# Desenvolvimento de Sistemas Web III

## Análise e Desenvolvimento de Sistemas

Paulo Maurício Gonçalves Júnior

Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco

24 de janeiro de 2017

Parte I

# Introdução a Ajax

# Introdução a Ajax

- É um conjunto de técnicas de programação ou uma abordagem particular para programação web.
- Estas técnicas de programação envolvem ser capaz de atualizar uma página ou uma seção de uma página através de dados do servidor, mas sem necessidade de uma atualização completa da página.
- Isso não significa que o navegador não precise realizar uma conexão com o servidor.
- Exemplos: Google Gmail, Maps, Suggest, Talk.

# Bons exemplos

- AutoSave: salvar dados digitados em caixas de texto automaticamente.
- AutoComplete: predizer textos sendo digitados.
- Paginação: quando uma grande quantidade de dados é retornada por uma pesquisa, o resultado pode ser melhor gerenciado usando Ajax.
- Comunicação entre usuários: permite a comunicação entre usuários diretamente pelo navegador.

# Ajax: o acrônimo

- *Asynchronous JavaScript and XML*. Ajax não precisa usar XML nem ser assíncrono.
- Ajax pode lidar com diversas tecnologias:
  - XHTML e CSS
  - DOM (*Document Object Model*)
  - JavaScript
  - XML e XSLT
  - O objeto `XMLHttpRequest`
- O mínimo a ser usado é XHTML, DOM, e JavaScript.
- Provavelmente também será necessário uma linguagem para gerenciar a interação com o servidor: PHP, ASP.NET, Java.

# O objeto XMLHttpRequest

- Permite aos desenvolvedores submeter e receber dados textuais ou documentos XML em background.
- `var xHRObject = new XMLHttpRequest();.`

# O modelo de aplicação Ajax

- No modelo tradicional, o usuário digita informações, envia-as ao servidor e espera uma resposta. Este é chamado um modelo síncrono. O navegador sempre inicia a conversação.
- O usuário tem que esperar a resposta e enquanto ela não chega, nada acontece, nenhuma nova interação pode ser realizada. O servidor não pode iniciar a comunicação.
- Em Ajax então é possível atualizar somente os elementos que precisam ser atualizados.
- Ao invés do modelo síncrono, existem os modelos assíncronos e de pedido.

# O modelo de aplicação Ajax

- Em uma aplicação Ajax, o servidor pode deixar uma notificação hoje, e o cliente obterá a informação quando desejar.
- Ou então, o cliente pode realizar pedidos em intervalos regulares e verificar se o servidor está pronto, mas pode continuar com outras operações.

# Por que usar Ajax?
## Quando usar Ajax?

- Realizar atualizações parciais da página.
- Obtenção de dados de forma invisível.
- Atualizações contantes.
- Interfaces mais elaboradas.
- Simplicidade e funcionalidades ricas.
- Arrastar e soltar.

# Por que usar Ajax?
## Quando não usar Ajax?

- Respostas lentas.
- Quebra o botão de voltar do navegador.
- Quebra os favoritos e bloqueia índices em engenhos de busca.
- Colocar muito processamento no navegador.

# Quem não pode usar Ajax?

- Navegadores mais antigos.
- Navegadores com scripts desabilitados.
- Pessoas que navegam offline.

# Parte II

# Ajax e Tecnologias do Lado Servidor

# Tecnologias do Lado Servidor

- Até agora não usamos nenhum código no lado do servidor.
- A forma na qual Ajax (leia-se o objeto `XMLHttpRequest`) se comunica com o servidor é fundamentalmente a mesma para cada tecnologia. Mas uma vez que a informação chegou, a forma na qual ela é lidada difere um pouco para cada tecnologia (ASP.NET, PHP, e Java).
- Vamos nos focar na tecnologia Java.

# Forms e controles HTML

- Ajax não muda a forma com que os controles HTML (botões radio, caixas de checagem, de texto, etc.) funcionam.
- Existem duas mudanças:
  1. A primeira é a forma na qual a página no servidor é chamada.
  2. A segunda é o fato do formulário HTML poder ser completamente removido ou modificado de forma a não funcionar da forma esperada.

# Os modelos de submissão

- Em qualquer tecnologia do lado servidor, o modo normal de submissão é um formulário com um botão que é submetido ao servidor quando o usuário clica no botão.
- No modelo de submissão Ajax, os formulários podem ser removidos completamente. Quando um evento ocorre, um script inicia a comunicação com o servidor.

# Do lado servidor em diante

- JavaScript é usado para começar a interação cliente/servidor.
- Enquanto existem vários métodos para submeter dados ao servidor, vamos focar apenas no `XMLHttpRequest`. Para submeter dados ao servidor usando este objeto, existem três passos envolvidos:
  1. Instanciar o objeto.
  2. Chamar o método `open` juntamente com o pedido.
  3. Indicar o evento que executará quando os dados forem recebidos.
  4. Enviar o pedido.

# Do lado servidor em diante

- O método `open` recebe três parâmetros (`object.open(method, URL to call, asynchronous or synchronous);`):
  1. O método HTTP a ser utilizado (get ou post).
  2. O recurso a ser chamado (URL).
  3. Se a chamada é assíncrona (true) ou síncrona (false). Omitindo-se este parâmetro, o padrão é assíncrona.
- Se submetermos usando GET:

```
XMLHttpRequestObject.open("get", "response.jsp?value=1", true);
```

- Se submetermos usando POST:

```
var argument = "value=";
argument += encodeURIComponent(data);
xHRObject.open("post", "response.jsp", true);
xHRObject.setRequestHeader("Content-Type", "application/x-www-form
    -urlencoded");
xHRObject.send(argument);
```

# GET x POST

- A string de pesquisa no método GET possui uma limitação de tamanho dependendo o navegador utilizado.
- Com GET, dados só podem ser enviados se forem texto.
- GET sofre cache, enquanto POST não.
- GET deve ser usado para consultas que não causam mudanças no servidor.

# O servidor recebe o pedido

- Não existe diferença observável entre receber dados através de um formulário HTML ou um pedido originado por JavaScript.
- O que difere da sequencia normal de eventos que você teria com a tecnologia do lado servidor é que a informação desejada não pode ser imediatamente escrita na página. Ao invés disso, você deve escrever a informação no objeto `Response` de HTTP.
- Em JSP isso significa:

```
String data = "This is our data";
response.getWriter().write(data);
```

# O objeto XMLHttpRequest

- O primeiro estágio para o recebimento dos dados é criar uma função que é executada quando os dados foram recebidos do servidor.
- Isto é feito através do evento `onreadystatechange`, registrado no objeto XMLHttpRequest.
- Dentro da função, a primeira coisa a fazer é verificar se os dados foram recebidos. Isso é feito verificando se a propriedade `readyState` do objeto `XMLHttpRequest` é igual a 4 (completo) e se o atributo `status` é igual a 200.

```
function getData() {
  if (xHRobject.readystate === 4 && xHRobject.status === 200) {
    //Do our processing here
  }
}
```

# A propriedade `readyState`

- Esta propriedade indica em que ponto o objeto `XMLHttpRequest` está comm relação ao envio/recebimento de dados.
  1. **0 – Não inicializado** — Quando o objeto foi criado mas ainda não inicializado. Em outras palavras, o método `open` ainda deve ser chamado.
  2. **1 – Aberto** — Quando o objeto foi criado e inicializado, mas o método `send` ainda não foi chamado.
  3. **2 – Enviado** — Quando o método `send` foi chamado, mas o objeto está esperando pelo retorno do código de status e pelos cabeçalhos.
  4. **3 – Recebendo** — Quando algum dado já foi recebido, mas não todo. Não é possível usar as propriedades do objeto para ver resultados parciais porque cabeçalhos e status ainda não estão completamente disponíveis.
  5. **4 – Carregado** — Estágio final onde todos os dados foram recebidos. Normalmente o estado que estaremos verificando.

# responseText

- Uma vez recebidos os dados, podemos usar uma de duas propriedades de `XMLHttpRequest`: `responseText` ou `responseXML`.
- Esta é a abordagem mais comum e mais simples.
- Você pode criar uma variável e obter o conteúdo vindo do servidor.

```
var text = xHRobject.responseText;
```

- Também é possível utilizá-lo para obter XML, mas os dados virão como texto.

# responseXML

- Permite tratar as respostas como documentos XML e navegar por ele usando DOM.
- Antes de enviar os dados, é importante setar o `Content Type` para `text/xml`.

```
response.setContentType("text/xml");
response.getWriter().write("<valid>Response</valid>");
```

- E mudar o código JavaScript para:

```
var document = xHRobject.responseXML;
```

# FormData

- O objeto `FormData` pode ser utilizado para simplificar o envio de dados através de Ajax.
- Ao invés de criar uma string com os dados a serem enviados ao servidor, o objeto `FormData` encapsula os dados (inclusive arquivos) no formato de Map.
- O construtor possui um parâmetro opcional que pode receber um formulário.

```
var formData = new FormData();
var formData = new FormData(form);
```

# FormData
## append

- Podemos acrescentar um novo valor utilizando o método `append`.

```
var formData = new FormData();
formData.append('username', 'Chris');
formData.append('userpic', myFileInput.files[0], 'chris.jpg');
```

# FormData

- Ao finalizar a criação/modificação dos dados do formulário, podemos enviar seus dados passando o objeto no método `send`.

```
var formData = new FormData();
formData.append('username', 'Chris');
formData.append('userpic', myFileInput.files[0], 'chris.jpg');
xhr.send(formData);
```

# Parte III

# Trabalhando com XML

# Trabalhando com XML

- Usando o comando `XMLHttpRequest`, o servidor pode retornar os dados em dois formatos: texto ou XML.
- Veremos agora
  - Usando CSS (CSS e dados XML, usando JavaScript com CSS)
  - Transformando XML com XSLT no navegador

# CSS

- Documentos XML não contém nenhuma informação de formatação.
- Pode-se aplicar CSS diretamente a um documento XML.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="classes.css"?>
```

# XSLT

- Permite aplicar XSLT a um documento XML e o próprio navegador faz a transformação.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="classes.xsl"?>
```

# XSLT

- Podemos realizar a transformação através de Javascript, adicionando o resultado à página HTML.
- Utiliza-se o objeto `XSLTProcessor`.
- Importa-se um documento XSLT.
- Aplica-se a um documento XML.
- O resultado pode ser acrescentado à página HTML.

```
var xsltProcessor = new XSLTProcessor();
xsltProcessor.importStylesheet(xslt);
var result = xsltProcessor.transformToFragment(xml,document);
document.body.appendChild(result);
```

# XSLT

- Pode-se utilizar parâmetros para ajustar as pesquisas:
  - `setParameter(namespace,nome do atributo, valor do atributo)`
  - `getParameter(namespace,nome do atributo)`
  - `removeParameter(namespace,nome do atributo)`

# XSLT

- O objeto `XMLSerializer` permite converter uma árvore em sua representação textual.

```
var oSerializer = new XMLSerializer();
var sXML = oSerializer.serializeToString(doc);
```

- O objeto `DOMParser` permite converter uma string em uma árvore.

```
var parser = new DOMParser();
var doc = parser.parseFromString(stringContainingXMLSource, "
    application/xml");
```

# XPath

- `var xpathResult = document.evaluate(xpathExpression, contextNode, namespaceResolver, resultType, result);`
  - `xpathExpression`: String com a expressão XPath.
  - `contextNode`: Nó raiz a partir do qual a expressão será executada.
  - `namespaceResolver`: Uma função que tratará os namespaces.
  - `resultType`: Uma constante com o tipo de resposta esperada.
  - `result`: Um resultado prévio a ser reutilizado.

# XPath
## Tipos de retorno

- ANY_TYPE
- NUMBER_TYPE
- STRING_TYPE
- BOOLEAN_TYPE
- UNORDERED_NODE_ITERATOR_TYPE
- ORDERED_NODE_ITERATOR_TYPE
- UNORDERED_NODE_SNAPSHOT_TYPE
- ORDERED_NODE_SNAPSHOT_TYPE
- ANY_UNORDERED_NODE_TYPE
- FIRST_ORDERED_NODE_TYPE

# Parte IV

# JSON

# JSON

- Ao invés de transmitir os dados no formato XML, pode-se utilizar o formato JSON (*JavaScript Object Notation*).
- É um formato texto baseado em um subconjunto da sintaxe de JavaScript.
- Existem bibliotecas JSON para várias linguagens que convertem dados formatados nestas linguagens para JSON.

# Sintaxe JSON

- JSON consiste de duas estruturas de dados:
  - **Objetos:** Uma coleção não ordenada de pares nome/valor.
  - **Arrays:** Uma coleção ordenada de valores.
- Não existem variáveis ou estruturas de controle em JSON. Ele é feito unicamente para transmitir dados.

# Tipos de dados JSON

- As estruturas de dados JSON podem incluir qualquer um dos seguintes tipos de dados:
  - String
  - Number
  - Valor booleano (true/false)
  - null
  - Object
  - Array
- Strings JSON devem estar entre aspas duplas.

# Objetos literais

- Objetos em JavaScript podem ser definidos através de um construtor ou literal.
- Construtor:

```
var member = new Object();

member.name = "Jobo";
member.address = "325 Smith Rd";
member.isRegistered = true;

member["name"] = "Jobo";
member["address"] = "325 Smith Rd";
member["isRegistered"] = true;
```

# Objetos literais

- Literal:

```
var member =
{name: "Jobo",
address: "325 Smith Road",
isRegistered: true
};
```

- Em JSON:

```
{name: "Jobo",
address: "325 Smith Road",
isRegistered: true
}
```

# Arrays literais

- Arrays em JavaScript também podem ser definidos através de um construtor ou literal.
- Construtor:

```
var myArray = new Array();

myArray[0] = 1218;
myArray[1] = "Crawford";
myArray[2] = "Drive";
```

- Literal:

```
var myArray = [1218, "Crawford", "Drive"];
```

- Em JSON:

```
[1218, "Crawford", "Drive"]
```

# Usando um parser JSON

- Navegadores disponibilizam um parser JSON através do objecto `JSON` para criar textos JSON a partir de objetos e arrays, e criar objetos e arrays a partir de texto JSON.
- Ele define dois métodos estáticos: `stringify` e `parse`.
- O método `stringify` converte valores em JavaScript para um string JSON.
- O método `parse` converte uma string para JSON.

# Formatos de Transmissão de dados

- Tanto JSON como XML podem ser usados como meio de transmissão de dados em Ajax.
- JSON possui algumas vantagens:
  - **JSON é JavaScript:** Texto JSON pode ser facilmente convertido para objetos ou arrays em JavaScript.
  - **JSON é tipado:** Um objeto JSON possui tipo: string, number, boolean, null, object ou array.
  - **JSON pode ser interpretado como JavaScript:** Não é necessário DOM para interpretar a resposta do servidor, como em XML.
- Geralmente, os arquivos JSON são menores que seus correspondentes em XML.

Parte V

# Upload de arquivos com Ajax

# Upload

- Upload de arquivos é realizado a partir do elemento `<input type="file">`.
- Usando Ajax, podemos também obter os arquivos através de uma operação de arrastar e soltar.
- Ao selecionar arquivos, os dados dos arquivos ficam disponíveis através da propriedade `files` (tipo **`FileList`**) do elemento `<input>`.
- Este tipo retorna um conjunto de objetos do tipo **`File`**, podendo-se obtê-los como um vetor ou usando o método `item()`.

# Upload

- Para ler os dados dos arquivos e carregá-los no navegador, utilizaremos o objeto **`FileReader`**.
- Três métodos podem ser utilizados para ler arquivos:
  - `readAsDataURL()`: Converte para uma string codificada no formato base64.
  - `readAsArrayBuffer()`: Converte para um array de dados binários.
  - `readAsText()`: Converte para uma string de texto.
- Durante a leitura do arquivo, podemos obter o estado da leitura através da propriedade `readyState`:

| | | |
|---|---|---|
| EMPTY | 0 | Nenhum dado foi carregado ainda. |
| LOADING | 1 | Dados estão sendo carregados. |
| DONE | 2 | A leitura completa dos dados foi concluída. |

# Upload

- Ao final da leitura do arquivo, o evento `onloadend` é chamado e o arquivo está disponível no atributo `result`.
- Finalmente, acrescentaremos os dados através do objeto **`FormData`** e o método `append`. Para indicar que serão mais de um arquivo, acrescenta-se um par de colchetes [] ao fim do nome do elemento.

# Barra de progresso

- Podemos incluir uma barra de progresso para informar o percentual de envio dos dados para o servidor. Para isso, podemos utilizar o elemento `<progress>`.
- Para isso, incluiremos o evento `onprogress` ao objeto `upload` do objeto **`XMLHttpRequest`**.
- O evento recebido possui duas propriedades: `loaded` e `total`, indicando quanto do total dos dados foi efetivamente enviado.

Parte VI

# WebSockets

# Introdução

- WebSockets é uma tecnologia que permite abrir uma sessão de comunicação interativa entre o navegador e o servidor. Desta forma, pode-se enviar mensagens para o servidor e receber respostas através de eventos sem precisar constantemente questionar o servidor por uma resposta.
- Ideal para casos em que a troca de dados entre o navegador e o servidor se dará de forma constante.

# WebSockets no navegador

- Para enviar dados do navegador para o servidor, devemos, inicialmente, criar uma conexão:

```
var ws = new WebSocket("ws://localhost:8080");
```

- Para enviar dados para o servidor, utilizamos o seguinte método:

```
ws.send('Hello Server!');
```

- E finalmente para fechar a conexão:

```
ws.close();
```

# WebSockets no navegador

- Quando dados são recebidos pelo servidor, devemos registrar os seguintes eventos para processá-los:
  - `onopen`: Chamado quando o servidor confirma a abertura de conexão e pode-se começar a enviar mensagens.
  - `onmessage`: Chamado quando o servidor envia uma mensagem.
  - `onerror`: Quando ocorre um erro ocorre na conexão.
  - `onclose`: Chamado quando a conexão foi fechada.

# WebSockets no navegador

- Quando os dados são recebidos, os dados podem ser obtidos através da propriedade `data` do evento `MessageEvent` recebido:

```
function onMessage(evt) {
  writeResponse("received: " + evt.data);
}
```

# WebSockets no servidor

- No servidor, a abordagem é semelhante. Primeiramente, criamos uma classe, e acrescentamos a anotação `@ServerEndpoint` para indicar que ela receberá dados através do protocolo WebSockets.
- Ela recebe um texto informando o nome do websocket, sempre começando com uma barra:

```
import javax.websocket.server.ServerEndpoint;
@ServerEndpoint("/echo") ou @ServerEndpoint(value="/echo")
public class EchoServer {
   ...
}
```

# WebSockets no servidor

- Três métodos podem ser implementados, acrescentando anotações para indicar em que situação serão chamados:
  - `@OnOpen`: Chamado quando o servidor recebe uma solicitação de abertura de conexão.
  - `@OnMessage`: Chamado quando o servidor recebe uma mensagem.
  - `@OnClose`: Chamado quando foi solicitado o fechamento da conexão.
- Apenas uma anotação pode ser acrescentada por classe.

# WebSockets no servidor
## ServerEndpoint

- O atributo **`value`** permite utilizar URI path templates, que são URIs com variáveis embutidas na sintaxe da URI. Variáveis são denotadas por chaves.

```
@ServerEndpoint("/users/{username}")
```

- Para obter o valor das variáveis, utiliza-se `PathParam` em um dos métodos anteriores.

```
@ServerEndpoint("/users/{username}")
public class UserEndpoint {
    @OnMessage
    public String getUser(String message, @PathParam("username")
        String userName) {
        ...
    }
}
```

# WebSockets no servidor
## ServerEndpoint

- O atributo `decoders` pode conter uma lista de classes que serão usadas para decodificar mensagens recebidas. Isso significa transformar mensagens de texto / binário para algum tipo definido pelo usuário. Cada decodificador necessita implementar a interface `Decoder`.

```
@ServerEndpoint(value = "/sample", decoders = SampleDecoder.class)
```

- Alguns tipos de decodificadores podem ser usados:

```
public class SampleDecoder implements Decoder.Text<SampleType>
public class SampleDecoder implements Decoder.TextStream<
    SampleType>
public class SampleDecoder implements Decoder.Binary<SampleType>
public class SampleDecoder implements Decoder.BinaryStream<
    SampleType>
```

# WebSockets no servidor
## ServerEndpoint

- O atributo `encoders` pode conter uma lista de classes que serão usadas para codificar mensagens a serem enviadas. Isso significa transformar mensagens de algum tipo definido pelo usuário para texto / binário. Cada codificador necessita implementar a interface `Encoder`.

```
@ServerEndpoint(value = "/sample", encoders = SampleEncoder.class)
```

- Alguns tipos de decodificadores podem ser usados:

```
public class SampleDecoder implements Encoder.Text<SampleType>
public class SampleDecoder implements Encoder.TextStream<
    SampleType>
public class SampleDecoder implements Encoder.Binary<SampleType>
public class SampleDecoder implements Encoder.BinaryStream<
    SampleType>
```

# WebSockets no servidor
## OnOpen

- Pode receber três parâmetros opcionais:
  - Session

```
public void open(Session session)
```

  - EndpointConfig

```
public void open(Session session, EndpointConfig config)
```

  - Zero a n parâmetros com a anotação PathParam do tipo String ou tipo primitivo

```
public void open(Session session, @PathParam("room") String
    room)
```

# WebSockets no servidor I

## OnMessage

- Os parâmetros permitidos são exatamente uma das opções abaixo:
  - Se estiver lidando com mensagens de texto:
    - String para receber a mensagem completa
    - Valor primitivo ou equivalente para receber a mensagem convertida para o tipo
    - String e par booleano para receber a mensagem em partes
    - `Reader` para receber toda a mensagem como um *blocking stream*
    - Qual parâmetro do tipo objeto para o qual foi declarado um decodificador de texto (`Decoder.Text` ou `Decoder.TextStream`).
  - Se estiver lidando com mensagens binárias:
    - byte[] ou ByteBuffer para receber a mensagem completa
    - byte[] e boolean, or ByteBuffer e boolean pair para receber a mensagem em partes
    - InputStream para receber a mensagem como um *blocking stream*
    - Qual parâmetro do tipo objeto para o qual foi declarado um decodificador binário (`Decoder.Binary` ou `Decoder.BinaryStream`).

# WebSockets no servidor II

## OnMessage

- Zero a n parâmetros com a anotação PathParam do tipo String ou tipo primitivo
- Um parâmetro opcional do tipo `Session`

- Os parâmetros podem ser listados em qualquer ordem.

# WebSockets no servidor

## OnClose

- Pode receber três parâmetros opcionais:
  - Session
  - CloseReason
  - Zero a n parâmetros com a anotação PathParam do tipo String

Parte VII

# Introdução a JavaServer Faces

# Introdução a JavaServer Faces I

- JSF é um framework baseado em componentes.
- JSF possui três partes:
  - Um conjunto de componentes de interface gráfica pré-fabricados.
  - Um modelo de programação baseado em eventos.
  - Um modelo de componentes que permite desenvolvedores terceiros suprir componentes adicionais.
- Vejamos um exemplo básico.
- Em JSF, o código da lógica da aplicação está contido em *beans*, e a interface gráfica nas páginas web.
- O processador das páginas JSF é chamado de *Facelets*. Ele processa páginas XHTML com *tags* JSF, usando a extensão `.xhtml`.
- O arquivo web.xml é basicamente igual para todas as aplicações JSF.
- Usando mapeamento prefixado, todos os pedidos que comecem com "/faces/" serão tratados como JSF.

# Introdução a JavaServer Faces II

- Exemplo: `http://www.sumatra.com/faces/mypage.xhtml`

```
<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>
```

- Usando mapeamento sufixado (ou de extensão), todos os pedidos que terminem com ".faces" serão tratados como JSF.
- Exemplo: `http://www.mrcat.org/mypage.faces`

# Introdução a JavaServer Faces III

```
<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>
```

- Para mudar a extensão padrão para as páginas JSF, usar:

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.jsf</param-value>
</context-param>
```

- Podemos também usar um parâmetro para informar o estágio do projeto:

# Introdução a JavaServer Faces IV

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

- Os valores para este parâmetro são *Development*, *UnitTest*, *SystemTest*, e *Production*. No desenvolvimento, você obtém mensagens de erro mais informativas.
- Alguns servidores automaticamente provêm mapeamento para `/faces/*`, `*.faces` e `*.jsf`, se uma das condições abaixo ocorrerem:
  - Qualquer classe usar uma anotação JSF
  - Qualquer parâmetro de inicialização começar com `javax.faces`
  - O arquivo `WEB-INF/faces-config.xml` estiver presente
- Não é necessário prover o arquivo `web.xml` se não setar um parâmetro de inicialização.
- Os serviços mais importantes que o *framework* JSF provê são:

# Introdução a JavaServer Faces V

- Arquitetura MVC
- Conversão de dados
- Validação e tratamento de erros
- Internacionalização
- Componentes customizados
- Suporte a Ajax
- Visualizadores alternativos

- Ciclo de vida

# Introdução a JavaServer Faces VI

| Phase | Description | Events fired |
|---|---|---|
| Restore View | Finds or creates a tree of components for the selected view. Some components, like `HtmlCommandButton`, will generate action events (or other types of events) in this phase. | Phase events |
| Apply Request Values | Updates the value of the components to equal ones sent in the request, optionally using converters. Adds conversion errors if there is an error. Also generates events from request parameters. | Phase events, data model events, action events |
| Process Validations | Asks each component to validate itself (which may include using external validators). Validation error messages may be reported. | Phase events, data model events, value-change events |
| Update Model Values | Updates all the values of backing beans or model objects associated with components. Conversion error messages may be reported. | Phase events, data model events, |
| Invoke Application | Calls any registered action listeners. The default action listener will also execute action methods referenced by command components (like `HtmlCommandButton`) and choose the next view to be displayed. | Phase events, action events |
| Render Response | Displays the selected view using the current display technology (like JSP). | Phase events |

# Parte VIII

# Beans Gerenciados

# Introdução

- JSF usa *beans* para separar a apresentação e a lógica do negócio.
- *Beans* armazenam o estado das páginas web.
- Pode-se usar uma expressão de valor para acessar a propriedade de um *bean*.

# Criando e inicializando beans I

- Em JSF, pode-se configurar e inicializar beans em um arquivo de configuração. A ideia por trás dessa funcionalidade, chamada *Managed Bean Creation*, é simples: sempre que se referenciar um bean, o bean será criado, inicializado, e armazenado no escopo de aplicação correto se ele ainda não existir (Caso exista, ele será retornado.)
- Usar managed beans permite:
  - Declarar todos os beans e inicializar todas as suas propriedades em um único local.
  - Controlar o escopo (aplicação, sessão ou pedido) onde o bean é armazenado.
  - Modificar a classe de um bean ou seus valores iniciais sem modificar o código.

# Criando e inicializando beans II

- Inicializar uma propriedade do bean com *value-binding expressions*. Isso traz vários benefícios, como associar um bean com objetos de negócio ou de armazenamento ou inicializar objetos filhos.
- Acessar um bean gerenciado usando apenas expressões JSF EL.

# Criando e inicializando beans

- Beans gerenciados podem ser configurados através do elemento `<managed-beans>`.

```
<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>org.jia.examples.UserBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>firstName</property-name>
    <value>Mark</value>
  </managed-property>
  <managed-property>
    <property-name>lastName</property-name>
    <value>Pippins</value>
  </managed-property>
</managed-bean>
```

# Backing Beans

- Podemos projetar um *bean* que contenha alguns componentes de um formulário.

```
public class QuizFormBean {
  private UIInput answerComponent;
  private UIOutput scoreComponent;
  public UIInput getAnswerComponent() { return answerComponent; }
  public void setAnswerComponent(UIInput newValue) {
      answerComponent = newValue; }
  public UIOutput getScoreComponent() { return scoreComponent; }
  public void setScoreComponent(UIOutput newValue) {
      scoreComponent = newValue; }
  ...
}
```

- Para associar os componentes da página com os do *bean*, usamos o atributo `binding`:

```
<h:inputText binding="#{quizForm.answerComponent}".../>
```

# Mensagens

- Todas as mensagens devem ser colocadas em um local, usando o formato *properties*:

```
guessNext=Guess the next number in the sequence!
answer=Your answer:
```

- O arquivo deve ser salvo em um pacote Java e informar no arquivo `faces-config.xml`, no diretório `WEB-INF`:

```
<resource-bundle>
  <base-name>com.corejsf.messages</base-name>
  <var>msgs</var>
</resource-bundle>
```

- Para mensagens com idiomas distintos, devemos criar arquivos para cada idioma, acrescentando um sufixo com o código do idioma: um *underscore* seguido por duas letras descrevendo o idioma.
- Exemplo: `messages_de.properties`

# Internacionalização

- Podemos informar o idioma padrão de três formas. Usar o idioma informado pelo navegador:

```
<locale-config>
  <default-locale>en</default-locale>
  <supported-locale>de</supported-locale>
</locale-config>
```

- De forma programática:

```
UIViewRoot viewRoot = FacesContext.getCurrentInstance().
    getViewRoot();
viewRoot.setLocale(new Locale("de"));
```

- Para páginas individuais:

```
<f:view locale="de">
<f:view locale="#{user.locale}"/>
```

# Mensagens com partes variáveis

- Permite a criação de textos parametrizados, ou seja, a criação de uma string com pontos específicos que serão substituídos por valores determinados.
- Estes pontos de substituição são números englobados por chaves. Os parâmetros são especificados usando componentes `UIParameter`.

```
<h:outputFormat value="Hey {0}. This is a static value: {1}. {0}, you''
    re using: {2}.">
  <f:param value="#{user.firstName}"/>
  <f:param value="hardcoded"/>
  <f:param value="#{header['User-Agent']}"/>
</h:outputFormat>
```

# Apresentando dados com componentes de saída

## HtmlOutputFormat

- Podemos formatar os parâmetros usando uma sintaxe diferente. Após o número, podemos especificar o tipo de formato, e após o estilo ou padrão, todos separados por vírgulas.
- Existem duas classes de tipos de formatos: dados (datas, horas, e números) e escolha.
- Abaixo, um exemplo usando formato de dados:

```
<h:outputFormat value="Hey {0}, you were born on {1, date, medium}.">
  <f:param value="#{user.firstName}"/>
  <f:param value="#{user.dateOfBirth}"/>
</h:outputFormat>
```

# Apresentando dados com componentes de saída

## HtmlOutputFormat

- Às vezes, desejamos apresentar parte de uma string baseado no valor da propriedade de um bean. Para isso, usamos o formato de escolha.
- Ele é feito de uma série de comparações valor/apresentação.
- Abaixo, um exemplo usando formato de escolha:

```
<h:outputFormat value="You have visited us {0} {0, choice, 0#times|1#
    time|2#times}.">
  <f:param value="#{user.numberOfVisits}"/>
</h:outputFormat>
```

# Escopos

- Armazenam os *beans* e outros objetos. Quando de sua criação, devemos informar em que momento e durante quanto tempo eles estarão disponíveis.
- Podem ser declarados por anotações ou no arquivo `faces-config.xml`
  - session
  - request
  - application. O atributo `eager` permite criar o *bean* antes que a primeira página seja carregada.
  - conversation
  - view

# Escopos I
## Conversation

- Existe durante a execução de um conjunto de páginas.
- Associado a uma página ou aba do navegador.
  - Usar a anotação `@ConversationScope`
  - Usar a variável de instância

    `private @Inject Conversation conversation;` que será automaticamente inicializada
  - Chamar `conversation.begin()` para modificar o escopo de `request` para `conversation`.
  - Chamar `conversation.end()` para remover o *bean* do escopo `conversation`.

# Escopos II
## Conversation

```
@Named
@ConversationScoped
public class QuizBean implements Serializable {
  @Inject Conversation conversation;
  ...
  public void setAnswer(String newValue) {
    try {
      if (currentIndex == 0) conversation.begin();
      int answer = Integer.parseInt(newValue.trim());
      if (getCurrent().getSolution() == answer) score++;
      currentIndex = (currentIndex + 1) % problems.size();
      if (currentIndex == 0) conversation.end();
    } catch (NumberFormatException ex) {
    }
  }
}
```

# Escopos

- view
  - Persiste enquanto a mesma página JSF está sendo reapresentada. Quando o usuário navega para outra página, o *bean* sai do escopo.
  - Particularmente útil para aplicações Ajax.
- none
  - Não está associado a um escopo.
  - Sempre é criado um novo objeto quando chamado.
  - Útil para especificar *beans* que são propriedades de outros *beans*.

# Injeção

- Serve para utilizar *beans* de forma conjunta.

```
@Named
@SessionScoped
public class EditBean {
  @Inject private UserBean currentUser;
  ...
}
```

# Escopos

- Um bean gerenciado não pode referenciar um objeto com um ciclo de vida mais curto.
- A funcionalidade de criação de beans gerenciados não suporta ciclos.

| **Um objeto armazenado neste escopo...** | **Pode referenciar objetos neste escopo...** |
|---|---|
| none | none |
| application | none, application |
| session | none, application, session |
| view | none, application, view, session |
| request | none, application, session, view, request |

# Anotações do ciclo de vida

- Com as anotações `@PostConstruct` e `@PreDestroy` pode-se especificar métodos que são automaticamente chamados antes do *bean* ser construído e antes dele sair do escopo

```
public class MyBean {
  @PostConstruct
  public void initialize() {
    // initialization code
  }
  @PreDestroy
  public void shutdown() {
    // shutdown code
  }
  // other bean methods
}
```

# Criando e inicializando beans

- Beans gerenciados podem ser configurados através do elemento `<managed-beans>`.

```
<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>org.jia.examples.UserBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>firstName</property-name>
    <value>Mark</value>
  </managed-property>
  <managed-property>
    <property-name>lastName</property-name>
    <value>Pippins</value>
  </managed-property>
</managed-bean>
```

# Inicializando propriedades Listas ou Arrays

- Caso um atributo seja do tipo `List` ou array, podemos inicializá-lo com valores padrão, através do elemento `<list-entries>`.
- Caso seja o tipo `List`, o objeto criado será um `ArrayList`.

```
<managed-property>
  <property-name>favoriteSites</property-name>
  <list-entries>
    <value>http://www.jsfcentral.com</value>
    <value>http://www.theserverside.com</value>
    <value>http://www.ibm.com/developerworks/</value>
    <value>http://otn.oracle.com</value>
    <value>http://www.java.net</value>
    <value>http://www.manning.com</value>
  </list-entries>
</managed-property>
```

# Inicializando propriedades Listas ou Arrays

- Os valores padrão são do tipo String. Se os valores devem ser convertidos para um tipo específico, podemos usar o elemento `<value-class>`.

```
<managed-property>
  <property-name>favoriteNumbers</property-name>
  <list-entries>
    <value-class>java.lang.Integer</value-class>
    <value>31415</value>
    <value>278</value>
    <value>304</value>
    <value>18</value>
    <value>811</value>
    <value>914</value>
  </list-entries>
</managed-property>
```

# Inicializando propriedades `Map`

- Caso um atributo seja do tipo `Map`, podemos inicializá-lo com valores padrão, através do elemento `<map-entries>`, com elementos filho `<map-entry>`.
- Cada elemento filho possui dois subelementos: `<key>` e `<value>`.

```
<managed-property>
  <property-name>favoriteSitesMap</property-name>
  <map-entries>
    <map-entry>
      <key>JSF Central</key>
      <value>http://www.jsfcentral.com</value>
    </map-entry>
    <map-entry>
      <key>TheServerSide.com</key>
      <value>http://www.theserverside.com</value>
    </map-entry>
</managed-property>
```

# Inicializando propriedades `Map`

- Podemos informar os tipos padrão da chave e dos valores. Para isso, usamos os elementos `<key-class>` e `<value-class>`.

```
<managed-property>
  <property-name>favoriteNumbersMap</property-name>
  <map-entries>
    <key-class>java.lang.Integer</key-class>
    <map-entry>
      <key>31415</key>
      <value>A pi-like integer.</value>
    </map-entry>
    <map-entry>
      <key>278</key>
      <null-value/>
    </map-entry>
  </map-entries>
</managed-property>
```

# Declarando listas e mapeamentos como beans gerenciados

- Podemos declarar uma lista ou um mapeamento como beans gerenciados. Basta especificarmos o tipo concreto e a lista ou mapeamento de valores.

```
<managed-bean>
  <description>List of favorite sites.</description>
  <managed-bean-name>favoriteSites</managed-bean-name>
  <managed-bean-class>java.util.ArrayList</managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <list-entries>
    <value>http://www.jsfcentral.com</value>
    <value>http://www.theserverside.com</value>
    <value>http://www.ibm.com/developerworks/</value>
    <value>http://otn.oracle.com</value>
    <value>http://www.java.net</value>
    <value>http://www.manning.com</value>
  </list-entries>
</managed-bean>
```

# Setando valores com expressões de valores vinculados I

- Quando se está especificando um valor de um bean gerenciado, pode-se usar uma expressão de valor vinculado (*value-binding expression*). Elas são expressões do tipo JSF EL que referenciam uma propriedade de um bean ou um elemento de coleção, sendo úteis para inicialização de propriedades de beans baseado nos valores de outros beans.

# Setando valores com expressões de valores vinculados II

```
<managed-bean>
  <managed-bean-name>defaultFavoriteSites</managed-bean-name>
  <managed-bean-class>java.util.ArrayList</managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
  <list-entries>
    <value>http://www.jsfcentral.com</value>
    <value>http://www.theserverside.com</value>
    <value>http://www.ibm.com/developerworks/</value>
    <value>http://otn.oracle.com</value>
    <value>http://www.java.net</value>
    <value>http://www.manning.com</value>
  </list-entries>
</managed-bean>
<managed-bean>
  <description>Default user object.</description>
  <managed-bean-name>newUser</managed-bean-name>
  <managed-bean-class>org.jia.examples.UserBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
```

# Setando valores com expressões de valores vinculados III

```
  <managed-property>
    <property-name>favoriteSites</property-name>
    <value>#{defaultFavoriteSites}</value>
  </managed-property>
  <managed-property>
    <property-name>favoriteAnimal</property-name>
    <value>donkey</value>
  </managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>exampleForm</managed-bean-name>
  <managed-bean-class>org.jia.examples.TestForm</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>user</property-name>
    <value>#{newUser}</value>
  </managed-property>
</managed-bean>
```

Parte IX

# Navegação em JSF

# Static Navigation

- How your application can move from one page to the next, depending on user actions and the outcomes of decisions in the business logic.
- **Static Navigation**:
  - Clicking a particular button always selects a fixed JSF page for rendering the response.
  - Example:

```
<h:commandButton label="Login" action="welcome"/>
```

- The value of the action attribute is called the *outcome*. It can be optionally mapped to a *view ID*.
- A JSF page is called a *view*.
- If you don't provide such a mapping for a particular outcome, the outcome is transformed into a view ID, using the following steps:
  1. If the outcome doesn't have a file extension, then append the extension of the current view.
  2. If the outcome doesn't start with a / , then prepend the path of the current view.

# Dynamic Navigation

- The page flow does not just depend on which button you click but also on the inputs that you provide.
- For example, submitting a login page may have two outcomes: success or failure.
- The submit button must have a *method expression*, such as:

```
<h:commandButton label="Login" action="#{loginController.
    verifyUser}"/>
```

- A method expression in an `action` attribute has no parameters. It can have any return type.
- Steps when the `action` attribute is a method expression:
  1. The specified bean is retrieved.
  2. The referenced method is called and returns an outcome string.
  3. The outcome string is transformed into a view ID.
  4. The page corresponding to the view ID is displayed.

# Mapping Outcomes to View IDs

- Separation of presentation from business logic.

```
<navigation-rule>
  <from-view-id>/index.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/welcome.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>failure</from-outcome>
    <to-view-id>/newuser.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

- Example

# Redirection

- If you don't use navigation rules, add the string `?faces-redirect=true`

```
<h:commandButton label="Login" action="welcome?faces-redirect=true
    "/>
```

- In a navigation rule:

```
<navigation-case>
  <from-outcome>success</from-outcome>
  <to-view-id>/success.xhtml</to-view-id>
  <redirect/>
</navigation-case>
```

# Redirection
## Flash

- In a redirect, request scoped beans no longer exist.
- JSF provides a flash object that can be populated in one request and used in the next.

```
ExternalContext.getFlash().put("message", "Your password is about
    to expire");
```

- In a JSF page, you reference the flash object with the flash variable.

```
#{flash.message}
```

- You can even keep a value in the flash for more than one request.

```
#{flash.keep.message}
```

# Using from-action

- Can be useful if you have two separate actions with the same outcome string.
- Suppose the startOverAction returns the string "again" instead of "startOver", similarly to the answerAction method.

```
<navigation-case>
  <from-action>#{quizBean.answerAction}</from-action>
  <from-outcome>again</from-outcome>
  <to-view-id>/again.xhtml</to-view-id>
</navigation-case>
<navigation-case>
  <from-action>#{quizBean.startOverAction}</from-action>
  <from-outcome>again</from-outcome>
  <to-view-id>/index.xhtml</to-view-id>
</navigation-case>
```

# Conditional Navigation Cases

- You can supply an if element that activates a navigation case only when a condition is fulfilled.
- Supply a value expression for the condition.

```
<navigation-case>
  <from-outcome>previous</from-outcome>
  <if>#{quizBean.currentQuestion != 0}</if>
  <to-view-id>/main.xhtml</to-view-id>
</navigation-case>
```

# Dynamic Target View IDs

- The to-view-id element can be a value expression, in which case it is evaluated. The result is used as the view ID.

```
<navigation-rule>
  <from-view-id>/main.xhtml</from-view-id>
  <navigation-case>
    <to-view-id>#{quizBean.nextViewID}</to-view-id>
  </navigation-case>
</navigation-rule>
```

Parte X

# Standard JSF Tags

# Attributes, Parameters, and Facets

- The f:attribute, f:param, and f:facet tags are general-purpose tags to add information to a component.
- f:attribute: You can set an attribute in a page and later retrieve it programatically (attribute map).
- f:param: you need to supply a number of values with the same name (or no name at all).
- f:facet: adds a named component to a component's facet map. Usually rendered in a special place.

```
<h:outputText value="#{payment.card}">
  <f:attribute name="separator" value="-" />
</h:outputText>
```

# An Overview of the JSF HTML Tags

- Inputs (input...)
- Outputs (output..., graphicImage)
- Commands (commandButton and commandLink)
- GET Requests (button, link, outputLink)
- Selections (checkbox, listbox, menu, radio)
- HTML pages (head, body, form, outputStylesheet, outputScript)
- Layouts (panelGrid, panelGroup)
- Data table (dataTable and column)
- Errors and messages (message, messages)

# Common Attributes
## Basic

- The id attribute lets you do the following:
  - Access JSF components from other JSF tags

```
<h:inputText id="name" .../>
<h:message for="name"/>
```

  - Obtain component references in Java code

```
UIComponent component = event.getComponent().findComponent("
    name");
```

  - Access HTML elements with scripts

# Common Attributes
## Basic

- The binding attribute
- Define the component as an instance field of a class.

```
<h:inputText binding="#{form.nameField}" .../>

private UIComponent nameField = new UIInput();
public UIComponent getNameField() { return nameField; }
public void setNameField(UIComponent newValue) { nameField =
    newValue; }
```

# Common Attributes
## Basic

- You use the rendered attribute to include or exclude a component, depending on a condition.

```
<h:commandButton ... rendered="#{user.loggedIn}"/>
```

- To conditionally include a group of components, include them in an h:panelGrid with a rendered attribute.

# Common Attributes
## Basic

- You can place stylesheets, JavaScript files, images, and other files into a resources directory in the root of your web application.
- Subdirectories of this directory are called libraries.

```
<h:outputStylesheet library="css" name="styles.css"/>
<h:outputScript name="jsf.js" library="javascript" target="head"
    />
<h:graphicImage name="logo.png" library="images"/>
```

- The target informs where the element should appear.
- If there is no target element, it is inserted in the current location.

# Common Attributes
## Basic

- h:panelGrid generates the HTML markup for laying out components in rows and columns.

```
<h:panelGrid columns="3">
...
</h:panelGrid>
```

- The columnClasses and rowClasses specify lists of CSS classes that are applied to columns and rows.

```
rowClasses="evenRows, oddRows"
```

- h:panelGroup groups two or more components so they are treated as one.

# Form Element I

- prependId: ID of this form is prepended to the IDs of its components
- JSF form submissions are implemented with the POST method.
- id: we can access them with JavaScript.

```
<h:form>
...
<h:inputSecret id="password" .../>
<h:inputSecret id="passwordConfirm" .../>
...
<h:commandButton type="button" onclick="checkPassword(this.form)"
    />
...
</h:form>
```

- All form controls generated by JSF have names that conform to formName:componentName

# Form Element II

```
<form id="_id0" method="post" action="/javascript/faces/index.
    xhtml" enctype="application/x-www-form-urlencoded">
  ...
  <input id="_id0:password"
  type="text" name="registerForm:password"/>
  ...
  <input type="button" name="_id0:_id5"
  value="Submit Form" onclick="checkPassword(this.form)"/>
  ...
</form>
```

# Displaying Text and Images I

- h:outputText: generates mere text (exception: specifying the style or styleClass attributes)
- You can insert value expressions, such as `#{msgs.namePrompt}` into your page.
- Usage:
  - To produce styled output
  - In a panel grid to make sure that the text is considered one cell of the grid
  - To generate HTML markup
- The h:outputFormat tag formats a compound message with parameters.

```
<h:outputFormat value="{0} is {1} years old">
  <f:param value="Bill"/>
  <f:param value="38"/>
</h:outputFormat>
```

# Displaying Text and Images II

- The h:graphicImage tag generates an HTML img element.

```
<h:graphicImage library="images" name="de_flag.gif"/>
<h:graphicImage url="/resources/images/de_flag.gif"/>
<h:graphicImage value="#{resources['images:de_flag.gif']}"/>
```

# Apresentando mensagens com HtmlMessage

- JSF suporta mensagens que reportam erros de validação e conversão, bem como informações gerais. Cada mensagem possui um nível de severidade.

| **Nível de severidade** | **Descrição** |
|---|---|
| Info | Representa texto a ser enviado para o usuário que não é erro. |
| Warn | Indica que um erro pode ter ocorrido. |
| Error | Indica um erro. Recomendado para mensagens de validação. |
| Fatal | Indica um erro sério. |

- Mensagens estão associadas a componentes. Se várias mensagens forem geradas, somente a última será apresentada.

# Apresentando mensagens com HtmlMessage

- Permite modificar o estilo baseado na severidade da mensagem. Pode-se também controlar o nível de detalhe da mensagem a ser apresentado.
- O atributo `for` é obrigatório e referencia o componente a serem apresentadas as mensagens. Abaixo, alguns atributos de `<h:message>`:

| Propriedade | Tipo | Padrão | Obrigatório | Descrição |
|---|---|---|---|---|
| `for` | String | Nenhum | Sim | O identificador do componente a serem apresentadas as mensagens. |
| `showDetail` | boolean | false | Não | Indica se deve mostrar os detalhes da mensagem. |
| `showSummary` | boolean | true | Não | Indica se deve mostrar o resumo da mensagem. |
| `errorClass` | String | Nenhum | Não | Classe CSS para mensagem com severidade Error. |

# Apresentando mensagens com HtmlMessages

- Utilizado para o caso de se desejar apresentar várias mensagens. Não está limitado a um único componente.
- Pode-se apresentar mensagens criadas em eventos e/ou criadas por validadores e conversores.

| **Propriedade** | **Tipo** | **Padrão** | **Obrigatório** | **Descrição** |
|---|---|---|---|---|
| `layout` | String | list | Não | Especifica como mostrar a mensagem. Valores possíveis: list, table |
| `globalOnly` | boolean | false | Não | Controla se deve mostrar mensagens globais (eventos x componentes). |

# Parte XI

# Facelets

# Introdução

- Designed to implement flexible UIs.
- Facelets tags can be grouped in these categories:
  - Including content from other XHTML pages (ui:include)
  - Building pages from templates (ui:composition, ui:decorate, ui:insert, ui:define, ui:param)
  - Creating custom components without writing Java code (ui:component, ui:fragment)
  - Miscellaneous utilities (ui:debug, ui:remove, ui:repeat)
- To use Facelets tags, add the following namespace declaration to your JSF page: `xmlns:ui="http://java.sun.com/jsf/facelets"`

# Templating with Facelets

- Facelets lets you encapsulate common layout and styling in a template, so that you can update the look of your site by making changes to the template, not the individual pages.
- Templating splits a view into two XHTML pages: one that defines common functionality (a template), and another that defines functionality that differs between views (a composition).
- ui:insert: specify default content
- ui:include: include default content from another file
- To make use of a template, you use a ui:composition tag with a template attribute
- When the template is loaded, each ui:insert is replaced with the contents of the corresponding ui:define.

# Templating with Facelets

- Decorators are a more content-centric approach.
- You write your pages as usual, but you surround the contents with a ui:decorate tag that has a template attribute.
- ui:composition trims all surrounding contents, whereas ui:decorator doesn't (and therefore requires a ui:composition in the template).
- ui:param sets an EL variable for use in a template

# Templating with Facelets

- Implementing a custom Facelets tag with JSF 2.0 is a two-step process:
  1. Implement the custom tag (or component) in an XHTML file.
  2. Declare the custom tag in a tag library descriptor.
- The tag library file defines:
  1. A namespace for the tags in this library
  2. A name for each tag
  3. The location of the template
- Custom Facelets tags are not as powerful as full-fledged JSF components.
- You cannot attach functionality, such as validators or listeners.

# Loose Ends

- The ui:debug tag is useful during development, so developers can instantly see the current page's component tree and the application's scoped variables.
- Put the ui:debug tag in a template.
- ui:remove: do not evaluate JSF tags or EL.
- If you set the context parameter `javax.faces.FACELETS_SKIP_COMMENTS` to true in web.xml, then XML comments are skipped.

# Parte XII

# Data Tables

# Data Tables I

- The h:dataTable tag iterates over data to create an HTML table.

```
<h:dataTable value="#{items}" var="item">
  <h:column>
    <!-- left column components -->
    #{item.aPropertyName}
  </h:column>
  <h:column>
    <!-- next column components -->
    <h:commandLink value="#{item.anotherPropertyName}" action="...
        "/>
  </h:column>
  <!-- add more columns, as desired -->
</h:dataTable>
```

- The value attribute must be one of the following:
  - A Java object
  - An array
  - An instance of java.util.List
  - An instance of java.sql.ResultSet

# Data Tables II

  - An instance of javax.servlet.jsp.jstl.sql.Result
  - An instance of javax.faces.model.DataModel
- The body of h:dataTable tags can contain only h:column tags; all other component tags are ignored.

# Headers, Footers, and Captions

- Caption, column headers and footers are specified with facets.

```
<h:dataTable captionClass="caption">
  <f:facet name="caption">An Array of Names:</f:facet>
  ...
  <h:column headerClass="columnHeader" footerClass="columnFooter">
    <f:facet name="header">
      <!-- header components go here -->
    </f:facet>
    <!-- column components go here -->
    <f:facet name="footer">
      <!-- footer components go here -->
    </f:facet>
  </h:column>
  ...
</h:dataTable>
```

# Styles

- h:dataTable has attributes that specify CSS classes for the following:
  - The table as a whole (styleClass)
  - Column headers and footers (headerClass and footerClass)
  - Individual columns (columnClasses)
  - Individual rows (rowClasses)

```
<h:dataTable value="#{order.all}" var="order" styleClass="orders"
    headerClass="ordersHeader" columnClasses="oddColumn,evenColumn
    ">
```

- In this case, h:dataTable reuses the column classes, starting with the first.

# The ui:repeat tag I

- The ui:repeat tag repeatedly inserts its body into the page.

```
<table>
  <ui:repeat value="#{tableData.names}" var="name">
  <tr>
    <td>#{name.last},</td>
    <td>#{name.first}</td>
  </tr>
  </ui:repeat>
</table>
```

- Attributes:
  - `offset` is the index at which the iteration starts (default: 0)
  - `step` is the difference between successive index values (default: 1)
  - `size` is the number of iterations (default: (size of the collection – `offset`) / `step`)
- The `varStatus` attribute sets a variable that reports on the iteration status.

# The ui:repeat tag II

- Boolean properties `even`, `odd`, `first`, and `last`, which are useful for selecting styles.
- Integer properties `index`, `begin`, `step`, and `end`, which give the index of the current iteration and the starting offset, step size, and ending offset.

# Database Tables

- Don't use a result set returned from the Statement.executeQuery method.
- In order to render that result set, the underlying database connection has to stay open.
- A better way is to use a wrapper that holds the query result, such as javax.sql.CachedRowSet (recommended) or javax.servlet.jsp.jstl.Result (older).

# Parte XIII

# Conversion and Validation

# Overview of the Conversion and Validation Process

- First, the user fills in a field of a web form. We call this value the *request value*.
- In the Apply Request Values phase, the JSF implementation stores the request values in component objects. A value stored in the component object is called a *submitted value*.
- A *conversion* process transforms the submitted values to business logic values. The values are first stored inside the component objects as *local values*. After conversion, the local values are *validated*.
- After all local values have been validated, the Update Model Values phase starts, and the local values are stored in beans, as specified by their value references.

# Using Standard Converters
## Conversion of Numbers and Dates

```
<h:inputText value="#{payment.amount}">
  <f:convertNumber minFractionDigits="2"/>
</h:inputText>

<h:inputText value="#{payment.date}">
  <f:convertDateTime pattern="MM/yyyy"/>
</h:inputText>

<h:outputText value="#{payment.date}" converter="javax.faces.DateTime"
    />

<h:outputText value="#{payment.date}">
  <f:converter converterId="javax.faces.DateTime"/>
</h:outputText>
```

# Conversion Errors

- When a conversion error occurs, the JSF implementation carries out the following actions:
  - The component whose conversion failed posts a message and declares itself invalid.
  - The JSF implementation redisplays the current page immediately after the Process Validations phase has completed. The redisplayed page contains all values that the user provided—no user input is lost.
- Add h:message tags to see the messages that are caused by conversion and validation errors. It produces only the first message.
- The h:messages tag shows all messages from all components.
- To replace a standard message, set up a message bundle. Add the replacement message, using the appropriate key.

```
<faces-config>
  <application>
    <message-bundle>com.corejsf.messages</message-bundle>
  </application>
  ...
</faces-config>
```

# Using Standard Validators
## Validating String Lengths and Numeric Ranges

- JavaServer Faces has built-in mechanisms that let you carry out the following validations:
  - Checking the length of a string
  - Checking limits for a numerical value (for example, $> 0$ or $\leq 100$)
  - Checking against a regular expression
  - Checking that a value has been supplied

```
<h:inputText id="card" value="#{payment.card}">
  <f:validateLength minimum="13"/>
</h:inputText>

<h:inputText id="amount" value="#{payment.amount}">
  <f:validateLongRange minimum="10" maximum="10000"/>
</h:inputText>

<h:inputText id="card" value="#{payment.card}">
  <f:validator validatorId="javax.faces.validator.LengthValidator">
    <f:attribute name="minimum" value="13"/>
  </f:validator>
</h:inputText>
```

# Using Standard Validators
## Bypassing Validation

- Validation errors (as well as conversion errors) force a redisplay of the current page.
- This behavior can be problematic with certain navigation actions. For example, with a Cancel button.
- If a command has the immediate attribute set, then the command is executed during the Apply Request Values phase.

```
<h:commandButton value="Cancel" action="cancel" immediate="true"/>
```

# Bean Validation

- JSF integrates with the *Bean Validation Framework*, a general framework for specifying validation constraints. Validations are attached to fields or property getters of a Java class, like this:

```
public class PaymentBean {
  @Size(min=13) private String card;
  @Future public Date getDate() { ... }
  ...
}
```

- The JSF pages are not concerned with validation. The model class contains the validation annotations.

# Programming with Custom Converters and Validators
## Implementing Custom Converter Classes

- A converter is a class that converts between strings and objects.
- A converter must implement the Converter interface, which has the following two methods:

```
Object getAsObject(FacesContext context, UIComponent component,
    String newValue)
String getAsString(FacesContext context, UIComponent component,
    Object value)
```

# Programming with Custom Converters and Validators I
## Specifying Converters

- One mechanism for specifying converters involves a symbolic ID that you register with the JSF application.

```
@FacesConverter("com.corejsf.Card")
public class CreditCardConverter implements Converter
```

```
<converter>
  <converter-id>com.corejsf.Card</converter-id>
  <converter-class>com.corejsf.CreditCardConverter</converter-
      class>
</converter>
```

- Now we can use the f:converter tag and specify the converter ID:

```
<h:inputText value="#{payment.card}">
  <f:converter converterId="com.corejsf.Card"/>
</h:inputText>
```

```
<h:inputText value="#{payment.card}" converter="com.corejsf.Card"
    />
```

# Programming with Custom Converters and Validators II

## Specifying Converters

- If you are confident that your converter is appropriate for all conversions between String and CreditCard objects, then you can register it as the default converter for the CreditCard class.

```
@FacesConverter(forClass=CreditCard.class)
```

```
<converter>
  <converter-for-class>com.corejsf.CreditCard</converter-for-class
      >
  <converter-class>com.corejsf.CreditCardConverter</converter-
      class>
</converter>
```

- Now you do not have to mention the converter any longer. It is automatically used whenever a value reference has the type CreditCard.

# Programming with Custom Converters and Validators
## Reporting Conversion Errors

- When a converter detects an error, it should throw a ConverterException.

```
if (foundInvalidCharacter) {
  FacesMessage message = new FacesMessage("Conversion error
      occurred. ", "Invalid card number. ");
  message.setSeverity(FacesMessage.SEVERITY_ERROR);
  throw new ConverterException(message);
}
```

- You can set an attribute of the component to which you attach a converter.

```
<h:outputText value="#{payment.card}">
  <f:converter converterId="CreditCard"/>
  <f:attribute name="separator" value="-"/>
</h:outputText>

separator = (String) component.getAttributes().get("separator");
```

# Programming with Custom Converters and Validators
## Implementing Custom Validator Classes

- Your validator class must implement the javax.faces.validator.Validator interface.

```
void validate(FacesContext context, UIComponent component, Object
    value)
```

- If validation fails, generate a FacesMessage that describes the error, construct a ValidatorException from the message, and throw it:

```
if (validation fails) {
  FacesMessage message = ...;
  message.setSeverity(FacesMessage.SEVERITY_ERROR);
  throw new ValidatorException(message);
}
```

# Programming with Custom Converters and Validators I
## Registering Custom Validators

- Now that we have created a validator, we need to give it an ID.

```
@FacesValidator("com.corejsf.Card")
public class CreditCardValidator implements Validator
```

```
<validator>
  <validator-id>com.corejsf.Card</validator-id>
  <validator-class>com.corejsf.CreditCardValidator</validator-
      class>
</validator>
```

- You specify the validator ID in the f:validator tag:

```
<h:inputText id="card" value="#{payment.card}" required="true">
  <f:converter converterId="com.corejsf.Card"/>
  <f:validator validatorId="com.corejsf.Card"/>
</h:inputText>
```

- You can also add the validation method to an existing class and invoke it through a method expression, like this:

# Programming with Custom Converters and Validators II

## Registering Custom Validators

```
<h:inputText id="card" value="#{payment.card}" required="true"
    validator="#{payment.luhnCheck}"/>

public class PaymentBean {
  ...
  public void luhnCheck(FacesContext context, UIComponent
      component, Object value) {
    ... // same code as in the preceding example
  }
}
```

# Programming with Custom Converters and Validators I
## Validating Relationships between Multiple Components

- The validation mechanism in JSF was designed to validate a single component.
- Attach the validator to the last of the components. By the time its validator is called, the preceding components have passed validation and had their local values set.
- The last component has passed conversion, and the converted value is passed as the Object parameter of the validation method.
- An alternative approach is to attach the validator to a hidden input field that comes after all other fields on the form:

```
<h:inputHidden id="datecheck" validator="#{bb.validateDate}" value
    ="needed"/>
```

Parte XIV

# Event Handling

# Introduction

- JSF supports four kinds of events:
  - Value change events
  - Action events
  - Phase events
  - System events

# Value Change Events

- You can keep dependent components in synch with value change events, which are fired by input components after their new value has been validated.

```
<h:selectOneMenu value="#{form.country}" onchange="submit()"
    valueChangeListener="#{form.countryChanged}">
  <f:selectItems value="#{form.countries}" var="loc" itemLabel="#{
      loc.displayCountry}" itemValue="#{loc.country}"/>
</h:selectOneMenu>
```

# Action Events

- Action events are fired by buttons and links.

```
<h:commandLink actionListener="#{bean.linkActivated}">
...
</h:commandLink>
```

- When you activate a button or link, the surrounding form is submitted and the JSF implementation subsequently fires action events.
- It is important to distinguish between action listeners and actions. In a nutshell, actions are designed for business logic and participate in navigation handling, whereas action listeners typically perform user interface logic and do not participate in navigation handling.

```
<h:commandButton image="/resources/images/mountrushmore.jpg"
    actionListener="#{rushmore.handleMouseClick}" action="#{
    rushmore.act}"/>
```

- Note that the JSF implementation always invokes action listeners before actions.

# Event Listener Tags I

- You can also add action and value change listeners to a component with the f:actionListener and f:valueChangeListener tags.
- The f:actionListener and f:valueChangeListener tags are analagous to the actionListener and valueChangeListener attributes.

```
<h:selectOneMenu value="#{form.country}" onchange="submit()">
  <f:valueChangeListener type="com.corejsf.CountryListener"/>
  <f:selectItems value="#{form.countryNames}"/>
</h:selectOneMenu>
```

- The tags have one advantage over the attributes: Tags let you attach multiple listeners to a single component.

# Event Listener Tags II

```
public class CountryListener implements ValueChangeListener {
  public void processValueChange(ValueChangeEvent event) {
    FacesContext context = FacesContext.getCurrentInstance();
    if ("US".equals(event.getNewValue()))
      context.getViewRoot().setLocale(Locale.US);
    else
      context.getViewRoot().setLocale(Locale.CANADA);
  }
}
```

- The f:actionListener tag is analogous to f:valueChangeListener

```
<h:commandButton image="mountrushmore.jpg" action="#{rushmore.
    navigate}">
  <f:actionListener type="com.corejsf.RushmoreListener"/>
</h:commandButton>
```

- If you specify multiple listeners for a component, as we did in the preceding code fragment, the listeners are invoked in the following order:
  1. The listener specified by the listener attribute

# Event Listener Tags III

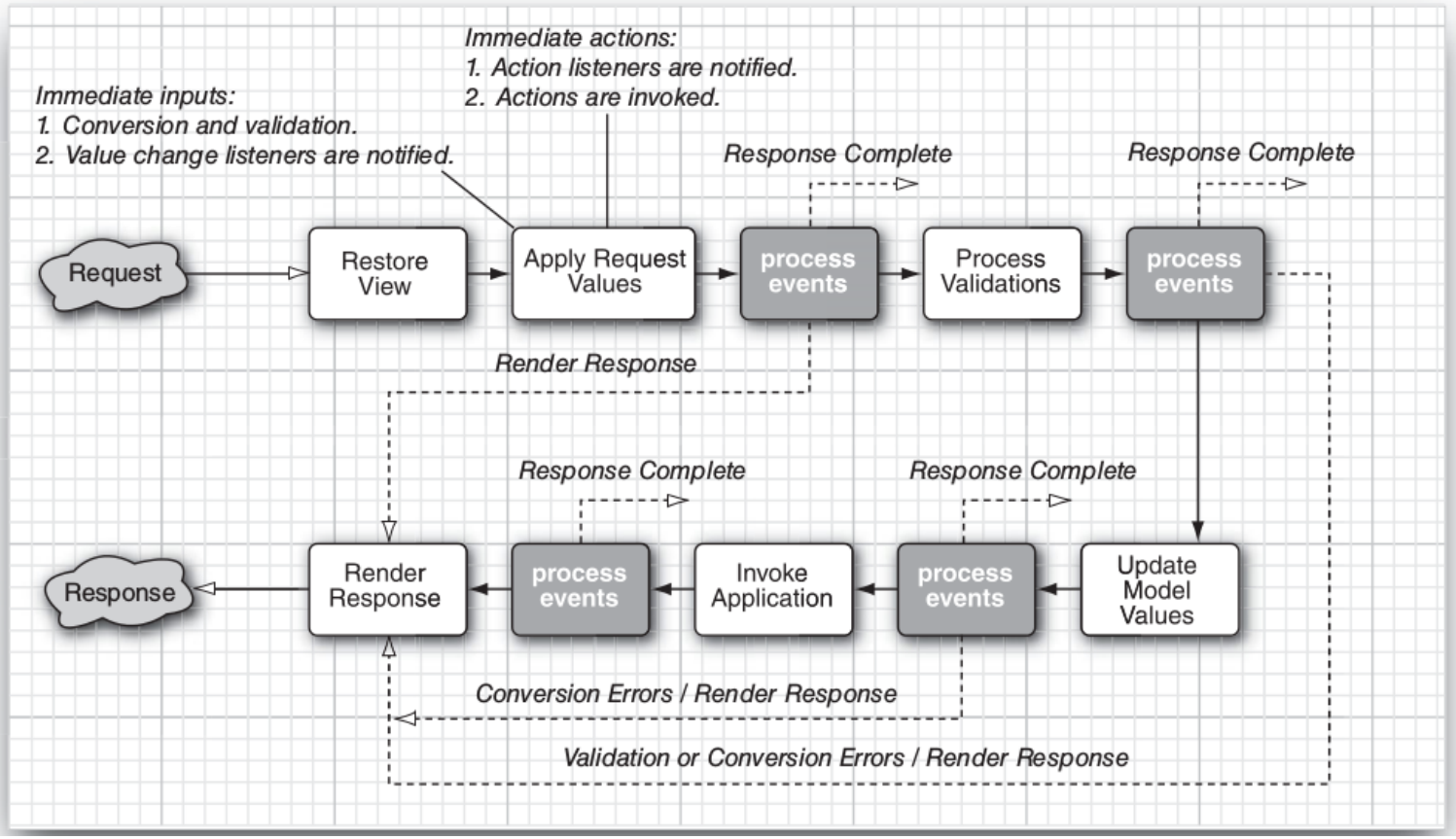
2. Listeners specified by listener tags, in the order in which they are declared

```
<h:commandButton image="mountrushmore.jpg" action="#{rushmore.
    navigate}">
  <f:actionListener type="com.corejsf.RushmoreListener"/>
  <f:actionListener type="com.corejsf.ActionLogger"/>
</h:commandButton>
```

# Immediate Components I

- Value change events are normally fired after the Process Validations phase, and action events are normally fired after the Invoke Application phase.
- When a component has its immediate attribute set, it fires events after the Apply Request Values phase. An immediate input component performs conversion and validation earlier than usual, after completing the Apply Request Values phase. Then it fires a value change event. An immediate command component fires action listeners and actions earlier than usual, after the Apply Request Values phase. That process kicks in the navigation handler and circumvents the rest of the life cycle, moving directly to Render Response.

# Immediate Components II




- You can skip validation when a value change event fires by doing the following:
  1. Adding an immediate attribute to your input tag

# Immediate Components III

2. Calling the renderResponse method of the FacesContext class at the end of your listener

```
public void countryChanged(ValueChangeEvent event) {
  FacesContext context = FacesContext.getCurrentInstance();
  if (US.equals((String) event.getNewValue()))
    context.getViewRoot().setLocale(Locale.US);
  else
    context.getViewRoot().setLocale(Locale.CANADA);
  context.renderResponse();
}
```

# Passing Data from the UI to the Server

- JSF gives us four mechanisms to pass information from the UI to the server:
  - Method expression parameters
  - The f:param tag
  - The f:attribute tag
  - The f:setPropertyActionListener tag

# Phase Events I

- The JSF implementation fires events, called phase events, before and after each life cycle phase.
- You can specify a phase listener for an individual page with a tag

```
<f:phaseListener type="com.corejsf.PhaseTracker"/>
```

- Alternatively, you can specify global phase listeners in a faces configuration file

```
<faces-config>
  <lifecycle>
    <phase-listener>com.corejsf.PhaseTracker</phase-listener>
  </lifecycle>
</faces-config>
```

- You can specify as many as you want. Listeners are invoked in the order in which they are specified in the configuration file.

# Phase Events II

- You implement phase listeners by means of the PhaseListener interface from the javax.faces.event package. That interface defines three methods:
  - PhaseId getPhaseId()
  - void afterPhase(PhaseEvent)
  - void beforePhase(PhaseEvent)

# System Events I

- JSF has a fine-grained notification system in which individual components as well as the JSF implementation notify listeners of many potentially interesting events.
- There are four ways in which a class can receive system events:
  - With the f:event tag:

```
<inputText value="#{...}">
  <f:event name="postValidate" listener="#{bean.method}"/>
</inputText>
```

```
public void listener(ComponentSystemEvent) throws
    AbortProcessingException
```

  - With an annotation for a UIComponent or Renderer class:

```
@ListenerFor(systemEventClass=PreRenderViewEvent.class)
```

  - By being listed as a system event listener in faces-config.xml :

# System Events II

```
<application>
  <system-event-listener>
    <system-event-listener-class>listenerClass</system-event-
        listener-class>
    <system-event-class>eventClass</system-event-class>
  </system-event-listener>
</application>
```

- By calling the subscribeToEvent method of the UIComponent or Application class.

# System Events III

| Event Class | Description | Source Type |
|---|---|---|
| PostConstructApplicationEvent<br>PreDestroyApplicationEvent | Immediately after the application has started; immediately before it is about to be shut down | Application |
| PostAddToViewEvent<br>PreRemoveFromViewEvent | After a component has been added to the view tree; before it is about to be removed | UIComponent |
| PostRestoreStateEvent | After the state of a component has been restored | UIComponent |
| PreValidateEvent<br>PostValidateEvent | Before and after a component is validated | UIComponent |
| PreRenderViewEvent | Before the view root is about to be rendered | UIViewRoot |
| PreRenderComponentEvent | Before a component is about to be rendered | UIComponent |
| PostConstructViewMapEvent<br>PreDestroyViewMapEvent | After the root component has constructed the view scope map; when the view map is cleared[a] | UIViewRoot |
| PostConstructCustomScopeEvent<br>PreDestroyCustomScopeEvent | After a custom scope has been constructed; before it is about to be destroyed | ScopeContext |
| ExceptionQueuedEvent | After an exception has been queued | ExceptionQueuedEvent-Context |

Parte XV

# Composite Components

# Composite Components

- JSF makes it easy to implement custom components in Java code, and by providing a new facility for composing new components from existing ones.
- JSF comes with a tag library for implementing composite components.

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

- Composite components have interfaces and implementations. Implementations are Facelet markup, using standard JSF tags. Interfaces let you expose configurable characteristics of your composite component.

# Using Composite Components I

- To use a composite component, you must first declare a namespace.

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

- You can use any name you want for the namespace, but the namespace's value must always start with `http://java.sun.com/jsf/composite/`.
- The rest of the namespace's value points to the directory, under resources , where the composite component resides.

# Configuring Composite Components I

- Usage

```
<util:icon image="#{resource['images:back.jpg']}" actionMethod="#{
    user.logout}" />
```

- Implementation

```
<composite:interface>
  <composite:attribute name="image"/>
  <composite:attribute name="actionMethod" method-signature="java.
      lang.String action()" />
</composite:interface>

<composite:implementation>
  <h:form>
    <h:commandLink action="#{cc.attrs.actionMethod}">
      <h:graphicImage url="#{cc.attrs.image}" styleClass="icon" />
    </h:commandLink>
  </h:form>
</composite:implementation>
```

# Configuring Composite Components II

- The type attribute must be specified with a fully qualified Java class name. Example:

```
<composite:attribute name ="date"type="java.util.Date"/>
```

- The `type` and `method-expression` attributes for the `composite:attribute` tag are mutually exclusive, with `type` having priority over `method-signature`, should you inadvertently specify both attributes.

# Localizing Composite Components I

- JSF lets you associate a resource bundle with a composite component.
- In your composite component's directory, you create a properties file with the same name of your composite component.
- Once you have a properties file, you can access its contents with the expression #{cc.resourceBundleMap.key} where key is the key from the properties file.

```
<composite:implementation>
...
<p>#{cc.resourceBundleMap.footer}</p>
...
</composite:implementation>
```

# Exposing a Composite's Components I

- editableValueHolder: used to attach validators

```
<composite:interface>
  <composite:editableValueHolder name="nameInput" targets="form:
      name"/>
  <composite:editableValueHolder name="passwordInput" targets="
      form:password"/>
  <composite:editableValueHolder name="inputs" targets="form:name
      form:password"/>
  ...
</composite:interface>

<util:login namePrompt="#{msgs.namePrompt}" passwordPrompt="#{msgs
    .passwordPrompt}" name="#{user.name}" password="#{user.
    password}" loginAction="#{user.login}" loginButtonText="#{msgs
    .loginButtonText}">
  <f:validateLength maximum="10" for="inputs"/>
  <f:validateLength minimum="4" for="nameInput"/>
  <f:validator id="com.corejsf.Password" for="passwordInput"/>
</util:login>
```

# Exposing a Composite's Components II

- The composite:valueHolder tag exposes components, such as output components, that have a non-editable value.
- The composite:actionSource tag exposes compo- nents, such as buttons and links, that fire action events.

```
<composite:interface>
  <composite:actionSource name="loginButton" targets="form:
      loginButton"/>
  ...
</composite:interface>

<composite:implementation>
  ...
  <h:form id="form"... >
    ...
    <h:commandButton id="loginButton" value="#{cc.attrs.
        loginButtonText}" action="#{cc.attrs.loginAction}"/>
    ...
  </h:form>
</composite:implementation>
```

# Exposing a Composite's Components III

```
<util:login ...>
  <f:actionListener for="loginButton" type="com.corejsf.
      LoginActionListener"/>
</util:login>
```

# Facets I

- Another way to add functionality letting page authors specify facets of a composite component.
- We declare the facets in the component's interface with the `composite:facet` tag, and use the `composite:renderFacet` tag in the component's implementation, to render the facets.

```
<util:login ...>
  <f:facet name="header">
    <div class="prompt">#{msgs.loginPrompt}</div>
  </f:facet>
  <f:facet name="error">
    <h:messages layout="table" styleClass="error"/>
  </f:facet>
  ...
</util:login>

<composite:interface>
  ...
  <composite:facet name="header"/>
```

# Facets II

```
  <composite:facet name="error"/>
</composite:interface>

<composite:implementation>
  ...
  <composite:renderFacet name="header"/>
    <h:form ...>
    ...
    </h:form>
  <composite:renderFacet name="error"/>
</composite:implementation>
```

- The `composite:renderFacet` tag renders the supplied facet as a child component. If you want to insert it as a facet instead, use the `composite:insertFacet` tag.

# Facets III

```
<composite:implementation>
  ...
  <h:dataTable>
    <composite:insertFacet name="header"/>
    ...
    <composite:insertFacet name="footer"/>
  </h:dataTable>
  ...
</composite:implementation>
```

# Children

- By default, if you put something in the body of a composite component's tag, JSF will just ignore that content, but you can use the `<composite:insertChildren/>` tag in your composite component's implementation to render the components in the body of your composite component's tag.

```
<util:login...>
  <f:facet name="header" styleClass="header">...</f:facet>
  <f:facet name="error" styleClass="error">...</f:facet>

  <h:link>#{msgs.registerLinkText}</h:link>
</util:login>

<composite:implementation>
  <composite:renderFacet name="header"/>
  ...
  <composite:renderFacet name="error"/>
  <composite:insertChildren/>
</composite:implementation>
```

# Backing Components I

- To add Java code to a composite component, you supply a backing component.
- A backing component has these requirements:
  - It is a subclass of UIComponent.
  - It implements the NamingContainer marker interface.
  - Its family property has the value "javax.faces.NamingContainer".
- There are two ways of designating a backing component for a composite component. The easiest is to follow a naming convention: use the class name libraryName.componentName . In our example, the class name is util.date —that is, a class date in a package util .

# Backing Components II

```
<util:date value="#{user.birthDay}"/>
```

```
package util;
...
public class date extends UIInput implements NamingContainer {
  public String getFamily() {
    return "javax.faces.NamingContainer";
  }
  ...
}
```

# Packaging Composite Components in JARs I

- You can package your composite components in JAR files.
- All you have to do is put your composite component, and its artifacts, such as JavaScript, stylesheets, or properties files, under a META-INF directory in the JAR.

# Parte XVI

# Ajax

# Introduction I

- You can handle most of the common Ajax use cases with a tag from JSF's core library: `f:ajax`. It attaches a behavior to a component.

```
<h:inputText value="#{someBean.someProperty}">
  <f:ajax event="keyup" render="someOtherComponentId"/>
</h:inputText>
```

- JSF Ajax requests partially process components on the server, and partially render components on the client when the request returns.
- Here's the recipe for using Ajax with JSF:
  1. Associate a component and an event with an Ajax request.
  2. Identify components to execute on the server during the Ajax request.
  3. Identify components to render after the Ajax request.

# Introduction II

```
<h:inputText id="nameInput" value="#{user.name}">
  <f:ajax event="blur" execute="@this passwordInput" render="
      nameError passwordError"/>
</h:inputText>
<h:outputText id="nameError"/>
...
<h:inputText id="passwordInput"/>
<h:outputText id="passwordError" value="#{user.passwordError}"/>
```

- JSF also lets you associate an Ajax request with a group of Ajax components:

```
<f:ajax>
  <h:form>
    <h:panelGrid columns="2">
      <h:inputText id="name" value="#{user.name}"/>
      <h:inputText id="password" value="#{user.password}"/>
      </h:commandButton value="Submit" action="#{user.login}"/>
    </h:panelGrid>
  </h:form>
</f:ajax>
```

# Ajax Request Monitoring I

- You can monitor Ajax requests with the `f:ajax` tag's `onevent` attribute. That attribute's value must be a JavaScript function. JSF calls that function at each stage of an Ajax request: begin, complete, and success.
- JSF passes a data object to any JavaScript function that's registered with an Ajax call via f:ajax's onevent attribute.

# Ajax Request Monitoring II

| Attribute | Description |
|---|---|
| status | The status of the current Ajax call. Must be one of: begin, complete, or error. |
| type | Either event or status. |
| source | The DOM element that is the source of the event. |
| responseXML | The response to the Ajax request. This object is undefined in the begin phase of the Ajax request. |
| responseText | The XML response, as text. This object is also undefined in the begin phase of the Ajax request. |
| responseCode | The numeric response code of the Ajax request. Like responseXML and responseText, this object is undefined in the begin phase of the Ajax request. |

# Handling Ajax Errors I

- You can use f:ajax's `onerror` attribute to handle errors

```
<f:ajax onerror="handleAjaxError"/>
```

- The value of the `onerror` attribute is a JavaScript function.
- For errors, the data object also contains three properties not present for events:
  - description
  - errorName
  - errorMessage

Tabela: data.status Values for Error Functions

| **Attribute** | **Description** |
|---|---|
| httpError | Response status null or undefined or status $<$ 200 or status $\geqslant$ 300. |
| emptyResponse | There was no response from the server. |
| malformedXML | The response was not well-formed XML. |
| serverError | The Ajax response contains an error element from the server. |

# Queueing Events I

- JSF automatically queues Ajax requests and executes those requests serially, so the last Ajax request always finishes before the next one starts.
- Mixing Ajax and HTTP requests can result in indeterminate behavior.

```
<h:form>
  <h:inputText ...>
    <f:ajax onblur="..."/>
  </h:inputText>
  <h:commandButton value="submit" action="nextPage"/>
</h:form>
```

- The solution to mixing Ajax and regular HTTP requests is simple: Don't do it.

# Queueing Events II

```
<h:form>
  <h:inputText ...>
    <f:ajax onblur="..."/>
  </h:inputText>
  <h:commandButton value="submit" action="nextPage">
    <f:ajax/>
  </h:commandButton>
</h:form>
```