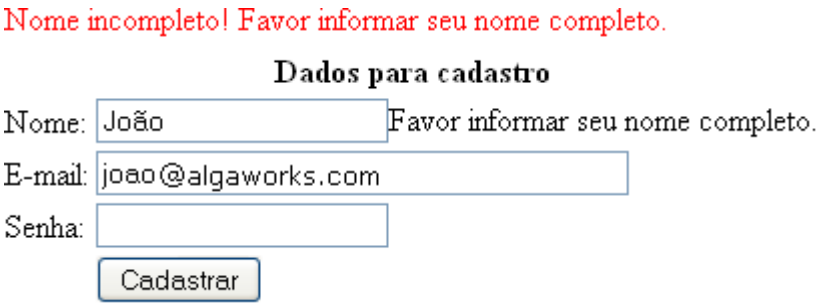


Perceba que o resumo da mensagem apareceu no topo, onde usamos `h:messages`, e a mensagem detalhada apareceu ao lado do campo, onde usamos `h:message`.

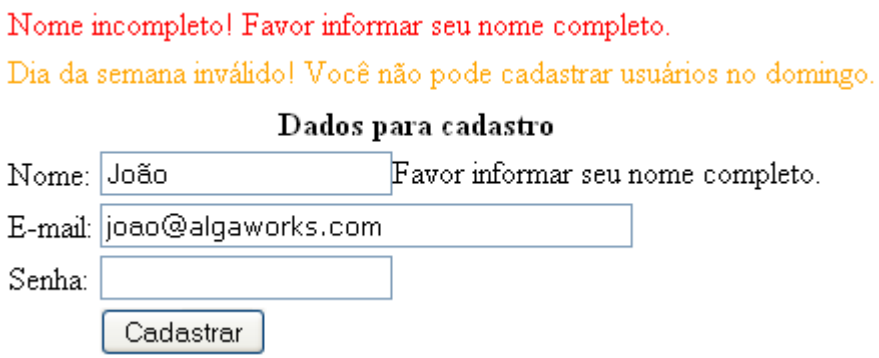
Se quisermos exibir a mensagem detalhada no topo, incluímos o atributo `showDetail` com o valor igual a "true".

```
<h:messages layout="table" errorStyle="color: red"
      infoStyle="color: green" warnStyle="color: orange"
      fatalStyle="color: gray" showDetail="true"/>
```

O resumo e a mensagem detalhada são exibidos agora.



No domingo não podemos usar este sistema, lembra? Veja o que acontece quando insistimos em usá-lo neste dia.

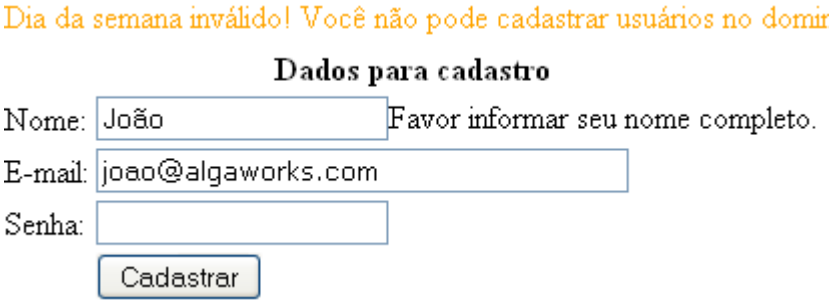


As duas mensagens foram exibidas, porém a mensagem da restrição do dia da semana apareceu apenas no topo, pois não referenciamos nenhum componente com `h:message`.

E se quiséssemos exibir no topo apenas as mensagens que não fazem referência a outros componentes, mas continuar exibindo as mensagens ao lado dos campos, para deixar a tela menos "poluída" de erros? Simplesmente colocamos o atributo `globalOnly` igual a "true" na tag `h:messages`.

```
<h:messages layout="table" errorStyle="color: red"
      infoStyle="color: green" warnStyle="color: orange"
      fatalStyle="color: gray" showDetail="true"
      globalOnly="true"/>
```

Agora apenas a mensagem da restrição do dia da semana apareceu no topo, mas a mensagem de validação do tamanho do nome continua aparecendo ao lado do campo.



10. Tabela de dados

10.1. O componente `h:dataTable`

O componente `h:dataTable` gera uma tabela HTML que pode ser associada a um *managed bean* para obter dados dinamicamente.

Nosso exemplo será de uma tabela de dados de clientes. Para isso, precisamos antes preparar o projeto com algumas classes e configurações. Primeiramente, criamos um JavaBean `Cliente` para armazenar os dados de clientes.

```
public class Cliente {

	private Integer codigo;
	private String nome;
	private String cidade;

	public Cliente(Integer codigo, String nome,
			String cidade) {
		super();
		this.codigo = codigo;
		this.nome = nome;
		this.cidade = cidade;
	}

	public Integer getCodigo() {
		return codigo;
	}
	public void setCodigo(Integer codigo) {
		this.codigo = codigo;
	}
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public String getCidade() {
		return cidade;
	}

	public void setCidade(String cidade) {
		this.cidade = cidade;
	}

}
```

Agora criamos o *managed bean* `ConsultaClienteBean`. Esta classe tem um atributo chamado `clientes` para armazenar a lista de clientes que devem aparecer na tabela e um método `consultar()` que popula a lista de clientes com dados informados manualmente.

```
public class ConsultaClienteBean {

	private List<Cliente> clientes = new ArrayList<Cliente>();

	public void consultar(ActionEvent event) {
		this.getClientes().clear();
		this.getClientes().add(new Cliente(1,
			"João da Silva", "Uberlândia"));
		this.getClientes().add(new Cliente(2,
```

```
                "Manoel Souza", "Uberaba"));
        this.getClientes().add(new Cliente(3,
                "Cristina Melo", "São Paulo"));
        this.getClientes().add(new Cliente(6,
                "Sebastião Cardoso", "Belo Horizonte"));
        this.getClientes().add(new Cliente(7,
                "Francisco Borges", "Uberlândia"));
        this.getClientes().add(new Cliente(10,
                "Juliano Messias", "Rio de Janeiro"));
        this.getClientes().add(new Cliente(11,
                "Maria Helena", "Uberlândia"));
    }

    public List<Cliente> getClientes() {
        return this.clientes;
    }

}
```

Configuramos o *managed bean* no arquivo “faces-config.xml”.

```
<managed-bean>
    <managed-bean-name>consultaClienteBean</managed-bean-name>
    <managed-bean-class>
        com.algaworks.dwjsf.visao.ConsultaClienteBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Veja agora a forma de usar o componente `h:dataTable`.

```
<h:form id="frm">
    <h:commandButton value="Consultar"
        actionListener="#{consultaClienteBean.consultar}"/>

    <h:dataTable value="#{consultaClienteBean.clientes}"
        var="item" border="1"
        rendered="#{not empty consultaClienteBean.clientes}">
        <h:column>
            <h:outputText value="#{item.codigo}"/>
        </h:column>
        <h:column>
            <h:outputText value="#{item.nome}"/>
        </h:column>
        <h:column>
            <h:outputText value="#{item.cidade}"/>
        </h:column>
    </h:dataTable>
</h:form>
```

Quando o componente `h:dataTable` é executado, cada item da lista, referenciada através do atributo `value`, fica disponível dentro do corpo da tag. O nome de cada item da lista é definido com o atributo `var`, possibilitando o acesso dentro das colunas.

No corpo da tag `h:dataTable` definimos três colunas com `h:column`. Dentro de cada coluna, podemos incluir qualquer quantidade de componentes. No caso deste exemplo, inserimos apenas um `h:outputText` em cada coluna.

Ao executar a tela, apenas o botão aparecerá, pois dissemos para a tabela não ser renderizada caso a lista de clientes esteja vazia. Quando clicamos no botão “Consultar”, aparecem os dados em uma tabela simples.

10.2. Cabeçalhos e rodapés

As tabelas de dados suportam cabeçalhos e rodapés. Os cabeçalhos são úteis para, por exemplo, incluir as descrições das colunas. Os rodapés também são úteis para, por exemplo, exibir cálculos, totalizadores, etc.

A inclusão de cabeçalhos e rodapés é conseguido usando a tag `f:facet` com o nome "header" e "footer", para, respectivamente, cabeçalho e rodapé.

Alteramos nosso exemplo para incluir cabeçalhos nas colunas.

```
<h:dataTable value="#{consultaClienteBean.clientes}"
      var="item" border="1"
      rendered="#{not empty consultaClienteBean.clientes}">
      <h:column>
            <f:facet name="header">
               <h:panelGroup>
                  <h:outputText value="Código"/>
                  <h:graphicImage value="/imagens/ajuda.gif"/>
               </h:panelGroup>
            </f:facet>
            <h:outputText value="#{item.codigo}"/>
      </h:column>
      <h:column>
            <f:facet name="header">
                  <h:outputText value="Nome"/>
            </f:facet>
            <h:outputText value="#{item.nome}"/>
      </h:column>
      <h:column>
            <f:facet name="header">
                  <h:outputText value="Cidade"/>
            </f:facet>
            <h:outputText value="#{item.cidade}"/>
      </h:column>
</h:dataTable>
```

A tag `f:facet` suporta apenas um componente dentro dela. Se você tentar incluir mais de um, apenas o primeiro será considerado, e o restante ignorado. Quando há a necessidade de incluir mais de um componente no cabeçalho, como por exemplo, um texto e uma imagem, deve-se englobar estes componentes dentro de um container, como o `h:panelGroup`. Fizemos isso no código acima para a coluna "Código". Veja o resultado.

Consultar

Código	Nome	Cidade
1	João da Silva	Uberlândia
2	Manoel Souza	Uberaba
3	Cristina Melo	São Paulo
6	Sebastião Cardoso	Belo Horizonte
7	Francisco Borges	Uberlândia
10	Juliano Messias	Rio de Janeiro
11	Maria Helena	Uberlândia

10.3. Componentes dentro de células

Você não é obrigado a colocar apenas textos dentro das tabelas de dados! Se precisar, é possível usar qualquer componente JSF nas colunas de tabelas. Veja um exemplo que usa diversos componentes dentro das colunas.

```
<h:dataTable value="#{consultaClienteBean.clientes}"
      var="item" border="1"
      rendered="#{not empty consultaClienteBean.clientes}">
      <h:column>
         <f:facet name="header">
            <h:panelGroup>
                <h:outputText value="Código"/>
                <h:graphicImage value="/imagens/ajuda.gif"/>
            </h:panelGroup>
         </f:facet>
         <h:inputText value="#{item.codigo}" size="2"
                  maxlength="4"/>
      </h:column>
      <h:column>
         <f:facet name="header">
            <h:outputText value="Nome"/>
         </f:facet>
         <h:outputLink target="_blank"
            value="http://www.google.com.br/#hl=pt-BR&q=#{item.nome}">
            <h:outputText value="#{item.nome}"/>
         </h:outputLink>
      </h:column>
      <h:column>
         <f:facet name="header">
            <h:outputText value="Cidade"/>
         </f:facet>
         <h:outputText value="#{item.cidade} "/>
            <h:graphicImage value="/imagens/bandeira_uberlandia.gif"
                  rendered="#{item.cidade eq 'Uberlândia'}"/>
      </h:column>
</h:dataTable>
```

Neste exemplo, colocamos um campo de entrada na coluna “Código”, um link para a pesquisa no Google sobre o nome do cliente e uma imagem da bandeira de Uberlândia, que é renderizada apenas se for esta a cidade.

10.4. Aplicando estilos à tabela

Apenas para mostrar que é possível deixar os sistemas desenvolvidos em JSF mais bonitos, iremos alterar o código do último exemplo e incluir estilos CSS na tabela.

Primeiro, criamos um arquivo CSS com o seguinte conteúdo:

```
.tabela {
	font-family: Arial;
	font-size: 10pt;
}

.cabecalho {
	text-align: center;
	color: white;
	background: blue;
}

.linha1 {
	background: #C9C9C9;
}

.linha2 {
	background: white;
}
```

Agora alteramos o arquivo JSP para incluir as classes do CSS.

```
<h:dataTable value="#{consultaClienteBean.clientes}"
	var="item"
	rendered="#{not empty consultaClienteBean.clientes}"
	styleClass="tabela" headerClass="cabecalho"
	rowClasses="linha1, linha2" width="70%">
	<h:column>
	   <f:facet name="header">
	      <h:panelGroup>
	          <h:outputText value="Código"/>
	          <h:graphicImage value="/imagens/ajuda.gif"/>
	      </h:panelGroup>
	   </f:facet>
	   <h:inputText value="#{item.codigo}" size="2"
	            maxlength="4"/>
```

```
            </h:column>
            <h:column>
               <f:facet name="header">
                  <h:outputText value="Nome"/>
               </f:facet>
               <h:outputLink target="_blank"
                  value="http://www.google.com.br/#hl=pt-BR&q=#{item.nome}">
                  <h:outputText value="#{item.nome}"/>
               </h:outputLink>
            </h:column>
            <h:column>
               <f:facet name="header">
                  <h:outputText value="Cidade"/>
               </f:facet>
               <h:outputText value="#{item.cidade} "/>
                  <h:graphicImage value="/imagens/bandeira_uberlandia.gif"
                        rendered="#{item.cidade eq 'Uberlândia'}"/>
            </h:column>
         </h:dataTable>
```

Especificamos os atributos `styleClass`, `headerClass` e `rowClasses` para incluir uma classe de CSS para a tabela, para o cabeçalho e para as linhas da tabela, respectivamente. No atributo `rowClasses`, colocamos nomes de duas classes CSS separados por vírgula. Quando fazemos isso, queremos dizer que as linhas da tabela devem intercalar as classes CSS a serem usadas entre as especificadas, dando o efeito, neste caso, de “cor sim e cor não”.

Consultar

Código	Nome	Cidade
1	João da Silva	Uberlândia
2	Manoel Souza	Uberaba
3	Cristina Melo	São Paulo
6	Sebastião Cardoso	Belo Horizonte
7	Francisco Borges	Uberlândia
10	Juliano Messias	Rio de Janeiro
11	Maria Helena	Uberlândia

11. Internacionalização

11.1. Usando *message bundles*

O JSF possui um mecanismo de mensagens que permite que você deixe sua aplicação internacionalizada, ou seja, com vários idiomas. A internacionalização, também conhecida por I18n, funciona através de arquivos de propriedades que possuem as mensagens do sistema, chamado de *message bundle*.

Antes de criar um exemplo para funcionar com vários idiomas, iremos apenas centralizar as mensagens em um único lugar, no *message bundle* (pacote de mensagens). Iremos alterar nosso último exemplo da tela de consulta de clientes.

Criamos um arquivo chamado "messages.properties" no pacote "com.algaworks.dwjsf.recursos" com o seguinte conteúdo:

```
search=Consultar
code=Código
name=Nome
city=Cidade
```

Cada linha deste arquivo possui pares de chaves e descrições, onde as chaves são usadas apenas como referência para quando precisarmos resgatar as descrições.

Incluímos a tag `f:loadBundle` na página JSF para carregar o pacote de mensagens que criamos e apelidamos como "msgs", através do atributo `var`:

```
<f:loadBundle basename="com.algaworks.dwjsf.recursos.messages"
      var="msgs" />
```

Agora, quando precisarmos de uma descrição para incluir à página, usamos uma expressão de ligação de valor para acessar essa descrição através da chave no pacote de mensagens, como por exemplo:

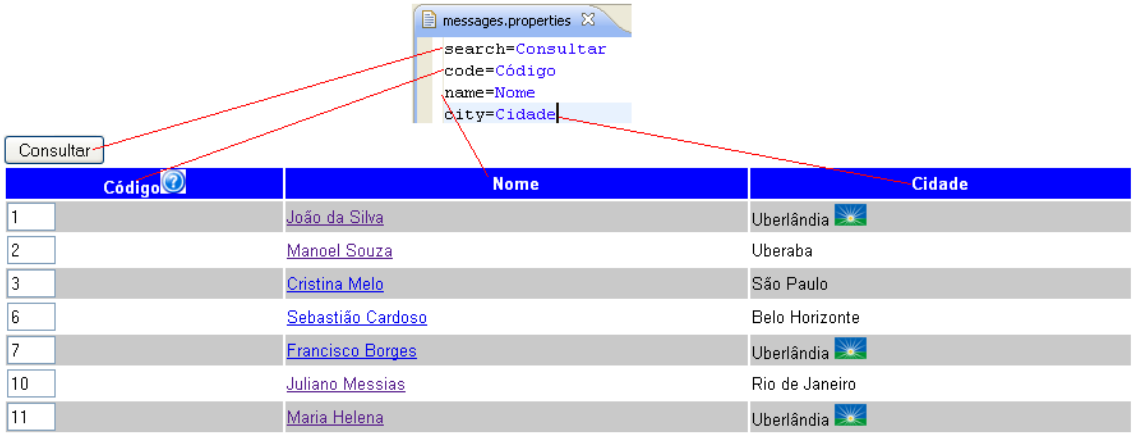
```
<h:outputText value="#{msgs.name}"/>
```

Vamos ver como ficou o código completo da página JSF, agora usando o *message bundle*.

```
<f:view>
 <f:loadBundle var="msgs"
      basename="com.algaworks.dwjsf.recursos.messages"/>

    <h:form id="frm">
      <h:commandButton value="#{msgs.search}"
            actionListener="#{consultaClienteBean.consultar}" />

      <h:dataTable value="#{consultaClienteBean.clientes}"
            var="item"
            rendered="#{not empty consultaClienteBean.clientes}"
            styleClass="tabela" headerClass="cabecalho"
            rowClasses="linha1, linha2" width="70%">
            <h:column>
               <f:facet name="header">
                  <h:panelGroup>
                     <h:outputText value="#{msgs.code}"/>
                     <h:graphicImage value="/imagens/ajuda.gif"/>
                  </h:panelGroup>
               </f:facet>
               <h:inputText value="#{item.codigo}" size="2"
                        maxlength="4"/>
            </h:column>
```

```
            <h:column>
               <f:facet name="header">
                  <h:outputText value="#{msgs.name}"/>
               </f:facet>
               <h:outputLink target="_blank"
                  value="http://www.google.com.br/#hl=pt-BR&q=#{item.nome}">
                  <h:outputText value="#{item.nome}"/>
               </h:outputLink>
            </h:column>
            <h:column>
               <f:facet name="header">
                  <h:outputText value="#{msgs.city}"/>
               </f:facet>
               <h:outputText value="#{item.cidade} "/>
                  <h:graphicImage value="/imagens/bandeira_uberlandia.gif"
                        rendered="#{item.cidade eq 'Uberlândia'}"/>
            </h:column>
      </h:dataTable>
   </h:form>
</f:view>
```

Agora você pode executar a tela novamente e verá o mesmo resultado. Visualmente o usuário não percebe que a aplicação está buscando os textos de um arquivo de propriedades.



11.2. Pacotes de mensagens para outras localidades

Se você desenvolver suas aplicações usando *message bundle*, quando surgir a necessidade de internacionalizá-las, bastará você criar arquivos de mensagens localizados, ou seja, para um idioma/região específico.

Vamos agora disponibilizar nosso exemplo em inglês. Precisamos criar um arquivo de propriedades com as descrições na língua inglesa. O nome do arquivo, neste caso, será "messages_en.properties", e terá o seguinte conteúdo:

```
search=Search
code=Code
name=Name
city=City
```

Perceba que fornecemos as descrições para todas as chaves que usamos.

O nome do arquivo possui o sufixo "en", indicando que é um pacote de mensagens em inglês (**en**glish). Este sufixo é um código da língua na norma ISO-639.

Poderíamos ter criado um arquivo chamado "messages_en_US.properties", indicando ser um arquivo em inglês (**en**glish) dos Estados Unidos (**US**), mas se fizéssemos isso, este pacote seria restrito apenas a esta localidade, e outros países que também falam este idioma

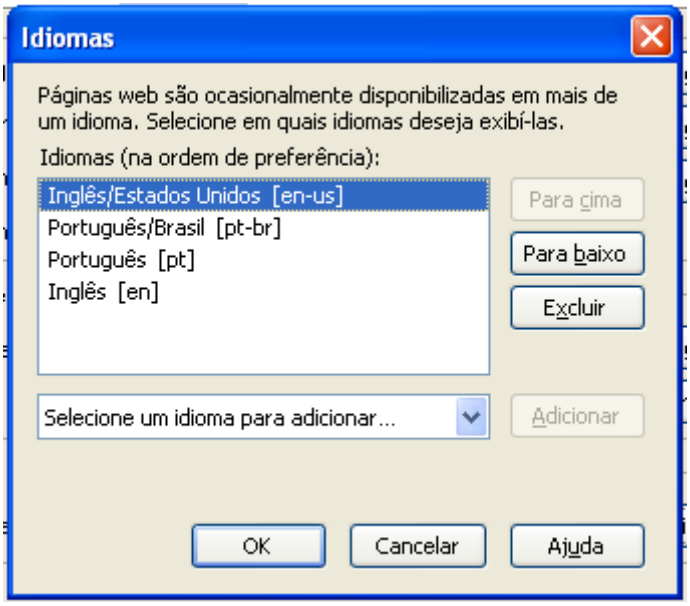
não iriam ler os textos em inglês, a não ser que criássemos outros pacotes de mensagens para isso.

Precisamos dizer ao framework JSF quais são as localidades suportadas e qual é a padrão. Para isso, incluímos o seguinte código de configuração no arquivo “faces-config.xml”:

```
<application>
	<locale-config>
		<default-locale>pt_BR</default-locale>
		<supported-locale>pt_BR</supported-locale>
		<supported-locale>en</supported-locale>
	</locale-config>
</application>
```

Quando o *browser* se comunica com aplicações web, normalmente ele inclui uma informação chamada “Accept-Language” no cabeçalho HTTP, que diz qual é o idioma preferido do usuário. O JSF usa essa informação para encontrar a melhor localização suportada pela aplicação.

Para testar nossa aplicação internacionalizada, alteramos a preferência de idiomas no navegador para priorizar a localização “en-US”.



Recarregamos a página e os textos são apresentados em inglês, pois o pacote de mensagens deste idioma foi selecionado.

Search

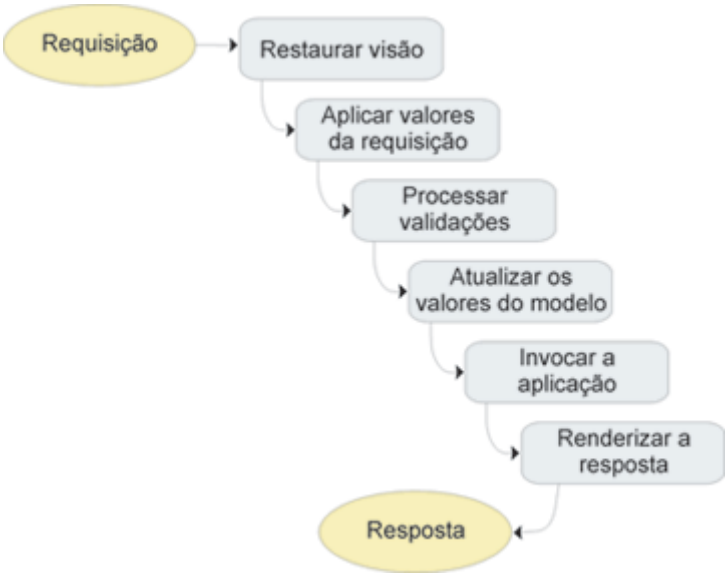
Code	Name	City
1	João da Silva	Uberlândia
2	Manoel Souza	Uberaba
3	Cristina Melo	São Paulo
6	Sebastião Cardoso	Belo Horizonte
7	Francisco Borges	Uberlândia
10	Juliano Messias	Rio de Janeiro
11	Maria Helena	Uberlândia

12. Conversão e validação

12.1. Introdução

Estudaremos neste capítulo como os dados informados pelos usuários são convertidos para objetos Java e como é feita a validação desses objetos convertidos com JavaServer Faces.

Para entender como funciona o processo de conversão e validação, precisamos nos lembrar do ciclo de vida do JSF.



Quando o usuário preenche um campo em um formulário e o submete, a informação digitada chega ao servidor e é chamada de “valor de requisição” (*request value*).

Na fase “Aplicar valores de requisição” do ciclo de vida, os valores de requisição são anexados em objetos de componentes e chamados de “valor submetido” (*submitted value*). Cada componente da página possui um objeto correspondente na memória do servidor, que armazena o valor submetido.

Os valores de requisição são do tipo *String*, pois o navegador envia o que o usuário informa como texto. Não existe outro tipo de dado para esta comunicação entre o navegador e o servidor. No código Java, precisamos lidar com tipos específicos, como um inteiro (*int*), um ponto-flutuante (*double* ou *float*), uma data (*Date*), etc. Existe então um processo de conversão que é executado pelo framework que converte os dados do tipo *String* para seus tipos específicos ainda na fase “Aplicar valores da requisição”. Essa conversão ocorre de acordo com configurações que podemos fazer nos componentes.

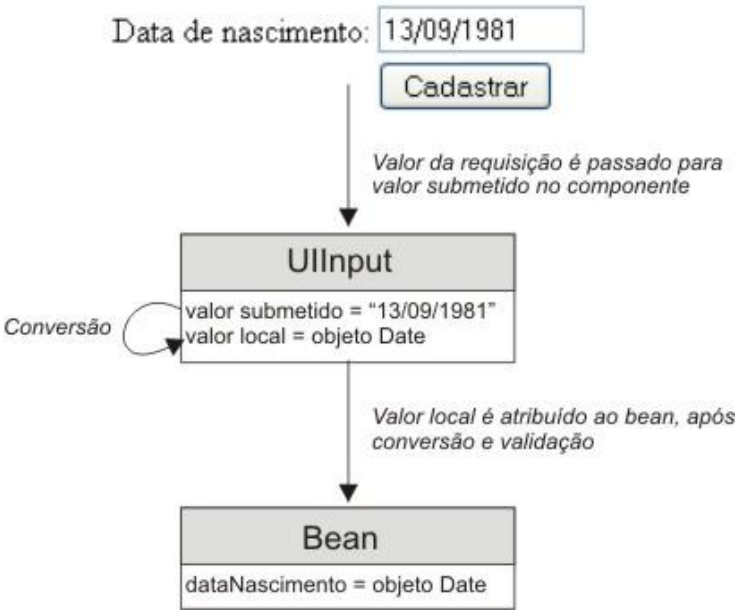
Os valores convertidos não são atribuídos aos beans, mas apenas anexados aos objetos que representam os componentes e chamados de “valores locais” (*local values*).

Neste momento, os objetos dos componentes possuem os valores submetidos em forma de texto e os valores locais já convertidos para o tipo específico.

O próximo passo no ciclo de vida é executar a fase “Processar validações”. Nesta etapa, entram em ação os validadores, que inspecionam os valores locais anexados aos componentes e os validam de acordo com o que desejamos. Estudaremos ainda neste capítulo como definir validadores de diferentes tipos, inclusive como criar o nosso próprio validador.

Inicia-se então a execução da fase “Atualizar os valores do modelo”, que atribui os valores locais já validados aos beans. Nesta fase já é seguro fazer isso, pois a entrada fornecida pelo usuário está correta, pois já foi convertida e validada.

Durante o processo de conversão e validação, se ocorrer erros ou inconsistências, a página é reexibida para que o usuário tenha a chance de corrigir o problema. Os valores locais são atualizados no modelo (nos beans) apenas se todas as conversões e validações forem bem sucedidas.



12.2. Conversores padrão de números e datas

Conversão é o processo que garante que os dados digitados pelos usuários se transformem em um tipo específico.

JSF fornece vários conversores prontos para serem usados. Todos os tipos primitivos, além de `BigInteger` e `BigDecimal`, usam conversores padrão do JSF automaticamente. Por exemplo, se você colocar um `h:inputText` referenciando um atributo do tipo `double` de um bean, o valor digitado pelo usuário será automaticamente convertido para o tipo `double` para ser atribuído ao bean.

```
<h:inputText size="12" value="#{pedidoBean.valorUnitario}"/>
```

```
public class PedidoBean {

	private double valorUnitario;

	//getters e setters
}
```

Para os tipos que não possuem um conversor padrão e também para os que já possuem, você pode especificar um conversor explicitamente, informando opções de conversão. Para testar, criaremos uma tela simples para emissão de pedido, onde o usuário deverá digitar o código do produto, a quantidade de itens, o valor unitário do produto, o número do cartão de crédito e a data do pedido. Ao clicar no botão "Enviar", uma tela de resumo do pedido será exibida, calculando o valor total do pedido baseado na quantidade de itens e valor unitário.

Criamos uma classe chamada `PedidoBean` e incluímos os atributos e métodos *getters* e *setters*.

```
package com.algaworks.dwjsf.visao;

import java.util.Date;

public class PedidoBean {
```

```
	private int codigoProduto;
	private short quantidade;
	private double valorUnitario;
	private String cartaoCredito;
	private Date dataPedido;

	public int getCodigoProduto() {
		return codigoProduto;
	}
	public void setCodigoProduto(int codigoProduto) {
		this.codigoProduto = codigoProduto;
	}
	public short getQuantidade() {
		return quantidade;
	}
	public void setQuantidade(short quantidade) {
		this.quantidade = quantidade;
	}
	public double getValorUnitario() {
		return valorUnitario;
	}
	public void setValorUnitario(double valorUnitario) {
		this.valorUnitario = valorUnitario;
	}
	public String getCartaoCredito() {
		return cartaoCredito;
	}
	public void setCartaoCredito(String cartaoCredito) {
		this.cartaoCredito = cartaoCredito;
	}
	public Date getDataPedido() {
		return dataPedido;
	}
	public void setDataPedido(Date dataPedido) {
		this.dataPedido = dataPedido;
	}

}
```

Registramos a classe `PedidoBean` como um *managed bean* e incluímos uma regra de navegação para a visão “resumoPedido.jsp” no arquivo “faces-config.xml”.

```
<managed-bean>
	<managed-bean-name>pedidoBean</managed-bean-name>
	<managed-bean-class>
		com.algaworks.dwjsf.visao.PedidoBean
	</managed-bean-class>
	<managed-bean-scope>request</managed-bean-scope>
</managed-bean>

<navigation-rule>
	<navigation-case>
		<from-outcome>enviarPedido</from-outcome>
		<to-view-id>/resumoPedido.jsp</to-view-id>
	</navigation-case>
</navigation-rule>
```

Agora criamos a página com o formulário para emissão de pedidos:

```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<body>
<f:view>
      <h:form id="frm">
            <h1>
              <h:outputText value="Informações sobre o pedido:"/>
            </h1>

            <h:panelGrid columns="2" >
                  <h:outputText value="Cód. do produto:"/>
                  <h:panelGroup>
                        <h:inputText id="codigoProduto" size="12"
                              value="#{pedidoBean.codigoProduto}"/>
                        <h:message for="codigoProduto"
                              showSummary="true" showDetail="false"/>
                  </h:panelGroup>

                  <h:outputText value="Quantidade:"/>
                  <h:panelGroup>
                        <h:inputText id="quantidade" size="12"
                              value="#{pedidoBean.quantidade}"/>
                        <h:message for="quantidade"
                              showSummary="true" showDetail="false"/>
                  </h:panelGroup>

                  <h:outputText value="Valor unitário:"/>
                  <h:panelGroup>
                        <h:inputText id="valorUnitario" size="12"
                              value="#{pedidoBean.valorUnitario}">
                              <f:convertNumber minFractionDigits="2"/>
                        </h:inputText>
                        <h:message for="valorUnitario"
                              showSummary="true" showDetail="false"/>
                  </h:panelGroup>

                  <h:outputText value="Cartão de crédito:"/>
                  <h:inputText id="cartaoCredito" size="20"
                        value="#{pedidoBean.cartaoCredito}"/>

                  <h:outputText value="Data do pedido:"/>
                  <h:panelGroup>
                        <h:inputText id="dataPedido" size="12"
                              value="#{pedidoBean.dataPedido}">
                              <f:convertDateTime pattern="dd/MM/yyyy"/>
                        </h:inputText>
                        <h:message for="dataPedido"
                              showSummary="true" showDetail="false"/>
                  </h:panelGroup>

                  <h:outputText/>
                  <h:commandButton id="cadastrar" value="Enviar"
                                   action="enviarPedido"/>
            </h:panelGrid>
      </h:form>
</f:view>
</body>
```

```
</html>
```

Não especificamos nenhum conversor para o campo de código do produto, pois será configurado automaticamente um conversor de inteiros.

```
<h:inputText id="codigoProduto" size="12"
      value="#{pedidoBean.codigoProduto}"/>
```

O mesmo acontece para o campo de quantidade, porém neste caso, um conversor de números short será incluído automaticamente.

```
<h:inputText id="quantidade" size="12"
      value="#{pedidoBean.quantidade}"/>
```

Para o campo de valor unitário, vinculamos um conversor numérico e dizemos para formatar o valor com no mínimo 2 casas decimais. Veja que para especificar o conversor numérico, usamos a tag `f:convertNumber`.

```
<h:inputText id="valorUnitario" size="12"
      value="#{pedidoBean.valorUnitario}">
      <f:convertNumber minFractionDigits="2"/>
</h:inputText>
```

O campo do número do cartão de crédito não possui um conversor, pois o tipo de dado é *String*.

```
<h:inputText id="cartaoCredito" size="20"
      value="#{pedidoBean.cartaoCredito}"/>
```

O campo de data do pedido usa um conversor de data/hora. Especificamos o atributo `pattern` para definir o formato da data que o usuário deverá digitar. Os possíveis *patterns* podem ser estudados na documentação da classe `java.text.SimpleDateFormat`.

```
<h:inputText id="dataPedido" size="12"
      value="#{pedidoBean.dataPedido}">
      <f:convertDateTime pattern="dd/MM/yyyy"/>
</h:inputText>
```

Para exibir as mensagens de erro de conversão (caso o usuário digite alguma informação incoerente), incluímos em vários pontos da tela a tag `h:message`, vinculando com campos do formulário.

```
<h:message for="dataPedido" showSummary="true"
      showDetail="false"/>
```

Criamos a página de resumo do pedido, que será usada apenas para exibir os dados digitados pelo usuário, com o valor total do pedido calculado.

```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<body>
<f:view>
      <h1><h:outputText value="Informações sobre o pedido:"/></h1>

      <h:panelGrid columns="2" >
            <h:outputText value="Cód. do produto:"/>
            <h:outputText value="#{pedidoBean.codigoProduto}"/>
```

```
            <h:outputText value="Quantidade:"/>
            <h:outputText value="#{pedidoBean.quantidade}"/>

            <h:outputText value="Valor unitário:"/>
            <h:outputText value="#{pedidoBean.valorUnitario}">
                  <f:convertNumber type="currency" currencyCode="BRL"/>
            </h:outputText>

            <h:outputText value="Valor total:"/>
            <h:outputText value="#{pedidoBean.valorUnitario
                        * pedidoBean.quantidade}">
                  <f:convertNumber type="currency" currencyCode="BRL"/>
            </h:outputText>

            <h:outputText value="Cartão de crédito:"/>
            <h:outputText value="#{pedidoBean.cartaoCredito}"/>

            <h:outputText value="Data do pedido:"/>
            <h:outputText value="#{pedidoBean.dataPedido}">
                  <f:convertDateTime pattern="dd, MMMM yyyy"/>
            </h:outputText>
      </h:panelGrid>
</f:view>
</body>
</html>
```

O código do produto, a quantidade e o número do cartão de crédito são exibidos normalmente, sem nenhum conversor.

```
      <h:outputText value="Cód. do produto:"/>
      <h:outputText value="#{pedidoBean.codigoProduto}"/>

      <h:outputText value="Quantidade:"/>
      <h:outputText value="#{pedidoBean.quantidade}"/>

      ...

      <h:outputText value="Cartão de crédito:"/>
      <h:outputText value="#{pedidoBean.cartaoCredito}"/>
```

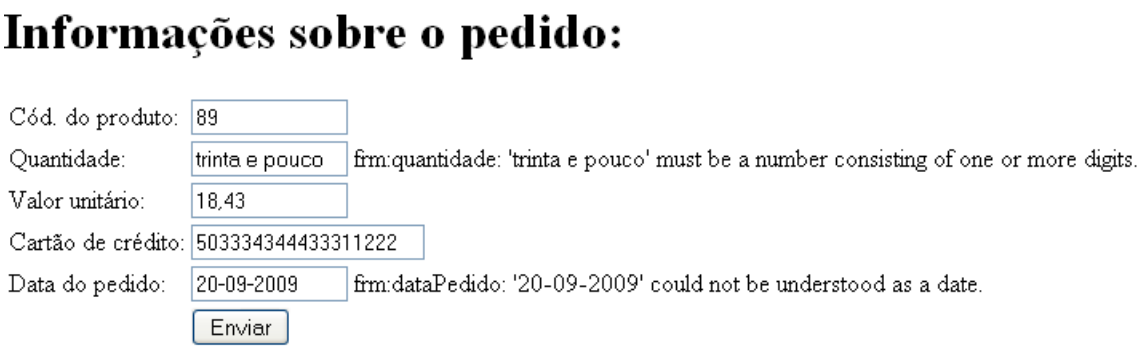
O valor unitário e valor total são exibidos usando um formatador de moeda. Para isso, incluímos o atributo `type` igual a `currency` e o `currencyCode` igual a `BRL`. O código “BRL” é uma representação da moeda brasileira de acordo com o padrão internacional ISO 4217, que define códigos de moedas.

```
      <h:outputText value="Valor unitário:"/>
      <h:outputText value="#{pedidoBean.valorUnitario}">
            <f:convertNumber type="currency" currencyCode="BRL"/>
      </h:outputText>

      <h:outputText value="Valor total:"/>
      <h:outputText value="#{pedidoBean.valorUnitario
            * pedidoBean.quantidade}">
            <f:convertNumber type="currency" currencyCode="BRL"/>
      </h:outputText>
```

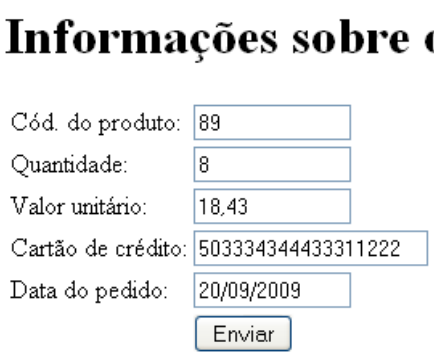
A data do pedido é especificada com o conversor de data/hora, usando um padrão diferente do usado para entrada da informação pelo usuário. Este padrão exibe a data com o mês por extenso.

```
<h:outputText value="Data do pedido:"/>
<h:outputText value="#{pedidoBean.dataPedido}">
      <f:convertDateTime pattern="dd, MMMM yyyy"/>
</h:outputText>
```

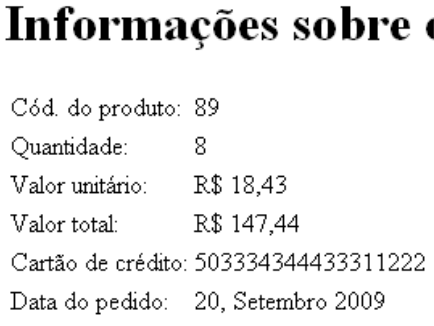
Para testar, executamos a página “pedido.jsp” e preenchemos todos os campos.
Erramos no preenchimento do campo de quantidade e data do pedido de propósito, para vermos as mensagens de erro:



Agora preenchemos corretamente e clicamos no botão “Enviar”.



A página “resumoPedido.jsp” é exibida. Veja que os conversores entraram em ação para formatar os valores monetários e a data.



12.3. Alternativas para definir conversores

Você pode especificar um conversor explicitamente em um componente adicionando o atributo `converter` na tag do componente, informando um ID do conversor como valor para este atributo. Por exemplo:

```
<h:outputText value="#{pedidoBean.codigoProduto}"
      converter="javax.faces.Integer"/>
```

O framework JSF já possui alguns IDs de conversores pré-definidos, como:

- javax.faces.Number
- javax.faces.Boolean
- javax.faces.Byte
- javax.faces.Character
- javax.faces.Double
- javax.faces.Float
- javax.faces.Integer
- javax.faces.Long
- javax.faces.Short
- javax.faces.BigDecimal
- javax.faces.BigInteger
- javax.faces.DateTime

Outra forma de explicitar um conversor em um componente é usando a tag `f:converter`, especificando o ID do conversor no atributo `converterId`.

```
<h:outputText value="#{pedidoBean.codigoProduto}">
      <f:converter converterId="javax.faces.Integer"/>
</h:outputText>
```

12.4. Customizando mensagens de erro de conversão

Você pode precisar customizar as mensagens de erros, pois as que vêem no framework são razoavelmente feias. Estudaremos agora como mudar estas mensagens para nosso idioma.

Primeiramente iremos customizar as mensagens de erro de conversão de campos do tipo de data. Para começar, criamos um arquivo de *messages bundle* e incluímos o seguinte conteúdo dentro dele:

```
javax.faces.converter.DateTimeConverter.DATE=Data inválida.
javax.faces.converter.DateTimeConverter.DATE_detail=O campo ''{2}'' não foi informado com uma data válida.
```

Este arquivo foi colocado dentro do pacote `com.algaworks.dwjsf.recursos`, com o nome “messages.properties”.

Customizamos as mensagens de erro de resumo e detalhe para conversão de data/hora.

Incluímos o seguinte código no arquivo “faces-config.xml”, para que este *messages bundle* seja carregado pela aplicação.

```
<application>
      <message-bundle>
            com.algaworks.dwjsf.recursos.messages
      </message-bundle>
</application>
```

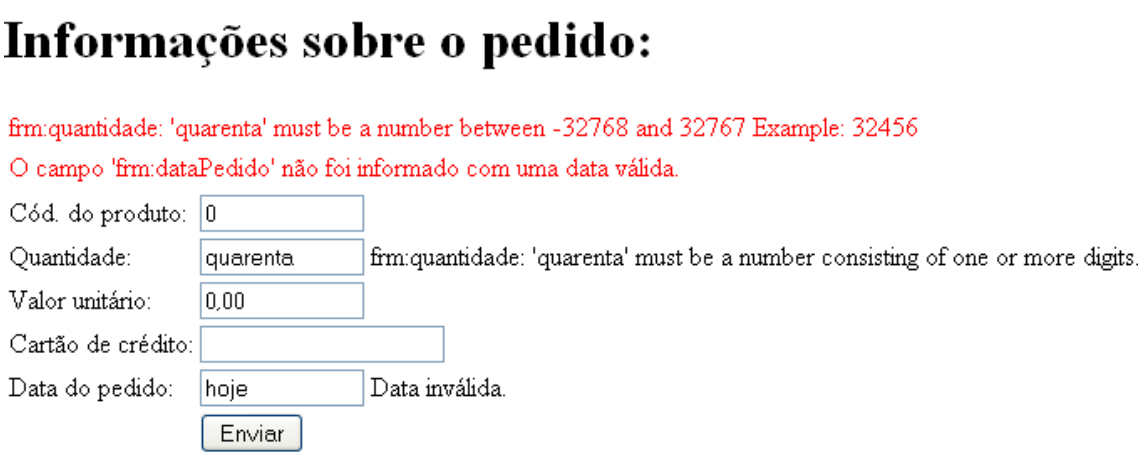
Incluímos a tag `h:messages` no arquivo “pedido.jsp”, antes dos campos do formulário:

```
<h:messages layout="table" errorStyle="color: red"
      infoStyle="color: green" warnStyle="color: orange"
      fatalStyle="color: gray"
```

```
            showSummary="false" showDetail="true"/>
```

Esta alteração foi feita apenas para testarmos também a customização da mensagem de detalhamento do erro, que deverá aparecer no formato de tabela antes dos campos do formulário.

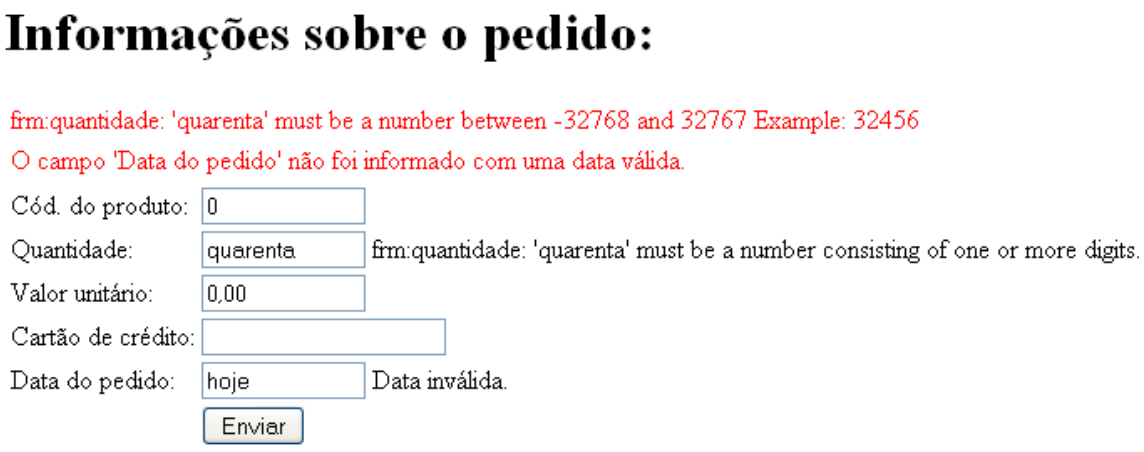
Ao digitar uma informação errada em um campo de data, veja que a mensagem que customizamos aparecerá. As mensagens de erro de conversão para campos numéricos continuam antigas, sem customizações. Melhoraremo-las também daqui a pouco.



A mensagem de detalhamento do erro de conversão de data ainda pode ser melhorada, pois provavelmente o usuário não gostará de ler "frm:dataPedido" como referência ao campo de data do pedido. Para ajustar o nome do campo e deixá-lo legível ao usuário leigo, basta adicionarmos o atributo `label` à tag `h:inputText`.

```
<h:inputText id="dataPedido" size="12"
      value="#{pedidoBean.dataPedido}" label="Data do pedido">
      <f:convertDateTime pattern="dd/MM/yyyy"/>
</h:inputText>
```

Agora a mensagem de erro de conversão de data está legível ao usuário final.



Ajustamos as mensagens de erro de conversão dos tipos numéricos que usamos na tela. Incluímos o seguinte conteúdo no arquivo de *messages bundle*:

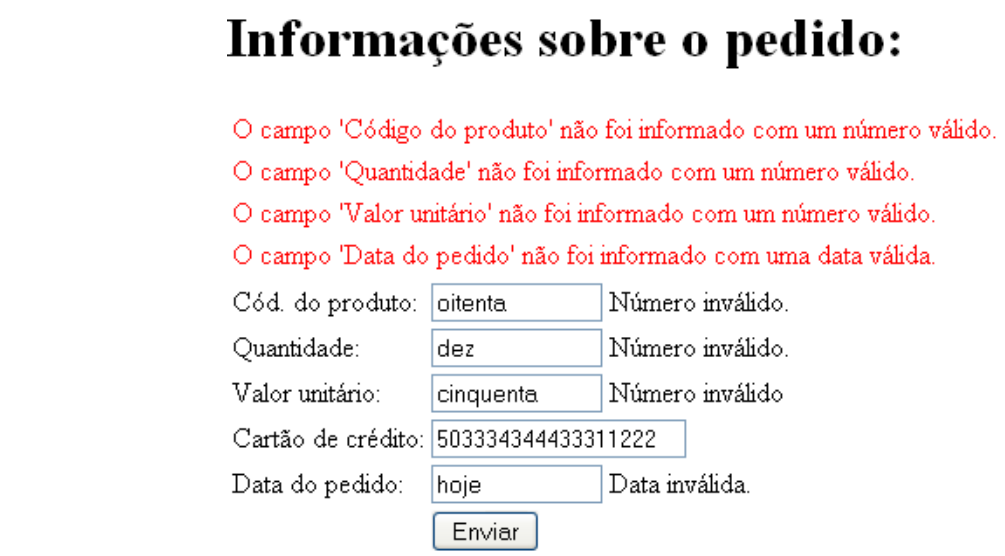
```
javax.faces.converter.IntegerConverter.INTEGER=Número inválido.
javax.faces.converter.IntegerConverter.INTEGER_detail=O campo ''{2}''
não foi informado com um número válido.
javax.faces.converter.ShortConverter.SHORT=Número inválido.
```

```
javax.faces.converter.ShortConverter.SHORT_detail=O campo ''{2}'' não
foi informado com um número válido.
javax.faces.converter.NumberConverter.NUMBER=Número inválido
javax.faces.converter.NumberConverter.NUMBER_detail=O   campo   ''{2}''
não foi informado com um número válido.
```

Alteramos também o JSP da tela, incluindo o atributo `label` em todos campos:

```
<h:outputText value="Cód. do produto:"/>
<h:panelGroup>
      <h:inputText id="codigoProduto" size="12"
            value="#{pedidoBean.codigoProduto}"
            label="Código do produto"/>
      <h:message for="codigoProduto"
            showSummary="true" showDetail="false"/>
</h:panelGroup>

<h:outputText value="Quantidade:"/>
<h:panelGroup>
      <h:inputText id="quantidade" size="12"
            value="#{pedidoBean.quantidade}" label="Quantidade"/>
      <h:message for="quantidade"
            showSummary="true" showDetail="false"/>
</h:panelGroup>

<h:outputText value="Valor unitário:"/>
<h:panelGroup>
      <h:inputText id="valorUnitario" size="12"
            value="#{pedidoBean.valorUnitario}"
            label="Valor unitário">
            <f:convertNumber minFractionDigits="2"/>
      </h:inputText>
      <h:message for="valorUnitario"
            showSummary="true" showDetail="false"/>
</h:panelGroup>

<h:outputText value="Cartão de crédito:"/>
<h:inputText id="cartaoCredito" size="20"
      value="#{pedidoBean.cartaoCredito}"/>

<h:outputText value="Data do pedido:"/>
<h:panelGroup>
      <h:inputText id="dataPedido" size="12"
            value="#{pedidoBean.dataPedido}"
            label="Data do pedido">
            <f:convertDateTime pattern="dd/MM/yyyy"/>
      </h:inputText>
      <h:message for="dataPedido"
            showSummary="true" showDetail="false"/>
</h:panelGroup>
```

Pronto, agora todas as mensagens de erro de conversão que precisamos em nossa tela estão traduzidas e bonitas.

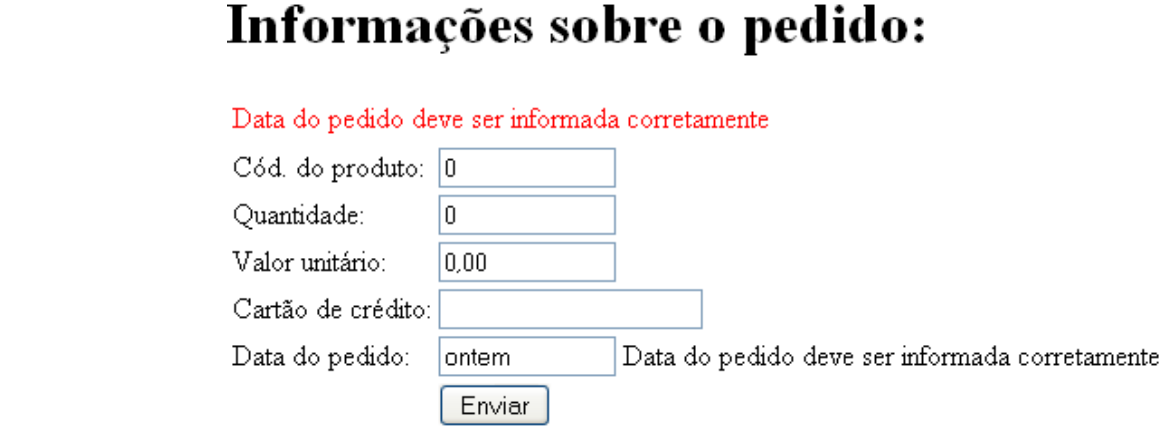


Se você precisar customizar outras mensagens de erro, você deverá descobrir as chaves das mensagens. Uma forma rápida e fácil para descobrir essas chaves é abrindo o arquivo original de *messages bundle* do JSF. Este arquivo está localizado no pacote `javax.faces`, dentro do arquivo "jsp-api.jar". Descompacte este JAR, encontre o arquivo e abra em um editor de texto.

A customização de mensagens que acabamos de fazer é bastante interessante, pois toda a aplicação utiliza as novas mensagens, sem precisarmos especificar tela por tela. Mas algumas vezes podemos precisar especificar uma mensagem de erro de conversão específica para um campo de uma tela. Se este for o caso, não podemos alterar a mensagem no *messages bundle*, pois se fizesse isso, as mensagens de todas as telas seriam alteradas. Para este cenário, devemos usar o atributo `converterMessage` do campo de entrada, como no exemplo abaixo:

```
<h:inputText id="dataPedido" size="12"
   value="#{pedidoBean.dataPedido}" label="Data do pedido"
   converterMessage="Data do pedido deve ser informada corretamente">
   <f:convertDateTime pattern="dd/MM/yyyy"/>
</h:inputText>
```

A partir de agora, apenas no campo de data do pedido desta tela, a mensagem de erro de conversão ficará diferente, de acordo com o que especificamos no atributo `converterMessage`.



Você deve ter percebido que essa última forma de customizar mensagens de erro de conversão não separa a mensagem de erro resumo e detalhamento do erro. O que informamos

como valor para o atributo `converterMessage` é definido como ambos os tipos (resumo e detalhamento).

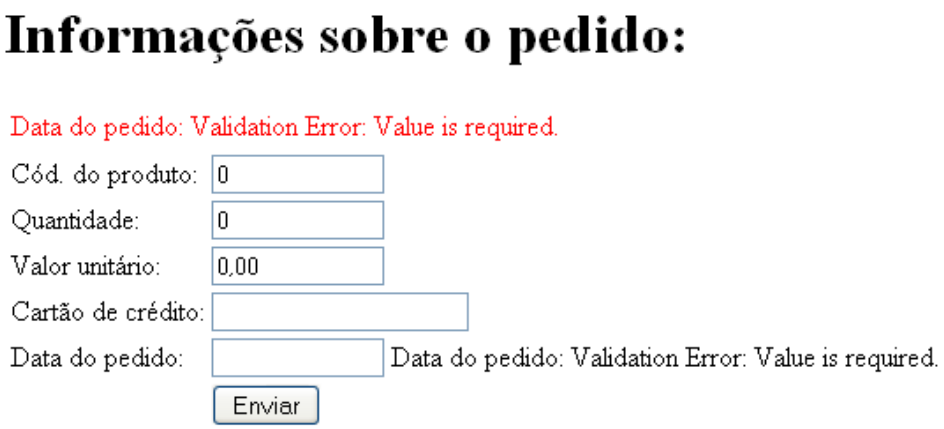
12.5. Usando validadores

O processo de validação é importante para qualquer aplicação. O JavaServer Faces possui um mecanismo de validação simples e flexível. O objetivo das validações no JSF é proteger o modelo (os beans). Eles devem ser atualizados com as informações digitadas pelos usuários apenas após a conversão e validação com sucesso de todos os campos do formulário. Dessa forma, evita-se que o modelo fique em um estado inconsistente, ou seja, que alguns atributos fiquem atualizados e outros não.

O validador mais simples de ser usado é o que obriga o preenchimento de um campo. Para incluir este validador, basta adicionarmos o atributo `required="true"` em um campo de entrada.

```
<h:inputText id="dataPedido" size="12"
     value="#{pedidoBean.dataPedido}" label="Data do pedido"
     converterMessage="Data do pedido deve ser informada corretamente"
     required="true">
     <f:convertDateTime pattern="dd/MM/yyyy"/>
</h:inputText>
```

Agora não conseguimos mais submeter o formulário sem informar a data do pedido, pois o tornamos obrigatório.

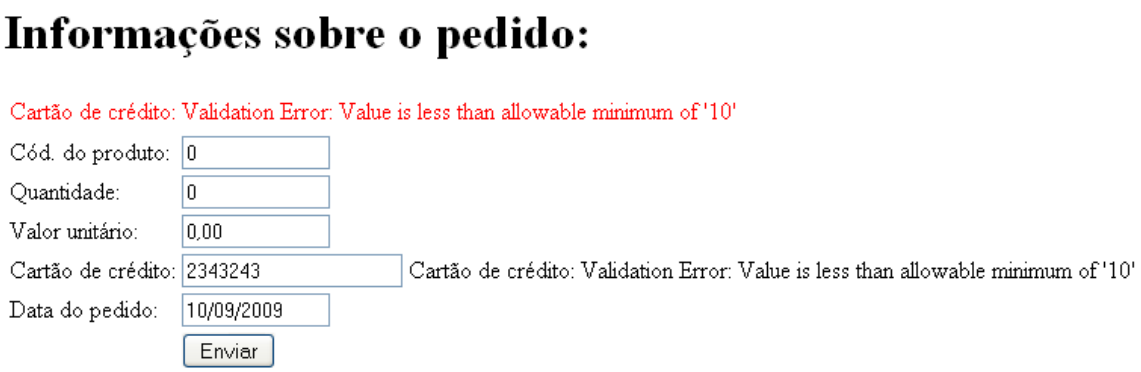


As mensagens de erro de validação também podem ser customizadas. Estudaremos isso no próximo tópico.

Existem outros validadores padrão do JSF que são fornecidos através de tags. No exemplo abaixo, adicionamos um validador que restringe o tamanho do texto digitado pelo usuário em no mínimo 10 caracteres. Veja que simplesmente adicionamos a tag `f:validateLength`.

```
<h:outputText value="Cartão de crédito:"/>
<h:panelGroup>
      <h:inputText id="cartaoCredito" size="20"
            value="#{pedidoBean.cartaoCredito}"
            required="true" label="Cartão de crédito">
            <f:validateLength minimum="10"/>
      </h:inputText>
      <h:message for="cartaoCredito"
            showSummary="true" showDetail="false"/>
</h:panelGroup>
```

O validador que acabamos de usar verifica se a string possui pelo menos 10 caracteres. Este validador não é executado se o valor do campo estiver vazio, por isso obrigamos o preenchimento colocando o atributo `required="true"`.



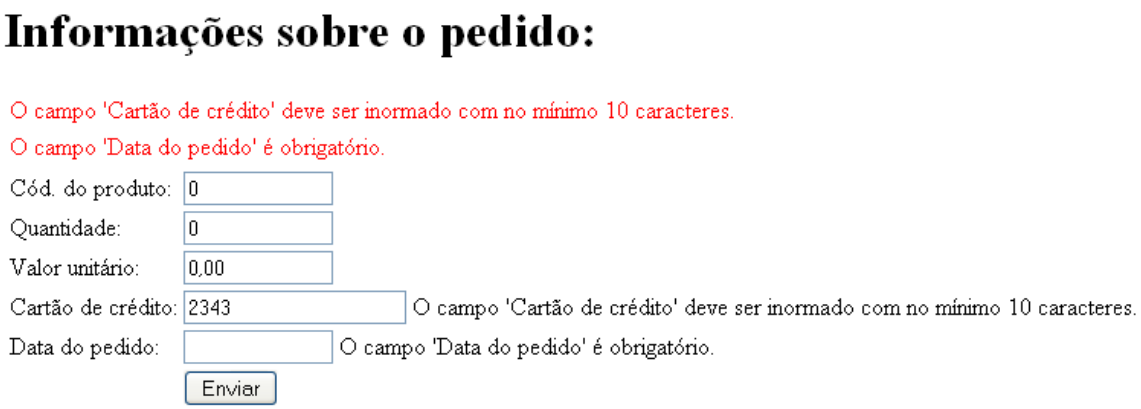
Além do validador fornecido pela tag f:validateLength, o JSF também possui outros dois validadores: `f:validateDoubleRange` e `f:validateLongRange`. O primeiro verifica se um valor numérico (ponto-flutuante) informado no campo está em um intervalo pré-definido e o segundo se um valor numérico (inteiro) também está em um intervalo de números. Ambos os validadores possuem os atributos `minimum` e `maximum` para especificar o intervalo.

12.6. Customizando mensagens de erros de validação

Assim como customizamos as mensagens de erros de conversão, também podemos customizar as mensagens de erros de validação. Vamos fornecer mensagens com as chaves apropriadas para a validação de obrigatoriedade (*required*) e para o validador de tamanho de strings. Editamos o arquivo “messages.properties” e incluímos o seguinte conteúdo:

```
javax.faces.component.UIInput.REQUIRED=O campo ''{0}'' é obrigatório.
javax.faces.validator.DoubleRangeValidator.MAXIMUM=O campo ''{1}''
deve ser informado com no máximo "{0}" caracteres.
javax.faces.validator.DoubleRangeValidator.MINIMUM=O    campo    ''{1}''
deve ser inormado com no mínimo "{0}" caracteres.
```

As mensagens de erro de validação que precisamos em nossa tela estão customizadas e traduzidas.

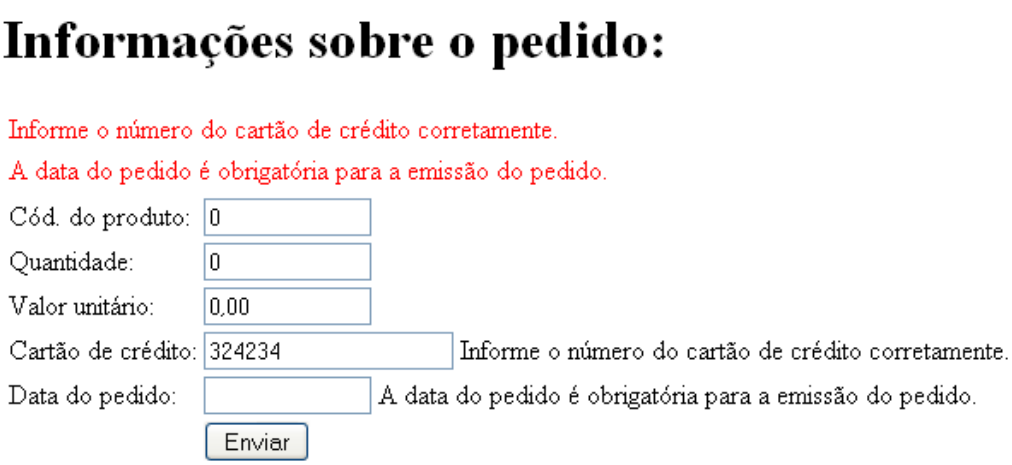


As mensagens de erro de validação que acabamos de customizar são genéricas, e por isso são utilizadas para todas as telas do sistema. Podemos customizar as mensagens de erro de validação para campos especificos, utilizando os atributos `requiredMessage` e `validatorMessage`, como no exemplo abaixo.

```
<h:inputText id="cartaoCredito" size="20"
```

```
        value="#{pedidoBean.cartaoCredito}"
        required="true" label="Cartão de crédito"
        requiredMessage="O número do cartão de crédito é obrigatório."
        validatorMessage="Informe o número do cartão de crédito
                          corretamente.">
        <f:validateLength minimum="10"/>
    </h:inputText>

    ...

    <h:inputText id="dataPedido" size="12"
        value="#{pedidoBean.dataPedido}"
        required="true" label="Data do pedido"
        converterMessage="Data do pedido deve ser informada corretamente."
        requiredMessage="A data do pedido é obrigatória para a
                         emissão do pedido.">
        <f:convertDateTime pattern="dd/MM/yyyy"/>
    </h:inputText>
```

Agora apenas nos campos de cartão de crédito e data do pedido da tela de emissão de pedidos, as mensagens foram customizadas como você pode ver abaixo:



12.7. Ignorando validações com o atributo `immediate`

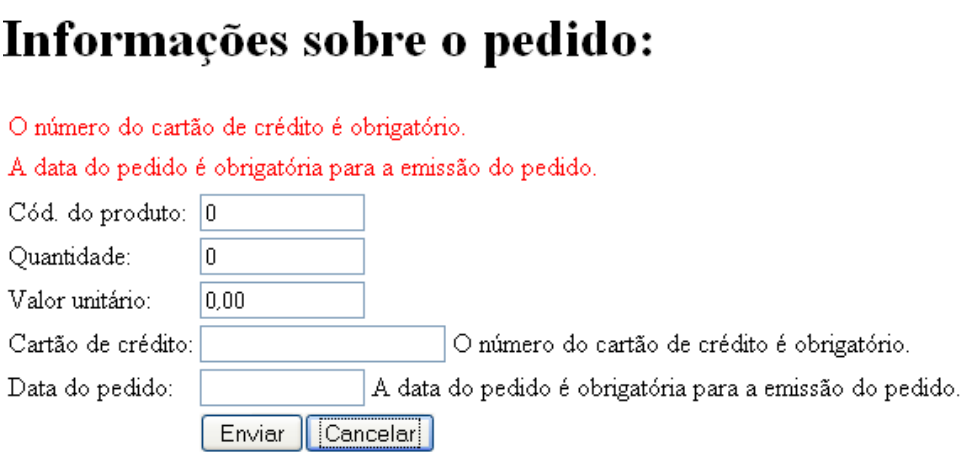
Na tela de emissão de pedidos, identificamos a necessidade de adicionar um botão “Cancelar”, o qual deve levar o usuário para outra tela do sistema, cancelando o processo de emissão do pedido. Para isso, incluímos o botão “Cancelar”.

```
<h:panelGroup>
      <h:commandButton id="cadastrar" value="Enviar"
                       action="enviarPedido"/>
      <h:commandButton value="Cancelar" action="cancelar"/>
</h:panelGroup>
```

Incluímos também a regra de navegação que responde à ação “cancelar”.

```
<navigation-rule>
      <navigation-case>
            <from-outcome>cancelar</from-outcome>
            <to-view-id>/menu.jsp</to-view-id>
      </navigation-case>
</navigation-rule>
```

Agora, na tela de emissão de pedidos, ao clicar no botão cancelar, queremos ser encaminhados para a tela principal do sistema (chamada "menu.jsp"), cancelando tudo que estávamos fazendo com a emissão do pedido, mas veja o que acontece:



O JSF não conseguiu nos encaminhar para outra tela, pois o mecanismo de validação entrou em ação e verificou que havia campos obrigatórios. O problema é que neste caso, não queremos que seja feita nenhuma validação, precisamos apenas que o usuário seja encaminhado para outra tela.

A solução é utilizarmos um artifício para antecipar a execução do comando do botão ainda na fase "Aplicar valores da requisição" do ciclo de vida, antes da conversão e validação serem executadas. Para fazer isso, basta adicionarmos o atributo `immediate="true"` na tag `h:commandButton`.

```
<h:commandButton value="Cancelar" action="cancelar"
      immediate="true"/>
```

Agora podemos clicar no botão "Cancelar" sem nos preocupar com as conversões e validações.

12.8. Criando conversores personalizados

Os conversores que já vêem embutidos no JSF são úteis na maioria dos casos, porém existem algumas situações que você pode precisar criar um conversor customizado. No exemplo a seguir, criamos um conversor personalizado para ser usado em campos de data. Sabemos que para campos de data já existe um conversor padrão, que inclusive já foi utilizado por nós em exemplos anteriores, mas nosso conversor será um pouco diferente. Queremos um conversor de datas que transforma a entrada do usuário em objetos do tipo `Date`, usando o formato `dd/MM/yyyy` e com um grande diferencial: o usuário poderá também informar palavras que serão convertidas em datas, como "amanhã", "hoje" ou "ontem".

O primeiro passo para fazer um novo conversor é criar uma classe que implementa a interface `Converter` (do pacote `javax.faces.convert`). Um conversor é uma classe que transforma strings em objetos e objetos em strings, por isso essa classe deve implementar os métodos `getAsObject()` e `getAsString()`.

O método `getAsObject()` deve converter de string para objeto e o método `getAsString()` deve converter de objeto para string. Ao converter de string para objeto, o método pode lançar a exceção `ConverterException`, caso ocorra algum problema durante a conversão.

Veja abaixo o código-fonte do nosso conversor.

```
package com.algaworks.dwjsf.conversores;

//várias importações

public class SmartDateConverter implements Converter {

	private static final DateFormat formatador =
		new SimpleDateFormat("dd/MM/yyyy");

	private Calendar getDataAtual() {
		Calendar dataAtual = new GregorianCalendar();

		//limpamos informações de hora, minuto, segundo
		//e milisegundos
		dataAtual.set(Calendar.HOUR_OF_DAY, 0);
		dataAtual.set(Calendar.MINUTE, 0);
		dataAtual.set(Calendar.SECOND, 0);
		dataAtual.set(Calendar.MILLISECOND, 0);

		return dataAtual;
	}

	@Override
	public Object getAsObject(FacesContext context,
			UIComponent component, String value) {
		if (value == null || value.equals("")) {
			return null;
		}

		Date dataConvertida = null;
		if ("hoje".equalsIgnoreCase(value)) {
			dataConvertida = getDataAtual().getTime();
		} else if ("amanha".equalsIgnoreCase(value)
			|| "amanhã".equalsIgnoreCase(value)) {
			Calendar amanha = getDataAtual();
			amanha.add(Calendar.DAY_OF_MONTH, 1);
			dataConvertida = amanha.getTime();
		} else if ("ontem".equalsIgnoreCase(value)) {
			Calendar amanha = getDataAtual();
			amanha.add(Calendar.DAY_OF_MONTH, -1);
			dataConvertida = amanha.getTime();
		} else {
			try {
				dataConvertida = formatador.parse(value);
			} catch (ParseException e) {
			    throw new ConverterException(
			        new FacesMessage(FacesMessage.SEVERITY_ERROR,
					"Data incorreta.",
					"Favor informar uma data correta."));
			}
		}
		return dataConvertida;
	}

	@Override
	public String getAsString(FacesContext context,
			UIComponent component, Object value) {
		Date data = (Date)value;
```

```
            return formatador.format(data);
        }

}
```

Após a criação da classe do conversor e a implentação dos métodos da interface `Converter`, precisamos registrar o conversor na aplicação JSF. Para isso, basta incluir o seguinte conteúdo no arquivo "faces-config.xml":

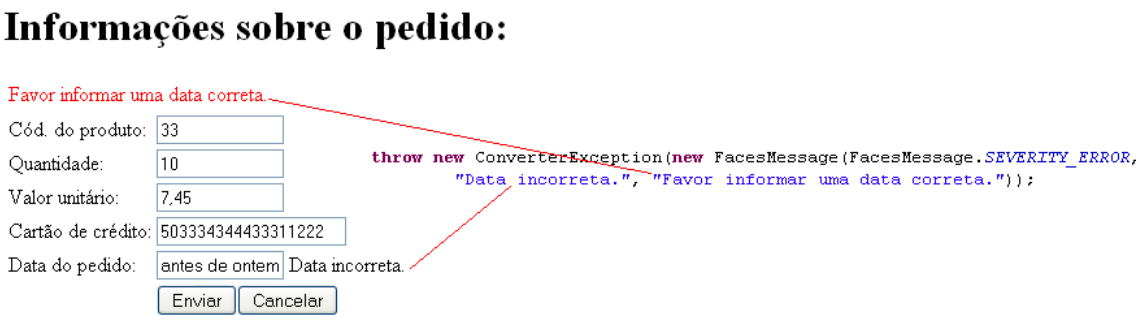
```
<converter>
        <converter-id>com.algaworks.Date</converter-id>
        <converter-class>
              com.algaworks.dwjsf.conversores.SmartDateConverter
        </converter-class>
</converter>
```

O código acima registra nosso conversor, de nome `SmartDateConverter`, para um ID igual a `com.algaworks.Date`. O ID funciona como um apelido, o qual nós utilizaremos para referenciar o conversor quando for preciso.

Agora podemos usar o conversor simplesmente informando o ID registrado no atributo converter da tag `h:inputText`.

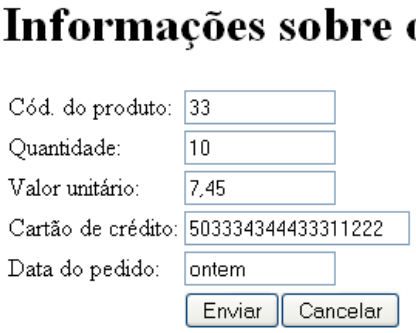
```
<h:inputText id="dataPedido" size="12"
        value="#{pedidoBean.dataPedido}" label="Data do pedido"
        requiredMessage="A data do pedido é obrigatória para a
                         emissão do pedido."
        converter="com.algaworks.Date"/>
```

Carregamos a tela de envio de pedido e informamos o texto "antes de ontem" no campo de data, veja:



Nosso conversor não foi programado para entender o significado de "antes de ontem", por isso uma exceção foi lançada e a mensagem de erro que escrevemos apareceu na tela.

Agora digitamos o texto "ontem" na data do pedido:



Veja que nosso conversor processou o texto digitado e converteu em uma data igual a um dia a menos da data atual.

Informações sobre o pedido:

Cód. do produto: 33
Quantidade: 10
Valor unitário: R$ 7,45
Valor total: R$ 74,50
Cartão de crédito: 503334344433311222
Data do pedido: 22, Setembro 2009

12.9. Criando validadores personalizados

Você pode também criar seus próprios validadores personalizados para usar com JSF. Para exemplificar, vamos adicionar um atributo `cpf` na classe `PedidoBean` e incluir o campo na tela.

```
public class PedidoBean {

	private String cpf;

	...

	public String getCpf() {
		return cpf;
	}
	public void setCpf(String cpf) {
		this.cpf = cpf;
	}

}
```

```
<h:outputText value="CPF:"/>
<h:panelGroup>
    <h:inputText id="cpf" size="12" value="#{pedidoBean.cpf}"/>
    <h:message for="cpf" showSummary="true" showDetail="false"/>
</h:panelGroup>
```

Vamos criar agora um validador que verifica se um CPF é válido. Para isso, criamos uma classe que implementa a interface `javax.faces.validator.Validator`.

```
package com.algaworks.dwjsf.validadores;

//várias importações

public class CPFValidator implements Validator {

	@Override
	public void validate(FacesContext context,
		UIComponent component, Object value)
			throws ValidatorException {
		if (value == null) {
			return;
		}
		String cpf = (String)value;

		if (cpf.length() != 11
```

```
                || !calcularDigitoVerificador(cpf.substring(0, 9))
                        .equals(cpf.substring(9, 11))) {
                throw new ValidatorException(
                        new FacesMessage(FacesMessage.SEVERITY_ERROR,
                                "CPF inválido.",
                                "Favor informar um CPF válido."));
        }
    }

    //Créditos ao JavaFree.org pelo algoritmo de validação de CPF
    private String calcularDigitoVerificador(String num) {
        Integer primDig, segDig;
        int soma = 0, peso = 10;
        for (int i = 0; i < num.length(); i++) {
            soma += Integer.parseInt(num.substring(i, i + 1))
                    * peso--;
        }

        if (soma % 11 == 0 | soma % 11 == 1) {
            primDig = new Integer(0);
        } else {
            primDig = new Integer(11 - (soma % 11));
        }

        soma = 0;
        peso = 11;
        for (int i = 0; i < num.length(); i++) {
            soma += Integer.parseInt(num.substring(i, i + 1))
                    * peso--;
        }

        soma += primDig.intValue() * 2;
        if (soma % 11 == 0 | soma % 11 == 1) {
            segDig = new Integer(0);
        } else {
            segDig = new Integer(11 - (soma % 11));
        }

        return primDig.toString() + segDig.toString();
    }

}
```

Nossa classe de validação precisou implementar o método `validate()`. Em nosso exemplo, este método precisou executar um outro método utilitário que executa o algoritmo de validação de CPF. Se a informação fornecida pelo usuário estiver incorreta, lançamos uma `ValidatorException`.

Precisamos registrar nosso validador na aplicação JSF. Basta incluir o seguinte fragmento de código no arquivo “faces-config.xml”:

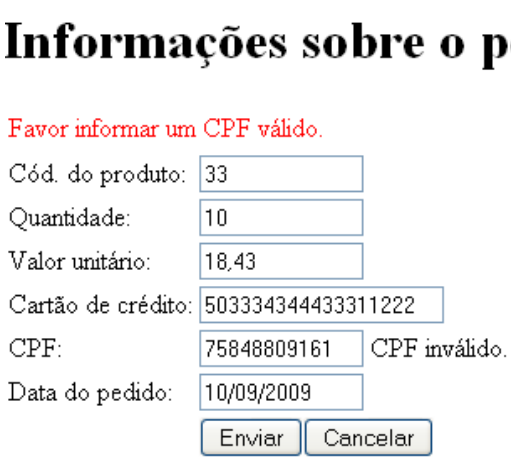
```
    <validator>
        <validator-id>com.algaworks.CPF</validator-id>
        <validator-class>
            com.algaworks.dwjsf.validadores.CPFValidator
        </validator-class>
    </validator>
```

Agora onde quisermos validar CPF, basta adicionarmos a tag `f:validator`, referenciando o ID do nosso validador.

```
    <h:outputText value="CPF:"/>
```

```
<h:panelGroup>
    <h:inputText id="cpf" size="12" value="#{pedidoBean.cpf}">
        <f:validator validatorId="com.algaworks.CPF"/>
    </h:inputText>
    <h:message for="cpf" showSummary="true" showDetail="false"/>
</h:panelGroup>
```

Um CPF inválido não é mais permitido a partir deste momento.



13. Manipulando eventos

13.1. Introdução

Qualquer aplicação que você desenvolver, por mais simples que seja, certamente precisará responder aos eventos do usuário, como cliques em botões, cliques em links, seleção de um item de um menu, alteração de um valor em um campo, etc.

Analisando rapidamente o modelo de eventos do JSF, notamos que se parece muito com o modelo usado em aplicações desktop. A diferença é que em JSF as ações do usuário acontecem no cliente (browser), que não possui uma conexão permanente com o servidor. Dessa forma, os eventos devem ser entregues ao servidor através de requisições HTTP para que ele processe a lógica de negócio referente ao evento.

O JSF possui mecanismos para tratar diferentes tipos de eventos. Neste capítulo, vamos estudar sobre eventos de ação e eventos de mudança de valor.

13.2. Capturando eventos de ação

Os eventos de ação são iniciados por componentes de comando, como `h:commandButton` e `h:commandLink`, quando os mesmos são ativados (clicados) pelo usuário. Estes eventos são executados na fase “Invocar a aplicação” do ciclo de vida, antes de renderizar a resposta ao cliente.

Os eventos de ação podem ser *listeners de ação* ou simplesmente *ações*. O primeiro tipo contribui para a navegação das páginas e não possui informações sobre o evento (como o componente que gerou a ação), já o segundo tipo não contribui para a navegação das páginas, porém possui informações sobre o evento. Os dois tipos podem trabalhar em conjunto em um mesmo componente, ou seja, pode-se incluir tratamento dos dois tipos de eventos para um mesmo botão.

Já usamos os dois tipos de eventos de ação em exemplos anteriores, por isso neste capítulo iremos apenas revisar a teoria de cada um.

Para criar um *listener de ação*, precisamos criar um método em um *managed bean* que recebe um objeto do tipo `javax.faces.event.ActionEvent`.

```
public void cadastrar(ActionEvent event) {
	...
}
```

Depois que o método estiver criado, vinculamos ele através do atributo `actionListener` de um componente de comando.

```
<h:commandButton value="Cadastrar"
	actionListener="#{nomeDoBean.cadastrar}"/>
```

Ao clicar no botão, o método `cadastrar()` é executado e a página reexibida.

Para criar uma ação que contribui para a navegação das páginas, precisamos criar um método em um *managed bean* sem nenhum parâmetro, porém com o retorno do tipo `String`. Este método deve retornar `null` para que a página atual seja reexibida ou um *outcome*, previamente configurado nas regras de navegação do arquivo “faces-config.xml”.

```
public String cadastrar() {
	return "umOutcomeQualquer";
}
```

O retorno deste método é usado para identificar qual será a próxima tela que o usuário irá visualizar. Dessa forma, é possível selecionar a próxima tela dinamicamente, dependendo da lógica executada.

Depois que o método estiver criado, vinculamo-lo através do atributo `action` de um componente de comando.

```
<h:commandButton value="Cadastrar"
      action="#{nomeDoBean.cadastrar}"/>
```

13.3. Capturando eventos de mudança de valor

Os eventos de mudança de valor são invocados quando valores de componentes de entrada são modificados e o formulário é submetido ao servidor. Para exemplificar este tipo de evento, criaremos uma tela simples com dois combos, sendo o primeiro com opções de unidades federativas (estados) e o segundo com cidades. O segundo combo será preenchido dinamicamente apenas após a seleção de uma UF.

Criamos uma classe chamada `EnderecoBean` com o código abaixo, que funcionará como nosso *managed bean*.

```
package com.algaworks.dwjsf.visao;

import java.util.ArrayList;
import java.util.List;

import javax.faces.component.UIInput;
import javax.faces.event.ActionEvent;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;

public class EnderecoBean {

      private List<SelectItem> estados =
            new ArrayList<SelectItem>();
      private List<SelectItem> cidades =
            new ArrayList<SelectItem>();
      private String estado;
      private String cidade;

      public EnderecoBean() {
          this.estados.add(new SelectItem(null, "Selecione"));
          this.estados.add(new SelectItem("MG", "Minas Gerais"));
          this.estados.add(new SelectItem("SP", "São Paulo"));
      }

      public void estadoAlterado(ValueChangeEvent event) {
            String novoEstado = (String)event.getNewValue();
            this.cidade = null;
            this.cidades.clear();
            this.cidades.add(new SelectItem(null, "Selecione"));

            if (novoEstado != null && novoEstado.equals("MG")) {
                  this.cidades.add(new SelectItem("UDI",
                        "Uberlândia"));
                  this.cidades.add(new SelectItem("BH",
                        "Belo Horizonte"));
            } else if (novoEstado != null
                  && novoEstado.equals("SP")) {
                  this.cidades.add(new SelectItem("SP",
                        "São Paulo"));
                  this.cidades.add(new SelectItem("RIB",
                        "Ribeirão Preto"));
                  this.cidades.add(new SelectItem("SJC",
```

```
                        "São José dos Campos"));
            }
        }

        public void enviar(ActionEvent event) {
            System.out.println("Estado selecionado: "
                    + this.getEstado() + " - Cidade selecionada: "
                    + this.getCidade());
        }

        public List<SelectItem> getEstados() {
            return estados;
        }
        public void setEstados(List<SelectItem> estados) {
            this.estados = estados;
        }
        public List<SelectItem> getCidades() {
            return cidades;
        }
        public void setCidades(List<SelectItem> cidades) {
            this.cidades = cidades;
        }
        public String getEstado() {
            return estado;
        }
        public void setEstado(String estado) {
            this.estado = estado;
        }
        public String getCidade() {
            return cidade;
        }
        public void setCidade(String cidade) {
            this.cidade = cidade;
        }

    }
```

Essa classe funcionará como um *managed bean*, por isso a registramos no "faces-config.xml".

```
<managed-bean>
    <managed-bean-name>enderecoBean</managed-bean-name>
    <managed-bean-class>
        com.algaworks.dwjsf.visao.EnderecoBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

No construtor da classe `EnderecoBean`, carregamos uma lista de itens de seleção com alguns estados:

```
public EnderecoBean() {
    this.estados.add(new SelectItem(null, "Selecione"));
    this.estados.add(new SelectItem("MG", "Minas Gerais"));
    this.estados.add(new SelectItem("SP", "São Paulo"));
}
```

Criamos um método chamado `estadoAlterado()`, que será invocado quando o estado for alterado pelo usuário.

```
	public void estadoAlterado(ValueChangeEvent event) {
		...
	}
```

Dentro do método `estadoAlterado()`, limpamos a lista de itens de seleção de cidades e carregamos uma nova lista, baseado no estado selecionado pelo usuário. Para descobrir qual estado foi escolhido, chamamos o método `getNewValue()` do objeto da classe `ValueChangeEvent`.

```
	String novoEstado = (String)event.getNewValue();
```

Neste momento, esta é a melhor forma de obter o valor selecionado pelo usuário, pois a fase "Atualizar os valores do modelo" ainda não foi processada, portanto o atributo `estado` do bean ainda está com o valor antigo.

O método `enviar()` simplesmente imprime o estado e a cidade selecionados pelo usuário.

```
	public void enviar(ActionEvent event) {
		System.out.println("Estado selecionado: "
			+ this.getEstado() + " - Cidade selecionada: "
			+ this.getCidade());
	}
```

Criamos um JSP chamado "endereco.jsp" para programar nossa tela:

```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<body>
<f:view>
	<h:form id="frm">
		<h:panelGrid columns="2" >
		   <h:outputText value="Estado:"/>
		   <h:selectOneMenu value="#{enderecoBean.estado}"
		       valueChangeListener="#{enderecoBean.estadoAlterado}"
		       onchange="submit();">
		       <f:selectItems value="#{enderecoBean.estados}"/>
		   </h:selectOneMenu>

		   <h:outputText value="Cidade:"/>
		   <h:selectOneMenu value="#{enderecoBean.cidade}">
		      <f:selectItems value="#{enderecoBean.cidades}"/>
		   </h:selectOneMenu>

		   <h:outputText/>
		   <h:commandButton value="Enviar"
		      actionListener="#{enderecoBean.enviar}"/>
		</h:panelGrid>
	</h:form>
</f:view>
</body>
</html>
```

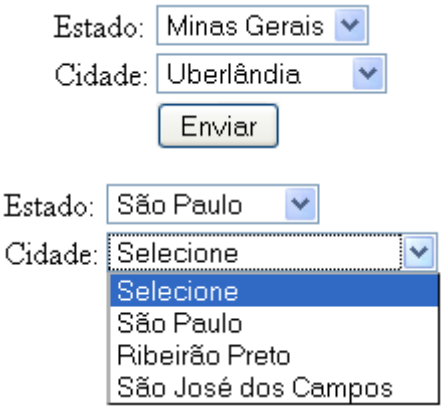
No código-fonte de nosso JSP, a única novidade é o uso do atributo `valueChangeListener` na tag `h:selectOneMenu`:

```
	<h:selectOneMenu value="#{enderecoBean.estado}"
		valueChangeListener="#{enderecoBean.estadoAlterado}"
		onchange="submit();">
```

```
            <f:selectItems value="#{enderecoBean.estados}"/>
        </h:selectOneMenu>
```

Incluímos um código JavaScript no evento `onchange`, que é executado *client-side* (no browser). Quando o usuário alterar a opção do menu de seleção, o formulário que contém o componente será submetido e o método `estadoAlterado()` será executado *server-side* (no servidor), pois vinculamos um *listener* de mudança de valor através do atributo `valueChangeListener`.

Agora podemos executar nossa tela e alternar entre os estados, que as cidades serão recarregadas automaticamente, após um pequeno *refresh* da página.



14. Sistema financeiro com JSF e Hibernate

14.1. Introdução

Agora que você já sabe bastante coisa sobre JSF, podemos desenvolver um sistema um pouco mais "completo" para demonstrar quase tudo que estudamos funcionando junto. Crie um novo projeto web no Eclipse chamado "Financeiro", inclua as bibliotecas do JavaServer Faces, configure o arquivo "web.xml" e certifique que um arquivo de modelo chamado "faces-config.xml" existe no diretório "WEB-INF".

Para criar um aplicativo um pouco mais interessante e real, precisamos ter persistência de dados. Mas afinal, o que é persistência de dados?

14.2. O que é persistência de dados?

A maioria dos sistemas desenvolvidos em uma empresa precisa de dados persistentes, portanto persistência é um conceito fundamental no desenvolvimento de aplicações. Se um sistema de informação não preservasse os dados quando ele fosse encerrado, o sistema não seria prático e usual.

Quando falamos de persistência de dados com Java, normalmente falamos do uso de sistemas gerenciadores de banco de dados relacionais e SQL, porém existem diversas alternativas para persistir dados, como em arquivos XML, arquivos texto e etc.

Nosso projeto de exemplo será desenvolvido usando um banco de dados MySQL, portanto podemos dizer que a persistência e dados será feita usando um banco de dados relacional.

14.3. Mapeamento objeto relacional (ORM)

Mapeamento objeto relacional (*object-relational mapping*, *ORM*, *O/RM* ou *O/R mapping*) é uma técnica de programação para conversão de dados entre banco de dados relacionais e linguagens de programação orientada a objetos.

Em banco de dados, entidades são representadas por tabelas, que possuem colunas que armazenam propriedades de diversos tipos. Uma tabela pode se associar com outras e criar relacionamentos diversos.

Em uma linguagem orientada a objetos, como Java, entidades são classes, e objetos dessas classes representam elementos que existem no mundo real. Por exemplo, um sistema de contas a pagar e receber possui a classe *Lancamento*, que no mundo real significa uma conta a pagar ou a receber, além de possuir uma classe se chama *Pessoa*, que pode ser o cliente ou fornecedor associado ao lançamento. Essas classes são chamadas de classes de domínio do sistema, pois fazem parte do negócio que está sendo desenvolvido.

Em banco de dados, podemos ter as tabelas "lancamento" e também "pessoa", mas a estrutura de banco de dados relacional está longe de ser orientado a objetos, e por isso a ORM foi inventada para suprir a necessidade que os desenvolvedores têm de visualizar tudo como objetos para programarem com mais facilidade.

Podemos comparar o modelo relacional com o modelo orientado a objetos conforme a tabela abaixo:

Modelo Relacional	Modelo OO
Tabela	Classe
Linha	Objeto
Coluna	Atributo
-	Método
Chave estrangeira	Associação

Essa comparação é feita em todo o tempo que se está desenvolvendo usando algum mecanismo de ORM. O mapeamento é feito usando metadados que descrevem a relação entre objetos e banco de dados.

Uma solução ORM consiste de uma API para executar operações CRUD simples em objetos de classes persistentes, uma linguagem ou API para especificar queries que se referem a classes e propriedades de classes, facilidades para especificar metadados de mapeamento e técnicas para interagir com objetos transacionais para identificarem automaticamente alterações realizadas, carregamento de associações por demanda e outras funções de otimização.

Em um ambiente ORM, as aplicações interagem com APIs e o modelo de classes de domínio e os códigos SQL/JDBC são abstraídos. Os comandos SQL são automaticamente gerados a partir dos metadados que relacionam objetos a banco de dados.

14.4. Hibernate e JPA

Hibernate é um framework de mapeamento objeto/relacional.

Antigamente, diversos programadores Java usavam os *entity beans* do EJB 2.1 como tecnologia para implementar a camada de persistência. Este modelo foi adotado largamente pela indústria de software.

Recentemente, especialistas da comunidade acharam que o modelo de desenvolvimento com EJB (principalmente os *entity beans*) deveria ser melhorado. A criadora do Java, Sun Microsystems, iniciou uma nova especificação para simplificar o EJB, que foi nomeada de Enterprise JavaBeans 3.0, conhecida também por JSR-220. Os desenvolvedores do Hibernate se tornaram parte da equipe de especislistas desta especificação e ajudaram nesta evolução, além de várias outras empresas e pessoas. Como o Hibernate já era um framework de persistência popular em todo o mundo por sua estabilidade e facilidade, vários conceitos foram aproveitados na especificação do EJB 3.0.

A especificação do EJB 3.0 possui várias partes: uma que define o modelo de programação EJB para *session beans*, *message-driven beans* e etc, outra parte, chamada de *Java Persistence API* (JPA), trata apenas de persistência, como entidades, mapeamento objeto/relacional e linguagem para consulta de dados (query).

Existem produtos que implementam um container completo EJB 3.0 e outros que implementam somente a parte de persistência.

As implementações de JPA devem ser plugáveis, o que significa que você poderá trocar um produto por outro sem afetar em nada seu sistema. Os produtos JPA devem também funcionar fora de um ambiente EJB 3.0 completo, ou seja, pode ser usado em aplicativos simples desenvolvidos em Java.

Consequentemente existem várias opções de produtos para empresas e desenvolvedores. A maioria dos produtos oferece funcionalidades que não fazem parte da especificação, como uma melhoria de desempenho ou uma API para consultas mais simples que a especificada. Você pode usar essas funcionalidades, porém deve-se lembrar que isso dificultará em uma futura migração de produto.

O Hibernate implementa a JPA, e aproveitando a flexibilidade que a especificação proporciona, você pode combinar diversos módulos do Hibernate:

- **Hibernate Core:** é a base para o funcionamento da persistência, com APIs nativas e metadados de mapeamentos gravados em arquivos XML. Possui uma linguagem de consultas chamada HQL (parecido com SQL), um conjunto de interfaces para consultas usando critérios (Criteria API) e queries baseadas em objetos exemplos.

- **Hibernate Annotations:** uma nova forma de definir metadados se tornou disponível graças ao Java 5.0. O uso de anotações dentro do código Java substituiu arquivos XML e tornou o desenvolvimento ainda mais fácil. O uso de Hibernate Annotations com Hibernate Core possibilita que os metadados sejam descritos dentro do código-fonte das classes de domínio em forma de anotações. A especificação JPA possui um conjunto de anotações para que o mapeamento objeto/relacional seja feito, e o Hibernate Annotations extende essas anotações para adicionar funcionalidades extras.

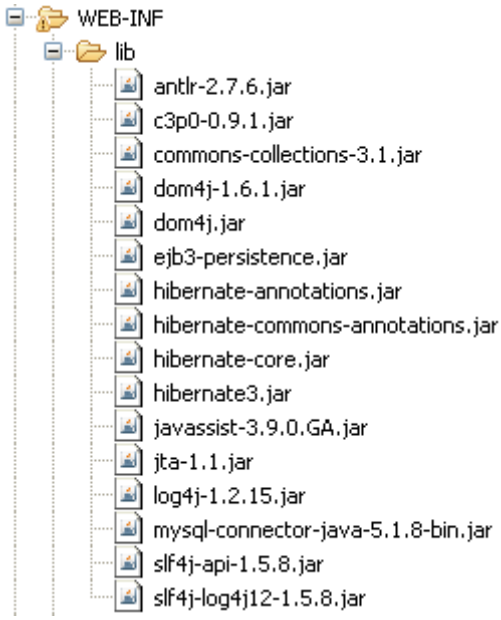
- **Hibernate EntityManager:** este módulo encapsula o Hibernate Core para fornecer compatibilidade com a JPA.

Aprenderemos apenas o básico do Hibernate neste curso. Se você gostaria de aprofundar mais, temos cursos específicos deste framework e também existem várias documentações disponíveis na internet.

14.5. Preparando o ambiente

Usaremos o Hibernate Core e Hibernate Annotations para desenvolver nosso projeto. Ambos estão disponíveis para download no próprio site do Hibernate (www.hibernate.org) e também no DVD do curso.

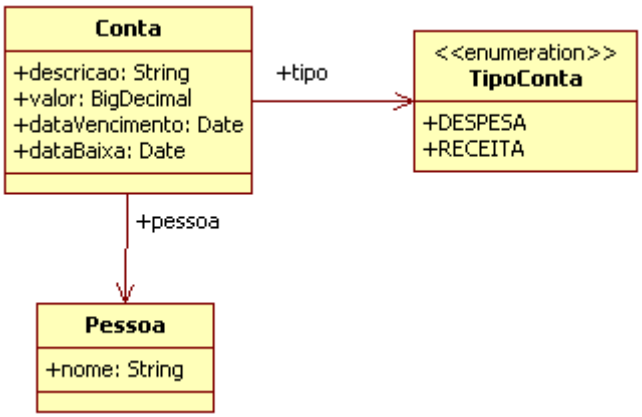
Para nosso projeto usar os recursos do Hibernate, precisamos incluir as bibliotecas deste framework e de suas dependências na pasta “WEB-INF/lib” do projeto.



Com exceção dos arquivos “log4j-1.2.15.jar”, “mysql-connector-java-5.1.8-bin.jar” e “slf4j-log4j12-1.5.8.jar”, todos são distribuídos através do Hibernate. Encontre os arquivos DVD e copie-os para seu diretório “lib”.

14.6. Nosso modelo de domínio

Nosso projeto será um sistema financeiro simples, que possibilitará lançamentos e consultas de contas a pagar e receber. Precisaremos de apenas duas entidades e uma enumeração.



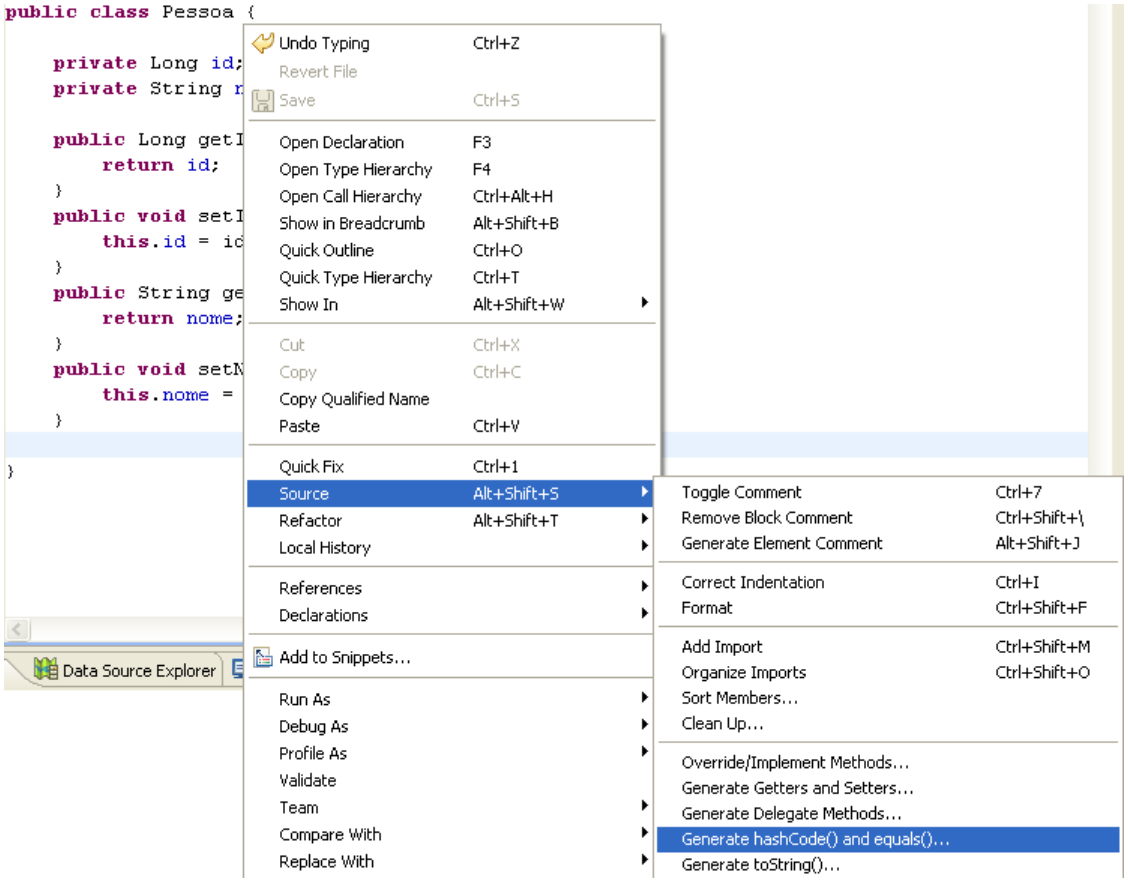
A classe `Pessoa` representará um cliente ou fornecedor a quem o usuário do sistema terá contas a receber ou pagar. A classe `Conta` representará uma conta a receber ou pagar, e a enumeração `TipoConta` representará apenas as constantes que qualificam uma conta como de "despesa" ou "receita".

14.7. Criando as classes de domínio

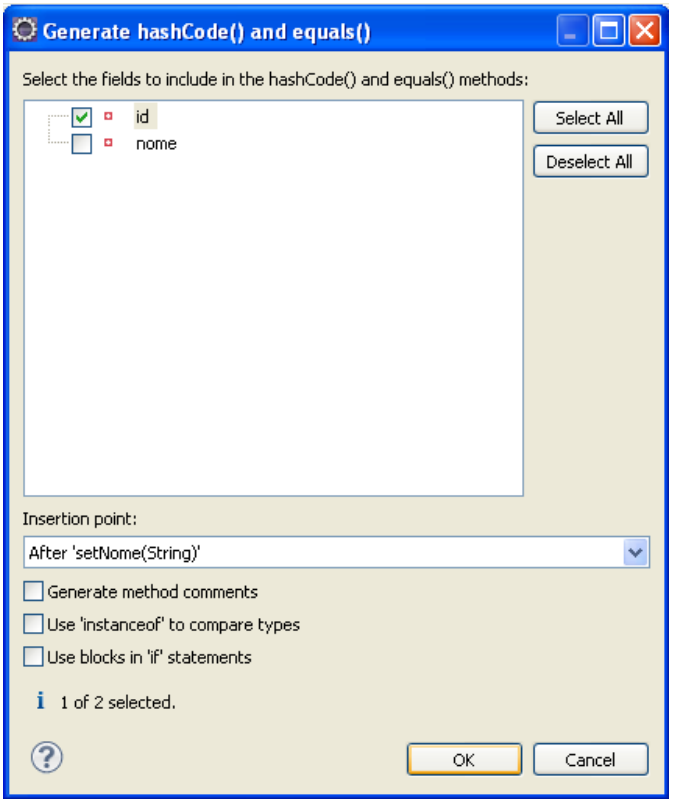
Com o diagrama de classes de domínio, fica fácil compreender o relacionamento entre as entidades. Vamos escrever primeiro o código-fonte da classe `Pessoa`.

```
package com.algaworks.dwjsf.financeiro.dominio;

public class Pessoa {

	private Long id;
	private String nome;

	public Long getId() {
		return id;
	}
	public void setId(Long id) {
		this.id = id;
	}
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}

}
```

A classe `Pessoa` é um bean que representa uma entidade no mundo real. Para finalizá-la, precisamos criar os métodos `equals()` e `hashCode()` baseados no atributo que é considerado como identificador da entidade. O Eclipse possui uma ferramenta que auxilia a geração desses métodos. Clique com o botão direito no código-fonte, selecione a opção "Source" e "Generate hashCode() and equals()...".



Na próxima tela, selecione apenas o atributo `id` (que representa o identificador da entidade) e clique no botão “OK”.



Agora o Eclipse irá gerar o código-fonte apropriado para estes métodos. Lembre-se de fazer isso em todas as entidades que você for criar.

```
@Override
public int hashCode() {
	final int prime = 31;
	int result = 1;
	result = prime * result + ((id == null) ? 0
		: id.hashCode());
	return result;
}
@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	Pessoa other = (Pessoa) obj;
	if (id == null) {
		if (other.id != null) return false;
	} else if (!id.equals(other.id)) return false;
	return true;
}
```

Precisamos criar a enumeração `TipoConta`, que identifica se uma conta é de despesa ou receita.

```
package com.algaworks.dwjsf.financeiro.dominio;

public enum TipoConta {

	DESPESA, RECEITA

}
```

Finalmente criamos a classe `Conta`, que representa um lançamento de uma conta a receber ou pagar (receita ou despesa). Algumas partes do código foram suprimidas.

```
package com.algaworks.dwjsf.financeiro.dominio;

import java.math.BigDecimal;
import java.util.Date;

public class Conta {

	private Long id;
	private Pessoa pessoa;
	private String descricao;
	private BigDecimal valor;
	private TipoConta tipo;
	private Date dataVencimento;
	private Date dataBaixa;

	//getters e setters

	//equals e hashCode

}
```

14.8. Criando as tabelas no banco de dados

Com as classes do modelo de domínio criadas, precisamos agora preparar as tabelas no nosso banco de dados. Usaremos o MySQL para nossos testes.

Conecte no servidor de banco de dados com um cliente MySQL (colocamos alguns no DVD deste curso) e execute a DDL (*Data Definition Language*) abaixo em um *schema* qualquer de testes. Se preferir, crie um novo *schema* chamado “financeiro”.

```
create table conta
(
   id                          bigint auto_increment          not null,
   pessoa_id                   bigint                         not null,
   descricao                   varchar(150)                   not null,
   valor                       decimal(10,2)                  not null,
   tipo                        varchar(10)                    not null,
   data_vencimento             date                           not null,
   data_baixa                  date,
   primary key (id)
);

create table pessoa
(
   id                          bigint auto_increment          not null,
   nome                        varchar(80)                    not null,
   primary key (id)
);

alter table conta add constraint fk_conta_pessoa
foreign key (pessoa_id) references pessoa (id);
```

14.9. Mapeando classes de domínio para tabelas do banco de dados

O Hibernate irá conectar com nosso banco de dados e executar todas as operações SQL necessárias para consultar, inserir, atualizar ou excluir registros nas tabelas, de acordo com o que precisarmos no momento. Para que tudo isso funcione, precisamos fazer o mapeamento objeto/relacional propriamente dito. Neste mapeamento, dizemos, por exemplo, que a classe `Conta` representa a tabela `conta`, o atributo `dataVencimento` representa a coluna `data_vencimento` do banco de dados, etc.

O mapeamento é feito através de metadados em um arquivo XML ou através de anotações do Java 5. Usaremos as anotações, que é o jeito mais elegante e produtivo.

O mapeamento da classe `Pessoa` é bastante simples. Incluímos as anotações `@Entity` e `@Table` na classe e `@Id` e `@GeneratedValue` no método `getId()`. Todas as anotações foram importadas do pacote `javax.persistence`.

```
@Entity
@Table(name="pessoa")
public class Pessoa {

	private Long id;
	private String nome;

	@Id
	@GeneratedValue
	public Long getId() {
		return id;
	}

	@Column(name="nome")
```

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

	...

}
```

A anotação `@Entity` diz que a classe é uma entidade e `@Table` especifica que a entidade representa uma tabela específica do banco de dados, descrita na propriedade `name`.

As anotações nos métodos *getters* configuram a relação dos atributos da classe com as colunas do banco de dados. As anotações `@Id` e `@GeneratedValue` são usadas para declarar o identificador do banco de dados, e esse identificador deve ter um valor gerado no momento de inserção (auto-incremento).

A anotação `@Column` especifica que um atributo deve referenciar uma coluna específica do banco de dados, descrita na propriedade `name`.

Agora é hora de mapear a classe `Conta`.

```
@Entity
@Table(name="conta")
public class Conta {

	private Long id;
	private Pessoa pessoa;
	private String descricao;
	private BigDecimal valor;
	private TipoConta tipo;
	private Date dataVencimento;
	private Date dataBaixa;

	@Id
	@GeneratedValue
	public Long getId() {
		return id;
	}

	@ManyToOne
	@JoinColumn(name="pessoa_id")
	public Pessoa getPessoa() {
		return pessoa;
	}

	public String getDescricao() {
		return descricao;
	}

	public BigDecimal getValor() {
		return valor;
	}

	@Enumerated(EnumType.STRING)
	public TipoConta getTipo() {
		return tipo;
	}

	@Column(name="data_vencimento")
	@Temporal(TemporalType.DATE)
	public Date getDataVencimento() {
		return dataVencimento;
```

```
	}

	@Column(name="data_baixa")
	@Temporal(TemporalType.DATE)
	public Date getDataBaixa() {
		return dataBaixa;
	}

	...

}
```

O método `getPessoa()` foi mapeado com as anotações `@ManyToOne` e `@JoinColumn`. A anotação `@ManyToOne` indica a multiplicidade do relacionamento entre contas e pessoas e a anotação `@JoinColumn` indica que essa relação é conseguida através da coluna especificada na propriedade `name`. Para facilitar o entendimento, esse mapeamento foi necessário para dizermos ao Hibernate que existe uma chave estrangeira na coluna `pessoa_id` da tabela `conta` que referencia a tabela `pessoa`.

Os *getters* `getDescricao()` e `getValor()` não foram mapeados explicitamente, mas isso não quer dizer que eles serão ignorados pelo Hibernate. Todos os *getters* sem mapeamentos explícitos são automaticamente reconhecidos e mapeados pelo Hibernate usando uma convenção de que o nome do atributo no código Java é o mesmo nome da coluna no banco de dados. No caso da descrição e valor da conta, isso é verdade, e, portanto podemos deixar o Hibernate mapeá-los automaticamente.

O método `getTipo()` foi mapeado com a anotação `@Enumerated` com `EnumType.STRING`. Isso é necessário para que o Hibernate entenda que na coluna do banco de dados será gravado o nome da constante da enumeração, e não o número que representa cada opção.

Já os *getters* `getDataVencimento()` e `getDataBaixa()` foram mapeados com `@Column` e `@Temporal`. A anotação `@Temporal` é útil para definir a precisão de colunas de data/hora. No caso das duas datas que mapeamos, dizemos que queremos armazenar apenas a data, e não data/hora.

14.10. Configurando o Hibernate

O Hibernate pode ser configurado de três formas: instanciando um objeto de configuração e inserindo propriedades programaticamente, incluindo um arquivo de propriedades com as configurações e informando os arquivos de mapeamento programaticamente ou usando um arquivo XML com as configurações e referências aos arquivos de mapeamento. Usaremos apenas a terceira opção, que é a mais usada e a melhor na maioria dos casos.

Crie um arquivo chamado “hibernate.cfg.xml” (que é o nome padrão) e coloque na raíz do diretório “src” de seu projeto.

Inclua o seguinte conteúdo dentro do arquivo XML:

```
<!DOCTYPE hibernate-configuration PUBLIC
   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
	<session-factory>
		<property name="connection.driver_class">
			com.mysql.jdbc.Driver
		</property>
		<property name="connection.url">
			jdbc:mysql://localhost/financeiro
		</property>
		<property name="dialect">
			org.hibernate.dialect.MySQL5Dialect
```

```
            </property>

            <property name="connection.username">usuario</property>
            <property name="connection.password">senha</property>

            <!-- Imprime os SQLs na saida padrao -->
            <property name="show_sql">false</property>
            <property name="format_sql">true</property>

            <!-- Pool de conexoes -->
            <property name="hibernate.c3p0.min_size">2</property>
            <property name="hibernate.c3p0.max_size">5</property>
            <property name="hibernate.c3p0.timeout">300</property>
            <property name="hibernate.c3p0.max_statements">
                  50
            </property>
            <property name="hibernate.c3p0.idle_test_period">
                  3000
            </property>

            <mapping
                  class="com.algaworks.dwjsf.financeiro.dominio.Pessoa"/>
            <mapping
                  class="com.algaworks.dwjsf.financeiro.dominio.Conta"/>
      </session-factory>
</hibernate-configuration>
```

Existem várias opções de configuração a serem informadas neste arquivo XML. Você pode ver todas estas opções na documentação do Hibernate. Vejamos as principais propriedades que usamos em nosso arquivo de configuração:

- **connection.driver_class:** nome completo da classe do driver JDBC.
- **connection.url:** descrição da URL de conexão com o banco de dados.
- **dialect:** dialeto a ser usado na construção de comandos SQL.
- **connection.username:** nome do usuário do banco de dados.
- **connection.password:** senha do usuário do banco de dados.
- **show_sql:** informa se os comandos SQL devem ser exibidos no console (importante para debug, mas deve ser desabilitado em ambiente de produção).
- **format_sql:** indica se os comandos SQL exibidos no console devem ser formatados (facilita a compreensão, mas pode gerar textos longos na saída padrão).

Na última parte do XML, indicamos as classes de entidades mapeadas que o Hibernate deve carregar antes de iniciar o trabalho.

Para evitar que conexões com o banco de dados sejam criadas e encerradas a cada ação dos usuários (aumentando o tempo de resposta), precisamos de um pool de conexões. Um pool é uma biblioteca que gerencia as conexões com o banco de dados do sistema de forma a atender todas as solicitações com o mínimo de tempo possível. Usaremos um provedor de pool chamado C3P0, e para que ele funcione, precisamos configurar algumas propriedades:

- **hibernate.c3p0.min_size:** número mínimo de conexões abertas com o banco de dados gerenciadas pelo pool.
- **hibernate.c3p0.max_size:** número máximo de conexões abertas com o banco de dados gerenciadas pelo pool.
- **hibernate.c3p0.timeout:** tempo máximo permitido para execução de comandos no banco de dados.
- **hibernate.c3p0.max_statements:** número máximo de comandos SQL que serão armazenados em cache. Isso é essencial para uma boa performance.

- **hibernate.c3p0.idle_test_period:** intervalo de tempo para que o pool teste se a conexão continua ativa.

14.11. Criando a classe `HibernateUtil`

Os sistemas que usam Hibernate precisam de uma instância da interface `SessionFactory` (e apenas uma), que é criada normalmente durante a inicialização da aplicação. Esta única instância pode ser usada por qualquer código que queira obter uma `Session`. A tradução de `SessionFactory` é "Fábrica de Sessão", e é exatamente esse o sentido.

Sessions ou sessões são representações de conexões com o banco de dados, que podem ser utilizadas para inserir, atualizar, excluir e pesquisar objetos persistentes.

A `SessionFactory` é *thread-safe* e por isso pode ser compartilhada, porém `Session` nunca deve ser compartilhada, pois não é um objeto *thread-safe*.

Como a instância de `SessionFactory` deve ser compartilhada, precisaremos de um lugar para colocá-la, onde qualquer código tenha acesso fácil e rápido. Uma forma de fazer isso é criando uma classe com o nome `HibernateUtil` (ou o nome que você desejar) que armazena essa instância em uma variável estática. Essa classe é bastante conhecida na comunidade de desenvolvedores Hibernate, pois é um padrão no desenvolvimento de aplicações com Java e Hibernate.

Veja abaixo um exemplo de implementação dessa classe:

```
package com.algaworks.dwjsf.financeiro.util;

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

public class HibernateUtil {

	private static SessionFactory sessionFactory;

	static {
		try {
			sessionFactory = new AnnotationConfiguration()
					.configure().buildSessionFactory();
		} catch (Throwable ex) {
			throw new ExceptionInInitializerError(ex);
		}
	}

	public static Session getSession() {
		return sessionFactory.openSession();
	}

}
```

Criamos um bloco estático para inicializar o Hibernate, instanciando a `SessionFactory`. Isso ocorrerá apenas uma vez, no carregamento da classe.

Agora, sempre que precisarmos de uma sessão, podemos chamar:

```
HibernateUtil.getSession()
```

14.12. Implementando classes de negócio

Antes de iniciar o desenvolvimento das telas do sistema, criaremos classes para tratar as regras de negócio e também um novo tipo de exceção que representa restrições de negócio.

A classe `RegraNegocioException` apenas herda a classe `Exception` e declara um construtor. Esta exceção será lançada sempre que uma regra de negócio for "quebrada".

```
package com.algaworks.dwjsf.financeiro.negocio;

public class RegraNegocioException extends Exception {

	private static final long serialVersionUID = 1L;

	public RegraNegocioException(String msg) {
		super(msg);
	}

}
```

A classe `PessoaService` foi criada com dois métodos apenas para pesquisa de pessoas. Usamos o método `get()` da sessão do Hibernate para obter um único objeto através do identificador e `createCriteria()` e depois `list()` para recuperar uma lista de objetos.

```
package com.algaworks.dwjsf.financeiro.negocio;

//importações

public class PessoaService {

	public Pessoa pesquisarPorId(Long id) {
		Session session = HibernateUtil.getSession();
		try {
			return (Pessoa) session.get(Pessoa.class, id);
		} finally {
			session.close();
		}
	}

	public List<Pessoa> listarTodas() {
		Session session = HibernateUtil.getSession();
		try {
			return session.createCriteria(Pessoa.class)
				.addOrder(Order.asc("nome")).list();
		} finally {
			session.close();
		}
	}

}
```

A classe `ContaService` foi criada com métodos para gravação e exclusão de objetos no banco de dados, pesquisa por identificador e listagem de todos os objetos do tipo `Conta`. Veja que alguns métodos desta classe lançam a exceção `RegraNegocioException` quando alguma restrição de negócio é encontrada.

```
package com.algaworks.dwjsf.financeiro.negocio;

//importações

public class ContaService {

	public void salvar(Conta conta) throws RegraNegocioException {
		if (conta.getValor().compareTo(BigDecimal.ZERO) <= 0) {
			throw new RegraNegocioException(
				"Valor da conta deve ser maior que zero.");
		}

		Session session = HibernateUtil.getSession();
		Transaction tx = session.beginTransaction();

		session.saveOrUpdate(conta);

		tx.commit();
		session.close();
	}

	public Conta pesquisarPorId(Long id) {
		Session session = HibernateUtil.getSession();
		try {
			return (Conta) session.get(Conta.class, id);
		} finally {
			session.close();
		}
	}

	@SuppressWarnings("unchecked")
	public List<Conta> listarTodas() {
		Session session = HibernateUtil.getSession();
		try {
			return session.createCriteria(Conta.class)
				.addOrder(Order.desc("dataVencimento")).list();
		} finally {
			session.close();
		}
	}

	public void excluir(Conta conta) throws RegraNegocioException {
		if (conta.getDataBaixa() != null) {
			throw new RegraNegocioException(
			  "Esta conta não pode ser excluída, pois já foi "
			  + "baixada!");
		}
		Session session = HibernateUtil.getSession();
		Transaction tx = session.beginTransaction();

		session.delete(conta);

		tx.commit();
		session.close();
	}

}
```

14.13. Criando uma folha de estilos e incluindo imagens

Para deixar nosso sistema com uma aparência agradável, podemos criar um arquivo de folha de estilos. Crie um novo arquivo chamado “estilo.css” e coloque na pasta “css” (dentro de “WebContent”). Crie esta pasta se ainda não existir.

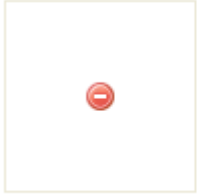
```
a {
	font: 12px Arial, sans-serif;
	font-weight: bold;
	color: #003399;
}

h1 {
	font: 18px Arial, sans-serif;
	font-weight: bold;
	color: #009900;
	text-decoration: underline;
}

.linkComEspaco {
	padding-right: 10px;
}

.imagemLink {
	border: none
}

.msgErro {
	font: 11px Arial, sans-serif;
	color: red;
}

.msgInfo {
	font: 11px Arial, sans-serif;
	color: blue;
}

.botao {
    background-color: #999999;
    color: #FFFFFF;
    border: 1px outset;
}

label {
	font: 11px Arial, sans-serif;
	font-weight: bold;
	color: #333333;
}


.tabela {
	border: 1px solid #666666;
}

.cabecalhoTabela {
	text-align: center;
	font: 11px Arial, sans-serif;
	font-weight: bold;
	color: white;
	background: #666666;
```

```
}

.linha1Tabela {
	font: 11px Arial, sans-serif;
	background: #DBDBDB;
}

.linha2Tabela {
	font: 11px Arial, sans-serif;
	background: #E6E6E6;
}

.colunaTipo {
	width: 20px;
	text-align: center;
}

.colunaPessoa {
	width: 200px;
}

.colunaDescricao {
}

.colunaValor {
	width: 100px;
	text-align: center;
}

.colunaVencimento {
	width: 100px;
	text-align: center;
}

.colunaAberta {
	width: 80px;
	text-align: center;
}

.colunaAcoes {
	width: 40px;
	text-align: center;
}
```

Usaremos alguns ícones para deixar o sistema mais amigável. Você pode utilizar os mesmos ou encontrar na internet ícones semelhantes. Coloque-os em uma pasta chamada “imagens”, dentro de “WebContent”.


despesa.png

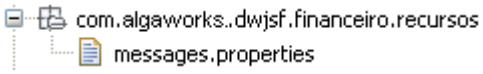

editar.png


excluir.png


receita.png

14.14. Traduzindo as mensagens padrão do JSF

Vamos traduzir algumas mensagens do JSF também para deixar o sistema mais amigável. Crie um pacote chamado `com.algaworks.dwjsf.financeiro.recursos` e depois crie um arquivo "messages.properties".



Coloque o seguinte conteúdo no arquivo "messages.properties":

```
javax.faces.converter.DateTimeConverter.DATE=Data inválida.
javax.faces.converter.DateTimeConverter.DATE_detail={2} não foi
informado com uma data válida.
javax.faces.converter.NumberConverter.NUMBER=Valor inválido.
javax.faces.converter.NumberConverter.NUMBER_detail={2} não foi
informado com um valor válido.
javax.faces.component.UIInput.REQUIRED={0} é obrigatório.
```

Referencie o *messages bundle* no arquivo "faces-config.xml".

```
<application>
    <message-bundle>
       com.algaworks.dwjsf.financeiro.recursos.messages
    </message-bundle>
</application>
```

14.15. Configurando *managed beans* e regras de navegação

Criaremos 2 *managed beans* e 3 telas JSF. Antes de ver o código-fonte deles, deixe-os configurados no arquivo "faces-config.xml", incluindo o seguinte:

```
<managed-bean>
      <managed-bean-name>cadastroContaBean</managed-bean-name>
      <managed-bean-class>
            com.algaworks.dwjsf.financeiro.visao.CadastroContaBean
      </managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
      <managed-bean-name>consultaContaBean</managed-bean-name>
      <managed-bean-class>
            com.algaworks.dwjsf.financeiro.visao.ConsultaContaBean
      </managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<navigation-rule>
      <navigation-case>
            <from-outcome>cadastroConta</from-outcome>
            <to-view-id>/contas/cadastroConta.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>consultaConta</from-outcome>
            <to-view-id>/contas/consultaConta.jsp</to-view-id>
      </navigation-case>
</navigation-rule>
```

```
<navigation-rule>
	<navigation-case>
		<from-outcome>menu</from-outcome>
		<to-view-id>/menu.jsp</to-view-id>
		<redirect/>
	</navigation-case>
</navigation-rule>
```

14.16. Conversor genérico para tipo `Enum`

Na tela de cadastro de conta, incluiremos um menu de seleção de tipo de conta (despesa ou receita). Para que a conversão desse tipo seje transparente de objeto para string e de string para objeto, precisamos criar um conversor. A classe abaixo é o código-fonte de um conversor genérico para enumerações, e é muito útil em qualquer projeto que você for desenvolver.

```
package com.algaworks.dwjsf.financeiro.conversores;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;

public class EnumConverter implements Converter {

	public Object getAsObject(FacesContext context,
			UIComponent component, String value)
			throws ConverterException {
		Class enumType = component.getValueExpression("value")
			.getType(context.getELContext());
		return Enum.valueOf(enumType, value);
	}

	public String getAsString(FacesContext context,
			UIComponent component, Object object)
			throws ConverterException {
		if (object == null) {
			return null;
		}
		Enum type = (Enum) object;
		return type.toString();
	}

}
```

Em Java, toda enumeração herda implicitamente da classe `Enum`. Para deixar nosso conversor genérico e pronto para qualquer enumeração que possa surgir no futuro em nosso projeto, registramo-lo para o tipo genérico `java.lang.Enum` no arquivo “faces-config.xml”.

```
<converter>
	<converter-for-class>java.lang.Enum</converter-for-class>
	<converter-class>
	    com.algaworks.dwjsf.financeiro.conversores.EnumConverter
	</converter-class>
</converter>
```

14.17. Conversor para entidade `Pessoa`

Na tela de cadastro de conta, incluiremos um menu de seleção de pessoas. Novamente precisamos e um conversor, pois a classe `Pessoa` não possui um conversor padrão, como existem para os tipos primitivos. Crie o conversor da classe `Pessoa` usando o código-fonte abaixo.

```
package com.algaworks.dwjsf.financeiro.conversores;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;

import com.algaworks.dwjsf.financeiro.dominio.Pessoa;
import com.algaworks.dwjsf.financeiro.negocio.PessoaService;

public class PessoaConverter implements Converter {

	public Object getAsObject(FacesContext context,
			UIComponent component, String value)
			throws ConverterException {
		if (value == null) {
			return null;
		}
		return new PessoaService()
			.pesquisarPorId(Long.parseLong(value));
	}

	public String getAsString(FacesContext context,
			UIComponent component, Object object)
			throws ConverterException {
		if (object == null) {
			return null;
		}
		Pessoa pessoa = (Pessoa) object;
		return pessoa.getId().toString();
	}

}
```

Veja que o método `getAsObject()`, que retorna um objeto convertido a partir de uma string necessita da classe de negócio `PessoaService` para consultar a pessoa pelo identificador.

Registre o conversor no arquivo “faces-config.xml”.

```
<converter>
    <converter-for-class>
        com.algaworks.dwjsf.financeiro.dominio.Pessoa
    </converter-for-class>
    <converter-class>
        com.algaworks.dwjsf.financeiro.conversores.PessoaConverter
    </converter-class>
</converter>
```

14.18. Construindo uma tela de menu do sistema

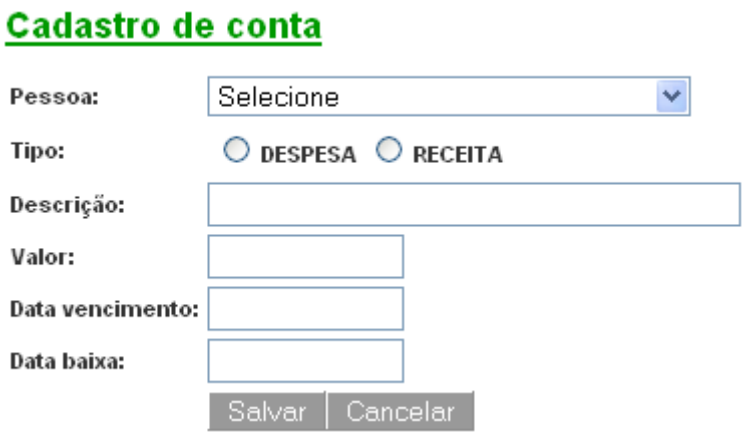
Criamos uma tela principal muito simples que guiará o usuário através de alguns links. Essa tela se parecerá com o seguinte:

Coloque o código em um arquivo chamado “menu.jsp”, em “WebContent”.

```
<%@ page contentType="text/html; charset=ISO-8859-1"
         pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<head>
   <title>Sistema Financeiro</title>
   <link rel="stylesheet" type="text/css" href="../css/estilo.css" />
</head>
<body>
<f:view>
      <h:form id="frm">
            <h1><h:outputText value="Sistema Financeiro"/></h1>

            <h:commandLink action="#{cadastroContaBean.inicializar}">
                  <h:outputText value="Cadastro de contas"/>
            </h:commandLink>
            <br/>
            <h:commandLink action="consultaConta"
                  actionListener="#{consultaContaBean.consultar}">
                  <h:outputText value="Consulta de contas"/>
            </h:commandLink>
      </h:form>
</f:view>
</body>
</html>
```

14.19. Construindo a tela para cadastro de contas

Agora iremos criar uma tela para cadastro de conta, que terá a seguinte aparência:



Crie um arquivo chamado “cadastroConta.jsp” em uma pasta “contas”, em “WebContent”, e coloque o código-fonte a seguir:

```
<%@ page contentType="text/html; charset=ISO-8859-1"
         pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<head>
    <title>Cadastro de conta</title>
    <link rel="stylesheet" type="text/css" href="../css/estilo.css" />
</head>
<body>
<f:view>
      <h:form id="frm">
            <h1><h:outputText value="Cadastro de conta"/></h1>

            <h:messages layout="table" showSummary="true"
                  showDetail="false" globalOnly="true"
                  styleClass="msgErro" infoClass="msgInfo"
                  style="font-weight: bold"/>

            <h:panelGrid columns="2">
                  <h:outputLabel value="Código:"
                     rendered="#{cadastroContaBean.contaEdicao.id
                                 != null}"/>
                  <h:panelGroup
                        rendered="#{cadastroContaBean.contaEdicao.id
                                    != null}">
                        <h:inputText id="codigo" size="10"
                          value="#{cadastroContaBean.contaEdicao.id}"
                          label="Código da conta" disabled="true"/>
                        <h:message for="codigo" showSummary="true"
                              showDetail="false" styleClass="msgErro"/>
                  </h:panelGroup>

                  <h:outputLabel value="Pessoa:"/>
                  <h:panelGroup>
                        <h:selectOneMenu id="pessoa"
                        value="#{cadastroContaBean.contaEdicao.pessoa}"
                        label="Pessoa" required="true">
                              <f:selectItems
                                 value="#{cadastroContaBean.pessoas}"/>
                        </h:selectOneMenu>
                        <h:message for="pessoa" showSummary="true"
                              showDetail="false" styleClass="msgErro"/>
                  </h:panelGroup>

                  <h:outputLabel value="Tipo:"/>
                  <h:panelGroup>
                        <h:selectOneRadio id="tipo"
                          value="#{cadastroContaBean.contaEdicao.tipo}"
                          label="Tipo da conta" required="true">
                          <f:selectItems
                            value="#{cadastroContaBean.tiposLancamentos}"/>
                        </h:selectOneRadio>
                        <h:message for="tipo" showSummary="true"
                              showDetail="false" styleClass="msgErro"/>
                  </h:panelGroup>
```

```
                    <h:outputLabel value="Descrição:"/>
                    <h:panelGroup>
                      <h:inputText id="descricao" size="40"
                       maxlength="150"
                       value="#{cadastroContaBean.contaEdicao.descricao}"
                       required="true" label="Descrição"/>
                      <h:message for="descricao" showSummary="true"
                          showDetail="false" styleClass="msgErro"/>
                    </h:panelGroup>

                    <h:outputLabel value="Valor:"/>
                    <h:panelGroup>
                          <h:inputText id="valor" size="12"
                           maxlength="10"
                           value="#{cadastroContaBean.contaEdicao.valor}"
                           required="true" label="Valor">
                                <f:convertNumber minFractionDigits="2"/>
                          </h:inputText>
                          <h:message for="valor" showSummary="true"
                                showDetail="false" styleClass="msgErro"/>
                    </h:panelGroup>

                    <h:outputLabel value="Data vencimento:"/>
                    <h:panelGroup>
                      <h:inputText id="dataVencimento" size="12"
                          maxlength="10"
                          value="#{cadastroContaBean.contaEdicao.dataVenc
                          imento}" required="true"
                          label="Data vencimento">
                                <f:convertDateTime pattern="dd/MM/yyyy"/>
                       </h:inputText>
                       <h:message for="dataVencimento" showSummary="true"
                          showDetail="false" styleClass="msgErro"/>
                    </h:panelGroup>

                    <h:outputLabel value="Data baixa:"/>
                    <h:panelGroup>
                      <h:inputText id="dataBaixa"  size="12"
                       maxlength="10"
                       value="#{cadastroContaBean.contaEdicao.dataBaixa}"
                       label="Data baixa">
                                <f:convertDateTime pattern="dd/MM/yyyy"/>
                      </h:inputText>
                      <h:message for="dataBaixa" showSummary="true"
                          showDetail="false" styleClass="msgErro"/>
                    </h:panelGroup>

                    <h:panelGroup/>
                    <h:panelGroup>
                          <h:commandButton value="Salvar"
                             actionListener="#{cadastroContaBean.salvar}"
                             styleClass="botao"/>
                          <h:commandButton value="Cancelar" action="menu"
                             immediate="true" styleClass="botao"/>
                    </h:panelGroup>
              </h:panelGrid>
        </h:form>
</f:view>
</body>
</html>
```

Código-fonte da classe do *managed bean*:

```
package com.algaworks.dwjsf.financeiro.visao;

//importações

public class CadastroContaBean {

	private Conta contaEdicao;
	private List<SelectItem> tiposContas;
	private List<SelectItem> pessoas;

	public String inicializar() {
		this.contaEdicao = new Conta();
		this.tiposContas = null;
		this.pessoas = null;
		return "cadastroConta";
	}

	public void salvar(ActionEvent event) {
		FacesContext context = FacesContext.getCurrentInstance();
		try {
			new ContaService().salvar(this.contaEdicao);
			this.contaEdicao = new Conta();
			FacesMessage msg = new FacesMessage(
				"Conta salva com sucesso!");
			msg.setSeverity(FacesMessage.SEVERITY_INFO);
			context.addMessage(null, msg);
		} catch (RegraNegocioException e) {
			context.addMessage(null,
				new FacesMessage(FacesMessage.SEVERITY_ERROR,
					e.getMessage(), e.getMessage()));
		} catch (Exception e) {
			e.printStackTrace();
			FacesMessage msg = new FacesMessage(
					"Erro inesperado ao salvar conta!");
			msg.setSeverity(FacesMessage.SEVERITY_ERROR);
			context.addMessage(null, msg);
		}
	}

	public List<SelectItem> getPessoas() {
		if (this.pessoas == null) {
			this.pessoas = new ArrayList<SelectItem>();
			List<Pessoa> pessoas = new PessoaService()
				.listarTodas();
			this.pessoas.add(new SelectItem(null, "Selecione"));
			for (Pessoa pessoa : pessoas) {
				this.pessoas.add(new SelectItem(pessoa,
					pessoa.getNome()));
			}
		}
		return this.pessoas;
	}

	public List<SelectItem> getTiposLancamentos() {
		if (this.tiposContas == null) {
			this.tiposContas = new ArrayList<SelectItem>();
			for (TipoConta tipo : TipoConta.values()) {
				this.tiposContas.add(new SelectItem(tipo,
```

```
                                    tipo.toString()));
                        }
                  }
                  return tiposContas;
            }

            public Conta getContaEdicao() {
                  return contaEdicao;
            }

            public void setContaEdicao(Conta contaEdicao) {
                  this.contaEdicao = contaEdicao;
            }

}
```

14.20. Construindo a tela para consulta de contas

Por último, vamos criar uma tela para consulta de contas, que terá a seguinte aparência:

Consulta de contas

Tipo	Pessoa	Descrição	Valor	Vencimento	Aberta	Ações
	Condomínio Ed. Jardim das Couves	Condomínio	R$ 200,00	10/10/2009	Sim	
	Escola Infantil Abelha Rainha	Escola filho	R$ 450,00	09/10/2009	Sim	
	Prefeitura de Uberlândia	Salário	R$ 1.000,00	08/10/2009	Não	
	Udi Imobiliária	Aluguel do AP	R$ 600,00	01/10/2009	Sim	
	Sebastião Costa Silva	Venda notebook	R$ 900,00	28/09/2009	Não	
	Supermercado Andrade	Compras	R$ 234,35	20/09/2009	Não	

Nova conta Menu do sistema

Crie um arquivo chamado "consultaConta.jsp" em uma pasta "contas", em "WebContent", e coloque o código-fonte a seguir:

```
<%@ page contentType="text/html; charset=ISO-8859-1"
         pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>
<head>
    <title>Consulta de contas</title>
    <link rel="stylesheet" type="text/css" href="../css/estilo.css" />
</head>
<body>
<f:view>
      <h:form id="frm">
            <h1><h:outputText value="Consulta de contas"/></h1>

            <h:messages layout="table" showSummary="true"
                  showDetail="false" globalOnly="true"
                  styleClass="msgErro" infoClass="msgInfo"
                  style="font-weight: bold"/>

            <h:dataTable value="#{consultaContaBean.contas}" var="item"
                  width="790px" styleClass="tabela"
                  headerClass="cabecalhoTabela"
                  rowClasses="linha1Tabela, linha2Tabela"
                  columnClasses="colunaTipo, colunaPessoa,
```

```
        colunaDescricao, colunaValor, colunaVencimento,
        colunaAberta, colunaAcoes">
    <h:column>
        <f:facet name="header">
            <h:outputText value="Tipo"/>
        </f:facet>
        <h:graphicImage value="/imagens/receita.png"
          title="Conta a receber"
          rendered="#{item.tipo eq 'RECEITA'}"/>
        <h:graphicImage value="/imagens/despesa.png"
          title="Conta a pagar"
          rendered="#{item.tipo eq 'DESPESA'}"/>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Pessoa"/>
        </f:facet>
        <h:outputText value="#{item.pessoa.nome}"/>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Descrição"/>
        </f:facet>
        <h:outputText value="#{item.descricao}"/>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Valor"/>
        </f:facet>
        <h:outputText value="#{item.valor}"
          style="color: #{item.tipo eq 'RECEITA' ?
              'blue' : 'red'}">
              <f:convertNumber minFractionDigits="2"
                  currencyCode="BRL" type="currency"/>
        </h:outputText>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Vencimento"/>
        </f:facet>
        <h:outputText value="#{item.dataVencimento}">
            <f:convertDateTime pattern="dd/MM/yyyy"/>
        </h:outputText>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Aberta"/>
        </f:facet>
        <h:outputText value="#{item.dataBaixa == null ?
                        'Sim' : 'Não'}"/>
    </h:column>
    <h:column>
        <f:facet name="header">
            <h:outputText value="Ações"/>
        </f:facet>
        <h:commandLink
            action="#{consultaContaBean.excluir}">
            <f:setPropertyActionListener
                value="#{item}"
                target="#{consultaContaBean
```

```
                                                .contaExclusao}"/>
                                    <h:graphicImage
                                       value="/imagens/excluir.png"
                                       title="Excluir"
                                       styleClass="imagemLink"/>
                                </h:commandLink>
                                <h:commandLink action="cadastroConta">
                                      <f:setPropertyActionListener
                                            value="#{item}"
                                            target="#{cadastroContaBean
                                                  .contaEdicao}"/>
                                      <h:graphicImage
                                       value="/imagens/editar.png"
                                       title="Editar" styleClass="imagemLink"/>
                                </h:commandLink>
                          </h:column>
                    </h:dataTable>

                    <br/>

                    <h:commandLink action="#{cadastroContaBean.inicializar}"
                                styleClass="linkComEspaco">
                          <h:outputText value="Nova conta"/>
                    </h:commandLink>
                    <h:commandLink action="menu">
                          <h:outputText value="Menu do sistema"/>
                    </h:commandLink>
              </h:form>
</f:view>
</body>
</html>
```

Código-fonte da classe do *managed bean*:

```
package com.algaworks.dwjsf.financeiro.visao;

//importações

public class ConsultaContaBean {

      private Conta contaExclusao;
      private List<Conta> contas = new ArrayList<Conta>();

      public void consultar(ActionEvent event) {
            this.contas = new ContaService().listarTodas();
      }

      public String excluir() {
            FacesContext context = FacesContext.getCurrentInstance();
            try {
                  new ContaService().excluir(this.contaExclusao);
                  this.contas.remove(this.contaExclusao);

                  this.contaExclusao = null;
                  FacesMessage msg = new FacesMessage(
                              "Conta excluída com sucesso!");
                  msg.setSeverity(FacesMessage.SEVERITY_INFO);
                  context.addMessage(null, msg);
            } catch (RegraNegocioException e) {
                  context.addMessage(null, new FacesMessage(
```

```
                                    FacesMessage.SEVERITY_ERROR,
                                    e.getMessage(), e.getMessage()));
            } catch (Exception e) {
                  e.printStackTrace();
                  FacesMessage msg = new FacesMessage(
                              "Erro inesperado ao excluir conta!");
                  msg.setSeverity(FacesMessage.SEVERITY_ERROR);
                  context.addMessage(null, msg);
            }

            return null;
      }

      public List<Conta> getContas() {
            return contas;
      }

      public Conta getContaExclusao() {
            return contaExclusao;
      }

      public void setContaExclusao(Conta contaExclusao) {
            this.contaExclusao = contaExclusao;
      }

}
```

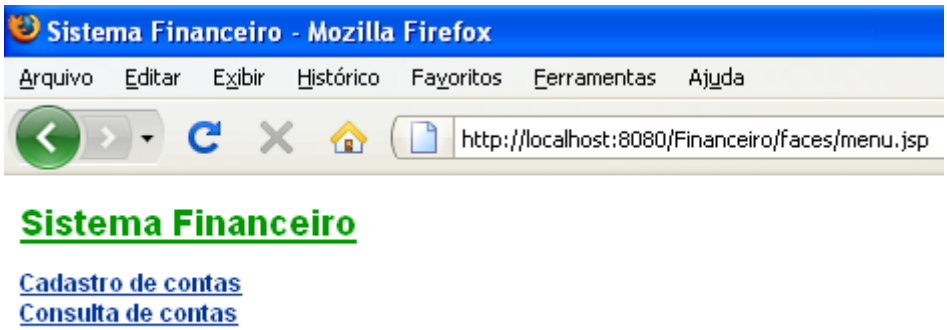
14.21. Um sistema financeiro funcionando!

Para finalizar, crie um arquivo “index.jsp” na pasta “WebContent” com o código a seguir:

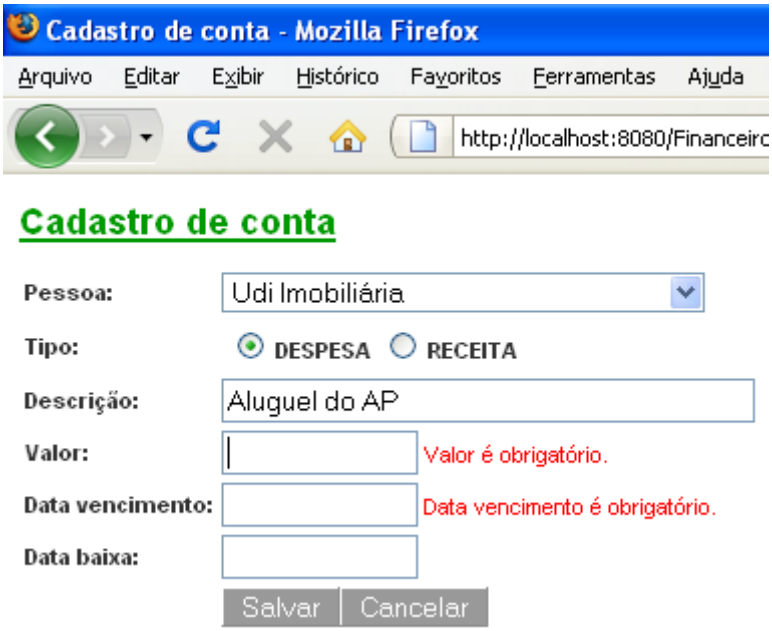
```
<%
      response.sendRedirect("faces/menu.jsp");
%>
```

Agora adicione o projeto no Tomcat, inicie o servidor e acesse seguinte endereço no browser: http://localhost:8080/Financeiro.

Na tela principal, você pode escolher se deseja cadastrar uma nova conta ou consultar contas existentes.



Na tela de cadastro de conta, todos os campos são obrigatórios, com exceção da data da baixa. Tente cadastrar contas sem preencher todas as informações ou informe dados incorretos para testar os conversores e as mensagens personalizadas.



Na tela de consulta de contas, ícones identificam se a conta é de despesa ou receita, a tabela é apresentada com "cor sim – cor não" e ainda há possibilidade e excluir ou editar algum registro, através de ícones de ações.

Teste a exclusão e alteração de contas. Veja que não é possível excluir contas fechadas (com data baixa informada). Neste caso, uma mensagem sobre a restrição é informada no topo da tabela.

15. Data Access Object

15.1. Introdução

DAO, ou Data Access Object, é um padrão de projeto centralizador de operações persistentes. Quando dizemos "operações persistentes", estamos nos referindo ao acesso a qualquer dado "eternizável", ou seja, dados armazenáveis em algum local, seja banco de dados, arquivo XML, arquivo texto e etc.

O DAO pode ser considerado também como um tradutor. Ele abstrai a origem e o modo de obtenção/gravação dos dados, de modo que o restante do sistema manipula os dados de forma transparente, sem se preocupar com o que acontece nos bastidores. Isso ajuda muito em processos de migrações de fonte de dados.

É muito comum vermos em códigos de programadores iniciantes a base de dados sendo acessada em diversos pontos da aplicação, de maneira extremamente explícita e repetitiva. O padrão DAO tenta resolver esse problema abstraindo e encapsulando todo o acesso a dados.

Normalmente, temos um DAO para cada classe de domínio do sistema. No projeto do sistema financeiro, estudado no último capítulo, tínhamos duas classes de domínio, Pessoa e Conta.

Cada DAO deve possuir uma interface, que especifica os métodos de manipulação de dados e uma classe concreta, que implementa realmente a obtenção/gravação dos dados diretamente na fonte de dados.

Com o surgimento de tecnologias como JPA e Hibernate, o padrão DAO começou a ser questionado por alguns especialistas, pois pode parecer um trabalho extra desnecessário, já que essas novas tecnologias abstraem o acesso ao banco de dados. Não existe certo ou errado, por isso você deve avaliar o que é melhor para você, para seu projeto e principalmente para sua empresa.

15.2. Criando um DAO genérico

Antes do surgimento do Java 5, costumávamos criar uma classe e uma interface DAO para cada classe de domínio, e implementávamos todos os métodos CRUD (*create*, *read*, *update* e *delete*) e outros que fosse necessário. O resultado disso era bastante código repetitivo. Por exemplo, a entidade Pessoa teria um DAO com métodos para salvar, listar, excluir e etc, a entidade Conta também teria os mesmos métodos para salvar, listar e excluir, porém referenciando outra tabela/entidade no banco de dados.

A partir do Java 5, diversos desenvolvedores na comunidade publicaram versões do que chamaram de "DAO genérico", graças ao poder de *generics*, incorporado na plataforma a partir da versão 1.5.

O DAO genérico possibilita uma economia significativa de código e trabalho repetitivo, e por isso criaremos uma versão simplificada para nosso projeto.

Para começar, criamos uma interface chamada `GenericDAO`, que define alguns métodos que um DAO genérico deve possuir.

```
package com.algaworks.dwjsf.financeiro.dao;

//importações

public interface GenericDAO<T> {

	public Session getSession();
	public T pesquisarPorId(Serializable id);
	public List<T> listarTodos();
	public T salvar(T entidade);
	public void excluir(T entidade);

}
```

Depois criamos uma classe que implementa a interface `GenericDAO`, chamada `HibernateGenericDAO`.

```
package com.algaworks.dwjsf.financeiro.dao.hibernate;

//importações

public abstract class HibernateGenericDAO<T>
            implements GenericDAO<T> {

      private Class<T> persistentClass;
      private Session session;

      public HibernateGenericDAO(Session session) {
            this.session = session;
            this.persistentClass = (Class<T>)((ParameterizedType)
                  getClass().getGenericSuperclass())
                  .getActualTypeArguments()[0];
      }

      public Session getSession() {
            return this.session;
      }

      public T pesquisarPorId(Serializable id) {
            return (T)this.session.get(this.persistentClass, id);
      }

      public List<T> listarTodos() {
            return this.session.createCriteria(this.persistentClass)
                  .list();
      }

      public T salvar(T entidade) {
            return (T)this.session.merge(entidade);
      }

      public void excluir(T entidade) {
            this.session.delete(entidade);
      }

}
```

Neste momento temos um DAO genérico, que pode ser usado para executar operações persistentes básicas de qualquer entidade.

O uso de *generics* do Java 5 foge do escopo deste curso. Existe vasta documentação na internet sobre o assunto. A AlgaWorks também possui um curso específico sobre as novidades do Java 5 que aborda este tema a fundo.

15.3. Criando DAOs específicos

Vamos criar DAOs específicos para as entidades `Pessoa` e `Conta`. Cada uma delas terá o seu próprio DAO responsável por realizar operações persistentes.

Iniciamos pela entidade `Pessoa`, criando uma interface que herda `GenericDAO`.

```
package com.algaworks.dwjsf.financeiro.dao;

import java.util.List;
```

```
import com.algaworks.dwjsf.financeiro.dominio.Pessoa;

public interface PessoaDAO extends GenericDAO<Pessoa> {

	public List<Pessoa> listarTodosOrdenadoPorNome();

}
```

Essa interface é uma especialização de `GenericDAO`. Veja que definimos que o tipo da entidade é `Pessoa`, quando herdamos `GenericDAO<Pessoa>`. Além disso, incluímos outro método a ser implementado pela classe concreta, pois apenas os métodos do DAO genérico não seriam suficientes.

Criamos uma classe chamada `HibernatePessoaDAO` que herda as implementações do DAO genérico para Hibernate (class `HibernateGenericDAO`) e implementa a interface `PessoaDAO`, que é uma interface que define métodos genéricos e específicos.

```
package com.algaworks.dwjsf.financeiro.dao.hibernate;

//importações

public class HibernatePessoaDAO extends HibernateGenericDAO<Pessoa>
		implements PessoaDAO {

	public HibernatePessoaDAO(Session session) {
		super(session);
	}

	public List<Pessoa> listarTodosOrdenadoPorNome() {
		return this.getSession().createCriteria(Pessoa.class)
				.addOrder(Order.asc("nome")).list();
	}

}
```

Agora vamos criar a interface DAO para a entidade `Conta`.

```
package com.algaworks.dwjsf.financeiro.dao;

import java.util.List;

import com.algaworks.dwjsf.financeiro.dominio.Conta;

public interface ContaDAO extends GenericDAO<Conta> {

	public List<Conta> listarTodasOrdenadasPelasMaisAntigas();

}
```

E a classe DAO para a mesma entidade.

```
package com.algaworks.dwjsf.financeiro.dao.hibernate;

//importações

public class HibernateContaDAO extends HibernateGenericDAO<Conta>
		implements ContaDAO {

	public HibernateContaDAO(Session session) {
		super(session);
	}
```

```
	public List<Conta> listarTodasOrdenadasPelasMaisAntigas() {
		return this.getSession().createCriteria(Conta.class)
			.addOrder(Order.desc("dataVencimento")).list();
	}

}
```

15.4. Criando uma fábrica de DAOs

Apesar de termos a interfaces DAO, não há nenhuma vantagem se instanciarmos DAOs desta maneira:

```
Session session = HibernateUtil.getSession();
PessoaDAO dao = new HibernatePessoaDAO(session);
```

Dessa forma não estaríamos abstraindo os DAOs, e perderíamos o maior benefício em utilizar interfaces. O melhor a ser feito é utilizar o padrão de projeto *Factory*.

O padrão *Factory* implementa uma fábrica de objetos, abstraindo e isolando a forma de criação dos objetos.

É possível abstrair também a fábrica de objetos, e desta forma, podemos possuir fábricas de diferentes tipos (Hibernate, JDBC, arquivos texto e etc). Para isso, criamos a seguinte classe abstrata:

```
package com.algaworks.dwjsf.financeiro.dao;

//importações

public abstract class DAOFactory {

    public static DAOFactory getDAOFactory() {
        return new HibernateDAOFactory();
    }

    public abstract void iniciarTransacao();
    public abstract void cancelarTransacao();
    public abstract void encerrar();

    public abstract PessoaDAO getPessoaDAO();
    public abstract ContaDAO getContaDAO();

}
```

Essa classe abstrai uma fábrica de DAOs. Ela especifica os métodos que uma fábrica de DAOs é obrigada a ter, e, ao mesmo tempo, possui um método estático que instancia uma fábrica padrão chamada HibernateDAOFactory, que iremos criar agora.

```
package com.algaworks.dwjsf.financeiro.dao.hibernate;

//importações

public class HibernateDAOFactory extends DAOFactory {

	private Session session;
	private Transaction tx;

	public HibernateDAOFactory() {
		this.session = HibernateUtil.getSession();
	}
```

```
	@Override
	public void cancelarTransacao() {
		this.tx.rollback();
		this.tx = null;
	}

	@Override
	public void iniciarTransacao() {
		this.tx = this.session.beginTransaction();
	}

	@Override
	public void encerrar() {
		if (this.tx != null) {
			this.tx.commit();
		}
		this.session.close();
	}

	@Override
	public PessoaDAO getPessoaDAO() {
		return new HibernatePessoaDAO(this.session);
	}

	@Override
	public ContaDAO getContaDAO() {
		return new HibernateContaDAO(this.session);
	}

}
```

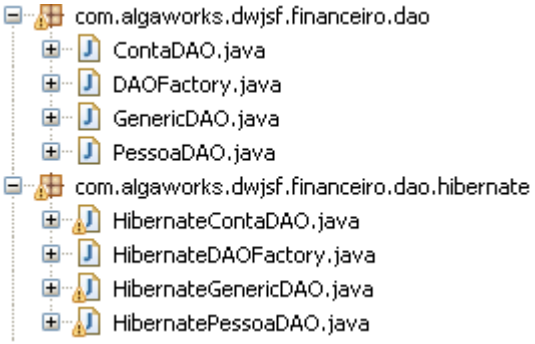
15.5. Instanciando e usando os DAOs

Quando adotamos o padrão DAO, não podemos permitir mais o acesso direto à sessão do Hibernate pelas classes de negócio para executar operações persistentes. O correto é sempre fabricar um DAO e usar os métodos que ele oferece. Por exemplo, na classe `PessoaService`, alteramos os métodos para usar DAO.

```
	public Pessoa pesquisarPorId(Long id) {
		DAOFactory daoFactory = DAOFactory.getDAOFactory();
		PessoaDAO pessoaDAO = daoFactory.getPessoaDAO();
		Pessoa pessoa = pessoaDAO.pesquisarPorId(id);
		daoFactory.encerrar();
		return pessoa;
	}

	public List<Pessoa> listarTodas() {
		DAOFactory daoFactory = DAOFactory.getDAOFactory();
		PessoaDAO pessoaDAO = daoFactory.getPessoaDAO();
		List<Pessoa> pessoas = pessoaDAO
			.listarTodosOrdenadoPorNome();
		daoFactory.encerrar();
		return pessoas;
	}
```

15.6. Valeu a pena usar DAO?

Para usar o padrão DAO no projeto do sistema financeiro, veja as classes e interfaces que foram necessárias.



Você deve analisar o custo-benefício de se usar esse padrão. Para projetos pequenos, como é o caso de nosso sistema financeiro, a adoção do DAO custou muito caro (8 novas classes/interfaces), pois houve mais esforço do que ganho, porém para sistemas maiores, a relação custo-benefício pode ser bastante interessante.

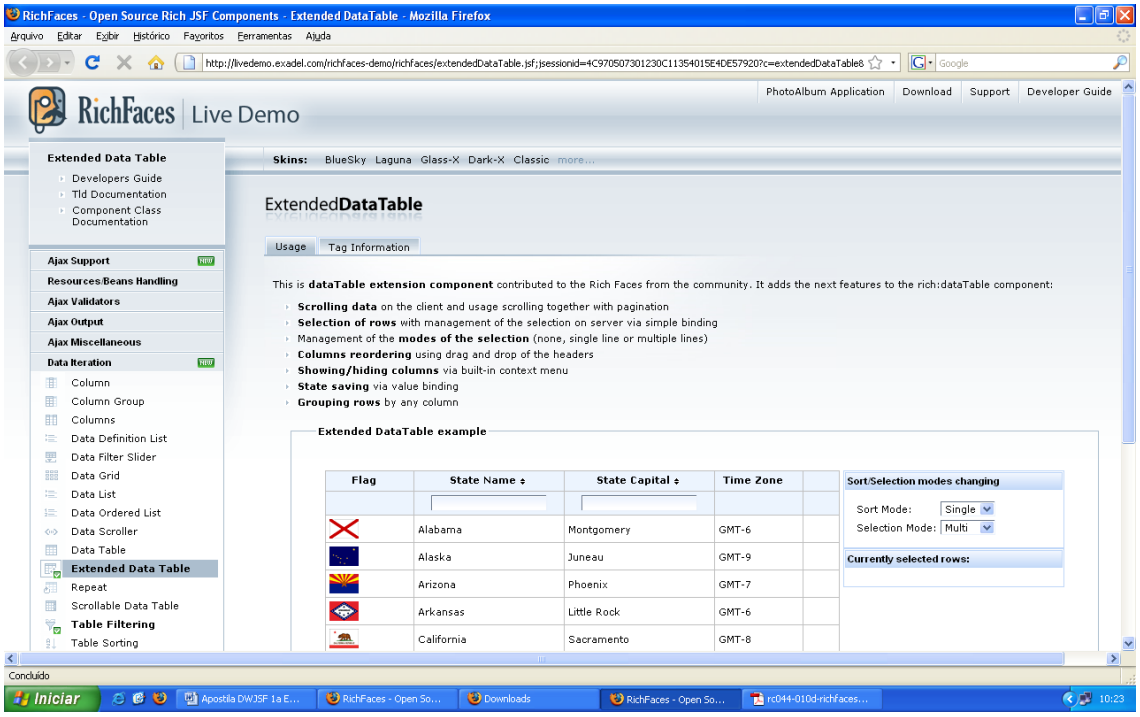
16. JBoss RichFaces e AJAX

16.1. Introdução

AJAX – *Asynchronous JavaScript and XML* – é uma técnica usada para criar aplicações web mais interativas, usando uma combinação de tecnologias que permitem que as requisições feitas pelo browser ao servidor aconteçam sem recarregar a tela.

RichFaces é uma biblioteca JSF open source, fornecida pela JBoss, que ofecere uma variedade de componentes visuais ricos e também um framework capaz de tornar aplicações web aptas para trabalhar com AJAX sem complicações.

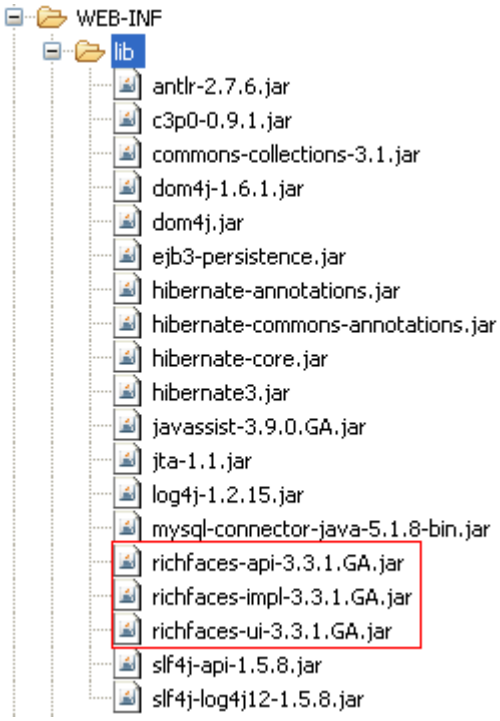
Você pode ver todos os componentes do RichFaces funcionando e baixar a última versão da biblioteca e documentações através do site www.jboss.org/richfaces.



Aprenderemos o básico de RichFaces neste curso. Para guiar nosso estudo, modificaremos nosso sistema financeiro para usar alguns componentes ricos do RichFaces com AJAX.

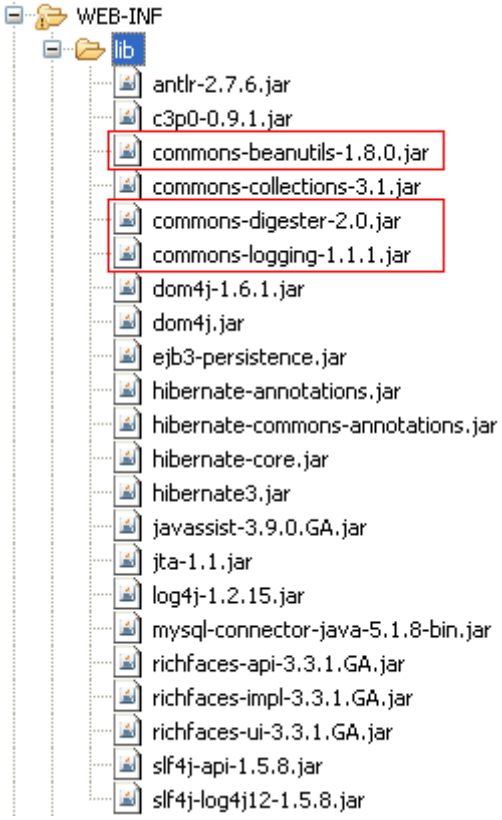
16.2. Configurando e testando o ambiente

Para começarmos a usar RichFaces, precisamos adicionar as bibliotecas ao projeto. Encontre os arquivos “richfaces-api-3.3.1.GA.jar”, “richfaces-impl-3.3.1.GA.jar” e “richfaces-ui-3.3.1.GA.jar” no DVD ou faça download do binário no site da JBoss. Copie estes arquivos para o diretório “WEB-INF/lib” do projeto.



Dependendo do Servlet Container que você estiver usando, será necessário adicionar outros JARs que são dependências do RichFaces. Como estamos usando o Apache Tomcat, precisamos incluir as bibliotecas do Apache Commons Logging, Apache Commons Digester e Apache Commons BeansUtils, disponíveis para download nos seguintes endereços:

http://commons.apache.org/logging/
http://commons.apache.org/digester/
http://commons.apache.org/beanutils/



Depois que as bibliotecas foram adicionadas ao projeto, é necessário configurar o RichFaces no arquivo "web.xml", adicionando as seguintes linhas:

```
<context-param>
	<param-name>org.richfaces.SKIN</param-name>
	<param-value>blueSky</param-value>
</context-param>

<context-param>
	<param-name>org.richfaces.CONTROL_SKINNING</param-name>
	<param-value>enable</param-value>
</context-param>

<filter>
	<display-name>RichFaces Filter</display-name>
	<filter-name>richfaces</filter-name>
	<filter-class>org.ajax4jsf.Filter</filter-class>
</filter>

<filter-mapping>
	<filter-name>richfaces</filter-name>
	<servlet-name>Faces Servlet</servlet-name>
	<dispatcher>REQUEST</dispatcher>
	<dispatcher>FORWARD</dispatcher>
	<dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

Agora nosso projeto está pronto para usar os recursos do RichFaces!

Para testar o ambiente, criaremos uma tela simples com um campo de entrada e um botão. Ao clicar no botão, uma mensagem "Olá, <nome>" aparecerá logo abaixo, substituindo o nome pelo que foi digitado no campo, tudo com AJAX.

Para esse exemplo, criamos uma classe que será o nosso *managed bean*:

```
package com.algaworks.dwjsf.financeiro.visao;

public class HelloRichBean {

	private String nome;

	public String getNome() {
		return nome;
	}

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

}
```

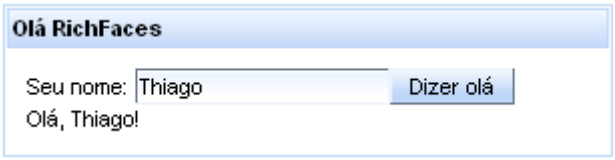
Registre este managed bean no "faces-config.xml" com o nome "helloRichBean" e escopo de requisição.

Vamos criar um arquivo chamado "helloRich.jsp" na raiz do diretório "WebContent", com o seguinte código-fonte:

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html"%>
<%@ taglib uri="http://richfaces.org/a4j" prefix="a4j"%>
<%@ taglib uri="http://richfaces.org/rich" prefix="rich"%>
<html>
```

```
<body>
<f:view>
	<h:form>
		<rich:panel header="Olá RichFaces" style="width: 300px">
			<h:outputText value="Seu nome: " />
			<h:inputText value="#{helloRichBean.nome}"/>
			<a4j:commandButton value="Dizer olá"
				reRender="olaPainel" />
			<h:panelGroup id="olaPainel" layout="block">
				<h:outputText value="Olá, "
				  rendered="#{not empty helloRichBean.nome}" />
				<h:outputText value="#{helloRichBean.nome}" />
				<h:outputText value="!"
				  rendered="#{not empty helloRichBean.nome}" />
			</h:panelGroup>
		</rich:panel>
	<h:form>
</f:view>
</body>
</html>
```

E é só isso! Agora execute a página e você verá um pequeno painel com os componentes que adicionamos. Digite seu nome, clique no botão “Dizer olá” e veja o RichFaces com AJAX funcionando com seus próprios olhos.



Para usar os componentes do RichFaces importamos duas bibliotecas de tags com o prefixo “a4j” e “rich”.

```
<%@ taglib uri="http://richfaces.org/a4j" prefix="a4j"%>
<%@ taglib uri="http://richfaces.org/rich" prefix="rich"%>
```

A biblioteca “a4j” possui vários componentes que dão suporte a AJAX. Essa biblioteca é conhecida como Ajax4JSF, que era um framework que adicionava suporte AJAX a aplicações JSF, e foi incorporada pelo RichFaces.

A biblioteca “rich” disponibiliza vários componentes ricos, que dão um “tcham” nas aplicações web. Conheceremos alguns deles ainda neste capítulo.

Usamos o componente `rich:panel` para adicionar um painel visual para agrupar os outros componentes e `a4j:commandButton` para dar suporte a AJAX e renderizar dinamicamente (e sem *refresh*) a mensagem “Olá <nome>”.

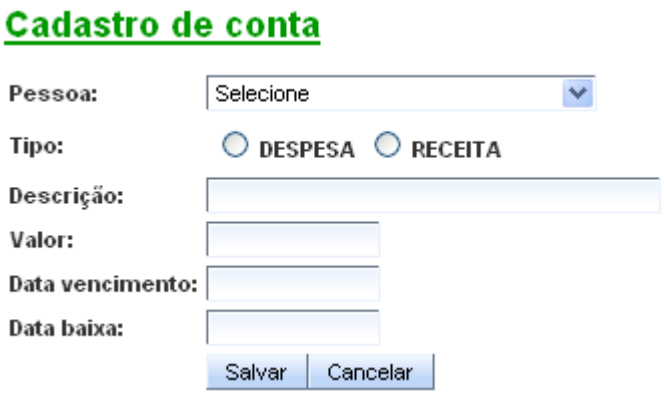
Perceba que fazermos o clique do botão funcionar com AJAX, bastou usarmos um componente de botão da biblioteca “a4j”. O atributo `reRender` indicou qual componente deveria ser atualizado após o clique.

```
<a4j:commandButton value="Dizer olá" reRender="olaPainel" />
```

O componente identificado como “olaPainel” é um painel que agrupa as mensagens com os dizeres que queremos que apareça.

```
<h:panelGroup id="olaPainel" layout="block">
	...
</h:panelGroup>
```

Você deve ter percebido que os campos e botões dessa pequena tela possuem estilos que o próprio RichFaces colocou. Para ficar mais surpeendido, acesse a tela de cadastro de conta do sistema financeiro e veja que, mesmo sem alterar nada no código-fonte dessa tela, sua aparência também mudou.



16.3. Adicionando suporte AJAX em componentes não-AJAX

Existe um componente chamado `a4j:support`, que é um dos mais importantes do núcleo da biblioteca do RichFaces. O `a4j:support` é um componente não-visual que dá suporte a AJAX para componentes não-AJAX, como os componentes padrões do JSF.

Para demonstrar o uso desse componente, alteramos o arquivo "helloRich.jsp" do último exemplo. Removemos o botão "Dizer olá", pois o usuário não precisará mais clicar em botão algum. O sistema deverá atualizar a mensagem "Olá <nome>" incrementalmente, em tempo de digitação.

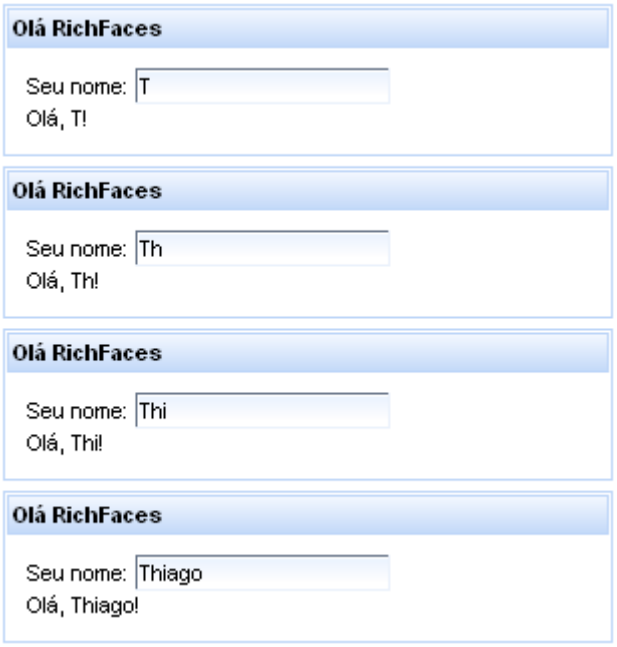
Para que isso funcione, incluímos o componente `a4j:support` como filho de `h:inputText`.

```
<h:inputText value="#{helloRichBean.nome}">
      <a4j:support event="onkeyup" reRender="olaPainel"/>
</h:inputText>
```

Definimos o atributo `event` igual a "onkeyup". Esse atributo recebe o nome de um evento JavaScript que será usado para ativá-lo. O evento "onkeyup" é ativado sempre que digitamos alguma coisa no campo.

O atributo `reRender` foi definido com o nome do identificador do painel que contém as mensagens "Olá <nome>", ou seja, a cada tecla digitada pelo usuário, o painel de mensagens deverá ser re-renderizado.

Execute novamente a tela e veja o resultado.



16.4. Adicionando suporte AJAX no cadastro de contas

Agora que já sabemos como usar AJAX, vamos integrar esta tecnologia na tela de cadastro de contas (arquivo "cadastroConta.jsp"). A única alteração que precisamos fazer é no botão "Salvar". Substituímos este botão por um `a4j:commandButton`, atribuindo `reRender` com "frm", para que o formulário inteiro seja re-renderizado no clique do botão e também informamos o valor "submit" no atributo `type`, para que a tecla *<enter>* dispare a submissão do formulário.

```
<a4j:commandButton value="Salvar"
      actionListener="#{cadastroContaBean.salvar}"
      reRender="frm" type="submit"/>
<h:commandButton value="Cancelar" action="menu"
      immediate="true"/>
```

Removemos o atributo `styleClass` dos botões, pois o RichFaces adiciona seus próprios estilos.

16.5. Componente de calendário

O componente `rich:calendar` renderiza um campo para entrada de data/hora. Este componente é altamente customizável e bastante rico para o usuário. Podemos configurá-lo para receber data e hora ou apenas data, exibir um campo e um popup de um calendário ou apenas exibir o calendário diretamente na tela, etc.

Alteramos a tela de cadastro de conta e incluímos este componente para os campos "Data vencimento" e "Data baixa".

```
<h:outputLabel value="Data vencimento:"/>
<h:panelGroup>
      <rich:calendar id="dataVencimento" inputSize="12"
          datePattern="dd/MM/yyyy" enableManualInput="true"
          value="#{cadastroContaBean.contaEdicao.dataVencimento}"
          required="true" label="Data vencimento"/>
      <h:message for="dataVencimento" showSummary="true"
            showDetail="false" styleClass="msgErro"/>
```

```
</h:panelGroup>

<h:outputLabel value="Data baixa:"/>
<h:panelGroup>
	<rich:calendar id="dataBaixa" inputSize="12"
		datePattern="dd/MM/yyyy" enableManualInput="true"
		value="#{cadastroContaBean.contaEdicao.dataBaixa}"
		label="Data baixa"/>
	<h:message for="dataBaixa" showSummary="true"
		showDetail="false" styleClass="msgErro"/>
</h:panelGroup>
```

O atributo `inputSize` define o tamanho do campo digitável, `datePattern` é usado para alterar o formato da data e `enableManualInput` especifica se é permitido a digitação da data pelo usuário (*true*) ou se a data pode ser informada apenas pelo calendário (*false*).

16.6. Componente de caixa de sugestão

O componente `rich:suggestionbox` adiciona a capacidade de auto-completar campos de entrada de texto normais, como o `h:inputText`.

Quando o usuário pressiona uma tecla em um campo vinculado ao `rich:suggestionbox`, uma requisição AJAX é enviada ao servidor para consultar possíveis valores a serem sugeridos ao usuário e esses valores são apresentados em uma janela flutuante que fica abaixo do campo.

Alteramos a tela de cadastro de conta do sistema financeiro para que, ao digitar a descrição da conta, o sistema sugira outras descrições já usadas anteriormente, poupando esforços de nossos queridos usuários, pois não precisam digitar as descrições completas de contas recorrentes.

Antes de alterar a tela, precisamos criar uma consulta de descrições no DAO e na classe de negócio. Primeiro adicionamos o seguinte método na classe `HibernateContaDAO`.

```
public List<String> pesquisarDescricoes(String descricao) {
	return this.getSession().createCriteria(Conta.class)
		.setProjection(Projections.distinct(
```

```
                Projections.property("descricao")))
            .add(Restrictions.ilike("descricao",
                descricao, MatchMode.ANYWHERE))
            .addOrder(Order.asc("descricao")).list();
    }
```

Esse método pesquisa descrições distintas no banco de dados, semelhantes à descrição recebida por parâmetro. Não podemos nos esquecer de incluir a assinatura desse método na interface `ContaDAO`.

Na classe `ContaService`, apenas adicionamos um método que invoca a pesquisa de descrições do DAO.

```
public List<String> pesquisarDescricoes(String descricao) {
    DAOFactory daoFactory = DAOFactory.getDAOFactory();
    ContaDAO contaDAO = daoFactory.getContaDAO();
    List<String> descricoes = contaDAO
        .pesquisarDescricoes(descricao);
    daoFactory.encerrar();
    return descricoes;
}
```

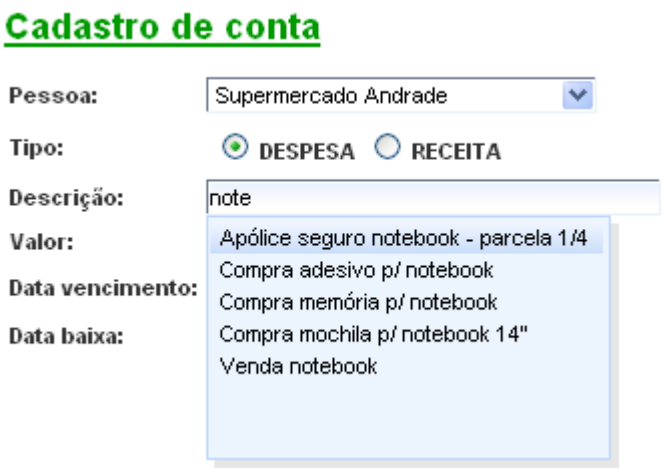
Na classe `CadastroContaBean`, que é o *managed bean* que usamos para a tela de cadastro de conta, incluímos um método que retorna uma lista de Strings e recebe um objeto como parâmetro. Este objeto recebido como parâmetro é exatamente o que o usuário digitou no campo, e por isso precisamos usá-lo para criar nossa lista de sugestões.

```
public List<String> sugerirDescricao(Object event) {
    return new ContaService()
        .pesquisarDescricoes(event.toString());
}
```

Finalmente, no arquivo "cadastroConta.jsp", apenas incluímos o componente `rich:suggestionbox`, o vinculamos ao `h:inputText` de descrição e especificamos o método que deve ser chamado no *managed bean* sempre que o usuário pressionar uma tecla.

```
<h:outputLabel value="Descrição:"/>
<h:panelGroup>
    <h:inputText id="descricao" size="40" maxlength="150"
        value="#{cadastroContaBean.contaEdicao.descricao}"
        required="true" label="Descrição"/>
    <rich:suggestionbox for="descricao"
        suggestionAction="#{cadastroContaBean.sugerirDescricao}"
        width="230" height="120" var="item">
        <h:column>
            <h:outputText value="#{item}"/>
        </h:column>
    </rich:suggestionbox>
    <h:message for="descricao" showSummary="true"
        showDetail="false" styleClass="msgErro"/>
</h:panelGroup>
```

O resultado final é fantástico! Prepare pare receber uma bela bonificação dos usuários.



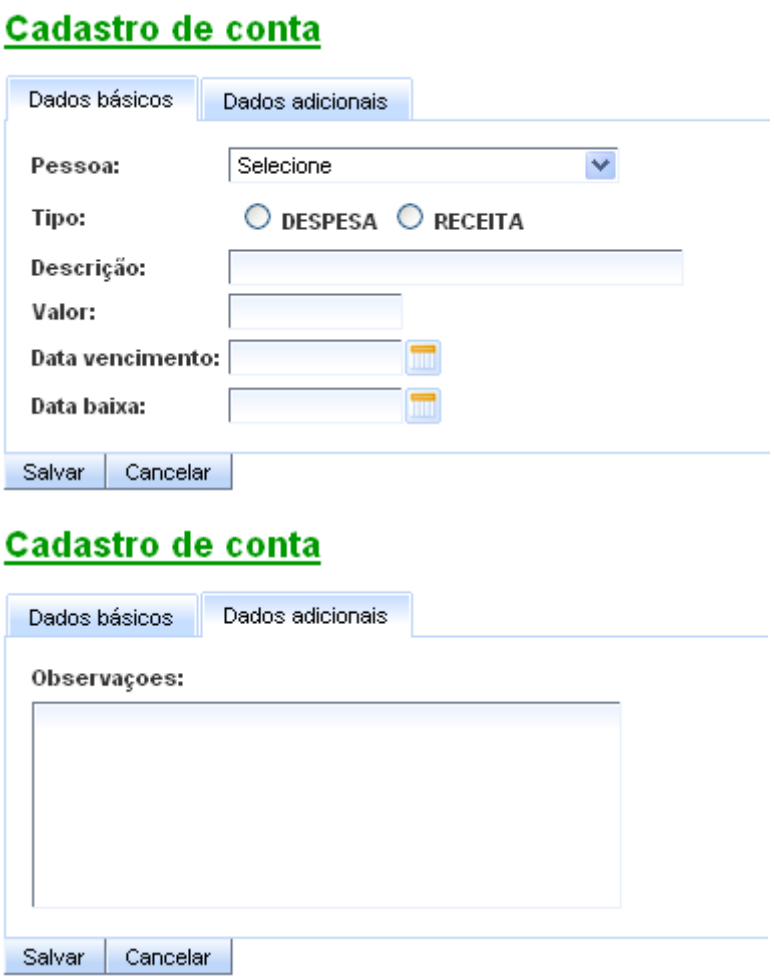
16.7. Componente de painel com abas

O componente `rich:tabPanel` é usado para criar painéis com abas. Este tipo de componente é muito utilizado em aplicativos desktop, pois permite separar vários componentes de uma mesma tela em áreas diferentes.

Não temos nenhuma tela com muitos componentes no sistema financeiro e que precise ser organizada, por isso tentaremos "forçar" a necessidade desse componente. Na tela de cadastro de conta, incluiremos um campo para o usuário digitar observações. Este campo não estará vinculado a nenhum atributo do *managed bean* e nem ao objeto de domínio, pois estamos apenas simulando uma necessidade. Como o campo de observações será um `h:inputTextarea`, vamos colocá-lo em uma aba separada, para deixar a tela com uma impressão compacta. Veja o código-fonte:

```
<rich:tabPanel>
	<rich:tab label="Dados básicos" switchType="client">
		<h:panelGrid columns="2">
			<!-- os componentes do formulário ficam aqui -->
		</h:panelGrid>
	</rich:tab>
	<rich:tab label="Dados adicionais" switchType="client">
		<h:panelGrid columns="1">
			<h:outputLabel value="Observaçoes:"/>
			<h:inputTextarea rows="6" cols="50"/>
		</h:panelGrid>
	</rich:tab>
</rich:tabPanel>

<h:panelGroup>
	<a4j:commandButton value="Salvar"
		actionListener="#{cadastroContaBean.salvar}"
		reRender="frm" type="submit"/>
	<h:commandButton value="Cancelar" action="menu"
		immediate="true"/>
</h:panelGroup>
```

O componente `rich:tabPanel` cria um painel capaz de conter abas. As abas propriamente ditas são representadas pelo componente `rich:tab`. O atributo `label` do componente `rich:tab` deve ser informado com o rótulo da aba e `switchType` com o mecanismo a ser utilizado para alternar as abas. Como informamos "client", as abas serão alternadas no browser, sem nenhuma requisição ao servidor.



16.8. Componente de tabela de dados

O componente `rich:dataTable` permite apresentar dados tabulares, assim como o `h:dataTable`. A diferença é que a tabela de dados do RichFaces incorpora várias funcionalidades interessantes, como mesclagem de linhas e colunas, layout do cabeçalho e rodapé da tabela flexível, conceito de sub-tabelas, etc.

O `rich:dataTable` funciona em conjunto com as colunas padrão do JSF, o componente `h:column`, porém o RichFaces também possui seu próprio componente de coluna, chamado `rich:column`. Este componente junto com `rich:dataTable` oferecem várias facilidades que com `h:column` você não teria, como por exemplo ordenação de dados, mesclagem de linhas e colunas, etc.

Para demonstrar o uso de `rich:dataTable` e `rich:column`, iremos alterar a tela de consulta de contas do sistema financeiro.

```
<rich:dataTable value="#{consultaContaBean.contas}" var="item"
            width="790px">
      <rich:column sortBy="#{item.tipo}" width="20px"
            style="text-align: center">
            <f:facet name="header">
                  <h:outputText value="Tipo"/>
            </f:facet>
            <h:graphicImage value="/imagens/receita.png"
                  title="Conta a receber"
                  rendered="#{item.tipo eq 'RECEITA'}"/>
            <h:graphicImage value="/imagens/despesa.png"
```

```
                title="Conta a pagar"
                rendered="#{item.tipo eq 'DESPESA'}"/>
    </rich:column>
    <rich:column sortBy="#{item.pessoa.nome}" width="200px">
          <f:facet name="header">
                <h:outputText value="Pessoa"/>
          </f:facet>
          <h:outputText value="#{item.pessoa.nome}"/>
    </rich:column>
    <rich:column sortBy="#{item.descricao}">
          <f:facet name="header">
                <h:outputText value="Descrição"/>
          </f:facet>
          <h:outputText value="#{item.descricao}"/>
    </rich:column>
    <rich:column sortBy="#{item.valor}" width="100px"
          style="text-align: center">
          <f:facet name="header">
                <h:outputText value="Valor"/>
          </f:facet>
          <h:outputText value="#{item.valor}"
                style="color: #{item.tipo eq 'RECEITA' ? 'blue'
                                  : 'red'}">
                <f:convertNumber minFractionDigits="2"
                      currencyCode="BRL" type="currency"/>
          </h:outputText>
    </rich:column>
    <rich:column sortBy="#{item.dataVencimento}" width="100px"
          style="text-align: center">
          <f:facet name="header">
                <h:outputText value="Vencimento"/>
          </f:facet>
          <h:outputText value="#{item.dataVencimento}">
                <f:convertDateTime pattern="dd/MM/yyyy"/>
          </h:outputText>
    </rich:column>
    <rich:column width="80px" style="text-align: center">
          <f:facet name="header">
                <h:outputText value="Aberta"/>
          </f:facet>
          <h:outputText value="#{item.dataBaixa == null
                                  ? 'Sim' : 'Não'}"/>
    </rich:column>
    <rich:column width="40px" style="text-align: center">
          <f:facet name="header">
                <h:outputText value="Ações"/>
          </f:facet>
          <h:commandLink action="#{consultaContaBean.excluir}">
                <f:setPropertyActionListener value="#{item}"
                  target="#{consultaContaBean.contaExclusao}"/>
                <h:graphicImage value="/imagens/excluir.png"
                      title="Excluir" styleClass="imagemLink"/>
          </h:commandLink>
          <h:commandLink action="cadastroConta">
                <f:setPropertyActionListener value="#{item}"
                  target="#{cadastroContaBean.contaEdicao}"/>
                <h:graphicImage value="/imagens/editar.png"
                      title="Editar" styleClass="imagemLink"/>
          </h:commandLink>
    </rich:column>
```

```
</rich:dataTable>
```

Para começar, tiramos todas as classes CSS da tabela, pois o componente do RichFaces já inclui alguns estilos próprios, porém é importante falar que não somos obrigados a usar os estilos do RichFaces.

No componente `rich:column`, incluímos expressões na propriedade `sortBy` para possibilitar a ordenação dos dados através de um clique na coluna. Perceba também que a coluna do RichFaces possui atributos interessantes como `width` e `style`. Parece básico, mas só para você ficar sabendo, `h:column` não possui esses atributos.

Veja como ficou nossa tabela de dados:

Consulta de contas

Tipo	Pessoa	Descrição	Valor	Vencimento	Aberta	Ações
	Condomínio Ed. Jardim das Couves	teste	R$ 10,00	08/09/2009	Não	
	Sebastião Costa Silva	Compra adesivo p/ notebook	R$ 20,00	22/09/2009	Sim	
	Supermercado Andrade	Compra memória p/ notebook	R$ 100,00	30/09/2009	Sim	
	Condomínio Ed. Jardim das Couves	Condomínio	R$ 200,00	10/10/2009	Sim	
	Sebastião Costa Silva	Apólice seguro notebook - parcela 1/4	R$ 200,00	29/09/2009	Sim	
	Supermercado Andrade	Compras	R$ 234,35	20/09/2009	Não	
	Sebastião Costa Silva	Compra mochila p/ notebook 14"	R$ 290,00	02/09/2009	Sim	
	Escola Infantil Abelha Rainha	Escola filho	R$ 450,00	09/10/2009	Sim	
	Udi Imobiliária	Aluguel do AP	R$ 600,00	01/10/2009	Sim	
	Sebastião Costa Silva	Venda notebook	R$ 900,00	28/09/2009	Não	
	Prefeitura de Uberlândia	Salário	R$ 1.500,00	08/10/2009	Não	

16.9. Componente de paginação de dados

O componente `rich:datascroller` adiciona um controle de paginação visual que alterna as páginas de acordo com a escolha do usuário, usando AJAX.

Para usar este componente, identificamos a tabela de dados da tela de consulta de contas com o nome “contas” e informamos um número máximo de linhas que queremos permitir que seja exibido na tabela através do atributo `rows`. Depois disso, adicionamos o componente `rich:datascroller` e o vinculamos com a tabela e dados através do atributo `for`.

```
<rich:dataTable id="contas" value="#{consultaContaBean.contas}"
      var="item" width="790px" rows="5">

      ...

</rich:dataTable>

<rich:datascroller for="contas" maxPages="20" align="left"/>
```

Agora temos uma tabela que apresenta no máximo 5 linhas de cada vez, controlada por um componente de paginação.

Consulta de contas

Tipo	Pessoa	Descrição	Valor	Vencimento	Aberta	Ações
	Condomínio Ed. Jardim das Couves	Condomínio	R$ 200,00	10/10/2009	Sim	
	Escola Infantil Abelha Rainha	Escola filho	R$ 450,00	09/10/2009	Sim	
	Prefeitura de Uberlândia	Salário	R$ 1.500,00	08/10/2009	Não	
	Udi Imobiliária	Aluguel do AP	R$ 600,00	01/10/2009	Sim	
	Supermercado Andrade	Compra memória p/ notebook	R$ 100,00	30/09/2009	Sim	

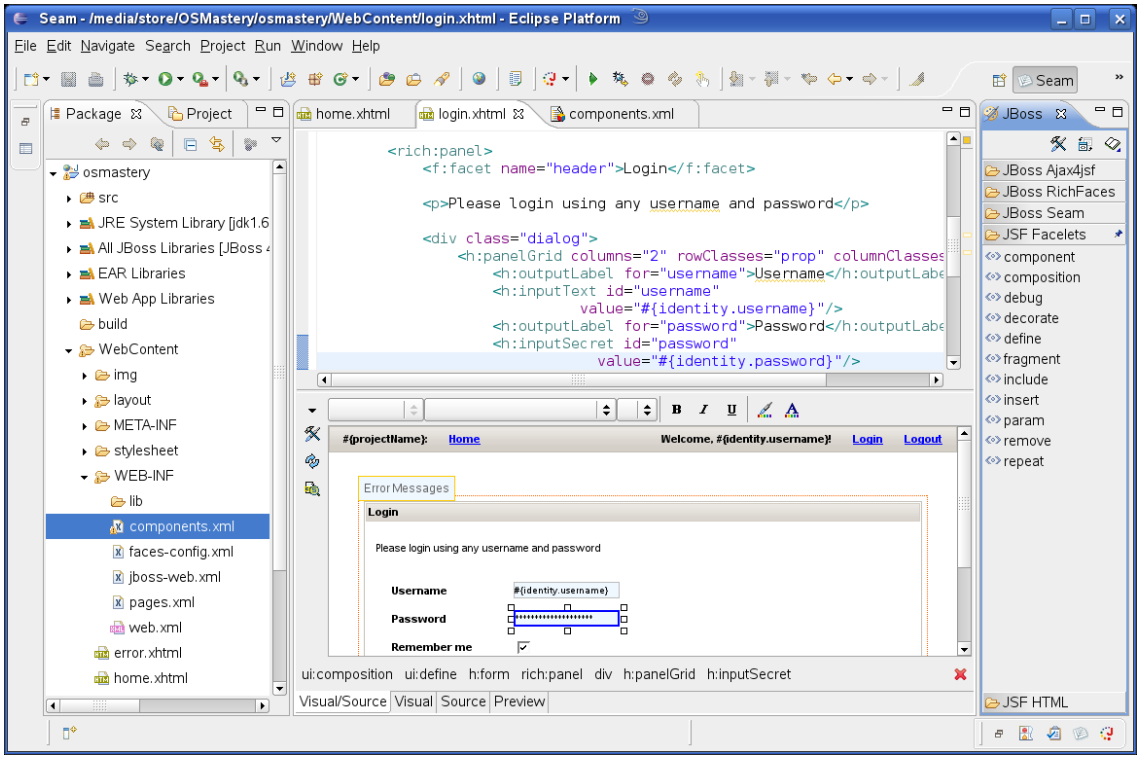
1 2 3

16.10. Clicar e arrastar com JBoss Tools

JBoss Tools é um projeto guarda-chuva que contém vários plugins do Eclipse que suportam tecnologias relacionadas ao JBoss, RichFaces, Hibernate, Seam, etc.

Com esta ferramenta, é possível desenvolver telas JSF clicando e arrastando componentes. No DVD deste curso, disponibilizamos algumas versões do JBoss Tools para você "brincar" em casa.

O site oficial deste projeto, que contém toda documentação e as últimas versões para download é www.jboss.org/tools.



17. Facelets

17.1. Introdução

Até a versão 1.2 do JSF, existem algumas deficiências que dificultam a vida do programador de páginas, como a falta de um mecanismo para criação de templates de telas integrado ao JSF e também para criação de novos componentes visuais a partir da composição de outros componentes já existentes. Para resolver essas deficiências, o framework chamado Facelets foi desenvolvido.

O site oficial do Facelets é http://facelets.dev.java.net.

Para trabalhar com Facelets, precisamos programar nossas páginas usando XHTML (e***X***tensible ***H***yper***T***ext ***M***arkup ***L***anguage), e não JSP, pois o framework inclui um compilador de XHTML para fazer toda a "mágica".

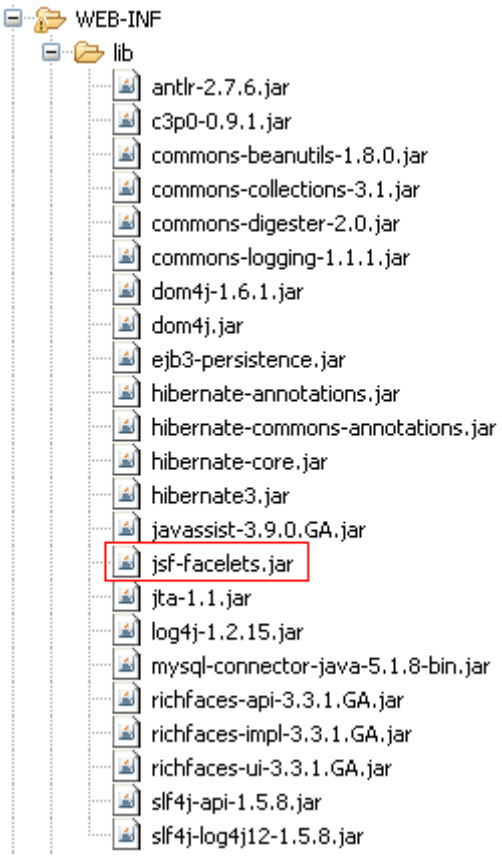
XHTML é uma reformulação da linguagem HTML baseada em XML, por isso muitas pessoas consideram XHTML como o sucessor de HTML.

XHTML combina as tags de marcação HTML com regras da XML, por isso consegue ser interpretado por qualquer dispositivo, independente da plataforma, pois as marcações possuem sentido semântico.

A boa notícia é que não existem muitas diferenças entre HTML e XHTML. Uma das regras mais importantes é que toda tag aberta deve ser fechada. Quantas vezes você já escreveu uma quebra de linha em HTML sem fechar, usando apenas `<BR>`? Pois é, em XHTML, o correto é `<BR/>`. Lembre-se: abriu, fechou.

17.2. Instalando Facelets no projeto

O Facelets é distribuído através de um JAR chamado "jsf-facelets.jar". Para começar a usá-lo, encontre este arquivo no DVD do curso ou faça download no site do framework e copie para o diretório "WEB-INF/lib" do projeto.



Edite o arquivo “web.xml” do projeto do sistema financeiro e inclua o seguinte fragmento de código:

```
<context-param>
      <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
      <param-value>.jspx</param-value>
</context-param>

<context-param>
      <param-name>facelets.DEVELOPMENT</param-name>
      <param-value>true</param-value>
</context-param>
```

Apesar de não ser obrigatório, vamos renomear os arquivos das telas que desenvolvemos para o sistema financeiro para usar a extensão “jspx”, por isso nós definimos o primeiro parâmetro do fragmento de código anterior como “.jspx”.

Queremos também que o Facelets trabalhe em “modo de desenvolvimento”, por isso definimos como “true” o parâmetro “facelets.DEVELOPMENT”. Trabalhar no “modo de desenvolvimento” significa que queremos ajuda do Facelets para conseguirmos *debugar* os erros.

Agora precisamos alterar o arquivo “faces-config.xml” para incluir a tag `<view-handler>` dentro de `<application>`. No projeto do sistema financeiro, a tag `<application>` ficará como abaixo:

```
<application>
      <view-handler>
            com.sun.facelets.FaceletViewHandler
      </view-handler>
      <message-bundle>
            com.algaworks.dwjsf.financeiro.recursos.messages
      </message-bundle>
</application>
```

Nosso ambiente está pronto para funcionar com Facelets! Agora o único problema é que nossas páginas não foram desenvolvidas usando XHTML e também não foram renomeadas para usar a extensão “jspx”.

17.3. Adequando nossas páginas para XHTML

Precisamos adequar nossas 3 páginas (menu, cadastro de conta e consulta de contas) do sistema financeiro para XHTML. Vamos começar pelo arquivo “menu.jsp”. Renomeie-o para “menu.jspx”.

Agora edite o arquivo “menu.jspx” e substitua o código:

```
<%@ page contentType="text/html; charset=ISO-8859-1"
         pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<html>

      ...

</html>
```

Por:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```
<!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:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

      ...

</html>
```

A declaração *DOCTYPE* não é uma tag HTML, mas uma instrução ao browser sobre qual versão da linguagem de marcação a página é escrita. Definimos que usaremos o XHTML 1.0 Transitional.

A referência a bibliotecas de tags não deve mais ser feita usando a diretiva `taglib` do JSP, mas *XML NameSpace* (*xmlns*).

Pronto! Essas alterações devem ser suficientes para que a tela de menu funcione. Acesse http://localhost:8080/Financeiro/faces/menu.jspx e verifique.

Faça o mesmo com os arquivos “cadastroConta.jsp” e “consultaConta.jsp”. Na declaração dos namespaces desses arquivos, você deve adicionar as bibliotecas “a4j” e “rich”. Veja um exemplo:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:a4j="http://richfaces.org/a4j"
      xmlns:rich="http://richfaces.org/rich">
```

Edite o arquivo “faces-config.xml” e altere as referências dos arquivos JSP para JSPX.

Agora confira se todas as telas do sistema estão funcionando.

17.4. Reutilização com *templating*

Com Facelets, podemos criar páginas *templates* e reutilizá-las em outras páginas específicas. Dessa forma, evitamos replicação de código de layout de página.

Para testar, crie um arquivo chamado “template.jspx” na pasta “WebContent” do projeto, com o seguinte conteúdo:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!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:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:a4j="http://richfaces.org/a4j"
      xmlns:rich="http://richfaces.org/rich"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
      <title>Sistema financeiro -
            <ui:insert name="tela"/></title>
      <link rel="stylesheet" type="text/css"
            href="/Financeiro/css/estilo.css" />
</head>
<body>
      <h:outputText value="Bem vindo ao sistema financeiro!"
            style="font-weight: bold; font-size: 17pt;
                  font-family: Arial"/>
```

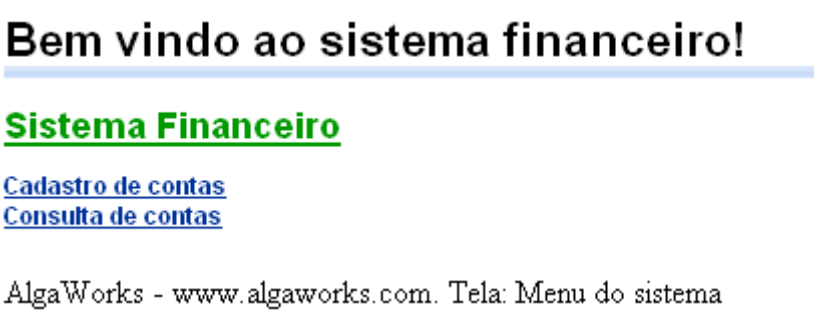
```
            <rich:separator/>

            <ui:insert name="corpo"/>

            <br/>
            AlgaWorks - www.algaworks.com. Tela:
            <ui:insert name="tela"/>
      </body>
      </html>
```

Definimos “regiões” variáveis no template através da tag `<ui:insert>` que serão substituídas em tempo de execução através do atributo `name`. As regiões foram nomeadas por “tela” e “corpo”.

Agora todas nossas páginas podem usar este arquivo de template e se beneficiar da reutilização de código e da fácil manutenabilidade alcançada. Edite o arquivo “menu.jspx” e adéqüe-o para que ele fique da seguinte forma:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!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:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/template.jspx">

      Este texto não será exibido!

      <ui:define name="tela"> Menu do sistema</ui:define>

      Este texto também não será exibido.
      Facelets ignora textos neste local.

      <ui:define name="corpo">
            <h:form id="frm">
                  <h1><h:outputText value="Sistema Financeiro"/></h1>

                  <h:commandLink
                        action="#{cadastroContaBean.inicializar}">
                        <h:outputText value="Cadastro de contas"/>
                  </h:commandLink>
                  <br/>
                  <h:commandLink action="consultaConta"
                       actionListener="#{consultaContaBean.consultar}">
                        <h:outputText value="Consulta de contas"/>
                  </h:commandLink>
            </h:form>
      </ui:define>

</ui:composition>
</html>
```

Na página “menu.jspx”, dizemos que ela deve usar como template o arquivo “template.jspx” através da tag `<ui:composition>` e depois especificamos o conteúdo das regiões “tela” e “corpo” através da tag `<ui:define>`.



17.5. Reutilização com *composition*

Com Facelets, a criação de componentes personalizados baseado em composições é muito fácil. Como exemplo, criaremos um componente chamado `labeledInputText` que ao incluí-lo na página renderizará um `h:outputLabel` juntamente com um `h:inputText`.

Para começar, criamos um arquivo chamado "labeledInputText.jspx" e colocamos na pasta "WEB-INF/tags" (crie-a primeiro) com o conteúdo:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">
      <c:if test="#{not empty label}">
            <h:outputLabel value="#{label}"/>
      </c:if>
      <h:inputText value="#{value}"/>
</ui:composition>
```

Este é o código-fonte do nosso componente! Provavelmente você não irá querer usá-lo em seus sistemas, pois está muito simples, mas a intenção é apenas demonstrar o funcionamento. Perceba que podemos usar tags JSTL, como a bilioteca *core*.

Precisamos registrar nosso componente para que ele fique visível às páginas. Primeiro, crie um arquivo chamado "algaworks.taglib.xml" na pasta "WEB-INF" e coloque o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE facelet-taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
  "http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib>
      <namespace>http://www.algaworks.com/dwjsf</namespace>
      <tag>
            <tag-name>labeledInputText</tag-name>
            <source>tags/labeledInputText.jspx</source>
      </tag>
</facelet-taglib>
```

Neste arquivo, especificamos qual é o *namespace* da biblioteca de componentes e também definimos todas as tags customizadas que criamos.

Para finalizar a configuração de nossa biblioteca no projeto, edite o arquivo "web.xml" e inclua:

```
<context-param>
      <param-name>facelets.LIBRARIES</param-name>
      <param-value>/WEB-INF/algaworks.taglib.xml</param-value>
</context-param>
```

Pronto! Para testar, crie ou edite uma página JSPX qualquer, importe o *namespace* “http://www.algaworks.com/dwjsf” e use o componente normalmente:

```
<o:labeledInputText label="Descrição:"
      value="#{bean.propriedade}"/>
```

18. Segurança com JAAS

18.1. Introdução

Java Authentication and Authorization Service, ou JAAS (pronunciado “Jazz”), é um conjunto de APIs padrão do Java usado para dar segurança às aplicações. O JAAS é responsável pela:

- **Autenticação:** validação de usuário/senha, ou seja, login do usuário no sistema.
- **Autorização:** verificação se existe um usuário autenticado e com as devidas permissões de acesso para acessar determinado recurso protegido do sistema, como uma tela, diretório, arquivo, etc.

Com JAAS, os controles de acesso ficam desacoplados do sistema que você está desenvolvendo, ou seja, a responsabilidade da autorização fica a cargo da API. As configurações de acesso e seus perfis são feitas de forma declarativa no descritor da aplicação, no arquivo “web.xml”.

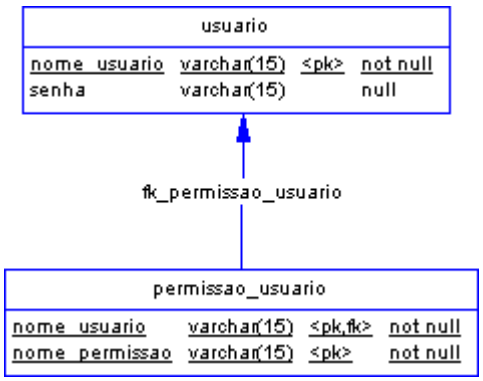
A vantagem de utilizar JAAS em suas aplicações é que você economiza tempo e dinheiro, pois não precisa criar mecanismos de segurança do zero para cada sistema desenvolvido. Além disso, você ganha por estar utilizando uma API padrão de mercado, com baixa curva de aprendizado, conhecido por milhares de programadores, testado e aprovado por especialistas em segurança.

O JAAS é nativo da plataforma Java e todos containers web o implementa, por isso não é necessário baixar nenhuma biblioteca para usá-lo.

18.2. Criando tabelas de segurança no banco de dados

Para demonstrar JAAS funcionando, iremos implementar a segurança no sistema financeiro.

Precisamos de algumas tabelas no banco de dados para armazenar o usuário, senha e as permissões de cada usuário. Nosso modelo de dados será bastante simples, mas você poderia ter tabelas com muito mais campos e relacionamentos.



A tabela “usuario” armazena o nome e a senha do usuário e a tabela “permissao_usuario” associa o usuário previamente cadastrado a um nome de permissão.

Para criar as tabelas no MySQL, execute a DDL abaixo:

```
create table usuario (
  nome_usuario  varchar(15) not null primary key,
  senha         varchar(15) not null
);

create table permissao_usuario (
  nome_usuario    varchar(15) not null,
```

```
  nome_permissao  varchar(15) not null,
  primary key (nome_usuario, nome_permissao)
);

alter table permissao_usuario
add constraint fk_permissao_usuario
foreign key (nome_usuario) references usuario (nome_usuario);
```

Agora que as tabelas estão criadas no banco de dados, iremos cadastrar alguns usuários e associar algumas permissões diferentes para cada um deles. Use o script DML abaixo:

```
insert into usuario values ('joao', 'joao');
insert into permissao_usuario values ('joao', 'cadastro');

insert into usuario values ('manoel', 'manoel');
insert into permissao_usuario values ('manoel', 'consulta');

insert into usuario values ('frederico', 'frederico');
insert into permissao_usuario values ('frederico', 'cadastro');
insert into permissao_usuario values ('frederico', 'consulta');
```

O usuário “joao” terá acesso apenas para cadastros, “manoel” apenas para consultas e “frederico" para cadastros e consultas. As senhas dos usuários ficaram iguais aos nomes dos usuários para facilitar os testes.

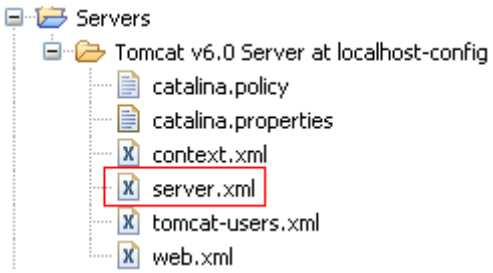
18.3. Configurando o domínio de segurança

A autenticação usando JAAS é flexível, pois ela é feita através de *Login Modules*. *Login Module* é uma interface que é implementada por classes que consultam o usuário e senha em algum lugar e realizam as regras de autenticação. Se você tiver aplicações legadas e deseja usar o mesmo mecanismo de autenticação existente, poderá implementar o seu próprio *Login Module*.

Os containers web implementam alguns módulos de login básicos para autenticação a partir de arquivos de propriedades ou XML contendo os nomes e permissões dos usuários, banco de dados, etc.

O Tomcat implementa esses módulos de login em forma de domínios (*Realms*). Domínios significam um conjunto de usuários e senhas que identificam usuários válidos para uma aplicação web, mais uma lista de permissões (*roles*) associadas para cada usuário.

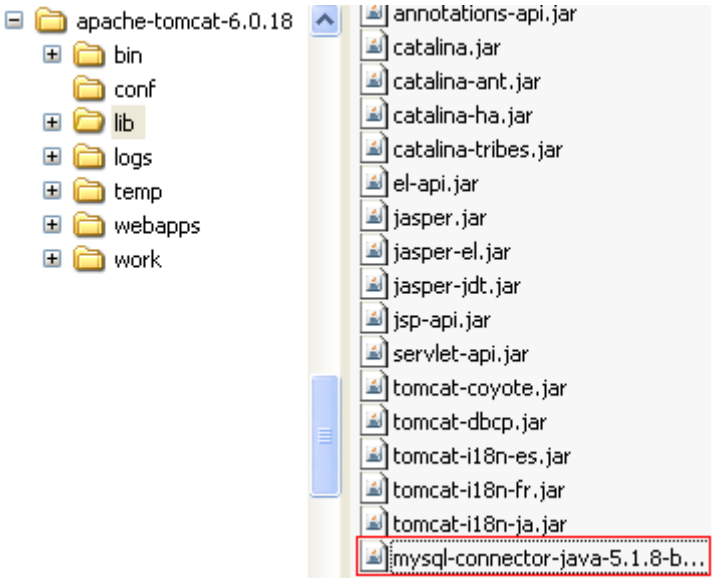
Para o nosso exemplo, usaremos o *realm* `JDBCRealm`, que suporta autenticação através de bancos de dados relacionais, acessados via JDBC. Para configurar o `JDBCRealm`, precisamos editar o arquivo “server.xml”. Se você estiver iniciando o Tomcat isoladamente, este arquivo se encontra na pasta “conf” do container, mas se estiver iniciando o Tomcat integrado ao Eclipse, você deve encontrar o arquivo na pasta “Servers” do seu workspace.



Encontre o elemento `Host`, onde `appBase` é igual a “webapps” e insira o a tag `Realm` com todos os atributos, conforme fragmento de código abaixo. Substitua o atributo `connectionURL`, `connectionName` e `connectionPassword` com a string de conexão, o usuário e a senha correta do seu banco de dados, respectivamente.

```
<Host appBase="webapps" autoDeploy="true" name="localhost" ...>

      <Realm className="org.apache.catalina.realm.JDBCRealm"
            driverName="com.mysql.jdbc.Driver"
            connectionURL="jdbc:mysql://localhost/financeiro"
            connectionName="usuario" connectionPassword="senha"
            userTable="usuario" userNameCol="nome_usuario"
            userCredCol="senha" userRoleTable="permissao_usuario"
            roleNameCol="nome_permissao"/>

      ...

</Host>
```

O módulo de login é executado em nível de servidor, e não da aplicação. Por isso devemos incluir o arquivo do driver JDBC do MySQL na pasta “lib” do Tomcat.



O domínio de segurança de usuários armazenados no banco de dados está configurado! Agora precisamos configurar a aplicação para integrar ao domínio de segurança.

18.4. Integrando a aplicação ao domínio de segurança

Para deixar nossa aplicação financeira segura, primeiramente precisamos editar o arquivo “web.xml” para incluir algumas configurações. A primeira delas é o método de autenticação, que usaremos, por enquanto, o básico. Inclua o seguinte fragmento de código no seu *deployment descriptor*:

```
<!-- Configurações de login -->
<login-config>
      <auth-method>BASIC</auth-method>
</login-config>
```

O método de autenticação básica deixa a cargo do browser do usuário o fornecimento das credenciais para efetuar o login, por isso o usuário visualizará uma tela do próprio navegador solicitando o usuário e senha. Apenas para conhecimento, existem outros métodos de autenticação, como *FORM*, *DIGEST* e *CLIENT-CERT*.

Novamente no arquivo "web.xml", precisamos incluir todas as possíveis *roles* (permissões) que o sistema pode usar. No sistema financeiro, estamos trabalhando apenas com as permissões "cadastro" e "consulta".

```
<!-- Regras (permissões) da aplicação -->
<security-role>
      <role-name>cadastro</role-name>
</security-role>
<security-role>
      <role-name>consulta</role-name>
</security-role>
```

Agora precisamos definir quais permissões são necessárias para acessar os recursos do sistema (páginas, imagens, relatórios, etc). Talvez essa seja a parte mais trabalhosa, pois é onde cada recurso é especificado e vinculado com as *roles*. Existe a opção de especificar um padrão de nome de recurso usando * (asterisco), como por exemplo, "/faces/contas/*", para todos os recursos iniciados por "/faces/contas/". Este não é o nosso caso, por isso devemos relacionar recurso por recurso.

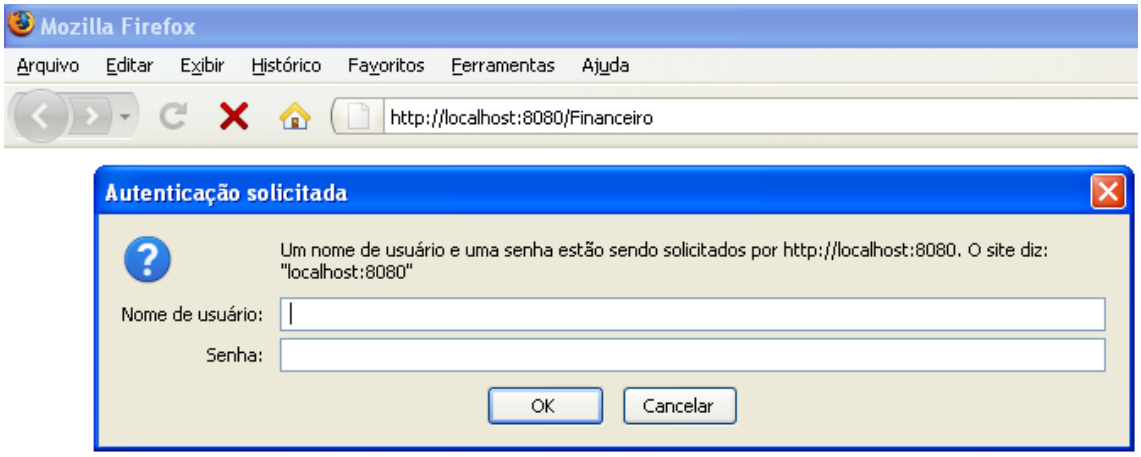
```
<!-- Mapeamento das permissões para os recursos web -->
<security-constraint>
   <web-resource-collection>
      <web-resource-name>Cadastro de Conta</web-resource-name>
      <url-pattern>/faces/contas/cadastroConta.jspx</url-pattern>
      </web-resource-collection>
      <auth-constraint>
            <role-name>cadastro</role-name>
      </auth-constraint>
</security-constraint>

<security-constraint>
   <web-resource-collection>
      <web-resource-name>Consulta de Conta</web-resource-name>
      <url-pattern>/faces/contas/consultaConta.jspx</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>consulta</role-name>
   </auth-constraint>
</security-constraint>

<!-- É necessário possuir permissão de cadastro ou consulta
     para acessar o menu do sistema -->
<security-constraint>
   <web-resource-collection>
      <web-resource-name>Página Inicial</web-resource-name>
      <url-pattern>/faces/menu.jspx</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>consulta</role-name>
      <role-name>cadastro</role-name>
   </auth-constraint>
</security-constraint>
```

Estamos quase no fim! A última coisa a ser feita agora é apenas um ajuste no arquivo "faces-config.xml". Como estamos protegendo alguns recursos, precisamos incluir a tag `<redirect/>` em alguns casos de navegação do JSF, pois sem esse elemento as páginas não são redirecionadas para a URL, mas apenas incluídas internamente pelo framework JSF, e neste caso, não são interceptadas pelo mecanismo de segurança do servidor. Portanto, edite o arquivo "faces-config.xml" e faça o ajuste.

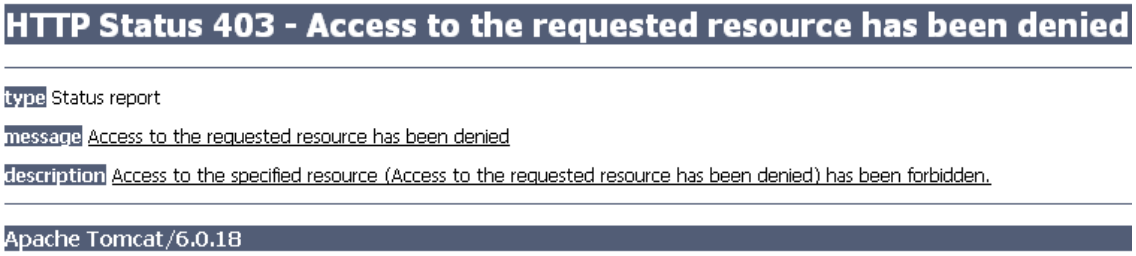
```
<navigation-rule>
	<navigation-case>
		<from-outcome>cadastroConta</from-outcome>
		<to-view-id>/contas/cadastroConta.jspx</to-view-id>
		<redirect/>
	</navigation-case>
	<navigation-case>
		<from-outcome>consultaConta</from-outcome>
		<to-view-id>/contas/consultaConta.jspx</to-view-id>
		<redirect/>
	</navigation-case>
</navigation-rule>
```

18.5. Validando a segurança

Todas as configurações necessárias já foram feitas, agora precisamos testar. Acesse o sistema e a tela de autenticação do browser deverá aparecer, solicitando o usuário e senha.



Se usarmos o usuário “joao", que possui acesso apenas para cadastros, e tentarmos acessar a tela de consulta, a tela de erro abaixo é apresentada:



Faça testes com todos os usuários e tente navegar em todo o sistema. Para efetuar logout, limpe os cookies de seu navegador ou feche-o e abra novamente.

18.6. Criando o formulário de login e personalizando telas de erro

A tela de login e as telas de erro de segurança (usuário/senha inválidos e acesso negado) podem ser personalizadas para ficar com a “cara” da sua aplicação. Vamos aprender agora como fazer isso, mas criaremos telas simples, sem nenhum estilo CSS, pois nossa intenção é apenas mostrar como funciona.

Crie um arquivo chamado “login.jsp” na pasta “WebContent” com o conteúdo abaixo:

```
<%@ page language="java"
      contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"%>
      <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
      Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
      <title>Login do sistema financeiro</title>
</head>
<body>
      <form action="j_security_check" method="POST">
            Usuário: <input type="text" name="j_username"/><br/>
            Senha: <input type="password"
                        name="j_password"/><br/>
            <input type="submit" value="Acessar"/>
      </form>
</body>
</html>
```

Este arquivo será chamado sempre que um usuário tentar acessar a aplicação e ainda não estiver logado.

A action “j_security_check” e os nomes dos campos “j_username” e “j_password” são obrigatórios, pois só assim o servidor conseguirá identificar que a requisição é uma tentativa de login usando JAAS e conseguirá obter os dados digitados.

Crie agora um arquivo chamado “falhaLogin.jsp”, também na raíz web do projeto, e digite o código-fonte a seguir:

```
<%@ page language="java"
      contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"%>
      <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
      Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
      <title>Falha no login</title>
</head>
<body>
      <h1>Usuário e/ou senha não conferem!</h1>
      <a href="javascript:history.back();">Voltar</a>
</body>
</html>
```

Esta página será exibida sempre que um usuário fornecer as informações de login inválidas.

Finalmente, crie o arquivo “acessoNegado.jsp” com o conteúdo:

```
<%@ page language="java"
      contentType="text/html; charset=ISO-8859-1"
      pageEncoding="ISO-8859-1"%>
      <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
      Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
      <title>Acesso negado</title>
</head>
<body>
      <h1>O recurso que você tentou acessar está protegido
          e você<br/>não possui privilégios para acessá-lo!</h1>
```

```
        <a href="javascript:history.back();">Voltar</a>
</body>
</html>
```

Esta página será exibida na tentativa de acesso indevido de um recurso protegido, ou seja, quando o usuário não possuir acesso a uma determinada página.

Agora é necessário dizer à aplicação o significado de cada arquivo criado. Para isso, edite novamente o arquivo "web.xml". Altere o conteúdo da tag `login-config` para o seguinte:

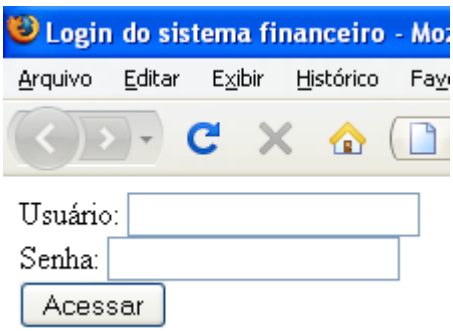
```
<!-- Configurações de login -->
<login-config>
        <auth-method>FORM</auth-method>
        <form-login-config>
                <form-login-page>/login.jsp</form-login-page>
                <form-error-page>/falhaLogin.jsp</form-error-page>
        </form-login-config>
</login-config>
```

Veja que alteramos o método de autenticação para *FORM*. Isso quer dizer que existe um formulário de login, e não é mais para ser usada a autenticação básica, implementada pelo navegador. Os caminhos do formulário de login e da página de erro exibida quando o usuário e/ou senha não conferem são especificados nos elementos `form-login-page` e `form-error-page`, respectivamente.

Para configurar a página de erro exibida quando o usuário tenta acessar um recurso protegido sem ter permissão, inclua o fragmento de código abaixo no seu "web.xml".

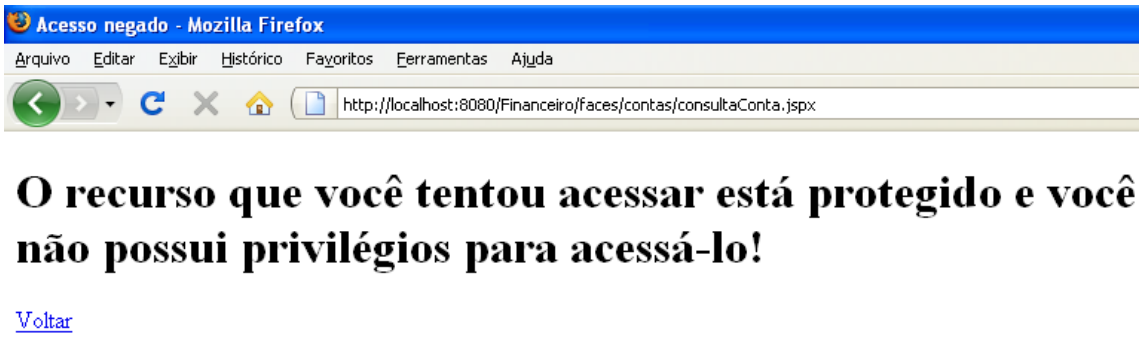
```
<error-page>
        <error-code>403</error-code>
        <location>/acessoNegado.jsp</location>
</error-page>
```

Pronto! Agora, ao tentar acessar um recurso protegido sem que você esteja logado, aparecerá o formulário de login que criamos.



Se o usuário e/ou senha foram informados incorretamente no formulário de login, veremos a seguinte página:

Se tentarmos acessar uma tela sem termos permissão, veremos a tela a seguir:

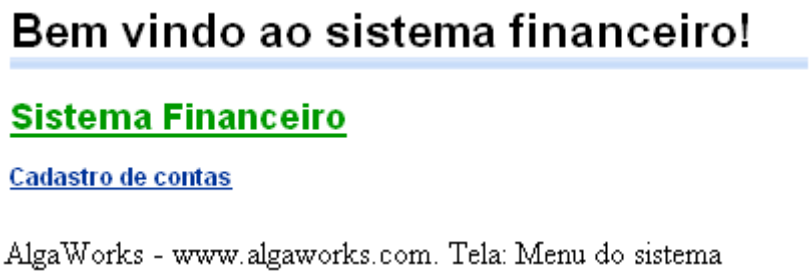


18.7. Protegendo componentes contra ações dos usuários

Quando o usuário loga no sistema financeiro, o menu do sistema apresenta os links para o cadastro e consulta de contas, mesmo que o usuário não tenha permissão para esses links. Seria interessante ocultar ou desabilitar os links caso o usuário não possa acessá-los. O RichFaces possui uma função chamada `rich:isUserInRole()`, que, dado um nome de permissão (*role*), retorna um valor boleano indicando se o usuário logado possui o privilégio. Essa função pode ser usada dentro de uma expressão no atributo `rendered` dos componentes, como no exemplo abaixo.

```
<h:commandLink action="#{cadastroContaBean.inicializar}"
      rendered="#{rich:isUserInRole('cadastro')}">
      <h:outputText value="Cadastro de contas"/>
</h:commandLink>
<br/>
<h:commandLink action="consultaConta"
      actionListener="#{consultaContaBean.consultar}"
      rendered="#{rich:isUserInRole('consulta')}">
      <h:outputText value="Consulta de contas"/>
</h:commandLink>
```

Agora, quando um usuário que não possui permissão para consultar contas acessar o sistema, por exemplo, não visualizará mais o link “Consulta de contas”.



A função `rich:isUserInRole()` pode ser usada em qualquer componente. Por exemplo, você poderia desabilitar um `h:commandButton` usando o atributo `disabled` se o usuário não possuir a permissão adequada ou ocultar uma coluna de uma tabela de dados que possui uma informação restrita usando o atributo `rendered` do componente `h:column` ou `rich:column`.

18.8. Exibindo o nome do usuário logado e criando um link para logout

Os usuários gostam de ser chamados pelo nome, por isso, aprenderemos agora como incluir o nome do usuário logado na mensagem de boas vindas do sistema. Editamos o arquivo "template.jspx" e alteramos o valor do `outputText`, como pode ver no código abaixo.

```
<h:outputText value="Bem vindo ao sistema financeiro,
#{facesContext.externalContext.userPrincipal.name}!"
style="font-weight: bold; font-size: 17pt; font-family: Arial"/>
```

Agora quando o João acessar o menu do sistema, que utiliza o arquivo "template.jspx", veja a mensagem de boas vindas apresentada.

Para criar a funcionalidade de logout da aplicação, criamos um arquivo JSP chamado "logout.jsp" com o código-fonte abaixo:

```
<%
	session.invalidate();
	response.sendRedirect("/Financeiro");
%>
```

O código acima invalida (limpa) a sessão do usuário e o redireciona para a tela inicial do sistema, que estará protegida (e o usuário estará deslogado) e encaminhará o usuário para a tela de login.

Precisamos também incluir um link para o usuário sair (efetuar logout) do sistema. Para isso, basta adicionarmos um `h:outputLink` na tela de menu do sistema.

```
<h:outputLink value="/Financeiro/logout.jsp">
	<h:outputText value="Sair"/>
</h:outputLink>
```

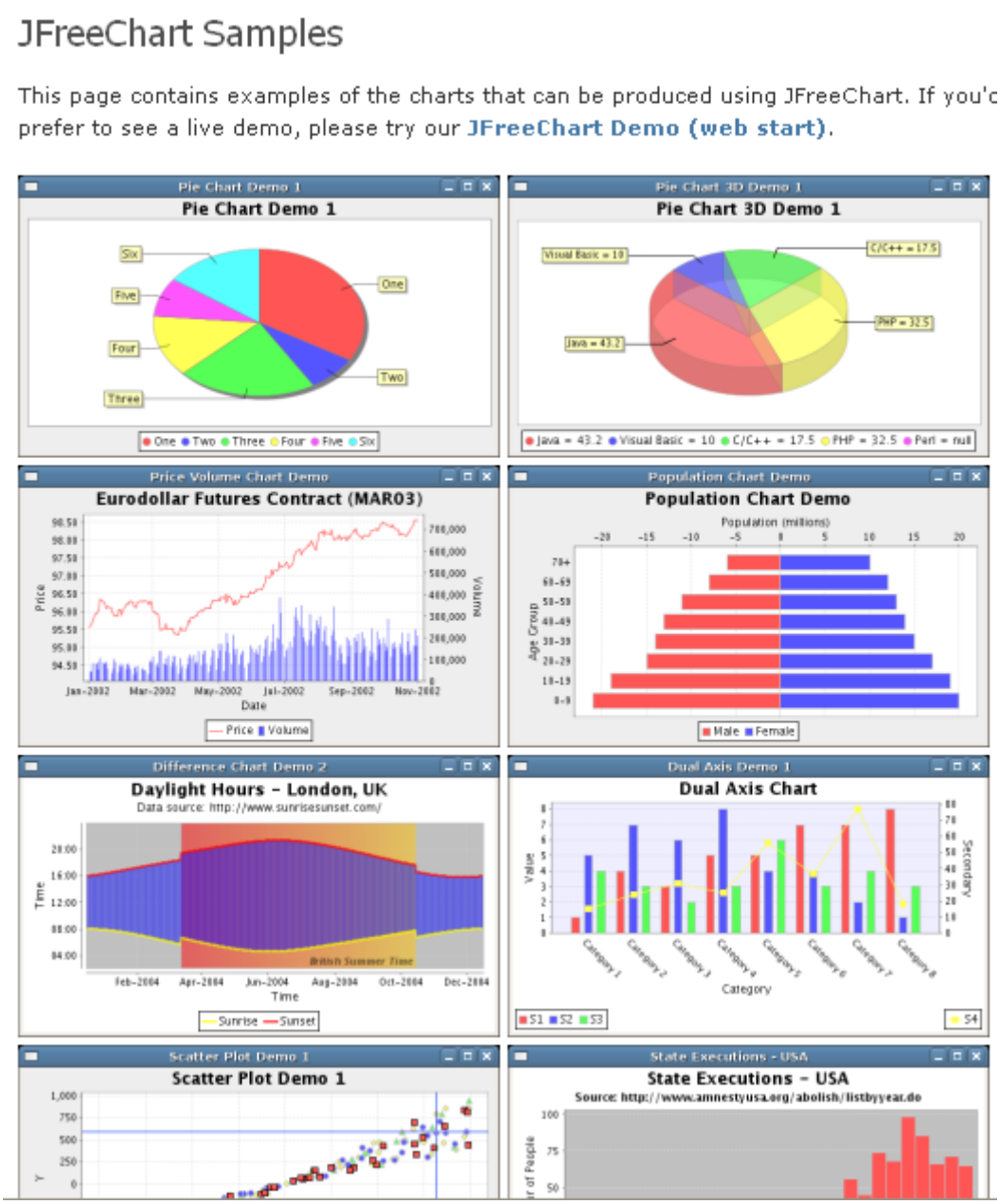
19. Conhecendo o JFreeChart

19.1. Introdução

JFreeChart é uma biblioteca *open source* (grátis) escrita 100% em Java que possibilita a criação de gráficos em aplicações desktop ou web de forma rápida, fácil, flexível e profissional.

Com JFreeChart, é possível gerar gráficos para diversas saídas, como por exemplo para uma tela desktop (Swing), arquivos de imagens (PNG, GIF e JPG) e formatos vetorizados (PDF, EPS e SVG).

O JFreeChart pode ser usado para gerar gráficos de área, linha, pizza, bolha, gantt, etc. Todos os tipos de gráficos possibilitam customização de quase tudo, como legendas, bordas, cores, fontes, etc. Para conhecer um pouco o que é possível fazer com esta biblioteca, acesse o endereço www.jfree.org/jfreechart/samples.html ou veja alguns exemplos na imagem abaixo.

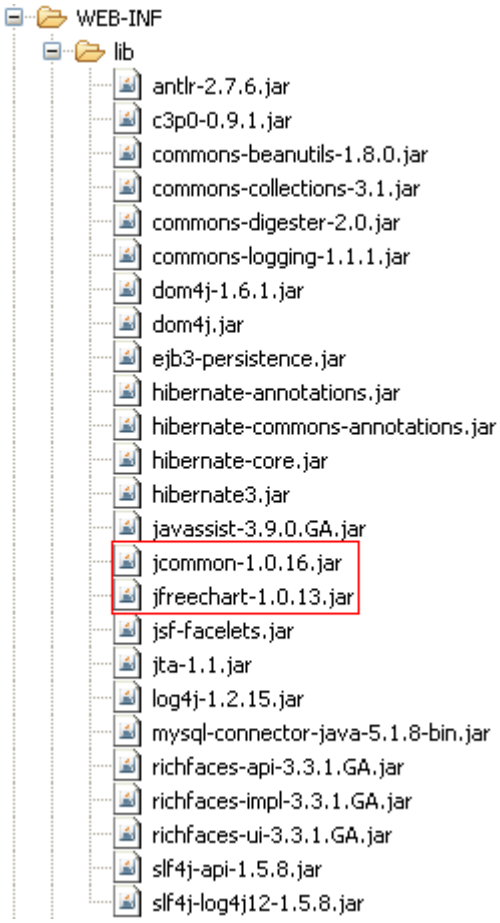


O site oficial do JFreeChart é www.jfree.org/jfreechart. Neste endereço, pode ser baixada a última versão da biblioteca.

19.2. Criando gráficos em páginas JSF

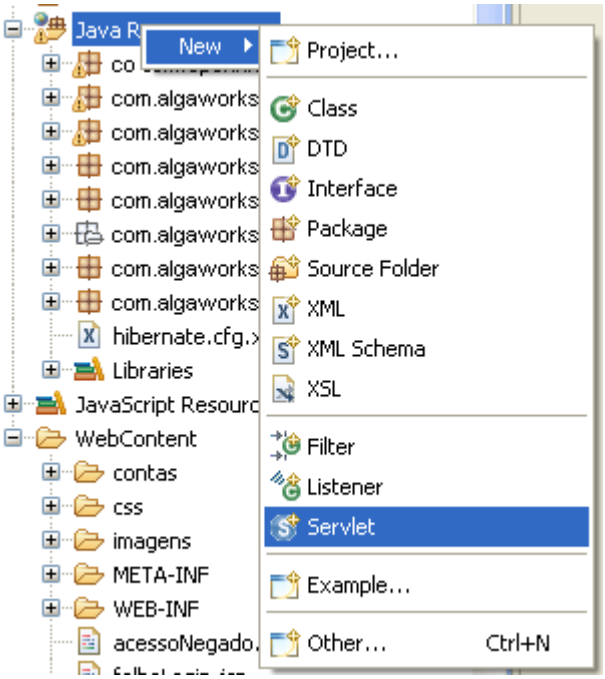
Para demonstrar o uso do JFreeChart com JSF, criaremos um gráfico de pizza 3D que exibirá 2 fatias, sendo que uma representará o valor total de despesas e a outra o valor total de receitas inseridas no sistema financeiro.

Antes de começar a programação, precisamos incluir as bibliotecas necessárias na pasta "WEB-INF/lib". Copie os arquivos "jcommon-1.0.16.jar" e "jfreechart-1.0.13.jar" do DVD para esta pasta.

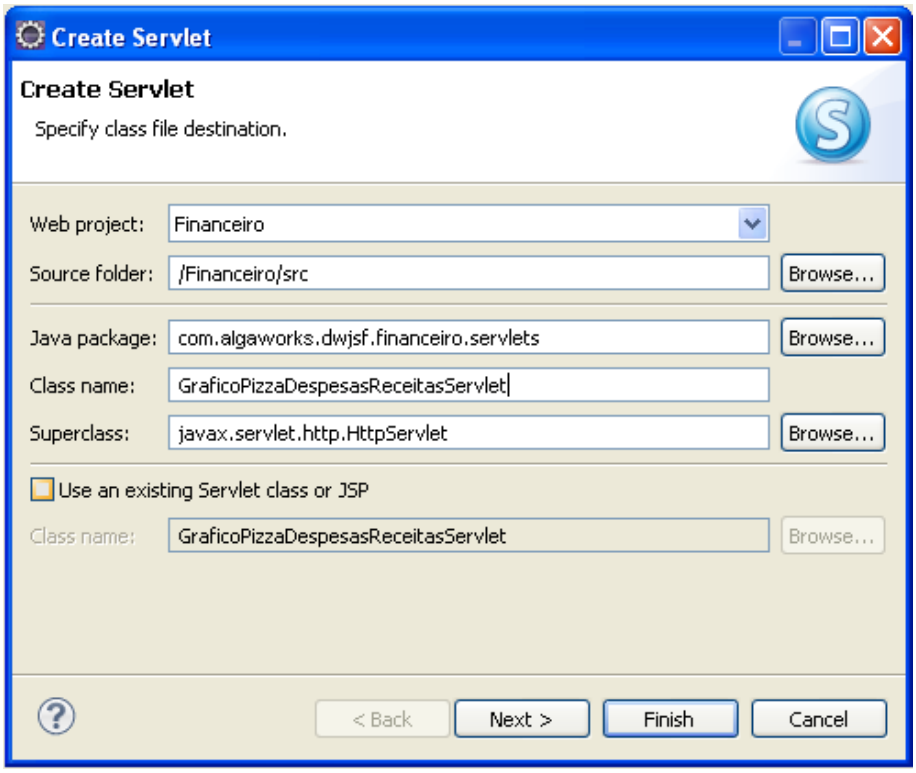


A programação do gráfico ficará em um Servlet, que consultará os valores no banco de dados através da classe de negócio `ContaService` e renderizará uma imagem no formato PNG para representar os dados obtidos.

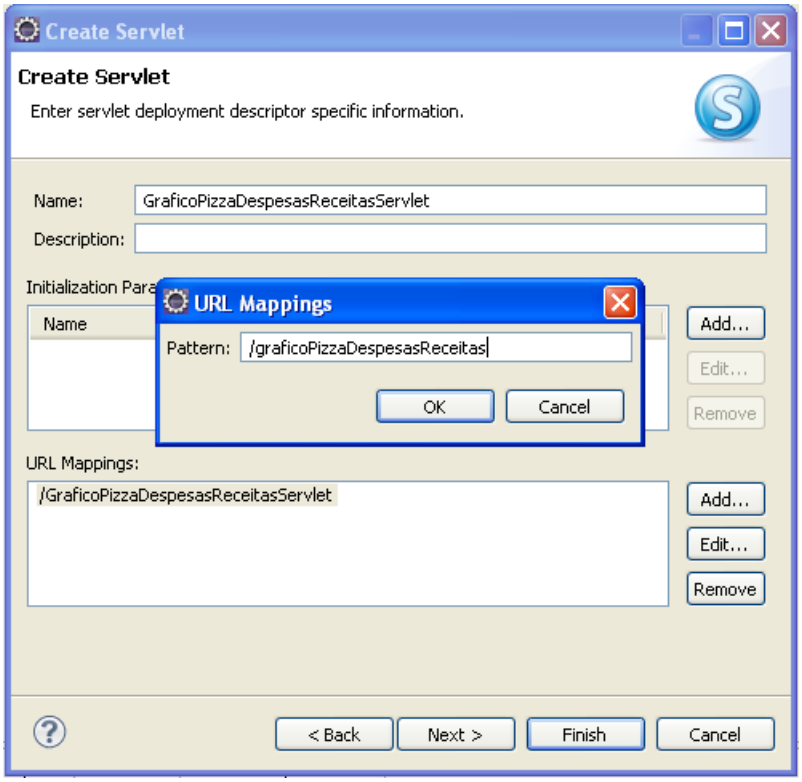
Para criar o Servlet no Eclipse, clique com o botão direito no projeto, selecione "New" e depois "Servlet".



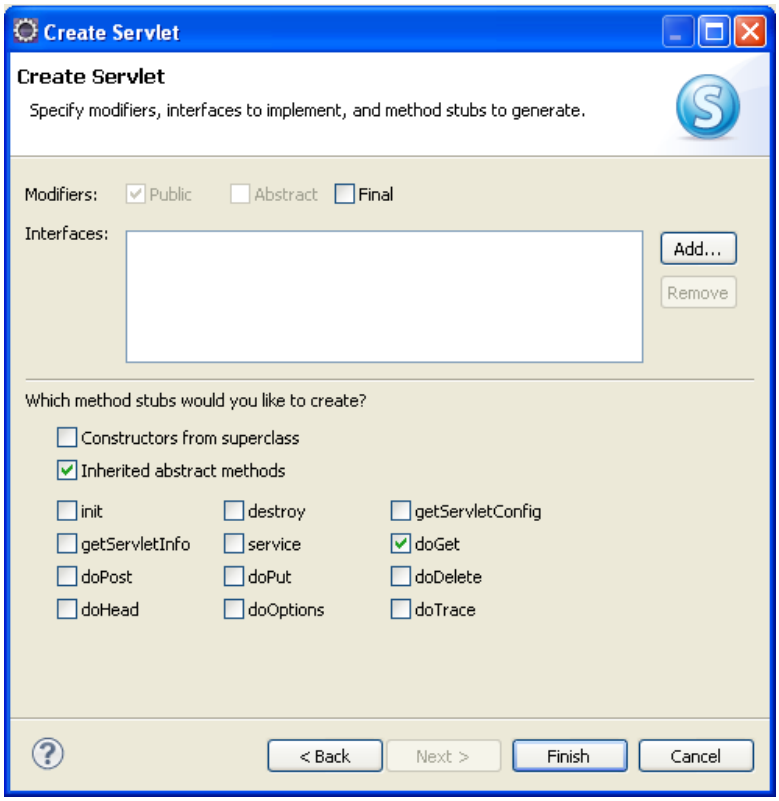
Digite o nome do pacote e da classe do Servlet. Nosso servlet terá o nome `GraficoPizzaDespesasReceitasServlet`. Clique em "Next".



Edite o mapeamento de URL e informe "/graficoPizzaDespesasReceitas". Clique em "Next".



Selecione apenas o método `doGet()` para ser declarado no código-fonte e pressione o botão “Finish”.



Digite o código-fonte abaixo na classe do Servlet.

```
package com.algaworks.dwjsf.financeiro.servlets;

import java.awt.Color;
import java.awt.image.BufferedImage;

import javax.imageio.ImageIO;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.JFreeChart;
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.data.general.PieDataset;

//vários outros imports

public class GraficoPizzaDespesasReceitasServlet extends HttpServlet {

	private static final long serialVersionUID = 1L;

	protected void doGet(HttpServletRequest request,
		HttpServletResponse response)
			throws ServletException, IOException {
		response.setContentType("image/png");
		OutputStream output = response.getOutputStream();

		BufferedImage image = criarGrafico(criarDataSet())
			.createBufferedImage(650, 250);
		ImageIO.write(image, "png", output);

		output.close();
	}

	private PieDataset criarDataSet() {
		NumberFormat formatador = DecimalFormat
			.getNumberInstance(new Locale("pt", "BR"));
		ContaService contaService = new ContaService();
		BigDecimal valorDespesas = contaService
			.consultarValorTotalPorTipoConta(TipoConta.DESPESA);
		BigDecimal valorReceitas = contaService
			.consultarValorTotalPorTipoConta(TipoConta.RECEITA);

		DefaultPieDataset dataSet = new DefaultPieDataset();

		dataSet.setValue("Despesas [" + formatador
			.format(valorDespesas) + "]", valorDespesas);
		dataSet.setValue("Receitas [" + formatador
			.format(valorReceitas) + "]", valorReceitas);
		return dataSet;
	}

	private JFreeChart criarGrafico(PieDataset dataSet) {
		JFreeChart chart = ChartFactory
			.createPieChart3D("Receitas x Despesas",
				dataSet, false, true, false);
		PiePlot3D plot = (PiePlot3D) chart.getPlot();
		plot.setBackgroundPaint(Color.WHITE);
		plot.setStartAngle(290);
		plot.setForegroundAlpha(0.5f);
		return chart;
```

```
        }

}
```

O método `criarDataSet()` apenas consulta os valores no banco de dados através da classe de negócio e preenche um objeto do tipo `DefaultPieDataset`, que é um conjunto de dados que será usado para renderizar o gráfico.

O método `criarGrafico()` cria o gráfico passando alguns parâmetros, como o título do gráfico, opções de apresentação, como por exemplo, se deseja mostrar a legenda do gráfico, rótulos de dicas e etc, além do próprio conjunto de dados (*dataset*). Neste mesmo método, alteramos a cor de fundo da área do gráfico para branco e especificamos o ângulo e a transparência.

O método `doGet()`, que responde requisições HTTP do tipo *GET*, altera o tipo de saída para PNG, chama os métodos `criarDataSet()` e `criarGrafico()` e transforma o objeto do gráfico em uma imagem redimensionada, que é passada para o fluxo de saída (*output stream*) da resposta do Servlet.

No arquivo “menu.jspx”, incluímos o fragmento de código abaixo:

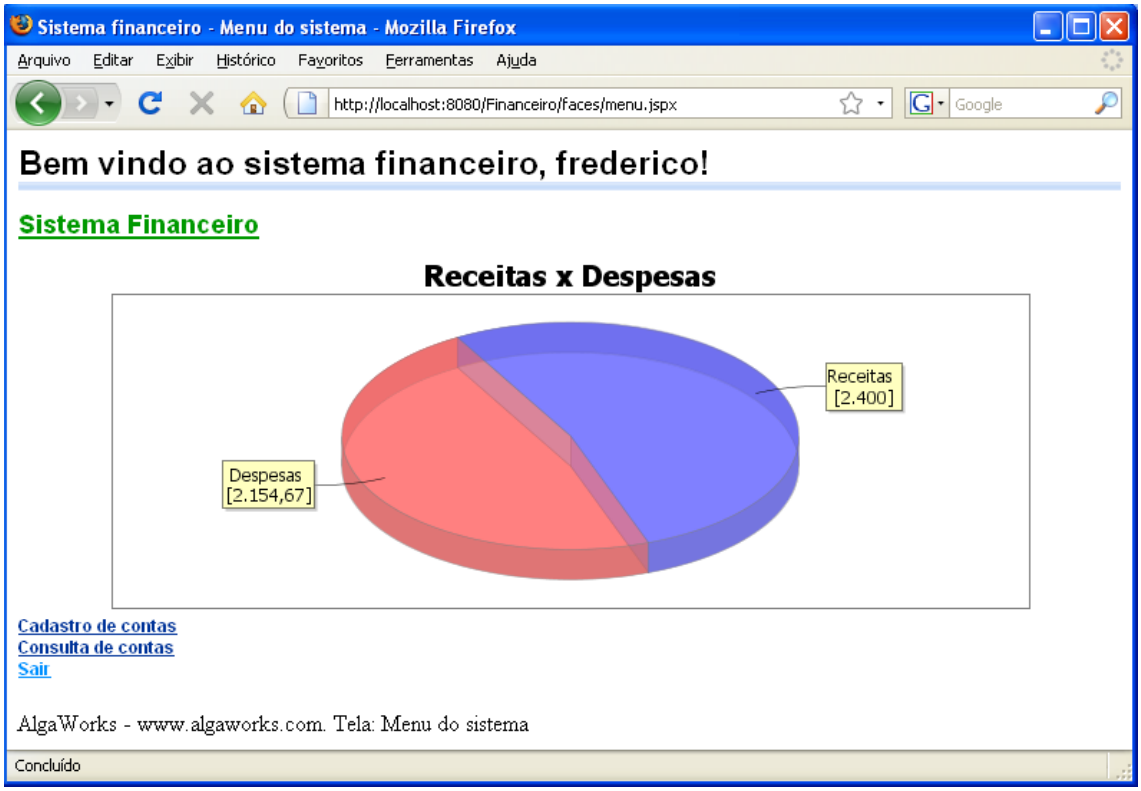
```
<h:panelGroup style="text-align: center" layout="block"
              rendered="#{rich:isUserInRole('consulta')}">
      <h:graphicImage value="/graficoPizzaDespesasReceitas"/>
      <br/>
</h:panelGroup>
```

Veja que incluímos um componente que renderiza uma imagem, referenciando a URL que mapeamos para o Servlet. Colocamos um painel envolvendo a imagem para incluir o estilo CSS de alinhamento e também checagem de segurança no atributo `rendered`, pois se o usuário não possuir permissão de consulta, a imagem não poderá ser exibida.

A verificação de segurança que colocamos no painel não é suficiente, pois o usuário pode usar a esperteza e digitar a URL do gráfico diretamente no navegador, por isso, protegemos este recurso no arquivo “web.xml”, como você já aprendeu no capítulo anterior.

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>Consulta de Conta</web-resource-name>
      <url-pattern>/faces/contas/consultaConta.jspx</url-pattern>
    </web-resource-collection>
    <web-resource-collection>
      <web-resource-name>Gráf. Desp. x Rec.</web-resource-name>
      <url-pattern>/graficoPizzaDespesasReceitas</url-pattern>
    </web-resource-collection>
    <auth-constraint>
       <role-name>consulta</role-name>
    </auth-constraint>
</security-constraint>
```

Achou fácil? Nosso trabalho chegou ao fim. Acesse o sistema e veja o gráfico na tela de menu do sistema.


Sistema financeiro - Menu do sistema - Mozilla Firefox
Arquivo Editar Exibir Histórico Favoritos Ferramentas Ajuda
http://localhost:8080/Financeiro/faces/menu.jspx
Google
Bem vindo ao sistema financeiro, frederico!
Sistema Financeiro
Receitas x Despesas
Receitas [2.400]
Despesas [2.154,67]
Cadastro de contas
Consulta de contas
Sair
AlgaWorks - www.algaworks.com. Tela: Menu do sistema
Concluído

20. Conclusão

O objetivo do curso **DWJSF (Desenvolvimento Web com JavaServer Faces)** foi de apresentar a tecnologia JavaServer Faces e a integração com vários outros frameworks, bibliotecas e especificações, como Hibernate, Richfaces, Facelets, JAAS, JFreeChart e etc. O conhecimento que você adquiriu é suficiente para que você possa desenvolver sistemas completos, com alta qualidade e produtividade, de acordo com o que o mercado precisa.

A AlgaWorks possui diversos outros cursos Java e de tecnologias relacionadas. Acesse nosso endereço na internet: **http://www.algaworks.com**.

Foi um prazer contribuir para o aperfeiçoamento de seu conhecimento.

21. Bibliografia

1. http://www.oracle.com/technology/tech/java/newsletter/articles/introjsf/index.html
2. http://www.guj.com.br/content/articles/jsf/jsf.pdf
3. http://www.ibm.com/developerworks/library/j-jsf2/
4. http://www.ibm.com/developerworks/library/j-jsf3/
5. http://www.jsftoolbox.com/documentation/
6. http://evandropaes.wordpress.com/2007/05/11/integrando-jsf-e-css-%E2%80%93-aplicando-estilos-ao-datatable/
7. http://docs.jboss.org/richfaces/latest_3_3_X/en/devguide/html/
8. http://pt.wikipedia.org/wiki/XHTML
9. http://www.w3schools.com/tags/tag_DOCTYPE.asp
10. https://facelets.dev.java.net/nonav/docs/dev/docbook.html
11. http://tomcat.apache.org/tomcat-5.5-doc/realm-howto.html
12. Core JavaServer Faces, O Guia Autorizado; Geary, David; Horstmann, Cay; Alta Books
13. Pro JSF e Ajax, Construindo Componentes Ricos para a Internet; Jacobi, Jonas; Fallows, John R.; Ciência Moderna

22. Fique atualizado!

Além de praticar todo o conteúdo deste curso, é importante você estar sempre atualizado com as novidades do mercado, por isso, indicamos que você acesse regularmente o **Blog e Twitter da AlgaWorks**, o portal de nosso parceiro **JavaFree** e se possível, assine ou compre exemplares da revista **MundoJ**, também parceira da AlgaWorks. Através destes canais, você pode ler artigos interessantes, participar de fóruns de discussão e contribuir para a comunidade Java.

Blog da AlgaWorks
http://www.algaworks.com/blog

No blog da AlgaWorks, você fica atualizado em relação aos principais trabalhos da empresa, turmas e promoções de treinamentos e artigos técnicos, escritos pelos instrutores e consultores da empresa.

Twitter da AlgaWorks
http://twitter.com/algaworks

No Twitter da AlgaWorks, você fica muito mais próximo do time de profissionais da empresa, e mantém atualizado sobre todas as promoções de treinamentos e atividades do dia-a-dia da empresa.

JavaFree – O melhor conteúdo Java
http://www.javafree.org

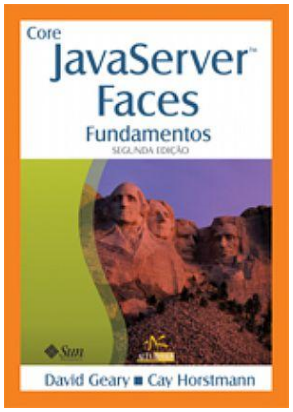
O JavaFree é considerado um dos maiores portais sobre Java do mundo, e tem como objetivo desenvolver o Java e o Software Livre nos países de língua portuguesa, com artigos, tutoriais, entrevistas, ofertas de empregos, fórum, entre outros.

MundoJ – A revista para quem gosta de desenvolver software
http://www.mundoj.com.br

A revista MundoJ é uma das mais importantes na área de desenvolvimento de software no Brasil, com edições bimestrais, que abordam assuntos como Java, SOA, agile, design, linguagens e arquitetura.

Assine a MundoJ através da parceria que temos com esta revista e ganhe 10% de desconto. Acesse http://www.algaworks.com/assine-mundoj.

23. Livros recomendados



Core JavaServer Faces
Autor: Geary, David / Horstmann, Cay
Editora: Alta Books
ISBN: 8-576081-60-1
Idioma: Português
Páginas: 544
Publicação: 2007
Edição: 2

Java Persistence with Hibernate
Autor: Bauer, Christian / King, Gavin
Editora: Manning
ISBN: 1-932394-88-5
Idioma: Inglês
Páginas: 880
Publicação: 2006
Edição: 2