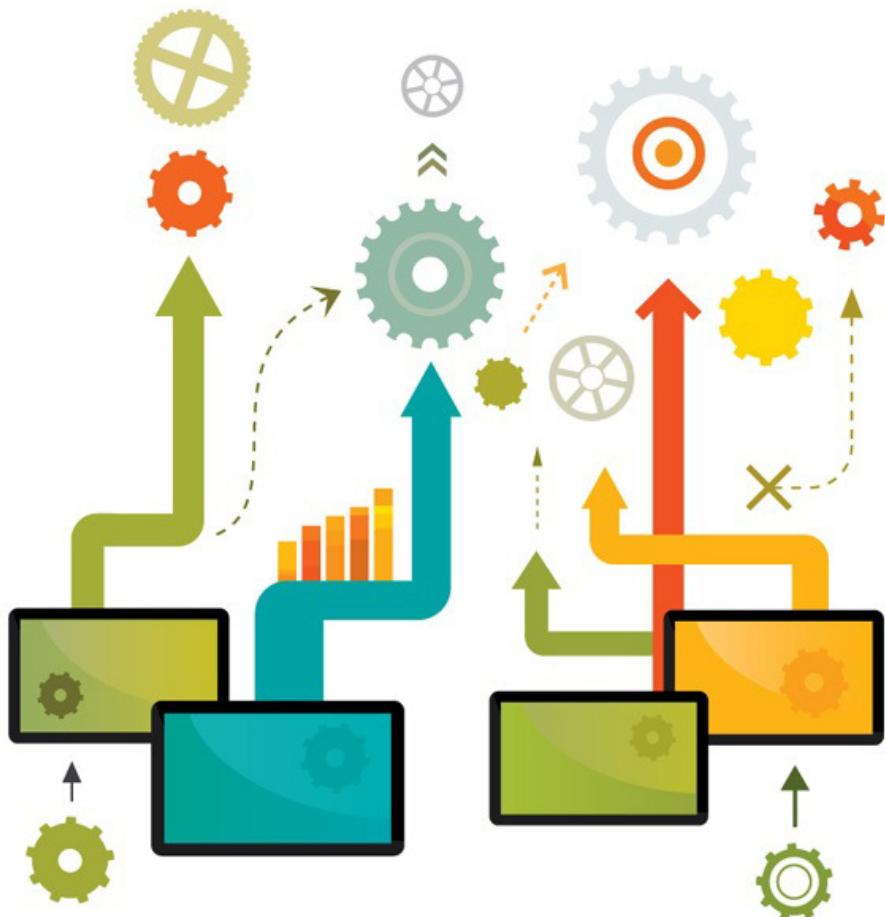


JSF Eficaz

As melhores práticas para
o desenvolvedor web Java



Casa do
Código

HÉBERT COELHO

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Os ISBNs do livro são:

- Impresso e PDF: 978-85-66250-19-0
- EPUB: 978-85-66250-82-4

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Primeiramente, agradeço a Deus por me capacitar para escrever o livro.

Agradeço à minha esposa por toda sua paciência durante o processo de escrita deste livro, e por sempre me animar nos momentos mais difíceis.

Aos meus pais, que sempre me apoiaram. Sou feliz também pelas minhas sobrinhas que, mesmo pequenas (Louanne e Fernanda), participaram no processo do livro — afinal estiveram durante uma semana em meus braços enquanto eu escrevia o livro. Não posso esquecer da minha irmã que sempre briga comigo. =D

Finalmente, agradeço ao pessoal da Casa do Código por me darem essa oportunidade e apoio durante todo o processo.

SOBRE O AUTOR

Hébert Coelho de Oliveira trabalha há mais de 10 anos com desenvolvimento de softwares. Possui as certificações SCJP, SCWCD, OCBCD, OCJPAD.

Criador do blog <http://uaiHebert.com>, visualizado por 170 países totalizando mais de 500 mil visualizações em seus 2 anos e meio de vida, é ainda autor do framework EasyCriteria (<http://easycriteria.uaihebert.com>), que ajuda na utilização da Criteria do JPA, sendo testado com Hibernate, OpenJPA e EclipseLink e com 100% de cobertura nos testes.

Foi revisor de um livro específico sobre Primefaces e criador de posts em seu blog com aplicações completas utilizando JSF. Escreveu um post sobre JSF com diversas dicas que alcançou mais de 3 mil visualizações no primeiro dia. Um post com 18 páginas, que foi a ideia original deste livro.

Pós-graduado em MIT Engenharia de Software — desenvolvimento em Java. Atualmente, atua como professor para o curso de pós-graduação, ensinando o conteúdo de Java Web (JSP, Servlet, JSF e Struts) e tópicos avançados, como EJB, Spring e WebServices.

SOBRE O LIVRO

O JSF é uma tecnologia muito útil e prática de ser aplicada, mas que diversas vezes é mal utilizada. Muitas vezes por falta de conhecimento de quem estrutura a aplicação, o projeto acaba ficando lento e de difícil manutenção.

Este livro tem por objetivo dar dicas e explicar conceitos que são necessários para que seja criada uma boa aplicação utilizando JSF. Às vezes, o mínimo detalhe — que vai desde como chamar um método, passar um valor para um ManagedBean ou até mesmo utilizar um converter — pode levar a horas perdidas de pesquisas na internet e testes na aplicação.

Este livro demonstrará boas práticas, dicas e o uso correto do JSF em diversos aspectos e diferentes situações.

Sumário

Use os escopos corretamente	1
1 Escolhas que afetam o desenvolvimento da aplicação	2
1.1 Suspeite se a aplicação está usando bem o JSF	2
1.2 Devo seguir todas as dicas ao pé da letra?	4
2 @RequestScoped para escopos curtos	5
3 Mantenha o bean na sessão com @SessionScoped	9
4 Entenda o novo @ViewScoped	14
5 Crie escopos longos e customizáveis com @ConversationScoped	20
6 A praticidade do escopo @Dependent	24
7 Guarde dados para toda a aplicação com o @ApplicationScoped	27
8 Quando usar o @NoneScoped?	30
9 Exibindo Objetos e Mensagens após Redirect e o FlashScoped	33

Cuidados com seus Managed Beans	39
10 Colocando lógica de rendered no MB	40
11 Inicializando Objetos	44
12 Injetando ManagedBeans	50
13 Target Unreachable: enfrente a NullPointerException do JSF	53
14 Cuidado com o Value is not valid	56
Front-ends JSF	58
15 Utilizar JSP ou xhtml?	59
16 Utilizando imagens/CSS/JavaScript de modos simples	62
17 Boa utilização do Facelets	65
18 Enviar valores para o ManagedBean	71
18.1 Envie valor como parâmetro pelo f:setPropertyActionListener	72
18.2 Envie valor como parâmetro	73
18.3 Envie valor por Binding	73
19 Temas dinâmicos	75
20 O que eu uso? Action ou ActionListener?	77
20.1 Redirecione o usuário com action	77
20.2 Trate eventos com actionPerformed	78

Aproveite as bibliotecas de componentes	81
21 Primefaces	83
21.1 DataTable com seleção com um click	83
21.2 Drag and drop	87
21.3 Notificador	92
21.4 AutoComplete	94
21.5 Poll	98
21.6 Considerações finais	100
22 Temas dinâmicos com Primefaces	102
23 Componentes do Primefaces não aparecem	108
24 Richfaces	110
24.1 DataTable com colunas congeladas	110
24.2 Tool Tip	113
24.3 Log	116
24.4 Panel Menu	117
24.5 Repeat	119
24.6 Considerações finais	120
25 Icefaces	122
25.1 Menu	122
25.2 Dialog	125
25.3 RichText	127
25.4 Data	128
25.5 Resize	129
25.6 Considerações finais	131

26 Evite misturar as implementações/bibliotecas de componentes	133
27 Não faça paginação no lado do servidor	135
Funcionalidades ricas com JSF, segurança e otimização do JSF	147
28 Facilitando o uso do Ajax	148
28.1 Sempre indique que a requisição está acontecendo	151
28.2 Previna-se das várias ações do usuário em requisições assíncronas	153
28.3 Cuidado ao usar ManagedBeans RequestScoped com Ajax	
29 Internacionalização e localização da sua aplicação	156¹⁵⁴
29.1 Permita que o usuário mude o idioma	158
30 Utilizando recursos dentro de um Converter	164
30.1 Acesse um ManagedBean programaticamente através de Expression Language	164
31 CDI com JSF	167
32 Evite o Cross Site Scripting em seu sistema	169
33 Otimizando a navegação e performance	171
Debug e inspeção de aplicações	172
34 Limpeza de comentários e debug	173
34.1 Esconda os comentários da página	173
34.2 Debug dos componentes	174

35 Organize funcionalidades por ambiente do projeto	178
--	------------

36 Refresh automático dos arquivos	181
---	------------

Versão: 20.9.8

Use os escopos corretamente

O JSF é um framework que tem um comportamento *component-based*. Ele tem por característica principal o fato de que a página buscará a informação no ManagedBean.

Cada ManagedBean tem um tipo de escopo ideal para cada situação. É fácil encontrar apologias ao uso indiscriminado do `SessionScoped`, assim como a defesa acirrada de que o ideal para todos os casos é o `RequestScoped`. Vamos analisar cada caso e ver qual a melhor solução para cada abordagem.

Antes, é importante salientar os dois modos de se declarar um ManagedBean. É possível utilizar um ManagedBean por CDI, usando o pacote `javax.inject`, ou por ManagedBean encontrado no pacote `javax.faces.bean`. Existem diferenças entre cada contexto que for utilizado, ambos com vantagens e desvantagens. Portanto, é importante conhecer os escopos para conseguir usá-los corretamente.

CAPÍTULO 1

ESCOLHAS QUE AFETAM O DESENVOLVIMENTO DA APLICAÇÃO

Você já sentiu um desânimo por ter de alterar uma funcionalidade? Ou ter de procurar por aquele bug que está aparecendo há meses? Muitas vezes, esse desânimo pode acontecer por decisões erradas durante a criação da aplicação. A pior parte é saber que, ao alterar um trecho do código, podemos ter efeitos colaterais indesejados em outros locais.

É possível encontrar diversos problemas técnicos, ou até mesmo conceituais, por escolhas erradas ao iniciar o desenvolvimento de uma nova aplicação. É necessário estruturá-la com conhecimento das ferramentas utilizadas; uma aplicação que tem frameworks mal usados será refém deles para sempre.

1.1 SUSPEITE SE A APLICAÇÃO ESTÁ USANDO BEM O JSF

Certa vez, me foi dada a trivial tarefa de mudar uma aba de lugar. A tarefa era apenas pegar uma aba que estava entre outras e exibi-la primeiro.



Figura 1.1: Exibir primeiro a aba Pessoa

A figura mostra como era o layout e como, teoricamente, seria simples passar a aba Pessoa para ser exibida antes da aba Carro .

O que seria uma tarefa de 15 minutos se transformou em uma caça às bruxas de 3 dias. Ao alterar as abas de posição, diversos erros começaram a acontecer. O primeiro erro que apareceu foi o cruel `NullPointerException`. Como um desenvolvedor poderia imaginar que, ao alterar uma aba de lugar, esse erro apareceria?

O principal problema dessa aplicação eram os escopos dos ManagedBeans. Todos eram `SessionScoped` e dependiam de informações em comum. Ao entrar na primeira aba (`carro`), diversos dados eram armazenados na sessão e usados em outras abas diretamente no ManagedBean que cuidava da aba `carro`. Ao trocar as abas de lugar, diversas informações não foram preenchidas nesse ManagedBean e, quando o ManagedBean da aba `Pessoa` fosse acessar essas informações, a `NullPointerException` aparecia.

Infelizmente, esse era um dos problemas da aplicação. Outro era que, ao carregar a tela, todas as informações de todas as outras abas eram carregadas. Era muita informação em memória, e erros começavam a acontecer sem explicação.

Às vezes, a escolha é feita pelos desenvolvedores, outras vezes

por algum desenvolvedor que fala: "*assim sempre funcionou e vamos continuar desse modo*". É preciso entender o framework com o qual estamos trabalhando, para só então apresentar argumentos e melhores técnicas na criação da aplicação.

1.2 DEVO SEGUIR TODAS AS DICAS AO PÉ DA LETRA?

Não. Você deve manter seu espírito crítico. O livro vai abrir sua mente para que você evite cair em armadilhas já tradicionais. Mas há sim situações em que você acaba subvertendo o framework.

CAPÍTULO 2

@REQUESTSCOPED PARA ESCOPOS CURTOS

O escopo RequestScoped funciona como um simples HTTP request. O ManagedBean não manterá seu estado entre as chamadas do usuário.

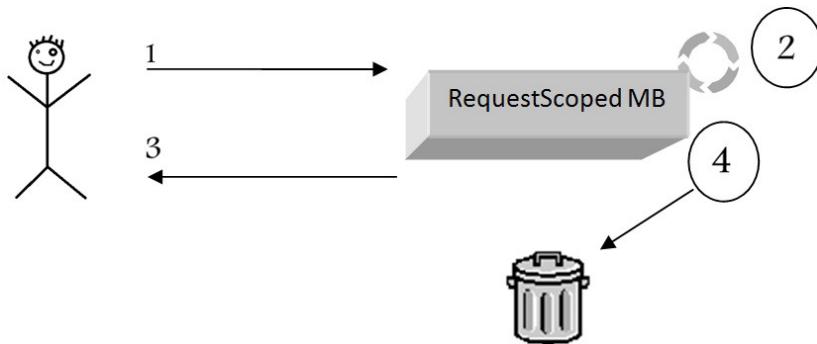


Figura 2.1: RequestScoped ManagedBean tratando requisição

Essa figura mostra como o JSF tratará a requisição ao se utilizar um ManagedBean RequestScoped:

1. O usuário iniciará uma requisição;
2. O ManagedBean processará as informações necessárias;
3. As informações do ManagedBean ficaram disponíveis para o

processamento da tela;

4. Caso algum valor tenha sido armazenado no ManagedBean, essas informações serão descartadas.

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.

```
import javax.faces.bean.*;  
  
@ManagedBean  
@RequestScoped  
public class RequestScopedMB {  
    private int numeroDeAcessos;  
  
    public int getNumeroDeAcessos() {  
        return ++numeroDeAcessos;  
    }  
  
    public void setNumeroDeAcessos(int numeroDeAcessos) {  
        this.numeroDeAcessos = numeroDeAcessos;  
    }  
}
```

Ao analisarmos o código da classe `RequestScopedMB`, é possível ver que, mesmo a classe tendo o atributo privado `numeroDeAcessos`, o seu valor será sempre igual a cada chamada. Note que, no método `getNumeroDeAcessos`, o valor do `numeroDeAcessos` é alterado. Não importa quantas vezes a página seja apresentada ao usuário, o valor retornado será sempre um.

BOA PRÁTICA

Os ManagedBeans por padrão são `@RequestScoped` e, com isso, a anotação pode ser omitida na declaração dos beans. Considere como boa prática sempre deixar seu ManagedBean anotado com `@RequestScoped`, pois com ela fica claro para quem lê o código qual é o escopo do ManagedBean, mesmo para um desenvolvedor que acaba de entrar no projeto e ainda não conhece o JSF.

O melhor uso de um ManagedBean no escopo de *request* é em telas que não necessitam de chamada AJAX, ou em de algum objeto salvo na memória. Considere uma situação na qual não é necessário nenhuma informação adicional de um objeto em memória. Basta enviar os dados presentes do formulário que um objeto será criado e estará pronto para ser persistido no banco de dados.

Incluir Profissional

* Nome:	Pedreiro de Software
Marretar senhas no código fonte	
Descrição:	Internacionalização, só os fracos usam EJB é para os marrentos, prefiro enums
<input checked="" type="button"/> Confirmar <input type="button"/> Cancelar	

Figura 2.2: Utilizando o RequestScoped de um bom modo

Essa figura mostra um exemplo de quando poderíamos utilizar um ManagedBean `@RequestScoped`. Note que a figura trata de uma tela de inclusão de dados, e não é necessário nenhum dado já processado anteriormente.

Um ManagedBean do tipo `RequestScoped` é considerado *ThreadSafe*, ou seja, ele não terá problemas relacionados a vários usuários acessando o mesmo MB ao mesmo tempo. É uma boa utilização na geração de relatório, pois não haveriam duas pessoas usando a mesma instância do ManagedBean, e o relatório poderia ter inconsistência de dados. Outro bom uso seria, ao enviar dados para uma outra tela, essa outra poderia exibir todos os dados do objeto presente no request.

CAPÍTULO 3

MANTENHA O BEAN NA SESSÃO COM @SESSIONSCOPED

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

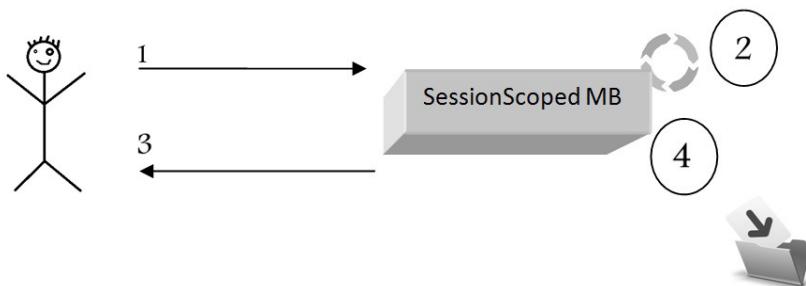


Figura 3.1: SessionScoped ManagedBean tratando requisição

A figura mostra como o JSF tratará a requisição ao se utilizar um ManagedBean SessionScoped :

1. O usuário iniciará uma requisição;
2. O ManagedBean processará as informações necessárias;

3. As informações do ManagedBean ficaram disponíveis para o processamento da tela;
4. Toda informação que foi atribuída ao ManagedBean será mantida enquanto durar a sessão do usuário no servidor.

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

```
import javax.faces.bean.*;  
  
@ManagedBean  
@SessionScoped  
public class SessionScopedMB {  
    private int numeroDeAcessos;  
  
    public int getNumeroDeAcessos() {  
        return ++numeroDeAcessos;  
    }  
  
    public void setNumeroDeAcessos(int numeroDeAcessos) {  
        this.numeroDeAcessos = numeroDeAcessos;  
    }  
  
    public boolean isAdministrador(){  
        return usuario.getPerfil().equals(Perfil.ADMINISTRADOR);  
    }  
}
```

No código da classe SessionScopedMB , o valor de numeroDeAcessos será alterado a cada requisição do usuário, e um novo valor será exibido para ele a cada exibição da página. O tipo SessionScoped está disponível tanto para ManagedBean do pacote javax.faces.bean como para os ManagedBeans que usam CDI.

A melhor utilização para um ManagedBean SessionScoped é

armazenar dados relativos ao usuário. Esses dados devem servir para facilitar o acesso às informações do usuário, ou até mesmo trabalhar "regras de view" — ou seja, regras que definirão o que um usuário pode visualizar ou acessar, entre outras funcionalidades.

```
// imports omitidos
@ManagedBean
@SessionScoped
public class UsuarioLoginMB {

    private Usuario usuario;
    private String localeUsuario = "pt_BR";

    public void logout() throws ServletException, IOException {
        HttpSession session = // busca a session

        session.invalidate();

        // efetua redirect
    }

    public boolean isAdministrador(){
        return usuario.getPerfil().equals(Perfil.ADMINISTRADOR);
    }

    // gets e sets omitidos
}
```

No código da classe `UsuarioMB`, é possível ver qual seria uma boa utilização para uma ManagedBean `SessionScoped`. Métodos e atributos poderiam ser adicionados/removidos para facilitar o acesso aos dados do usuário, por exemplo, um método para verificar os papéis do usuário no próprio ManagedBean para diminuir o caminho da EL. Em vez de `#{usuarioMB.usuario.administrador}`, bastaria fazer `#{usuarioMB.administrador}` e, no próprio ManagedBean, seria feito todo o tratamento dos dados.

O problema do SessionScoped é que ele é muito fácil de usar. Se um desenvolvedor precisar enviar valor de uma tela para outra, o modo mais fácil de fazer seria salvar esse valor em um ManagedBean do tipo SessionScoped . Mas ao mesmo tempo que ele é o mais fácil, é também o mais perigoso.

É preciso ter em mente que sempre que um valor é atribuído a um SessionScoped , ele permanecerá na memória. Se o servidor tiver mil usuários ativos e cada instância de UsuarioMB tiver pelo menos uma lista de carros que contenham 1.000 itens... Bom, aí é possível perceber como o uso da memória do servidor começaria a aumentar.

Quando o SessionScoped é muito usado, podemos ter ManagedBeans que utilizam informações de outros ManagedBeans, sendo que todos esses valores estão salvos na sessão. Imagine os ManagedBeans CarroMB , PessoaMB e CasaMB . Caso PessoaMB remova o carro do CarroMB , e a CasaMB esteja precisando desse valor, já é possível sentir o cheiro de NullPointerException .

O SessionScoped é um escopo muito útil, mas pode causar diversos problemas e dor de cabeça se mal usado. É necessário bastante cuidado ao usá-lo, ainda mais em telas de listagens e cadastro.

Caso o usuário abra o navegador na tela de BairrosMB , e depois acabe abrindo uma aba nova no mesmo navegador apontando para a tela da BairrosMB . A alteração realizada em uma aba pode interferir no funcionamento da outra.

Bairros	
Nome	Cidade
Asa Norte	Alandia - Acre
Bifucarção	Blandia - Bahia
Centro	Clandia
Dirlindo	Dlandia

Atualizar

* Nome:

* Cidade:

Confirmar Cancelar

Figura 3.2: Aba 1

Bairros	
Nome	Cidade
Asa Norte	Alandia - Acre
Bifucarção	Blandia - Bahia
Centro	Clandia
Dirlindo	Dlandia

Atualizar

* Nome:

* Cidade:

Confirmar Cancelar

Figura 3.3: Aba 2

É possível perceber que nas figuras que, se o usuário confirmar a alteração da Aba 2 e depois confirmar a alteração da Aba 1, alguma informação será perdida.

O SessionScoped é muito útil, mas se mal utilizado pode causar sérios problemas à aplicação.

CAPÍTULO 4

ENTENDA O NOVO @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. Veja as figuras seguintes para entender melhor o funcionamento desse escopo.

(4 of 10)		<	<<	1	2	3	4	5	6	7	8	9	10	>	>>	10	▼
		Name												Age			
Player: 30														30			
Player: 31														31			
Player: 32														32			
Player: 33														33			
Player: 34														34			
Player: 35														35			
Player: 36														36			
Player: 37														37			
Player: 38														38			
Player: 39														39			

Figura 4.1: ViewScoped com DataTable — 1

(4 of 10)		1	2	3	4	5	6	7	8	9	10	10
Name		Age										
Player: 30											30	
Player: 31											31	
Player: 32											32	
Player: 33											33	
Player: 34											34	
Player: 35											35	
Player: 36											36	
Player: 37											37	
Player: 38											38	
Player: 39											39	

Figura 4.2: ViewScoped com DataTable — 2

Se o usuário abrir uma tela com um `dataTable`, 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.

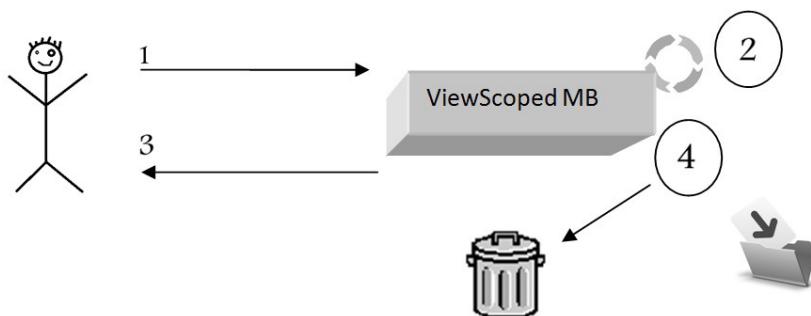


Figura 4.3: ViewScoped ManagedBean tratando requisição

Essa figura mostra como o JSF tratará a requisição ao se utilizar um `ManagedBean ViewScoped`:

1. O usuário iniciará uma requisição;
2. O ManagedBean processará as informações necessárias;
3. As informações do ManagedBean ficaram 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.

```
@ManagedBean  
@ViewScoped  
public class ViewScopedMB implements Serializable {  
    private int numeroDeAcessos;  
  
    public int getNumeroDeAcessos() {  
        return ++numeroDeAcessos;  
    }  
  
    public void setNumeroDeAcessos(int numeroDeAcessos) {  
        this.numeroDeAcessos = numeroDeAcessos;  
    }  
  
    public String somar() {  
        numeroDeAcessos++;  
  
        return null;  
    }  
}
```

No exemplo da classe `ViewScopedMB` , o valor do `numeroDeAcessos` permanecerá na memória enquanto o usuário permanecer na página. Note que o método `somar` retorna `null` . Isso faz com que a ação redirecione para a própria página e, como o bean é `ViewScoped` , manterá na memória os seus valores.

É preciso ter em mente que os dados permanecerão "vivos" no ManagedBean quando os métodos executados retornarem `null` ou sejam `void` , assim indicando que devem permanecer na

mesma tela. Qualquer navegação para outra página, ou até mesmo um método retornando o endereço da página atual, causará a perda dos dados do ManagedBean.

Um bom uso para um ManagedBean do tipo ViewScoped é em uma página com diversas chamadas Ajax. Imagine um `h:dataTable` sendo exibido e diversos `dialogs` para exibir informações de registros relacionados ao `dataTable`. Desse modo, as informações estariam na memória e não haveria a necessidade de voltar ao banco de dados a cada chamada.

Atente-se a um problema que pode ocorrer com relação a acesso de diversos usuários ao banco de dados. Veja novamente a figura *ViewScoped com DataTable — 1* e note uma coisa: o valor que está sendo exibido está armazenado em memória. Imagine que um usuário altere os dados do Player 35 no banco de dados. Nesse caso, se um outro usuário estivesse com essa tela aberta a mais tempo, ele não veria essa alteração.

No caso de uma tela com edição, o problema seria maior ainda. Veja as figuras a seguir para entender como o problema pode se agravar.

(8 of 10)		<	<	1	2	3	4	5	6	7	8	9	10	>	>	10 ▾
		Name						Age								
Player: 70													70			
Player: 71													71			
Player: 72													72			
Player: 73													73			
Player: 74													74			
Player: 75													75			
Player: 76													76			
Player: 77													77			
Player: 78													78			
Player: 79													79			

Player

x

Id: 76

Name: Player: 75

Age: 75

Alterar Cancelar

Figura 4.4: Edição com ViewScoped — 1

(8 of 10)		<	<	1	2	3	4	5	6	7	8	9	10	>	>	10 ▾
		Name						Age								
Player: 70													70			
Player: 71													71			
Player: 72													72			
Player: 73													73			
Player: 74													74			
Player: 75													75			
Player: 76													76			
Player: 77													77			
Player: 78													78			
Player: 79													79			

Player

x

Id: 76

Name: Minhocá

Age: 77

Alterar Cancelar

Figura 4.5: Edição com ViewScoped — 2

Imagine que o usuário Astrogildo abriu a tela de edição (*Edição com ViewScoped — 1*) e foi ao banheiro. Enquanto isso, a usuária

Antonieta, por meio do seu PC, entrou no mesmo registro, alterou e persistiu a alteração no banco de dados (*Edição com ViewScoped — 1*). O que aconteceria, ao retornar do banheiro, se Astrogildo apertasse o botão *Alterar*? Toda a alteração realizada pela Antonieta seria perdida.

Para evitar o problema de dados fora de sincronia com o banco de dados, uma rotina de atualização poderia ser executada. A cada 1 minuto, uma verificação poderia ser realizada, ou talvez antes de exibir cada registro. Uma outra abordagem seria adotar algum mecanismo ou *Framework* para tomar conta dessa falta de sincronia.

O JPA conta com a opção de verificar se o objeto foi alterado por outra pessoa. Desse modo, quando Astrogildo apertasse o botão *Alterar*, uma mensagem de erro seria exibida.

CAPÍTULO 5

CRIE ESCOPOS LONGOS E CUSTOMIZÁVEIS COM @CONVERSATIONSCOPED

O ManagedBean do tipo ConversationScoped tem um funcionamento parecido com o ViewScoped . A característica principal do conversationScoped é que o controle da existência do ManagedBean é feito manualmente.

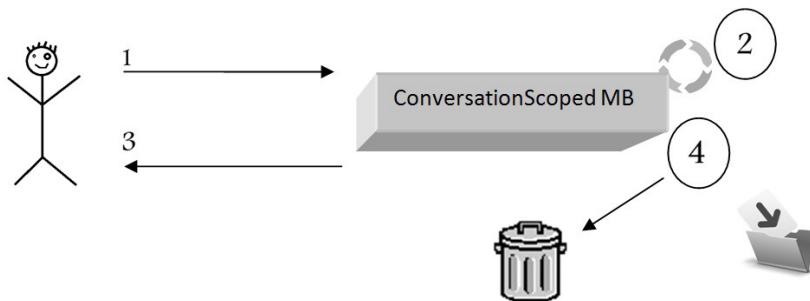


Figura 5.1: ConversationScoped ManagedBean tratando requisição

A figura mostra como o JSF tratará a requisição ao se utilizar um ManagedBean ConversationScoped :

1. O usuário inicia uma requisição;

2. O ManagedBean processará as informações necessárias;
3. As informações do ManagedBean ficaram disponíveis para o processamento da tela;
4. Enquanto o ManagedBean não for finalizado, os dados serão mantidos;
5. Uma vez que o comando para finalizar o ManagedBean for executado, seus dados serão eliminados da memória.

O ConversationScoped tem seu tempo de vida definido programaticamente. Ele funciona basicamente como uma transação de banco de dados. Para iniciar seu ciclo de vida, é necessário um comando de início, da mesma forma que, para encerrar o escopo, é preciso de um comando que identifique o fim.

Para um ManagedBean do tipo ConversationScoped funcionar corretamente, ele deve seguir algumas normas:

- Como só pode ser usado com CDI, o arquivo beans.xml deve existir dentro da pasta WEB-INF .
- Utilizar na classe a anotação javax.enterprise.context.ConversationScoped .
- Injetar um objeto do tipo javax.enterprise.context.Conversation .
- Chamar os métodos conversation.begin() e conversation.end() para iniciar e finalizar o escopo do ManagedBean.

```
// imports omitidos
@Named
@ConversationScoped
public class ConversationScopedMB implements Serializable {
```

```
@Inject
private Conversation conversation;

private int numeroDeAcessos;

public String iniciar(){
    conversation.begin();
    return null;
}

public boolean isTransient(){
    return conversation.isTransient();
}

public String somar(){
    if(!conversation.isTransient()){
        numeroDeAcessos++;
    }

    return null;
}

public String finalizar(){
    conversation.end();
    return
    "/paginas/parte2/
        conversationScoped.xhtml?faces-redirect=true";
}

public String navegar(){
    if(conversation.isTransient()){
        return null;
    }

    return
    "/paginas/parte2/
        conversationScoped2.xhtml?faces-redirect=true";
}

public int getNumeroDeAcessos() {
    return ++numeroDeAcessos;
}

public void setNumeroDeAcessos(int numeroDeAcessos) {
    this.numeroDeAcessos = numeroDeAcessos;
```

```
    }  
}
```

No código da classe `ConversationScopedMB`, tem um método para abrir e finalizar a conversação do ManagedBean, nesse caso `iniciar` e `finalizar`, respectivamente. É necessário sempre se lembrar do escopo, ou então recursos poderão ficar alocados na memória desnecessariamente.

Na classe `ConversationScopedMB`, é possível ver que realizamos uma navegação e mantivemos o escopo vivo. A navegação deve começar de dentro do ManagedBean, por exemplo, pelo método `navegar`, porque assim os dados permanecerão na memória.

O ManagedBean do tipo `ConversationScoped` tem apenas dois estados: `transient` e `long running`. O estado será `transient` antes do início e após o fim da conversação.

O `ConversationScoped` é ideal para ser usado em uma tela na qual existe um `h:dataTable` com muitas chamadas ajax, e os dados devem ser mantidos até uma determinada invocação. Outra vantagem em utilizar o `ConversationScoped` é manter um objeto vivo entre diversas telas: a navegação entre telas não eliminará objetos da memória, diferentemente do `ViewScoped`.

O `ConversationScoped` só pode ser utilizado com CDI. Não existe essa opção para os ManagedBeans do pacote `javax.faces.bean`. Caso a sessão do usuário termine, o ManagedBean do tipo `ConversationScoped` será eliminado da sessão.

CAPÍTULO 6

A PRATICIDADE DO ESCOPO @DEPENDENT

O escopo Dependent tem como característica não ter um comportamento próprio, mas sim herdar o comportamento de outro ManagedBean em que for injetado.

```
//Imports omitidos
@Named
@Dependent
public class DependentScopedMB implements Serializable {
    private int numeroDeAcessos;

    public int getNumeroDeAcessos() {
        return ++numeroDeAcessos;
    }

    public void setNumeroDeAcessos(int numeroDeAcessos) {
        this.numeroDeAcessos = numeroDeAcessos;
    }
}
```

Esse é o escopo padrão ao se usar JSF com CDI. Não é necessário utilizar a anotação @Dependent na classe.

Quando a @Dependent é usada diretamente em uma página, cada EL resultará na criação de um ManagedBean novo. Caso uma página contenha `#{{dependentScopedMB.total}}` #`{dependentScopedMB.total}`, será impresso 1 1 .

Em geral, o escopo `Dependent` é injetado dentro de outros `ManagedBeans`. Desse modo, ele herdará o escopo da classe principal. Se o `DependentScopedMB` fosse injetado no `SessionScopedMB`, seu escopo também seria de sessão.

```
// imports omitidos
public class MeuMB{
    @Inject
    private DependentScopedMB dependent;

    public void encaminharRequisicao(){
        int tempoSessao =
            dependent.buscarConfiguracao("tempoSessao");
    }

    // outros metodos omitidos
}
```

BOA PRÁTICA

Considero como boa prática sempre deixar anotado com `@Dependent`. Por mais que algumas pessoas da equipe possam saber que o valor padrão é esse, com o passar do tempo haverá pessoas que talvez não saibam disso. Com o valor `@Dependent`, fica claro para quem lê o código qual é o escopo do `ManagedBean`.

Tome bastante cuidado ao injetar um `Dependent ManagedBean` dentro de um `ApplicationScoped`. Essa abordagem pode levar ao problema de *Memory Leak*. Após um método usar um `ManagedBean` do tipo `Dependent`, ele não é totalmente descartado, mantendo uma referência para o `ApplicationScoped`. Se o mesmo método for chamado

novamente, outra referência será criada para ser utilizada, e assim em diante.

CAPÍTULO 7

GUARDE DADOS PARA TODA A APLICAÇÃO COM O @APPLICATIONSCOPED

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

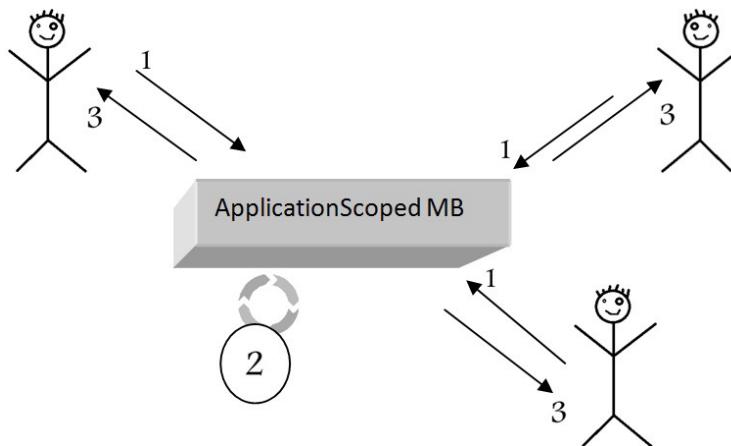


Figura 7.1: ApplicationScoped ManagedBean tratando requisição

A figura mostra como o JSF tratará a requisição ao se utilizar um ManagedBean ApplicationScoped :

1. O usuário iniciará uma requisição;
2. O ManagedBean processará as informações necessárias;
3. As informações do ManagedBean ficaram 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.

Note que não existe individualidade quando se fala de `ApplicationScoped`. No caso de um ManagedBean `ApplicationScoped`, todo usuário terá acesso à mesma instância. Seria uma prática muito errada salvar alguma informação em um `ApplicationScoped` que pertencesse somente a um determinado usuário.

O tipo `ApplicationScoped` está disponível tanto para ManagedBean do pacote `javax.faces.bean` como para os ManagedBeans que utilizem CDI. O melhor uso para o `ApplicationScoped` seria para conter valores utilizados por toda a aplicação.

```
@ManagedBean
@ApplicationScoped
public class ApplicationScopedMB implements Serializable {
    private int numeroDeAcessos;

    public int getNumeroDeAcessos() {
        return ++numeroDeAcessos;
    }

    public void setNumeroDeAcessos(int numeroDeAcessos) {
        this.numeroDeAcessos = numeroDeAcessos;
    }
}
```

A classe `ApplicationScopedMB` mostra como usar o

`ApplicationScoped` . Todos os usuários do sistema que acessarem esse ManagedBean verão o contador aumentar. Esse tipo de escopo é ideal para conter valores como configuração, ou objetos caros de se criar e que devem ser instanciados apenas uma vez.

Uma outra observação importante sobre o `ApplicationScoped` é que, às vezes, se faz necessário que ele seja iniciado antes de todos ManagedBeans. Isso é possível através da configuração: `@ManagedBean(eager = true)` . Com essa configuração, um `ApplicationScoped` ManagedBean será iniciado antes que qualquer tela da aplicação seja acessada.

Esse configuração é bastante útil quando já queremos alguma informação carregada em memória antes que seja solicitada pelo usuário. Imagine que um cache de Cidades seja feito: se não usasse o `eager` , o cache se faria apenas quando o valor fosse solicitado pela primeira vez, causando demora na chegada da informação.

Outra aplicação para a opção `eager=true` é caso um `ApplicationScoped` ManagedBean seja injetado dentro de outro ManagedBean. Pode acontecer que o JSF ainda não tenha inicializado o `ApplicationScoped` e os valores estejam nulos.

CAPÍTULO 8

QUANDO USAR O @NONESCOPED?

O `NoneScoped` ManagedBean tem por característica servir apenas a uma chamada da EL, e depois ser eliminado da memória. Veja que sua utilização difere do `RequestScoped` quanto a duração das informações. O `RequestScoped` dura por quantas chamadas de EL forem realizadas durante uma requisição. O `NoneScoped` será eliminado após uma chamada de EL.

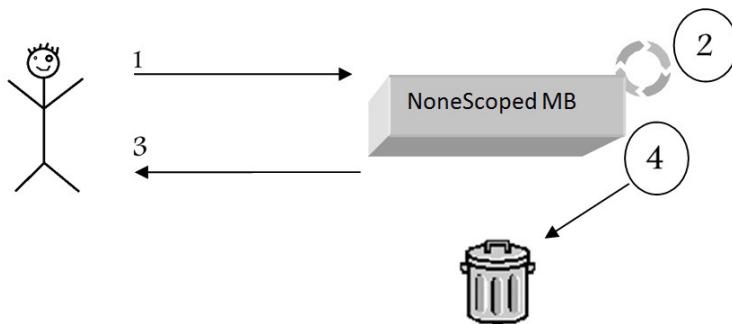


Figura 8.1: NoneScoped Scoped ManagedBean tratando requisição

A figura mostra como o JSF tratará a requisição ao se utilizar um ManagedBean `NoneScoped` :

1. O usuário iniciará uma requisição;

2. O ManagedBean processará as informações necessárias;
3. As informações do ManagedBean ficaram disponíveis para o processamento da tela;
4. O mesmo ManagedBean não responderá a outras chamadas.
Ele será descartado após receber uma chamada.

É necessário entender que esse escopo tem por característica ser altamente descartável.

```
// imports omitidos
@ManagedBean
@NoneScoped
public class NoneScopedMB {
    private int numeroDeAcessos;

    public int getNumeroDeAcessos() {
        return ++numeroDeAcessos;
    }

    public void setNumeroDeAcessos(int numeroDeAcessos) {
        this.numeroDeAcessos = numeroDeAcessos;
    }
}
```

Se o `NoneScopedMB` fosse usado da seguinte forma de uma EL, como:

```
#{noneScopedMB.numeroDeAcessos} #{noneScopedMB.numeroDeAcessos}
```

O valor impresso seria `1 1`. O `NoneScoped` atenderia apenas a uma chamada do EL e depois deixaria de existir.

O melhor uso para o `NoneScoped` seria para tarefas pontuais, como formatar um campo, exibir hora ou realizar um cálculo. Outro bom uso seria para acessar propriedades do `faces-config.xml`.

O `NoneScoped` ManagedBean está disponível através do

pacote javax.faces.bean .

CAPÍTULO 9

EXIBINDO OBJETOS E MENSAGENS APÓS REDIRECT E O FLASHSCOPED

No projeto de exemplo do livro, é possível encontrar uma tela de cadastro como na figura:

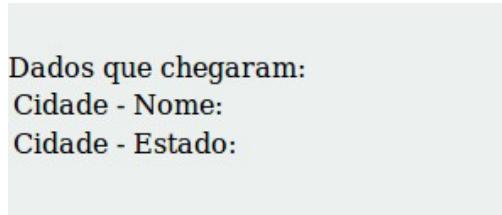
Cadastre uma Cidade:

Nome

Estado

Figura 9.1: Cadastro de cidade

Cadastramentos de dados na aplicação costumam vir com alguma mensagem indicativa para o usuário de que aquele registro foi salvo no banco de dados com sucesso. O problema acontece quando, após realizar a persistência com sucesso, nenhuma mensagem é exibida ao usuário, algo como:



Dados que chegaram:
Cidade - Nome:
Cidade - Estado:

Figura 9.2: Não exibe dados após uma ação

```
// finalizando cadastro  
super.exibirMensagemParaUsuario(FacesMessage.SEVERITY_INFO,  
        "Nome da cidade cadastrada: " + cidade.getNome());  
  
return "/cadastroComSucesso.xhtml?faces-redirect=true";
```

Mesmo adicionando a mensagem para ser exibida ao usuário, como no código visto, a mensagem não é exibida e os dados da cidade recém-cadastrada não chegam na próxima tela. Isso acontece por um pequeno detalhe, o `redirect`.

Muitos desenvolvedores percebem que, ao realizar um `redirect`, informações que deveriam ser exibidas na tela não aparecem, como se tivessem sido perdidas. Na verdade, é um comportamento esperado essas informações se perderem, mas para entendermos a razão, precisamos compreender o impacto em se utilizar um `redirect`.

Veja na figura a seguir como ele funciona:

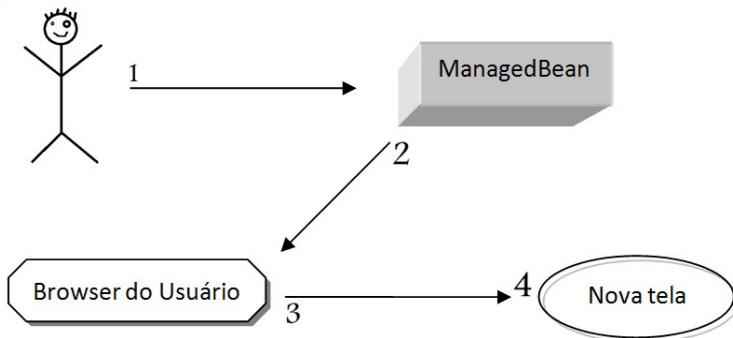


Figura 9.3: Como funciona o `SendRedirect`

Conseguimos então perceber que:

1. O usuário enviará um `request` ao `ManagedBean`;
2. Ao executar o comando `SendRedirect`, o `ManagedBean` envia para o browser do usuário uma URL com um novo destino (esse valor vai em um `%header` do `request`);
3. O browser do usuário analisará o `response` e fará uma nova chamada para a URL que recebeu;
4. A nova tela será exibida ao usuário.

Quando o `ManagedBean` redireciona o usuário para uma nova tela, todos os dados do `request` original se perdem. Dados enviados no `form`, mensagens para serem exibidas ao usuário e outros valores se perderão.

Para ajudar o desenvolvedor a tratar esse comportamento, foi criado o chamado `FlashScope`. Ao usar o `FlashScope` para manter os objetos na memória, o próprio JSF manterá o objeto na memória até que ele seja utilizado.

Veja o código da classe `SendRedirectMB` que mostra quando

o problema poderá acontecer:

```
@ManagedBean  
@RequestScoped  
public class SendRedirectMB extends AbstractMB{  
  
    private int valorParaExibir;  
    private static final String URL = "URL PARA PÁGINA";  
  
    public int getValorParaExibir() {  
        return valorParaExibir;  
    }  
  
    public void setValorParaExibir(int valorParaExibir) {  
        this.valorParaExibir = valorParaExibir;  
    }  
  
    public String redirecionarSemFlash() {  
        super.exibirInformacao("Valor Enviado é de: " +  
                               valorParaExibir);  
        return URL + "?faces-redirect=true";  
    }  
  
    public String redirecionarComFlash() {  
        super.exibirInformacao("Valor Enviado é de: " +  
                               valorParaExibir);  
        FacesContext instance =  
            FacesContext.getCurrentInstance();  
        ExternalContext externalContext =  
            instance.getExternalContext();  
        externalContext.getFlash().put("valorParaExibir",  
                                      valorParaExibir);  
        externalContext.getFlash().setKeepMessages(true);  
        return URL + "?faces-redirect=true";  
    }  
}
```

Uma vez que o método `redirecionarSemFlash` for executado, todos os dados presentes no request original serão perdidos. O método `super.exibirInformacao` nada mais faz do que exibir uma mensagem para o usuário, que pode ser visualizada com um `h:messages`.

Para exibir um objeto ou um atributo após um `SendRedirect`, basta fazer como no método `redirecionarComFlash`. Basta utilizar o `FlashScope externalContext.getFlash().put("valorParaExibir", valorParaExibir);` para armazenar o valor como se fosse um mapa. Esse mapa armazenará o valor até o final do redirecionamento.

Para exibir esse valor na outra tela, basta fazer:
`<h:outputText value="#{flash.valorParaExibir}" />`. Após o valor ser acessado no `FlashScope`, ele será eliminado da memória. É possível também manter o objeto na sessão, basta acessar o objeto por
`<h:outputText value="#{flash.keep.valorParaExibir}" />`.

Para exibir uma mensagem para o usuário, existem diferentes maneiras: duas fáceis e uma mais complexa. Após inserir a mensagem no contexto para exibi-la para o usuário, basta executar o seguinte comando:
`externalContext.getFlash().setKeepMessages(true)`. Desse modo, a mensagem será exibida para o usuário após ser redirecionado.

O outro modo de exibir a mensagem é adicionar a tag a seguir na página:
`<c:set target="#{flash}" property="keepMessages" value="true" />`. Assim, a mensagem será salva no flash e exibida ao usuário.

O último modo envolve criar um `PhaseListener` que faça o trabalho manual de persistir o valor na sessão e depois remover. Para maiores informações dessa abordagem, visite o link:
<http://uaihebert.com/?p=499>.

Veja na figura seguinte os dados sendo exibidos após utilizar as técnicas apresentadas aqui.

- Nome da cidade cadastrada: Itaoca da Pedra

Dados que chegaram:

Cidade - Nome: Itaoca da Pedra

Cidade - Estado: Espírito Santo

Figura 9.4: Cadastro de cidade

Cuidados com seus Managed Beans

Todo desenvolvedor precisa ter bastante cuidado ao adicionar códigos que lidam com a lógica de visualização da sua aplicação, como definições de "o que pode ser exibido para determinado usuário?", "como exibir determinada informação para perfis diferentes de visualizações".

Um código criado sem pensar em boas práticas pode levar a sérios problemas de manutenção, acabamos por ter classes grandes e complexas, e muitas funções e códigos repetidos.

Cito aqui a teoria da janela quebrada apresentada pelos cientistas James Q. Wilson e George L. Kelling. Imagine um prédio com janelas quebradas. Se essas janelas não forem rapidamente reparadas, a chance de vândalos quebrarem mais janelas aumentam. Do mesmo modo, é possível acontecer com sistemas. Basta que o primeiro código *feio* seja adicionado ao projeto, que aumentará a chance de um segundo código *feio* aparecer e assim por diante.

CAPÍTULO 10

COLOCANDO LÓGICA DE RENDERED NO MB

Considere o código de um panel, no qual existe uma lógica para determinar se ele será exibido ou não:

```
<h1>Relatório de Salários</h1>
<h:panelGrid columns="2"
    rendered="#{relatorioMB.usuario.papel eq 'ADM'}" >
    <h:outputText value="Salário do Funcionário:" />
    <h:outputText value="#{funcionario.salario}" />
</h:panelGrid>
```

O trecho de um relatório de salários poderia claramente exemplificar essa situação. O código do *Relatório de Salários* é simples e fácil de entender. Nele existe apenas a lógica para ver se o usuário é ou não administrador. Caso o usuário seja um administrador, então será mostrado o salário do funcionário.

Uma primeira boa prática seria aplicar o princípio *Tell, don't ask* ("Diga, não pergunte"). Essa prática diz que é melhor deixar que o próprio objeto diga se ele pode ou não fazer tal ação, e não outro objeto calcular essa ação por ele. Veja o código da classe `Usuario`, que mostra como ficaria um método aplicando esse princípio.

```
public enum Papel {
```

```

ADM, GERENTE, USUARIO_SIMPLES;
}

public class Usuario {
    private String login;
    private String email;
    private Papel papel;

    public boolean isAdm() {
        return Papel.ADM.equals(papel);
    }

    public boolean isGerente() {
        return Papel.GERENTE.equals(papel);
    }

    public boolean isUsuarioSimple() {
        return Papel.USUARIO_SIMPLES.equals(papel);
    }

    // getters e setters omitidos
}

```

Note que a classe `Usuario` informa qual o papel do usuário. A vantagem dessa abordagem é que, caso a cláusula `rendered="#{usuario.papel eq 'ADM'}` passasse para `rendered="#{usuario.papel eq 'ADM' and usuario.dataAdmissao > '10-10-2000'}`, bastaria alterar na classe `Usuario` que todas as ELs que utilizassem o novo método não precisariam ser alteradas. Veja como ficará o código alterado do relatório.

```

<h1>Relatório de Salários</h1>
<h:panelGrid columns="2" rendered="#{relatorioMB.usuario.adm}" >
    <h:outputText value="Salário do Funcionário:" />
    <h:outputText value="#{funcionario.salario}" />
</h:panelGrid>

```

Note que não importa quantas vezes a regra para calcular se o usuário é um administrador mude, isso não afetará mais a EL. Independente de quantas vezes a EL for utilizada, uma alteração na

regra não seria necessária uma alteração nas páginas.

Ainda assim, é possível notar que esse código pode ser melhorado. Imagine que agora para ver o salário do funcionário, além de ser administrador, o usuário precisa ser do RH. Seria bem simples alterar o relatório e adaptar essa regra.

```
<h1>Relatório de Salários</h1>
<!-- relMB foi reduzido de relatorioMB para melhor
visualização -->
<h:panelGrid
    rendered="#{relMB.usuario.rh and relMB.usuario.adm}" >
    <h:outputText value="Salário do Funcionário:" />
    <h:outputText value="#{funcionario.salario}" />
</h:panelGrid>
```

Realmente, o código está simples e legível. Mas imagine se esse mesmo `if` está repetido 40x por essa página, e talvez até em outras. É agora que entra a triste realidade do desenvolvedor, aquele trágico momento em que percebemos que será necessário vasculhar cada página e ver onde essa regra foi aplicada. E a pior parte é que talvez a condição `rendered` esteja assim:

```
rendered="#{relatorioMB.usuario.rh and relatorioMB.usuario.adm}"
```

Ou assim:

```
rendered="#{relatorioMB.usuario.adm and relatorioMB.usuario.rh}"
```

Ou assim:

```
rendered="#{rel.user.adm and rel.user.rh}"
```

Essa variação poderia causar a impossibilidade de usar uma pesquisa para localizar o que necessitaria de alteração.

Existe mais uma abordagem simples que ajudará a evitar muita dor de cabeça: basta isolrar essas regras em algum lugar, como o

ManagedBean. Não é necessário mágica, nem um conhecimento profundo em arquitetura para aplicar essa simples abordagem.

A tag `rendered` ficaria assim:

```
rendered="#{relatorioMB.usuarioPodeVerSalario}"
```

Note que, a partir de agora, quem está tomando conta dessa regra de view é o próprio ManagedBean. É ele quem dirá se o usuário poderá ver ou não o relatório. Esse método poderia ser replicado por milhares de vezes que, caso a regra mudasse, não haveria impacto nas páginas.

```
public boolean isUsuarioPodeVerSalario(){  
    return usuario.isADM() && usuario.isRH();  
}
```

O método `isUsuarioPodeVerSalario` mostra exatamente a vantagem da abordagem de centralizar as regras de view em um ManagedBean. Caso o método fosse usado em 30 validações diferentes, não haveria problema algum se a regra alterasse.

CAPÍTULO 11

INICIALIZANDO OBJETOS

Quando estamos implementando os Managed Beans do projeto, precisamos tomar bastante cuidado com a inicialização dos seus atributos, pois algumas armadilhas podem nos esperar. No código a seguir, inicializamos a lista de carros no construtor, mas será que essa é a melhor prática?

```
@ManagedBean  
public class InicializandoMB{  
    private List<Carro> carros;  
  
    @EJB  
    private CarroDAO carroDAO;  
  
    public InicializandoMB(){  
        carros = carroDAO.listAll();  
    }  
    // getters e setters omitidos  
}
```

Usamos um EJB no construtor da classe para buscar uma lista de carro no banco de dados. No entanto, a injeção de dependências acontece apenas após a invocação do construtor. Nesse caso, receberemos uma `NullPointerException`, pois o EJB ainda não estará injetado e pronto para ser invocado.

Um modo de evitar todos os problemas listados seria inicializar os atributos em um método anotado com

`javax.annotation.PostConstruct`. Por contrato, uma classe Java EE terá todos seus recursos inicializados antes do método anotado com `@PostConstruct` ser invocado.

```
@ManagedBean
public class InicializandoMB {
    private List<Carro> carros;

    @EJB
    private CarroDAO carroDAO;

    @PostConstruct
    public void init() {
        carros = carroDAO.listAll();
    }

    // getters e setters omitidos
}
```

Após alterar o código da classe `InicializandoMB`, é possível notar que o código que havia no construtor da classe foi movido para o método `init()`, e o construtor foi removido do código. O próprio servidor ficará encarregado de chamar o método `init()` e garantir que ele seja executado antes que o ManagedBean interaja com chamadas externas.

Infelizmente, há uma desvantagem na abordagem de inicialização de atributos em um método anotado com `@PostConstruct`. Considere agora que esse ManagedBean tem outros métodos que não necessitam do valor da lista `carros`.

```
@ManagedBean
public class InicializandoMB{
    private List<Carro> carros;

    @EJB
    private CarroDAO carroDAO;

    @PostConstruct
```

```

public void init(){
    carros = carroDAO.listAll();
}

public List<String> getMarcas(){
    return carroDAO.getMarcas();
}

public List<String> getModelos(){
    return carroDAO.getModelos();
}
// getters e setters omitidos
}

```

Repare que o método `getMarcas()` e o método `getModelos()` podem ser chamados por uma página que não utilize a lista de `carros` para nada. A desvantagem dessa abordagem é: mesmo que apenas os métodos `getMarcas()` e `getModelos()` sejam usados, a lista de `carros` continuará a ser carregada do banco de dados desnecessariamente.

Nesse caso, podemos usar a abordagem *lazy* (preguiçosa) de se carregar objetos no ManagedBean. Veja na nova versão da classe `InicializandoMB` como utilizar essa abordagem de inicialização.

```

@ManagedBean
public class InicializandoMB{
    private List<Carro> carros;

    @EJB
    private CarroDAO carroDAO;

    public List<Carro> getCarros(){
        if (carros == null) {
            carros = carroDAO.listAll();
        }
        return carros;
    }

    public List<String> getMarcas(){

```

```
        return carroDAO.getMarcas();
    }

    public List<String> getModelos(){
        return carroDAO.getModelos();
    }
    // getters e setters omitidos
}
```

O método `init()` foi removido do código e o método `getCarros` foi alterado. Caso a lista `carros` seja `null`, a consulta será realizada no banco de dados. Com essa abordagem, se o método `getCarros` nunca for invocado, essa lista nunca será inicializada e muitas viagens ao banco de dados deixarão de existir.

A desvantagem dessa abordagem é o fato de que cada método `get` deve ter um teste do tipo `if (objeto == null)`. A quantidade de linhas da classe poderá subir consideravelmente.

SERVE PARA TODO ESCOPO

Note que iniciar um objeto de modo *lazy* ou pelo `@PostConstruct` é uma boa prática e pode/deve ser aplicada a qualquer tipo de escopo, não apenas no `@RequestScoped`, como exibido na classe `InicializandoMB`.

É possível utilizar uma abordagem híbrida na qual temos atributos carregados em um método anotado com `@PostConstruct` e atributos inicializados de modo *lazy*.

```
@ManagedBean
public class InicializandoMB{
    private List<Carro> carros;
    private List<Cidade> cidades;
```

```

@EJB
private CarroDAO carroDAO;

@EJB
private CidadeDAO cidadeDAO;

@PostConstruct
public void init() {
    cidades = cidadeDAO.listAll();
}

public List<Carro> getCarros() {
    if (carros == null) {
        carros = carroDAO.listAll();
    }
    return carros;
}

public List<String> getMarcas() {
    return carroDAO.getMarcas();
}

public List<String> getModelos() {
    return carroDAO.getModelos();
}
// getters e setters omitidos
}

```

No código da classe `InicializandoMB`, é possível ver que ambas abordagens foram usadas. Preferencialmente, métodos anotados `@PostConstruct` devem carregar objetos que serão de uso comum em todas, ou na maior parte, das chamadas realizadas ao ManagedBean.

Uma outra maneira de iniciar objetos é informando ao JSF que um método deve ser executado antes de carregar toda a página. Esse método é informado através da página. Veja como a classe `InicializandoMB` ficará após uma leve refatoração.

`@ManagedBean`

```
public class InicializandoMB{  
    private List<Carro> carros;  
    private List<Cidade> cidades;  
  
    @EJB  
    private CarroDAO carroDAO;  
    @EJB  
    private CidadeDAO cidadeDAO;  
  
    public void inicializar(){  
        cidades = cidadeDAO.listAll();  
        carros = carroDAO.listAll();  
    }  
    // outros métodos omitidos  
}
```

Temos o método `inicializar` que será disparado quando a página for chamada pela tag `f:event`.

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

Basta adicionar o código da tag `f:event` que o JSF chamará o método que carrega todos os atributos da tela antes de executar qualquer ação.

CAPÍTULO 12

INJETANDO MANAGEDBEANS

Considere um ManagedBean em que, para decidir se o usuário pode ou não ver determinada informação, ele precisa determinar se o usuário tem o perfil necessário.

```
// imports omitidos
@ManagedBean
public class CarroMB {
    public List<Carro> getCarrosPorUsuario() {
        List<Carro> carros = new ArrayList<Carro>();
        usuario = // recupera usuário logado

        adicionarCarrosSimples(carros, usuario);
        adicionarCarrosEconomicos(carros, usuario);
        adicionarCarrosLuxuosos(carros, usuario);

        return carros;
    }
    // outros métodos...
}
```

O método `getCarrosPermitidosPorUsuario()` retornaria apenas os carros necessários para determinados tipos de usuário. Note que o ManagedBean `CarroMB` precisa de um usuário para realizar a busca dos carros. E de onde viria esse objeto usuário? Esse objeto poderia ser buscado na sessão, mas o caminho até chegar a ela é um tanto quanto longo.

```
FacesContext context = FacesContext.getCurrentInstance();
ExternalContext externalContext = context.getExternalContext();
HttpServletRequest request =
    (HttpServletRequest) externalContext.getRequest();
HttpSession session = request.getSession();
Usuario usuario =
    (Usuario) session.getAttribute("USUARIO_LOGADO");
```

Para conseguir acessar o `HttpSession`, chamamos 5 métodos em um trecho de código muito extenso e que fica difícil de ler. Uma prática comum é de ter um ManagedBean que facilite o acesso aos dados do usuário logado. Por exemplo, para acessar o usuário da sessão, bastaria fazer `usuarioMB.getUsuario()`.

Existe um modo de acessar um ManagedBean buscando o valor da sessão, que é um código tão verboso quanto o já mostrado anteriormente:

```
// código reduzido
userMB = (UsuarioMB) externalContext.getSessionMap()
    .get("UsuarioMB");
userMB.getUsuario();
```

É possível extrair um ManagedBean diretamente do contexto do JSF, mas ainda assim é um código que ficará longo. Uma abordagem mais simples seria através da injeção do ManagedBean como dependência.

```
@ManagedBean
public class CarroMB{

    @ManagedProperty(value = "#{usuarioMB}")
    private UsuarioMB usuarioMB;

    public void setUsuarioMB(UsuarioMB usuarioMB) {
        this.usuarioMB = usuarioMB;
    }

    public List<Carro> getCarrosPorUsuario(){
        List<Carro> carros = new ArrayList<Carro>();
```

```
    usuario = usuarioMB.getUsuario();
    adicionarCarrosSimples(carros, usuario);
    adicionarCarrosEconomicos(carros, usuario);
    adicionarCarrosLuxuosos(carros, usuario);

    return carros;
}
// outros métodos...
}
```

Para que a injeção funcione, alguns passos são necessários:

- Utilizar a anotação `@ManagedProperty` sobre o atributo do ManagedBean a ser injetado;
- O ManagedBean a ser injetado precisa de um método `set`.

Utilizar injeção de ManagedBean deixa o código mais simples de ler e entender. Outro detalhe sobre a injeção de ManagedBean é que só podemos injetar um ManagedBean dentro de outro com mesmo escopo, ou um escopo mais abrangente. Por exemplo, é possível injetar um `SessionScoped` dentro de um `RequestScoped`, mas não é possível fazer o inverso.

CAPÍTULO 13

TARGET UNREACHABLE: ENFRENTE A NULLPOINTEREXCEPTION DO JSF

Como desenvolvedores Java, estamos acostumados a nos deparar com as famosas `NullPointerException`. Também não estaremos livres disso ao trabalhar com JSF; portanto, é preciso aprender a lidar com elas, na forma como podem acontecer no JSF. Considere o simples ManagedBean a seguir:

```
@ManagedBean
public class CidadeMB {
    private Cidade cidade;

    public Cidade getCidade() {
        return cidade;
    }

    // outros métodos
}
```

Caso o código `#{{cidadeMB.cidade.nome}}` seja executado, a seguinte mensagem de erro será exibida:

```
Target Unreachable, 'null' returned null
```

No exemplo do ManagedBean `CidadeMB`, é possível ver que o atributo `cidade` é retornado, mas em nenhum momento é instanciado. Esse erro pode ser facilmente contornado pelas abordagens do capítulo *Inicializando Objetos*.

```
@ManagedBean
public class CidadeMB {
    private Cidade cidade;

    public Cidade getCidade() {
        if(cidade == null){
            cidade = new Cidade();
        }

        return cidade;
    }

    // outros métodos
}
```

Note que, no código da classe `CidadeMB`, o método `getCidade()` agora contém um `if` para verificar se o atributo `cidade` está `null`. Se o código `# {cidadeMB.cidade.estado.nome}` for executado, o mesmo erro poderá acontecer. É necessário que todos os atributos não primitivos sejam inicializados antes de serem usados.

Um outro motivo para esse erro aparecer poderia ser utilizar a EL `# {cidadeMB.cidade.estado.nome}`, mas esquecer de anotar o ManagedBean com `@ManagedBean` ou `@Named`. Esse erro também pode acontecer por má utilização da EL. Caso a EL `# {cidade.nome}` seja escrita `#cidade.nome`, a mesma mensagem poderá aparecer.

Para finalizar, essa mensagem de erro pode vir descrita dos seguintes modos:

Target Unreachable, 'null' returned null

javax.el.PropertyNotFoundException: /pagina.xhtml @13,154
value="#{cidade.nome)": Target Unreachable, 'null' returned null

Alvo inalcançável, SEU_ATRIBUTO retornou null

CUIDADO COM O VALUE IS NOT VALID

Existem erros que acontecem por detalhes sutis, cujos motivos às vezes demoramos horas para descobrir. O erro `Value is not valid` é um deles, comumente encontrado quando usamos componentes do tipo `select`.

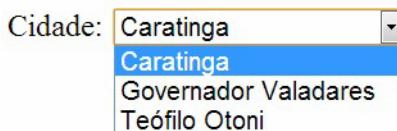


Figura 14.1: Value is not valid

Devido ao fato de que o `select` é utilizado em classes, o JSF tentará comparar os valores através dos métodos `hashCode` e `equals`. Esse erro pode acontecer caso sua classe não esteja implementando esses dois métodos corretamente.

O método `hashCode` deve sempre retornar um inteiro que represente numericamente a classe. Esse inteiro é comumente utilizado pela interface `Set` e outros componentes. O método `equals` deve sempre retornar se um objeto é igual ao outro, e o resultado não pode variar. Para melhores detalhes veja:

<http://tutorials.jenkov.com/java-collections/hashcode-equals.html>.

```
public class Cidade{  
    private int id;  
    private String nome;  
  
    // outros métodos  
  
    @Override  
    public int hashCode() {  
        int hashCode = 33;  
        hashCode = hashCode * 17 + id;  
        hashCode = hashCode * 31 + nome.hashCode();  
        return hashCode;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (obj instanceof Cidade) {  
            Cidade cidade = (Cidade) obj;  
            return cidade.id == id;  
        }  
        return false;  
    }  
}
```

A classe `Cidade` está descrita no formato adequado para ser utilizada em um componente `select`.

Front-ends JSF

Um dos maiores problemas de pessoas que usam o JSF começa pelos arquivos de páginas. Má utilização do JSF pode levar a problemas de arquitetura, a comportamentos estranhos e até mesmo a desenvolvedores achando que é falha do próprio JSF.

Escolher entre JSP ou `xhtml`, otimizar o uso do Facelets, saber quando utilizar `action` ou `actionListener` são assuntos que serão vistos nesta parte do livro.

CAPÍTULO 15

UTILIZAR JSP OU XHTML?

O JSP é um velho guerreiro e conhecido do mundo Java Web. Com o JSP , veio a grande utilização do Servlet para fazer a integração e o desacoplamento entre o código HTML e o código Java .

Nas primeiras versões do JSF, o JSP foi intensamente utilizado para exibir as informações para o usuário. Já com a versão 2.0 do JSF, apareceu o xhtml , com uma tecnologia chamada Facelets .

Note que as vantagens do JSP sobre o xhtml estão mais relacionadas com a questão de estudo e aprendizado. Muitas empresas adotam o JSP ainda pelo fato de ser a primeira tecnologia a ser utilizada para Java Web, e acaba sendo mais prático e barato continuar a usar a mesma tecnologia.

Conheço pessoas que, até hoje, são contra determinados frameworks. Por exemplo, algum conhecido que usou a versão 0.5 do JSF e não gostou espalhou isso para frente. Reconheço que, antes da versão 2.0, era muito difícil de se trabalhar com o JSF, mas poucos são os que não gostaram dessa nova versão.

Posso citar um caso em que uma pessoa me disse: "odeio Hibernate, pois tem muito XML". Eu respondi que atualmente poderia se fazer a aplicação toda com anotação, e a pessoa me disse

que ainda assim preferia nem olhar o Hibernate.

Com relação ao `xhtml`, quando utilizado com Facelets (que veremos em breve), é possível ressaltar as seguintes vantagens:

- XHTML com Facelets podem executar até 50% mais rápido que o JSP;
- Todas as tags utilizadas nas páginas são declaradas de modo dinâmico, não necessitam estar declaradas em um TLD (*Tag Library Descriptor*);
- Facelets permitem Templates de modo avançado, em que a reutilização de código é muito alta.

Na codificação das páginas, é possível ver como o código de um JSP é semelhante a uma página `xhtml`. Só é possível notar essa semelhança em página simples, uma vez que as páginas aumentam sua complexidade. E em número de componentes, o código ficará bastante distinto. Um JSP ficaria como:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
    <head>
        <title>Exemplo JSP</title>
    </head>
    <body>
        <h:form>
            <h:outputText value="Olá Mundo JSP!" />
        </h:form>
    </body>
</html>
```

Enquanto que o código equivalente no `xhtml` ficaria:

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

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
    <title>Exemplo XHTML</title>
</h:head>
<h:body>
    <h:form>
        <h:outputText value="Olá Mundo XHTML!" />
    </h:form>
</h:body>
</html>
```

CAPÍTULO 16

UTILIZANDO IMAGENS/CSS/JAVASCRIPT DE MODOS SIMPLES

Quando precisamos trabalhar com imagens em nossas aplicações, uma abordagem muito utilizada é passar o caminho completo de um arquivo.

```
<!-- código que podem ou não exibir a imagem -->
<h:graphicImage value="../imagens/minhoca.jpg" />
<h:graphicImage
    value="#{facesContext.getExternalContext.requestContextPath}
        /WebContent/imagens/minhoca.jpg" />
```

E a mesma prática também ocorre quando falamos de CSS e JavaScript.

```
<!-- código que podem importar o css/javascript -->
<link rel="stylesheet" type="text/css"
    href="../meu_css/style.css">
<script type="text/javascript"
    src="../meu_javascript/script.js" />
```

Veja que o caminho físico foi passado para a página poder exibir uma imagem, ou utilizar os recursos do CSS e do JavaScript. O problema é que basta mudar a página de lugar que os códigos já vistos param de funcionar. É muito complicado refatorar os

diretórios das páginas, uma vez que existe uma dependência tão forte entre o recurso utilizado.

Para resolver esse problema, o JSF 2 tem o recurso chamado *Libraries* (bibliotecas). Ele tratará cada tipo de recurso como uma biblioteca. Ou seja, na aplicação pode existir uma ou mais bibliotecas para imagens, CSS e JavaScript.

Para utilizar os recursos de nossa aplicação, basta criar uma pasta chamada `resources`, que ficará na raiz do projeto. Esta conterá a estrutura de pastas e arquivos necessários para serem usados como bibliotecas. A figura a seguir mostra como deve ficar essa estrutura.

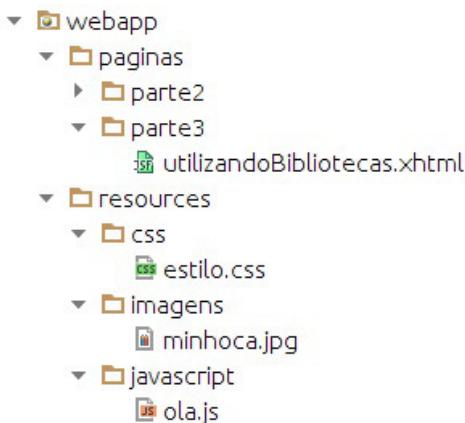


Figura 16.1: Estrutura das pastas

Note que outras 3 pastas existem abaixo de `resources` : `css` , `imagens` e `javascript` . Cada diretório que for adicionado abaixo da pasta `resources` será tratado como uma biblioteca. A vantagem dessa abordagem é a facilidade para acessar os recursos.

```
<!-- Código para importar os arquivos -->
```

```
<h:outputStylesheet library="css" name="estilo.css" />
<h:outputScript library="javascript" name="ola.js" />

<!-- Código para utilizar os recursos -->
<h:graphicImage library="imagens" name="minhoca.jpg" />
```

Para adicionar o JavaScript, o CSS e utilizar uma imagem, basta apontar para a biblioteca e para o nome do recurso. Ao usar essa abordagem, é possível mudar as estruturas das páginas, que não haverá problemas para localizar as bibliotecas.

Note que, no trecho do código que mostra como utilizar os valores, em nenhum momento foi utilizado o caminho físico para se chegar aos arquivos.

CAPÍTULO 17

BOA UTILIZAÇÃO DO FACELETS

Incorporado oficialmente à versão 2 do JSF, uma das vantagens do *Facelets* é a possibilidade de se utilizar um esquema de *Template* nas páginas da aplicação. Veja na figura a seguir como ficará a estrutura das páginas usadas no código-fonte deste livro.

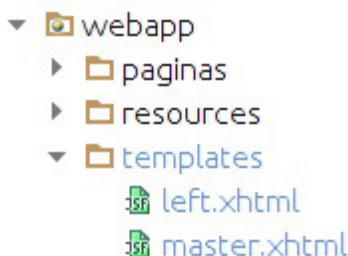


Figura 17.1: Estrutura das pastas

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

<h:head>
    <h:outputStylesheet library="css" name="estilo.css" />
    <h:outputScript library="javascript" name="ola.js" />
</h:head>
```

```

<h:body>
    <f:view>
        <div id="divLeft">
            <ui:insert name="divLeft">
                <ui:include src="left.xhtml" />
            </ui:insert>
        </div>

        <div id="divMain">
            <ui:insert name="divMain" />
        </div>
    </f:view>
</h:body>
</html>

```

Para utilizar uma página que servirá como *Template*, basta fazer como no código da página `master.xhtml`. Esse layout é simples, apenas com um menu lateral e uma área principal para exibir as informações. Sobre o código da página `master.xhtml`, é possível dizer:

- `<ui:insert name="divLeft"></ui:insert>` define uma área que será substituída. Note que essa área pode conter um valor padrão ou estar em branco.
- `<ui:include src="left.xhtml" />` faz a inclusão de uma página.

A página do menu é uma ainda mais simples, apenas com botões que serão inseridos em todas as páginas que usarem o *Template*.

```

<!-- left.xhtml -->
<h:body>
    <ui:composition>
        <h:form>
            <p:button outcome="/applicationScoped.xhtml"
                       value="@ApplicationScoped" />
            <p:button outcome="/conversationScoped.xhtml"
                       value="@ConversationScoped" />

```

```

<p:button outcome="/dependentScoped.xhtml"
           value="@DependentScoped" />
</h:form>
</ui:composition>
</h:body>

```

Note que o código da página left.xhtml não tem conhecimento da página de *Template*. É possível encontrar a tag <ui:composition> que define que o código a ser inserido se encontra ali dentro.

DICA

É considerada boa prática deixar todo o código que será inserido via *Template* envolvido pela tag <ui:composition>. A vantagem disso é que todo código que estiver fora do <ui:composition> será ignorado pelo *view handler* do JSF. Com essa prática, é possível adicionar código no <h:head> para que se possa visualizar a página sem que seja necessário compilar o projeto.

Lembra da figura da minhoca exibida no capítulo *Utilizando imagens/css/javascript de modos simples?* O código da página está usando a funcionalidade de *Template* do JSF 2.0.

```

<!-- Página que exibe a minhoca utilizandoBibliotecas.xhtml -->
<h:body>
<ui:composition template="/templates/master.xhtml">
    <ui:define name="divMain">
        <h:button value="Ola via JavaScript"
                  onclick="ola();"/>
        <br />
        <p class="titulo">#{mensagens.bibliotecasTexto}</p>
        <h:graphicImage library="imagens"

```

```
        name="minhocinha.jpg" />
    </ui:define>
</ui:composition>
</h:body>
```

Veja como é simples e objetivo o código da página `utilizandoBibliotecas.xhtml`. Pela tag `ui:composition`, é indicado qual template será utilizado. E com a tag `ui:define`, indicamos qual parte do código vamos sobrescrever.

A vantagem de utilizar Facelets como *Template* é que é possível mudar toda a estrutura de uma aplicação sem afetar todas as páginas. Note que a página `utilizandoBibliotecas.xhtml` não tem a mínima ideia de quantos menus existem na aplicação, ou se há duas ou três barras laterais no layout.

Caso uma reestruturação geral fosse necessária, com Facelets, o impacto aconteceria apenas nas páginas de *Template*, e as páginas que exibem as informações não seriam modificadas.

É preciso ter muito cuidado para não aninhar `form`s HTML. Para aninhar um `form`, bastaria fazer `<h:form><h:form></h:form></h:form>`. O problema de aninhar `form` é que cada browser poderá ter comportamentos inesperados, como se perder ao dar foco a um componente, tab e outros.

```
<!-- aninhar.xhtml -->
<ui:composition template="/templates/master.xhtml">
    <h:form>
        <ui:define name="divMain"><br/>
            <p>Alguma coisa</p>
        </ui:define>
        <ui:define name="divRight"/>
    </h:form>
</ui:composition>
```

Veja no código da página `aninar.xhtml` que, ao substituir o `ui:define` em alguma, com algum código que contenha o componente `h:form`, os `form`s serão aninhados.

É preciso estar atento para não deixar que essa situação de `form` aninhado aconteça na aplicação. Um problema já presenciado em fórum era que a `action` de um botão não era chamada, pois o botão estava dentro de um `form` aninhado.

Outra facilidade do Facelets é colocar no Template os componentes que exibirão as mensagens para o usuário. Pode ser um `h:messages` ou um `p:growl` do Primefaces, por exemplo. Desse modo, não será necessário declarar esse componente em todas as páginas. Veja no código da página `reutilizandoComponenteMessage.xhtml` como seria possível utilizar um único componente de mensagens para toda a aplicação:

```
<!-- reutilizandoComponenteMessage.xhtml -->
<ui:composition template="/templates/master.xhtml">
    <h:messages />
    <ui:define name="divMain">
        <p>Alguma coisa</p>
    </ui:define>
</ui:composition>
```

Cuidado ao ter de sobrescrever diversas regiões nas páginas apenas para apagar um texto que não deveria ser exibido. Seria possível ter um código do tipo:

```
<!-- codigo_feio.xhtml -->
<ui:composition template="/templates/master.xhtml">
    <ui:define name="divLeft01"/>
    <ui:define name="divLeft02"/>
    <ui:define name="divMain"><br/><br/><br/>
        <p>Alguma coisa</p>
    </ui:define>
    <ui:define name="divRight"/>
```

```
</ui:composition>
```

Veja o exemplo da página `codigo_feio.xhtml`. Nela são sobreescritas diversas áreas para não exibirem determinado valor padrão. Em uma página ou outra, não seria tanto incômodo, mas caso aconteça em diversas páginas, existe um modo de evitar esse problema: basta criar um outro *Template*. É possível ter mais que um *Template* em um projeto, de modo que ficaria organizado se seu projeto tivesse 33% das páginas em um, e outros 67% em outro *Template*.

CAPÍTULO 18

ENVIAR VALORES PARA O MANAGEDBEAN

Muitas vezes precisamos transferir dados que estão sendo exibidos em uma tabela por meio de um `datatable`, como por exemplo, o `id` de uma informação que queremos editar. É possível enviar valores de um `datatable` para um `ManagedBean` de diversas maneiras.

Para enviar os objetos, como demonstrado a seguir, basta colocar um botão em uma coluna:

```
<!-- dataTable.xhtml -->
<h:form>
    <p:datatable value="#{cidadeMB.cidades}" var="cidade" >
        <p:column>
            <f:facet name="header">
                #{mensagens.cidadeNome}
            </f:facet>
            <h:outputText value="#{cidade.nome}" />
        </p:column>
        <p:column>
            <!-- BOTÃO AQUI -->
        </p:column>
    </p:datatable>
</h:form>
```

O código da página `dataTable.xhtml` mostra onde deve ficar o botão para enviar um valor para o `ManagedBean`. Ele deve estar

dentro de um `h:form` para funcionar corretamente.

A seguir, veremos modos de enviar o valor o ManagedBean mostrando apenas o código do botão. Serão mostrados aqui três modos para enviar um valor: utilizando `f:setPropertyActionListener` por parâmetro e por binding.

18.1 ENVIE VALOR COMO PARÂMETRO PELO F:SETPROPERTYACTIONLISTENER

Esse é o modo de enviar o parâmetro que necessita de menos métodos em comparação com os outros. Ele precisa apenas da tag `f:setPropertyActionListener` para informar ao ManagedBean que uma linha do `dataTable` foi selecionada.

```
<p:commandButton value="#{mensagens.enviarValorSetProperty}"
    onclick="exibirCidadeWidget.show()"
    update=":exibirCidadeGrid">
    <f:setPropertyActionListener value="#{cidade}"
        target="#{cidadeMB.cidade}" />
</p:commandButton>

public String atualizarCidade(){
    cidadeDAO.atualizar(cidade);
    return "cidade.xhtml";
}
```

A tag `f:setPropertyActionListener` tem como atributos `value` e `target`. O `value` indica qual o objeto que será enviado ao ManagedBean; `target` indica qual o ManagedBean receberá o objeto.

Lembre-se de que o escopo do ManagedBean faz diferença. Caso você use o `RequestScoped`, o valor poderá se perder. É necessário entender como funciona cada escopo e utilizar o tipo

ideal para cada caso. Para um ManagedBean, o ideal seria `ViewScoped` (veja capítulo *Entenda o novo @ViewScoped*).

Para que o objeto seja corretamente enviado ao ManagedBean, é necessário ter métodos *getters* e *setters* implementados para o atributo.

18.2 ENVIE VALOR COMO PARÂMETRO

Outro modo simples de se enviar um atributo é por método. Esse método receberá o objeto diretamente como parâmetro através da página.

```
<p:commandButton value="Enviar"
    actionListener="#{cidadeMB.atualizarCidade(cidade)}" />

public String atualizarCidade(Cidade cidade){
    cidadeDAO.atualizar(cidade);
    return "cidade.xhtml";
}
```

É necessário utilizar a versão 3.0 do Servlet e a versão 2.2 de EL para habilitar esse modo de envio de valor. A versão do EL vem do jar utilizado. A versão do Servlet é definida no `web.xml`.

```
<web-app version="3.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
```

18.3 ENVIE VALOR POR BINDING

Existe uma técnica chamada `binding`, que nada mais é que ligar um componente visual diretamente a um objeto no ManagedBean.

```

<p:dataTable value="#{cidadeMB.cidades}" var="cidade"
    binding="#{cidadeMB.dataTable}" rowKey="#{cidade.id}" >
    <!-- outras colunas -->
    <p:column>
        <p:commandButton value="Enviar"
            actionListener="#{cidadeMB.selecionarCidade}" />
    </p:column>
</p:dataTable>

private DataTable dataTable;

public String atualizarCidade(){
    Integer rowIndex = dataTable.getRowIndex() + 1;
    cidade = (Cidade) dataTable.getRowData(rowIndex.toString());
    cidadeDAO.atualizar(cidade);
    return "cidade.xhtml";
}

```

Para utilizar o `binding`, basta indicar na página qual objeto do tipo `DataTable` estará ligado ao ManagedBean. E para usar no ManagedBean, basta pegar o index da linha que foi selecionada e depois buscar o valor dentro da tabela pelo método `getRowData`. O ruim dessa abordagem é que cria uma acoplamento muito forte entre a página e o ManagedBean, tornando mais difícil a manutenção do código.

TEMAS DINÂMICOS

É sempre bom pensar em agradar o usuário, não apenas com maravilhosas funcionalidades, mas também com um visual ao mesmo tempo bonito e funcional em se tratando de cores, imagens de fundo, tipografia etc.

Uma prática interessante poderia ser possuir um conjunto de temas pré-definidos, no qual o usuário pudesse selecionar sua opção preferida. Um ManagedBean de sessão poderia conter o valor default, ou carregar algum valor padrão do banco de dados.

```
@ManagedBean  
@SessionScoped  
public class UsuarioMB {  
    private String userCSS = "estilo.css";  
    private List<String> cssDisponivel;  
  
    @PostConstruct  
    private void init(){  
        cssDisponivel = new ArrayList<String>();  
        cssDisponivel.add("estilo.css");  
        cssDisponivel.add("estilo2.css");  
    }  
  
    public String alterarEstilo(){  
        return null;  
    }  
    // getters e setters omitidos  
}
```

O método `alterarEstilo()` apenas retorna `null`, mantendo o usuário na mesma página. E basta permitir que o usuário escolha o estilo que será utilizado, por exemplo, através de um `selectOneMenu`:

```
<!-- utilizando o estilo dinâmico -->
<h:outputStylesheet library="css" name="#{usuarioMB.userCSS}" />

<!-- utilizando action que mantém usuário na mesma página -->
<p><h:outputText value="#{mensagens.troqueTema}:" /></p>
<p:selectOneMenu value="#{usuarioMB.userCSS}">
<f:selectItems value="#{usuarioMB.cssDisponivel}" />
</p:selectOneMenu>
<p:commandButton value="#{mensagens.troqueTemaTrocar}"
    action="#{usuarioMB.alterarEstilo()}" ajax="false" />
```

Para utilizar o estilo dinâmico nas páginas, basta apontar a origem do arquivo CSS para a saída do ManagedBean

```
<h:outputStylesheet      library="css"      name="#
{usuarioMB.userCSS}" />
```

 e um método que retorne o estilo do usuário.

MANTENHA A OPÇÃO ESCOLHIDA PELO USUÁRIO

Apesar do exemplo desse capítulo utilizar todos os valores em memória, é possível salvar essa configuração no banco de dados. Quando o usuário entrar na aplicação, a classe `UsuarioMB` poderia simplesmente acessar um objeto da sessão e buscar o valor do CSS padrão dele.

CAPÍTULO 20

O QUE EU USO? ACTION OU ACTIONLISTENER?

Um dos maiores dilemas durante o desenvolvimento em JSF é o momento em que devemos utilizar `action` ou `actionListener`. O JSF tem dois modos de tratar as ações do usuário.

O primeiro é possível definir como: "esse click vai executar uma ação e essa ação decidirá se o usuário permanece ou muda de tela". Enquanto o segundo podemos definir como: "esse click realizará alguma ação, mas o usuário permanecerá na mesma tela ou será forçadamente redirecionado".

20.1 REDIRECIONE O USUÁRIO COM ACTION

```
<h:commandButton value="Click aqui!"  
action="#{managedBean.metodo}" />  
  
public String metodo(){  
    // faz alguma coisa  
    return "nova_pagina";  
}
```

RETORNO SEM EXTENSÃO

O código que mostra um exemplo de navegação `return "nova_pagina";` não adiciona a extensão da página. O JSF procurará por uma página com a mesma extensão da URL chamada. Se o usuário que chamou o método estiver na página `http://site/index.jsf`, o JSF procurará uma página chamada `nova_pagina.jsf`. Ao encontrar uma página chamada `nova_pagina.xhtml`, ele entenderá que o destino final é esse arquivo.

O que melhor caracteriza uma `action` é um método que retornará um objeto do tipo `String`. A `action` é melhor utilizada em um método que definirá o destino do usuário. Dessa forma, ele pode permanecer na tela que estava ou mudar para uma nova tela.

Considere que o usuário esteja na tela de cadastro, e uma vez finalizado, ele será redirecionado para a tela de login. É possível retornar `null` e fazer com que o usuário permaneça na mesma tela caso algum erro aconteça.

```
public String metodo(){
    // mantém o usuário na mesma tela com todos os dados
    return null;
}
```

20.2 TRATE EVENTOS COM ACTIONLISTENER

```
<!-- como utilizar uma actionListener -->
<h:commandButton value="Click aqui!"  
actionListener="#{managedBean.metodo}" />  
  
public void metodo(javax.faces.event.ActionEvent event){  
    // faz alguma coisa  
}
```

Uma das características do `actionListener` é que o usuário sempre permanecerá na mesma tela. Esse tipo de método é muito usado em chamada *Ajax* e em componentes próprios. Um `actionListener` recebe como parâmetro um `javax.faces.event.ActionEvent`, com o qual você tem acesso, por exemplo, ao componente que iniciou a ação. Uma boa analogia é você associar o `actionListener` com o tratamento de um evento de um clique, de uma ação.

`ActionListener` naturalmente não realizará uma navegação, mesmo retornando uma String como feito em uma *Action*. Para realizar uma navegação a partir de um método `ActionListener`, será necessário utilizar o método `sendRedirect`. Para executar um `sendRedirect`, basta fazer como no código exibido:

```
FacesContext.getCurrentInstance()  
.getExternalContext()  
.redirect("/site.jsf");
```

Note que, apesar de o método se chamar `redirect`, ele é comumente chamado de `sendRedirect`, utilizado pela classe `HttpServletResponse`. `sendRedirect`.

Caso apareça alguma mensagem de erro ao executar, faça como a seguir:

```
response.sendRedirect();  
FacesContext.getCurrentInstance().responseComplete();
```

O método `responseComplete()` implicitamente informa ao JSF que não será necessário trabalhar o `response`.

Aproveite as bibliotecas de componentes

Quando estamos desenvolvendo uma aplicação, muitas vezes precisamos que os elementos que envolvem a nossa tela tenham características diferentes das que elas possuem por padrão. Por exemplo, gostaríamos de ter um campo de texto onde fosse possível adicionar uma máscara. Ou então, um `dataTable` que fizesse ordenação dos dados dentro dele, e assim por diante. Nesse caso, precisamos de componentes diferentes. Podemos encontrar esses componentes em bibliotecas, desenvolvidas e disponíveis em variadas formas na comunidade.

Bibliotecas de componentes são focadas mais na parte de funcionalidades, em geral, criando `dialogs` , `dataTable` , `galerias` etc. Entre as principais, se destacam o Primefaces, Richfaces, Icefaces e OmniFaces. Aliás, é possível que qualquer pessoa possa criar sua biblioteca de componente e disponibilizá-la para outros desenvolvedores, ou reutilizá-la em diferentes projetos.

Nesta parte do livro, veremos sobre algumas características do Primefaces, Richfaces, Icefaces e OmniFaces. Vantagens, desvantagens e exemplos de código de funcionalidades ricas e interessantes que conseguimos fazer com essas bibliotecas.

Os exemplos demonstrados aqui não são para comparar uma biblioteca com a outra. Seus exemplos foram escolhidos apenas aleatoriamente. Há componentes que existem em todas as bibliotecas, mas outros apenas em uma.

DICA

Nos códigos apresentados no livro, foram utilizados diversos nomes de métodos, atributos e classes em português, justamente para mostrar que o nome não influencia no comportamento da ação. Alguns tutoriais da internet usam de nomes estranhos e, às vezes, abusivos por acharem que é só com aquele nome que tudo se resolve.

PRIMEFACES

É considerado por muitos o mais avançado do mercado. Ele tem diversas funcionalidades prontas que facilitam e muito a vida do desenvolvedor. Possui mais de 130 componentes em seu showcase com código que explica como utilizar.

21.1 DATATABLE COM SELEÇÃO COM UM CLICK

Um `DataTable` pode ter diversas configurações e ajustes para melhorar a usabilidade do usuário. Um modo de utilizar o `DataTable` do Primefaces bem interessante é permitir que, com apenas um clique, ele exiba o valor da linha selecionada.

(1 of 100)		<=>	<<	1	2	3	4	5	6	7	8	9	10	>>	>><	10
Id		Nome										Estado				
1	CIDADE_1											ESTADO_1				
2	CIDADE_2											ESTADO_2				
3	CIDADE_3											ESTADO_3				
4	CIDADE_4											ESTADO_4				
5	CIDADE_5											ESTADO_5				
6	CIDADE_6											ESTADO_6				
7	CIDADE_7											ESTADO_7				
8	CIDADE_8											ESTADO_8				
9	CIDADE_9											ESTADO_9				
10	CIDADE_10											ESTADO_10				

Figura 21.1: DataTable Primefaces seleção em um clique

(1 of 100)		<=>	<<	1	2	3	4	5	6	7	8	9	10	>>	>><	10
Id		Nome										Estado				
1	CIDADE_1											ESTADO_1				
2	CIDADE_2											ESTADO_2				
3	CIDADE_3											ESTADO_3				
4	CIDADE_4											ESTADO_4				
5	CIDADE_5											ESTADO_5				
6	CIDADE_6											ESTADO_6	Cidade			
7	CIDADE_7											ESTADO_7	Id 5			
8	CIDADE_8											ESTADO_8	Nome CIDADE_5			
9	CIDADE_9											ESTADO_9	Estado ESTADO_5			
10	CIDADE_10											ESTADO_10				

Figura 21.2: DataTable Primefaces seleção em um clique

Pelas imagens, é possível ver como é interessante a abordagem de um clique para exibir os dados da linha selecionada. A ação do

clique poderia ser qualquer ação como excluir, alterar etc.

E como ficaria o código? Veja em seguida como é simples.

```
<!-- dataTable.xhtml -->
<h:form>
    <p:dataTable value="#{primefacesMB.cidades}"
        var="cidade"
        selection="#{primefacesMB.cidade}"
        selectionMode="single"
        paginator="true"
        rows="10"
        paginatorTemplate="{CurrentPageReport}
            {FirstPageLink}
            {PreviousPageLink} {PageLinks}
            {NextPageLink} {LastPageLink}
            {RowsPerPageDropdown}"
        rowsPerPageTemplate="5,10,15"
        style="width: 40%"
        lazy="true">
        <p:ajax event="rowSelect"
            update=":cidadeDialogForm"
            onComplete="cidadeWidget.show()"/>
        <p:column>
            <f:facet name="header">
                #{mensagens.cidadeId}
            </f:facet>
            <h:outputText value="#{cidade.id}" />
        </p:column>
        <p:column>
            <f:facet name="header">
                #{mensagens.cidadeNome}
            </f:facet>
            <h:outputText value="#{cidade.nome}" />
        </p:column>
        <p:column>
            <f:facet name="header">
                #{mensagens.cidadeEstado}
            </f:facet>
            <h:outputText value="#{cidade.estado}" />
        </p:column>
    </p:dataTable>
</h:form>
```

Veja que, no `dataTable.xhtml`, um código fácil de entender e de trabalhar, o atributo `selection="#{primefacesMB.cidade}"` aponta onde ficará armazenada a linha selecionada. O atributo `selectionMode="single"` é usado para definir os tipos de seleção possível. O outro pedaço de código relativo ao clique único é a chamada Ajax para mostrar o valor em um `dialog`.

```
<p:ajax event="rowSelect"
        update=":cidadeDialogForm"
        oncomplete="cidadeWidget.show()"/>
```

A tag `p:ajax` tem o atributo `event` que informa qual ação deve ser executada. O `update` realizará atualização de um componente. O `oncomplete="cidadeWidget.show()"` exibirá o `dialog` de cidades.

```
<!-- dialog.xhtml -->
<p:dialog widgetVar="cidadeWidget"
           modal="true"
           header="#{mensagens.cidade}>
    <h:form id="cidadeDialogForm">
        <h:panelGrid columns="2">
            <h:outputText value="#{mensagens.cidadeId}"/>
            <h:outputText value="#{primefacesMB.cidade.id}"/>
            <h:outputText value="#{mensagens.cidadeNome}"/>
            <h:outputText value="#{primefacesMB.cidade.nome}"/>
            <h:outputText value="#{mensagens.cidadeEstado}"/>
            <h:outputText value="#{primefacesMB.cidade.estado}"/>
        </h:panelGrid>
    </h:form>
</p:dialog>
```

No código da página `dialog.xhtml`, é possível ver como utilizar uma `dialog` para exibir o valor. A propriedade `widgetVar` é uma forma de facilitar a utilização de jQuery/JavaScript dentro de um `xhtml`. A chamada *Ajax* executa o comando do `widgetVar` assim: `cidadeWidget.show()`. O

ManagedBean tem apenas `get/set` do atributo: `List<CIDADE> cidades`.

21.2 DRAG AND DROP

Uma função que agrada muito aos usuários de aplicações é o chamado *drag and drop*, ou seja, arrastar objetos de um lado e largá-los em outro.



Figura 21.3: Drag and drop — 1



Figura 21.4: Drag and drop — 2

Escolha fotos da Minhocá

Aniversariante



Roqueira



Fotos Escolhidas

Anos 60



Figura 21.5: Drag and drop — 3

Essas figuras mostram que é uma funcionalidade muito interessante que atualmente é fácil de encontrar em diversos programas. O código é composto de dois componentes

`p:fieldset` é um outro chamado `p:droppable`.

```
<p:fieldset legend="#{mensagens.primefacesEscolhaFotoMinhoca}">
    <p:dataGrid id="fotosDisponiveis"
        var="foto"
        value="#{primefacesMB.fotos}"
        columns="3">
        <p:column>
            <p:panel id="panelFotos"
                header="#{foto.nome}"
                style="text-align:center">
                <h:panelGrid columns="1"
                    style="width:100%">
                    <p:graphicImage value="#{foto.path}" />
                </h:panelGrid>
            </p:panel>

            <p:draggable for="panelFotos"
                revert="true"
                handle=".ui-panel-titlebar"
                stack=".ui-panel"/>
        </p:column>
    </p:dataGrid>
</p:fieldset>
```

O primeiro `p:fieldset` é o que exibe os elementos disponíveis para serem arrastados. Dentro dele, é possível encontrar um `dataTable`. A lista utilizada no `dataTable` é `List<Foto> fotos`, onde a classe `foto` tem apenas duas `String`s: a primeira `String` é o nome da foto, e a segunda, o caminho físico da foto (por exemplo, `/home/uaihebert/fotos`).

O componente `p:draggable` é quem configura qual objeto da tela poderá ser arrastado. Ele tem o atributo `for`, que aponta quem poderá ser arrastado; o parâmetro `revert`, que indica que, caso o objeto seja largado em um lugar inválido, ele deve voltar ao seu lugar de origem; os parâmetros `handle`, que estão falando qual classe servirá de ativador para se mover o objeto, e no caso foi

escolhido o título do panel — o usuário não conseguirá mover o objeto a não ser pelo título; e stack , que controla automaticamente o "arrastar" do componente.

```
<p:fieldset id="fotosSelecionadas"
    legend="#{mensagens.primefacesFotosSelecionada}"
    columns="3">
    <p:outputPanel id="fotosDespejadas">
        <h:outputText value="#{mensagens.primefacesArrasteAqui}"
            rendered="#{empty primefacesMB.fotosSelecionadas}"
            style="font-size:24px;"/>
        <p:dataGrid var="foto"
            value="#{primefacesMB.fotosSelecionadas}"
            rendered="#{not empty
                primefacesMB.fotosSelecionadas}">
            <p:column>
                <p:panel id="panelFotos"
                    header="#{foto.nome}">
                    <p:graphicImage value="#{foto.path}"/>
                </p:panel>
            </p:column>
        </p:dataGrid>
    </p:outputPanel>
</p:fieldset>
```

O segundo p:fieldset nada mais é do que o responsável por exibir os dados das fotos que foram arrastadas. Logo abaixo do p:fieldset , virá o último componente que definirá onde um objeto poderá ser jogado.

```
<p:droppable for="fotosSelecionadas"
    tolerance="touch"
    activeStyleClass="ui-state-highlight"
    datasource="fotosDisponiveis"
    onDrop="handleDrop">
    <p:ajax listener="#{primefacesMB.fotoDespejada}"
        update="fotosDespejadas fotosDisponiveis"/>
</p:droppable>
```

O componente p:droppable define qual objeto receberá os

objetos jogados. O `for` aponta qual componente segurará o objeto jogado; `tolerance` aponta qual o tipo de ação para que ele considere que o objeto foi jogado; e `activeStyleClass` define qual estilo o componente que receberá um objeto terá enquanto ele não for jogado (ver na figura *Drag and drop — 2*).

O `datasource` é o componente que contém os objetos que poderão ser arrastados, e `onDrop` é ação que será executada (que veremos em breve). Existe também um listener que realiza a transferência da foto de uma lista (`#{primefacesMB.fotos}`) para outra(`#{primefacesMB.fotosSelecionadas}`).

```
// método executado pelo listener
public void fotoDespejada(DragDropEvent event) {
    Foto foto = (Foto) event.getData();

    getFotos().remove(foto);
    getFotosSelecionadas().add(foto);
}

function handleDrop(event, ui) {
    var droppedCar = ui.draggable;

    droppedCar.fadeOut('fast');
}
```

Efeito *jQuery* disparado pelo evento `onDrop` definido no componente `droppable`

21.3 NOTIFICADOR

É normal enviar uma resposta para usuário indicando se a ação foi realizada ou não. "Salvo com sucesso", "Erro ao alterar" ou "Você tem certeza? Sério mesmo?" são mensagens que são exibidas para os usuários de aplicações.

Veja nas figuras a seguir um interessante recurso chamado Growl.



Figura 21.6: Drag and drop



Figura 21.7: Drag and drop

Para utilizar o Growl, basta usar um código bastante simples e objetivo:

```
<!-- growl.xhtml -->
<h:form>
    <p:growl id="mensagens"/>
    <h:outputText value="#{mensagens.mensagemSucesso}"/>
    <p:commandButton value="#{mensagens.enviar}"
        actionListener="#{primefacesMB.mensagemSucesso}"
        update="mensagens"/>
    <h:outputText value="#{mensagens.mensagemErro}: "/>
    <p:commandButton value="#{mensagens.enviar}"
        actionListener="#{primefacesMB.mensagemErro}"
        update="mensagens"/>
</h:form>
```

O código da página growl.xhtml tem um componente

`p:growl` , responsável por mostrar a mensagem ao usuário. Seu comportamento é exatamente igual ao componente `h:messages` do JSF. Os botões exibidos nas páginas nada mais fazem do que uma chamada *Ajax* que cria a mensagem a ser exibida. Ao final da chamada *Ajax*, os botões estão configurados para atualizarem o `growl` pelo atributo `update="mensagens"` .

Para fazer com que uma mensagem seja exibida para um usuário, basta fazer como o código a seguir:

```
private final String SUCESSO = "Operação realizada com sucesso";
private final String ERRO = "Ops! Ocorreu um erro inesperado";
public void mensagemSucesso(ActionEvent event) {
    FacesContext instance = FacesContext.getCurrentInstance();
    instance.addMessage(null,
        new FacesMessage(FacesMessage.SEVERITY_INFO,
            SUCESSO,
            SUCESSO)
    );
}

public void mensagemErro(ActionEvent event) {
    FacesContext instance = FacesContext.getCurrentInstance();
    instance.addMessage(null,
        new FacesMessage(FacesMessage.SEVERITY_ERROR,
            ERRO,
            ERRO)
    );
}
```

21.4 AUTOCOMPLETE

O `autoComplete` , como diz o nome, completa uma informação que o usuário está digitando em um `input` . Assim como no Google, é possível ter esse mesmo comportamento com o Primefaces. Veja nas figuras seguintes como fica esse componente.

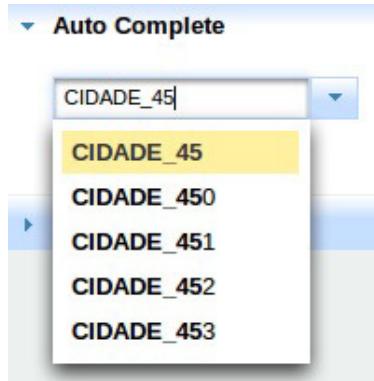


Figura 21.8: AutoComplete

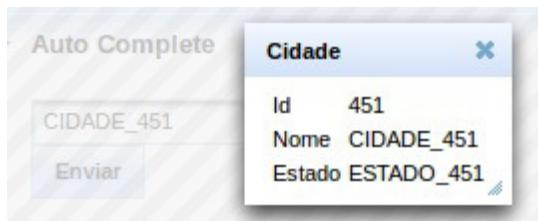


Figura 21.9: AutoComplete

Esse componente é altamente configurável e deve ser usado com cautela.

```
<!-- autoComplete.xhtml -->
<p:autoComplete id="autoComplete"
    forceSelection="true"
    minQueryLength="3"
    value="#{primefacesMB.cidade}"
    completeMethod="#{primefacesMB.autoComplete}"
    var="cidade"
    itemLabel="#{cidade.nome}"
    itemValue="#{cidade}"
    label="#{mensagens.cidade}"
    dropdown="true"
    required="true"
    queryDelay="3000"/>
```

```
<br/>
<p:commandButton value="#{mensagens.enviar}"
    update=":cidadeDialogForm"
    oncomplete="cidadeWidget.show()"/>
```

Note no código da página autoComplete.xhtml a quantidade de configurações (a maioria não obrigatória) que é possível ter, e existem outras que não foram utilizadas. O forceSelection indica se a seleção de algum valor é obrigatória, para evitar que o usuário deixe um valor qualquer, como uma cidade chamada 123***. O minQueryLength é a quantidade mínima de caracteres necessários para disparar a chamada no ManagedBean.

O value terá o valor selecionado pelo usuário. completeMethod="#{primefacesMB.autoComplete}" é o método que retornará a lista filtrada pelo valor informado pelo usuário. O var é o nome que um item da lista retornada no método definido no completeMethod retornará; itemLabel é o nome que será exibido ao usuário; e itemValue o valor selecionado que será enviado ao ManagedBean. O dropdown="true" exibe ou não o botão para exibir os valores.

O queryDelay é um contador para disparar a chamada Ajax. Imagine que o usuário digitou ABC , que é o tamanho mínimo de uma pesquisa definido em minQueryLength . Ao atingir 3 caracteres, o Primefaces iniciará um contador com um tempo definido em milissegundos na opção queryDelay . Caso o usuário edite o texto, o tempo de espera será reiniciado. A chamada Ajax somente será disparada quando o tempo de espera for zerado.

```
// metodoAutoComplete.java
public List<Cidade> autoComplete(String valorPesquisado) {
    final int maximoResultadosExibidos = 5;

    if (valorPesquisado == null || valorPesquisado.isEmpty()) {
```

```

        int comecarPesquisaPosicao = 1;
    return cidadeDAO.buscaPorPaginacao(comecarPesquisaPosicao,
        maximoResultadosParaExibido);
}

return cidadeDAO.findByNameLike(valorPesquisado,
    maximoResultadosParaExibido);

```

O método da classe `MetodoAutoComplete.java` é bem simples e retorna uma lista. Ele recebe um valor enviado pelo usuário e testa se o valor tem um parâmetro preenchido. Se o usuário clicar no botão do `dropdown`, esse valor não viria preenchido.

Fique atento a um detalhe: toda vez que um valor de um componente `select` for um objeto, será necessário um `Converter`. Em nosso caso, o `itemValue` foi definido com `#
{cidade}`.

```

// CidadeConverter.java
@FacesConverter(forClass = Cidade.class)
public class CidadeConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext arg0, U
        IComponent arg1,
        String key) {
        CidadeDAO cidadeDAO = // busca o DAO;

        return cidadeDAO.findById(Integer.parseInt(key));
    }

    @Override
    public String getAsString(FacesContext arg0,
        UICOMPONENT arg1,
        Object cidadeObject) {
        if(cidadeObject != null &&
            cidadeObject instanceof Cidade){
            Cidade cidade = (Cidade) cidadeObject;
            return String.valueOf(cidade.getId());
        }
    }
}

```

```
        return "";
    }
}
```

Um Converter normal de JSF que simplesmente converte um valor enviado pela View em objeto, ou retorna o ID do objeto.

DICA

É preciso tomar bastante cuidado com a configuração/utilização de um Converter. A anotação `@FacesConverter(forClass = Cidade.class)` foi usada para definir que toda classe `Cidade` da aplicação utilizará esse Converter.

Infelizmente, algumas implementações simplesmente ignoram esse valor da anotação e não encontram o converter. A solução seria usar a anotação assim: `@FacesConverter(value = "cidadeConverter")`. E em cada componente que necessitasse de um Converter, fazer uso do seguinte atributo: `converter="cidadeConverter"`.

21.5 POLL

Existe um recurso interessante do JSF que é realizar chamadas a um ManagedBean sem a necessidade de ser disparada por um usuário. Em outras palavras, o Primefaces automaticamente dispara chamadas em um período predeterminado por meio do componente `Poll`.

▼ Contador

Chamadas Ajax serão feitas automaticamente: 3

Figura 21.10: Poll

Esse componente é simples de configurar.

```
<h:form>
    <h:outputText id="contadorText"
                  value="#{mensagens.contadorTexto}:
                  #{primefacesMB.contador}" />

    <p:poll interval="3"
              listener="#{primefacesMB.somar}"
              update="contadorText"/>
</h:form>
```

O componente `p:poll` realizará uma chamada *Ajax* a cada 3 segundos ao ManagedBean e, ao final da chamada, atualizará o `outputText` para exibir o novo valor.

```
private int contador;

public void somar(){
    contador++;
}

public int getContador() {
    return contador;
}

public void setContador(int contador) {
    this.contador = contador;
}
```

O componente `p:poll` aqui foi utilizado como um contador, mas poderia ser usado para verificar se o usuário continua na página do sistema, por exemplo.

21.6 CONSIDERAÇÕES FINAIS

Através dos componentes exibidos, foi possível ver que existem diversas funcionalidades prontas que funcionam sem a necessidade de códigos complicados e enormes. Todos os exemplos foram retirados do showcase do próprio Primefaces: <http://primefaces.org/showcase/>.

É possível fazer o download do código-fonte (SVN) de todos os exemplos [aqui](http://repository.primefaces.org/org/primefaces/prime-showcase/): <http://repository.primefaces.org/org/primefaces/prime-showcase/>.

Em seu fórum, é possível falar diretamente com os mantenedores do Primefaces. Lá eles tiram dúvidas e até registram bugs para futuros reparos. É muito utilizado em meio acadêmico (por já vir com um belo CSS) e por sua facilidade de uso. Os exemplos citados são bem fáceis de serem usados e aplicados com poucas linhas de código.

Infelizmente, nem tudo são flores: existe também o "*Dark Side*" do Primefaces. Eles não se preocupam nem um pouco com *retrocompatibilidade*. É muito comum ver na internet pessoas dizendo: "eu mudei da versão x para a x.1, e a tela parou de funcionar".

É preciso ter bastante cuidado ao mudar de versão, pois esse componente ainda não está confiável quanto a mudanças. Um novo comportamento interessante deles é que, antigamente, ao mudar de versão, caso um parâmetro fosse eliminado de um componente, a tela apresentaria erro. Hoje em dia, diversos componentes que encontram um parâmetro que foi eliminado não dão mais a mensagem de erro. Aquele parâmetro simplesmente

passa a ser ignorado.

ESTEJA ATENTO

No capítulo *O que eu uso? Action ou ActionListener?*, vimos quando e como utilizar `action` e `actionListener`. É preciso estar atento ao fato de que o componente `p:commandButton` do Primefaces utiliza Ajax por padrão. É normal encontrar na internet desenvolvedores reclamando que determinado método não é chamado, ou então que a navegação do método não está sendo feita.

Como vimos naquele capítulo, apenas uma `action` executa uma navegação de modo natural. É necessário desativar o Ajax do `commandButton` para que uma `action` se comporte normalmente. Basta definir que a opção `ajax` seja igual a `false`.

```
<p:commandButton ajax="false" />
```

Na documentação do Primefaces (<http://primefaces.org/documentation.html>), é possível encontrar todos os estilos de todos os componentes, como cor, fonte, configuração de CSS, e podem ser alterados.

CAPÍTULO 22

TEMAS DINÂMICOS COM PRIMEFACES

Uma das vantagens do Primefaces é a questão do tema. Existem diversos tipos de temas já prontos para serem utilizados, e a maior vantagem é que o desenvolvedor também pode criar um e usar.

Veja com é simples fazer uso de um tema do Primefaces. Primeiro vamos utilizar o ManagedBean de sessão do usuário para saber qual o seu tema.

```
// imports omitidos
@ManagedBean
@SessionScoped
public class UsuarioMB {
    private String tema;

    public String getTema() {
        if(tema == null){
            tema = "casablanca";
        }
        return tema;
    }
    // outros métodos omitidos
}
```

Veja que na classe UsuarioMB encontra-se uma String

chamada `tema`, que indicará ao Primefaces qual o tema do usuário. Um detalhe interessante é que, no `get`, é feito um teste para saber se o `tema` está `null` ou não. Seria nesse momento que, caso o `tema` estivesse `null`, se pegaria a informação do usuário logado na aplicação.

Para deixar a escolha do tema habilitado para o usuário, bastaria utilizar o componente chamado `ThemeSwitcher` do próprio Primefaces.

```
<p:themeSwitcher value="#{usuarioMB.tema}">
    <f:selectItems value="#{temasDisponiveis.temas}" />
</p:themeSwitcher>

@ApplicationScoped
@ManagedBean
public class TemasDisponiveis {
    private List<String> temas;

    public List<String> getTemas() {
        if(temas == null){
            temas = new ArrayList<String>();
            temas.add("casablanca");
            temas.add("cupertino");
            temas.add("dark-hive");
            temas.add("bluesky");
            temas.add("blitzer");
        }
        return temas;
    }

    public void setTemas(List<String> temas) {
        this.temas = temas;
    }
}
```

Basta colocar o componente `ThemeSwitcher` dentro de um `h:form` que, por *Ajax*, o Primefaces trocará toda a interface de seus componentes. Note que foi criado um `ManagedBean` com o

escopo ApplicationScoped que conterá todos os temas da aplicação.

Por último, será necessária uma pequena configuração no web.xml da aplicação. O código adicionado no web.xml informará ao Primefaces onde buscar o tema; em nosso caso, no ManagedBean.

```
<context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>#{usuarioMB.tema}</param-value>
</context-param>
```

E para o usuário trocar o tema atual por outros, será bem simples. Veja nas figuras a seguir.



Figura 22.1: Tema selecionado

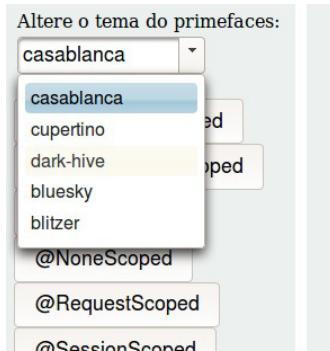


Figura 22.2: Escolhendo novo tema

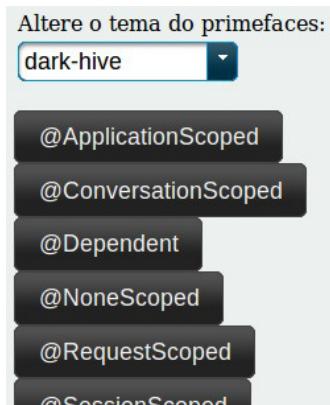


Figura 22.3: Novo tema aplicado

É possível criar um evento Ajax para persistir a alteração no banco de dados para o usuário. Basta adicionar a linha `<p:ajax listener="#{usuarioMB.salvarTemaNoDB}"/>` dentro do componente `ThemeSwitcher` que, via *Ajax*, o Primefaces chamará esse método e esse valor poderá ser persistido.

DICA

Note que o objeto `tema` da classe `UsuarioMB` é apenas uma String. Em diversos tutoriais, livros e até mesmo no showcase do Primefaces, são utilizadas classes para esse valor.

Neste livro, foi usada a abordagem de String para ficar mais fácil o entendimento e aplicação.

A melhor parte dessa história é que, para utilizar um desses temas, basta adicionar os temas predefinidos no `pom.xml` ou na pasta `WEB-INF/lib`.

```
<!-- pom -->
<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>cupertino</artifactId>
    <version>${primefaces.tema}</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>dark-hive</artifactId>
    <version>${primefaces.tema}</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>casablanca</artifactId>
    <version>${primefaces.tema}</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>bluesky</artifactId>
    <version>${primefaces.tema}</version>
    <scope>runtime</scope>
```

```
</dependency>
<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>blitzer</artifactId>
    <version>${primefaces.tema}</version>
    <scope>runtime</scope>
</dependency>
```

Para criar seu próprio tema, vá ao site: <http://jqueryui.com/themeroller/>. Nele é possível estilizar os aspectos visuais da aplicação. No manual do Primefaces, é mostrado como aplicar esse tema recém-criado ao seu projeto.

CAPÍTULO 23

COMPONENTES DO PRIMEFACES NÃO APARECEM

É fácil perder bons minutos procurando na internet a solução para esse problema. Às vezes, temos um código todo correto, mas o infeliz do componente do Primefaces não aparece por nada.

```
<head>
</head>
<body>
    <!-- exemplo.xhtml -->
    <p:button value="Olá" />
</body>
```

Por mais que um código pareça correto, o Primefaces tem suas regras para funcionar. O código do arquivo `exemplo.xhtml` mostra uma página com todos os blocos necessários da linguagem HTML. A grande pegadinha aí é que os componentes no Primefaces necessitam de `h:head` e não `head`, e de `h:body` em vez de apenas `body`. Esse pequeno detalhe faz toda diferença.

Ainda existe um porém. Às vezes, mesmo em uma página com `h:head` e `h:body`, o componente não vai aparecer.

```
<h:head>
</h:head>
```

```
<h:body>
    <!-- exemplo2.xhtml -->
    <p:editor value="" />
</h:body>
```

Mesmo a página exemplo2.xhtml contendo h:head e h:body , em algumas versões do Primefaces, esse componente pode não aparecer. Qual seria o motivo?

Diversos componentes do Primefaces necessitam estar envoltos pela tag h:form . No código anterior, bastaria colocar a tag h:form antes do p:editor e fechá-la após o p:editor .

CAPÍTULO 24

RICHFACES

O Richfaces é uma velha biblioteca de componentes do JSF, massivamente utilizado no JSF 1.2. Sua utilização no JSF 1.2 era ideal para cenários que necessitassem de *Ajax*.

Atualmente, a recém-versão 4 foi lançada para voltar a concorrer com as outras bibliotecas do mercado, que hoje tem o foco no JSF 2.0. O Richfaces tem bastantes componentes desenvolvidos, e que facilitam o uso do desenvolvedor.

24.1 DATATABLE COM COLUNAS CONGELADAS

A função de congelar colunas é possível de ser utilizada no Excel e pode ser algo muito prático nas aplicações JSF. O Richfaces tem essa função.

Cidades				
Nome	Estado	Total de Ruas	Total de Bairros	
CIDADE_1	ESTADO_1	0	0	
CIDADE_2	ESTADO_2	33	15	
CIDADE_3	ESTADO_3	66	30	
CIDADE_4	ESTADO_4	99	45	
CIDADE_5	ESTADO_5	132	60	
CIDADE_6	ESTADO_6	165	75	
CIDADE_7	ESTADO_7	198	90	
CIDADE_8	ESTADO_8	231	105	
CIDADE_9	ESTADO_9	264	120	

Figura 24.1: DataTable com colunas congeladas

Cidades				
Nome	Estado	Total de Ruas	Total de Bairros	Total Eleit
CIDADE_1	ESTADO_1	0	0	0
CIDADE_2	ESTADO_2	15	3150	
CIDADE_3	ESTADO_3	30	6300	
CIDADE_4	ESTADO_4	45	9450	
CIDADE_5	ESTADO_5	60	12600	
CIDADE_6	ESTADO_6	75	15750	
CIDADE_7	ESTADO_7	90	18900	
CIDADE_8	ESTADO_8	105	22050	
CIDADE_9	ESTADO_9	120	25200	

Figura 24.2: DataTable com colunas congeladas

Cidades			
Nome	Estado	Total de Bairros	Total de Eleitores
CIDADE_1	ESTADO_1	0	0
CIDADE_2	ESTADO_2	15	3150
CIDADE_3	ESTADO_3	30	6300
CIDADE_4	ESTADO_4	45	9450
CIDADE_5	ESTADO_5	60	12600
CIDADE_6	ESTADO_6	75	15750
CIDADE_7	ESTADO_7	90	18900
CIDADE_8	ESTADO_8	105	22050
CIDADE_9	ESTADO_9	120	25200

Figura 24.3: DataTable com colunas congeladas

As figuras mostram como esse recurso é interessante para melhorar a utilização dos usuários acostumados com essa facilidade.

```
<!-- dataTable.xhtml -->
<rich:extendedDataTable value="#{richfacesMB.cidades}"
    var="cidade"
    frozenColumns="2"
    style="height:250px; width:400px;"
    selectionMode="none">

    <f:facet name="header">
        <h:outputText value="#{mensagens.cidadePlural}" />
    </f:facet>
    <rich:column>
        <f:facet name="header">
            <h:outputText value="#{mensagens.cidadeNome}" />
        </f:facet>
        <h:outputText value="#{cidade.nome}" />
    </rich:column>
    <rich:column>
        <f:facet name="header">
            <h:outputText value="#{mensagens.cidadeEstado}" />
        </f:facet>
        <h:outputText value="#{cidade.estado}" />
    </rich:column>

```

```

        </f:facet>
        <h:outputText value="#{cidade.estado}" />
    </rich:column>
    <rich:column>
        <f:facet name="header">
            <h:outputText value="#{mensagens.cidadeTotalRuas}" />
        </f:facet>
        <h:outputText value="#{cidade.totalRuas}" />
    </rich:column>
    <!-- outras 1000 colunas -->
</rich:extendedDataTable>

```

O `DataTable`, apesar de ter um funcionamento diferente, tem sua configuração bem parecida com a de qualquer outro `DataTable`. O que o torna diferente é a configuração `frozenColumns="2"` que determina a quantidade de colunas a serem congeladas.

24.2 TOOL TIP

A ferramenta de Tool Tip é bem comum no mundo web. Basta passar o mouse que uma informação sobre aquela área será exibida. A figura seguinte mostra como essa ferramenta funciona com o Richfaces.



Figura 24.4: Tool Tip Normal

```

<rich:panel>
    #{mensagens.richfacesToolTipNormal}
    <rich:tooltip>
        #{mensagens.richfacesToolTipNormalTexto}
    </rich:tooltip>
</rich:panel>

```

O código para exibir um `ToolTip` precisa apenas do componente `rich:tooltip` dentro da região em que ele deve ser exibido. O Richfaces tem outras configurações possíveis para um `ToolTip`.

A figura a seguir mostra como seria um `ToolTip` com *delay*.

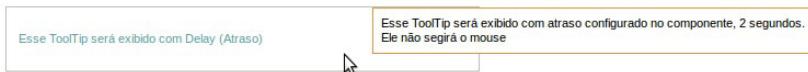


Figura 24.5: Tool Tip com delay

```
<rich:panel>
    <h:outputText value="#{mensagens.richfacesToolTipDelay}" />
    <rich:tooltip followMouse="false" showDelay="2000">
        <h:outputText
            value="#{mensagens.richfacesToolTipDelayTexto}" />
    </rich:tooltip>
</rich:panel>
```

O *delay* é usado para atrasar a exibição do `ToolTip` para o usuário. Para utilizar o *delay*, basta configurar com o atributo `showDelay="2000"` que o texto do `Tool Tip` será exibido após dois segundos. O atributo `followMouse="false"` foi usado para informar ao RichFaces que o `ToolTip` não deve seguir o mouse.

A figura a seguir mostra como seria um `ToolTip` com um texto chamado via *Ajax*.

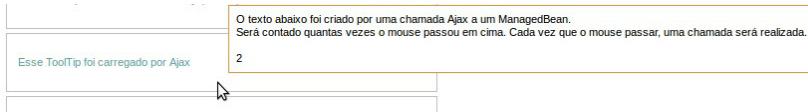


Figura 24.6: Tool Tip via Ajax

```
<h:form>
    <rich:panel>
```

```

<h:outputText value="#{mensagens.richfacesToolTipAjax}" />
<rich:tooltip mode="ajax">
    <h:outputText
        value="#{mensagens.richfacesToolTipAjaxTexto}" />
    <h:outputText value="#{richfacesMB.contadorAjax}" />
</rich:tooltip>
</rich:panel>
</h:form>

```

Para que uma mensagem exibida seja carregada via *Ajax*, basta utilizar o atributo `mode="ajax"`. O `get/set` do atributo `contadorAjax` são normais com a única diferença de que aumentam o contador a cada chamada.

```

public int getContadorAjax() {
    return ++contadorAjax;
}

public void setContadorAjax(int contadorAjax) {
    this.contadorAjax = contadorAjax;
}

```

Uma outra opção interessante é a de exibir o `ToolTip` apenas após o clique. E para tornar essa opção mais interessante, uma chamada *Ajax* poderia ser executada após o clique. A figura adiante mostra como seria um `ToolTip` com um clique e *Ajax*.



Figura 24.7: Tool Tip via Ajax

```

<rich:panel>
    <h:outputText
        value="#{mensagens.richfacesToolTipPorClique}" />
    <rich:tooltip followMouse="false" showEvent="click"
        mode="ajax">
        <h:outputText value="#{mensagens.porCliqueTexto}" />
        <h:outputText value="#{richfacesMB.contadorClique}" />
    </rich:tooltip>
</rich:panel>

```

```
</rich:tooltip>  
</rich:panel>
```

Para definir que um `ToolTip` pode ser apenas exibido após o clique, basta utilizar o atributo `showEvent="click"`, e um `get/set` no ManagedBean.

```
public int getContadorClique() {  
    return contadorClique++;  
}  
  
public void setContadorClique(int contadorClique) {  
    this.contadorClique = contadorClique;  
}
```

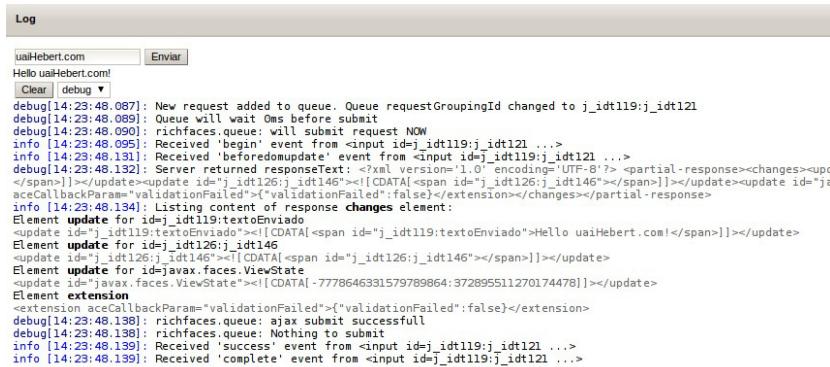
24.3 LOG

Uma ferramenta que pode ajudar em requisições *Ajax* é a Log. A figura seguinte mostra como ela trabalha:



Figura 24.8: Log de chamadas Ajax

Note que são exibidas as informações relacionadas à chamada *Ajax*. É possível também realizar chamadas com outros níveis de Log (ver nível debug na figura a seguir).



The screenshot shows a log window with the title 'Log'. It contains several lines of debug and info messages related to an Ajax request. The messages include details about the request being added to a queue, the server returning a response with XML content, and various update and extension events. The log ends with a note about a 'validationFailed' event.

```

uaHebert.com Enviar
Clear debug ▾
debug[14:29:48.087]: New request added to queue. Queue requestGroupId changed to j_idt119:j_idt121
debug[14:29:48.089]: Queue will wait 0ms before submit
richfaces.queue: will submit request NOW
info [14:29:48.095]: Received 'begin' event from <input id=j_idt119:j_idt121 ...>
info [14:29:48.131]: Received 'beforeondomupdate' event from <input id=j_idt119:j_idt121 ...>
debug[14:29:48.132]: Server returned responseText: <?xml version='1.0' encoding='UTF-8'?><partial-response><changes><update id="j_idt119:j_idt126:j_idt146"><![CDATA[<span id=j_idt126:j_idt146></span>]]></update><update id="j_idt119:j_idt126:j_idt146"><![CDATA[<span id=j_idt126:j_idt146></span>]]></update><update id="j_idt119:j_idt126:j_idt146"><![CDATA[<span id=j_idt126:j_idt146></span>]]></update><extension acelockbackForSam="validationFailed"><validationFailed>false</validationFailed></extension>
debug[14:29:48.138]: richfaces.ajax.ajax submit successful
debug[14:29:48.138]: richfaces.queue: Nothing to submit
info [14:29:48.139]: Received 'success' event from <input id=j_idt119:j_idt121 ...>
info [14:29:48.139]: Received 'complete' event from <input id=j_idt119:j_idt121 ...>
info [14:29:48.139]: Listing content of response changes element:
Element update for id=j_idt119:textoEnviado
<update id="j_idt119:textoEnviado"><![CDATA[<span id=j_idt119:textoEnviado>Hello uaHebert.com!</span>]]></update>
Element update for id=j_idt126:j_idt146
<update id="j_idt126:j_idt146"><![CDATA[<span id=j_idt126:j_idt146></span>]]></update>
Element update for id=javax.faces.ViewState
<update id="javax.faces.ViewState"><![CDATA[-7778646331579789864:37289511270174478]]></update>
Element extension
<extension acelockbackForSam="validationFailed"><validationFailed>false</validationFailed></extension>
debug[14:29:48.138]: richfaces.ajax.ajax submit successful
debug[14:29:48.138]: richfaces.queue: Nothing to submit
info [14:29:48.139]: Received 'success' event from <input id=j_idt119:j_idt121 ...>
info [14:29:48.139]: Received 'complete' event from <input id=j_idt119:j_idt121 ...>
info [14:29:48.139]: validationFailed=false

```

Figura 24.9: Log de chamadas Ajax com log em debug

E para utilizar esse recurso, é necessário adicionar um simples componente à página: `<a4j:log/>`. Ele realizará todo o trabalho de colocar a chamada *Ajax* e exibir. O código da página exibida nas figuras pode ser encontrado a seguir:

```

<h:form>
    <h:inputText value="#{richfacesMB.nomeEnviado}" />
    <a4j:commandButton value="#{mensagens.enviar}"
        render="textoEnviado"/>
    <br/>
    <a4j:outputPanel id="textoEnviado">
        <h:outputText value="Hello #{richfacesMB.nomeEnviado}!" 
            rendered="#{not empty richfacesMB.nomeEnviado}"/>
    </a4j:outputPanel>
</h:form>
<a4j:log/>

```

24.4 PANEL MENU

O *panelMenu* é um componente que ajuda na organização de material, e até mesmo facilita a navegação do usuário. Veja na figura adiante como ele é visualmente.



Figura 24.10: Panel Menu

O interessante desse componente é que ele funciona como um agrupador de conteúdo, tem efeito Jquery ao se trocar de blocos de códigos, e é possível integrar com chamadas Ajax.

```
<rich:panelMenu itemMode="ajax"
    groupMode="ajax"
    groupExpandedLeftIcon="triangleUp"
    groupCollapsedLeftIcon="triangleDown"
    topGroupExpandedRightIcon="chevronUp"
    topGroupCollapsedRightIcon="chevronDown"
    itemLeftIcon="disc"
    itemChangeListener=
        "#{richfacesMB.atualizarGrupoSelecionado}">

<rich:panelMenuGroup label="#{mensagens.richfacesGrupo} 1">
    <rich:panelMenuItem label="Item 1.1" name="Item_1_1"/>
    <rich:panelMenuItem label="Item 1.2" name="Item_1_2"/>
    <rich:panelMenuItem label="Item 1.3" name="Item_1_3"/>
```

```

</rich:panelMenuGroup>
<rich:panelMenuGroup label="#{mensagens.richfacesGrupo} 2">
    <rich:panelMenuItem label="Item 2.1" name="Item_2_1"/>
    <rich:panelMenuItem label="Item 2.2" name="Item_2_2"/>
    <rich:panelMenuItem label="Item 2.3" name="Item_2_3"/>
    <rich:panelMenuGroup
        label="#{mensagens.richfacesGrupo} 2.4">
        <rich:panelMenuItem label="Item 2.4.1"
            name="Item_2_4_1"/>
        <rich:panelMenuItem label="Item 2.4.2"
            name="Item_2_4_2"/>
        <rich:panelMenuItem label="Item 2.4.3"
            name="Item_2_4_3"/>
    </rich:panelMenuGroup>
    <rich:panelMenuItem label="Item 2.5" name="Item_2_5"/>
</rich:panelMenuGroup>
<rich:panelMenuGroup label="#{mensagens.richfacesGrupo} 3">
    <rich:panelMenuItem label="Item 3.1" name="Item_3_1"/>
    <rich:panelMenuItem label="Item 3.2" name="Item_3_2"/>
    <rich:panelMenuItem label="Item 3.3" name="Item_3_3"/>
</rich:panelMenuGroup>
</rich:panelMenu>

```

Os atributos `itemMode="ajax"` e `groupMode="ajax"` são configurações relacionadas ao *Ajax*.

```
//indica qual método será chamado via Ajax a cada mudança de item
itemChangeListener="#{richfacesMB.atualizarGrupoSelecionado}"
```

24.5 REPEAT

Uma função interessante do Richfaces é o componente que pode repetir outros componentes. De certo modo, funciona como um loop, mas com valores diferentes.

```

<a4j:repeat value="#{richfacesMB.cidadosParaRepeat}"
    var="cidade"
    rows="20">
    <rich:panel>
        <f:facet name="header">
            <h:panelGroup>
```

```

        <h:outputText value="#{cidade.nome}" />
    </h:panelGroup>
</f:facet>
<h:panelGrid columns="2">
    <h:outputText value="#{mensagens.cidadeEstado}" />
    <h:outputText value="#{cidade.estado}" />
    <h:outputText
        value="#{mensagens.cidadeTotalHabitantes}" />
    <h:outputText value="#{cidade.totalDeHabitacoes}" />
</h:panelGrid>
</rich:panel>
</a4j:repeat>

```

O componente `a4j:repeat` receberá uma lista de objetos da classe `Cidade`, e o atributo `rows="20"` define que serão repetidos 20 objetos por página. Veja como esse componente é visualmente:

Repeat			
CIDADE_1 Estado ESTADO_1 Total de Habitantes 0	CIDADE_2 Estado ESTADO_2 Total de Habitantes 75	CIDADE_3 Estado ESTADO_3 Total de Habitantes 150	CIDADE_4 Estado ESTADO_4 Total de Habitantes 225
CIDADE_5 Estado ESTADO_5 Total de Habitantes 300	CIDADE_6 Estado ESTADO_6 Total de Habitantes 375	CIDADE_7 Estado ESTADO_7 Total de Habitantes 450	CIDADE_8 Estado ESTADO_8 Total de Habitantes 525
CIDADE_9 Estado ESTADO_9 Total de Habitantes 600	CIDADE_10 Estado ESTADO_10 Total de Habitantes 675	CIDADE_11 Estado ESTADO_11 Total de Habitantes 750	CIDADE_12 Estado ESTADO_12 Total de Habitantes 825
CIDADE_13 Estado ESTADO_13 Total de Habitantes 900	CIDADE_14 Estado ESTADO_14 Total de Habitantes 975	CIDADE_15 Estado ESTADO_15 Total de Habitantes 1050	CIDADE_16 Estado ESTADO_16 Total de Habitantes 1125
CIDADE_17 Estado ESTADO_17 Total de Habitantes 1200	CIDADE_18 Estado ESTADO_18 Total de Habitantes 1275	CIDADE_19 Estado ESTADO_19 Total de Habitantes 1350	CIDADE_20 Estado ESTADO_20 Total de Habitantes 1425
... 1 2 3 4 5 » >>>			

Figura 24.11: Repeat

24.6 CONSIDERAÇÕES FINAIS

Existem diversos componentes hoje que ajudam o

desenvolvedor com funcionalidades e facilidades. Outra vantagem do Richfaces é que seu código do lado do servidor é considerado o mais robusto. Ele tem um excelente enfileiramento de chamadas Ajax, utiliza JMS para o componente `a4j:push`, e já implementa a validação através da *JSR-303 (Bean Validation)*.

Infelizmente, o Richfaces tem um pequeno número de componentes, sua diversidade é baixa. Outra melhoria que poderia ser feita no Richfaces seria sua documentação que contivesse mais detalhes.

CAPÍTULO 25

ICEFACES

Das bibliotecas citadas aqui, o Icefaces é o mais velho de todos, e um dos menos utilizados. Ele tem uma biblioteca com diversos componentes para ajudar tanto na parte visual como no *server-side*. Seu showcase pode ser acessado em: <http://icefaces-showcase.icesoft.org/showcase.jsf>.

25.1 MENU

O Icefaces fornece um componente de menu que permite chamadas *Ajax*, além de colocar imagens de modo atrativo ao usuário.

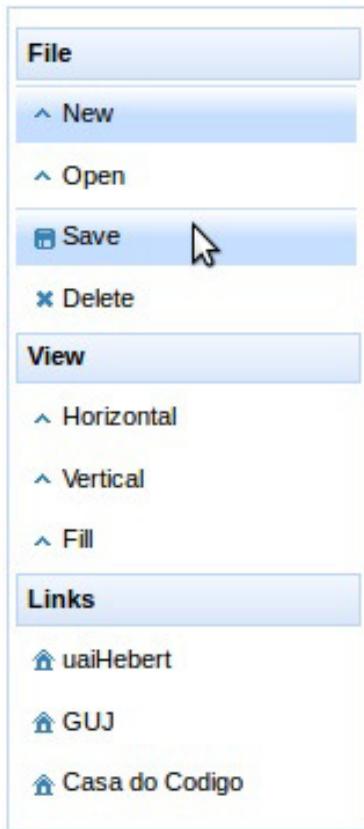


Figura 25.1: Menu

Essa figura mostra um menu que poderia ficar acessível a toda aplicação. Ele não tem apenas um tipo de ações, mas indica que pode haver tanto ações específicas (salvar, incluir etc.) como navegações.

```
<ace:menu type="plain" id="exampleMenu" >
    <ace:submenu label="File" id="file">
        <ace:menuItem id="new"
            value="New"
            actionListener="#{icefacesMB.fireAction}"
```

```

        icon="ui-icon">
    <ace:ajax event="activate"
              execute="@this"
              render="message" />
</ace:menuItem>
<ace:menuItem id="open"
              value="Open"
              actionListener="#{icefacesMB.fireAction}"
              icon="ui-icon">
    <ace:ajax event="activate"
              execute="@this"
              render="message" />
</ace:menuItem>
<ace:menuItem id="save"
              value="Save"
              actionListener="#{icefacesMB.fireAction}"
              icon="ui-icon ui-icon-disk">
    <ace:ajax event="activate"
              execute="@this"
              render="message" />
</ace:menuItem>
<ace:menuItem id="delete"
              value="Delete"
              actionListener="#{icefacesMB.fireAction}"
              icon="ui-icon ui-icon-close">
    <ace:ajax event="activate"
              execute="@this"
              render="message" />
</ace:menuItem>
</ace:submenu>

<ace:submenu id="view" label="View">
    <ace:menuItem id="horizontal"
                  value="Horizontal"
                  actionListener="#{icefacesMB.fireAction}"
                  icon="ui-icon">
        <ace:ajax event="activate"
                  execute="@this"
                  render="message" />
    </ace:menuItem>
    <ace:menuItem id="vertical"
                  value="Vertical"
                  actionListener="#{icefacesMB.fireAction}"
                  icon="ui-icon">
        <ace:ajax event="activate"
                  execute="@this"
                  render="message" />
    </ace:menuItem>
</ace:submenu>
```

```

        execute="@this"
        render="message" />
    </ace:menuItem>
    <ace:menuItem id="fill"
        value="Fill"
        actionListener="#{icefacesMB.fireAction}"
        icon="ui-icon">
        <ace:ajax event="activate"
            execute="@this"
            render="message" />
    </ace:menuItem>
</ace:submenu>

<ace:submenu label="Links">
    <ace:menuItem value="uaiHebert"
        url="http://uaihebert.com"
        target="_blank"
        icon="ui-icon ui-icon-home"/>
    <ace:menuItem value="GUJ"
        url="http://www.guj.com.br"
        target="_blank"
        icon="ui-icon ui-icon-home"/>
    <ace:menuItem value="Casa do Código"
        url="http://www.casadocodigo.com.br"
        target="_blank"
        icon="ui-icon ui-icon-home"/>
</ace:submenu>
</ace:menu>
```

O menu é composto de dois componentes: ace:menu e ace:submenu . Ambos estão envolvendo uma tag chamada ace:ajax que executará uma ação *Ajax* quando cada item for selecionado.

25.2 DIALOG

O Icefaces tem um Dialog que é simples de montar e suas ações são fáceis de definir.



Figura 25.2: Dialog



Figura 25.3: Dialog

As figuras mostram esse útil componente que, em geral, é utilizado para confirmar ações em muitas partes do sistema.

```
<h:panelGrid styleClass="centeredPanelGrid"
    id="panelGridDialog">
    <h:commandButton id="save"
        value="Save"
        onclick="confirmation.show()"
        type="button"/>
    <h:outputText id="outcome"
        value="#{icefacesMB.resultadoDialog}"
        rendered=
            "#{icefacesMB.resultadoDialog != null}"/>
</h:panelGrid>

<h:form id="formConfirmDialog">
    <ace:confirmationDialog id="confirmDialog"
        widgetVar="confirmation"
        message="Are you sure about this?"
        header="Confirmation"
        width="250"
```

```
height="100"
closable="true"
position="center">
<h:panelGrid columns="2" styleClass="centeredPanelGrid">
    <h:commandButton id="yes" value="Yes"
        onclick="confirmation.hide()"
        actionListener=
            "#{icefacesMB.dialogSim}"/>
    <h:commandButton id="no" value="No"
        onclick="confirmation.hide()"
        actionListener=
            "#{icefacesMB.dialogNao}"/>
</h:panelGrid>
</ace:confirmationDialog>
</h:form>
```

O componente `ace:confirmationDialog` é usado para exibir a confirmação.

25.3 RICHTEXT

Se necessário simular um editor de texto, o Icefaces tem uma boa opção para agradar o usuário.

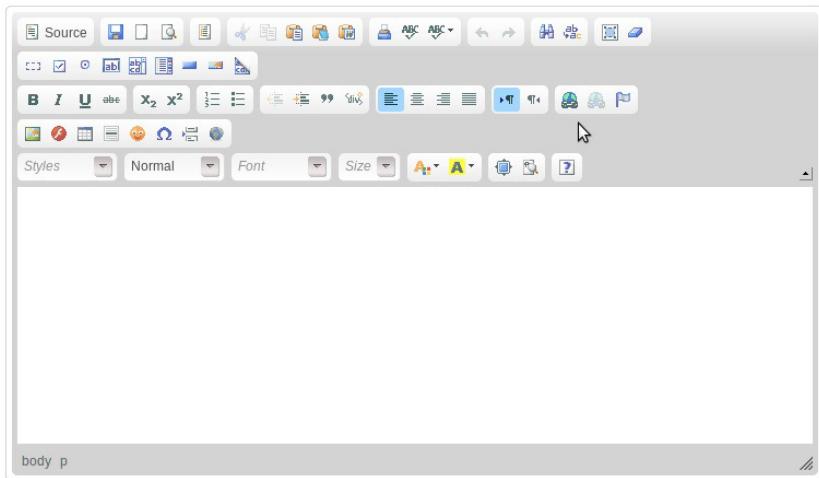


Figura 25.4: RichText

E para usar esse componente, basta um código bastante simples.

```
<ace:richTextEntry value="#{icefacesMB.richText}"  
skin="kama" toolbar="Default"/>
```

25.4 DATA

Por diversas vezes, criar um componente de data que seja agradável ao usuário é complicado. Diversos códigos de JavaScript e CSS existem no mercado para tentar fazer uma interface bonita. O Icefaces já tem um componente bastante simples de usar.

Selecionar Data



Figura 25.5: Componente de Data

```
<ace:dateTimeEntry value="#{icefacesMB.data}"  
pattern="dd/MM/yyyy"  
renderAsPopup="true"/>
```

O `richTextEntry` é o componente utilizado para criar um editor de texto com diversas opções de formatação. A opção `skin` habilita escolher algum tema disponível do Icefaces, e a opção `toolbar` permite escolher quais componentes estarão disponíveis para o usuário.

25.5 RESIZE

O `Resize` é um componente do Icefaces que permite ao usuário alterar o tamanho do componente na tela.

Resize com efeito após o redirecionamento apenas em um
inputTextarea

Panel com Resize

Resize para todas as direções sem efeitos
"especiais"

Figura 25.6: Resize

Na figura, é possível ver que existe uma opção de Resize com muitas opções e uma opção mais simples.

```
<h:inputTextarea value="#{mensagens.icefacesResizeEfeito}">
    <ace:resizable animate="true"
        ghost="true"
        effectDuration="slow"
        minHeight="50"
        minWidth="200"
        maxHeight="250"
        maxWidth="650"/>
</h:inputTextarea>
<ace:panel header="#{mensagens.icefacesResizePanel}">
    <h:outputText value="#{mensagens.icefacesResizeGradual}" />
    <ace:resizable handles="all"
        grid="20">
```

```
    minHeight="160"
    minWidth="300"
    maxHeight="300"
    maxWidth="700"/>
</ace:panel>
```

Para utilizar o `ace:resizable`, basta colocá-lo dentro de outro componente. As configurações `minHeight`, `minWidth`, `maxHeight` e `maxWidth` definem configurações relacionadas ao tamanho mínimo e máximo de larguras (`width`) e alturas (`height`). A configuração `animate` indica para o Icefaces que uma animação deve ser utilizada; `ghost="true"` exibirá um helper semitransparente durante o `resize`; e `effectDuration` define a velocidade do efeito a ser executado.

25.6 CONSIDERAÇÕES FINAIS

O Icefaces tem a vantagem de usar uma abordagem chamada *Direct-2-Dom*. Ele mantém uma cópia do DOM exibido na página do usuário, quando este envia uma solicitação: em vez de responder com uma página inteira, o Icefaces enviará apenas o que foi alterado como resposta.

Ele verificará a diferença entre o DOM no servidor e o DOM do usuário para enviar apenas o necessário. Com essa tecnologia, o tráfego de dados é menor.

O Icefaces tem uma grande quantidade de documentação e exemplos pela internet. Seu maior defeito é a quantidade de problemas abertos em seu Jira sem correção.

CÓPIA DESONESTA? OU, SE É CÓDIGO LIVRE, POSSO COPIAR?

Um outro fato interessante foi que o Icefaces foi acusado de uma cópia do código-fonte do Primefaces de modo desleal. No blog do Primefaces (<http://blog.primefaces.org/?p=1692>) foi exibido um código que foi totalmente copiado, mas não foi dado o devido crédito a quem realmente o criou.

Muita coisa aconteceu depois desse dia, e hoje o Icefaces reconheceu que muito dos seus componentes foram baseados e "powered" pelo Primefaces. Isso só aconteceu depois que o criador do Primefaces fez uma reclamação.

CAPÍTULO 26

EVITE MISTURAR AS IMPLEMENTAÇÕES/BIBLI OTECAS DE COMPONENTES

É muito fácil encontrar em uma implementação ou em uma biblioteca de componente algum componente que nos agrade, mas esse componente pode não existir na implementação/biblioteca que é utilizada na aplicação em que trabalhamos. Imagine uma aplicação que use Mojarra com Icefaces, mas, para um novo requisito da aplicação, um componente existente no Richfaces seria perfeito.

O problema é que cada implementação/biblioteca tem suas peculiaridades e suas próprias rotinas para funcionar. Por exemplo, o Primefaces criou seu processador de *Ajax* através da tag `p:ajax`.

É possível encontrar na internet diversos relatos de problemas relacionados ao misturar o componente `rich:calendar` com `p:tooltip`. O Richfaces tem seu modo próprio de trabalhar, assim como o próprio Primefaces. É necessário entender que cada

implementação/biblioteca, apesar de ter sua base no JSF, não tem compatibilidade garantida com as outras.

Aviso do JBoss

Ao executar uma aplicação no JBoss com o MyFaces, a seguinte mensagem é exibida: "WARN [JBossJSFConfigureListener] MyFaces JSF implementation found! This version of JBoss AS ships with the java.net implementation of JSF. There are known issues when mixing JSF implementations. This warning does not apply to MyFaces component libraries such as Tomahawk. However, myfaces-impl.jar and myfaces-api.jar should not be used without disabling the built-in JSF implementation. See the JBoss wiki for more details".

Note que o próprio JBoss avisa que o ideal seria apenas utilizar uma implementação, a que ele fornece.

CAPÍTULO 27

NÃO FAÇA PAGINAÇÃO NO LADO DO SERVIDOR

Uma das grandes dificuldades do JSF é sobre como fazer paginação por demanda. Quando utilizamos o `h:datatable`, é possível ver que não existe opção nativa de paginação. Veja a figura:

Id	Nome	Estado
0	CIDADE_0	ESTADO_0
1	CIDADE_1	ESTADO_1
2	CIDADE_2	ESTADO_2
3	CIDADE_3	ESTADO_3
4	CIDADE_4	ESTADO_4
5	CIDADE_5	ESTADO_5
6	CIDADE_6	ESTADO_6
7	CIDADE_7	ESTADO_7
8	CIDADE_8	ESTADO_8
9	CIDADE_9	ESTADO_9
10	CIDADE_10	ESTADO_10
11	CIDADE_11	ESTADO_11

Figura 27.1: Datatable do JSF

Conseguimos essa tabela com o seguinte código:

```
<!-- dataTable nativo do JSF -->
<h:datatable value="#{cidadesPaginadas.cidades}"
    var="cidade">
    <h:column>
```

```
<f:facet name="header">
    #{mensagens.cidadeId}
</f:facet>
<h:outputText value="#{cidade.id}" />
</h:column>
<h:column>
    <f:facet name="header">
        #{mensagens.cidadeNome}
    </f:facet>
    <h:outputText value="#{cidade.nome}" />
</h:column>
<h:column>
    <f:facet name="header">
        #{mensagens.cidadeEstado}
    </f:facet>
    <h:outputText value="#{cidade.estado}" />
</h:column>
</h:dataTable>
```

Infelizmente, o `dataTable` do JSF não é dos mais bonitos, nem práticos. Se você colocar uma coleção de 100.000 itens para o `dataTable`, todos eles serão exibidos. Dessa forma, além de você manter no mínimo 100.000 objetos na memória, a renderização da tela demorará muito, pois a quantidade de informação que deverá ser mostrada é muito grande.

Indo mais além, é ruim também para o usuário, já que a usabilidade ficará prejudicada. Como encontrar uma informação numa lista com outras 100.000? É como encontrar uma agulha no palheiro.

Existem diversas formas de fazer uma paginação dos dados em um `dataTable`. Porém, vamos precisar da ajuda do Primefaces, que já possui toda a infraestrutura necessária para facilitar nossa tarefa.

Veja na figura a seguir como fica um `DataTable` paginado pelo Primefaces.

(6 of 100)		1	2	3	4	5	6	7	8	9	10	10
Id	Nome	Estado										
50	CIDADE_50	ESTADO_50										
51	CIDADE_51	ESTADO_51										
52	CIDADE_52	ESTADO_52										
53	CIDADE_53	ESTADO_53										
54	CIDADE_54	ESTADO_54										
55	CIDADE_55	ESTADO_55										
56	CIDADE_56	ESTADO_56										
57	CIDADE_57	ESTADO_57										
58	CIDADE_58	ESTADO_58										
59	CIDADE_59	ESTADO_59										

Figura 27.2: Datatable do Primefaces

```
<!-- paginação nativa do primefaces -->
<p:datatable value="#{cidadesPaginadas.cidades}"
    var="cidade"
    paginator="true"
    rows="10"
    paginatorTemplate="{CurrentPageReport} {FirstPageLink}
        {PreviousPageLink} {PageLinks}
        {NextPageLink} {LastPageLink}
        {RowsPerPageDropdown}"
    rowsPerPageTemplate="5,10,15"
    style="width: 40%">
    <p:column>
        <f:facet name="header">
            #{mensagens.cidadeId}
        </f:facet>
        <h:outputText value="#{cidade.id}" />
    </p:column>
    <p:column>
        <f:facet name="header">
            #{mensagens.cidadeNome}
        </f:facet>
        <h:outputText value="#{cidade.nome}" />
    </p:column>
    <p:column>
        <f:facet name="header">
            #{mensagens.cidadeEstado}
        </f:facet>
```

```
</f:facet>
<h:outputText value="#{cidade.estado}" />
</p:column>
</p:dataTable>
```

O próprio Primefaces cuida de nos oferecer de antemão um layout agradável para o `dataTable`. Note que o código é bem parecido com um `dataTable` nativo do JSF, e adicionamos poucas configurações do Primefaces relacionadas à paginação.

Ao utilizar a paginação nativa do Primefaces, toda a lista é alocada na sessão do usuário. Caso uma consulta retornasse muitos objetos, facilmente poderia estourar a memória do servidor.

Temos nesse momento uma paginação que acontece apenas no lado do cliente. Ela melhora a usabilidade da aplicação, mas ainda onera o uso do servidor, mantendo objetos demais e possivelmente desnecessários na memória.

Para evitar isso, pode-se usar a técnica chamada *paginação por demanda (Lazy Pagination)*. É uma técnica bem simples que trabalha em cima da seguinte filosofia: "qual a razão de carregar objetos que o usuário pode nem chegar a ver?".

O princípio da paginação por demanda é simples: a cada clique para ir à próxima página do `dataTable`, uma nova consulta será feita no banco de dados buscando novos valores para exibir. A vantagem dessa técnica é que, em vez de trazer os 100.000 registros de uma só vez, virá do banco de dados apenas o que será exibido ao usuário.

Veja nas figuras seguintes como ficará o `DataTable` com paginação por demanda.

(1 of 100)		
Id	Nome	Estado
1	CIDADE_1	ESTADO_1
2	CIDADE_2	ESTADO_2
3	CIDADE_3	ESTADO_3
4	CIDADE_4	ESTADO_4
5	CIDADE_5	ESTADO_5
6	CIDADE_6	ESTADO_6
7	CIDADE_7	ESTADO_7
8	CIDADE_8	ESTADO_8
9	CIDADE_9	ESTADO_9
10	CIDADE_10	ESTADO_10

Figura 27.3: Datatable do Primefaces com LazyLoad

(1 of 1)		
Id	Nome	Estado
45		9
945	CIDADE_945	ESTADO_945
459	CIDADE_459	ESTADO_459

Figura 27.4: Datatable do Primefaces com LazyLoad

```
<!-- paginação por demanda -->
<h:form>
    <p:datatable value="#{cidadesPaginadasMB.cidadesLazy}"
        var="cidade" paginator="true" rows="10"
        paginatorTemplate="{CurrentPageReport} {FirstPageLink}
            {PreviousPageLink} {PageLinks}
            {NextPageLink} {LastPageLink}
            {RowsPerPageDropdown}"
        rowsPerPageTemplate="5,10,15"
        style="width: 40%"
        lazy="true">
        <p:column sortBy="#{cidade.id}">
            <f:facet name="header">
                #{mensagens.cidadeId}</f:facet>
    
```

```

        </f:facet>
        <h:outputText value="#{cidade.id}" />
    </p:column>
    <p:column sortBy="#{cidade.nome}"
               filterBy="#{cidade.nome}">
        <f:facet name="header">
            #{mensagens.cidadeNome}
        </f:facet>
        <h:outputText value="#{cidade.nome}" />
    </p:column>
    <p:column sortBy="#{cidade.estado}"
               filterBy="#{cidade.estado}">
        <f:facet name="header">
            #{mensagens.cidadeEstado}
        </f:facet>
        <h:outputText value="#{cidade.estado}" />
    </p:column>
</p:dataTable>
</h:form>

```

Veja que o código da página é quase o mesmo, com um detalhe muito importante. Precisamos informar que a paginação será feita de forma `lazy`, pelo atributo `lazy="true"`. Foram adicionadas no componente `p:column` os atributos `sortBy` para habilitar a ordenação, e `filterBy` para indicar o filtro.

Toda coluna que utilizar o `sortBy` enviará esse valor na hora em que for alterado. Esse parâmetro funciona do modo Lazy e do modo normal. O `filterBy` fará o filtro do campo assim que seu valor for alterado. Essa opção também funciona tanto com Filtro Lazy como do modo normal.

Uma outra diferença está no atributo `value`. Agora indicamos o valor `#{cidadesPaginadas.cidadesLazy}`. Precisamos de algumas mudanças no nosso ManagedBean. Quem é esse `cidadesLazy`?

```

@ManagedBean
@ViewScoped

```

```

public class CidadesPaginadasMB implements Serializable {
    private List<Cidade> cidades;

    private LazyDataModel<Cidade> cidadesLazy;

    public List<Cidade> getCidades() {
        if(cidades == null){
            CidadeDAO cidadeDAO =
                AbstractManagedBean.getCidadeDAO();
            cidades = cidadeDAO.listAll();
        }

        return cidades;
    }

    public void setCidades(List<Cidade> cidades) {
        this.cidades = cidades;
    }

    public LazyDataModel<Cidade> getCidadesLazy() {
        if(cidadesLazy == null){
            cidadesLazy = new CidadeLazyList();
        }

        return cidadesLazy;
    }

    public void
        setCidadesLazy(LazyDataModel<Cidade> cidadesLazy) {
        this.cidadesLazy = cidadesLazy;
    }
}

```

Veja que o código da classe `CidadesPaginadasMB` sofreu algumas alterações. Agora `cidadesLazy` é apenas um atributo de uma classe abstrata chamada `LazyDataModel`. O `cidadesLazy` tem apenas um `new` em seu `get`, o que indica que todo o funcionamento da paginação por demanda foi delegado para a classe `CidadeLazyList`.

```
public class CidadeLazyList extends LazyDataModel<Cidade> {
```

```

private List<Cidade> cidades;

@Override
public List<Cidade> load(int posicaoPrimeiraLinha,
                           int maximoPorPagina,
                           String ordernarPeloCampo,
                           SortOrder ordernarAscOuDesc,
                           Map<String, String> filtros) {

    String ordenacao = ordernarAscOuDesc.toString();

    if(SortOrder.UNSORTED.equals(ordernarAscOuDesc)){
        ordenacao = SortOrder.ASCENDING.toString();
    }

    cidades =
        getDAO().buscaPorPaginacao(posicaoPrimeiraLinha,
                                     maximoPorPagina,
                                     ordernarPeloCampo,
                                     ordenacao, filtros);

    // total encontrado no banco de dados,
    // caso o filtro esteja preenchido dispara a consulta
    // novamente
    if (getRowCount() <= 0 ||
        (filtros != null && !filtros.isEmpty())) {
        setRowCount(getDAO().countAll(filtros));
    }

    // quantidade a ser exibida em cada página
    setPageSize(maximoPorPagina);

    return cidades;
}

private CidadeDAO getDAO() {
    return AbstractManagedBean.getCidadeDAO();
}

@Override
public Cidade getRowData(String rowKey) {
    for (Cidade cidade : cidades) {
        if (rowKey.equals(String.valueOf(cidade.getId())))
            return cidade;
    }
}

```

```
        }

        return null;
    }

@Override
public Object getRowKey(Cidade cidade) {
    return cidade.getId();
}

@Override
public void setRowIndex(int rowIndex) {
    // solução para evitar ArithmeticException
    if (rowIndex == -1 || getPageSize() == 0) {
        super.setRowIndex(-1);
    }
    else
        super.setRowIndex(rowIndex % getPageSize());
}
}
```

A classe `CidadeLazyList` apresenta diversos métodos e configurações, que veremos uma a uma. O método `public List<Cidade> load` recebe como atributo todos os parâmetros necessários para se fazer uma consulta.

O `int posicaoPrimeiraLinha` indica de qual linha do banco de dados a pesquisa deverá iniciar. Dessa forma, caso seu valor seja 10, a pesquisa iniciará a partir do décimo registro que a consulta devolver do banco de dados.

O `int maximoPorPagina` indica a quantidade a ser exibida em cada página. Caso esse valor esteja definido como 20, a cada query disparada no banco de dados apenas 20 resultados deverão ser utilizados.

O `String ordenarPeloCampo` indica em qual campo será ordenada a pesquisa. Já o `SortOrder ordenarAscOUdesc` é um

Enum do próprio Primefaces, que demonstra se é para ordenar ASCENDING (crescente), DESCENDING (decrescente) ou se é UNSORTED (sem ordenação).

```
String ordenacao = ordernarAscOUdesc.toString();

if(SortOrder.UNSORTED.equals(ordernarAscOUdesc)) {
    ordenacao = SortOrder.ASCENDING.toString();
}

cidades = getDAO().buscaPorPaginacao(posicaoPrimeiraLinha,
                                         maximoPorPagina,
                                         ordernarPeloCampo,
                                         ordenacao, filtros);
```

O DAO realizará a pesquisa no banco de dados e trará toda a informação necessária. Mais adiante, veremos o DAO .

```
// total encontrado no banco de dados,
// caso o filtro esteja preenchido dispara a consulta novamente
if (getRowCount() <= 0 ||
    (filtros != null && !filtros.isEmpty())) {
    setRowCount(getDAO().countAll(filtros));
}
```

Informa ao DataTable a quantidade de registros no banco de dados no total. Caso o filtro seja alterado, a consulta deve ser refeita. Caso contrário, a consulta será disparada apenas uma vez.

```
// quantidade a ser exibida em cada página
setPageSize(maximoPorPagina);
```

Informa ao DataTable a quantidade de registros a ser exibido por página.

```
@Override
public Cidade getRowData(String rowKey) {
    for (Cidade cidade : cidades) {
        if (rowKey.equals(String.valueOf(cidade.getId())))
            return cidade;
    }
}
```

```

        return null;
    }

    @Override
    public Object getRowKey(Cidade cidade) {
        return cidade.getId();
    }

    @Override
    public void setRowIndex(int rowIndex) {
        // solução para evitar ArithmeticException
        if (rowIndex == -1 || getPageSize() == 0) {
            super.setRowIndex(-1);
        }
        else
            super.setRowIndex(rowIndex % getPageSize());
    }
}

```

Os métodos `setRowIndex`, `getRowKey` e `getRowData` são usados quando alguma linha do `DataTable` é selecionada. Caso um `DataTable` seja utilizado sem paginação por demanda, esses métodos não são necessários. Eles apenas indicam qual a linha selecionada, o objeto que se encontra na linha e qual o ID do objeto da linha determinada.

É interessante notar que a classe `CidadeLazyList` estende da classe `org.primefaces.model.LazyDataModel` e, por consequência, acaba herdando diversos métodos que já estão implementados. É uma prática comum: um dos seus mais famosos exemplos é quando se cria um `Servlet` e a classe estendida é a `HttpServlet`.

Nos métodos do `DAO`, é sempre importante usar os recursos do seu banco de dados para buscar apenas as informações que deverão ser exibidas. No MySQL, por exemplo, é possível usar a instrução `limit`, que devolverá apenas uma determinada faixa de

resultados.

Funcionalidades ricas com JSF, segurança e otimização do JSF

O JSF, por si só, nos traz alguns recursos que permitem enriquecer as aplicações que desenvolvemos. Vimos que as bibliotecas de componentes potencializam ainda mais a criação de funcionalidades avançadas. Mas será que sempre precisamos delas? E quando precisarmos, como podemos implementá-las?

CAPÍTULO 28

FACILITANDO O USO DO AJAX

O *Ajax* é uma técnica que, entre outras coisas, permite atualizar parte de uma tela sem precisar recarregar toda a tela para o usuário. Apesar de poder ser considerado relativamente simples de se implementar, é importante manter alguns cuidados para que não se caia em algumas armadilhas, tanto do ponto de vista da usabilidade quanto do desenvolvimento.

Nada melhor do que um exemplo clássico de Ajax, que pode ser visto nas figuras:

The screenshot shows a user interface with three dropdown menus. The first menu, labeled "Selecionar um Estado:", has options "Escolha um", "Escolha um", "MG", "ES", and "RJ". The option "MG" is highlighted with a red background. The second menu, labeled "Selecionar uma Cidade:", has the option "Escolha um". The third menu, labeled "Bairro:", also has the option "Escolha um".

Figura 28.1: Utilizando Ajax para selectOne — 1

Selecionar um Estado: MG

Selecionar uma Cidade: Goverandor Valadares

Bairro: Escolha um

Figura 28.2: Utilizando Ajax para selectOne — 2

Selecionar um Estado: MG

Selecionar uma Cidade: Goverandor Valadares

Bairro: Vila Bretas

Figura 28.3: Utilizando Ajax para selectOne — 3

Note que é necessário selecionar um estado, para depois selecionar uma cidade e finalmente um bairro. Mas carregar todos esses objetos em memória pode ser um problema dependendo da quantidade de objetos retornados. Salvar todas opções escondidas na tela e utilizar JavaScript é outra opção, mas caso a quantidade de registros seja muito grande, o usuário levará mais tempo para carregar os valores.

A solução mais prática é utilizar *Ajax*, que felizmente já vem nativo no JSF 2.0. No JSF 1.x, era necessário recorrer a alguma biblioteca de componentes para conseguir usar requisições assíncronas, enquanto que a partir do JSF 2, basta utilizar a tag `f:ajax`.

```
<h:form>
    <h:panelGrid columns="3">
        <h:outputText value="#{mensagens.ajaxSelecionaEstado}: "/>
        <h:outputText value="#{mensagens.ajaxSelecionaCidade}: "/>
```

```

<h:outputText
    value="#{mensagens.ajaxBairro}: " />
<h:selectOneMenu value="#{ajaxMB.estadoSelecionado}">
    <f:selectItem itemLabel="#{mensagens.ajaxSelecione}"
                  itemValue="" />
    <f:selectItems value="#{ajaxMB.estados}" />
    <f:ajax execute="@this"
            render="selectCidade selectBairro"
            listener="#{ajaxMB.alterarEstado}" />
</h:selectOneMenu>
<h:selectOneMenu id="selectCidade">
    value="#{ajaxMB.cidadeSelecionada}">
    <f:selectItem itemLabel="#{mensagens.ajaxSelecione}"
                  itemValue="" />
    <f:selectItems value="#{ajaxMB.cidades}" />
    <f:ajax execute="@this"
            render="selectBairro"
            listener="#{ajaxMB.alterarCidade}" />
</h:selectOneMenu>
<h:selectOneMenu id="selectBairro">
    value="#{ajaxMB.bairroSelecionado}">
    <f:selectItem itemLabel="#{mensagens.ajaxSelecione}"
                  itemValue="" />
    <f:selectItems value="#{ajaxMB.bairros}" />
</h:selectOneMenu>
</h:panelGrid>
</h:form>

```

O componente `f:ajax` é responsável por executar a chamada assíncrona. Toda vez que o `selectOne` for alterado, ele enviará o valor definido em `execute="@this"`. Nesse caso, o `@this` significa o componente atual e o `selectOne` está envolvendo o `f:ajax`.

O atributo `render` indica quais componentes serão atualizados quando a resposta da requisição assíncrona for devolvida. No `selectOne` do estado, indicamos `selectCidade` `selectBairro` como tendo de sofrer atualização. Ou seja, quando o estado for selecionado, os campos de cidade e de bairro serão recarregados, para mostrar as informações condizentes com a

escolha do usuário. A figura a seguir mostra como funciona uma requisição Ajax.

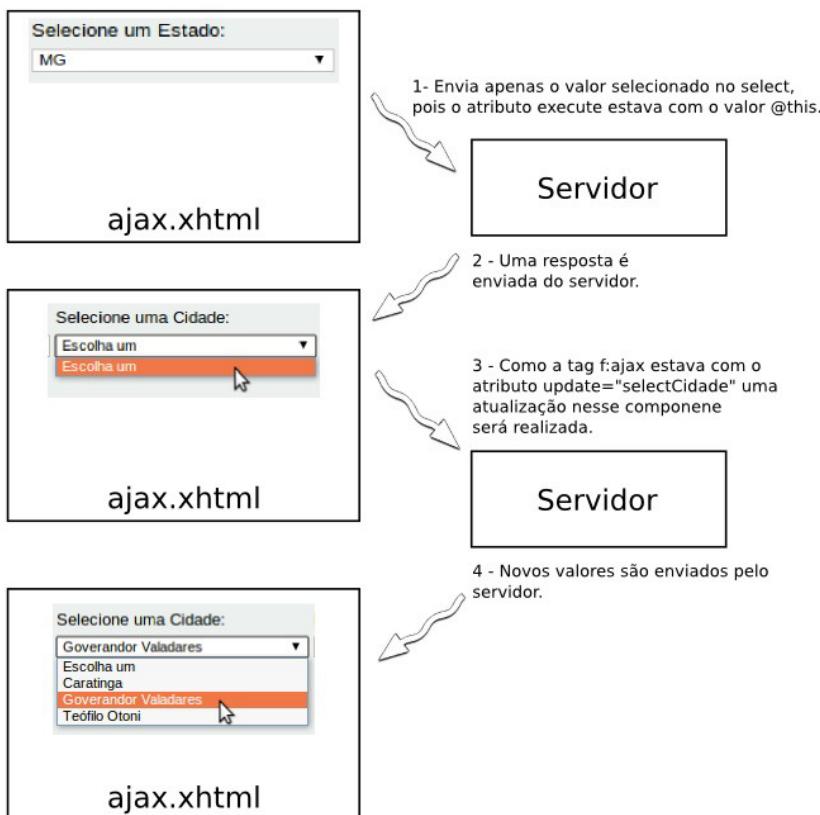


Figura 28.4: Requisição Ajax

O ManagedBean apenas recebe o valor selecionado em um `selectOne` para decidir qual o valor a ser exibido no próximo.

28.1 SEMPRE INDIQUE QUE A REQUISIÇÃO ESTÁ ACONTECENDO

Devido ao fato de a requisição Ajax ser feita de forma assíncrona, em alguns momentos pode não haver a certeza de que algo está acontecendo ou sendo processado. Com isso, o usuário da aplicação pode ter uma reação que não seja a adequada, como por exemplo, realizar repetidas vezes a ação até que algo aconteça, dessa forma, enfileirando várias requisições. Uma boa prática é sempre indicar que uma ação está sendo executada, seja através de uma imagem ou uma animação.

Uma solução interessante é apresentada pelo Primefaces (também presente em outras bibliotecas): o componente chamado `p:ajaxStatus`.

```
<p:ajaxStatus onstart="statusDialog.show();"  
              onsuccess="statusDialog.hide();"/>  
  
<p:dialog modal="true" widgetVar="statusDialog" header="Status"  
              draggable="false" closable="false">  
    <p:graphicImage library="gifs" name="ajaxloadingbar.gif" />  
</p:dialog>
```

O componente `p:ajaxStatus` automaticamente detectará que uma chamada Ajax foi iniciada e exibirá um dialog para o usuário. Ao final do processamento Ajax, o componente fechará o dialog automaticamente, como demonstrado na figura:

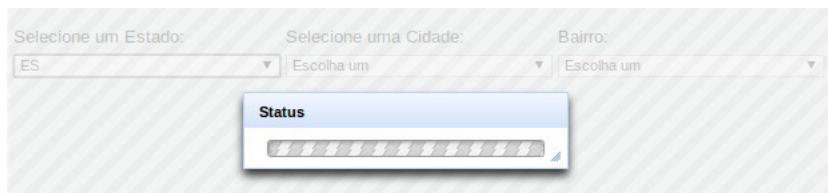


Figura 28.5: Feedback de ação iniciada

Dê mensagens de feedback para o usuário

Sempre dê o feedback do que está acontecendo para o usuário. Uma forma alternativa à imagem é usar simples mensagens que indiquem ao usuário o que está acontecendo. Toda vez que uma ação é executada, o usuário naturalmente espera por um resultado. Mas como ele vai saber se algo aconteceu e o quê, sem que nenhuma mensagem indique isso a ele? O componente Growl explicado no capítulo *Primefaces* mostra como ele funciona.

28.2 PREVINA-SE DAS VÁRIAS AÇÕES DO USUÁRIO EM REQUISIÇÕES ASSÍNCRONAS

Caso a ação disparada demore mais do que o esperado para finalizar e o usuário dependa do resultado dessa execução para continuar alguma ação, podemos mostrar para ele uma mensagem indicando a demora do processamento.

Na maioria das vezes, é necessário desabilitar o botão, pois o usuário incansavelmente apertará o mesmo botão milhares de vezes. Para fazer isso, teremos de usar um pequeno código jQuery.

```
<h:form>
    <h:commandButton value="#{mensagens.ajaxBotaoTravado}">
        <f:ajax execute="@form"
            render="@form"
            listener="#{ajaxMB.chamadaAjax}"
            onevent="desabilitarBotao" />
    </h:commandButton>
</h:form>
<script type="text/javascript">
    function desabilitarBotao(evento) {
        var botao = evento.source;
        var statusAjax = evento.status;

        switch (statusAjax) {
            case "begin":
                botao.disabled = true;
```

```
        break;

    case "complete":
        break;

    case "success":
        botao.disabled = false;
        break;
    }
}

</script>
```

Pela função `desabilitarBotao`, o botão será desabilitado ao iniciar a chamada Ajax e, ao final da chamada, será habilitado novamente. As figuras a seguir mostram como ficará esse botão.

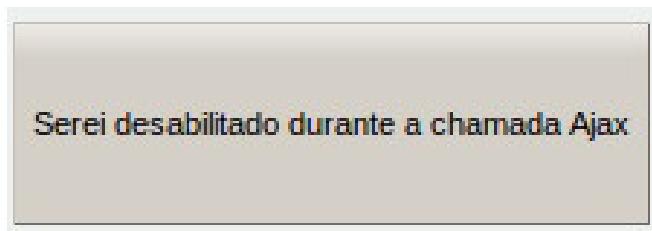


Figura 28.6: Desabilitando botão

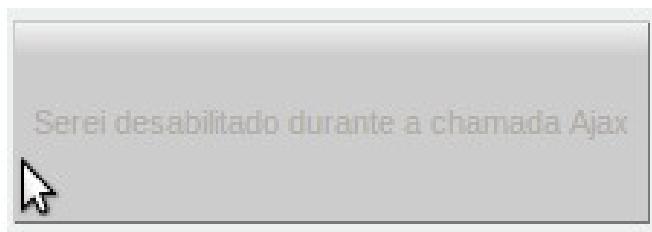


Figura 28.7: Desabilitando botão

28.3 CUIDADO AO USAR MANAGEDBEANS REQUESTSCOPED COM AJAX

Quando usamos requisições *Ajax*, precisamos tomar alguns cuidados com os ManagedBeans RequestScoped . Veja o exemplo a seguir, em que temos um cenário clássico de uso de requisições assíncronas.

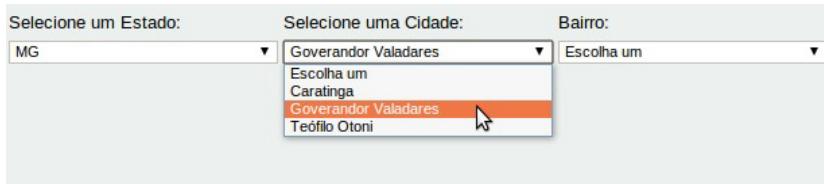


Figura 28.8: Utilizando Ajax para selectOne

Na figura, é possível ver um exemplo no qual, quando cada `selectOne` for atualizado, um outro `selectOne` será populado. Note que uma chamada *Ajax* a um ManagedBean do tipo `RequestScoped` sempre precisa enviar os dados necessários para o processamento, já que o ManagedBean não guardará as informações entre as diferentes requisições.

Nesse caso, seria necessário enviar o `Estado` selecionado para habilitar o `selectOne` de cidades. Ao escolher uma `Cidade`, seria necessário enviar novamente `Estado` e `Cidade` para poder calcular os valores do `selectOne` de Bairros.

Note que, com escopo de `request` , poderá funcionar sem problemas, mas a desvantagem é esse trabalho de enviar todos os valores necessários a cada chamada. Poderíamos evitar essa necessidade através de um bean `ViewScoped` ou `ConversationScoped` .

CAPÍTULO 29

INTERNACIONALIZAÇÃO E LOCALIZAÇÃO DA SUA APLICAÇÃO

Muitas vezes, desenvolvemos aplicações que serão usadas por pessoas de vários lugares do mundo, com diversas características culturais, sendo o idioma uma das mais marcantes. Dessa maneira, o ideal é que nossa aplicação se adapte às diferentes necessidades de locais distintos.

Para isso, precisamos que haja algum tipo de suporte do framework para adaptarmos a aplicação a essas necessidades. Isso é o que chamamos de **internacionalização**.

O JSF provê internacionalização para nós por meio de simples configurações e arquivos `.properties`, que conterão todas as mensagens do sistema. A tradução das diversas mensagens para diferentes idiomas, adaptação de sinais monetários, abreviações e vários outros recursos são conhecidos como **localização**.

Para começarmos a adaptar a aplicação, vamos colocar os arquivos com as mensagens. A figura a seguir mostra como ficaria o arquivo utilizando a *IDE Eclipse*.

-->

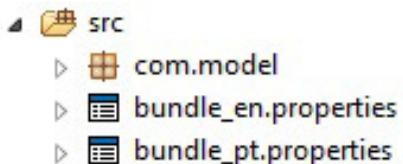


Figura 29.1: Arquivos com mensagens do sistema no Eclipse

Dentro desses arquivos, podemos ter as mensagens junto de suas respectivas chaves identificadoras:

```
# arquivo bundle_pt.properties  
bemVindo=Seja Bem Vindo  
index=Pagina Principal
```

```
# arquivo bundle_en.properties  
bemVindo=You are welcome  
index=Welcome Page
```

O próximo passo é configurar a existência dos arquivos da internacionalização para o JSF. Podemos fazer isso pelo `faces-config.xml`, no qual adicionamos a tag `locale-config` para indicar qual será o idioma padrão, que em nosso caso será `pt`, indicando o português.

Definimos também o nome arquivo com os idiomas, que chamamos de `bundle`. Com isso, teremos para cada idioma um arquivo `bundle_xx.properties`, em que podemos substituir `xx` pelo código do idioma. Vamos definir também uma variável chamada `mensagens`, que é configurada também no arquivo `faces-config.xml`.

```
<application>  
  <locale-config>
```

```
<default-locale>pt</default-locale>
</locale-config>
<resource-bundle>
    <base-name>bundle</base-name>
    <var>mensagens</var>
</resource-bundle>
</application>
```

É possível especificar ainda melhor qual a linguagem do arquivo se seu nome for `bundle_pt_BR` e ele pode ser aplicado para as outras linguagens. Nesse caso, estamos indicando a variação do português a ser usado, no caso, o do Brasil.

Outro comportamento importante de se prestar atenção é que, caso o usuário esteja em uma linguagem não definida no sistema, o JSF escolherá a linguagem padrão, definida na tag `default-locale`.

29.1 PERMITA QUE O USUÁRIO MUDE O IDIOMA

Sempre que o usuário acessar a aplicação, um idioma será escolhido para que o conteúdo seja mostrado. Não necessariamente a língua padrão será a que o usuário gostaria. Por isso, precisamos permitir que ele escolha a que preferir.

Com pequenos ajustes na aplicação, é possível fazer com que a linguagem seja escolhida pelo usuário. Primeiro, é necessário criar uma opção para ele selecionar qual é o idioma desejado. Podemos fazer através de um `selectOneMenu`, ou então, disponibilizando bandeirinhas das línguas disponíveis, que, quando clicadas, muda para o idioma adequado.

```
<h:outputText value="#{mensagens.internacionalizacaoTexto}" />
<h:form>
```

```

<h:selectOneMenu value="#{usuarioMB.linguaEscolhida}">
    <f:selectItem itemValue="pt"
        itemLabel="#{mensagens.internacionalizacaoPT}"/>
    <f:selectItem itemValue="en"
        itemLabel="#{mensagens.internacionalizacaoEN}"/>
</h:selectOneMenu>
<h:commandButton action="#{usuarioMB.alterarIdioma}"
    value="#{mensagens.internacionalizacaoTrocar}" />
</h:form>

```

É possível notar que o `selectOneMenu` aponta para um `ManagedBean` que define qual a linguagem a ser selecionada. E existe uma `action`, chamada `alterarIdioma`, que fará todo o trabalho para nós.

```

@ManagedBean
@SessionScoped
public class UsuarioMB implements Serializable {
    private String linguaEscolhida = "pt";
    private Locale locale;

    public String alterarIdioma() {
        locale = new Locale(linguaEscolhida);
        FacesContext instance =
            FacesContext.getCurrentInstance();
        instance.getViewRoot().setLocale(locale);
        return null;
    }

    public Locale getLocale() {
        if(locale == null){
            locale = new Locale(linguaEscolhida);
        }
        return locale;
    }

    // getters e setters necessários
}

```

Note que o `ManagedBean` `UsuarioMB` tem os atributos para informar qual o `Locale` atual, e uma `String` que contém uma

língua padrão, utilizada no `selectOne`. Dentro do método `alterarIdioma`, a chamada para o método `setLocale` é feita, passando como parâmetro o idioma `Locale` que o usuário escolheu.

Uma outra opção é envolver um pedaço de um texto por uma linguagem qualquer. Caso em um ponto específico da página seja necessário ter outra língua, basta utilizar o atributo `locale` do componente `f:view`.

```
<f:view locale="#{usuarioMB.locale}">
    <!-- códigos do sistema aqui-->
</f:view>
```

BOA PRÁTICA

Tenha em mente que colocar todo o texto da sua aplicação em arquivos é uma boa prática mesmo que se tenha apenas um idioma. Podemos considerar a internacionalização como boa prática pelos seguintes fatos:

- Facilita a alteração de textos. Caso seja necessário trocar a palavra de Carro para Automóvel, todos os textos se encontram em um arquivo apenas. Não será necessário entrar em cada página do sistema.
- Caso após um deploy se perceba que existe uma mensagem errada, bastaria alterar o arquivo de propriedades. Não seria necessário editar as páginas do sistema.
- Se fosse necessário criar uma nova linguagem para o sistema, bastaria traduzir o arquivo. Não seria necessária alteração em todas as páginas.

Para finalizar esse assunto, é possível sobrescrever as mensagens default do JSF. Caso um campo esteja marcado como `required=true`, uma mensagem em inglês aparecerá: *ValidationError: Value is required*. Para um sistema em português, essa mensagem pode ser bastante desconfortável para o usuário.

Para alterar as mensagens padrões do JSF, basta adicionar a seguinte configuração ao `faces-config.xml` : `<message-bundle>bundle</message-bundle>` . Com essa configuração, o

JSF procurará no arquivo de bundle pelas chaves utilizadas pelo JSF.

```
<application>
<locale-config>
    <default-locale>pt</default-locale>
</locale-config>
<resource-bundle>
    <base-name>bundle</base-name>
    <var>mensagens</var>
</resource-bundle>
<!-- configuração adicionada -->
<message-bundle>bundle</message-bundle>
</application>

# bundle_pt.properties
javax.faces.component.UIInput.REQUIRED=
    Campo necessário não preenchido

# bundle_en.properties
javax.faces.component.UIInput.REQUIRED=A required field is Empty
```

Basta adicionar a linha com a chave de campo obrigatória ao arquivo de mensagens, no projeto do livro chamado de bundle, e pronto. O JSF identificará qual a língua do usuário e procurará no respectivo arquivo de mensagens.

Veja nas figuras seguintes como ficará essa troca dinâmica da língua do sistema.

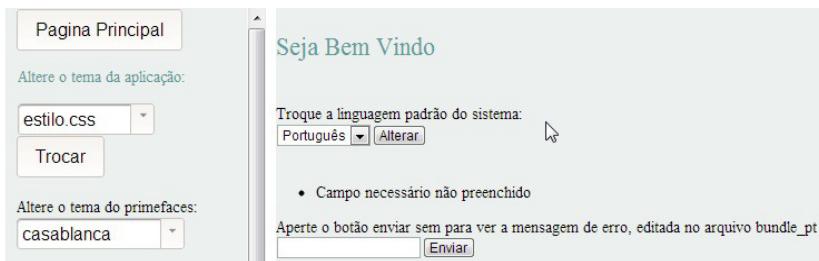


Figura 29.2: Mensagens em português

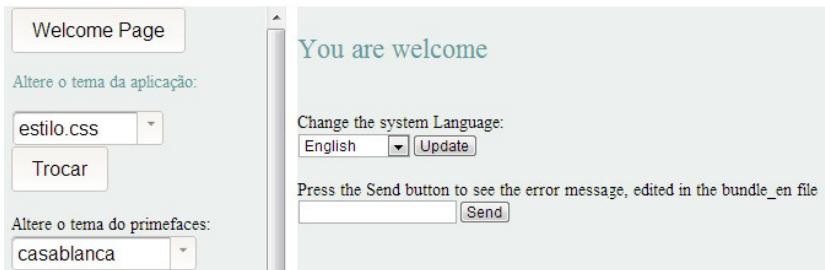


Figura 29.3: Mensagens em inglês

No projeto do livro, algumas chaves foram traduzidas, por isso que, mesmo alterando para inglês, nem todas as palavras serão traduzidas. Para utilizar os valores descritos no arquivo de propriedades, basta utilizar `#{{mensagens.bemVindo}}`. O JSF escolherá qual língua utilizar de acordo com as configurações e o idioma do usuário.

CAPÍTULO 30

UTILIZANDO RECURSOS DENTRO DE UM CONVERTER

Criar conversores é uma tarefa bastante recorrente em projetos JSF. No entanto, uma grande dificuldade é que, até a versão 2.1 do JSF, não era possível trabalhar com injecção de dependências dentro desses componentes. Dessa forma, tendemos a ter códigos complexos e de difícil manutenção dentro do Bean. Mas como podemos fazer pra mantermos nosso código limpo e elegante dentro dos conversores?

30.1 ACESSE UM MANAGEDBEAN PROGRAMATICAMENTE ATRAVÉS DE EXPRESSION LANGUAGE

Sempre que precisamos, nas páginas `xhtml`, temos acesso a diversos recursos através das *Expression Languages*. Seria interessante se de alguma maneira também tivéssemos como acessá-los dentro dos conversores. E se houvesse uma possibilidade de usarmos a *Expression Language* de dentro do conversor?

```
@FacesConverter(value = "bairroConverter")
```

```

public class BairroConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext fc, UIComponent uic,
        String key) {
        FacesContext context =
            FacesContext.getCurrentInstance();
        ELContext elContext = context.getELContext();
        ELResolver elResolver = elContext.getELResolver();

        BairroMB bairroMB = (BairroMB)
            elResolver.getValue(elContext, null, "BairroMB");
        return bairroMB.find(Integer.valueOf(key));
    }

    // outros métodos omitidos
}

```

O código da classe `BairroConverter` mostra como buscar um ManagedBean e utilizá-lo para realizar as rotinas necessárias. Note que o método `find` do ManagedBean teria acesso às classes injetadas.

Ao utilizar essa abordagem, o ideal é ter uma classe para abstrair todos os passos necessários para se acessar um ManagedBean.

```

// imports omitidos
public class ManagedBeanLocator {
    public static AbstractMB find(String managedBean) {
        FacesContext context =
            FacesContext.getCurrentInstance();
        ELContext elContext = context.getELContext();
        ELResolver elResolver = elContext.getELResolver();
        return (AbstractMB) elResolver.getValue(elContext, null,
            managedBean);
    }
}

```

E para utilizar o método bastaria fazer:

```
BairroMB bairroCrudMB =
```

```
(BairroMB) ManagedBeanLocator.find("BairroMB");
```

CDI COM JSF

A vantagem do CDI é injetar as dependências e utilizar aquele recurso sem que a classe tenha de se preocupar em instanciá-lo. Um bom exemplo seria utilizar o EntityManager do JPA dentro de um ManagedBean.

```
@ManagedBean(name="usuarioLoginMB")
@SessionScoped
public class UsuarioLoginMB implements Serializable {
    private String login;
    private String senha;

    @PersistenceUnit(unitName="meuPU")
    private EntityManagerFactory emFactory;

    @Resource
    private UserTransaction userTransaction;

    public String login() {
        // ...
    }
}
```

No código do ManagedBean `UsuarioLoginMB`, é possível ver que temos injetado um `EntityManagerFactory` e um objeto do tipo `UserTransaction`. Note que o ManagedBean não tem ideia de onde vieram essas duas instâncias, ele apenas utiliza esses objetos.

A grande vantagem do CDI é que um desacoplamento é criado entre classes. Note que não foi necessário passar configuração alguma relacionada ao arquivo `persistence.xml`. O próprio servidor localizou essas informações e criou uma instância do `EntityManagerFactory`. É possível também injetar um EJB e outros recursos dentro de um ManagedBean.

CAPÍTULO 32

EVITE O CROSS SITE SCRIPTING EM SEU SISTEMA

Em um campo de texto de um formulário, é possível que o usuário digite qualquer informação, mesmo não sendo a solicitada pelo campo. Por exemplo, em um campo cuja descrição é "Nome", o usuário pode digitar a "Idade", o seu "Endereço" ou qualquer outra informação que ele queira. Sendo assim, já que nesse campo ele pode digitar qualquer informação que seja alfanumérica, o que o impede de preencher esse campo com um pequeno código JavaScript? Algo como:

```
<script>alert('ola!');</script>
```

Com essa informação digitada no campo, ela será cadastrada no banco de dados. Até aí, tudo bem. O problema acontece quando queremos mostrar o nome na tela.

```
<h:outputText value="#{cliente.nome}" />
```

O conteúdo do nome que, no caso é o pequeno trecho de código JavaScript, será inserido dentro do HTML. Isso vai fazer com que durante a exibição da tela, a mensagem de alerta apareça, ou seja, o código JavaScript cadastrado pelo usuário será executado

posteriormente. Quão perigoso isso pode ser?

Muitos podem imaginar que isso é inofensivo, pois o usuário não vai digitar algo dessa natureza no campo. No entanto, essa brecha pode ser usada para um ataque poderoso, até mesmo para tirar o sistema do ar.

Por exemplo, já que é possível introduzir código JavaScript que em algum momento futuro será executado, o que impede a pessoa de cadastrar um *looping* infinito que envia requisições assíncronas para a aplicação? Pronto, sua aplicação se tornou alvo de um ataque.

Atacar aplicações através da injeção de código JavaScript é uma técnica chamada *Cross Site Scripting*. A solução para esse problema é extremamente simples. Para exibir as informações, utilize a tag `h:outputText`. Ela já vem com um mecanismo de proteção, pelo atributo `escape`.

O `true` já é o valor padrão utilizado, e ele fará com que um código JavaScript seja apenas impresso, mas não executado. É uma solução extremamente simples que evita dores de cabeça com ataques a que muitos sistemas estão vulneráveis.

CAPÍTULO 33

OTIMIZANDO A NAVEGAÇÃO E PERFORMANCE

O JSF dispara seu ciclo de vida quando uma navegação é disparada utilizando `h:commandLink` e o `h:commandButton`. No entanto, quando apenas queremos trocar de página sem disparar ação alguma, não precisamos que todo o ciclo seja executado. A saída para essa situação é usar o componente `h:outputLink`, que gera apenas um link convencional.

Seu uso é extremamente simples, bastando indicar no atributo `outcome` qual é a página de destino:

```
<h:outputLink outcome="pagina.xhtml" value="Pagina 01"/>
```

A vantagem de utilizar o `h:outputLink` para navegação é que ele não dispara os ciclo de vida completo do JSF, tornando a navegação um processo mais leve.

Debug e inspeção de aplicações

Durante o desenvolvimento das aplicações, é comum inspecionarmos o que estamos fazendo por meio das mensagens de debug e comentários que deixamos pelo código. Mas como podemos fazer esse debug de uma maneira que não polua nosso código, ou atrapalhe a nossa aplicação?

Nessa parte, você aprenderá dicas de desenvolvimento e como resolver erros comuns do dia a dia do desenvolvimento com JSF.

LIMPEZA DE COMENTÁRIOS E DEBUG

34.1 ESCONDA OS COMENTÁRIOS DA PÁGINA

É comum desenvolvedores olharem o código-fonte de sistemas e sites que não foram feitos por eles. No entanto, geralmente, ao abrir o código-fonte de uma página, é possível encontrar comentários no HTML resultante, que além de poluir a saída, tornam a resposta mais pesada.

Com o JSF, existe um modo simples de esconder todos os comentários que são colocados nos XHTML. Basta adicionar uma configuração no `web.xml` e os comentários não serão mais exibidos.

```
<!-- adicionar ao web.xml -->
<context-param>
    <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
    <param-value>true</param-value>
</context-param>
```

É considerada boa prática esconder os comentários do código XHTML. Primeiro, pelo fato de que comentários em HTML são enviados para o navegador, que o ignora, porém aumenta o

tamanho do conteúdo da resposta a ser dada. E segundo, pois pessoas podem burlar de alguma maneira o sistema desenvolvido caso hajam comentários que exponham a existência de alguma falha na implementação.

Um desenvolvedor poderia escrever `<!-- corrigindo falha da data -->` e, a partir dessa mensagem, ele poderia descobrir qual falha é essa. Isso poderia acarretar em uma mensagem simples de erro, até falhas em pagamentos, prejuízo financeiro ou queda do servidor em casos mais extremos.

34.2 DEBUG DOS COMPONENTES

Muitas vezes fazemos manipulações na tela e perdemos o controle do que está ou não na árvore de componentes, por exemplo, ou então qual é o estado dos componentes e assim por diante. Para resolver essas e outras questões sobre a situação dos componentes, podemos utilizar a tag `<ui:debug>`.

Para acessar os dados fornecidos pelo `<ui:debug />`, basta utilizar o atalho `ctrl + shift + d`, e aparecerá uma tela como a mostrada nas figuras:

A paginação por Demanda ou Lazy Pagination carregará apenas o que será exibido. Desse modo será pouparado a quantidade de registros trazidos do banco de dados. Se a consulta retornasse 150.000 registros, haveria um estouro de memória. Com

The screenshot shows a JSF application running on localhost:8080. At the top, there is a navigation bar with icons for search, refresh, and other functions. Below it is a header bar with the text '(1 of 100)' and a page number selector from 1 to 10. The main content area contains a table with three columns: 'Id', 'Nome', and 'Estado'. The table has 10 rows, each containing an ID from 0 to 9 and a corresponding name and state. To the right of the table is a 'Debug Output' window titled 'Debug - /paginas/parte5/dataTablePaginacaoPorDemanda'. The output window shows the URL '/paginas/parte5/dataTablePaginacaoPorDemanda.xhtml' and two expandable sections: '+ Component Tree' and '+ Scoped Variables'.

Id	Nome	Estado
0	CIDADE_0	ESTADO_0
1	CIDADE_1	ESTADO_1
2	CIDADE_2	ESTADO_2
3	CIDADE_3	ESTADO_3
4	CIDADE_4	ESTADO_4
5	CIDADE_5	ESTADO_5
6	CIDADE_6	ESTADO_6
7	CIDADE_7	ESTADO_7
8	CIDADE_8	ESTADO_8
9	CIDADE_9	ESTADO_9

Figura 34.1: Função Debug do JSF

- Component Tree

```
<UIViewRoot id="j_id1" inView="true" locale="pt" renderKitId="HTML_BASIC"
rendered="true" transient="false" viewId="/paginas/parte5
/dataTablePaginacaoPorDemanda.xhtml">
| javax.faces.location_HEAD
| <ComponentResourceContainer id="javax_faces_location_HEAD" inView="true"
| rendered="true" transient="false">
|   | <UIOutput id="j_idt4" inView="false" rendered="true"
|   | transient="false"/>
|   | <UIOutput inView="true" rendered="true" transient="false"/>
|   | </ComponentResourceContainer>
|   | <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE html PUBLIC "-//W3C//DTD
|   | XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
|   | transitional.dtd">
|   | <html xmlns="http://www.w3.org/1999/xhtml">
|   | <UIOutput id="j_idt3" inView="true" rendered="true" transient="false">
|   |   | <UIOutput id="j_idt5" inView="true" rendered="true"
|   |   | transient="false"/>
|   | </UIOutput>
|   | <UIOutput id="j_idt6" inView="true" rendered="true" transient="false">
|   |   | <UIDebug hotkey="D" id="j_idt7" inView="true" rendered="true"
|   |   | transient="true"/>
```

Figura 34.2: Função Debug do JSF

+ Component Tree

- Scoped Variables

Request Parameters

Name	Value
None	

View Attributes

Name	Value
cidadesPaginadas	parte5.CidadesPaginadas@321b1228

Request Attributes

Name	Value
None	

Flash Attributes

Name	Value
None	

Figura 34.3: Função Debug do JSF

Entre outras coisas, o *debug* é útil para resolver problemas de Ajax quando um *update* não funciona. Outra situação na qual o *debug* pode ser usado é para ver se os valores enviados por um ManagedBean estão chegando à *view*.

Apenas tenha o cuidado de colocar o `<ui:debug />` ao final de sua página. Caso contrário, ele disparará os *getters* e *setters* dos ManagedBean monitorados, e assim poderá ocasionar comportamentos inesperados.

CAPÍTULO 35

ORGANIZE FUNCIONALIDADES POR AMBIENTE DO PROJETO

O JSF entende que cada projeto passa por **estágios**, e cada um possui características específicas que ajudam de diferentes modos. Hoje, no JSF, é possível encontrar os seguintes estágios: `Development` , `UnitTest` , `SystemTest` , `Production` e `Extension` .

Os estágios `SystemTest` e `UnitTest` permitem que ações sejam configuradas para serem executadas em um `Listener` . Caso algum dado, objeto ou configuração sejam necessários apenas para teste, basta configurar no `web.xml` qual o estágio atual da aplicação.

O método `configurarAmbiente` mostra como o estágio poderia ser configurado em um `ManagedBean` do tipo `@ApplicationScoped` . Desse modo, bastaria chamar o método para que a aplicação se comportasse dependendo da configuração.

```
public void configurarAmbiente() {  
  
    FacesContext facesContext =  
        FacesContext.getCurrentInstance();
```

```
Application application = facesContext.getApplication();
ProjectStage projectStage = application.getProjectStage();

if (projectStage.equals(ProjectStage.Development)) {

    // realiza ações de desenvolvimento

} else if (projectStage.equals(ProjectStage.Production)) {

    // realiza ações de produção

} else if (projectStage.equals(ProjectStage.SystemTest)) {

    // realiza ações de testes de sistema

} else if (projectStage.equals(ProjectStage.UnitTest)) {

    // realiza ações de testes de units

}
}
```

Os estágios `Development` e `Production` são os mais utilizados. É normal que, ao se desenvolver com JSF, erros estejam acontecendo e nenhuma mensagem seja exibida. Às vezes, são erros tão simples que, caso alguma mensagem fosse exibida rapidamente, esses erros seriam resolvidos.

Uma das vantagens do estágio `Development` é que o JSF adiciona automaticamente `h:messages` nas páginas para exibir mensagens de erro, o que é muito prático. Já o estágio de `Production` (que é o valor padrão) tem uma característica oposta ao `Development`: ele esconde erros que poderiam ser exibidos ao usuário final.

As implementações do JSF podem realizar ações para otimizar cada estágio escolhido para o projeto. O MyFaces, por exemplo, no estágio `Development` faz o deploy dos arquivos de JavaScript

separadamente. Já no estágio Production , ele vai reduzir, minificar o arquivo e compactar os vários arquivos, a fim de otimizar a performance da aplicação.

Para configurar o estágio da aplicação, basta adicionar o parâmetro javax.faces.PROJECT_STAGE no web.xml :

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

CAPÍTULO 36

REFRESH AUTOMÁTICO DOS ARQUIVOS

Em alguns momentos, alterações são realizadas diretamente nas páginas do projeto (arquivo `xhtml/jsp`), mas não são refletidas no código. É possível informar ao JSF que ele deve verificar se algum arquivo de um projeto foi alterado. Com isso, ele atualizará o arquivo alterado no projeto em tempo de execução no servidor.

Para ativar essa funcionalidade, basta adicionar a configuração `javax.faces.FACELETS_REFRESH_PERIOD` no arquivo `web.xml`.

```
<context-param>
    <param-name>javax.faces.FACELETS_REFRESH_PERIOD</param-name>
    <param-value>2</param-value>
</context-param>
```

O tempo da configuração `javax.faces.FACELETS_REFRESH_PERIOD` é tratado em segundos.