JSF - Java Server Faces

Conceitos do framework

Alisson G. Chiquitto[1]

[1]Sistemas para Internet

Umuarama, 2016

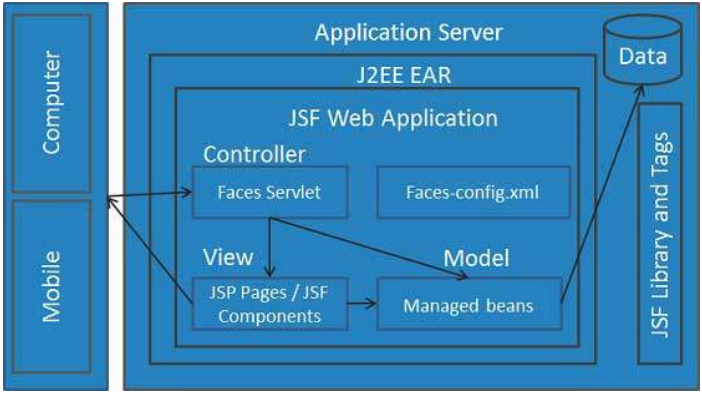
Outline - Seção

Outline - Subseção

JSF

Java Server Faces (JSF) é uma especificação Java para a construção de interfaces de usuário baseadas em componentes para aplicações web.

JSF


Computer
Mobile
Application Server
J2EE EAR
JSF Web Application
Controller
Faces Servlet
Faces-config.xml
View
Model
JSP Pages / JSF Components
Managed beans
Data
JSF Library and Tags

Outline - Seção

Outline - Subseção

XML

XML (*Extensible Markup Language* ou Linguagem de Marcação Extensível) é uma linguagem de marcação baseada em textos.

Tem sido adotada como padrão para a representação e troca de dados entre sistemas.

XML

Exemplo: Cliente

```
<?xml version="1.0" encoding="UTF-8"?>
<cliente>
  <nome apelido="Zezinho">José da Silva</nome>
  <fone>(11) 99999-9999</fone>
  <vivo valor="1" />
</cliente>
```

XML

Exemplo: Lista de Clientes

```
<?xml version="1.0" encoding="UTF-8"?>
<clientes>
  <cliente>
    <nome apelido="Zezinho">José da Silva</nome>
    <fone>(11) 99999-9999</fone>
    <vivo valor="1" />
  </cliente>
  <cliente>
    <nome apelido="Huguinho">Hugo Paulino</nome>
    <fone>(55) 88888-888888</fone>
    <vivo valor="0" />
  </cliente>
</clientes>
```

Declaração XML

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

- Um arquivo XML pode opcionalmente ter uma declaração XML;
- Se tiver a declaração, ela deve ser o primeiro elemento do arquivo;
- A declaração é *case-sensitive*;

Elementos (Tags)

- Um XML é estruturado por uma série de elementos XML, também chamados de *tags* ou *nodes*;
- Os nomes dos elementos devem estar entre os caracteres < e >;
- Todos os elementos devem ser fechados após a sua abertura;

```
<elemento> CONTEUDO </elemento>

<elemento />
```

Elementos (Tags)

Fechamento dos elementos

```
<elemento> CONTEUDO </elemento>

<elemento />
```

- Elementos que possuem conteúdo devem possuir a *tag* de fechamento;
- Elementos que não possuem conteúdo não necessitam da *tag* de fechamento, mas precisam ser fechados com o uso da barra (/) antes do >.

Aninhamento de elementos

- Cada elemento pode conter vários outros elementos;

```
<email>
  <para>chiquitto@gmail.com</para>
  <de>garotabonita@gmail.com</de>
  <assunto>Elogio</assunto>
  <msg>Como você é gato!</msg>
</email>
```

Elemento pai

- Um arquivo XML pode conter apenas um elemento pai, ou seja, dentro do elemento pai deve estar contido todos os outros elementos;
- A única exceção é a declaração XML;

```
<?xml version="1.0" encoding="UTF-8"?>
<empresa>
  <nome>Coca Glue</nome>
  <clientes>...</clientes>
  <fornecedores>...</fornecedores>
  <filiais />
</empresa>
```

Case sensitive

- O XML é *case sensitive*;

Atributos

- Um atributo especifica uma propriedade única do elemento, usando um par `chave=valor`;
- Um elemento pode conter vários atributos;
- Os atributos devem ser colocados na *tag* de abertura;
- Valores devem estar entre aspas (");

```
<telefone ddd="44" op="TIM">9999-9999</telefone>
```

Outline - Seção

Outline - Subseção

Facelets

- Em MVC, representam a *view* pois contêm os componentes que compõem a GUI (*Graphical User Interface*);
- Linguagem XML;
- Utiliza *Expression Language* (EL) para se comunicar com os *Managed Beans*;
- Extensão `.xhtml`;

Facelets

Exemplo

```
<?xml version="1.0" encoding="UTF-8"?>
<html lang="en"
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Facelets Hello World</title>
    </h:head>
    <h:body>
        <p>Olá mundo.</p>
        <p>Bem vindo #{helloWorldMb.nome}</p>
    </h:body>
</html>
```

Expression Language (EL)

- Permite que em um *facelet*, façamos uma referência para algo de um *Managed Bean*;
- Pode ser utilizada para acessar valores ou disparar tarefas no *Managed Bean*;
- A EL fica contida em #{};

```
<p>Bem vindo #{helloWorldMb.nome}</p>
```

Outline - Seção

Outline - Subseção

ManagedBeans

- Os *ManagedBeans* são os *controllers* em MVC;
- Classe Java anotada com `javax.faces.bean.ManagedBean`;

ManagedBeans

Exemplo

```
package mb;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean(name="helloWorldMb")
@RequestScoped
public class HelloWorldMb {
    public String getNome() {
        return "Jean Claude Van Dame Da Silva";
    }
}
```

Outline - Seção

Outline - Subseção

Componentes JSF

O JSF fornece a capacidade de criar Aplicações para Web a partir de coleções de componentes UI (User Interface) que podem entregar conteúdo para os seus clientes, por exemplo, HTML para os Navegadores Web.

Taglibs

Uma *taglib* é uma referência a uma biblioteca JSF, a qual contém uma série de componentes reutilizáveis.

- h HTML: abstrai os componentes de HTML, tais como *head*, *body*, *links*, *dropdown boxes*, caixas de texto, etc.;
- f Facelets: inclui conversores, validadores, *listeners*, passagem de parâmetros, *facets*, internacionalização, etc.
- ui *User Interface*: permite fazer *templating* e outras manipulações de fragmentos de interface;
- cc *Composite Component*: permite criar os seus próprios componentes reutilizáveis;

Taglibs

Carregamento de taglibs

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

h:form

- Responsável por gerar um formulário;

Propriedades:

id Identificação do elemento;

h:form

Exemplo de utilização

```
<h:form id="form">
  Conteúdo
</h:form>
```

h:inputText/h:inputTextarea

- Responsável por gerar elementos HTML de entradas de dados `<input>` e `<textarea>`;

Propriedades:

id	Identificação do elemento;
value	Valor do elemento;

h:inputText/h:inputTextarea

Exemplo de utilização

```
<h:form enctype="multipart/form-data">
  <h:inputText id="nome" label="Nome completo" value="">
</h:form>
```

```
<h:form enctype="multipart/form-data">
  <h:inputTextarea id="comentarios"
  label="Comentários" value="">
</h:form>
```

h:inputSecret

- Responsável por gerar elementos HTML de entradas de dados `<input>` do tipo senha;

Propriedades:

id	Identificação do elemento;
value	Valor do elemento;

h:inputSecret

Exemplo de utilização

```
<h:form enctype="multipart/form-data">
  <h:inputSecret id="senha" label="Senha" value="" />
</h:form>
```

h:selectOneMenu

- Exibe para o usuário uma lista de alternativas;
- Trabalha em conjunto com os elementos `<f:selectItem>` e `<f:selectItems>`;

Propriedades:

id	Identificação do elemento;
value	Valor do elemento;

f:selectItem

- Um item estático para para elementos de lista;

Propriedades:

itemLabel Label da alternativa;

itemValue Valor do elemento;

noSelectionOption Valor booleano indicado sem a alternativa é selecionável;

f:selectItems

- Um conjunto de itens para elementos de lista;

Propriedades:

itemLabel Label da alternativa;

itemValue Valor da alternativa;

value Lista de valores do elemento;

var Nome de variável utilizado para referenciar cada item da lista;

h:selectOneMenu

Exemplo de utilização

```
<h:form enctype="multipart/form-data">
  <h:selectOneMenu value="#{userData.data}">
    <f:selectItem itemValue="1" itemLabel="Item 1" />
    <f:selectItem itemValue="2" itemLabel="Item 2" />
  </h:selectOneMenu>

  <h:selectOneMenu value="#{userData.data}">
    <f:selectItem itemLabel="-- Selecione --" noSelectionOption="true"/>
    <f:selectItems value="#{mb.marcas}" var="marca"
      itemValue="#{marca}" itemLabel="#{marca.nome}"/>
  </h:selectOneMenu>
</h:form>
```

h:inputFile

- Representa um `<input>` do tipo *file*;

Propriedades:

value Valor do elemento;

h:inputFile

Exemplo de utilização

```
<h:form enctype="multipart/form-data">
    Imagem: <h:inputFile value="#{mb.uploadedFile}"/>
</h:form>
```

h:outputText

- Apresenta o valor de `value`;
- Possibilidade de usa-lo em conjunto com conversores;

Propriedades:

value Valor a ser apresentado;

h:outputText

Exemplo de utilização

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

<h:outputText value="#{mb.texto}" />
```

h:outputLabel

- Representa o elemento `<label>` do HTML;

Propriedades:

- **for** Referencia para um `id` de um elemento de entrada de dados;
- **value** Valor a ser apresentado;

h:outputLabel

Exemplo de utilização

```
<h:form id="form">
  <h:outputLabel value="Email" for="email" />
  <h:inputText id="email" value="#{mb.email}" />
</h:form>
```

h:panelGrid

- Renderiza uma tabela cujo número de colunas é definido pela propriedade `columns`;

Propriedades:

columns Número de colunas;

h:panelGrid

Exemplo de utilização

```
<h:form id="form">
  <h:panelGrid id="panel" columns="2">
    <h:outputLabel value="Email" />
    <h:inputText  />
    <h:outputLabel value="Senha" />
    <h:inputSecret />
  </h:panelGrid>
</h:form>
```

h:commandLink/h:commandButton

- Renderiza um *link*/botão *submit*;
- A propriedade `action` pode ser associada a um método de um Managed Bean;
- Navegação disparada por `POST`;
- Necessário estar contido em um `<h:form>`;

Propriedades:

action Método do Managed Bean. O retorno (String) deste método é utilizado para fazer a navegação do usuário;

value Label do elemento;

h:commandLink/h:commandButton

Exemplo de utilização

```
<h:form id="form">
  <h:commandLink value="Editar" action="#{mb.salvar()}" />
</h:form>
```

```
<h:form id="form">
  <h:commandButton value="Editar" action="#{mb.salvar()}" />
</h:form>
```

h:link/h:button

- Renderiza um *link*/botão *submit*;
- Navegação disparada por `GET`;

Propriedades:

outcome Regra de navegação;

value Label do elemento;

h:link/h:button

Exemplo de utilização

```
<h:link value="Editar" outcome="update">
  <f:param name="id" value="#{mb.id}"/>
</h:link>
```

```
<h:button value="Editar" outcome="update">
  <f:param name="id" value="#{mb.id}"/>
</h:button>
```

h:message

- Exibe mensagens de um componente;

Propriedades:

for `id` do elemento que será exibido as mensagens;

h:message

Exemplo de utilização

```
<h:form>
  <h:panelGrid columns="3">
    <h:outputLabel value="Nome" for="nome" />
    <h:inputText id="nome" value="#{mb.nome}" label="Nome" />
    <h:message for="nome" />
  </h:panelGrid>
  <h:commandButton value="Salvar" action="#{mb.salvar()}" />
</h:form>
```

h:messages

- Exibe todas as mensagens geradas para qualquer componente;

Propriedades:

globalOnly Valor booleano indicando se apenas mensagens globais devem ser exibidas;

h:messages

Exemplo de utilização

```
<h:messages />

<h:messages globalOnly="true" />
```

Outline - Seção

Outline - Subseção

Conversores de dados JSF

Os conversores realizam duas tarefas básicas:

- Conversão de String para Objeto: Converter a String que vem do usuário para um objeto;
- Conversão de Objeto para String: Converter um objeto para um formato que pode ser utilizado para gerar tela (String);

O JSF possui alguns conversores nativos para transformação de dados, como por exemplo, Data, Hora e Números.

f:convertDateTime

- Faz a conversão de datas.

```
<h:inputText value="#{mb.valor}">
  <f:convertDateTime pattern="dd/MM/yyyy" />
</h:inputText>

<h:outputText value="#{mb.valor}">
  <f:convertDateTime dateStyle="full" type="both" />
</h:outputText>
```

f:convertDateTime

Propriedade type

- Permite informar se queremos trabalhar com data, hora ou ambos.

Valores:

date	(padrão) Apenas com data;
time	Apenas com hora;
both	Ambos;

f:convertDateTime

Propriedade dateStyle

- Tipo da data.

Valores:

default	(padrão) 01/01/2001;
short	01/01/01;
medium	01/01/2001;
long	01 de Janeiro de 2001;
full	Segunda-Feira, 01 de Janeiro de 2001;

f:convertDateTime

Propriedade timeStyle

- Tipo da hora.

Valores:

default	(padrão) 00:00:00;
short	00:00;
medium	00:00:00;
long	0h0min0s BRT;
full	00h00min00s BRT;

f:convertDateTime

Propriedade pattern

- Formatação do valor;
- Conforme a classe `java.text.SimpleDateFormat`.

Valores:

yy/yyyy	Ano (16/2016);
M/MM/MMM	Mês do ano (1/01/Jan);
d/dd	Dia do mês (1/01);
E/EEEE	Dia da semana (Seg/Segunda-Feira);
HH/KK	Hora (00-23/00-11);
mm	Minutos (00-59);
ss	Segundos (00-59);

f:convertDateTime

Propriedade timeZone

- Define o fuso horário;
- Padrão é GMT (+0);

Exemplos de valores:

GMT Fuso horário 0 (zero);

America/Sao_Paulo Fuso horário para São Paulo (-3);

America/Campo_Grande Fuso horário para Campo Grande/MS (-4);

Japan Fuso horário para Japão (+9);

f:convertDateTime

Definir fuso horário local para toda a aplicação

Adicionar o seguinte parâmetro no arquivo `web.xml`.

```
<web-app>
  <context-param>
    <param-name>
       javax.faces.DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE
    </param-name>
    <param-value>true</param-value>
  </context-param>
</web-app>
```

f:convertDateTime

Propriedade locale

- Informa a localidade.
- A localidade define a formatação e o fuso horário;
- Pode ser uma instância de `java.util.Locale` ou *String*;

Exemplos de valores:

pt_BR	Português/Brasil;
en	Inglês;
en_US	Inglês/EUA;
ja	Japônes;
ja_JP	Japônes/Japão;

f:convertNumber

- Faz a conversão de números.

```
<h:inputText value="#{mb.valor}">
  <f:convertNumber type="currency"/>
</h:inputText>

<h:outputText value="#{mb.valor}">
  <f:convertNumber type="number"/>
</h:outputText>
```

f:convertNumber

Propriedade type

- Define como o número será formatado;

Valores:

currency	Moeda;
number	(padrão) Número;
percent	Porcentagem;

f:convertNumber

Propriedade currencyCode

- Define o código da moeda para ser aplicado quando formatamos valores monetários;
- Valores devem seguir o padrão `ISO 4217`;

Exemplos de valores:

BRL Brasil (Real);

JPY Japão (Iene);

CAD Canadá (Dolar Canadense);

USD Estados Unidos da América (Dolar Americano);

EUR União Européia (Euro);

f:convertNumber

Propriedade currencySymbol

- Define o simbolo da moeda para ser aplicado quando formatamos valores monetários;

Exemplos de valores:

R$ Real;

US$ Dólar Americano;

f:convertNumber

Propriedade locale

- Informa a localidade.
- A localidade define a formatação;
- Pode ser uma instância de `java.util.Locale` ou *String*;

Exemplos de valores:

pt_BR	Português/Brasil;
en	Inglês;
en_US	Inglês/EUA;
ja	Japônes;
ja_JP	Japônes/Japão;

f:convertNumber

Propriedade groupingUsed

- Define o agrupamento de casas de milhar;
- Se for utilizado EL, a expressão deve retornar um valor booleano;

Valores:

true	(padrão) As casas de milhar serão agrupadas;
false	As casas de milhar não serão agrupadas;

f:convertNumber

Propriedade integerOnly

- Define se será exibido apenas os valores inteiros;
- Se for utilizado EL, a expressão deve retornar um valor booleano;

Valores:

true	Exibição sem casas decimais;
false	(padrão) Exibição com casas decimais;

Conversores Personalizados

Podemos criar Conversores Personalizados para realizar a conversão de um tipo de dado específico para *String* e vice versa.

Conversores são implementações da interface `javax.faces.convert.Converter` e possuem dois métodos:

getAsString Recebe um objeto e retorna uma *String*;

getAsObject Recebe uma *String* e retorna um objeto;

Conversores Personalizados

Passos para a criação

1. Criar uma classe que implementa `javax.faces.convert.Converter`;
2. Implementar os métodos `getAsObject()` e `getAsString()` da interface acima;
3. Usar a anotação de classe `@javax.faces.convert.FacesConverter` para atribuir um ID para o conversor;

Conversores Personalizados

Passos para a criação

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

@FacesConverter("uppercaseConverter")
public class UppercaseConverter implements Converter {
    @Override
    public Object getAsObject(FacesContext context,
    UIComponent component, String value) {
        return ((String) value).toUpperCase();
    }

    @Override
    public String getAsString(FacesContext context,
    UIComponent component, Object value) {
        return (String) value;
    }
}
```

Conversores Personalizados

Passos para a utilização

1. Utilizar a *tag* `<f:converter>` no local que deseja aplicar o conversor;
2. Adicionar na *tag* o atributo `id`, contendo a identificação do conversor;

Conversores Personalizados

Passos para a utilização

```
<h:inputText id="nome" value="#{mb.nome}" label="Nome">
  <f:converter converterId="uppercaseConverter"/>
</h:inputText>
```

Outline - Seção

Outline - Subseção

Validadores de dados JSF

O JSF fornece validadores nativos para validar seus componentes UI.

Os validadores testam um valor de entrada, e de acordo com os seus requisitos, ele pode determinar se o valor é válido ou inválido.

f:validateLength

- Utilizada para validar se o comprimento de uma *String* esta dentro de um intervalo;

Propriedades:

minimum	Comprimento mínimo;
maximum	Comprimento máximo;

f:validateLength

```
<h:inputText id="nameInput" value="" label="name" >
  <f:validateLength minimum="5" maximum="8" />
</h:inputText>
```

f:validateLongRange

- Utilizada para validar se um valor inteiro esta dentro de um intervalo;

Propriedades:

minimum	Valor mínimo;
maximum	Valor máximo;

f:validateLongRange

```
<h:inputText id="ageInput" value="" label="age" >
  <f:validateLongRange minimum="5" maximum="200" />
</h:inputText>
```

f:validateDoubleRange

- Utilizada para validar se um valor com casas decimais esta dentro de um intervalo;

Propriedades:

minimum	Valor mínimo;
maximum	Valor máximo;

f:validateDoubleRange

```
<h:inputText id="salaryInput" value="" label="salary" >
  <f:validateDoubleRange minimum="1000.50" maximum="10000.50" />
</h:inputText>
```

f:validateRegex

- Utilizada para validar se uma *String* esta em um formato específico;
- A especificação do padrão deve ser em Expressões Regulares (REGEX);

Propriedades:

pattern Padrão em Expressão Regular;

f:validateRegex

```
<h:inputText value="">
  <f:validateRegex pattern="[a-z]{6,18}"/>
</h:inputText>
```

Validadores Personalizados

Podemos criar Validadores Personalizados para realizar validações de dados específicos.

Validadores são implementações da interface `javax.faces.validator.Validator` e possui um método chamado `validate()`.

Validadores Personalizados

Passos para a criação

1. Criar uma classe que implementa `javax.faces.validator.Validator`;
2. Implementar o método `validate()` da interface acima;
3. Usar a anotação de classe `@javax.faces.validator.FacesValidator` para atribuir um ID para o validador;

Validadores Personalizados

Passos para a criação

```
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

@FacesValidator("emailValidator")
public class EmailValidator implements Validator {
  @Override
  public void validate(FacesContext context, UIComponent component,
  Object value) throws ValidatorException {
    if (!((String) value).contains("@")) {
      FacesMessage msg = new FacesMessage("Email deve conter @");
      msg.setSeverity(FacesMessage.SEVERITY_ERROR);
      throw new ValidatorException(msg);
    }
  }
}
```

Validadores Personalizados

Passos para a utilização

1. Utilizar a *tag* `<f:validator>` no elemento UI que deseja validar o seu valor de entrada;
2. Adicionar na *tag* o atributo `id`, contendo a identificação do validador;

Validadores Personalizados

Passos para a utilização

```
<h:inputText id="email" label="Email" value="#{loginMb.email}">
  <f:validator validatorId="emailValidator" />
</h:inputText>
```

Outline - Seção

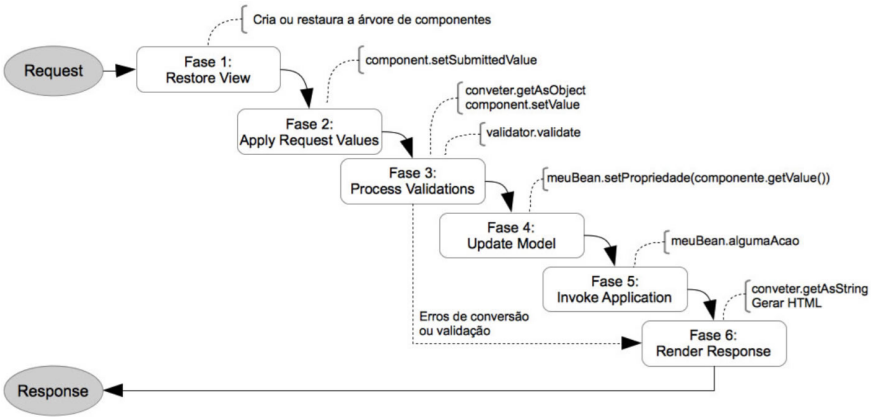
Outline - Subseção

Ciclo de Vida JSF

O ciclo de vida do JSF é dividido em 6 fases:

- Fase 1 Criar ou restaurar a árvore de componentes da tela (*Restore View*);
- Fase 2 Aplicar valores da requisição na árvore de componentes (*Apply Request Values*);
- Fase 3 Converter e Validar (*Validate*);
- Fase 4 Atualizar o modelo (*Update Model*);
- Fase 5 Invocar ação da aplicação (*Invoke Application*);
- Fase 6 Renderizar a resposta (*Render Response*);

Ciclo de Vida JSF



Fase 1

- a primeira tarefa realizada pelo JSF é construir (ou restaurar) a árvore de componentes do arquivo XHTML;
- se o usuário estiver requisitando o XHTML pela primeira vez, após a criação da árvore de componentes, o *request* vai direto para a fase 6;

Fase 2

- buscar os valores informados pelo usuário e atribui-los aos seus respectivos componentes;
- o JSF atribui os valores sem se preocupar se eles são válidos/inválidos;

Fase 3

- faz a conversão dos valores informados pelo usuário (*String*) para os tipos necessários;
- JSF possui conversores "nativos" para `Integer`, `Long` e `java.util.Date`;
- depois da conversão, é feita a validação;

Fase 4

- avalia as *Expression Language* e faz atribui os valores ao XHTML;

Fase 5

- dispara a ação solicitada pelo usuário;

Fase 6

- geração da resposta (HTML) para o usuário;
- cada tag JSF possui um renderizador correspondente;
- utiliza os conversores para gerar *String*;

PhaseListener

Os `PhaseListener` são objetos que serão notificados no início e término de cada etapa.

PhaseListener

Passos para a criação do PhaseListener

- Criar uma classe que implementa `javax.faces.event.PhaseListener`;
- Implementar os métodos do contrato com a *interface*;

PhaseListener

Métodos da classe PhaseListener

getPhaseId	Devolve qual fase do JSF o *listener* irá escutar;
beforePhase	Método chamado antes da fase do ciclo;
afterPhase	Método chamado após a fase do ciclo;

Devemos informar à aplicação da existência do nosso `PhaseListener`. Adicione o código ao arquivo `faces-config.xml`.

```
<faces-config>
  ...
  <lifecycle>
    <phase-listener>
        CLASSE_PHASELISTENER
    </phase-listener>
  </lifecycle>
  ...
</faces-config>
```

PhaseListener

Exemplo: LifeCycleListener.java

```
public class LifeCycleListener implements PhaseListener {

    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }

    public void beforePhase(PhaseEvent event) {
        System.out.println("INICIANDO FASE: " + event.getPhaseId());
    }

    public void afterPhase(PhaseEvent event) {
        System.out.println("FINALIZANDO FASE: " + event.getPhaseId());
    }
}
```

PhaseListener

Exemplo: faces-config.xml

```
<faces-config>
  ...
  <lifecycle>
    <phase-listener>
      package.LifeCycleListener
    </phase-listener>
  </lifecycle>
  ...
</faces-config>
```

Outline - Seção

Outline - Subseção

Escopos de ManagedBeans

O comportamento padrão de um *ManagedBean* é durar apenas uma requisição.

Em outras palavras, o escopo padrão de um *ManagedBean* é de *request*.

Com apenas uma anotação podemos alterar essa duração.

Escopos de ManagedBeans

Os quatro escopos principais do JSF são:

@RequestScoped é o escopo padrão. A cada requisição um novo objeto do *bean* será criado;

@ViewScoped escopo da página. Enquanto o usuário estiver na mesma página, o *bean* é mantido;

@SessionScoped escopo de sessão. Enquanto a sessão com o servidor não expirar, o mesmo *bean* atenderá o mesmo cliente;

@ApplicationScoped escopo de aplicação. Mantém uma única instância do *bean* para todos os usuários.

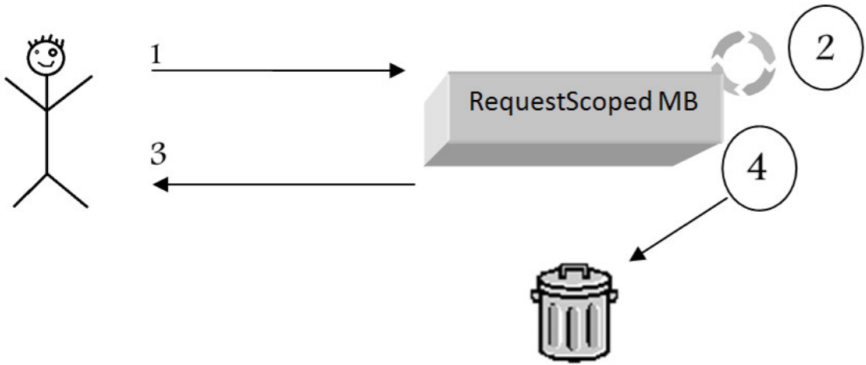
RequestScoped

A cada requisição, uma nova instância do *ManagedBean* será criada e usada, dessa maneira, não há o compartilhamento das informações do *ManagedBean* entre as requisições.

RequestScoped

1. O usuário iniciará uma requisição;
2. O *ManagedBean* processará as informações necessárias;
3. As informações do *ManagedBean* ficam disponíveis para o processamento da tela;
4. Caso algum valor tenha sido armazenado no *ManagedBean*, essas informações serão descartadas;

RequestScoped



SessionScoped

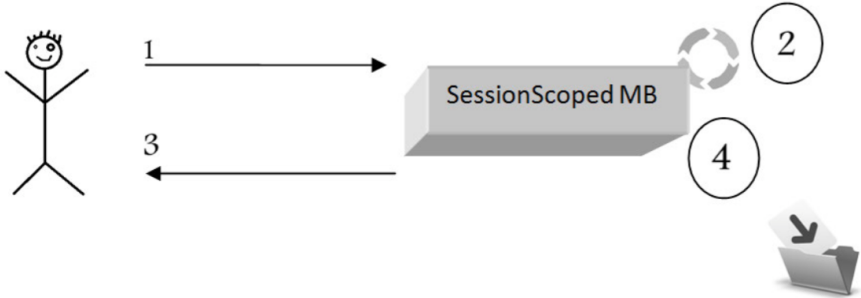
Os *ManagedBeans* `@SessionScoped` tem o mesmo comportamento da sessão web (`HttpSession`). Todo atributo de um *ManagedBean* `SessionScoped` terá seu valor mantidos até o fim da sessão do usuário.

Todo valor que for alterado em um *ManagedBean* `SessionScoped` será mantido, e todos os valores desse *bean* serão mantidos na memória.

SessionScoped

1. O usuário iniciará uma requisição;
2. O *ManagedBean* processará as informações necessárias;
3. As informações do *ManagedBean* ficam disponíveis para o processamento da tela;
4. Toda informação que foi atribuída ao *ManagedBean* será mantida enquanto durar a sessão do usuário no servidor.

SessionScoped



ViewScoped

O *ManagedBean* `ViewScoped` é um tipo de escopo que se encontra entre o `SessionScoped` e o `RequestScoped`.

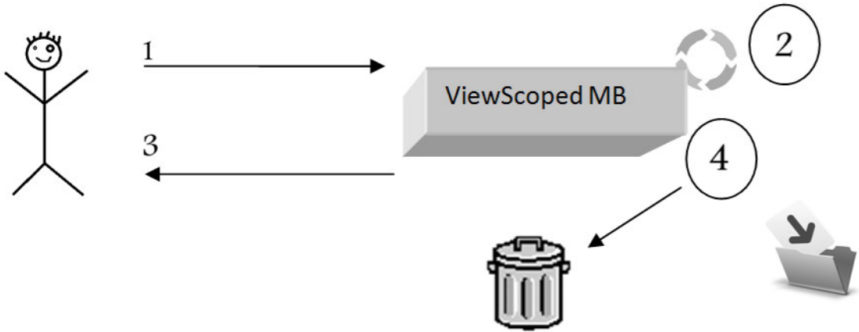
Ele tem a característica de existir na memória enquanto o usuário permanecer na página exibida.

Se o usuário abrir uma tela, o *ManagedBean* com os dados permanecerá vivo e pronto para exibir as informações a qualquer chamada realizada. Uma vez que o usuário mude de página, o *ManagedBean* será descartado pelo servidor.

ViewScoped

1. O usuário iniciará uma requisição;
2. O *ManagedBean* processará as informações necessárias;
3. As informações do *ManagedBean* ficam disponíveis para o processamento da tela;
4. O *ManagedBean* manterá os dados caso o usuário permaneça na página ou navegará para outra página e os dados em memória serão descartados.

ViewScoped



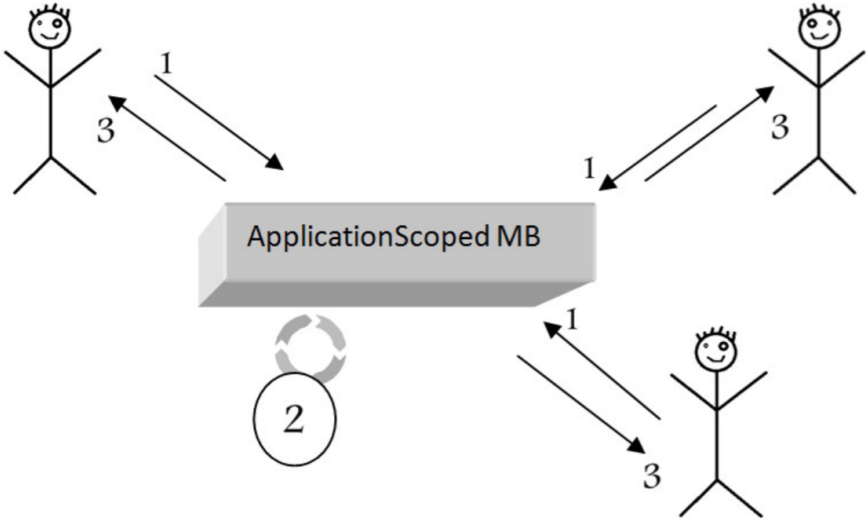
ApplicationScoped

O `ApplicationScoped` *ManagedBean* tem o comportamento semelhante ao do padrão de projeto *Singleton*, mantendo uma única instância de determinado *bean* na memória.

ApplicationScoped

1. O usuário iniciará uma requisição;
2. O *ManagedBean* processará as informações necessárias;
3. As informações do *ManagedBean* ficam disponíveis para o processamento da tela;
4. O mesmo *ManagedBean* responderá a outros *requests* de usuários, dessa forma, não haverá distinção de qual usuário poderá ou não ter acesso ao dados do *ManagedBean*.

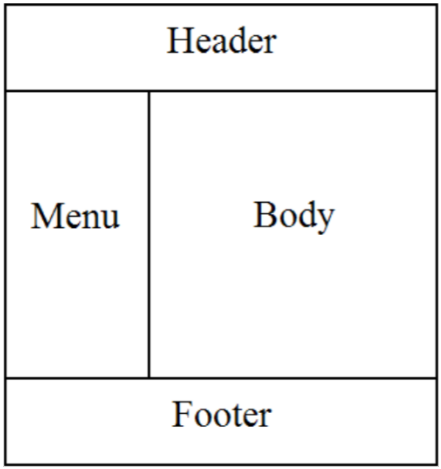
ApplicationScoped



Outline - Seção

Outline - Subseção

Templates



"A utilização de *templates* facilita a padronização da aplicação, já que conseguimos deixar fixo tudo aquilo que raramente muda."

Templates

```
<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">
<h:head>
    <title><ui:insert name="pageTitle">Template</ui:insert></title>
</h:head>
<h:body>
    <div id="topo">
        <ui:insert name="topo"><ui:include src="topo.xhtml"/></ui:insert>
    </div>
    <div id="menu"><ui:include src="menu.xhtml"/></div>
    <div id="body"><ui:insert name="body"/></div>
    <div id="rodape"><ui:include src="rodape.xhtml"/></div>
</h:body>
</html>
```

ui:include

Incluindo conteúdo de uma página na outra

A *tag* `ui:include` é usada quando queremos incluir o conteúdo de uma página dentro de outra.

ui:include

Exemplo de uso

No exemplo, o conteúdo do arquivo `menu.xhtml` será exibido dentro de `<div id="menu"></div>`.

```
<div id="menu">
  <ui:include src="menu.xhtml"/>
</div>
```

ui:insert

Abrindo espaço no template

Usado no arquivo de *template* onde será inserido um conteúdo proveniente da página cliente, ou seja, da página que usa o *template*. Temos apenas a propriedade `name` para podermos referenciar essa "área editável" dentro da página cliente.

`ui:define` pode substituir o conteúdo de `ui:insert`.

ui:insert

Abrindo espaço no template

- Define uma área que será substituída;
- Pode estar vazio ou conter um valor padrão (`ui:include`);

```
<div id="topo">
    <ui:insert name="topo">
      <ui:include src="topo.xhtml"/>
    </ui:insert>
</div>
<div id="body"><ui:insert name="body"/></div>
```

ui:composition

Carregando o template

- `ui:composition` carrega um *template* usando o atributo `template`;
- `ui:define` define o conteúdo que será inserido no *template*;

ui:composition

Carregando o template

```
<h:body>
  <ui:composition template="template.xhtml">
    <ui:define name="pageTitle">Cadastro</ui:define>
    <ui:define name="body">
      <h:form>
        Nome:
        <h:inputText id="nome" value="" />
        <h:commandButton value="Salvar" action="#{mb.action()}" />
      </h:form>
    </ui:define>
  </ui:composition>
</h:body>
```

ui:composition

Uso com o ui:insert

Quando utilizado em conjunto do `ui:include`, define o código a ser inserido em uma página.

Neste caso o atributo `template` é desnecessário.

```
<h:body>
  <ui:composition>
    <h:form>
    ...
    </h:form>
  </ui:composition>
</h:body>
```

Leitura complementar I


Coelho, Hébert.
JSF Eficaz: As melhores práticas para o desenvolvedor web Java.
Casa do Código


Cordeiro, Gilliard.
Aplicações Java para a web com JSF e JPA.
Casa do Código


JSF Mini Livro – Dicas, conceitos e boas práticas
http://uaihebert.com/jsf-mini-livro-dicas-conceitos-e-boas-praticas/


JavaServer Faces Technology - Oracle
http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html

Leitura complementar II

- JavaServer Faces (JSF) - Tutorials Point
  http://www.tutorialspoint.com/jsf/
- JSF Core Tag Reference
  http://www.jsftoolbox.com/documentation/help/12-TagReference/core/index.jsf
- Managed Beans. Não complique, simplifique. - Rafael Ponte
  http://www.rponte.com.br/2009/08/27/managed-beans-nao-complique-simplifique/
- Introdução ao JSF Managed Bean
  http://www.devmedia.com.br/introducao-ao-jsf-managed-bean/29390

Leitura complementar III

Como Depurar o Ciclo de Vida do JSF - DevMedia
http://www.devmedia.com.br/como-depurar-o-ciclo-de-vida-do-jsf-java-server-faces/29038

Introdução ao JSF e Primefaces - Caelum
https://www.caelum.com.br/apostila-java-testes-jsf-web-services-design-patterns/introducao-ao-jsf-e-primefaces/

Java Server Faces
http://www.dsc.ufcg.edu.br/~jacques/cursos/daca/html/jsf/jsf.htm