JSF

Ciclo de vida das Requisições:

Fase1. Le o arquivo XHTML e cria a arvore de componentes. Se o usuário já esteja usando a aplicação da arvore, ela não será criada e sim recuperada.

Fase2. O JSF irá buscar valores informados pelo usuário e coloca-los nos seus respectivos componentes.

Fase3. O JSF irá converter o formato das entradas dos componentes para o formato da variável, em seguida, o JSF irá validar os valores quando temos os métodos validadores declarados.

Fase4. O JSF irá atualizar o modelo com os valores inseridos.

Fase5. Nessa fase acontece a lógica da aplicação. Aqui dentro não precisamos e não devemos ficar buscando objetos baseados em algum id, uma vez que isso é responsabilidade dos conversores.

Fase6. Nessa fase acontece a geração do HTML a partir da árvore de componentes do JSF.

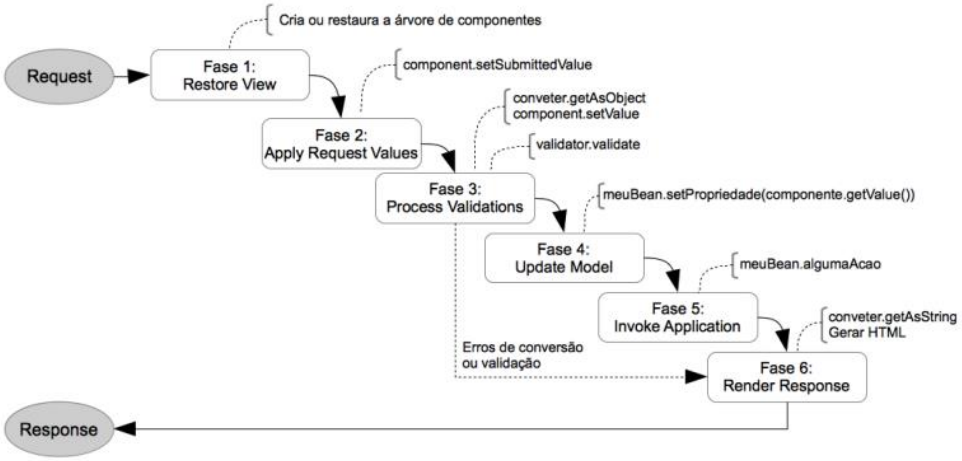


Figura 7.2: Ciclo de vida do JSF

Componentes JSF

Todos os componentes do JSF utilizam dois modificadores que permitem modificações de CSS são eles o “style” onde podemos dar atributos CSS a ele; E o “styleClass” que nos permite dar um nome a classe CSS, que poderá ser modificada depois.

<h:form> : Usado como caixa de componentes, geralmente usada sem atributos porém podemos usar atributos como o “id” para referencia-lo depois, porém vale lembrar que se declararmos um id para o form os componentes dentro dele também receberão esse id, para que isso não ocorra modificamos a propriedade “prepenId=false”.

Inputs:

No JSF quase todos os componentes são inputs, mesmo que muitos declaramos de outras formas quase todos por essência são inputs, por isso, a propriedade mais importante é o value, onde ligamos uma **Expression Language** que nos possibilitara fazer a ligação entre a linguagem web e uma propriedade de um objeto ou classe.

<h:inputText> : Possibilita ao usuário inserir uma string.

<h:inputTextarea> : Igual o “inputText” porém possui a propriedade “rows” que possibilita especificar a quantidade de linhas, e a propriedade “cows” que especifica a quantidade de colunas.

<h:inputSecret> : Equivalente ao <input type= “password”> do HTML ou seja nos possibilita inserir senhas sem mostrar elas.

<h:inputHidden> : Se comporta igual o <h:inputText> porém fica oculto para o usuário.

SelectOne:

<h:selectOneMenu> : Input com opções pré-definidas, os itens podem ser estáticos com **<f:selectItem>** e dinâmicos, com o **<f:selectItems>**.

Exemplo:

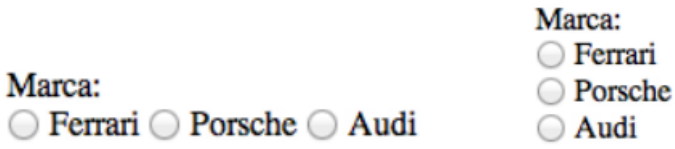
<h:selectOneMenu value="#{modeloBean.modelo.marca}">

<f:selectItem itemLabel="-- Selecione --" noSelectionOption="true"/>

<f:selectItems value="#{marcaBean.marcas}" var="marca"

itemValue="#{marca}" itemLabel="#{marca.nome}"/>

<h:selectOneRadio> : Igual **<h:selectOneMenu>** porém a visualização é diferente, por padrão ela fica em **lineDirection**, porém, podemos acrescentar **pageDirection**, nesse caso elas serão apresentadas de cima para baixo.



<h:selectOneListbox> : Igual **<h:selectOneMenu>** porém as opções ficam visíveis, podemos limitar o tamanho da lista usamos a propriedade **size**, se estiver mais elementos que o size ira aparecer uma barra de rolagem.

SelectMany:

A dinâmica é a mesma dos selectOne porém o **value** guarda uma lista em vez de apenas um elemento.

<h:selectManyListbox> : Lista selectMany igual a selectOneListbox com a propriedade size.

<h:selectManyMenu> : Igual selectManyListbox porém com tamanho único, sem a propriedade size.

<h:selectManyCheckbox> : Versão many do selectOneRadio, possui a propriedade **layout** com **lineDirection** e **pageDirection**.

<h:selectBooleanCheckbox> : Usado para ligação com propriedades booleanas a objetos, pode ser usado também para ativar ou não a exibição de outros componentes.

InputFile:

<h:inputFile> : Apenas a partir da versão 2.2 do JSF, utilizamos para enviar um arquivo para a API Servelet 3.0 dentro de uma classe. **Exemplo:**

```
<h:form enctype="multipart/form-data">
Imagem:  <h:inputFile value="#{automovelBean.uploadedFile}"/>
<h:commandButton value="Salvar" action="#{automovelBean.salvar}"/>
</h:form>
public void class AutomovelBean {

 private javax.servlet.http.Part uploadedFile;
 private Automovel automovel;
 // getters e setters

 public void salvar(){
try {
 InputStream is = uploadedFile.getInputStream();
 byte[] bytes = IOUtils.toByteArray(is);
automovel.setImagem(bytes);
 em.persist(automovel);
} catch (IOException e) {
// tratar exceção
}
}
}
```

Tabelas:

<h:painelGrid> : Renderiza uma tabela onde o numero de colunas é definido pela propriedade **columns,** assim sendo quando criamos por exemplo 2 colunas a cada 2 elementos seria criado uma outra linha.

<h:painelGroup> : É apenas um container para outros componentes.

<h:dataTable> : É usado para mostrar dados tabulares, usamos a propriedade **value** para ligar a lista que queremos tabular e a propriedade **var** para dar um nome ao objeto.

Dentro do dataTable temos a tag **<h:column>** para definirmos colunas, dentro do **column** podemos definir a propriedade **<f:facet>** definindo um name como **header** e **footer,** para cabeçalho e rodapé.

Temos também as propriedades **headerClass** e **footerClass** que usamos para criar um nome de referencia no css para as propriedades do cabeçalho e do rodapé. Temos também **rowClasses** que permite o zebramento das linhas no css e **columnClasses** que faz o mesmo para as colunas.

Exemplo:

```
<h:dataTable value="#{automovelBean.automoveis}" var="auto" rowClasses="table-linha-par,table-linha-impar" border="1">

<h:column>

<f:facet name="header">Marca</f:facet>

#{auto.modelo.marca}

</h:column>

…

</h:dataTable>
```

Outputs:

<h:outputText> : É um componente de texto, não necessariamente precisamos usar outputText podemos simplesmente escrever o texto, porém se for feito assim será contado apenas 1 componente e as vezes o numero de componentes interfere na exibição.

<h:outputLabel> : É a semântica correta dos labels, renderiza a tag label do HTML, o atributo **for** referencia o **id** do **input** ao qual ele se refere, Exemplo:

```
<h:outputLabel value="Ano de Fabricação:" for="anoFabricacao"/>
<h:inputText value="#{automovel.anoFabricacao}" id="anoFabricacao"/>
```

<h:outputFormat> : É utilizado quando precisamos de textos parametrizados, Exemplo:

```
<h:outputFormat value="#{preferenciasBean.bemVindo}" >
<f:param value="#{session.usuarioLogado.nome}" />
<f:param value="#{session.usuarioLogado.tratamento}" />
<f:param value="#{session.usuarioLogado.cargo}" />
</h:outputFormat>
```

<h:graphicImage> : Utilizamos para exibir imagens em nossa aplicação através do JSF Exemplos:

```
<h:graphicImage url="www.servidor.com/imagem.jpg"/>
<h:graphicImage library="images" name="imagem.jpg"/>
```

Exportando css e javaScript:

javaScript:

<h:outputScript>

Css:

<h:outputStylesheet>

Exemplos:

```
<h:outputStylesheet library="css" name="facesmotors.css"/>
<h:outputScript library="scripts" name="facesmotors-commons.js" target="head"/>
```

<ui:repeat> : É usada também para iterar elementos, porém, não cria uma tabela, simplesmente percorre a lista, deixando-nos livres para escolhermos a saída.

Possui os mesmos **value** e **var**, possui a propriedade **varStatus** que possibilita definir uma variável relacionada ao status da iteração, podemos declarar a variável **status** onde podemos fazer as seguintes checagens:

#{status.first}: é booleano e indica se é o primeiro elemento da lista;

#{status.last}: é booleano e indica se é o último elemento da lista;

#{status.index}: do tipo inteiro que representa o índice da iteração. Se fosse um for tradicional, seria o int i;

#{status.even}: é booleano e indica se o index é par;

#{status.odd}: é booleano e indica se o index é impar.

```
<ui:repeat value="#{automovelBean.automoveis}" var="auto" varStatus="status">
<motors:automovel automovel="#{auto}"/>
</ui:repeat>
```

Command Buttons:

<h:commandButton> : Cria um botão de comando.

<h:commandLink> : Igual ao **commandButton** porem não cria um botão e sim um link.

Ambos os componentes **command** possuem a propriedade **action** que é uma **String** que pode ser uma string presente nos componentes ou uma string do **managedBean** seja uma variável ou retorno de método ou objeto.

Exemplo:

```
<h:dataTable value="#{automovelBean.automoveis}" var="automovel">
<h:commandLink value="Editar" action="editar">
<f:setPropertyActionListener value="#{automovel}"
        target="#{automovelBean.automovel}"/>
        </h:commandLink>
        </h:dataTable>
        <h:commandButton value="Salvar"
                action="#{automovelBean.salvar(automovel)}"/>
```

Action e ActionListener

Action: É usado para executar a lógica da aplicação.

<f:ActionListener> : É usado para observamos exemplos de tela, ou seja podemos visualizar ações de tela no código java porém sem interação com a tela; Podemos receber parâmetros da execução do componente através da biblioteca **javax.faces.event.ActionEvent.**

Exemplo:

```
<h:commandButton id="botaoSalvar" value="Salvar"
        actionListener="#{automovelBean.listener}"
                action="#{automovelBean.salvar(automovel)}"/>
```

```
public void listener(ActionEvent event){
 UIComponent source = event.getComponent();
 System.out.println("Ação executada no componente " + source.getId()); }
```

Conversores:

<f:convertDateTime> : Usado para convertermos os dados de hora adquiridos pelo código java no JSF.

Exemplo:

<h:inputText value="#{managedBean.objeto.data}">

<f:convertDateTime pattern="dd/MM/yyyy"/>

</h:inputText>

<h:outputText value="#{managedBean.objeto.data}">

<f:convertDateTime dateStyle="full" type="both"/>

</h:outputText>

dateStyle Pode assumir os seguintes valores com os respectivos resultados:

• default (valor padrão): 21/12/2012

• short: 21/12/12 • medium: 21/12/2012

• long: 21 de Dezembro de 2012

• full: Sexta-feira, 21 de Dezembro de 2012

type

A propriedade type permite configurar se queremos trabalhar somente com data, hora, ou ambos:

• date (valor padrão): 21/12/2012

• time: 23:59:59

• both: 21/12/2012 23:59:59

locale

Podemos informar a localidade. Deve ser uma instância de java.util.Locale ou então uma String, como “ en” ou “ pt”. Caso não informemos o locale, o JSF usará o devolvido por FacesContext.getViewRoot().getLocale(), que, se não for alterado de alguma forma, será a localidade padrão do sistema (browser) do usuário.

timeZone

Uma característica do JSF que não agrada muito com relação ao f:convertDateTime é que ele usa o time zone GMT por padrão, e não o horário do sistema. Para mudar isso teríamos que informar timeZone="GMT-3", no caso do horário oficial do Brasil, mas aí teríamos algo fixo. Para resolver esse problema podemos colocar o seguinte parâmetro no web.xml.

<context-param>

<param-name>

javax.faces.DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE

</param-name>

<param-value>true

</context-param>

<f:convertNumber> : Esse conversor possui propriedades que auxiliam na formatação, como **maxFractionDigits** e **minFractionDigits** que definem o mínimo e o máximo de casas decimais a serem usadas, **currencyCode** e **currencySymbol** para exibir R$ ou US$, **groupingUsed="false"** para desabilitar a separação de milhar.

Podemos usar **type** para informar se é **number**(valor padrão) ou **currency**(valor dinheiro) ou **percent**.

Podemos também usar **locale** para pegarmos a localidade do usuário e formatar à partir disso.

Exemplo:

```
<h:outputText value="#{automovel.preco}">

<f:convertNumber type="currency"/>

</h:outputText>

<h:outputText value="#{automovel.kilometragem}">

<f:convertNumber type="number"/>

</h:outputText>
```

Conversores customizados:

Precisamos usar de conversores customizados quando por exemplo precisamos enviar um objeto do JSF para um código java, por padrão ele ira como String, o problema é que o java não possui um **fromString** para transformar string em um objeto.

Nesse caso usamos a biblioteca **javax.faces.convert.Converter** ela possui dos métodos que são equivalentes ao **toString** e um método equivalente ao **fromString**. A interface Converter possui o método **getAsString** que recebe um objeto e devolve uma String, e o método **getAsObject** que recebe a String e devolve o objeto.

Exemplo:

```
<h:inputText value="#{automovelBean.automovel}" />

Import javax.faces.convert.Converter;

@FacesConverter(forClass=Automovel.class)

public class AutomovelConverter implements Converter {


public String getAsString(FacesContext context, UIComponent component, Object object) {

Automovel automovel = (Automovel) object;

 if(automovel == null || automovel.getId() == null) return null;

 return String.valueOf(automovel.getId());

 }

public Object getAsObject (FacesContext context, UIComponent component, String string) { if(string == null || string.isEmpty()) return null; Integer id = Integer.valueOf(string);

Automovel automovel = entityManager.find(Automovel.class, id);

 return automovel;

}

}
```

Validadores Nativos:

Os validadores possuem propriedades em comum são elas:

.disabled :É usado para desativar os validadores, podendo ser colocado uma propriedade a ser cumprida ou usando um Expression Language.

.for : Usada para trabalhar com composição de componentes.

.binding :A propriedade **binding** pode ser usada em todos os componentes do JSF para ligar-los com uma propriedade do Managed Bean.

<f:validatorLongRange> e <f:validateDoubleRange> : Usados para validar números inteiros e reais, Temos a propriedade **minimum** que indica o valor mínimo e a propriedade **maximum** que indica o valor máximo, ambos aceitam Expression Languages.

<f:validateLenght> : Usado para validar tamanho de Strings, aceita as propriedades **minimum** e **maximum**.

<f:validateRequired> : Na pratica gera o mesmo resultado que a propriedade **required** ou seja torna obrigatório uma condição podendo ou não ser um Expression Language.

<f:validateRegex> : Cria uma validação baseada em expressões regulares, possui apenas a propriedade **pattern**, que nos permite colocar qualquer expressão, Exemplo:

```
<h:inputText value="#{user.login}">

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

</h:inputText>
```

<f:validateBean> : Usada em Bean Validation, é usada para especificar grupos que desejamos validar, pode ser desabilitada quando acrescentado a propriedade **disable**, podemos usar o validateBean tanto dentro de um componente como acrescentando um grupo de componentes dentro dele.

- **validationGroups**: informamos o grupo de propriedades que desejamos validar. Caso queiramos informar mais de um grupo, informamos todos eles separados por vírgula.

Criando validadores:

Utilizamos o **<f:validator>** com a alguma propriedade como o **disable** com uma Expression Language ligada a uma variável boolean do Managed Bean ou um método com alguma refêrencia, temos que criar a propriedade **validatorId** com o nome de referência dessa validação.

Vale lembrar que os validadores atuam na terceira fase, porém se usamos um **<f:validator>** ligado a uma variável do ManagedBean, essa variável só será atualizada na quarta fase da requisição, dessa forma essa validação só será efetiva na segunda requisição feita.

Podemos usar a tag **<f:AJAX>** sem nenhuma propriedade, dessa forma fazemos um pequeno request no componente atualizando seus valores.

Exemplo managedBean com método validador de exceção:

```
@ManagedBean
 public class PedidoBean {
 private Pedido pedido;
 ...
 public void validaProdutoPedidoPopular(FacesContext context, UIComponent component, Object value) throws ValidatorException {
 //objeto vem convertido
Produto produto = (Produto) value;
 if(pedido.isPedidoPopular() && produto.getValor() > 200.0){
 FacesMessage message = new FacesMessage("Pedido popular: máximo R$ 200"); message.setSeverity(FacesMessage.SEVERITY_ERROR);
 throw new ValidatorException(message);
}
 }
}
```

Exemplo aplicação do método no XHTML:

```
<h:inputText id="produto"
 value="#{pedidoBean.pedido.produto}"
 converter="produtoConverter"
 validator="#{pedidoBean.validaProdutoPedidoPopular}"/>
```

<h:button> e <h:link> : Tem o mesmo comportamento dos **command** porém não submetem os formulários onde estão, por isso, nem precisam estar em um formulário, são do tipo **get** e não **post**.

A propriedade **outcome** é muito importante e baseado nela o JSF executará a regra de navegação para descobrir qual link deverá montar.

Regras de Navegação

A navegação é feita através das **outcome** antes do suporte as navegações implícitas, utilizava-se o faces-config.xml, onde configurávamos para qual pagina o resultado iria levar o usuário.

A estrutura era a seguinte utilizava-se uma **<navigation-rule>** onde essa regra iria se aplicar a uma pagina especificada pela tag **<from-view-id>**, podemos usar ***** para especificar um local especifico da pagina ou deixar apenas o asterisco e assim seria uma regra global.

Na regra de navegação, após informarmos a pagina de origem criamos a tag **<navigation-case>** onde iremos criar as regras de outcome, podemos usar **<from-action>** quando o outcome vem de uma ação seja do **bean** seja do próprio xhtml, lembrando que a ação deverá ter um retorno em String para que se tenha o outcome, devemos então criar a tag **<from-outcome>** com um nome especifico que se encontrado será passado para o outro link através da tag **<to-view-id>**, por fim devemos colocar a tag **<redirect>** que irá informar a ação de redirecionamento.

Exemplo faces-config.xml:

```
<faces-config ...>
<navigation-rule>
<from-view-id>/marca/editar.xhtml</from-view-id>
<navigation-case>
<from-action>#{marcaBean.excluir}</from-action>
<from-outcome>sucesso</from-outcome>
<to-view-id>/marca/listar.xhtml</to-view-id>
<redirect/>
</navigation-case>
</navigation-rule>
</faces-config>
```

Navegação implicita:

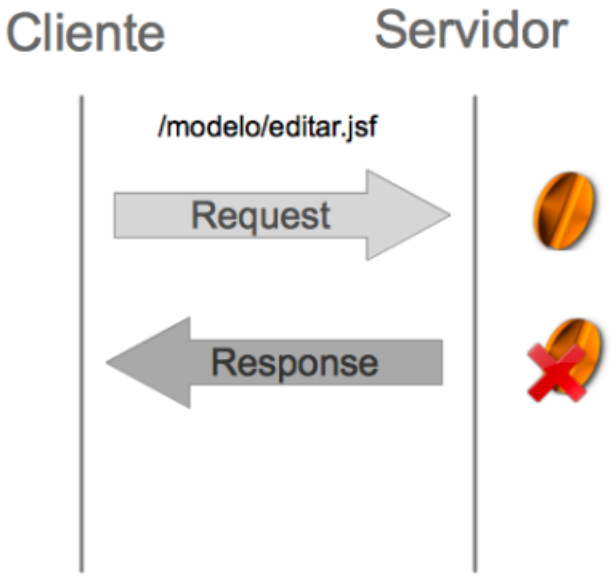
A partir da versão 2.0 do JSF foi adicionado a navegação implícita, através do **outcome** o JSF busca um correspondente no **faces-config.xml**, não possuindo um correspondente o JSF busca uma pagina na mesma pasta com o mesmo nome.

Na navegação implícita como no **faces-config.xml** o JSF faz apenas o **forward**, mudamos isso no faces-config.xml através da tag **<redirect />**, na navegação implícita temos que indicar a necessidade de **redirect** na String com a propriedade **?faces-redirect=true** a partir do nome.

Escopos

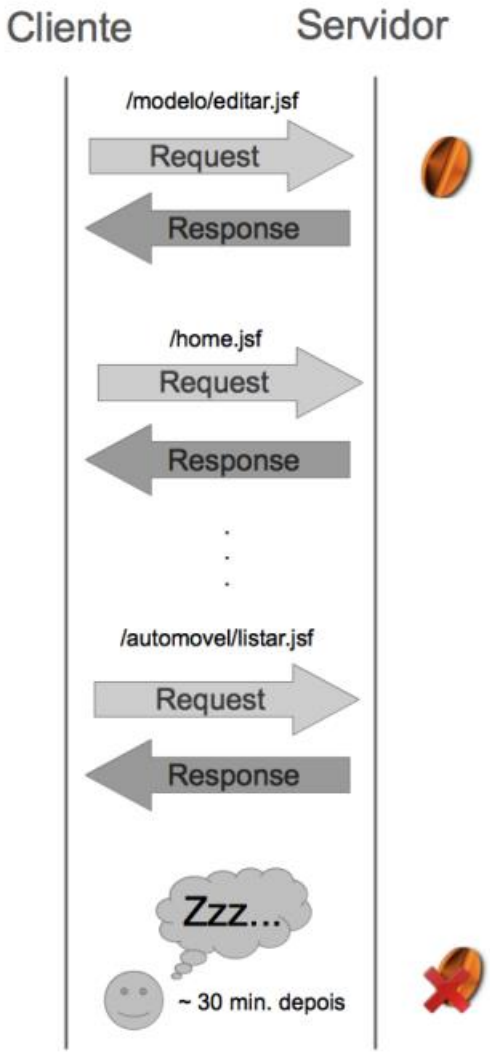
Temos 4 tipos de escopo, **request, session** e **application**, a partir da versão 2 do JSF temos também o **view**.

Request Scope:



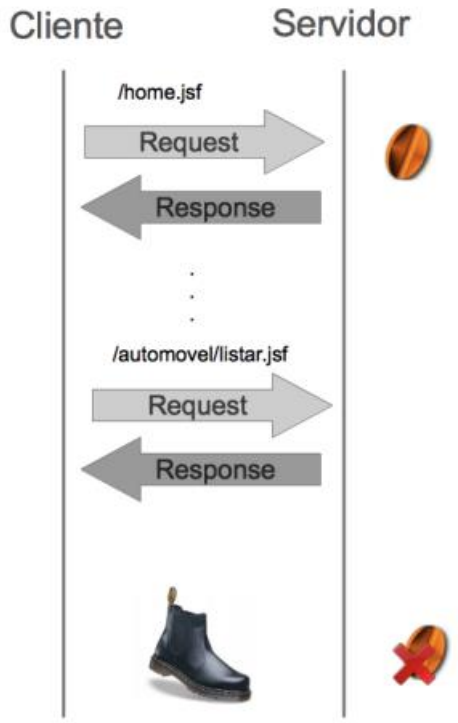
Os objetos criados através do escopo do **request** só sobrevivem por uma passagem pelas fases.

Session Scope:



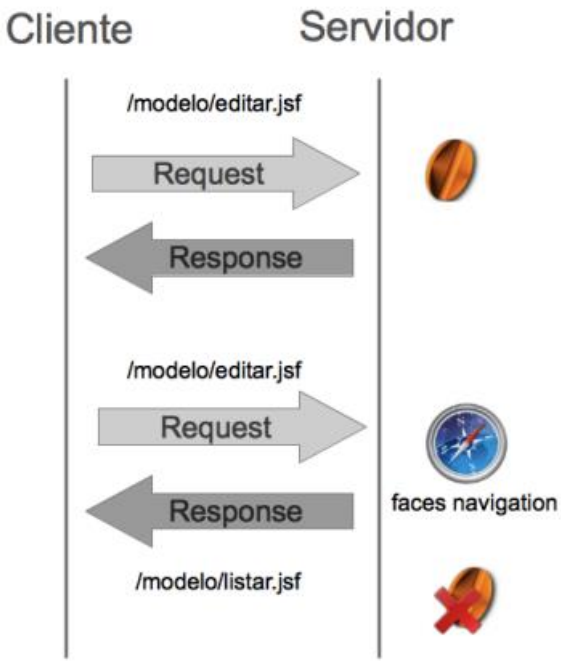
Tudo ficara armazenado enquanto a sessão do usuário estiver ativa.

Application Scope:



Tudo que é armazenado no escopo de aplicação permanece enquanto a aplicação estiver executando, e é compartilhada entre todos os usuários, dessa forma, esse tipo de escopo oferece um excelente controle de todos os usuários e suas informações.

View Scope:



Nesse escopo o foco é o view, sendo assim os dados são mantidos independente de quantas requisições sejam feitas, porém ao trocar de view a memória é liberada.

Vale lembrar que a memoria so é liberada se a troca de view for feita através de um **POST**, dessa forma, se abrirmos outra view com outro link através do método get, podemos manter os dados da outra página.