

**LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO
DO RIO GRANDE DO NORTE**

Sidney Mendes de Oliveira

**DESENVOLVIMENTO DE COMPONENTES JSF PARA CRIAÇÃO DE
APLICAÇÕES RIA COM AJAX**

**DESENVOLVIMENTO DE COMPONENTES JSF PARA CRIAÇÃO DE
APLICAÇÕES RIA COM AJAX**

Monografia apresentada ao Conselho
Especialização em Desenvolvimento
de Sistemas Corporativos da
Faculdade Natalense para o
Desenvolvimento do Rio Grande do
Norte como requisito obrigatório para
obtenção do grau de Especialista em
Desenvolvimento de Sistemas
Corporativos.

Orientador:
Gleydson Lima, MSc.

**NATAL
2007**

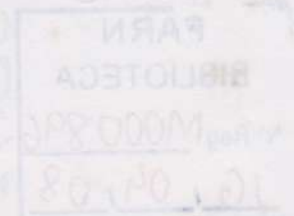
SIDNEY MENDES DE OLIVEIRA

DESENVOLVIMENTO DE COMPONENTES JSF PARA CRIAÇÃO DE APLICAÇÕES RIA COM AJAX

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte como requisito obrigatório para obtenção do grau de Especialista em Desenvolvimento de Sistemas Corporativos.

Orientador:
Gleydson Lima, MSc.

Natal
2007



Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Oliveira, Sidney Mendes de

Desenvolvimento de componentes JSF para criação de aplicações RIA
com AJAX / Sidney Mendes de Oliveira. – Natal: FARN, 2007.
96f.

Orientador: Gleydson Lima (M.Sc.)

Monografia (Especialização em Desenvolvimento de Sistemas
Corporativos) – Faculdade Natalense para o Desenvolvimento do Rio
Grande do Norte.

1. Linguagem de Programação – Monografia. 2. Componentes JSF com
AJAX – Monografia. 3. Desenvolvimento de Páginas WEB – Monografia. I.
Título.

RN/FARN/BC

CDU 004.43

RB. 30-705

FARN

BIBLIOTECA

Nº Reg M000896

16, 04, 08

004.43

0461d

2007

EX 01

SIDNEY MENDES DE OLIVEIRA

DESENVOLVIMENTO DE COMPONENTES JSF PARA CRIAÇÃO DE
APLICAÇÕES RIA COM AJAX

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas
Corporativos da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte,
aprovada em ____ de ____ de ____, pela banca examinadora composta
pelos seguintes membros:

Gleydson de Azevedo F. Lima

Gleydson Lima, MSc.
Orientador

AGRADECIMENTOS

Eu agradeço e dedico este trabalho ao Senhor, meu Deus, em reconhecimento por toda a sua misericórdia e bondade que tem sempre me acompanhado em todos os dias da minha vida.

"Porque dele e por ele, e para ele, são todas as coisas; glória, pois, a ele eternamente. Amém." (Rm 11:36)

Agradeço também a todos que intercederam junto ao Senhor por mim em oração.

"A oração feita por um justo pode muito em seus efeitos." (Jd 5:16)

Por fim, agradeço ao apoio e suporte da minha mãe, dos meus irmãos e familiares e de minha amada esposa, bem como ao suporte provido pela empresa a qual faço parte, a Natalsoft.

"Honra a teu pai e a tua mãe, para que se prolonguem os teus dias na terra que o Senhor teu Deus te dá." (Êx 20:12)

"A casa e os bens são herança dos pais; porém do SENHOR vem a esposa prudente." (Pv 19:14)

Senhor Jesus, obrigado pelo teu sacrifício vicário na cruz do calvário por mim.

"Porque Deus amou o mundo de tal maneira que deu o seu Filho unigênito, para que todo aquele que nele crê não pereça, mas tenha a vida eterna." (Jo 3:16)

RESUMO

O presente trabalho demonstra como pode ser encapsulado o AJAX em componentes reutilizáveis, ágeis e interativos para construir aplicações RIA produtivas. Para isso, é desenvolvida a criação de um framework JavaScript que encapsula as técnicas AJAX para aumentar a responsividade da aplicação. A seguir, este framework é encapsulado em um componente escrito usando o framework JavaServer Faces (JSF). O produto final será uma nova marcação JSF, a ser inserida na página Web da aplicação RIA de uma forma bastante simples pelo desenvolvedor da aplicação Web, ou designer, provendo maior produtividade e desempenho, usando componentes customizados e adequados às necessidades da aplicação.

Palavras-Chave: JSF. Componentes JavaServer Faces. AJAX. JavaScript.

LISTA DE FIGURAS

ABSTRACT

This work shows how AJAX can be encapsulated into reusable, interactive and smart components to create productive RIA applications. A JavaScript framework is developed in order to have AJAX techniques within that, improving the application's reliability. Forward, this framework is encapsulated into a component built in JavaServer Faces framework (JSF). The final product will be a new JSF tag, that will be inserted into a web page of the RIA application in a very simple way, by the developer or designer, where the requirements of performance and productivity will be higher, by using customized components that would certainly fit the needs of the application in a regular basis.

Figura 9. Declaração de um managed bean..... 44

Figura 10. Barra de navegação JSF..... 44

Keywords: JSF. JavaServer Faces. Components. AJAX. JavaScript.

Figura 11. Exemplo JSF EL..... 47

Figura 12. Exemplo de registro de um ActionListener..... 47

Figura 13. Exemplo de uso de um componente JSF em uma página..... 49

Figura 14. Declarações no faces-config.xml do componente Spinner..... 53

Figura 15. Protótipo HTML do componente Spinner..... 54

Figura 16. Protótipo parametrizado do componente Spinner..... 55

Figura 17. Definição da declaração do componente Spinner..... 55

Figura 18. Métodos de HtmlRenderer usados para a codificação..... 58

Figura 19. Método HtmlRenderer writeStyleResource..... 58

Figura 20. Método HtmlRenderer _getStyleResourcesAlreadyWritten..... 59

Figura 21. Constantes chave para os recursos..... 59

Figura 22. Método de codificação do componente Spinner..... 61

Figura 23. Script de validação da entrada de dados no cliente..... 61

Figura 24. Diferentes métodos para a codificação do Spinner..... 63

Figura 25. Método HtmlSpinnerRenderer decode..... 65

Figura 26. Chave UISpinner..... 65

Figura 27. Declarações no arquivo faces-config.xml para o exemplo do Spinner..... 67

LISTA DE LISTAGENS

Listagem 1. Criação do objeto XMLHttpRequest com <i>factory method</i>	25
Listagem 2. Página usando FactoryXMLHttpRequest.....	28
Listagem 3. Singleton JavaScript.....	33
Listagem 4. Propriedades para reconhecimento do browser.	33
Listagem 5. Factory Method reformulado.....	34
Listagem 6. Gerenciador de eventos <i>Cross-Browsing</i>	35
Listagem 7. Classe HttpRequest.....	38
Listagem 8. Exemplo de uso do objeto HttpRequest.	39
Listagem 9. Declaração de um <i>managed bean</i>	44
Listagem 10. Regra de navegação estática.....	44
Listagem 11. Exemplo JSF EL.....	47
Listagem 12. Exemplo de registro de um ActionListener.....	47
Listagem 13. Exemplo de uso de um componente JSF em uma página.....	49
Listagem 14. Declarações no faces-config.xml do componente Spinner.....	53
Listagem 15. Protótipo HTML do componente Spinner.....	54
Listagem 16. Protótipo parametrizado do componente Spinner.....	55
Listagem 17. Definição da declaração do componente Spinner.	55
Listagem 18. Métodos de HtmlRenderer usados para a codificação.....	58
Listagem 19. Método HtmlRenderer.writeStyleResource.....	58
Listagem 20. Método HtmlRenderer._getStyleResourcesAlreadyWritten.	59
Listagem 21. Constantes chave para os recursos.....	59
Listagem 22. Método de codificação do componente Spinner.....	61
Listagem 23. Script de validação da entrada de dados no cliente.	62
Listagem 24. Demais métodos para a codificação do Spinner.....	63
Listagem 25. Método HtmlSpinnerRenderer.decode.....	65
Listagem 26. Classe UISpinner.....	65
Listagem 27. Declarações no arquivo faces-config.xml para o exemplo do Spinner.	67

Listagem 28. Arquivo TLD do Spinner.....	68
Listagem 29. Classe manipuladora da tag do componente Spinner.	70
Listagem 30. Protótipo parametrizado do componente LabelDetailsAjax.	79
Listagem 31. Definição da declaração do componente LabelDetailsAjax.	79
Listagem 32. Métodos para a codificação do componente LabelDetailsAjax.....	82
Listagem 33. Classe UILabelDetailsAjax.....	82
Listagem 34. Declarações no arquivo faces-config.xml para o componente LabelDetailsAjax.....	83
Listagem 35. Arquivo TLD do componente LabelDetailsAjax.	84
Listagem 36. Classe LabelDetailsAjaxTag.....	85
Listagem 37. Ajustes na codificação do componente LabelDetailsAjax.	88
Listagem 38. Construtor e <i>getters</i> de outputTextLinked.....	89
Listagem 39. Método outputTextLinked.init.	90
Listagem 40. Uso do objeto HttpRequest pelo outputTextLinked.	90

LISTA DE FIGURAS

Figura 1. Modelo clássico da Web.	13
Figura 2. Modelo assíncrono com AJAX.....	14
Figura 3. Comparação do modelo tradicional com o modelo AJAX.	17
Figura 4. Sessão do usuário mantida no servidor.....	19
Figura 5. Nova abordagem da sessão do usuário usando AJAX.	20
Figura 6. Gráfico comparativo de aproveitamento da banda de rede.....	21
Figura 7. Comunicação com frames ocultos.....	23
Figura 8. Uso do objeto XMLHttpRequest.....	24
Figura 9. Uso de AJAX nas empresas de software.	30
Figura 10. Procura por profissionais em AJAX.	31
Figura 11. Facade do gerenciador de eventos.....	35
Figura 12. Arquitetura de uma aplicação JEE complexa com JSF e vários RenderKits... 41	
Figura 13. Camadas JSF.....	42
Figura 14. Interação dos serviços JSF com a aplicação Web.	42
Figura 15. Arquitetura MVC nível 2 do JSF.	43
Figura 16. Representação hierárquica da view JSF.....	43
Figura 17. Ciclo de vida JSF.	44
Figura 18. Hierarquia de componentes de interface extensíveis do JSF.....	50
Figura 19. Novo componente sem renderizador específico.....	52
Figura 20. Tela do Protótipo do Componente Spinner.....	55
Figura 21. Diagrama de classes do componente Spinner.....	56
Figura 22. Classes de renderização do componente Spinner.....	57
Figura 23. Decodificação na fase Apply Request Values.	64
Figura 24. Primeira estratégia JSF AJAX.....	72
Figura 25. Segunda estratégia JSF AJAX.....	74
Figura 26. Terceira estratégia JSF AJAX.	75
Figura 27. Protótipo do componente LabelDetailsAjax.....	79

Figura 28. Diagrama de classes do componente LabelDetailsAjax.	80
Figura 29. Diagrama de sequência do comportamento AJAX.	86
Figura 30. Visualização do estado inicial do componente.....	86
Figura 31. Visualização do componente aguardando dados, após clique do usuário..	87
Figura 32. Visualização do componente após receber os dados e atualizar a view.	87

2.2.1	Princípios de estabilidade e utilidade	17
2.2.2	Características gerais	18
2.3	TÉCNICAS DE COMUNICAÇÃO.....	22
2.3.1	Formas órfãs (Dead-end forms)	22
2.3.2	XMLHttpRequest	23
2.3.3	Vantagens e Desvantagens das técnicas.....	29
2.4	VANTAGENS E DESVANTAGENS NO USO DO AJAX.....	30
2.4.1	Vantagens em usar Ajax	30
2.4.2	Desvantagens da adoção de Ajax.....	31
2.5	Padrão W3C DOM Level 3 Load and Save	38
2.6	Framework JAVASCRIPT AJAX	33
2.7	criando um novo objeto ajax	33
2.8	Tratamento de eventos	34
2.9	Encapsulando o objeto "XMLHttpRequest"	36
2.10	iniciando a aplicação ajax	39
2.11	Fundamentos do JST	40
2.12	DEFINIÇÃO	40
2.13	CICLO DE VIDA.....	44
2.13.1	Função render view	45
2.13.2	Função apply request values.....	45
2.13.3	Função process validation	46
2.13.4	Função update model values.....	47
2.13.5	Função invoke application	47
2.13.6	Função render response	48

SUMÁRIO

1	INTRODUÇÃO	13
2	FUNDAMENTOS DE AJAX	16
2.1	DEFINIÇÃO	16
2.2	CARACTERÍSTICAS E PRINCÍPIOS DE UMA APLICAÇÃO COM AJAX	17
2.2.1	Princípios de usabilidade e utilidade	17
2.2.2	Características gerais	18
2.3	TÉCNICAS DE COMUNICAÇÃO	22
2.3.1	Frames ocultos (hidden frames)	22
2.3.2	XMLHttpRequest	23
2.3.3	Vantagens e desvantagens das técnicas	29
2.4	VANTAGENS E DESVANTAGENS NO USO DO AJAX	30
2.4.1	Vantagens em usar AJAX	30
2.4.2	Desvantagens da adoção de AJAX	31
2.5	PADRÃO W3C DOM LEVEL 3 LOAD AND SAVE	32
3	FRAMEWORK JAVASCRIPT AJAX	33
3.1	CRIANDO UM NOVO OBJETO AJAX	33
3.2	TRATAMENTO DE EVENTOS	34
3.3	ENCAPSULANDO O OBJETO "XMLHTTPREQUEST"	36
3.4	INICIANDO A APLICAÇÃO AJAX	39
4	FUNDAMENTOS DE JSF	40
4.1	DEFINIÇÃO	40
4.2	CICLO DE VIDA	44
4.2.1	Fase restore view	45
4.2.2	Fase apply request values	45
4.2.3	Fase process validations	46
4.2.4	Fase update model values	47
4.2.5	Fase invoke applications	47
4.2.6	Fase render response	48

5 CRIAÇÃO DE COMPONENTES JSF.....	49
5.1 ESTRATÉGIAS PARA CRIAÇÃO DE COMPONENTES JSF.....	50
5.1.1 Sem Renderizador Específico.....	51
5.1.2 Com Renderizador Específico.....	52
5.2 CRIAÇÃO DO COMPONENTE USANDO BOAS PRÁTICAS.....	53
5.2.1 Criando o protótipo da interface.....	54
5.2.2 Design das classes necessárias.....	55
5.2.3 Criando o renderizador específico para o cliente Web.....	57
5.2.3.1 A classe <i>HtmlRenderer</i>	57
5.2.3.2 A classe <i>HtmlSpinnerRenderer</i>	59
5.2.3.2.1 Manipulação dos valores dos atributos.....	60
5.2.3.2.2 Identificação do componente.....	60
5.2.3.2.3 Escrita da saída para o cliente.....	60
5.2.3.2.4 Decodificando no postback.....	63
5.2.4 Estendendo o componente de interface JSF.....	65
5.2.5 Registrando o componente e renderizador.....	66
5.2.6 Criando o arquivo TLD.....	67
5.2.7 Criando a classe para manipular a tag JSP.....	69
6 JSF + AJAX.....	72
6.1 ESTRATÉGIAS PARA INSERIR AJAX NO JSF.....	72
6.1.1 Primeira estratégia: usando um <i>Phase Listener</i>	72
6.1.2 Segunda estratégia: usando um <i>Servlet</i> dentro do contexto JSF.....	73
6.1.3 Terceira estratégia: usando um <i>Servlet</i> fora do ciclo JSF.....	75
6.1.4 Comparação entre as estratégias.....	77
6.2 DEFINIÇÃO DO COMPONENTE JSF.....	78
6.2.1 Criando o protótipo da interface.....	78
6.2.2 Design das classes necessárias.....	79
6.2.3 Criando o renderizador específico para o cliente Web.....	80
6.2.4 Estendendo o componente de interface JSF.....	82

6.2.5	Registrando o componente e renderizador.....	82
6.2.6	Criando o arquivo TLD	83
6.2.7	Criando a classe para manipular a tag JSP	84
6.3	IMPLEMENTANDO NO COMPONENTE JSF O FRAMEWORK AJAX JAVASCRIPT	85
6.3.1	A estratégia usada.....	87
6.3.2	Os ajustes no renderizador	88
6.3.3	O código JavaScript.....	89
7	CONCLUSÃO.....	91
7.1	PRIMEIRA ESTRATÉGIA: CODIFICAR AJAX DIRETAMENTE	91
7.1.1	Vantagens	91
7.1.2	Desvantagens.....	91
7.2	SEGUNDA ESTRATÉGIA: USAR BIBLIOTECA QUE ENCAPSULE O AJAX	92
7.2.1	Vantagens	92
7.2.2	Desvantagens.....	92
7.3	TERCEIRA ESTRATÉGIA: USAR UM FRAMEWORK CLIENTE-SERVIDOR QUE ENCAPSULA A BIBLIOTECA JAVASCRIPT COM AJAX	93
7.3.1	Vantagens	93
7.3.2	Desvantagens.....	93
7.4	TRABALHOS FUTUROS.....	93
	REFERÊNCIAS	95

1 INTRODUÇÃO

Tradicionalmente, a porção multimídia da internet, a World Wide Web, ou apenas, Web, foi originalmente projetada para navegação por documentos HTML¹. Usando um modelo de comunicação síncrono, a Web era limitada às ações: clicar, aguardar e recarregar. Portanto, o conteúdo, ou página, é transferido do servidor para o cliente (*browser*) a cada evento, envio de dados ou navegação, como ilustra a Figura 1.

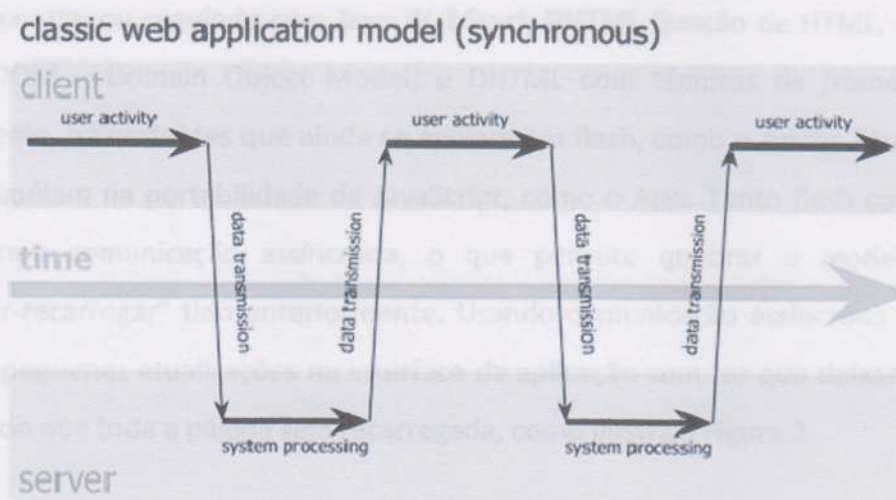


Figura 1. Modelo clássico da Web.
Fonte: Garret (2005).

Com o advento da “Web 2.0”, termo lançado pela O’Reilly Media em 2004, a corrida agora é para transformar o que antes era puramente estático, em verdadeiras aplicações similares às aquelas executadas em *desktops*. Desta forma, a Web agora é vista como uma plataforma e agrega novas características como inteligência coletiva, filtros de SPAM colaborativos, Blogs, Mashups – uso de mais de uma fonte para criar outros serviços - Housing Maps (Google Maps + Craigslist), enfim, aplicações que provêem maior experiência com o usuário. Nascem novos serviços como aplicações que realizam armazenamento e processamento no servidor, como o Writely, Flickr e Gmail. Nestas novas aplicações não há lançamento de versões, pois o aprimoramento é contínuo, não há necessidade de portar para várias plataformas e ainda podem contar com APIs *on line*, como o Google Maps API, Amazon Web Services, eBay API etc.

¹ HTML (Hiper-Text Markup Language) é uma linguagem de especificação de *hyperlinks* e layout de documentos.

Outros serviços bem mais dinâmicos e interativos como o Wikipedia, MySpace, YouTube, del.icio.us e tantos outros, mostram novas oportunidades.

Nesta nova visão, é essencial que as aplicações Web respondam de forma intuitiva e rápida para que tudo aconteça naturalmente para o usuário. O termo “aplicações ricas para internet”, ou “rich internet applications” (RIA), é designado para essa nova família de aplicações. No entanto, a idéia de aplicações RIA não é tão nova, pelo contrário, nasceu basicamente em 1995 com Java Applets, depois com Macromedia Flash e continuou seguindo com Java WebStart, DHTML (junção de HTML, JavaScript, CSS e DOM – Domain Object Model) e DHTML com técnicas de *frames* ocultos. Atualmente, há vertentes que ainda se apóiam em flash, como o Adobe Flex, e outros que se apóiam na portabilidade de JavaScript, como o Ajax. Tanto flash como o Ajax incorporam comunicação assíncrona, o que permite quebrar o modelo “clique-aguardar-recarregar” tido anteriormente. Usando comunicação assíncrona é possível realizar pequenas atualizações na interface da aplicação sem ter que deixar o usuário esperando que toda a página seja recarregada, como ilustra a Figura 2.

Ajax web application model (asynchronous)

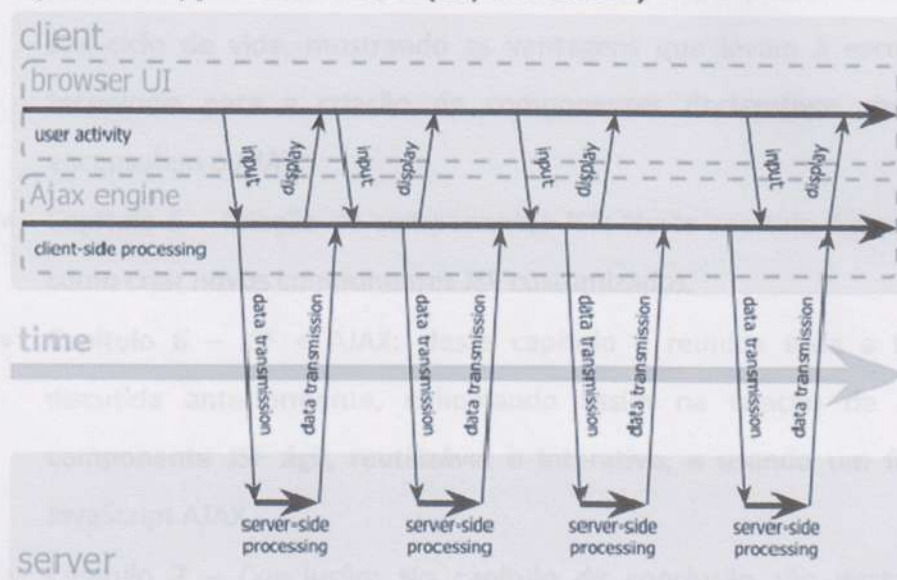


Figura 2. Modelo assíncrono com AJAX.
Fonte: Garret (2005).

Moore, Budd e Benson (2007), enumeram as seguintes características para aplicações RIA: plasticidade, quebra de paradigmas, invisibilidade, quebra de padrões proprietários, criação e manipulação de conteúdo compartilhado, sempre podem ser

atualizadas, independentes de sistemas operacionais, centradas no browser, centradas na rede. As aplicações RIA trazem ao desenvolvedor, ao arquiteto e ao usuário uma nova forma de pensar, transformando aplicações em serviços, centradas no usuário, inerentemente colaborativas, com desenvolvimento mais simples e mais ágil, estão ligadas às mudanças, e mais sedutoras.

Neste trabalho será demonstrado como pode ser encapsulado o AJAX em componentes reutilizáveis, ágeis e interativos para construir aplicações RIA produtivas. Para alcançar este objetivo, o trabalho foi organizado como segue:

- Capítulo 2 – Fundamentos de AJAX: O objetivo deste capítulo é definir o AJAX e suas características para mostrar as vantagens que levam à escolha desta tecnologia.
- Capítulo 3 – Framework JavaScript AJAX: Neste capítulo são destacadas as características importantes para um framework JavaScript AJAX, associando padrões de projetos e boas práticas para a criação de componentes.
- Capítulo 4 – Fundamentos de JSF: O objetivo deste capítulo é definir JSF e o seu ciclo de vida, mostrando as vantagens que levam à escolha desta tecnologia para a criação de componentes declarativos simples que encapsulem o AJAX.
- Capítulo 5 – Criação de componentes JSF: Neste capítulo é demonstrado como criar novos componentes JSF customizados.
- Capítulo 6 – JSF + AJAX: Neste capítulo é reunida toda a tecnologia discutida anteriormente, culminando assim na criação de um novo componente JSF ágil, reutilizável e interativo, e usando um framework JavaScript AJAX.
- Capítulo 7 – Conclusão: No capítulo de conclusão são destacados os aspectos positivos e negativos do AJAX em cada nível de abstração discutido, partindo da codificação direta do AJAX até o uso de componentes JSF de alto nível, nos quais é abstraído todo o mecanismo necessário ao funcionamento do AJAX em uma aplicação RIA.

2 FUNDAMENTOS DE AJAX

Este capítulo aborda o AJAX como a peça fundamental para a criação de componentes a serem usados na implementação de uma aplicação RIA. Após a exploração da definição e das características desta técnica, serão apresentados os princípios para a criação de um framework AJAX em JavaScript usando as boas práticas e padrões já bem estabelecidos atualmente. O intuito é dar o suporte à criação de componentes de mais alto nível que poderão encapsular este framework, abstraindo para o desenvolvedor da aplicação toda a problemática envolvida na implementação de uma aplicação RIA com AJAX.

2.1 DEFINIÇÃO

O termo AJAX foi primariamente definido por Jesse James Garret, em fevereiro de 2005, no seu artigo "Ajax: A New Approach to Web Applications". Ajax é um acrônimo para "Asynchronous Javascript and XML", ou seja, JavaScript assíncrono e XML. Não se trata de uma nova tecnologia, apenas da união de várias tecnologias para criar um padrão de desenvolvimento de aplicações web ricas:

- CSS e HTML/XHTML para os padrões de apresentação;
- Document Object Model para a interação e o dinamismo da visualização;
- XML e XSTL para o intercâmbio e manipulação de dados;
- JavaScript para ligar todas as partes.

A forma como Ajax transforma o modelo tradicional de aplicações web é centrada em um objeto ou uma função JavaScript, denominada por Garret (2005) como "Ajax engine". O "Ajax engine" provê uma camada intermediária entre o browser e o servidor, de forma que pode iniciar e processar requisições. A idéia é que a comunicação seja dada através do envio e recebimento de pequenas quantidades de informações sempre que forem solicitados novos dados pelo usuário. A Figura 3 ilustra como funciona o "Ajax engine" comparando com o modelo clássico.

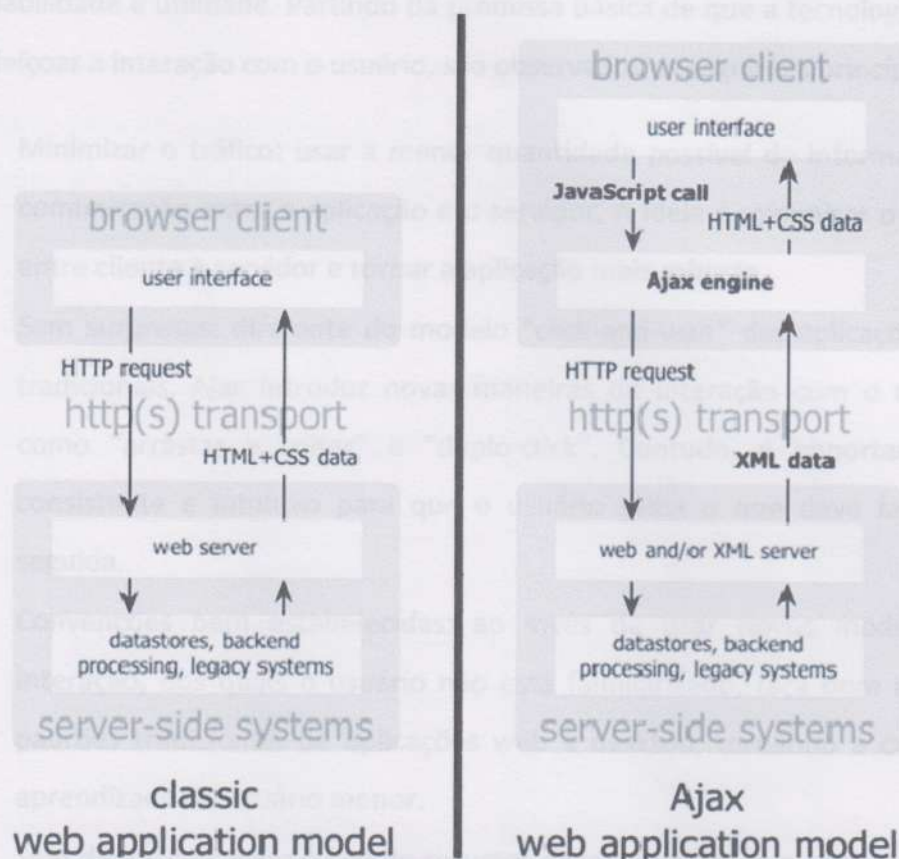


Figura 3. Comparação do modelo tradicional com o modelo AJAX.

Fonte: Garret (2005)

As solicitações de recursos ao servidor são agendadas e executadas pelo "Ajax engine", que realizará as requisições de forma assíncrona, ou seja, a execução do código não esperará pelo retorno, continuando a sua execução normalmente. Qualquer formato de dados pode ser retornado pelo servidor, desde que o "Ajax engine" consiga entendê-lo e interpretá-lo, usualmente XML ou texto simples.

O "Ajax engine" recebe a resposta do servidor, analisa a informação e realiza as alterações na interface do usuário. Como essa comunicação se dá através de pequenas quantidades de informações, a atualização da interface é bem mais rápida.

2.2 CARACTERÍSTICAS E PRINCÍPIOS DE UMA APLICAÇÃO COM AJAX

2.2.1 Princípios de usabilidade e utilidade

De acordo com o expert em desenvolvimento e usabilidade de software, Michael Mahemoff apud Zakas, McPeak e Fawcett (2006), alguns princípios devem ser levados em consideração para a construção de aplicações Ajax bem implementadas no que

tange a usabilidade e utilidade. Partindo da premissa básica de que a tecnologia serve para aperfeiçoar a interação com o usuário, são observados os seguintes princípios:

- Minimizar o tráfico: usar a menor quantidade possível de informação na comunicação entre a aplicação e o servidor. A idéia é minimizar o tráfego entre cliente e servidor e tornar a aplicação mais robusta.
- Sem surpresas: diferente do modelo "click-and-wait" das aplicações web tradicionais, Ajax introduz novas maneiras de interação com o usuário, como "arrastar e soltar" e "duplo-click". Contudo, é importante ser consistente e intuitivo para que o usuário saiba o que deve fazer em seguida.
- Convenções bem estabelecidas: ao invés de usar novos modelos de interação, dos quais o usuário não está familiarizado, faça bom uso dos padrões tradicionais de aplicações web e desktop, tornando a curva de aprendizado do usuário menor.
- Sem distrações: evite o uso de recursos desnecessários que podem distrair o usuário, como elementos piscando ou animações demasiadas, tirando seu foco da operação que ele deseja completar.
- Acessibilidade: os usuários não devem ser forçados a usar algum browser em especial, ou depender de algum outro componente de software para executar sua aplicação Ajax.
- Evitar downloads de páginas inteiras: toda a comunicação com o servidor depois do carregamento da página inicial deve ser controlada pelo "Ajax engine". Portanto, carregar pequenas quantidades de dados em uma parte e páginas inteiras em outra está fora de cogitação.
- Usuário em primeiro lugar: os casos de uso em aplicações Ajax devem ser simples de serem operados, evitando a preocupação com efeitos e propagandas e tendo o usuário em mente acima de tudo.

2.2.2 Características gerais

Explorando princípios de aplicações Ajax quanto às suas características gerais, Crane, Pascarello e James (2006), apontam quatro definições:

O browser executa uma aplicação, não apenas exibe conteúdo: no modelo tradicional de aplicações web, o browser não tem influência sobre o fluxo ou lógica de negócio, ele apenas exibe conteúdo, tal qual um terminal “burro”. Toda a informação das transações de negócio é mantida no servidor nas “sessões de usuário”. A Figura 4 ilustra com mais detalhes esta visão.

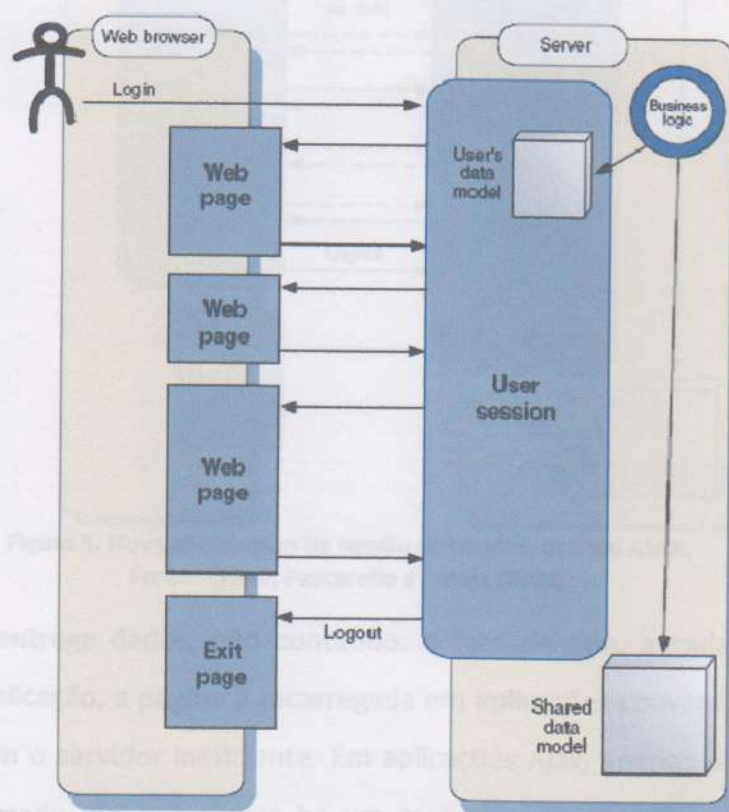


Figura 4. Sessão do usuário mantida no servidor.

Fonte: Crane, Pascarello e James (2006)

Aplicações Ajax transportam parte da lógica de negócio para o browser. Ao invés de apenas conteúdo estático e sem semântica alguma, o servidor envia um documento complexo permitindo ao usuário interagir com ele e alterar o seu comportamento e visualização. Este documento complexo, normalmente contendo bastante código JavaScript, é capaz de decidir quando é capaz de resolver por si só a solicitação do usuário e quando deverá passar uma requisição para o servidor web, ou até mesmo fazer uma combinação destas ações. Desta forma, pode-se até mesmo armazenar estados da aplicação no browser ao invés de tudo estar na sessão do usuário no servidor web. Na Figura 5 é possível perceber como a abordagem anterior fica modificada.

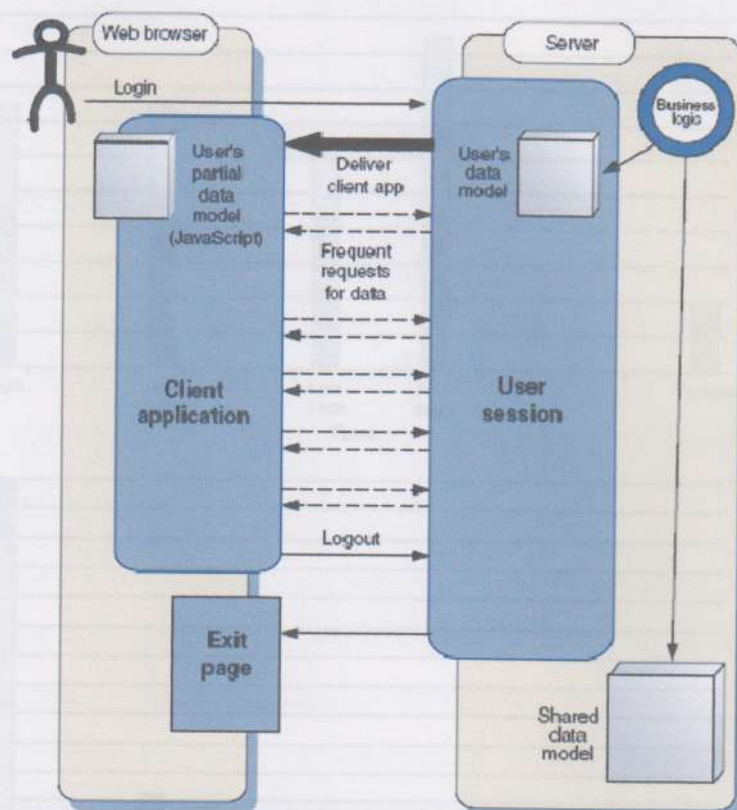


Figura 5. Nova abordagem da sessão do usuário usando AJAX.

Fonte: Crane, Pascarello e James (2006)

O servidor entrega dados, não conteúdo: o fato de que, a cada interação do usuário com a aplicação, a página é recarregada em aplicações convencionais, torna a comunicação com o servidor ineficiente. Em aplicações Ajax, apenas o carregamento inicial é mais pesado, daí por diante há um melhor aproveitamento da largura de banda da rede, pois a comunicação é realizada com pequenas porções de informações de cada vez.

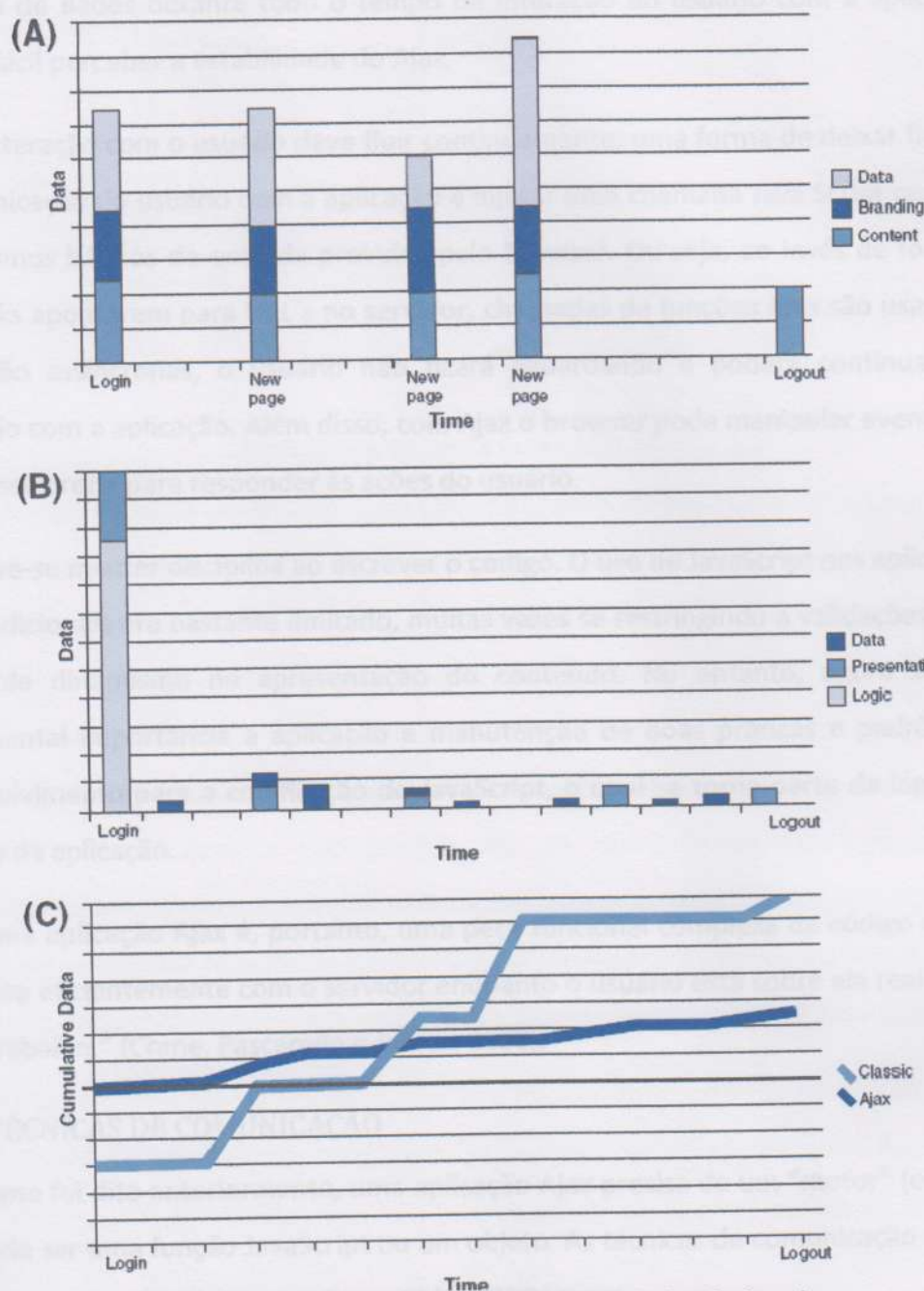


Figura 6. Gráfico comparativo de aproveitamento da banda de rede.
Fonte: Crane, Pascarello e James (2006)

Nos gráficos ilustrados na Figura 6, percebe-se a diferença entre aplicações clássicas, onde grandes porções de dados, conteúdo e marcadores são transmitidas a cada interação (A) ; e aplicações Ajax, onde inicialmente é transmitida uma grande quantidade de informação de lógica de negócio e parte da apresentação, e, em seguida, apenas pequenas porções de informações referentes a dados e apresentação são necessárias a cada interação com o usuário (B). E, no último gráfico, compara-se o

acúmulo de dados durante todo o tempo de interação do usuário com a aplicação, onde é fácil perceber a estabilidade do Ajax.

A interação com o usuário deve fluir continuamente: uma forma de deixar fluente a comunicação do usuário com a aplicação é injetar uma chamada Java Script nos dois mecanismos básicos de entrada providos pelo browser. Ou seja, ao invés de forms e hiperlinks apontarem para URL's no servidor, chamadas de funções Ajax são usadas e, como são assíncronas, o usuário não ficará aguardando e poderá continuar sua interação com a aplicação. Além disso, com Ajax o browser pode manipular eventos de forma assíncrona para responder às ações do usuário.

Deve-se manter disciplina ao escrever o código. O uso de JavaScript nas aplicações web tradicionais era bastante limitado, muitas vezes se restringindo a validações e um pouco de dinamismo na apresentação do conteúdo. No entanto, agora são de fundamental importância a aplicação e manutenção de boas práticas e padrões de desenvolvimento para a codificação do JavaScript, o qual se torna parte da lógica de negócio da aplicação.

“Uma aplicação Ajax é, portanto, uma peça funcional complexa de código que se comunica eficientemente com o servidor enquanto o usuário está sobre ela realizando o seu trabalho.” (Crane, Pascarello e James, 2006).

2.3 TÉCNICAS DE COMUNICAÇÃO

Como foi dito anteriormente, uma aplicação Ajax precisa de um “motor” (engine), que pode ser uma função JavaScript ou um objeto. As técnicas de comunicação usadas no “motor” Ajax são: frames ocultos e XMLHttpRequest.

2.3.1 Frames ocultos (hidden frames)

Esta técnica consiste, basicamente, em criar uma página HTML com um *frameset* onde um dos frames será ocultado, atribuindo a propriedade “width” ou “height” para 0 pixels, para servir, exclusivamente, na comunicação entre o browser e o servidor.

Na Figura 7 é ilustrado o padrão de 4 passos para comunicação com hidden frames:

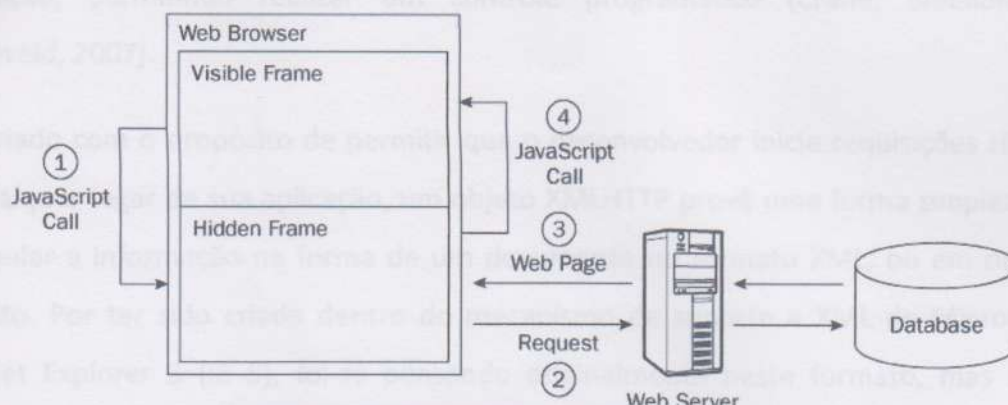


Figura 7. Comunicação com frames ocultos.

Fonte: Zakas, Mcpeak e Fawcett (2006)

No frame visível, o usuário interage com a página e executa uma ação que requer novas informações do servidor, disparando uma chamada JavaScript para o frame invisível (1). Esta chamada pode ser tão simples quanto o envio de dados em requisição do tipo GET, bem como um envio complicado de dados de um formulário com uma requisição do tipo POST, de qualquer forma, uma requisição será enviada ao servidor (2). Por seu turno, o servidor retorna uma página Web inteira, pois estamos trabalhando com frames e não poderia ser de outra forma (3). Esta página, necessariamente, contém os dados solicitados assim como um código JavaScript, usualmente associado ao evento "onload", para transferir os dados ao frame visível (4). Com os dados requisitados em seu poder, o frame visível pode então manipulá-los.

Uma especialização do hidden frame é o *iframe*, originado no HTML 4.0 e tem como vantagem poder ser colocado em qualquer página, não somente em *framesets*. Apesar de seguir o padrão dos mesmos passos acima, os *iframes* permitem a adição incremental de funcionalidades, uma vez que podem ser criados até mesmo dinamicamente em tempo de execução. Com o detalhe de que apenas o browser Internet Explorer reconhece automaticamente os frames adicionados, para os demais browsers é necessário colocar um tempo de espera para que eles reconheçam o novo frame. Podem ser usados também em requisições do tipo GET e POST.

2.3.2 XMLHttpRequest

Introduzido pelo Internet Explorer 5.0, dentro da biblioteca ActiveX chamada MSXML, o objeto XMLHttpRequest fornece uma granularidade melhor de controle sobre a

requisição, permitindo realizar um controle programático (Crane, Bibeault e Sonneveld, 2007).

Criado com o propósito de permitir que o desenvolvedor inicie requisições HTTP de qualquer lugar de sua aplicação, um objeto XMLHttpRequest provê uma forma simples de manipular a informação na forma de um documento no formato XML, ou em outro formato. Por ter sido criado dentro do mecanismo de suporte a XML do Microsoft Internet Explorer 5 (IE 5), foi-se pensando originalmente neste formato, mas não limitado a ele.

Desta forma, finalmente, podemos ter aplicações web complexas com clientes tratando dados e não apenas conteúdo. E, por ser um ActiveX, também pode ser usado em aplicações desktops, porém se popularizou na Web, o que impulsionou os outros browsers a também suportar este recurso. Iniciando pelo Mozilla Firefox, seguido do Safari (versão 1.2) e do Opera (versão 7.6), atualmente todos os browsers possuem uma implementação do objeto XMLHttpRequest.

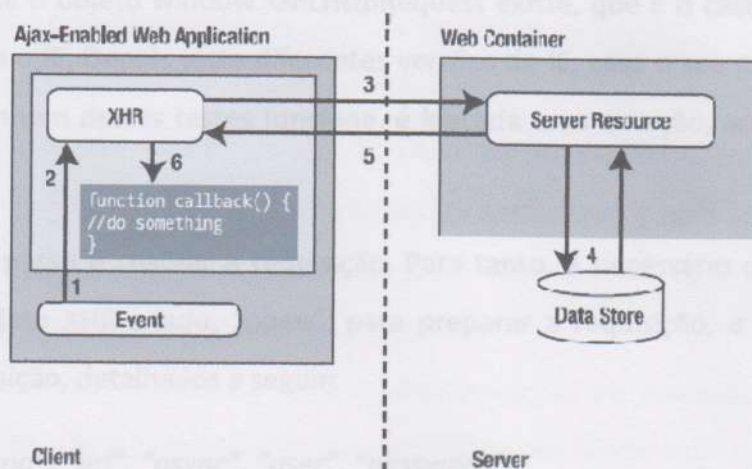


Figura 8. Uso do objeto XMLHttpRequest.

Fonte: Asleson e Schutta (2006b)

A primeira etapa para a utilização do objeto XMLHttpRequest é a sua criação, como mostrado no segundo passo da Figura 8. Devido às várias diferenças nas versões da Microsoft e na implementação dos demais browsers, é necessário um pouco de cuidado na sua criação para manter a compatibilidade com todos os browsers. De acordo com Gross (2006, p.27), uma boa prática é usar o padrão de projeto *factory method* (SHALLOWAY, TROTT, 2000, p. 285) para instanciação de um tipo, como demonstrado na Listagem 1.

```
function FactoryXMLHttpRequest() {  
    if (window.XMLHttpRequest) {  
        return new XMLHttpRequest();  
    } else {  
        if (window.ActiveXObject) {  
            var msxmls = new Array(  
                "Msxml2.XMLHTTP.5.0",  
                "Msxml2.XMLHTTP.4.0",  
                "Msxml2.XMLHTTP.3.0",  
                "Msxml2.XMLHTTP",  
                "Microsoft.XMLHTTP");  
            for (var i = 0; i < msxmls.length; i++) {  
                try {  
                    return new ActiveXObject(msxmls[i]);  
                }  
                catch (e) {  
                }  
            }  
        }  
        throw new Error("Could not instantiate XMLHttpRequest");  
    }  
}
```

Listagem 1. Criação do objeto XMLHttpRequest com factory method.

Neste caso, o padrão foi implementado fazendo uso de um método apenas, `FactoryXMLHttpRequest`, o qual retorna uma instância de um objeto `XMLHttpRequest`, denotado daqui para frente apenas por XHR. A função testa em primeiro lugar se o objeto `window.XMLHttpRequest` existe, que é o caso de todos os browsers exceto o IE. Depois testa diferentes versões do IE, caso o seu objeto `ActiveX` existir. Caso nenhum destes testes funcione, é lançada uma exceção, ao invés de um valor "null".

O segundo passo é realizar a requisição. Para tanto, é necessário o uso de dois métodos do objeto XHR criado, "open", para preparar a requisição, e "send", para disparar a requisição, detalhados a seguir:

`open("method", "url", "async", "user", "password")`

Parâmetros:

- "method": Requerido. Indica qual o método usado na requisição. GET e POST são suportados por todos os browsers;
- "url": Requerido. Indica a url requisitada;
- "async": Não requerido. Indica o se a requisição será assíncrona ou não, o padrão é "true", ou seja, assíncrona;

- “user” e “password”: Não requeridos. Usuário e respectiva senha para acesso à url requisitada.

send(“content”)

Parâmetros:

- “content”: Requerido. Define o conteúdo da mensagem HTTP. Serve para conter os dados que serão postados em requisição POST, ou mesmo um arquivo para upload. Deve ser usado “null” caso não haja dados a serem enviados.

Caso seja usado o método assíncrono, é necessário informar a propriedade *onreadystatechange*, apontando para uma função, chamada também de *callback function* que irá ser acionada assincronamente de acordo com a mudança no *status* da requisição. A função de *callback* deve ler a propriedade *readystate* para decidir o que fazer. Esta propriedade tem os seguintes valores possíveis:

- 0: não inicializado - O objeto foi criado, mas o método *open* não foi chamado ainda;
- 1: carregando - O método *open* foi chamado, mas não o método *send*;
- 2: carregado - Indica que a conexão foi estabelecida com sucesso com o servidor. Os cabeçalhos de resposta do servidor e o *status code*, código HTTP de retorno, já podem ser acessados;
- 3: interativo - Indica que o conteúdo começou a ser recebido, e pode ser acessado pela propriedade *responseText*, mas ainda está incompleto. É disparado a cada 4kb recebidos ao começar a receber os dados.
- 4: completo - O conteúdo foi totalmente recebido - a requisição foi finalizada. Não indica, entretanto, que o arquivo foi encontrado, ou que não houve erros no processamento do arquivo. Indica apenas que o processo de conectar ao servidor, fazer uma requisição HTTP simples e receber a resposta foi concluído, mas o arquivo pode não ter sido encontrado ou ter acontecido algum erro no processamento.

De acordo com Zakas, Mcpeak e Fawcett (2006), é uma boa prática em aplicações Ajax reservar o uso do método síncrono para mensagens mais curtas, e usar o método assíncrono para a maior parte da aplicação. Desta forma, o usuário interage com a aplicação ao invés de esperar até a requisição estar completa.

Na Listagem 2 segue a continuação do exemplo anterior, usando `FactoryXMLHttpRequest` para uma requisição assíncrona:

```
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <script language="JavaScript" src="/lib/factory.js"></script>
  <script language="JavaScript" type="text/javascript">
var xmlhttp = FactoryXMLHttpRequest();
function AsyncUpdateEvent() {
  switch (xmlhttp.readyState) {
    case 0:
      document.getElementById("status").innerHTML = "não
inicializada";
      break;
    case 1:
      document.getElementById("status").innerHTML = "carregando";
      break;
    case 2:
      document.getElementById("status").innerHTML = "carregada";
      break;
    case 3:
      document.getElementById("status").innerHTML = "interativa";
      break;
    case 4:
      document.getElementById("status").innerHTML = "completa";
      document.getElementById("result").innerHTML =
xmlhttp.responseText;
      break;
  }
}
function GetIt(url) {
  if (xmlhttp) {
    xmlhttp.open("GET", url, true);
    xmlhttp.onreadystatechange = AsyncUpdateEvent;
    xmlhttp.send(null);
  }
}
  </script>
</head><body>
  <button onclick="GetIt('/serverhang.aspx')">
    Get a document
  </button>
  <p>
    <table border="1">
      <tr>
        <td>
          Document
        </td>
        <td>

```

```

        <span id="status">No Result</span>
      </td>
      <td>
        <span id="result">No Result</span>
      </td>
    </tr>
  </table>
</p>
</body>
</html>

```

Listagem 2. Página usando FactoryXMLHttpRequest.

O terceiro e último passo é acessar e formatar os dados retornados. No exemplo da Listagem 2, é mostrado o uso da propriedade *responseText*, a qual armazena a resposta da requisição em uma *string* comum. Para recuperar os dados requisitados e tratar seu conteúdo pode ser lida também a propriedade *responseXML*, de forma compatível com o DOM e manipulável tal qual um documento XML.

Outros métodos do objeto XHR:

- **abort:** aborta a requisição corrente, se possível. Seu impacto se dá apenas do lado cliente, o servidor continuará a processar a requisição, no entanto, o browser irá descartá-la;
- **setRequestHeader(header, value):** atribui um valor a um cabeçalho da requisição http. Usada normalmente para atribuir o tipo do conteúdo do corpo da requisição. Qualquer cabeçalho HTTP válido pode ser atribuído. Pode ser usada também para definir o MIME da requisição, atribuindo *x-www-form-urlencoded* para emular o envio de um formulário HTML, ou *'application/xml; charset=UTF-8'* para envio de um arquivo XML a ser tratado no servidor;
- **getAllResponseHeaders():** lista todos os cabeçalhos HTTP da resposta da requisição;
- **getResponseHeader(header):** recupera o valor de um cabeçalho HTTP específico da resposta da requisição.

Outras propriedades do objeto XHR:

- **status:** recupera o código do status HTTP retornado pelo servidor (ex.: 200 = OK);
- **statusText:** recupera o texto do status HTTP retornado pelo servidor.

2.3.3 Vantagens e desvantagens das técnicas AJAX

A vantagem interessante em usar *hidden frames* e *iframes* é a preservação do histórico de navegação. Ou seja, permite ao usuário usar os botões *back* e *forward* do *browser*. Contudo, para *iframes*, somente o IE continuamente guarda o histórico, no FIREFOX somente se criado por HTML e no Safari o histórico não é armazenado de forma nenhuma.

A grande desvantagem é o cliente, ou seja, o *browser*, continuar sem saber o que está acontecendo, que tipo de informação está sendo transitada e qual é o negócio da aplicação. Além disso, é necessário usar um limite de tempo, pois, do contrário, se houver algum erro na requisição o usuário, vai esperar um longo tempo sem saber o que está acontecendo. Desta forma, não há informação sobre a requisição HTTP que está ocorrendo nos bastidores da aplicação.

Notoriamente, o uso do objeto XHR traz grandes vantagens para a comunicação cliente-servidor sobre *hidden frames*. A codificação é mais clara e mais fácil de entender o seu propósito usando XHR do que várias funções de *callback* de uma aplicação complexa com *hidden frames*. O acesso aos cabeçalhos HTTP de requisição e resposta no XHR permite melhor interatividade com o usuário, permitindo determinar rapidamente e com facilidade se a requisição teve sucesso.

Uma desvantagem no uso do XHR é a perda do histórico de navegação. Os botões do *browser* "voltar" e "avançar" não funcionam em requisições XHR. Outra desvantagem, exclusiva para o caso do uso do IE, é a dependência da habilitação de controles ActiveX na sua configuração de segurança.

Apesar destas questões, o uso do XHR tem se sobressaído. A opção por chamadas assíncronas tem tomado conta do cenário Ajax e, portanto, tem tido maiores aprofundamentos, definições de padrões e técnicas mais apuradas e está sendo bem mais aplicado na maioria das aplicações.

2.4 VANTAGENS E DESVANTAGENS NO USO DO AJAX

2.4.1 Vantagens em usar AJAX

São citados em Johnson, White e Charland (2007) como vantagens do uso de AJAX em aplicações Web corporativas pelo menos três pontos essenciais:

- AJAX pode melhorar e maximizar a experiência do usuário final, fazendo-os mais produtivos e satisfeitos;
- AJAX pode reduzir a demanda da rede e da infra-estrutura do servidor, reduzindo assim custos com manutenção e banda, e ainda assim melhorar a qualidade do serviço para os usuários finais;
- AJAX pode criar a possibilidade de novos tipos de funcionalidades antes impossíveis ou impraticáveis no modelo tradicional, dando aos usuários novas ferramentas para atingir seus objetivos.

Além disso, a grande e crescente demanda por aplicações RIA têm levado muitas empresas de desenvolvimento de softwares a criarem novas ferramentas, e, na sua grande maioria, elas optam por AJAX. De acordo com uma pesquisa do BZ Research (Figura 9), em setembro de 2006, apenas 9.5% das empresas não tinham planos de usar AJAX.

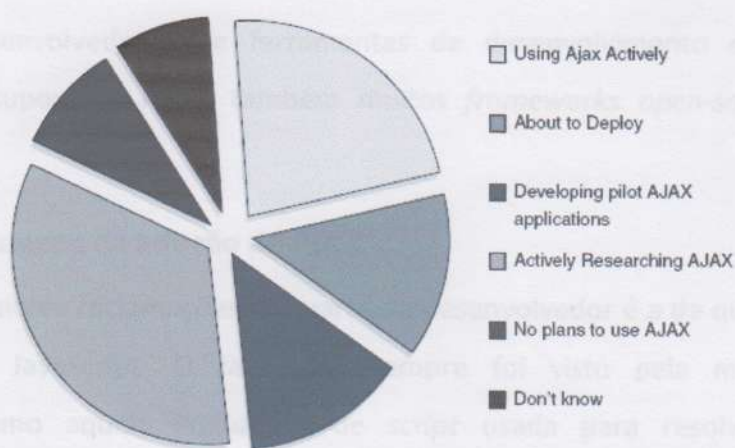


Figura 9. Uso de AJAX nas empresas de software.

Fonte: Johnson, White e Charland (2007)

Esta tendência também se confirma na pesquisa realizada pelo INDEED, onde é verificada a crescente busca por profissionais com habilidades em AJAX, conforme Figura 10:



Figura 10. Procura por profissionais em AJAX.

Fonte: Johnson, White e Charland (2007).

Aplicações Web desenvolvidas em AJAX recentemente têm se popularizado, como é o caso das novas aplicações da Google (Google GMail, Google Suggests, Google Maps). As pessoas hoje estão pensando em aplicações, não apenas em sites. E tudo isso sem criar novos investimentos em estrutura, todo o conhecimento Web já adquirido é aproveitado, tal como JavaScript, CSS, XML, XHTML, DOM etc.

Não é necessário se preocupar com distribuição de atualizações das aplicações e também é possível criar aplicações *Cross-Browsing* em AJAX, ou seja, independente de *browser*.

Muitos desenvolvedores de ferramentas de desenvolvimento estão criando soluções com suporte a AJAX, também muitos *frameworks open-source* estão à disposição.

2.4.2 Desvantagens da adoção de AJAX

Uma das maiores reclamações por parte do desenvolvedor é a de que vai precisar programar em JavaScript. O JavaScript sempre foi visto pela maioria destes profissionais como aquela linguagem de *script* usada para resolver pequenos problemas no *browser*. Também por ser uma linguagem dinâmica, fracamente tipada e sem completo suporte a orientação a objetos, uma vez que ela é apenas baseada e não orientada a objetos, alguns programadores não gostam desta linguagem. Portanto, deve-se recorrer a alguma biblioteca ou *framework* para tornar o desenvolvimento mais fácil e prazeroso. Outro empecilho para os desenvolvedores é ter que lidar com

as diferenças de implementações do DOM nos *browsers*. As diferenças entre *browsers* também implica em dificuldades para testar aplicação, que deve ser retestada a cada lançamento de nova versão de *browsers*.

Outras dificuldades inerentes a adoção do AJAX podem se tornar mais uma complicação que precisa ser tratada, como: os botões de navegação do *browser* não poderem ser usados, não será armazenado no *browser* o histórico da navegação do usuário, recuperação de recursos fora do domínio da aplicação etc.

Apesar das dificuldades e desvantagens, as vantagens em usar AJAX para o desenvolvimento de aplicações RIA ainda são mais relevantes. O que leva à sua adoção ser a mais usual atualmente.

2.5 PADRÃO W3C DOM LEVEL 3 LOAD AND SAVE

Apesar de sua popularidade, é bom lembrar que o AJAX não é um padrão W3C. O padrão W3C que especifica a maneira independente de linguagem ou de *browser* de modificar o conteúdo de um documento DOM a partir de conteúdo XML é o **DOM Level 3 Load and Save**. Apesar de sua versão 1.0 ter sido lançada em abril de 2004, ainda nenhum *browser* a tem implementada. É difícil afirmar hoje se o Leve 3 irá suplantará o AJAX, no entanto já existe alguns esforços no sentido de viabilizá-lo. A implementação do XMLHttpRequest no *browser* Opera já é baseada em um futuro suporte ao DOM Level 3, e a API Java para processamento de arquivos XML (JAXP) já o suporta a partir da versão 1.3. A rapidez com que o W3C lançou esta especificação aponta como são importantes as técnicas AJAX.

A maior vantagem trazida pela especificação DOM Level 3 é justamente a forma *cross-browser* com que permite que seja implementado o código, trazendo uma forma padrão a ser seguida independentemente de sistema operacional e de linguagem. Além disso, será possível salvar a árvore DOM diretamente como um documento XML ou pegar este documento e carregá-lo diretamente no DOM. A especificação também provê suporte ao XML 1.1, XML Schema 1.0 e SOAP 1.2. Uma vez disponível, espera-se que seja bastante usada pelos desenvolvedores.

3 FRAMEWORK JAVASCRIPT AJAX

A construção de uma biblioteca ou um *framework* em JavaScript para o suporte AJAX deve atender a alguns pré-requisitos funcionais:

- Implementação do mecanismo de comunicação
- Compatibilidade com os *browsers* (*Cross-Browsing*)
- Controles visuais e efeitos
- Suporte ao servidor e ligação de dados
- Tratamento de erros

Apesar do JavaScript não dar suporte a *namespaces*² é importante implementar um mecanismo para prevenir que outros *frameworks* ou bibliotecas venham a interferir uns nos outros, por terem nomes iguais sendo usados entre si. Portanto, o *framework* deve ser implementado partindo-se de um objeto vazio e adicionando funcionalidades a ele depois, graças a mutabilidade do JavaScript.

```
if (typeof entAJAX == "undefined") {
    entAJAX = {}; //Singleton
}
```

Listagem 3. Singleton JavaScript.

A Listagem 3 implementa um *singleton* (SHALLOWAY, TROTT, 2000, p. 256) que será responsável por encapsular toda a funcionalidade do *framework*.

3.1 CRIANDO UM NOVO OBJETO AJAX

Apesar de já ter sido visto anteriormente a implementação de um *factory method* (SHALLOWAY, TROTT, 2000, p. 285) para a criação do objeto XHR, algumas modificações serão realizadas para fazer uso das novas propriedades criadas na Listagem 4.

```
var ua = navigator.userAgent;
entAjax.IE = (ua.indexOf("MSIE") > 0?true:false); // Internet Explorer
entAjax.FF = (ua.indexOf("Firefox") > 0?true:false); // Firefox
entAjax.SF = (ua.indexOf("Apple") > 0?true:false); // Safari
```

Listagem 4. Propriedades para reconhecimento do browser.

Estas propriedades serão usadas ao longo da implementação cross-browsing. Logo, têm-se a nova implementação do *factory method* na Listagem 5.

² "Namespace é uma forma de prevenir que nomes de objetos, propriedades, funções, ou métodos interfiram uns nos outros". (JOHNSON, WHITE, CHARLAND, 2007, p.41)

```

entAjax.XHRFactory = {createXHR:function () {
    try {
        if (entAjax.IE6) {
            xhr = new ActiveXObject("Microsoft.XMLHTTP");
        } else {
            if (entAjax.IE7 || entAjax.FF || entAjax.SF ||
entAjax.OP) {
                xhr = new XMLHttpRequest();
            } else {
                throw ("XHR não suportado em seu browser");
            }
        }
    }
    catch (e) {
    }
    return xhr;
}};

```

Listagem 5. Factory Method reformulado.

Desta forma, qualquer objeto que precise realizar uma requisição terá que fazê-la através do mesmo objeto, mantendo-se assim as requisições estruturadas e organizadas.

3.2 TRATAMENTO DE EVENTOS

AJAX faz uso do paradigma de programação dirigida a eventos (*event-driven programming paradigm*), portanto, é necessário que seja criado um mecanismo de tratamento de eventos implementando boas práticas.

A boa prática na associação de eventos a um elemento HTML é usar a definição do manipulador do evento de forma programática, por exemplo:

```
domFirstName.onclick = nameClicked;
```

É importante que seja testado em qual *browser* será executada a aplicação para manipulação correta do evento, uma vez que o IE tem uma implementação diferente do padrão W3C usado nos demais *browsers*. Caso seja necessário vincular mais de um evento, deve-se associar um evento que encapsule a chamada dos vários eventos necessários. Este padrão está em conformidade com as orientações do W3C no que se refere a separar o código do *design*. É preciso ter cuidado com o método de fluxo de chamada de eventos, o *Event Capture* ou o *Bubbling*.

Uma melhor alternativa ao esquema de vincular evento é o registro de eventos. Neste esquema é possível facilmente adicionar vários eventos a um elemento, porém não há garantia da sequência em que estes eventos serão acionados e, além disso, é

preciso ter uma implementação que trate a forma como o IE funciona e a forma como os demais *browsers* padrão W3C funcionam. O código para tratar registro de eventos de maneira transparente, independente do browser, implementa o padrão de projeto *facade* (SHALLOWAY, TROTT, 2000, p. 87). Para começar, devemos atentar para o fato de que no IE o evento de retorno, ou *callback*, é acionado no contexto de execução do código Java Script, logo a palavra reservada *this* refere-se ao objeto JavaScript. No entanto, nos demais *browsers* o contexto é o próprio elemento onde foi acionado o evento, ou seja, a palavra reservada *this* refere-se ao elemento HTML em si. Abaixo, na Figura 11, segue uma explanação onde será optado por padronizar a segunda abordagem, independente do *browser* usado.

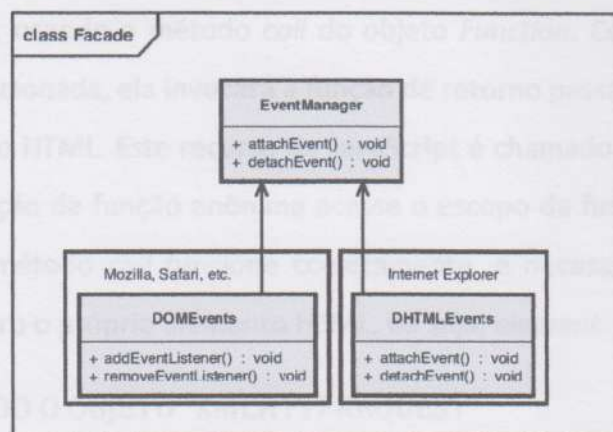


Figura 11. Facade do gerenciador de eventos.

Fonte: Johnson, White e Charland (2007)

```

entAjax.EventManager = {};
entAjax.EventManager.attachEvent = function (element, type, callback,
setCapture) {
  // Checa se o browser é o IE ou um compatível com o W3C
  if (element.attachEvent) {
    // Criar duas propriedades "expando" com referências a funções
    element["ntb_" + type] = function () {
      callback.call(element);
    };
    // Associa ao evento uma função referenciada pela propriedade expando
    element.attachEvent("on" + type, element["ntb_" + type]);
    // Liga o "Capture", se especificado
    if (setCapture) {
      element.setCapture(true);
    }
  } else {
    if (element.addEventListener) {
      element.addEventListener(type, callback, setCapture);
    }
  }
};

```

Listagem 6. Gerenciador de eventos *Cross-Browsing*.

O objeto para tratar os eventos, o `EventManager` demonstrado na listagem 6, é iniciado usando o padrão de projetos *singleton*. Na linha 2, é adicionado ao `EventManager` um método para tratar o registro de eventos independente do *browser*, sua *interface* se assemelha ao padrão W3C: *element* = elemento HTML, *type* = tipo do evento, *callback* = função de retorno e *setCapture* = opção do método de fluxo de chamada de eventos.

Se o *browser* for padrão W3C, será executada apenas a chamada de *addEventListener*, do contrário, será adaptada a execução para o IE. Para isso, é criada via código JavaScript uma propriedade expandida no elemento HTML passado em *element* para referenciar uma função anônima que invocará a função de retorno passada em *callback*, usando o método *call* do objeto *Function*. Com isso, quando a função anônima for acionada, ela invocará a função de retorno passada em *callback* no contexto do elemento HTML. Este recurso do JavaScript é chamado de *closure*, o qual permite que a execução da função anônima acesse o escopo da função externa. Para que a chamada ao método *call* funcione corretamente, é necessário que seja lhe passado por parâmetro o próprio elemento HTML, ou seja, *element*.

3.3 ENCAPSULANDO O OBJETO "XMLHTTPREQUEST"

Para contornar o fato de que JavaScript é *single-threaded*³, realizar requisições assíncronas ao servidor é uma boa alternativa. Ao invés de bloquear a execução no cliente até obter a resposta, o programa continua executando. Quando a resposta do servidor é recebida pelo *browser*, a *thread* JavaScript é usada para executar a função de retorno, ou *callback function*, que fora registrada para aquela requisição em particular antes de ter sido enviada. Esta abordagem incrementa a experiência do usuário porque a aplicação continua disponível enquanto os dados são passados para o servidor e enviados de volta "por detrás das câmeras".

Devido ao fato de que existem várias características que são tratadas de forma diferente entre os *browsers* padrão W3C e o IE, e também para tornar mais confortável o uso do objeto `XMLHttpRequest`, doravante denominado XHR, abstraindo detalhes e tornando mais produtivo o seu uso, é interessante construir um

³ Uma aplicação *single-threaded* é caracterizada por ter apenas uma sequência de execução, ou seja, não suporta processamento paralelo.

framework. Inicialmente, é necessário tratar a criação do objeto XHR. Usando o padrão *singleton* para centralizar a criação e evitar desperdício de memória, usando também o padrão *factory method* para tratar as diferentes abordagens de cada *browser*. O próximo passo é a criação do objeto que será responsável por encapsular a requisição e realizar tratamentos específicos, será chamado de `HttpRequest`, conforme ilustra a Listagem 7.

```
entAJAX.HttpRequest = function () {
    this.handler = "";
    this.async = true;
    this.responseType = "xml";
    this.httpObj = entAJAX.XHRFactory.createXHR();
    this.params = {};
    completeCallback = null;
};

entAJAX.HttpRequest.prototype.get = function () {
    this.httpObj.open("GET", this.handler, this.async);
    this.httpObj.onreadystatechange = entAJAX.close(this,
this.requestComplete);
    if (this.responseType == "xml") {
        this.httpObj.setRequestHeader("Content-Type", "text/xml");
    }
    this.httpObj.send(null);
};

entAJAX.close = function (context, func, params) {
    if (null == params) {
        return function() {return func.apply(context, arguments);};
    } else {
        return function() {return func.apply(context, params);};
    }
};

entAJAX.HttpRequest.prototype.requestComplete = function () {
    if (this.httpObj.readyState == 4) {
        if (this.httpObj.status == 200) {
            completeCallback.call(this, this);
        }
    }
};

entAJAX.HttpRequest.prototype.abort = function () {
    this.httpObj.onreadystatechange = function () {};
    this.httpObj.abort();
};

entAJAX.HttpRequest.prototype.post = function (sData) {
    // Envia os dados providos ou os parâmetros
    if (sData == null) {
        sData = "";
        for (var name in this.params) {
            sData += escape(name) + "=" +
escape(this.params[name]) + "&";
        }
        sData = sData.substring(0, sData.length - 1);
    }
    // Envia os dados usando POST
    this.httpObj.open("POST", this.handler, this.async);
    this.httpObj.onreadystatechange = entAJAX.close(this,
this.requestComplete);
    if (this.responseType == "xml") {
        this.httpObj.setRequestHeader("Content-Type", "text/xml");
    } else {
```

```

        this.httpObj.setRequestHeader("Content-Type", "text/text");
    }
    this.httpObj.send(sData);
};

entAJAX.HttpRequest.prototype.setParam = function (name, value) {
    if (value == null) {delete this.params[name];}
    else {this.params[name] = value;}
};

```

Listagem 7. Classe HttpRequest.

Propriedades:

- *handler*: URL do recurso desejado;
- *async*: indica se será usado o método assíncrono, padrão verdadeiro;
- *responseType*: indica qual o tipo de dados de resposta, padrão XML;
- *httpObj*: objeto XHR, criado usando a técnica descrita *factory+singleton*;
- *completeCallback*: função que tratará o retorno da requisição;
- *params*: parâmetros da requisição a serem passados ao servidor.

Acessando o protótipo do objeto `HttpRequest`, será implementado o seu comportamento adicionando-se outras funções. A função `get()` será usada para requisições usando o método GET. Uma vez que a palavra reservada *this* tem seu valor dependendo de onde é executada, principalmente quando se programa JavaScript orientado a objeto dirigido à eventos. Portanto, para garantir que *this* se trata do objeto XHR, a função `close` implementa este tratamento. Com esta abordagem evita-se referências circulares no DOM e problemas no gerenciamento de memória do IE.

Usando uma *closure* têm-se a vantagem de que funções anônimas podem acessar a variável XHR mesmo depois de a função associada ter terminado sua execução. Desta forma, quando o valor de *readyState* mudar, o objeto *xhr* ainda será acessível no controlador do evento. São usadas funções anônimas e *closures* para tratar as mudanças na propriedade *readyState* do objeto XHR.

O evento *onreadystatechange* notifica quando a requisição está completa e o conteúdo do seu retorno acessível. Por isto, a função `requestComplete` é associada a este evento. Quando o servidor terminou de preencher a requisição e o seu conteúdo estiver acessível, ou seja, quando *readyStatus=4* e *status=200*, então é hora de executar a função que irá realizar o tratamento dos dados retornados.

Existem dois métodos no objeto XHR para acessar o conteúdo da resposta: *responseText* e *responseXML*. O primeiro método traz a resposta como texto simples, a segunda como um documento XML DOM válido. No último caso, é importante a definição do cabeçalho *content-type* da requisição para *text/xml*. A função `abort()` será usada quando se desejar cancelar a requisição assíncrona.

Desta forma, a execução de uma requisição com parâmetros pode ser como ilustra a Listagem 8.

```
var myXHR = new XMLHttpRequest();
myXHR.setRequestHeader("first Name", "John");
myXHR.setRequestHeader("last Name", "Doe");
myXHR.setRequestHeader("id", "1234");
myXHR.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        // Do something with the response
    }
};
myXHR.open("POST", "customers/save", true);
myXHR.send();
```

Listagem 8. Exemplo de uso do objeto XMLHttpRequest.

3.4 INICIANDO A APLICAÇÃO AJAX

A forma mais comum de se inicializar um componente de uma aplicação AJAX é associar um método ao evento *onload*. Este método, doravante denominado *init()*, é uma função JavaScript que será executada depois da página ser totalmente carregada. Existem dois grandes problemas nessa abordagem: o primeiro problema no uso desta técnica é a de possivelmente estar sobrescrevendo outra inicialização de outro componente JavaScript na mesma página; o segundo é ter que esperar o carregamento total da página.

Para resolver o primeiro problema é bastante simples, basta praticar um pouco da política da boa vizinhança. Para tanto, basta que antes de associar o evento *onload* à função *init()*, seja guardada a função *init()* anterior e executá-la logo após.

O segundo problema é mais difícil de ser tratado, devido à forma diferente em que os *browsers* funcionam. A forma mais simples, porém a menos recomendada, é colocar o elemento `<script>` sempre após os elementos da árvore DOM que seriam necessários, no entanto esta abordagem é muito frágil e exige que o *design* conheça do código dos componentes JavaScript, ou seja, a *view* fica atrelada ao código.

Em *browsers* baseados no Mozilla pode ser feito uso do evento *DOMContentLoaded*, o qual é disparado quando todos os elementos DOM são carregados, enquanto que os outros elementos não-DOM ainda estão sendo carregados.

No IE, é possível usar o atributo *DEFER* no elemento `<script>`, que funcionará de maneira análoga a se fosse colocado o `<script>` no final da página. Podem ser feito uso de comentários condicionais para “esconder” este código de demais *browsers*, fazendo com que funcione apenas no IE.

Na verdade, nenhuma das soluções acima é completamente independente de *browser* e são complicadas de agirem em conjunto, portanto, na maioria das vezes, a melhor opção é esperar mesmo o carregamento total da página.

4 FUNDAMENTOS DE JSF

Ao longo deste capítulo serão apresentados as principais características desta tecnologia, bem como os motivos que levaram à sua escolha para a implementação de componentes reusáveis para construção de aplicações RIA com AJAX. Os detalhes de sua especificação estão fora deste foco, apenas os pontos mais relevantes para a criação de componentes serão abordados de uma maneira sucinta e clara. Será mostrado como é possível integrar um framework AJAX em JavaScript a um componente JSF de forma prática, reutilizável e extensível, o que é o objetivo final deste trabalho.

4.1 DEFINIÇÃO

Conforme a JSR-127, "JavaServer Faces (JSF) é um framework de interface com o usuário (UI) para aplicações em Java para a Web. Ele foi desenhado para facilitar significativamente a escrita e a manutenção de aplicações executadas em um servidor de aplicações Java e renderizar as interfaces de usuário de volta para o cliente.

De acordo com Jacobi e Fallows (2006, p. 8), o diferencial que JSF traz frente aos outros *frameworks* similares é justamente ser um padrão JEE. Isso faz com que seja uma prioridade para qualquer construtor de ferramentas JEE no mercado (incluindo Oracle, IBM, SUN e Borland), o que garante uma boa possibilidade de uma escolha de qualidade.

De acordo com Olson (2007), o JSF provê uma clara separação entre a lógica de negócio e as camadas de apresentação. Estabelecendo um mapeamento um-para-um entre requisições HTTP e componentes. Com JSF, há um ciclo de vida que para a comunicação entre a página Web e os *JavaBeans* que fazem o pano de fundo. Há também ouvintes, ou *listeners*, que estão atentos para serem acionados mediante o disparo de um evento e renderizadores que criam a visualização a ser exibida ao usuário. Escrever uma aplicação JSF, segundo eles, não é muito diferente de escrever uma aplicação para *desktop*, em *Swing*, e é radicalmente diferente de escrever uma aplicação Web típica.

Como descrito por Geary e Horstmann (2007), a promessa do JSF é trazer o desenvolvimento rápido de *interfaces* de usuário para o lado servidor em Java, contendo todo o código necessário para gerenciar eventos e organização de componentes. Com isso, o desenvolvedor pode se abstrair destes detalhes e despendar seu esforço na lógica da aplicação.

As principais características do JSF são:

- Um conjunto de componentes de interface pré-fabricados
- Um modelo de programação orientada a eventos
- Um modelo de componentes que permite desenvolvedores estenderem suas funcionalidades criando novos componentes
- Um modelo de navegação declarativo simplificado
- Um ciclo de vida para a aplicação

Apesar do foco principal do JSF ser Web, ele pode ser usado para renderizar a interface do usuário para as mais diversas plataformas, basta para isso que seja desenvolvido o RenderKit (renderizador) específico. Um RenderKit é uma biblioteca de renderizadores com uma tecnologia comum para a renderização (HTML, WML etc). Abaixo, na Figura 12, segue uma ilustração de como pode ser adaptada facilmente a mesma aplicação usando o *framework* para renderizar para outros tipos de *interface* apenas alterando esta camada:

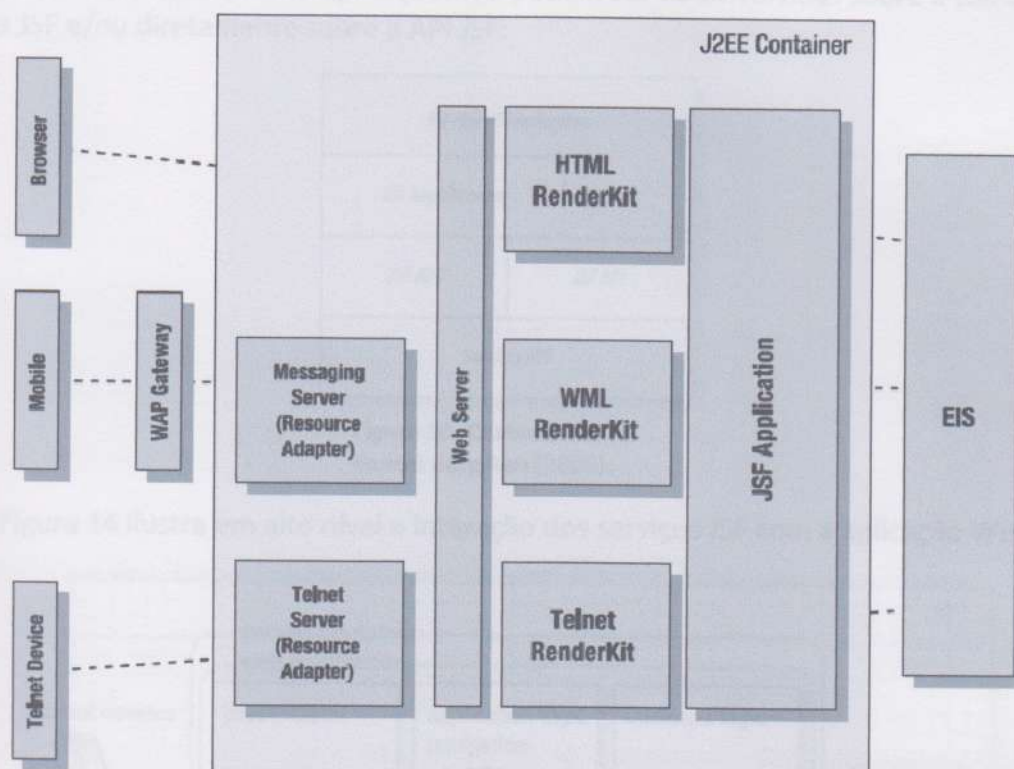


Figura 12. Arquitetura de uma aplicação JEE complexa com JSF e vários RenderKits.
Fonte: Jacobi e Fallows (2006).

Geary e Horstmann (2007) destacam como principais serviços providos pelo *framework*:

- Conversão de dados: os usuários entram os dados no formulário HTML como simples texto, mas os objetos de negócio recebem os dados como datas, números e outros tipos de dados e ainda é possível personalizar e especificar regras de conversão;
- Validação e gerenciamento de erros: validação do tipo "este campo é requerido" ou "este campo deve ser numérico" são facilmente associadas aos elementos da *interface*, e também exibe as mensagens de erro apropriadas para entradas inválidas;
- Internacionalização: JSF pode tratar codificação de caracteres bem como seleção de pacotes de recursos;
- Customização de componentes: este será o foco do próximo capítulo, permite a extensão e criação de componentes simples de usar e reusar;
- Renderizadores alternativos: como comentado anteriormente, apesar do padrão JSF ser HTML, ele pode ser usado para renderizar para outros formatos, como WML, XUL etc;

- Suporte de ferramentas: JSF é otimizado para uso com ferramentas automatizadas e várias IDEs (Interface Development Enviromment) já provêm suporte integrado.

Na sua essência, o JSF é uma API Java desenvolvida sobre a API Servlet. Através de sua biblioteca de tags JSP personalizadas, o *framewrok* abstrai a API para tornar mais fácil o desenvolvimento Web adicionando componentes JSF a páginas JSP. A Figura 13 mostra estas camadas e como aplicações JSF podem ser desenvolvidas sobre a camada de tags JSF e/ou diretamente sobre a API JSF:

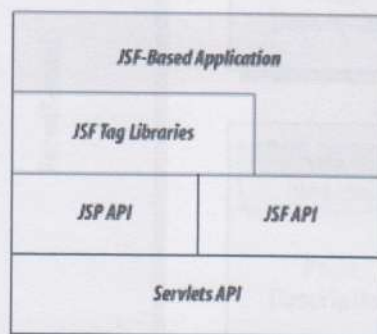


Figura 13. Camadas JSF.

Fonte: Bergsten (2006).

A Figura 14 ilustra em alto nível a interação dos serviços JSF com a aplicação Web:

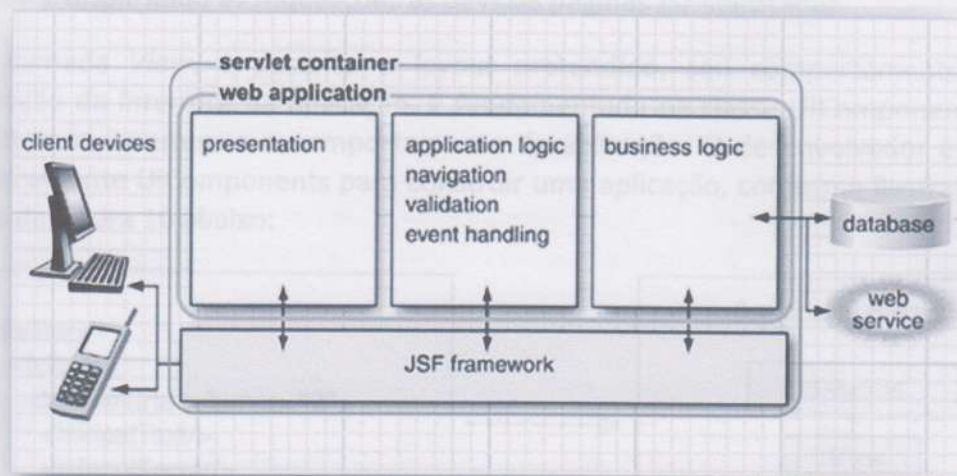


Figura 14. Interação dos serviços JSF com a aplicação Web.

Fonte: Geary e Horstmann (2007).

Outro serviço disponibilizado pelo *framework* que merece um destaque é a arquitetura MVC. O JSF provê a implementação de um MVC nível 2 (Figura 15), conforme mencionado por Moore, Budd e Benson (2007, p. 171), baseado em três elementos: a visualização, o modelo de navegação e a lógica da aplicação.

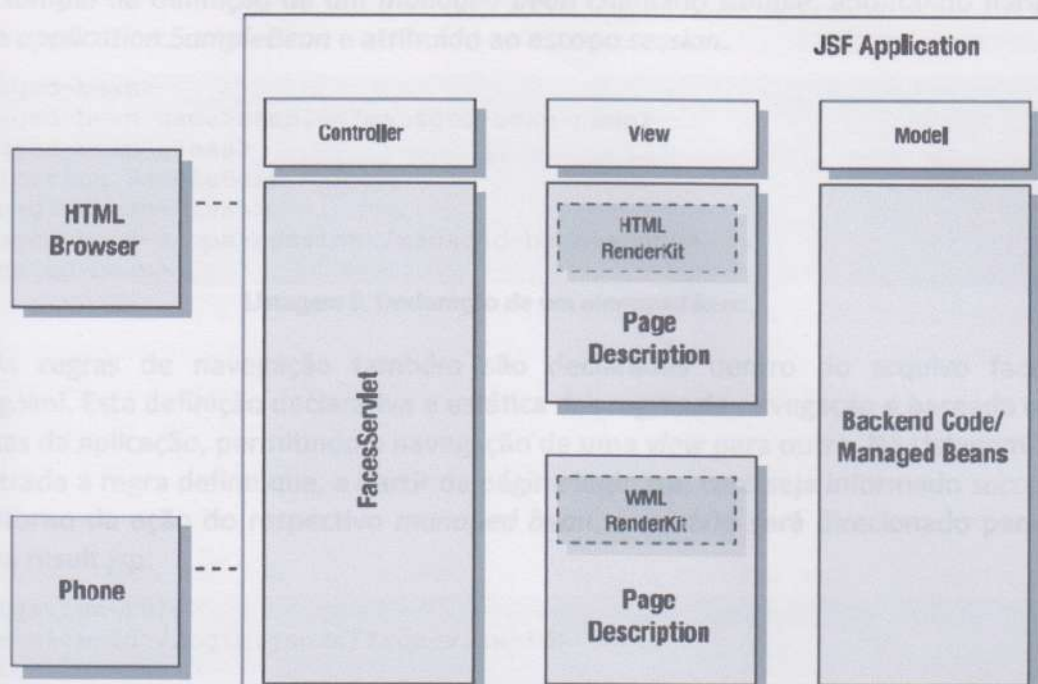


Figura 15. Arquitetura MVC nível 2 do JSF.

Fonte: Jacobi e Fallows (2006)

O *controller* é apenas a classe *FacesServlet*. Ele age controlando o fluxo de navegação e disparando as requisições às devidas páginas JSF solicitadas.

Na camada *View*, é descrito o *layout* pretendido, seu comportamento e a renderização da *interface* da aplicação. É fundamentada na classe *UIComponent* que representam a estrutura e o comportamento da aplicação. O desenvolvedor agrupa hierarquicamente *UIComponents* para construir uma aplicação, conforme ilustrado no exemplo da Figura 16 abaixo:

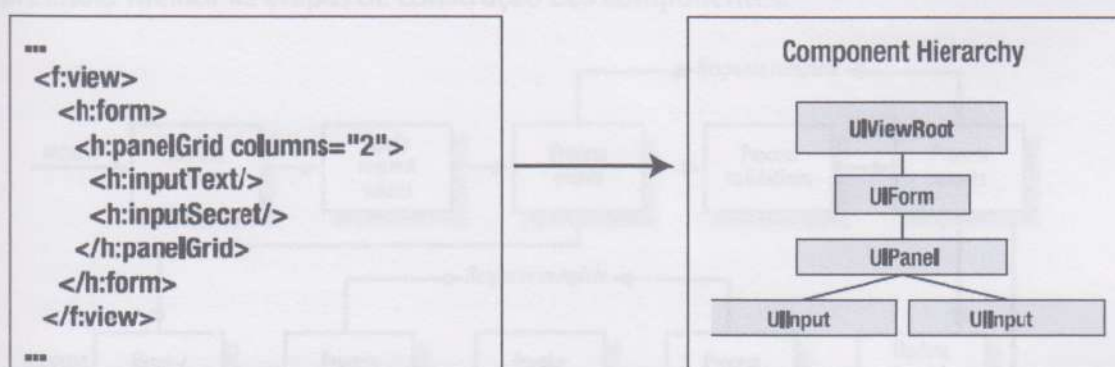


Figura 16. Representação hierárquica da view JSF.

Fonte: Jacobi e Fallows (2006)

O *Model* do JSF é baseado em elementos chamados de *managed beans* e *backing beans*. Definidos no arquivo *faces-config.xml*, estes elementos são classes que não dependem de implementar nenhuma outra *interface* u outra classe específica e realizam a ligação com a lógica da aplicação. O JSF usa o conceito de IoC – inversão de controle, descrito por Fowler (2005), para dispor ao desenvolvedor da aplicação acesso completo a todos os métodos implementados nos *backing beans*. Na Listagem 9 segue

um exemplo de definição de um *managed bean* chamado *sample*, apontando para a classe *application.SampleBean* e atribuído ao escopo *session*.

```
<managed-bean>
<managed-bean-name>sample</managed-bean-name>
<managed-bean-class>
application.SampleBean
</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Listagem 9. Declaração de um *managed bean*.

As regras de navegação também são declaradas dentro do arquivo *faces-config.xml*. Esta definição declarativa e estática das regras de navegação é baseada nas páginas da aplicação, permitindo a navegação de uma *view* para outra. Na Listagem 10 é ilustrada a regra define que, a partir da página *login.jsp*, caso seja informado *success* no retorno da ação do respectivo *managed bean*, o usuário será direcionado para a página *result.jsp*:

```
<navigation-rule>
<from-view-id>/login.jsp</from-view-id>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/result.jsp</to-view-id>
</navigation-case>
</navigation-rule>
```

Listagem 10. Regra de navegação estática.

A navegação em JSF também pode ser definida dinamicamente de forma programática dentro dos métodos dos *managed beans*.

4.2 CICLO DE VIDA

É importante conhecer o funcionamento do ciclo de vida do *framework* para compreender melhor as etapas de construção dos componentes.

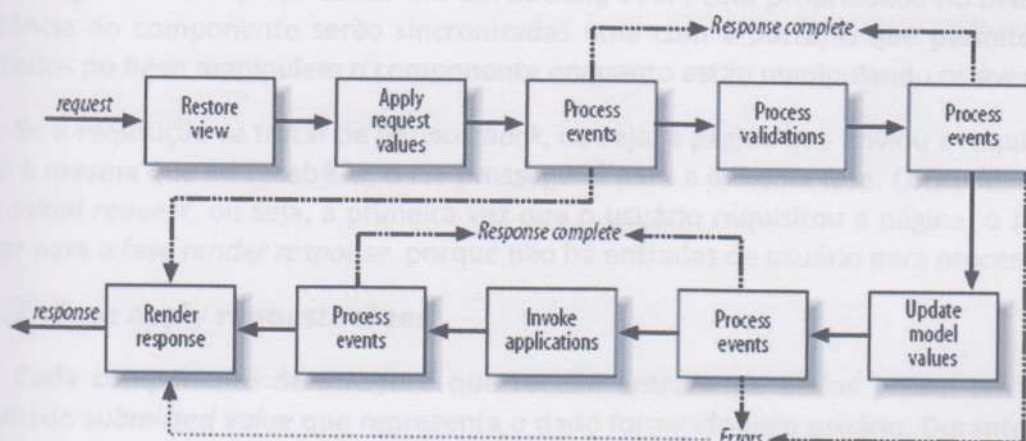


Figura 17. Ciclo de vida JSF.

Fonte: Olson (2007)

A Figura 17 acima ilustra, em uma visão de alto nível, o princípio básico do ciclo de vida do JSF. O ciclo inicia na fase *restore view* e termina na fase *render response*, porém, uma requisição JSF não tem que necessariamente passar por todas estas fases.

Cada fase de *process events* fornece à aplicação a possibilidade de tratar qualquer evento ocorrido durante a fase anterior, dando a resposta da requisição por completa.

Para uma compreensão geral do ciclo de vida do *framework* JSF, será explicada cada fase em seguida, baseado na explanação provida por Mann (2005, p. 57).

4.2.1 Fase restore view

A principal tarefa desta fase é encontrar a visualização corrente e aplicar as entradas do usuário a ela. A visualização representa de forma hierárquica todos os componentes de uma página em particular (ver figura 18 no tópico anterior), podendo ser armazenada no cliente (normalmente em um campo oculto no *browser*) ou, como é o padrão, no servidor (normalmente na sessão). A visualização sempre é iniciada pelo componente *UIViewRoot*, que é um container para todos os outros controles da página.

Quando uma aplicação JSF recebe uma requisição de uma página existente, ela tem que descobrir qual página enviou a requisição para que possa identificar quais eventos foi gerado pelo usuário, e então associá-los aos componentes na página que está sendo enviada.

Se a visão não estiver disponível (o usuário ainda não tinha acessado a página antes), o *framework* descarta a visualização corrente (se existir uma), e cria uma nova baseada no identificador da visualização requisitada. Uma vez que a visão foi recuperada ou criada, ela é armazenada no *FacesContext* corrente, que é a representação contextual da informação associada com a requisição e, eventualmente, com a resposta.

Restaurar a visualização implica em restaurar também os valores dos componentes, assim como os *listeners* dos eventos, validadores ou conversores associados com os componentes da árvore. Se algum dos componentes da árvore estiver ligado a uma propriedade em um *backing bean*, esta propriedade no *bean* e a instância do componente serão sincronizadas uma com a outra, o que permite que métodos no *bean* manipulem o componente enquanto estão manipulando os eventos.

Se a requisição se tratar de um *postback*, ou seja, a página que enviou a requisição será a mesma que irá recebê-la, o JSF prosseguirá para a próxima fase. Contudo, se for um *initial request*, ou seja, a primeira vez que o usuário requisitou a página, o JSF irá pular para a fase *render response*, porque não há entradas de usuário para processar.

4.2.2 Fase apply request values

Cada componente de *interface* que recebe entrada de dados possui um valor chamado *submitted value* que representa o dado fornecido pelo usuário. Durante esta fase, o *framework* preenche o *submitted value* baseado nos parâmetros passados na requisição. Este processo é chamado de decodificação, ou *decoding*.

Quando o JSF codifica um componente como HTML, o identificador enviado ao *browser* é composto com o um prefixo que é o identificador do componente acima dele na árvore de componentes que compõem a visualização. Este atributo, normalmente atribuído ao *id* do elemento HTML no *browser*, é chamado de

identificador do cliente, ou *client identifier*. Nesta fase, o JSF solicita a cada componente que decodifique a si próprio. O componente, ou seu renderizador, faz esta tarefa procurando em primeiro lugar por um parâmetro que tenha o mesmo nome do seu *client identifier*. Uma vez que o parâmetro é encontrado, o valor do componente é atribuído ao valor do parâmetro.

Todo componente de entrada de dados tem um parâmetro do tipo booleano chamado *immediate*. Se esta propriedade for verdadeira, a validação será realizada nesta fase, ao invés da fase *process validation*. No caso de botões ou *hiperlinks*, se *immediate* for verdadeiro, eles irão acionar eventos durante esta fase também, sendo muito útil nos casos em que um botão cancelar ignora todas as entradas de um formulário, ou um *hiperlink* que aceita valores apenas de um controle específico etc.

Outro parâmetro enviado pela requisição é o *welcomeForm*. Quando o usuário submete o formulário, o respectivo componente *HtmlForm* tem sua propriedade *submitted value* associada a verdadeiro, tornando possível à aplicação realizar um processo específico na lógica de negócio dependendo de qual formulário o usuário submeteu, o que é muito útil numa *view* com vários formulários.

Nesta fase, os parâmetros da requisição referentes a ações de um formulário são convertidos em ações de eventos por controles como o *HtmlCommandButton* de um respectivo *client identifier*. Uma vez que uma ação foi criada, ela é adicionada ao *FacesContext*, para posterior execução na fase *invoke application*. Ou seja, o código de decodificação pode também adicionar eventos ou realizar outras operações baseado na requisição.

Depois que esta fase é completada, há um *broadcast* de todos os eventos existentes aos seus devidos *listeners*. O ciclo de vida pode ser interrompido e direcionado direto para a fase *render response*, ou ainda ser terminado completamente (caso ele mesmo renderize a resposta), por qualquer renderizador, ou componente, ou *listener*. Caso não seja interrompido, cada componente fez sua própria decodificação e conseqüentemente terá um valor que será seu estado atualizado baseado na requisição.

4.2.3 Fase *process validations*

Com os componentes já tendo seus valores atualizados ao final da fase anterior, nesta fase, o *framework* percorre toda a árvore de componentes da *view* e pergunta a cada um deles se o seu respectivo valor submetido pela requisição (*submitted value*) é aceitável. Mas, antes da validação, cada *submitted value* é convertido, ou usando um conversor específico registrado para o componente, ou usando o conversor padrão. Daí a validação é então realizada pelo próprio componente, ou delegada a um ou vários validadores.

Uma vez que o componente tem seu *submitted value* validado seu valor local é atribuído a este valor, já no tipo de dado apropriado. Se este valor local do componente for alterado, também serão disparados eventos. Neste ponto, todos os eventos de alteração de valores, bem como todos os outros eventos associados a esta fase, são acionados e consumidos pelos seus respectivos *listeners*, os quais têm a opção de responder diretamente ou saltar para a fase *render response*.

Se a conversão e a validação ocorrem com sucesso para todos os componentes que tiveram valores submetidos, o ciclo de vida passa para a próxima fase. Do contrário, poderá saltar para a fase *render response* com as mensagens de erro da conversão e/ou validação.

4.2.4 Fase update model values

Com todos os componentes com seus valores atualizados, validados e convertidos para os seus devidos tipos de dados, é hora de conversar com os *backing beans* associados ou com os objetos do modelo. O JSF usa sua linguagem de expressões, JSF EL (JavaServer Faces Expression Language), para associar objetos (*beans*) aos seus componentes, como mostra a Listagem 11.

```
<h:inputText id="helloInput" value="#{helloBean.numControls}"
            required="true">
</h:inputText>
```

Listagem 11. Exemplo JSF EL

Esta é a fase onde estas expressões são avaliadas e as propriedades destes objetos associados são atualizadas baseadas nos valores locais dos componentes. No exemplo acima, a propriedade *value* do componente é a expressão `#{helloBean.numControls}` que associa o objeto *helloBean* ao componente JSF `<h:inputText>`. Uma instância deste objeto será procurada em todos os escopos disponíveis – *request*, *session* ou *application*. Uma vez encontrado o componente irá atribuir à propriedade específica, neste caso, *numControls*, o seu valor local atualizado.

Uma vez que esta fase for finalizada, o JSF irá disparar um *broadcast* de todos os eventos para todos os *listeners* interessados. Novamente, qualquer *listener* pode renderizar por si próprio a resposta e finalizar o processo, ou pode direcionar direto para a fase *render response*.

É importante destacar que, neste ponto do ciclo de vida, todos os componentes da *view* estão com seus valores atualizados com base na entrada do usuário, com todos estes valores validados e todos os *beans* associados atualizados sem seja escrita nenhuma linha de código pelo desenvolvedor da aplicação.

4.2.5 Fase invoke applications

Finalmente, agora com todo o modelo atualizado, é possível focar nas regras de negócio da aplicação. Nesta fase, o JSF realiza um *broadcast* dos eventos desta fase para todos os *listeners* registrados. Os eventos da fase *apply requested values*, como o *HtmlCommandButton* comentado anteriormente no tópico relativo a esta fase, na fase *invoke application*, o JSF irá enviar as ações de eventos como aqueles para todos os *listeners* registrados para aquela ação naquele componente.

No Listagem 12 abaixo, há dois *listeners* registrados: o método restaurado na fase *restore view* (o *action listener*), e o *action listener* padrão (o qual é automaticamente registrado para todos os componentes de comando)

```
<h:commandButton id="redisplayCommand" type="submit" value="Redisplay"
                actionListener="#{helloBean.addControls}" />
```

Listagem 12. Exemplo de registro de um ActionListener.

A propriedade *actionListener* é a expressão JSF EL `#{helloBean.addControls}`. Nesta fase, o JSF avalia a expressão e executa o método *addControls* da instância de *helloBean* armazenada no escopo em que foi definido na sua respectiva declaração no arquivo *faces-config.xml*. Depois que o método do *bean* for executado é a vez do *action listener* padrão. Ele irá então delegar o controle para quaisquer métodos de ação (*action methods*) que estejam registrados para o componente que acionou o evento, e então escolherá para qual página baseado na lógica de navegação estática definida no arquivo *faces-config.xml*. No caso acima, não há métodos de ação vinculados, o que seria feito através da propriedade *action*. Quando não houver regra de navegação definida, o *action listener* padrão irá simplesmente re-exibir a visualização corrente.

Os *action listeners* e os *actions methods* podem realizar várias operações, como renderizar a resposta por si mesmos, adicionar mensagens da aplicação, gerar eventos, e ainda executar a lógica da aplicação. Eles também podem saltar para a fase *render response* para evitar que qualquer outro evento da aplicação seja processado. Os *action methods* são integrados ao sistema de navegação, portanto, eles podem definir também de forma programática determinar uma navegação dinâmica para a próxima página a ser carregada.

4.2.6 Fase render response

Neste ponto, todo o processamento realizado pelo *framework* e pela aplicação já tem sido realizado. Agora só falta enviar a resposta de volta para o usuário, e este é o objetivo primário desta fase. O segundo objetivo é salvar o estado da *view* para que ela possa ser restaurada na fase *restore view* para quando o usuário requisitá-la novamente. O estado da *view* é salvo nesta fase porque usualmente é armazenado no cliente, portanto, esta parte deve ser retornada para ele. No caso de ser armazenada no servidor, a *view* será armazenada muito provavelmente na sessão do usuário dentro do servidor de aplicação.

Como foi dito anteriormente, a renderização da resposta ao usuário pode ser realizada com outras tecnologias, não apenas para a Web, a implementação do JSF é livre para que sejam implementadas também outras abordagens. Durante o processo de codificação nesta fase, para cada componente, os conversores também são invocados para transformar os valores dos componentes no texto que será exibido para o usuário no *browser*, no caso de visualização na Web. Esta é a fase final do ciclo de vida do processamento de uma requisição JSF.

5 CRIAÇÃO DE COMPONENTES JSF

De acordo com Mann (2005), o real poder do JSF se encontra no seu modelo de componentes e na sua arquitetura extensível, permitindo que sejam criadas novas *interfaces* com o usuário estendendo os componentes de base do *framework*. Desta forma, tanto componentes, como renderizadores, validadores e conversores podem ser personalizados e adequados às necessidades de cada cenário de negócio.

Sempre vale lembrar que o objetivo principal na criação de componentes é abstrair a complexidade de uma *interface* mais elaborada, assim, o desenvolvedor da aplicação pode se prender mais às regras de negócio que estão sendo implementadas do que a detalhes de construção de *interfaces*.

Novamente, Mann (2005), sugere que antes de pensar em fazer um componente é melhor se perguntar "O que exatamente eu preciso fazer?". Como resposta para este questionamento é citada situações nas quais é uma boa solução o desenvolvimento de novos componentes:

- Prover suporte a características padrões HTML, como *frames* ou *upload* de arquivos, que não são suportadas pelos componentes padrões;
- Prover funcionalidades não visuais para um componente de *interface* existente, como ordenar uma grade automaticamente ao clicar no cabeçalho de uma coluna;
- Prover efeitos visuais mais elaborados aos componentes já existentes, apenas estendendo o renderizador do componente é possível adicionar código JavaScript para torná-lo esteticamente mais atraente;
- Criar um tipo completamente novo de componente, como um leitor de RSS, uma barra de tarefas, árvores de componentes etc.
- Prover renderização para outros tipos de linguagens de marcação, para serem usados, por exemplo, em dispositivos móveis.

Um novo componente JSF implica na criação de uma marcação JSP nova, a ser usada em uma página através da declaração de uma nova biblioteca de marcações (*taglib*). Abaixo segue a Listagem 13 com um exemplo simples de um componente sendo usado em uma página, seu nome é *spinner*, importado para a página através do uso da *taglib* designada pela URI <http://corejsf.com/spinner>, associada ao *namespace* *corejsf*:

```
<%@ taglib uri="http://corejsf.com/spinner" prefix="corejsf" %>
...
<corejsf:spinner value="#{cardExpirationDate.month}"
    id="monthSpinner" minimum="1" maximum="12" size="3"/>
<h:message for="monthSpinner"/>
```

Listagem 13. Exemplo de uso de um componente JSF em uma página.

Neste capítulo será abordada a criação de componentes JSF destinados ao ambiente Web, ou seja, para criação de aplicações RIA que é o foco do trabalho. Apesar de ser possível também criar novos validadores e conversores especializados, não será discutida neste trabalho a personalização desses elementos JSF.

- Uma lista de componentes filhos;
 - Um mapa de componentes *facet* (Similares a componentes filhos, porém um *facet* tem uma chave e não uma posição em uma lista, cada componente define o *layout* de seus *facets*);
 - Um mapa de atributos para serem usados para quaisquer propósitos, na forma de pares do tipo “nome-valor”;
 - Um mapa de valores de expressões. Similar ao mapa anterior, só que o valor é uma expressão JSF EL;
 - Uma coleção de *listeners*, mantida pelo próprio *framework* JSF.
2. `javax.faces.webapp.UIComponentELTag`: Esta classe, disponível a partir do JSF 1.2, faz o que sua antecessora (`UIComponentTag`) fazia no JSF 1.1, só que agora também tratando expressões JSF EL. Esta classe provê a *interface* da *tag* JSP que corresponde a uma instância de um *UIComponente* na *view*. Como no JSF 1.2 todas estas *tags* são `BodyTag JSP`, toda a página é executada criando a árvore completa de componentes antes que seja renderizada. Assim, *tags* JSP e JSF podem conviver harmonicamente;
 3. `javax.faces.context.FacesContext`: Esta classe contém informação relacionada à requisição JSF. Entre outras coisas, é possível acessar os parâmetros da requisição, recuperar uma referência ao objeto *Application*, recuperar a árvore de componentes da visualização corrente, ou recuperar uma referência para o escritor de resposta (*ResponseWriter*), o qual será usado para codificar as marcações;
 4. `javax.faces.application.Application`: Esta classe mantém o rastreamento dos objetos compartilhados dentro de uma aplicação, como as localidades suportadas, e os conversores e validadores disponíveis. Serve também como uma fábrica, implementando um *factory method* para componentes, conversores e validadores, e ainda, para obter uma fábrica de expressões para expressões de métodos e valores.
 5. `javax.faces.context.ResponseWriter`: Nesta classe estão disponibilizados métodos para facilitar a geração das marcações HTML. Ela tem métodos para iniciar, finalizar e escrever atributos de elementos HTML.

Geary e Horstmann (2005) citam como estratégias para o desenvolvimento de novos componentes JSF duas abordagens mais comuns que serão detalhadas a seguir.

5.1.1 Sem Renderizador Específico

Esta abordagem permite a criação de um novo componente com apenas duas novas classes:

- Uma classe para processar os atributos da nova marcação JSP – que, por convenção, tem como sufixo a palavra *Tag*. Esta classe cria o componente e transfere os valores dos atributos da marcação para os atributos e propriedades do componente;
- Uma classe que mantém o estado do componente, renderiza sua interface e processa entrada de dados – convencionalmente, tem o prefixo “UI”. Nesta classe será implementado o mecanismo de renderização pela

codificação de marcações HTML, bem como o mecanismo de processamento da entrada pela decodificação da requisição HTTP.

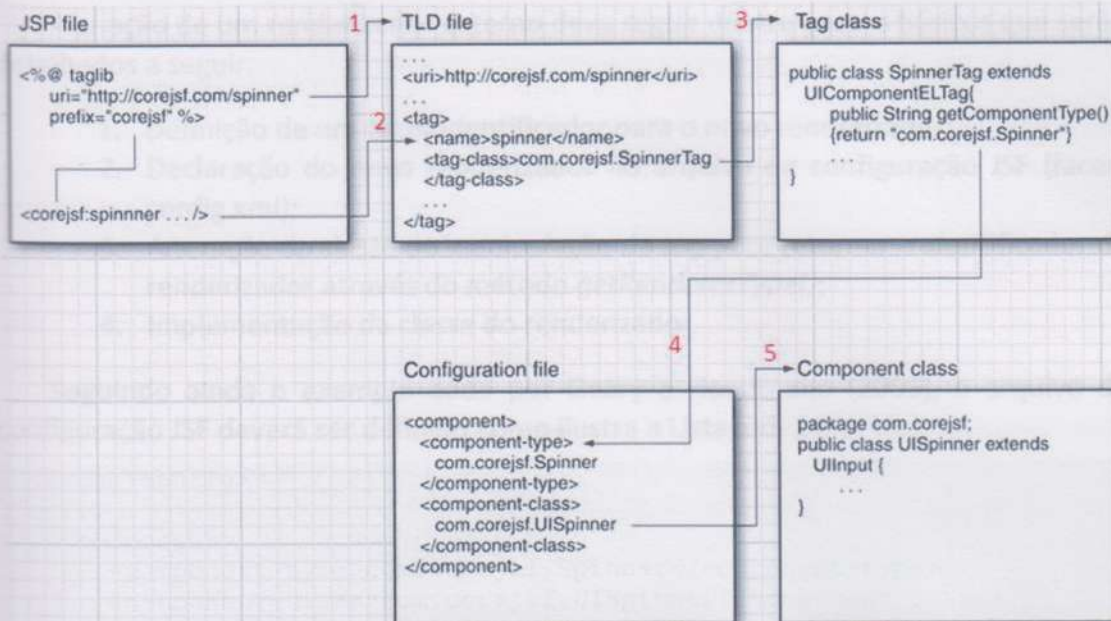


Figura 19. Novo componente sem renderizador específico.

Fonte: Geray e Horstmann (2007).

Na Figura 19 está um exemplo de criação de um novo componente JSF sem renderizador específico. Cada passo destacado na figura é explicado abaixo:

1. No arquivo JSP da página que faz uso do componente há uma declaração da *taglib* (arquivo TLD) onde está o componente denotado pelo URI `http://corejsf.com/spinner`, definindo o *namespace corejsf*;
2. O uso efetivo do componente se dá pela utilização do seu nome definido no arquivo TLD (*spinner*);
3. O arquivo TLD tem a declaração do componente apontando para a primeira classe do esquema, a qual irá processar os atributos JSP, a classe *SpinnerTag* (esta classe precisa estender *UIComponentTag* ou *UIComponentELTag*);
4. Na classe *SpinnerTag* há necessariamente o método *getComponentType*, o qual retorna o nome completo do tipo do componente que precisa estar declarado no arquivo de configuração do JSF, o *faces-config.xml*;
5. No arquivo *faces-config.xml*, na declaração do componente, está a definição do nome completo da segunda classe deste esquema, esta classe manterá o estado do componente, renderizará sua *interface* e processará a entrada de dados.

5.1.2 Com Renderizador Específico

Apesar de mais simples, o esquema anterior traz um alto acoplamento entre o comportamento do componente e a sua renderização. Ou seja, caso a aplicação precise agora deste componente sendo renderizado para um aparelho de telefone celular, será necessário alterar o componente. Para resolver este problema, a

renderização deve ser separada, criando uma classe específica para este fim, ou seja, um renderizador externo.

A criação de um renderizador externo deve seguir quatro passos básicos que serão detalhados a seguir.

1. Definição de um texto identificador para o novo renderizador;
2. Declaração do novo renderizador no arquivo de configuração JSF (`faces-config.xml`);
3. Alteração da classe de manipulação da *tag* para retornar o identificador do renderizador através do método `getRendererType()`;
4. Implementação da classe do renderizador.

Seguindo ainda o exemplo dado por Geary e Horstmann (2005), o arquivo de configuração JSF deverá ser definido como ilustra a Listagem 14.

```
<faces-config>
...
<component>
  <component-type>com.corejsf.Spinner</component-type>
  <component-class>com.corejsf.UISpinner</component-class>
</component>

<render-kit>
  <renderer>
    <component-family>javax.faces.Input</component-family>
    <renderer-type>com.corejsf.Spinner</renderer-type>
    <renderer-class>com.corejsf.SpinnerRenderer</renderer-class>
  </renderer>
</render-kit>
</faces-config>
```

Listagem 14. Declarações no `faces-config.xml` do componente `Spinner`.

Nesta configuração foi omitido o elemento `<render-kit-id>`. Quando isto acontece, é usado o valor padrão para este elemento, que é o padrão de renderização da *interface* JSF em HTML (`RenderKitFactory.HTML_BASIC_RENDER_KIT`).

O elemento *component-family* serve para estabelecer a reusabilidade de um renderizador por vários componentes. Ao invés de identificar individualmente os renderizadores de cada componente, os renderizadores são determinados pelo *renderer-type* e o *component-family*. Geary e Horstmann (2005), assumem como uma boa prática a definição do *renderer type* também no construtor da classe do componente. Com isso, se o componente for manipulado programaticamente, será possível identificar seu renderizador sem a necessidade da *tag*.

5.2 CRIAÇÃO DO COMPONENTE USANDO BOAS PRÁTICAS

Mann (2005) definiu como sendo os três passos chave para criação de novos componentes JSF:

1. Estender uma classe apropriada da base de componente JSF, implementando as *interfaces* necessárias;
2. Adicionar as configurações ao arquivo de configuração JSF;

3. Integrar a nova classe de componente com uma tecnologia de exibição, como o JSP

Esses três passos chave definem, em alto nível, como sendo uma boa prática a estratégia definida anteriormente de escrever um renderizador específico.

É citada também como uma boa prática, por Jacobi e Fallows (2006), a criação de um protótipo de *interface*, antes dos passos definidos acima. Quando é criado um protótipo de *interface* na própria linguagem de marcação pretendida, no caso HTML, além de servir como base para testar o resultado do componente, servirá também para encontrar quais elementos o componente precisará gerar e quais atributos são específicos da sua renderização. Desta forma, é possível visualizar o que será parametrizado na geração das marcações.

A seguir será abordado o passo-a-passo básico para a criação de um componente JSF, considerando as boas práticas citadas. Será tomado por base o exemplo dado por Geary e Horstmann (2005) e mesclando com as boas práticas citadas por Jacobi e Fallows (2006) e Mann (2005).

5.2.1 Criando o protótipo da interface

Como discutido anteriormente, a criação do protótipo da *interface* do componente é uma boa prática a ser seguida, deve ser usada para dar melhor visão sobre a construção do componente. Abaixo segue a Listagem 15 com o código em HTML relevante ao protótipo do componente que será desenvolvido:

```
...
<style type="text/css" >
    .overlayLess
    {
        position:relative;
        left:0px;
        bottom:-5px;
        cursor:pointer;
        background-color:transparent;
    }
    .overlayMore
    {
        position:relative;
        left:-2px;
        bottom:-6px;
        cursor:pointer;
        background-color:transparent;
    }
</style>
...
<div title="Spinner Component">
    <input name="spinnerForm:monthSpinner" value="2" size="3" />
    
    
</div>
...
```

Listagem 15. Protótipo HTML do componente Spinner.

Com o protótipo definido, fica mais claro definir quais deverão ser os parâmetros necessários. Além disso, é fácil destacar quais os parâmetros que precisarão ser criados especificamente para o novo componente. Na Listagem 16 são destacados os parâmetros desejados entre colchetes.

```
...
<div title=[título]>
  <input name=[clientId] value=[valor_convertido] size=[tamanho]
  minimum=[valor_mínimo] maximum=[valor_máximo]/>
  
  
</div>
...
```

Listagem 16. Protótipo parametrizado do componente Spinner.

O resultado do protótipo é exibido na Figura 20, onde mostra simplesmente um campo de entrada de dados e as duas figuras que indicam a funcionalidade de incremento e decremento do componente.

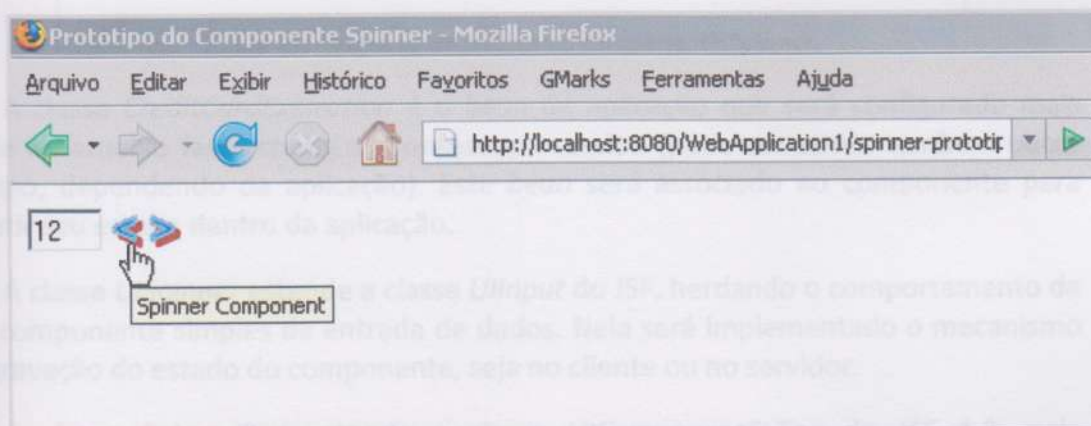


Figura 20. Tela do Protótipo do Componente Spinner.

A partir deste protótipo, a definição do uso do novo componente JSF dentro da página JSP, substituindo as marcações HTML, será de acordo com a Listagem 17.

```
...
<corejsf:spinner id="monthSpinner" title="Componente Spinner"
  value="#{cardExpirationDate.month}" minimum="1" maximum="12"
  size="3"/>
...
```

Listagem 17. Definição da declaração do componente Spinner.

A seguir, serão criadas as peças integrantes do componente que será usado como demonstrado acima e renderizará a interface como descrito no protótipo.

5.2.2 Design das classes necessárias

No diagrama de classes simplificado na Figura 21 são destacadas as classes que são necessárias.

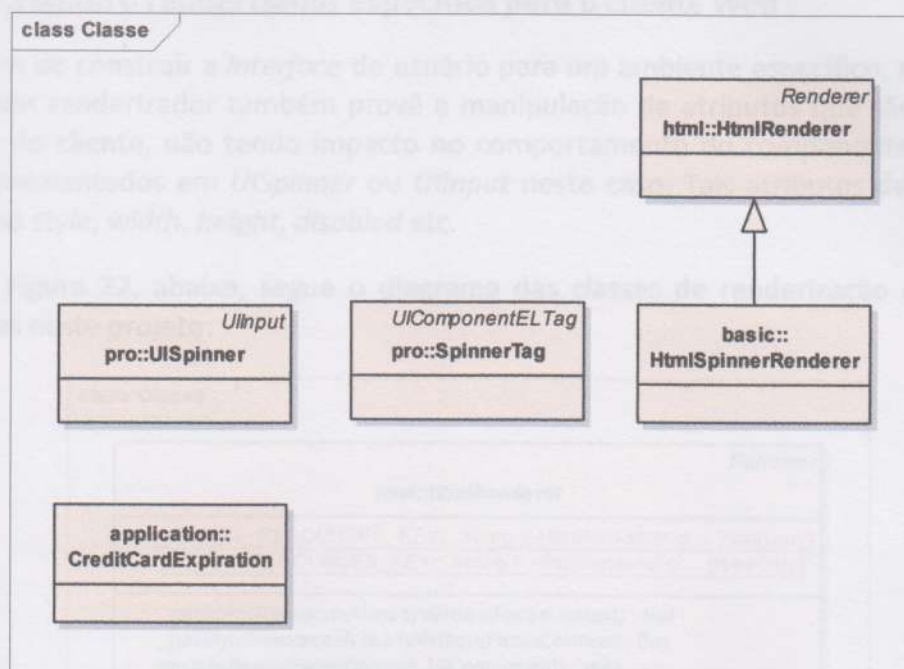


Figura 21. Diagrama de classes do componente Spinner.

A classe *CreditCardExpiration* é o *bean* da aplicação que será configurado mais tarde no arquivo *faces-config.xml* no escopo *session* (pode ser usado também outro escopo, dependendo da aplicação). Este *bean* será associado ao componente para refletir seu estado dentro da aplicação.

A classe *UISpinner* estende a classe *UIInput* do JSF, herdando o comportamento de um componente simples de entrada de dados. Nela será implementado o mecanismo de gravação do estado do componente, seja no cliente ou no servidor.

A classe *SpinnerTag* estende a classe *UIComponentELTag* do JSF 1.2, nela implementar-se-á o tratamento dos parâmetros da *tag* JSP do novo componente JSF. Será referenciada pelo arquivo "tld" que será desenvolvido a seguir.

Tanto a classe *UISpinner* como a *SpinnerTag* estão definidas dentro do sub-pacote "pro", mas dentro de pacotes diferentes: *com.posfarn.component.pro.UISpinner* e *com.posfarn.taglib.pro.SpinnerTag*.

A divisão de pacotes adotada torna mais organizada a visão funcional do projeto do componente.

A classe *HtmlRenderer* estende a classe *Renderer* do JSF, esta classe auxiliará nas tarefas comuns a serem implementadas pelos renderizadores de componentes HTML que venham a ser desenvolvidos. Nela estão métodos que irão, a princípio, ajudar com o manuseio de recursos que precisarão ser adicionados à página para que o componente funcione a contento, inicialmente, arquivos JavaScript e CSS.

Finalmente, a classe *HtmlSpinnerRenderer* estende a classe *HtmlRenderer* para implementar a renderização propriamente dita do novo componente JSF em marcações HTML.

5.2.3 Criando o renderizador específico para o cliente Web

Além de construir a *interface* de usuário para um ambiente específico, neste caso HTML, um renderizador também provê a manipulação de atributos que são restritos ao lado do cliente, não tendo impacto no comportamento do componente, ou seja, não implementados em *UISpinner* ou *UIInput* neste caso. Tais atributos de *interface* são como *style*, *width*, *height*, *disabled* etc.

Na Figura 22, abaixo, segue o diagrama das classes de renderização que serão utilizadas neste projeto:

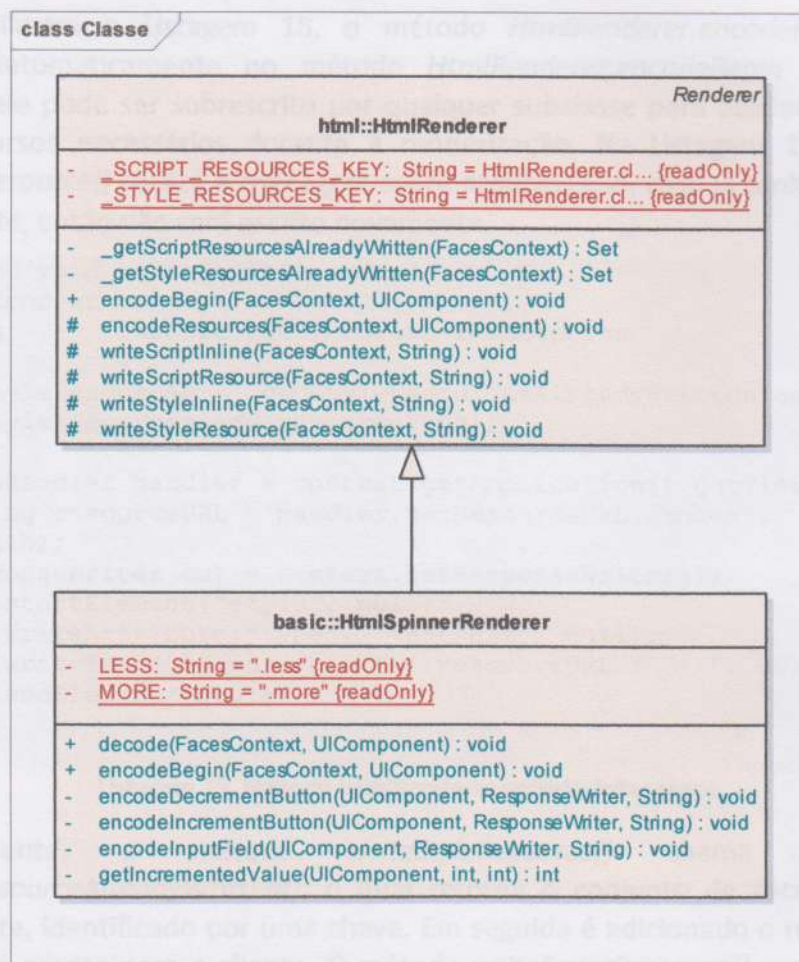


Figura 22. Classes de renderização do componente Spinner.

5.2.3.1 A classe *HtmlRenderer*

Nesta classe são codificados alguns métodos convenientes para o tratamento de recursos necessários pelo renderizador HTML. Se um desenvolvedor incluir múltiplos componentes de um mesmo tipo em uma página, fatalmente, um recurso usado pelo componente, tal como um arquivo CSS ou JavaScript, será importado várias vezes para a página desnecessariamente.

A semântica por trás dos métodos de *HtmlRenderer* garantirão que a importação dos recursos necessários seja escrita apenas uma vez, não importando quantas vezes o componente seja usado na página.

```

    public void encodeBegin(FacesContext context, UIComponent
component) throws IOException {
        // escreve os recursos
        encodeResources(context, component);
    }

    protected void encodeResources(FacesContext context, UIComponent
component) throws IOException {
        // deixado em branco para que subclasses reescrevam quando
necessário
    }

```

Listagem 18. Métodos de `HtmlRenderer` usados para a codificação.

Como ilustra a Listagem 18, o método `HtmlRenderer.encodeResources()` é chamado automaticamente no método `HtmlRenderer.encodeBegin` (herdado de `Renderer`), ele pode ser sobrescrito por qualquer subclasse para adicionar quaisquer outros recursos necessários durante a renderização. Na Listagem 19, o método `writeStyleResource()` realiza a checagem sobre arquivos CSS, caso já tenha sido escrito para o cliente, então não será escrito novamente.

```

protected void writeStyleResource(
    FacesContext context,
    String resourcePath) throws IOException
{
    Set styleResources = _getStyleResourcesAlreadyWritten(context);
    if (styleResources.add(resourcePath))
    {
        ViewHandler handler = context.getApplication().getViewHandler();
        String resourceURL = handler.getResourceURL(context,
resourcePath);
        ResponseWriter out = context.getResponseWriter();
        out.startElement("style", null);
        out.writeAttribute("type", "text/css", null);
        out.writeText("@import url(" + resourceURL + ")", null);
        out.endElement("style");
    }
}

```

Listagem 19. Método `HtmlRenderer.writeStyleResource`.

Inicialmente, o método `writeStyleResource()` chama o método `_getStyleResourceAlreadyWritten()`, o qual retorna o conjunto de recursos escritos para o cliente, identificado por uma chave. Em seguida é adicionado o recurso se não já existir e é escrito para o cliente. O método `writeScriptResource()` realiza a mesma semântica para arquivos JavaScript.

O método `_getStyleResourceAlreadyWritten()`, ilustrado na Listagem 20, implementa a semântica de somente uma escrita do recurso através da adição da chave `_STYLE_RESOURCE_KEY` com um mapa associado ao escopo da requisição no servidor.

```

private Set _getStyleResourcesAlreadyWritten(
    FacesContext context)
{
    ExternalContext external = context.getExternalContext();
    Map requestScope = external.getRequestMap();
    Set written = (Set)requestScope.get(_STYLE_RESOURCES_KEY);

    if (written == null)

```

```

{
    written = new HashSet();
    requestScope.put(_STYLE_RESOURCES_KEY, written);
}

return written;
}

```

Listagem 20. Método `HtmlRenderer.getStyleResourcesAlreadyWritten`.

Similarmente, é realizado o mesmo procedimento para recursos JavaScript através do método `_getScriptResourceAlreadyWritten()`.

Finalmente, as chaves são implementadas usando o nome completo da classe, neste caso, `com.posfarn.render.html.HtmlRenderer`, e acrescentando `STYLE_WRITTEN` ou `SCRIPT_WRITTEN` para distinguir entre os recursos de *script* e de estilos, como demonstra a Listagem 21.

```

static private final String _STYLE_RESOURCES_KEY =
    HtmlRenderer.class.getName() + ".STYLES_WRITTEN";
static private final String _SCRIPT_RESOURCES_KEY =
    HtmlRenderer.class.getName() + ".SCRIPTS_WRITTEN";

```

Listagem 21. Constantes chave para os recursos.

5.2.3.2 A classe `HtmlSpinnerRenderer`

As classes `UIComponent` e `Renderer` são parte do contrato para escrita de renderizadores específicos de um cliente. O `FacesContext` é a parte do contrato responsável por acionar os métodos `decode()`, `encodeBegin()`, `encodeChildren()` e `encodeEnd()`, e todas as marcações são escritas para o cliente usando um `ResponseWriter`.

A classe `UllInput` é a superclasse comportamental que será provida com a nova classe que estende a superclasse de rederização, `Renderer`. A classe `ExternalContext` permite acesso aos parâmetros da requisição, permitirá também às aplicações baseadas em JSF executem tanto em um ambiente *servlet* quanto em um ambiente *portlet* permitindo acesso à instância da sessão, ao objeto de resposta, à informação de caminhos (*paths*) etc.

A classe `HtmlSpinnerRenderer` realizará a codificação do componente nas marcações HTML através dos métodos:

- `encodeBegin()`: é geralmente usado para escrever a marcação de abertura do elemento do componente, por exemplo (`<table>`);
- `encodeChildren()`: o *framework* JSF irá executar este método apenas se o retorno do método `getRendersChildren()` for verdadeiro, ou seja, se este componente tem elementos que estão abaixo dele na hierarquia de componentes. O retorno padrão de `getRendersChildren()` é falso;
- `encodeEnd()`: é geralmente usado para escrever a marcação de fechamento do elemento do componente, por exemplo (`</table>`).

No final do ciclo de vida JSF, na fase *render response*, estes métodos são executados, e nesta mesma ordem acima.

No caso do componente *Spinner* que está sendo construído neste exemplo, não será necessário implementar o método *encodeChildren()*, pois não há elementos hierarquicamente dependentes nele.

5.2.3.2.1 Manipulação dos valores dos atributos

Usualmente, componentes têm atributos específicos que precisam ser renderizados, além daqueles que seriam os padrões para uma marcação específica. O componente *Spinner* tem definido alguns parâmetros HTML (*title*, *name*, *value*, *size*, *minimum* e *maximum*) que são renderizados usando o identificador (*clientId*) do componente e o valor convertido do atributo.

O método *getIncrementedValue()*, implementado na classe *HtmlSpinnerRenderer*, recupera o valor dos atributos *minimum* e *maximum* para validar e atualizar o valor do componente com o novo valor entrado pelo usuário. O método *getAttributes()* retorna um mapa de atributos (e propriedades) associado a este *UIComponent*, chaveando pelo nome do atributo, e daí acessa-se o valor do respectivo atributo.

5.2.3.2.2 Identificação do componente

Como a classe *Renderer* é um *singleton*, é necessário assegurar-se de que a decodificação e a associação dos valores estão sendo realizadas no componente correto, ou seja, aquele em que o usuário digitou um valor. Antes de construir as marcações HTML da resposta é preciso determinar um identificador único para o componente na hierarquia que permaneça consistente também em requisições *postback*, o *clientId*. Esse objetivo é alcançado através do método *getClientId()* do componente. O método *getClientId()* percorrerá a árvore de componentes subindo em busca do primeiro componente pai que implemente a *interface NamingContainer* (ver comentário em 5.1). Assim, o identificador do componente será uma composição do identificador do componente pai encontrado e o seu próprio, por exemplo: *[NamingContainer clientId]:[id]*. Caso não tenha sido especificado um identificador para o componente na página JSP, será chamado o método *UIViewRoot.createUniqueId()* para o *framework* criar o identificador único.

Um componente implementará a *interface NamingContainer* somente se agregar outros componentes abaixo de si na hierarquia de componentes, tal como o componente de tabela JSF (`<h:table>`).

Jacobi e Fallows (2006) citam como uma boa prática a definição de identificadores nos componentes de formulário JSF (`<h:form>`) nas páginas JSP.

5.2.3.2.3 Escrita da saída para o cliente

A escrita das marcações HTML que serão enviadas para o cliente é realizada através do escritor de resposta, o *ResponseWriter*. Ele é obtido a partir do método *FacesContext.getResponseWriter()*. A classe *ResponseWriter* estende *Java.io.Writer* e implementa métodos para a geração de marcações, tal como HTML e XML. Jacobi e Fallows (2006), citam como uma boa prática o uso destes métodos ao invés de escrever todo o código HTML diretamente, pois:

- Esta abordagem provê melhor desempenho sobre o que será escrito e quando;
- Permite portabilidade entre linguagens de marcação, como HTML e XHTML;
- Possibilita melhor depuração, através da verificação do balanceamento das chamadas dos métodos *startElement()* e *endElement()*.

O código responsável pela codificação do componente *Spinner* será explorado em seguida. No método *encodeBegin()*, ilustrado na Listagem 22, faz-se uso dos métodos de escrita de recursos da classe *HtmlRenderer*, é iniciada a escrita do elemento *div* para conter os demais elementos necessários e são chamados outros métodos para a escrever o restante das marcações dos outros elementos.

```
public void encodeBegin(FacesContext context, UIComponent spinner)
throws IOException {
    //Recupera o escritor de resposta
    ResponseWriter writer = context.getResponseWriter();
    //Recupera o identificador do componente em questão
    String clientId = spinner.getClientId(context);
    //Recupera o identificador do formulário em que o componente
    está inserido
    String formId = getFormId(context, spinner);

    //Escreve o elemento inicial, o div
    writer.startElement("div", spinner);
    //Escreve o atributo title do div
    writer.writeAttribute("title", (String)
spinner.getAttributes().get("title"), null);

    //Usa o método de HtmlRenderer para escrever o recurso CSS
    writeStyleResource(context, "resources/spinnerstyle.css");

    //Chama método para escrita do elemento HTML input
    encodeInputField(spinner, writer, clientId);

    //Usa método de HtmlRenderer para escrever os recursos de
    script, um arquivo e outro inline
    writeScriptResource(context, "js/spinnervvalidation.js");
    writeScriptInline(context, "window.onload = init(" +
        String.valueOf((Integer)
spinner.getAttributes().get("minimum")) + ", " +
        String.valueOf((Integer)
spinner.getAttributes().get("maximum")) + ", \"\" +
        formId + "\", \"\" +
        clientId + "\");");

    //Chama o método para escrever o botão de decrementar valor do
    spin
    encodeDecrementButton(spinner, writer, clientId, formId);
    //Chama o método para escrever o botão de incrementar valor do
    spin
    encodeIncrementButton(spinner, writer, clientId, formId);
}
```

Listagem 22. Método de codificação do componente *Spinner*.

Como pode ser visualizado no método *encodeBegin()*, o projeto inicial teve algumas leves mas importantes modificações que visam manter as boas práticas. Em primeiro lugar, o estilo a ser aplicado às *tags* do componente *Spinner* agora são

importadas de um arquivo CSS (`resources/spinnerstyle.css`). Em seguida, é importado um arquivo JavaScript. Esse *script* realizará a importante função de evitar que toda e qualquer validação precise de uma requisição ao servidor para ser realizada. Ou seja, algumas validações simples, como valor máximo e mínimo do *spin*, poderão ser realizadas no cliente mesmo, poupando recursos e aumentando a interação com o usuário que esperará menos pela resposta. O código do *script* está na Listagem 23.

```
init = function(minValue, maxValue, formId, inputId) {
    document.forms[formId][inputId].spin = function (increment) {
        var v = parseInt(this.value) + increment;
        if (isNaN(v)) return;
        if ('min' in this && v < this.min) return;
        if ('max' in this && v > this.max) return;
        this.value = v;
    };
    document.forms[formId][inputId].min = minValue;
    document.forms[formId][inputId].max = maxValue;
};
```

Listagem 23. Script de validação da entrada de dados no cliente.

A estratégia para realizar esta validação no cliente é criar dinamicamente no elemento *input* três novos atributos: a função genérica de validação (*spin*), o valor mínimo (*min*) e o valor máximo (*max*).

Nos métodos *encodeInputField()*, *encodeDecrementButton()* e *encodeIncrementButton()* é realizada a codificação das marcações dos demais elementos HTML renderizados pelo componente *Spinner*. Com destaque para os dois últimos métodos, onde é usado o método *MessageFormat.format()* para simplificar a codificação do valor do atributo *onclick* do elemento HTML *img*. O evento *onclick* acionará a função inserida pelo *script* acima para realizar a validação no cliente do novo valor do elemento *input*. No método *encodeEnd()* é fechada a *tag div*, abaixo segue a Listagem 24 com os códigos destes métodos:

```
public void encodeEnd(FacesContext context, UIComponent spinner)
throws IOException {
    ResponseWriter writer = context.getResponseWriter();
    writer.endElement("div");
}

private void encodeInputField(UIComponent spinner, ResponseWriter
writer, String clientId) throws IOException {
    writer.startElement("input", spinner);
    writer.writeAttribute("name", clientId, "clientId");

    Object v = ((UIInput) spinner).getValue();
    if (v != null) {
        writer.writeAttribute("value", v.toString(), "value");
    }
    Integer size = (Integer) spinner.getAttributes().get("size");
    if (size != null) {
        writer.writeAttribute("size", size, "size");
    }
    writer.endElement("input");
}
```

```

    private void encodeDecrementButton(UIComponent spinner,
ResponseWriter writer, String clientId, String formId) throws
IOException {
    writer.startElement("img", spinner);
    writer.writeAttribute("name", clientId + LESS, null);
    writer.writeAttribute("class", "overlayLess", null);
    writer.writeAttribute("src", "resources/spinnerLess.gif",
null);
    writer.writeAttribute("onclick",
        MessageFormat.format(
            "document.forms['{0}']['{1}'].spin(-1);",
            new Object[] { formId, clientId }),
        null);
    writer.endElement("img");
}

    private void encodeIncrementButton(UIComponent spinner,
ResponseWriter writer, String clientId, String formId) throws
IOException {
    writer.startElement("img", spinner);
    writer.writeAttribute("name", clientId + MORE, null);
    writer.writeAttribute("class", "overlayMore", null);
    writer.writeAttribute("src", "resources/spinnerMore.gif",
null);
    writer.writeAttribute("onclick",
        MessageFormat.format(
            "document.forms['{0}']['{1}'].spin(1);",
            new Object[] { formId, clientId }),
        null);
    writer.endElement("img");
}

```

Listagem 24. Demais métodos para a codificação do Spinner.

5.2.3.2.4 Decodificando no postback

No *postback*, todas as seis fases do ciclo de vida JSF são executadas, salvo se em alguma fase o ciclo for forçado a terminar, como visto em 4.3.4. Na fase *apply request values*, o método *processDecodes()* é chamado em *UIViewRoot*, que é a raiz da hierarquia de componentes.

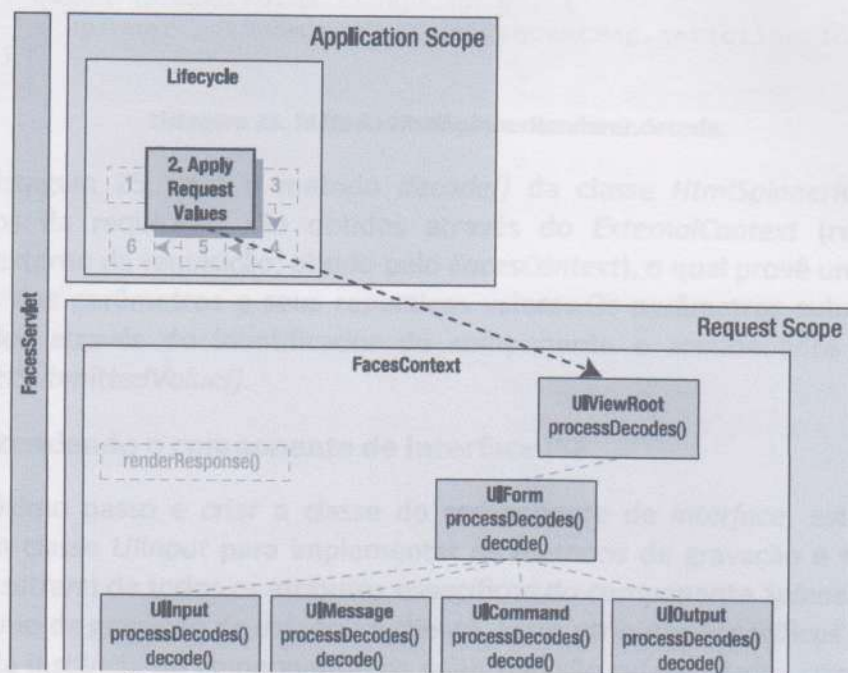


Figura 23. Decodificação na fase Apply Request Values.

Fonte: Jacobi e Fallows (2006).

O método `processDecodes()` em `UIViewRoot` é responsável por chamar recursivamente o método `processDecodes()` em cada `UIComponent` na hierarquia, como mostra a Figura 23 acima.

Se existir um renderizador associado ao `UIComponent`, a responsabilidade pela renderização será delegada para ele. No caso do componente `Spinner`, o `HtmlSpinnerRenderer`. O método `decode()` do renderizador é responsável por observar os parâmetros da requisição e atribuir os valores submetidos de acordo com o `UIComponent`. Depois que o método `processDecodes()` termina, o ciclo passa para a fase `process validations`.

```
public void decode(FacesContext context, UIComponent component) {
    EditableValueHolder spinner = (EditableValueHolder) component;
    Map<String, String> requestMap =
context.getExternalContext().getRequestParameterMap();
    String clientId = component.getClientId(context);
    int increment;
    if (requestMap.containsKey(clientId + MORE)) {
        increment = 1;
    } else if (requestMap.containsKey(clientId + LESS)) {
        increment = -1;
    } else {
        increment = 0;
    }
    try {
        int submittedValue =
Integer.parseInt(requestMap.get(clientId));
        int newValue = getIncrementedValue(component,
submittedValue, increment);
        spinner.setSubmittedValue("" + newValue);
        spinner.setValid(true);
    }
```

```

    } catch (NumberFormatException ex) {
        spinner.setSubmittedValue(requestMap.get(clientId));
    }
}

```

Listagem 25. Método `HtmlSpinnerRenderer.decode`.

No Listagem 25, está o método `decode()` da classe `HtmlSpinnerRenderer`, os parâmetros da requisição são obtidos através do `ExternalContext` (representa o contexto externo da requisição, obtido pelo `FacesContext`), o qual provê um mapa com os nomes dos parâmetros e seus respectivos valores. Os parâmetros submetidos são recuperados através do identificador do componente e armazenados através do método `setSubmittedValue()`.

5.2.4 Estendendo o componente de interface JSF

O próximo passo é criar a classe do componente de *interface*, esta classe irá estender a classe `UIInput` para implementar os métodos de gravação e recuperação (*getters* e *setters*) de todos os atributos específicos do componente `Spinner` e também o mecanismo de gravação de estado no cliente. Esses atributos específicos são aqueles próprios da instância do componente, os quais não são referenciados como atributos da *tag* que referencia o componente JSF. Se todo o estado do componente estiver mapeado em atributos da *tag* não é necessário implementar métodos *getters* e *setters*, e nem métodos para salvar e recuperar o estado do componente. Pois, neste caso, o estado destes atributos será automaticamente mantido pelo *framework* JSF, através da classe `StateManager`.

Como o `Spinner` não tem outro atributo senão aqueles referenciados na *tag*, o seu código é tão simplesmente a Listagem 26.

```

public class UISpinner extends UIInput {
    // Constantes públicas: definição do tipo do componente e
    // renderizador
    public static final String COMPONENT_TYPE =
        "com.posfarn.pro.Spinner";
    public static final String RENDERER_TYPE =
        "com.posfarn.pro.Spinner";

    public UISpinner() {
        // Atribuição de conversor padrão para o valor submetido
        setConverter(new IntegerConverter());
        // Boa prática: Atribuir no construtor o tipo do
        // renderizador do
        // componente
        setRendererType(RENDERER_TYPE);
    }
}

```

Listagem 26. Classe `UISpinner`.

Não há restrição quanto ao nome do componente e o nome do renderizador serem os mesmos.

Para o caso do componente possuir algum atributo específico da instância do componente, e que não é referenciado pela *tag*, deve-se implementar os métodos `saveState()` e `restoreState()`, da interface `StateHolder`.

Para o caso da aplicação estar usando o parâmetro `SAVING_STATE_METHOD` igual a *server*, estes métodos serão ignorados, pois o estado da instância do componente será armazenado no servidor. Porém, para o caso do componente ser levado para outra aplicação, onde este parâmetro do arquivo `faces-config.xml` esteja com o valor igual a *client*, o estado do componente no tocante a estes atributos específicos não será mantido. Logo, é também uma boa prática a implementação destes métodos.

Gerary e Horstmann (2007) comentam o uso do parâmetro `SAVING_STATE_METHOD` igual a *client* como sendo importante para manter a escalabilidade da aplicação e, para os usuários que desabilitam *cookies* no seu *browser*, a aplicação não funcionaria de outra forma. Contudo, há a desvantagem de que, a cada requisição e resposta, a informação trocada entre o cliente e servidor tem seu volume acrescido pelos dados do estado dos componentes.

5.2.5 Registrando o componente e renderizador

Depois de finalizados o componente (*UISpinner*) e o seu renderizador (*HtmlSpinnerRenderer*), é hora de registrá-los no arquivo de configuração do JSF, o `faces-config.xml`.

Quando a aplicação inicia, o JSF processa a sua configuração de acordo com os passos a seguir:

1. Em primeiro lugar, procura o arquivo `faces-config.xml` no diretório `META-INF`, dentro dos caminhos de recursos do *ClassLoader* para a aplicação web em questão;
2. Depois, checa a existência do parâmetro de inicialização `javax.faces.CONFIG_FILES`. Se o parâmetro existir, será tratado como uma lista de caminhos de recursos relativos ao *ServletContext*, todos separados por vírgula e iniciando com `"/`. Estes arquivos de configuração encontrados nestes recursos serão carregados na instância do objeto *Application* do JSF;
3. Em seguida, será procurado e carregado, caso exista, o arquivo `/WEB-INF/faces-config.xml`;
4. No final, o JSF junta todas as definições de meta-dados encontradas em todos os arquivos `faces-config.xml` encontrados.

Esta abordagem do JSF permite aos desenvolvedores a distribuição de seus componentes em arquivos *jar* e também redefinir configurações de outros componentes.

Abaixo, na Listagem 27, segue a definição do arquivo `faces-config.xml` com o registro do componente e do renderizador para o *Spinner*:

```
<component>
  <component-type>com.posfarn.pro.Spinner</component-type>
  <component-class>
com.posfarn.component.pro.UISpinner
  </component-class>
</component>

<render-kit>
```

```

<renderer>
  <component-family>javax.faces.Input</component-family>
  <renderer-type>com.posfarn.pro.Spinner</renderer-type>
  <renderer-class>
com.posfarn.render.html.basic.HtmlSpinnerRenderer
  </renderer-class>
</renderer>
</render-kit>

<managed-bean>
  <managed-bean-name>cardExpirationDate</managed-bean-name>
  <managed-bean-class>
    com.posfarn.application.CreditCardExpiration
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<application>
  <resource-bundle>
    <base-name>com.posfarn.application.messages</base-name>
    <var>msgs</var>
  </resource-bundle>
</application>

```

Listagem 27. Declarações no arquivo `faces-config.xml` para o exemplo do Spinner.

Neste trecho também está a definição do *managed bean* que irá realizar as operações específicas da aplicação e a definição de um arquivo de mensagens para uso nos textos de interface com o usuário.

5.2.6 Criando o arquivo TLD

O arquivo TLD (*Tag Library Descriptor*) é responsável por especificar o nome da classe que irá manipular as tags dos componentes e os atributos permitidos para a tag (no caso, <http://pro.posfarn.com/spinner>).

A marcação *uri*, dentro do arquivo TLD, define a identificação da classe:

```
<uri>http://pro.posfarn.com/spinner</uri>
```

Esta será a *uri* referenciada pela directiva *taglib* na página JSF para definição do prefixo do *namespace* que será usado nas tags.

```
<@ taglib uri="http://pro.posfarn.com/spinner" prefix="propos" %>
```

O JSF procura por arquivos TLD no diretório `WEB-INF`, e em seus sub-diretórios; no diretório `META-INF`; ou em qualquer arquivo `jar` dentro do diretório `WEB-INF/lib`. O arquivo TLD para o componente *Spinner* está na Listagem 28.

```

<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd">
  <tlib-version>1.1</tlib-version>
  <short-name>spinner</short-name>
  <uri>http://pro.posfarn.com/spinner</uri>
  <tag>
    <name>spinner</name>
    <tag-class>com.posfarn.taglib.pro.SpinnerTag</tag-class>
    <body-content>empty</body-content>
  </tag>

```

```

<attribute>
  <description>
    Ligação de valor para uma propriedade em um bean
  </description>
  <name>binding</name>
  <deferred-value>
    <type>javax.faces.component.UIComponent</type>
  </deferred-value>
</attribute>
<attribute>
  <description>Identificador do cliente</description>
  <name>id</name>
  <rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
  <description>Indica se será renderizado</description>
  <name>rendered</name>
  <deferred-value>
    <type>boolean</type>
  </deferred-value>
</attribute>
<attribute>
  <description>O valor mínimo do spinner</description>
  <name>minimum</name>
  <deferred-value>
    <type>int</type>
  </deferred-value>
</attribute>
<attribute>
  <description>O valor máximo do spinner</description>
  <name>maximum</name>
  <deferred-value>
    <type>int</type>
  </deferred-value>
</attribute>
<attribute>
  <description>Tamanho do elemento input</description>
  <name>size</name>
  <deferred-value>
    <type>int</type>
  </deferred-value>
</attribute>
<attribute>
  <description>O Valor do spinner</description>
  <name>value</name>
  <required>true</required>
  <deferred-value>
    <type>int</type>
  </deferred-value>
</attribute>
<attribute>
  <description>Titulo do componente</description>
  <name>title</name>
  <deferred-value>
    <type>String</type>
  </deferred-value>
</attribute>
</tag>
</taglib>

```

Listagem 28. Arquivo TLD do Spinner.

Como pode ser visualizada na Listagem 28, a maioria das definições dos atributos contém um elemento chamado *deferred-value*, o qual define que o atributo pode conter uma *String* ou expressões JSF EL. Dentro do *deferred-value* tem outra *tag* para definir qual o tipo esperado após a avaliação da expressão (*type*). Há ainda a possibilidade de criar atributos que são métodos e também serem avaliados por expressões, através da marcação *method-signature*.

No JSF 1.2 é possível usar tanto expressões do tipo JSP 2.0 (no formato “\${}”), como JSF 1.1 (no formato “#{}”).

Para o caso do identificador do componente, atributo *id*, a definição é um pouco diferente. No atributo *id* a atribuição do valor verdadeiro (*true*) para a *tag rtexprvalue* define que o valor de *id* é uma expressão avaliada em tempo de execução, ou seja, uma expressão JSP, mas não uma expressão JSF.

5.2.7 Criando a classe para manipular a tag JSP

Para cada *tag* informada no arquivo TLD é necessária uma classe para sua manipulação. Estas classes devem herdar da classe *UIComponentELTag* (para o caso de validadores, devem herdar *ValidatorELTag*, e, para conversores, *ComponenteELTag*).

A classe manipuladora da *tag* do componente no arquivo TLD tem como principais responsabilidades:

- Identificar o tipo do componente;
- Identificar o tipo do renderizador;
- Prover métodos de atribuição (*setters*) dos valores dos atributos da *tag*;
- Armazenar os atributos da *tag* no seu respectivo componente;
- Liberar recursos.

O código da classe manipuladora da *tag* do componente *Spinner* é ilustrado na Listagem 29.

```
package com.posfarn.taglib.pro;

import javax.el.MethodExpression;
import javax.el.ValueExpression;
import javax.faces.component.ActionSource;
import javax.faces.component.UIComponent;
import javax.faces.event.MethodExpressionActionListener;
import javax.faces.webapp.UIComponentELTag;

public class SpinnerTag extends UIComponentELTag {

    //Constantes públicas
    public static final String COMPONENT_TYPE =
"com.posfarn.pro.Spinner";
    public static final String RENDERER_TYPE =
"com.posfarn.pro.Spinner";

    //Atributos privados
    private ValueExpression minimum = null;
    private ValueExpression maximum = null;
    private ValueExpression size = null;
```

```

private ValueExpression value = null;
private ValueExpression title = null;
private MethodExpression valueChangeListener = null;
public String getRendererType() {
    return RENDERER_TYPE;
}
public String getComponentType() {
    return COMPONENT_TYPE;
}
public void setMinimum(ValueExpression newValue) {
    minimum = newValue;
}
public void setMaximum(ValueExpression newValue) {
    maximum = newValue;
}
public void setSize(ValueExpression newValue) {
    size = newValue;
}
public void setValue(ValueExpression newValue) {
    value = newValue;
}
public void setTitle(ValueExpression newTitle) {
    title = newTitle;
}
public void setValueChangeListener(MethodExpression
valueChangeListener) {
    this.valueChangeListener = valueChangeListener;
}
public void setProperties(UIComponent component) {
    // always call the superclass method
    super.setProperties(component);
    component.setValueExpression("size", size);
    component.setValueExpression("minimum", minimum);
    component.setValueExpression("maximum", maximum);
    component.setValueExpression("value", value);
    component.setValueExpression("title", title);
    if (valueChangeListener != null)
        ((ActionSource) component).addActionListener(
            new
MethodExpressionActionListener(valueChangeListener));
}
public void release() {
    super.release();
    minimum = null;
    maximum = null;
    size = null;
    value = null;
    title = null;
}
}

```

Listagem 29. Classe manipuladora da tag do componente Spinner.

Cada atributo da *tag* tem um respectivo atributo de instância na classe *SpinnerTag*, com exceção dos atributos *id*, *binding* e *rendered* que são controlados pela sua super classe, a classe *UIComponentELTag*.

O tipo do componente é definido como *com.posfarn.pro.Spinner* e o tipo do renderizador tem também este mesmo nome, como dito anteriormente, não há problema em usar nomes iguais para o componente e seu renderizador. Caso o

renderizador fosse definido como *null*, o componente teria a responsabilidade de renderizar-se.

Quando a *tag* é processada na página JSF, cada valor de atributo da *tag* é convertido para um objeto do tipo *ValueExpression* e então o respectivo método de atribuição (*setter*) é invocado em *SpinnerTag*. Métodos de recuperação (*getters*) não são necessários. Para copiar os valores destes atributos para o componente, a classe deve sobrescrever o método *setProperties()*.

O *framework* JSF pode fazer cache destes objetos e reusá-los, portanto, é necessário sobrescrever também o método *release()* para retornar os valores padrões dos atributos.

Os métodos *setProperties()* e *release()* devem sempre chamar em primeiro lugar a super-classe.

6.1.1 Primeira estratégia: usando um Phase Listener

A Figura 24 ilustra a arquitetura desta estratégia, onde componentes JSF customizados renderizam scripts AJAX no cliente e processam requisições AJAX.

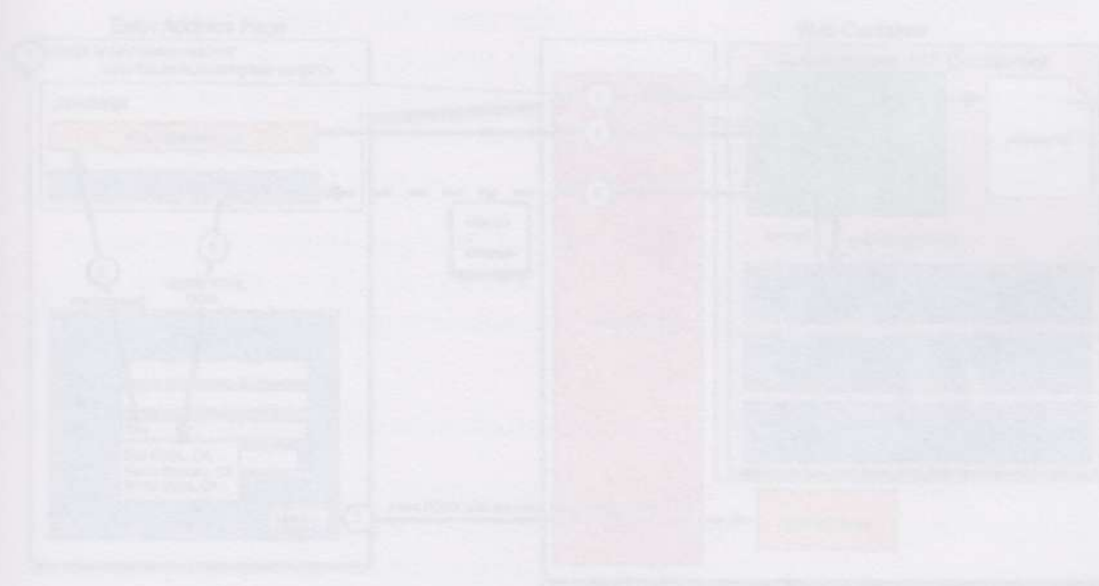


Figura 24. Primeira estratégia JSP AJAX.
Fonte: Murray, Murphy e Bondi (2007).

Neste exemplo é usado um componente que tem a função de auto completar o texto que está sendo digitado pelo usuário, o componente *AutoComplete*, denotado pela *tag completionField*. Este componente foi inserido na página usando:

```
<h:input type="text" value="#{user.name}" style="width: 100%; border: 1px solid #ccc;"/>  
<h:completionField value="#{user.name}" style="width: 100%; border: 1px solid #ccc;"/>  
</h:form>
```

Os passos são detalhados a seguir:

6 JSF + AJAX

Este capítulo aborda como encapsular as nuances das técnicas AJAX em componentes JSF, abstraindo para o desenvolvedor da aplicação Web a problemática envolvida em adicionar características de aplicações RIA ao seu projeto.

As estratégias para adicionar AJAX ao JSF serão abordadas em primeiro plano para possibilitar uma visão mais ampla. Em seguida, será construído um exemplo de um componente JSF AJAX usando a estratégia mais difundida atualmente.

Para facilitar e abstrair os detalhes que precisam ser tratados, como discutido no capítulo 4, será usado um *framework* JavaScript na criação do novo componente.

6.1 ESTRATÉGIAS PARA INSERIR AJAX NO JSF

A seguir, são explanadas as estratégias para o desenvolvimento de aplicações JSF com AJAX, baseado nas técnicas mais aceitas atualmente e referenciadas por vários autores como Olson (2007), Jacobi e Fallows (2006), Gerary e Horstmann (2007). E em vários artigos do catálogo de boas práticas da Sun Microsystems.

6.1.1 Primeira estratégia: usando um Phase Listener

A Figura 24 ilustra a arquitetura desta estratégia, onde componentes JSF customizados renderizam *scripts* AJAX no cliente e processam requisições AJAX.

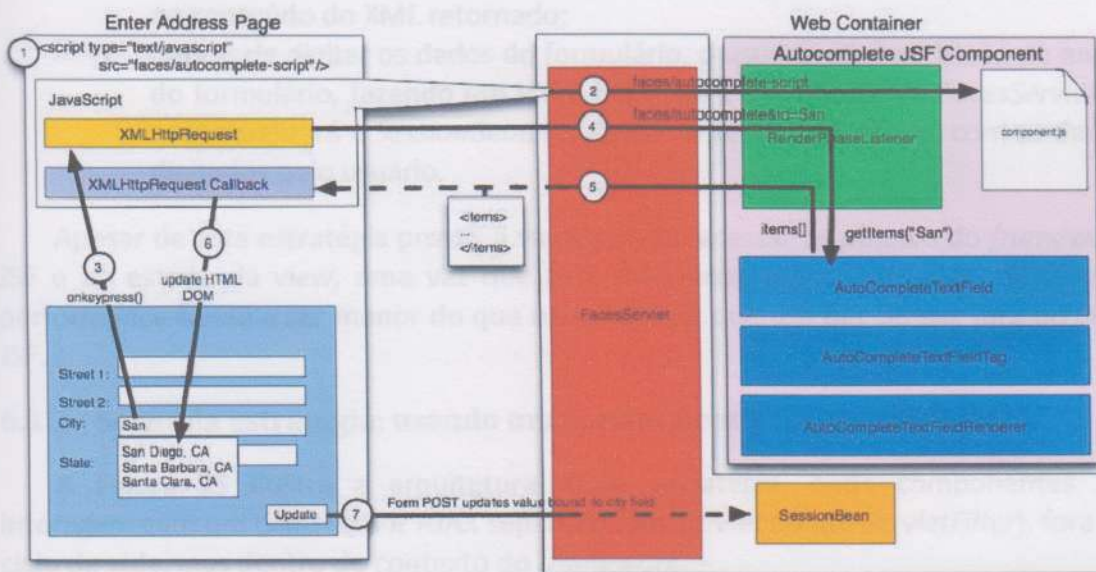


Figura 24. Primeira estratégia JSF AJAX.
Fonte: Murray, Norbye e Burns (2007).

Neste exemplo é usado um componente que tem a função de auto completar o texto que está sendo digitado pelo usuário, o componente *AutoComplete*, denotado pela tag *completionField*. Este componente foi inserido na página usando:

```
<ajaxTags:completionField size="40" id="cityField"
completionMethod="#{AutoCompleteTextfield.completeCity}"
value="#{SessionBean.city}" required="true"
/>
```

Os passos são detalhados a seguir:

1. A página "Enter Address Page" contém um elemento HTML, renderizado por *AutoCompleteTextRenderer*, que insere o arquivo JavaScript, o qual realizará a chamada AJAX;
2. Uma chamada é feita para a URL, *faces/autocomplete-script*, a qual é mapeada para uma instância do *FacesServlet*. Esta instância processa a instância de *RenderPhaseListener*, o qual reconhece a URI e retorna o arquivo com o código JavaScript e em seguida termina sua participação no ciclo de vida JSF;
3. Quando o usuário começa a teclar no campo do nome da cidade (*City*), um evento *onkeypress* ocorre. A função JavaScript associada a este evento cria um objeto *XMLHttpRequest* e o configura com a URL da instância do *FacesServlet* (*faces/autocomplete&id=San*), a qual é requisitada enquanto a página continua a ser processada;
4. A instância do *FacesServlet* processa a instância de *RenderPhaseListener* que reconhece a URL *faces/autocomplete* e procura pelo componente *AutoCompleteTextField* que proverá a lógica de negócio (o auto completar);
5. A instância de *RenderPhaseListener* gera um documento XML contendo as cidades que combinam com o texto digitado pelo usuário e retorna-o para o objeto *XMLHttpRequest*, o qual chamará sua função de retorno (*callback*);
6. A função de retorno do *XMLHttpRequest* atualiza o HTML DOM baseado no conteúdo do XML retornado;
7. Depois de digitar os dados do formulário, o usuário clica no botão de envio do formulário, fazendo um HTTP POST para a instância do *FacesServlet*, o qual atualizará o *SessionBean* (o objeto de sessão associado) com os dados digitados pelo usuário.

Apesar de esta estratégia prover a vantagem de acessar os objetos do *framework* JSF e ao estado da *view*, uma vez que está mais acoplado ao seu ciclo de vida, a performance tende a ser menor do que na estratégia que usa um *Servlet* fora do ciclo JSF.

6.1.2 Segunda estratégia: usando um Servlet dentro do contexto JSF

A Figura 25 ilustra a arquitetura desta estratégia, onde componentes JSF interagem com um controlador AJAX separado (um *Servlet* ou um *ServletFilter*), fora do ciclo de vida, mas dentro do contexto do *framework*.

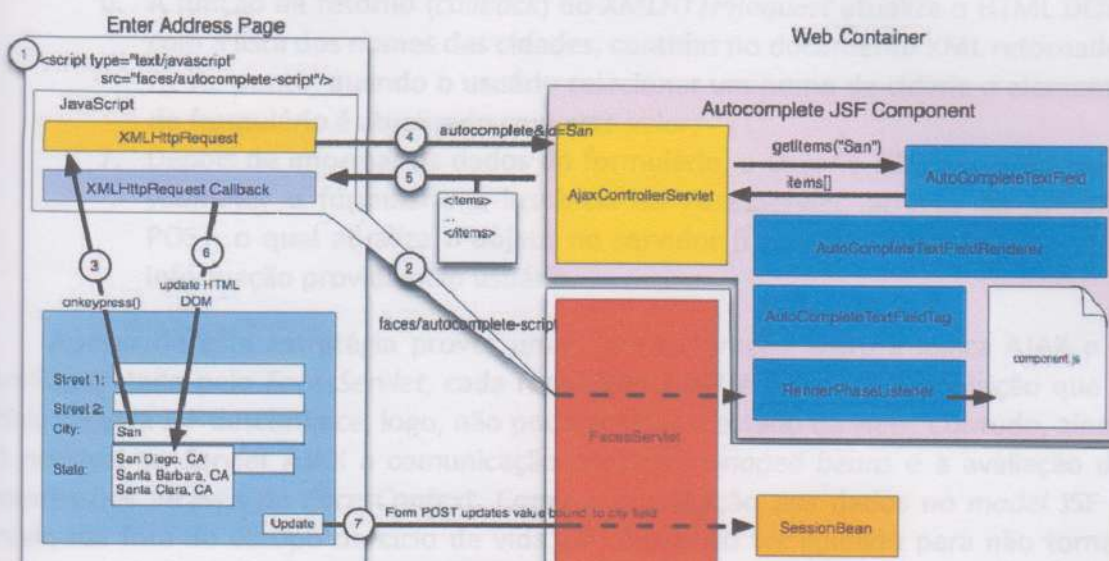


Figura 25. Segunda estratégia JSF AJAX.
Fonte: Murray, Norbye e Burns (2007).

Nesta estratégia o JavaScript renderizado pelo componente *JavaServer Faces* se comunica com um *Servlet* específico. Todas as requisições assíncronas serão tratadas fora do processamento do ciclo de vida do *framework* JSF. Contudo, o referido *Servlet* poderá acessar a instância do *FacesContext* e avaliar expressões JSF EL para realizar tarefas como encontrar *managed beans* e realizar chamadas a eles.

As interações destacadas nos passos da figura acima são descritos com segue:

1. A página "Enter Address Page" contém um elemento HTML, renderizado por *AutoCompleteTextRenderer*, que insere o arquivo JavaScript, o qual realizará a chamada AJAX;
2. Uma chamada é feita para a URL, *faces/autocomplete-script*, a qual é mapeada para uma instância do *FacesServlet*. Esta instância processa a instância de *RenderPhaseListener*, o qual reconhece a URI e retorna o arquivo com o código JavaScript e em seguida termina sua participação no ciclo de vida JSF;
3. Quando o usuário começa a teclar no campo do nome da cidade (*City*), um evento *onkeypress* ocorre. A função JavaScript associada a este evento cria um objeto *XMLHttpRequest* e o configura com a URL da instância do *AjaxControllerServlet* (*faces/autocomplete&id=San*), a qual é requisitada enquanto a página continua a ser processada;
4. O objeto *XMLHttpRequest* realiza a chamada para a instância de *AjaxControllerServlet* enquanto os eventos da página HTML continuam sendo processados no cliente. A instância de *AjaxControllerServlet* recupera a partir do componente *AutoCompleteTextField* a lista dos itens que combinam com a entrada do usuário submetida;
5. A instância do *AjaxControllerServlet* gera o documento XML contendo a lista obtida de *AutoCompleteTextField* e a retorna ao objeto *XMLHttpRequest*, o qual realiza a chamada à sua função de retorno (função *callback*);

6. A função de retorno (*callback*) do *XMLHttpRequest* atualiza o HTML DOM com a lista dos nomes das cidades, contidas no documento XML retornado. Neste ponto, quando o usuário selecionar um nome de cidade o elemento do formulário é atualizado com este valor;
7. Depois de informar os dados do formulário, o usuário clica no botão para submeter o formulário à instância do *FacesServlet*, através do método POST, o qual atualiza o objeto no servidor (no caso, *SessionBean*) com a informação provida pelo usuário.

Apesar de esta estratégia prover uma clara separação entre a lógica AJAX e o tráfego tratado pelo *FacesServlet*, cada requisição AJAX é uma nova requisição que o ciclo de vida JSF desconhece, logo, não pode acessar o estado da *view*. Contudo, ainda é possível ao *Servlet* AJAX a comunicação com os *managed beans* e a avaliação de expressões através do *FacesContext*. Como a atualização dos dados no *model* JSF é realizada fora do escopo do ciclo de vida JSF, é preciso ter cuidado para não tornar este modelo de dados inconsistente.

6.1.3 Terceira estratégia: usando um Servlet fora do ciclo JSF

A Figura 26 ilustra a arquitetura desta estratégia, onde é usado um controlador AJAX (um *Servlet* ou um *ServletFilter*) completamente fora do ciclo de vida e do contexto JSF.

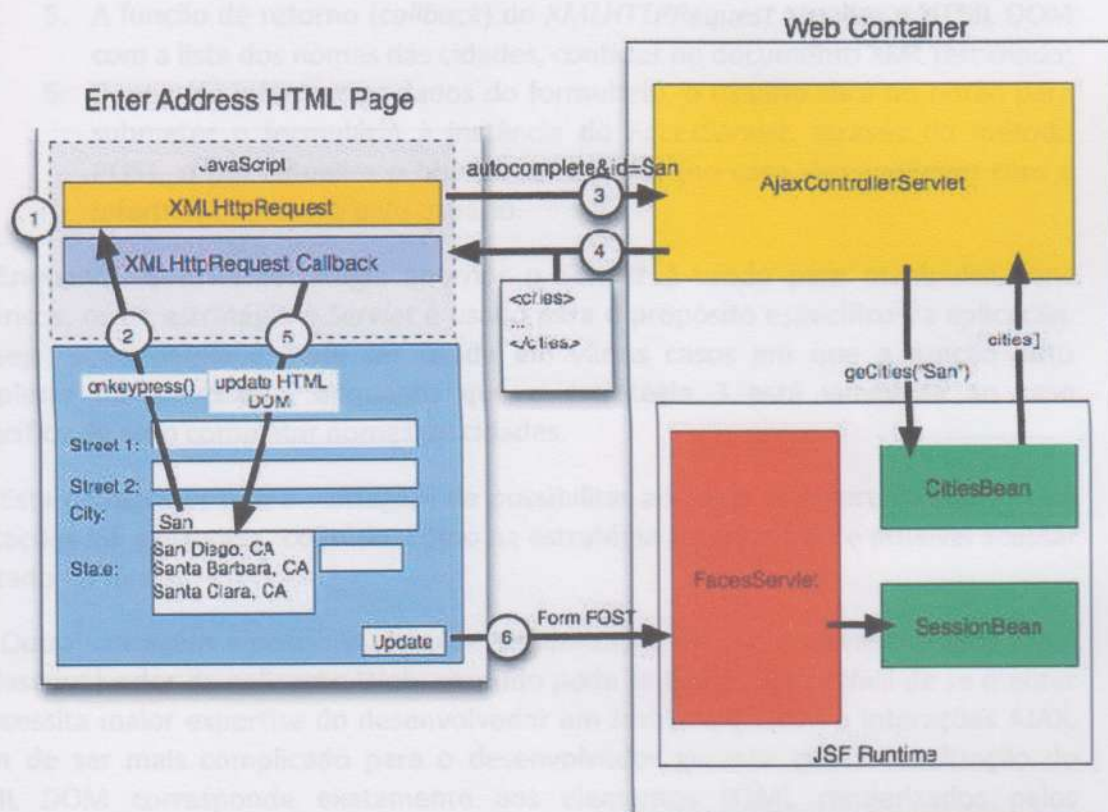


Figura 26. Terceira estratégia JSF AJAX.

Fonte: Murray, Norbye e Burns (2007).

Nesta estratégia nenhum componente JSF customizado é usado. Um *Servlet* completamente separado do JSF é usado para tratar as requisições AJAX. Todo o

código JavaScript necessário para a comunicação com este *Servlet* é adicionado à página JSP diretamente pelo desenvolvedor da página. O desenvolvedor deverá prover todo o código que irá tratar os eventos JavaScript associados aos componentes JSF, realizando as chamadas assíncronas e atualizando o documento HTML quando a resposta for recebida.

Os passos na figura acima relativos à arquitetura desta estratégia são explanados abaixo:

1. A página "Enter Address Page" contém o código JavaScript necessário para as interações AJAX;
2. Quando o usuário digita no campo do nome da cidade (*City*) um evento *onkeypress* é gerado. A função JavaScript associada a este evento cria um objeto *XMLHttpRequest* e o configura com uma URL para o *AjaxControllerServlet* (*faces/autocomplete&id=San*), a qual é requisitada enquanto a página continua a ser processada;
3. O *AjaxControllerServlet* recupera a partir do *managed bean* chamado *CitiesBean* a lista dos itens que combinam com a entrada do usuário submetida;
4. A instância do *AjaxControllerServlet* gera o documento XML contendo a lista obtida de *CitiesBean* e a retorna ao objeto *XMLHttpRequest*, o qual realiza a chamada à sua função de retorno (função *callback*);
5. A função de retorno (*callback*) do *XMLHttpRequest* atualiza o HTML DOM com a lista dos nomes das cidades, contidas no documento XML retornado;
6. Depois de informar os dados do formulário, o usuário clica no botão para submeter o formulário à instância do *FacesServlet*, através do método POST, o qual atualiza o objeto no servidor (no caso, *SessionBean*) com a informação provida pelo usuário.

Enquanto que na estratégia anterior o *Servlet* é usado para manipular itens genéricos, nesta estratégia o *Servlet* é usado para o propósito específico da aplicação. Ou seja, a estratégia 2 pode ser usada em vários casos em que a função auto completar for necessária, enquanto que a estratégia 3 está vinculada ao caso específico de auto completar nomes de cidades.

Esta estratégia tem a vantagem de possibilitar adicionar comportamento AJAX à aplicações JSF existentes, contudo, como na estratégia anterior, não é possível acessar o estado da *view*.

Outra vantagem é possibilitar maior customização do código JavaScript por parte do desenvolvedor da aplicação Web, contudo pode se tornar mais difícil de se manter e necessita maior expertise do desenvolvedor em JavaScript, DOM e interações AJAX. Além de ser mais complicado para o desenvolvedor garantir que a atualização do HTML DOM corresponde exatamente aos elementos HTML renderizados pelos componentes JSF da página.

Da mesma forma que na estratégia anterior, como a atualização dos dados no model JSF é realizada fora do escopo do ciclo de vida JSF, é preciso ter cuidado para não tornar este modelo de dados inconsistente.

6.1.4 Comparação entre as estratégias

Abaixo segue um quadro, o qual resume as características principais das estratégias abordadas para incluir a funcionalidade AJAX à aplicações JSF:

	Estratégia 1	Estratégia 2	Estratégia 3
Onde fica a lógica do controle?	Em uma instância do <code>PhaseListener</code> .	Em um <code>Servlet</code> ou <code>ServletFilter</code> .	Em um <code>Servlet</code> ou <code>ServletFilter</code> .
O desenvolvedor precisa prover código JavaScript AJAX no cliente e mapear elementos do formulário?	Não. A renderização do componente JSF proverá o código JavaScript necessário.		Sim. Inclusive, deve-se ter o cuidado de garantir que os elementos mapeados correspondam aos renderizados pelo componente JSF.
É uma solução reusável?	Sim. Contudo, o JavaScript incluído com o componente JSF deve ser escrito de forma a ser capaz de processar requisições de vários componentes. Há um número limitado de requisições <code>XMLHttpRequest</code> por página.		Não. Cada página JSF deve ter o seu próprio JavaScript. Mesmo compartilhando um arquivo, geralmente, a função callback se vincula a um caso específico.
É uma solução customizável?	Sim. A customização é provida pelo componente.		Sim. Porém, isto se torna um problema, pois cada página JSF tem uma solução customizada.
O componente tem acesso ao estado da view do JSF?	Sim. Porém, há diferentes formas de acessar a view, de acordo com a versão do JSF.	Não.	Não.
A lógica de controle AJAX do componente JSF pode ser sobrescrita?	Sim. A lógica é mantida na instância do <code>PhaseListener</code> , que pode ser alterado.	Sim. O controlador é uma <code>Servlet</code> que pode ser alterada ou substituída.	
A lógica do	Sim. Cada	Sim. O código da lógica do controle	

controle AJAX pode acessar managed beans e avaliar expressões?	requisição é uma requisição JSF.	AJAX pode acessar managed beans e avaliar expressões através do FacesContext, mas não é simples.
É possível usar bibliotecas JavaScript existentes?	Sim. Tanto o código JavaScript pode ser encapsulado dentro do componente JSF como pode ser obtido de um JavaScript externo.	Sim.
A lógica do controle AJAX pode ficar fora da execução do JSF?	Não.	Sim. Servlets e ServletFilters não requerem o ambiente de execução JSF. Contudo, não é possível ter a lógica de controle do AJAX associada à managed beans ou avaliação de expressões JSF.

Fonte: Murray, Norbye e Burns (2007).

6.2 DEFINIÇÃO DO COMPONENTE JSF

O componente JSF que será desenvolvido para o uso do Ajax será bem simples, porém fará o uso básico de toda a facilidade provida pelo JSF para construir um componente prático e configurável para uso do desenvolvedor da aplicação Web.

O objetivo deste componente será estender o componente JSF *UIOutput* para adicionar-lhe um novo comportamento. A idéia é agregar a funcionalidade de um *link* para exibição dinâmica de uma janela flutuante com detalhes relativos à informação que está sendo mostrada.

6.2.1 Criando o protótipo da interface

O protótipo da *interface* será construído com base na visualização estática do componente. O *layout* desejado para a janela de exibição dos detalhes da informação principal é dado a seguir:

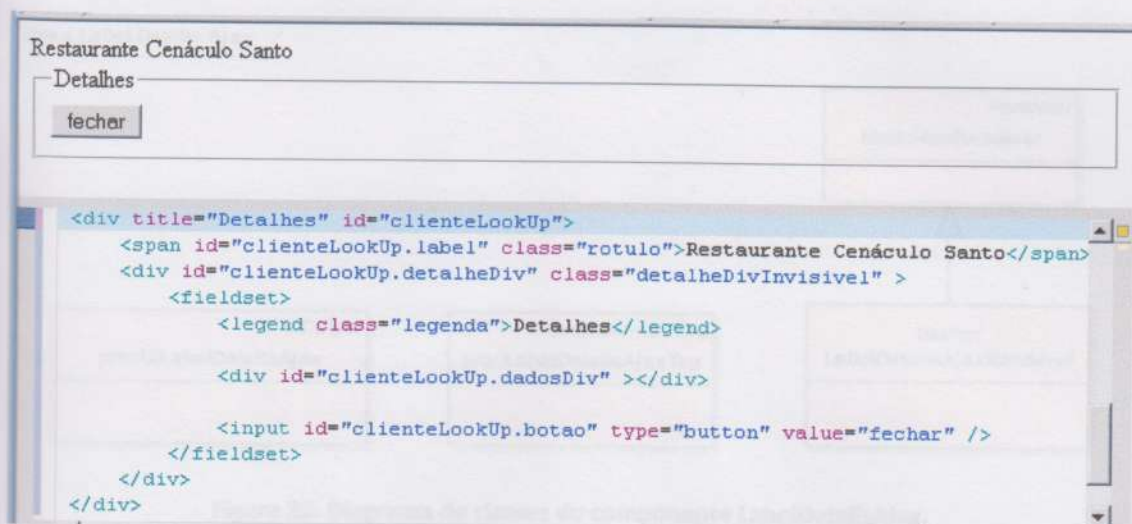


Figura 27. Protótipo do componente LabelDetailsAjax.

Na Figura 27 acima, está o primeiro esboço da *interface* pretendida, e logo abaixo o código usado para sua geração. E, já neste código, é possível destacar os primeiros parâmetros que serão necessários, como ilustra a Listagem 30.

```

<div title=[título] id=[id]>
  <span id="[id].label" class="rotulo">[valor_convertido]</span>
  <div id="[id].detalheDiv" class="detalheDivInvisivel" >
    <fieldset>
      <legend class="legenda">Detalhes</legend>

      <div id="[id].dadosDiv" ></div>

      <input id="[id].botao" type="button" value="fechar"/>
    </fieldset>
  </div>
</div>

```

Listagem 30. Protótipo parametrizado do componente LabelDetailsAjax.

A partir dos parâmetros destacados entre colchetes na Listagem 30, a definição do uso do novo componente JSF dentro da página JSP, substituindo as marcações HTML, será de acordo com o a Listagem 31.

```

<propos:labelDetailsAjax
  id="clienteLookUp"
  value="Restaurante Cenáculo Santo"
  title="Detalhes"
  targetId="100"
  entity="Cliente"/>

```

Listagem 31. Definição da declaração do componente LabelDetailsAjax.

Os dois últimos parâmetros, assim como o elemento *div* interno – *dadosDiv*, serão usados no comportamento dinâmico do componente, portanto, serão detalhados apenas no tópico 6.3 mais adiante.

6.2.2 Design das classes necessárias

No diagrama de classes simplificado abaixo, pela Figura 28, são destacadas as classes que são necessárias.

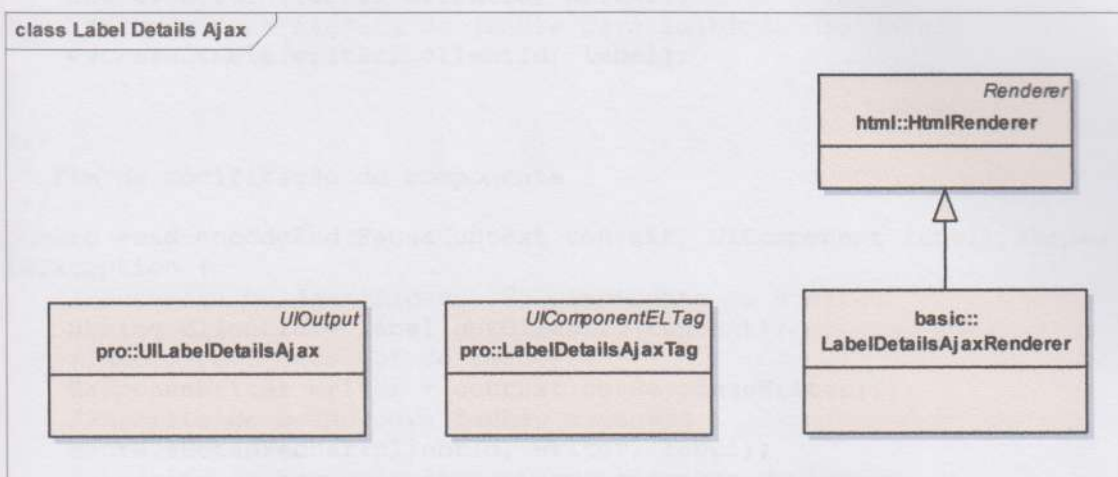


Figura 28. Diagrama de classes do componente LabelDetailsAjax.

A classe *UILabelDetailsAjax* estende a classe *UIOutput* do JSF, herdando o comportamento de um componente simples de entrada de dados, que pode guardar seu estado no cliente ou no servidor.

A classe *LabelDetailsAjaxTag* estende a classe *UIComponentELTag* do JSF 1.2, nela implementar-se-á o tratamento dos parâmetros da tag JSP do novo componente JSF. Será referenciada pelo arquivo *labeldetailsajax.tld* que será desenvolvido a seguir.

Tanto a classe *UILabelDetailsAjax* como a *LabelDetailsAjaxTag* estão definidas dentro do sub-pacote *pro*, mas dentro de pacotes diferentes: *com.posfarn.component.pro.UILabelDetailsAjax* e *com.posfarn.taglib.pro.LabelDetailsAjaxTag*.

Assim como no capítulo 5, será feito uso da classe *HtmlRenderer* para ajudar nas tarefas comuns. Portanto, a classe *LabelDetailsAjaxRenderer* estende *HtmlRenderer* para prover a renderização dos elementos HTML necessários.

6.2.3 Criando o renderizador específico para o cliente Web

O código do renderizador específico é trivial, considerando tudo o que já foi explorado nos capítulos anteriores. Ele tem a seu cargo tão simplesmente a escrita dos elementos HTML necessários para a funcionalidade básica do componente. O código dos seus métodos é dado na Listagem 32.

```

/**
 * Início da codificação do componente
 */
public void encodeBegin(FacesContext context, UIComponent label)
throws IOException {
    //Recupera o escritor de resposta
    ResponseWriter writer = context.getResponseWriter();
    //Recupera o identificador do componente em questão
    String clientId = label.getClientId(context);
    //Escreve o elemento inicial, o div
    writer.startElement("div", label);
    //Escreve o atributo title do div
    writer.writeAttribute("title", (String)
label.getAttributes().get("title"), null);
    //Método para escrita do valor enviado
  
```

```

    escreveRotulo(label, clientId, writer);
    //Metodo para escrita da janela para exibição dos detalhes
    escreveJanela(writer, clientId, label);
}

/**
 * Fim da codificação do componente
 */
public void encodeEnd(FacesContext context, UIComponent label) throws
IOException {
    //Recupera o identificador do componente em questão
    String clientId = label.getClientId(context);
    //Recupera o escritor de resposta
    ResponseWriter writer = context.getResponseWriter();
    //Escrita do botao para fechar a janela
    escreveBotaoFechar(clientId, writer, label);
    //Escrita da tag de fechamento do elemento de layout
    writer.endElement("fieldset");
    //Escrita do fechamento do div interno
    writer.endElement("div");
    //Escrita do fechamento do div externo
    writer.endElement("div");
}

/**
 * Método de decodificação
 */
public void decode(FacesContext context, UIComponent component) {
    ValueHolder label = (ValueHolder) component;
    Map<String, String> requestMap =
context.getExternalContext().getRequestParameterMap();
    String clientId = component.getClientId(context);

    try {
        String submittedValue = requestMap.get(clientId);
        label.setValue(" " + submittedValue);
    } catch (NumberFormatException ex) {
        label.setValue(requestMap.get(clientId));
    }
}

//Escreve botão para fechar janela
private void escreveBotaoFechar(String clientId, ResponseWriter
writer, UIComponent label) throws IOException {
    writer.startElement("input", label);
    writer.writeAttribute("id", clientId + ".botao", null);
    writer.writeAttribute("type", "button", null);
    writer.writeAttribute("value", "fechar", null);
    writer.endElement("input");
}

//Escreve janela e seu layout
private void escreveJanela(ResponseWriter writer, String clientId,
UIComponent label) throws IOException {
    writer.startElement("div", label);
    writer.writeAttribute("id", clientId + ".detalheDiv", null);
    writer.writeAttribute("class", "detalheDivInvisivel", null);
    //Escreve elemento de layout dentro da janela
    writer.startElement("fieldset", label);
    //Escreve legenda da janela
    writer.startElement("legend", label);

```

```

writer.writeAttribute("class", "legenda", null);
writer.writeText("Detalhes", null);
writer.endElement("legend");
//Escreve div para manipulação por Ajax
writer.startElement("div", label);
writer.writeAttribute("id", clientId + ".dadosDiv", null);
writer.endElement("div");
}

//Escreve o valor enviado
private void escreveRotulo(UIComponent label, String clientId,
ResponseWriter writer) throws IOException {
    writer.startElement("span", label);
    writer.writeAttribute("id", clientId + ".label", null);
    writer.writeAttribute("class", "rotulo", null);
    writer.writeText((java.lang.String)
label.getAttributes().get("value"), null);
    writer.endElement("span");
}

```

Listagem 32. Métodos para a codificação do componente `LabelDetailsAjax`.

6.2.4 Estendendo o componente de interface JSF

O código necessário para implementação da classe `UILabelDetailsAjax` também não tem nenhum detalhe especial. A exemplo do componente discutido no capítulo 5, não será necessário implementar os métodos para guardar e recuperar estado. Como não são realizadas entradas de dados, o componente a ser estendido é o `UIOutput`. Desta forma, o código é simplesmente tal qual a Listagem 33.

```

public class UILabelDetailsAjax extends UIOutput {

    // Constantes públicas
    public static final String COMPONENT_TYPE =
"com.posfarn.pro.LabelDetailsAjax";
    public static final String RENDERER_TYPE =
"com.posfarn.pro.LabelDetailsAjax";

    public UILabelDetailsAjax() {
        // Boa prática: Atribuir no construtor o tipo do
renderizador do
        // componente
        setRendererType(RENDERER_TYPE);
    }
}

```

Listagem 33. Classe `UILabelDetailsAjax`.

6.2.5 Registrando o componente e renderizador

Depois de finalizados o componente (`UILabelDetailsAjax`) e o seu renderizador (`LabelDetailsAjaxRenderer`), é hora de registrá-los no arquivo de configuração do JSF, o `faces-config.xml`.

Na Listagem 34, segue a definição do arquivo `faces-config.xml` com o registro do componente e do renderizador para o `LabelDetailsAjax`:

```

...

<component>
    <description>LabelDetailsAjax</description>

```

```

    <component-type>com.posfarn.pro.LabelDetailsAjax</component-type>
    <component-
class>com.posfarn.component.pro.UILabelDetailsAjax</component-class>
</component>

<render-kit>
    <renderer>
        <component-family>javax.faces.Output</component-family>
        <renderer-type>com.posfarn.pro.LabelDetailsAjax</renderer-type>
        <renderer-
class>com.posfarn.render.html.basic.LabelDetailsAjaxRenderer</renderer-
class>
    </renderer>
</render-kit>

...

```

Listagem 34. Declarações no arquivo faces-config.xml para o componente LabelDetailsAjax.

6.2.6 Criando o arquivo TLD

A marcação *uri*, dentro do arquivo TLD, que define a identificação da classe manipuladora das *tags* dos componentes e dos atributos permitidos é dada a seguir:

```
<uri>http://pro.posfarn.com/labelDetailsAjax</uri>
```

Esta será a *uri* referenciada pela directiva *taglib* na página JSF para definição do prefixo do *namespace* que será usado nas *tags*.

```
<%@ taglib uri="http://pro.posfarn.com/labelDetailsAjax"
prefix="propos" %>
```

O arquivo TLD para o componente *LabelDetailsAjax* é ilustrado na Listagem 35.

```

<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd">
    <tlib-version>1.1</tlib-version>
    <short-name>labelDetailsAjax</short-name>
    <uri>http://pro.posfarn.com/labelDetailsAjax</uri>
    <tag>
        <name>labelDetailsAjax</name>
        <tag-class>com.posfarn.taglib.pro.LabelDetailsAjaxTag</tag-
class>
        <body-content>empty</body-content>
        <attribute>
            <description>
                Ligação de valor para uma propriedade em um bean
            </description>
            <name>binding</name>
            <deferred-value>
                <type>javax.faces.component.UIComponent</type>
            </deferred-value>
        </attribute>
        <attribute>
            <description> Identificador do cliente </description>
            <name>id</name>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
        <attribute>

```

```

        <description> Indica se será renderizado </description>
        <name>rendered</name>
        <deferred-value>
            <type>boolean</type>
        </deferred-value>
    </attribute>
    <attribute>
        <description>O texto a ser exibido</description>
        <name>value</name>
        <required>true</required>
        <deferred-value>
            <type>String</type>
        </deferred-value>
    </attribute>
    <attribute>
        <description>Titulo do componente</description>
        <name>title</name>
        <deferred-value>
            <type>String</type>
        </deferred-value>
    </attribute>
    <attribute>
        <description>Id do objeto que vai ter seus detalhes
exibidos</description>
        <name>targetId</name>
        <deferred-value>
            <type>String</type>
        </deferred-value>
    </attribute>
    <attribute>
        <description>Entidade a que pertence o
objeto</description>
        <name>entity</name>
        <deferred-value>
            <type>String</type>
        </deferred-value>
    </attribute>
</tag>
</taglib>

```

Listagem 35. Arquivo TLD do componente LabelDetailsAjax.

6.2.7 Criando a classe para manipular a tag JSP

O código da classe manipuladora da *tag* do componente *LabelDetailsAjax* demonstrado na Listagem 36.

```

public class LabelDetailsAjaxTag extends UIComponentELTag {

    //Constantes públicas
    public static final String COMPONENT_TYPE =
"com.posfarn.pro.LabelDetailsAjax";
    public static final String RENDERER_TYPE =
"com.posfarn.pro.LabelDetailsAjax";

    //Atributos privados
    private ValueExpression value = null;
    private ValueExpression title = null;
    private ValueExpression targetId = null;
    private ValueExpression entity = null;

    public String getRendererType() {

```

```

    return RENDERER_TYPE;
}

public String getComponentType() {
    return COMPONENT_TYPE;
}

public void setValue(ValueExpression newValue) {
    value = newValue;
}

public void setTitle(ValueExpression newTitle) {
    title = newTitle;
}

public void setTargetId(ValueExpression newTargetId) {
    targetId = newTargetId;
}

public void setEntity(ValueExpression newEntity) {
    entity = newEntity;
}

public void setProperties(UIComponent component) {
    // always call the superclass method
    super.setProperties(component);

    component.setValueExpression("value", value);
    component.setValueExpression("title", title);
    component.setValueExpression("targetId", targetId);
    component.setValueExpression("entity", entity);
}

public void release() {
    // always call the superclass method
    super.release();

    value = null;
    title = null;
    targetId = null;
    entity = null;
}
}

```

Listagem 36. Classe LabelDetailsAjaxTag.

A exemplo do componente *Spinner* do capítulo 5, neste componente também não há para os atributos *id*, *binding* e *rendered* um respectivo atributo na classe *LabelDetailsAjaxTag*, pois, como dito anteriormente, estes são controlados pela super-classe *UIComponentELTag*.

O tipo do componente é definido como "*com.posfarn.pro.LabelDetailsAjax*" e o tipo do renderizador tem também este mesmo nome.

6.3 IMPLEMENTANDO NO COMPONENTE JSF O FRAMEWORK AJAX JAVASCRIPT

Para implementar o comportamento dinâmico e assíncrono no componente *LabelDetailsAjax* será necessário o JavaScript. Para encapsular as diferenças entre os *browsers* e os detalhes do mecanismo AJAX, será usado o *framework* descrito no capítulo 3. Toda a interação é demonstrada no diagrama de sequência da Figura 29.

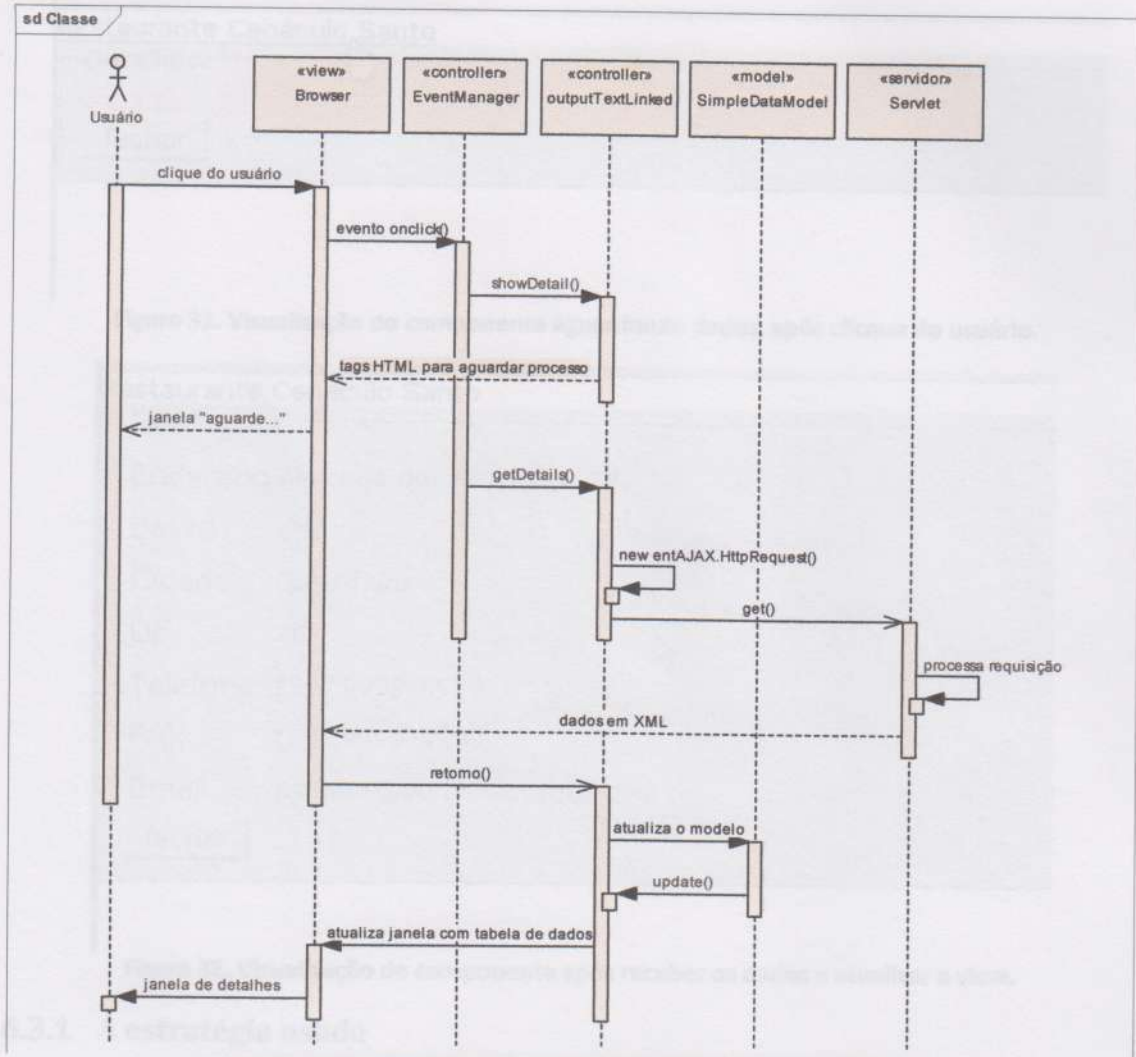


Figura 29. Diagrama de sequência do comportamento AJAX.

A Figura 29 ilustra o diagrama de sequência que reflete as interações entre cada elemento relevante da arquitetura montada.

Abaixo seguem as Figuras 30, 31 e 32 que exibem os estados mais relevantes da visualização do componente em uma página: inicialmente com a janela fechada, com a janela aberta aguardando os dados solicitados e depois com a janela sendo exibida já atualizada por AJAX.

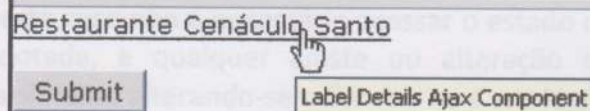


Figura 30. Visualização do estado inicial do componente.

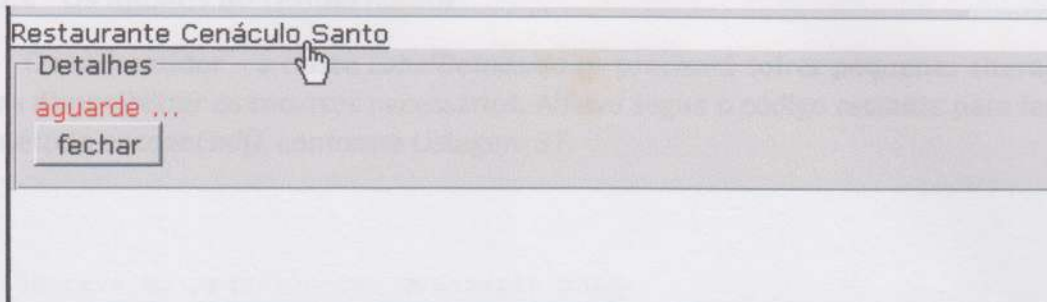


Figura 31. Visualização do componente aguardando dados, após clique do usuário.

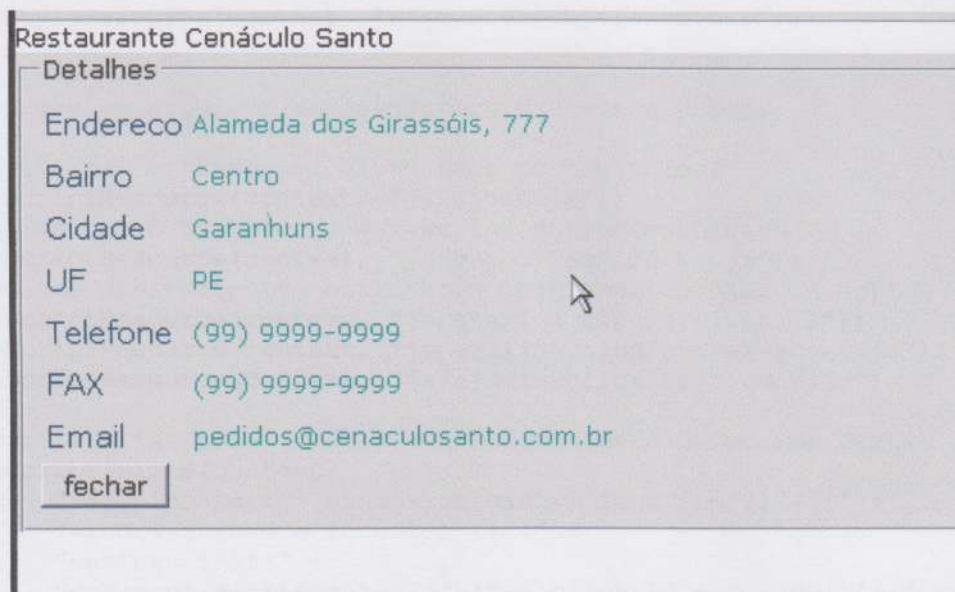


Figura 32. Visualização do componente após receber os dados e atualizar a view.

6.3.1 A estratégia usada

A estratégia para inserir o comportamento AJAX no JSF que será usada é a segunda abordagem descrita anteriormente, ou seja, usando um componente específico acionando um *Servlet* para tratamento das requisições AJAX.

Todo o código JavaScript usado na execução do componente será provido por ele mesmo, não sendo necessário que o desenvolvedor da aplicação precise implementar nenhum código de *script*.

Esta solução permite que o componente seja reutilizado e customizado. E, como neste caso não é necessário acessar o estado da *view* JSF, é a opção mais simples a ser adotada, e qualquer ajuste ou alteração que apareça, pode ser implementado facilmente alterando-se o *Servlet*, ou a sua lógica.

O código JavaScript que será usado, exceto o código da sua inicialização, será separado em arquivos e embutidos através das facilidades implementadas na classe *HtmlRenderer*, bem como os arquivos CSS.

A aparência do componente será delegada para os arquivos CSS, dando mais liberdade para o desenvolvedor da aplicação alterar seu comportamento visual.

6.3.2 Os ajustes no renderizador

O renderizador – a classe *LabelDetailsAjax* – precisará sofrer pequenas alterações para disponibilizar os recursos necessários. Abaixo segue o código restante para fechar o método *encodeEnd()*, conforme Listagem 37.

```
...
/*
 * Escreve os arquivos css que serão usados
 */
writeStyleResource(context, "resources/labelDetailsAjax.css");
writeStyleResource(context, "resources/objviewer.css");

/*
 * Escreve os arquivos de JavaScript que serão usados
 */
//Arquivo com o framework JavaScript do capítulo 4
writeScriptResource(context, "js/ajaxf.js");
//Arquivo com o componente JavaScript outputTextLinked
writeScriptResource(context, "js/outputTextLinked.js");
//Arquivos JavaScript do utilitário objviewer de Ajax in Action
writeScriptResource(context, "js/ajaxinaction/styling.js");
writeScriptResource(context, "js/ajaxinaction/extras-array.js");
writeScriptResource(context, "js/ajaxinaction/objviewer.js");

//Script in-line para inicializar o componente outputTextLinked
writeScriptInline(context,
    "var oldLoader = window.onload || function () {};" +
    "window.onload = function () {" +
    "outText.init(" +
    "document.getElementById(\"" + clientId + ".label\")," +
    "\"" + clientId + ".detalheDiv\")," +
    "document.getElementById(\"" + clientId + ".botao\")," +
    "\"" + clientId + ".dadosDiv\")," +
    "\"" + (String) label.getAttributes().get("targetId") + "\", " +
    "\"" + (String) label.getAttributes().get("entity") + "\"" +
    ");" +
    "oldLoader.call(this)}");
...
```

Listagem 37. Ajustes na codificação do componente *LabelDetailsAjax*.

Os arquivos CSS que são incorporados ditam o comportamento visual do componente. No arquivo *.labelDetailsAjax.css* estão as classes que desempenham o papel principal:

- *rotulo*: é a classe associada ao texto que é exibido inicialmente;
- *detalheDivVisivel*: classe associada ao elemento *div* interno da janela que irá aparecer quando o usuário clicar no texto exibido;
- *detalheDivInvisivel*: classe associada ao elemento *div* interno da janela enquanto este não está visível ao usuário. O comportamento de visibilidade é delegado ao usuário, normalmente, nesta classe apenas precisa-se definir o atributo *visibility* para *hidden*;
- *legenda*: esta classe é associada ao elemento que exibe a legenda da janela;

No arquivo `objviewer.css` está definido o comportamento visual do componente que exibirá a grade dos dados dos detalhes solicitados.

Quanto aos arquivos `js`, no arquivo `ajaxf.js` está o framework JavaScript; no arquivo `outputTextLinked.js` está o componente JavaScript; no arquivo `objviewer.js` está o componente de visualização automática de objetos JavaScript em `tags` HTML; nos demais arquivos estão funções gerais utilitárias.

Depois destas definições, está a renderização do elemento de *script in line* que inicializa o componente JavaScript. A função `init()` é vinculada ao evento `window.onload`, usando a política da boa vizinhança, citada em 3.4. Nesta função, o componente `outputTextLinked` é inicializado passando-lhe os parâmetros necessários ao seu funcionamento.

Com estes arquivos, o cliente é capaz de realizar o comportamento dinâmico pretendido, usando as definições de exibição escolhidas pelo desenvolvedor da aplicação ou pelo *designer* do site.

6.3.3 O código JavaScript

O código do construtor da classe do componente `outputTextLinked` e dos `getters` necessários é definido como ilustra a Listagem 38. Os atributos privados do construtor serão providos pelo método `init()`, cujo código é detalhado na Listagem 39.

```
if (typeof outputTextLinked == "undefined") {
    outputTextLinked = function () {
        identifierDivDetalhe = "";
        identifierDivDados = "";
        servletTargetId = "";
        servletEntity = "";
    };
}
outputTextLinked.prototype.getIdentifierDivDados = function () {
    return identifierDivDados;
};
outputTextLinked.prototype.getIdentifierDivDetalhe = function () {
    return identifierDivDetalhe;
};
```

Listagem 38. Construtor e `getters` de `outputTextLinked`.

```
//Inicialização do componente outputTextLinked
outputTextLinked.prototype.init = function (label, idDetalheDiv,
button, idDadosDiv, targetId, entity) {
    //inicialização dos atributos privados do objeto outputTextLinked
    identifierDivDetalhe = idDetalheDiv;
    identifierDivDados = idDadosDiv;
    servletTargetId = targetId;
    servletEntity = entity;
    //associação dos eventos aos elementos DOM
    entAJAX.EventManager.attachEvent(label, "click",
outputTextLinked.showDetail, this, true);
    entAJAX.EventManager.attachEvent(label, "click",
outputTextLinked.getDetails, this, true);
    entAJAX.EventManager.attachEvent(button, "click",
outputTextLinked.hideDetail, this, true);
};
```

Listagem 39. Método `outputTextLinked.init`.

Os parâmetros do método `init()` são detalhados abaixo:

- **label:** é o elemento DOM correspondente ao texto exibido, no qual serão associados ao seu evento `onclick` os métodos `outputTextLinked.showDetail()` e o método `outputTextLinked.getDetails()`;
- **idDetalheDiv:** é o identificador do elemento `div` que encapsula a janela;
- **button:** é o elemento DOM responsável por fechar a janela, no qual será associado ao seu evento `onclick` o método `outputTextLinked.hideDetail()`;
- **idDadosDiv:** é o identificador do elemento `div` que exibirá os dados retornados do servidor;
- **targetId:** é o identificador do objeto a ser recuperado do servidor, será passado por parâmetro na requisição AJAX;
- **entity:** é o nome da entidade ao qual o `targetId` se refere.

O uso do comportamento AJAX encapsulado na biblioteca `entAJAX`, no arquivo `ajaxf.js`, é bem simples. Basta apenas o código definido no método abaixo para efetuar a requisição tal como na Listagem 40.

```
outputTextLinked.getDetails = function () {
var xhr = new entAJAX.HttpRequest();
    xhr.handler = "/WebApplication/servletJSFAjax";
xhr.setParam("targetId", servletTargetId);
    xhr.setParam("entity", servletEntity);
    xhr.setCompleteCallback(outputTextLinked.retorno);
    xhr.get();
};
```

Listagem 40. Uso do objeto `HttpRequest` pelo `outputTextLinked`.

Este código realizará a requisição assíncrona ao `servlet` na URL `"/WebApplication/servletJSFAjax"`, passando os parâmetros `targetId` e `entity`. O `servlet` realizará seu processamento e retornará um arquivo XML correspondente à requisição. Como o `servlet` realizará sua tarefa não será detalhado aqui, pois, na verdade, não interessa se obterá estes dados acionando um `web service`, ou acessando uma camada de mapeamento objeto-relacional, ou um container EJB, para o componente, é irrelevante. Porém, o `servlet` deve respeitar o formato do XML como dado a seguir:

```
<detalhes>
  <dados field_1="valueField_1"... field_n="valueField_n"/>
</detalhes>
```

O tratamento do retorno da requisição é resolvido pelo método `retorno`.

7 CONCLUSÃO

Ao longo deste trabalho foi possível desenvolver o entedimento de vários aspectos na construção de uma aplicação RIA com AJAX. Desde a implementação do AJAX pelo próprio desenvolvedor da aplicação, codificando JavaScript diretamente, até a implementação de componentes que propiciam o uso prático e descomplicado de funcionalidades AJAX. Principalmente, no que tange a construir componentes que auxiliem o desenvolvedor da aplicação. Componentes personalizados que são reutilizáveis e extensíveis e sua visualização flexível e customizável.

Com base em tudo que foi demonstrado e discutido, consegue-se extrair uma relação das vantagens e desvantagens das estratégias desenvolvidas, bem como uma interpretação de indicações do uso de cada uma delas.

7.1 PRIMEIRA ESTRATÉGIA: CODIFICAR AJAX DIRETAMENTE

Esta estratégia é indicada em casos que necessitam um controle de baixa granularidade das funcionalidades da aplicação.

7.1.1 Vantagens

Oferece controle de baixa granularidade sobre o processamento das requisições AJAX: Quando todo o código relativo ao processamento da requisições AJAX é escrito pelo desenvolvedor da aplicação, sem o uso de bibliotecas JavaScript ou qualquer outro *framework*, todo o controle sobre o AJAX está nas suas mãos. Funcionalidades que a biblioteca não provê podem ser implementadas, o código AJAX pode ser otimizado de acordo com suas necessidades específicas, como controles de segurança ou melhoras de performance.

7.1.2 Desvantagens

Requer o desenvolvimento de muito código relacionado ao AJAX: Quando não é usada nenhuma biblioteca JavaScript ou *framework*, muito código precisa ser escrito para processar as requisições AJAX. Esta abordagem toma tempo e aumenta o custo e o risco do projeto, pois o desenvolvedor poderia estar mais focado nas regras de negócio, ao invés de codificando detalhes técnicos que poderiam ser abstraídos.

Requer o conhecimento de várias tecnologias: A codificação das funcionalidades AJAX necessita de conhecimento avançado de várias tecnologias, como JavaScript, CSS e DOM, o que torna mais complexo tarefas que poderiam ser bem mais simples para o desenvolvedor.

Requer do desenvolvedor o conhecimento de incompatibilidades entre os browsers: Desde a criação do objeto para realizar a requisição AJAX, a customização da visualização com CSS, até a codificação em JavaScript, existem várias divergências entre os *browsers*. Vale lembrar que o AJAX em si não é um padrão W3C.

Requer do desenvolvedor a solução de questões de interface: Aplicações AJAX trazem várias questões de interface que precisam ser contornadas pelo desenvolvedor da aplicação: guardar o histórico de navegação do usuário, tratar os botões de voltar, avançar e recarregar, etc.

Pode se tornar um código muito difícil de depurar: Como o desenvolvimento de aplicações RIA com AJAX envolvem várias tecnologias, a depuração do código construído pelo desenvolvedor da aplicação pode se tornar muito difícil.

Expõe muito código JavaScript, se tornando um risco à segurança: Todo o código JavaScript da aplicação é visível pelo *browser*. Engenharia reversa da aplicação, interceptação de requisições AJAX, retornando respostas destrutivas são possibilidades que precisam ser bem gerenciadas.

7.2 SEGUNDA ESTRATÉGIA: USAR BIBLIOTECA QUE ENCAPSULE O AJAX

No caso em que apenas é desejado incluir a funcionalidade de requisições AJAX, pode-se usar esta estratégia para abstrair os detalhes e tornar a codificação mais simples.

7.2.1 Vantagens

Encapsula os detalhes do AJAX: Bibliotecas JavaScript podem esconder do desenvolvedor nuances técnicas, como a criação do objeto *XMLHttpRequest* e as operações sobre ele. Isto torna a codificação mais produtiva.

Reduz o código JavaScript necessário: Várias rotinas de manipulação de objetos e funções utilitárias, além de mecanismos mais elaborados de controle da visualização, podem ser usados diretamente das bibliotecas, reduzindo bastante o código necessário.

Incompatibilidades no processamento AJAX são encapsuladas: Nas bibliotecas JavaScript, as questões relativas as diferenças entre os *browsers*, referente ao AJAX, são tratadas transparentemente para o desenvolvedor. Assim como o tratamento das diferenças na manipulação de eventos e do DOM também são encapsulados.

Tratamento das questões de navegação: O suporte para os botões de navegação do *browser* e seu histórico, é tratado pelas bibliotecas.

Comunidades de usuários das bibliotecas podem ajudar na resposta a questionamentos: Bibliotecas como Dojo já possuem uma comunidade de usuários bem formada. Nestas comunidades encontra-se fóruns, blogs e outras fontes de informação. Além do mais, estes projetos também recebem apoio de outras empresas do mercado de desenvolvimento de *software*.

7.2.2 Desvantagens

Requer conhecimento de JavaScript: As bibliotecas não eliminam a necessidade de conhecimento e codificação em JavaScript.

Provavelmente, será necessário juntar várias bibliotecas para atender às necessidades do desenvolvedor: Apesar de ser colocado aqui como uma desvantagem, devido ao fato do desenvolvedor precisar se ambientar com cada uma, a possibilidade de escolha, e até o uso concomitante de várias bibliotecas, é uma flexibilidade interessante.

7.3 TERCEIRA ESTRATÉGIA: USAR UM FRAMEWORK CLIENTE-SERVIDOR QUE ENCAPSULA A BIBLIOTECA JAVASCRIPT COM AJAX

Em aplicações onde é mais importante o foco na lógica de negócios, e a produtividade e desempenho da equipe de desenvolvimento é o ponto crucial, é necessário um nível de abstração mais elevado, levando à construção de componentes reusáveis, extensíveis e fáceis de usar.

Usando o *framework* cliente-servidor *JavaServer Faces*, possibilita maior produtividade em IDEs que dão suporte *drag-and-drop* para seus componentes, como o NetBeans e o Eclipse (através de vários *plug-ins*).

7.3.1 Vantagens

O desenvolvedor da aplicação não precisa ter conhecimento de JavaScript, CSS e DOM: Como foi demonstrado no capítulo anterior, apenas é necessário referenciar a *tag library*, e usar a *tag* específica do componente. Todo o código necessário ao processamento AJAX e à manipulação do DOM, bem como a vinculação com classes de estilos em arquivos CSS, é realizado pela renderização do componente provida pelo JSF.

Componentes JSF-AJAX customizados são reutilizáveis: Assim como os componentes padrões, os componentes construídos para prover suporte ao AJAX no JSF também o são.

Desenvolvedores de componentes podem tirar vantagens das características da tecnologia JSF: A arquitetura JSF provê tratamento de eventos, conversão de dados e validação de dados informados pelo usuário da aplicação. Todas estas vantagens estão disponíveis para o desenvolvedor de componentes JSF AJAX.

Desenvolvedores de aplicações podem usar IDEs para construir suas aplicações em AJAX apenas usando arrastar-e-soltar componentes: Os componentes desenvolvidos para prover AJAX em JSF podem ser incorporados às várias IDEs presentes no mercado para tornar ainda mais fácil o desenvolvimento das aplicações, apenas arrastando e soltando os componentes para a *interface*.

7.3.2 Desvantagens

Para o desenvolvedor de componentes customizados para prover AJAX no JSF, ainda é necessário conhecimento e codificação em JavaScript, CSS e DOM, além do tratamento de incompatibilidades entre browsers: Antes de usufruir de todas as vantagens acima, é necessário, durante a construção dos componentes customizados, resolver todas as questões levantadas anteriormente. Porém, já existem no mercado vários *frameworks* desenvolvidos sobre JSF para prover AJAX aos componentes, como o Dynamic Faces, Ajax4JSF, Rich Faces, etc.

7.4 TRABALHOS FUTUROS

Extensões deste trabalho e outros estudos de aspectos desenvolvidos aqui podem ser criados, como, por exemplo, sobre algum dos temas citados abaixo:

- Análise do comportamento do MVC do *framework* JavaScript trabalhando com grande volume de dados: Uso de XSLT para melhorar a performance;
- Implementar *pool* de objetos *XMLHttpRequest* no *framework* JavaScript;
- Análise de performance entre as estratégias de implementação do AJAX no JSF;
- Uso de bibliotecas JavaScript com recursos avançados, como o "ext", para incrementar a camada de *view* da aplicação RIA;
- Desenvolvimento de componentes que usem *frameworks* Java que encapsulam bibliotecas JavaScript, como o jMaki;
- Desenvolvimento de componentes que usem *frameworks* remotos, como o DWR (Direct Web Remoting) e JSON-RPC (JavaScript Object Notation – Remote Procedure Call).

GARRETT, J. J. *Ajax: A New Approach to Web Applications*. [s.l.]: Adaptive Path, 2005. Disponível em: <http://www.adaptivepath.com/publications/essays/archives/000185.php>. Acesso em: 26 jul. 2007.

GEARY, D.; HORSTMANN, C. *Core JavaScript Faces*. 2nd. Santa Clara: Prentice Hall, 2007.

GOODMAN, D. *JavaScript Bible, Gold Edition*. New York: Hungry Minds, 2001.

GROSS, C. *Ajax Patterns and Best Practices*. New York: Apress, 2006.

HADLOCK, K. *Ajax for Web Application Developers*. Indianapolis: Sams Publishing, 2006.

HOLZNER, S. *Ajax Bible*. Indianapolis: Wiley Publishing, 2007.

JACOBI, J.; FALLOWS, J. R. *Pro JSF and Ajax: Building Rich Internet Components*. New York: Apress, 2006.

JOHNSON, D.; WHITE, A.; CHARLAND, A. *Enterprise AJAX: Strategies for Building High Performance Web Applications*. Santa Clara: Prentice Hall, 2007.

MAHEMOFF, M. *Ajax Design Patterns*. Sebastopol: O'Reilly, 2006.

MANN, K. D. *JavaScript Faces in Action*. Greenwich: MANNING, 2005.

MOORE, D.; BUDD, R.; BENSON, E. *Professional Web Internet Applications: AJAX and Beyond*. Indianapolis: Wiley Publishing, 2007.

MURRAY, G.; HOMBVE, T.; BURNS, T. *Using JavaScript Faces Technology with AJAX*. [s.l.]: Sun Microsystems, 2007. Disponível em: <https://opcsatlog.scy.java.net/ajax/jsf-ajax/>. Acesso em: 22 nov. 2007.

OLSON, S. D. *Ajax on Java*. Sebastopol: O'Reilly, 2007.

REFERÊNCIAS

- ASLESON, R.; SCHUTTA, N. T. **Pro Ajax and Java**. New York: APRESS, 2006a.
- ASLESON, R.; SCHUTTA, N. T. **Foundations of Ajax**. New York: APRESS, 2006b.
- BERGSTEN, H. **JavaServer Faces**. Sebastopol: O'Reilly, 2004.
- CRANE, D.; PASCARELLO, E.; JAMES, D. **Ajax in Action**. Greenwich: MANNING, 2006.
- CRANE, D.; BIBEALULT, B.; SONNEVELD, J. **Ajax in Practice**. Greenwich: MANNING, 2007.
- FOWLER, M. **Inversion of Control Design**. [s.l.]: [s.n.], 2005. Disponível em: <<http://www.martinfowler.com/bliki/InversionOfControl.html>>. Acesso em: 05 mai. 2007.
- GARRET, J. J. **Ajax: A New Approach to Web Applications**. [s.l.]: Adaptive Path, 2005. Disponível em: <<http://www.adaptivepath.com/publications/essays/archives/000385.php>>. Acesso em: 16 jul. 2007.
- GEARY, D.; HORSTMANN, C. **Core JavaServer Faces**. 2 ed. Santa Clara: Prentice Hall, 2007.
- GOODMAN, D. **JavaScript Bible, Gold Edition**. New York: Hungry Minds, 2001.
- GROSS, C. **Ajax Patterns and Best Practices**. New York: APRESS, 2006.
- HADLOCK, K. **Ajax for Web Applications Developers**. Indianapoles: Sams Publishing, 2006.
- HOLZNER, S. **Ajax Bible**. Indianapoles: Wiley Publishing, 2007.
- JACOBI, J.; FALLOWS, J. R. **Pro JSF and Ajax: Building Rich Internet Components**. New York: APRESS, 2006.
- JOHNSON, D.; WHITE, A.; CHARLAND, A. **Enterprise AJAX: Strategies for Building High Performance Web Applications**. Santa Clara: Prentice Hall, 2007.
- MAHEMOFF, M. **Ajax Design Patterns**. Sebastopol: O'Reilly, 2006.
- MANN, K. D. **JavaServer Faces in Action**. Greenwich: MANNING, 2005.
- MOORE, D.; BUDD, R.; BENSON, E. **Professional Rich Internet Applications: AJAX and Beyond**. Indianapoles: Wiley Publishing, 2007.
- MURRAY, G.; NORBYE, T.; BURNS, E. **Using JavaServer Faces Technology with AJAX**. [s.l.]: Sun Microsystems, 2007. Disponível em: <<https://bpcatalog.dev.java.net/ajax/jsf-ajax/>>. Acesso em: 22 nov. 2007.
- OLSON, S. D. **Ajax on Java**. Sebastopol: O'Reilly, 2007.

ORT, E.; BASLER, M. **Ajax Design Strategies**. [s.l.]: Sun Microsystems, 2006. Disponível em: <<http://java.sun.com/developer/technicalArticles/J2EE/AJAX/DesignStrategies/>>. Acesso em: 22 nov. 2007.

SHALLOWAY, A.; TROTT J. R. **Design Patterns Explained: A New Perspective on Object Oriented Design**. Boston: Addison-Wesley, 2000.

WYKE, R. A. et al. **Pure JavaScript**. 2 ed. Indianapoles: Sams Publishing, 2002.

ZAKAS, N. C.; MCPEAK, J.; FAWCETT, J. **Professional Ajax**. Indianapoles: Wrox Press, 2006.

ZAMMETTI, F. W. **Practical Ajax Projects with Java Technology**. New York: APRESS, 2006.