01

Navegação e o ciclo de vida dos componentes JSF

Transcrição

Caso queira começar o treinamento a partir desse vídeo, pode baixar o projeto aqui (http://s3.amazonaws.com/caelum-online-public/JSF/livraria-capitulo7.zip). Só baixe este arquivo se **não tiver feito os exercícios dos capítulos anteriores**.

Navegação entre as páginas

Continuaremos a melhorar passo a passo nossa aplicação aprendendo ainda mais sobre JSF. Ao acessar a página `livro.xhtml` podemos ver o formulário para cadastrar livros. Uma situação bastante comum no dia a dia será cadastrar um livro quando ainda não há autor relacionado. Nesse caso é preciso chamar o formulário do autor manualmente, digitando a página no navegador. Com certeza isso não vai agradar nenhum usuário. Vamos melhorar então o formulário e criar um link abaixo do combobox .

O componente h:commandLink

No eclipse, vamos abrir a página `livro.xhtml` e procurar o `fieldset` do autor. Nele, abaixo do `h:commandButton` adicionaremos mais um comando, mas agora do tipo link, ou seja `h:commandLink` .

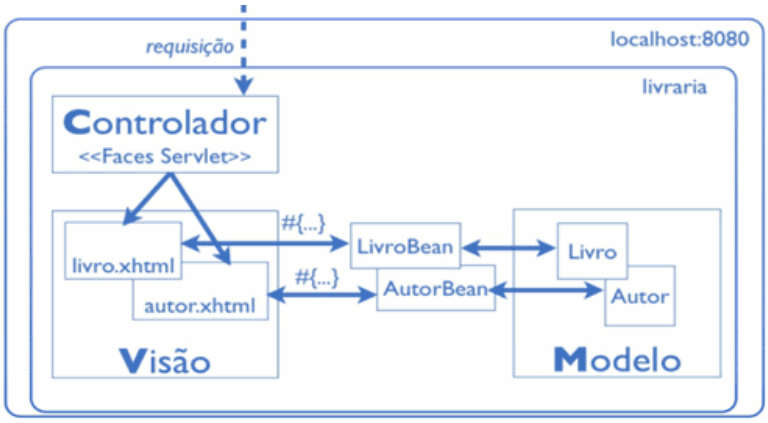
Esse componente também recebe no atributo value o nome do link, que neste caso é *Cadastrar novo autor*. No atributo `action` podemos associar uma expressão ou colocar diretamente o nome da página que queremos chamar. Não podemos usar a extensão da página, apenas o nome sem `xhtml` . Só falta adicionar um nova linha para melhorar o layout, basta usar a tag HTML `<br>` .

```
<h:commandLink value="Cadastrar novo autor" action="autor" />
```

Pronto, podemos testar no navegador. Ao atualizar aparece o link. Vamos apertar uma vez para verificar a funcionalidade, mas percebemos uma coisa estranha: no lugar de chamar a página `autor.xhtml` a validação foi executada! Repare nos erros de validação no início do formulário. Não foi bem isso que queríamos.

Introdução ao ciclo de vida do JSF

Para entender melhor o que o JSF faz por baixo dos panos, vamos lembrar do modelo de camadas. Aprendemos que o JSF segue um padrão arquitetural *Model-View-Controller*. Nele cada camada possui uma responsabilidade bem definida. A camada do controlador recebe todas as requisições e decide que tela ou árvore de componentes utilizar. A visão define toda interface e a camada do modelo fornece os dados. Os *ManagedBean*s representam a ligação entre modelo e visão.

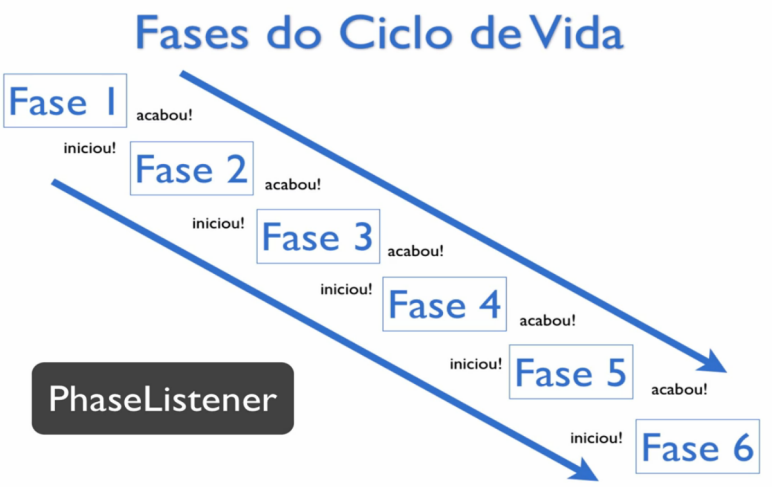


Além dessa separação já percebemos que há um fluxo bem definido dentro desse modelo. Por exemplo, vimos que o JSF converte e valida automaticamente os parâmetros da requisição. Esses dados, caso não haja nenhum problema com conversão e validação, serão passados para o modelo. Depois os *ManagedBean*s usam os valores para executar um método.

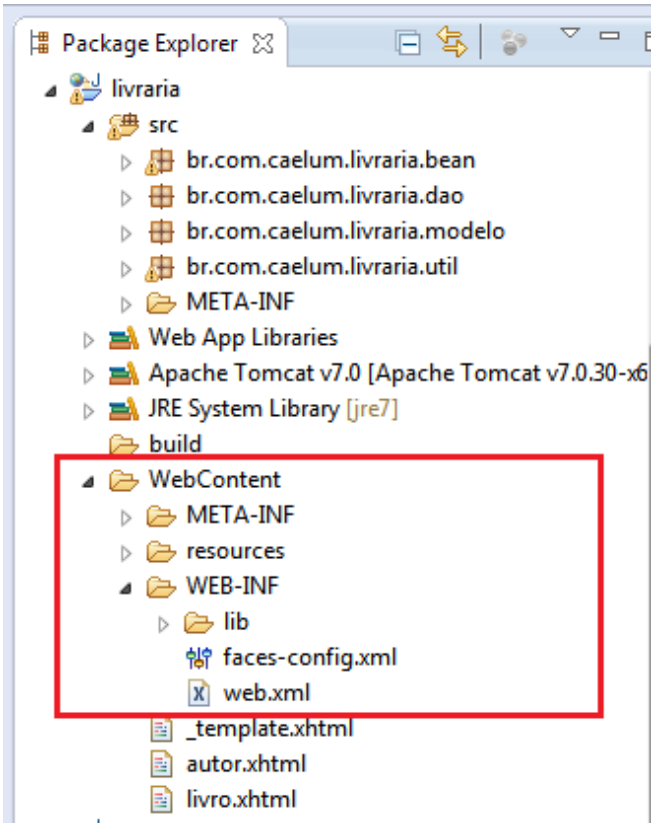
Esses passos ou fases foram bem definidos pela especificação e fazem parte do **ciclo de vida** do JSF. Para entender por que o link não funcionou é preciso entender essas fases. Em geral, um bom entendimento sobre o ciclo de vida facilita muito o trabalho com o JSF.

Definição do PhaseListener

A especificação define 6 fases. Para visualizar e acompanhar o ciclo de vida podemos plugar uma classe chamada automaticamente pelo JSF quando uma fase inicia ou acaba. Criaremos esse *PhaseListener* para entender o ciclo de vida.



No Eclipse vamos abrir o arquivo de configuração do JSF: `faces-config.xml` que está na pasta `WEB-INF`. Nele precisamos declarar uma tag `lifecycle` que por sua vez terá um elemento `phase-listener`. Dentro do `phase-listener` definiremos uma classe `br.com.caelum.livraria.util.LogPhaseListener`.



A classe não existe ainda, então vamos para a pasta `src`, digitando `ctrl + n` para criar uma nova classe `LogPhaseListener` dentro do package `br.com.caelum.livraria.util`. Após confirmação é gerado o esboço da classe. Ela deve implementar a interface `PhaseListener` que já vem com a JAR do JSF.

```
<lifecycle>
    <phase-listener>br.com.caelum.livraria.util.LogPhaseListener</phase-listener>
</lifecycle>
```

Ao implementar a `PhaseListener` somos obrigados a implementar três métodos. Com a ajuda do Eclipse podemos gerar esses métodos automaticamente, basta digitar `ctrl + 1` antes do `PhaseListener`. Os métodos são `afterPhase()`, `beforePhase()` e `getPhaseId()`.

`beforePhase(..)` e `afterPhase(..)` são chamados antes e depois de uma fase e o `getPhaseId()` define qual fase o listener atende. Para nosso objetivo acionaremos todas as fases, indicado pelo retorno `PhaseId.ANY_PHASE`. Para fins educativos usaremos apenas o método `beforePhase(..)` e imprimir o nome da fase do evento recebido como parâmetro.

Segue a implementação do `LogPhaseListener`:

```
package br.com.caelum.livraria.util;

import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

public class LogPhaseListener implements PhaseListener{

    private static final long serialVersionUID = 1L;

    @Override
```

```
    public void afterPhase(PhaseEvent arg0) {
    }

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

    @Override
    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }
}
```

Ajustando os logs da aplicação

Antes de testarmos o Listener, alteraremos o log de Hibernate. No arquivo `persistence.xml` desabilitaremos o log do SQL. Nosso foco agora são as fases do JSF, o SQL gerado pode atrapalhar nesse momento. Para tal basta alterar a propriedade show_sql para false. No `persistence.xml` :

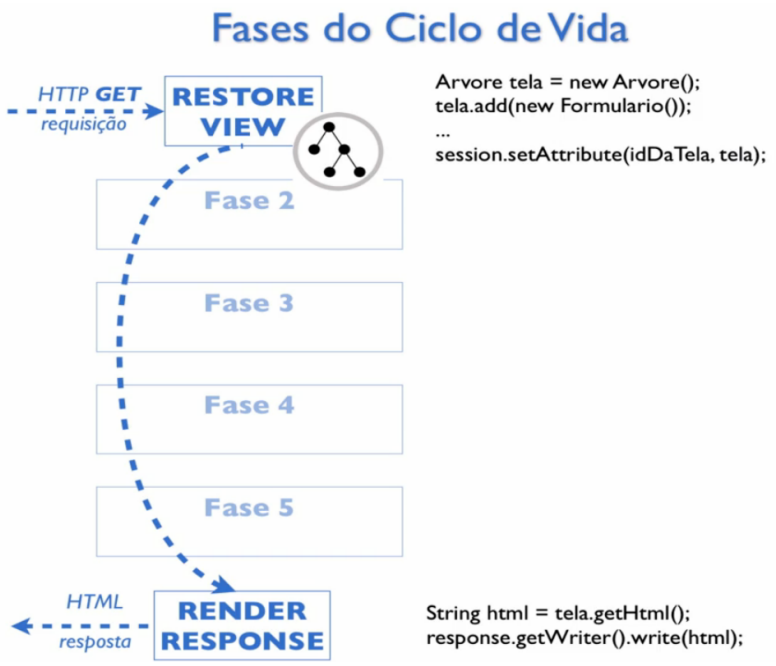
```
<property name="hibernate.show_sql" value="false" />
```

Pronto, podemos reiniciar o Tomcat e atualizar a página `livro.xhtml` no navegador. Ao voltar para Eclipse podemos ver que o console foi poluído pelo logs do Tomcat e Hibernate. Isso só aconteceu na inicialização da aplicação. Vamos limpar o console e repetir o processo. Novamente chamaremos a página pelo navegador.

Criação da view e renderização: RESTORE_VIEW e RENDER_RESPONSE

Agora o console do Eclipse só mostra a saída do nosso `PhaseListener` . Podemos ver que duas fases foram acionadas, a primeira chamada RESTORE_VIEW e outra, a sexta fase, chamada `RENDER_RESPONSE` . Repare que o `PhaseListener` automaticamente numerou as fases.

Ao receber uma requisição HTTP do tipo GET o controlador iniciou o ciclo de vida da tela. Isso significa que ele leu o arquivo xhtml e instanciou todos os componentes. Como isso foi disparado pela requisição inicial está claro que não há nada a fazer além de renderizar a resposta.



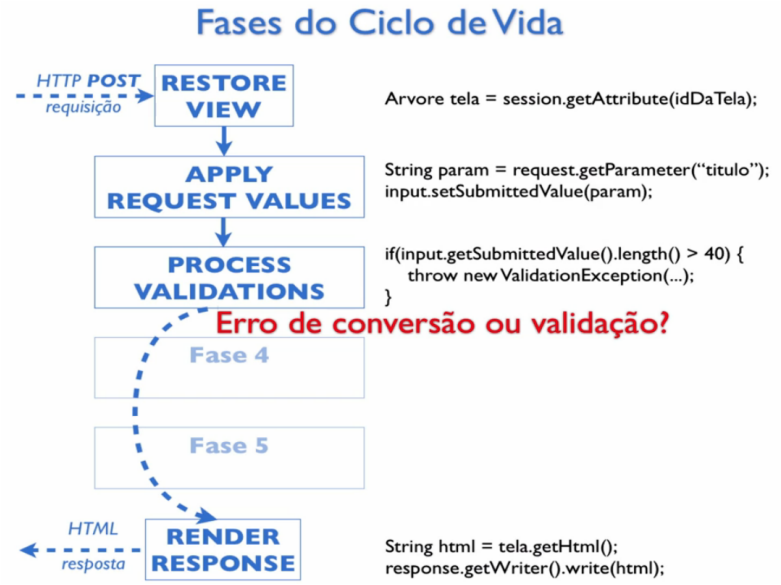
Analisando as fases: APPLY_REQUEST_VALUES e PROCESS_VALIDATION

No Eclipse novamente limparemos o console e mais uma vez no navegador usaremos o formulário para executar uma validação. Lembrando que o campo do título automaticamente envia uma requisição AJAX ao perder foco e consequentemente é executada a validação no lado do servidor.

Olhando o console no Eclipse podemos ver que dessa vez foram executadas 4 fases. Temos como segunda fase o `APPLY_REQUEST_VALUES`, e como terceira fase `PROCESS_VALIDATION`.

Vamos analisar também esse caso. Como sempre o controlador recebeu a requisição mas agora ela é do tipo POST. Isso significa que o controlador apenas recupera a árvore (por isso se chama `RESTORE_VIEW`). Após a recuperação da tela os componentes recebem o valor digitado pelo usuário que vem da requisição (dai vem o nome `APPLY_REQUEST_VALUE`). Nesse caso submetemos apenas o valor do título, mas em branco. Em outras palavras, o componente do título recebe um String vazio.

Na terceira fase acontece a conversão, se for preciso, e a validação. O título não precisa ser convertido pois o título é do tipo String, mas associamos uma validação com o componente. É justamente essa validação que falha. Em geral, se há um problema de conversão ou validação, o controlador pula automaticamente as fases quatro e cinco e pode renderizar o HTML com as mensagens de erro.



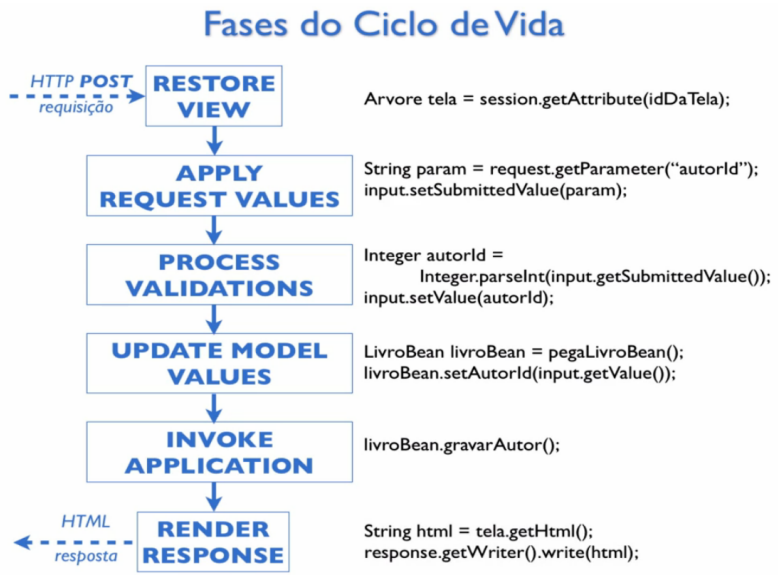
Todas as fases do ciclo

Agora testaremos o que acontece quando escolhemos um autor no combobox. Vamos apertar o botão e verificar o console de novo. Repare que agora foram executadas todas as seis fases. Na primeira fase (`RESTORE_VIEW`) foi recuperada a árvore de componentes da sessão, na segunda (`APPLY_REQUEST_VALUES`) os componentes receberam os parâmetros da requisição e na terceira (`PROCESS_VALIDATION`) todos os dados foram convertidos e validados.

Como nenhum erro ocorreu na terceira fase o controlador continua com a próxima fase, que se chama `UPDATE_VALUES`. Aqui o modelo será atualizado com os valores convertidos na fase anterior. Neste caso, `LivroBean` recebe apenas o ID do autor, já que só enviamos esse parâmetro pelo AJAX. Em geral, nesta fase, tudo que definimos com a *expression language* nos componentes de inputs é atualizado no modelo.

Agora que o modelo já foi atualizado, nosso `LivroBean` já pode executar o comando definido via expression language. Este comando é um método no próprio `LivroBean` e opera sobre os dados convertidos. Esta fase de chamada de métodos no ManagedBean é chamada de `INVOKE_APPLICATION`.

Por fim, o JSF devolve uma resposta para o usuário, o que é feito na última fase, `RENDER_RESPONSE`.



Resolvendo o problema de navegação e o atributo immediate

Abriremos a página e testaremos novamente o link. Ao voltarmos ao Eclipse, vemos no console as fases que foram processadas. Houve um salto da fase `PROCESS_VALIDATION` para `RENDER_RESPONSE`, sendo assim, nenhum dado do modelo foi atualizado e nenhum método foi invocado, isto porque as fases `UPDATE_MODEL` e `INVOKE_APPLICATION` foram puladas.

Vamos realizar outro tipo de navegação, mas desta vez através de um método em `LivroBean`. Vamos alterar o atributo `action` que agora apontará para o método `formAutor()`. Fazemos isso através de expression language `#{livroBean.formAutor}`. Com a classe aberta, adicionaremos o método, mas agora precisamos que seu retorno seja uma String, pois o retorno é a regra de navegação que o JSF seguirá. Aqui vamos devolver apenas o nome da página, sem extensão. Adicionaremos também uma saída no console que será impressa apenas se o JSF executar a fase `PROCESS_VALIDATION`:

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

    //codigo omitido

    public String formAutor() {
        System.out.println("Chamando o formulário do Autor");
        return "autor";
    }

    //codigo omitido
}
```

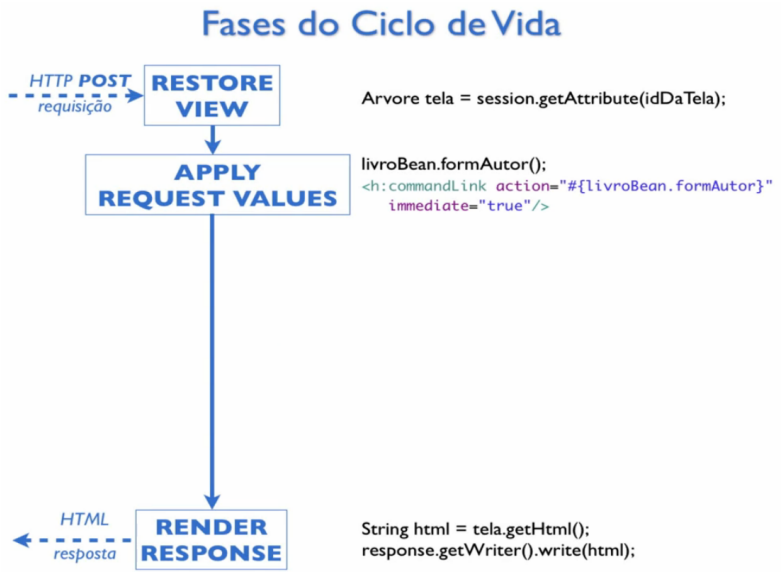
Vamos reiniciar o servidor, pois alteramos uma classe. Abrindo mais uma vez e testando o link, vemos que o fluxo não mudou. Olhando no console do Eclipse, percebemos que a fase `INVOKE_APPLICATION` não foi invocada, isto é, o JSF não executou o método `formAutor()`.

O problema é que o nosso link com a regra de navegação está disparando a validação do formulário, quando na verdade queremos apenas seguir para outra página. Uma maneira de resolver este problema é adicionar o atributo `immediate=true` em nosso link. Isso fará com que o método `formAutor()` seja executado durante a segunda fase

`APPLY_REQUEST_VALUES`, antes da validação. Assim a nossa regra de navegação será processada e automaticamente pularemos para última fase.

```
<h:commandLink value="Cadastrar novo autor" action="#{livroBean.formAutor}" immediate="true"/>
```

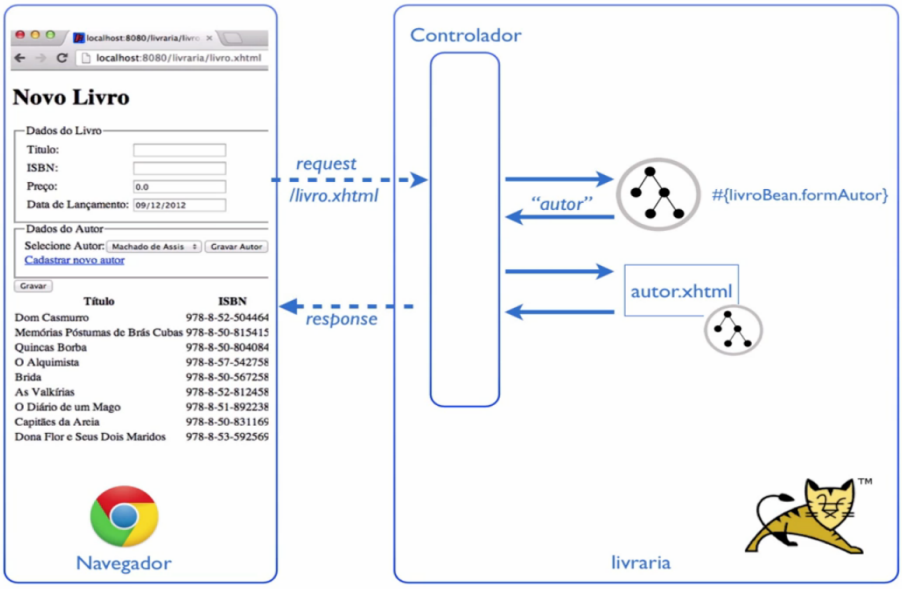
Finalmente a página do autor foi chamada. Podemos verificar também na saída do console. Foram executadas a primeira, segunda e sexta fase.



Redirecionamento pelo controlador no lado do servidor

Após ter executado o link podemos reparar que na barra de endereço do navegador é mostrado o endereço do livro e não o endereço do formulário que estamos vendo. Isto acontece porque a regra de navegação padrão do JSF é feita através de forward.

Vamos visualizar novamente os acontecimentos para entender melhor. Enviamos pelo link um request do tipo POST que caiu na árvore de componentes, mas a árvore do Livro. Aqui o comando executou o método `formAutor()`, que devolveu a `String autor`. O controlador recebeu esse retorno e chamou a página `autor.xhtml` para instanciar os componentes. Após isso devolveu o HTML dessa tela para o navegador. Repare que o controlador fez um redirecionamento, mas tudo no lado do servidor. Enviamos apenas uma requisição e por isso o endereço no navegador ficou com `livro.xhtml`. A consequência disto é que o navegador sempre ficará um URL atrás.



Redirecionamento no cliente

Uma maneira de resolver isso é através de redirecionamento pelo navegador, no lado do cliente. Aqui o controlador devolve a resposta após ter recebido o retorno do comando. Nessa resposta avisamos o navegador para enviar um segundo request com o novo endereço `/autor.xhtml`. Ou seja, enviamos duas requisições. Para tal, vamos indicar ao controlador que queremos redirecionar pelo navegador. Isso é feito no método `formAutor()` com o retorno `autor?faces-redirect=true`.

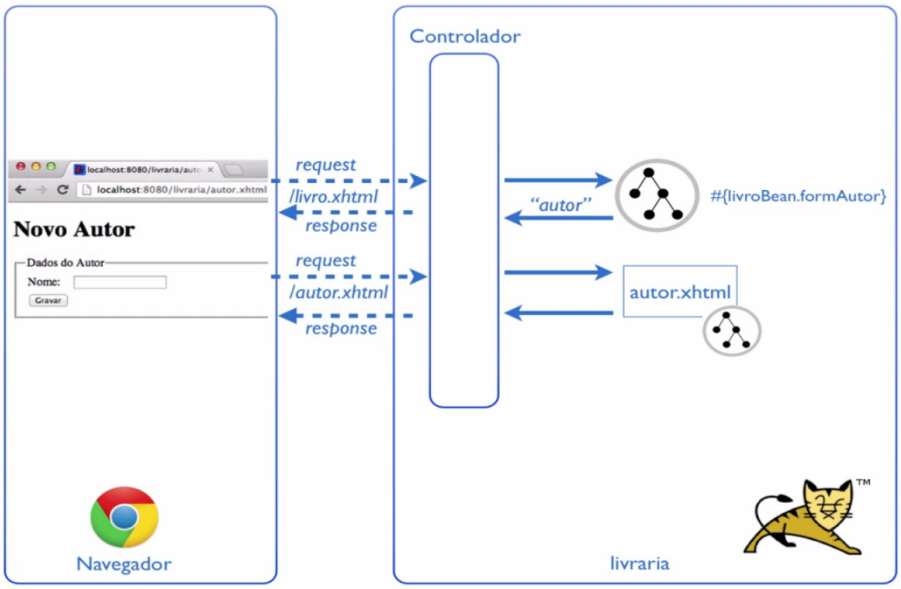
```
@ManagedBean
@ViewScoped
public class LivroBean implements Serializable {

    //codigo omitido

    public String formAutor() {
        System.out.println("Chamanda o formulario do Autor");
        return "autor?faces-redirect=true";
    }

    //codigo omitido
}
```

Reiniciando a aplicação e testando mais uma vez. A regra de navegação é executada e a barra de endereço do navegador corresponde à página que estamos visualizando.



Faremos a mesma coisa com o método `gravar()` de `AutorBean` . Após salvar um autor, redirecionaremos para a página de livros. Para isso retornamos uma String indicando o redirecionamento para a página de livros. Salvando as alterações, reiniciando e visualizando. Após testar, vemos que o redirecionamento é feito com sucesso.