THIAGO FARIA

JAVA EE 7

Com JSF, PrimeFaces e CDI

algaworks

Java EE 7 com JSF, PrimeFaces e CDI
por Thiago Faria

Edição de 24/12/2013

AlgaWorks Softwares, Treinamentos e Serviços Ltda
www.algaworks.com
contato@algaworks.com
+55 (11) 3509-3100

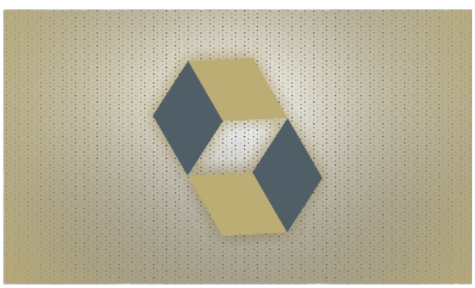
algaworks

CURSOS ONLINE

COM VÍDEO AULAS E SUPORTE

Fundamentos Java e
Orientação a Objetos

Desenvolvimento Web
com JSF 2



Persistência de Dados com
JPA 2 e Hibernate

Sistemas Comerciais Java EE
com CDI, JPA e PrimeFaces

www.**algaworks**.com

Cursos presenciais in-company? Entre em contato.

Sobre o autor

Thiago Faria de Andrade

@ThiagoFAndrade

Fundador, instrutor e consultor da AlgaWorks. Graduado em Sistemas de Informação e certificado como programador Java pela Sun. Iniciou seu interesse por programação em 1995, quando desenvolveu um software para entretenimento e se tornou um dos mais populares no Brasil e outros países de língua portuguesa. Já foi sócio e trabalhou em outras empresas de software como programador, gerente e diretor de tecnologia, mas nunca deixou de programar.

Sumário

Capítulo 1

Introdução ao desenvolvimento web

1.1. O que é Java EE?

A Java EE (Java Platform, Enterprise Edition) é uma plataforma padrão para desenvolver aplicações Java de grande porte e/ou para a internet, que inclui bibliotecas e funcionalidades para implementar software Java distribuído, baseado em componentes modulares que executam em servidores de aplicações e que suportam escalabilidade, segurança, integridade e outros requisitos de aplicações corporativas ou de grande porte.

A plataforma Java EE possui uma série de especificações (tecnologias) com objetivos distintos, por isso é considerada uma plataforma guarda-chuva. Entre as especificações da Java EE, as mais conhecidas são:

- Servlets: são componentes Java executados no servidor para gerar conteúdo dinâmico para a web, como HTML e XML.
- JSP (JavaServer Pages): uma especialização de Servlets que permite que aplicações web desenvolvidas em Java sejam mais fáceis de manter. É similar às tecnologias como ASP e PHP, porém mais robusta por ter todas as facilidades da plataforma Java.
- JSF (JavaServer Faces): é um framework web baseado em Java que tem como objetivo simplificar o desenvolvimento de interfaces (telas) de sistemas para a web, através de um modelo de componentes reutilizáveis. A proposta é que os sistemas sejam desenvolvidos com a mesma facilidade e produtividade

que se desenvolve sistemas desktop (até mesmo com ferramentas que suportam clicar-e-arrastar componentes).

- JPA (Java Persistence API): é uma API padrão do Java para persistência de dados, que usa um conceito de mapeamento objeto-relacional. Essa tecnologia traz alta produtividade para o desenvolvimento de sistemas que necessitam de integração com banco de dados. Só para citar, essa API possibilita que você desenvolva aplicações usando banco de dados sem precisar escrever uma linha sequer de SQL.

- EJB (Enterprise Java Beans): são componentes que executam em servidores de aplicação e possuem como principais objetivos, fornecer facilidade e produtividade no desenvolvimento de componentes distribuídos, transacionados, seguros e portáveis.

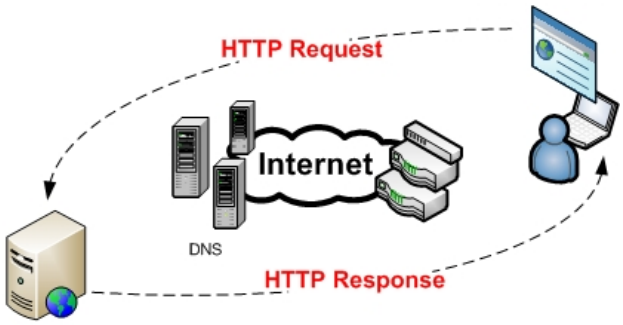
Neste livro, abordaremos sobre JSF e uma breve introdução de JPA.

1.2. O protocolo HTTP

O protocolo HTTP é utilizado na navegação de páginas da Internet. Quando você abre uma janela de um *browser*, acessa uma página Web e navega em seus links, você está, na verdade, utilizando esse protocolo para visualizar, em sua máquina, o conteúdo que está armazenado e/ou é processado em servidores remotos.

O HTTP é um protocolo *stateless* de comunicação cliente-servidor: o cliente envia uma requisição para o servidor, que processa a requisição e devolve uma resposta para o cliente, sendo que, a princípio, nenhuma informação é mantida no servidor em relação às requisições previamente recebidas.

Assim, quando digitamos o endereço de uma página em um *browser*, estamos gerando uma requisição a um servidor, que irá, por sua vez, devolver para o *browser* o conteúdo da página HTML requisitada.



A requisição enviada por um cliente deve conter, basicamente, um comando (também chamado de método), o endereço de um recurso no servidor (também chamado de *path*) e uma informação sobre a versão do protocolo HTTP sendo utilizado.

Supondo, por exemplo, que queremos buscar o conteúdo do endereço *http://www.uol.com.br/index.html*. Utilizemos o método *GET*, o *path* `/index.html` e a versão `1.1` do protocolo HTTP. Temos a seguinte requisição enviada:

```
GET /index.html HTTP/1.1
Host: www.uol.com.br
```

Existem diversos métodos HTTP que podem ser especificados em requisições, sendo os mais comuns o método *GET*, normalmente utilizado para obter o conteúdo de um arquivo no servidor, e o método *POST*, utilizado para enviar dados de formulários HTML ao servidor.

Uma requisição pode conter parâmetros adicionais, chamados *headers*. Alguns *headers* comuns são, por exemplo, *Host*, *User-Agent* e *Accept*.

Uma vez processada a requisição, o servidor, por sua vez, manda uma resposta para o cliente, sendo que essa resposta também tem um formato pré-determinado: a primeira linha contém informações sobre a versão do protocolo, um código de status da resposta e uma mensagem associada a esse status. Em seguida, são enviados os *headers* da resposta, e finalmente, é enviado o conteúdo da resposta. Veja um exemplo simples de resposta HTTP:

```
HTTP/1.1 200 OK
Date: Thu, 26 Sep 2013 15:17:12 GMT
Server: Apache/2.2.15 (CentOS)
Content-Type: text/html; charset=utf-8

<html>
    <body>
```

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

No exemplo anterior, o código de status `200` indica que houve sucesso no atendimento da requisição enviada pelo cliente, e os *headers* indicam a data e hora do servidor, o servidor usado, tipo do conteúdo e, por fim, temos o código-fonte da página HTML.

Outros códigos de status bastante comuns são o `404`, que indica que o recurso não foi localizado no servidor e o código `500`, que indica que houve erro no processamento da requisição enviada.

1.3. Desenvolvimento web com Java

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.

Quando falamos em aplicações web, estamos nos referindo a sistemas ou sites onde grande parte da programação fica hospedada em servidores na internet, e o usuário (cliente) normalmente não precisa ter nada instalado em sua máquina para utilizá-las, além de um navegador (browser).

O acesso às páginas desses sistemas é feita utilizando o 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 através da API de Servlets. Um Servlet estende a funcionalidade de um servidor web para servir páginas dinâmicas aos navegadores, utilizando o protocolo HTTP.

No mundo Java, os servidores web são chamados de Servlet Container, pois implementam a especificação de Servlet. O servidor converte a requisição em um objeto do tipo `HttpServletRequest`. Este objeto é então passado aos componentes web, que podem executar qualquer código Java para que possa ser gerado um conteúdo dinâmico. Em seguida, o componente web devolve um objeto `HttpServletResponse`, que representa a resposta ao cliente. Este objeto é utilizado para que o conteúdo gerado seja enviado ao navegador do usuário.

Desde o lançamento de Servlets, 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. JSF, assim como os outros frameworks web, foram baseados em Servlets.

1.4. Containers

Containers são interfaces entre componentes e funcionalidades de baixo nível específicas de uma plataforma. Para uma aplicação web desenvolvida em Java ou um componente corporativo ser executado, eles precisam ser implantados em um container.

Os containers também são chamados de servidores de objetos, ou servidores de aplicação, pois oferecem serviços de infra-estrutura para execução de componentes.

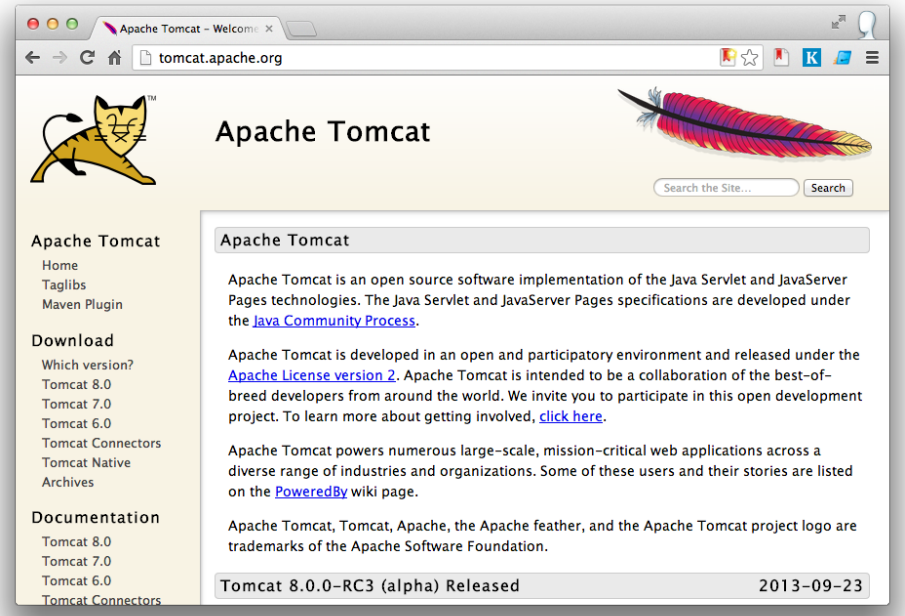
O EJB Container suporta Enterprise JavaBeans (EJB), que são componentes corporativos distribuídos. Os Servlets, JSP, páginas JSF e arquivos estáticos (HTML, CSS, imagens e etc) necessitam de um Web Container para ser executado.

Existem diversas organizações que desenvolvem containers Java EE, por exemplo: Oracle, IBM, Red Hat, Apache, etc. Apesar de tantas ofertas gratuitas, 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.

Para testar nossos exemplos, usaremos o **Apache Tomcat**, pois é leve, gratuito e muito popular.

Como estes servidores são baseados nas especificações da tecnologia Java EE, teoricamente, você pode implantar os exemplos que desenvolveremos neste livro em qualquer container compatível com a Java EE.

O download do Apache Tomcat pode ser feito em http://tomcat.apache.org.



1.5. Instalando o Apache Tomcat

O processo de instalação do Apache Tomcat é simples, basta descompactar o arquivo baixado no local desejado. Neste livro, usaremos o **Tomcat 7.0**.

Uma vez finalizado, tem-se um container 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. O Tomcat também pode gerar arquivos de log com tudo o que as aplicações desenvolvidas enviam para a saída padrão do sistema.

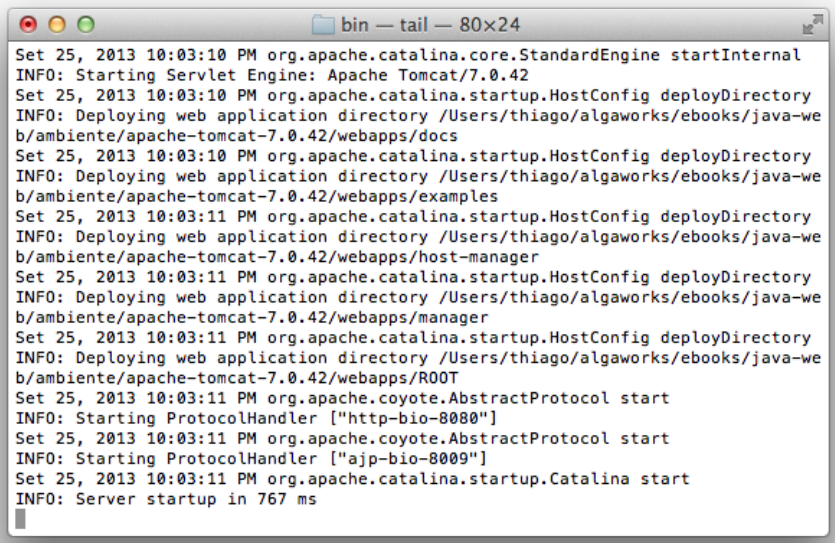
- **work**: Diretório temporário do Tomcat. Esse diretório é utilizado, por exemplo, para realizar a recompilação automática de páginas JSP.

- **webapps**: Nesse diretório são implantadas as diversas aplicações web desenvolvidas.

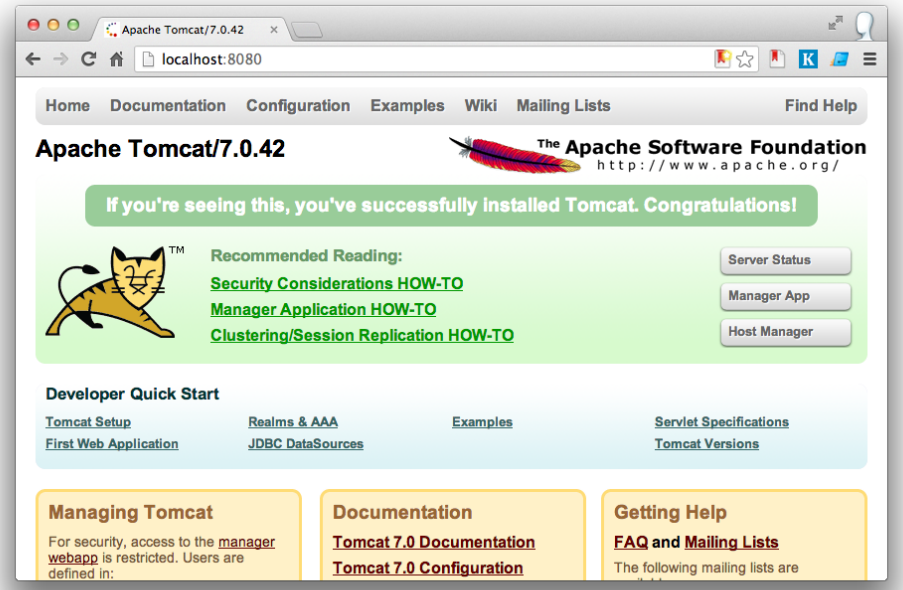
Para verificar se a instalação foi bem sucedida, no Windows, acesse o diretório *bin* e execute o arquivo *startup.bat* para iniciar o servidor. Você já deve ter a JDK (Java Development Kit) instalado em seu computador para executar este passo.

No Linux ou Mac, acesse o diretório *bin* do Tomcat e digite:

```
$ chmod +x *.sh
$ ./startup.sh; tail -f ../logs/catalina.out
```



Abra um *browser* e acesse o endereço **http://localhost:8080**. Se a tela abaixo aparecer para você, parabéns, o Tomcat está instalado e funcionando em seu computador.



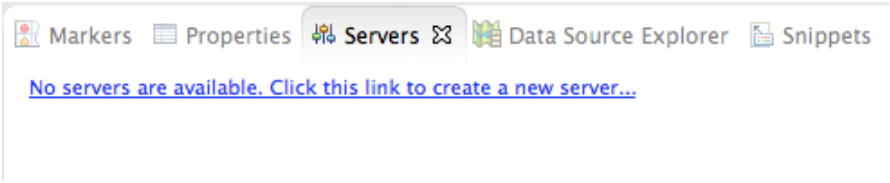
Para interromper a execução do Tomcat, execute o arquivo *shutdown.bat* (Windows) ou *shutdown.sh* (Linux ou Mac).

1.6. Integrando o Eclipse com o Apache Tomcat

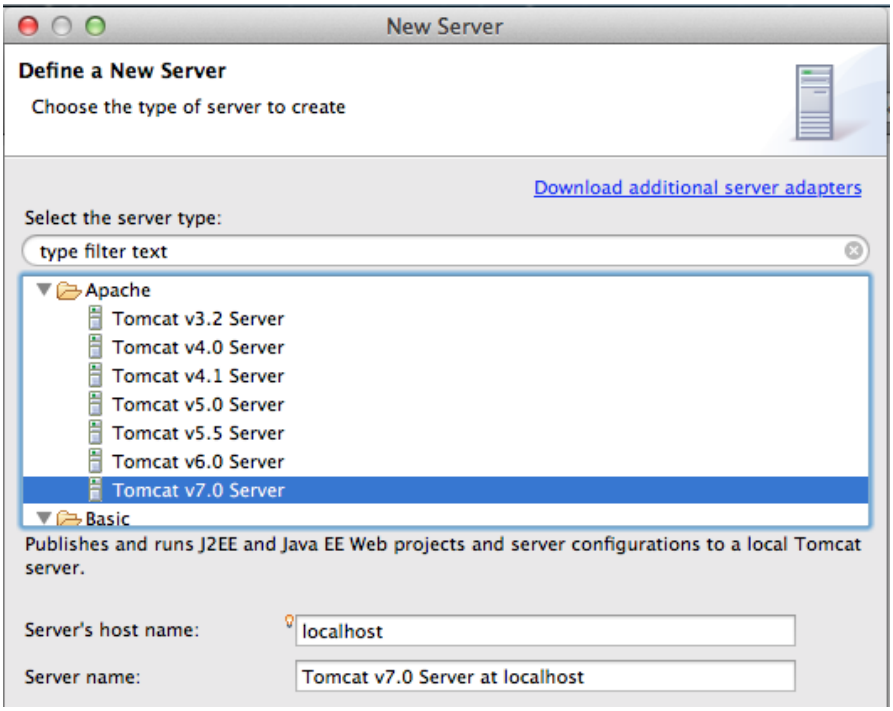
Podemos iniciar o Tomcat dentro do Eclipse para ganharmos mais produtividade, mas para isso, precisamos do plugin WTP (*Web Tools Platform*). O **Eclipse IDE for Java EE Developers** já vem com esse plugin, que além de possibilitar a integração de *containers* ao Eclipse, vem com diversos editores que auxiliam o desenvolvimento web e novas perspectivas.

Vamos integrar o Apache Tomcat ao Eclipse, para podermos iniciar e parar o Tomcat, além de implantar as aplicações a partir do ambiente de desenvolvimento.

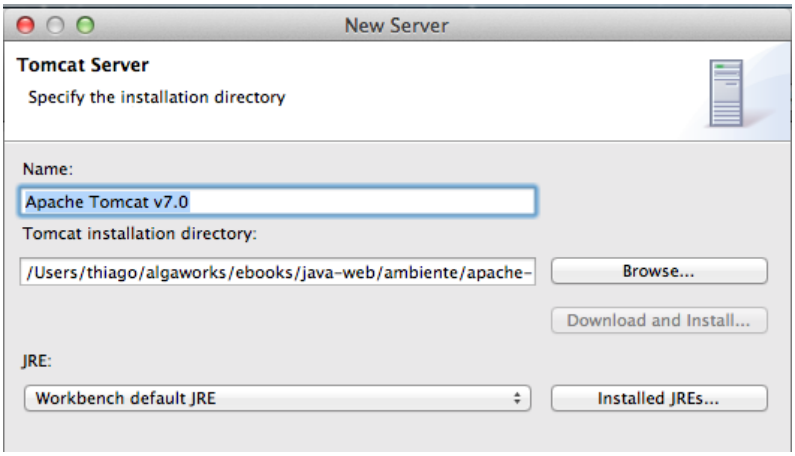
Acesse a *view* **Servers** e clique no único link que aparece para adicionar um novo servidor.



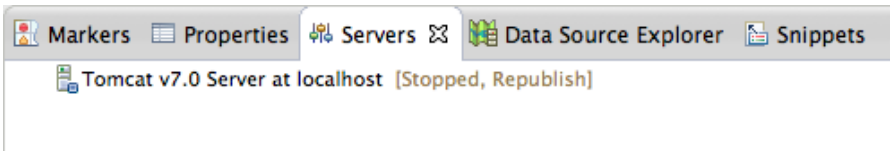
Na tela que abrir, encontre e selecione **Tomcat 7.0 Server**. Depois, clique em **Next**.



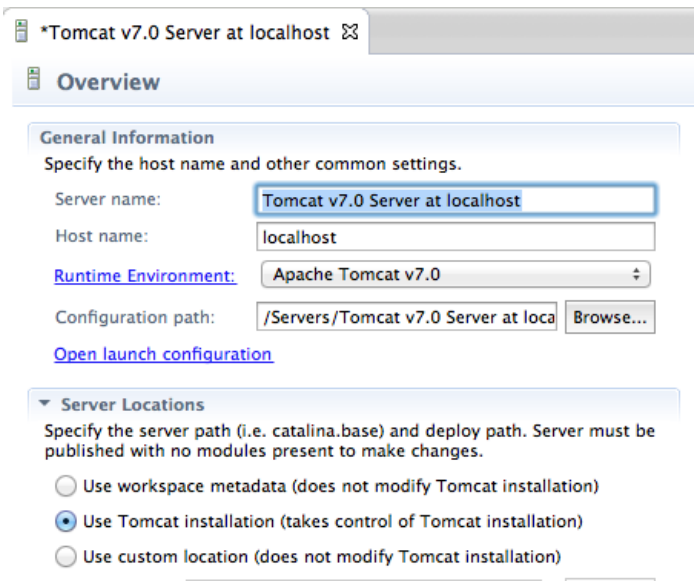
Clique no botão **Browse...**, selecione o diretório raiz onde o Tomcat foi descompactado e clique em **Finish**.



Você verá o Tomcat adicionado na *view* **Servers**.

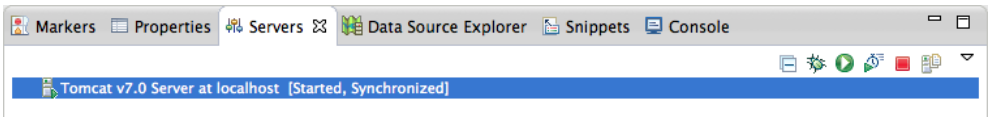


Dê um duplo clique no servidor do Tomcat adicionado na *view*. Marque a opção **Use Tomcat installation** em **Server Locations** e salve a alteração. Fazemos isso para que o WTP use as mesmas configurações da instalação do Tomcat.



Para iniciar o Tomcat dentro do Eclipse, primeiramente, confirme que o servidor não está rodando fora do Eclipse. Depois, selecione a linha que representa o servidor adicionado e clique no ícone  .

Se tudo der certo, você verá na *view* **Servers** que o Tomcat está iniciado (*Started*).



Abra um *browser* e acesse o endereço **http://localhost:8080**.

1.7. Apache Maven

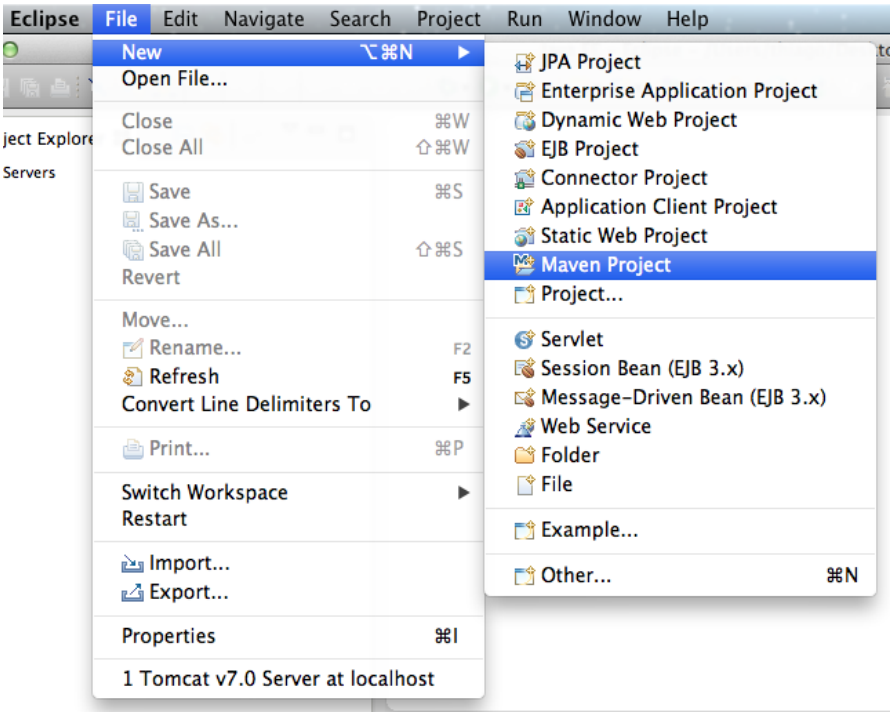
Maven é uma ferramenta da Apache Software Foundation para gerenciamento de dependências e automação de *build*, principalmente em projetos Java.

Um projeto que usa Maven possui um arquivo XML (*pom.xml*) que descreve o projeto, suas dependências, detalhes do *build*, diretórios, plugins requeridos, etc. Este arquivo é conhecido como POM (*Project Object Model*).

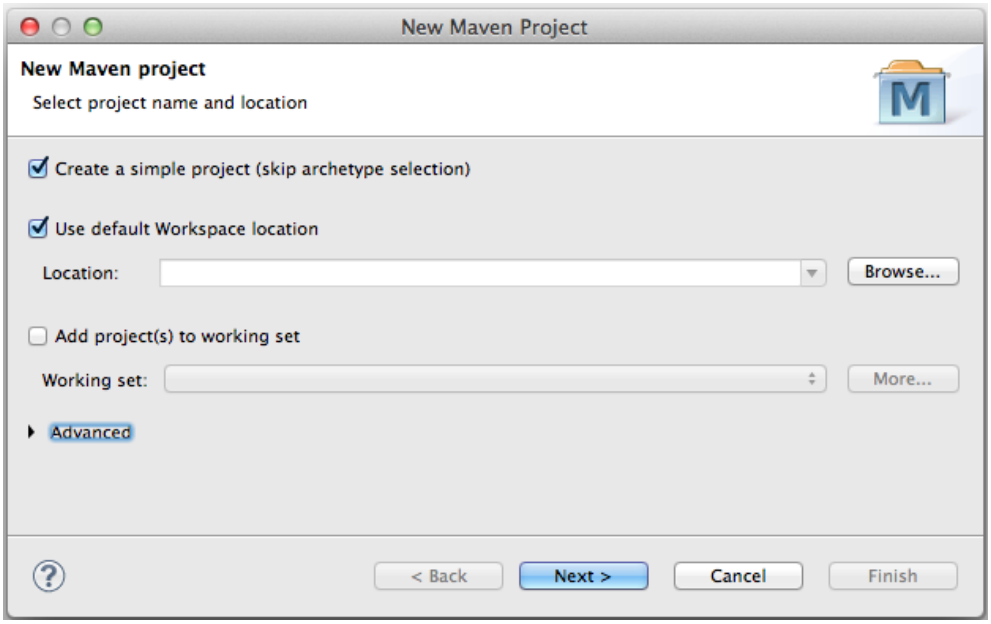
Usaremos Maven neste livro para criar os exemplos. As versões mais recentes do Eclipse já possui um plugin para criar projetos com Maven.

1.8. Primeiro projeto web com Apache Maven

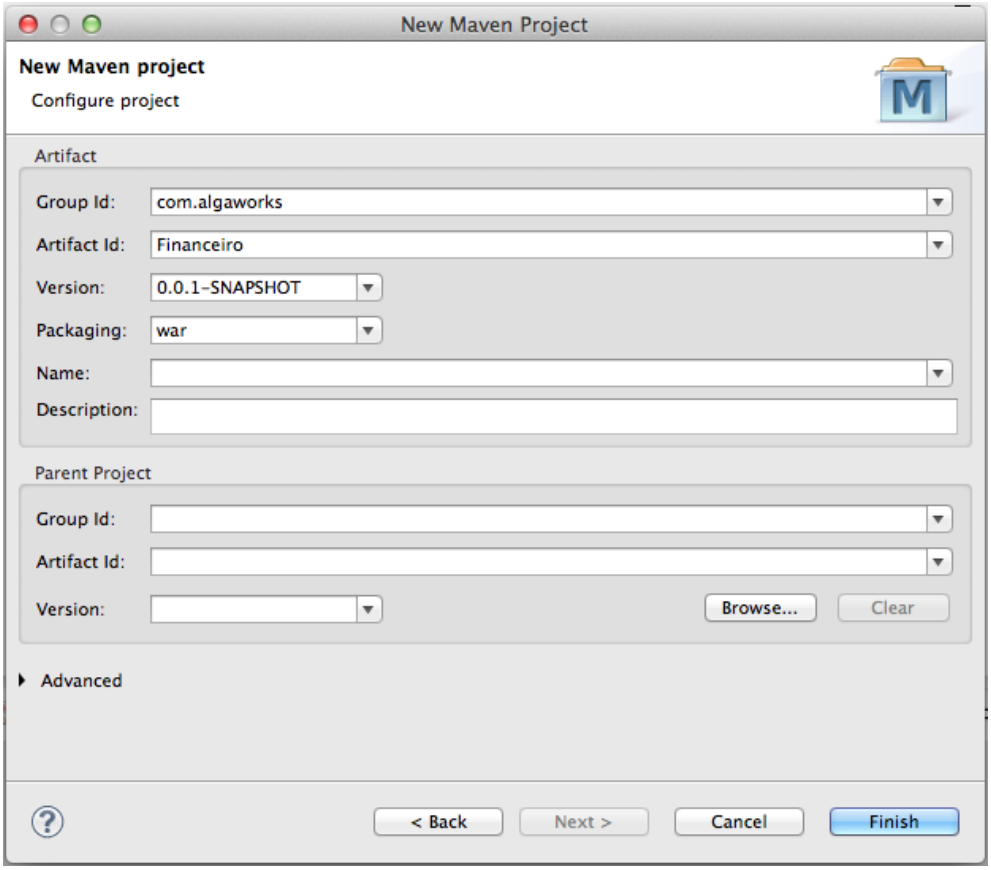
Para criar um novo projeto com Maven, acesse o menu **File**, **New**, **Maven Project**.



Marque a opção **Create a simple project** e clique em **Next >**.



Preencha o campo **Group Id** com um nome único que identifica sua organização (domínio ao contrário) e **Artifact Id** um identificador único do projeto dentro da organização. Selecione **war** na seleção **Packaging**. Depois, clique em **Finish**.



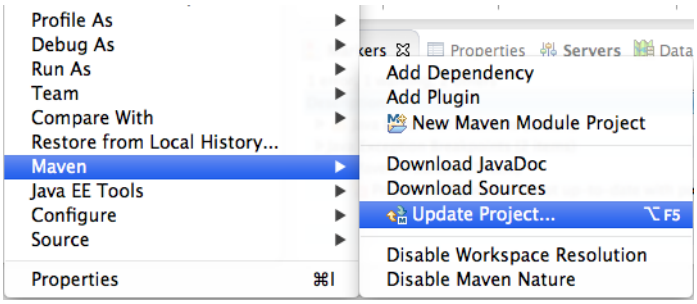
Um projeto web será criado dentro do workspace do Eclipse, com um arquivo *pom.xml* básico:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.algaworks</groupId>
  <artifactId>Financeiro</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
</project>
```

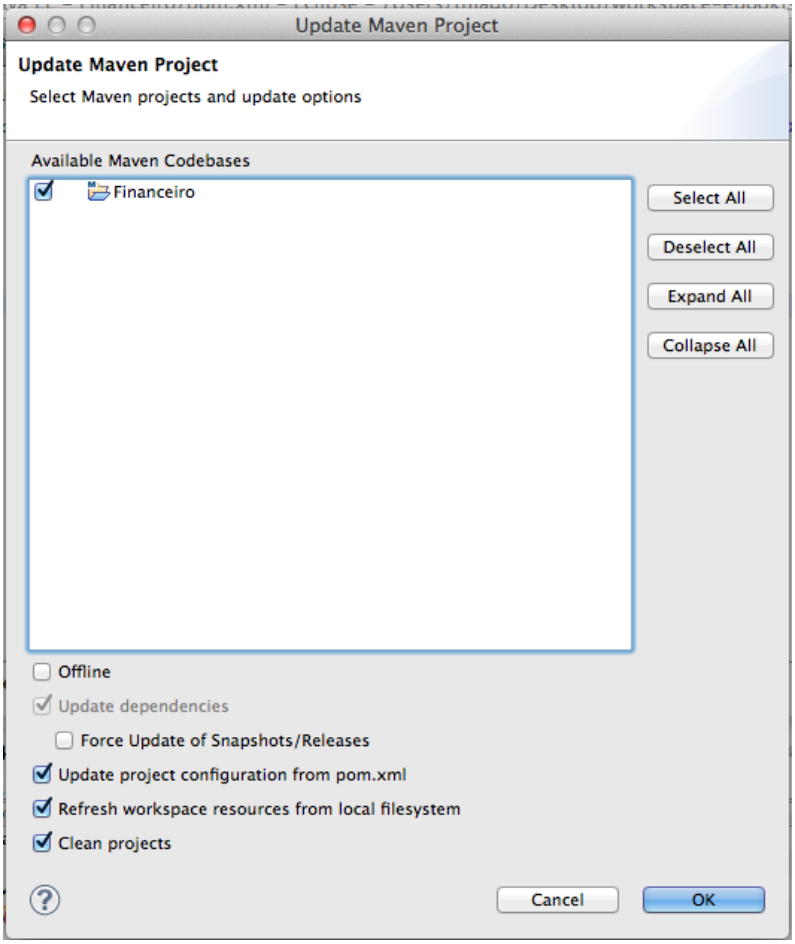
Vamos configurar uma propriedade do projeto para que o processo e *build* use a codificação UTF-8 para copiar arquivos e, também, configurar o plugin de compilação para dizer que nosso projeto deve usar Java 7.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.algaworks</groupId>
  <artifactId>Financeiro</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.0</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

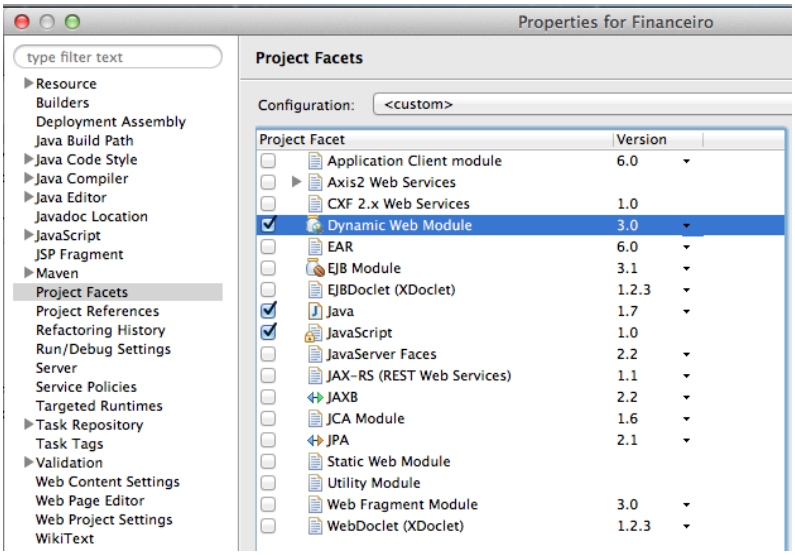
Precisamos atualizar o projeto no Eclipse baseado nessas alterações. Para isso, clicamos com o botão direito no projeto e acessamos o menu **Maven**, **Update Project...**.



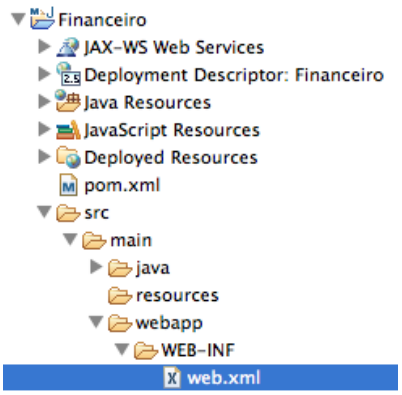
Apenas verifique se o projeto está selecionado e clique em **OK**.



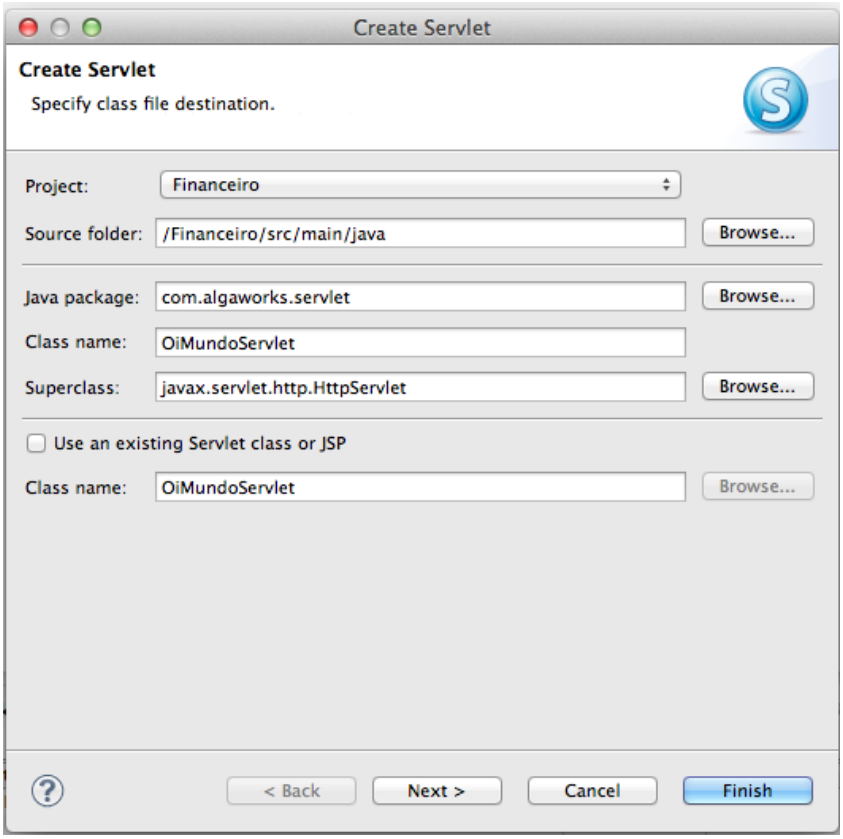
Precisamos verificar os *Project Facets* instalados no projeto pelo Eclipse. Acesse as propriedades do projeto e encontre o menu lateral **Project Facets**. Selecione a versão `3.0` no *facet* **Dynamic Web Module**, depois, clique em **OK**.



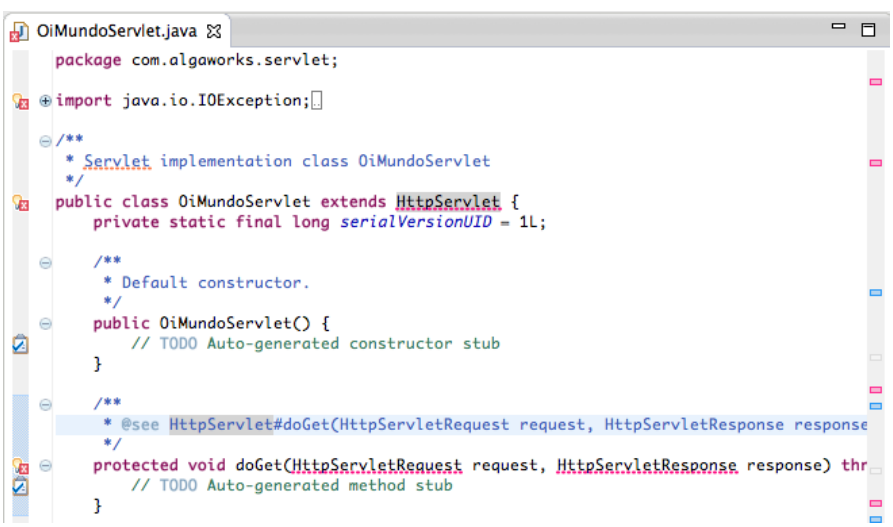
Excluiremos o arquivo *web.xml* do diretório *src/main/webapp/WEB-INF*, pois não precisaremos dele agora.



Vamos criar uma servlet muito simples, que apenas exibe "Oi Mundo" para o usuário. Clique com o botão direito no projeto criado, acesse a opção **New** e clique em **Servlet**. Na tela que abrirá, informe o nome do pacote e da classe da servlet. Depois, clique em **Finish**.



A classe `OiMundoServlet` será criada no pacote `com.algaworks.servlet` com vários erros.



A API de Servlets não foi encontrada, pois não foi declarada como uma dependência do projeto. Precisamos adicionar essa dependência no *pom.xml*, incluindo o trecho de código abaixo:

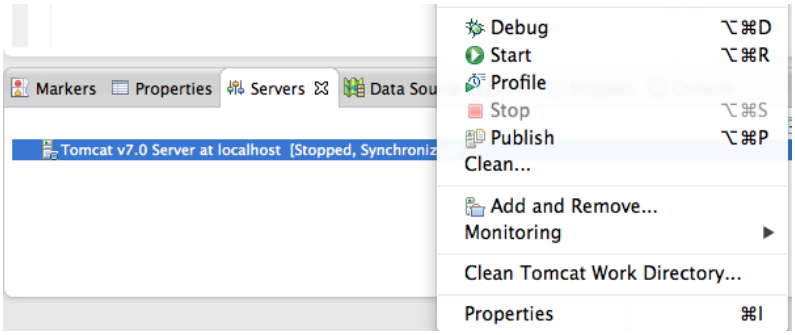
```
<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

O Maven irá baixar a dependência e instalar no repositório local da máquina.

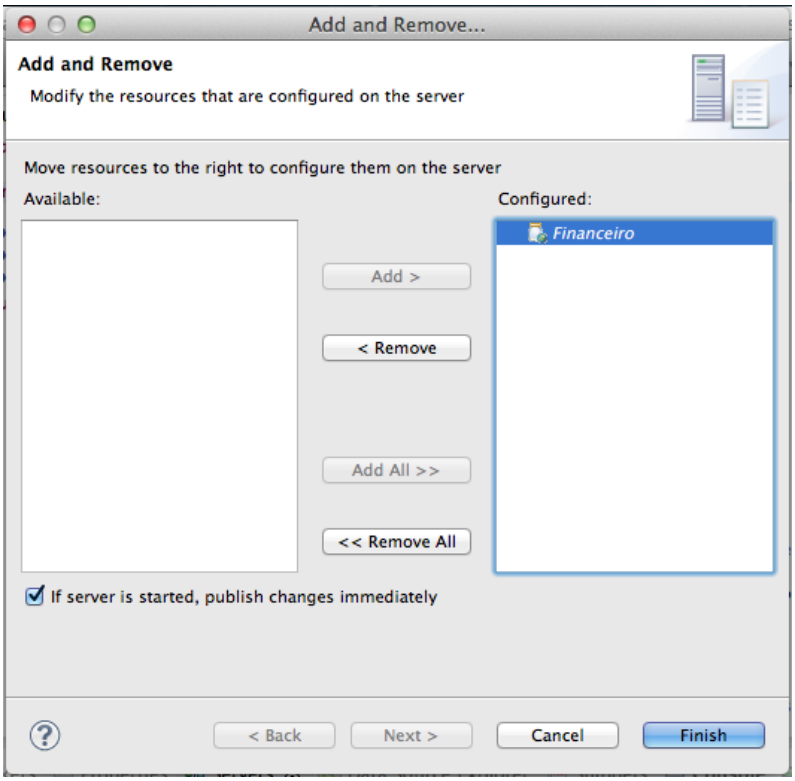
Programamos a servlet `OiMundoServlet`, deixando o código como abaixo:

```
package com.algaworks.servlet;

// imports...

public class OiMundoServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
            IOException {
        PrintWriter out = response.getWriter();
        out.print("<html>");
        out.print("<body><h1>Oi Mundo</h1></body>");
        out.print("</html>");
    }

}
```

Agora, precisamos adicionar o projeto ao Tomcat, para que ele faça a implantação sempre que houver alguma modificação. Na *view* **Servers**, clique com o botão direito no servidor do Tomcat e acesse a opção **Add and Remove...**

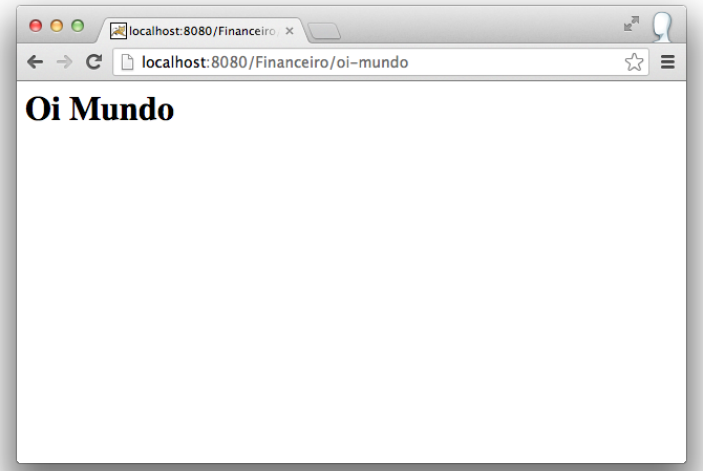


Marque o projeto na listagem da esquerda e transfira para a listagem da direita, clicando no botão **Add >**. Depois, clique em **Finish**.



Inicie o Tomcat, se ele não estiver sendo executado, depois, acesse **http://localhost:8080/Financeiro/oi-mundo**.

A servlet responderá a requisição e apresentará a mensagem "Oi Mundo" no navegador.


localhost:8080/Financeiro
localhost:8080/Financeiro/oi-mundo
Oi Mundo

Capítulo 2

Persistência de dados com JPA

2.1. O que é persistência?

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 outras alternativas para persistir dados, como em arquivos XML, arquivos texto e etc.

2.2. 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 faturamento possui a classe `NotaFiscal`, que no mundo real existe e todo mundo já viu alguma pelo menos uma vez, além de possuir uma classe que pode se chamar `Imposto`, que infelizmente todo mundo sente no bolso. 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 *nota_fiscal* e também *imposto*, 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.

2.3. Porque usar ORM?

Uma implementação ORM é mais complexa que outro framework qualquer para desenvolvimento web, porém os benefícios de desenvolver utilizando esta tecnologia são grandes.

Códigos de acesso a banco de dados com queries SQL são chatos de se desenvolver. JPA elimina muito do trabalho e deixa você se concentrar na lógica de negócio. JPA trará uma produtividade imensa para você.

A manutenabilidade de sistemas desenvolvidos com ORM é excelente, pois o mecanismo faz com que menos linhas de código sejam necessárias. Além de facilitar o entendimento, menos linhas de código deixam o sistema mais fácil de ser alterado.

Existem outras razões que fazem com que um sistema desenvolvido utilizando JPA seja melhor de ser mantido. Em sistemas com a camada de persistência desenvolvida usando JDBC e SQL, existe um trabalho na implementação para representar tabelas como objetos de domínio, e alterações no banco de dados ou no modelo de domínio geram um esforço de readequação que pode custar caro.

ORM abstrai sua aplicação do banco de dados e do dialeto SQL. Com JPA, você pode desenvolver um sistema usando um banco de dados e colocá-lo em produção usando diversos outros banco de dados, sem precisar alterar códigos-fontes para adequar sintaxe de queries que só funcionam em SGBDs de determinados fornecedores.

2.4. Java Persistence API e Hibernate

A *Java Persistence API* (JPA) é um framework para persistência em Java, que oferece uma API de mapeamento objeto-relacional e soluções para integrar persistência com sistemas corporativos escaláveis.

Com JPA, os objetos são POJO (*Plain Old Java Objects*), ou seja, não é necessário nada de especial para tornar os objetos persistentes. Basta adicionar algumas anotações nas classes que representam as entidades do sistema e começar a persistir ou consultar objetos.

JPA é uma especificação, e não um produto. Para trabalhar com JPA, precisamos de uma implementação.

O projeto do Hibernate ORM possui alguns módulos, sendo que o **Hibernate EntityManager** é a implementação da JPA que encapsula o Hibernate Core.

O **Hibernate Core** é a base para o funcionamento da persistência, com APIs nativas e metadados de mapeamentos em arquivos XML. Possui uma linguagem de consultas chamada HQL (parecido com SQL), um conjunto de interfaces para consultas usando critérios (Criteria API), etc.

Neste livro, estudaremos apenas o básico de JPA e Hibernate, para implementarmos exemplos mais interessantes com JSF, com acesso ao banco de dados.

2.5. Configuração de JPA e Hibernate com Maven

Como estamos usando Maven, não precisamos acessar o site do Hibernate para baixar os arquivos necessários e incluir manualmente no projeto. Podemos incluir todas as dependências no arquivo *pom.xml*, que o Maven baixará os arquivos necessários automaticamente.

```
<dependencies>
    <!-- Núcleo do Hibernate -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>4.2.6.Final</version>
        <scope>compile</scope>
    </dependency>

    <!-- Implementação de EntityManager da JPA -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>4.2.6.Final</version>
        <scope>compile</scope>
    </dependency>

    <!-- Driver JDBC do MySQL -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.26</version>
        <scope>compile</scope>
    </dependency>

    ...
</dependencies>
```

Adicionamos o núcleo do Hibernate, a implementação de JPA e o driver JDBC do MySQL em nosso projeto.

2.6. Criação do Domain Model

Em nosso projeto de exemplo, implementaremos um sistema financeiro simples. Antes de qualquer coisa, precisamos criar nosso modelo de domínio para o negócio em questão.

Nosso sistema possuirá as classes `Lancamento` e `Pessoa`, que representarão as entidades de mesmo nome, além de uma enumeração `TipoLancamento`..

```
package com.algaworks.financeiro.model;

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;
    }

}
```

```
package com.algaworks.financeiro.model;

public enum TipoLancamento {

    RECEITA, DESPESA

}
```

```
package com.algaworks.financeiro.model;

// imports...
```

```
public class Lancamento {

    private Long id;
    private Pessoa pessoa;
    private String descricao;
    private BigDecimal valor;
    private TipoLancamento tipo;
    private Date dataVencimento;
    private Date dataPagamento;

    public Long getId() {
        return id;
    }

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

    public Pessoa getPessoa() {
        return pessoa;
    }

    public void setPessoa(Pessoa pessoa) {
        this.pessoa = pessoa;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public BigDecimal getValor() {
        return valor;
    }

    public void setValor(BigDecimal valor) {
        this.valor = valor;
    }

    public TipoLancamento getTipo() {
        return tipo;
    }

    public void setTipo(TipoLancamento tipo) {
        this.tipo = tipo;
    }

    public Date getDataVencimento() {
        return dataVencimento;
    }
```

```
    public void setDataVencimento(Date dataVencimento) {
        this.dataVencimento = dataVencimento;
    }

    public Date getDataPagamento() {
        return dataPagamento;
    }

    public void setDataPagamento(Date dataPagamento) {
        this.dataPagamento = dataPagamento;
    }

}
```

Os atributos identificadores (chamados de `id`) são referentes às chaves primárias no banco de dados. Por exemplo, se existirem duas instâncias de `Pessoa` com o mesmo identificador, eles representam a mesma linha no banco de dados.

As classes de entidades devem seguir o estilo de JavaBeans, com métodos *getters* e *setters*. É obrigatório que essas classes possuam um construtor sem argumentos.

2.7. Implementação do equals() e hashCode()

Para que os objetos de entidades sejam diferenciados uns de outros, precisamos implementar os métodos `equals()` e `hashCode()`.

No banco de dados, as chaves primárias diferenciam registros distintos. Quando mapeamos uma entidade de uma tabela, devemos criar os métodos `equals()` e `hashCode()`, levando em consideração a forma em que os registros são diferenciados no banco de dados.

O Eclipse possui um gerador desses métodos que usa uma propriedade (ou várias, informadas por você) para criar o código-fonte. Veja como deve ficar a implementação dos métodos para a entidade `Pessoa`.

```
@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 gerar esses métodos também para a entidade `Lancamento`, e o Hibernate conseguirá comparar objetos para descobrir se são os mesmos.

2.8. Mapeamento básico

Para que o mapeamento objeto/relacional funcione, precisamos informar à implementação do JPA mais informações sobre como a classes `Lancamento` e `Pessoa` devem se tornar persistentes, ou seja, como instâncias dessas classes podem ser gravadas e consultadas no banco de dados. Para isso, devemos anotar os *getters* ou os atributos, além das próprias classes.

```
@Entity
@Table(name = "pessoa")
public class Pessoa {

    private Long id;
    private String nome;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

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

    @Column(length = 60, nullable = false)
    public String getNome() {
        return nome;
    }

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

```
    }

    // hashCode e equals

}
```

As anotações foram importadas do pacote `javax.persistence`. Dentro desse pacote estão todas as anotações padronizadas pela JPA.

A anotação `@Entity` diz que a classe é uma entidade, que representa uma tabela do banco de dados, e `@Table` define detalhes da tabela no banco de dados, como por exemplo o nome da tabela.

```
@Entity
@Table(name = "pessoa")
public class Pessoa {
```

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).

```
@Id
@GeneratedValue
public Long getId() {
    return id;
}
```

Definimos que a propriedade `nome` tem tamanho que comporta até 60 caracteres e não aceita valores nulos, ou seja, queremos criar uma restrição *not null* no banco de dados. Como não informamos o nome da coluna no banco de dados, ela receberá o mesmo nome da propriedade.

```
@Column(length = 60, nullable = false)
public String getNome() {
    return nome;
}
```

Vamos mapear a classe `Lancamento`, que é um pouco mais trabalhosa e usa novas anotações JPA.

```
@Entity
@Table(name = "lancamento")
public class Lancamento {

    private Long id;
    private Pessoa pessoa;
    private String descricao;
```

```
    private BigDecimal valor;
    private TipoLancamento tipo;
    private Date dataVencimento;
    private Date dataPagamento;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

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

    @ManyToOne(optional = false)
    @JoinColumn(name = "pessoa_id")
    public Pessoa getPessoa() {
        return pessoa;
    }

    public void setPessoa(Pessoa pessoa) {
        this.pessoa = pessoa;
    }

    @Column(length = 80, nullable = false)
    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    @Column(precision = 10, scale = 2, nullable = false)
    public BigDecimal getValor() {
        return valor;
    }

    public void setValor(BigDecimal valor) {
        this.valor = valor;
    }

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    public TipoLancamento getTipo() {
        return tipo;
    }

    public void setTipo(TipoLancamento tipo) {
        this.tipo = tipo;
    }

    @Temporal(TemporalType.DATE)
```

```
    @Column(name = "data_vencimento", nullable = false)
    public Date getDataVencimento() {
        return dataVencimento;
    }

    public void setDataVencimento(Date dataVencimento) {
        this.dataVencimento = dataVencimento;
    }

    @Temporal(TemporalType.DATE)
    @Column(name = "data_pagamento", nullable = true)
    public Date getDataPagamento() {
        return dataPagamento;
    }

    public void setDataPagamento(Date dataPagamento) {
        this.dataPagamento = dataPagamento;
    }

    // hashCode e equals

}
```

A primeira novidade é o mapeamento muitos-para-um em `getPessoas`. A anotação `@ManyToOne` indica a multiplicidade do relacionamento entre lançamentos 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 provedor JPA que existe uma chave estrangeira na coluna `pessoa_id` da tabela `lancamento`, que referencia a tabela `pessoa`.

```
@ManyToOne(optional = false)
@JoinColumn(name = "pessoa_id")
public Pessoa getPessoa() {
    return pessoa;
}
```

Atribuímos a precisão de `10` com escala de `2` casas na coluna de número decimal.

```
@Column(precision = 10, scale = 2, nullable = false)
public BigDecimal getValor() {
    return valor;
}
```

O tipo do lançamento é uma enumeração, por isso, mapeamos com a anotação `@Enumerated`, indicando que queremos armazenar a string da constante na coluna da tabela, e não o índice da constante.

```
@Enumerated(EnumType.STRING)
@Column(nullable = false)
public TipoLancamento getTipo() {
```

```
    return tipo;
}
```

As propriedades de datas foram mapeadas usando a anotação `@Temporal`, indicando que queremos armazenar apenas a data (sem informações da hora).

```
@Temporal(TemporalType.DATE)
@Column(name = "data_vencimento", nullable = false)
public Date getDataVencimento() {
    return dataVencimento;
}
```

2.9. O arquivo persistence.xml

O *persistence.xml* é um arquivo de configuração padrão da JPA. Ele deve ser criado no diretório *META-INF* da aplicação ou do módulo que contém os beans de entidade. No Eclipse, opcionalmente, você pode adicionar o *Project Facet* **JPA** no seu projeto, que a estrutura básica desse arquivo é criada automaticamente, além de ter outras facilidades.

O arquivo *persistence.xml* define unidades de persistência, conhecidas como *persistence units*.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="FinanceiroPU">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>

        <properties>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:mysql://localhost/financeiro" />
            <property name="javax.persistence.jdbc.user"
                value="usuario" />
            <property name="javax.persistence.jdbc.password"
                value="senha" />
            <property name="javax.persistence.jdbc.driver"
                value="com.mysql.jdbc.Driver" />

            <property name="hibernate.dialect"
                value="org.hibernate.dialect.MySQL5Dialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
```

```
        </properties>
    </persistence-unit>

</persistence>
```

O nome da unidade de persistência foi definido como `FinanceiroPU`. Precisaremos desse nome daqui a pouco, quando formos colocar tudo para funcionar.

O `provider` diz qual é a implementação que será usada como provedor de persistência.

Existem várias opções de configuração que podem ser informadas neste arquivo XML. Vejamos as principais propriedades que usamos em nosso arquivo de configuração:

- `javax.persistence.jdbc.url`: descrição da URL de conexão com o banco de dados.
- `javax.persistence.jdbc.driver`: nome completo da classe do driver JDBC.
- `javax.persistence.jdbc.user`: nome do usuário do banco de dados.
- `javax.persistence.jdbc.password`: senha do usuário do banco de dados.
- `hibernate.dialect`: dialeto a ser usado na construção de comandos SQL.
- `hibernate.show_sql`: informa se os comandos SQL devem ser exibidos na console (importante para *debug*, mas deve ser desabilitado em ambiente de produção).
- `hibernate.format_sql`: indica se os comandos SQL exibidos na console devem ser formatados (facilita a compreensão, mas pode gerar textos longos na saída).
- `hibernate.hbm2ddl.auto`: cria ou atualiza automaticamente a estrutura das tabelas no banco de dados.

2.10. Gerando as tabelas do banco de dados

Como ainda não temos as tabelas representadas pelas classes `Pessoa` e `Lancamento` no banco de dados, precisamos criás-la.

O Hibernate pode fazer isso pra gente, graças à propriedade `hibernate.hbm2ddl.auto` com valor `update`, que incluímos no arquivo *persistence.xml*.

Precisamos apenas criar um `EntityManagerFactory`, que todas as tabelas mapeadas pelas entidades serão criadas ou atualizadas.

```java
import javax.persistence.Persistence;

public class CriaTabelas {

    public static void main(String[] args) {
        Persistence.createEntityManagerFactory("FinanceiroPU");
    }

}
```

O parâmetro do método `createEntityManagerFactory` deve ser o mesmo nome que informamos no atributo `name` da *tag* `persistence-unit`, no arquivo *persistence.xml*.

Ao executar o código, as tabelas são criadas.

```
...
Set 30, 2013 1:50:52 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000228: Running hbm2ddl schema update
Set 30, 2013 1:50:52 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000102: Fetching database metadata
Set 30, 2013 1:50:52 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000396: Updating schema
Set 30, 2013 1:50:52 PM org.hibernate.tool.hbm2ddl.DatabaseMetadata...
INFO: HHH000262: Table not found: lancamento
Set 30, 2013 1:50:52 PM org.hibernate.tool.hbm2ddl.DatabaseMetadata...
INFO: HHH000262: Table not found: pessoa
Set 30, 2013 1:50:52 PM org.hibernate.tool.hbm2ddl.DatabaseMetadata...
INFO: HHH000262: Table not found: lancamento
Set 30, 2013 1:50:52 PM org.hibernate.tool.hbm2ddl.DatabaseMetadata...
INFO: HHH000262: Table not found: pessoa
Set 30, 2013 1:50:52 PM org.hibernate.tool.hbm2ddl.DatabaseMetadata...
INFO: HHH000262: Table not found: lancamento
Set 30, 2013 1:50:52 PM org.hibernate.tool.hbm2ddl.DatabaseMetadata...
INFO: HHH000262: Table not found: pessoa
Set 30, 2013 1:50:53 PM org.hibernate.tool.hbm2ddl.SchemaUpdate execute
INFO: HHH000232: Schema update complete
```

2.11. Próximos passos

Já temos nossas entidades `Pessoa` e `Lancamento` mapeadas e as tabelas criadas no banco de dados. Precisaremos persistir objetos, consultar, excluir e atualizar, mas deixaremos para apresentar como funciona esses detalhes apenas quando for necessário, pois este não é um livro de JPA, ok?

Capítulo 3

Introdução ao JSF

3.1. O que é JavaServer Faces?

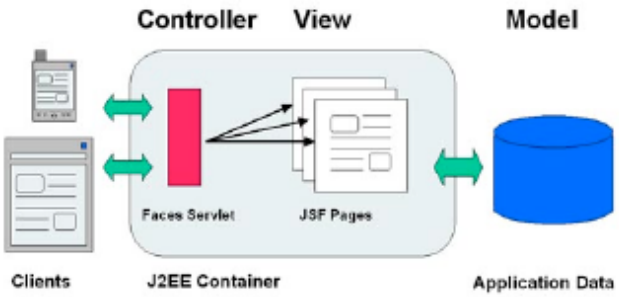
JavaServer Faces, também conhecido como JSF, é uma tecnologia para desenvolvimento web que utiliza um modelo de interfaces gráficas baseado em eventos. Esta tecnologia foi definida pelo JCP (*Java Community Process*), o que a torna um padrão de desenvolvimento e facilita o trabalho dos fornecedores de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

JSF é baseado no padrão de projeto MVC (Model View Controller), o que torna o desenvolvimento de sistemas menos complicado. 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 uma servlet chamada *Faces Servlet*, opcionalmente, por arquivos XML de configuração e por vários manipuladores de ações e observadores de eventos. A *Faces Servlet* recebe as requisições dos usuários na web, redireciona para o modelo e envia uma resposta.

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.



3.2. Principais componentes

O verdadeiro poder de JavaServer Faces está em seu modelo de componentes de interface do usuário, que gera 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.

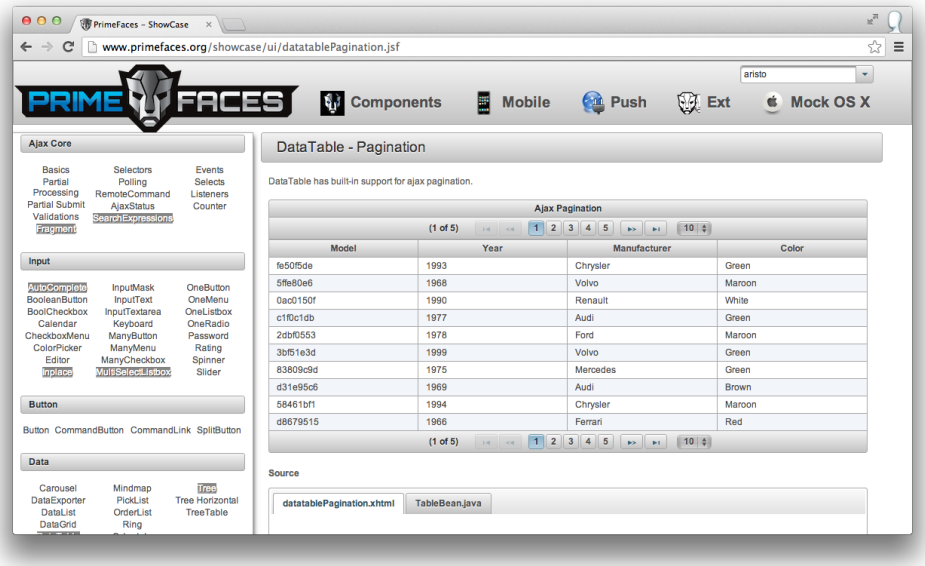
Existem vários componentes JSF, desde os mais simples, como um *Output Label*, que apresenta simplesmente um texto, ou um *Data Table*, que representa dados tabulares de uma coleção que pode vir do banco de dados.

A API de 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: formulário, campos de entrada de texto e senhas, rótulos, links, botões, mensagens, painéis, tabela de dados, etc.

3.3. Bibliotecas de componentes de terceiros

Atualmente, existem diversas organizações que trabalham na criação de componentes personalizados, como exemplo, podemos citar a Oracle (ADF Faces Rich Client), IceSoft (IceFaces), Red Hat (RichFaces), Prime Technology (PrimeFaces) e etc.

As bibliotecas de componentes terceiras incluem muitos componentes interessantes, como tabelas de dados avançadas, menus suspensos, botões, barras de progressão, diálogos, componentes para captura de datas e cores, etc.



Usaremos PrimeFaces no projeto deste livro, mas antes, usaremos apenas os componentes básicos do JSF.

3.4. 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 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. 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, a implementação mais conhecida atualmente é a **Mojarra**, que pode ser obtida em https://javaserverfaces.java.net/.

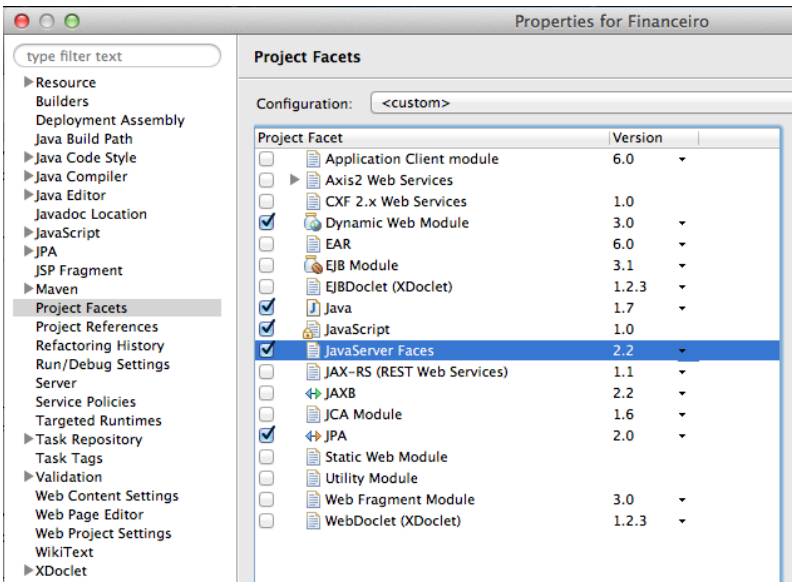
3.5. Adicionando JSF ao projeto Maven

Como estamos usando o Maven, não precisaremos baixar a implememtação de JSF manualmente. Podemos apenas adicionar a dependência no POM do projeto.

```
<!-- Mojarra (implementacao do JSF) -->
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.faces</artifactId>
    <version>2.2.2</version>
    <scope>compile</scope>
</dependency>
```

Usaremos o JSF 2.2, lançado dentro da Java EE 7.

Adicionaremos o *Project Facet* **JavaServer Faces** com a versão 2.2 para o Eclipse nos auxiliar melhor durante o desenvolvimento.



3.6. Managed bean

Antigamente, alguns programadores desenvolviam todo o comportamento de uma página no próprio arquivo de layout, na verdade, infelizmente, ainda existem programadores que fazem isso.

Em JSF, não conseguimos fazer isso. O arquivo que inclui os componentes da página deve ficar separado da classe que gerencia o comportamento dela, chamada de managed bean.

Os managed beans nada mais são que Java Beans, que servem como canais entre a interface gráfica (a página) e o *back-end* da aplicação (regras de negócio, acesso ao banco de dados, etc). Os beans gerenciados do JSF podem receber dados digitados pelos usuários através de alguma página, processar métodos através de ações dos usuários e fornecer dados para apresentação na página.

Para um bean ser reconhecido como um managed bean JSF, precisamos registrá-lo. A maneira mais fácil de fazer isso é através da anotação `@ManagedBean`, do pacote `javax.faces.bean`. Por padrão, todas as classes do projeto serão escaneadas para encontrar beans anotados.

Nosso primeiro exemplo será o managed bean `OlaBean`. Os atributos `nome` e `sobrenome` serão informados pelo usuários, por isso, possuem os *getters* e *setters* correspondentes. O atributo `nomeCompleto` será montado pelo método `dizerOla` e apresentado na página, portanto, não precisamos do *setter* para esse atributo. O método `dizerOla` será chamado a partir de um botão da página.

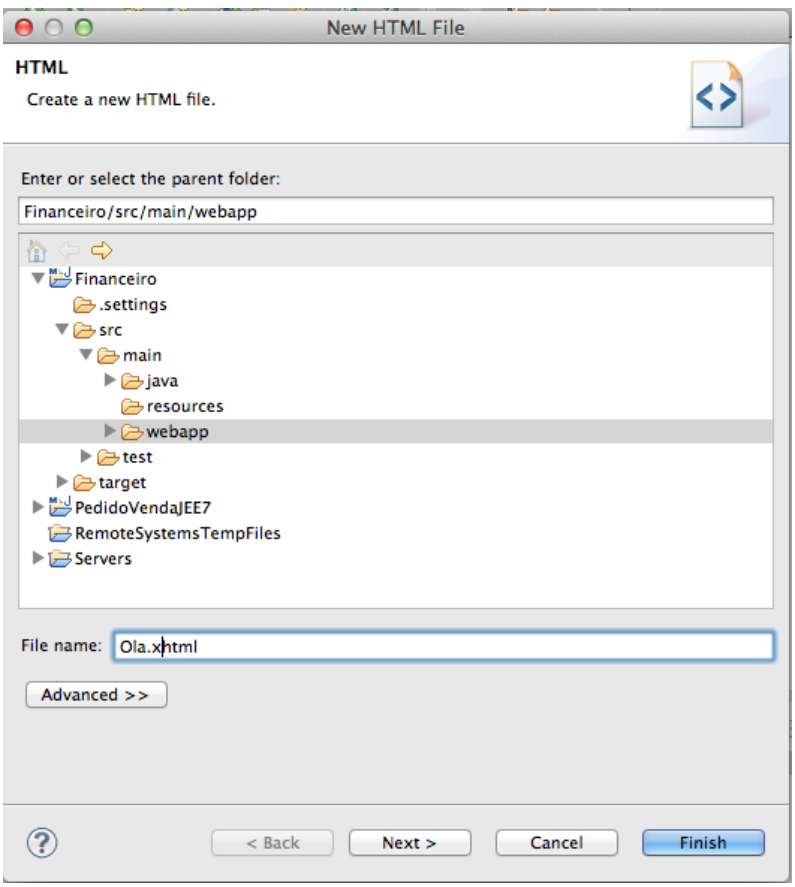
```
@ManagedBean
public class OlaBean {

    private String nome;
    private String sobrenome;
    private String nomeCompleto;

    public void dizerOla() {
        this.nomeCompleto = this.nome.toUpperCase()
            + " " + this.sobrenome;
    }

    public String getNome() {
        return nome;
    }

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

    public String getSobrenome() {
        return sobrenome;
    }

    public void setSobrenome(String sobrenome) {
        this.sobrenome = sobrenome;
    }
```

```
    public String getNomeCompleto() {
        return nomeCompleto;
    }

}
```

3.7. Criando uma página XHTML

Vamos criar uma página simples em JSF, que por enquanto, não fará ligação com o managed bean que programamos.

Criaremos um arquivo chamado *Ola.xhtml*, clicando com o botão direito no projeto e acessando **New**, **HTML File**. Na tela **New HTML File**, digite o nome do arquivo e clique em **Finish**. O arquivo será criado no diretório *src/main/webapp* do projeto.



Deixaremos o código-fonte do arquivo *Ola.xhtml* como a seguir:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Olá JSF</title>
    </h:head>

    <h:body>
        <h:form>
            <h1>Olá</h1>

            Nome: <h:inputText />
            <br/>
            Sobrenome: <h:inputText />
            <br/>

            <h:commandButton value="Dizer olá" />
        </h:form>
    </h:body>

</html>
```

A declaração *DOCTYPE* foi usada para dizer aos *browsers* dos usuários a versão do HTML que estamos usando, para que eles possam exibir os elementos de forma adequada. Em nosso exemplo, declaramos que é o HTML 5.

```
<!DOCTYPE html>
```

Importamos a biblioteca de componentes **HTML** através do namespace *http://xmlns.jcp.org/jsf/html*. A letra `h` é o prefixo usado para acessar os componentes dessa biblioteca.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

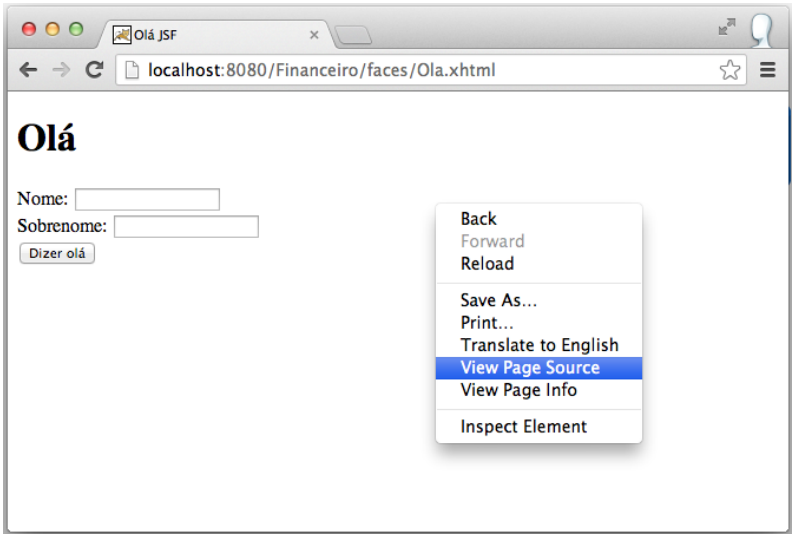
As tags `<h:head>` e `<h:body>` são importantes para o funcionamento da página JSF, para definir o cabeçalho e corpo da página.

```
<h:head>
    <title>Olá JSF</title>
</h:head>

<h:body>
...
</h:body>
```

No corpo da página, usamos um componente de formulário, representado pela tag `<h:form>`, dois componentes de entrada de texto, representados pela tag `<h:inputText>` e um botão, com `<h:commandButton>`.

```
<h:form>
    <h1>Olá</h1>

    Nome: <h:inputText />
    <br/>
    Sobrenome: <h:inputText />
    <br/>

    <h:commandButton value="Dizer olá" />
</h:form>
```

Considerando que esse exemplo esteja no projeto "Financeiro", podemos acessar a página que criamos pela URL *http://localhost:8080/Financeiro/faces/Ola.xhtml*.



Acesse o código-fonte da página pelo navegador. No caso do Google Chrome, clique com o botão direito na página e depois em **View Page Source**.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"><head id="j_idt2">
        <title>Olá JSF</title></head><body>
<form id="j_idt5" name="j_idt5" method="post"
    action="/Financeiro/faces/Ola.xhtml"
    enctype="application/x-www-form-urlencoded">
<input type="hidden" name="j_idt5" value="j_idt5" />

    <h1>Olá</h1>

    Nome: <input type="text" name="j_idt5:j_idt7" />
    <br />
    Sobrenome: <input type="text" name="j_idt5:j_idt9" />
    <br /><input type="submit" name="j_idt5:j_idt11" value="Dizer olá" />
    <input type="hidden" name="javax.faces.ViewState"
    id="j_id1:javax.faces.ViewState:0"
```

```
    value="7050636473587579887:292279611572368337" autocomplete="off" />
</form></body>

</html>
```

A implementação do JSF gerou o código-fonte HTML a partir dos componentes que adicionados à página XHTML.

3.8. Ligando valores e ações com EL

Depois que o managed bean é registrado, ele pode ser acessado pelos componentes das páginas do projeto. A maioria dos componentes JSF possui propriedades que nos permitem especificar um valor ou uma ligação de valor que está associado a um bean.

Por exemplo, podemos especificar um valor estático no componente *InputText*:

```
<h:inputText value="Olá mundo!" />
```

Expression Language (EL) torna possível o acesso rápido a managed beans. O avaliador de expressões é responsável por tratar expressões EL que estão entre os caracteres `#{ }`. No exemplo abaixo, ligamos o valor do componente *InputText* à propriedade `nome` do managed bean `OlaBean`, através do *getter* e *setter*.

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

Quando o componente for renderizado, o método `getNome` será invocado. Já o método `setNome` será chamado quando o usuário digitar algo no componente de entrada de texto e submeter o formulário.

O nome `olaBean` (com inicial em minúsculo) é definido por padrão, de acordo com o nome da classe do managed bean, quando não especificamos um outro nome.

```
@ManagedBean
public class OlaBean {
}
```

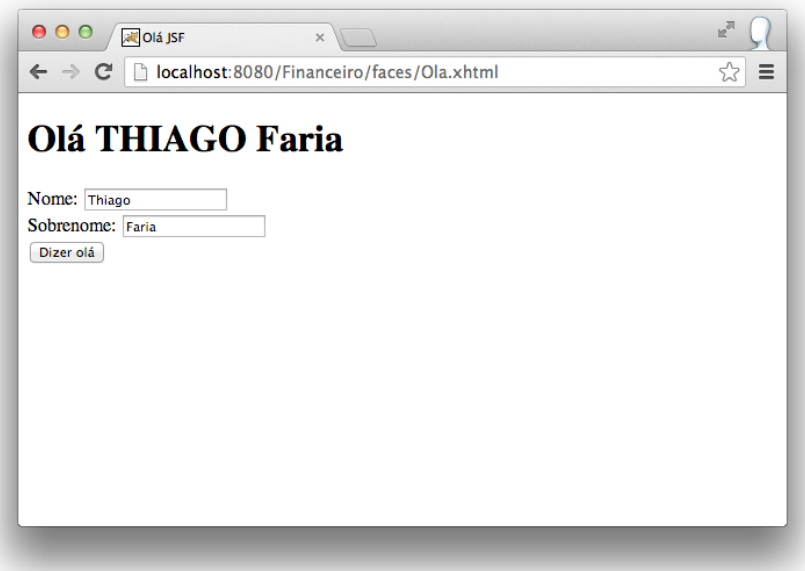
Poderíamos definir um nome diferente atribuindo `name` da anotação `@ManagedBean`.

```
@ManagedBean(name = "ola")
public class OlaBean {
}
```

Agora, associaremos os valores e/ou ações dos componentes às propriedades e métodos de `OlaBean`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Olá JSF</title>
    </h:head>

    <h:body>
        <h:form>
            <h1>Olá #{ola.nomeCompleto}</h1>

            Nome: <h:inputText value="#{ola.nome}" />
            <br/>
            Sobrenome: <h:inputText value="#{ola.sobrenome}" />
            <br/>

            <h:commandButton value="Dizer olá"
                action="#{ola.dizerOla}" />
        </h:form>
    </h:body>

</html>
```

Quando o botão "Dizer olá" for acionado, o framework chamará o método `dizerOla` do managed bean, que passará o nome para maiúsculo, concatenará com o sobrenome e atribuirá à variável `nomeCompleto`, que é acessada pela página através do *getter*, para dizer "Olá NOME sobrenome".



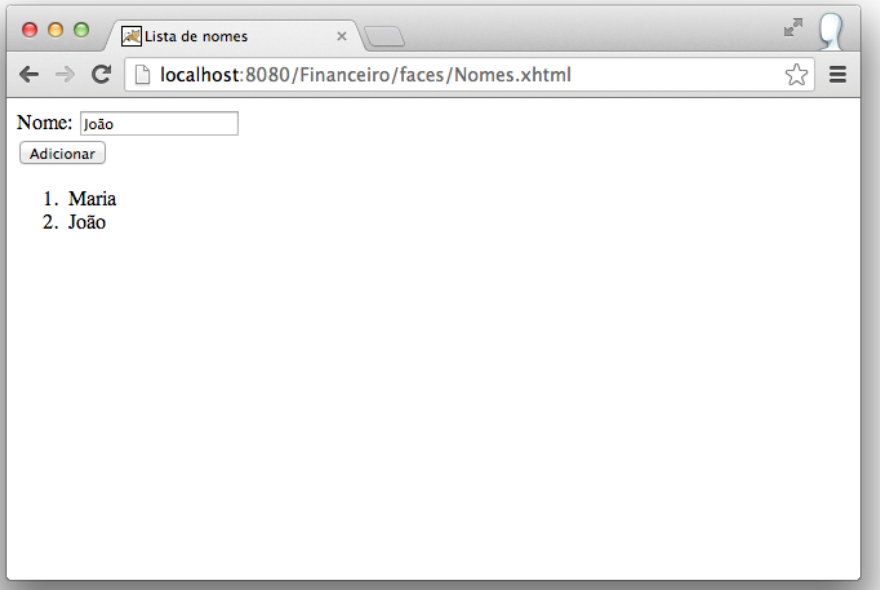
3.9. Escopos de managed beans

Quando referenciamos um managed bean via EL, o framework do JSF instanciará um objeto da classe do managed bean, ou recuperará uma instância existente. Todas as instâncias possuem um tempo de vida, que é definido dependendo do escopo usado no managed bean.

Os escopos de managed beans JSF podem ser definidos através de anotações do pacote `javax.faces.bean`. Os principais são:

- `@NoneScoped`: o bean será instanciado a cada vez que for referenciado.
- `@RequestScoped` (padrão): tem vida curta, começando quando é referenciado em uma única requisição HTTP e terminando quando a resposta é enviada de volta ao cliente.
- `@ViewScoped`: a instância permanece ativa até que o usuário navegue para uma próxima página.
- `@SessionScoped`: mantém a instância durante diversas requisições e até mesmo navegações entre páginas, até que a sessão do usuário seja invalidada ou o tempo limite é atingido. Cada usuário possui sua sessão de navegação, portanto, os objetos não são compartilhados entre os usuários.
- `@ApplicationScoped`: mantém a instância durante todo o tempo de execução da aplicação. É um escopo que compartilha os objetos para todos os usuários do sistema.

Para exemplificar o funcionamento de alguns escopos, criaremos uma página com uma lista de nomes, com um campo e um botão para adicionar novos nomes.



O managed bean abaixo é registrado no escopo de requisição, pois anotamos com `@RequestScoped`.

```
@ManagedBean
@RequestScoped
public class NomesBean {

    private String nome;
    private List<String> nomes = new ArrayList<>();

    public void adicionar() {
        this.nomes.add(nome);
    }

    public String getNome() {
        return nome;
    }

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

    public List<String> getNomes() {
        return nomes;
    }

}
```

Criamos uma página chamada *Nomes.xhtml*. A única novidade neste arquivo é a importação do namespace *http://xmlns.jcp.org/jsf/facelets* e o uso da tag `<ui:repeat>`. O componente *Repeat* funciona como um repetidor, onde cada elemento da lista passada para a propriedade `values` é atribuído a uma variável com o nome definido em `var`, renderizando o conteúdo da tag a cada iteração.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

    <h:head>
        <title>Lista de nomes</title>
    </h:head>

    <h:body>
        <h:form>
            Nome: <h:inputText value="#{nomesBean.nome}" />
            <br/>
            <h:commandButton value="Adicionar"
                action="#{nomesBean.adicionar}" />
            <br/>

            <ol>
                <ui:repeat var="nome" value="#{nomesBean.nomes}">
                    <li>#{nome}</li>
                </ui:repeat>
            </ol>
        </h:form>
    </h:body>

</html>
```

Quando executamos esse exemplo, conseguimos inserir o primeiro nome normalmente, mas ao tentar inserir o segundo nome, o nome anterior que estava na lista desaparece. Isso ocorre porque o escopo que usamos foi de requisição, ou seja, quando inserimos o segundo nome, a primeira requisição já não existe mais, e o managed bean é instanciado novamente pelo framework.

Para mudar o escopo de `NomesBean` para *view*, basta anotarmos a classe com `@ViewScoped`.

```
@ManagedBean
@ViewScoped
public class NomesBean {

    ...

}
```

Agora conseguimos inserir diversos nomes na lista em sequência, mas se recarregarmos a página ou navegarmos para uma outra página e voltar, a lista estará vazia, pois os dados permanecem enquanto o usuário estiver na mesma página.

A anotação `@SessionScoped` define o escopo de sessão.

```
@ManagedBean
@SessionScoped
public class NomesBean {

    ...

}
```

Se usarmos este escopo, podemos adicionar diversos nomes, navegar em outras páginas e voltar para a listagem de nomes, que eles ainda estarão lá. O managed bean será perdido apenas quando o tempo limite da sessão for alcançado ou se o usuário solicitar a invalidação da sessão (geralmente, através de *logout* do sistema).

O escopo de aplicação compartilha a instância do managed bean com todos os usuários.

```
@ManagedBean
@ApplicationScoped
public class NomesBean {

    ...

}
```

Se adicionarmos alguns nomes e acessarmos a página de outro navegador ou até mesmo de outro computador, os mesmos nomes aparecerão, mostrando que realmente este escopo compartilha os dados.

3.10. Backing bean

Quando você digita o endereço da aplicação no *browser* e acessa uma página do sistema, o framework JSF lê e processa o arquivo XHTML. Esse arquivo contém tags de componentes, como formulários, campos de entrada de textos, botões e etc.

JSF fornece um conjunto de classes que representam os componentes. Essas classes são instanciadas de acordo com as tags adicionadas na página XHTML e constroem uma hierarquia de componentes, que representam os elementos da página e seus relacionamentos.

Por exemplo, a classe `HtmlForm` representa o componente de formulário, `HtmlInputText` representa o componente de entrada de texto e `HtmlCommandButton` o botão.

Durante o processamento da página, um código HTML é gerado (renderizado) e enviado para o navegador do usuário. Cada componente JSF possui um renderizador que é responsável por gerar código HTML, refletindo o estado de seu componente.

Em algumas ocasições, seu bean pode precisar ter acesso às instâncias dos componentes da página. Este acesso dá possibilidade de inspecionar e até modificar propriedades do componente que está sendo renderizado para o usuário. Por exemplo, um componente de entrada de texto `<h:inputText/>`, representado como objeto Java do tipo `HtmlInputText`, pode ter a propriedade `disabled` modificada em tempo de execução pelo código Java, através do acesso direto a este objeto.

Para fazer 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 valores, pode fazer também ligações de componentes.

Para um bean ser caracterizado como um *backing bean*, no código-fonte da página é feita uma amarração (*binding*) em uma tag de um componente JSF para uma propriedade de um managed bean.

No backing bean `NomesBean`, criamos os atributos que receberão instâncias dos componentes:

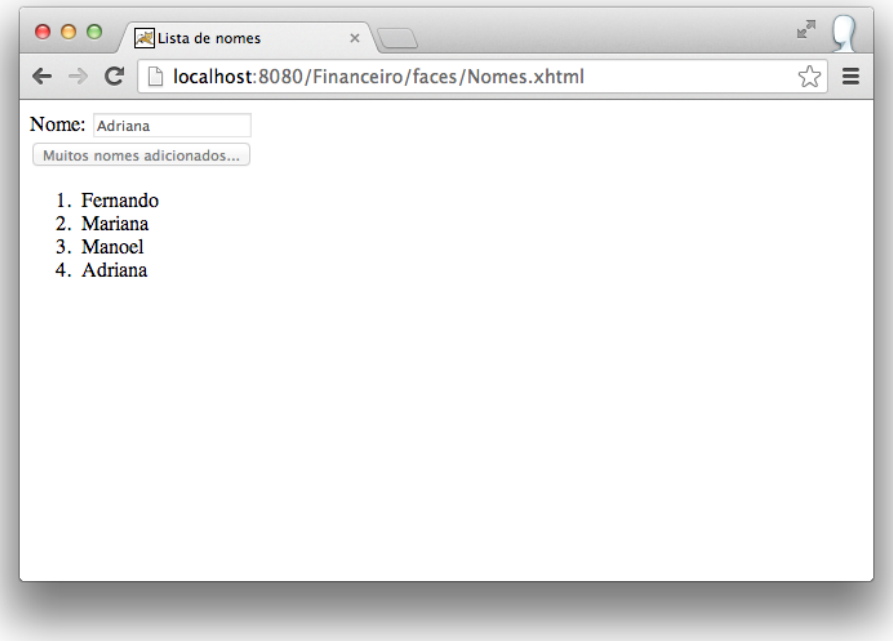
```
@ManagedBean
@ViewScoped
public class NomesBean {

    private String nome;
    private List<String> nomes = new ArrayList<>();

    private HtmlInputText inputNome;
    private HtmlCommandButton botaoAdicionar;

    public void adicionar() {
        this.nomes.add(nome);

        // desativa campo e botão quando mais que 3 nomes
        // forem adicionados
        if (this.nomes.size() > 3) {
            this.inputNome.setDisabled(true);
            this.botaoAdicionar.setDisabled(true);
            this.botaoAdicionar.setValue(
```

```
                "Muitos nomes adicionados...");
        }
    }

    // getters e setters

}
```

Para conectar os componentes do formulário com as propriedades criadas, usamos o o atributo `binding` das tags dos componentes.

```
Nome: <h:inputText value="#{nomesBean.nome}"
    binding="#{nomesBean.inputNome}" />
<br/>
<h:commandButton value="Adicionar" action="#{nomesBean.adicionar}"
    binding="#{nomesBean.botaoAdicionar}" />
```

Podemos acessar a página normalmente e adicionar até 4 nomes. A partir daí, o campo e botão são desabilitados e o texto do botão também é modificado.



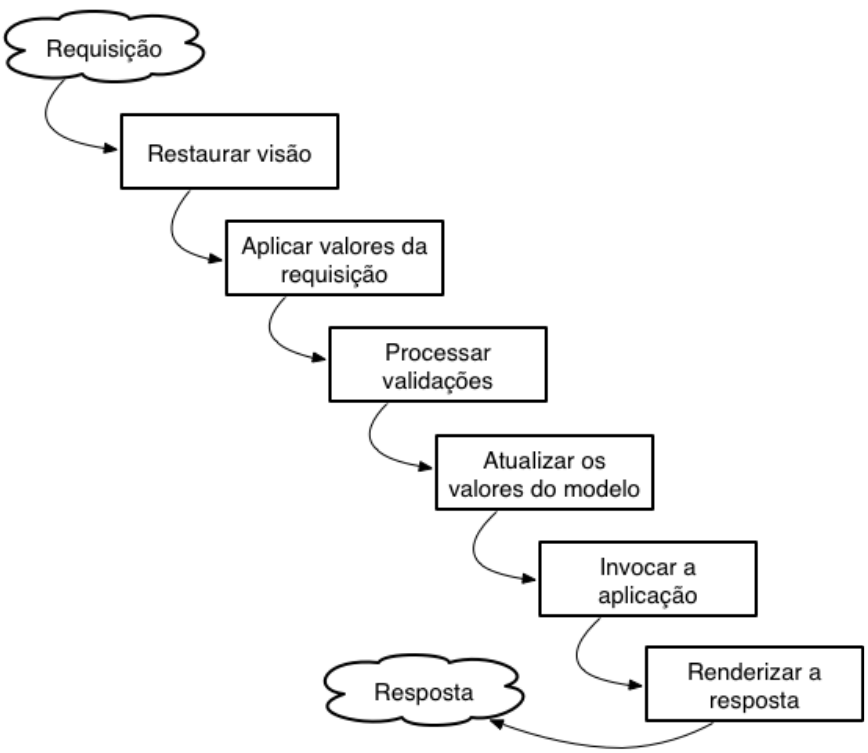
Apesar de poderoso, este recurso deve ser usado com bastante cuidado. O uso excessivo pode deixar o código-fonte grande e difícil de entender. Na maioria das vezes, conseguimos fazer o que precisamos usando apenas expressões de ligação de valor.

3.11. 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:

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



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.

Aplicar valores de requisição

Nesta fase, cada componente da hierarquia de componentes criada na fase anterior tem a chance de atualizar seu próprio estado com informações que vieram da requisição.

Processar validações

Os valores submetidos são convertidos em tipos específicos e anexados aos componentes. Quando você programa uma página em JSF, você pode incluir validadores que atuam nos valores recebidos pelos usuários. Neste momento, os validadores entram em ação e, se surgirem 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.

Atualizar os valores do modelo

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

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 da ação deste botão deve ser executada. Em alguns casos, o método executado pode retornar um identificador dizendo qual é a próxima página a ser exibida, ou simplesmente não retornar nada para exibir a mesma página.

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.

3.12. O arquivo faces-config.xml

Projetos que usam JavaServer Faces podem ter um arquivo de configuração, chamado *faces-config.xml*. Este arquivo é opcional, mas se precisar dele, crie no diretório *src/main/webapp/WEB-INF* do projeto, com o conteúdo mínimo abaixo:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
    version="2.2">

</faces-config>
```

Dentre diversas coisas que podemos configurar nesse arquivo, uma delas é o registro de managed beans, como alternativa às anotações. Por exemplo, veja como registraríamos o managed bean `NomesBean` neste arquivo.

```
<managed-bean>
    <managed-bean-name>nomesBean</managed-bean-name>
    <managed-bean-class>
        com.algaworks.financeiro.controller.NomesBean
    </managed-bean-class>
    <managed-bean-scope>view</managed-bean-scope>
</managed-bean>
```

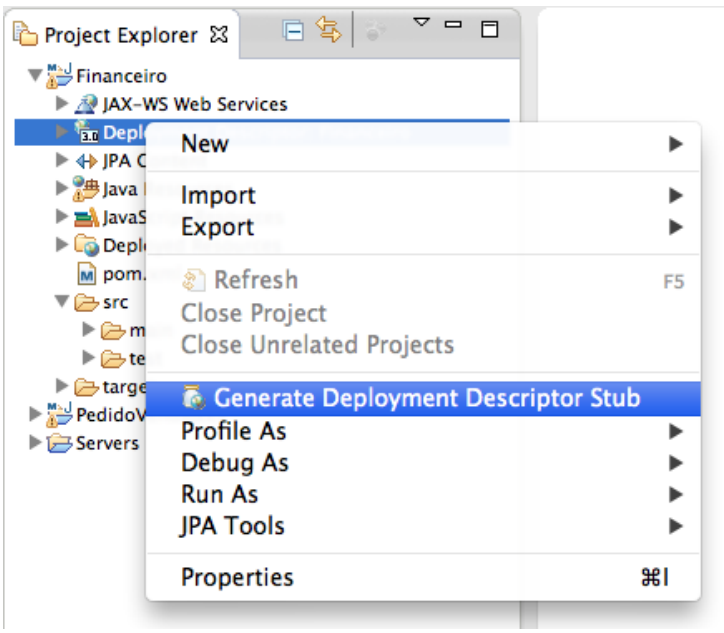
Nas versões anteriores ao JSF 2, não existiam as anotações para registrar managed beans, e por isso, o arquivo *faces-config.xml* era obrigatório. Felizmente, precisaremos

desse arquivo apenas para definir outras configurações da aplicação, que veremos mais adiante.

3.13. O arquivo web.xml

As aplicações web em Java podem ter um arquivo especial, chamado *web.xml*, que deve ficar na pasta *src/main/webapp/WEB-INF* do projeto. Este arquivo também é chamado de **Deployment Descriptor**, pois ele descreve algumas configurações e detalhes de implantação dos recursos da aplicação.

O arquivo *web.xml* não é obrigatório, mas podemos criá-lo clicando com o botão direito em **Deployment Descriptor** (dentro do projeto) e depois em **Generate Deployment Descriptor Stub**.



O arquivo será criado com um conteúdo semelhante ao código abaixo:

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

    <display-name>Financeiro</display-name>
```

```
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>

</web-app>
```

O arquivo gerado descreve o nome da aplicação, através de `display-name`, e alguns arquivos de boas vindas, através de `welcome-file-list`, que são usados caso o usuário acesse o sistema apenas pelo *context path*, sem informar o nome de um recurso.

O *deployment descriptor* de uma aplicação pode descrever também servlets, filtros, mapeamentos e outras configurações.

Para o framework do JavaServer Faces funcionar, ele registra automaticamente uma servlet chamada *Faces Servlet*. Esta servlet é mapeada para o padrão de URL */faces/**, por isso, quando acessamos uma página, escrevemos */faces/NomeDaPagina.xhtml*.

Podemos sobrescrever o registro da *Faces Servlet* e seu mapeamento.

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
</servlet>

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

Com este novo mapeamento da *Faces Servlet*, podemos acessar a página de listagem de nomes através da URL *http://localhost:8080/Financeiro/Nomes.xhtml*.

Vamos aproveitar que estamos editando o arquivo *web.xml* para incluir a configuração abaixo:

```
<context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
</context-param>
```

O parâmetro de contexto adicionado acima diz que estamos em ambiente de desenvolvimento. Quando cometermos algum erro, a página pode exibir informações técnicas adicionais, para nos ajudar a resolver o problema.

Capítulo 4

Navegação

4.1. Introdução à navegação

Em JSF, navegação é um conjunto de regras que define a próxima página a ser exibida quando uma ação é executada pelo usuário. Por exemplo, quando um usuário clica em um botão para se inscrever em um site, qual a próxima página ele deverá visualizar? Se os dados estiverem incompletos, provavelmente deverá visualizar a mesma página, com as mensagens de erro apropriadas, porém se tudo estiver correto e a inscrição for efetuada com sucesso, ele poderá ver uma página de boas vindas ao serviço.

A navegação pode ser implícita ou explícita. Estudaremos como configurar os dois tipos.

4.2. Navegação implícita

Quando incluímos um `<h:commandButton>` com um valor na propriedade `action`, o mecanismo de tratamento de navegação tentará encontrar uma página adequada automaticamente. No exemplo abaixo, o mecanismo de navegação encontrará e encaminhará a requisição para a página *Ola.xhtml*.

```
<h:commandButton value="Próxima página" action="Ola" />
```

O valor passado para a propriedade `action` é chamado de *outcome*, ou resultado da ação.

Navegação dinâmica

Quando um `action` tem uma expressão de ligação de método, o retorno do método deve ser o *outcome* da navegação.

```
<h:commandButton value="Adicionar" action="#{nomesBean.adicionar}" />
```

Quando o método `adicionar` retornar `null`, o usuário permanecerá na mesma página. Quando o retorno for "Ola", a requisição será encaminhada para a página *Ola.xhtml*.

```
public String adicionar() {
    this.nomes.add(nome);

    if (this.nomes.size() > 3) {
        return "Ola";
    }

    return null;
}
```

Redirecionamento

Por padrão, uma requisição é encaminhada para a página do *outcome*, ou seja, a requisição é única e o framework apenas despacha a requisição para uma nova página.

Se for necessário que seja feito o redirecionamento, podemos passar o parâmetro `faces-redirect=true` para o *outcome*.

```
public String adicionar() {
    this.nomes.add(nome);

    if (this.nomes.size() > 3) {
        return "Ola?faces-redirect=true";
    }

    return null;
}
```

4.3. Navegação explícita

As regras de navegação explícitas são declaradas no arquivo *faces-config.xml*. Para declarar uma regra explícita, precisamos informar o caminho da página de origem, um *outcome* com um nome qualquer e o caminho da página de destino.

```
<navigation-rule>
    <from-view-id>/Nomes.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>oi</from-outcome>
        <to-view-id>/Ola.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

A regra que acabamos de declarar define que a ação de nome "oi" a partir da origem */Nomes.xhtml* deve encaminhar para a página */Ola.xhtml*. Para usar esta regra, podemos simplesmente adicionar um botão na página *Nomes.xhtml* com a propriedade `action` referenciando o *outcome*.

```
<h:commandButton value="Próxima página" action="oi" />
```

Wildcard

Podemos usar *wildcard* no elemento `from-view-id` para selecionar diversos arquivos de origem, como por exemplo `*` ou `/admin/*`.

```
<navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
        <from-outcome>oi</from-outcome>
        <to-view-id>/Ola.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

Redirecionamento

Por padrão, a requisição será encaminhada para a página de destino. Se precisarmos fazer o redirecionamento, basta incluir o elemento `<redirect/>`.

```
<navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
        <from-outcome>oi</from-outcome>
        <to-view-id>/Ola.xhtml</to-view-id>
        <redirect/>
    </navigation-case>
</navigation-rule>
```

Capítulo 5

Componentes de interface

5.1. Bibliotecas

Para desenvolver um sistema completo em JavaServer Faces, você deve conhecer pelo menos os principais componentes básicos. Quando falamos em componentes básicos, queremos dizer os componentes padrões da especificação da tecnologia.

As bibliotecas de tags são:

- **Core**: existe para dar suporte às outras bibliotecas, e não possui componentes visuais. Aprenderemos esta biblioteca no decorrer do livro, sempre que for necessário
- **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.
- **Facelets**: é a biblioteca para criação de templates de páginas.
- **Composite**: usada para criar componentes customizados.

Neste capítulo, focaremos nos componentes visuais, da biblioteca **HTML**.

Para usar as bibliotecas, temos que importá-las através dos *namespaces* correspondentes.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
```

```
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:composite="http://xmlns.jcp.org/jsf/composite">
```

Importamos as bibliotecas de tags e nomeamos com os prefixos `f`, `h`, `ui` e `composite`. Os prefixos podem ser modificados, mas os que usamos são os convencionais.

5.2. Cabeçalho e corpo da página

Os componentes `<h:head>` e `<h:body>` renderizam as tags HTML `<head>` e `<body>`, respectivamente.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
    </h:head>

    <h:body>
    </h:body>

</html>
```

A importância destes componentes não é apenas a renderização das tags em HTML, pois seria muito simples escrever as tags HTML diretamente no código da página. Na etapa de renderização da resposta, o JSF pode incluir recursos necessários dinamicamente, como por exemplo referência a arquivos JavaScript e CSS.

5.3. Formulários

JavaServer Faces possui um componente para renderizar formulários HTML, chamado `<h:form>`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Form</title>
    </h:head>

    <h:body>
        <h:form>

        </h:form>
    </h:body>
```

```
</html>
```

A tag <h:form> gerou um elemento <form> da HTML. Apesar da tag <form> da HTML possuir as propriedades `method` e `action`, o componente 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.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"><head id="j_idt2">
        <title>Form</title></head><body>
<form id="j_idt5" name="j_idt5" method="post"
    action="/Financeiro/Form.xhtml"
    enctype="application/x-www-form-urlencoded">
<input type="hidden" name="j_idt5" value="j_idt5" />
<input type="hidden" name="javax.faces.ViewState"
    id="j_id1:javax.faces.ViewState:0"
    value="-1136606116435564932:-1470026908624937807"
    autocomplete="off" />
</form></body>

</html>
```

5.4. Propriedades comuns

As tags dos componentes possuem propriedades que dizem como eles devem funcionar. As propriedades podem ser básicas, quando são compartilhadas 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.

A propriedade id

A propriedade `id` está presente em quase todos os componentes. Ela nos permite identificar os componentes da página para referência 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 um componente JSF. Ao clicar no botão, um código JavaScript irá alterar o conteúdo do campo de entrada.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Propriedades comuns</title>

        <script>
            function alterarValorCampo() {
                var campo = document.getElementById('meuForm:meuCampo');
                campo.value = 'Valor alterado';
            }
        </script>
    </h:head>

    <h:body>
        <h:form id="meuForm">
            <h:inputText id="meuCampo" />

            <input type="button" value="Alterar valor"
                onclick="alterarValorCampo();"/>
        </h:form>
    </h:body>

</html>
```

Veja que criamos uma função JavaScript chamada `alterarValorCampo()`. Esta função é chamada no evento `onclick` do botão "Alterar valor", 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 usado para definir o id do elemento HTML do campo de entrada. Com este identificador, usamos a seguinte programação JavaScript para acessá-lo:

```
var campo = document.getElementById('meuForm:meuCampo');
```

A propriedade binding

A propriedade `binding` pode ser especificada com uma expressão ligação que referencia uma propriedade do bean do tipo do componente. Já usamos este atributo quando falamos sobre *backing beans*.

A propriedade rendered

A propriedade `rendered` também está presente na maioria das tags. Ela controla a renderização do componente. Se o valor ou o resultado da expressão for `false`, o componente não será renderizado.

```
<h:inputText rendered="false" />
<h:inputText rendered="#{meuBean.usuarioAtivo}" />
```

As propriedades style e styleClass

É possível utilizar estilos CSS (*Cascade Style Sheet*) em componentes de modo *inline* ou usando classes CSS.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Estilos</title>

        <style>
            .campo {
                background-color: #ccc;
                color: white
            }
        </style>
    </h:head>

    <h:body>
        <h:form>
            <h:inputText styleClass="campo"
                style="border-color: blue; border-style: dotted" />
        </h:form>
    </h:body>

</html>
```

Falaremos sobre importação de arquivos CSS mais adiante. Nesse exemplo, definimos uma classe CSS no próprio arquivo da página.

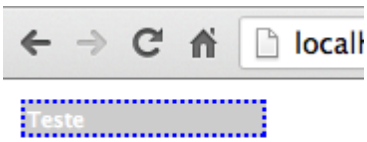
```
<style>
    .campo {
        background-color: #ccc;
        color: white
    }
</style>
```

A classe CSS "campo" configura a cor de fundo para cinza e a cor da fonte para branco.

O campo que adicionamos referencia a classe CSS através da propriedade `styleClass`, e além disso, adiciona novos estilos *inline* na propriedade `style`, configurando uma cor e estilo de borda.

```
<h:inputText styleClass="campo"
    style="border-color: blue; border-style: dotted" />
```

Ao executar a página, podemos ver que as características do campo foram modificadas.



A maioria dos componentes da biblioteca **HTML** possuem as propriedades `style` e `styleClass`.

As propriedades da HTML

As propriedades 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 exemplo.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Propriedades HTML</title>
    </h:head>

    <h:body>
        <h:form>
            <h:inputText size="40" maxlength="50"
                title="Informe seu nome" />
        </h:form>
    </h:body>

</html>
```

Veja como este componente é renderizado:

```
<input type="text" name="j_idt5:j_idt6" maxlength="50"
    size="40" title="Informe seu nome" />
```

As propriedades `maxlength`, `size` e `title` simplesmente foram repassadas para o código de saída, por isso são chamadas de *pass-through* HTML.

As propriedades de eventos DHTML

As propriedades que suportam scripts, que são executados em algum evento do usuário, são chamadas de propriedades de eventos DHTML. Quase todas as tags da biblioteca HTML possuem essas propriedades.

No campo de entrada do exemplo abaixo, usamos algumas propriedades para incluir códigos JavaScript que tratam eventos do usuário.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Propriedades DHTML</title>
    </h:head>

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

</html>
```

Os códigos JavaScript serão renderizados na saída para o navegador. Quando o usuário clicar sobre o campo, seu valor será alterado para vazio, ao alterar o valor do campo, ele será modificado para letras maiúsculas, ao passar o cursor do mouse por cima do campo, a cor de fundo será alterada e ao retirar o cursor do mouse de cima do campo, a cor de fundo ficará branca.

5.5. 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. No exemplo a seguir, o campo de entrada está ligado à propriedade `nome` do managed 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.

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

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}"/>
```

5.6. 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}"
    style="font-weight: bold" />
```

O exemplo acima renderiza o trecho HTML a seguir (considerando que o bean `segurancaBean` exista).

```
Bem-vindo <span style="font-weight: bold">Thiago</span>
```

Veja que o texto "Bem-vindo" não está envolvido 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>`, como aconteceu com "Thiago".

O componente <h:outputFormat>

O componente `<h:outputFormat>` renderiza textos parametrizados na página. Os textos parametrizados são compostos por espaços reservados (*placeholders*), 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 **core**.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>OutputFormat</title>
    </h:head>

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

</html>
```

O componente <h:outputLabel>

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

```
<h:form id="frm">
    <h:outputLabel value="Nome:" for="nomeInput" />
    <h:inputText id="nomeInput" />
</h:form>
```

O código acima renderiza a seguinte saída em HTML:

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

5.7. 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 permite você usar o caminho relativo ao contexto, ou seja, você não precisa informar o caminho do contexto da aplicação, pois é colocado automaticamente.

```
<h:graphicImage value="/imagens/logo.png" width="200" height="36"/>
```

O código acima renderiza a seguinte saída em HTML:

```
<img src="/Financeiro/imagens/logo.png" height="36" width="200" />
```

O arquivo *logo.png* deve ser colocado em *src/main/webapp/imagens*.

Biblioteca de recursos

A partir do JSF 2, todos os recursos web, como imagens, CSS e JavaScript, podem ser colocados no diretório *src/main/webapp/resources*.

Um subdiretório dentro de *resources* é considerado como uma biblioteca de recursos do projeto, e você pode referenciar qualquer um dos recursos pelos atributos `library` e `name` de alguns componentes.

```
<h:graphicImage library="algaworks" name="logo.png"
    width="200" height="36"/>
```

Estamos referenciando o recurso *logo.png* da biblioteca *algaworks*. O caminho do arquivo no projeto é *src/main/webapp/resources/algaworks/logo.png*.

Veja a saída em HTML

```
<img src="/Financeiro/javax.faces.resource/logo.png.xhtml?ln=algaworks"
    height="36" width="200" />
```

5.8. Menus e caixas de listagem

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

O componente <h:selectOneMenu>

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

Precisamos passar uma relação de itens que aparecerão na lista, usando a tag `<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:

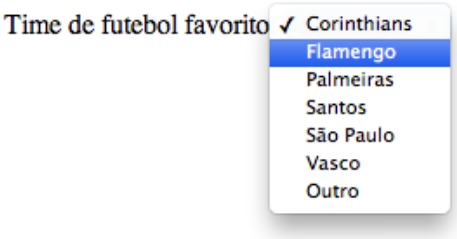
```
<label for="frm:timeFutebol">Time de futebol favorito: </label>
<select id="frm:timeFutebol" name="frm:timeFutebol" size="1">
    <option value="Corinthians">Corinthians</option>
    <option value="Flamengo">Flamengo</option>
    <option value="Palmeiras">Palmeiras</option>
```

```
    <option value="Santos">Santos</option>
    <option value="São Paulo">São 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 a propriedade do managed bean usada na ligação do valor do componente, por isso, precisamos criar um atributo com seu *getter* e *setter* no bean `CadastroTorcedorBean`.

```
private String timeFavorito;

public String getTimeFavorito() {
    return timeFavorito;
}

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

O resultado no browser é o seguinte:



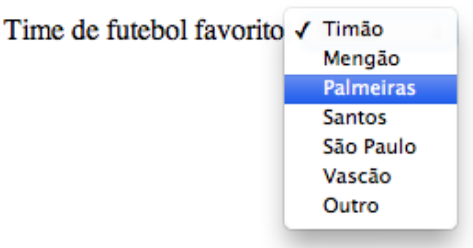
Os valores especificados na tag `<f:selectItem>` são usados também como rótulos dos itens do menu. Algumas vezes temos a necessidade de ter os valores diferentes dos rótulos, e por isso a tag `<f:selectItem>` possui a propriedade `itemLabel`.

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

Veja o código HTML gerado:

```
<label for="frm:timeFutebol">Time de futebol favorito: </label>
<select id="frm:timeFutebol" name="frm:timeFutebol" size="1">
    <option value="Corinthians">Timão</option>
    <option value="Flamengo">Mengão</option>
    <option value="Palmeiras">Palmeiras</option>
    <option value="Santos">Santos</option>
    <option value="São Paulo">São Paulo</option>
    <option value="Vasco">Vascão</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 `multiple` igual a `multiple`.

```
<h:outputLabel value="Times de futebol favoritos: " 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">Times de futebol favoritos: </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="São Paulo">São Paulo</option>
    <option value="Vasco">Vasco</option>
    <option value="Outro">Outro</option>
</select>
```

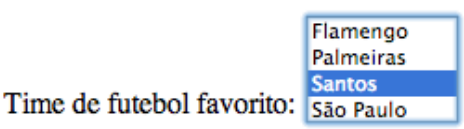
Como este componente possibilita a seleção de mais de um item da seleção, o managed bean deve possuir uma propriedade do tipo de array para comportar os elementos selecionados. No caso deste exemplo, precisamos criar uma propriedade do tipo `String[]` com o nome `timesFavoritos`.

```
private String[] timesFavoritos;

public String[] getTimesFavoritos() {
    return timesFavoritos;
}

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

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>
```

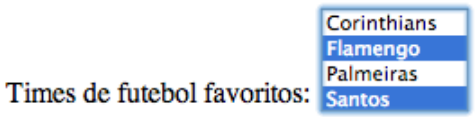


O componente <h:selectManyListbox>

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="Times de futebol favoritos: " for="timeFutebol" />
<h:selectManyListbox value="#{cadastroTorcedorBean.timesFavoritos}"
        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:selectManyListbox>
```

O usuário pode selecionar nenhum, um ou vários itens da seleção.



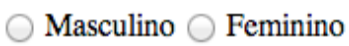
5.9. Campos de checagem e botões rádio

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.

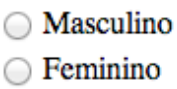
```
<h:selectOneRadio id="sexo">
    <f:selectItem itemValue="M" itemLabel="Masculino" />
    <f:selectItem itemValue="F" itemLabel="Feminino" />
</h:selectOneRadio>
```



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

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

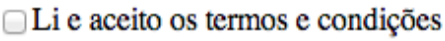
O layout `pageDirection` exibe os itens na vertical.



O componente <h:selectBooleanCheckbox>

O componente `<h:selectBooleanCheckbox>` renderiza o elemento da HTML `<input>` do tipo `checkbox`. Use este componente para definir atributos booleanos.

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



No managed bean, precisamos criar uma propriedade booleana para receber a opção do usuário. Se o campo for selecionado, a propriedade receberá o valor `true`, caso contrário, receberá o valor `false`.

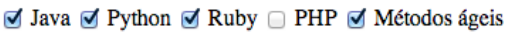
```
private boolean termoAceito;

public boolean isTermoAceito() {
    return termoAceito;
}

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

O componente <h:selectManyCheckbox>

Este componente renderiza uma lista de campos de checagem. 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="3" itemLabel="Python" />
```

```
    <f:selectItem itemValue="4" itemLabel="Ruby" />
    <f:selectItem itemValue="4" itemLabel="PHP" />
    <f:selectItem itemValue="5" itemLabel="Métodos ágeis" />
</h:selectManyCheckbox>
```



Como definimos os valores dos itens da seleção como numéricos e o usuário pode selecionar múltiplos elementos da lista, criamos um atributo no managed bean do tipo `Integer[]`.

```
private Integer[] assuntosInteresse;

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

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

É possível alterar o layout para apresentação dos itens. 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="3" itemLabel="Python" />
    <f:selectItem itemValue="4" itemLabel="Ruby" />
    <f:selectItem itemValue="4" itemLabel="PHP" />
    <f:selectItem itemValue="5" itemLabel="Métodos ágeis" />
</h:selectManyCheckbox>
```

O layout `pageDirection` exibe os itens na vertical.

5.10. Itens de seleção

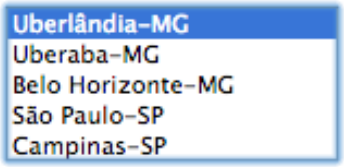
Existem situações que precisamos que uma lista de itens seja obtida dinamicamente, para disponibilizar ao usuário a seleção em qualquer um dos componentes que vimos nas última seções. Esta lista pode vir de um banco de dados, de um arquivo ou de qualquer outra origem. Quando existe essa necessidade,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

Neste exemplo, criaremos uma lista de opções com nomes de cidades. Primeiramente, vamos criar nosso managed bean.

```
@ManagedBean
@ViewScoped
public class MinhaCidadeBean {

    private String cidadeNatal;
    private List<String> todasCidades;

    public MinhaCidadeBean() {
        this.todasCidades = new ArrayList<>();
        this.todasCidades.add("Uberlândia-MG");
        this.todasCidades.add("Uberaba-MG");
        this.todasCidades.add("Belo Horizonte-MG");
        this.todasCidades.add("São Paulo-SP");
        this.todasCidades.add("Campinas-SP");
        this.todasCidades.add("São José dos Campos-SP");
        this.todasCidades.add("Rio de Janeiro-RJ");
        this.todasCidades.add("Goiânia-GO");
        this.todasCidades.add("Fortaleza-CE");
        this.todasCidades.add("Porto Velho-RO");
    }

    public String getCidadeNatal() {
        return cidadeNatal;
    }

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

    public List<String> getTodasCidades() {
        return todasCidades;
    }

}
```

Na página JSF, incluímos o componente `<h:selectOneListbox>` com uma lista de seleção.

```
<h:selectOneListbox size="5" value="#{minhaCidadeBean.cidadeNatal}">
    <f:selectItems value="#{minhaCidadeBean.todasCidades}" />
</h:selectOneListbox>
```



5.11. Botões e links

Aprenderemos a incluir botões e links com os componentes `<h:commandButton>`, `<h:commandLink>` e `<h:outputLink>`.

O componente <h:commandButton>

Já usamos este componente em exemplos anteriores. 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, a propriedade `action` pode ser especificada com expressões de ligação de métodos ou *outcome* de navegação.

```
<h:commandButton id="cadastrar" value="Cadastrar" type="submit"
    action="#{cadastroBean.cadastrar}"/>

<h:commandButton id="limpar" value="Limpar" type="reset"/>

<h:commandButton id="voltar" value="Voltar"
    image="/imagens/voltar.png" action="home"/>
```

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"
    action="#{cadastroBean.cadastrar}"/>
```

```
<h:commandLink id="voltar" action="home">
    <h:graphicImage value="/imagens/voltar.png" />
    <h:outputText value="Voltar"/>
</h:commandLink>
```

Para os links reagirem como um botão e submeter o formulário usando o método *POST*, o framework gera vários códigos em JavaScript. Veja a saída em HTML:

```
<a id="frm:cadastrar" href="#" onclick="mojarra.jsfcljs(
    document.getElementById('frm'),{'frm:cadastrar':'frm:cadastrar'},'');
    return false">Cadastrar</a>
<a id="frm:voltar" href="#" onclick="mojarra.jsfcljs(
    document.getElementById('frm'),{'frm:voltar':'frm:voltar'},'');
    return false"><img src="/Financeiro/imagens/voltar.png" />Voltar</a>
```

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 library="algaworks" name="logo.png" />
</h:outputLink>
```

Este componente facilita a construção de URLs com parâmetros (*querystring*). Os parâmetros são especificados usando a tag `<f:param>`.

```
<h:outputLink value="http://www.algaworks.com" target="_blank">
    <f:param name="codigo" value="931"/>
    <f:param name="grupo" value="Java"/>
    <h:graphicImage library="algaworks" name="logo.png" />
</h:outputLink>
```

Ao clicar no link gerado pelo código acima, somos redirecionados para o endereço *http://www.algaworks.com/?codigo=931&grupo=Java*.

5.12. Painéis

JavaServer Faces possui componentes de painéis que ajudam a organizar os outros componentes da página.

O componente <h:panelGrid>

O componente `<h:panelGrid>` renderiza uma tabela da HTML. Ele funciona como uma grade organizadora de itens no layout da página, e pode acomodar qualquer outro componente.

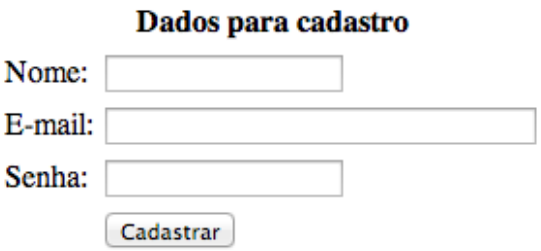
```
<h:panelGrid columns="2">
    <f:facet name="header">
        <h:outputText value="Dados para cadastro" />
    </f:facet>

    <h:outputText value="Nome:" />
    <h:inputText size="20" />

    <h:outputText value="E-mail:" />
    <h:inputText size="40" />

    <h:outputText value="Senha:" />
    <h:inputSecret size="20" />

    <h:outputText />
    <h:commandButton value="Cadastrar" />
</h:panelGrid>
```

O atributo `columns` especifica o número de colunas que queremos por linha. O renderizador do `<h:panelGrid>` organiza os componentes em colunas, da esquerda para a direita, e de cima para baixo.

Cada componente que estiver dentro do painel 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 do painel 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, o nome do *facet* deve ser "header", e para rodapé, "footer".



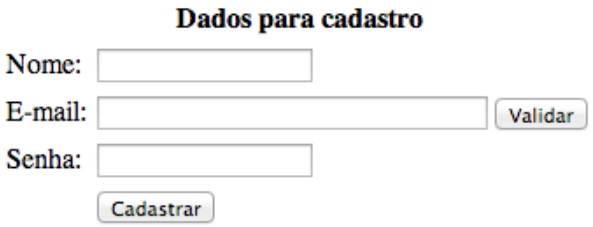
O componente <h:panelGroup>

Este componente é um container que agrupa seus filhos. Ele deve ser usado, especialmente, em situações onde é desejado a inclusão de vários componentes em um lugar que apenas um é permitido, por exemplo, como uma única célula do `<h:panelGrid>`.

Vamos ver como faríamos para incluir um botão ao lado de um campo de entrada de texto usando `<h:panelGrid>` para organizar os componentes.

```
<h:panelGrid columns="2">
    <f:facet name="header">
        <h:outputText value="Dados para cadastro" />
    </f:facet>

    <h:outputText value="Nome:" />
    <h:inputText size="20" />

    <h:outputText value="E-mail:" />
    <h:panelGroup>
        <h:inputText size="40" />
        <h:commandButton value="Validar" />
    </h:panelGroup>

    <h:outputText value="Senha:" />
    <h:inputSecret size="20" />

    <h:outputText />
    <h:commandButton value="Cadastrar" />
</h:panelGrid>
```

Veja o resultado:



5.13. Mensagens

As aplicações precisam 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 mensagens de feedback para os usuários, pois os managed beans possuem o poder de adicionar mensagens a uma fila, que em algum momento pode ser exibida na página.

Criaremos um managed bean que instancia uma mensagem e adiciona ao contexto do framework.

```
@ManagedBean
public class CadastroBean {

    private String nome;

    public void cadastrar() {
        FacesContext context = FacesContext.getCurrentInstance();

        FacesMessage mensagem = new FacesMessage(
            FacesMessage.SEVERITY_INFO, "Cadastro efetuado.",
                "Cliente " + this.nome + " cadastrado com sucesso.");
        context.addMessage(null, mensagem);
    }

    public String getNome() {
        return nome;
    }

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

}
```

Instanciamos um objeto da classe `FacesMessage`, passando como parâmetro a severidade da mensagem, uma mensagem resumida e uma mensagem detalhada. Essa classe fornece outros construtores também.

O método `FacesContext.addMessage` recebe o id do componente que a mensagem será associada e uma instância de `FacesMessage`. Veja que não associamos a mensagem com nenhum componente, pois passamos `null` no primeiro parâmetro.

Para exibir todas as mensagens na página, usamos o componente `<h:messages>`, podendo especificar se queremos exibir a mensagem de detalhe e/ou de resumo, além de aceitar também propriedades de estilos CSS.

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

    <h:outputLabel value="Nome:" />
    <h:inputText value="#{cadastroBean.nome}" />

    <h:commandButton value="Cadastrar"
        action="#{cadastroBean.cadastrar}" />
</h:form>
```

Veja o resultado:

5.14. Tabelas de dados

O componente `<h:dataTable>` gera uma tabela HTML, que deve ser associada a uma lista de elementos de um managed bean, através de uma expressão de ligação de valor. O managed bean pode obter dados dinamicamente, através de um banco de dados, arquivo, etc.

Nosso exemplo será de uma tabela de dados de livros favoritos. Precisamos criar uma classe que representará um livro, chamada `Livro`.

```
public class Livro {

    private String titulo;
    private String autor;

    public Livro() {
    }

    public Livro(String titulo, String autor) {
```

```
        super();
        this.titulo = titulo;
        this.autor = autor;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getAutor() {
        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

}
```

Nosso managed bean `LivrosBean` possuirá uma lista de livros, que será preenchida na instanciação de objetos da classe.

```
@ManagedBean
@ViewScoped
public class LivrosBean {

    private List<Livro> livros;

    public LivrosBean() {
        this.livros = new ArrayList<>();
        this.livros.add(new Livro("Java e Orientação a Objetos",
            "Thiago Faria"));
        this.livros.add(new Livro("JPA 2 e Hibernate", "Thiago Faria"));
        this.livros.add(new Livro("JavaServer Faces", "Thiago Faria"));
        this.livros.add(new Livro("Test Driven Development", "Kent Beck"));
        this.livros.add(new Livro("Start Small, Stay Small",
            "Rob Walling"));
        this.livros.add(new Livro("Trabalhe 4 Horas Por Semana",
            "Timothy Ferris"));
        this.livros.add(new Livro("Getting Real", "Jason Fried"));
        this.livros.add(new Livro("Rework", "Jason Fried"));
    }

    public List<Livro> getLivros() {
        return livros;
    }

    public void setLivros(List<Livro> livros) {
        this.livros = livros;
    }
```

```
}
```

Criaremos agora uma página *Livros.xhtml*, que listará os livros favoritos através do componente `<h:dataTable>`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Livros favoritos</title>
    </h:head>

    <h:body>
        <h:form>
            <h:dataTable value="#{livrosBean.livros}" var="livro"
                    border="1" cellspacing="0" cellpadding="2">
                <f:facet name="header">
                    Relação de livros favoritos
                </f:facet>

                <h:column>
                    <f:facet name="header">Título</f:facet>
                    <h:outputText value="#{livro.titulo}" />
                </h:column>
                <h:column>
                    <f:facet name="header">Autor</f:facet>
                    <h:outputText value="#{livro.autor}" />
                </h:column>
            </h:dataTable>
        </h:form>
    </h:body>

</html>
```

Quando este componente é processado, 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 pelo atributo `var`, possibilitando o acesso dentro das colunas.

```
<h:dataTable value="#{livrosBean.livros}" var="livro"
        border="1" cellspacing="0" cellpadding="2">
```

Definimos o *facet* **header** da tabela de dados, que corresponde ao cabeçalho da própria tabela.

```
<f:facet name="header">Relação de livros favoritos</f:facet>
```

Inserimos duas colunas com `<h:column>`. Dentro de cada coluna, podemos incluir qualquer quantidade de componentes. No caso deste exemplo, inserimos apenas um `<h:outputText>`, além do *facet* **header**, que corresponde ao cabeçalho da coluna.

```
<h:column>
    <f:facet name="header">Título</f:facet>
    <h:outputText value="#{livro.titulo}" />
</h:column>
<h:column>
    <f:facet name="header">Autor</f:facet>
    <h:outputText value="#{livro.autor}" />
</h:column>
```

Relação de livros favoritos	
Título	**Autor**
Java e Orientação a Objetos	Thiago Faria
JPA 2 e Hibernate	Thiago Faria
JavaServer Faces	Thiago Faria
Test Driven Development	Kent Beck
Start Small, Stay Small	Rob Walling
Trabalhe 4 Horas Por Semana	Timothy Ferris
Getting Real	Jason Fried
Rework	Jason Fried

Formulário para inclusão de livros

Deixaremos a página de livros favoritos um pouco mais dinâmica, possibilitando que o próprio usuário adicione novos livros. Vamos alterar o managed bean `LivrosBean` para o código abaixo:

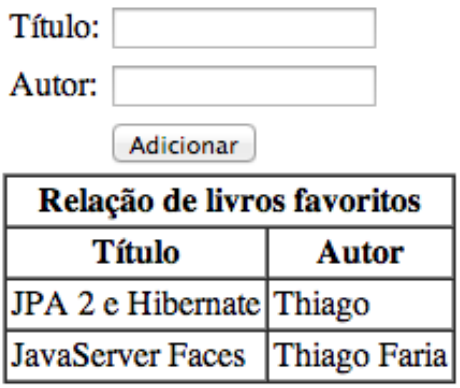
```
@ManagedBean
@ViewScoped
public class LivrosBean {

    private List<Livro> livros;
    private Livro novoLivro;

    public LivrosBean() {
        this.livros = new ArrayList<>();
        this.novoLivro = new Livro();
    }
```

```
    public void adicionar() {
        this.livros.add(this.novoLivro);
        this.novoLivro = new Livro();
    }

    public List<Livro> getLivros() {
        return livros;
    }

    public void setLivros(List<Livro> livros) {
        this.livros = livros;
    }

    public Livro getNovoLivro() {
        return novoLivro;
    }

}
```

No arquivo *Livros.xhtml*, incluiremos um painel com os campos e botão para adição de novos livros à relação de livros favoritos.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">

    <h:head>
        <title>Livros favoritos</title>
    </h:head>

    <h:body>
        <h:form>
            <h:panelGrid columns="2">
                <h:outputLabel value="Título: "/>
                <h:inputText value="#{livrosBean.novoLivro.titulo}" />

                <h:outputLabel value="Autor: "/>
                <h:inputText value="#{livrosBean.novoLivro.autor}" />

                <h:outputLabel />
                <h:commandButton value="Adicionar"
                    action="#{livrosBean.adicionar}" />
            </h:panelGrid>

            <h:dataTable value="#{livrosBean.livros}" var="livro"
                    border="1" cellspacing="0" cellpadding="2">
                <f:facet name="header">
                    Relação de livros favoritos
                </f:facet>

                <h:column>
                    <f:facet name="header">Título</f:facet>
                    <h:outputText value="#{livro.titulo}" />
```

```
                </h:column>
                <h:column>
                    <f:facet name="header">Autor</f:facet>
                    <h:outputText value="#{livro.autor}" />
                </h:column>
            </h:dataTable>
        </h:form>
    </h:body>

</html>
```



5.15. Arquivos JavaScript e CSS

Os componentes `<h:outputStylesheet>` e `<h:outputScript>` podem ser usados para adicionar arquivos CSS e JavaScript da bilioteca de recursos do projeto.

Para exemplificar, criaremos um arquivo CSS simples chamado *estilo.css* no diretório *src/main/webapp/resources/algaworks* do projeto.

```
body {
    background-color: yellow
}
```

Criaremos também, no mesmo diretório, um arquivo JavaScript chamado *script.js*.

```
alert('Olá!');
```

No arquivo XHTML, usamos os componentes para importar os arquivos CSS e JavaScript.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
```

```
    <h:head>
        <title>Biblioteca de recursos</title>
    </h:head>

    <h:body>
        <h:outputStylesheet library="algaworks" name="estilo.css" />
        <h:outputScript library="algaworks" name="script.js"
            target="head" />
    </h:body>

</html>
```

A tag <h:outputScript> aceita uma propriedade target, para você especificar um destino diferente de onde o componente está. Por exemplo, mesmo o componente estando dentro de <h:body>, ele será renderizado em <head> no HTML gerado. Veja:

```
<head id="j_idt2">
<title>Biblioteca de recursos</title>
<link type="text/css" rel="stylesheet"
    href="/Financeiro/javax.faces.resource/estilo.css.xhtml?ln=algaworks"
/>
<script type="text/javascript"
    src="/Financeiro/javax.faces.resource/script.js.xhtml?ln=algaworks">
</script>
</head>
```

Capítulo 6

Página de consulta de lançamentos

6.1. Criando EntityManager

Os sistemas que usam JPA precisam de apenas uma instância de `EntityManagerFactory`. Esta única instância pode ser usada por qualquer código que queira obter um `EntityManager`.

Um `EntityManager` é responsável por gerenciar entidades no contexto de persistência. Através dos métodos dessa interface, é possível persistir, pesquisar e excluir objetos do banco de dados.

A inicialização de `EntityManagerFactory` pode demorar alguns segundos, por isso a instância dessa interface deve ser compartilhada na aplicação.

Precisaremos de um lugar para colocar a instância compartilhada de `EntityManagerFactory`, onde qualquer código tenha acesso fácil e rápido. Criaremos a classe `JpaUtil` para armazenar a instância em uma variável estática.

```
package com.algaworks.financeiro.util;

// imports...

public class JpaUtil {

    private static EntityManagerFactory factory;

    static {
        factory = Persistence.createEntityManagerFactory(
            "FinanceiroPU");
```

```
    }

    public static EntityManager getEntityManager() {
        return factory.createEntityManager();
    }

}
```

Criamos um bloco estático para inicializar a fábrica de *Entity Manager*. Isso ocorrerá apenas uma vez, no carregamento da classe. Agora, sempre que precisarmos de uma `EntityManager`, podemos chamar:

```
EntityManager manager = JpaUtil.getEntityManager();
```

6.2. Persistindo pessoas e lançamentos

Neste capítulo, criaremos uma página JSF que consulta lançamentos do banco de dados, por isso, precisamos de registros na tabela de lançamentos.

O código abaixo persiste (insere) pessoas e lançamentos nas tabelas correspondentes.

```
// imports...

public class CriaLancamentos {

    public static void main(String[] args) {
        EntityManager manager = JpaUtil.getEntityManager();
        EntityTransaction trx = manager.getTransaction();
        trx.begin();

        Calendar dataVencimento1 = Calendar.getInstance();
        dataVencimento1.set(2013, 10, 1, 0, 0, 0);

        Calendar dataVencimento2 = Calendar.getInstance();
        dataVencimento2.set(2013, 12, 10, 0, 0, 0);

        Pessoa cliente = new Pessoa();
        cliente.setNome("WWW Indústria de Alimentos");

        Pessoa fornecedor = new Pessoa();
        fornecedor.setNome("SoftBRAX Treinamentos");

        Lancamento lancamento1 = new Lancamento();
        lancamento1.setDescricao("Venda de licença de software");
        lancamento1.setPessoa(cliente);
        lancamento1.setDataVencimento(dataVencimento1.getTime());
        lancamento1.setDataPagamento(dataVencimento1.getTime());
        lancamento1.setValor(new BigDecimal(103_000));
        lancamento1.setTipo(TipoLancamento.RECEITA);
```

```
        Lancamento lancamento2 = new Lancamento();
        lancamento2.setDescricao("Venda de suporte anual");
        lancamento2.setPessoa(cliente);
        lancamento2.setDataVencimento(dataVencimento1.getTime());
        lancamento2.setDataPagamento(dataVencimento1.getTime());
        lancamento2.setValor(new BigDecimal(15_000));
        lancamento2.setTipo(TipoLancamento.RECEITA);

        Lancamento lancamento3 = new Lancamento();
        lancamento3.setDescricao("Treinamento da equipe");
        lancamento3.setPessoa(fornecedor);
        lancamento3.setDataVencimento(dataVencimento2.getTime());
        lancamento3.setValor(new BigDecimal(68_000));
        lancamento3.setTipo(TipoLancamento.DESPESA);

        manager.persist(cliente);
        manager.persist(fornecedor);
        manager.persist(lancamento1);
        manager.persist(lancamento2);
        manager.persist(lancamento3);

        trx.commit();
        manager.close();
    }

}
```

Depois que pegamos um `EntityManager`, iniciamos uma transação para a operação de persistência.

```
EntityManager manager = JpaUtil.getEntityManager();
EntityTransaction trx = manager.getTransaction();
trx.begin();
```

Instanciamos os objetos que queremos persistir e chamamos o método `EntityManager.persist()` em cada um deles. Depois, fazemos *commit* na transação e fechamos o *Entity Manager*.

```
manager.persist(cliente);
manager.persist(fornecedor);
manager.persist(lancamento1);
manager.persist(lancamento2);
manager.persist(lancamento3);

trx.commit();
manager.close();
```

6.3. Managed bean que consulta lançamentos

Criaremos um managed bean que consulta os lançamentos no banco de dados e atribui a uma variável de instância, que poderá ser consultada pela página JSF através de uma expressão de ligação de valor.

Podemos consultar objetos de entidades JPA com a linguagem JPQL (*Java Persistence Query Language*). A JPQL é uma extensão da SQL, porém com características da orientação a objetos. Com essa linguagem, não referenciamos tabelas do banco de dados, mas apenas entidades de nosso modelo, que foram mapeadas para tabelas.

Quando fazemos pesquisas em objetos, não precisamos selecionar as colunas do banco de dados, como é o caso de SQL. O código em SQL a seguir:

```
select * from lancamento
```

Fica da seguinte forma em JPQL:

```
from Lancamento
```

A sintaxe acima em JPQL significa que queremos buscar os objetos persistentes da entidade `Lanamento`.

Para fazer uma consulta usando JPQL, pegamos uma instância de `EntityManager` e chamamos o método `createQuery`, passando como parâmetro a string da *query* e o tipo esperado como retorno. O retorno será do tipo `TypedQuery`, que podemos usar para obter uma lista com o resultado da consulta, através do método `getResultList`.

```
EntityManager manager = JpaUtil.getEntityManager();
TypedQuery<Lancamento> query = manager.createQuery(
    "from Lancamento", Lancamento.class);
List<Lancamento> lancamentos = query.getResultList();
```

Sabendo disso, podemos implementar o managed bean que consulta os lançamentos no banco de dados.

```
@ManagedBean
@ViewScoped
public class ConsultaLancamentosBean {

    private List<Lancamento> lancamentos;

    public void consultar() {
        EntityManager manager = JpaUtil.getEntityManager();

        TypedQuery<Lancamento> query = manager.createQuery(
```

```
            "from Lancamento", Lancamento.class);
        this.lancamentos = query.getResultList();

        manager.close();
    }

    public List<Lancamento> getLancamentos() {
        return lancamentos;
    }

}
```

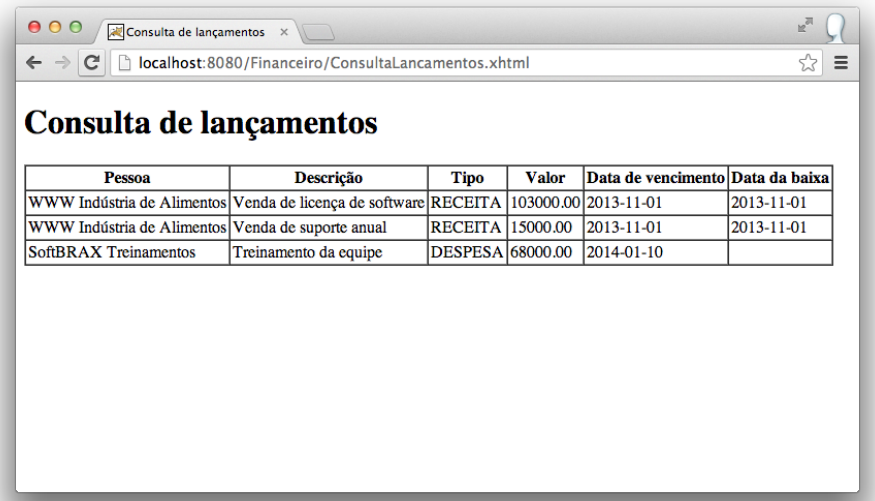
6.4. Página de resultado da consulta

Criaremos um arquivo *ConsultaLancamentos.xhtml*, que terá um `<h:dataTable>` com diversas colunas, que exigem os dados dos lançamentos.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:head>
    <title>Consulta de lançamentos</title>
</h:head>

<h:body>
    <f:metadata>
        <f:viewAction action="#{consultaLancamentosBean.consultar}" />
    </f:metadata>

    <h1>Consulta de lançamentos</h1>

    <h:form id="frm">
        <h:dataTable value="#{consultaLancamentosBean.lancamentos}"
                var="lancamento" border="1" cellspacing="0"
                cellpadding="2">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Pessoa"/>
                </f:facet>
                <h:outputText value="#{lancamento.pessoa.nome}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Descrição"/>
                </f:facet>
                <h:outputText value="#{lancamento.descricao}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
```

```
                    <h:outputText value="Tipo"/>
                </f:facet>
                <h:outputText value="#{lancamento.tipo}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Valor"/>
                </f:facet>
                <h:outputText value="#{lancamento.valor}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Data de vencimento"/>
                </f:facet>
                <h:outputText value="#{lancamento.dataVencimento}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Data de pagamento"/>
                </f:facet>
                <h:outputText value="#{lancamento.dataPagamento}"/>
            </h:column>
        </h:dataTable>

    </h:form>
</h:body>

</html>
```

Usamos `<f:viewAction>` para chamar o método `consultar()` do managed bean na fase de invocação da aplicação, ou seja, antes de renderizar a página. Este recurso está disponível a partir do JSF 2.2 (Java EE 7).



Por enquanto, não iremos nos preocupar com as formatações de datas, números, etc.

6.5. O padrão Repository

No managed bean `ConsultaLancamentosBean` possui código de acesso a dados, e isso pode não ser interessante, principalmente em aplicações médias e grandes, pois não conseguiremos reaproveitar a lógica de acesso aos dados. Por isso, vamos aprender e implementar o padrão *Repository*.

O padrão *Repository* tem como objetivo isolar o código de acesso a dados de qualquer outra lógica da aplicação, ou seja, atua como um mediador entre a camada de domínio (negócio) e acesso a dados, sendo representado, conceitualmente, como se fosse uma coleção de objetos de um tipo específico. Um repositório deve fornecer métodos para adicionar, atualizar, remover e/ou buscar objetos nessa "coleção".

Nosso primeiro repositório representará uma coleção de objetos do tipo `Lancamento`. Podemos chamar nosso repositório de `LancamentoRepository`, `RepositorioLancamento`, etc, mas preferimos chamá-lo de `Lancamentos`.

```
package com.algaworks.financeiro.repository;

// imports...

public class Lancamentos {
```

```
    private EntityManager manager;

    public Lancamentos(EntityManager manager) {
        this.manager = manager;
    }

    public List<Lancamento> todos() {
        TypedQuery<Lancamento> query = manager.createQuery(
            "from Lancamento", Lancamento.class);
        return query.getResultList();
    }

}
```

O construtor do repositório `Lancamentos` receberá um *Entity Manager*, ou seja, não é responsabilidade do repositório obter um *Entity Manager*.

O método `todos` faz a consulta de todos os lançamentos e retorna um `List<Lancamento>`.

Agora, alteramos nosso managed bean para usar o repositório `Lancamentos`.

```
@ManagedBean
@ViewScoped
public class ConsultaLancamentosBean {

    private List<Lancamento> lancamentos;

    public void consultar() {
        EntityManager manager = JpaUtil.getEntityManager();
        Lancamentos lancamentos = new Lancamentos(manager);

        this.lancamentos = lancamentos.todos();

        manager.close();
    }

    public List<Lancamento> getLancamentos() {
        return lancamentos;
    }

}
```

Talvez você ainda não consiga ver uma grande vantagem em usar este padrão, pois nosso exemplo ainda é muito simples, mas a medida que o projeto for crescendo, o uso de repositórios começará a fazer mais sentido.

Capítulo 7

Templates com Facelets

7.1. Qual é o problema de repetir?

Geralmente, as páginas de um mesmo sistema seguem um layout muito similar. Por exemplo, o cabeçalho, menu e rodapé são os mesmos durante toda a navegação na aplicação. Repetir esse código em todas as páginas torna difícil a manutenção do sistema, por isso, é recomendado o uso de Facelets para criar templates que podem ser reutilizados por todas as páginas do projeto.

Neste capítulo, criaremos o template do sistema financeiro.

7.2. Incluindo um cabeçalho e rodapé

Incluiremos um cabeçalho e rodapé na página *ConsultaLancamentos.xhtml*, além de alguns estilos CSS.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:head>
    <title>Consulta de lançamentos</title>
    <h:outputStylesheet library="algaworks" name="estilo.css" />
</h:head>

<h:body>
    <f:metadata>
        <f:viewAction action="#{consultaLancamentosBean.consultar}" />
    </f:metadata>
```

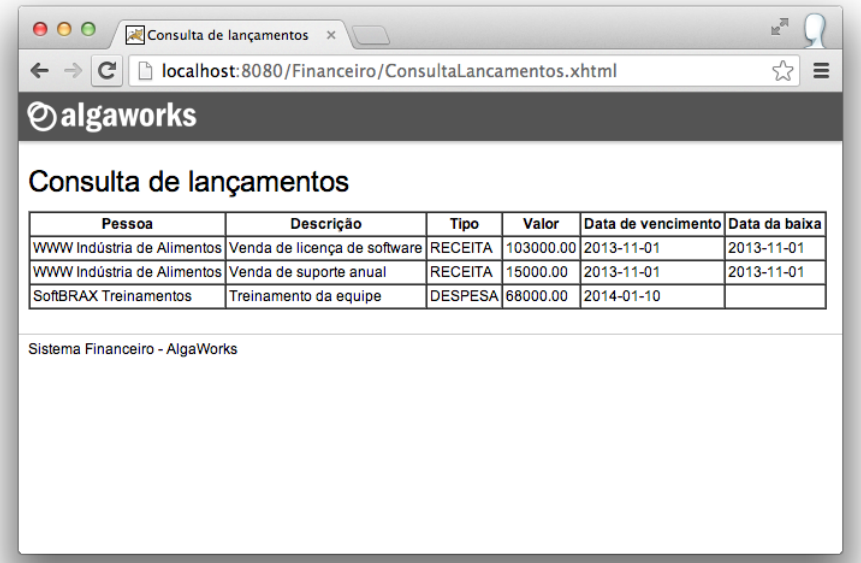
```
<header>
    <h:graphicImage library="algaworks" name="logo.png" />
</header>

<div id="conteudo">
    <h1>Consulta de lançamentos</h1>

    <h:form id="frm">
        <h:dataTable value="#{consultaLancamentosBean.lancamentos}"
                var="lancamento" border="1" cellspacing="0"
                cellpadding="2">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Pessoa"/>
                </f:facet>
                <h:outputText value="#{lancamento.pessoa.nome}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Descrição"/>
                </f:facet>
                <h:outputText value="#{lancamento.descricao}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Tipo"/>
                </f:facet>
                <h:outputText value="#{lancamento.tipo}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Valor"/>
                </f:facet>
                <h:outputText value="#{lancamento.valor}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Data de vencimento"/>
                </f:facet>
                <h:outputText value="#{lancamento.dataVencimento}"/>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Data de pagamento"/>
                </f:facet>
                <h:outputText value="#{lancamento.dataPagamento}"/>
            </h:column>
        </h:dataTable>

    </h:form>
</div>

<footer>
    Sistema Financeiro - AlgaWorks
```

```
    </footer>

</h:body>

</html>
```

O arquivo *estilo.css* foi criado com o conteúdo abaixo:

```
@charset "utf-8";

body {
    font-size: 12px;
    font-family: Arial, Helvetica, sans-serif;
    margin: 0px;
    font-weight: normal
}

header {
    padding: 5px;
    margin-bottom: 20px;
    height: 30px;
    background-color: #545454;
    color: #fff;
    box-shadow: 0px 2px 2px #ccc
}

#conteudo {
    padding: 0px 8px
}

footer {
    border-top: 1px solid #ccc;
    padding: 5px 8px;
    margin-top: 20px;
    margin-bottom: 10px
}

h1 {
    font-size: 24px;
    font-weight: 500;
    padding: 0px;
    margin: 0px;
    margin-bottom: 10px
}
```

Ainda não está perfeita, mas o visual da página de consulta de lançamentos já melhorou bastante.



7.3. Criando um template

Podemos criar templates de páginas e reutilizá-los em todas as páginas de nosso sistema, evitando replicação de código de layout comum. Incluímos um cabeçalho e rodapé na página de consulta de lançamentos, e nas próximas páginas do sistema financeiro, precisaríamos fazer a mesma coisa.

Vamos criar um template para o sistema financeiro. Criaremos um arquivo *Layout.xhtml* no diretório *src/main/webapp/WEB-INF/template* com o código abaixo:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

<h:head>
    <title>
        <ui:insert name="titulo">Sistema Financeiro</ui:insert>
    </title>
    <h:outputStylesheet library="algaworks" name="estilo.css" />
</h:head>

<h:body>
    <header>
```

```
        <h:graphicImage library="algaworks" name="logo.png" />
    </header>

    <div id="conteudo">
        <ui:insert name="corpo" />
    </div>

    <footer>
        Sistema Financeiro - AlgaWorks
    </footer>
</h:body>

</html>
```

Importamos o *namespace* de Facelets (*http://xmlns.jcp.org/jsf/facelets*) e usamos a tag `<ui:insert>`. Esta tag define regiões no template, que em tempo de execução receberão trechos de código fornecidos pelas páginas que utilizarem o template. Neste template criamos duas regiões, com os nomes "titulo" e "corpo".

7.4. Usando o template

Agora, todas nossas páginas podem usar o arquivo de template *Layout.xhtml*. Editaremos o arquivo *ConsultaLancamentos.xhtml* e adequaremos para que ele fique da seguinte forma:

```
<!DOCTYPE html>
<ui:composition template="/WEB-INF/template/Layout.xhtml"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

    <f:metadata>
        <f:viewAction action="#{consultaLancamentosBean.consultar}" />
    </f:metadata>

    <ui:define name="titulo">Consulta de lançamentos</ui:define>

    <ui:define name="corpo">
        <h1>Consulta de lançamentos</h1>

        <h:form id="frm">
            <h:dataTable value="#{consultaLancamentosBean.lancamentos}"
                    var="lancamento" border="1" cellspacing="0"
                    cellpadding="2">
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Pessoa"/>
                    </f:facet>
```

```
                    <h:outputText value="#{lancamento.pessoa.nome}"/>
                </h:column>
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Descrição"/>
                    </f:facet>
                    <h:outputText value="#{lancamento.descricao}"/>
                </h:column>
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Tipo"/>
                    </f:facet>
                    <h:outputText value="#{lancamento.tipo}"/>
                </h:column>
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Valor"/>
                    </f:facet>
                    <h:outputText value="#{lancamento.valor}"/>
                </h:column>
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Data de vencimento"/>
                    </f:facet>
                    <h:outputText value="#{lancamento.dataVencimento}"/>
                </h:column>
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Data de pagamento"/>
                    </f:facet>
                    <h:outputText value="#{lancamento.dataPagamento}"/>
                </h:column>
            </h:dataTable>
        </h:form>
    </ui:define>

</ui:composition>
```

Usamos a tag `<ui:composition>` para referenciar o arquivo de template e importar as bibliotecas de componentes. Depois, especificamos os conteúdos das regiões "tela" e "corpo" através da tag `<ui:define>`.

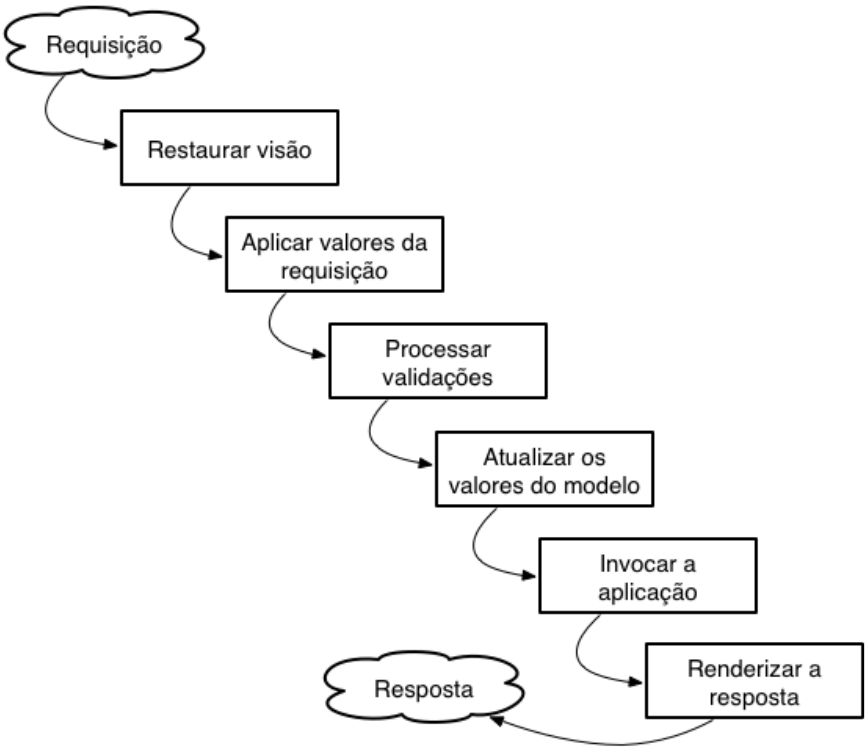
Capítulo 8

Conversão e validação

8.1. Introdução

Estudaremos neste capítulo como os dados informados pelos usuários são convertidos para objetos Java e como é feita a validação desses objetos 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 chega ao servidor e é chamada de **valor de requisição** (*request value*).

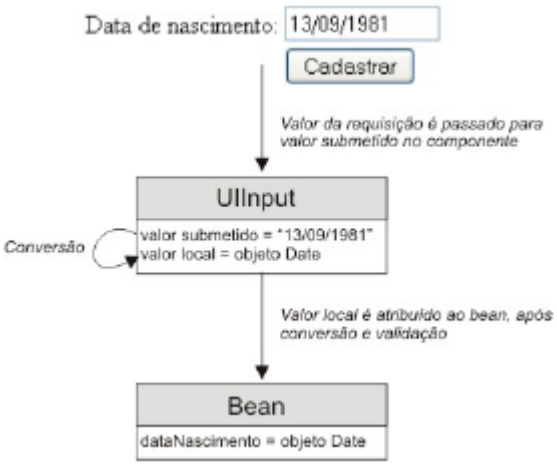
Na fase **Aplicar valores de requisição** do ciclo de vida, os valores de requisição são anexados em objetos de componentes e chamados de **valor submetido** (*submitted value*). Cada componente da página possui um objeto correspondente na memória do servidor, que armazena o valor submetido.

Os valores de requisição são do tipo `String`, pois tudo em HTTP é enviado como texto. No código Java, precisamos lidar com tipos específicos, como um inteiro, ponto-flutuante, data, etc. Existe então um processo de conversão e validação que é executado pelo framework, que converte os dados em string para seus tipos específicos e valida na fase **Processar validações**.

Os valores convertidos e validados 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 e validados.

Inicia-se então a execução da fase **Atualizar os valores do modelo**, que atribui os valores locais convertidos e validados aos beans.

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.



8.2. Conversores padrão

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

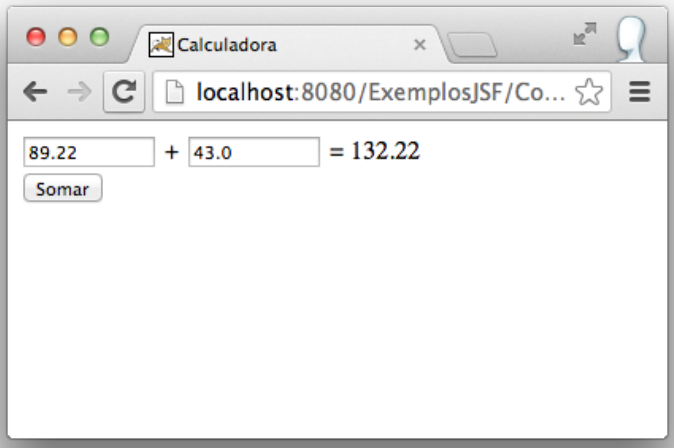
JavaServer Faces fornece vários conversores prontos para serem usados. Todos os tipos primitivos e classes *wrappers*, além de `BigInteger` e `BigDecimal`, usam conversores padrão do JSF automaticamente. Por exemplo, se você colocar um `<h:inputText>` referenciando um atributo do tipo `Double` de um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

```
@ManagedBean
public class CalculadoraBean {

    private Double valorA;
    private Double valorB;
    private Double resultado;

    public void somar() {
        this.resultado = this.valorA + this.valorB;
    }
```

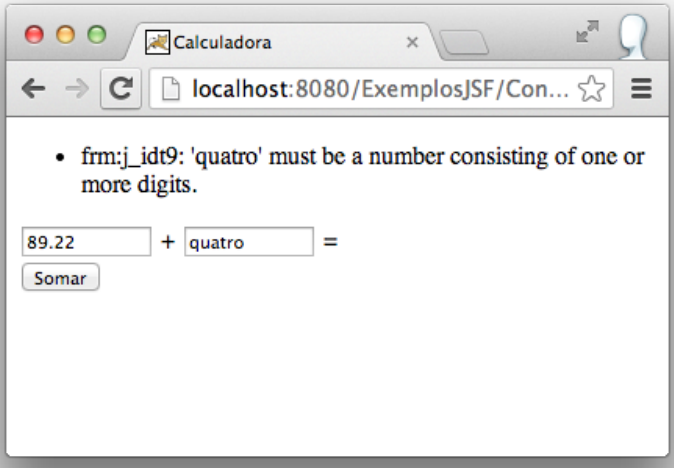
```
    // getters e setters

}
```

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:head>
    <title>Calculadora</title>
</h:head>

<h:body>
    <h:form id="frm">
        <h:messages />

        <h:inputText size="12" value="#{calculadoraBean.valorA}" />
        +
        <h:inputText size="12" value="#{calculadoraBean.valorB}" />
        =
        <h:outputText value="#{calculadoraBean.resultado}" />
        <br />

        <h:commandButton value="Somar"
            action="#{calculadoraBean.somar}" />
    </h:form>
</h:body>

</html>
```

O managed bean recebeu os valores digitados como `Double` e efetuou a soma deles.



Se informarmos um valor que não pode ser convertido para `Double`, recebemos uma mensagem de erro de conversão.



A tag <f:convertNumber>

As vezes precisamos especificar detalhes para que a conversão ocorra como desejamos. A tag `<f:convertNumber>` permite algumas configurações para conversão de entradas numéricas.

```
<h:inputText size="12" value="#{calculadoraBean.valorA}">
    <f:convertNumber minFractionDigits="2" maxFractionDigits="2"
        locale="pt_BR" />
</h:inputText>
+
<h:inputText size="12" value="#{calculadoraBean.valorB}">
    <f:convertNumber minFractionDigits="2" maxFractionDigits="2"
        locale="pt_BR" />
</h:inputText>
=
<h:outputText value="#{calculadoraBean.resultado}">
    <f:convertNumber type="currency" locale="pt_BR" />
</h:outputText>
```

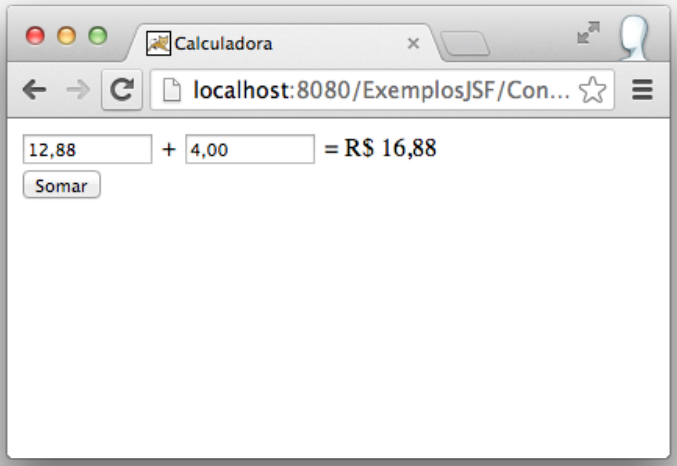
Incluímos a tag `<f:convertNumber>` nos campos de entrada e especificamos que as casas decimais podem ter no mínimo e no máximo 2 dígitos, além de especificar o *locale* brasileiro, que forçará o uso de vírgulas para separação decimal.

```
<h:inputText size="12" value="#{calculadoraBean.valorA}">
    <f:convertNumber minFractionDigits="2" maxFractionDigits="2"
        locale="pt_BR" />
</h:inputText>
```

Podemos especificar um conversor para componentes de saída também. Neste caso, eles funcionam de forma inversa, ou seja, convertem de tipos específicos para strings. No exemplo, usamos um conversor para exibir o resultado da soma no formato de moeda brasileira.

```
<h:outputText value="#{calculadoraBean.resultado}">
    <f:convertNumber type="currency" locale="pt_BR" />
</h:outputText>
```

Veja o resultado:



A tag <f:convertDateTime>

Esta tag faz conversões de/para o tipo `java.util.Date`. Criaremos uma calculadora de datas, onde o usuário informará uma data base e uma quantidade de dias para adicionar à data.

```
@ManagedBean
public class CalculadoraDataBean {

    private Date dataBase;
    private Integer dias;
    private Date resultado;

    public void adicionar() {
        Calendar dataCalculo = Calendar.getInstance();
        dataCalculo.setTime(this.dataBase);
        dataCalculo.add(Calendar.DAY_OF_MONTH, dias);

        this.resultado = dataCalculo.getTime();
    }

    // getter e setter

}
```

Incluímos os campos no formulário e usamos a tag `<f:convertDateTime>`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:head>
    <title>Calculadora</title>
</h:head>

<h:body>
    <h:form id="frm">
        <h:messages showDetail="true" showSummary="false" />

        <h:inputText size="12" value="#{calculadoraDataBean.dataBase}">
            <f:convertDateTime pattern="dd/MM/yyyy" />
        </h:inputText>
        +
        <h:inputText size="12" value="#{calculadoraDataBean.dias}" />
        dias =
        <h:outputText value="#{calculadoraDataBean.resultado}">
            <f:convertDateTime dateStyle="full" locale="pt_BR"/>
        </h:outputText>
        <br />

        <h:commandButton value="Adicionar"
            action="#{calculadoraDataBean.adicionar}" />
    </h:form>
</h:body>

</html>
```

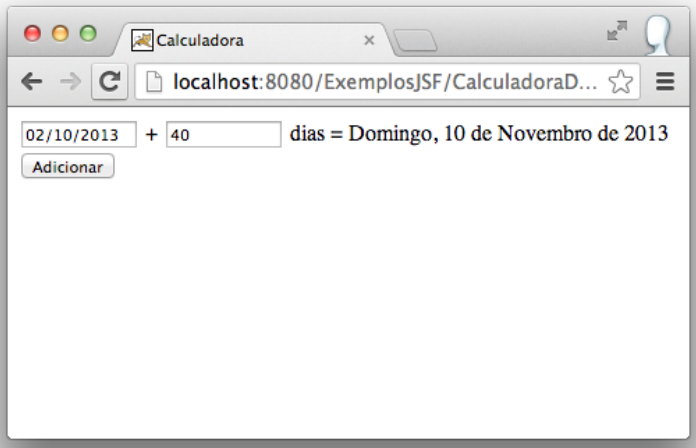
A propriedade `pattern` define o padrão de formatação da data que o usuário precisará digitar. Esse formato é o mesmo usado na classe `SimpleDateFormat` da API.

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

A data de resultado será exibida em um estilo completo, em português brasileiro. Usamos as propriedades `dateStyle` e `locale` para configurar isso.

```
<h:outputText value="#{calculadoraDataBean.resultado}">
    <f:convertDateTime dateStyle="full" locale="pt_BR"/>
</h:outputText>
```

Veja o resultado:



8.3. Alternativas para definir conversores

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

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

O JavaServer Faces já possui alguns IDs de conversores pré-definidos:

- 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 deixar explícito qual conversor queremos usar é com tag `<f:converter>`, especificando o ID do conversor no atributo `converterId`.

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

8.4. Customizando mensagens de erro de conversão

Você pode precisar customizar as mensagens de erros, quando as conversões não são bem sucedidas. Primeiramente, customizaremos as mensagens de erro de conversão de campos de datas.

Para começar, criaremos um arquivo de *messages bundle* e incluiremos 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.
```

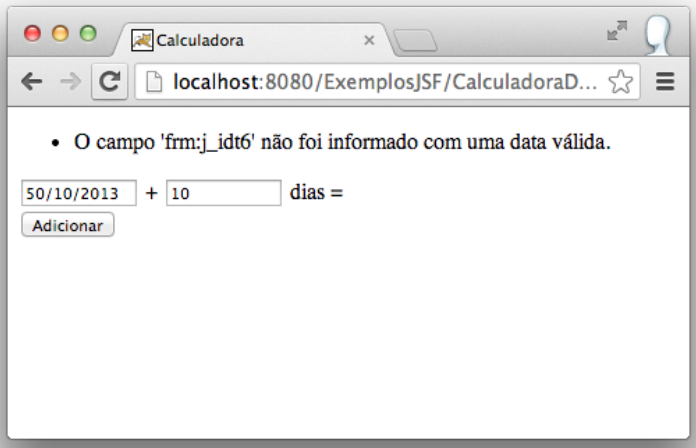
Este arquivo foi colocado dentro do pacote `com.algaworks.resources`, com o nome *Messages.properties*. Como estamos usando Maven, a convenção é que criemos o pacote e coloquemos arquivos de recursos no diretório *src/main/resources* do projeto.

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

Agora, incluiremos o seguinte código no arquivo *faces-config.xml*, para que este pacote de mensagens seja carregado pela aplicação.

```
<application>
    <message-bundle>
        com.algaworks.resources.Messages
    </message-bundle>
</application>
```

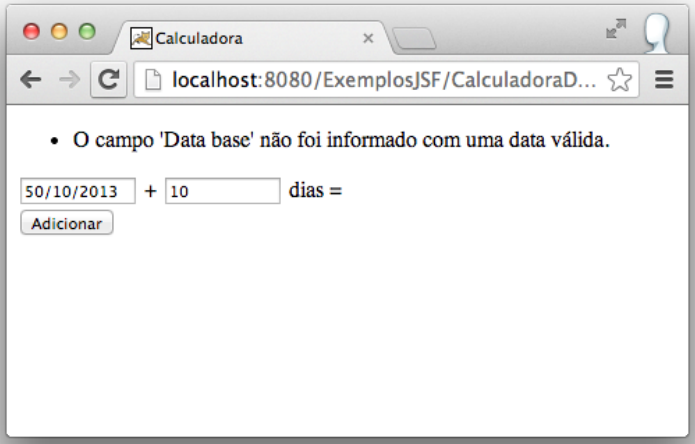
Vamos executar o exemplo de uma seção anterior, que criamos uma calculadora de data, e informar uma data incorreta.



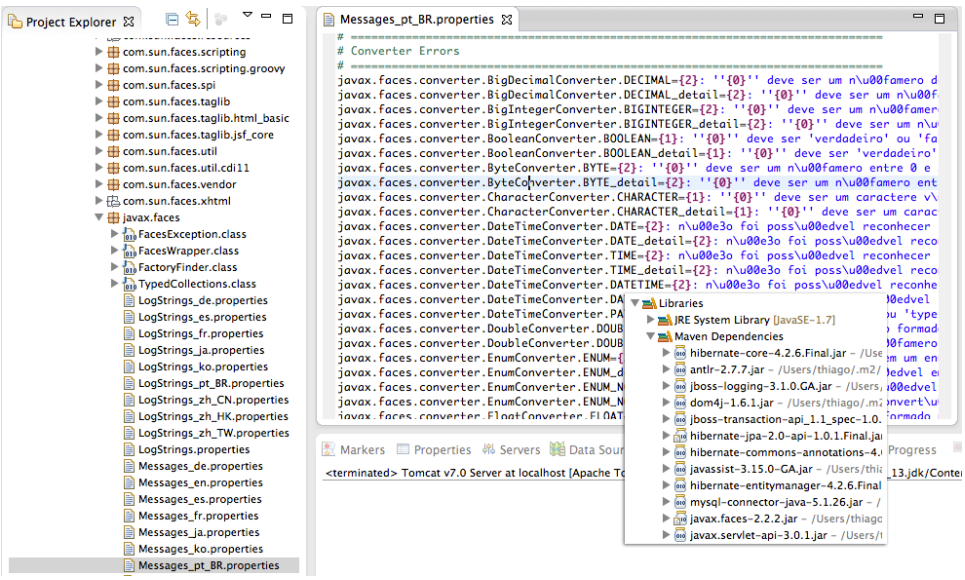
A mensagem que customizamos foi usada! Podemos melhorar a mensagem incluindo a propriedade `label` no campo.

```
<h:inputText size="12" value="#{calculadoraDataBean.dataBase}"
        label="Data base">
    <f:convertDateTime pattern="dd/MM/yyyy" />
</h:inputText>
```

Agora o valor da propriedade `label` foi usado no *placeholder* que incluímos na mensagem de detalhe.



Para customizar outras mensagens, precisamos conhecer as chaves delas. Podemos consultar a especificação de JavaServer Faces ou, ainda mais fácil, abrir o arquivo *Messages_pt_BR.properties* que fica no pacote `javax.faces` do arquivo *javax.faces.2.2.x.jar*.



A propriedade converterMessage

A customização de mensagens que fizemos é muito interessante, pois toda a aplicação utiliza as novas mensagens, mas algumas vezes podemos querer uma mensagem de erro de conversão específica para um componente da página. Para este cenário, devemos usar a propriedade `converterMessage` do campo de entrada, como no exemplo abaixo:

```
<h:inputText size="12" value="#{calculadoraDataBean.dataBase}"
        converterMessage="Informe uma data base corretamente.">
    <f:convertDateTime pattern="dd/MM/yyyy" />
</h:inputText>
```

8.5. Validadores padrão

JavaServer Faces possui um mecanismo de validação simples e flexível, e tem como objetivo proteger o modelo contra valores inválidos informados pelos usuários, evitando que fique em um estado inconsistente. O processo de validação ocorre após a conversão dos dados.

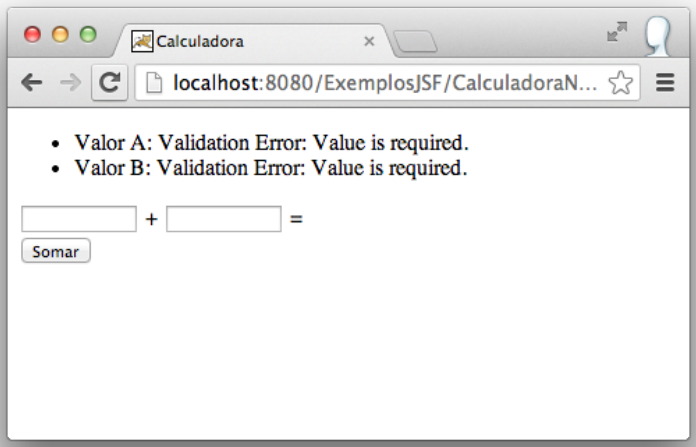
A propriedade required

O validador mais simples de ser usado é o que obriga o preenchimento de um campo. Para incluir este validador, basta adicionarmos a propriedade `required` em um componente de entrada de dados.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:head>
    <title>Calculadora</title>
</h:head>

<h:body>
    <h:form id="frm">
        <h:messages />

        <h:inputText size="12" value="#{calculadoraBean.valorA}"
            label="Valor A" required="true" />
        +
        <h:inputText size="12" value="#{calculadoraBean.valorB}"
            label="Valor B" required="true" />
```

```
        =
        <h:outputText value="#{calculadoraBean.resultado}" />
        <br />

        <h:commandButton value="Somar"
            action="#{calculadoraBean.somar}" />
    </h:form>
</h:body>

</html>
```

Agora não conseguimos mais submeter o formulário da calculadora, sem informar os valores A e B.



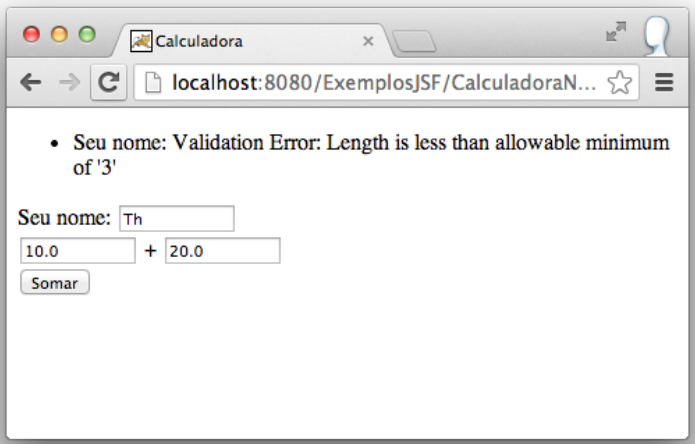
A tag <f:validateLength>

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 3 e no máximo 15 caracteres. Veja que adicionamos a tag `<f:validateLength>`.

```
<h:form id="frm">
    <h:messages />

    <h:outputLabel value="Seu nome: " for="seuNome"/>
    <h:inputText size="12" value="#{calculadoraBean.nome}"
            id="seuNome" label="Seu nome" required="true">
        <f:validateLength minimum="3" maximum="15" />
```

```
    </h:inputText>
    <br/>

    <h:inputText size="12" value="#{calculadoraBean.valorA}"
        label="Valor A" required="true" />
    +
    <h:inputText size="12" value="#{calculadoraBean.valorB}"
        label="Valor B" required="true" />
    <br/>
    <h:panelGroup rendered="#{calculadoraBean.resultado != null}">
        <h:outputText value="#{calculadoraBean.nome}, o resultado é "/>
        <h:outputText value="#{calculadoraBean.resultado}" />
        <br />
    </h:panelGroup>

    <h:commandButton value="Somar" action="#{calculadoraBean.somar}" />
</h:form>
```

Este validador não é executado se o valor do campo estiver vazio, por isso obrigamos o preenchimento colocando o atributo `required`.



Outros validadores

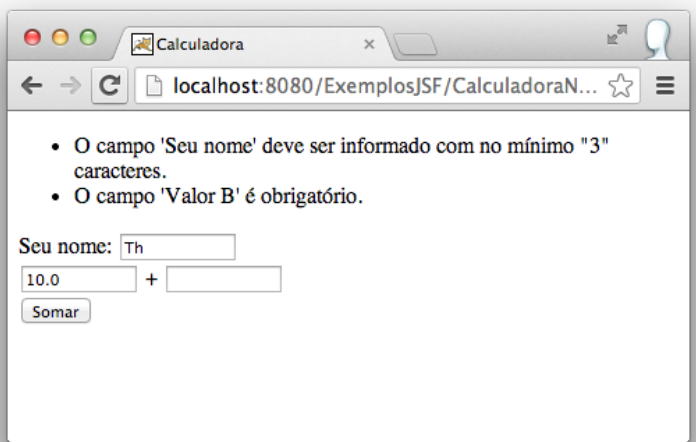
Além do validador fornecido pela tag `<f:validateLength>`, o JavaServer Faces também possui outros, como `<f:validateDoubleRange>`, `<f:validateLongRange>`, `<f:validateRegex>` e `<f:validateRequired>`.

8.6. 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 e para o validador de tamanho de strings. Editaremos o arquivo *Messages.properties* e incluiremos o seguinte conteúdo:

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

As mensagens de erro de validação que precisamos em nossa página foram customizadas.



As propriedades requiredMessage e validatorMessage

As mensagens de erro de validação que acabamos de customizar serão usadas por todas as páginas do sistema. Podemos customizar as mensagens de erro de validação para campos específicos, utilizando as propriedades `requiredMessage` e `validatorMessage`.

```
<h:inputText size="12" value="#{calculadoraBean.nome}" id="seuNome"
        label="Seu nome" required="true"
        requiredMessage="Informe o seu nome."
        validatorMessage="Informe o seu nome de 3 a 15 caracteres.">
    <f:validateLength minimum="3" maximum="15" />
</h:inputText>
```

8.7. Criando conversores personalizados

Os conversores que já veem 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, 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 `javax.faces.convert.Converter` e é anotada com `@FacesConverter`. 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.

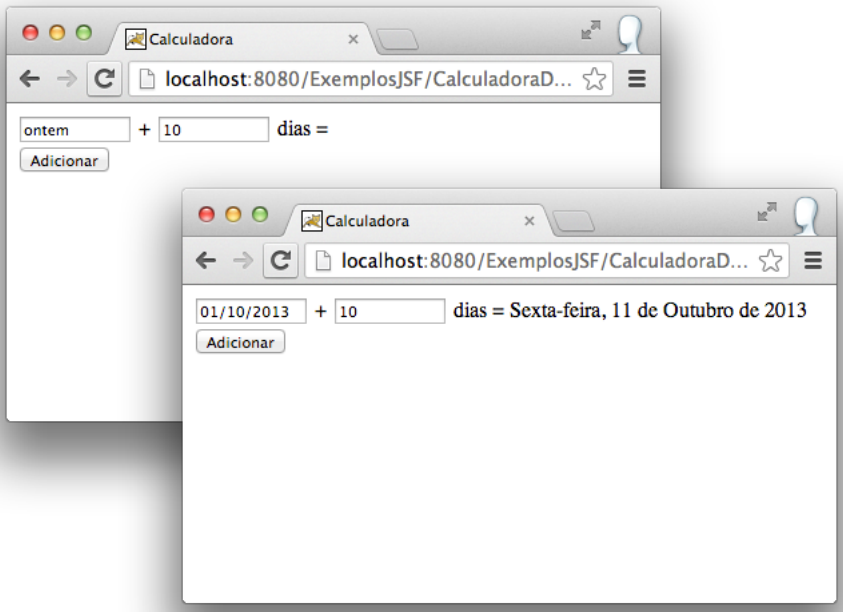
```
@FacesConverter("smartDate")
public class SmartDateConverter implements Converter {

    private static final DateFormat FORMATADOR =
        new SimpleDateFormat("dd/MM/yyyy");

    @Override
    public Object getAsObject(FacesContext context,
            UIComponent component, String value) {
        Date dataConvertida = null;

        if (value == null || value.equals("")) {
            return 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 ontem = getDataAtual();
            ontem.add(Calendar.DAY_OF_MONTH, -1);
            dataConvertida = ontem.getTime();
        } else {
            try {
                dataConvertida = FORMATADOR.parse(value);
            } catch (ParseException e) {
                throw new ConverterException(new FacesMessage(
                    FacesMessage.SEVERITY_ERROR, "Data incorreta.",
                    "Informe uma data correta."));
            }
        }

        return dataConvertida;
    }

    private Calendar getDataAtual() {
        Calendar dataAtual = new GregorianCalendar();

        // limpamos informações de hora, minuto, segundo
        // e milissegundos
        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 String getAsString(FacesContext context,
            UIComponent component, Object value) {
        return FORMATADOR.format((Date) value);
    }

}
```

Implementamos nosso conversor esperto de datas e registramos ele com o identificador *smartDate*, através da anotação `@FacesConverter`. Agora, podemos usar o conversor simplesmente informando o ID na propriedade `converter` da tag `<h:inputText>`.

```
<h:inputText size="12" value="#{calculadoraDataBean.dataBase}"
    converter="smartDate" />
```

Alteramos a página da calculadora de data, que usamos no exemplo de uma seção anterior, e incluímos a possibilidade de informar algumas palavras especiais para referenciar datas, como por exemplo "ontem".



8.8. Criando validadores personalizados

O processo para criar validadores personalizados é semelhante à criação de conversores. Criaremos um validador de datas que restringe os valores apenas em dias úteis.

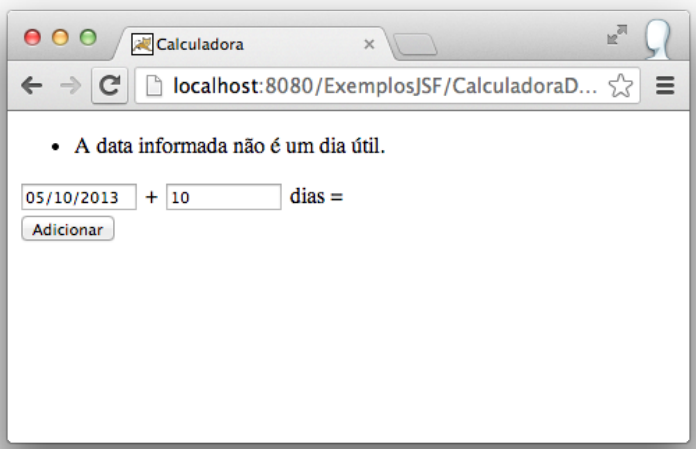
Para criar um validador customizado, precisamos implementar a interface `javax.faces.validator.Validator` e anotar a classe com `@FacesValidator`.

```
@FacesValidator("diaUtil")
public class DiaUtilValidator implements Validator {

    @Override
    public void validate(FacesContext context, UIComponent component,
            Object value) throws ValidatorException {
        Date data = (Date) value;

        Calendar calendar = Calendar.getInstance();
```

```
            calendar.setTime(data);

            int diaSemana = calendar.get(Calendar.DAY_OF_WEEK);

            if (diaSemana < Calendar.MONDAY
                    || diaSemana > Calendar.FRIDAY) {
                throw new ValidatorException(new FacesMessage(
                        FacesMessage.SEVERITY_ERROR, "Data inválida.",
                        "A data informada não é um dia útil."));
            }
        }

}
```

Registramos nosso validador com o id **diaUtil**. Agora, podemos usá-lo através da tag `<f:validator>`.

```
<h:inputText size="12" value="#{calculadoraDataBean.dataBase}">
    <f:convertDateTime pattern="dd/MM/yyyy"
        timeZone="America/Sao_Paulo" />
    <f:validator validatorId="diaUtil" />
</h:inputText>
```



Capítulo 9

Página de cadastro de lançamento

9.1. Implementando o repositório

Neste capítulo, criaremos uma página de cadastro e lançamentos. A primeira coisa que vamos fazer, é incluir um método `adicionar()` no repositório `Lancamentos`.

```
public void adicionar(Lancamento lancamento) {
    this.manager.persist(lancamento);
}
```

Depois, criaremos um novo repositório chamado `Pessoas`, com métodos para consulta de todas as pessoas e de uma única pessoa por um código específico.

```
public class Pessoas {

    private EntityManager manager;

    public Pessoas(EntityManager manager) {
        this.manager = manager;
    }

    public Pessoa porId(Long id) {
        return manager.find(Pessoa.class, id);
    }

    public List<Pessoa> todas() {
        TypedQuery<Pessoa> query = manager.createQuery(
                "from Pessoa", Pessoa.class);
        return query.getResultList();
    }

}
```

9.2. Implementando as regras de negócio

As regras de negócio de uma aplicação não devem ficar em managed beans. Precisamos criar classes que tratam apenas de processos de negócio, facilitando a manutenção e reaproveitamento da lógica do sistema.

Vamos criar um tipo de exceção chamado `NegocioException`, para representar erros de negócio.

```
public class NegocioException extends Exception {

    private static final long serialVersionUID = 1L;

    public NegocioException(String msg) {
        super(msg);
    }

}
```

Agora, criaremos nossa classe de negócio que trata de cadastro de lançamentos. Essa classe dependerá do respositório de lançamentos, por isso, ela deve receber o objeto do repositório no construtor.

```
public class CadastroLancamentos {

    private Lancamentos lancamentos;

    public CadastroLancamentos(Lancamentos lancamentos) {
        this.lancamentos = lancamentos;
    }

    public void salvar(Lancamento lancamento) throws NegocioException {
        if (lancamento.getDataPagamento() != null &&
                lancamento.getDataPagamento().after(new Date())) {
            throw new NegocioException(
                    "Data de pagamento não pode ser uma data futura.");
        }

        this.lancamentos.adicionar(lancamento);
    }

}
```

Da forma que implementamos, podemos usar a classe `CadastroLancamentos` em um sistema desktop ou web, inclusive usando qualquer framework MVC.

9.3. Programando o managed bean de cadastro

Vamos criar o managed bean que será usado pela página de cadastro de lançamentos. O método `salvar()` deve iniciar uma transação, instanciar um objeto do tipo `CadastroLancamentos` e salvar o lançamento. Caso ocorra algum erro, deve-se fazer *rollback* e apresentar uma mensagem de erro.

Criamos também um método chamado `prepararCadastro()`, que carrega uma lista com todas as pessoas cadastradas. Usaremos essa lista para preencher o menu de pessoas na página.

```
@ManagedBean
@ViewScoped
public class CadastroLancamentoBean implements Serializable {

    private static final long serialVersionUID = 1L;

    private Lancamento lancamento = new Lancamento();
    private List<Pessoa> todasPessoas;

    public void prepararCadastro() {
        EntityManager manager = JpaUtil.getEntityManager();
        try {
            Pessoas pessoas = new Pessoas(manager);
            this.todasPessoas = pessoas.todas();
        } finally {
            manager.close();
        }
    }

    public void salvar() {
        EntityManager manager = JpaUtil.getEntityManager();
        EntityTransaction trx = manager.getTransaction();
        FacesContext context = FacesContext.getCurrentInstance();

        try {
            trx.begin();
            CadastroLancamentos cadastro = new CadastroLancamentos(
                new Lancamentos(manager));
            cadastro.salvar(this.lancamento);

            this.lancamento = new Lancamento();
            context.addMessage(null, new FacesMessage(
                "Lançamento salvo com sucesso!"));

            trx.commit();
        } catch (NegocioException e) {
            trx.rollback();

            FacesMessage mensagem = new FacesMessage(e.getMessage());
            mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
```

```
            context.addMessage(null, mensagem);
        } finally {
            manager.close();
        }
    }

    public List<Pessoa> getTodasPessoas() {
        return this.todasPessoas;
    }

    public TipoLancamento[] getTiposLancamentos() {
        return TipoLancamento.values();
    }

    public Lancamento getLancamento() {
        return lancamento;
    }

    public void setLancamento(Lancamento lancamento) {
        this.lancamento = lancamento;
    }

}
```

9.4. Programando o conversor de Pessoa

Criamos um conversor que é capaz de converter uma string com o código da pessoa em objeto do tipo `Pessoa`, e vice-versa.

Registramos o conversor e configuramos ele como padrão para o tipo `Pessoa`, usando o atributo `forClass` da anotação `@FacesConverter`.

```
@FacesConverter(forClass = Pessoa.class)
public class PessoaConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext context,
            UIComponent component, String value) {
        Pessoa retorno = null;
        EntityManager manager = JpaUtil.getEntityManager();

        try {
            if (value != null) {
                Pessoas pessoas = new Pessoas(manager);
                retorno = pessoas.porId(new Long(value));
            }

            return retorno;
        } finally {
            manager.close();
```

```
        }
    }

    @Override
    public String getAsString(FacesContext context,
            UIComponent component, Object value) {
        if (value != null) {
            return ((Pessoa) value).getId().toString();
        }
        return null;
    }

}
```

9.5. Criando o formulário de cadastro

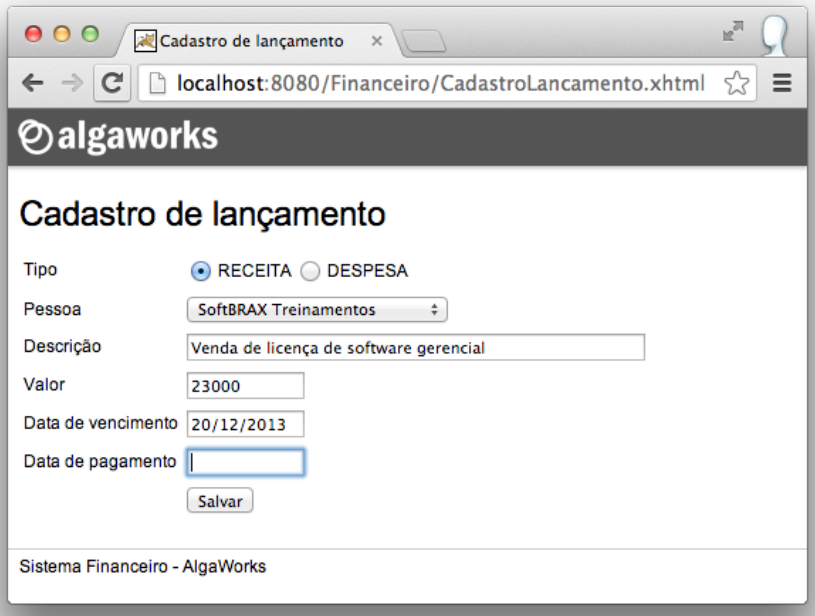
A página de cadastro usa o que já falamos em capítulos anteriores. Não incluímos validações nos campos de entrada, por isso, por enquanto, é importante preencher corretamente pelo menos os campos obrigatórios.

```
<!DOCTYPE html>
<ui:composition template="/WEB-INF/template/Layout.xhtml"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

<f:metadata>
    <f:viewAction action="#{cadastroLancamentoBean.prepararCadastro}" />
</f:metadata>

<ui:define name="titulo">Cadastro de lançamento</ui:define>

<ui:define name="corpo">
    <h1>Cadastro de lançamento</h1>

    <h:form id="frm">
        <h:messages showDetail="false" showSummary="true" />

        <h:panelGrid columns="2">
            <h:outputLabel value="Tipo" />
            <h:selectOneRadio
                    value="#{cadastroLancamentoBean.lancamento.tipo}"
                    label="Tipo do lançamento">
                <f:selectItems
                    value="#{cadastroLancamentoBean.tiposLancamentos}"
                    var="tipoLancamento"
                    itemValue="#{tipoLancamento}" />
            </h:selectOneRadio>

            <h:outputLabel value="Pessoa" />
```

```
            <h:selectOneMenu
                    value="#{cadastroLancamentoBean.lancamento.pessoa}"
                    label="Pessoa">
                <f:selectItem
                    itemLabel="Selecione" noSelectionOption="true" />
                <f:selectItems
                    value="#{cadastroLancamentoBean.todasPessoas}"
                    var="pessoa" itemValue="#{pessoa}"
                    itemLabel="#{pessoa.nome}" />
            </h:selectOneMenu>

            <h:outputLabel value="Descrição" />
            <h:inputText size="60"
                value="#{cadastroLancamentoBean.lancamento.descricao}"
                label="Descrição" />

            <h:outputLabel value="Valor" />
            <h:inputText size="12"
                    value="#{cadastroLancamentoBean.lancamento.valor}"
                    label="Valor">
                <f:convertNumber locale="pt_BR" maxFractionDigits="2"
                    minFractionDigits="2" />
            </h:inputText>

            <h:outputLabel value="Data de vencimento" />
            <h:inputText size="12"
                    value="#{cadastroLancamentoBean.lancamento
                        .dataVencimento}"
                    label="Data de vencimento">
                <f:convertDateTime pattern="dd/MM/yyyy"
                    timeZone="America/Sao_Paulo" />
            </h:inputText>

            <h:outputLabel value="Data de pagamento" />
            <h:inputText size="12"
                    value="#{cadastroLancamentoBean.lancamento
                        .dataPagamento}"
                    label="Data de pagamento">
                <f:convertDateTime pattern="dd/MM/yyyy"
                    timeZone="America/Sao_Paulo" />
            </h:inputText>

            <h:outputLabel />
            <h:commandButton value="Salvar"
                action="#{cadastroLancamentoBean.salvar}" />
        </h:panelGrid>
    </h:form>
</ui:define>

</ui:composition>
```

Usamos a tag `<f:viewAction>` em `<f:metadata>` para chamar o método `prepararCadastro()` do managed bean na fase de invocação da aplicação.

Vejamos o resultado de nossa página recém criada:



Capítulo 10

Bean Validation

10.1. O que é Bean Validation?

A API de Bean Validation fornece uma facilidade para validar objetos em diferentes camadas da aplicação. JavaServer Faces integra com esta tecnologia para validar objetos preenchidos pelas páginas que criamos.

A vantagem de usar Bean Validation é que as restrições ficam inseridas nas classes de modelo, e não em páginas XHTML, por isso, podem ser usadas por outras camadas da aplicação.

As restrições de Bean Validation são em forma de anotações, que podem ser usadas, por exemplo, em entidades ou classes de managed beans.

Diversas anotações de restrições estão disponíveis no pacote `javax.validation.constraints`. Vejamos um exemplo de uma classe com atributos anotados com restrições Bean Validation.

```
public class Usuario {

    @NotNull
    @Size(min = 5, max = 20)
    private String nome;

    @NotNull
    @Size(min = 5, max = 40)
    private String sobrenome;

}
```

Bean Validation é uma especificação, e não um produto, por isso, precisamos de uma implementação para tudo funcionar. As implementações podem adicionar restrições customizadas, além das fornecidas pela especificação. Usaremos o Hibernate Validator, que implementa Bean Validation.

10.2. Adicionando o artefato no pom.xml

O Hibernate Validator pode ser baixado em http://www.hibernate.org/subprojects/validator.html, mas como estamos usando Maven, podemos facilmente adicionar o artefato no arquivo *pom.xml* de nosso projeto.

```
<!-- Implementacao do Bean Validation -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.0.1.Final</version>
    <scope>compile</scope>
</dependency>
```

Neste capítulo, implementaremos os exemplos no projeto do sistema financeiro.

10.3. Adicionando restrições no modelo

Incluiremos algumas restrições em nosso modelo, nas classes `Lancamento` e `Pessoa`.

```
@Entity
@Table(name = "lancamento")
public class Lancamento {

    private Long id;
    private Pessoa pessoa;
    private String descricao;
    private BigDecimal valor;
    private TipoLancamento tipo;
    private Date dataVencimento;
    private Date dataPagamento;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

    @NotNull
    @ManyToOne(optional = false)
    @JoinColumn(name = "pessoa_id")
```

```
    public Pessoa getPessoa() {
        return pessoa;
    }

    @NotEmpty
    @Size(max = 80)
    @Column(length = 80, nullable = false)
    public String getDescricao() {
        return descricao;
    }

    @NotNull
    @DecimalMin("0")
    @Column(precision = 10, scale = 2, nullable = false)
    public BigDecimal getValor() {
        return valor;
    }

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    public TipoLancamento getTipo() {
        return tipo;
    }

    @NotNull
    @Temporal(TemporalType.DATE)
    @Column(name = "data_vencimento", nullable = false)
    public Date getDataVencimento() {
        return dataVencimento;
    }

    @Temporal(TemporalType.DATE)
    @Column(name = "data_pagamento", nullable = true)
    public Date getDataPagamento() {
        return dataPagamento;
    }

    // setters, hashCode e equals

}

@Entity
@Table(name = "pessoa")
public class Pessoa {

    private Long id;
    private String nome;

    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }
```

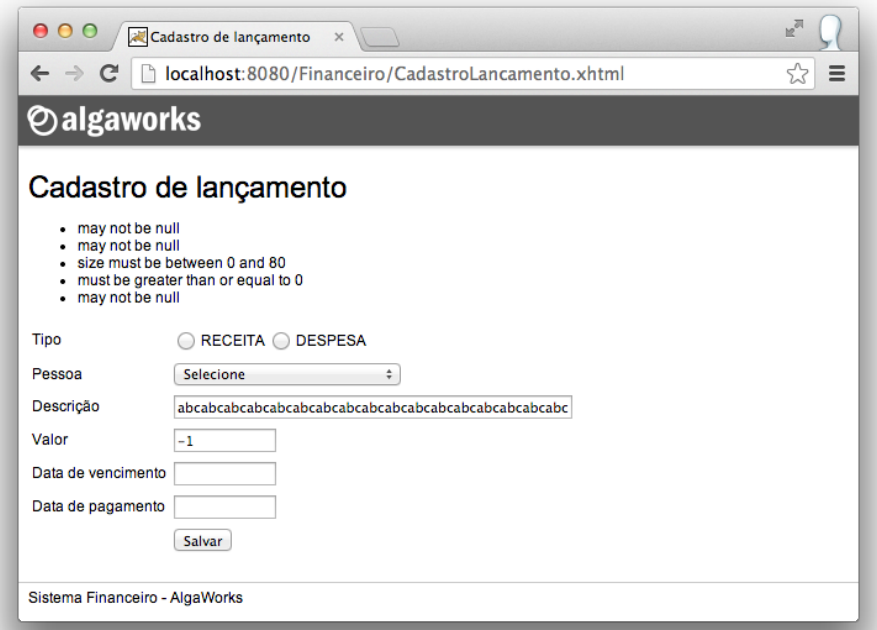
```
    @NotEmpty
    @Size(max = 60)
    @Column(length = 60, nullable = false)
    public String getNome() {
        return nome;
    }

    // setters, hashCode e equals

}
```

Vejamos o significado das anotações que usamos para adicionar restrições.

Restrição	Descrição
@NotNull	O valor da propriedade não pode ser `null`.
@NotEmpty	Valida que uma string, coleção, mapa ou array não é nulo e nem vazio. Esta restrição é específica do Hibernate Validator.
@Size	O tamanho do valor da propriedade deve estar entre os limites configurados. Funciona com string, coleção, mapa e array.
@DecimalMin	O valor da propriedade deve ser um número decimal maior ou igual ao número especificado.

Existem diversas outras anotações Bean Validation e também do Hibernate Validator. Você pode encontrá-las nos pacotes `javax.validation.constraints` e `org.hibernate.validator.constraints`.

Como um passe de mágica, podemos acessar a página de cadastro de lançamentos e tentar submeter o formulário com dados inválidos ou insuficientes, e o JSF irá usar a integração com Bean Validation para validar os valores.



10.4. Customizando mensagens de validação

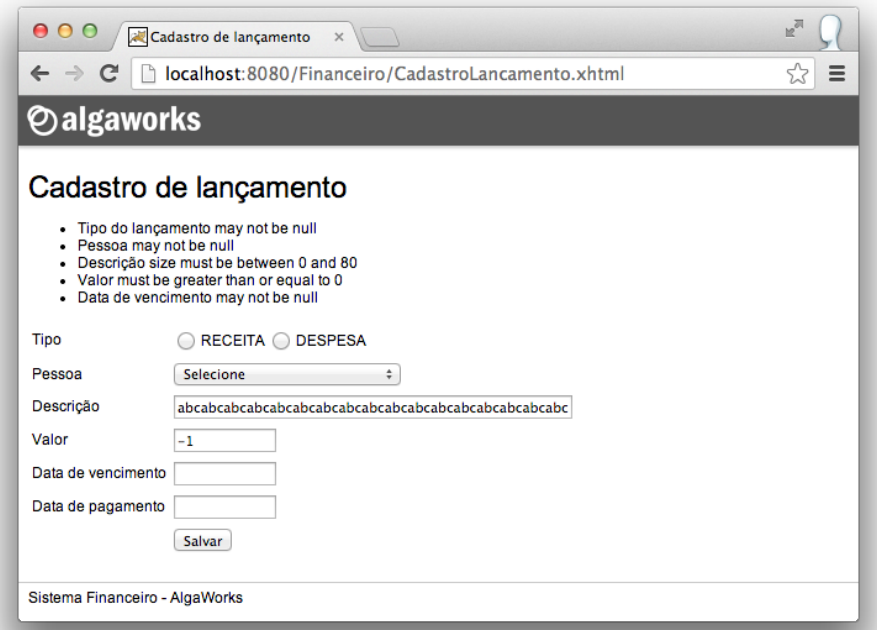
Por padrão, o JavaServer Faces não adiciona o rótulo do campo nas mensagens de erro geradas através das restrições Bean Validation, mas podemos alterar esse formato, criando um arquivo *Messages.properties* no pacote `com.algaworks.financeiro.resources` de nosso projeto, com o seguinte conteúdo:

```
javax.faces.validator.BeanValidator.MESSAGE={1} {0}
```

Incluímos dois *placeholders* no formato da mensagem usada para restrições Bean Validation. O *placeholder* `{1}` representa o rótulo do campo e `{0}` a mensagem.

Depois disso, precisamos incluir o seguinte código no arquivo *faces-config.xml*, para que este pacote de mensagens seja carregado pela aplicação.

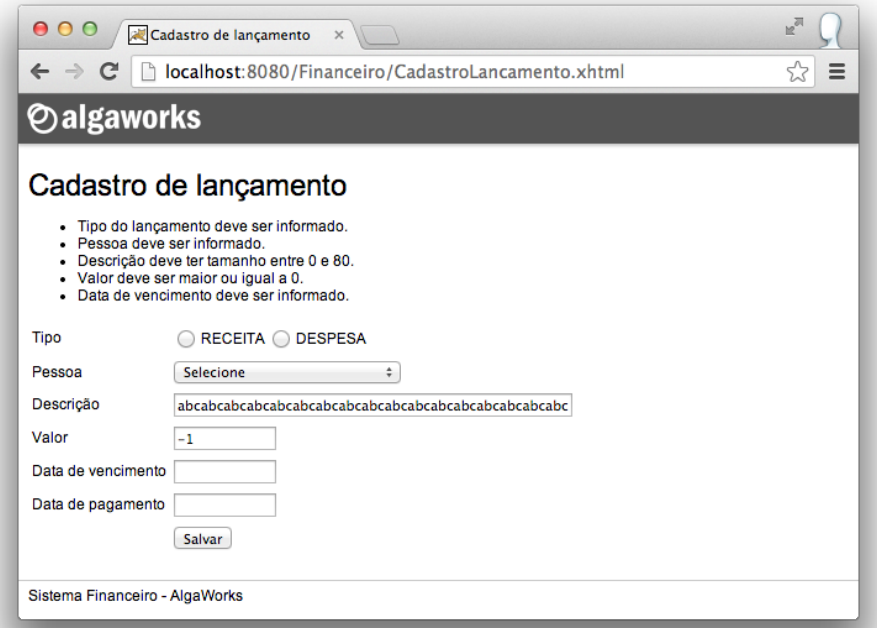
```
<application>
    <message-bundle>
        com.algaworks.financeiro.resources.Messages
    </message-bundle>
</application>
```



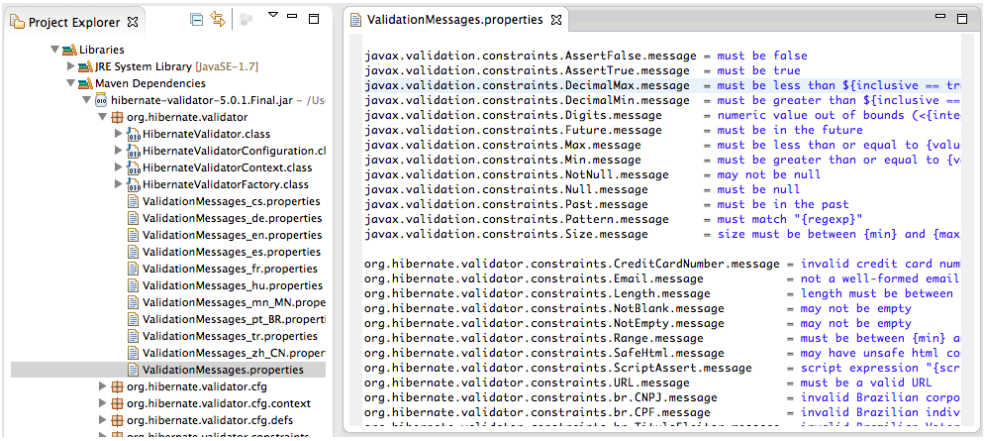
Conseguimos melhorar a identificação dos campos que estão com erro, mas podemos deixar ainda melhor, customizando as mensagens das restrições. Para isso, basta criar um arquivo *ValidationMessages.properties* na raíz do diretório *src/main/resources* do projeto, com o conteúdo a seguir:

```
javax.validation.constraints.NotNull.message = deve ser informado.
javax.validation.constraints.Size.message = deve ter tamanho entre \
                {min} e {max}.
javax.validation.constraints.DecimalMin.message = deve ser maior ou \
                igual a {value}.

org.hibernate.validator.constraints.NotEmpty.message = deve ser \
                informado.
```

Customizamos apenas as mensagens das restrições que estamos usando.



Para conhecer as outras chaves de mensagens, consulte o arquivo *ValidationMessages.properties* do arquivo *hibernate-validator-5.0.x.Fina.jar*, referenciado pelo projeto.



10.5. Compondo uma nova restrição

Podemos criar restrições Bean Validation customizadas a partir das anotações existentes. Por exemplo, imagine que precisaremos adicionar as restrições `@NotNull` e `@DecimalMin` em diversas propriedades em nosso sistema. Teríamos que repetir o mesmo código em diversos lugares no código-fonte.

```
@NotNull
@DecimalMin("0")
@Column(precision = 10, scale = 2, nullable = false)
public BigDecimal getValor() {
    return valor;
}
```

Podemos facilitar um pouco, criando uma única restrição chamada `@DecimalPositivo`, que é simplesmente uma composição das restrições `@NotNull` e `@DecimalMin` juntas.

```
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@NotNull
@DecimalMin("0")
public @interface DecimalPositivo {

    @OverridesAttribute(constraint = DecimalMin.class, name = "message")
    String message() default "{com.algaworks.NumeroDecimal.message}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

Definimos, por padrão, que a mensagem da restrição `@DecimalMin` será obtida da chave *com.algaworks.NumeroDecimal.message* do arquivo *ValidationMessages.properties*, por isso, precisamos adicionar uma nova linha nele.

```
com.algaworks.NumeroDecimal.message = deve ser um número decimal \
                positivo.
```

Agora, sempre que precisarmos validar um número decimal positivo, usaremos a restrição `@DecimalPositivo`.

```
@DecimalPositivo
@Column(precision = 10, scale = 2, nullable = false)
public BigDecimal getValor() {
```

```
    return valor;
}
```

Capítulo 11

Manipulando eventos

11.1. 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 de JavaServer Faces, 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*). Dessa forma, os eventos devem ser entregues ao servidor através de requisições HTTP para que ele processe a lógica de negócio referente ao evento.

JavaServer Faces possui mecanismos para tratar diferentes tipos de eventos. Neste capítulo, vamos estudar sobre eventos de ação e eventos de mudança de valor do usuário.

11.2. Eventos de ação

Os eventos de ação são iniciados por componentes de comando, como `<h:commandButton>` e `<h:commandLink>`, quando os mesmos são disparados 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 não contribui para a navegação das páginas, porém possui informações sobre o evento, já o segundo tipo contribui para a navegação das páginas e não possui

informações sobre o evento (como o componente que gerou a ação). Os dois tipos podem trabalhar em conjunto em um mesmo componente.

Para criar um *listener* de ação, precisamos criar um método em um managed bean que recebe um objeto do tipo `javax.faces.event.ActionEvent`.

Um método de ação, como já vimos em outros capítulos, contribui para a navegação das páginas. Este método deve retornar uma string ou `void`.

```
public void registrarLogCadastro(ActionEvent event) {
    System.out.println("Cadastrando...");
}

public String cadastrar() {
    return "Consulta";
}
```

Vinculamos o *listener* de ação e a ação através das propriedades `actionListener` e `action`.

```
<h:commandButton value="Cadastrar" action="#{bean.cadastrar}"
    actionListener="#{bean.registrarLogCadastro}"/>
```

Ao clicar no botão, o método `registrarLogCadastro()` é executado antes do método `cadastrar()`.

Use *listener* de ação se você quer executar algum código antes da lógica real do botão, como por exemplo, gerar um log, preparar algum objeto que o método de ação irá usar ou se precisar ter acesso ao componente de origem da ação.

Use `action` para executar a lógica real da ação e, se necessário, tratar a navegação entre páginas.

11.3. Eventos de mudança de valor e propriedade immediate

Os eventos de mudança de valor são invocados quando valores de componentes de entrada são modificados e o formulário é submetido ao servidor. Os métodos chamados quando eventos de mudança de valor ocorrem devem receber um parâmetro do tipo `ValueChangeEvent`.

Criaremos um método que receberá eventos de mudança de valor no managed bean `CadastroLancamentoBean`, do projeto financeiro.

```
public void descricaoModificada(ValueChangeEvent event) {
    System.out.println("Valor antigo: " + event.getOldValue());
    System.out.println("Novo valor: " + event.getNewValue());
}
```

Podemos obter o valor anterior e o novo valor a partir do objeto do tipo `ValueChangeEvent`.

Os métodos de mudança de valor são chamados no final da fase **Processar validações**, antes de atribuir os valores convertidos e validados ao modelo.

Na página *CadastroLancamento.xhtml*, alteraremos o campo de descrição para chamar o método que criamos, usando a propriedade `valueChangeListener`.

```
<h:inputText size="60" label="Descrição"
    value="#{cadastroLancamentoBean.lancamento.descricao}"
    valueChangeListener="#{cadastroLancamentoBean
        .descricaoModificada}" />
```

Se acessarmos a página, informarmos apenas uma descrição com valor "Teste" e submetermos o formulário, não conseguiremos cadastrar um lançamento, mas o método `descricaoModificada()` será chamado, e imprimirá na console:

```
Valor antigo: null
Novo valor: Teste
```

O método `descricaoModificada()` não está fazendo nada de útil, mas a ideia é mostrar como os eventos de mudança de valor funcionam.

Durante a execução de um método de evento de mudança de valor, podemos dizer ao contexto do JSF que queremos que a página seja renderizada, pulando todas as outras fases do ciclo de vida.

```
public void descricaoModificada(ValueChangeEvent event) {
    System.out.println("Valor antigo: " + event.getOldValue());
    System.out.println("Novo valor: " + event.getNewValue());
    FacesContext.getCurrentInstance().renderResponse();
}
```

Se acessarmos a página de cadastro de lançamento, preenchermos os campos e submetermos o formulário, nada irá acontecer! Se não alterarmos nada e submetermos o formulário novamente, conseguimos cadastrar o lançamento. Isso acontece porque, na primeira vez, o método `descricaoModificada()` é chamado, mas nós desviamos a execução para a fase de renderização de resposta. Se nada for alterado no formulário, o método não é invocado novamente, pois como o próprio nome diz, é um evento de mudança de valor.

Propriedade immediate

Componentes de entrada e de comando possuem uma propriedade chamada `immediate`, que quando atribuída para `true`, faz com que as validações, conversões e eventos desse componente sejam executados durante a fase **Aplicar valores de requisição**.

Vamos incluir essa propriedade no campo de descrição, e veremos o resultado.

```
<h:inputText size="60" label="Descrição"
    value="#{cadastroLancamentoBean.lancamento.descricao}"
    valueChangeListener="#{cadastroLancamentoBean
        .descricaoModificada}"
    immediate="true" />
```

Agora, se acessarmos a página de cadastro de lançamento, preenchermos apenas a descrição e submetermos o formulário, nada acontecerá, nem mesmo as mensagens de validação dos outros campos serão exibidas (e nem executadas), pois continuamos chamando o método `renderResponse()` no método do evento, porém em uma fase que ocorre antes das validações!

Eventos de mudança de valor podem ser usados, por exemplo, para implementar caixas de seleção dependentes ou componente cuja visualização depende do estado de um outro componente, que pode ser alterado a qualquer momento pelo usuário. Com os recursos de Ajax, que falaremos em outro capítulo, podemos fazer isso de uma forma mais fácil.

Capítulo 12

CDI - Contexts and Dependency Injection

12.1. Injeção de dependências

Injeção de dependências (*dependency injection* ou DI) é um padrão de desenvolvimento de software usado para manter o baixo acoplamento entre classes do sistema. As dependências de um objeto não são instanciadas programaticamente, mas sim injetadas de alguma forma.

Vamos supor que temos uma classe `EmissorNotaFiscal`, que deve registrar um *log* sempre que uma nova nota fiscal for emitida.

```
public class EmissorNotaFiscal {

    private Logging logging;

    public EmissorNotaFiscal() {
        this.logging = new ConsoleLogging();
    }

    public void emitir(NotaFiscal nf) {
        // emite nota fiscal...

        this.logging.registrar("Nota fiscal emitida: "
            + nf.getNumero());
    }

}
```

`Logging` é uma interface.

```
public interface Logging {

    public void registrar(String mensagem);

}
```

`ConsoleLogging` é uma classe que implementa a interface `Logging`.

```
public class ConsoleLogging implements Logging {

    @Override
    public void registrar(String mensagem) {
        System.out.println(mensagem);
    }

}
```

A estrutura está quase perfeita, se não fosse pela instanciação de `ConsoleLogging` no construtor de `EmissorNotaFiscal`.

```
public class EmissorNotaFiscal {

    private Logging logging;

    public EmissorNotaFiscal() {
        this.logging = new ConsoleLogging();
    }

    ...

}
```

Essa instanciação gera um acoplamento desnecessário entre a classe `EmissorNotaFiscal` e `ConsoleLogging`.

A classe `EmissorNotaFiscal` não precisa saber da existência de `ConsoleLogging`. Tudo que ela precisa conhecer é a interface. Por isso, receberemos uma instância do tipo da interface pelo construtor.

```
public class EmissorNotaFiscal {

    private Logging logging;

    public EmissorNotaFiscal(Logging logging) {
        this.logging = this.logging;
    }

    ...

}
```

Deixamos a classe `EmissorNotaFiscal` muito mais interessante. Podemos usá-la com qualquer tipo de mecanismo de *logging*. Por exemplo, poderíamos criar uma nova classe `ArquivoLogging`, que implementa a interface `Logging`, e passar uma instância no construtor de `EmissorNotaFiscal`, e não precisaríamos mudar nenhuma linha dessa classe. Isso indica que conseguimos um baixo acomplamento!

Mas quem irá instanciar um tipo de `Logging` e um objeto `EmissorNotaFiscal`? É aí que entra os mecanismos de injeção de dependências!

CDI (*Contexts and Dependency Injection*) é a especificação da Java EE que trabalha com injeção de dependências. Podemos usar CDI para instanciar e injetar objetos de nossa aplicação.

12.2. Configurando CDI no projeto

Para configurar CDI em um projeto, primeiro precisamos de uma implementação, pois CDI é apenas uma especificação. Usaremos Weld, que pode ser baixado em http://weld.cdi-spec.org/. Como estamos usando Maven, apenas adicionaremos a dependência no arquivo *pom.xml* do projeto.

```
<!-- Weld (implementação do CDI) -->
<dependency>
    <groupId>org.jboss.weld.servlet</groupId>
    <artifactId>weld-servlet</artifactId>
    <version>2.0.4.Final</version>
    <scope>compile</scope>
</dependency>
```

Precisamos adicionar um arquivo chamado *context.xml* no diretório *src/main/webapp/META-INF* do projeto.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
    <Manager pathname=""/>
    <Resource name="BeanManager" auth="Container"
        type="javax.enterprise.inject.spi.BeanManager"
        factory="org.jboss.weld.resources.ManagerObjectFactory"/>
</Context>
```

No arquivo *web.xml*, adicionamos o código abaixo.

```
<listener>
    <listener-class>
        org.jboss.weld.environment.servlet.Listener
    </listener-class>
```

```
</listener>

<resource-env-ref>
    <resource-env-ref-name>BeanManager</resource-env-ref-name>
    <resource-env-ref-type>
        javax.enterprise.inject.spi.BeanManager
    </resource-env-ref-type>
</resource-env-ref>
```

No diretório *src/main/resources/META-INF*, criamos um arquivo vazio chamado *beans.xml*. A existência desse arquivo habilita CDI no projeto.

12.3. Beans CDI, EL Names e @Inject

Em um projeto que usa CDI, **quase** todas as classes são consideradas como beans CDI, também conhecidas como *CDI managed beans*. Beans CDI podem ser injetados em outros beans.

O projeto do sistema financeiro já foi configurado com CDI, agora, vamos apenas adequar algumas classes.

Repositórios

`Lancamentos` é um bean CDI! Veja que não foi necessário dizer isso através de anotações ou arquivos de configuração. Apenas pelo fato de uma classe pública ter um construtor sem argumentos, ou ter um construtor com argumentos injetados, faz com que ela seja um bean CDI.

Injetaremos `EntityManager` através do construtor da classe `Lancamentos`, anotando com `@Inject`. O problema é que `EntityManager` não é um bean CDI, mas vamos ignorar isso por enquanto.

```
public class Lancamentos {

    private EntityManager manager;

    @Inject
    public Lancamentos(EntityManager manager) {
        this.manager = manager;
    }

    ...

    public void adicionar(Lancamento lancamento) {
```

```
        EntityTransaction trx = this.manager.getTransaction();
        trx.begin();
        this.manager.persist(lancamento);
        trx.commit();
    }

}
```

Provisoriamente, iniciamos uma transação no método `adicionar()`. Este não é o melhor lugar para controlar transações, mas resolveremos isso em breve.

Precisamos fazer a mesma coisa no construtor de `Pessoas`.

Classe de negócio

Na classe `CadastroLancamentos`, removeremos o construtor e injetaremos o repositório diretamente no atributo. Veja que esta é uma outra alternativa de injeção, mesmo o atributo sendo privado.

```
public class CadastroLancamentos {

    @Inject
    private Lancamentos lancamentos;

    ...

}
```

Beans CadastroLancamentoBean e ConsultaLancamentosBean

Managed beans CDI não podem ser injetados em managed beans JSF, por isso, precisamos usar o bean `CadastroLancamentoBean` como um bean CDI.

A anotação `@Named` torna possível o acesso ao bean CDI por *Expression Language*, através de seu nome.

Beans CDI também possuem escopos quando integrados com JSF, inclusive o escopo *view*. Na API de JSF, existem duas anotações `@ViewScoped`. Para funcionar com CDI, a anotação correta é do pacote `javax.faces.view`.

Injetaremos objetos do tipo `CadastroLancamentos` e `Pessoas` no bean `CadastroLancamentoBean`. Veja como o código fica mais simples.

```
@Named
@javax.faces.view.ViewScoped
public class CadastroLancamentoBean implements Serializable {

    private static final long serialVersionUID = 1L;

    @Inject
    private CadastroLancamentos cadastro;

    @Inject
    private Pessoas pessoas;

    private Lancamento lancamento = new Lancamento();
    private List<Pessoa> todasPessoas;

    public void prepararCadastro() {
        this.todasPessoas = this.pessoas.todas();
    }

    public void salvar() {
        FacesContext context = FacesContext.getCurrentInstance();

        try {
            this.cadastro.salvar(this.lancamento);

            this.lancamento = new Lancamento();
            context.addMessage(null, new FacesMessage(
                "Lançamento salvo com sucesso!"));
        } catch (NegocioException e) {

            FacesMessage mensagem = new FacesMessage(e.getMessage());
            mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
            context.addMessage(null, mensagem);
        }
    }

    public List<Pessoa> getTodasPessoas() {
        return this.todasPessoas;
    }

    ...

}
```

Vamos fazer a mesma coisa com o bean `ConsultaLancamentosBean`.

```
@Named
@ViewScoped
public class ConsultaLancamentosBean implements Serializable {

    private static final long serialVersionUID = 1L;

    @Inject
    private Lancamentos lancamentosRepository;
```

```
    private List<Lancamento> lancamentos;

    public void consultar() {
        this.lancamentos = lancamentosRepository.todos();
    }

    ...

}
```

12.4. Escopos de beans CDI

Os beans CDI possuem escopos, que podem ser:

Escopo	Duração
@RequestScoped	Interação com usuário em uma única requisição HTTP.
@SessionScoped	Interação com usuário entre muitas requisições HTTP, ou seja, a sessão do usuário.
@ApplicationScoped	Estado compartilhado com todos os usuários durante toda a execução da aplicação.
@Dependent	É o escopo padrão, se nenhum for especificado. Mantém o mesmo ciclo de vida do bean que o injetou.
@ConversationScoped	Interação com usuário entre muitas requisições HTTP, com o início e término controlado pelo programador.

Todas essas anotações devem ser importadas do pacote `javax.enterprise.context`. As 3 primeiras, possuem os mesmos nomes de anotações do pacote `javax.faces.bean`. Quando usamos CDI, temos que tomar cuidado para não importar do pacote errado!

A anotação `@javax.faces.view.ViewScoped`, apesar de funcionar com CDI, é da API de JavaServer Faces.

12.5. Produtor de EntityManager

`EntityManager` não é um bean CDI, por isso, não conseguimos injetá-lo automaticamente. Se iniciarmos nossa aplicação da forma que está, uma exceção será lançada.

```
org.jboss.weld.exceptions.DeploymentException: WELD-001408 Unsatisfied
dependencies for type EntityManager with qualifiers @Default
  at injection point [BackedAnnotatedParameter] Parameter 1 of
  [BackedAnnotatedConstructor] @Inject public com.algaworks
  .financeiro.repository.Pessoas(EntityManager)
  at com.algaworks.financeiro.repository.Pessoas.(Pessoas.java:0)
```

Para resolver esse problema, criaremos um método produtor (*producer method*) de `EntityManager`. Um método produtor gera um objeto que pode ser injetado. Normalmente, usamos métodos produtores quando queremos injetar um objeto que não é um bean CDI, quando o tipo concreto do objeto a ser injetado pode variar em tempo de execução ou quando a instanciação do objeto requer algum procedimento adicional.

```
package com.algaworks.financeiro.util;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.context.RequestScoped;

// outros imports...

@ApplicationScoped
public class EntityManagerProducer {

    private EntityManagerFactory factory;

    public EntityManagerProducer() {
        this.factory = Persistence.createEntityManagerFactory(
            "FinanceiroPU");
    }

    @Produces
    @RequestScoped
    public EntityManager createEntityManager() {
        return factory.createEntityManager();
    }

    public void closeEntityManager(@Disposes EntityManager manager) {
        manager.close();
    }

}
```

A classe `EntityManagerProducer` possui um método produtor chamado `createEntityManager`. O objeto produzido por esse método terá o escopo de requisição, pois o método foi anotado com `@RequestScoped`.

Criamos também um método de evacuação (*disposer method*) de `EntityManager`, chamado de `closeEntityManager`. Este método será chamado automaticamente

quando o contexto onde o objeto produzido estiver for encerrado. Aproveitamos este evento para fechar o `EntityManager`.

Podemos acessar as páginas do sistema financeiro normalmente, e tudo deve funcionar!

12.6. Controlando as transações com interceptadores

Um interceptador é uma classe usada para intervir em chamadas de métodos de classes. Podemos usar interceptadores para várias coisas, como registrar logs ou executar tarefas repetitivas e que não fazem parte da regra de negócio do sistema.

Criaremos um interceptador para controlar as transações de nossa aplicação. Antes de começar, vamos voltar o código do método `adicionar()` do repositório de lançamentos para a versão original, que não gerenciava transações manualmente.

```
public class Lancamentos {

    ...

    public void adicionar(Lancamento lancamento) {
        this.manager.persist(lancamento);
    }

}
```

Antes de criar um interceptador, precisaremos de uma anotação que associará o interceptador ao método ou classe que será interceptada. Esta anotação deve ser anotada com `@InterceptorBinding`.

```
@InterceptorBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD })
public @interface Transactional {
}
```

A classe responsável pelas interceptações deve ser anotada com `@Interceptor` e também a anotação que acabamos de criar, no caso, `@Transactional`.

O método que realizará as interceptações deve receber `InvocationContext` como parâmetro, retornar um `Object` e pode lançar `Exception`, além de ser anotado com `@AroundInvoke`. Este método deve chamar `context.proceed()`, que faz com que o método interceptado seja realmente chamado.

```
@Interceptor
@Transactional
public class TransactionInterceptor implements Serializable {

    private static final long serialVersionUID = 1L;

    private @Inject EntityManager manager;

    @AroundInvoke
    public Object invoke(InvocationContext context) throws Exception {
        EntityTransaction trx = manager.getTransaction();
        boolean criador = false;

        try {
            if (!trx.isActive()) {
                // truque para fazer rollback no que já passou
                // (senão, um futuro commit, confirmaria até mesmo
                // operações sem transação)
                trx.begin();
                trx.rollback();

                // agora sim inicia a transação
                trx.begin();

                criador = true;
            }

            return context.proceed();
        } catch (Exception e) {
            if (trx != null && criador) {
                trx.rollback();
            }

            throw e;
        } finally {
            if (trx != null && trx.isActive() && criador) {
                trx.commit();
            }
        }
    }

}
```

Precisamos registrar o interceptador no arquivo *beans.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
            http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">

    <interceptors>
        <class>
```

```
            com.algaworks.financeiro.util.TransactionInterceptor
        </class>
    </interceptors>

</beans>
```

Sempre que precisarmos de uma transação em um bean CDI, basta anotarmos o método com `@Transactional`.

```
public class CadastroLancamentos {

    @Inject
    private Lancamentos lancamentos;

    @Transactional
    public void salvar(Lancamento lancamento) throws NegocioException {
        ...
    }

}
```

12.7. Injeção em conversores JSF

Infelizmente, não é possível injetar beans CDI em conversores JSF, mas provavelmente esse recurso estará disponível em uma futura versão do JSF. Enquanto isso não se torna realidade, podemos tratar essa situação criando uma fábrica de beans CDI.

```
package com.algaworks.financeiro.util;

// imports...

public class CDILocator {

    private static BeanManager getBeanManager() {
        try {
            InitialContext initialContext = new InitialContext();
            return (BeanManager) initialContext.lookup(
                "java:comp/env/BeanManager");
        } catch (NamingException e) {
            throw new RuntimeException(
                "Não pôde encontrar BeanManager no JNDI.");
        }
    }

    @SuppressWarnings("unchecked")
    public static <T> T getBean(Class<T> clazz) {
        BeanManager bm = getBeanManager();
        Set<Bean<?>> beans = (Set<Bean<?>>) bm.getBeans(clazz);

        if (beans == null || beans.isEmpty()) {
```

```
            return null;
        }

        Bean<T> bean = (Bean<T>) beans.iterator().next();

        CreationalContext<T> ctx = bm.createCreationalContext(bean);
        T o = (T) bm.getReference(bean, clazz, ctx);

        return o;
    }

}
```

Nosso conversor `PessoaConverter` pode usar a classe `CDILocator` para obter o bean CDI referente ao repositório de pessoas.

```
@FacesConverter(forClass = Pessoa.class)
public class PessoaConverter implements Converter {

    // @Inject (ainda não é suportado)
    private Pessoas pessoas;

    public PessoaConverter() {
        this.pessoas = CDILocator.getBean(Pessoas.class);
    }

    @Override
    public Object getAsObject(FacesContext context,
            UIComponent component, String value) {
        Pessoa retorno = null;

        if (value != null) {
            retorno = this.pessoas.porId(new Long(value));
        }

        return retorno;
    }

    ...

}
```

Capítulo 13

Ajax

13.1. Introdução

Ajax (*Asynchronous JavaScript and XML*) é um grupo de tecnologias web que permite a criação de aplicações interativas. Quando usamos Ajax, as aplicações web podem fazer requisições de conteúdo ao servidor sem recarregar a tela, buscando apenas fragmentos da página que precisam ser atualizados.

Em versões anteriores do framework JSF, para que as aplicações pudessem funcionar com Ajax, era necessário adicionar bibliotecas terceiras. A partir do JSF 2, o suporte a Ajax é fornecido por uma biblioteca JavaScript integrada. Com esta biblioteca, podemos usar Ajax em componentes visuais, como botões, campos, etc.

13.2. Renderização parcial

A grande vantagem em utilizar Ajax é poder atualizar apenas os componentes necessários em uma página. Vamos exemplificar essa funcionalidade com um exemplo muito simples, que possui um campo e um componente de saída de texto.

Nosso managed bean tem um atributo `nome`, com seu *getter* e *setter*.

```
@Named
@RequestScoped
public class MeuBean {

    private String nome;

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

```
    }

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

}
```

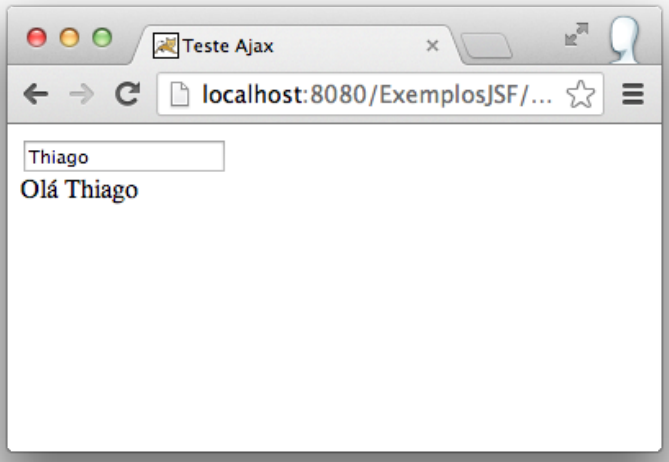
A página XHTML usa a tag <f:ajax> para adicionar funcionalidade de Ajax no componente <h:inputText>.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html">

<h:head>
    <title>Teste Ajax</title>
</h:head>

<h:body>
    <h:form id="frm">
        <h:inputText value="#{meuBean.nome}">
            <f:ajax render="ola" />
        </h:inputText>

        <br/>
        <h:outputText value="Olá #{meuBean.nome}" id="ola" />
    </h:form>
</h:body>

</html>
```

Definimos o id "ola" para o componente <h:outputText> que adicionamos, e referenciamos esse mesmo id na propriedade render de <f:ajax>.

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

Quando a funcionalidade de Ajax é adicionada em um <h:inputText>, por padrão, uma requisição Ajax é criada no evento de mudança de valor (*onchange*). No retorno dessa requisição, o componente especificado em render será atualizado. O restante da página fica como estava, sem atualizações.



13.3. A propriedade event

A propriedade `event` de `<f:ajax>` especifica o evento que criará uma nova requisição Ajax. No caso de um `<h:inputText>`, podemos usar, por exemplo, `change`, `keyup`, `mouseover`, `focus`, `blur`, `click` e etc.

Mudaremos o evento que acionará uma nova requisição Ajax de nosso exemplo para `keyup`.

```
<h:inputText value="#{meuBean.nome}">
    <f:ajax event="keyup" render="ola" />
</h:inputText>
```

Agora, a cada nova letra digitada no campo, uma nova requisição Ajax é disparada e o componente "ola" atualizado.

13.4. A propriedade listener

A propriedade `listener` deve ser informada com uma expressão de ligação de método, que será executado no servidor sempre que uma requisição Ajax for disparada. Esse método pode ou não receber um argumento do tipo `AjaxBehaviorEvent`.

Incluiremos um método `transformar` no managed bean, que será chamado nas requisições Ajax para transformar o nome digitado para letras maiúsculas.

```
public void transformar(AjaxBehaviorEvent event) {
    this.nome = this.nome.toUpperCase();
}
```

Agora, fazemos referência ao método usando EL.

```
<h:inputText value="#{meuBean.nome}">
    <f:ajax event="keyup" render="ola"
        listener="#{meuBean.transformar}" />
</h:inputText>
```

A cada nova letra digitada no campo, o método `transformar` será chamado.

13.5. Renderizações múltiplas

Além de transformar o nome digitado para maiúsculo, o método `transformar` também contará quantos caracteres foram digitados e atribuirá a uma nova variável de instância, chamada `quantidadeCaracteres`.

```
@Named
@RequestScoped
public class MeuBean {

    private String nome;
    private int quantidadeCaracteres;

    public void transformar(AjaxBehaviorEvent event) {
        this.nome = this.nome.toUpperCase();
        this.quantidadeCaracteres = this.nome.length();
    }

    public String getNome() {
        return nome;
    }

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

    public int getQuantidadeCaracteres() {
        return quantidadeCaracteres;
    }

}
```

Na página XHTML, incluímos um novo `<h:outputText>` para exibir a quantidade de caracteres digitados.

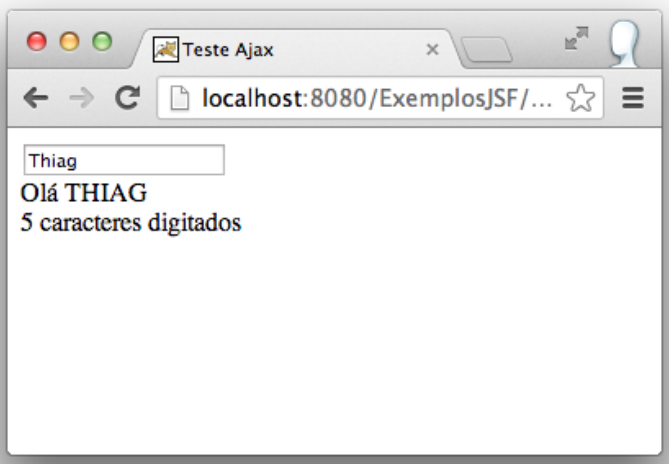
```
<h:inputText value="#{meuBean.nome}">
    <f:ajax event="keyup" render="ola"
        listener="#{meuBean.transformar}" />
</h:inputText>

<br/>
<h:outputText value="Olá #{meuBean.nome}" id="ola" />

<br/>
<h:outputText id="contador"
    value="#{meuBean.quantidadeCaracteres} caracteres digitados" />
```

Embora os caracteres estejam sendo contados pelo método `transformar`, o componente "contador" não é atualizado!

Podemos especificar mais de um componente na propriedade `render`, separando os ids por espaço.

```
<h:inputText value="#{meuBean.nome}">
    <f:ajax event="keyup" render="ola contador"
        listener="#{meuBean.transformar}" />
</h:inputText>
```

Ao digitar um nome, os dois componentes são atualizados durante as requisições Ajax.



13.6. Processamento parcial

Iremos alterar nosso exemplo para iniciar uma requisição Ajax a partir de um botão, chamando um método de ação do managed bean.

Vamos adequar o código do método `transformar`, removendo o argumento do tipo `AjaxBehaviorEvent`.

```
public void transformar() {
    this.nome = this.nome.toUpperCase();
    this.quantidadeCaracteres = this.nome.length();
}
```

No arquivo da página, removemos a funcionalidade de Ajax do `<h:inputText>` e incluímos em um novo botão.

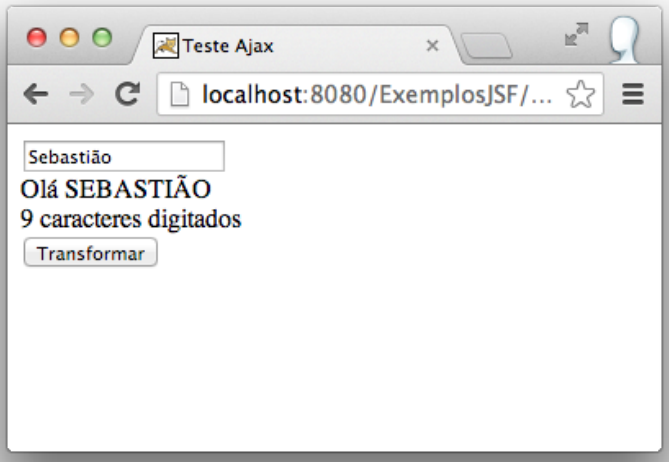
```
<h:inputText id="nome" value="#{meuBean.nome}" />

<br/>
<h:outputText value="Olá #{meuBean.nome}" id="ola" />

<br/>
<h:outputText id="contador"
    value="#{meuBean.quantidadeCaracteres} caracteres digitados" />

<br/>
<h:commandButton value="Transformar" action="#{meuBean.transformar}">
    <f:ajax render="ola contador" />
</h:commandButton>
```

Quando acessamos a página, digitamos um nome e acionamos o botão, uma exceção `NullPointerException` é lançada, pois o atributo `nome` está nulo! Uma requisição Ajax está sendo disparada, mas o componente `<h:inputText>` não está sendo processado.

Por padrão, apenas o próprio componente que adicionamos a funcionalidade de Ajax é processado pelo JavaServer Faces. Para adicionar outros componentes, usamos a propriedade `execute` de `<f:ajax>`, incluindo seus ids.

```
<h:commandButton value="Transformar" action="#{meuBean.transformar}">
    <f:ajax render="ola contador" execute="nome" />
</h:commandButton>
```

A página funciona agora, disparando requisições Ajax a partir do clique no botão "Transformar" e processando o valor digitado no campo de entrada.



13.7. Palavras-chave para render e execute

Se quisermos atualizar ou processar um grupo de componentes na página, podemos usar algumas palavras-chave nos atributos `render` e `process` de `<f:ajax>`.

Palavra-chave	Descrição
@all	Todos os componentes da página
@form	Todos os componentes do formulário que possui o componente que disparou a requisição Ajax
@none	Nenhum componente
@this	Apenas o componente que disparou a requisição Ajax

Apenas para exemplificar, vamos alterar a configuração de Ajax do botão "Transformar", para usar algumas palavras-chave.

```
<h:commandButton value="Transformar" action="#{meuBean.transformar}">
    <f:ajax render="@all" execute="@form" />
</h:commandButton>
```

Neste caso, o recomendado seria não usar a palavra-chave `@all` para atualizar a página inteira, já que apenas a atualização dos *outputs* que precisamos seria muito mais rápida.

13.8. Página de cadastro de lançamento com Ajax

Para adicionar Ajax na página de cadastro de lançamento do sistema financeiro, precisamos apenas usar a tag <f:ajax> no botão "Salvar".

```
<h:commandButton value="Salvar" action="#{cadastroLancamentoBean
        .salvar}">
    <f:ajax execute="@form" render="@form" />
</h:commandButton>
```

Para deixar a página um pouco mais interativa e facilitar a vida dos usuários, adicionaremos uma nova funcionalidade nesta página. Quando o usuário informar a data de vencimento do lançamento, automaticamente, atribuiremos a data de pagamento com a mesma data de vencimento. Se o usuário preencher a data de pagamento antes da data de vencimento, neste caso, não iremos substituir o que já foi informado.

Vamos criar um método no bean `CadastroLancamentoBean` para tratar o evento de mudança de data de vencimento.

```
public void dataVencimentoAlterada(AjaxBehaviorEvent event) {
    if (this.lancamento.getDataPagamento() == null) {
        this.lancamento.setDataPagamento(this.lancamento
            .getDataVencimento());
    }
}
```

No campo de entrada de data de vencimento, adicionamos <f:ajax> com os atributos `render`, `execute` e `listener`.

```
<h:outputLabel value="Data de vencimento" />
<h:inputText size="12" value="#{cadastroLancamentoBean.lancamento
        .dataVencimento}" label="Data de vencimento">
    <f:ajax render="@this dataPagamento"
        execute="@this dataPagamento"
        listener="#{cadastroLancamentoBean
            .dataVencimentoAlterada}" />
    <f:convertDateTime pattern="dd/MM/yyyy"
        timeZone="America/Sao_Paulo" />
</h:inputText>

<h:outputLabel value="Data de pagamento" />
```

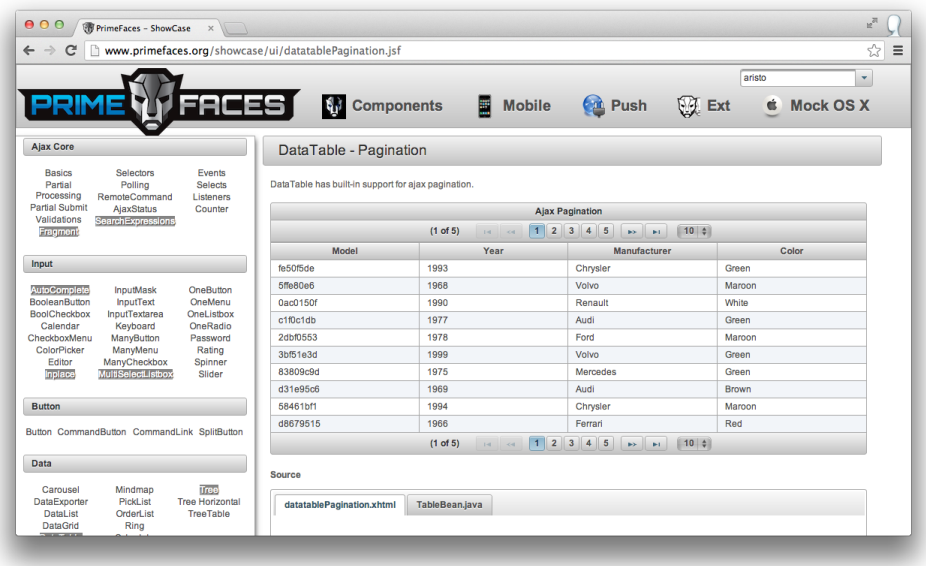
```
<h:inputText size="12" value="#{cadastroLancamentoBean.lancamento
        .dataPagamento}" label="Data de pagamento"
        id="dataPagamento">
    <f:convertDateTime pattern="dd/MM/yyyy"
        timeZone="America/Sao_Paulo" />
</h:inputText>
```

Capítulo 14

PrimeFaces

14.1. Introdução

PrimeFaces é uma bibliotecas de componentes ricos em JavaServer Faces. A suíte de componentes inclui diversos campos de entrada, botões, tabelas de dados, árvores, gráficos, diálogos, etc.



Os componentes do PrimeFaces possuem funcionalidade de Ajax integrado, baseado na API de Ajax do JSF. Para conhecer todos os componentes disponíveis, acesse o *showcase* em http://www.primefaces.org/showcase/.

14.2. Configurando o projeto

A biblioteca PrimeFaces está disponível para download em http://www.primefaces.org/. Como estamos usando Maven, vamos apenas adicionar a dependência no arquivo *pom.xml* do projeto.

```
<!-- PrimeFaces (biblioteca de componentes) -->
<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>4.0</version>
    <scope>compile</scope>
</dependency>
```

Pronto! Não precisamos configurar mais nada. Agora, é só começar a usar os componentes do PrimeFaces.

14.3. OutputLabel e InputText

Para usar componentes do PrimeFaces, precisamos importar a biblioteca de componentes na página XHTML. Iremos alterar o arquivo *CadastroLancamento.xhtml* do sistema financeiro.

```
<!DOCTYPE html>
<ui:composition template="/WEB-INF/template/Layout.xhtml"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    xmlns:p="http://primefaces.org/ui">
```

Usaremos dois componentes do PrimeFaces: `<p:outputLabel>` e `<p:inputText>`.

```
<p:outputLabel value="Descrição" for="descricao" />
<p:inputText id="descricao" size="60"
    value="#{cadastroLancamentoBean.lancamento.descricao}" />
```

O componente `<p:outputLabel>` é uma extensão do componente de mesmo nome da biblioteca do JSF. Quando associamos esse componente a um `<p:inputText>`, não precisamos especificar o `label` no campo de entrada, pois é automaticamente atribuído. Além disso, erros de validação alteram a cor do texto do *label*.

O componente `<p:inputText>` é uma extensão do componente de mesmo nome da biblioteca do JSF. O campo é renderizado com um *skin* diferente do componente

padrão. Além disso, em caso de erros associados ao componente, ele fica destacado para o usuário.

14.4. SelectOneMenu

O componente `<p:selectOneMenu>` é baseado no componente padrão do JSF, mas inclui *skinning*, suporte a edição, efeitos, filtro e conteúdo customizado.

Se mudarmos apenas o prefixo `h` para `p`, já estaremos usando o componente do PrimeFaces.

```
<p:outputLabel value="Pessoa" for="pessoa" />
<p:selectOneMenu value="#{cadastroLancamentoBean.lancamento.pessoa}"
        id="pessoa">
    <f:selectItem itemLabel="Selecione" noSelectionOption="true" />
    <f:selectItems value="#{cadastroLancamentoBean.pessoas}"
        var="pessoa" itemValue="#{pessoa}"
        itemLabel="#{pessoa.nome}" />
</p:selectOneMenu>
```

Para habilitar a funcionalidade de filtro do componente, basta atribuir `true` à propriedade `filter`. Podemos também escolher a forma que queremos filtrar os

elementos através de `filterMatchMode`, que aceita `startsWith`, `contains`, `endsWith` e `custom`.

```
<p:outputLabel value="Pessoa" for="pessoa" />
<p:selectOneMenu value="#{cadastroLancamentoBean.lancamento.pessoa}"
        id="pessoa" filter="true" filterMatchMode="contains">
    <f:selectItem itemLabel="Selecione" noSelectionOption="true" />
    <f:selectItems value="#{cadastroLancamentoBean.pessoas}"
        var="pessoa" itemValue="#{pessoa}"
        itemLabel="#{pessoa.nome}" />
</p:selectOneMenu>
```

Temos um menu de seleção prático e intuitivo!

14.5. SelectOneButton

O PrimeFaces possui o componente `<p:selectOneRadio>`, que é baseado no componente de mesmo nome da biblioteca do JSF. Um outro componente que funciona como um *radio* é o `<p:selectOneButton>`. Usaremos ele na página de cadastro de lançamentos para seleção do tipo do lançamento.

Antes de alterar a página, vamos fazer uma pequena implementação no enum `TipoLancamento`, para associarmos uma descrição de usuário para cada constante.

```
public enum TipoLancamento {

    RECEITA("Receita"),
    DESPESA("Despesa");

    private String descricao;

    TipoLancamento(String descricao) {
        this.descricao = descricao;
```

```
    }

    public String getDescricao() {
        return descricao;
    }

}
```

Agora, iremos alterar o componente de seleção do tipo de lançamento para `<p:selectOneButton>`.

```
<p:outputLabel value="Tipo" for="tipo" />
<p:selectOneButton id="tipo"
        value="#{cadastroLancamentoBean.lancamento.tipo}">
    <f:selectItems value="#{cadastroLancamentoBean.tiposLancamentos}"
        var="tipoLancamento" itemValue="#{tipoLancamento}"
        itemLabel="#{tipoLancamento.descricao}" />
</p:selectOneButton>
```

Veja o resultado:

14.6. Calendar

O `<p:calendar>` é um componente usado para selecionar data/hora, com suporte a paginação, localização, Ajax, etc.

Vamos alterar os campos de datas na página de cadastro de lançamentos para usar este componente. A propriedade `pattern` define o formato da data.

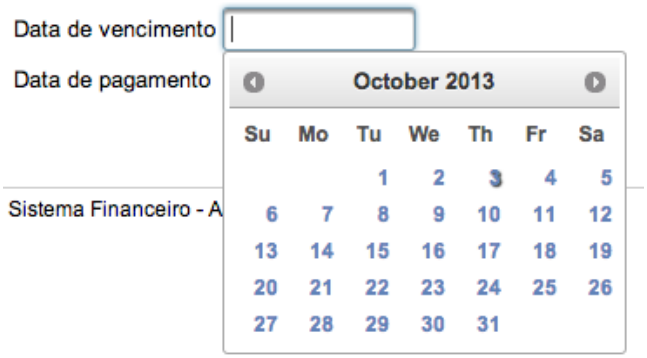
```
<p:outputLabel value="Data de vencimento" for="dataVencimento" />
<p:calendar id="dataVencimento" size="12" pattern="dd/MM/yyyy"
        value="#{cadastroLancamentoBean.lancamento.dataVencimento}">
    <p:ajax event="dateSelect" update="@this dataPagamento"
        process="@this dataPagamento"
        listener="#{cadastroLancamentoBean.dataVencimentoAlterada}" />
    <p:ajax event="change" update="@this dataPagamento"
        process="@this dataPagamento"
```

```
        listener="#{cadastroLancamentoBean.dataVencimentoAlterada}" />
</p:calendar>

<p:outputLabel value="Data de pagamento" for="dataPagamento" />
<p:calendar size="12" id="dataPagamento" pattern="dd/MM/yyyy"
    value="#{cadastroLancamentoBean.lancamento.dataPagamento}" />
```

Neste exemplo, usamos a tag `<p:ajax>` para habilitar Ajax para o primeiro componente de calendário. Esta tag é uma extensão da `<f:ajax>`, e a principal diferença é que a tag do PrimeFaces usa jQuery para fazer requisições Ajax, enquanto a tag padrão do JSF usa uma implementação própria. Além disso, a tag `<p:ajax>` muda os nomes de algumas propriedades, como é o caso de `update` e `process`.

Incluímos a tag `<p:ajax>` duas vezes para disparar requisições Ajax no evento `dateSelect` e `change`.

Veja como o componente de calendário é exibido:



14.7. AutoComplete

O componente `<p:autoComplete>` fornece sugestões enquanto o usuário digita um valor para o campo.

Iremos alterar o campo de descrição de lançamento, para sugerir ao usuário as descrições já usadas (cadastradas) anteriormente. Primeiramente, precisaremos criar um novo método no repositório `Lancamentos`.

```
public List<String> descricoesQueContem(String descricao) {
    TypedQuery<String> query = manager.createQuery(
            "select distinct descricao from Lancamento "
            + "where upper(descricao) like upper(:descricao)",
            String.class);
    query.setParameter("descricao", "%" + descricao + "%");
```

```
    return query.getResultList();
}
```

O método `descricoesQueContem` seleciona todas as descrições distintas de lançamentos, filtrando por uma parte da descrição.

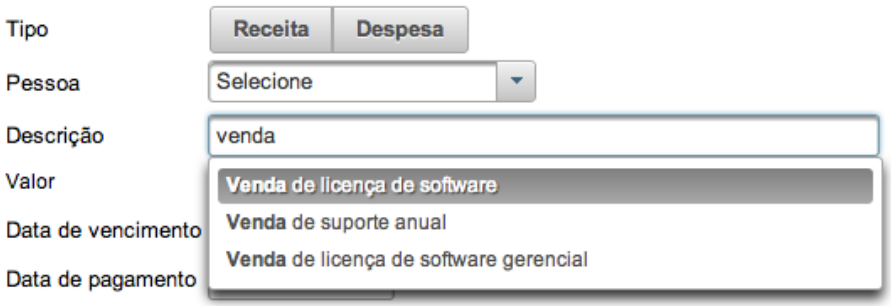
No bean `CadastroLancamentoBean`, incluiremos um método que será usado para preencher as sugestões do componente.

```
@Inject
private Lancamentos lancamentos;

public List<String> pesquisarDescricoes(String descricao) {
    return this.lancamentos.descricoesQueContem(descricao);
}
```

Na página de cadastro, basta incluirmos o componente `<p:autoComplete>` e associar o método `pesquisarDescricoes`, através da propriedade `completeMethod`.

```
<p:outputLabel value="Descrição" for="descricao" />
<p:autoComplete id="descricao" size="60"
    value="#{cadastroLancamentoBean.lancamento.descricao}"
    completeMethod="#{cadastroLancamentoBean.pesquisarDescricoes}" />
```

Agora o campo de entrada da descrição do lançamento sugere descrições completas que foram usadas em outros lançamentos.



14.8. Messages

O componente `<p:messages>` é uma extensão do componente padrão do JSF, com um visual mais atraente e suporte a Ajax.

```
<h:form id="frm">
    <p:messages showDetail="false" showSummary="true"
        autoUpdate="true" />
```

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

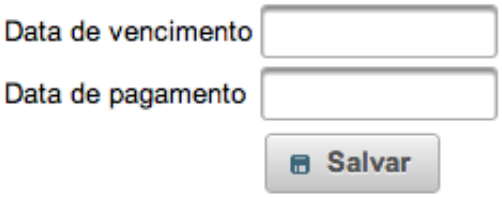
Incluímos o atributo `autoUpdate` com valor `true` para o componente ser atualizado automaticamente quando houver uma requisição Ajax, sem que seja necessário listar o id do componente de mensagem no atributo `update` de `<p:ajax>`.

14.9. CommandButton

O componente `<p:commandButton>` é uma versão mais atraente do botão padrão do JSF, além de ter a funcionalidade de Ajax integrada, sem necessidade de usar `<p:ajax>`.

```
<p:commandButton value="Salvar"
    action="#{cadastroLancamentoBean.salvar}"
    icon="ui-icon-disk" update="@form" />
```

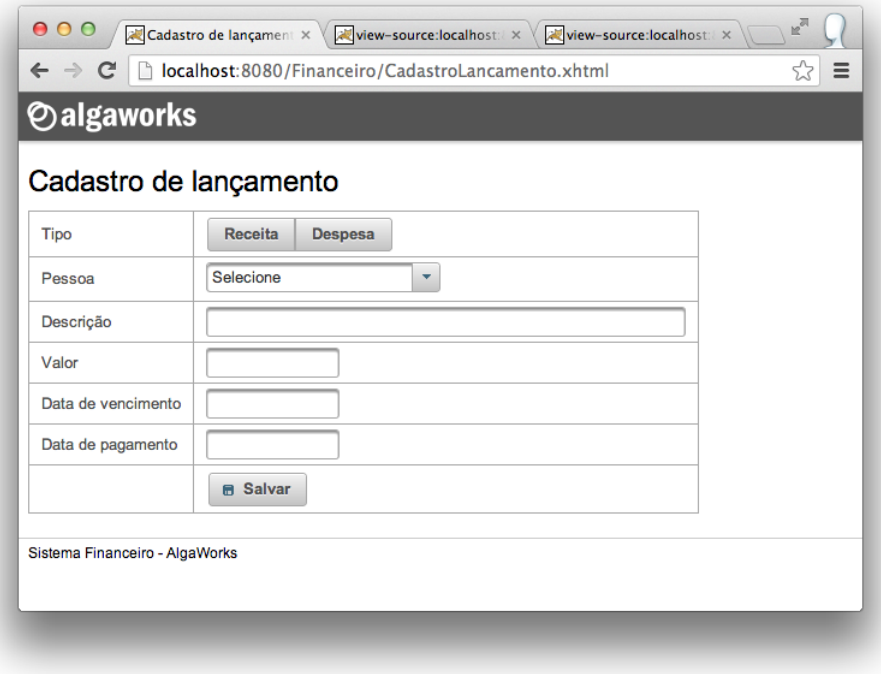
A propriedade `icon` pode ser usada para definir uma classe CSS de um ícone do botão. O PrimeFaces já possui algumas classes de ícones, que podem ser consultadas em http://jqueryui.com/themeroller/.



14.10. PanelGrid

O componente `<p:panelGrid>` é uma versão extendida do componente padrão do JSF, com inclusão do tema do PrimeFaces e suporte a mesclagem de colunas e linhas.

```
<p:panelGrid columns="2">
    ...
</p:panelGrid>
```



14.11. DataTable

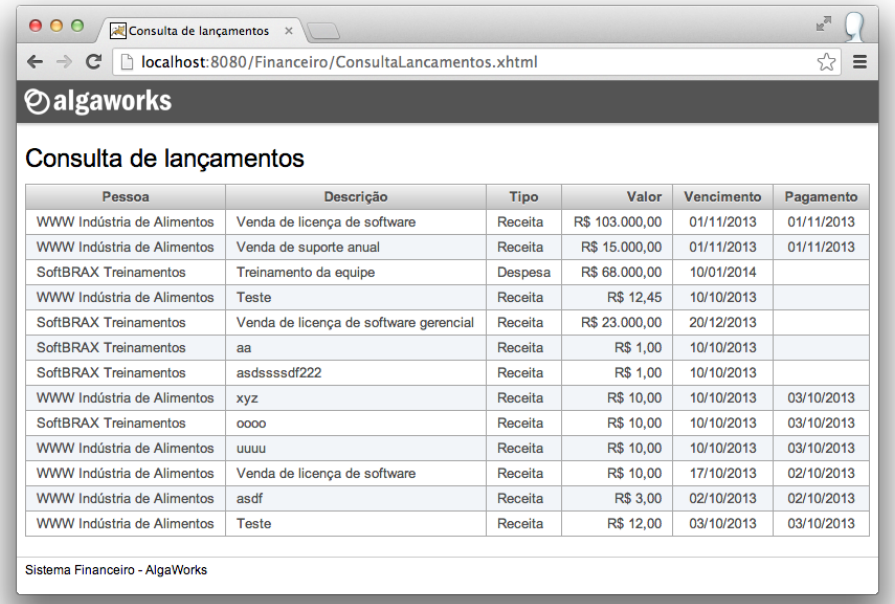
O componente `<p:dataTable>` é uma versão evoluída da tabela de dados padrão do JSF, com suporte a paginação, ordenação, seleção, filtros e muito mais.

Usamos a tag `<p:column>` para especificar as colunas da tabela de dados do PrimeFaces. Essa tag possui a propriedade `headerText`, que recebe o texto do cabeçalho da coluna.

Mudaremos a página de consulta de lançamentos do sistema financeiro para usar `<p:dataTable>`.

```
<p:dataTable value="#{consultaLancamentosBean.lancamentos}"
        var="lancamento" border="1" cellspacing="0"
        cellpadding="2">
    <p:column headerText="Pessoa">
        <h:outputText value="#{lancamento.pessoa.nome}" />
    </p:column>
    <p:column headerText="Descrição">
        <h:outputText value="#{lancamento.descricao}" />
    </p:column>
    <p:column headerText="Tipo">
        <h:outputText value="#{lancamento.tipo.descricao}" />
    </p:column>
    <p:column headerText="Valor" style="text-align: right">
        <h:outputText value="#{lancamento.valor}">
            <f:convertNumber type="currency" locale="pt_BR" />
        </h:outputText>
    </p:column>
    <p:column headerText="Vencimento" style="text-align: center">
        <h:outputText value="#{lancamento.dataVencimento}">
            <f:convertDateTime pattern="dd/MM/yyyy"
                timeZone="America/Sao_Paulo" />
        </h:outputText>
    </p:column>
    <p:column headerText="Pagamento" style="text-align: center">
        <h:outputText value="#{lancamento.dataPagamento}">
            <f:convertDateTime pattern="dd/MM/yyyy"
                timeZone="America/Sao_Paulo" />
        </h:outputText>
    </p:column>
</p:dataTable>
```

Veja o resultado:

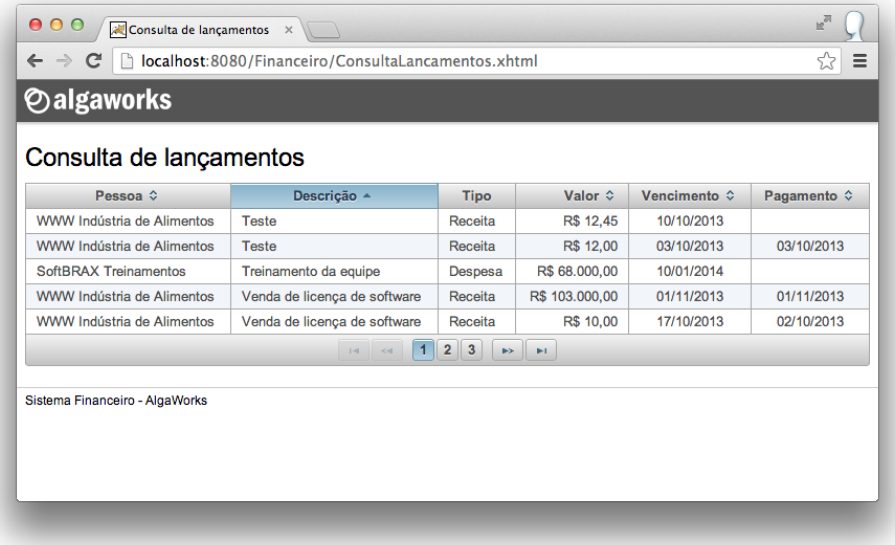


Adicionaremos o recurso de paginação na tabela de dados, através das propriedades `paginator`, `paginatorPosition` e `rows`. Habilitaremos também a ordenação por algumas colunas, especificando a propriedade `sortBy` de `<p:column>`.

```
<p:dataTable value="#{consultaLancamentosBean.lancamentos}"
        var="lancamento" border="1" cellspacing="0"
        cellpadding="2" paginator="true" rows="5"
        paginatorPosition="bottom">
    <p:column headerText="Pessoa" sortBy="#{lancamento.pessoa.nome}">
        <h:outputText value="#{lancamento.pessoa.nome}" />
    </p:column>
    <p:column headerText="Descrição" sortBy="#{lancamento.descricao}">
        <h:outputText value="#{lancamento.descricao}" />
    </p:column>
    <p:column headerText="Tipo">
        <h:outputText value="#{lancamento.tipo.descricao}" />
    </p:column>
    <p:column headerText="Valor" style="text-align: right"
            sortBy="#{lancamento.valor}">
        <h:outputText value="#{lancamento.valor}">
            <f:convertNumber type="currency" locale="pt_BR" />
        </h:outputText>
    </p:column>
    <p:column headerText="Vencimento" style="text-align: center"
            sortBy="#{lancamento.dataVencimento}">
        <h:outputText value="#{lancamento.dataVencimento}">
            <f:convertDateTime pattern="dd/MM/yyyy"
```

```
                timeZone="America/Sao_Paulo" />
        </h:outputText>
    </p:column>
    <p:column headerText="Pagamento" style="text-align: center"
            sortBy="#{lancamento.dataPagamento}">
        <h:outputText value="#{lancamento.dataPagamento}">
            <f:convertDateTime pattern="dd/MM/yyyy"
                timeZone="America/Sao_Paulo" />
        </h:outputText>
    </p:column>
</p:dataTable>
```

Temos uma tabela de dados com paginação e ordenação.



14.12. Menubar

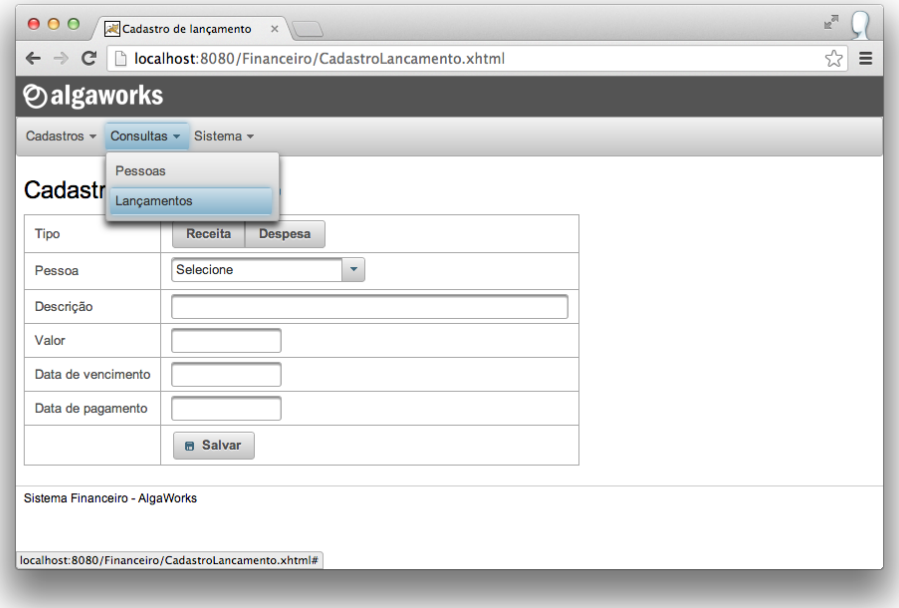
O componente <p:menubar> renderiza uma barra de menu horizontal, parecida com as barras em sistemas desktop.

Incluiremos uma barra de menu no arquivo *Layout.xhtml* do sistema financeiro.

```
<h:body>
    ...

    <h:form>
        <p:menubar style="margin-top: -20px; margin-bottom: 20px">
            <p:submenu label="Cadastros">
                <p:menuitem value="Pessoa" />
```

```
                <p:menuitem value="Lançamento"
                    action="/CadastroLancamento?faces-redirect=true" />
            </p:submenu>
            <p:submenu label="Consultas">
                <p:menuitem value="Pessoas" />
                <p:menuitem value="Lançamentos"
                    action="/ConsultaLancamentos?faces-redirect=true" />
            </p:submenu>
            <p:submenu label="Sistema">
                <p:menuitem value="Sair" />
            </p:submenu>
        </p:menubar>
    </h:form>

    ...
</h:body>
```

Veja que, para usar um <p:menubar>, tivemos que usar outros componentes auxiliares, como <p:submenu> e <p:menuitem>.



14.13. AjaxStatus

O componente <p:ajaxStatus> é um indicador global de requisições Ajax. Podemos, por exemplo, adicionar uma imagem que indica que algo está sendo carregado, para que o usuário espere o processamento da página.

No arquivo *Layout.xhtml* do sistema financeiro, incluiremos o seguinte código:

```
<p:ajaxStatus styleClass="ajax-status">
    <f:facet name="start">
        <h:graphicImage library="algaworks" name="loading.gif" />
    </f:facet>
    <f:facet name="complete">
        <h:outputText value="" />
    </f:facet>
</p:ajaxStatus>
```

O *facet* `start` define o que será exibido quando uma requisição Ajax for iniciada, e o *facet* `complete` substitui o conteúdo do componente quando a requisição Ajax for encerrada.

Usamos uma classe CSS chamada `ajax-status` para posicionar o componente na tela, por isso, precisamos incluir o código abaixo no arquivo *estilo.css*.

```
.ajax-status {
    position: fixed;
    top: 85px;
    right: 10px;
    width: 35px;
    height: 35px
}
```

14.14. Programando a alteração de lançamentos

Para deixar o sistema financeiro um pouco mais completo, implementaremos a edição de lançamentos nesta seção.

No repositório `Lancamentos`, criaremos um método `guardar`, que servirá tanto para inserir como atualizar objetos, e outro `porId`, que retorna um lançamento pelo identificador.

```
public Lancamento porId(Long id) {
    return manager.find(Lancamento.class, id);
}

public Lancamento guardar(Lancamento lancamento) {
    return this.manager.merge(lancamento);
}
```

Na classe `CadastroLancamentos`, chamaremos o método `guardar`, e não mais `adicionar`.

```
public void salvar(Lancamento lancamento) throws NegocioException {
    ...

    this.lancamentos.guardar(lancamento);
}
```

Precisaremos de um conversor de lançamentos, por isso, já vamos deixá-lo pronto.

```
@FacesConverter(forClass = Lancamento.class)
public class LancamentosConverter implements Converter {

    // @Inject (ainda não é suportado)
    private Lancamentos lancamentos;

    public LancamentosConverter() {
        this.lancamentos = CDILocator.getBean(Lancamentos.class);
    }

    @Override
    public Object getAsObject(FacesContext context,
            UIComponent component, String value) {
        Lancamento retorno = null;

        if (value != null) {
            retorno = this.lancamentos.porId(new Long(value));
        }

        return retorno;
    }

    @Override
    public String getAsString(FacesContext context,
            UIComponent component, Object value) {
        if (value != null) {
            Lancamento lancamento = ((Lancamento) value);
            return lancamento.getId() == null ? null
                : lancamento.getId().toString();
        }
        return null;
    }

}
```

Na tabela de dados da página *ConsultaLancamentos.xhtml*, adicionaremos uma coluna com um botão de edição. Usamos o componente `<p:button>`, que recebe um *outcome* para direcionar o usúario para a página de cadastro de lançamento. Passaremos como parâmetro o id do lançamento, pois vamos precisar dessa informação para exibir a página de cadastro preenchida com as informações do lançamento selecionado.

```
<p:column>
    <p:button icon="ui-icon-pencil" title="Editar"
```

```
            outcome="/CadastroLancamento">
        <f:param name="id" value="#{lancamento.id}" />
    </p:button>
</p:column>
```

Usaremos uma biblioteca de utilitários para JSF, chamada OmniFaces. Vamos incluir a dependência no *pom.xml*.

```
<!-- OmniFaces (utilitarios para JSF) -->
<dependency>
    <groupId>org.omnifaces</groupId>
    <artifactId>omnifaces</artifactId>
    <version>1.6</version>
    <scope>compile</scope>
</dependency>
```

Na página *CadastroLancamento.xhtml*, importamos a biblioteca do OmniFaces e incluímos `<o:viewParam>` em `<f:metadata>`.

```
<!DOCTYPE html>
<ui:composition template="/WEB-INF/template/Layout.xhtml"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://xmlns.jcp.org/jsf/core"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    xmlns:p="http://primefaces.org/ui"
    xmlns:o="http://omnifaces.org/ui">

    <f:metadata>
        <o:viewParam name="id"
            value="#{cadastroLancamentoBean.lancamento}" />
        <f:viewAction
            action="#{cadastroLancamentoBean.prepararCadastro}" />
    </f:metadata>
```

A tag `<o:viewParam>` do OmniFaces receberá o id do lançamento como parâmetro na *querystring* e atribuirá à variável `lancamento` do bean. O conversor de lançamentos entrará em ação neste momento.

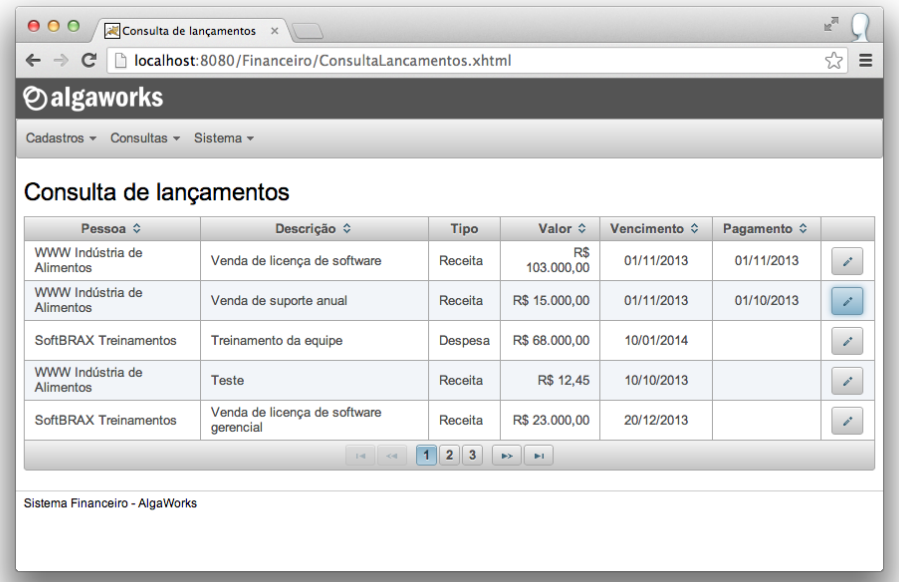
A diferença entre `<f:viewParam>` e `<o:viewParam>`, é que a tag do OmniFaces não chama o *setter* da propriedade em *postback*, e isso é importante para o objeto não ser substituído no meio de uma edição.

No bean `CadastroLancamentoBean`, alteramos o código do método `prepararCadastro`, referenciado pela tag `<f:viewAction>`, para instanciar um novo `Lancamento` quando ele estiver nulo.

```
public void prepararCadastro() {
    this.todasPessoas = this.pessoas.todas();
```

```
    if (this.lancamento == null) {
        this.lancamento = new Lancamento();
    }
}
```

Temos a edição de lançamentos funcionando!



14.15. Programando a exclusão de lançamentos

Adicionaremos a funcionalidade de exclusão de lançamentos na página de consulta de lançamentos do sistema financeiro.

No repositório de lançamentos, vamos adicionar um novo método para remover lançamentos.

```
public void remover(Lancamento lancamento) {
    this.manager.remove(lancamento);
}
```

Na classe `CadastroLancamentos`, implementaremos as regras de negócio para excluir lançamentos no método `excluir`.

```
@Transactional
public void excluir(Lancamento lancamento) throws NegocioException {
    lancamento = this.lancamentos.porId(lancamento.getId());

    if (lancamento.getDataPagamento() != null) {
        throw new NegocioException(
            "Não é possível excluir um lançamento pago!");
    }

    this.lancamentos.remover(lancamento);
}
```

No managed bean `ConsultaLancamentosBean`, adicionamos um método `excluir` e uma variável de instância `lancamentoSelecionado`. Esta variável será atribuída pela página, quando o usuário selecionar o lançamento que deseja excluir.

```
public class ConsultaLancamentosBean implements Serializable {

    ...

    private Lancamento lancamentoSelecionado;

    public void excluir() {
        FacesContext context = FacesContext.getCurrentInstance();

        try {
            this.cadastro.excluir(this.lancamentoSelecionado);
            this.consultar();

            context.addMessage(null, new FacesMessage(
                "Lançamento excluído com sucesso!"));
        } catch (NegocioException e) {

            FacesMessage mensagem = new FacesMessage(e.getMessage());
            mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
            context.addMessage(null, mensagem);
        }
    }

    ...

}
```

Precisaremos adicionar o componente `<h:messages>` na página *ConsultaLancamentos.xhtml*, para darmos feedback para o usuário sobre a solicitação de exclusão de lançamentos.

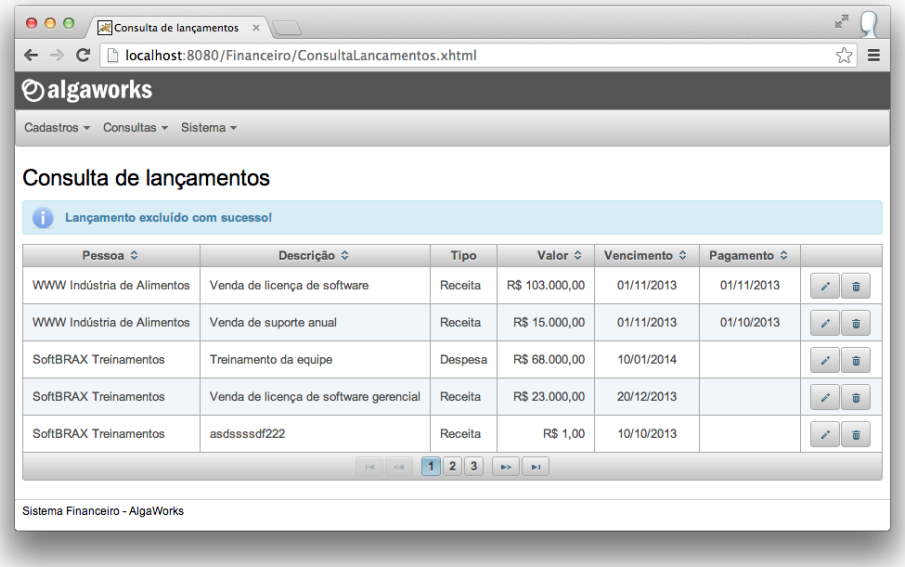
```
<h:form id="frm">
    <p:messages showDetail="false" showSummary="true"
        autoUpdate="true" />
```

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

Na última coluna da tabela de dados, adicionamos um botão de comando que invoca o método `excluir` do managed bean. Usamos a tag `<f:setPropertyActionListener>`, que é um *action listener* que atribui um objeto especificado em `value` em um destino especificado em `target`. *Action listeners* são executados antes de *actions*, portanto, quando o método `excluir` for chamado, o lançamento selecionado já estará atribuído na variável de instância do managed bean.

```
<p:commandButton icon="ui-icon-trash" title="Excluir" process="@this"
        update="@form" action="#{consultaLancamentosBean.excluir}">
    <f:setPropertyActionListener value="#{lancamento}"
        target="#{consultaLancamentosBean.lancamentoSelecionado}" />
</p:commandButton>
```

Pronto! Já podemos excluir lançamentos a partir da tela de consulta.



Capítulo 15

Segurança da aplicação

15.1. Escolhendo uma solução

Existem diversas maneiras de implementar autenticação e autorização em sistemas JSF, tais como:

- *Realms* de segurança do container
- Frameworks de segurança, como Spring Security
- Manualmente, com criação de filtros servlet

Escolheremos a última opção, que é a mais simples.

15.2. Login

Vamos criar um mecanismo de login simples, mas que pode ser usado em sistemas reais. Não buscaremos usuários e senhas de um banco de dados, mas deixaremos *hardcoded*. Claro que você deve buscar os dados de algum outro lugar, por exemplo usando JPA, mas neste capítulo focaremos em uma solução de autenticação e autorização.

Criaremos uma classe `Usuario` que representará um usuário do sistema.

```
@Named
@SessionScoped
public class Usuario implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nome;
```

```
    private Date dataLogin;

    public boolean isLogado() {
        return nome != null;
    }

    public String getNome() {
        return nome;
    }

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

    public Date getDataLogin() {
        return dataLogin;
    }

    public void setDataLogin(Date dataLogin) {
        this.dataLogin = dataLogin;
    }

}
```

Agora, criaremos um managed bean que será usado pela página de login. Este bean injeta `Usuario` e atribui algumas variáveis quando a autenticação ocorre com sucesso.

```
@Named
@RequestScoped
public class LoginBean {

    @Inject
    private Usuario usuario;

    private String nomeUsuario;
    private String senha;

    public String login() {
        FacesContext context = FacesContext.getCurrentInstance();

        if ("admin".equals(this.nomeUsuario)
                && "123".equals(this.senha)) {
            this.usuario.setNome(this.nomeUsuario);
            this.usuario.setDataLogin(new Date());

            return "/ConsultaLancamentos?faces-redirect=true";
        } else {
            FacesMessage mensagem = new FacesMessage(
                "Usuário/senha inválidos!");
            mensagem.setSeverity(FacesMessage.SEVERITY_ERROR);
            context.addMessage(null, mensagem);
        }

        return null;
```

```
    }

    public String getNomeUsuario() {
        return nomeUsuario;
    }

    public void setNomeUsuario(String nomeUsuario) {
        this.nomeUsuario = nomeUsuario;
    }

    public String getSenha() {
        return senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }

}
```

Criamos uma página chamada *Login.xhtml*, com o seguinte código:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://xmlns.jcp.org/jsf/html"
    xmlns:p="http://primefaces.org/ui">

<h:head>
    <title>Login</title>
    <h:outputStylesheet library="algaworks" name="estilo.css" />
</h:head>

<h:body>

    <div id="login-dialog">
        <h1>Login</h1>

        <h:form id="frm">
            <p:messages autoUpdate="true" />

            <h:panelGrid columns="2" styleClass="grid-login">
                <p:outputLabel value="Usuário" for="usuario" />
                <p:inputText id="usuario" size="20"
                    value="#{loginBean.nomeUsuario}" />

                <p:outputLabel value="Senha" for="senha" />
                <p:password id="senha" size="20"
                    value="#{loginBean.senha}" />

                <p:outputLabel />
                <p:commandButton value="Acessar"
                    action="#{loginBean.login}" />
            </h:panelGrid>
        </h:form>
```

```
    </div>

</h:body>

</html>
```

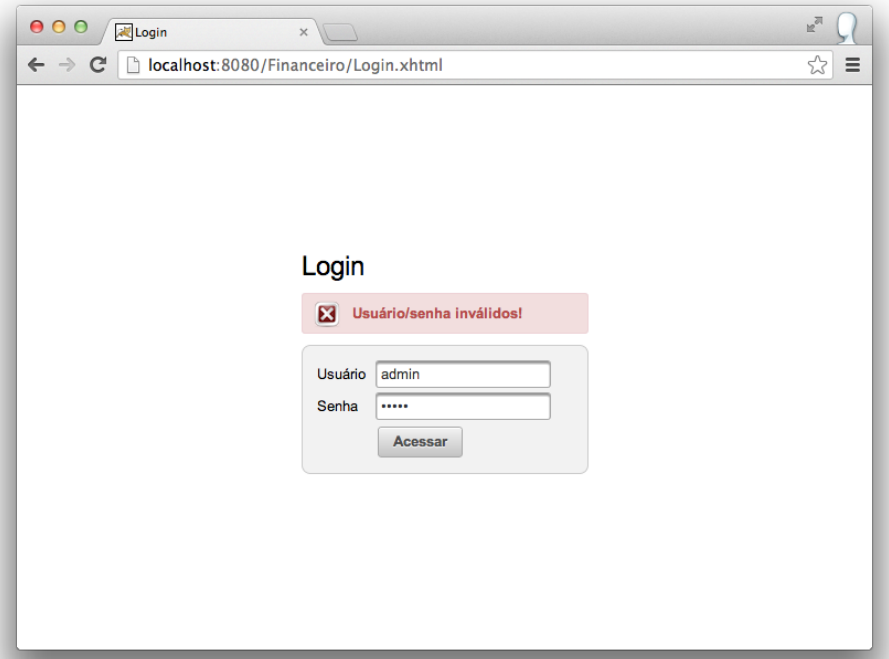
A página de login usa algumas classes CSS para deixar o formulário mais atraente. Precisamos incluir o código abaixo no arquivo *estilo.css*.

```
#login-dialog {
    width: 260px;
    margin: auto;
    margin-top: 150px;
}

.grid-login {
    background-color: #f2f2f2;
    border-radius: 8px;
    border: 1px solid #ccc;
    margin-top: 8px;
    padding: 10px;
    width: 100%
}
```

No arquivo *Layout.xhtml*, incluímos uma mensagem com o nome do usuário logado no cabeçalho.

```
<header>
    <h:graphicImage library="algaworks" name="logo.png" />

    <div style="float: right; margin-right: 20px; margin-top: 8px">
        Olá #{usuario.nome}!
    </div>
</header>
```

Já podemos acessar a página de login diretamente e logar com o usuário "admin".



15.3. Logout

Para fazer logout do sistema, basta invalidarmos a sessão do usuário. Criaremos um método `logout` no managed bean `LoginBean` para fazer isso.

```
public String logout() {
    FacesContext.getCurrentInstance().getExternalContext()
        .invalidateSession();
    return "/Login?faces-redirect=true";
}
```

Este método é chamado no item de menu "Sair", que fica no arquivo *Layout.xhtml*.

```
<p:submenu label="Sistema">
    <p:menuitem value="Sair" action="#{loginBean.logout}" />
</p:submenu>
```

15.4. Filtro de autorização

O usuário já pode fazer login e logout no sistema, mas as páginas ainda estão abertas, ou seja, se o usuário souber as URLs das páginas, ainda não bloquearemos o acesso.

Para fazer a autorização de usuários, criaremos um filtro servlet, que redireciona o usuário para a página de login quando o mesmo não é reconhecido como um usuário logado.

```
package com.algaworks.financeiro.filter;

// imports...

@WebFilter("*.xhtml")
public class AutorizacaoFilter implements Filter {

    @Inject
    private Usuario usuario;

    @Override
    public void doFilter(ServletRequest req, ServletResponse res,
            FilterChain chain) throws IOException, ServletException {
        HttpServletResponse response = (HttpServletResponse) res;
        HttpServletRequest request = (HttpServletRequest) req;

        if (!usuario.isLogado()
                && !request.getRequestURI().endsWith("/Login.xhtml")
                && !request.getRequestURI()
                    .contains("/javax.faces.resource/")) {
            response.sendRedirect(request.getContextPath()
                + "/Login.xhtml");
        } else {
            chain.doFilter(req, res);
        }
    }

    @Override
    public void init(FilterConfig config) throws ServletException {
    }

    @Override
    public void destroy() {
    }

}
```

O sistema de segurança está funcionando completamente!

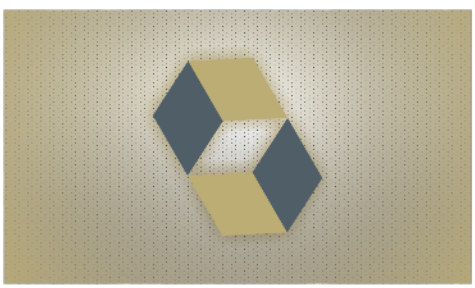
algaworks

CURSOS ONLINE

COM VÍDEO AULAS E SUPORTE

Fundamentos Java e
Orientação a Objetos

Desenvolvimento Web
com JSF 2



Persistência de Dados com
JPA 2 e Hibernate

Sistemas Comerciais Java EE
com CDI, JPA e PrimeFaces

www.**algaworks**.com

Cursos presenciais in-company? Entre em contato.

JAVA EE 7

Com JSF, PrimeFaces e CDI

THIAGO FARIA

algaworks