argonavis
tecnologia e arte

JSF
javaserver faces

JAVA EE 7

Helder da Rocha

O texto foi elaborado como material de apoio para treinamentos especializados em linguagem Java e explora assuntos detalhados nas especificações e documentações oficiais sobre o tema, utilizadas como principais fontes. A autoria deste texto é de inteira responsabilidade do seu autor, que o escreveu independentemente com finalidade educativa e não tem qualquer relação com a Oracle.

O código-fonte relacionado aos tópicos abordados neste material estão em:

`github.com/helderdarocha/javaee7-course`
`github.com/helderdarocha/CursoJavaEE_Exercicios`
`github.com/helderdarocha/ExercicioMinicursoJMS`
`github.com/helderdarocha/JavaEE7SecurityExamples`

www.argonavis.com.br

R672p Rocha, Helder Lima Santos da, 1968-

Programação de aplicações Java EE usando Glassfish e WildFly.

360p. 21cm x 29.7cm. PDF.

Documento criado em 16 de outubro de 2013.

Atualizado e ampliado entre setembro e dezembro de 2016.

Volumes (independentes): *1: Introdução, 2: Servlets, 3: CDI, 4: JPA, 5: EJB, 6: SOAP, 7: REST, 8: JSF, 9: JMS, 10: Segurança, 11: Exercícios*.

1. Java (*Linguagem de programação de computadores*). 2. Java EE (*Linguagem de programação de computadores*). 3. Computação distribuída (*Ciência da Computação*). I. Título.

CDD 005.13'3

Capítulo 8: Java Server Faces (JSF)

1 Introdução

Java Server Faces é um framework de interface do usuário (UI) para aplicações Web em Java. Possui uma arquitetura baseada em uma árvore de componentes cujo estado e comportamento são mapeados a tags, e sincronizados em tempo de execução. Implementa uma arquitetura Web-MVC (Model-View-Controller) ao permitir a separação das responsabilidades da apresentação gráfica (View), processamento de informações (Controller) e estado dos componentes (Model). Essa separação de responsabilidades, um dos fundamentos da arquitetura Java EE, permite a criação de GUIs em HTML puro, delegando tarefas de controle envolvendo ciclos de vida, estado e eventos para objetos Java. As camadas são interligadas com baxíssimo acoplamento através de uma Linguagem de Expressões. A separação de responsabilidades facilita testes, desenvolvimento, manutenção, evolução. Também é uma tecnologia baseada em padrões e independente de ferramentas. É também um framework extensível, que pode ser ampliado e melhorado através de frameworks integrados, desenvolvidos por terceiros.

Alguns benefícios do JSF 2.2 incluem

- Transparência no gerenciamento do estado e escopo das requisições síncronas e assíncronas
- Controle declarativo e condicional de navegação e suporte a processamento multi-página de formulários
- Suporte nativo e extensível a *validação*, *eventos* e *conversão* automática de tipos entre as camadas da aplicação
- Encapsulamento de diferenças entre browsers
- Plataforma extensível (através de bibliotecas de componentes criadas por terceiros)

O JSF esconde detalhes de baixo nível da plataforma HTTP, retirando do desenvolvedor a necessidade de lidar com eles, mas permite que sejam monitorados e configurados se necessário.

Assim como os outros padrões Java EE, a especificação JSF (através da arquitetura MVC) promove a separação de responsabilidades permitindo a divisão de tarefas durante o desenvolvimento entre diferentes perfis de desenvolvedores. A especificação define os seguintes perfis:

- **Autores de página**: *Programadores Web* que poderão construir páginas em HTML usando tags, facelets, bibliotecas de terceiros. *Responsabilidades*: construir views em

XHTML, imagens, CSS, etc., interligar views com componentes usando a linguagem declarativa EL, e declarar namespaces para usar bibliotecas de tags e extensões.

- **Autores de componentes**: *Programadores Java SE* que poderão construir os componentes que serão mapeados a tags, ou que irão suportar páginas e aplicações. *Responsabilidades*: criar conversores, validadores, managed beans, event handlers e eventualmente escrever componentes novos.
- **Desenvolvedores de aplicação**: *Programadores Java EE* que irão utilizar o JSF como interface para serviços e aplicações. *Responsabilidades*: configurar a integração de aplicações JSF com dados e serviços (componentes EJBs, JPA, MDB, SOAP, REST), através de CDI e JNDI.
- **Fornecedores de ferramentas** e **implementadores JSF**: Usam a especificação para construir ferramentas e containers/servidores que irão suportar aplicações JSF.

As principais *desvantagens* do JSF incluem: a complexidade e flexibilidade de sua arquitetura e a grande quantidade de componentes, tags e atributos, que impõe uma longa curva de aprendizado; e o controle rígido que detém sobre a estrutura das views, baseada em geração de código HTML, JavaScript e CSS, que dificulta o uso independente dessas tecnologias em aplicações baseadas em HTML5 e frameworks populares como Bootstrap e JQuery. É possível a integração com essas tecnologias, mas não de forma transparente.

1.1 Como construir aplicações JSF

O desenvolvimento JSF envolve a criação dos componentes de uma arquitetura MVC: *Views*, representados principalmente por documentos XHTML (compostos de tags especiais chamados de *facelets*), e o uso e criação de *Controllers* e *Models*, representados principalmente por componentes Java (*managed beans*, conversores, modelos de componentes, handlers de eventos, etc.)

O *managed bean* é um papel exercido por uma classe Java comum (POJO, ou *JavaBean*) que serve para *mapear* o estado dos componentes de um formulário (ou os modelos dos próprios componentes) e para definir operações que podem ser realizadas, de forma sincronizada, sobre os dados e componentes. Managed beans são simplesmente POJOs que são gerenciados pelo runtime do JSF. Geralmente eles são disponibilizados para injeção via EL (Expression Language) através de anotações CDI ou EJB.

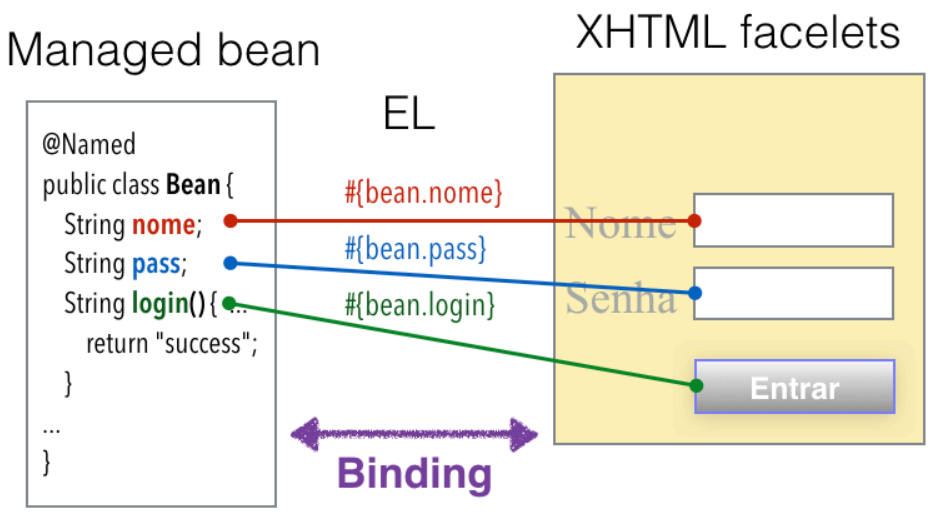
Exemplo de um POJO exercendo o papel de managed bean:

```
@Named
@RequestScoped
public class Bean {
    private String name;
    private String pass;
```

```
    public String getName() { return name; }
    public String getPass() { return pass; }
    public String login() {
        if(service.check(name, pass)) {
          return "success";
        }
        return "fail";
    }
}
```

O design das interfaces é realizado usando tags especiais que são mapeados a componentes de bibliotecas do JSF. Os tags são *templates* para tags HTML que serão gerados pelo JSF e preenchidos pelo *managed bean*. Para indicar em que parte da View e como serão usados ou transformados os dados recebidos pelo bean, o autor de uma página JSF usa o *Expression Language (EL)* que, através de uma sintaxe declarativa, permite acesso a componentes do contexto Web, managed beans, coleções e mapas, além de permitir operações aritméticas, booleanas e de strings.

O desenho abaixo ilustra como um managed bean interage com uma View do JSF:



Os managed beans contém *propriedades,* que são mapeadas a dados de formulários, e *métodos* que implementam ações, handlers de eventos, operações de conversão e validação e que podem ser chamados por eventos gerados na View. O managed bean também cuida da navegação. Típicamente um managed bean está associado a uma View, que normalmente é composto de um único documento XHTML (mas também que pode ser representado por múltiplas páginas embutidas de um template ou componente, ou ainda ser fragmento de uma página maior).

A configuração de aplicações JSF é realizada através de anotações aplicados em classes Java, ou através dos arquivos *faces-config.xml* e *web.xml*.

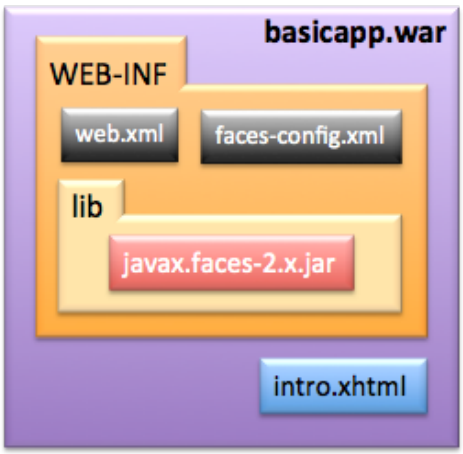
1.2 Exemplos de aplicações JSF

Uma aplicação JSF *mínima* consiste de um arquivo XHTML usando facelets (tags JSF) empacotado em uma aplicação Web (arquivo WAR). Se o servidor usado não tiver uma implementação nativa do runtime JSF, ela pode ser incluída da distribuição do componente (na pasta WEB-INF/lib). Uma opção é a implementação de referência (codinome Mojarra) que é distribuída como um único JAR.

O arquivo *faces-config.xml* é opcional em JSF 2.x, e se presente, pode estar vazio.

Se o *web.xml* não for incluído, o mapeamento da URL de acesso é determinada pela implementação usada. O *Mojarra*, por exemplo, mapeia automaticamente arquivos terminados em *.faces ou *.jsf para processamento pelo runtime JSF. Para escolher outro mapeamento é preciso criar um *web.xml* e definir um *servlet-mapping* próprio.

A figura a seguir ilustra uma aplicação JSF mínima. Apesar dos arquivos *web.xml* e *faces-config.xml* serem opcionais, é uma boa prática tê-los presentes pois não é incomum serem necessários, mesmo que se decida usar os defaults. Há configurações que não são possíveis apenas através de anotações (ex: navegação condicional), e registro de listeners e filtros ainda requerem o *web.xml*.



web.xml é opcional em aplicações simples como esta
faces-config.xml é opcional em muitas aplicações típicas

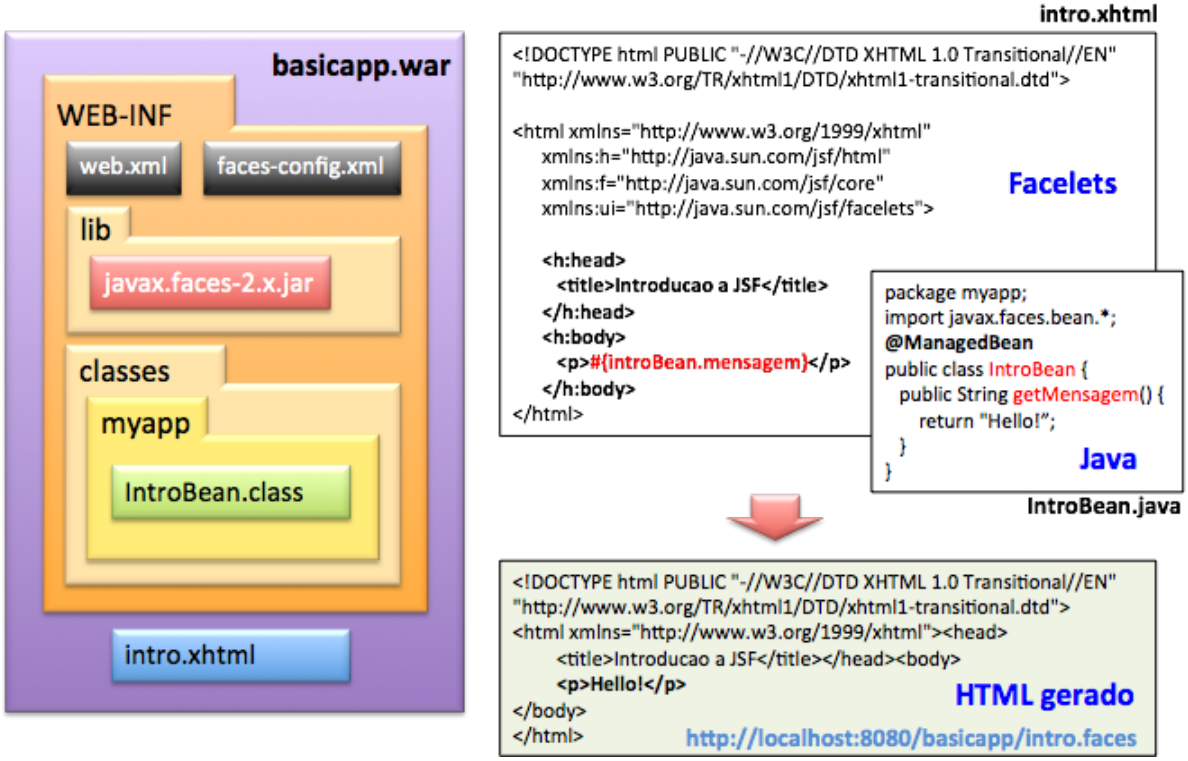
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets">
                                                    Facelets
    <h:head>
      <title>Introducao a JSF</title>
    </h:head>
    <h:body>
      <h:outputText value="Esta funcionando!"/>
    </h:body>
</html>                                             intro.xhtml
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title>Introducao a JSF</title>
</head>
<body>Esta funcionando!</body>                  HTML gerado
</html>
```

http://localhost:8080/basicapp/intro.faces

A aplicação mostrada é mínima e serve para testar um ambiente quanto ao suporte JSF. Uma aplicação típica contém pelo menos um managed bean, como ilustrado na figura a seguir.



A ilustração mostra um bean anotado com *@ManagedBean* em vez de *@Named*. *@ManagedBean* é uma anotação válida, mas será deprecada no próximo lançamento do Java EE (junto com as outras anotações do mesmo pacote) em favor de *@Named*, que é a anotação do CDI. Neste tutorial usaremos apenas as anotações CDI.

1.2.1 Página web mínima

Um *web.xml* com a configuração default já é fornecido no JAR da implementação de referência (Mojarra 2.x), portanto não é preciso criar um *web.xml* a menos que se deseje configurar o ambiente adicionando listeners e outros recursos. Se criado, o *web.xml* irá substituir o fornecido pelo Mojarra e deve ter a configuração mínima abaixo:

```
<web-app version="2.5"
      xmlns="http://java.sun.com/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
     <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>*.faces</url-pattern>
    </servlet-mapping>
</web-app>
```

O exemplo acima mantém o mapeamento *.faces* padrão do Mojarra, mas ele pode ser alterado. Um mapeamento comum é usar */faces/.xhtml* ou simplesmente **.xhtml*.

O *faces-config.xml* é usado para configurar mapeamentos de componentes, navegação, conversores, validadores, managed beans e outros recursos do JSF. Tudo isto pode ser feito via comportamento default e anotações, mas às vezes é mais prático, legível e eficiente fazer via XML (quando se usa ferramentas gráficas que geram XML, por exemplo).

A configuração *faces-config.xml* tem precedência e sobrepõe a configuração via anotações. Se usada, deve ter a seguinte configuração mínima

```
<?xml version="1.0"?>
<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
    version="2.0">
</faces-config>
```

Ambos devem ser colocados na raiz da pasta WEB-INF.

Para usar JSF 2.0, a versão do *web.xml* deve ser 2.5 ou superior e do faces-config 2.0 ou superior. JSF 2.1 requer *web.xml* 3.0 e JSF 2.2 usa *web.xml* 3.1.

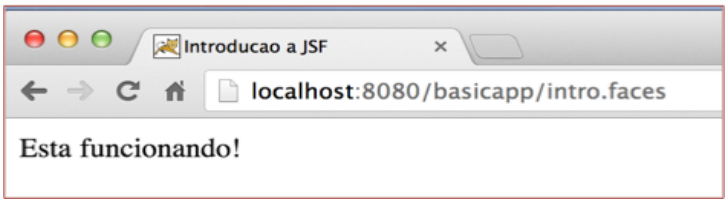
1.2.2 Como executar

Para executar uma aplicação JSF, ela precisa ser implantada (deploy) em um servidor de aplicações ou servidor Web que contenha um servlet container. No exemplo abaixo usamos o servidor de referência para containers Web do Java EE, o Tomcat, mas as regras valem para qualquer servidor de aplicações.

O deploy no Tomcat consiste em transferir o WAR para a *$TOMCAT_HOME/webapps/* ou usar a interface de administração para fazer upload do WAR e iniciá-lo. Neste exemplo transferimos o arquivo *basicapp.war* para *webapps*. Uma vez implantada a aplicação, é preciso acessar através da URL *servidor/porta/contexto/página*. No caso do Tomcat, o nome do contexto é o nome do WAR (*basicapp*). Portanto, a URL completa para executar a aplicação é:

`http://localhost:8080/basicapp/intro.faces`

Se não houver erros, o resultado deve ser similar ao ilustrado abaixo:



A instalação usando outro servidor ou IDE deve produzir o mesmo resultado.

1.2.3 Com managed bean

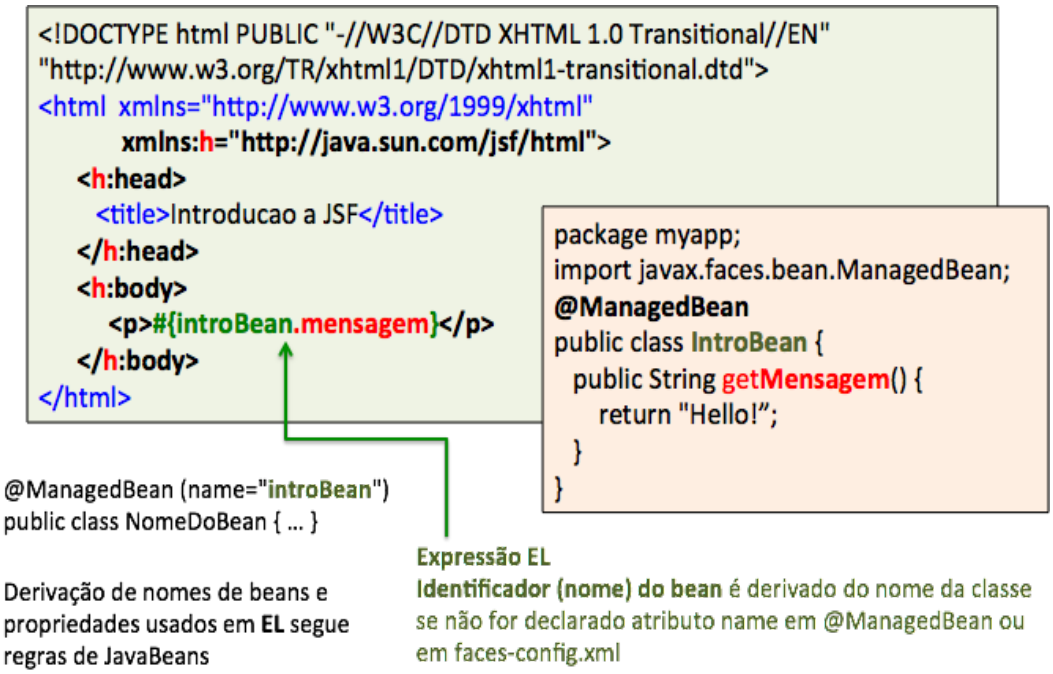
O managed bean obedece regras de formação de JavaBeans. Ao anotá-lo com *@Named* (ou *@ManagedBean*), é criado um *identificador* para o bean que pode ser usado em expressões EL, que por default é derivado do nome da classe. *Propriedades de leitura* são derivadas dos métodos *getter* (iniciados em "get" ou "is"). *Propriedades de gravação* são derivadas de métodos *setter* (iniciados em "set"). Nomes de beans e propriedades são derivadas removendo o *get/set/is* e convertendo o formato da primeira letra para minúscula. A conversão só ocorre se a segunda letra também for maiúscula. Por exemplo:

- `getNome` → `nome`
- `NomeDoBean` → `nomeDoBean`
- `getID` → `ID`
- `RESTBean` → `RESTBean`

A comunicação entre o componente e página é realizada através da expression language (EL), usada na página através dos marcadores *#{...}* ou *${...}*. A EL permite ler ou gravar propriedades do bean, transformar dados, executar expressões sobre os dados, usar resultados de operações e executar métodos no bean. Expressões EL geralmente são usadas em atributos de tags que as suportam, mas expressões de leitura de propriedades podem ser usadas diretamente na página.

- `#{identificadorDoBean.propriedade}`
- `#{identificadorDoBean.metodo}`
- `#{identificadorDoBean.prop1.prop2.prop3}`
- `#{identificadorDoBean.colecao[5].value + 20}`

A ilustração abaixo mostra o mapeamento entre um método de ação HTTP e um botão na View através de uma expressão EL:



1.2.4 Identificadores de navegação e métodos de ação HTTP

Alguns tags recebem o valor de retorno de métodos de ação HTTP (*action events*), que retornam strings que funcionam como identificadores com regras de navegação (indicam qual será a próxima View a ser mostrada). Exemplos típicos são tags usados para submissão de formulários (botões), tags de links e expressões que causam action events.

Os identificadores retornados podem ser mapeados a nomes de páginas no *faces-config.xml*. Na ausência de mapeamentos explícitos, o string retornado é interpretado por default como nome de uma página sem a extensão e contexto do mapeamento padrão, dentro do contexto da aplicação. Por exemplo:

- Se o mapeamento padrão do *FacesServlet* no *web.xml* for */faces/*.xhtml*, *"pagina"* refere-se à URL */faces/pagina.xhtml* dentro do contexto da aplicação.
- Se o mapeamento padrão for **.faces*, "pagina" refere-se a *pagina.faces*
- Se o mapeamento padrão for **.xhtml*, "pagina" refere-se a *pagina.xhtml*

No exemplo abaixo, em uma aplicação *jsf-intro.war* rodando em *localhost:8080* com mapeamento **.faces*, o método de ação *throwCoin()* retorna aleatoriamente ou a URL *http://localhost:8080/jsf-intro/coroa.faces* ou *http://localhost:8080/jsf-intro/cara.faces*:

```
@Named
@RequestScoped
public class CoinBean implements Serializable {
    public String throwCoin() {
       int coin = (int)(Math.random()*2);
```

```
        if(coin == 0) {
              return "coroa";
        } else {
              return "cara";
        }
    }
}
```

1.2.5 Uso de resources

Muitas aplicações contém *resources*, como arquivos de imagem, CSS, etc. que são parte essencial da aplicação. Resources podem ser colocados na pasta raiz e carregados normalmente do HTML através de URIs relativas se forem resources do HTML, ou na pasta classes se forem resources do Java. Mas o JSF recomenda que elas sejam colocadas em pastas especiais para que sejam carregadas pelo runtime JSF. As pastas são (relativas ao contexto):

- */META-INF/resources* – para resources que serão carregadas por arquivos Java (ex: arquivos **.properties* como resource bundles).
- */resources* – para resources carregadas pelas páginas Web.

Resources não são processados durante a requisição. Se referenciados em tags HTML, devem ser carregados informando o caminho completo.

Considere a seguinte árvore de resources Web:

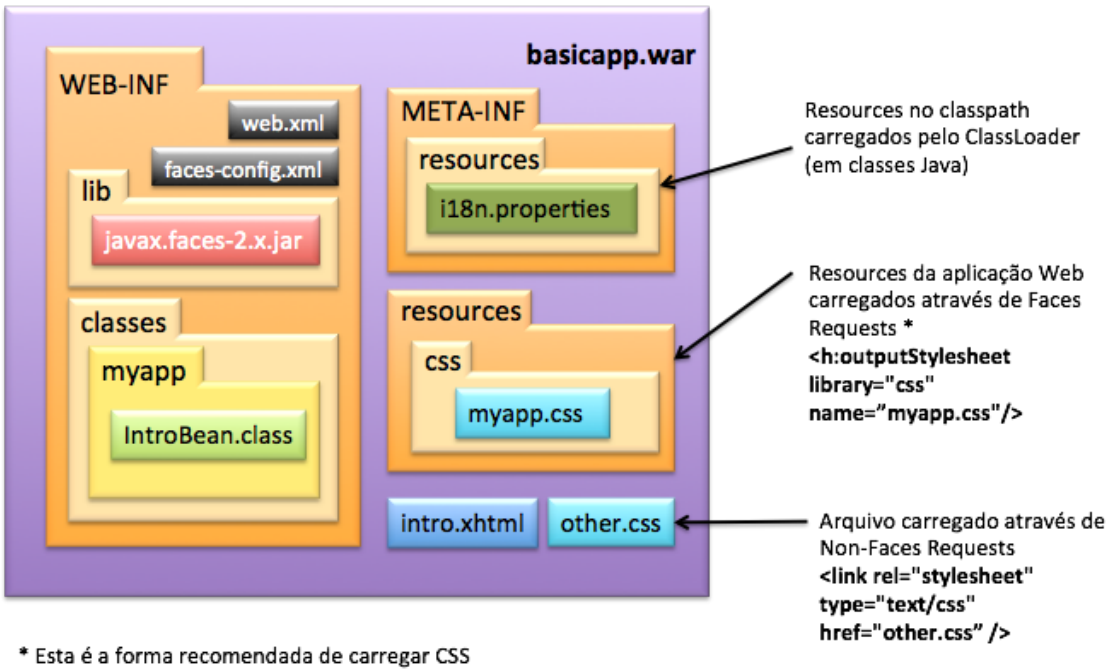
```
WEB-INF
resources/
    css/
        arquivo.css
    imagens/
        icone.jpg
```

Sem usar JSF, os resources podem ser carregados da seguinte forma, em *caminho relativo ao contexto da aplicação*:

```
<link rel="stylesheet" href="resources/css/arquivo.css"/>
<img src="resources/imagens/icone.jpg"/>
```

Se carregados via tags JSF, que é a forma recomendada, os resources devem ser referenciados usando o atributo "name" e caminho *relativo a essa pasta*.

```
<h:outputStylesheet name="css/arquivo.css"/>
<h:graphicImage name="imagens/icone.jpg"/>
```



JSF 2.2 ainda permite outras duas formas. Uma delas é usar as pastas de contexto como uma library (que atualmente é a melhor prática):

```
<h:outputStylesheet library="css" name="arquivo.css"/>
<h:graphicImage library="imagens" name="icone.jpg"/>
```

Pode-se também usar a seguinte sintaxe (que é compatível com URLs em CSS):

```
<h:outputStylesheet value="#{resource['css:arquivo.css']}"/>
<h:graphicImage value="#{resource['imagens:icone.jpg']}"/>
```

Exemplo de uso em CSS:

```
div {
    background-image: url( #{resource['imagens:icone.jpg']} );
}
```

Veja e explore os exemplos simples de aplicações JSF usando resources e beans no projeto *jsf-intro*.

2 Linguagem de expressões (EL)

Expression Language (EL) é uma linguagem simples que permite a comunicação entre as camadas de apresentação (view) e aplicação (beans). Fornece objetos implícitos, comandos, operadores para construir expressões para gerar valores, localizar componentes e chamar métodos, transformar e converter valores, definir e usar variáveis e constantes, gravar e recuperar dados, executar operações aritméticas, booleanas, de strings, etc.

Expressões podem ser usadas dentro de atributos e às vezes diretamente na página (quando geram dados de saída).

Há duas sintaxes possíveis e que podem produzir resultados diferentes em JSF. Elas determinam *quando* as expressões serão processadas.

A sintaxe:

```
${ ... expressão ... }
```

é usada para expressões que precisam ser executadas *imediatamente* (assim que a página é processada pela primeira vez). Esta é a forma normalmente usada em JSP. Funciona em JSF apenas em expressões *somente-leitura*.

O ciclo de vida do JSF envolve etapas distintas para leitura de propriedades, validação, conversão, gravação e geração da View (e será explorado em uma seção mais adiante.) Expressões ${} *não são afetadas* por esse ciclo de vida.

A sintaxe.

```
#{ ... expressão ... }
```

é usada para expressões que podem ter a execução adiada para uma *fase posterior* da geração da página. Esta é a forma mais usada e *recomendada* em aplicações onde a geração da resposta passa por um ciclo de vida de múltiplas fases como JSF. Expressões #{} podem ser usadas em expressões de leitura e gravação. A leitura e gravação normalmente ocorrerão em fases diferentes do ciclo de vida do JSF.

No campo abaixo:

```
<h:inputText id="name" value="#{produto.nome}" />
```

O valor de *#{produto.nome}* é *lido* em uma fase inicial do ciclo de vida. Mas uma fase posterior poderá preenchê-lo com um valor obtido em um request após a validação e antes de ser enviado para atualizar o campo mapeado no bean.

2.1 Method & value expressions

Existem dois tipos de expressões EL:

- *Method expressions* - expressões que chamam métodos em um bean
- *Value expressions* - expressões que referenciam valores.

E existem dois tipos de *Value expressions*:

- *Rvalue* - somente leitura
- *Lvalue* - leitura e gravação

As expressões *${}* são sempre *rvalue* mas expressões *#{}* podem ser *lvalue* ou *rvalue*.

Expressões que referenciam valores podem acessar componentes JavaBean, coleções e arrays, enums, objetos implícitos, além de propriedades ou atributos declarados no contexto de outros JavaBeans, coleções, arrays, enums e objetos.

A primeira variável de uma *value expression* é (geralmente) o nome de um bean. O bean deve ter sido previamente *registrado* e armazenado em um contexto (*Map*) com escopo limitado (ex: request, sessão, aplicação, página). Normalmente isto é feito de forma declarativa através de CDI.

A expressão:

```
#{produto}
```

irá procurar um componente registrado como "produto" nos escopos *page*, *request*, *session* e *application*.

Como foi mostrado na seção anterior, o registro pode ter sido feito através de tags no *faces-config.xml*, via anotação *@Named("produto")* (ou *@ManagedBean(name="produto")*) em uma classe com qualquer nome, ou ainda com a anotação *@Named* (ou *@ManagedBean*) em uma classe de nome Produto (ou ainda programaticamente através de inserção direta nos mapas do contexto do JavaServer Faces). O escopo default nos registros declarativos é *request*.

2.2 Como acessar as propriedades de um JavaBean

Normalmente beans são acessados por causa de suas propriedades. Elas podem ser acessadas via bean através do operador "."(ponto) ou através de "[...]" (colchetes). Normalmente, como convenção, o ponto "." é usado para acessar propriedades enquanto que colchetes "[...]" são normalmente usados para acessar coleções, mapas e arrays, ou ainda quando é necessário referenciar uma propriedade usando um literal String. As duas formas abaixo são equivalentes:

```
#{produto.nome}
#{produto["nome"]}
```

Para acessar item de um array ou lista, é preciso usar o *índice* (ou número que contenha valor que seja traduzido em inteiro), por exemplo:

```
#{item.itens[5]}
```

Um *Map* pode ser acessado através de sua chave:

```
#{banco.contas["3928-X"]}
```

As propriedades, itens de coleções e mapas podem ser *rvalues* ou *lvalues*. A forma como serão tratadas (leitura ou gravação) depende de onde são usadas. A configuração sobre os tipos de expressões que podem ser recebidos em cada atributo é feita na definição do tag. Os tags das bibliotecas padrão informam os tipos aceitos na sua documentação.

Expressões *rvalue* podem ser usadas em atributos configurados para *receber* valores e diretamente no texto da página. Expressões *lvalue* podem ser usadas em atributos configurados para *receber e enviar* valores.

Expressões usando *valores literais* sem referência a um bean também podem ser usadas para construir expressões *rvalue*:

- `#{false}`
- `#{123}`
- `#{produto.preco + 12.5}`
- `#{"Texto"}`

2.3 Sintaxe da EL

EL possui 16 palavras reservadas:

- `and, or, not`
- `eq, ne, lt, gt, le, ge`
- `true, false`
- `null`
- `instanceof,`
- `empty`
- `div, mod`

Os literais do JSF são

- `true` e `false` (boolean)
- números (inteiros e ponto flutuante)
- strings entre aspas ou apóstrofes (escapes \″, \′ e \\)
- `null`

Escapes também são necessários quando uma expressão literal usada em atributos precisa imprimir as sequencias *${* ou *#{*. Use

- `\${` ou `\#{`
- `${'${'}` ou `#{'#{'}`

Atributos que recebem expressões também podem receber expressões *literais* fixas que são expressas em texto sem os *#{}*:

```
<h:inputText value="texto" />
```

Expressões compostas são concatenadas e executadas da *esquerda para a direita* (convertendo, em cada passo, o resultado em string). O String final é convertido no tipo esperado pelo atributo:

```
<h:inputText value="${produto.codigo or produto.nome}" />
```

Expressões de método permitem a chamada de métodos públicos de um bean (que geralmente retornam um valor). É possível chamar métodos com ou sem parâmetros.

A sintaxe para chamar métodos deve sempre ser *#{}* uma vez que podem ser executados em fases diferentes do ciclo de vida.

Métodos são chamados através de atributos configurados para executar ações ou receber os valores retornados. Podem ser chamadas a validadores, métodos de ação (navegação), listeners de eventos, etc. O exemplo abaixo ilustra métodos que serão chamados em fases diferentes:

```
<h:form>
        <h:inputText id="codigo"
                    value="#{device.codigo}"
            validator="#{device.validarCodigo}"/> <!-- fase validação -->
        <h:commandButton id="sincronizar"
            action="#{device.sincronizar}" /> <!-- fase aplicação -->
</h:form>
```

Métodos de ação HTTP (navegação) são geralmente chamados *sem* parâmetro, mas qualquer método pode ser parametrizado. Eles devem passar a lista de parâmetros (separados por vírgula) entre parênteses como normalmente são chamados em Java:

```
<h:inputText value="#{produtoBean.porCodigo('100')}">
```

Vários operadores podem ser usados para formar expressões aritméticas e booleanas. São todas do tipo *rvalue*.

- Aritméticos *(+,-, *, /, %, mod, div)*
- Lógicos *(and, or, not, &&, ||, !)*
- Relacionais *(==, eq, !=, ne, <, lt, >, gt, <=, ge, >=, le)*
- *empty* (testa se é *null* ou vazio)
- A ? B : C (condicional).

Se combinados os operadores seguem regras de precedência similares às de Java que podem ser alteradas com parênteses.

Veja no projeto *jsf-el* vários exemplos de expressões EL e explore os resultados.

2.4 Funçoes JSTL

O JSTL (JSP Standard Tag Library) contém uma coleção de tags e funções reutilizáveis que podem ser usadas em JSF. Geralmente operações sobre arrays e strings devem ser feitas em Java nos managed beans e terem apenas seus resultados transferidos para a View. Mas há ocasiões em que é justificável usar operações na View, para, por exemplo, verificar se um string contem um substring, contar o número de caracteres ou itens de um array, ou ainda fazer transformações mínimas. Para isto, pode-se usar as funções JSTL.

Para usar essas funções é preciso declarar o namespace das funções JSTL e definir um prefixo:

```
<html ...
      xmlns:fn="http://java.sun.com/jsp/jstl/functions">
```

Depois as funções podem ser usadas dentro de expressões EL:

```
Número de filmes: #{fn:length(cinemateca.filmes)}
```

A tabela abaixo relaciona as funções disponíveis:

`fn:contains(s1, s2)`	Retorna true se string s1 contem o substring s2.
`fn:containsIgnoreCase(s1, s2)`	Mesmo que *fn:contains()*, com ignorando formato mauúscula/minúscula.
`fn:endsWith(s1, s2)`	Retorna true se string s1 termina com s2.
`fn:escapeXml(s)`	Converte caracteres especiais do XML em escapes <, > etc.
`fn:indexOf(s1, s2)`	Retorna posição da ocorrência de s2 dentro de s2, ou -1 se não ocorrer.
`fn:join(a[], delim)`	Concatena os elementos do array com o delimitador e retorna um string.
`fn:length(obj)`	Retorna número de elementos de uma coleção ou número de caracteres de um string.
`fn:replace(s1, s2, s3)`	Retorna um string no qual todas as ocorrências de s2 ocorridas dentro de s1 são trocadas por s3.
`fn:split(s, delim)`	Divide o string s pelo delimitador e devolve um array
`fn:startsWith(s1, s2)`	Retorna true se o string s1 começar com s2.

`fn:substring(s1, inicio, fim)`	Retorna o substring entre as posições inicio (inclusive) até antes da posição fim.
`fn:substringAfter(s1, s2)`	Retorna o substring depois da ocorrência de s2, no string s1.
`fn:substringBefore(s1, s2)`	Retorna o substring antes da ocorrência de s2, no string s1.
`fn:toLowerCase(s)`	Retorna o string em caixa baixa.
`fn:toUpperCase(s)`	Retorna o string em caixa alta.
`fn:trim(s)`	Remove os espaços em branco (tabulações, espaços, quebras de linha) antes e depois do string.

3 Arquitetura e ciclo de vida

O JavaServer faces possui uma arquitetura complexa, que envolve APIs Java, bibliotecas de componentes em Java, bibliotecas de tags em XHTML, diversos modelos abstratos e um mecanismo de tempo de execução baseado em um ciclo de vida com fases e responsabilidades bem definidas.

3.1 Principais APIs

A maior parte das APIs Java usadas para construir aplicações JSF 2.2 estão no pacote *javax.faces* e subpacotes nas distribuições do Java EE desde a versão 6.0. A seguir uma lista de alguns desses sub-pacotes e uma breve descrição das classes e interfaces ou funcionalidades mais importantes.

- **javax.faces.component**: contém uma hierarquia de classes baseada em UIComponent, que compõe o modelo de componentes abstrato do JSF, representando componentes gráficos genéricos. O subpacote *javax.faces.component.html* baseia-se nesse modelo abstrato e define um modelo de componentes para renderização de HTML.
- **javax.faces.application:** coleção de classes utilitárias que ajudam a integrar objetos relacionados à logica de negócios de uma aplicação ao runtime do JSF (FacesMessage, Resource, Application).
- **javax.faces.model:** API que contém classes que representam modelos para estruturas de dados usados em JSF, como arrays, tabelas, listas, resultados e itens de menu.
- **javax.faces.context**: API para acesso ao estado da requisição (via FacesContext); permite acesso a vários contextos e objetos criados durante a execução do JSF. As classes deste pacote permitem que beans e classes Java tenham acesso a componentes Faces, componentes HTML, runtime e contexto do EL, objetos de escopo, etc.

- **javax.faces.convert**: API para conversores de dados (interface Converter). Contém interfaces para a construção de conversores e várias implementações nativas.
- **javax.faces.event**: API para eventos (FacesEvent, FacesListener). Contém interfaces para a construção de listeners e classes de eventos.
- **javax.faces.render**: API para renderização gráfica (RenderKit). Permite a construção de componentes customizados e extensões ao kit de renderização HTML que é default.
- **javax.faces.validator**: API de validação. Contém interfaces para a criação de validadores e diversas implementações nativas.
- **javax.faces.flow** – API do Faces Flow, que define um escopo para tarefas que envolvem várias páginas, com um ponto de entrada e um ponto de saída.
- ~~**javax.faces.bean**~~: Contém as anotações para managed beans (@ManagedBean, @SessionScoped, etc.) que devem ser *deprecadas* em um futuro próximo, portanto, não devem mais ser usadas. O mesmo comportamento pode ser obtido usando CDI.

3.2 Componentes

O Java Server Faces define diversos componentes que são abstrações de objetos que podem ter seus dados renderizados graficamente. São classes que representam componentes de interface gráfica (UI) e estendem a classe abstrata *UIComponent*. Cada classe de *UIComponent* representa um tipo específico de componente e descreve seu estado e comportamento.

Também compõem a arquitetura do JSF quatro outros modelos abstratos: renderização, conversão, eventos e validação.

3.2.1 Modelo de componentes UI

O pacote *javax.faces.component* contém a hierarquia fundamental dos componentes UI que descreve a sua estrutura e comportamento independente da interface. Em geral, programadores de aplicações JSF não instanciam componentes diretamente, mas os utilizam indiretamente através de tags. As classes, porém, são importantes para construir compontentes programaticamente e para realizar *binding*, mapeando objetos Java a facelets para poder alterar suas propriedades via código Java.

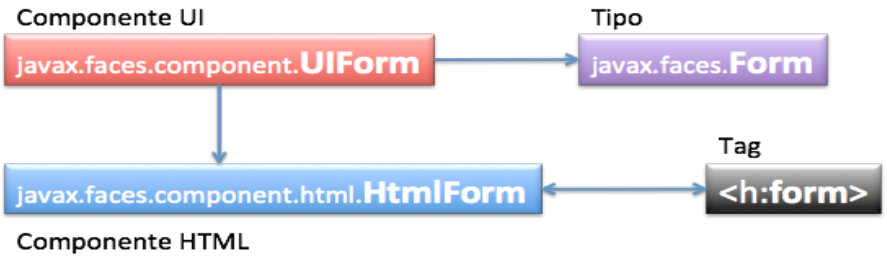
As principais classes de *javax.faces.component* estão relacionadas na tabela abaixo.

UIComponentBase	Superclasse concreta de todos os componentes UI (implementa todos os métodos abstratos de UIComponent). Define estado e comportamento default para todos os componentes.

`UIData`	Mapeamento de dados a uma coleção de dados representados por um DataModel (em HTML, representa uma tabela)
`UIColumn`	Coluna de um componente UIData (representa uma coluna de uma tabela HTML)
`UICommand`	Controle que dispara eventos de ação (em HTML, representa um botão, um iem de menu ou um link)
`UIForm`	Formulário de entrada de dados. Componentes contidos no formulário representam os campos de entrada de dados enviados quando o campo é submetido. Representa um elemento <form> em HTML
`UIGraphic`	Representa uma imagem.
`UIOutput`	Representa dados de saída (somente leitura) em uma View.
`UIInput`	Representa dados de entrada. É subclasse de UIOutput. Em HTML tipicamente representa elementos <input> e <textarea>.
`UIMessage, UIMessages`	UIIMessage representa mensagens (geralmente de erro) associadas a um componente UIComponent. UIMessages obtém mensagens do contexto inteiro.
`UIOutcomeTarget`	Em HTML, representa um link como link (a href) ou botão
`UIPanel`	Componente para controlar o layout de componentes filho (qualquer UIComponent)
`UIParameter`	Representa parâmetros para um elemento pai.
`UISelectBoolean`	Subclasse de UIInput. Permite seleção de valor booleano em um controle. Em HTML representa um checkbox ou radio button individual.
`UISelectItem; UISelectItems`	UISelectItem representa um ítem individual dentro de um conjunto de itens; UISelectItens representa o conjunto de itens como uma coleção. Em HTML representam coleções contendo opções <option> de um <select>, ou os checkboxes/radio buttons individuais pertencentes a um mesmo grupo.
`UISelectMany`	Permite que um usuário selecione vários itens de um grupo. Em HTML representa um elemento <select> com opção de seleção múltipla, ou uma coleção de checkboxes de um mesmo grupo.

UISelectOne	Permite selecionar um item de um grupo. Em HTML representa um elemento <select> ou uma coleção de radio buttons de um mesmo grupo.
UIViewParameter	Representa um parâmetro de um request HTTP. Possui métodos para decodificar e extrair nome e valor.
UIViewRoot	Representa a raiz da árvore de componentes, através do qual pode-se navegar e obter todos os elementos-filho. Em DOM/HTML representa o elemento document/<body>.

Os componentes *javax.faces.component* são associados a elementos HTML através de um kit de renderização HTML, que combina cada *tipo* com uma implementação de *UIComponent* no pacote *javax.faces.component.html*, mapeando os componentes a tags XHTML. Este mapeamento é mostrado ilustrado para o tipo *Form*, abaixo:



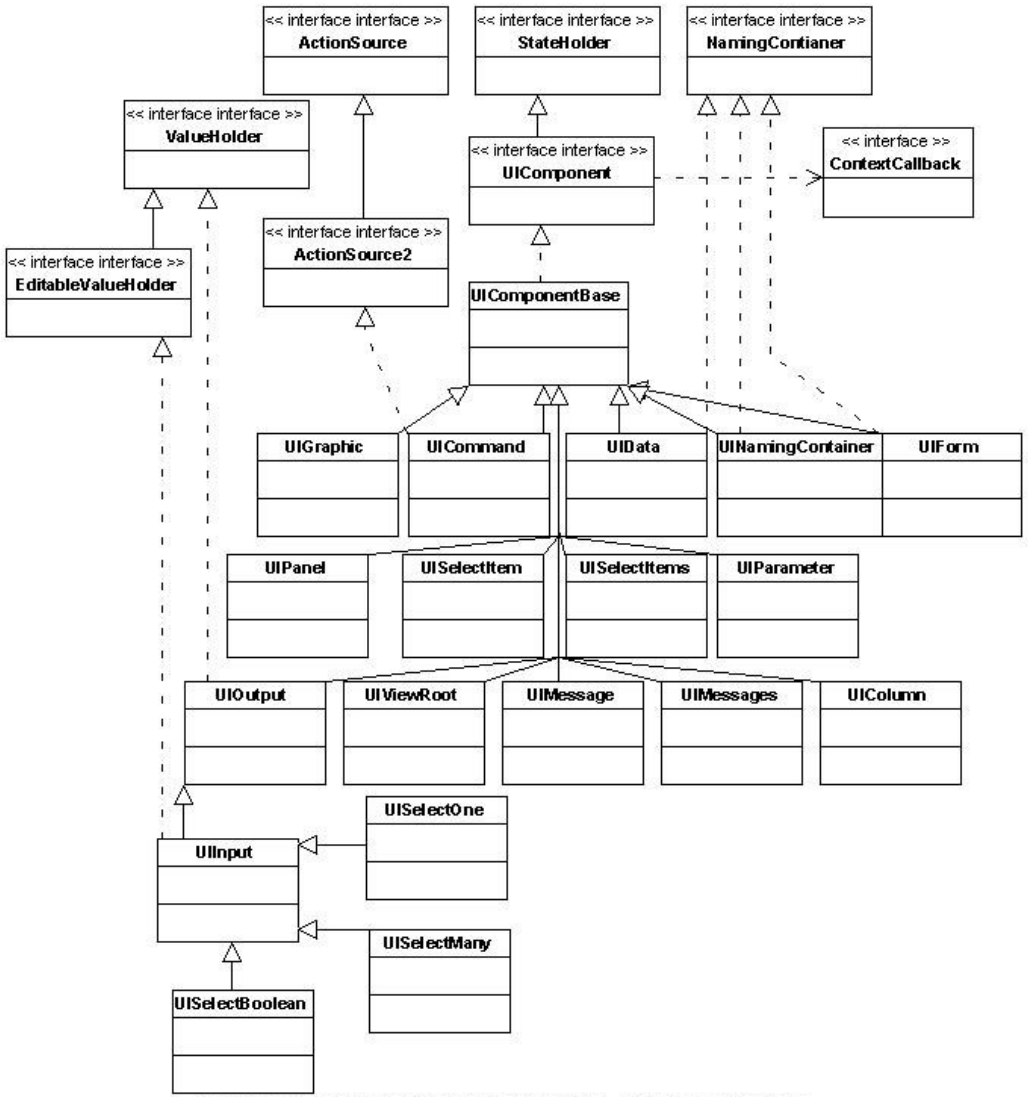
3.2.2 Interfaces de comportamento

Além das classes de componentes, o pacote javax.faces.component contém uma coleção de *interfaces de comportamento* que estão associados aos componentes e que mapeiam diferentes tipos de valores e ações. A tabela abaixo lista algumas dessas interfaces.

ActionSource2	Permite que o componente possa disparar evento de ação HTTP (evento action). Links e botões (UICommand) implementam esta interface.
ValueHolder	Permite que o componente possa manter um valor local e acessar dados na camada do modelo de dados através de uma expressão de valor. A maior parte dos elementos de texto e de formulário (UIOutput) implementam esta interface.
EditableValueHolder	Estende ValueHolder e especifica recursos adicionais para componentes editáveis. Implementado por elementos de entrada de dados (UIInput).
StateHolder	Indica que o componente possui estado que precisa ser preservado entre requisições. Implementado por todos os componentes UIComponent.
NamingContainer	Obriga ID único a componente que implementa essa interface. Todos os

	componentes (UIComponent) implementam esta interface e definem um Client ID, que é formado pelo ID do elemento pai, seguido por um separador (JSF usa ":" por default) e o ID local do componente.

A ilustração abaixo (extraída da especificação JSF) ilustra a hierarquia completa dos componentes UI do JSF e sua relação com as interfaces de comportamento.



3.2.3 Modelo de eventos

Há seis tipos de eventos suportados por componentes JSF 2.2:

1. **Ação** HTTP (*javax.faces.event.ActionEvent*) gerados por *UIComponent*
2. **Mudança de valor** (*javax.faces.event.ValueChangeEvent*) gerados por *UIComponent*
3. Alteração no **modelo de dados** (*javax.faces.model.DataModelEvent*) por *UIData*
4. **Fase do ciclo de vida** (*javax.faces.event.PhaseEvent*) gerados por *FacesContext*
5. Comportamento **Ajax** (*javax.faces.event.AjaxBehaviorEvent*) gerados por Ajax

6. **Sistema** (Subclasses de *javax.faces.event.SystemEvent*) gerados por outros objetos.

Além desses eventos, aplicações JSF também podem capturar outros eventos lançados em aplicações Java, como por exemplo *java.beans.PropertyChangeEvent* para detectar mudanças em propriedades de beans.

Todos os eventos possuem interfaces listener que podem ser implementados para capturá-los. Em seguida devem ser registrados. Os eventos mais importantes são os eventos gerados por UIComponent: *ActionEvent* e *ValueChangeEvent*, que são disparados por todos os componentes. Eles podem ser registrados através de facelets.

Há duas formas de registrar eventos de *UIComponent*:

- Pode-se *implementar uma classe* do event listener correspondente e registrá-lo no componente (aninhando um *<f:valueChangeListener>* ou *<f:actionListener>* dentro do tag), ou
- Pode-se *criar um método* em um managed bean vinculá-lo através de uma expressão EL em atributo compatível com expressões de método (ex: atributo *listener*, *actionListener*, *valueChangeListener*).

Maiores detalhes sobre registro e captura de eventos de UIComponent serão explorados em outras seções deste tutorial.

3.2.4 Modelo de renderização

Um *Render Kit* define de que forma as classes de componentes serão mapeadas a tags para um determinado tipo de cliente. Atualmente o único Render Kit incluído no JSF serve para renderização HTML.

Mas mesmo em HTML há benefícios no modelo e reuso. Renderizações de um mesmo componente podem produzir aparências diferentes. Por exemplo a classe *UISelectOne* pode ser renderizada como um grupo de opções (rádio button) *<input type="radio">*, um combo, lista ou menu pull-down *(<select><option/>...</select>*). Já um *UICommand* pode ser renderizado como botão ou link*(<a href>*, *<input type="submit>*).

O modelo de renderização é usado nas fases de restauração da view (1) para construir uma árvore de componentes a partir de um documento XHTML com raiz em UIRootView, e na fase de renderização da resposta (6) onde o UIRootView, depois de ser atualizado é usado para gerar o HTML da página da resposta.

Em versões anteriores a JSF 2.0 usava-se o render kit para construir componentes reutilizáveis. Desde JSF 2.0 há uma API declarativa muito mais simples para criar bibliotecas de tags e componentes, portanto neste tutorial não exploraremos a API de renderização.

3.2.5 Modelo de conversão

Quando um componente é mapeado a um objeto, a aplicação passa a ter *duas visões dos dados*. A visão do modelo (*Model View*) onde os dados são representados por *tipos* (int, long, objetos), e a visão de apresentação (*Presentation View*), onde os dados são apresentados em formado legível (ex: String).

Por exemplo, um objeto *Produto* pode ser um objeto com *preço*, *nome*, *descrição*, *código*, *fornecedor*, etc. (*Model*) mas aparecer em uma combo representado apenas pelo seu nome ou código (*Presentation*). Quando um usuário selecionar o produto ou código, a seleção precisará ser convertida de volta em um objeto *Produto*. O modelo de conversão do JSF pode cuidar dessa conversão transparentemente.

A conversão para tipos primitivos e tipos básicos do Java (como BigDecimal) é automática. Para converter outros tipos é preciso implementar e registrar um Converter. Tipos comuns, como Date, Number, moeda, etc. têm conversores nativos e internacionalizados que podem ser reusados, e a API permite criar conversores customizados através da implementação de uma interface.

A conversão de dados ocorre em duas fases do ciclo de vida do JSF. Na fase de processamento de validação (3) ocorre a conversão dos dados para produzir uma representação em forma de objeto, que possa ser incluída na árvore que será processada. Na fase de renderização da resposta (6) o conversor é usado para converter o objeto em uma representação String.

3.2.6 Modelo de validação

A validação dos dados em uma fase específica (fase 3 - processamento de validação) *antes* do modelo de dados ser atualizado. A validação pode ter vários níveis de detalhamento. Pode ser apenas uma validação de campo obrigatório (configurada simplesmente incluindo um atributo *required="true"* no componente), ou um algoritmo mais complexo que envolve várias etapas.

Pode-se usar validadores existentes, para ou criar validadores customizados. Os existentes geralmente suportam testes de limites (em números, listas), comprimentos (strings), valor

nulo e expressões regulares (que permitem uma validação muito detalhada). É possível também usar Bean Validation, que fornece validação declarativa (nativa em CDI).

Validadores customizados, para tarefas mais complexas (ex: validação com XML Schema) podem ser criados ou através da implementação da interface *Validator*, ou através da criação de um método de validação no próprio bean, que será identificado como validador ao ser chamado em um atributo.

3.2.7 Modelo de navegação

A navegação consiste de regras usadas para escolher a próxima pagina a ser mostrada (*outcome*). Pode ser implícita – quando regras de navegação não são configuradas e o sistema usa *defaults*, ou explícita – quando regras são declaradas em *faces-config.xml*.

O *NavigationHandler* é responsável por localizar a próxima view a ser exibida com base nas regras declaradas. Se nenhuma regra combinar com o resultado*, a mesma View é exibida*. Geralmente um processamento terminará com a chamada de um método de ação HTTP (no envio do formulário) que devolve um string. O string devolvido produz um identificador implícitos de navegação, que será usado para localizar a URL da próxima *View* a ser exibida. Retornar *null* é a forma padrão de provocar a recarga da mesma página.

Saber a próxima View a ser exibida é necessário para renderizar uma resposta de uma nova View, portanto o processamento da ação é realizada na fase de execução da aplicação (5).

3.3 Tipos de requisições JSF

Uma aplicação JSF distingue as requisições HTTP que são direcionadas a aplicações JSF e respostas produzidas por aplicações JSF. Essas requisições e respostas são chamadas de *Faces Requests* e *Faces Responses*. Elas são interceptadas pelo *Faces Runtime* e tratadas de forma diferenciada. O Faces Runtime processa dois tipos de Faces Request/Response:

- Faces *Resource* Request/Response (para transferir imagens, CSS, etc.)
- Faces Request/Response (para processar uma página JSF)

Uma requisição HTTP envolvendo JSF pode incluir, portanto, *seis* diferentes tipos de requisições:

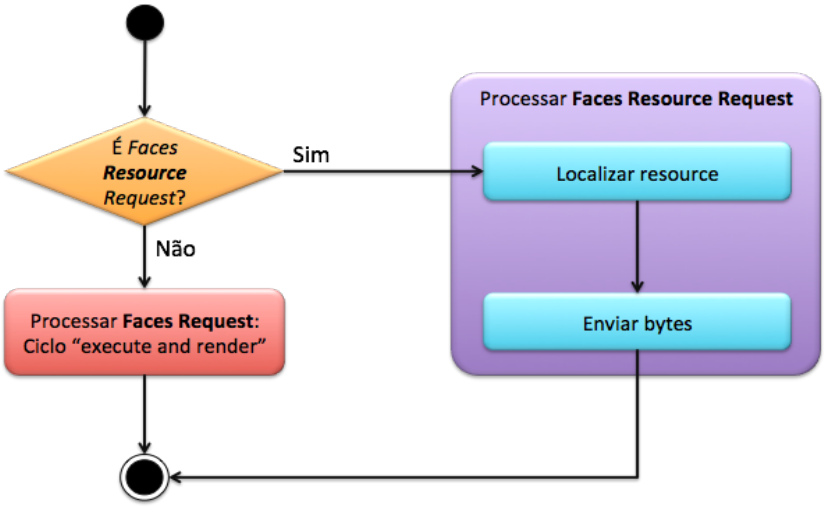
- Faces Response (FRs): Resposta criada pela execução da fase Render Response
- Faces Request (FRq): Requsição iniciada a partir de uma FRs prévia
- Faces Resource Request (FRRq): um FRq para um resource (imagem, CSS, etc.)
- Non-Faces Request (NFRq): Requisição não iniciada a partir de FRs prévia

- Non-Faces Response (NFRs): Resposta que não passou pelo Faces Runtime, e
- Faces Resource Response (FRRs): um NFRs para um resource iniciado por um FRRq

Cenários *relevantes* para JSF são as respostas iniciadas por um FRq e a requisição inicial (que é um NFRq). São, portanto, três os cenários relevantes:

- NFRq gerando FRs (requisição inicial)
- FRq gerando FRs (requisição que causa ciclo execute & render)
- FRq (FRRq) gerando NFRs (FRRs)

Apenas um dos Faces request participa do ciclo de vida que é responsável pela renderização gráfica da página. A ilustração abaixo mostra como o Faces Runtime distingue os dois.



3.4 Ciclo de vida do JSF

O ciclo de vida (chamado de *Execute & Render*) processa todos os Faces Requests e realiza o tratamento de conversão, eventos, renderização, etc. em fases distintas da requisição. Existem *seis* fases. Os nomes abaixo em maiúsculas são as constantes *default* da classe *PhaseId*:

1. **RESTORE_VIEW** (restaurar a view)– na primeira requisição não faz nada, nas requisições seguintes, recupera o UIViewRoot com o estado da View armazenado no FacesContext.
2. **APPLY_REQUEST_VALUES** (aplicar valores da requisição) – copia os valores da requisição de cada componente para um local temporário (submittedValue).
3. **PROCESS_VALIDATIONS** (processar validações) – processa validações e conversões, e se obtiver sucesso, copia os valores do local temporário para o componente (value).

4. **UPDATE_MODEL_VALUES** (atualizar valores do modelo) – copia os valores validados e convertidos para o modelo (managed beans)
5. **INVOKE_APPLICATION** (chamar aplicação) – executa os métodos de ação HTTP e escolhe a próxima View.
6. **RENDER_RESPONSE** (renderizar a resposta) - constrói um UIViewRoot com os componentes atualizados e gera o HTML da nova View.

Suponha que a View seja o XHTML seguinte:

```
<h:form id="formulario">

    <h:panelGrid columns="3">
        <h:outputText value="Mensagem" />
        <h:inputText value="#{mensagemBean.mensagem}" id="msg" required="true">
            <f:converter converterId="mensagemConverter" />
           <f:validator validatorId="mensagemValidator" />
            <f:valueChangeListener type="br...MensagemValueChangeListener"/>
        </h:inputText>
       <h:message for="msg" style="color: red"/>
    </h:panelGrid>
    <h:commandButton action="#{mensagemBean.processar}" value="Processar" />
    <h:commandButton action="index" value="Cancelar" immediate="true"/>

</h:form>
```

Este é o managed bean usado:

```
@Named
@SessionScoped
public class MensagemBean implements Serializable {

    private Mensagem mensagem;

    public Mensagem getMensagem() {
        return mensagem;
    }

    public void setMensagem(Mensagem mensagem) {
        this.mensagem = mensagem;
    }

    public String processarMensagem() {
       // processar a mensagem
        return "resultado";
    }

}
```

Na primeira requisição, para mostrar a view, haverá apenas duas fases, e nada acontece de especial na primeira fase, já que não há view ainda para ser restaurada. As fases são:

1. RESTORE_VIEW
6. RENDER_RESPONSE: gera uma árvore de componentes UIViewRoot, que contém um UIForm contendo um UIInput (campo de entrada) e UICommand (botão).

Com a página mostrada no browser, digitamos um texto qualquer (ex: "teste") no campo de entrada e apertamos o botão. Desta vez as fases serão:

1. RESTORE_VIEW: os dados do formulário são usados para reconstruir a *UIViewRoot* . Os valores do *UIInput* (que está mapeado ao campo de entrada que contém o texto "teste") ainda não recebeu nada. O valor armazenado no objeto Mensagem no bean é *null*.
2. APPLY_REQUEST_VALUES: O texto digitado é usado para atualizar o valor do método setSubmittedValue() do UIInput da árvore de componentes. O valor armazenado no objeto Mensagem no bean ainda é *null*.
3. PROCESS_VALIDATIONS: O texto passa pelo conversor (método getAsObject) e pelo validador. Como o texto digitado não causa erro de validação, ele é transferido para o *setValue()* do *UIInput*; como o valor novo ("teste") é diferente do anterior (null), um evento *ValueChange* é disparado. O valor armazenado no objeto Mensagem no bean ainda é *null*.
4. UPDATE_MODEL_VALUES: os valores válidos e convertidos da árvore de componentes são finalmente usados para atualizar os atributos do managed bean (O valor armazenado no objeto Mensagem no bean finalmente recebe o valor que estava em *UIInput.getValue()*).
5. INVOKE_APPLICATION: Agora que o bean está atualizado, seus dados podem ser usados por outras aplicações. O método processarMensagem() configurado como *action* do formulário é executado, e a View de resposta é selecionada.
6. RENDER_RESPONSE: Os dados a serem exibidos são obtidos do bean (getMensagem()), convertidos em string (chama getAsString() no conversor) e usados para montar a árvore de componentes UIViewRoot que irá gerar o HTML da resposta.

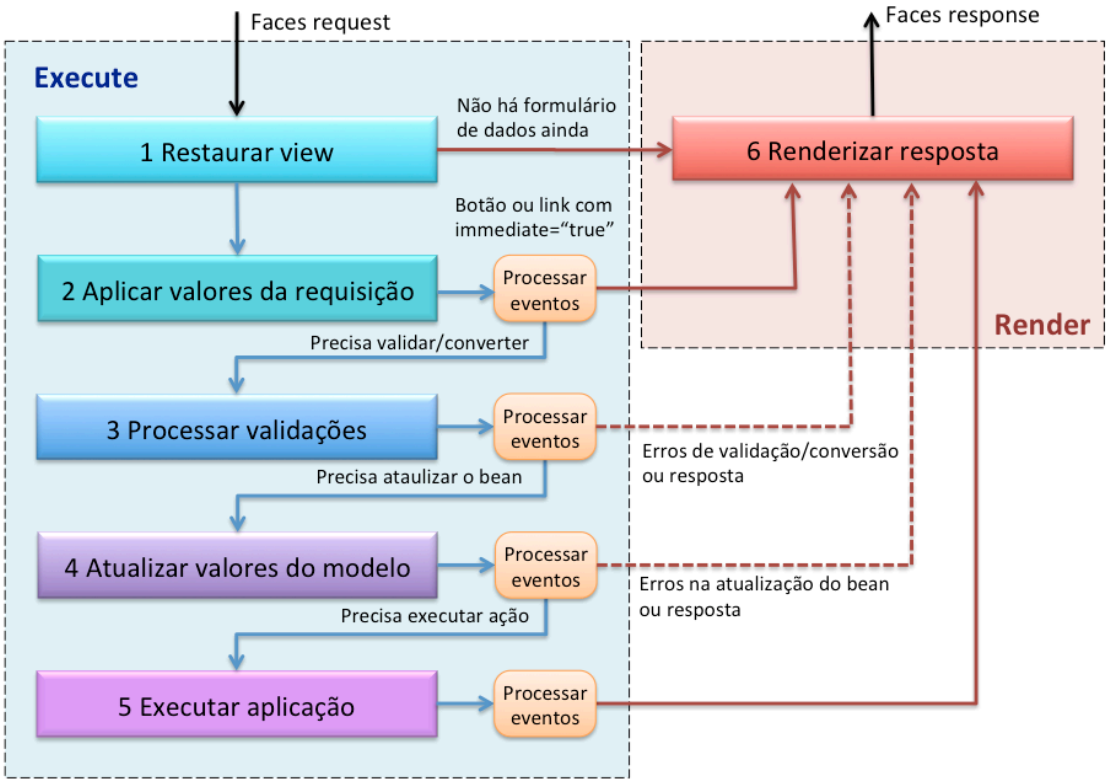
Se houver erros de validação ou de conversão (na etapa 3), *as etapas 4 e 5 serão puladas*, e o controle pula para a etapa 6, onde a resposta será renderizada, possivelmente com dados de mensagem (FacesMessage) de erro incluídas na etapa 3 que serão exibidas no lugar do componente <h:message>. Portanto as etapas executadas serão:

1. RESTORE_VIEW
2. APPLY_REQUEST_VALUES

3. PROCESS_VALIDATIONS: Neste ponto, ocorre um erro de conversão ou validação. Uma exceção é lançada e FacesMessage são criadas. O controle pula para a última etapa. *UIInput.getValue()* continua *null*, e *UIInput.getSubmittedValue()* continua com o valor enviado.
4. RENDER_RESPONSE: O formulário é exibido com uma FacesMessage contendo a mensagem de erro.

Expressões EL são executadas *imediatamente* (na etapa 2) se forem do tipo ${}. Se forem do tipo #{} a execução ocorre em várias outras fases. O uso do atributo *immediate="true"* pode fazer o JSF pular etapas (se for usado em um botão ou link) ou priorizar o processamento (se for usado em um componente que guarda valor). No exemplo mostrado, o botão Cancelar tem o atributo immediate="true". Apertá-lo irá fazer o controle pular as etapas 3, 4 e 5.

A imagem abaixo ilustra o ciclo de vida completo do JSF (sem levar em conta uso do atributo *immediate* em componentes de entrada de dados), mostrando como o fluxo da requisição e da resposta passa por todas as seis fases.



3.4.1 Monitoração do ciclo de vida

A depuração do ciclo de vida de uma aplicação JSF pode ser feita com um *PhaseListener*, que captura os eventos do Faces Runtime e permite executar código antes e depois de cada

fase. Rode os exemplos do projeto *jsf-architecture* que monitora os eventos de fase. Um exemplo de PhaseListener é mostrado no capítulo sobre listeners.

4 Facelets e componentes HTML

4.1 O que são Facelets?

JSF fornece uma arquitetura que mapeia tags, componentes e renderizadores de interface do usuário. Os tags usados em aplicações JSF compõem uma linguagem de declaração de páginas (*View Declaration Language*) através da qual se pode montar árvores de componentes declarativamente. Essa árvore é posteriormente renderizada e transformada em uma tela ou página Web.

Esta linguagem é chamada de *Facelets*. Trata-se de uma aplicação do XML similar ao XHTML que serve de template para gerar páginas XHTML dinamicamente, e que permite combinar tags nativos do XHTML com tags de outras bibliotecas, prefixados, agrupados em coleções identificadas por um *namespace*. Alguns tags representam componentes gráficos em que são renderizados em HTML, outros representam transformações que geram código dinamicamente (loops, condicionais) e outras encapsulam código JavaScript e cabeçalhos HTTP. É uma linguagem extensível. Pode-se ainda criar facelets para renderizar outras coisas como JavaScript, SVG, XML, etc.

O código abaixo, similar aos exemplos mostrados e executados na seção anterior, é um documento XHTML usando facelets que geram elementos HTML. Observe os diferentes prefixos usados e os atributos contendo expressões EL:

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

<h:head></h:head>
<body>
<h:form>
    <h1>Exemplos</h1>
    <h:panelGrid columns="2" columnClasses="pre-col1,col2,col3">
        <h:selectOneListbox id="sol" value="#{bean.filme}">
            <f:selectItems value="#{cinemateca.filmes}" var="filme"
                           itemLabel="#{filme.titulo}" itemValue="#{filme.imdb}" />
        </h:selectOneListbox>
        <h:message for="sol" />
    </h:panelGrid>
    <h:commandButton value="Enviar" action="#{bean.processar}" />
</h:form>
```

```
</body>
</html>

</html>
```

Este é o código-fonte do HTML final gerado no browser:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="j_idt2"></head>
<body>
<form id="j_idt4" name="j_idt4"
      method="post"
      action="/jsf-facelets/facelets-example.faces"
      enctype="application/x-www-form-urlencoded">

    <input type="hidden" name="j_idt4" value="j_idt4" />
    <h1>Exemplos</h1><table>
    <tbody>
    <tr>
        <td class="pre-col1"><select id="j_idt4:sol" name="j_idt4:sol" size="6">
            <option value="tt0062622">2001: A Space Odyssey</option>
            <option value="tt0013442">Nosferatu, eine Symphonie des Grauens</option>
            <option value="tt1937390">Nymphomaniac</option>
            <option value="tt1527186">Melancolia</option>
            <option value="tt0113083">La Flor de mi Secreto</option>
            <option value="tt0101765">La double vie de Véronique</option>
        </select></td>
        <td class="col2"></td>
    </tr>
    </tbody></table>
    <input type="submit" name="j_idt4:j_idt13" value="Enviar" />
    <input type="hidden" name="javax.faces.ViewState"
           id="j_id1:javax.faces.ViewState:0"
           value="2302106755987392480:3479051282761317321" autocomplete="off" />
</form>
</body>
```

Através de facelets é possível executar expressões (na linguagem *EL - Expression Language*) para gerar strings, números e outros valores, transferir dados entre componentes e a página, realizar mapeamento (*binding*) entre tags, páginas e componentes, construir e reusar componentes. As principais bibliotecas de tags nativamente suportadas e seus namespaces com os prefixos recomendados estão listadas abaixo:

- Templating: *xmlns:**ui**="http://java.sun.com/jsf/facelets"*
  Exemplos: ui:component, ui:insert, ui:repeat
- HTML: *xmlns:**h**="http://java.sun.com/jsf/html"*
  Exemplos: h:head, h:body, h:outputText, h:inputText

- Core library JSTL1.2: *xmlns:**c**="http://java.sun.com/jsp/jstl/core"*
  Exemplos: c:forEach, c:catch
- Core library do JSF: *xmlns:**f**="http://xmlns.jcp.org/jsf/core"*
  Exemplos: f:actionListener, f:attribute, f:ajax
- Funções JSTL1.2: *xmlns:**fn**=" http://java.sun.com/jsp/jstl/functions".*
  Exemplos: fn:toUpperCase, fn:toLowerCase
- Passthrough (suporte a atributos não suportados e HTML5):
  xmlns:a="http://xmlns.jcp.org/jsf/passthrough"

O namespace *default* (target namespace da página XML que afeta todos os elementos que não têm prefixo) deve ser declarado da forma *xmlns="http://www.w3.org/1999/xhtml"*. Todos os namespaces devem ser declarados no elemento raiz *<html>*.

Outros namespaces podem ser declarados para suporte a extensões de bibliotecas de componentes que não fazem parte do JSF, como componentes customizados ou bibliotecas de terceiros como PrimeFaces.

Só é necessário declarar os namespaces dos tags que forem usados. Declará-los desnecessariamente não afeta a performance (apenas acrescenta alguns caracteres a mais na página).

Tags do HTML, SVG e XML *podem* ser usados em páginas JSF. Eles apenas não participam do ciclo de vida e são copiados diretamente para a saída como texto. Não é necessário usar templating para tudo. Pode-se usar *<body>* em vez de *<h:body>*, *<img>* em vez de *<h:graphicImage>* e até mesmo *<form>* ou *<input>* em uma página se o templating do JSF for indesejado, mas é boa prática usar esses tags nas páginas que serão processadas pelo runtime do JSF. Declarar *<h:head>*, mesmo que vazio é obrigatório em páginas onde há geração de JavaScript ou CSS, como páginas que usam Ajax, já que a geração de JavaScript procura o *<h:head>* para inserir código.

4.2 Componentes e tags que geram HTML

As páginas HTML são os principais componentes da camada de apresentação de aplicações Web. Uma pagina de uma aplicação Web típica de aplicações JSF inclui:

- Um conjunto de declarações de namespace no elemento raiz *(<html>)*
- Elementos *<h:head>* e *<h:body>*
- Um elemento *<h:form>* que contém componentes de entrada de dados

No mínimo é preciso declarar o namespace raiz do XHTML

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

e a biblioteca HTML padrão (prefixo "h").

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

A maior parte das páginas também irá precisar do JSF Core Library (normalmente usada com prefixo "f"):

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

A tabela abaixo relaciona os tags da biblioteca HTML e os tags ou efeitos gerados na página HTML final.

Tag JSF	O que gera em HTML
h:form	<form>
h:commandButton	<input type="submit|reset|image">
h:button	<input type="button">
h:commandLink	<a href="">
h:outputLink	<a href="">
h:link	<a href="">
h:dataTable	<table>
h:column	Uma coluna em uma tabela HTML
h:graphicImage	<img>
h:inputFile	<input type="file">
h:inputHidden	<input type="hidden">
h:inputSecret	<input type="password">
h:inputText	<input type="text">
h:inputTextarea	<textarea>
h:message e h:messages	Um ou mais se estilos forem usados
h:outputFormat	Texto (ou Texto se houver ID)
h:outputLabel	<label>
h:outputText	Texto (ou Texto se houver ID)
h:panelGrid	<table><tr><td>...</td></tr></table>
h:panelGroup	

h:selectBooleanCheckbox	<input type="checkbox">
h:selectManyCheckbox	<input type="checkbox">
h:selectManyListbox	<select>
h:selectManyMenu	<select>
h:selectOneListbox	<select>
h:selectOneMenu	<select>
h:selectOneRadio	<input type="checkbox">

Os nomes e funções dos atributos nos tags de facelets geralmente possuem muitas semelhanças com os tags HTML que eles geram. Muitas vezes têm o mesmo nome e funcionam de forma idêntica. Às vezes é possível descobrir o tag HTML associado pela semelhança do nome, que pode ser igual ou parecido (ex: <h:form> e <form>). Mas os componentes às vezes têm funções completamente diferentes e os atributos usados em JSF muitas vezes não existem em HTML.

4.3 Atributos

Cada tag tem atributos próprios, e vários componentes compartilham atributos comuns, descritos brevemente a seguir (para maiores detalhes consulte a documentação).

- **id** – identificação do componente (usado por vários outros atributos JSF); o JSF poderá gerar um ID diferente no HTML resultante.
- **immediate** – se *true*, eventos, validação e conversão devem ocorrer tão logo parâmetros de requisição sejam aplicados (dependendo do componente e do resultado da validação, isto poderá fazer com que etapas do ciclo de vida sejam puladas). Veja detalhes na seção sobre arquitetura e ciclo de vida.
- **rendered** – renderizado; se *true*, componente é exibido na tela; se *false*, ele é removido da árvore de componentes e não é exibido (similar a CSS display).
- **style** – contém regras de estilo CSS (mesmo comportamento que style em HTML)
- **styleClass** – define uma classe CSS (mesmo comportamento que class em HTML)
- **value** – define o valor do componente (o tipo de dados depende do componente e da forma como é usado)
- **binding** – mapeia uma instância a uma propriedade de bean

4.3.1 Propriedades binding e rendered

Os tags são mapeados a componentes que são instanciados como objetos Java. As classes dos componentes podem ser declaradas como membros de um managed bean para quese

possa estabelecer mapeamentos (**binding**) entre suas propriedades e o tag. Por exemplo, um bean pode declarar um atributo UIOutput:

```
@Named("mybean")
public class MyBean {
    private UIOutput output;
    ...
}
```

Ele pode ser mapeado a um elemento da página usando o atributo *binding* e uma expressão em EL que associa o atributo declarado na classe do bean um tag <h:outputText>:

```
<h:outputText binding="#{mybean.output}" />
```

Agora, o bean poderá definir o texto que será impresso na página através da propriedade value do componente:

```
public setOutput(UIOutput output) {
    this.output = output;
    this.output.setValue("Texto inserido via binding");
}
```

Nem sempre esse tipo de binding é necessário ou desejável, sendo mais comum e eficiente mapear apenas *os valores de entrada e saída* (*value*) utilizados, dos componentes/tags, a propriedades declaradas no bean. No exemplo acima, para apenas alterar o *valor* do componente, pode-se declarar um campo do tipo String no bean:

```
@Named("mybean")
public class MyBean {
    private String outputValue;
    ...
}
```

e mapear apenas o atributo *value* (e não o componente *UIOutput* inteiro):

```
<h:outputText value="#{mybean.outputValue}" />
```

Mas o *binding* é mais poderoso. Através dele pode-se mudar quaisquer propriedades do componente dinamicamente. Por exemplo, pode-se controlar quando exibir ou não um componente variando o valor da sua propriedade *rendered*, se ele estiver mapeado ao bean. Por exemplo, o bean poderia ter um método para controlar a exibição do texto:

```
public mostrarTexto(boolean mostrar) {
    this.output.setRendered(mostrar);
}
```

É possível também ter acesso aos valores mapeados em outras fases do ciclo de vida, antes deles serem validados, convertidos e transferidos para o bean, através dos métodos *getValue()* e *getSumbittedValue()* de *UIInput*.

4.4 Estrutura de uma página

O código abaixo mostra um template básico XHTML mínimo para uma página JSF que gera HTML:

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

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

4.4.1 Bloco <h:form>

Grande parte das páginas JSF são formulários HTML. Um bloco *<h:form>* deve ser usado sempre que houver necessidade de entrada de dados, mas também pode incluir links (*commandLink*) e botões (*commandButton*) e outros elementos HTML e componentes que permite-se usar fora de um bloco *<h:form>*. Uma página pode ter vários *<h:form>* mas apenas um será enviado em cada requisição.

A menos que a página tenha apenas um pequeno formulário simples, não é considerado uma boa pratica colocar todos os elementos dentro de um único *<h:form>* pai. Uma boa pratica e agrupar em um cada *<h:form>* apenas os elementos necessários para cumprir uma determinada responsabilidade de entrada de dados.

Aninhar blocos *<h:form>* é ilegal em HTML e JSF. É preciso tomar cuidado principalmente quando se usa templates e fragmentos de facelets que são inseridos dinamicamente nas páginas.

4.4.2 NamingContainer, atributo name e client ids

Um *<h:form>* possui vários atributos, mas todos são opcionais. E uma boa prática definir um ID explícitamente, já que o *<h:form>* pertence a um grupo de componentes que é derivado de *NamingContainer*. Componentes desse tipo redefinem o ID dos componentes filho. Portanto, o ID *local* do formulário será prefixado nos ids gerados para todos os seus componentes, por default. Uma sintaxe mínima recomendada é, portanto:

```
<h:form id="f1">
   …
</h:form>
```

Todos os elementos de entrada de dados em HTML possuem um atributo *name*, que contém o nome do campo que será enviado através de parâmetros HTTP. Nos componentes UI do JSF não existe atributo *name*, e se for usado ele será *ignorado* e removido do HTML final. Deve-se sempre usar *id*, e no HTML final um atributo *name* com o mesmo conteúdo será *gerado*. Se um *id* não for informado, o JSF ira automaticamente *gerar* um ID e um nome para o componente. Os atributos *id* e *name* gerados são chamados de *Client ID*, e têm um formato especial recursivo:

```
client-id-do-elemento-pai:id-local-do-componente
```

Por exemplo, se o formulário estiver na raiz da página, seu ID (local) for "*f1*", e ele contiver um componente declarado com ID local de "*comp5*" em JSF:

```
<h:form id="f1">
...
   <h:inputText id="comp5" .../>
...
```

No HTML gerado, o *client ID* desse componente será gerado pelo JSF e usado nos atributos *id* e *name*. O HTML resultante é:

```
<input type="text" id="f1:comp5"  name="f1:comp5" >
```

Se houver muitos componentes aninhados, o cliente-id pode ter vários componentes separados por ":". Por exemplo, se o *form* estiver dentro de outro *NamingContainer* com ID local *c1*, o client-id do *inputText* mudará para:

```
c1:f1:comp5
```

E assim por diante.

Em facelets XHTML, entre componentes localizados dentro de um mesmo *form* (ou qualquer *NamingContainer*), pode-se usar o ID ou o Client ID para referenciá-los (ex: f1:comp ou comp), mas se o componente referenciado estiver em outro escopo, é preciso usar o client ID (ex: f2:comp).

Se um elemento não estiver contido em um *NamingContainer*, ele tanto pode ser localizado por seu ID local (ex: *form1*), como por seu client ID (*:form1*). Um elemento com client ID começando com ":" não pertence a nenhum *NamingContainer*.

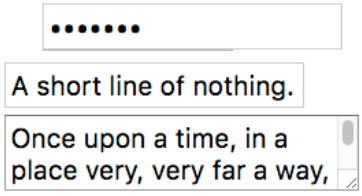
É importante saber desse comportamento porque isto afeta a intgração de aplicações JSF com outras aplicações lado-servidor (que usem nomes dos parâmetros da requisição HTTP), bibliotecas JavaScript e até mesmo folhas de estilo CSS que acessem elementos pelo seu ID.

Existem estratégias para impedir esse comportamento, mas todas elas afetam o funcionamento do JSF de alguma maneira. Por exemplo, é possível desligar esse comportamento em um *<h:form>* usando o atributo *prependId="false"*, porém isto impedirá que código JavaScript gerado pelo JSF funcione (*<f:ajax>,* por exemplo, não irá funcionar).

4.4.3 Componentes para entrada e saída de texto

Dentro de um formulário podem ser usados vários diferentes componentes de entrada de texto. Existem quatro tipos. Geram elementos <input> ou <textarea> em HTML:

- **<h:inputHidden>** - campo oculto *<input type="hidden">*
- **<h:inputSecret>** - campo de senha *<input type="password">*
- **<h:inputText>** - campo de entrada de texto *<input type="text">*
- **<h:inputTextArea>** - campo multilinha - *<textarea>*



Atributos comuns a todos os input tags são: *converter*, *converterMessage*, *label*, *required*, *requiredMessage*, *validator*, *validatorMessage*, *valueChangeListener*.

A maioria são auto-explicativos e referem-se a configurações conversores, validadores e listeners, e suas mensagens de erro. Serão detalhados mais adiante.

Existem também vários componentes usados para a geração de texto *somente leitura*, que podem ser mapeados a propriedades de um managed bean:

- **<h:outputText>** - Gera uma linha de texto (sem ID não gera nenhum elemento HTML; com ID inclui o texto gerado em um **)
- **<h:outputFormat>** - Um *<h:outputText>* que gera uma linha de texto que pode ser formatada e parametrizada.

```
<h:outputFormat value="Hello, {0}!">
      <f:param value="#{hello.name}"/>
</h:outputFormat>
```

- **<h:outputLink>** - gera um *<a href>* para outra página sem causar um action event no clique (ex: para links externos ou que geram non-faces request)
- **<h:outputLabel>** - exibe texto apenas leitura e deve ser associado a um componente usando o atributo *for* – gera um *<label>*.

Exemplos:

```
<h:inputTextarea id="description" value="#{bean.description}"/>

<h:inputText id="input" value="#{bean.input}">
   <f:validateLength minimum="5" />
</h:inputText>

<h:outputText value="#{listings.outputFormatCode}" escape="true" />

<h:outputFormat value="O número da sorte de hoje é {1} e o de ontem foi {0}.">
   <f:param value="#{bean.sorte}" />
   <f:param value="#{bean.jogar()}" />
</h:outputFormat>
```

4.4.4 Ações e navegação

Componentes de comando realizam uma ação quando ativados. Os dois elementos abaixo são mapeados a *UICommand*:

- <**h:commandButton**> - gera um <*input type="submit"*>
- <**h:commandLink**> - gera um <*a href*>

Podem usar os seguintes atributos (além dos comuns a todos os elementos)

- **action** – executa um método de ação HTTP que retorna um string ou contém um string. O string é usado para identificar a próxima página a acessar.
- **actionListener** – expressão que aponta para o método que processa um evento de clique (*ActionEvent*).

```
<h:commandLink value="Página principal" action="#{bean.goHome()}" />

<h:commandButton action="index" value="Página principal">
     <f:actionListener type="a.b.c.ExitListener" />
</h:commandLink>
```

4.4.5 Botões e links que não disparam eventos de action

Nem todo botão é um *UICommand*. Alguns links e botões são *UIInput*. <*h:link*> e <*h:outputLink*> também geram <*a href*> só que bem mais simples e que não estão automaticamente associados a uma ação HTTP. <*h:button*> gera um botão que não faz o submit do formulário.

O código abaixo mostra como usá-los para redirecionar para uma página index.faces. Observe o uso diferente do atributo *value*:

```
<h:link value="Página principal" outcome="index" />
```

```
<h:outputLink value="index.faces">Página principal</h:outputLink>

<h:button value="Página principal" outcome="index" />
```

Na verdade, <*h:outputLink*> é mais antigo. Em <*h:link*> e <*h:button*> o atributo outcome funciona de forma similar ao atributo action de *commandLink/commandButton*. Esses links e botões podem ser usados fora de <h:form>.

Todos os links também aceitam parâmetros passados como elementos-filho <*f:param*>. O exemplo abaixo ilustra um link com parâmetros. Se a opção *includeViewParams* for *true*, os parâmetros passados da requisição da URL atual serão incluídos no link. O exemplo abaixo mostra como passar para outra página os parâmetros de uma requisição:

```
<h:link outcome="segunda" value="Clique para continuar"
        includeViewParams="true">
    <f:param name="nome" value="#{bean.nome}" />
</h:link>
```

4.4.6 Gráficos e imagens

Para gerar um <*img*> dentro de um Faces Request usa-se <*h:graphicImage*>. A imagem pode ser referenciada usando os atributos

- **url** – caminho absoluto (relativo ao contexto) para a imagem.

  ```
  <h:graphicImage url="resource/imagens/bg.jpg" />
  ```

- **library** e **name** – *library* é o nome de uma pasta dentro de resources (library) e *name* é o nome do arquivo dentro dessa pasta.

  ```
  <h:graphicImage library="imagens" name="bg.jpg" />
  ```

- **value** – uma expressão EL usando objeto implícito *resource* e chave *library:name.*

  ```
  <h:graphicImage value="#{resource['imagens:bg.jpg']}" />
  ```

  Esta sintaxe pode ser usada em código CSS:

  ```
  h1 {background: url(#{resource['imagens:bg.jpg']}) repeat-x; }
  ```

4.4.7 Seleção simples e múltipla

Há quatro elementos para selecionar *um* item de uma lista:

- <**h:selectOneMenu**> - gera <select> e <option> em menu drop-down
- <**h:selectBooleanCheckbox**> - gera <checkbox>
- <**h:selectOneListBox**> - gera <select> e <option> em lista
- <**h:selectOneRadio**> - gera <radio>

Há três componentes para selecionar múltiplos itens de uma lista:

- <**h:selectManyCheckbox**> – que gera <checkbox>
- <**h:selectManyMenu**> – que gera <select> e <option>
- <**h:selectManyListbox**> – que gera <select> e <option>

O **value** do componente guarda o elemento selecionado apenas *quando o formulário for enviado* (fase 5: chamada da aplicação). A lista é povoada por elementos <**f:selectItem**> ou <**f:selectItems**>. Este último recebe no seu atributo *value* uma *coleção* de objetos, que define as opções da lista que pode ser fornecida pelo bean.

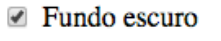
Exemplos de cada componente estão ilustrados abaixo com o código usado para criá-los. O bean usado no exemplo fornece, na propriedade *filmes*, um array de objetos do tipo *Filme*, que possui a seguinte estrutura:

```
public class Filme implements Serializable {
     private String imdb;
     private String titulo;
     private String diretor;
     private int ano;
     private int duracao;  ...
}
```

1) *<h:selectOneMenu>*

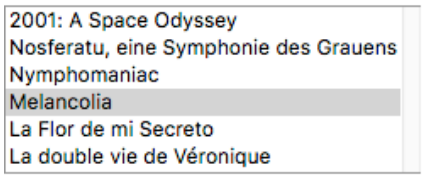

```
<h:selectOneMenu value="#{bean.filme}">
     <f:selectItems value="#{cinemateca.filmes}"          coleção ou array de itens a exibir
                    var="filme"                           variável para acessar itens individuais
                    itemLabel="#{filme.titulo}"     texto que será exibido
                    itemValue="#{filme.imdb}" />    valor que será enviado
</h:selectOneMenu>
```

2) *<h:selectBooleanCheckbox>*




```
<h:selectBooleanCheckbox value="false"/> Fundo escuro
```

3) *<h:selectOneListbox>*




```
<h:selectOneListbox id="sol" value="#{bean.filme}">
     <f:selectItems value="#{cinemateca.filmes}"
                    var="filme"
                    itemLabel="#{filme.titulo}"
                    itemValue="#{filme.imdb}" />
</h:selectOneListbox>
```

4) *<h:selectOneRadio>*

- 2001: A Space Odyssey
- Nosferatu, eine Symphonie des Grauens
- Nymphomaniac
- Melancolia
- La Flor de mi Secreto
- La double vie de Véronique

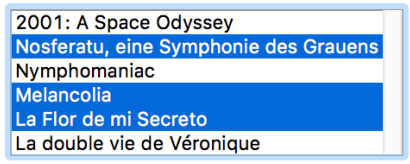
```
<h:selectOneRadio id="sor" value="#{bean.filme}" layout="pageDirection">
    <f:selectItems value="#{cinemateca.filmes}"
                    var="filme"
                    itemLabel="#{filme.titulo}"
                    itemValue="#{filme.imdb}" />
</h:selectOneRadio>
```

5) *<h:selectManyCheckbox>*

manha tarde noite

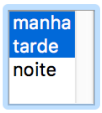
```
<h:selectManyCheckbox value="#{bean.turnos}">
     <f:selectItem itemValue="1" itemLabel="manha" />
     <f:selectItem itemValue="2" itemLabel="tarde" />
     <f:selectItem itemValue="3" itemLabel="noite" />
</h:selectManyCheckbox>
```

6) *<h:selectManyListbox>*




```
<h:selectManyListbox value="#{bean.selecaoFilmes}">
     <f:selectItems value="#{cinemateca.filmes}"
                                var="filme"
                                itemLabel="#{filme.titulo}"
                                itemValue="#{filme.imdb}" />
</h:selectManyListbox>
```

7) *<h:selectManyMenu>*




```
<h:selectManyMenu value="#{bean.turnos}">
     <f:selectItem itemValue="1" itemLabel="manha" />
     <f:selectItem itemValue="2" itemLabel="tarde" />
     <f:selectItem itemValue="3" itemLabel="noite" />
</h:selectManyMenu>
```

4.4.8 Layout e tabelas

O elementos <**h:panelGrid**> e <**h:panelGroup**> são usados para realizar layout simples em forma de tabela. *<h:panelGrid>* organiza uma tabela considerando cada bloco de texto ou bloco HTML como uma célula e encaixando no número de colunas declarado *<h:panelGrid columns="número de colunas">*.

Se houver necessidade de tratar vários blocos de texto ou HTML como uma única célula, pode-se usar *<h:panelGroup>* para agrupar vários componentes em uma única célula.


```
<h:panelGrid columns="3">
  <h:inputText value="1.1" />
  <h:inputText value="1.2" />
  <h:panelGroup>
    <h:panelGrid>
      <h:inputText value="1.3.a" />
      <h:inputText value="1.3.b" />
      <h:inputText value="1.3.c" />
    </h:panelGrid>
  </h:panelGroup>

  <h:inputText value="2.1" />
  <h:inputText value="2.2" />
  <h:inputText value="2.3" />
</h:panelGrid>
```

Para apresentar uma coleção de itens na forma de uma tabela de dados pode-se usar *<h:dataTable>* que realiza *binding* com uma coleção de objetos (array, *List*, *ResultSet*, *DataModel*) e apresenta os dados como uma tabela HTML. A coleção deve ser passada ao atributo *value*, e os elementos individuais podem ser acessados através de uma variável declarada em *var*. Dentro da tabela os dados são agrupados em colunas com *<h:column>*

```
<h:dataTable value="#{cinemateca.filmes}" var="filme" rows="3" first="2">
     <h:column>#{filme.titulo}</h:column>
     <h:column>#{filme.diretor}</h:column>
     <h:column>#{filme.ano}</h:column>
...
</h:dataTable>
```

Para incluir cabeçalhos e legendas use *<f:facet>*. O componente *<h:dataTable>* reconhece as facetas *header*, *footer* e *caption* (cabeçalho, rodapé e legenda). Outros atributos de *h:dataTable* incluem:

- *border*, *bgcolor*, *cellpadding*, etc. (todos os atributos do elemento HTML *<table>*)
- **styleClass**, **captionClass**, **headerClass**, **footerClass**, **rowClasses**, **columnClasses** – classes CSS para diferentes partes da tabela
- **first** - Primeiro item da coleção a mostrar na tabela (para usar em paginação)
- **rows** - Número de linhas da coleção a exibir na tabela (para usar em paginação)

Exemplo (usando o mesmo bean dos exemplos anteriores):

Título	Diretor	Ano
2001: A Space Odyssey	Stanley Kubrick	1968
Nosferatu, eine Symphonie des Grauens	F. W. Murnau	1922
Nymphomaniac	Lars von Trier	2013
Melancolia	Lars von Trier	2011
La Flor de mi Secreto	Pedro Almodovar	1995
La double vie de Véronique	Krzysztof Kieslowski	1991

```
<h:dataTable value="#{cinemateca.filmes}" var="filme" rows="3" first="2">
  <h:column>
    <f:facet name="header">Título</f:facet>
    #{filme.titulo}
  </h:column>
  <h:column>
    <f:facet name="header">Diretor</f:facet>
    #{filme.diretor}
  </h:column>
  <h:column>
    <f:facet name="header">Ano</f:facet>
    #{filme.ano}
  </h:column>
</h:dataTable>
```

4.5 Core tags

Os core tags do JSF interagem com os tags de componentes gráficos acrescentando comportamentos diversos como tratamento de eventos, conversores, validadores, comunicação assíncrona, metadados e configuração. Alguns foram usados na seção anterior, para incluir itens em uma lista ou menu. Outros serão abordados em seções próprias. As tabelas abaixo listam os core tags mais importantes.

Todos são usados como *elementos-filho* de outros tags e afetam o contexto no qual foram inseridos (geralmente o elemento pai e outros filhos).

4.5.1 Tags para tratamento de eventos

É uma das formas de associar um handler de eventos ao componente pai (vários componentes permitem também fazer isto através de atributos). Todos possuem um atributo *type* que recebe uma expressão EL vinculando o tag a um método que irá processar o evento quando ele ocorrer.

`f:actionListener`	Adiciona um action listener, que reage a eventos de ação (ex: cliques)
`f:valueChangeListener`	Adiciona um value-change listener, que reage a eventos de mudança de valor
`f:phaseListener`	Adiciona um PhaseListener que monitora as fases do ciclo de vida do JSF
`f:setPropertyActionListener`	Registra um action listener que define uma propriedade no managed bean quando formulário é enviado
`f:event`	Registra um *ComponentSystemEventListener* em um componente

Estes tags serão explorados em maior detalhe no capitulo sobre Listeners.

4.5.2 Tags para conversão de dados

É uma das formas de associar um conversor de dados ao componente pai (vários componentes também permitem associar conversores através de atributos).

`f:converter`	Registra um conversor ao componente no qual este tag é incluído. O conversor deve ter sido declarado via metadados (anotações ou faces-config.xml) é identificado através do atributo converterId.
`f:convertDateTime`	Registra um conversor pra formatos de data e hora ao componente pai. Atributos são usados para configurar o formato.
`f:convertNumber`	Registra um conversor de número ao componente pai. Atributos são usados para configurar o formato.

Estes tags serão explorados em maiores detalhes no capitulo sobre Converters.

4.5.3 Tags para validação

É uma das formas de associar um validador ao componente pai (vários componentes também permitem associar validadores através de atributos, ou via *Bean Validation*).

f:validateDoubleRange	Adiciona um DoubleRangeValidator
f:validateLength	Adiciona um Length Validator
f:validateLongRange	Adiciona um LongRangeValidator
f:validator	Adiciona um validador customizado
f:validateRegEx	Adiciona um validador de expressão regular
f:validateBean	Delega a validação para um BeanValidator
f:validateRequired	Requer validação em um componente

Esses tags serão explorados em maiores detalhes no capitulo sobre Validators.

4.5.4 Outros tags

Alguns desses tags já foram usados na construção de componentes como listas, tabelas, links. Outros são usados para tratamento de parâmetros em requests HTTP, carregar bundles de mensagens e propriedades de configuração, configuração de requisições Ajax, etc. A seguir um resumo de cada um:

f:facet	Adiciona um componente aninhado que tem algum relacionamento com o componente pai (ex: em tabelas e usado para definir os cabeçalhos das colunas)
f:param	Substitui parâmetros e adiciona pares nome-valor a uma URL. (ex: usado em links para passar parâmetros para a URI gerada)
f:selectItem	Representa um item em uma lista (ex: usado em componentes de menu)
f:selectItems	Representa um conjunto de itens (ex: usado em componentes de menu)
f:view	Representa uma View. O tag, em JSF 2.2 é opcional (mas é inserida na árvore automaticamente por default). Pode ser incluída na página (em volta de <h:head> e <body>) para configurar defaults, como locale, encoding e tipo MIME.
f:viewParam	Permite o processamento de parâmetros da requisição HTTP GET (usado dentro de um bloco <f:metadata>)

f:viewAction	Permite associar uma operação diretamente a parâmetros recebidos em HTTP GET (geralmente usado para executar uma operação logo após a carga da página).
f:metadata	Registra uma faceta em um componente pai (frequentemente usado em processamento de parâmetros HTTP (<f:viewParam>) onde o componente pai é <h:head>)
f:attribute	Adiciona atributos a um componente (ex: pode ser usado para passar dados para listeners, conversores, etc.)
f:loadBundle	Carrega um resource bundle (properties) que é acessível como um Map
f:ajax	Associa uma ação Ajax com um componente ou grupo de componentes. Este elemento sera explorado em um capítulo especifico.

4.6 Core tags do JSTL

Os core tags do JSTL são incluídos através do namespace *http://xmlns.jcp.org/jsp/jstl/core*.

É importante observar que eles apenas são processados quando a árvore de componentes está sendo montada (fase 1), em contraste com os facelets, que são processados na fase 6.. Nesta fase, dados que estão na página ainda não foram transferidos para os seus respectivos componentes e nem para o bean. Se uma expressão condicional depender desses valores, o resultado poderá ser diferente daquele esperado.

O resultado é mais fácil de prever se estes tags forem usados apenas junto com outros tags JSTL, e que apenas *leiam* dados. Em geral, há boas alternativas JSF. Há uma ótima discussão sobre diferenças entre tags JSTL e JSF em *http://stackoverflow.com/questions/3342984/jstl-in-jsf2-facelets-makes-sense*.

c:if	Tag condicional. Equivale a um if, mas não suporta um bloco complementar (default/else). Para situações desse tipo use <c:choose>. Para exibir/ocultar componentes em JSF é preferível usar expressões EL condicionais como valor para atributo rendered.
c:choose c:when c:otherwise	Tag condicional. Equivale a if-then-else ou switch. <choose> deve conter um ou mais blocos <when test=””> (equivalente a if-else/case) e opcionalmente um <otherwise> (equivalente a else/default).
c:forEach	Tag de repetição que ocorre na primeira fase do JSF (pode ser usado em JSF para dados estáticos já carregados antes). Alternativas JSF são <ui:repeat>, ou mesmo <h:dataTable> se o objetivo da repetição for gerar uma tabela.

c:set	Define uma propriedade em um escopo com base no valor dos atributos. Em JSF pode ser usado com valores estáticos que irão ser usados apenas na página (default). No mundo JSF o ideal é usar <ui:param>.
c:catch	Captura qualquer objeto Throwable (exceção) que ocorrer dentro do <c:catch>.

4.7 Tags de templating

A maior parte destes tags sao usados principalmente na construção e uso de templates e componentes reutilizáveis. Este assunto será tratado em um capitulo a parte.

ui:component ui:fragment	Usados na construção de componentes. <ui:component> ignora todo o HTML fora do elemento.
ui:composition ui:decorate	Representam uma composição de elementos. Podem ser usados para construir templates reusáveis. <ui:composition> ignora todo o HTML fora do elemento.
ui:debug	Se incluído em uma página, a combinaçãoo Ctrl-Shift-D (default – pode ser alterada com o atributo hotkey) faz com que informações sobre a árvore de componentes seja mostrada em uma janela pop-up.
ui:define	Define parte de uma página que irá ser inserido em um ponto de inserção de um template.
ui:include	Inclui conteúdo de outra página.
ui:insert	Usado em templates para definir área de inserção.
ui:param	Usado em elementos <ui:include> para passar parâmetros para um template e para setar variáveis em um escopo. Pode também ser usado em um tag raiz (<f:view>) para definir uma constante para toda a view.
ui:repeat	Permite a repetição do seu conteúdo.
ui:remove	Remove o conteúdo da árvore de componentes. Pode ser usado para "comentar" código JSF, eliminando-o da árvore de componentes para que não seja processado nem gere HTML.

4.7.1 Repetição com ui:repeat

Embora a maior parte dos tags desta coleção em componentes e templates, *<ui:repeat>* é o principal tag usado para repetição em JSF. O ideal seria usar um componente, já que repetição é uma tarefa que deveria estar dentro de um componente reusável, mas como não há

um componente similar a *<h:dataTable>* em JSF para outras estruturas HTML que se repetem, como listas *<li>*, *<ui:repeat>* é uma alternativa muito melhor que *c:forEach*.

O trecho de código abaixo, usando *c:forEach* e *c:if*:

```
<ui:forEach items="#{[1,2,3,4,5,6,7,8]}" var="numero">
     <c:if test="#{numero mod 2 ne 0}">
         <li><h:outputText value="#{numero}" /></li>
     </c:if>
</ui:forEach>
```

Irá gerar os mesmos dados que o código abaixo, usando *ui:repeat* e *rendered*:

```
<ui:repeat value="#{[1,2,3,4,5,6,7,8]}" }" var="numero">
     <ui:fragment rendered="#{numero mod 2 ne 0}">
         <li><h:outputText value="#{numero}" /></li>
     </ui:fragment>
</ui:repeat>
```

E importante destacar a principal diferença entre *h:dataTable/ui:repeat* e *c:forEach*: os primeiros atuam sobre a *árvore de componentes* (objeto que é alterado durante seis fases do JSF), enquanto que o segundo atua sobre a própria página, gerando o HTML:

- Em JSF o tag *<h:outputText>* é reusado para construir uma coleção de 4 itens *na memória* que será usada para gerar 4 tags <li> contendo números diferentes.
- Em JSTL, o *<h:outputText>* é copiado 4 vezes *na página*, e depois cada cópia gera 1 tag <li>. O resultado final é o mesmo, 4 tags <li> (mas nem sempre é assim.!)

Por exemplo, com base no funcionamento "esperado" de cada tag, poderíamos imaginar que não faria diferença usar um *c:if* dentro de um *ui:repeat*:

```
<ui:repeat value="#{[1,2,3,4,5,6,7,8]}" }" var="numero">
    <c:if test="#{numero mod 2 ne 0}">
      <li><h:outputText value="#{numero}"/></li>
    </c:if>
</ui:repeat>
```

Mas o que ocorre é que neste caso não é impresso nada! E se a condição for invertida, para:

```
#{numero mod 2 eq 0}"
```

Ele imprime todos! A condicional if aparentemente não está funcionando como se espera.

O problema acontece porque *<ui:repeat>* age na *árvore de componentes*, e reusa o *<h:outputText>*. Ele não faz cópias dele na página como em *<c:forEach>*. Assim, o *c:if* é executado apenas uma vez, e testando se *numero* é zero ou não.

5 Managed beans

Uma aplicação típica JSF contém um ou mais beans gerenciados pelo container, ou *managed beans*. Um managed bean pode ser um POJO configurado com anotaçoes CDI ou um Session Bean (ou ainda um componente anotado com @ManagedBean, mas esta última forma está caindo em desuso e não é recomendada para projetos novos). Na arquitetura MVC, este bean é um *controller*, mas faz parte da camada de apresentação. Deve estar mais fortemente relacionado com a interface gráfica de comunicação com o usuário que com os componentes do modelo de domínio da aplicação.

Cada managed bean pode ser associado com os componentes usados em uma página através de mapeamento de valores ou *binding* do componente. Um managed bean segue as regras gerais para a construção de Java Beans:

- Construtor sem argumentos
- Estado na forma de *propriedades* (acessíveis via métodos get/set)
- Atributos privativos

Além disso, se o bean for compartilhado em um escopo que envolva passivação, deve ser serializável (declarar Serializable e conter apenas atributos de estado serializáveis).

Muitos beans contém propriedades que serão usadas em coleções, para construir tabelas, menus, etc. É muito importante que esses beans sejam serializáveis, para que possam participar de escopos que requerem passivação, e que estejam bem implementados com os métodos *equals()* e *hashCode()*, sob pena de introduzirem bugs muito difíceis de encontrar nas aplicações em que forem usados devido a objetos duplicados.

A classe a seguir é um managed bean CDI, acessível via EL através do nome *livros*, e que contém uma propriedade *allLivros*:

```
@Named("livros")
public class LivroDAOManagedBean {
     @Inject EntityManager em;
     public List<Livro> getAllLivros() {
             return em.createNamedQuery("selectAll", Livro.class).getResultList();
     }
}
```

O session bean a seguir *também* é um managed bean, acessível via EL através do nome *livroDAOSessionBean*, e com uma propriedade somente-leitura *livros*:

```
@Stateless
public class LivroDAOSessionBean {
     @PersistenceContext
     EntityManager em;
```

```
    public List<Livro> getLivros() {
        return (List<Livro>)em.createNamedQuery("selectAll").getResultList();
    }
}
```

5.1 Mapeamento de propriedades em um managed bean

Cada *propriedade* de um managed bean pode ser mapeada a uma

- Instância de um componente (binding)
- Valor de um componente (value)
- Instância de um conversor (converter)
- Instância de um listener (listener)
- Instância de um validador (validator)

5.1.1 Binding de componentes

O mapeamento direto *instâncias dos componentes* (*binding*) permite um grande dinamismo da interface, pois é possível controlar seus atributos diretamente. Propriedades mapeadas via *binding* devem ter tipo idêntico ao do componente.

Por outro lado, *binding* é considerada uma prática rara e usar como principal meio de conectar beans a componentes é uma prática não-recomendável. Componentes UI sempre têm escopo limitado à requisição. Se usados em um escopo mais duradouro (View, Session, etc.) eles serão recriados e podem causar erros de duplicação de ID. Em geral, binding só deve ser usado em escopos locais.

Para mapear um componente representado por um facelet a um bean, é preciso descobrir qual a classe *UIComponent* que está mapeado a este facelet, e declarar a classe como propriedade do managed bean. Por exemplo, o *<h:inputText>* da página abaixo declara estar mapeado à propriedade *inputComponent* de mensagemBean::

```
<h:form id="formulario">
    <h:inputText value="#{mensagemBean.mensagem}"
                 binding="#{mensagemBean.inputComponent}"/>
     ...
</h:form>
```

O facelet *<h:inputText>* é um objeto UIInput na árvore de componentes, portanto na classe que implementa *mensagemBean*, a propriedade deve ser declarada com este tipo:

```
@Named
@SessionScoped
public class MensagemBean implements Serializable {
```

```
    private Mensagem mensagem;
    private UIInput inputComponent;
    ...
}
```

Assim, dentro do bean é possível ter acesso ao estado do componente, e alterar propriedades como *rendered* (via setRendered()) que podem incluir/remover o objeto da renderização da página:

```
public void método() {
    ...
    inputComponent.setRendered(false);
    ...
}
```

5.1.2 Mapeamento de valores

É muito mais comum mapear o valor dos componentes às propriedades do bean. Os valores não se limitam aos tipos básicos do Java. Através do uso de conversores, pode-se associar objetos, beans, arrays, coleções, enums, preservando as informações encapsuladas nos objetos.

Para realizar o mapeamento é preciso descobrir o tipo de dados que está armazenado no componente. O componente deve ser um *ValueHolder*, ou seja, capaz de armazenar valor. Se o valor for um string ou tipo numérico, ou coleção desses tipos, não é necessário implementar conversores (a menos que se deseje formatar os dados).

Componentes *UIInput* e *UIOutput* guardam valores que podem ser representados como *texto*, e fazem a conversão automática de tipos numéricos (inclusive BigDecimal), datas e strings em geral. Requerem conversores para outros tipos de objetos, ou para formatar moeda, datas em relação ao locale, etc. Para isto o JSF disponibiliza alguns conversores prontos.

Os mesmos bean e tag mostrado anteriormente tinha um mapeamento de valor para um *UIInput*. As listagens a seguir destacam esse mapeamento:

```
<h:form id="formulario">
    <h:inputText value="#{mensagemBean.mensagem}"
                 binding="#{mensagemBean.inputComponent}"/>

    ...
</h:form>

@Named
@SessionScoped
public class MensagemBean implements Serializable {
```

```
    private Mensagem mensagem;
    private UIInput inputComponent;
...
```

O value de um componente *UISelectBoolean* deve ser mapeado a um tipo *boolean* (ou *Boolean*), e elementos que armazenam coleções, como *UIData*, *UISelectMany*, *UISelectItems*, etc., são mapeados a coleções ou arrays.

Componentes de seleção representam a coleção de dados via <f:selectItem> ou <f:selectItems>, como foi mostrado no capítulo sobre Facelets. Esses itens podem ser mapeados via binding a UISelectItem/UISelectItems e via value a coleções. Por exemplo, considere o bean abaixo que contém um array de objetos Filme:

```
@Named("cinemateca")
@ApplicationScoped
public class CinematecaBean {
    private Filme selecionado;
    private Filme[] filmes;
    ...
}
```

A lista de opções de um menu pull-down pode ser construída mapeando o atributo value de *<f:selectItems>* ao array, e o item selecionado, mapeando o value do *<h:selectOneMenu>* ao atributo selecionado.

```
<h:selectOneMenu id="som" value="#{bean.filme}">
    <f:selectItems value="#{cinemateca.filmes}"
                   var="filme"
                   itemLabel="#{filme.titulo}"
                   itemValue="#{filme.imdb}" />
    </h:selectOneMenu>
```

Em ambos os casos, um converter será necessário para que o objeto possa ser selecionado, enviado para processamento, e o resultado exibido.

Os atributos de um bean podem ser mapeados a vários componentes ao mesmo tempo, e refletirão as mudanças ocorridas em cada um.

5.2 Comportamento de um managed bean

Um managed bean pode conter métodos., Tipicamente eles lidam com:

- Ação de processamento e navegação (chamado no *action*, do form)
- Tratamento de eventos (listeners)
- Processamento de validação (validators)

Criar esses métodos no próprio managed bean elimina a necessidade de implementar validators e listeners em classes separadas, quando são simples e não há interesse em reusá-los. A vantagem é que terão acesso aos campos privativos do bean e poderão ser implementados mais facilmente e sem quebrar encapsulamento.

5.2.1 Métodos de ação

É o método que é chamado para processar um formulário, e que retorna uma instrução de navegação, informando a próxima View. Um método de ação deve ser público, pode ou não receber parâmetros e deve retornar um objeto identificador de navegação, que é geralmente um String (pode ser um enum também).

Um método de navegação é referenciado pelo atributo *action* dos componentes que suportam ações (CommandUI)

```
public String submit() { ...
  if ((compra.getTotal()> 1000.00)) {
    desconto.setRendered(true);
     return null;
  } else if (compra.getItens() > 10) {
    oferta.setRendered(true);
     return null;
  } else { ...
    carrinho.clear();
     return ("fatura");
  }
}
```

Retornar *null* reapresenta a mesma página. É o default em processamento Ajax.

5.2.2 Métodos de processamento de eventos

Os métodos de tratamento de eventos devem ser métodos públicos que retornam void. O argumento depende do tipo de evento gerado. Os componentes UI provocam dois tipos de eventos: *ActionEvent*, disparados por botões e links, e *ValueChangeEvent*, disparados por elementos que retém valor.

ActionEvent possui *getComponent()* que pode ser usado para identificar o componente que provocou o evento. O método a seguir, implementado em um managed bean, é chamado quandoum botão é apertado, e imprime o seu ID local:

```
public void processadorAction(ActionEvent e) {
     this.setButtonId(e.getComponent().getId());
     System.out.println("Bean.processadorAction: " + this.buttonId);
}
```

Para registrar o listener usa-se o atributo actionListener do componente:

```
<h:commandButton value="Button 2" id="btn2"
                 actionListener="#{bean.processadorAction}" />
```

Componentes que suportam eventos *ValueChangeEvent* não têm atributo *actionListener*, mas *valueChangeListener*, que deve conter uma expressão EL para vinculá-lo ao método apropriado:

```
<h:inputText id="name" valueChangeListener="#{bean.processarValueChange}" />
```

ValueChangeEvent possui *getNewValue()* e *getOldValue()* que permite acesso ao valor novo e o valor antigo.

```
public void processarValueChange(ValueChangeEvent event) {
   if (null != event.getNewValue()) {
       FacesContext.getCurrentInstance()
       .getExternalContext()
       .getSessionMap().put("name", event.getNewValue());
   }
}
```

Listeners e eventos serão explorados em maior detalhe em um capítulo próprio.

5.2.3 Métodos para realizar validação

Um método que realiza validação deve receber como parâmetros

- O *FacesContext*
- O *UIComponent* cujos dados devem ser validados
- Os dados a serem validados

```
public void validarNull(FacesContext context, UIComponent comp, Object value) {
    if(value == null || value.toString().length() == 0) {
        throw new ValidatorException(new FacesMessage("Campo obrigatório!"));
     }
}
```

É referenciado usando o atributo *validator* de componentes descendentes de *UIInput*:

```
<h:inputText value="#{transporte.destinatario.nome}" id="nome"
             validator="#{transporte.validarNull}"/>
```

Validação será explorada em maior detalhe em um capítulo próprio.

5.3 Escopos

Escopos representam duração da vida de um bean. São escopos de persistência da plataforma Web. Aplicações Web em Java (WebServlets, JSP) definem quatro escopos, em ordem de durabilidade:

- *Página*: durante a execução do método *service()* de um *Servlet*, um objeto de requisição é criado e repassado a uma ou mais páginas. Objetos que são instanciados quando a página recebe a requisição deixam de existir quando a requisição é repassada a outra página. Este é o escopo mais curto.
- *Requisição*: tem o escopo do método *service()* do *Servlet*, que instancia objetos quando a requisição HTTP é recebida, e os destrói quando a resposta é devolvida ao cliente.
- *Sessão*: como HTTP é um protocolo que não mantém estado, a sessão precisa ser mantida artificialmente usando um mecanismo externo. O mais comum é usar Cookies de sessão, que são criadas na primeira requisição feita para um domínio/caminho por um cliente, e mantido enquanto o cliente continuar enviando requisições para o mesmo domínio/caminho. Objetos associados à sessão são destruídos apenas quando o cliente encerra a conexão com o domínio/caminho, geralmente fechando a aba do browser ou criando um novo Cookie com data de expiração no passado.
- *Aplicação*, ou *Contexto Web*: representa a instância do container ou servidor Web enquanto ele estiver executando. Objetos que têm escopo de aplicação só deixam de existir se o container ou servlet que representa o contexto de execução for destruído (ex: se o servidor for reiniciado).

Em aplicações Ajax, o conceito de página foi redefinido, pois tornou-se possível realizar várias requisições sem carregar uma nova página. Em JSF, que implementa suporte a Ajax, o escopo de página, do JSP (que dura menos que uma requisição), foi substituído pelo escopo de *View* (que pode durar várias requisições). O escopo de página ainda existe, mas não tem a importância em JSF que tinha antes em JSP. Escopo de View também não existe em HTTP, e é, na verdade, é um tipo especial de sessão que se encerra quando o usuário deixa uma página. O JSF 2.2 e CDI ainda define mais dois escopos, que permitem controlar melhor a duração de uma sessão: *Conversation*, que permite controlar programaticamente o início e o fim de uma sessão, e *Flow*, que atrela a sessão a um conjunto de páginas agrupadas por regras de navegação.

5.3.1 Escopos em managed beans

O pacote *javax.faces.beans* define várias anotações para definir escopos em beans controlados pelo runtime JSF, mas essa responsabilidade hoje é duplicada por outros pacotes do Java EE, como CDI e EJB. A partir do próximo lançamento de Java EE, o uso de escopos do JSF será deprecado. Portanto, atualmente recomenda-se que para beans que não são EJB, os escopos usados sejam declarados usando contextos CDI, disponíveis no pacote *javax.enterprise.context.*

Os escopos JSF (antigos) são criados guardando referências para os objetos em um *Map*. Os mapas são obtidos do *FacesContext*.

```
FacesContext ctx = FacesContext.getCurrentInstance();
ExternalContext ectx = ctx.getExternalContext();

Map requests = ectx.getRequestMap();
Map sessions = ectx.getSessionMap();
Map appContexts = ectx.getApplicationMap();
```

O mapa do View é obtido do ViewRoot:

```
Map view = ctx.getViewRoot().getViewMap();
```

Tendo-se o *Map*, pode-se obter a instância de um bean que tenha sido declarado com *@ManagedBean*:

```
Bean bean = (Bean)sessions.get("bean");
String propriedade = bean.getMensagem();
```

Outra forma de localizar um bean é executar o *ELResolver*, que irá procura-lo em todos os escopos que o runtime EL tiver acesso (inclusive escopos CDI e EJB):

```
FacesContext ctx = FacesContext.getCurrentInstance();
            ELContext elCtx = ctx.getELContext();
            Application app = ctx.getApplication();
            Bean bean = (Bean)app.getELResolver().getValue(elCtx, null, "bean");
```

Esse procedimento é ideal quando não se sabe em que escopo está o bean.

Finalmente, se o bean foi anotado com *@Named*, e foi armazenado em um escopo CDI, que é a forma recomendada para JSF 2.2 e Java EE 7, a instância pode ser recuperada através de um *@Inject*:

```
public class MyListener ... {
    @Inject
    Bean bean;
    ...
    public void listenerMethod(...) {
        String propriedade = bean.getMensagem();
         ...
```

As seções a seguir apresentam um resumo dos escopos do CDI e seu significado. Para maiores informações, consulte a documentação do CDI.

5.3.2 Escopos fundamentais: requisição, sessão e contexto

Estes três escopos de persistência são abstrações importantes em qualquer aplicação Web de qualquer tecnologia. Em JSF/CDI o escopo mais curto é a *requisição*. A *sessão* tem geralmente a duração definida pelo cliente. O *contexto Web/aplicação* dura o tempo em que o

WAR da aplicação está instaslado, ativo e executando no container Web. A vida de um managed bean pode ser configurada para ter a duração desses contextos simplesmente anotando a classe do bean com:

- `@javax.enterprise.context.RequestScoped` – escopo de sessão
- `@javax.enterprise.context.SessionScoped` – escopo de sessão
- `@javax.enterprise.context.ApplicationScoped` – escopo da aplicação (contexto Web)

Um bean declarado com **@RequestScoped** neste escopo será instanciado a cada requisição HTTP, e ficará disponível pelo tempo que durar a requisição (até a resposta ser recebida). Sem Ajax, a requisição termina com a recarga da página ou com a renderização de outra página. Usando Ajax, várias requisições podem ocorrer sem que haja mudança do endereço da página.

O bean declarado com **@SessionScoped** será criado na primeira requisição e estará disponível até que a sessão seja invalidada, que geralmente depende do cliente e da duração da sessão, que pode envolver várias requisições.

Uma sessão também pode ser invalidada através de comando enviado pelo servidor. O método invalidateSession do ExternalContext pode ser usado para este fim.

```
public String logout() {
    ctx.getExternalContext().invalidateSession();
    return "/home.xhtml?faces-redirect=true";
}
```

O redirect é fundamental para que a página não permaneça mostrando dados obsoletos.

É importante observar que o *invalidateSession()* destrói *todas* as sessões, não apenas aquela identificada por *@SessionScoped*, mas também beans configurados como @ViewScoped, *@FlowScoped* e *@ConversationScoped*

Uma vez instanciado, um bean declarado como **@ApplicationScoped** estará disponível durante toda existência da aplicação (enquanto a aplicação Web no container não for reiniciada). Este escopo é ideal para serviços e estados que serão compartilhados por vários usuários e aplicações. Pode haver problemas de concorrência se vários usuários tentam alterar mesmos dados.

Escopos similares a *@ApplicationScoped* podem ser implementadom usando singletons (*@javax.ejb.Singleton* ou *@javax.inject.Singleton*)

5.3.3 Escopos de sessão: view, conversation e flow

Estes três escopos estão atrelados à sessão HTTP. São intermediárias entre a duração de uma requisição e a duração de uma sessão. A principal diferença entre elas consiste no controle da sua duração. Como são atraladas à sessão, chamar um método para invalidar a sessão destrói não apenas os beans que são *@SessionScoped*, mas também qualquer um anotado com as sessões abaixo:

- `@javax.faces.bean.ViewScoped` – uma view (ex: página + ajax)
- `@javax.faces.flow.FlowScoped` – um flow (conjunto de páginas relacionadas)
- `@javax.enterprise.context.ConversationScoped` – um diálogo (com início e fim)

Um bean anotado com **@ViewScoped** dura o tempo de uma View. Este escopo é adequado a aplicações Ajax que realizam várias requisições na mesma página. O bean será destruído e removido do escopo quando a página mudar.

@ConversationScoped pode também ser usado para aplicações Ajax, e oferece mais controle pois permite que a aplicação defina quando o escopo começa e quando termina. Para controlar é preciso injetar um *javax.enterprise.context.Conversation* no bean que irá determinar a duração do escopo. Usando a instância injetada, pode-se iniciar o escopo com o método begin(), e encerrá-lo com end():

```
@Inject Conversation conv;
...
public void inicioDaConversa() {  ...
     conv.begin(); // escopo está ativo
}
...
public void conversaTerminada() {...
     conv.end();
     return "/home.xhtml?faces-redirect=true";
}
```

O redirecionamento no final é importante, pelos mesmos motivos que a invalidação de uma sessão.

@FlowScoped é usado em componentes que fazem parte de um *Faces Flow*: uma coleção de Views usadas em conjunto, relacionadas entre si por regras de navegação (configuradas em faces-config.xml), com um único ponto de entrada e saída. O escopo inicia quando o Flow é iniciado, e encerra automaticamente quando o usuário sai do Flow. Um Flow pode envolver várias páginas.

5.3.4 Outros escopos

Outros dois escopos são usados em beans CDI. O primeiro ***@Dependent*** não é realmente um escopo, mas um mecanismo que indica que o bean anotado com ele herda o escopo do bean que o utiliza. @Dependent é default em CDI e aplicado em beans que não têm escopo declarado.

@TransactionScoped dura o tempo de uma transação, que geralmente é menos que uma requisição, já que aplicações CDI e EJB típicas demarcam transações que duram no máximo o escopo de um método de negócios, e o uso de transações distribuídas em WebServlets nunca devem ter escopo maior que o método *service()*. Beans anotados com esta anotação, portanto, deixam de existir logo que acontecer o *commit()* da transação.

6 Conversores

Conversores possibilitam que JSF trabalhe com objetos que nem sempre podem ser exibidos em uma página sem algum tipo de conversão. Conversores encapsulam em um objeto as regras de conversão Objeto-String-Objeto, simplificando o desenvolvimento e facilitando o reuso. Grande parte dos objetos usados em aplicações Web consistem de textos, números e datas, que são facilmente convertidos em String. Esses formatos são automaticamente convertidos pelo JSF, que também é capaz de extrair e converter objetos individuais de uma coleção. Em situações em que a conversão não é óbvia, o JSF fornece implementações prontas com atributos que permitem configurar e ajustar a conversão de certas estruturas como datas, locales, moeda, etc. Essas implementações estão disponíveis no pacote *javax.faces.convert* e incluem:

- *BigDecimalConverter*
- *BooleanConverter*
- *DateTimeConverter*
- *EnumConverter*
- *IntegerConverter*, etc.

Cada *converter* tem uma mensagem de erro padrão associada, que será exibida se conversão falhar. Por exemplo, BigIntegerConverter contém a mensagem: *"{0} must be a number consisting of one or more digits"*. Essas mensagens, e outros aspectos de configuração, podem ser ajustados de maneira declarativa.

Há três tags que permitem configurar converters:

- *<f:converter>* - usado para registrar um conversor nativo ou customizado.
- *<f:numberConverter>* - para conversão e formatação de números e moeda.
- *<f:dateTimeConverter>* - para conversão e formatação de datas.

6.1 Como usar um conversor

Há várias formas de usar conversores. A mais simples é automática nao requer nada além do mapeamento de valor. Por exemplo, se um managed bean declara o seguinte tipo:

```
Integer idade = 0;
public Integer getIdade(){ return idade;}
public void setIdade(Integer idade) {this.idade = idade;}
```

Mapeado, na página XHTML, da seguinte forma:

```
<h:inputText value="#{bean.idade}" />
```

O texto (string) digitado, contendo o número, será automaticamente convertido para Integer antes de ser gravado no bean. Se o usuário digitar algo que não pode ser convertido em inteiro, será lançado um erro de conversão/validação, que impedirá o envio do formulário. Conversões similares podem ser feitas com qualquer tipo primitivo e tipos básicos como *BigDecimal* e datas.

é incluir um dos facelets de converter dentro do elemento que representa o componente UI que precisa converter dados.

A conversão automática é realizada por um objeto conversor. Ele pode ser indicado explicitamente (embora neste caso não seja necessário) de duas formas. Uma delas é usando o atributo *converter* do componente :

```
<h:inputText value="#{...}" converter="javax.faces.convert.IntegerConverter" />
```

Outra forma é usar o tag *<f:converter>* dentro de um componente e referenciá-lo pelo seu ID (o ID default é o tipo convertido):

```
<h:inputText value="#{bean.idade}" />
     <f:converter converterId="Integer" />
</h:inputText>
```

Embora não sejam necessárias para tipos básicos, essas formas são usadas para incluir converters customizados.

6.2 Conversão de datas

DateTimeConverter converte datas em objeto e vice-versa. A conversão default usa apenas a representação *toString()* para a data, que geralmente é insuficiente e dificulta o processamento. Incluindo e configurando o facelet *<f:convertDateTime>* em um componente

que recebe ou exibe datas, oferece muito mais controle. O componente encapsula vários formatos, baseados nos disponíveis na classe utilitária *DateFormat*.

```
<h:outputText value="#{bean.aniversario}">
     <f:convertDateTime type="date" dateStyle="full" />
</h:outputText>
```

O tag acima gera: **`Saturday, October 5, 2016`**

Pode-se também usar um padrão

```
<f:convertDateTime pattern="EEEEEEEE, MMM dd, yyyy" />
```

ou aplicar um Locale

```
<f:convertDateTime dateStyle="full" locale="pt" timeStyle="long" type="both" />
```

O exemplo acima produz:

`Quarta-feira, 5 de outubro de 2016 07:15:04 GMT-04`

A tabela abaixo lista alguns atributos de *f:convertDateTime*:

dateStyle	Baseado em java.text.DateFormat. Pode ser default, short, medium, long, full.
locale	Default é FacesContext.getLocale(). Pode ser um Locale ou String (“pt”, “pt-BR”, “en”, “fr”, “ru”, etc.)
pattern	Formatação baseado em padrão java.text.DateFormat. Este atributo anula o efeito de dateStyle, timeStyle e type.
timeStyle	Baseado em java.text.DateFormat. Pode ser default, short, medium, long, full.
type	Pode ser date, time ou both (default é date)

6.3 Conversão numérica e de moeda

NumberConverter permite converter números, moedas, aplicar locale, formatar, etc. Os recursos também são similares aos disponíveis nas classes do pacote *java.text*. Considere o fragmento a seguir onde *total* é do tipo *BigDecimal*:

```
<h:outputText value="#{pedido.total}">
   <f:convertNumber currencyCode="BRL" pattern="#,###.##"
                    type="currency" locale="pt"/>
</h:outputText>
```

Quando o *outputText* for renderizar o valor *320.0*, recebido em formato *BigDecimal*, ele será convertido na string *“R$320,00”*. Usando em um campo de entrada, o recebimento do texto *“R$1.230,55”* causará a conversão para *BigDecimal.valueOf(1235.55)*

(automaticamente trocando a vírgula pelo ponto, uma vez que o Locale escolhido usa outra representação).

Alguns dos atributos de *f:convertNumber* estão listados abaixo:

currencyCode	Código ISO 4217 para moeda: USD, BRL, EUR, RUB, CNY, JPY, etc.
currencySymbol	"$", "¥", "£", "R$", "₩", "€"
locale	Default é FacesContext.getLocale(). Pode ser um Locale ou String ("pt", "pt-BR", "en", "fr", "ru", etc.)
pattern	Formato (baseado em java.text.DecimalFormat)
type	number (default), currency ou percent.

6.4 Conversores customizados

Pode-se criar conversores para outros tipos use a interface *javax.faces.convert.Converter*, e implemente os métodos *getAsObject()* e *getAsString()*, que servem para converter o objeto em uma representação unívoca, e vice-versa. O conversor é registrado com um ID, que é fornecido como argumento para a anotação *@FacesConverter*, usada na classe (também é possível registrar o conversor em *faces-config.xml*). Para usar, deve-se referenciar o ID em *<f:converter>*.

O conversor abaixo traduz no método *getAsString()* um objeto Filme em uma representação String (seu código IMDB, que é único). No método *getAsObject()* ele usa o código IMDB para realizar um query no banco e recuperar o objeto:

```
@FacesConverter("filmeConverter")
public class FilmeConverter implements Converter {

    @Inject
    private FilmDAO filmDatabase;

    @Override
    public Object getAsObject(FacesContext ctx, UIComponent comp, String imdb) {
            if(imdb == null || imdb == "") return null;
            return filmDatabase.findByIMDB(imdb);
    }

    @Override
    public String getAsString(FacesContext ctx, UIComponent comp, Object objFilme) {
            if(objFilme == null) return "";
            return ((Filme)objFilme).getImdb();
    }
```

```
}
```

O conversor está sendo usado dentro de um componente <h:selectOneMenu> para que seja possível exibir filmes e gravar o valor selecionado como objeto:

```
<h:selectOneMenu id="menu" value="#{cinemateca.filme}">
    <f:selectItem itemLabel="Selecione" noSelectionOption="true" />
    <f:selectItems value="#{cinemateca.filmes}"
                   var="filme"
                   itemLabel="#{filme.titulo}"
                   itemValue="#{filme}"/>
        <f:converter converterId="filmeConverter" />
</h:selectOneMenu>
```

Este outro componente lê o objeto e imprime sua representação String (que é apenas o IMDB do filme). Sem o Converter, ele usaria o *toString()* do objeto:

```
<h:outputText id="imdb" value="#{cinemateca.filme}">
    <f:converter converterId="filmeConverter"/>
</h:outputText>
```

7 Listeners

Listeners são usados para tratar e processar *eventos*. Um listener precisa registrar-se com uma fonte produtora de *notificações*. Quando um evento ocorre, o listener é notificado e recebe um objeto contendo informações sobre o evento, e tem a oportunidade de executar e realizar qualquer processamento disparado pelo evento. Para criar um listener de eventos disparados por componentes JSF é preciso criar e registrar uma classe ou um método.

Vários tipos de eventos ocorrem durante a execução de uma aplicação JSF: eventos disparados por componentes da interface do usuário, eventos de Ajax, eventos de ação HTTP, eventos do sistema e eventos de modelo de dados. Desses, os mais importantes são:

- *Action events* – lidam com envio de formulários. São lançados *após* preenchimento do bean (fase 5) ou após validação (fase 3) e retornam strings que identificam regras de navegação.
- *UI events* – apenas afetam a interface do usuário. Geralmente lançados antes do bean ser preenchido (fase 2, 3 ou 5 – depende do tipo e do atributo *immediate*) e ignoram lógica de validação. Esses eventos nunca afetam (diretamente) a navegação.

Os eventos disparados são úteis se são capturados por listeners que devem se cadastrar para recebe-los e serão notificados quando (ou se) o evento ocorrer. Os listeners devem implementar interfaces relacionadas ao tipo de evento que se deseja tratar.

7.1 Listeners de eventos disparados por componentes (UI)

Existem duas interfaces:

- *ActionListener*: usado para capturar eventos disparados por botões, links, etc. – enviam automaticamente o formulário ao final do processamento.
- *ValueChangeListener*: para capturar eventos disparados por checkboxes, radio, combos, etc – não enviam o formulário.

Como foi mostrado no capítulo sobre managed beans, pode-se escrever o handler de eventos no próprio managed bean e não precisar implementar uma interface. Isto é suficiente se o evento precisar de apenas um listener, tiver necessidade de ter acesso ao estado interno do bean e não ter potencial de reuso.

Listeners também podem ser registrados através dos tags *<f:actionListener>* ou *<f:valueChangeListener>*. Neste caso é necessário escrever uma classe para implementar a interface correspondente. As interfaces são:

```
public interface ActionListener {
   void processAction(ActionEvent event);
}

public interface ValueChangeListener {
   void processValueChange(ValueChangeEvent event) ;
}
```

Desta forma é possível registrar vários listeners para um objeto. No exemplo abaixo várias implementações de ActionListener são registradas para um componente:

```
<h:commandButton value="Button 1" id="btn1">
     <f:actionListener type="br.com...RegistroDeClique" />
     <f:actionListener type="br.com...NotificarUsuarios" />
     <f:actionListener type="br.com...GravarLog" />
</h:commandButton>
```

Esta é uma das implementações, que usa *ActionEvent.getComponent().getComponent()* para obter dados do componente que causou o evento:

```
public class RegistroDeClique implements ActionListener {
     @Inject Bean bean;
     @Override
     public void processAction(ActionEvent evt) throws AbortProcessingException {
             String componenteId = evt.getComponent().getId();
             bean.addClicked(componenteId);
     }
}
```

Este outro exemplo, incluído em um menu, obtém de *ValueChangeEvent* detectar mudança no estado do componente. :

```
public class RegistroDeAlteracao implements ValueChangeListener {

    @Override
    public void processValueChange(ValueChangeEvent evt)
                            throws AbortProcessingException {
            Filme antigo = (Filme)evt.getOldValue();
            Filme novo   = (Filme)evt.getNewValue();
            System.out.println("Filme anterior: " + antigo);
            System.out.println("Filme atual: " + novo + "\n");
    }
}
```

Um listener já está sendo chamado no bean através do atributo *valueChangeListener*, portanto para que seja possível incluir um *segundo* listener, o componente usou *<f:valueChangeListener>:*

```
<h:selectOneMenu id="menu" value="#{cinemateca.filme}"
                valueChangeListener="#{bean.processadorValueChange}">
   <f:selectItem itemLabel="Selecione" noSelectionOption="true" />
   <f:selectItems value="#{cinemateca.filmes}" ... />
   <f:converter converterId="filmeConverter" />
   <f:valueChangeListener type="br.com...RegistroDeAlteracao" />
</h:selectOneMenu>
```

7.1.1 Eventos que pulam etapas

Em listeners que precisam alterar apenas a interface gráfica (e não enviar dados) é necessário incluir o atributo *immediate="true"* para forçar a execução na fase 2 (antes da validação e preenchimento dos beans). Se isto não for feito a expressão só será executada após a validação.

Por exemplo, no formulário abaixo o botão *Cancelar* usa *immediate=true*, para pular diretamente para a fase de renderizar página (6) sem precisar validar os campos do formulário (que não faz sentido para um cancelamento). Mais sobre *immediate="true"* no capítulo sobre arquitetura.

```
<h:form id="transporte">
   Código <h:inputText value="#{transporte.codigo}" id="codigo"/>
   Destinatário <h:inputText value="#{transporte.destinatario.nome}" id="nome"/>
   Cidade <h:inputText value="#{transporte.destinatario.cidade}" id="cidade"/>
   <h:commandButton action="#{transporte.validarEnviar}" value="Enviar pedido" />
    <h:commandButton action="#{transporte.cancelar}" value="Cancelar"
                                                     immediate="true"/>
</h:form>
```

7.2 Eventos do sistema e ciclo de vida

Uma das formas de depurar aplicações JSF é monitorar o que ocorre em cada fase. É também uma ótima maneira de entender melhor o funcionamento do JSF. Isto é possível através de listeners que reagem a eventos do sistema e às fases do JSF.

7.2.1 PhaseListener

Para monitorar as fases de uma aplicação inteira, deve-se implementar um *PhaseListener*. Implemente a interface, os métodos de interesse (antes ou depois de cada fase) e as fases a serem monitoradas em *getPhaseId()*:

```
public class ExamplePhaseListener implements PhaseListener {

    @Override
    public void afterPhase(PhaseEvent evt) {
       String clientId   = "form:input1";
       UIViewRoot view   = evt.getFacesContext().getViewRoot();
       UIComponent input = view.findComponent(clientId);

       System.out.println("AFTER PhaseID: " + evt.getPhaseId()
                                           + input.getValue());
    }

    @Override
    public void beforePhase(PhaseEvent evt) {...}

    @Override
    public PhaseId getPhaseId() {
       return PhaseId.ANY_PHASE; // qualquer fase
    }

}
```

Depois registre em faces-config.xml:

```
<faces-config ... version="2.2">
    <lifecycle>
        <phase-listener>br.com...ExamplePhaseListener</phase-listener>
    </lifecycle>
</faces-config>
```

Assim todas as fases irão chamar os métodos do *PhaseListener*, em todos os beans e views da aplicação.

Para associar um *PhaseListener* apenas a um componente, use *<f:phaseListener>*:

```
<h:inputText ...>
    <f:phaseListener type="br.com...ExamplePhaseListener" />
</h:inputText>
```

7.2.2 <f:event>

O tag *<f:event>* permite capturar eventos do sistema e redirecionar para métodos que irão processá-los. Cinco tipos de eventos do sistema podem ser capturados:

- *preRenderComponent* – antes do componente ser renderizado
- *preRenderView* – antes da View ser renderizada
- *postAddToView* – antes do componente ser adicionado à View
- *preValidate* – antes da validação
- *postValidate* – após a validação

Esses identificadores são passados ao atributo type, de *<f:event>*. O atributo listener deve indicar um método do managed bean que irá processar o evento. Por exemplo, para capturar o evento preRenderView:

```
<f:event type="preRenderView" listener="#{bean.antesDaView}" />
```

E no bean:

```
public void antesDaView(ComponentSystemEvent event) {
    System.out.println("PreRenderView");
}
```

8 Validadores

A validação é o processo de verificar que campos necessários estão presentes em um formulário, e se estão no formato correto. Se dados estiverem incorretos a aplicação deve reapresentar o formulário para que o usuário tenha a oportunidade de corrigir os dados. A aplicação deve identificar o problema e descrevê-lo com uma mensagem de erro, preservando os valores que foram inseridos corretamente.

8.1 Tipos e processo de validação

JSF oferece várias alternativas para facilitar esse trabalho

- Validação **explícita**: envolve o uso de tags, métodos validadores, *Bean Validation* ou outros recursos que indicam como a validação deve ser realizada. A validação explícita pode ser "*manual*" através da implementação dos algoritmos de validação escritos em Java e preenchimento das mensagens de erro, ou *automática* através de tags ou Bean Validation.
- Validação **implícita**: ocorre de forma implícita durante a checagem de campos vazios e conversão de tipos nativos. .

A validação implícita verifica *restrições básicas* (ex: se o campo pode ou não ser vazio ou nulo) e *conversão de tipos* (ex: se o tipo digitado tem um formato que pode ser convertido no tipo da propriedade).

A validação explícita requer configuração ou programação, e envolve o uso de tags que definem os constraints, classes que implementam validadores ou métodos com algoritmos de validação.

O fluxo de validação segue uma ordem bem-definida.

1. Validação implícita através da verificação do atributo *required*: se o componente tiver atributo *required="true"* e o campo de dados contiver valor *vazio* ou *null*, uma violação é gerada.
2. Validação implícita através da *conversão de tipos*: se um tipo não puder ser convertido, uma violação é gerada.
3. Validação explícita através da execução de *validadores*: se houver validadores explícitos eles são executados. A validação pode ser cancelada se antes o componente mudar o atributo *immediate* para *"true"* (se o componente tiver *binding* no managed bean, pode fazer isto baseado em condições, como eventos).

8.2 Exibição de mensagens

Em todos os casos, *mensagens* são geradas quando ocorrem erros de validação. Essas mensagens podem ser apresentadas ao usuário através dos tags *<h:messages>* e *<h:message>*.

Mensagens de erro são colecionadas pelo contexto atual do runtime JSF (FacesContext). Dentro de um método (validação manual) a chamada precisa ser explítica, obtendo o contexto e chamando métodos *addMessage()*:

```
FacesContext ctx = FacesContext.getCurrentInstance();
ctx.addMessage(null, "msg para todo o form");
ctx.addMessage("form1:id1", "msg para elemento id1 dentro de form1");
ctx.getMessageList().size() // quantas mensagens há?
```

Para mostrar as mensagens na View existem dois tags:

- *<h:messages>* - mostra em um único lugar varias mensagens.
- *<h:message for="id">* - mostra mensagem individual para um determinado componente.

Um exemplo clássico é usar *<h:panelGrid>* e *<h:panelGroup>* com três colunas, sendo a primeira para o *label*, a segunda para o *componente* de entrada, e a terceira para *mensagens*, que só irão aparecer durante a validação.

```
<h:panelGrid columns="3">
      Nome: <h:inputText id="n"/> <h:message for="n"/>
      Email: <h:inputText id="e"/> <h:message for="e"/>
</h:panelGrid>
```

8.3 Validação implícita

A conversão é automática e causa validação implícita. O sistema tenta converter automaticamente *Integer*, *Double*, etc. ou um tipo que possui um conversor customizado. Se houver erro, a mensagem é armazenada automaticamente. Mas mesmo componentes que não precisam de conversão podem ser validados implicitamente, simplesmente incluindo um atributo *required="true"*.

Em caso de erro, a mensagem armazenada será uma mensagem default, em inglês ou de acordo com o locale/bundle usado. O atributo *requiredMessage* pode ser usado para customizar a mensagem de erro. O tag *<h:message>* deve ser posicionado no local onde deseja-se mostrar a mensagem. A mensagem só é renderizada se houver erro. Normalmente se adiciona um atributo style ou styleClass, para que seja possível configurar o estilo da mensagem de erro:

```
<h:inputText value="#{pacoteBean.codigo}"
     required="true"
     requiredMessage="É necessário fornecer um código"
     id="codigo"/>
<h:message for="codigo" style="color:red"/>

<h:inputText value="#{pacoteBean.peso}"
     required="true"
     requiredMessage="Peso é obrigatório" id="peso"/>
<h:message for="peso" style="color:red"/>
```

Uma alternativa ao atributo required é a inclusão do tag *<f:validateRequired>* dentro do elemento que se deseja validar.

8.4 Validação explícita

8.4.1 Validação manual com método validator ou action

A validação explícita pode ser "manual" implementando as regras que testam a validade dos dados em um método disparado por eventos. Pode ser o próprio método de *action* ou um método específico para validação.

Por exemplo, considere as seguintes restrições para um objeto "Pacote":

- Codigo (não pode estar ausente – 3 letras)

- Peso (não pode ser mais que 1000)
- Altura (não pode ser menos que 1 ou mais que 10)
- Largura (não pode ser menos que 1 ou mais que 10)
- Profundidade (não pode ser menos que 1 ou mais que 5)

O método de validação no bean deve testar cada condição e se houver violação acrescentar mensagem (FacesMessage) no contexto. Se houver mais que 0 mensagens no contexto, retornar *null* no método de ação (isto exibe o formulário novamente).

Na página deve-se usar *<h:messages>* se *addMessage()* foi criado com primeiro argumento *null* ou usar *<h:message>* para cada componente se este foi identificado com *ID*.

```
<h:form>
<h:messages/>
…
Codigo: <h:inputText value="#{bean.codigo}"/>
…
```

No exemplo abaixo a validação está sendo feita em um método de *action* e será executado apenas na fase 5 (e não na fase 3, dedicada à validação).

```
public String enviar() {
   FacesContext context = FacesContext.getCurrentInstance();
   if (getCodigo().length() == 3) { // falta checar se sao letras
     context.addMessage(null,
         new FacesMessage("Codigo deve ter 3 caracteres"));
   }
   if (getPeso() < 1000) {
     context.addMessage(null,
         new FacesMessage("Peso deve ser menos que 1000"));
   }
   if (getAltura() >= 1 && getAltura() <= 10) {
     context.addMessage(null,
         new FacesMessage("Altura deve ser entre 1 e 10."));
   }
   if (getLargura() >= 1 && getLargura() <= 10) {
     context.addMessage(null,
         new FacesMessage("Largura deve ser entre 1 e 10."));
    }
    if (context.getMessageList().size() > 0) {
         return(null);
     } else {
         processarEnvio();
         return("mostrarDados");
     }
}
```

A validação também pode ser feita através de um método *validator* criado no managed bean ou em uma classe separada (usando <f:validator>). Essas alternativas são melhores pois respeitam o ciclo de vida natural do JSF (o método é chamado na fase 3).

No facelet de um componente o atributo *validator* é usado para informar o nome do método. Este método tem uma assinatura definida (é a mesma declarada do método da interface *javax.faces.validator.Validator*.) O método deve causar *ValidatorException* com o *FacesMessage* como parâmetro se a validação falhar, e não fazer nada se passar.

```
void validar(FacesContext ctx,
             UIComponent fonte,
             Object dados) throws ValidatorException
```

8.4.2 Validação automática

Existem vários tags de validação que cuidam de validações simples de strings e números, mas também permitem a declaração de validações elaboradas usando expressões regulares, evitando a necessidade de se escrever um método de validação em Java. A validação determinada pelos tags também ocorre na fase de *validação explícita* (após teste *required* e *conversão*). Os tags e seus atributos são:

- *f:validateLength* (minimum, maximum)
- *f:validateDoubleRange* (minimum, maximum)
- *f:validateLongRange* (minimum, maximum)
- *f:validateRegEx* (pattern)

Para usar, esses tags devem ser incluídos dentro dos tags dos componentes a serem validados.:

```
<h:inputText value="#{transporte.destinatario.telefone}" id="telefone">
    <f:validateRegex pattern="^([+]?[\d]+)?$"/>
</h:inputText>
```

8.5 Validators customizados

O tag *<f:validator>* permite que se associe a um componente qualquer validador que implemente a interface *javax.faces.validator.Validator*. Além de implementar a interface, o validador deve ser registrado com um ID, através da anotação *@FacesValidator*.

Por exemplo:

```
@FacesValidator("CEPValidator")
public class CEPValidator implements Validator {

    private Pattern pattern = Pattern.compile("\\d{5}-\\d{3}");
```

```
    @Override
    public void validate(FacesContext ctx, UIComponent comp, Object value)
                                                        throws ValidatorException {
        Matcher matcher = pattern.matcher(value.toString());
       if(!matcher.matches()) {
          FacesMessage msg = new FacesMessage("CEP deve ter o formato NNNNN-NNN.");
          throw new ValidatorException(msg);
       }
    }
}
```

Exemplo de uso:

```
CEP
<h:inputText value="#{transporte.destinatario.endereco.cep}" id="cep">
    <f:validator validatorId="CEPValidator" />
</h:inputText>
```

8.6 Bean Validation

Uma alternativa à validação do JSF é o *Bean Validation*, que pode ser usado em conjunto com os recursos nativos do JSF. Bean Validation pode simplificar a validação "manual" substituindo o algoritmo de validação para casos comuns a uma simples anotação. A API também permite que novas anotações sejam criadas para validadores customizados.

A vantagem de usar *Bean Validation* é que as validações podem ser reusadas e compartilhadas em outras partes da aplicação, e não apenas no JSF, já que é uma API nativa do Java EE 7 e integra com containers Java EE e serviços.

Bean validation é configurado através de anotações nos atributos e métodos do bean. Por exemplo:

```
public class Name {
   @NotNull
   private String firstname;

   @NotNull
   private String lastname;
}
```

Em ambientes CDI, Bean Validation está habilitado por default. Em outros ambientes, ele pode ser usado com JSF incluindo a tage *<f:validateBean>* no componente que se deseja validar com a Bean Validation API:

```
<h:inputText value="#{bean.email}">
   <h:validateBean />
</h:inputText>
```

9 Templates

Um *template* pode conter a estrutura básica de várias páginas, permitindo o reuso de estrutura, e garantindo um estilo consistente para uma aplicação ou website. Combinado com recursos como imagens, CSS e JavaScript, permite tratar partes de uma View como componentes configuráveis, evitando duplicação de código e simplificando o desenvolvimento.

9.1 Por que usar templates?

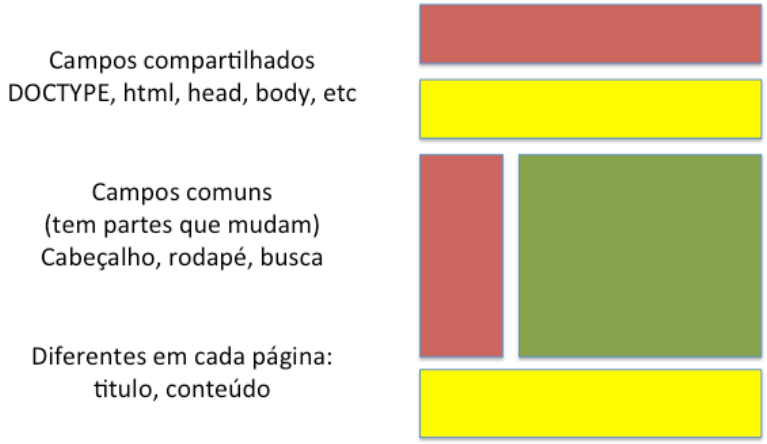
Templates existem desde as primeiras implementações de tecnologias de servidor em HTTP. Alguns exemplos de soluções similares incluem:

- Server-side includes (1994)
- Tiles (Struts)
- *@include* e *<jsp:include>* em HTTP

Templates servem para criar *composições*, que são páginas que reusam uma estrutura similar e alteram sua aparência através de configuração. Use templates para

- Evitar repetir o mesmo código
- Manter consistência de apresentação
- Facilitar a aplicação de estilos
- Facilitar o uso em dispositivos diferentes

Páginas de um site costumam ter áreas quase *estáticas*, que se repetem quase sem alteração. Por exemplo, talvez todas as páginas carreguem as mesmas folhas de estilo e scripts, e tenham uma estrutura de <head>/<body> muito semelhante, um rodapé, um menu superior. Essa estrutura poderia ser definida em um template.



Outras seções talvez tenham pequenas *alterações*, mas preservem semelhanças de estrutura. Por exemplo, um menu contextual, sempre localizado à direita, mas refletindo o contexto de uma seção do site, o estilo de uma seção, que altera fontes e cores, ou mesmo o comportamento diferenciado, de uma seção que mostra um álbum de imagens, omite algumas estruturas e altera outras radicalmente.

Finalmente é provável que haja áreas da página que sejam sempre diferentes, com texto diferentes, ou outras estruturas como formulários, imagens, recursos interativos, etc.

Através de um design cuidadoso do site, buscando identificar estruturas que possam ser reusadas, pode-se identificar as partes estáticas, e que podem ser reusadas *sem alterações*, seções que se repetem com *poucas alterações*, e seções que são *diferentes em cada página*. A partir dessas informações podemos criar um ou mais templates e determinar *pontos de inserção*, onde serão incluídos os fragmentos que irão compor a View final.

O recurso de componentes compostos do JSF (composite components), que é abordado em outra seção, tem semelhanças com templates, mas templates são mais simples. Templates também podem ser usados para criar componentes, mas com menos recursos que composite components.

9.2 Templates em JSF

JSF oferece uma coleção de tags criados com a finalidade de construir templates e componentes reutilizáveis, de forma declarativa, sem a necessidade de programação em Java. Tags usados na construção de templates fazem parte da biblioteca padrão *"http://java.sun.com/jsf/facelets"* e usam normalmente o prefixo *ui*. Alguns tags dessa biblioteca, como foi mostrado anteriormente, não servem apenas para templating mas podem ser usados em outras situações (ex: *<ui:repeat>*).

JSF oferece *duas estratégias* para construção de templates:

- Páginas comuns usando tags ***<ui:include src="uri do fragmento">*** para inserir partes dinâmicas. Nesta estratégia, os fragmentos que se repetem, como cabeçalhos, menus, rodapés são guardados em fragmentos de XHTML, inseridos em uma página que contém partes dinâmicas através de tags <ui:include>. Os próprios fragmentos podem conter outros fragmentos.
- Páginas de template contendo pontos de inserção usando ***<ui:insert name="id da inserção">***. Nesta estratégia é usada uma página-cliente contendo uma coleção de definições *<ui:define>* (fragmentos de XHTML) para cada inserção que deve ser substituída. Inserções que não estão na coleção de definições mantém seu conteúdo

original. Quando a página cliente é referenciada, ela carrega seu template e substitui os pontos de inserção com suas definições.

É possível combinar as duas, usando <*ui:include*> em páginas de template e dentro de blocos <*ui:define*> das páginas cliente, quando for interessante incluir pequenos fragmentos.

Os fragmentos, apesar do nome, precisam ser páginas XHTML bem formadas. Precisam ter um elemento raiz. Normalmente usa-se <*ui:fragment*>, ou <*ui:component*> que ignora os tags de fora do componente.

Em JSF 2.2 ainda há uma terceira estratégia chamada de "contracts", que é baseada na segunda solução e consiste em reorganizar os templates e seus resources dentro de uma estrutura padrão, dentro pasta *resources* e configurada em *faces-config.xml*, que pode ser recuperada através de um identificador (como "temas" ou "skins")

Os tags de UI relacionados com a construção de templates estão listados na tabela abaixo:

`ui:component` `ui:fragment`	Representam UIComponent. Usados na construção de componentes. <ui:component> é um container genérico que ignora todo o HTML fora do elemento. <ui:fragment> é idêntico, mas considera o conteúdo externo. E geralmente é usado como um container para usar o atributo rendered.
`ui:composition` `ui:decorate`	Representam uma composição ou coleção arbitrária de elementos. <ui:composition> ignora todo o HTML fora do elemento e é frequentemente usado para construir páginas-cliente para templates de página reutilizáveis (contendo uma coleção de <ui:define> e <ui:param>); <ui:decorate> é idêntico mas pode ser usado para construir componentes-cliente inseridos em um contexto, já que não ignora o HTML fora do elemento.
`ui:define`	Define parte de uma página que irá ser inserido em um ponto de inserção de um template. Atributo name indica o nome do ponto de inserção a ser substituído.
`ui:include`	Inclui conteúdo de outra página. Atributo src indica o caminho para o fragmento XHTML, relativo ao contexto.
`ui:insert`	Usado em templates para definir área de inserção. Atributo name especifica um identificador para o ponto de inserção.
`ui:param`	Representa um parâmetro usado em diversas situações onde há necessidade de passar parâmetros (ex: elementos <ui:include>, <ui:decorate>, <ui:composition>, <f:view>, etc.) Este elemento define uma constante que pode ser lida no destino através de uma expressão EL #{nome-do-parametro}.

9.3 Como construir um template

Usando a segunda estratégia descrita neste capítulo, uma página de template deve conter toda a estrutura comum de uma página HTML. Deve-se evitar elementos que não podem ser aninhados (ex: <h:form>).

Elementos cuja estrutura dependem da estrutura dos fragmentos devem ser usados com cautela, pois podem introduzir bugs e limitar a reusabilidade do template se não forem construídos corretamente. Exemplos são estruturas de layout como <h:dataTable>, <h:panelGrid>, <p:layout> do PrimeFaces, ou mesmo estruturas <table> HTML.

Templates devem ser armazenados em resources, já que são destinados a reuso.

O exemplo abaixo mostra o conteúdo de um *arquivo de template* com a estrutura que será compartilhada por todos os clientes. Este arquivo contém a estrutura básica do HTML (inclusive logotipos e arquivos CSS) como parte estática, e sete pontos de inserção possíveis, (destacados) contendo conteúdo *default* e identificados por blocos <*ui:insert*>.

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

<h:head>
  <h:outputStylesheet library="css" name="layout.css" />
  <title><!-- (1) -->
    <ui:insert name="titulo">PteroCargo</ui:insert>
  </title>
</h:head>

<body>
  <!-- (2) -->
  <ui:insert name="cabecalho">
    <header>
       <h:graphicImage library="images" name="pterosaur2.jpg" height="150" />
       <h1>PteroCargo</h1>
       <h2>Transporte de cargas no tempo e no espaço</h2>
      <!-- (3) - pode ser usado apenas se (2) não for substituído -->
      <ui:insert name="menu-superior">
         <ui:include src="menu.xhtml" />
      </ui:insert>
    </header>
  </ui:insert>
  <!-- (4) -->
  <ui:insert name="conteudo-pagina">
      <div id="conteudo">
        <!-- (5) pode ser usado apenas se (4) não for substituído -->
       <ui:insert name="menu-lateral">
          <nav class="menu-lat">
            <ul class="nav">
```

```
                <li>...</li>
                <li>...</li>
                <li>...</li>
              </ul>
            </nav>
        </ui:insert>
        <!-- (6) pode ser usado apenas se (4) não for substituído -->
        <ui:insert name="conteudo-lateral">
            <div id="principal">Conteúdo a ser incluído</div>
        </ui:insert>
      </div>
    </ui:insert>

    <!-- (7) -->
    <ui:insert name="rodape">
        <footer>Copyright (c) PteroCargo Ltda.</footer>
    </ui:insert>
</body>
</html>
```

Três pontos de inserção acima estão aninhados, e só poderão ser usados se o ponto que os contém não for substituído quando o template for usado. O exemplo mostrado também usa *<ui:include>* para inserir um menu (fragmento de XHTML) dentro do terceiro ponto de inserção (que o cliente também pode substituir, se quiser).

O arquivo acima será armazenado em */resources/templates/template.xhtml* para que possa ser referenciado pelas páginas-cliente.

9.4 Como usar templates

O template será usado por um ou mais *arquivos cliente* que irão prover o conteúdo e definir páginas individuais. Esse é o arquivo que define o ponto de acesso Web, e não o template, apesar da estrutura do arquivo ser toda do template.

Os *arquivos cliente* devem usar *<ui:composition>* para especificar o template usado, e conter um ou mais elementos de conteúdo em blocos *<ui:define>*. Tudo o que estiver fora do *<ui:composition>* será ignorado (a estrutura *<head>, <body>,* etc. normalmente é determinada pelo arquivo de template.)

Cada bloco *<ui:define>* contém conteúdo que irá substituir, no template, seções correspondentes marcadas com *<ui:insert>*. A ordem dos elementos *<ui:define>* não é relevante, já que eles serão identificados pelo nome e inseridos nos locais definidos pelo template, podendo inclusive ser repetidos. Basicamente, o *<ui:composition>* em uma página cliente é um conjunto de elementos *<ui:define>* contendo conteúdo de inserção no template.

Abaixo um *exemplo de um cliente* que define apenas três pontos de inserção para o template criado na seção anterior:

```
<body>
<!-- Tudo acima é ignorado -->

  <ui:composition template="resources/templates/template.xhtml">

    <ui:define name="conteudo-pagina"> <!-- Substitui (4) -->
       <div>
       <h3>Lorem</h3>
       <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
       <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
       <h3>Ipsum</h3>
       <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
       <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
       </div>
    </ui:define>

    <ui:define name="titulo">Página principal</ui:define> <!-- Substitui (1) -->
    <ui:define name="cabecalho"/> <!-- Substitui/elimina (2) -->

  </ui:composition>

  <!-- Tudo abaixo é ignorado -->
</body>
```

O conteúdo default usado em *<ui:insert>* é *preservado* se o arquivo-cliente não o substituir. Elementos que o cliente deseja eliminar, devem ser referenciados em elementos *<ui:define>* vazios. No exemplo acima o *cabecalho* foi eliminado, o *titulo* e *conteudo-pagina* do cliente foram inseridos. Os outros pontos de inserção foram mantidos (exceto os aninhados). Um documento JSF equivalente ao resultado da aplicação do template, destacando os trechos substituídos, está listado abaixo:

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

<h:head>
  <h:outputStylesheet library="css" name="layout.css" />
  <title><!-- (1) -->
    Página principal
  </title>
</h:head>

<body>
  <!-- (2) -->
  <!-- (4) -->
        <div>
        <h3>Lorem</h3>
```

```
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
            <h3>Ipsum</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit, .</p>
            </div>

        <!-- (7) -->
            <footer>Copyright (c) PteroCargo Ltda.</footer>
    </body>
    </html>
```

Combinando templates e arquivos cliente é possível obter vários níveis de compartilhamento:

- Conteúdo compartilhado por *todos* os clientes deve ser colocado no *arquivo de template*.
- Conteúdo para um *cliente específico* deve ser colocado no corpo de um *ui:define* no *arquivo-cliente*.
- Conteúdo que é compartilhado por *alguns clientes* (*fragmento*) deve ser colocado em arquivo separado para ser incluído em um arquivo cliente através de um *ui:include no corpo de um ui:define*.

9.5 Contratos

Contratos são uma forma de organizar templates como *temas*, que podem ser usados e reusados com facilidade, selecionados e configurados através de metadados. Permite a seleção automática de templates para finalidades diferentes (ex: mobile, desktop).

Para configurar, é preciso definir *subpastas* dentro de uma pasta chamada *contracts*, na raiz da aplicação Web. Cada subpasta é o nome de um contrato, e ela deve conter um template e os recursos usados (ex: pastas com CSS, imagens, scripts). Todos os templates de um contrato devem ter o mesmo nome. Por exemplo:

```
WEB-INF/faces-config.xml
contracts
    tema1/
        template.xhtml
        css/
        images/
    tema2/
        template.xhtml
        css/
        images/
index.xhtml
mobile/index.xhtml
```

Os contratos são mapeados ao padrão de URL no *faces-config.xml*, para que possam ser selecionados automaticamente:

```
<faces-config ...>
...
   <application>
      <contract-mapping>
         <url-pattern>*</url-pattern>
         <contracts>tema1</contracts>
       </contract-mapping>
       <contract-mapping>
         <url-pattern>/mobile/*</url-pattern>
         <contracts>tema2</contracts>
       </contract-mapping>
    </application>
</faces-config>
```

A seleção pode ser realizada através de ação do usuário ou evento.

10 Componentes

É possível fazer muita coisa com os componentes já existentes no JSF, e, se forem necessários componentes adicionais para aparência e comportamento existem várias bibliotecas disponíveis, como por exemplo o PrimeFaces e OmniFaces, por exemplo. Mesmo assim, pode ser necessário criar um componente customizado para funcionalidades exclusivas da aplicação Web que se está criando. O JSF oferece desde estratégias simples, que não requerem programação, a soluções mais elaboradas de baixo nível que envolvem a implementação de UIComponent.

10.1 Quando usar um componente customizado

Grande parte da API de componentes em JSF existe desde as primeiras versões do JSF. Em JSF 1.x não era incomum criar componentes programando e configurando Render Kits com implementações customizadas de UIComponent. Mas JSF 2.x oferece várias outras alternativas. Em JSF 2.x *não* é preciso programar um componente novo para:

- Manipular o estado ou funcionalidades de um componente (pode-se usar managed-beans com mapeamento de valor e binding)
- Conversão, validação ou listeners customizados (pode-se criar objetos customizados simplesmente estendendo uma interface, registrando com anotações e inserindo nos componentes via tags padrão)
- Agregar componentes para criar um componente composto (é possível fazer isto usando templates ou criando um composite component com facelets)

Finalmente, se ainda houver necessidade de componentes para suportar recursos como frames, HTML5, mobile, etc., veja se alguma extensão do JSF (como PrimeFaces ou OmniFaces) já não faz o que você quer.

Pode ser justificável investir na criação de um componente UIComponent se houver necessidade de explorar todos os recursos de um componente externo (ex: suporte DOM, eventos e CSS para componentes SVG). Se o objetivo é incluir suporte mínimo a um componente ou widget externo (ex: adaptar um componente JQuery), pode-se usar facelets.

10.2 Criação de componentes em JSF

Há três formas de criar componentes customizados em JSF 2.2.

- A forma mais complexa requer programação em Java, para criar um novo *UIComponent*. Depois pode ser necessário criar uma implementação de *Renderer* para adaptar o componente a um tag, e um *Tag Library Descriptor* (TLD) para especificar o tag e associá-lo com a implementação do UIComponent.
- Uma forma intermediária é construir um tag customizado. Tags customizados não são mapeados a componentes UI. São basicamente geradores de HTML que são processados de forma imediata (na primeira fase do ciclo de vida). É possível cria-los usando apenas facelets, mas eles também requerem uma configuração em XML (TLD), para declarar regras de uso e atributos.
- A terceira forma (que será explorada nesta seção), consiste na criação de componentes verdadeiros usando apenas facelets sem a necessidade de nenhuma configuração adicional.

Neste tutorial mostraremos como construir componentes usando esta terceira estratégia.

10.3 Componentes compostos

Os tags usados para construir componentes fazem parte do namespace

```
xmlns:composite="http://java.sun.com/jsf/composite"
```

que geralmente é declarado com o prefixo *cc* ou *composite*: Os tags mais importantes desta coleção estão brevemente resumidos abaixo:

`cc:interface`	Declara a estrutura (contrato) de um componente (conteúdo, atributos, etc) – como o tag vai ser usado
`cc:implementation`	Define a implementação (deve estar associada a um bloco cc:interface) – o que o tag vai mostrar

cc:actionSource	Permite declarar o componente como alvo de objetos associados (ex: listeners) compatíveis com a implementação de ActionSource2
cc:valueHolder cc:editableValueHolder	Permitem declarar componente como alvo de objetos associados (ex: conversores, validadores) compatíveis com a implementação de ValueHolder
cc:attribute	Declara os atributos do componente
cc:insertChildren	Adiciona elementos-filho no componente

10.3.1 Nome do tag e implementação

A criação de componentes compostos não requer o uso de TLD ou qualquer configuração XML. A meta-informação contida nesses arquivos é geralmente usada para declarar nome da biblioteca, namespace, tag, tipo e restrições de uso para atributos. Componentes compostos fornecem essas informações no próprio tag e na estrutura usada para guardar os arquivos dentro do contexto.

O nome da *taglib* é obtida automaticamente através do nome da *library*: pasta onde o componente é armazenado, dentro da pasta *resources*. E o nome do tag é obtido do nome do arquivo XHTML, armazenado dentro da pasta. Por exemplo, na biblioteca *geocomp* abaixo foram definidos três tags: *coords*, *mercator* e *qgis*:

```
WEB-INF/
resources/
   geocomp/
      coords.xhtml
      mercator.xhtml
      qgis.xhtml
      hello.xhtml
```

Cada tag definido no arquivo .xhtml consiste no mínimo de um bloco cc:implementation (se o tag não tiver atributos):

```
<cc:implementation>
     Hello, world!
</cc:implementation>
```

Uma página-cliente que usa o componente declara o namespace padrão que identifica a biblioteca, e associa um prefixo:

```
xmlns:geo="http://java.sun.com/jsf/composite/geocomp"
```

Em seguida, o tag pode ser usado:

```
<geo:hello/>
```

Um tag, portanto, pode ser usado para reusar coleções de componentes menores que formam um componente composto, reusável. Por exemplo, suponha uma página que contenha um componente que consiste de duas caixas de entrada de texto, para coordenadas geográficas:

```
<h:form>
  <h:panelGrid columns="3">
    <h:outputText value="Latitude:" />
    <h:inputText value="#{locationService.location.latitude}" id="lat" />
    <h:message for="lat" style="color: red" />

    <h:outputText value="Longitude:" />
    <h:inputText value="#{locationService.location.longitude}" id="lon" />
    <h:message for="lon" style="color: red" />
  </h:panelGrid>

  <h:commandButton actionListener="#{locationService.register}"
                   id="locBtn" action="map" value="Localizar" />
</h:form>
```

Se a mesma caixa é usada em várias partes do site, com dados idênticos, ela pode ser reusada usando um componente:

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

<cc:interface>
</cc:interface>

<cc:implementation>
  <h:form>
    <h:panelGrid columns="3">
      <h:outputText value="Latitude:" />
      <h:inputText value="#{locationService.location.latitude}" id="lat" />
      <h:message for="lat" style="color: red" />

      <h:outputText value="Longitude:" />
      <h:inputText value="#{locationService.location.longitude}" id="lon" />
      <h:message for="lon" style="color: red" />
    </h:panelGrid>

    <h:commandButton actionListener="#{locationService.register}"
      id="locBtn" action="map" value="Localizar" />
  </h:form>
</cc:implementation>
</html>
```

E inserida na páginas em que for usada:

```
<body>
```

```
<h1>Coordenadas</h1>
    <geo:coords />
</body>
```

10.3.2 Atributos

O bloco <*cc:interface*> define o o contrato de uso do tag, que consiste dos seus atributos que mapeiam valores e comportamentos associados. Atributos são declarados com <*cc:attribute*>.

```
<cc:interface>
    <cc:attribute name="nomeDoAtributo"/>
</cc:interface>
```

Dentro de <cc:implementation> é possível referenciar o valor dos atributos através de EL e da referência **cc**, que dá acesso ao componente que está mapeado ao tag (é uma implementação de *UIContainer* chamada de UINamingContainer, e que é fornecida pelo runtime JSF). Abaixo estão algumas das propriedades acessíveis via *UINamingContainer*:

`#{cc.attrs}`	Mapa de atributos
`#{cc.clientId}:`	O id do elemento (da forma form:id)
`#{cc.parent}:`	Permite acesso ao componente pai, e recursivamente a suas propriedades cc (cc.parent.attrs.nomeDoAtributo por exemplo)
`#{cc.children}`	Acesso aos componentes-filho
`#{cc.childCount}`	Número de filhos

Portanto, para ter acesso aos atributos declarados, usamos *cc.attrs*:

```
<composite:implementation>
    ...#{cc.attrs.nomeDoAtributo}...
</composite:implementation>
```

O tag agora pode ser usado com atributo:

```
<geo:hello nomeDoAtributo="..."/>
```

Um atributo pode ser obrigatório, assim como pode limitar seu uso a determinadas finalidades ou componentes. Os seguintes atributos podem ser usados em <*cc:atribute*>:

`name`	Nome do atributo
`required`	Se true, atributo é obrigatório (false é default)

`default`	Valor a ser usado como default para atributos não obrigatórios
`type`	Para atributos que recebem value expressions – o tipo que a expressão deve produzir: <cc:attribute ... type="java.util.Date"/>
`method-signature`	Para atributos que recebem method expressions, contém a assinatura do método que será executado: <cc:attribute method-signature="java.lang.String action()"/>.
`targets`	Usada em atributos com method-signature para identificar os componentes que devem receber a expressão declarada.

No exemplo abaixo, estendemos o tag *<geo:coords>* com alguns atributos, para que um managed bean e propriedades possam ser determinados pelo usuário da forma:

```
<geo:coords  latitude="#{locationService.location.latitude}"
             longitude="#{locationService.location.longitude}"
             actionListener="#{locationService.register}"/>
```

Assim ele poderá ser usado com um bean que guarda coordenadas e registra as coordenadas enviadas:

```
@Named("locationService")
@SessionScoped
public class LocationServiceBean implements Serializable {
     @Inject private Location location;
     @Inject @GIS EntityManager em;

     @Transactional
     public void register(ActionEvent evt) {
         em.persist(location);
         System.out.println("Location " + location + " registered!");
     }
     public Location getLocation() {
         return location;
     }
     public void setLocation(Location location) {
         this.location = location;
     }
}

@Entity
public class Location implements Serializable {
     private String latitude  = "-23";
     private String longitude = "-46";
    ...
}
```

Para isto, declaramos a seguinte interface, para permitir o registro de um ActionListener:

```
<cc:interface>
     <cc:attribute name="longitude" required="true"/>
     <cc:attribute name="latitude" required="true"/>
     <cc:attribute name="actionListener"
                   method-signature="void register(javax.faces.event.ActionEvent)"
                   targets="form1:locBtn"/>
</cc:interface>
```

O ID usado em *targets* é usado para referenciar o botão, na *implementation*:

```
<cc:implementation>
     <h:form id="form1">
            <h:panelGrid columns="3">
                   <h:outputText value="Latitude:" />
                   <h:inputText value="#{cc.attrs.latitude}" id="lat" />
                   <h:message for="lat" style="color: red" />

                   <h:outputText value="Longitude:" />
                   <h:inputText value="#{cc.attrs.longitude}" id="lon" />
                   <h:message for="lon" style="color: red" />
            </h:panelGrid>
            <h:commandButton id="locBtn" action="map" value="Localizar" />
     </h:form>
</cc:implementation>
```

Os objetos acessíveis via cc não se limitam apenas aos declarados em <cc:interface>. Como o UINamingComponent é um UIComponent, ele também tem acesso a objetos implícitos que estão instanciados neste componente, por exemplo:

```
<composite:implementation>
  ...
    <p>Host da requisição: #{request.remoteHost}</p>
</composite:implementation>
```

Como a intenção ao criar componentes compostos é o reuso, é importante seguir boas práticas da construção de artefatos reutilizáveis, como facilidade de uso da API, nomes auto-documentáveis, etc. Siga padrões e seja consistente. Por exemplo: crie atributos com nomes padrão (*styleClass*, *required*, etc.) e chame o principal atributo de "*value*".

O componente mostrado como exemplo contém um <h:form>. Como elementos <h:form> não podem ser aninhados, usar <h:form> em um componente pode limitar o seu uso (este componente, por exemplo, não pode ser usado dentro de um <h:form>.

O problema é que o cliente-ID usado no atributo actionListener *referencia* o *<h:form>*. Neste caso a solução é simples. Remover o *<h:form>* e associar o ID diretamente usando *:id*:

```
<cc:interface> ...
     <cc:attribute name="actionListener"
                   method-signature="void register(javax.faces.event.ActionEvent)"
                   targets=":locBtn"/>
```

```
</cc:interface>

<cc:implementation> ...
     <h:commandButton id="locBtn" action="map" value="Localizar" />
</cc:implementation>
```

Agora o componente precisa dserr usado dentro de um <h:form>:

```
<h:form id="f1">
     <geo:coords latitude="#{locationService.location.latitude}"
                 longitude="#{locationService.location.longitude}"
                 actionListener="#{locationService.register}"/>
</h:form>
```

10.3.3 Componente mapeado a classe UINamingContainer

Um binding entre o componente composto e uma classe Java oferece a possibilidade de usar recursos da linguagem e permite que componentes compostos tenham acesso a recursos que só seriam possível implementando *UIComponent*.

Para realizar o binding é preciso implementar *UINamingContainer* e dar um nome a ele com uma anotação *FacesComponent*:

```
@FacesComponent("componenteGeolocation")
public class ComponenteGeolocation
                    extends UINamingContainer { ... }
```

E informar o componente ao declarar a interface:

```
<cc:interface componentType="componenteGeolocation">
 …
</cc:interface>
```

11 Ajax

Ajax (Asynchronous JavaScript and XML) é uma tecnologia que utiliza-se de JavaScript para explorar uma requisição HTTP assíncrona (geralmente chamada de *XMLHttpRequest*). A aplicação JavaScript envia um *XMLHttpRequest* para o servidor, espera obter a resposta (geralmente dados em XML ou JSON) e usa esses dados para atualizar o modelo de objetos (DOM) da página, sem que seja necessário carregar uma nova página.

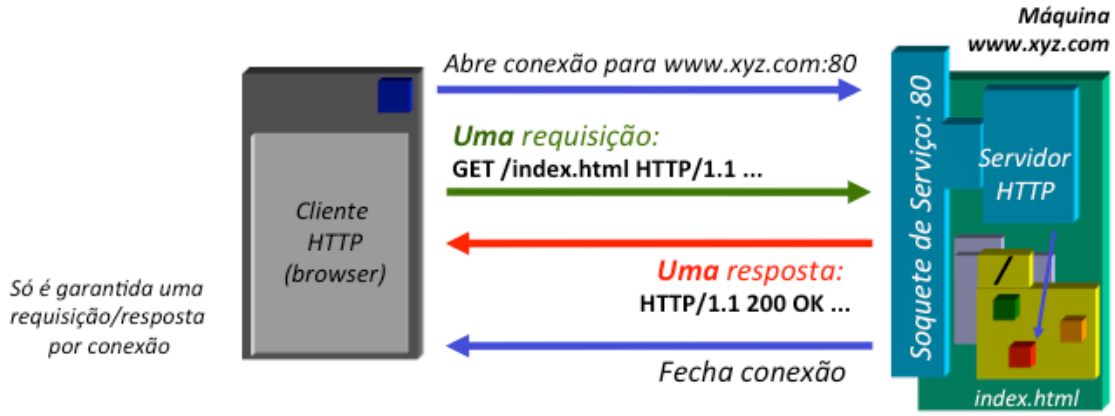
A atualização da página é feita usando recursos dinâmicos de JavaScript e CSS via Document Object Model – DOM, que permite alterar a estrutura da árvore da página. Frameworks como JQuery, JSF e PrimeFaces escondem os detalhes de uma requisição Ajax, que geralmente envolve o envio e recebimento de dados em texto, XML ou JSON em várias etapas, facilitando o seu uso.

O uso de Ajax permite a construção de páginas Web bastante interativas, e melhora a usabilidade das aplicações Web. Por outro lado, o modelo assíncrono aumenta de forma significativa a complexidade das aplicações Web, ja que requisições e atualizações de dados e páginas podem ocorrer em paralelo.

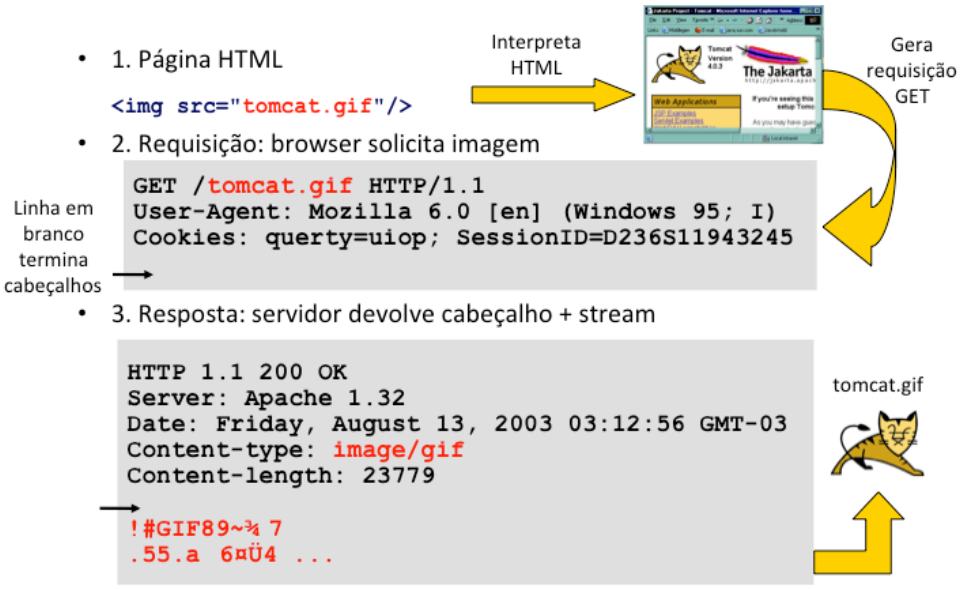
11.1 Por que usar Ajax?

A arquitetura Web é baseada em cliente, protocolo HTTP 1.x e servidor. Tem como principais características um protocolo de transferência de arquivos (HTTP: RFC 2068) que não mantém estado da sessão do cliente, um servidor que representa um sistema de arquivos virtual e responde a comandos representados uniformemente em URIs, e cabeçalhos com meta-informação de requisição e resposta.

A ilustração abaixo mostra um exemplo de requisição e resposta HTTP 1.x.

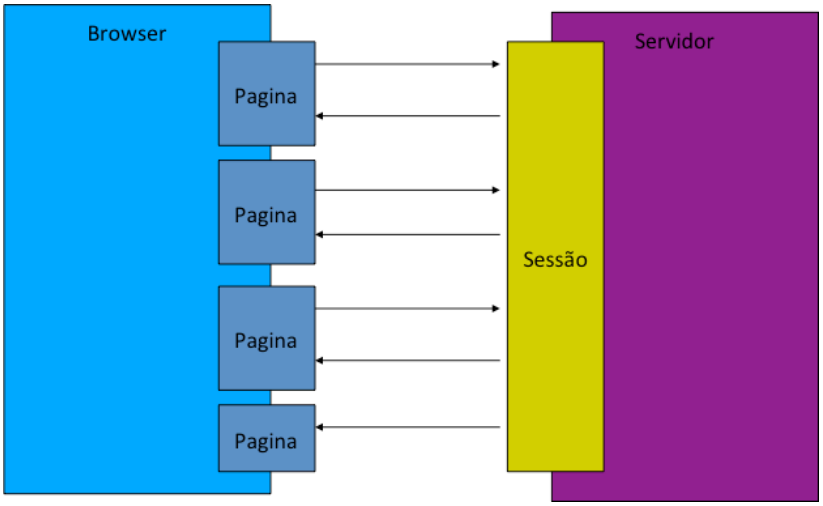


A sintaxe de uma requisição HTTP consiste de uma linha que contém método, URI e versão seguida de linhas de cabeçalho com dados do cliente. A resposta contém versão, código de status e informação de status na primeira linha e cabelalhos com dados do servidor. A ilustração abaixo mostra detalhes de uma requisição/resposta HTTP:



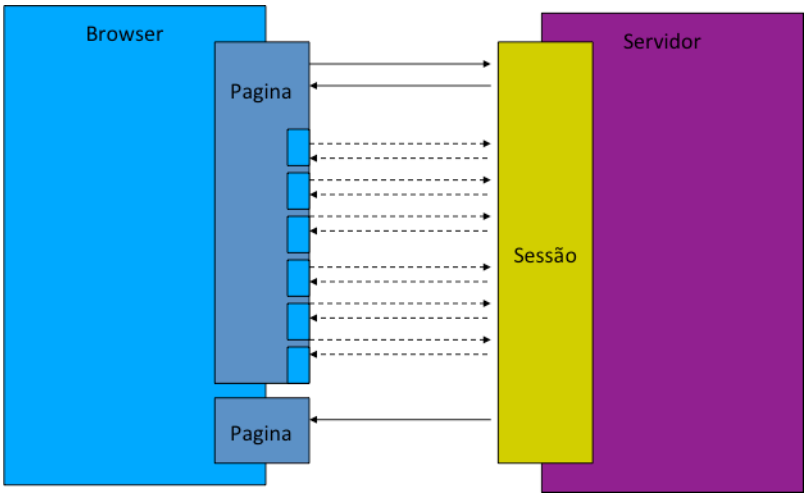
O ciclo de vida de uma aplicação Web convencional associa cada requisição/resposta a uma página. Portanto, *cada vez* que uma requisição é enviada ao servidor, a resposta provoca ou a recarga da mesma página ou o redirecionamento para outra página.

Uma sessão precisa ser mantida artificialmente, através de mecanismos externos ao HTTP, como Cookies, rescrita de URLs ou camadas intermediarias (como SSL). Uma sessão pode preservar o estado entre várias requisições, mas cada requisição devolve uma página.



O ciclo de vida de uma aplicação Ajax (Web 2.0) é diferente. Como as requisições são *assíncronas* e não forçam a recarga da página, é possível receber *fragmentos* de código ou dados (em texto, XML e JSON) para atualizar *parcialmente* a mesma página, alterando o HTML dessa página através de DOM e JavaScript, assim possibilitando a criação de interfaces dinâmicas. Assim várias requisições podem ser feitas sem que a página seja alterada. Ainda assim, o HTTP não preserva o estado da requisição, sendo necessário que a comunicação mantenha uma sessão ativa para que os dados entre as requisições sejam preservados. Com

Ajax, e possível manter o estado da sessão da forma usual (Session, usando cookies) e também com escopo menor limitado à própria página (View, usando a página), já que ela pode preservar estado entre requisições.



A programação de Ajax em JavaScript puro requer várias linhas de código, funções de call-back e monitoração de estados da requisição. Atualmente utilizamos frameworks que simplificam o uso do Ajax oferecendo funções para iniciar requisições, e callbacks para lidar com as respostas assíncronas.

11.2 Características do Ajax no JSF

Em JSF 2.2 (Java EE 7) o Ajax é executado por uma biblioteca JavaScript própria *(jsf.ajax.request())* que contém as funções necessárias para realizar o serviço. Aplicações JSF não precisam lidar diretamente com JavaScript. As requisições são geradas automaticamente e ativadas por eventos dos componentes, e o resultado causa execução e renderização de outros componentes da página. Toda a configuração é feita através do tag *<f:ajax>*.

Qualquer componente capaz de produzir eventos pode ser configurado para ativar requisições Ajax através da inclusão do tag *<f:ajax/>* como elemento filho:

```
<h:inputText value="#{bean.message}">
    <f:ajax />
</h:inputText>
```

O tag sem atributos define *comportamento default* para:

- O evento que dispara a requisição (é o *evento default do componente*, dependendo do tipo; ex: *ValueChangeEvent* para um *inputText*, *selectOneMenu*, etc., *ActionEvent* para *commandButton*, *commandLink*)

- O componente que é executado para produzir os dados da requisição (default é o form no qual o componente esta contido, identificado pelo ID genérico – identificador de agrupamento – *@form*)
- O componente que será atualizado quando os dados de resposta forem recebidos (default é o próprio componente, identificado pelo ID genérico *@this*)

Através de atributos é possível alterar esses defaults.

11.3 Atributos de <f:ajax>

A tabela abaixo lista os atributos que podem ser usados no tag *<f:ajax>*:

`disabled="true"`	Desabilita Ajax (o mesmo que não incluir o tag)
`event=` `"action |` `valueChange |` `click |` `change |` `keyup |` `mouseover |` `focus |` `blur |` `..."`	Evento do DOM a processar. O evento JSF default pode ser action (click) ou valueChange (change). Depende do componente.
`immediate="true"`	Processamento imediato (salta fases do ciclo de vida).
`listener="#{…}"`	Expressão EL mapeada a um método que poderá processar o evento AjaxBehaviorEvent. Este listener é chamado na fase JSF Invoke Application.
`execute="`*id | lista de IDs* `"@all | @form |` `@none | @this |` *#{... EL ...}*"	Componentes cujo estado deverá ser enviado para o servidor junto com a requisição – geralmente implementações de UIInput. @all = página inteira, @form = formulário no qual o componente está contido, @none = nenhum componente, @this = componente atual (default). Lista de IDs ou @nome separados por espaço. Expressão EL que gere IDs/@nomes.
`render="`*id | lista de IDs* `"@all | @form |` `@none | @this |` *#{... EL ...}"*	Componentes a serem renderizados. Mesmo conteúdo do atributo execute. Default é @this.
`onevent="`*JavaScript*"	Função JavaScript que será executada quando ocorrer um evento Ajax. A função recebe um parâmetro que possui uma propriedade status (“begin”, “complete”, “success”) que reflete diferentes fases do processamento Ajax,

	e source, que contém o elemento HTML que causou o disparo do evento.
onerror="*JavaScript*"	Função JavaScript executada em caso de erro da requisição Ajax.

Inclua o elemento *<f:ajax>* dentro de um elemento de componente UI. Caso o componente afetado pela resposta não seja o próprio componente (*@this*), e preciso informar o ID do componente que deve ser atualizado através do atributo *render*:

```
<h:commandButton ... action="...">
    <f:ajax render="id1"/>
</h:commandButton>
<h:outputText ... value="#{...}" id="id1"/>
```

O atributo *render* recebe uma lista de IDs de componentes que precisam ser atualizados, separadas por espaço.

Geralmente não é o componente que dispara o evento quem fornece dados para o servidor, mas esse componente geralmente faz parte do mesmo formulário. Esse é o default (*@form*). Caso outros componentes precisem ser executados, os seus IDs devem ser informados através do atributo *execute*:

```
<h:commandButton ... action="...">
    <f:ajax render="id1 id2" execute="id3 id4" />
</h:commandButton>
```

Nos exemplos acima, quando o botão for pressionado (evento *click*) será enviada uma requisição HTTP assíncrona para o servidor, que conterá como parâmetros os dados dos elementos declarados no atributo *execute*. Na resposta, os dados recebidos são atualizados no bean, e os componentes declarados em *render* são renderizados, fazendo com que seus dados exibidos sejam lidos novamente a partir do bean.

O atributo *event* define o evento que irá disparar a ação. O evento default é o evento do componente. Por exemplo o evento *click* (nome do evento em DOM) equivale a *action* (nome do evento em JSF) e é o evento default de botões.

```
<h:commandButton id="submit" value="Submit">
     <f:ajax event="click" render="result" />
</h:commandButton>
<h:outputText id="result"
                    value="#{userNumberBean.response}" />
```

Em Ajax, o evento JSF *action* representa o evento de clicar um *h:commandButton* ou *h:commandLink*. Ele é traduzido em DOM/JavaScript como *click*. O evento JSF *valueChange* é disparado em *h:inputText*, *h:inputSecret*, *h:inputTextarea*, *h:selectOneMenu*, etc. Ele é traduzido em DOM/JavaScript como *change*.

11.4 Ajax e eventos de ação/navegação

O atributo *action* de um botão refere-se ao evento que gera uma requisição e resposta HTTP sincrona (ele tem relação com o *action* do <*form*> em HTML e o botão do tipo *submit*), e informa um resultado de navegação que ira mudar a View (página) atual. Em uma requisição Ajax, que não retorna outra página, ele não é relevante.

Os métodos de ação *podem* ser chamados através de requisições Ajax, mas como Ajax é assíncrono o valor de retorno não é usado. Normalmente a assinatura de um método de controle retorna um string que é usado para informar a próxima página da navegação:

```
public String metodoDeAcao() {
   ...
   return "pagina";
}
```

Ajax ignora o valor retornado. Métodos que forem disparados por uma requisição Ajax devem declarar *String* como tipo de retorno, ja que são métodos de *action*, mas no final devem retornar *null*.

11.5 Monitoração de eventos Ajax

Eventos Ajax podem ser monitorados através de eventos, que fornecem dados da requisição e resposta HTTP, além de informações de erro. Por ser uma operação assíncrona, a resposta é processada em métodos de *callback* que podem ser configurados através dos atributos *onevent*, *onerror* em JavaScript, e *listener,* em Java.

11.5.1 Eventos Ajax tratados em JavaScript

Usado para obter dados em baixo nível da requisição Ajax, inclua o nome de uma função (*callback*) JavaScript como argumento do atributo *onevent*.

```
<f:ajax render="..." onevent="processar" ... />
```

A funçao será chamada nos varios estágios da requisição/resposta Ajax. Esta função deve ter um argumento, que será usado para obter dados da requisição Ajax:

```
function processar(dados) { … }
```

O argumento pode ter as seguintes propriedades.

- *status* (dados.status) – *begin*, *complete* ou *success*
- *source* (dados.source) – o evento DOM que iniciou a requisição
- *responseCode* – dados retornados em código numérico
- *responseText* – dados retornados em texto (ex: JSON)

- *responseXML* – dados retornados em XML

11.5.2 Eventos Ajax tratados em Java

O atributo *listener* permite redirecionar a um método no servidor o tratamento de uma ação do Ajax no cliente. Esse método é chamado na fase *Aplicação* do ciclo de vida (*AjaxBehaviorListener.processAjaxBehavior*)

Em vez de usar *onevent*, use *listener*:

```
<f:ajax event="change" render="total" listener="#{bean.calcularTotal}"/>
```

11.5.3 Erros de processamento tratados em JavaScript

O atributo *onerror* é similar ao *onevent*, mas ele captura apenas erros. Se houver um erro durante o processamento da requisição Ajax, JSF chama a função definida no atributo *onerror* e passa um objeto de dados.

A função deve ter um argumento:

```
function processar(dadosErro) { … }
```

Propriedades recebidas no argumento dadosErro são as mesmas propriedades do atributo *onevent*, mais:

- *description* (dadosErro.description)
- *errorName* (dadosErro.errorName)
- *errorMessage* (dadosErro.errorMessage)

Valores possíveis para o *dadosErro.status*: são *emptyResponse*, *httpError* (null, undefined, < 200, > 300), *malformedXML* ou *serverError*

11.6 Limitações do Ajax

Nem tudo é possível com <*f:ajax*>. Às vezes é necessário forçar atualizações e recargas de páginas para obter os resultados esperados. Algumas limitações podem ser resolvidas com JavaScript, reescrevendo a forma de atualização ou mesmo usando bibliotecas de terceiros como PrimeFaces.

<*h:outputText value="#{bean.prop}"/*> ou *#{bean.prop}* nem sempre pode ser atualizado com Ajax. Por não ter ID é necessário que esteja dentro de uma árvore de componentes que e renderizada para que seja atualizado.

Acrescentando um ID em <*h:outputText /*>, permite que ele seja identificado em um atributo *render*, mas isto também fará com que seja renderizado dentro de um <*span*>,

impondo limites onde pode ser usado (não é mais texto cru – não pode ser colocado dentro de um atributo ou usado para gerar JavaScript).

Não é possível usar *<f:ajax render="..."/>* para dinamicamente alterar os atributos de elementos HTML, mas é possível fazer isto com funções JavaScript/DOM.

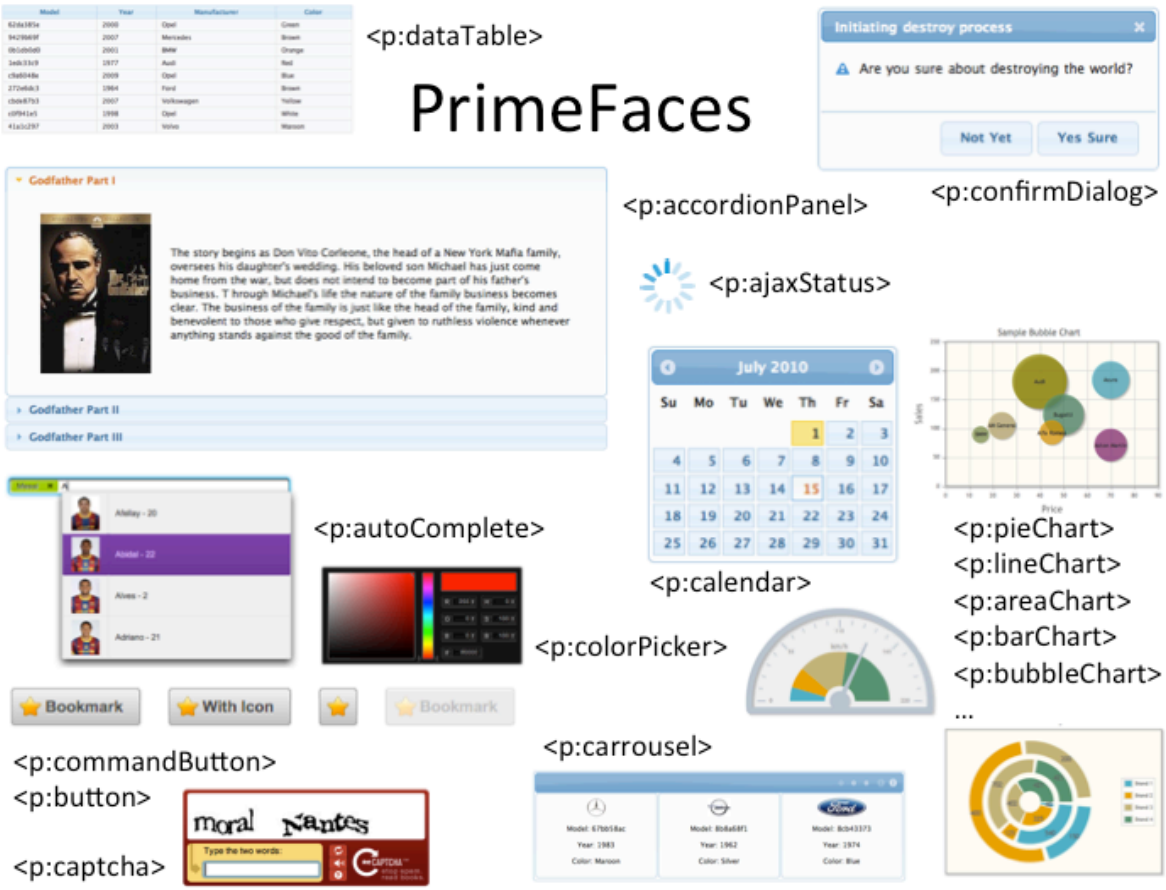
Por fim, Ajax depende de JavaScript, que pode ter comportamento diferente em browsers diferentes (fabricantes, versões e plataformas).

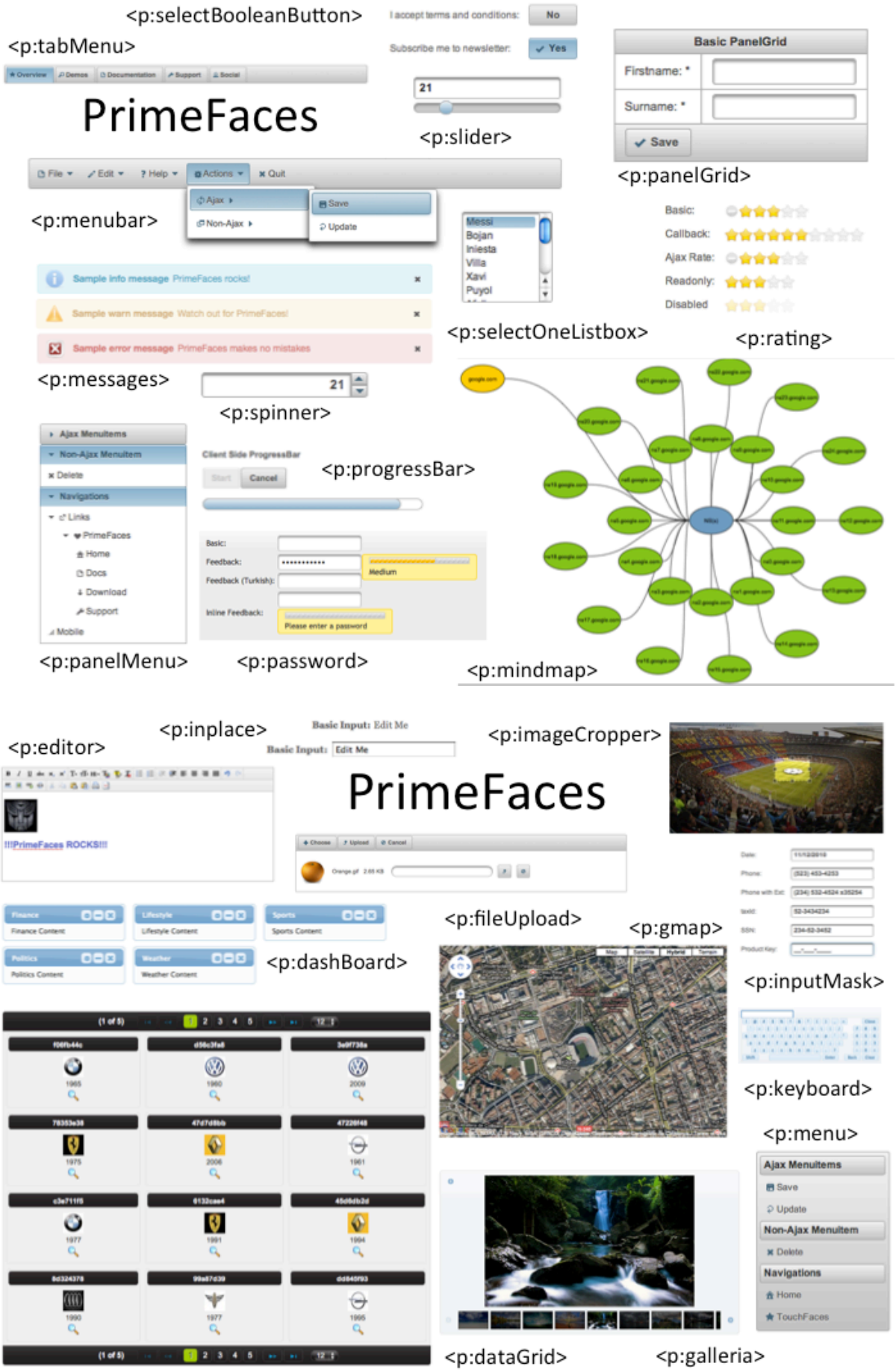
12 Primefaces

PrimeFaces é uma biblioteca de componentes para JSF que possui uma grande coleção de componentes, com suporte integrado a Ajax e websockets (Ajax Push). Vários componentes podem ser usados em substituição aos componentes JSF (também podem ser usados juntos). PrimeFaces integra naturalmente à arquitetura do JSF.

PrimeFaces possui componentes de várias bibliotecas populares. Muitos componentes são wrappers sobre componentes JQuery, YUI, Google e outros. É uma das bibliotecas mais populares para JSF.

As imagens abaixo ilustram diversos componentes do PrimeFaces.




<p:selectBooleanButton>
I accept terms and conditions:
No
Subscribe me to newsletter:
Yes
<p:tabMenu>
PrimeFaces
21
<p:slider>
Basic PanelGrid
Firstname: *
Surname: *
Save
<p:panelGrid>
File
Edit
Help
Actions
Quit
Ajax
Non-Ajax
Save
Update
<p:menubar>
Messi
Bojan
Iniesta
Villa
Xavi
Puyol
Basic:
Callback:
Ajax Rate:
Readonly:
Disabled
<p:selectOneListbox>
<p:rating>
Sample info message PrimeFaces rocks!
Sample warn message Watch out for PrimeFaces!
Sample error message PrimeFaces makes no mistakes
<p:messages>
21
<p:spinner>
Ajax Menuitems
Non-Ajax Menuitem
Delete
Navigations
Links
PrimeFaces
Home
Docs
Download
Support
Mobile
Client Side ProgressBar
Start
Cancel
<p:progressBar>
Basic:
Feedback:
Medium
Feedback (Turkish):
Inline Feedback:
Please enter a password
<p:panelMenu>
<p:password>
<p:mindmap>
<p:inplace>
Basic Input: Edit Me
Basic Input:
Edit Me
<p:editor>
<p:imageCropper>
!!!PrimeFaces ROCKS!!!
PrimeFaces
<p:fileUpload>
<p:gmap>
<p:inputMask>
<p:dashBoard>
<p:keyboard>
<p:menu>
Ajax Menuitems
Save
Update
Non-Ajax Menuitem
Delete
Navigations
Home
TouchFaces
<p:dataGrid>
<p:galleria>

Muitos componentes do PrimeFaces são variações do JSF padrão, mas têm funcionalidade adaptada ao look & feel e comportamentos do Primefaces, além de poderem ter outros atributos e comportamento. Por exemplo:

- *<p:ajax>* (substitui f:ajax em componentes PrimeFaces)
- *<p:commandButton>* e *<p:commandLink>*
- *<p:dataTable>* e *<p:column>*
- *<p:message>* e *<p:messages>*
- *<p:outputLabel>*
- *<p:panelGrid>*
- *<p:fieldset>*
- *<p:inputText>, <p:inputTextArea>, <p:password>, <p:selectOneMenu>,* etc.

12.1 Configuração e instalação

Baixe o arquivo *primefaces-VERSAO.jar* em *www.primefaces.org/downloads.html* ou configure a instalação automática no seu projeto usando *Maven*, incluindo a seguinte dependência no *pom.xml*:

```
<dependency>
   <groupId>org.primefaces</groupId>
   <artifactId>primefaces</artifactId>
   <version>6.0</version>
</dependency>
```

O Primefaces possui um repositório próprio, onde estão as últimas versões (pode ser necessário configurar se for usada uma versão recente):

```
<repository>
   <id>prime-repo</id>
   <name>Prime Repo</name>
   <url>http://repository.primefaces.org</url>
</repository>
```

12.1.1 Teste

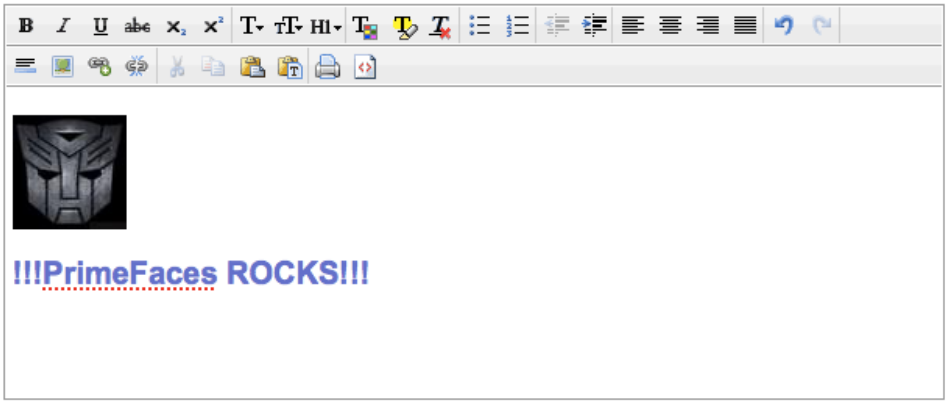
Crie um projeto e uma página XHTML. Inclua o JAR do PrimeFaces como dependência do seu projeto Web (automático se usada a configuração Maven acima: o JAR deve ser distribuído em *WEB-INF/lib*).

Em cada página que usar tags do Primefaces, declare o namespace do primefaces na página de Facelets:

```
xmlns:p="http://primefaces.org/ui
```

Depois escolha um ou mais componentes da biblioteca (veja documentação)

```
<html xmlns="http://www.w3c.org/1999/xhtml"
        xmlns:h="http://java.sun.com/jsf/html"
        xmlns:p="http://primefaces.org/ui">
   <h:head></h:head>
   <h:body>
         <p:editor />
   </h:body>
</html>
```



12.2 Temas (themes, skins)

Temas fornecem um look & feel comum para todos os componentes: fontes, cores, ícones, aparência geral dos componentes. Existem temas gratuitos e pagos. Podem ser baixados de:

- *http://www.primefaces.org/themes.html*
- *http://apps.jsf2.com/primefaces-themes/themes3.jsf*

Para instalar baixe o JAR do tema desejado, distribua-o como dependência do WAR (coloque em *WEB-INF/lib*) e defina um *<context-param>* em *web.xml* com o nome do tema:

```
<context-param>
     <param-name>primefaces.THEME</param-name>
    <param-value>bluesky</param-value>
</context-param>
```

Os temas podem ser alterados em tempo de execução usando *<p:themeSwitcher>* que é um tipo de *<h:selectOneMenu>*. Os temas devem ser incluidos com *<f:selectItem>* ou *<f:selectItems>:*

Exemplo de uso:

```
<p:themeSwitcher>
  <f:selectItem itemLabel="- Escolha um tema -" itemValue=""/>
  <f:selectItems value="#{bean.listaDeTemas}"/>
</p:themeSwitcher>

@Named
public class Bean {
```

```
public String[] getListaDeTemas() {
    String[] temas = { "afterdark", "bluesky", "casablanca", "eggplant", "glass-x"};
    return(temas);
}
```

12.2.1 Configuração de CSS

Os temas podem ser ajustados usando CSS. As principais classes estão listadas abaixo (veja mais na documentação – existem classes específicas para cada componente):

```
.ui-widget (afeta todos os componentes)
.ui-widget-header
.ui-widget-content
.ui-corner-all
.ui-state-default
.ui.state-hover
.ui.state-active
.ui-state-disabled
.ui-state-highlight
.ui-icon
```

Primefaces faz alterações dinâmicas em CSS inserindo tags *<link rel="stylesheet">* no final de *<h:head>*. Se você criar uma folha de estilos CSS para alterar classes do PrimeFaces e carregá-la usando *<link>*, ela não irá funcionar. OS CSS do PrimeFaces tem precedência e irá sobrepor os estilos! A alternativa para forçar a carga apos o CSS do Primefaces é carregar sua folha de estilos usando *<h:outputStyleSheet>* (que é a forma recomendada em JSF).

Outras alternativas são usar bloco *<style>* (que tem precedência sobre folhas carregadas externamente). Ainda assim, devido às regras de cascade do CSS, a aplicação do estilo pode não funcionar. Aplique estilos diretamente em elementos pelo ID, use o atributo *style* ou marcar as definições com *!important* para aumentar a precedência da declaração:

```
.ui-widget {font-size:90% !important}
```

Primefaces não é compatível com frameworks responsivos baseados em HTML5, CSS e JavaScript, e o uso combinado com esses frameworks pode causar erros e resultados inesperados devido a conflitos de JavaScript e CSS, dependendo dos recursos usados.

12.3 Alguns componentes Primefaces

A seguir será apresentada uma seleção de componentes do PrimeFaces em exemplos simples. Os exemplos são genéricos e se destinam a ilustrar a finalidade do componente e sua configuração basica. O código é baseado em componentes das versões 3.4 a 5.0, e pode não estar atualizado com a versão mais recente. Para usar, consulte a documentação online (que é

muito boa e detalhada) para exemplos utilizáveis, atributos e detalhes de como usar cada componente.

12.3.1 <p:spinner>

Campo de entrada de dados numéricos equivalente a HTML5 *<input type="number">*.Tipos são convertidos automaticamente (doubles, ints, etc).

Atributos: *min*, *max*, *stepFactor*, etc.

Como usar:

```
<p:spinner value="#{bean.numero}" />

@Named
public class Bean {
    private double numero;
    public double getNumero() {…}
    public void setNumero(double numero) {…}
}
```

Exemplo (*fonte: coreservlets.com*) usando *<p:ajax>* (similar a *<f:ajax>*)

```
<h:form>
&deg;F:
<p:spinner min="32" max="212" value="#{fBean2.f}">
     <p:ajax update="f c"/>
</p:spinner>
<h:outputText value="#{fBean2.f}&deg;F" id="f"/> = <h:outputText value="#{fBean2.c}&deg;C" id="c"/>
</h:form>
```

12.3.2 <p:calendar>

Campo de entrada para seleção de data (baseada em JQueryUI). É equivalente a HTML5 *<input type="date">*. Várias configurações visuais são possíveis (veja documentação).

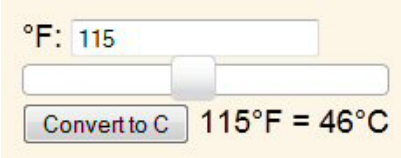
Como usar

```
<p:calendar value="#{bean.data}" />

@Named
public class Bean {
    private java.util.Date data;
    public java.util.Date getData() {…}
    public void setNumero(java.util.Date data) {…}
}
```

Alguns atributos de *<p:calendar>*:

- *value* - aponta para propriedade do tipo *Date*
- *pages* - número de meses a exibir de uma vez (default = 1)
- *showOn* (*focus* é default, pode ser *button* ou *both*) - quando o evento de mostrar o calendário será disparado
- *mode* (*popup* – default, ou *inline*) - mostrar o calendário o tempo todo ou apenas quando o usuário clicar
- *pattern="MM/dd/yyyy HH:mm"* - estabelece um padrão para a propriedade (o pattern obedece os padrões da classe *DateFormat*)
- *effect* – mostra calendário com efeito de animação <p:slider>



Campo de entrada para faixas de valores inteiros. É equivalente a usar HTML5 *<input type="range">*. Usado em sincronismo com *<h:inputText>*. *Slider* modifica os valores no *inputText* e *inputText* pode modificar posição do *slider*.

Exemplos de uso:

```
<p:slider for="id_do_input" … />
<h:inputText id="id_do_input" … />

<p:slider for="fonte" display="resultado" … />
<h:inputHidden id="fonte" … />
<h:outputText id="resultado" />
```

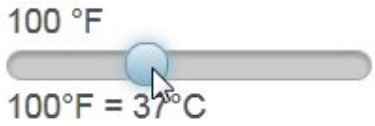
Exemplo (*fonte: coreservlets.com*) usando ajax para atualizar valores. Facelets:

```
<h:panelGrid width="200">
  <h:panelGroup>
     <h:inputHidden id="fInput3" value="#{fahrBean.f}" />
     <h:outputText id="fDisp3" value="#{fahrBean.f}"/> &deg;F
   </h:panelGroup>
```

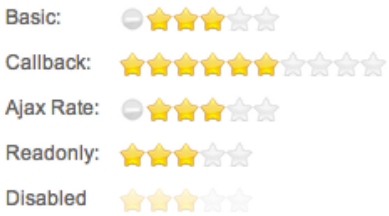
```
    <p:slider minValue="32" maxValue="212" for="fInput3" display="fDisp3">
        <p:ajax process="fInput3" update="status"/>
     </p:slider>
     <h:outputText value="#{fahrBean.status}" id="status" escape="false"/>
</h:panelGrid>
```

Bean:

```
public class FahrBean {
  ...
  public int getF() { ...}
  public String getStatus() { ...}
}
```



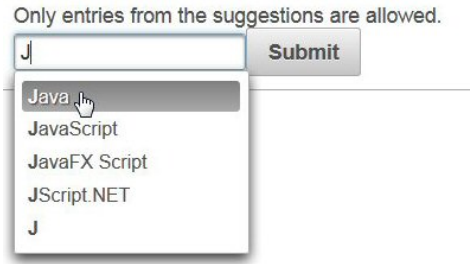
12.3.3 <p:rating>



Geralmente usado para representar um índice de avaliação. Exemplo:

```
<p:rating value="inteiro" />
```

12.3.4 <p:autoComplete>



Campo de texto que mostra um menu com lista reduzida de opções com base nos caracteres digitados.

Alguns atributos:

- *value* – propriedade string do bean que ira guardar o valor selecionado.

- *completeMethod* – método do bean que recebe o string parcial e retorna uma lista de itens que é compatível com esse valor
- *minQueryLength* – número minimo de caracteres que o value deve ter antes de iniciar o autocomplete.
- *queryDelay* – quantos milisegundos esperar antes de contactar o servidor

Usa ajax para fazer a requisição automaticamente. Pode-se usar *<p:ajax>* para fazer outras atividades durante os eventos do autoComplete (*itemSelect* ou *itemUnselect*):

```
<p:ajax event="itemSelect" listener="…" />
```

Exemplo de um método (não necessariamente eficiente) que devolve uma lista de opções (para chamar: *<p:autoComplete ... completeMethod="#{bean.completar}"…/>*)

```
public List<String> completar(String prefixo) {
     List<String> opcoes = new ArrayList<String>();
     for(String opcao: arrayDeOpcoes) {
       if(opcao.toUpperCase().startsWith(prefixo.toUpperCase())) {
            opcoes.add(opcao);
        }
     }
     return(opcoes);
}
```

Exemplo: facelets

```
<p:autoComplete value="#{bean.opcao}"
                completeMethod="#{bean.completar}" />
```

12.3.5 <p:input-mask>

Cria uma máscara para restringir um String de entrada. O formulário mostra como digitar o texto. Não é feita nenhuma conversão e todo o texto é enviado (inclusive a máscara).

Documentação:
http://digitalbush.com/projects/masked-input-plugin/

Como usar *<p:input-mask>*:

- Atributo *value*: o valor armazenado
- Atributo *mask*: texto – qualquer caractere – o texto literal não é editável mas é enviado para o servidor. Caracteres 9, a, * são curingas para número, letra ou ambos, respectivamente.

Ex: *"(999) 999-9999"* produz a máscara ilustrada acima. Se for digitado *1234567890* será enviado em *value* o texto *(123)456-7890*.

12.3.6 <p:colorPicker>

Um botão que quando clicado permite escolher uma cor. Valor é string *rrggbb* em hexadecimal e não começa com #. Atributos principais:

- *value*: contém a cor em hexadecimal
- *mode*: *popup* (default) ou *inline* (mostrado na página sem que o usuário aperte o botão)

12.3.7 <p:captcha>

Usa a API do Google (e requer configuração). É necessário registrar em *http://www.google.com/recaptcha/whyrecaptcha* para obter chaves públicas e privadas. Depois as chaves precisam ser registradas como parâmetros no *web.xml*:

```
<context-param>
     <param-name>primefaces.PRIVATE_CAPTCHA_KEY</param-name>
     <param-value>Private key enviada pelo Google</param-value>
</context-param>
<context-param>
      <param-name>primefaces.PUBLIC_CAPTCHA_KEY</param-name>
      <param-value>Public key enviada pelo Google</param-value>
 </context-param>
```

Veja documentação em *http://www.google.com/recaptcha/*.

Principais atributos de *<p:captcha />*: *required*, *requiredMessage*, *validatorMessage*, *theme* (*red*, *white*, *blackglass*, *clean*), *language* (default "*en*"), *validatorMessage* – sobrepõe a mensagem default.

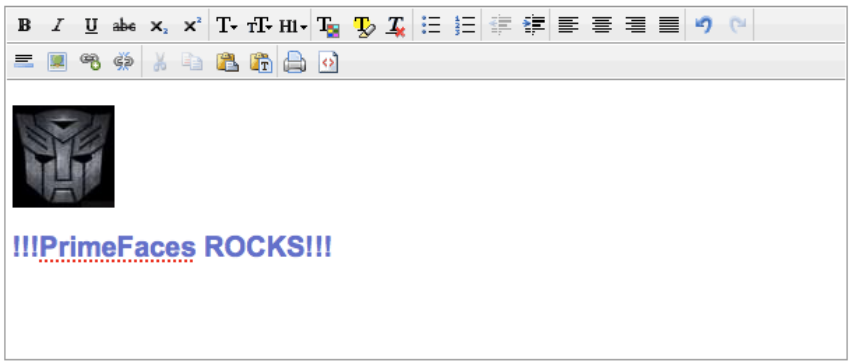
12.3.8 <p:password>

Campo de senha. Versão PrimeFaces para <p:inputSecret>. Atributos:

- *value* – propriedade que guarda a senha
- *feedback* (*true* ou *false*) - informa se deve haver feedback sobre a qualidade da senha

Pode-se substituir as mensagens default para a qualidade da senha nos atributos: *promptLabel*, *weakLabel*, *goodLabel*, *strongLabel*.

12.3.9 <p:editor>



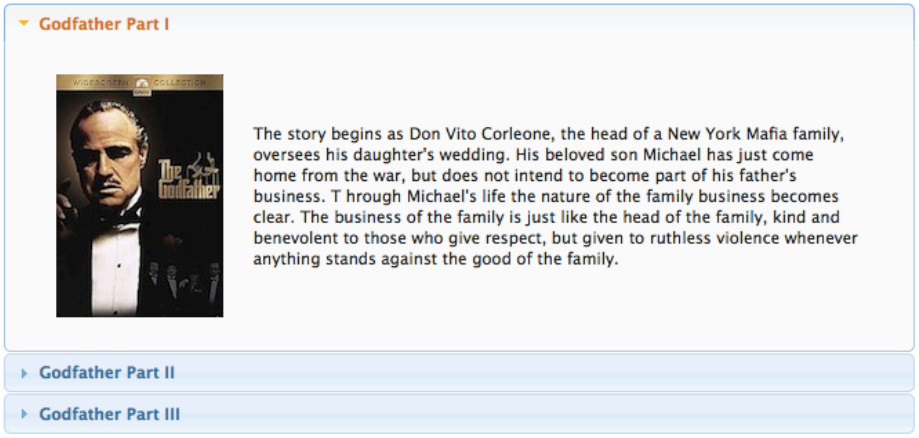
Editor de textos do *YUI* (usado no Yahoo Mail). O texto digitado pode conter HTML e o componente não protege contra riscos de Cross-Site Scripting (CSS). Usuário precisa tomar as medidas para evitar isso. Veja mais detalhes em *www.owasp.org*.

Atributos:

- *value* - conteúdo do editor (texto ou EL - opcional)
- *controls* – lista de controles que são exibidos (veja documentação)

12.3.10 Accordion

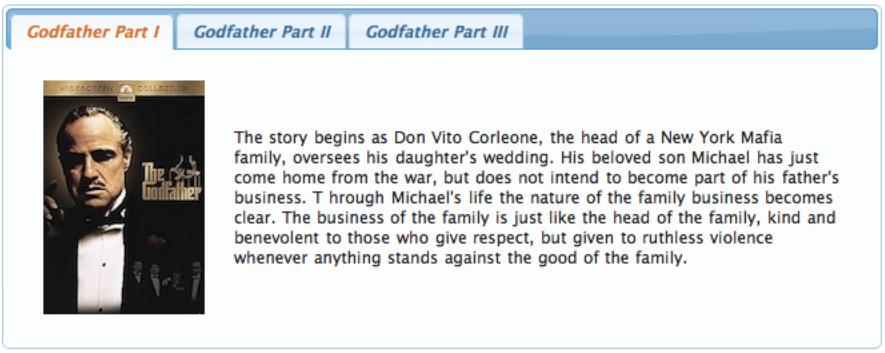
Wrapper para o painel **accordion** do JQuery UI: tabs horizontais para listar conteúdo e permitir detalhamento ao serem clicados.



Uso básico <*p:accordionPanel*> e <*p:tab*>:

```
<p:accordionPanel>
   <p:tab title="Titulo do primeiro tab">
      Conteúdo JSF
   </p:tab>
   <p:tab title="Titulo do segundo tab">
      Conteúdo JSF
   </p:tab>
...
</p:accordionPanel>
```

12.3.11 Tab view



E igual a um JQuery UI Tabbed Panel. Uso básico <*p:tabView*> e <*p:tab*>:

```
<p:tabView>
   <p:tab title="Titulo do primeiro tab">
      Conteúdo JSF
   </p:tab>
   <p:tab title="Titulo do segundo tab">
      Conteúdo JSF
   </p:tab>
...
</p:tabView>
```

12.3.12 <p:panelGrid>

Versão PrimeFaces do *h:panelGrid*. Funciona igual ao *h:panelGrid* mas bordas estão ativadas por default e usa o tema corrente para desenhar a tabela. Em vez de *h:panelGroup*, usa *p:row* e *p:column*. O elemento *p:column* tem atributos *rowspan* e *colspan* (como os que existem em <*td*>) que são ausentes em *h:panelGroup*. Por outro lado, não possui os atributos de <*table*> que estão presentes em *h:panelGrid* (*cellpadding*, etc.)

Exemplo de uso básico (fonte: *coreservlets.com*):

Major Stocks	
coreservlets.com	956.92 (+43.55%)
Prime Technology	887.48 (+37.78%)

```
<p:panelGrid columns="2">
  <f:facet name="header">Minor Stocks</f:facet>
  Google
  <h:outputText value="#{financeBean.google}"/>
  Facebook
  <h:outputText value="#{financeBean.facebook}"/>
  Oracle
  <h:outputText value="#{financeBean.oracle}"/>
</p:panelGrid>
```

Exemplo de row e column:

Mailing List Signup	
Name:	
Email:	
Sign Up	

```
<p:panelGrid>
  <f:facet name="header">
    <p:row>
      <p:column colspan="2">Mailing List Signup</p:column>
    </p:row>
  </f:facet>
  <p:row>
    <p:column>Name:</p:column>
    <p:column><p:inputText/></p:column>
  </p:row>
  <p:row>
    <p:column>Email:</p:column>
    <p:column><p:inputText/></p:column>
  </p:row>
```

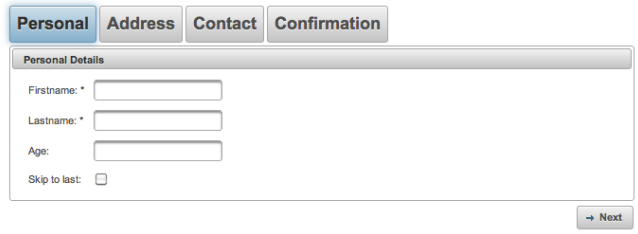
```
    <p:row>
      <p:column colspan="2" style="text-align: center">
        <p:commandButton value="Sign Up"/>
      </p:column>
    </p:row>
  </p:panelGrid>
```

12.3.13 Outros componentes

A seguir uma lista de outros componentes que *não* serão abordados aqui (consulte online detalhes na documentação e exemplos de uso).

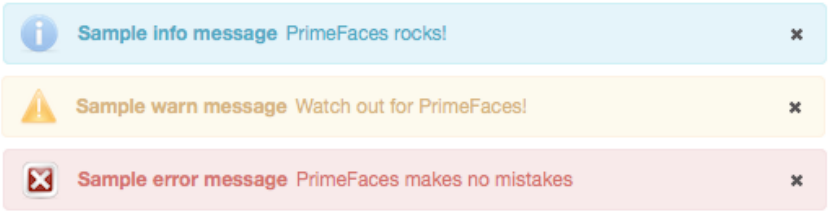
1) Componentes que permitem organizar o layout da página:

- p:dashboard
- p:scrollPanel
- p:layout
- p:outputPanel
- p:toolbar
- p:wizard



2) Formas alternativas de exibir mensagens

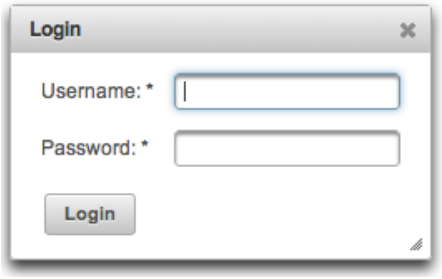
- <p:messages>
- <p:tooltip>
- <p:growl>



3) Janelas, popups, overlays

- <p:dialog>
- <p:confirmDialog>
- <p:overlayPanel>

- <p:lightBox>
- <p:notificationBar>



13 Mecanismos de extensão

13.1 Passthrough

JavaServer Faces gera código XHTML a partir de tags. Esse processo elimina atributos que são gerados automaticamente (ex: name, em formulários) e outros que não fazem parte da especificação). Pode-se usar diretamente quaisquer tags do HTML5 em páginas JSF, mas eles não serão processados como parte de requisições faces nem participação do ciclo de vida. Portanto, o HTML gerado pelos facelets não é compatível com HTML5.

Até a versão 2.1 de JSF era necessário programar com o RenderKit (escrever métodos Java e registrar em faces-config.xml) para lidar com atributos desconhecidos. Com a versão JSF 2.2 foi criado um mecanismo para permitir o uso de quaisquer atributos nos tags gerados. Isto é possível associando-os a um namespace próprio. Isto permite não apenas que o JSF 2.2 suporte HTML5, como também que inclua atributos específicos de outros frameworks, como o Bootstrap ou Foundation. O namespace é:

```
http://xmlns.jcp.org/jsf/passthrough
```

Para usá-lo ele deve ser associado a um prefixo. Na documentação da Oracle geralmente usa-se "p" que conflita com o uso de Primefaces, então é preciso escolher outro, por exemplo "a", e declarar o namespace:

```
<html ... xmlns:a="http://xmlns.jcp.org/jsf/passthrough">
```

Os atributos que não fazem parte do tag então podem ser usados desde que prefixados. Por exemplo, para gerar um <input type="number"> do HTML5 pode-se usar:

```
<h:inputText id="num" value="#{bean.numero}" a:type="number" />
```

Na verdade o JSF 2 não gosta muito dessas combinações. O ideal é escolher entre HTML5 e JSF, Bootstrap e JSF, e evitar combiná-los. *Passthrough* e outros mecanismos são uma espécie de "gambiarra" para viabilizar esse uso, mas é preciso ter cuidado com a maneira

como esses componentes serão renderizados no browser, e dos conflitos que eles irão causar. Isto envolve não apenas o conflito de CSS, mas também envolve a forma como os browser renderizam componentes HTML5. Por exemplo, a maior parte dos browsers renderiza <input type="date"> como um calendário. Embora identificar campos de data com "date" seja recomendado pelo HTML5, haverá problemas se o tag for gerado por um componente de calendário do Primefaces.

13.2 Integração com Bootstrap

Como foi mencionado nas seções anteriores, JSF não funciona muito bem com outros frameworks que interferem no look & feel da interface Web, e combiná-lo com frameworks baseados em HTML5, JavaScript e CSS é um desafio.

Mas, devido à popularidade do HTML5 e de frameworks como JQuery, AngularJS e Bootstrap, têm surgido vários pacotes para integrá-los com JSF. Algumas das soluções mais populares são distribuídos pelo Primefaces:

- *PrimeUI* – uma coleção de componentes HTML5 baseados em JQueryUI
- *PrimeNG* – uma coleção de componentes HTML5 baseados em AngularJS

Para Bootstrap, o Primefaces oferece um tema, que fornece um look & feel *similar* ao BootStrap (mas não é uma integração completa). E possível configurar as folhas de estilo e adaptar algumas funções especificas, removendo conflitos de CSS, mas ainda há conflitos de script em componentes interativos, devido às incompatibilidades das bibliotecas JavaScript.

Existe um framework chamado *BootsFaces*, baseado em JQuery UI e Bootstrap 3 que é compatível com JSF e PrimeFaces e oferece uma integração maior. A desvantagem é um novo vocabulário de tags e atributos para aprender.

Em suma, não existe uma solução que ofereça integração simples e completa, nem existe nenhuma recomendação oficial.

14 Referências

14.1 Especificações e documentação oficial

1. Oracle. Java Server Faces Specification. https://jcp.org/aboutJava/communityprocess/final/jsr344/index.html
2. Oracle. The Java EE 6 Tutorial. http://docs.oracle.com/javaee/6/tutorial/doc/
3. Oracle. The Java EE 7 Tutorial. http://docs.oracle.com/javaee/7/tutorial/doc/
4. Contexts and Dependency Injection Specification. http://www.cdi-spec.org/

5. W3C. Ajax. http://www.w3.org/TR/XMLHttpRequest/
6. IETF RFC 2616 HTTP: http://www.w3.org/Protocols/rfc2616/rfc2616.html
7. W3C. XHTML. http://www.w3.org/MarkUp/ (padrão descontinuado)
8. *Manual do PrimeFaces*. http://primefaces.org/documentation.html

14.2 Tutoriais e artigos

1. Bauke Scholtz. *The BalusC Code*. http://balusc.omnifaces.org/ e vários artigos sobre JSF do StackOverflow organizados em https://jsf.zeef.com/bauke.scholtz.
2. Marty Hall. PrimeFaces Tutorial Series. http://www.coreservlets.com/JSF-Tutorial/primefaces/
3. M. Kyong. JSF 2.0 Tutorials. http://www.mkyong.com/tutorials/jsf-2-0-tutorials/

14.3 Livros

1. David Geary & Cay Horstmann. *Core JavaServer Faces 2.0*. Prentice-Hall, 2010.
2. Ed Burns. *JavaServer Faces 2.0: The Complete Reference. McGraw-Hill, 2009.*
3. Arun Gupta. *Essential Java EE 7.* O'Reilly, 2013.

14.4 Produtos

1. Mojarra https://javaserverfaces.java.net/2.2/download.html
2. JBoss Weld http://weld.cdi-spec.org/
3. Tomcat http://tomcat.apache.org/
4. JBoss http://www.jboss.org/
5. Glassfish https://glassfish.java.net/
6. PrimeFaces http://primefaces.org/