

UNIVERSIDADE FEDERAL DE MINAS GERAIS
ESCOLA DE ENGENHARIA
PROGRAMA DE PÓS GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ARQUITETURA EXPANSÍVEL PARA DESENVOLVIMENTO DE PÁGINAS
AJAX

BELO HORIZONTE – MINAS GERAIS
2009

LUCAS EUSTÁQUIO GOMES DA SILVA

ARQUITETURA EXPANSÍVEL PARA DESENVOLVIMENTO DE PÁGINAS
AJAX

Dissertação apresentada ao Programa de Pós Graduação em Engenharia Elétrica da Universidade Federal Minas Gerais, como requisito para a obtenção do título de Mestre em Engenharia Elétrica.

Área de Conhecimento: Sistemas de Computação

Orientação: Prof. Renato Cardoso Mesquita

BELO HORIZONTE – MINAS GERAIS

2009

SILVA, Lucas
Arquitetura expansível desenvolvimento de páginas AJAX. 2009.
86 f.

Orientador: Renato Cardoso Mesquita
Dissertação (mestrado) – Universidade Federal de Minas Gerais.

Dedicatória

Dedico este trabalho a Deus, a Jesus, à minha namorada, aos meus familiares e aos meus amigos.

A Deus, por ter se mostrado a mim de maneira irrefutável através da fé que tenho aprendido a cultivar em meu coração.

A Jesus por ter vindo aqui e nos mostrado que o amor é possível. Não sei o que seria de mim hoje sem a presença desse meigo nazareno em minha vida.

À minha namorada Natália pelo muito amor e apoio que tem me dado em todas as horas da minha vida (é muito mesmo!). Sem ela, com certeza eu não seria a mesma pessoa que sou hoje.

Aos meus familiares, pelo muito que me têm amado e amparado, nas mais variadas situações. Especialmente aos meus pais, que todas as páginas seriam poucas para descrever tanta dedicação e sacrifício.

Aos amigos, pois, afinal, são meus amigos e isso me basta para lhes dedicar o meu melhor sempre (ou quase sempre). Fica aqui um abraço carinhoso ao pessoal do Manoel, de Patos, do trabalho, da faculdade, do mestrado e a todos aqueles que se consideram meus amigos.

Agradecimentos

Agradeço a todas as pessoas que tornaram esse trabalho possível.

Ao Fábio Cesso e ao Guilherme Salles, colegas de trabalho, por terem me ensinado os fundamentos do AJAX. Ao professor Renato Mesquita por ter acreditado em mim e orientado esse trabalho. Aos familiares e amigos que nunca me deixaram desistir do mestrado e sempre me incentivaram a continuar.

Epígrafe

“E disse Pedro: Não tenho prata nem ouro; mas o que tenho isso te dou.”

Pedro, Atos dos Apóstolos 3:6.

Resumo

Páginas web assíncronas estão cada vez mais presentes na Internet. Essas páginas suportam uma apresentação de conteúdo mais flexível, pois conseguem alterar o próprio conteúdo sem ter que recarregar-se inteiramente. O paradigma de comunicação assíncrona é conhecido como AJAX.

Entretanto, o nível de complexidade do desenvolvimento de páginas assíncronas é bem maior. Para facilitar esse desenvolvimento foram criados diversos frameworks web com suporte a AJAX, sendo que vários estão disponíveis para download gratuito na Internet. Cada um desses frameworks apresenta uma abordagem para o problema da comunicação assíncrona e a maior crítica a eles é que estão preparados para lidar apenas com páginas DHTML/HTML, não possuindo suporte AJAX a outras tecnologias como o Silverlight e o SVG (**S**calable **V**ector **G**raphics). Exceção feita apenas ao framework de código fechado da Microsoft, o Asp.Net, que suporta o Silverlight.

O objetivo deste trabalho é produzir uma arquitetura expansível para um framework web que suporte outras tecnologias além do HTML. Inicialmente foi inserido suporte para SVG e Silverlight. Por arquitetura deve-se entender a modelagem das classes com atribuições e responsabilidades bem definidas. Tanto as classes usadas no aplicativo servidor quanto as usadas no aplicativo cliente são contempladas.

Para atingir esse objetivo foram estudadas as tecnologias envolvidas em uma requisição AJAX. Problemas de implementação e arquiteturas de servidores também foram discutidos. A arquitetura proposta foi criada a partir do padrão MVC (**M**odel **V**iew **C**ontroller). Para demonstrar a viabilidade dessa arquitetura foi implementado, em C#, um framework nomeado KIS (**K**ee **I**t **S**imple).

Palavras-chave

AJAX, Framework, Silverlight, SVG, Web, DOM, C#.

Abstract

Asynchronous web pages have gained popularity over the Internet recently. These pages support more flexible content presentation, because they can change their content without having to reload themselves entirely. The paradigm of asynchronous communication is known as AJAX.

However, the development of asynchronous web pages is much more complex. To aid this development, several web frameworks with AJAX support have been created, and many of them are available for free download on the Internet. Each one of these frameworks presents a different approach to solve the problem of asynchronous communication. The biggest critic to all of them is that they are designed to work only with DHTML/HTML. They do not offer AJAX support to other technologies such as Silverlight and SVG (**S**calable **V**ector **G**raphics). Exception made only to the non open-source Microsoft .Net framework, which supports Silverlight.

The objective of this work is to produce an expandable web framework architecture that supports other technologies besides HTML. Initially, support for SVG and Silverlight was added. By architecture we mean the class modeling with well-defined roles and responsibilities. Classes used in both server-side application and client-side application are considered.

To achieve this objective, the technologies involved in an AJAX request were studied. Implementation problems and server-side architectures were also discussed. The proposed architecture was built upon the pattern MVC (**M**odel **V**iew **C**ontroller). To demonstrate the feasibility of this architecture, a framework named KIS (**K**eeP **I**t **S**imple) was implemented in C #.

Keywords

AJAX, Framework, Silverlight, SVG, Web, DOM, C#.

Lista de figuras

| | |
|---|----|
| Figura 1 - Comunicação síncrona..... | 7 |
| Figura 2 - Comunicação assíncrona | 8 |
| Figura 3 – Código HTML dinâmico | 9 |
| Figura 4 - Página antes e depois de passar o mouse sobre o texto | 10 |
| Figura 5 – Código HTML de uma tabela | 11 |
| Figura 6 - Representação DOM da tabela vista na Figura 5..... | 11 |
| Figura 7 - Definição da cor de um elemento via HTML e interface DOM..... | 12 |
| Figura 8 - Exemplo de CSS: Código HTML à esquerda e página à direita..... | 13 |
| Figura 9 - Exemplo fictício de requisição-resposta para o Google..... | 15 |
| Figura 10 - Página atualizada com as informações obtidas via Ajax | 15 |
| Figura 11 - Resposta real enviada pelo Google para o exemplo dado | 16 |
| Figura 12 - Exemplo de documento SVG | 18 |
| Figura 13 - Visualização do exemplo de documento SVG..... | 19 |
| Figura 14 - Exemplo de documento Silverlight | 20 |
| Figura 15 – Visualização do exemplo de documento Silverlight | 20 |
| Figura 16 - Macro arquitetura AJAX | 22 |
| Figura 17 - Página inicial do mecanismo de busca Google | 25 |
| Figura 18 - Trecho de código possível para o buscador do Google | 25 |
| Figura 19 - Requisição e resposta gerada pelo auto-complete do Google | 26 |
| Figura 20 - Resultado da atualização executada pelo mecanismo AJAX..... | 26 |
| Figura 21 - Envio de parâmetros na requisição assíncrona..... | 29 |
| Figura 22 - Registro de <i>callback</i> para uma chamada assíncrona | 31 |
| Figura 23 - Hierarquia das interfaces principais do DOM | 35 |
| Figura 24 - Codificação manual da resposta a uma solicitação AJAX..... | 37 |
| Figura 25 - Macro arquitetura AJAX | 43 |
| Figura 26 - Diagrama de sequência da Interface..... | 44 |
| Figura 27 - Diagrama de atividades da interface | 44 |
| Figura 28 - Interface do Controlador..... | 47 |
| Figura 29 - Diagrama de atividades do controlador síncrono | 48 |
| Figura 30 - Diagrama de atividades do controlador assíncrono..... | 50 |
| Figura 31 - Interfaces das classes do Modelo | 52 |
| Figura 32 - Requisição síncrona de pedido de novo documento | 55 |

| | |
|---|----|
| Figura 33 - Requisição de evento síncrono | 55 |
| Figura 34 - Requisição de evento assíncrono..... | 56 |
| Figura 35 - Modelo do cliente de um aplicativo AJAX..... | 57 |
| Figura 36 - Implementação da interface em C#..... | 61 |
| Figura 37 - Processamento de requisição síncrona..... | 62 |
| Figura 38 - Processamento de requisição assíncrona | 62 |
| Figura 39 - Método <i>WriteCommands</i> do <i>CommandLogger</i> | 62 |
| Figura 40 - Implementação dos métodos <i>GetInnerText</i> e <i>SetInnerText</i> | 63 |
| Figura 41 - Principais funções do mecanismo AJAX | 64 |
| Figura 42 - Metodo <i>SetInnerText</i> | 66 |
| Figura 43 – Interface gráfica da página Hello world..... | 67 |
| Figura 44 - Código da página de Hello world..... | 67 |
| Figura 45 - Código HTML do Hello world..... | 68 |
| Figura 46 – Código XML do documento SVG..... | 69 |
| Figura 47 - Código XML do documento Silverlight..... | 69 |
| Figura 48 - Scripts de inicialização do documento Silverlight | 70 |
| Figura 49 - Página com um sinótico SVG e Silverlight | 70 |
| Figura 50 - Criação de evento usando-se o KIS | 72 |
| Figura 51 - Solicitação assíncrona do evento de atualizar os valores de set point..... | 73 |
| Figura 52 - Resposta à solicitação assíncrona da figura anterior | 73 |
| Figura 53 – Listener do evento de atualização dos valores do tanque | 74 |
| Figura 54 - Resposta ao evento de atualização dos sinóticos..... | 75 |

Lista de tabelas

| | |
|--|----|
| Tabela 1 - Métodos do objeto XMLHttpRequest | 17 |
| Tabela 2 - Propriedades do objeto XMLHttpRequest | 17 |
| Tabela 3 - Interface <i>EventListener</i> | 33 |
| Tabela 4- Interface <i>Node</i> | 33 |
| Tabela 5 - Interface <i>Element</i> | 33 |
| Tabela 6 - Interface <i>EventTarget</i> | 34 |
| Tabela 7 - Comparação de dados trafegados por tipo de requisição | 76 |
| Tabela 8 - Comparação de tempo gasto por tipo de requisição | 76 |

Lista de abreviaturas

| Sigla | Significado |
|--------------|--|
| AJAX | Assynchronous Javascript and XML |
| API | Application Programming Interface |
| CSS | Cascading Style Sheets |
| DHTML | Dynamic HTML |
| DOM | Document Object Model |
| IIS | Internet Information Services |
| HTML | Hyper Text Markup Language |
| HTTP | HyperText Transfer Protocol |
| MVC | Mode View Controller |
| RIA | Rich Interactive Application |
| RPC | Remote Procedure Call |
| RFC | Request For Comments |
| SVG | Scalable Vector Graphics |
| W3C | World Wide Web Consortium |
| XML | eXtensible Markup Language |

Sumário

| | | |
|----------|--|-----------|
| 1 | INTRODUÇÃO | 1 |
| 1.1 | TEMA..... | 1 |
| 1.2 | OBJETIVO E METODOLOGIA..... | 3 |
| 1.2.1 | Metodologia..... | 3 |
| 1.3 | JUSTIFICATIVA E CONTRIBUIÇÃO..... | 4 |
| 1.4 | ESTRUTURA DA DISSERTAÇÃO | 5 |
| 2 | TECNOLOGIAS PARA PÁGINAS INTERATIVAS NA WEB..... | 6 |
| 2.1 | AJAX..... | 6 |
| 2.1.1 | Comunicação cliente-servidor | 6 |
| 2.1.1.1 | Comunicação síncrona..... | 6 |
| 2.1.1.2 | Comunicação assíncrona | 7 |
| 2.1.2 | Tecnologias AJAX | 9 |
| 2.1.2.1 | HTML Dinâmico | 9 |
| 2.1.2.2 | Document Object Model (DOM) | 10 |
| 2.1.2.3 | CSS (<i>Cascading Style Sheets</i>) | 12 |
| 2.1.2.4 | JavaScript | 13 |
| 2.1.2.5 | XML | 14 |
| 2.1.2.6 | XMLHttpRequest | 16 |
| 2.2 | SVG..... | 18 |
| 2.3 | SILVERLIGHT | 19 |
| 3 | ARQUITETURA AJAX | 22 |
| 3.1 | INCLUSÃO DE SUPORTE AJAX NO CLIENTE..... | 28 |
| 3.1.1 | Gerando uma requisição Assíncrona..... | 28 |
| 3.1.1.1 | Como enviar parâmetros..... | 29 |
| 3.1.1.2 | Parâmetro de identificação da solicitação | 29 |
| 3.1.1.3 | Parâmetros com dados para o processamento da solicitação | 30 |
| 3.1.2 | Tratamento da resposta de uma chamada assíncrona..... | 30 |
| 3.2 | COMANDOS DOM | 32 |
| 3.3 | INCLUSÃO DE SUPORTE AJAX NO SERVIDOR..... | 36 |
| 3.3.1 | Codificação manual da resposta a uma requisição assíncrona | 36 |
| 3.3.2 | Publicação Assíncrona no cliente de funções do servidor..... | 38 |
| 3.3.3 | Inclusão de suporte AJAX em arquiteturas MVC dos Servidor..... | 39 |
| 4 | PROPOSIÇÃO DE UMA ARQUITETURA AJAX EXPANSÍVEL | 42 |

| | | |
|----------|--|-----------|
| 4.1 | ARQUITETURA MVC DO SERVIDOR..... | 42 |
| 4.1.1 | Interface | 43 |
| 4.1.2 | Controlador..... | 45 |
| 4.1.2.1 | Pontos comuns entre os dois controladores..... | 45 |
| 4.1.2.2 | Controlador síncrono | 48 |
| 4.1.2.3 | Controlador assíncrono..... | 49 |
| 4.1.3 | Modelo..... | 51 |
| 4.2 | CÓDIGO INCLUÍDO NO CLIENTE..... | 57 |
| 4.3 | INCLUSÃO DE SUPORTE A NOVAS TECNOLOGIAS | 58 |
| 5 | KIS, UMA IMPLEMENTAÇÃO DA ARQUITETURA PROPOSTA..... | 60 |
| 5.1 | IMPLEMENTAÇÃO DA PARTE DO SERVIDOR..... | 60 |
| 5.1.1 | Interface | 60 |
| 5.1.2 | Controlador..... | 61 |
| 5.1.3 | Modelo..... | 63 |
| 5.2 | IMPLEMENTAÇÃO DA PARTE DO CLIENTE | 64 |
| 5.3 | EXEMPLO DE USO..... | 66 |
| 5.3.1 | Hello World!..... | 66 |
| 5.3.2 | Sinótico..... | 70 |
| 5.3.2.1 | Eventos do sinótico..... | 71 |
| 5.3.2.2 | Comparação de desempenho: Síncrono x Assíncrono | 75 |
| 6 | CONCLUSÕES E POSSIBILIDADES FUTURAS..... | 78 |
| | REFERÊNCIAS BIBLIOGRÁFICAS | 83 |

1 INTRODUÇÃO

1.1 TEMA

Na última década os aplicativos web foram ganhando o espaço antes ocupado por aplicativos desktop. Essa mudança aconteceu por várias razões, como por exemplo, o aumento na segurança do acesso a dados restritos, diminuição de tempo gasto com instalação de produtos e a popularização dos navegadores web.

Entretanto, se por um lado houve ganhos significativos, por outro a plataforma web ainda deixava a desejar quanto aos recursos que disponibilizava. Uma aplicação web tinha uma interação muito limitada com o usuário e possuía um tempo de resposta bem maior quando comparado ao dos aplicativos desktop. A limitação de velocidade se devia a dois fatores: a baixa velocidade da rede e o modelo síncrono de comunicação HTTP. A evolução tecnológica cuidou de resolver o primeiro problema e o AJAX se apresentou para contornar o segundo.

AJAX é um acrônimo consagrado por *GARRET* [1], e significa **A**synchronous **J**avaScript and **X**ML (**eX**tensible **M**arkup **L**anguage). Porém o que temos é muito mais que a junção de JavaScript (linguagem usada para atualização dinâmica de uma página) com XML (linguagem utilizada para codificação de uma resposta assíncrona), é todo um novo conceito de navegação e atualização de páginas web [2]. O resultado da abordagem AJAX para as aplicações web é aumentar o grau de interação com o usuário e diminuir o tempo de resposta da aplicação. O **GOOGLE MAPS** [3] e o **GMAIL** [4] são exemplos de aplicações web bem conhecidas que seguem essa filosofia.

Pode-se dizer, de maneira superficial, que o papel do AJAX é distribuir a comunicação cliente-servidor em pequenas requisições assíncronas, fazendo com o usuário tenha a impressão de que as coisas estão acontecendo em tempo real, como se fossem eventos no cliente. Como as requisições são mais simples, o tempo de tráfego na rede é menor e o tempo de processamento desses dados também decai, o que gera uma diminuição sensível no tempo de resposta.

Outro fator importante é que as requisições são assíncronas. Como o usuário não precisa ficar esperando a resposta do servidor sempre que o aplicativo faz uma requisição, ele consegue fazer várias tarefas em paralelo, diminuindo o tempo total que seria gasto caso a comunicação fosse síncrona.

A implementação do AJAX, entretanto, é bem trabalhosa e exige alguns cuidados. Entre os principais pontos a serem observados, temos:

- deve-se ter um conhecimento avançado de JavaScript;
- deve-se estabelecer uma padronização muito forte no formato da resposta à requisição assíncrona, já que essa resposta é codificada em texto puro e não há nenhum protocolo formal adotado universalmente;
- deve-se preparar uma infra-estrutura no servidor para habilitá-lo a tratar corretamente esse tipo de requisição;
- é necessário prover uma forma de enviar todas as informações relevantes na requisição AJAX.

E ainda assim, deve-se tomar cuidado com alguns problemas:

- incompatibilidade entre navegadores;
- exposição da lógica de negócio;
- código acoplado à apresentação tornando a reutilização pequena;

Para contornar estes problemas é que surge a figura do framework. Um framework pode ser definido como “um sistema que pode ser customizado, especializado, ou estendido para prover funcionalidades mais específicas e apropriadas” [5]. Dessa forma um framework AJAX consegue contornar os problemas acima, encapsulando a geração de uma solução AJAX para cada página criada a partir desta ferramenta. Um contraponto na adoção de frameworks, é que eles introduzem um overhead na utilização dos recursos para o encapsulamento dos problemas citados. Utilizá-los ou não depende da natureza da aplicação.

É neste contexto que este trabalho se insere. A proposta deste trabalho é criar uma arquitetura expansível de framework AJAX. O diferencial entre a arquitetura aqui apresentada e as arquiteturas de frameworks existentes é a expansibilidade. O conceito de expansibilidade adotado aqui é o de se poder incluir nativamente outras tecnologias que não o HTML (**H**yper**T**ext **M**arkup **L**anguage) / DHTML (**D**ynamic HTML), desde que essas tecnologias disponibilizem uma interface de atualização via JavaScript.

1.2 OBJETIVO E METODOLOGIA

O objetivo deste trabalho é o de se produzir uma arquitetura expansível para um framework web. Mais especificamente, o objetivo deste trabalho é produzir uma arquitetura que possibilite suporte AJAX a outras tecnologias além do HTML

Por arquitetura deve-se entender a modelagem das classes com atribuições e responsabilidades bem definidas, e por expansibilidade deve-se entender facilidade de inclusão de novas tecnologias que fazem uso de uma interface DOM (**D**ocument **O**bject **M**odel – interface usada para atualização via JavaScript).

1.2.1 Metodologia

Para alcançar os objetivos propostos, a metodologia usada foi a seguinte:

1. Análise da arquitetura cliente-servidor de aplicativos AJAX. Essa análise foi importante para se ter uma visão geral de um aplicativo AJAX. Conceitos importantes como mecanismo AJAX, criação e tratamento de uma requisição assíncrona, codificação dos dados enviados e recebidos foram estudados nessa fase.
2. Estudo da interface DOM (a interface DOM é usada via JavaScript para alterar uma página já carregada) do HTML e de outras tecnologias como, por exemplo, o SVG (**S**calable **V**ector **G**raphics) e o Silverlight, que são tecnologias usadas para a criação de gráficos vetoriais. As principais fontes usadas nesse estudo foram as especificações do W3C (**W**orld **W**ide **W**eb **C**onsortium) [6] que é o consórcio internacional

que trabalha para desenvolver padrões Web, além de diversos fóruns e páginas na Internet e livros sobre o assunto. Esse estudo teve a finalidade de identificar os tipos de alterações possíveis em uma página web.

3. Estudo das arquiteturas mais comuns de servidores AJAX. Neste ponto foram identificadas as três famílias principais de arquiteturas AJAX e a arquitetura MVC (**M**odel **V**iew **C**ontroller) [7] foi escolhida para basear o projeto da implementação.
4. Proposição da arquitetura expansível com base nas respostas a serem geradas. Um framework AJAX deve gerar dois tipos de respostas: código fonte dos documentos mostrados e comandos necessários para alterá-los. A estrutura de servidor proposta foi capaz de contemplar os dois casos.
5. Validação da arquitetura através de sua implementação em um framework. O nome dado a esse framework foi KIS (**K**ee**P** **I**t **S**imple). Para validar o conceito de expansibilidade, foi incluído o suporte AJAX a duas tecnologias, o SVG e o Silverlight, além do HTML.

1.3 JUSTIFICATIVA E CONTRIBUIÇÃO

O padrão HTML foi projetado para tratar dados de formulários, ou seja, caixas de texto, botões, figuras, etc. Não existe em HTML elementos para descrever elipses, retas ou outros elementos geométricos. Desse modo ele não é apropriado para fazer desenhos, gráficos, etc. Para suprir essas e outras limitações é que foram criados os plugins web. Plugins web, tais como o SVG e o Silverlight, complementam o HTML e possibilitam a criação de páginas com conteúdo mais rico.

Atualmente o AJAX tem sido usado principalmente para tratar conteúdo HTML, mas nada impede que seja usado em conjunto com outras tecnologias. Ele pode ser usado com qualquer outra tecnologia que disponibilize uma interface de atualização via JavaScript e muitos plugins web disponibilizam essa interface.

A contribuição desse trabalho é criar uma arquitetura que possibilite integrar essas tecnologias ao AJAX. Mas o produto dessa dissertação não é somente a criação dessa arquitetura, é também todo o estudo necessário para criá-la. É uma compilação de problemas e respectivas soluções adotadas, ou seja, padrões de projeto para AJAX. Como os frameworks AJAX apresentam similaridades em seu funcionamento, a estrutura proposta poderá ser aproveitada para eles se as adaptações necessárias forem feitas para os casos particulares.

1.4 ESTRUTURA DA DISSERTAÇÃO

O capítulo 1 apresentou a motivação, os objetivos, a metodologia e as contribuições desta dissertação. O capítulo 2 apresenta brevemente o AJAX, o SVG e o Silverlight. O objetivo é familiarizar o leitor com o assunto aqui tratado. O capítulo 3 apresenta conceitos gerais da arquitetura de aplicativos AJAX mostrando o papel de cada uma das partes dessa arquitetura. Apresenta também arquiteturas comumente encontradas em frameworks AJAX. O capítulo 4 mostra a arquitetura expansível proposta. É o capítulo principal deste trabalho. O capítulo 5 apresenta brevemente o KIS (**Keep It Simple**), um framework de código aberto, com suporte a SVG e Silverlight, desenvolvido para mostrar a viabilidade da arquitetura apresentada. E por fim, o capítulo 6 apresenta as conclusões e possibilidades futuras deste trabalho.

2 TECNOLOGIAS PARA PÁGINAS INTERATIVAS NA WEB

2.1 AJAX

O AJAX é uma sigla cunhada a partir de **A**ssynchronous **J**avascript **A**nd **X**ML, e representou uma grande mudança no desenvolvimento Web. Ele não é uma tecnologia, e sim a junção de várias. O AJAX usa essas tecnologias de uma maneira inovadora, fazendo com que trabalhem juntas. Cada uma com uma funcionalidade específica, criando um novo modelo de desenvolvimento para as aplicações Web [8]. A grande vantagem desse modelo é que ele rompe com a limitação da comunicação síncrona cliente-servidor.

2.1.1 Comunicação cliente-servidor

Por comunicação cliente-servidor devemos entender o binômio requisição-resposta. Cada requisição ou solicitação do cliente (navegador) gera uma resposta por parte do servidor. Essa comunicação pode acontecer de maneira síncrona ou assíncrona.

2.1.1.1 Comunicação síncrona

Comunicação síncrona é aquela na qual o navegador envia uma requisição para o servidor contendo todos os dados da página, passando a ficar em modo de espera até que obtenha a resposta. Neste caso o servidor retorna uma página completa, seja ela a mesma página (com ou sem alterações) ou outra inteiramente nova.

A Figura 1 mostra como o usuário é constantemente interrompido durante a comunicação síncrona, tendo que esperar a cada requisição. Por esse motivo esse padrão de comunicação é descrito por SMITH [9] como “um padrão de comunicação extremamente incômodo que causa diminuição a produtividade”.

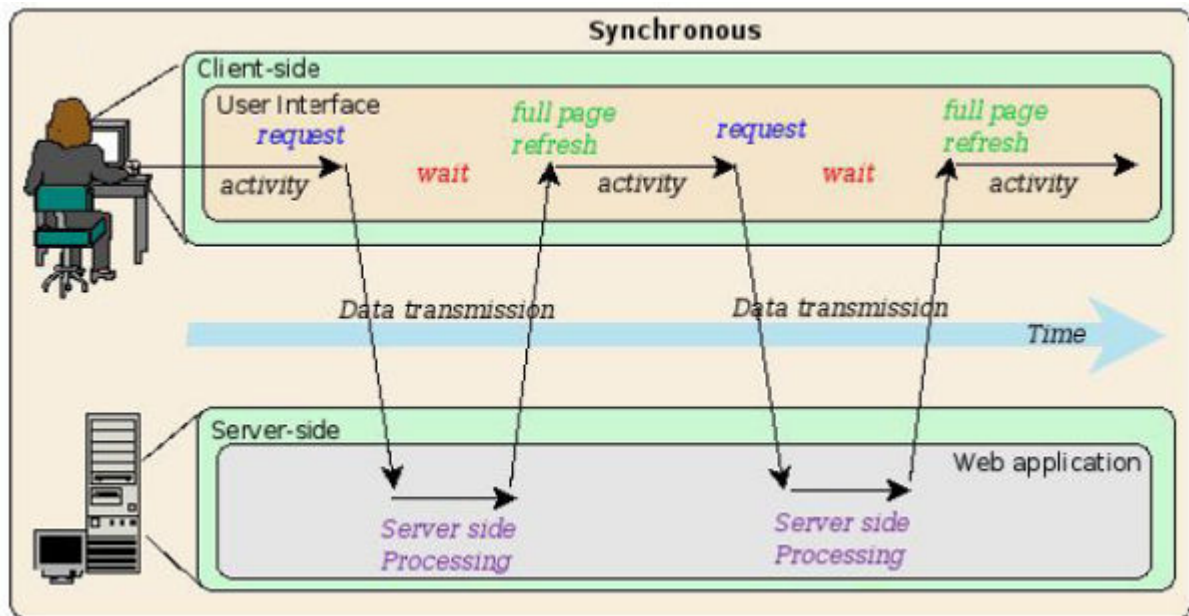


Figura 1 - Comunicação síncrona [10]

Esse modelo clássico cria muitos problemas para as aplicações Web devido ao desempenho lento, redundância desnecessária no envio e recebimento de informações, uso excessivo de banda e interação limitada [11]. Apesar das suas limitações é um modelo que atende bem a páginas com muita informação estática, por exemplo, páginas de notícias ou de busca [11].

2.1.1.2 Comunicação assíncrona

Neste tipo de comunicação o navegador envia uma requisição para o servidor e continua normalmente seu fluxo de execução. Quando a resposta chega o navegador é avisado para dar o devido tratamento. Assim o usuário não precisa esperar a resposta do servidor para continuar sua navegação. As requisições transportam apenas as informações necessárias e o servidor não precisa reenviar a página inteira, apenas os dados relevantes.

Para executar a comunicação assíncrona um mecanismo de tratamento AJAX é incluído. A função desse mecanismo é montar a requisição assíncrona com os dados esperados pelo servidor e realizar o tratamento da resposta quando esta for recebida. Esse tratamento consiste em interpretar essa resposta e atualizar os componentes necessários via DOM [12]. Não há, portanto a necessidade de se atualizar a página inteira.

A Figura 2 mostra como essa comunicação funciona. O usuário gera um evento que é tratado pelo mecanismo AJAX. Esse mecanismo envia uma requisição assíncrona para o servidor. Quando o servidor retorna a resposta, o mecanismo é acionado novamente para atualizar a página.

Nem todos os eventos precisam de informações do servidor e por isso nem sempre são geradas requisições. Há ainda os eventos que precisam de algumas informações disponíveis no cliente e de outras disponíveis no servidor. Nesse caso alguma atualização pode ser feita antes da resposta chegar. A atividade do usuário não é interrompida em momento algum.

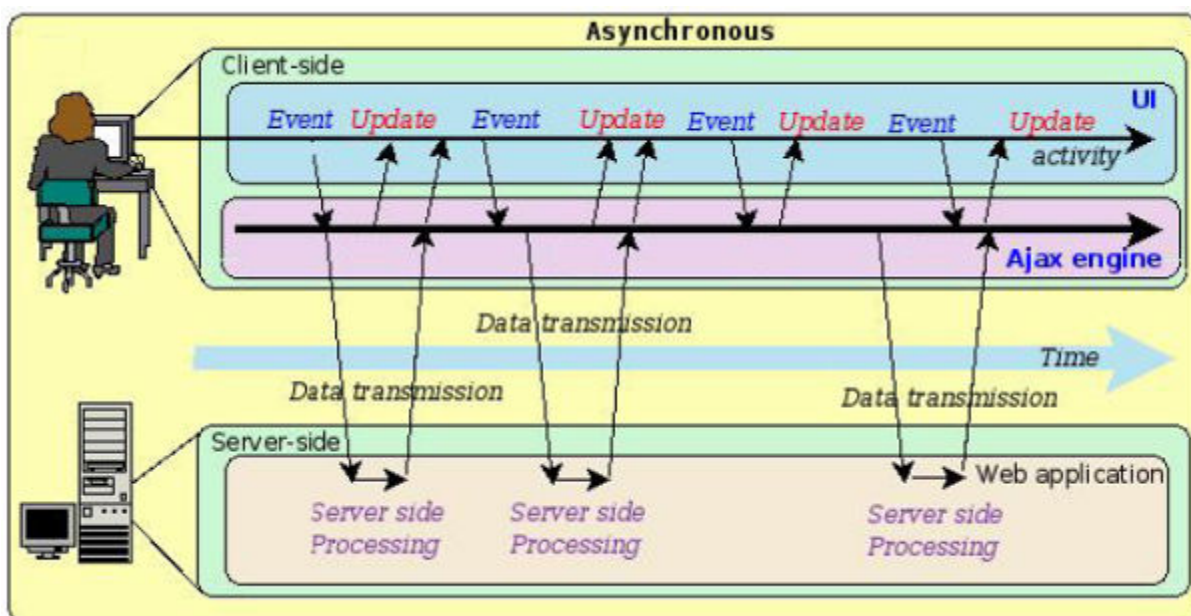


Figura 2 - Comunicação assíncrona [11]

O AJAX elimina a necessidade de se atualizar a página inteira a cada requisição do usuário (como acontece no modelo web clássico). Com a comunicação assíncrona, a transmissão de dados é mais eficiente, pois a atualização é feita somente com os dados necessários. Dessa maneira há mais interatividade e rapidez [11]. Essa mudança faz com que as aplicações web tenham um comportamento mais parecido com as aplicações desktop.

Como todas as tecnologias, o AJAX também tem as suas desvantagens. Como principais desvantagens podemos citar:

- aumento no uso de banda: a criação de várias requisições assíncronas simultaneamente pode levar ao aumento no consumo de banda, ainda que cada requisição seja menor que uma requisição síncrona equivalente;
- problema no histórico do navegador: como as páginas são atualizadas via JavaScript, os botões de navegação de páginas (anterior e próxima) não irão funcionar corretamente;
- possibilidade de criação de falhas de segurança que podem ser usadas para execução de código malicioso.

2.1.2 Tecnologias AJAX

Conforme o próprio nome indica, AJAX (**A**ssynchronous **J**avaScript **A**nd **X**ML) é uma técnica recente que usa um conjunto de tecnologias padrões. Essas tecnologias são suportadas por múltiplos navegadores em múltiplos sistemas operacionais [13]. As tecnologias não são novas e nem instáveis, ao contrário, são maduras, estáveis e bastante conhecidas. As principais tecnologias usadas pelo AJAX são listadas nos itens a seguir.

2.1.2.1 HTML Dinâmico

O HTML dinâmico consiste no uso conjunto de HTML, CSS e JavaScript. O resultado é uma página Web dinâmica, que vai alterando sua aparência à medida que interage com o usuário [14].

```
<html>
  <body>
    <span
      style="font-size: 25px; color: black;"
      onmouseover="this.style['color'] = 'red';"
      onmouseout="this.style['color'] = 'black';">Texto</span>
    </body>
  </html>
```

Figura 3 – Código HTML dinâmico

Texto

Texto

Figura 4 - Página antes e depois de passar o mouse sobre o texto

A Figura 3 e a Figura 4 mostram uma página que muda a cor de um texto quando se passa o mouse por cima dele. Nesse caso o código HTML, mostrado na Figura 3, descreveu um objeto de texto com tamanho, cor, fonte, etc. Quando se passa o mouse sobre esse texto ele gera o evento *onmouseover* e nesse evento suas propriedades são alteradas por meio dos objetos DOM via JavaScript. A aparência da página antes e depois da alteração pode ser vista na Figura 4.

2.1.2.2 Document Object Model (DOM)

O **D**ocument **O**bject **M**odel (DOM) [15] é uma interface de programação (API) para HTML e tecnologias baseadas em XML. Ela define a estrutura lógica de documentos e o modo como um documento é acessado e manipulado.

Com o DOM, programadores podem construir o documento, navegar em sua estrutura, adicionar, modificar ou apagar elementos e conteúdo. Qualquer elemento encontrado em um HTML ou XML pode ser acessado, modificado, apagado ou adicionado usando o DOM [16].

O DOM é uma API de programação para documentos. Ele representa fielmente a estrutura do documento que modela usando árvores. Para mostrar esse conceito considere o trecho da Figura 5, obtido de um documento HTML. O DOM representa o trecho da Figura 5 na estrutura mostrada na Figura 6. Essa figura mostra a estrutura em árvore usada pelo DOM. O nó *TABLE* é pai do nó, *TBODY* QUE possui dois filhos *TR* e assim por diante. Pode-se perceber que todas as informações e relacionamentos contidos no HTML foram mantidos.

O nome *Document Object Model* foi escolhido devido ao DOM ser um modelo de objetos no sentido tradicional de orientação a objetos. Documentos são modelados usando objetos, e o modelo engloba não somente a estrutura de um documento, mas também o comportamento e os objetos que o compõem [16]. Em outras palavras, os nós do diagrama da Figura 6 não representam uma estrutura de

dados, eles representam objetos que têm funções e identidade. Como um modelo de objetos, o DOM identifica:

- As interfaces e objetos usados para representar e manipular um documento;
- A semântica dessas interfaces e objetos – incluindo comportamento e atributos;
- Os relacionamentos e colaborações entre essas interfaces e objetos.

```
1 <TABLE>
2   <TBODY>
3     <TR>
4       <TD>Shady Grove</TD>
5       <TD>Aeolian</TD>
6     </TR>
7     <TR>
8       <TD>Over the River, Charlie</TD>
9       <TD>Dorian</TD>
10    </TR>
11  </TBODY>
12 </TABLE>
```

Figura 5 – Código HTML de uma tabela

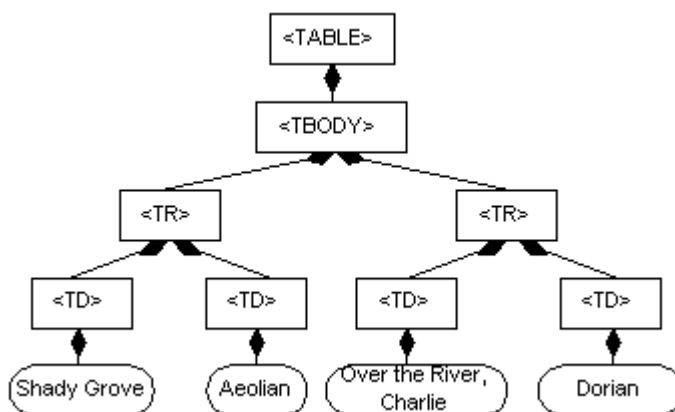


Figura 6 - Representação DOM da tabela vista na Figura 5

Cada nó expõe uma série de métodos, através dos quais é possível alterar as suas propriedades. Essas alterações são refletidas imediatamente na exibição dos documentos. O DOM de páginas HTML é acessado via JavaScript. Por exemplo, se temos um nó exibindo um texto em uma página HTML e, via JavaScript, alterarmos a cor do texto desse nó, imediatamente essa cor será mostrada pelo navegador. Através do DOM também é possível adicionar event listeners.

A Figura 7 mostra como definir a cor de um elemento via HTML e via interface DOM. O resultado final é o mesmo, mas um é feito através do código fonte HTML e o outro pode ser executado dinamicamente.

O DOM é essencial para o AJAX. É por meio dele que a resposta de uma requisição assíncrona é refletida na página.

```
<span id="span1" style="color: black;">Texto</span>
<script type="javascript">
var span1 = document.getElementById('span1');
span1.style['color'] = 'black';
</script>
```

Figura 7 - Definição da cor de um elemento via HTML e interface DOM

2.1.2.3 CSS (*Cascading Style Sheets*)

Com CSS é possível definir padrões de aparência e comportamento dos elementos de uma página. As cores, parágrafos, margens, espaçamentos, espessura ou qualquer outro elemento da página são definidos a parte em um ou mais estilos. Quando um objeto referencia um estilo ele herda as propriedades nele declaradas.

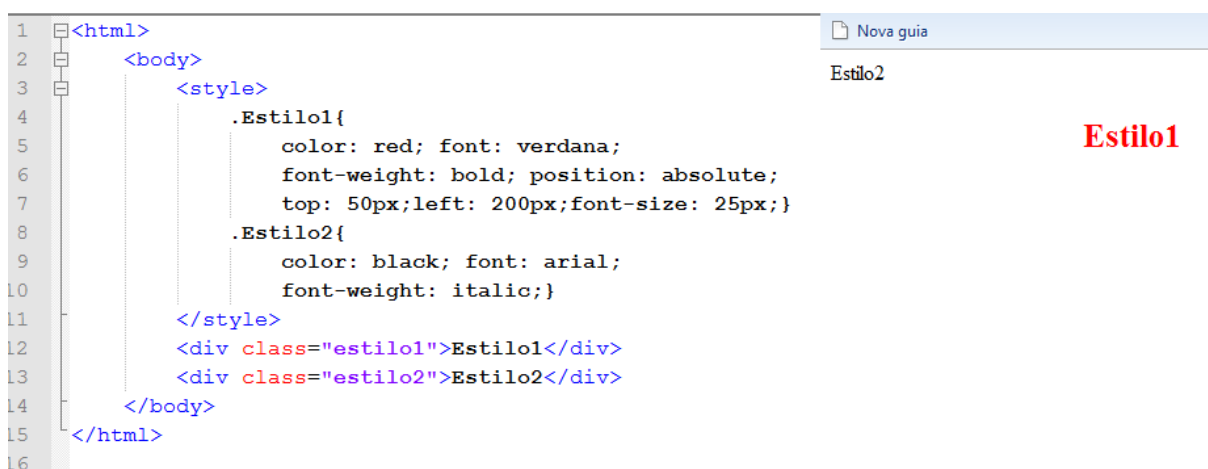
A Figura 8 mostra dois elementos *div* que possuem códigos muito parecidos, mas aparências diferentes (canto direito superior da figura). A diferença se deve ao atributo *class* que indica o estilo que cada elemento herdou. Isso significa que mudando apenas um atributo, pode-se modificar totalmente a renderização de um

objeto. Nesse exemplo os estilos são declarados dentro da página, mas poderiam ser declarados em um arquivo à parte.

O CSS possui mais funcionalidades. É possível, por exemplo, definir estilos para objetos que não possuem o atributo *class* declarado. Isso é feito através de operadores de seleção. Existem operadores que selecionam tipo de objetos (div, span, etc), descendentes (descendentes de uma tabela, de objetos que possuem um dado estilo) etc.

Um grande ganho de usar estilos pré-definidos é que isso torna possível alterar toda a aparência de uma página alterando apenas o atributo *class* de um elemento ou mesmo trocando a folha de estilos.

O tema CSS foi aqui apresentado para introduzir as tecnologias que são usadas em conjunto com o AJAX. Esse tema não será desenvolvido ao longo da dissertação já que o foco são as tecnologias Silverlight e SVG e não o HTML



```
1 <html>
2   <body>
3     <style>
4       .Estilo1{
5         color: red; font: verdana;
6         font-weight: bold; position: absolute;
7         top: 50px;left: 200px;font-size: 25px;}
8       .Estilo2{
9         color: black; font: arial;
10        font-weight: italic;}
11    </style>
12    <div class="estilo1">Estilo1</div>
13    <div class="estilo2">Estilo2</div>
14  </body>
15 </html>
16
```

Estilo2

Estilo1

Figura 8 - Exemplo de CSS: Código HTML à esquerda e página à direita

2.1.2.4 JavaScript

O JavaScript é uma linguagem de scripts voltada para executar operações no lado do cliente. Com ela, é possível fazer validações de campos, abertura de janelas, controle da utilização de botões, mensagens de alertas, confirmações, alterações de estilo. Enfim é possível criar uma interatividade maior do usuário com a página utilizada [17].

A linguagem JavaScript não é compilada, ela é interpretada e é inserida junto com o código fonte HTML das páginas. Apesar de não possuir alguns conceitos importantes de orientação a objeto como definição de classe e herança explícita, possui outros como o conceito de objeto, herança implícita, encapsulamento e o polimorfismo [17].

O JavaScript é considerado um dos fundamentos do AJAX, pois é ele que permite o acesso ao modelo DOM dos objetos que compõem uma página. É através desse acesso que uma página pode ser alterada dinamicamente durante a sua execução. A Figura 7 mostrou como alterar as propriedades de um elemento usando DOM e JavaScript. Nesta figura o conteúdo interno do nó `<script>` representa o JavaScript e o métodos e propriedades dos objetos *document* e *span1* são definidos pelo modelo DOM.

2.1.2.5 XML

O XML ou **eXtensible Markup Language** foi um dos pilares iniciais do AJAX, mais precisamente a última letra, o X [17]. É um padrão W3C [18] e foi criado para permitir transferência de dados entre plataformas diferentes. A primeira versão deste padrão foi aprovada pelo W3C em 1998.

O XML pode ser usado para codificar os dados transferidos entre cliente e servidor. A título de exemplo imagine uma aplicação que possui auto-complete, por exemplo, o mecanismo de busca Google. À medida que se digita a palavra a ser pesquisada, são gerados eventos assíncronos para obter do servidor as buscas mais pertinentes. Um exemplo fictício de requisição-resposta usando XML é mostrado na Figura 9. Nesse exemplo a requisição ou solicitação envia para o servidor os parâmetros da busca: país (*br*), linguagem (*pt-BR*) e texto (*T*). A resposta contém os dados que serão interpretados para preencher a lista de auto-complete, como pode ser visto na Figura 10.

O exemplo dado é fictício, pois na realidade a resposta enviada pelo Google não estava codificada em XML. A resposta foi um JavaScript que exibiu uma lista de auto-complete quando executado (Figura 11).

Cabeçalhos de Solicitação

```
Host clients1.google.com
User-Agent Mozilla/5.0 (Windows; U; Windows NT 6.0; pt-BR; rv:1.9.0.5) Gecko/2008120122 Firefox/3.0.5
Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language pt-br,en-us;q=0.5
Accept-Encoding gzip,deflate
Accept-Charset ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive 300
Connection keep-alive
Referer http://www.google.com.br/
Cookie PREF=ID=c392f6dce236823a:TM=1228072482:LM=1228475327:GM=1:8=d73pI9eUQiHMerHp; rememberme=true; NID=17=dEMKdFTL_ijFqJp9JxpMzdlbLeIsdKsen2BuUVptEP5G2Ho0rSruUUsY2MUz4vEa6U5-zsGq0WLbXkrqQlh68dMBb01kYV2a3mKGKZ4tb6DcMzLIQmIuif3kWgAlLbJ
```

Parâmetros
gl br
hl pt-BR
q T

Cabeçalhos de Resposta

```
Content-Type text/javascript; charset=utf-8
Date Thu, 05 Feb 2009 11:30:31 GMT
Expires Thu, 05 Feb 2009 12:30:31 GMT
Cache-Control public, max-age=3600
Content-Encoding gzip
Server Auto-Completion Server
Content-Length 190
X-Antivirus avast! 4
X-Antivirus-Status Clean
```

Resposta

```
<T>
<result word="tradutor" relevance="5.010.000 resultados" order="0" />
<result word="terra" relevance="174.000.000 resultados" order="0" />
<result word="tre" relevance="168.000.000 resultados" order="0" />
<result word="tam" relevance="256.000.000 resultados" order="0" />
<result word="tse" relevance="19.000.000 resultados" order="0" />
<result word="tim" relevance="299.000.000 resultados" order="0" />
<result word="tradutor online" relevance="516.000 resultados" order="0" />
<result word="telelista" relevance="49.100.000 resultados" order="0" />
<result word="tradutor google" relevance="1.070.000 resultados" order="0" />
</T>
```

Figura 9 - Exemplo fictício de requisição-resposta para o Google [19]



Pesquisa avançada
Preferências
Ferramentas de idiomas

| | |
|-----------------|------------------------|
| tradutor | 5.010.000 resultados |
| terra | 174.000.000 resultados |
| tre | 168.000.000 resultados |
| tam | 256.000.000 resultados |
| tse | 19.000.000 resultados |
| tim | 299.000.000 resultados |
| tradutor online | 516.000 resultados |
| telefonica | 49.100.000 resultados |
| telelista | 150.000 resultados |
| tradutor google | 1.070.000 resultados |

fechar

Figura 10 - Página atualizada com as informações obtidas via Ajax

Resposta

```
window.google.ac.h(["T", [{"tradutor", "5.010.000 resultados", "0"}, {"terra", "174.000.000 resultados", "1"}, {"tre", "168.000.000 resultados", "2"}, {"tam", "256.000.000 resultados", "3"}, {"tse", "19.000.000 resultados", "4"}, {"tim", "299.000.000 resultados", "5"}, {"tradutor online", "516.000 resultados", "6"}, {"telefonica", "49.100.000 resultados", "7"}, {"telelista", "150.000 resultados", "8"}, {"tradutor google", "1.070.000 resultados", "9"}]])
```

Figura 11 - Resposta real enviada pelo Google para o exemplo dado

Nem todas as requisições assíncronas codificam sua resposta em XML, conforme pôde ser constatado. O conteúdo da resposta pode ser qualquer informação de texto. Não há um padrão que deva ser obedecido. O que define qual o formato da resposta é a arquitetura do aplicativo. Quando a resposta chega, é preciso transformá-la em um JavaScript. Não importa como a resposta esteja codificada, apenas é preciso que o mecanismo de tratamento AJAX saiba decodificá-la.

2.1.2.6 XMLHttpRequest

É o objeto que faz a conexão assíncrona entre a página e o servidor de aplicações Web, ou seja, é a tecnologia principal do AJAX. Trata-se de um objeto JavaScript que pode ser usado para fazer requisições ao servidor Web em segundo plano, sem congelar o navegador ou recarregar a página atual [17]. O objeto XMLHttpRequest é hoje parte da especificação do DOM nível 3. Ou seja, qualquer navegador que queira oferecer suporte a esse padrão precisa implementar esse objeto [2].

O XMLHttpRequest foi criado inicialmente no navegador Internet Explorer 5 como um componente do ActiveX. Depois disso, outros navegadores passaram a adotar a idéia. Nessa época o XMLHttpRequest não era um padrão W3C e por isso cada navegador o incorporou de maneiras diferentes.

Atualmente, o Mozilla Firefox, Konqueror, Safari e Opera implementam o XMLHttpRequest como um objeto JavaScript nativo. Já o Internet Explorer prefere continuar usando o ActiveX.

O XMLHttpRequest possui os métodos mostrados na Tabela 1 e as propriedades mostradas na Tabela 2.

| Método | Descrição |
|--|--|
| Open(method:string, url:string, async:booleano, user:string, pass:string) | Define os parâmetros de comunicação com o servidor. O argumento <i>method</i> pode ser GET, POST ou PUT. O endereço da url pode ser relativo ou absoluto. <i>Async</i> indica se a requisição é síncrona ou não. Os 3 últimos parâmetros são opcionais.. |
| Send(content:string) | Envia uma solicitação com o conteúdo <i>content</i> para o Servidor. |
| setRequestHeader(name:string, value:string) | Configura o cabeçalho http especificado com o valor fornecido. |
| getResponseHeader(name:string) | Retorna o valor da string do cabeçalho especificado. |
| getAllResponseHeaders | Retorna uma string com todos os cabeçalhos HTTP especificados. |
| Abort | Interrompe o processamento atual do objeto XMLHttpRequest. |

Tabela 1 - Métodos do objeto XMLHttpRequest [20]

| Propriedades | Descrição |
|----------------------------------|---|
| Status:string | Contém o código de status enviado pelo servidor Web. (isto é, 200 para OK, 404 para Não Encontrado e assim por diante) |
| statusText:string | A versão em texto do código de status http. (isto é, OK ou Not Found e assim por diante) |
| readyState:int | O estado da solicitação. Os cinco valores possíveis são: 0:não inicializada, 1:carregando, 2:carregada, 3:interativa e 4:concluída. |
| responseText:string | A resposta do servidor na forma de uma string. |
| responseXML:document | A resposta do servidor em formato XML. |
| Onreadystatechange:method | O manipulador de eventos que é acionado a cada mudança de estado. É a função de callback chamada quando a requisição retorna. |

Tabela 2 - Propriedades do objeto XMLHttpRequest

2.2 SVG

O SVG é um padrão para descrever figuras bidimensionais e aplicações gráficas em XML [18]. É um padrão aberto e por ser basicamente um arquivo XML pode ser editado por diversos editores de textos simples como o Bloco de Notas. Por serem vetoriais, pode-se dar zoom em gráficos SVG sem perda de qualidade [21]. Diversas empresas, como a Sun Microsystems, Adobe, Apple, IBM, Kodak se envolveram em implementações da especificação do SVG [22].

Essa tecnologia suporta acesso DOM via JavaScript, ou seja, é possível alterar o seu documento dinamicamente. Também possui suporte a eventos. Assim, é possível, por exemplo, fazer com que um quadrado SVG mude de cor ao ser clicado, ou então arrastá-lo na tela através da manipulação dos eventos de clique e movimentação do mouse.

Para visualizar gráficos SVG usando o Internet Explorer, deve-se ter instalado um plugin, tal como o Adobe SVG Viewer [23], que é freeware. O Opera, o Google Chrome e Firefox são navegadores que oferecem suporte nativo a essa tecnologia [24].

```
1  <?xml version="1.0" standalone="no"?>
2  <svg width="290" height="290" version="1.1"
3      xmlns="http://www.w3.org/2000/svg">
4
5      <rect x="20" y="20" rx="20" ry="20" width="250" height="100"
6          style="fill:blue;stroke:black;stroke-width:3;opacity:0.5"/>
7
8      <g stroke="green" stroke-width="4" fill="darkgray">
9
10         <line x1="0" y1="0" x2="280" y2="280"/>
11
12         <circle cx="170" cy="170" r="40"
13             stroke-width="2" fill="darkgray" opacity="0.9"/>
14     </g>
15
16     <text x="170" y="250" style="font:36px verdana bold;">Texto</text>
17 </svg>
```

Figura 12 - Exemplo de documento SVG

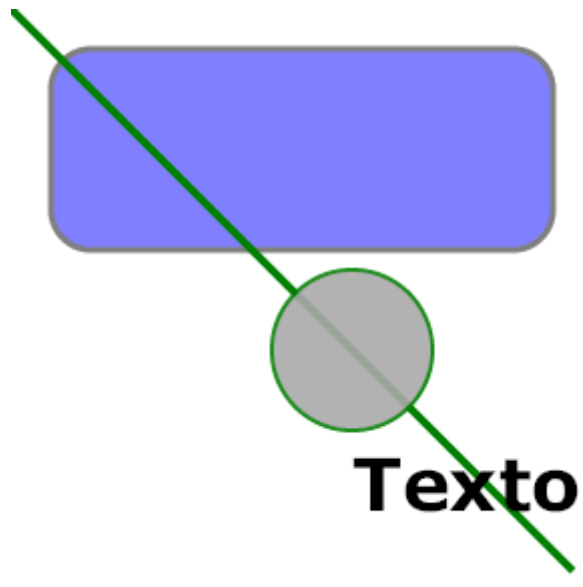


Figura 13 - Visualização do exemplo de documento SVG

Para ilustrar o que é o SVG, considere o exemplo mostrado na Figura 12. Cada nó do código XML desta figura pode ser facilmente identificável. Por exemplo, o nó `<rect>` desenha um retângulo, `<circle>` desenha um círculo, `<g>` agrupa elementos, `<line>` desenha um linha, `<text>` é utilizado para escrever um texto. Tudo isto está dentro no nó raiz `<svg>`, que define o documento. Cada um destes nós possui diversos atributos e propriedades, tais como sua posição na tela (x, y), cor (*fill*), espessura da linha (*stroke-width*), etc. A saída gerada pelo visualizador é mostrada na Figura 13.

2.3 SILVERLIGHT

O Silverlight provê para designers e desenvolvedores uma solução multi-plataforma para aplicações altamente interativas na Web. É uma parte chave da plataforma Web Microsoft da nova geração [25].

Assim como o SVG, ele é também um padrão para descrever gráficos bidimensionais e aplicações gráficas. Um documento Silverlight é definido em XAML (**eXtensible Application Markup Language**). O XAML é um padrão criado pela Microsoft e é um arquivo XML. Além de descrever animações bidimensionais pode – se combiná-las com áudio e vídeo.

O Silverlight é suportado por múltiplos sistemas operacionais (incluindo Macintosh OS X). O seu plugin foi desenvolvido para funcionar com os navegadores Internet Explorer, Mozilla e Safári. Em testes foi possível fazê-lo funcionar com o Google Chrome também.

Essa tecnologia suporta acesso DOM via JavaScript e também é orientado a eventos. Desse modo é possível a integração do Silverlight com AJAX.

Para ilustrar o funcionamento do Silverlight considere o exemplo da Figura 14 e da Figura 15.

```
<Canvas
  xmlns="http://schemas.microsoft.com/client/2007"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Ellipse Height="200" Width="200" Canvas.Left="30" Canvas.Top="30"
    Stroke="Black" StrokeThickness="10" Fill="SlateBlue"/>
  <Rectangle Height="100" Width="100" Canvas.Left="5" Canvas.Top="5"
    Stroke="Black" StrokeThickness="10" Fill="SlateBlue"/>
  <Line X1="200" Y1="10" X2="10" Y2="200"
    Stroke="black" StrokeThickness="5"/>
</Canvas>
```

Figura 14 - Exemplo de documento Silverlight

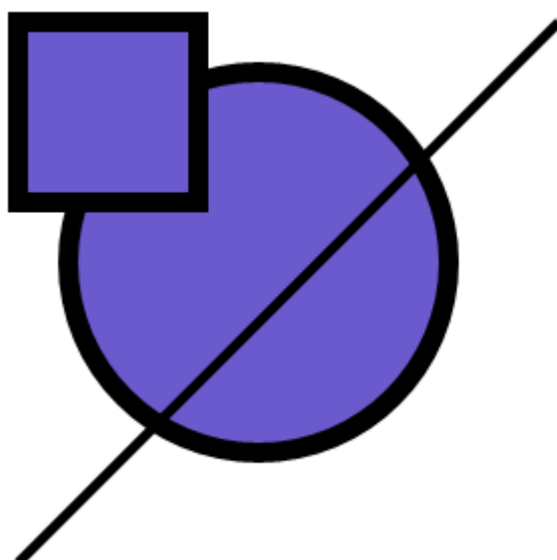


Figura 15 – Visualização do exemplo de documento Silverlight

Cada nó do código XML desta figura pode ser facilmente identificável. Por exemplo, o nó *<Rectangle>* desenha um retângulo, *<Ellipse>* desenha uma elipse, *<Line>* desenha um linha. Tudo isto está dentro do nó raiz *<Canvas>*, que

representa a área de desenho à qual as propriedades *Canvas.Left*, *Canvas.Top*, *X1*, *X2* etc se referem. Cada um destes nós possui diversos atributos e propriedades além das citadas, tais como cor (*Fill*), cor de contorno (*Stroke*) largura do contorno (*StrokeThickness*), etc.

3 ARQUITETURA AJAX

Esse capítulo apresenta conceitos gerais da arquitetura de um aplicativo AJAX.

Conforme descrito na seção 2.1, AJAX é o nome dado ao uso conjunto de várias tecnologias para o desenvolvimento Web. Não é uma tecnologia com uma especificação definida e por esse motivo cada aplicação o implementa de um jeito diferente.

Contudo, mesmo existindo variedades nos detalhes de implementação, existem vários pontos comuns na arquitetura desses aplicativos. Um esquema dessa arquitetura comum [26] pode ser visto na Figura 16.

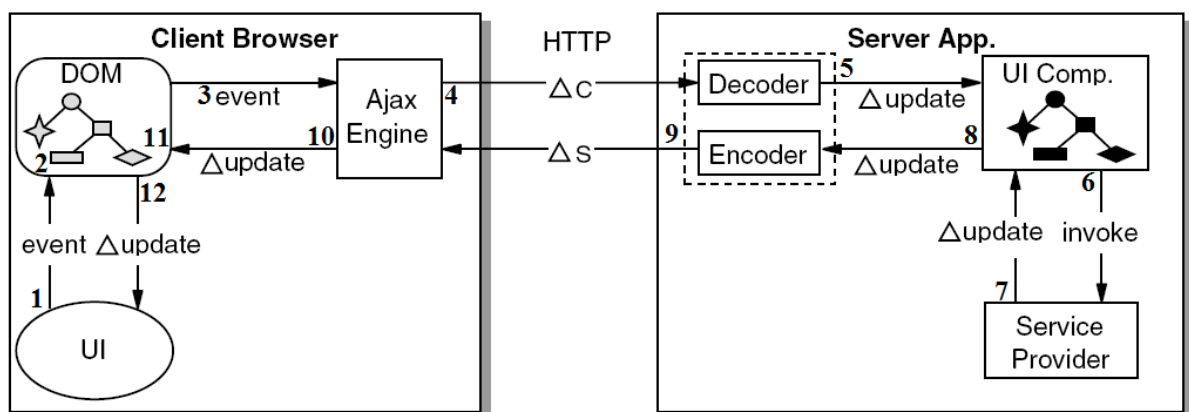


Figura 16 - Macro arquitetura AJAX [26]

A figura mostra a macro arquitetura presente em qualquer aplicativo AJAX, incluindo os componentes necessariamente presentes tanto no cliente quanto no servidor.

O cliente (*client browser*) oferece suporte ao conjunto de padrões usados no AJAX, tais como HTTP, HTML, CSS, JavaScript e DOM. Ele processa o modelo representativo de uma página Web para produzir a interface com o usuário (*UI*). Todos os efeitos visuais são apresentados através dessa interface. No cliente está localizado o mecanismo AJAX. Esse mecanismo é responsável por inicializar e

manipular o modelo representativo da página. Ele lida com os eventos gerados pelo usuário, comunica com o servidor e usa a resposta obtida para gerar modificações em uma página sem precisar recarregá-la.

O servidor (*server app.*) é responsável por receber, processar e responder as requisições geradas pelo cliente. O servidor recebe a requisição, interpreta os dados nela presentes e chama os listeners responsáveis por manipular esses dados. Logo após, ele interpreta os dados manipulados para gerar uma resposta à requisição recebida.

A numeração da Figura 16 mostra o caminho típico de uma requisição assíncrona. A saber:

1. No navegador (*client browser*) é gerado um evento. Esse evento pode ser um clique do usuário, a alteração do valor de uma caixa de texto, um evento cíclico (timer), etc;
2. O evento avisa o que ocorreu aos listeners registrados na interface DOM;
3. A implementação do listener gera uma chamada ao mecanismo AJAX (*AJAX engine*) para que alguma operação seja executada no servidor;
4. O mecanismo AJAX codifica as informações desse evento e também todas as outras informações que porventura sejam necessárias para o servidor e as envia através de uma requisição assíncrona;
5. O servidor (*server app.*) recebe a requisição, decodifica os dados e atualiza o componente responsável por tratar a requisição (*UI Comp.*). Dois tipos de componentes comuns são: representação em objeto da página ou função tipo web service. Dizemos tipo web service, pois não há a obrigatoriedade do uso do protocolo SOAP;
6. O componente responsável por tratar a requisição chama o listener (*service provider*) registrado no servidor para tratar aquele evento. Se o componente for uma representação em objeto da página, chamar o

listener significa chamar a representação do evento gerador da requisição. Se for uma função tipo web service, esse passo significa executar o corpo da função;

7. O listener executa suas operações e pode atualizar o componente. Se o componente for uma representação da página, esse passo significa que o listener alterou a estrutura da página. Se for a função significa que o corpo da função foi executado;
8. O componente registra as atualizações para que sejam enviadas pelo servidor. Se o componente for uma página, esta pode registrar as alterações em sua estrutura, à medida que vão acontecendo. Se for uma função, esse passo não faz sentido;
9. O servidor codifica as atualizações em algum formato pré-acordado com o mecanismo AJAX e as envia no conteúdo da resposta. Se o componente for a página, esse passo significa que o servidor transformou histórico de atualizações em texto. Se for uma função, significa que o retorno da função foi serializado;
10. O mecanismo AJAX recebe a resposta e a interpreta, gerando um script de atualização da página;
11. O script de atualização é executado através da interface DOM;
12. As modificações do DOM são refletidas na visualização da página.

Para entender como uma requisição é feita na prática, voltemos ao exemplo do auto-complete dado na seção 2.1.2.5. Nesse exemplo a interface com o usuário é a página de busca Google [19].

Para efeitos didáticos, suponhamos que a página mostrada na Figura 17 possua o trecho de código mostrado na Figura 18.

A figura mostra um trecho de código fonte possível para o auto-complete do buscador do Google. Não foi mostrado o código real, pois ele está ofuscado (sem endentação e com nome de funções ilegíveis).

As linhas 1 e 2 dessa figura mostram a declaração da caixa de texto usada. As duas funções JavaScript declaradas implementam um mecanismo AJAX simples mas funcional.



Figura 17 - Página inicial do mecanismo de busca Google

```
1 <input value="" title="Pesquisa Google" size="55" name="q" maxlength="2048"
2   onkeypress="obterSugestoesAutoComplete(this.value);" />
3 </script>
4 function obterSugestoesAutoComplete(valorTexto) {
5     var objetoReqAssincrona = null;
6     if (window.XMLHttpRequest) {
7         objetoReqAssincrona = new XMLHttpRequest();
8     }
9     else if (typeof ActiveXObject != "undefined") {
10        objetoReqAssincrona = new ActiveXObject("Microsoft.XMLHTTP");
11    }
12    objetoReqAssincrona.open("POST", obterEndecoServidor(), true);
13    objetoReqAssincrona.setRequestHeader("gl", obterPais());
14    objetoReqAssincrona.setRequestHeader("hl", obterLingua());
15    objetoReqAssincrona.setRequestHeader("q", valorTexto);
16    objetoReqAssincrona.onreadystatechange = function() {
17        recebeResultadoSugestoesAutoComplete(objetoReqAssincrona)
18    };
19    objetoReqAssincrona.send(null);
20 }
21
22 function receberResultadoSugestoesAutoComplete(requisicao) {
23     if ((requisicao.readyState != null) && (requisicao.readyState == 4)) {
24         if (requisicao.status == 200) {
25             eval(requisicao.responseText);
26         }
27         else {
28             alert("Erro na chamada ao servidor: " + requisicao.responseText);
29         }
30         delete requisicao;
31     }
32 }
33 </script>
```

Figura 18 - Trecho de código possível para o buscador do Google

A função *obterSugestoesAutoComplete* cria a requisição assíncrona (linhas de 5 a 11), codifica os dados em texto puro e os coloca no cabeçalho da solicitação (linhas 13, 14, 15), registra uma função *callback* para tratar a resposta (linhas 16, 17 e 18) e envia a solicitação ao servidor (linha 19).

Cabeçalhos de Solicitação

```

Host clients1.google.com
User-Agent Mozilla/5.0 (Windows; U; Windows NT 6.0; pt-BR; rv:1.9.0.5) Gecko/2008120122 Firefox/3.0.5
Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language pt-br,en-us;q=0.5
Accept-Encoding gzip,deflate
Accept-Charset ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive 300
Connection keep-alive
Referer http://www.google.com.br/
Cookie PREF=ID=c392f6dce236823a:TM=1228072482:LM=1228475327:GM=1:S=d73pI9eUQiHMerHp; rememberme=true; NID=17=dEMKdFTL_i_jFqJp9JxpMzdlbLeIsdXsnn2BuUVptEP5G2Ho00rSruUUsY2MUz4vEa6U5-raGq0WLbXkrqQlh68dMBb01kYV2a3mK GKZ4tb6DcMzLIQmIuif3kWgAlLbJ
  
```

Parâmetros

gl br
hl pt-BR
q T

Cabeçalhos de Resposta

```

Content-Type text/javascript; charset=utf-8
Date Thu, 05 Feb 2009 11:30:31 GMT
Expires Thu, 05 Feb 2009 12:30:31 GMT
Cache-Control public, max-age=3600
Content-Encoding gzip
Server Auto-Completion Server
Content-Length 190
X-Antivirus avast! 4
X-Antivirus-Status Clean
  
```

Resposta

```

window.google.ac.h(["T", [{"tradutor", "5.010.000 resultados", "0"}, {"terra", "174.000.000 resultados", "1"}, {"tre", "168.000.000 resultados", "2"}, {"tam", "256.000.000 resultados", "3"}, {"tse", "19.000.000 resultados", "4"}, {"tim", "299.000.000 resultados", "5"}, {"tradutor online", "516.000 resultados", "6"}, {"telefonica", "49.100.000 resultados", "7"}, {"telelista", "150.000 resultados", "8"}, {"tradutor google", "1.070.000 resultados", "9"}]])
  
```

Figura 19 - Requisição e resposta gerada pelo auto-complete do Google



T|

tradutor 5.010.000 resultados

terra 174.000.000 resultados

tre 168.000.000 resultados

tam 256.000.000 resultados

tse 19.000.000 resultados

tim 299.000.000 resultados

tradutor online 516.000 resultados

telefonica 49.100.000 resultados

telelista 150.000 resultados

tradutor google 1.070.000 resultados

[fechar](#)

[Pesquisa avançada](#)

[Preferências](#)

[Ferramentas de idiomas](#)

Figura 20 - Resultado da atualização executada pelo mecanismo AJAX

A função *receberResultadoSugestoesAutoComplete* trata a resposta do servidor, decodificando-a (linha 25) e usando-a para atualizar a página.

A requisição e a resposta do aplicativo podem ser encontradas na Figura 19 e a página após o processamento da solicitação é mostrada na Figura 20. Essa figura mostra o conteúdo da resposta transformado em uma lista de opções.

Fazendo um paralelo com os números vistos na Figura 16 que descreve o caminho típico de uma solicitação AJAX:

1. O usuário digita um caractere ou uma seqüência de caracteres;
2. O listener do evento *keypress* é chamado pelo DOM;
3. A implementação do listener (Figura 18, linha 2) chama o método *obterSugestoesAutoComplete* do mecanismo AJAX.
4. O mecanismo AJAX, através do método *obterSugestoesAutoComplete*, codifica as informações, isto é, coloca os valores do país, da língua e o texto digitado como os parâmetros *gl*, *hl* e *q* respectivamente da solicitação assíncrona (Figura 18, linhas 13, 14, 15);
5. O servidor recebe a requisição, obtém os valores dos parâmetros de país, língua e texto digitado, repassa a chamada ao componente responsável por tratar aquele tipo de solicitação. Suponhamos que o componente seja uma função a ser chamada.
6. O corpo do função é executado.
7. O corpo da função faz a busca desejada em uma base de dados, usando os valores dos parâmetros informados e obtém uma lista contendo palavras, número de resultados e ordem de seqüência.
8. A função retorna a lista como resultado.
9. O servidor codifica a lista no formato de chamada a uma função JavaScript e envia a resposta (Figura 19);

10. O mecanismo AJAX recebe a resposta através do método *receberSugestaoAutoComplete* (Figura 18, linha 21) e decodifica a resposta. Nesse caso a decodificação consiste em simplesmente obter o texto da resposta e executá-lo como um script;
11. O script de atualização é executado (Figura 18, linha 24);
12. A página é atualizada para mostrar a lista com os valores sugeridos (Figura 20).

Agora que os conceitos gerais da arquitetura de um aplicativo AJAX foram abordados, discutiremos como incluir suporte AJAX no servidor e no cliente.

3.1 INCLUSÃO DE SUPORTE AJAX NO CLIENTE

O cliente de um aplicativo AJAX contém três partes (Figura 16): interface com o usuário, interface DOM e mecanismo AJAX. A interface com o usuário se comunica com o mecanismo AJAX através de listeners registrados no DOM, e o mecanismo AJAX, por sua vez, altera os componentes de visualização através de chamadas DOM via JavaScript.

O mecanismo AJAX é responsável tanto por coletar dados do aplicativo cliente e enviá-los ao servidor de maneira assíncrona, quanto por receber a resposta do servidor e convertê-la em informação útil para o usuário. Dessa maneira ele é acessado em dois momentos: no envio de uma requisição assíncrona e no recebimento da resposta à solicitação gerada.

3.1.1 Gerando uma requisição Assíncrona

Quando um usuário interage com o aplicativo e esse último necessita fazer uma requisição assíncrona ao servidor, essa requisição é feita chamando-se o mecanismo AJAX. Um exemplo concreto de como isso é feito foi mostrado na Figura 18.

Contudo, algumas questões ainda não foram respondidas: Como saber quais são os dados relevantes para o servidor? Quais parâmetros devem ser enviados? Qual o significado desses parâmetros?

As respostas dependem da arquitetura do servidor e por isso não há como responder precisamente, mas, mesmo em uma discussão mais abstrata sobre o assunto, é possível perceber que certas informações devem estar sempre presentes em uma requisição assíncrona.

Abstraindo-se da codificação usada para enviar os parâmetros, é necessário que pelo menos dois tipos de parâmetros sejam enviados: parâmetro de identificação da solicitação e parâmetros contendo os dados usados para processar a solicitação.

3.1.1.1 Como enviar parâmetros

Existem duas maneiras de enviar parâmetros: através do cabeçalho e através do conteúdo de uma solicitação. Os dois modos são mostrados na Figura 21.

```
// Parâmetro no cabeçalho
objetoReqAssincrona.setRequestHeader("Parametro1", "Valor1");

// Parâmetro no conteúdo
objetoReqAssincrona.send("<Parametro1>Valor1</Parametro1>")
```

Figura 21 - Envio de parâmetros na requisição assíncrona

Em ambos os casos, o valor do parâmetro é o mesmo, a diferença está no modo de obtê-lo. É necessário que o componente responsável pelo tratamento da solicitação no servidor saiba exatamente como o parâmetro foi enviado para que saiba decodificá-lo.

3.1.1.2 Parâmetro de identificação da solicitação

Quando o servidor recebe uma solicitação, é necessário que ele saiba identificar o componente responsável por respondê-la. Essa identificação pode ser

dividida em duas categorias distintas: identificação através de um endereço específico e identificação através de parâmetros.

A identificação através de um endereço específico consiste em informar ao servidor o endereço de um componente especializado que sabe apenas responder a um tipo de requisição. Esse parece ser o caso do auto-complete do Google mostrado no capítulo 3. O cliente envia apenas três parâmetros, a língua, o país e o texto digitado. Nenhuma informação adicional sobre o estado da página ou sobre o objeto que gerou a requisição. O papel do componente requisitado ao servidor é o de apenas fazer a busca em uma base de dados e codificar a resposta em um formato compreensível para o mecanismo AJAX.

A identificação através de parâmetros é necessária quando o servidor está estruturado para receber vários tipos de solicitações em um único endereço. Neste caso basta enviar um ou mais parâmetros que informem o tipo de solicitação. Pode-se enviar, por exemplo, o nome de uma classe que deve ser instanciada ou algum identificador que será usado em uma cláusula condicional.

3.1.1.3 Parâmetros com dados para o processamento da solicitação

Uma vez identificado o componente responsável por responder a solicitação, é preciso fornecer a ele os dados necessários para que a solicitação seja processada. No exemplo do Google, os dados necessários eram o texto, a língua e o país. Se o aplicativo representasse um sinótico com o volume de tanques, seria necessário enviar o identificador dos tanques como parâmetros e por aí segue. Os parâmetros devem ser sempre adequados ao tipo de resposta que se deseja obter. Parâmetros que poderiam ser usados para identificar um evento DOM: identificador do documento, identificador do elemento e evento gerador da chamada.

3.1.2 Tratamento da resposta de uma chamada assíncrona

O objeto usado para se fazer uma requisição assíncrona é do tipo XMLHttpRequest descrito em 2.1.2.6. Esse objeto permite que seja registrada uma função de *callback* para que o tratamento da resposta seja realizado.

O objeto XMLHttpRequest possui uma propriedade chamada *onreadystatechange* que recebe um método sem parâmetros. Esse método será chamado sempre que o estado da requisição for alterado e pode ser usado para tratar a resposta a uma requisição assíncrona. Para mais informações sobre a interface do objeto XMLHttpRequest consulte a Tabela 1 e a Tabela 2.

```
1 <script>
2 function fazRequisicaoAssincrona() {
3     ...
4     objetoReqAssincrona.onreadystatechange = function() {
5         trataRespostaRequisicaoAssincrona(objetoReqAssincrona)
6     };
7     objetoReqAssincrona.send(obterParametros());
8 }
9
10 function trataRespostaRequisicaoAssincrona(objetoReqAssincrona) {
11     if ((objetoReqAssincrona.readyState != null)
12         && (objetoReqAssincrona.readyState == 4)) {
13         if (objetoReqAssincrona.status == 200) {
14             eval(objetoReqAssincrona.responseText);
15         }
16         else {
17             alert("Erro na chamada ao servidor: " +
18                 objetoReqAssincrona.responseText);
19         }
20         delete objetoReqAssincrona;
21     }
22 }
23 </script>
```

Figura 22 - Registro de *callback* para uma chamada assíncrona

A Figura 22 mostra como registrar a função *callback*. As linhas 4, 5 e 6 dessa figura mostram o registro da função *callback*. Neste exemplo foi usada uma função anônima como *callback*. Isso foi feito para possibilitar passar o objeto usado na requisição como parâmetro. Esse objeto é necessário para se obter o conteúdo da resposta, conforme visto na linha 18. Um ponto de atenção é que a função *callback* não é chamada somente quando a resposta à requisição está disponível. Por isso é necessário checar se a solicitação foi concluída (*readyState=4*).

O elemento mais importante de uma resposta assíncrona é o conteúdo. É a partir desse conteúdo que a página será atualizada. Este conteúdo pode estar

codificado como texto puro, XML, JSON (**JavaScript Object Notation**), etc, e cada tipo de codificação tem seus prós e contras. A codificação influencia na legibilidade da resposta, no tempo de processamento para decodificá-la e no seu tamanho. Cada aplicativo define, de acordo com as suas necessidades, o padrão que será usado. O essencial é converter os dados presentes na resposta em informação inteligível para o mecanismo AJAX. O papel do mecanismo AJAX passa a ser então o de converter essas informações em instruções de comando que afetarão o estado da página.

Existem vários tipos de instruções de comando, tais como alterar o valor de variáveis, criar funções, etc. Um tipo particularmente relevante de instruções para o SVG e para o Silverlight é a instrução DOM. É somente a partir desse tipo de instrução que a estrutura de um documento SVG ou Silverlight pode ser alterada. Qualquer comando que objetive mudanças nesse tipo de documento deve ser convertido em uma instrução DOM.

As instruções DOM são as mesmas em todos os aplicativos. Elas podem variar de navegador para navegador ou de plugin para plugin, mas são independentes da arquitetura do aplicativo AJAX. São definidas pela especificação de cada tecnologia.

3.2 COMANDOS DOM

O DOM define quatro interfaces principais: *Node*, *Element Document*, *EventListener* e *EventTarget*, através das quais é possível efetuar praticamente todo o tipo de alteração em um documento XML ou HTML.

A Tabela 3, a Tabela 4, a Tabela 5 e a Tabela 6 mostram as interfaces mais importantes dentro do DOM. As interfaces não são mostradas integralmente. Propriedades somente de leitura, constantes e métodos redundantes foram retirados. A descrição completa das interfaces e dos métodos pode ser encontrada em [27] e [28].

| Método | Descrição |
|--------------------------------|---|
| handleEvent(evt: Event) | É chamado toda vez que ocorre o evento para o qual esse <i>event listener</i> foi registrado. O parâmetro <i>evt</i> contém as informações relevantes de cada evento. |

Tabela 3 - Interface *EventListener*

| Método | Descrição |
|---|---|
| parentNode: Node | Nó pai de um nó |
| childNodes: NodeList | Lista de filhos de um nó |
| ownerDocument: Document | Documento ao qual o nó pertence |
| insertBefore(newChild: Node, refChild: Node) | Insere o nó <i>newChild</i> antes do nó <i>refChild</i> |
| removeChild(oldChild:Node) | Remove o nó filho <i>oldChild</i> |
| appendChild(newChild: Node) | Adiciona o nó <i>newChild</i> ao fim da lista de nós filhos |

Tabela 4- Interface *Node*

| Método | Descrição |
|--|--|
| tagName: string | Tag (nome) de um nó |
| getAttribute(name: string): string | Obtém o valor do atributo <i>name</i> |
| setAttribute(name: string, value: string) | Altera o valor do atributo <i>name</i> para <i>value</i> |
| removeAttribute(name: string) | Remove o atributo <i>name</i> |

Tabela 5 - Interface *Element*

| Método | Descrição |
|---|---|
| addEventListener (type: string, listener: EventListener, useCapture: boolean) | Adiciona um event listener ao evento do tipo type (mouseover, click, etc.). O parâmetro useCapture indica que o usuário deseja iniciar captura. Após iniciar captura, todos os eventos daquele tipo serão enviados para o <i>listener</i> registrado antes de serem despachados para qualquer <i>EventTargets</i> abaixo deles na árvore. Eventos que estão sendo repassados para cima ao longo da árvore não chamarão um <i>EventListener</i> designado para usar captura. |
| removeEventListener (type: string, listener: EventListener, useCapture: boolean) | Remove um <i>event listener</i> previamente registrado ao evento do tipo type. O parâmetro useCapture deve ser o mesmo que informado para registrar o evento que se deseja remover. |

Tabela 6 - Interface EventTarget

A hierarquia dessas interfaces pode ser vista na Figura 23 e é a seguinte: *Element* herda de *Node* e *Document* herda de *Element*. A maioria dos elementos em um documento DOM implementam a interface *Element*. Os elementos que possuem eventos também implementam a interface *EventTarget*. A interface *EventListener* é implementada por qualquer função JavaScript que possua somente um parâmetro.

A partir da análise das interfaces anteriores é possível catalogar as instruções primárias de um documento DOM. O termo instrução primária está sendo usado aqui em um sentido análogo ao de cores primárias: instruções que quando usadas em conjunto podem produzir qualquer tipo de alteração na estrutura de um documento.

Conforme se pode perceber pelas interfaces, as instruções se encaixam nas seguintes categorias:

- Navegação da estrutura DOM através das propriedades nó pai, nós filhos e documento e obtenção de um elemento a partir de seu identificador utilizando o método *getElementById*;
- Manipulação de atributos através dos métodos *getAttribute*, *setAttribute*, e *removeAttribute*;
- Manipulação de eventos através dos métodos *addEventListener* e *removeEventListener*;
- Inserção e posicionamento hierárquico de nós através dos métodos *appendChild*, *insertBefore*, e *removeChild*;
- Inclusão de novos elementos em um documento através dos métodos *createElement* e *createTextNode*.

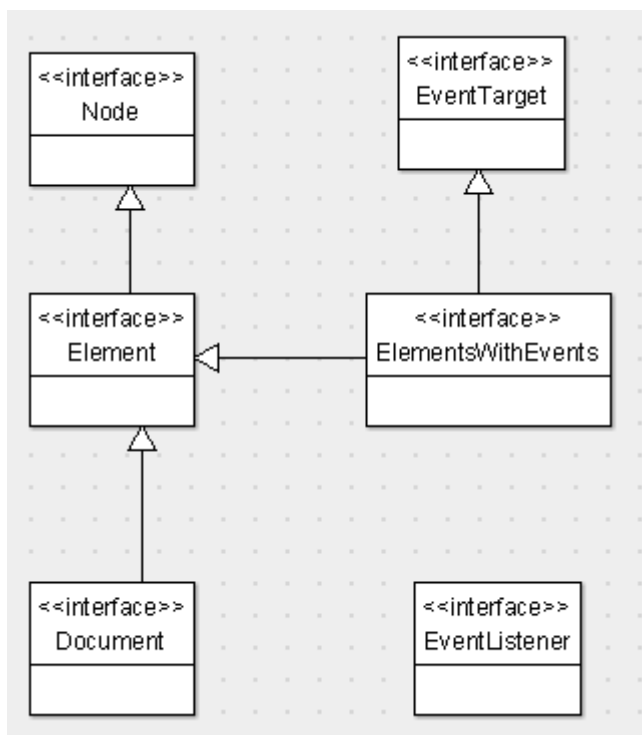


Figura 23 - Hierarquia das interfaces principais do DOM

A partir do uso conjunto dessas instruções primárias é que o mecanismo AJAX irá alterar um documento. Esse mecanismo decodificará as informações contidas na resposta enviada pelo servidor, e as transformará em chamadas DOM.

A estrutura DOM citada é aquela definida pelo W3C [15]. Nem todos os navegadores ou plugins seguem à risca essa definição, mas na prática as funcionalidades permanecem. Isso ocorre, pois todas as definições DOM buscam criar um modelo de objetos para representar a mesma estrutura.

A interface DOM do SVG é definida pelo W3C e, portanto contempla todos os métodos descritos na seção anterior. Entretanto, mesmo que a interface seja única, o efeito da chamada de algumas funções pode ser diverso dependendo do visualizador e por isso é importante pensar em implementações robustas que contornem esses entraves.

A interface DOM do Silverlight [29] é definida pela Microsoft e não segue o padrão definido pelo W3C, embora possua funções equivalentes àquelas citadas anteriormente. Por se tratar de uma tecnologia patenteada pela Microsoft essa interface DOM é única, isto é, não há diferença na sintaxe mesmo que o navegador usado varie.

A inclusão de suporte AJAX no cliente a Silverlight e ao SVG, consiste em adaptar o mecanismo AJAX para que ele também consiga gerar chamadas às funções DOM de cada uma dessas tecnologias.

3.3 INCLUSÃO DE SUPORTE AJAX NO SERVIDOR

Atualmente os aplicativos AJAX estão distribuídos por toda a Web. Vários ambientes de servidor já incorporaram esse paradigma ao seu fluxo de funcionamento. Cada um desses ambientes apresenta seus detalhes, suas convenções e limitações. São inúmeras implementações diferentes de AJAX na parte do servidor, porém podem-se reconhecer famílias de implementações dentro dessa variedade. Há possivelmente três maneiras principais de suporte AJAX em um servidor de aplicativo, discutidas a seguir.

3.3.1 Codificação manual da resposta a uma requisição assíncrona

Consiste na codificação manual do conteúdo da resposta usando diretamente os objetos que representam a requisição recebida e a resposta a ser enviada.

É um tipo de codificação que pode ser encontrado em vários componentes auto-contidos disponíveis gratuitamente pela Internet. Basta configurar o endereço da requisição nesses componentes e escrever uma resposta no formato especificado pela documentação.

O exemplo dado no capítulo 3 de auto-complete do Google parece se encaixar nessa categoria. Um exemplo de código do servidor, escrito em Asp.Net que gera a resposta dada para esse exemplo pode ser encontrado na Figura 24.

```
1 using System.Web;
2
3 public sealed class GoogleAutoComplete : IHttpHandler
4 {
5     public bool IsReusable { get { return true; } }
6
7     public void ProcessRequest(HttpContext context) {
8         string country = context.Request.Headers["gl"];
9         string language = context.Request.Headers["hl"];
10        string typedText = context.Request.Headers["q"];
11
12        AutoCompleteResult[] results = GetAutoCompleteResults(country, language, typedText);
13        if (results != null && results.Length > 0) {
14            context.Response.Write("window.google.ac.h[" + typedText + "],[");
15            for (int i = 0; i < results.Length; ++i) {
16                if (i != 0) context.Response.Write(", ");
17                context.Response.Write("'" + results[i].completedText +
18                    "'," + results[i].numberOfResults + ",'" + i + "']");
19            }
20            context.Response.Write("]");
21        }
22        context.Response.Flush();
23        context.Response.Close();
24
25    }
26
27    private AutoCompleteResult[] GetAutoCompleteResults(string country,
28        string language, string typedText) {
29        ...
30    }
31 }
32
33 public class AutoCompleteResult {
34     public string completedText;
35     public string numberOfResults;
36 }
```

Figura 24 - Codificação manual da resposta a uma solicitação AJAX

O importante a se perceber na figura é a manipulação direta dos objetos da requisição (*request*) e da resposta (*response*). As linhas 8, 9 e 10 mostram como os parâmetros podem ser obtidos. A linha 12 mostra a chamada ao método que executa a lógica de negócio e retorna os resultados desejados. A interface do método é mostrada na linha 27 e a declaração da classe *AutoCompleteResult* pode

ser vista nas linhas 33, 34 e 35. As linhas de 13 a 22 mostram como a resposta é escrita diretamente no objeto *response*. O formato do conteúdo escrito será o mesmo visto na Figura 19. O código mostrado não é código real da aplicação do Google, mas é capaz de gerar o mesmo resultado.

Esse tipo de comunicação AJAX é o mais simples de se implementar. É rápido, pois tanto a requisição como a resposta não carregam dados inúteis. O tratamento no servidor também é mais rápido, pois não exige a criação de uma estrutura complexa para efetuar o tratamento da solicitação. Em contrapartida essa abordagem exige um conhecimento avançado de JavaScript pois, o mecanismo AJAX usado deve ser projetado para tratar cada caso. Há também pouco reaproveitamento de código já que cada tipo de solicitação exige tratamento especial.

A adaptação desse tipo de comunicação AJAX para outras tecnologias DOM não necessita de modificações no servidor. Basta a inclusão no cliente dos scripts de atualização necessários. O exemplo do auto-complete mostra isso. Não é o servidor que decide se o que vai ser alterado é uma caixa de texto, mas sim a implementação, no cliente, da função invocada pelo servidor. Se ao invés de alterar a caixa de texto, essa função alterasse um documento Silverlight, isso seria indiferente para o servidor.

3.3.2 Publicação Assíncrona no cliente de funções do servidor

Essa é uma abordagem comum em *frameworks* baseados em RPC (*remote procedure calls*). O SAJAX [30], DWR [31] e GWT [32] são exemplos dessa abordagem.

Essa abordagem consiste na publicação, no cliente, de métodos existentes no servidor. Se esses métodos forem publicados assincronamente serão métodos AJAX.

O fluxo de uma chamada AJAX feita dessa maneira é o seguinte:

1. Um método publicado no cliente é chamado de maneira assíncrona. São informados para esse método uma função *callback* e alguns parâmetros.

2. O método chamado se comunica com o mecanismo AJAX, informando o seu nome, além da função *callback* e dos parâmetros recebidos.
3. O mecanismo AJAX serializa esses parâmetros e os envia ao servidor.
4. O servidor decodifica as informações recebidas e executa o método especificado com os parâmetros informados.
5. O servidor serializa o retorno do método chamado e o envia para o cliente.
6. O mecanismo AJAX recebe essa resposta e converte-a em objetos JavaScript que reflitam a estrutura desses mesmos objetos no servidor.
7. O mecanismo AJAX invoca a função *callback* da função que gerou a requisição AJAX, informando a resposta enviada pelo servidor.

Fazendo um paralelo com o exemplo dado anteriormente, é como se a função *GetAutoCompleteResults*, mostrada na Figura 24, fosse publicada diretamente na página via JavaScript. O resultado de uma chamada a essa função seria um vetor de objetos JavaScript cuja estrutura seria a mesma da classe *AutoCompleteResult*.

A ênfase nesse tipo de comunicação é sobre os dados e não sobre conteúdo de uma página. Dessa maneira, não seria necessário incluir modificações no servidor para adaptar um aplicativo dessa natureza ao Silverlight ou ao SVG. Bastaria consumir os dados de uma chamada assíncrona e convertê-los em instruções DOM que alterassem os documentos Silverlight e/ou SVG.

3.3.3 Inclusão de suporte AJAX em arquiteturas de servidor MVC

Vários *frameworks* AJAX implementam essa arquitetura, como exemplos podemos citar o ECHO2 [33], Java Server Faces [34], o Struts [35] e o Asp.net [36].

O Modelo-Interface-Controlador (*Model-View-Controller* ou MVC) é um padrão de projeto usado para descrever uma boa separação entre a parte de um programa que interage com o usuário e a parte que faz o trabalho pesado, processamento numérico ou outro “objetivo de negócios” do aplicativo.

O padrão de projeto MVC identifica três papéis que um componente do sistema pode desempenhar. O Modelo (*Model*) é a representação do domínio do problema, aquilo com que ele trabalha. Um servidor Web trabalha com o fornecimento de respostas a solicitações, e por isso sua modelagem geralmente é feita com base no formato da resposta. A solicitação feita a um servidor Web dessa natureza pode ser síncrona ou assíncrona. A resposta a solicitações síncronas é feita no formato HTML, e a resposta a solicitações assíncronas pode ser feita em qualquer formato que possa ser convertido em um JavaScript de atualização.

A Interface (*View*) é a parte que o programa apresenta ao usuário. O usuário para um servidor Web é a requisição. O que é apresentado para essa requisição é uma resposta. Dessa forma a Interface pode ser entendida como essa resposta e o código que a gera.

A regra de ouro do MVC é que a Interface e o Modelo não devem conversar. Atuar como intermediário nessa comunicação é papel do Controlador. A Interface informa ao Controlador sobre as interações do usuário. O Controlador, então, manipula o Modelo e decide se as alterações no Modelo requerem uma atualização da Interface. Se requerem, informa à Interface como alterar a si mesma.

A vantagem dessa arquitetura é o Modelo e a Interface permanecem fracamente acoplados.

A implementação de um servidor com base nesse padrão de projeto pode variar, mas em linhas gerais pode ser descrito como:

1. A Interface informa ao controlador que uma requisição foi recebida.
2. O Controlador atualiza o modelo com base nas informações enviadas na solicitação.
3. O Controlador invoca no modelo os eventos de inicialização registrados
4. O Controlador invoca no modelo o evento que gerou a requisição

5. Os eventos invocados podem gerar atualizações na interface e com base nessas alterações o Controlador decide o conteúdo da resposta (Interface).
6. Uma vez construída a resposta, ela é enviada para o cliente.

A desvantagem desse modelo de servidor é que ele introduz um custo adicional de processamento e memória para encapsular a complexidade do tratamento de uma requisição AJAX. A vantagem é que ele possibilita a criação de páginas web em uma linguagem de alto nível, diminuindo assim a complexidade do desenvolvimento. A arquitetura de um *framework* expansível proposta neste trabalho foi projetada com base nesse modelo.

4 PROPOSIÇÃO DE UMA ARQUITETURA AJAX EXPANSÍVEL

O objetivo deste capítulo é propor uma arquitetura expansível para um *framework* AJAX. O principal objetivo dessa arquitetura é possibilitar que novas tecnologias baseadas em XML possam ser facilmente *plugadas* nesse *framework*. Para alcançar esse objetivo a arquitetura foi modelada para conseguir trabalhar com o modelo DOM proposto pelo W3C.

4.1 ARQUITETURA MVC DO SERVIDOR

A codificação manual da resposta (3.3.1) não é uma arquitetura apropriada para se criar um *framework*, de vez que é pouco reaproveitável. Por isso essa arquitetura não foi escolhida.

A publicação de chamadas assíncronas (3.3.2) geralmente é usada em conjunto com outros *frameworks*, já que sua ênfase não é na interface, mas sim nos dados. Por isso não faz sentido incluir suporte a Silverlight e SVG em um *framework* baseado nesse conceito.

A arquitetura MVC (3.3.3) é usada para gerar interfaces com o usuário e o SVG e o Silverlight são tecnologias de interface com o usuário. Por essa razão a arquitetura MVC foi escolhida.

No padrão MVC o servidor contém três partes, o Modelo, a Interface e o Controlador. Para entender o papel de cada uma dessas partes, voltemos à macro arquitetura de um servidor AJAX (Figura 25). Um servidor deve ser capaz de:

1. Receber uma requisição;
2. Decodificar seus parâmetros;
3. Atualizar o modelo do servidor com base nos parâmetros recebidos;
4. Executar o código alvo da requisição neste modelo (um evento, por exemplo);

5. Criar uma resposta que faça refletir no cliente o novo estado do modelo no servidor;
6. Enviar para o cliente a resposta.

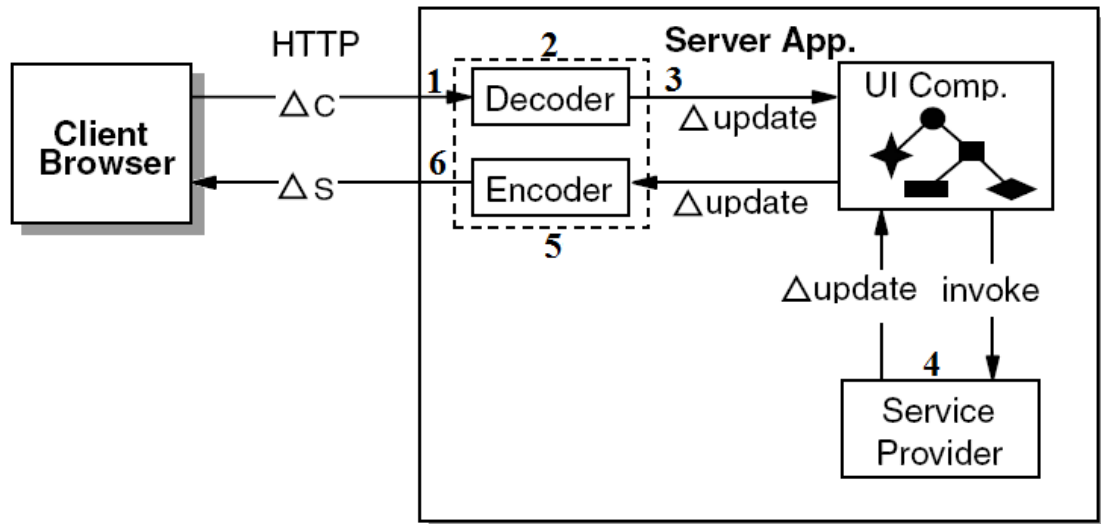


Figura 25 - Macro arquitetura AJAX [26]

Na arquitetura proposta as atribuições do Modelo, Interface e Controlador estão assim divididas:

- Interface: executa os passos 1 e 6.
- Controlador: executa os passos 2, 3 e 5.
- Modelo: executa o passo 4.

Os próximos itens mostram com mais detalhes cada um desses componentes da arquitetura.

4.1.1 Interface

A Interface é implementada em sua maior parte pelo servidor de aplicação usado. O seu papel consiste em transformar os dados recebidos da requisição em um objeto a ser usado pelo sistema, e em transformar os dados gerados pelo sistema em uma resposta ao cliente, trabalhando diretamente com o protocolo HTTP.

A parte que lida com o protocolo HTTP é implementada pelo servidor de aplicação. A parte restante é apenas uma ponte entre o servidor de aplicação e o controlador.

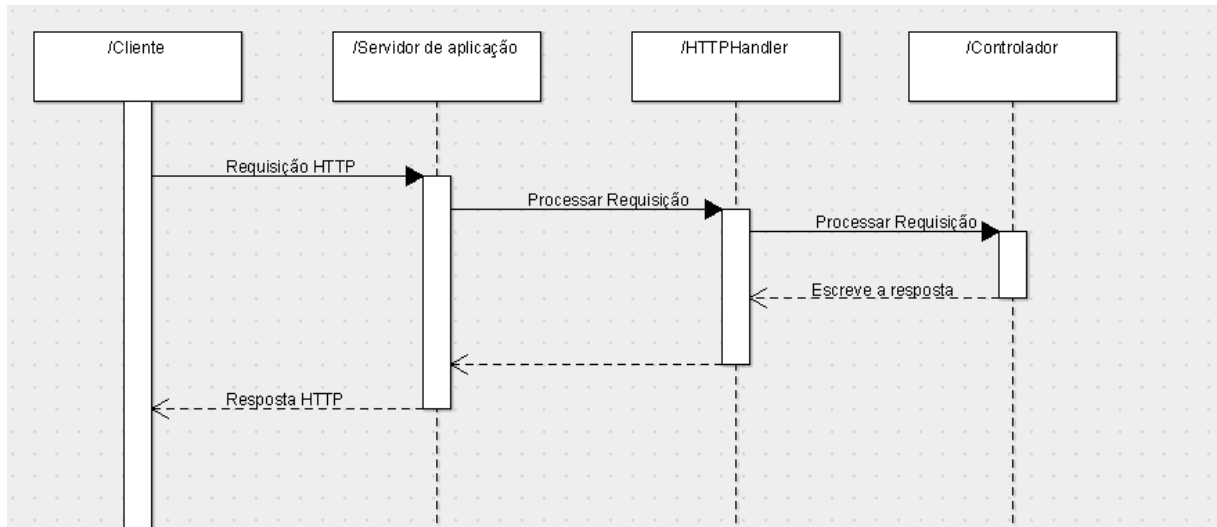


Figura 26 - Diagrama de seqüência da Interface

A Figura 26 mostra o diagrama de seqüência da recepção de uma requisição. A interface está representada pelo *Servidor de aplicação* e pelo *HTTPHandler*. O servidor de aplicação recebe uma requisição, monta os objetos que representarão a requisição e a resposta e os repassa ao *HTTPHandler*. O *HTTPHandler* verifica qual controlador deve ser chamado para tratar aquela requisição e invoca o controlador correspondente. O controlador passa a ser, então, o responsável por atualizar o modelo e escrever a resposta novamente. Neste caso a interface do *HTTPHandler* contém apenas o método de processar a requisição. O diagrama de atividades correspondente pode ser visto na Figura 27.

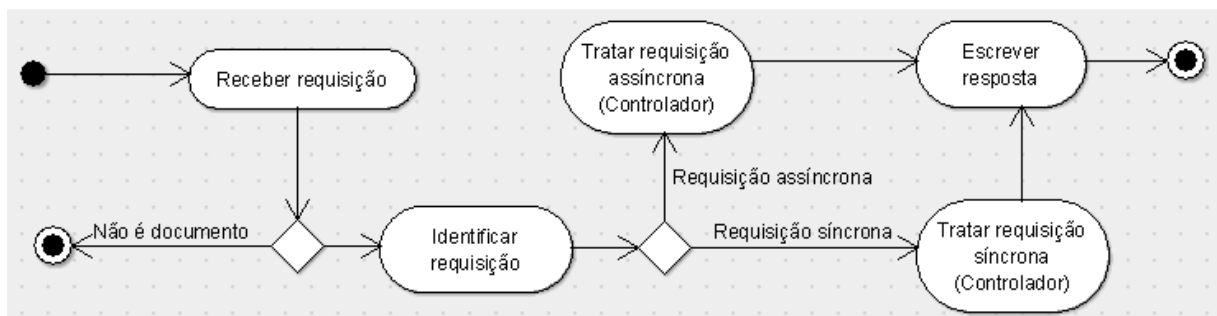


Figura 27 - Diagrama de atividades da interface

4.1.2 Controlador

O papel do Controlador na arquitetura apresentada é o de agir como intermediário entre o servidor de aplicação e o Modelo. Ele decodifica a requisição, atualiza o Modelo com base nesses dados e envia para o cliente uma resposta que o faça refletir o estado final do modelo.

A Figura 27 mostrou que em nossa arquitetura existem dois controladores: um síncrono e um assíncrono. Esses dois controladores servem justamente para tratar os casos síncrono e assíncrono de requisição. Essa separação foi feita, pois o tratamento de cada uma das requisições é diferente o suficiente para merecer tratamento diferenciado.

A seguir cada um desses controladores é detalhado. Mas antes apresentamos os pontos comuns entre eles.

4.1.2.1 Pontos comuns entre os dois controladores

O primeiro ponto comum entre os dois controladores é a interface. O Controlador em uma arquitetura MVC é acessado tanto pela Interface quanto pelo Modelo. Em nosso caso, esse acesso ocorre em dois momentos.

1. Quando uma requisição é recebida ele é acessado pela interface para tratá-la.
2. Quando ocorrem modificações no modelo, este avisa ao controlador para que essas modificações sejam incluídas na resposta.

Quando o controlador é acionado pela interface ele precisa obter o objeto no servidor que representa aquela página (modelo). Duas abordagens são comuns nesse caso.

A primeira é criar um objeto novo e usar os dados da requisição para alterá-lo de modo a reproduzir o estado desse objeto no cliente. Nessa abordagem devem ser sempre enviados para o servidor dados que representem o estado de todos os objetos da página, tornando o tráfego mais intenso. Na prática isso indica que nem

todas as informações são persistidas entre uma requisição e outra. Essa arquitetura é mais comum em sistemas legados, que foram concebidos para trabalhar com aplicativos clássicos e posteriormente foram alterados para incluir suporte a AJAX.

Outra abordagem é manter o objeto que representa a página na sessão. As únicas informações que precisam ser enviadas para o servidor nesse caso são o identificador da página e os dados alterados no cliente desde a última requisição. A implementação fica mais complexa nesse caso, pois é necessário fazer um controle a mais que é o da alteração de dados no cliente. Também há um consumo maior de memória. Em nossa arquitetura usaremos essa abordagem por considerarmos que as vantagens compensam a maior complexidade e o consumo extra de memória.

O segundo item da enumeração fala sobre monitoramento de alterações. Os tipos de modificações monitorados são aqueles que vão gerar mudança no estado atual da página que se encontra no cliente. Analisando-se a interface DOM e fazendo-se testes chegou-se a conclusão que o monitoramento das seguintes operações básicas é suficiente para reproduzir no cliente todas as modificações ocorridas no servidor:

- Atualização de atributos: propriedades como o valor de uma caixa de texto, se um botão está habilitado ou desabilitado, etc. são modeladas como atributos;
- Adição e remoção de filhos: a partir do monitoramento da adição e da remoção de filhos é possível efetuar qualquer movimentação de elementos em um documento web. A adição de um filho que ainda não existe no cliente demanda a criação deste primeiramente;
- Adição e remoção de eventos: os eventos servem para criar páginas mais interativas com o usuário. Na maior parte das vezes é através destes eventos que as requisições AJAX são geradas;
- Atualização do valor interno de um nó. O valor interno de um nó geralmente é usado para a exibição de textos em uma página. O DOM modela o valor interno como um nó texto e não como uma propriedade.

Isso é feito para modelar o caso em um mesmo elemento possui filhos texto e filhos elementos. Vamos desconsiderar, por simplificação, essa possibilidade em nossa arquitetura, pois é possível obter o mesmo efeito visual trocando um nó de texto por um elemento cuja única finalidade é conter aquele texto. Isso significa que em nossa arquitetura um nó ou terá filhos ou terá texto interno.

A interface do controlador pode ser vista na Figura 28:

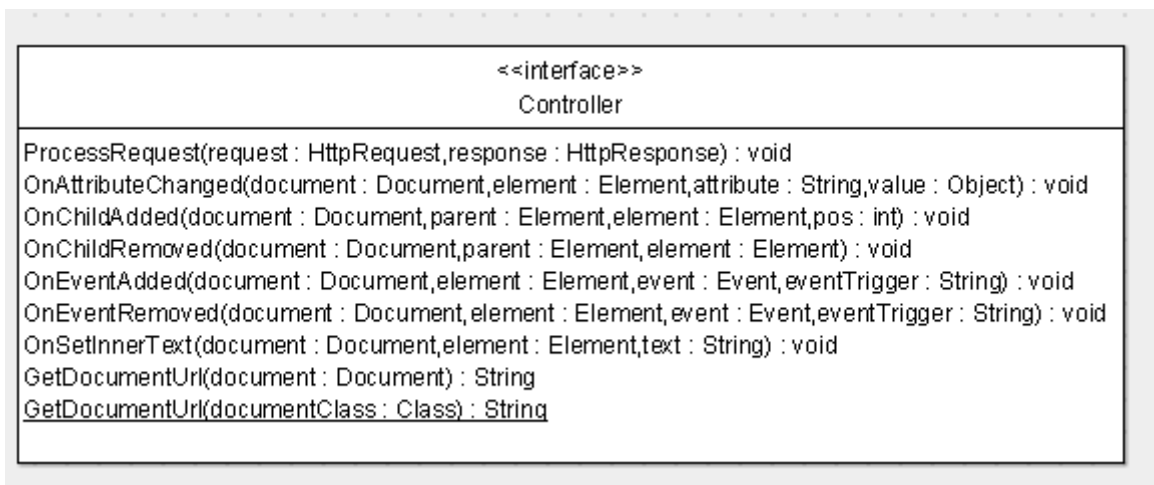


Figura 28 - Interface do Controlador

O método *ProcessRequest* é chamado pela interface quando uma requisição é recebida. Os métodos *On...* são chamados pelo modelo à medida que este vai sendo alterado. *OnAttributeChanged*, por exemplo, é chamado pelo documento *document* quando o elemento *element* teve o valor do seu atributo *attribute* alterado para *value*. Os métodos *GetDocumentUrl* são utilitários e são usados para criar urls a partir de documentos.

Mais considerações sobre a implementação desses métodos serão feitas na descrição de cada um dos tipos controladores. Existem métodos declarados nessa interface que só fazem sentido para um dos dois casos, mas colocá-los na interface comum e usar polimorfismo pode melhorar consideravelmente a organização do código das classes de infra-estrutura.

4.1.2.2 Controlador síncrono

O controlador síncrono é acionado pela interface sempre que uma requisição síncrona é recebida. Esse tipo de requisição pode indicar um evento de uma página existente ou o pedido de um documento que ainda não esteja sendo exibido. A diferenciação dos dois casos é feita através do envio de um parâmetro. A resposta a uma requisição síncrona é sempre o código fonte do documento inteiro.

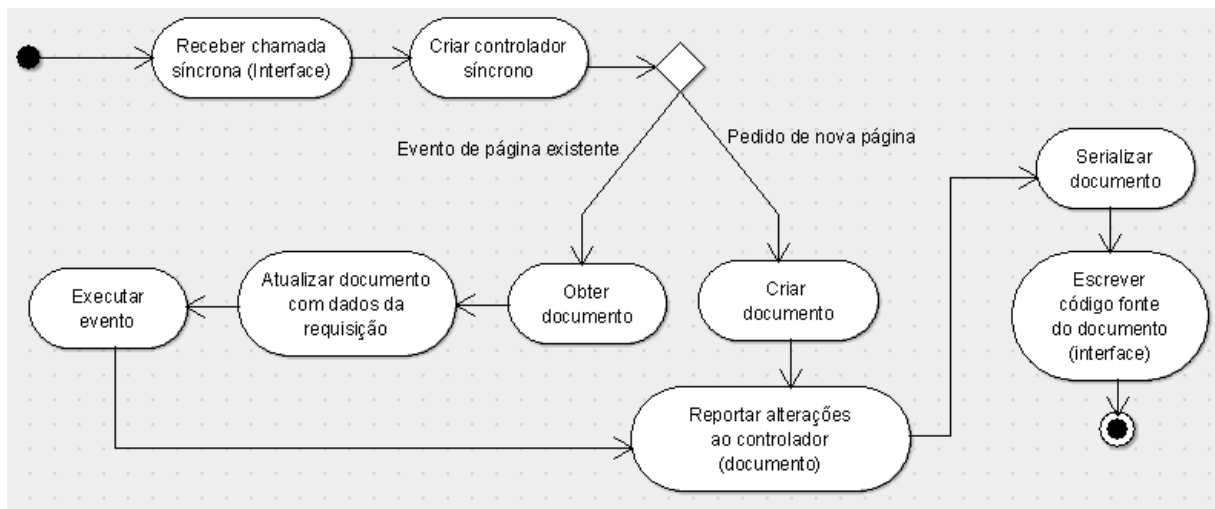


Figura 29 - Diagrama de atividades do controlador síncrono

O diagrama de atividades da Figura 29 mostra o fluxo do controlador síncrono. Quando uma chamada síncrona é recebida o controlador verifica se a requisição é oriunda de um evento de alguma página do cliente. Se for, ele obtém a página armazenada na sessão e atualiza seus dados com base no conteúdo da requisição. Logo após, ele pede ao modelo que execute o evento gerador da solicitação. O identificador desse evento foi enviado como um parâmetro pela solicitação. Durante a execução desse evento, a página pode ser modificada. Nesse caso, o modelo aciona o controlador informando cada uma das alterações ocorridas (chamando os métodos *On...*). O controlador síncrono apenas considera as chamadas a *OnChildAdded* e *OnChildRemoved* para chamar os eventos de servidor correspondentes nos elementos adicionados ou removidos. Após a execução do evento, o controlador converte o Modelo em código fonte e envia esse texto através da interface.

Quando a requisição é um pedido de um novo documento, o controlador analisa o endereço da requisição e cria o objeto correspondente. Se for criado um objeto novo, este irá informar ao controlador as modificações em sua estrutura. O controlador apenas considerará as informações relativas à adição e remoção de filhos para chamar os eventos de servidor correspondentes. Após a construção, o controlador converte esse objeto em código fonte para escrevê-lo na resposta.

4.1.2.3 Controlador assíncrono

O controlador assíncrono só é usado para responder eventos assíncronos e não há um formato definido para as respostas de uma solicitação dessa natureza. Por simplicidade vamos enviar como resposta o próprio código JavaScript que, quando executado, efetuará a sincronia dos estados da página no cliente e no servidor.

O controle de alterações é feito geralmente de duas maneiras. Uma delas é fazer um controle por região. O objetivo é garantir que trechos específicos sejam sincronizados. Neste tipo de controle não é necessário catalogar cada operação que altera o estado da página, basta gerar um script que substitua no cliente um trecho da página pelo seu equivalente no servidor. Suponhamos que a região controlada seja uma tabela HTML. O controlador iria transformar essa tabela em HTML e gerar um script que trocava a tabela mostrada no cliente pela tabela existente no modelo do servidor. Essa opção torna-se mais ou menos interessante dependendo do tamanho da região controlada. Se esta for pequena, esse método é mais eficaz que o controle pontual de alterações, se for grande estaremos transformando uma requisição AJAX em uma requisição síncrona. Quando a página é armazenada no servidor essa abordagem não é interessante, pois dessincroniza o modelo existente no servidor e a página no cliente já, que as alterações nem sempre se circunscrevem à região controlada.

A outra opção é fazer um controle de cada operação efetuada no servidor. O ponto fraco dessa opção ocorre quando um evento altera muito a página. Neste caso demora mais para executar o script de sincronia do que teria demorado se o evento fosse síncrono. Essa é a opção escolhida em nossa arquitetura.

Em nossa arquitetura é possível escolher se um dado evento será síncrono ou assíncrono através da mudança de um parâmetro na criação do evento, então é possível contornar esse problema.

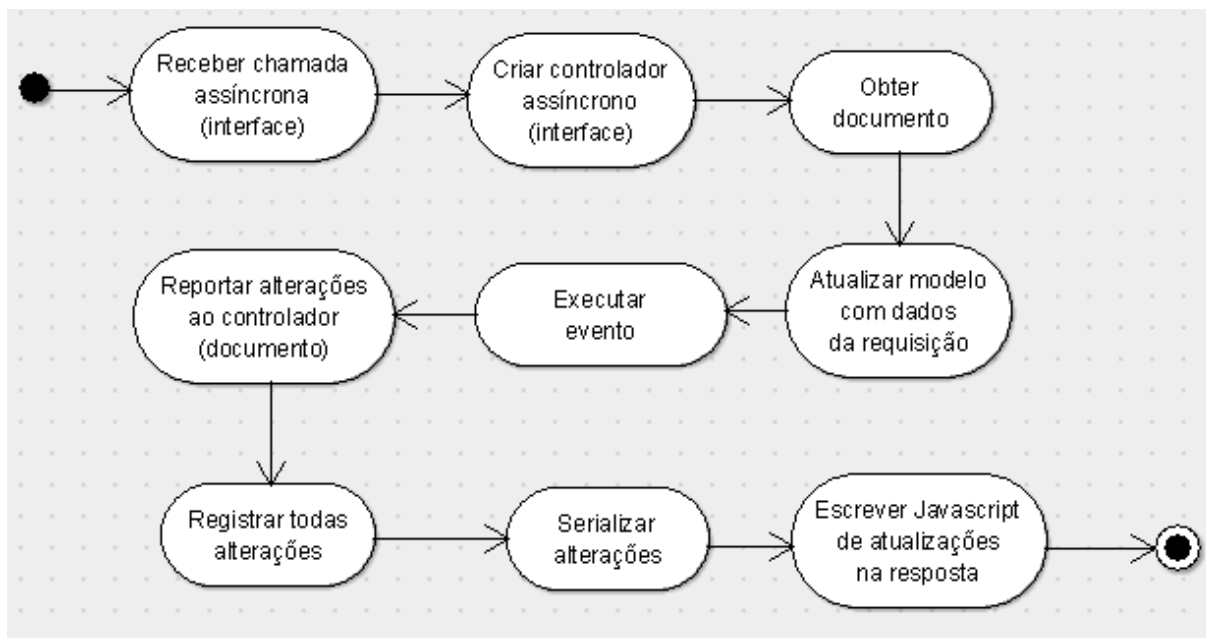


Figura 30 - Diagrama de atividades do controlador assíncrono

O diagrama de atividades da Figura 30 mostra o fluxo do controlador assíncrono. Quando uma requisição assíncrona é recebida, a interface cria um controlador assíncrono para tratá-la. Esse controlador obtém o documento correspondente a partir dos dados enviados. A solicitação pode conter parâmetros que indiquem a necessidade de atualização do modelo. Nesse caso o modelo é atualizado. Durante essa atualização, o histórico de alterações fica desligado, afinal essas atualizações já partiram do estado da página no cliente. Apenas os eventos de servidor correspondentes são chamados. Logo após, o evento que gerou a solicitação é executado e as alterações ocorridas no Modelo vão sendo registradas à medida que forem ocorrendo. Os eventos de servidor que porventura ocorram também são chamados. Após a execução do evento, o histórico de alterações catalogadas é transformado em um script de atualização. Esse script é então escrito na resposta através da interface. É importante usar uma seção crítica para garantir que cada documento só atenderá a uma chamada assíncrona por vez, evitando resultados inesperados no caso de eventos simultâneos acessarem o mesmo objeto.

4.1.3 Modelo

O modelo é usado pelo controlador para escrever a resposta. É a representação no servidor do estado da página no cliente. Este modelo deve ser capaz de produzir dois tipos de resposta diferentes: código fonte de um documento e código JavaScript de atualização.

Os navegadores usam a interface DOM para representar uma página no cliente. Os scripts de atualização gerados no servidor irão acessar essa interface para efetuar atualizações na página. Por outro lado, é relativamente simples realizar a conversão de DOM para código fonte baseado em XML, o que é o caso do HTML, do SVG e do Silverlight. Por esses dois motivos, iremos também projetar o Modelo como uma interface DOM simplificada. Dizemos simplificada, pois a interface DOM contida em um documento é muito complexa e possui muitos atributos calculados em tempo de execução que seriam inúteis para o servidor. Dessa maneira colocaremos no modelo apenas as características que se mostrarem necessárias ao uso.

A definição da interface das classes do modelo poder ser vista na Figura 31. As classes básicas são *Element*, *Document*, *Plugin*, *Event*, *ClientEvent*, *RequestEvent* e *ServerEvent*. A implementação do modelo consiste em criar objetos que implementam ou usam essas classes. Vejamos o significado de cada uma dessas classes abaixo.

Element é a classe que representa qualquer elemento. Páginas, caixas de textos, tabelas, círculos do SVG e do Silverlight são elementos. Essa interface representa as classes *Node* e *Element* do DOM. Aqui elas estão juntas, pois o modelo não considera texto interno como um nó. Ela possui dois tipos de métodos, os de infra-estrutura, usados para serializar o elemento (*getChildNode*, *getTag*, *getAttributeNames*, *getAttributeValue*, etc) e os de atualização (*setInnerText*, *setAttribute*, *addEvent*, etc).

A classe *Document* é usada para representar um documento DOM. Página, Documentos Silverlight ou SVG entram nessa categoria. Os métodos incluídos nessa interface são usados pelo controlador para tratar a requisição.

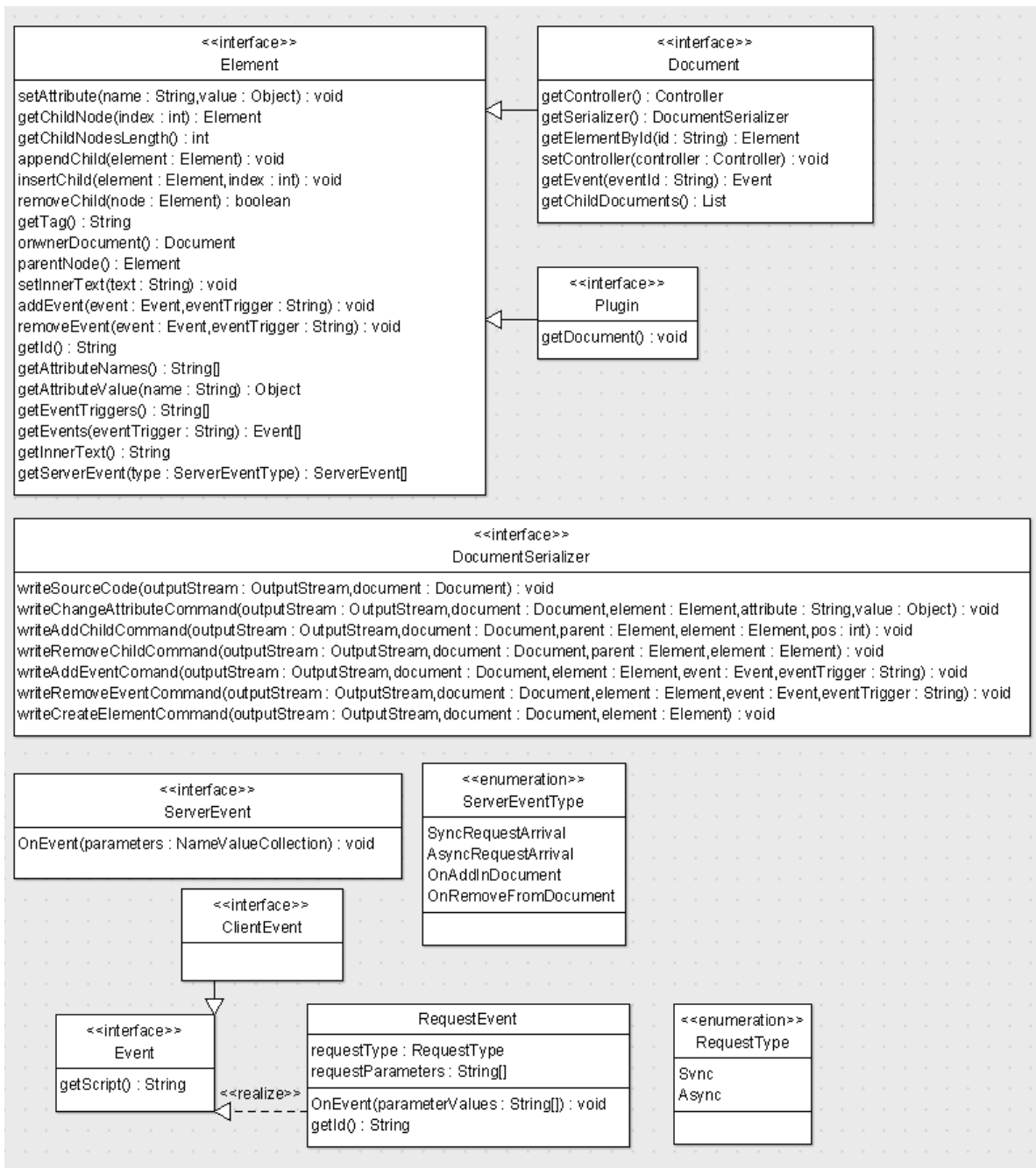


Figura 31 - Interfaces das classes do Modelo

A classe *Plugin* serve para modelar a inclusão de um documento dentro de outro. Ela modela as tags *embed* e *object* quando estas são usadas para inclusão do SVG ou Silverlight dentro de um documento HTML.

A classe *DocumentSerializer* é usada para serializar um documento tanto em uma requisição síncrona quanto em uma requisição assíncrona. Essa interface

poderia ser implementada pela interface *Document*, mas aqui elas estão separadas para facilitar o reaproveitamento de código de serialização.

A classe *Event* representa um event listener do JavaScript. Existem dois tipos de *Event*, *ClientEvent* e *RequestEvent*.

ClientEvent representa um evento puramente do cliente. Esse tipo de evento não gera uma requisição para o servidor. Pode ser usado, por exemplo, para alterar uma imagem quando se passa o mouse por cima. O script desse evento é escrito pelo desenvolvedor. *ClientEvent* pode ser registrado para qualquer evento declarado no DOM.

RequestEvent é usado para fazer requisições tanto síncronas quanto assíncronas ao servidor. O script que será serializado a partir desse evento deverá fazer uma chamada ao mecanismo AJAX informando o identificador do documento e do evento clicado. O script desse evento é escrito pelo *DocumentSerializer*. *RequestEvent* pode ser registrado para qualquer evento declarado no DOM.

ServerEvent modela os eventos ocorridos no servidor. Esses eventos são chamados exclusivamente pelo controlador durante o fluxo de processamento do documento.

Os eventos de servidor têm a dupla finalidade de atualizar o documento quando uma requisição chega e de informar ao elemento o que está ocorrendo no servidor. Essa última finalidade pode ser útil para criar elementos compostos. Alguns elementos necessitam de incluir no documento scripts para o seu funcionamento correto, então quando são adicionados no documento eles adicionam esses scripts e quando são removidos eles os removem. Os tipos de eventos do servidor são:

- *SyncRequestArrival*: Informa a chegada de uma requisição síncrona. Chamado pelo controlador para que um dado elemento se atualize. Os parâmetros informados são os parâmetros que vieram na requisição.
- *AsyncRequestArrival*: Informa a chegada de uma requisição assíncrona. Chamado pelo controlador para que um dado elemento se atualize. Os parâmetros informados são os parâmetros que vieram na requisição.

- *OnAddInDocument*: chamado pelo controlador quando um elemento é adicionado ao documento. O valor do parâmetro é vazio. É chamado para toda a árvore de elementos adicionados.
- *OnRemoveFromDocument*: chamado pelo controlador quando um elemento é removido do documento. O valor do parâmetro é vazio. É chamado para toda a árvore de elementos removidos.

Essas são as classes principais do modelo. Os objetos serão criados a partir da especialização de cada uma delas. Para o HTML será necessário criar elementos HTML e documentos HTML a partir da especialização das interfaces *Element* e *Document*. Uma página será um documento com a tag “*html*” filhos “*head*”, “*body*” etc.

Para o Silverlight e SVG o mesmo se aplica. O SVG será um documento com a tag “*svg*” e filhos “*g*”, “*circle*”, “*rect*”, etc. O Silverlight será um documento “*canvas*” com os filhos definidos pela especificação dessa tecnologia.

O essencial é garantir que um documento HTML só contenha elementos HTML, que um documento SVG só contenha elementos SVG e assim por diante. A inclusão de um documento dentro do outro é feita através dos elementos HTML *Object* ou *Embed*. Assim, um documento Silverlight ou SVG nunca será um filho direto de um elemento HTML. Para se incluir um documento dentro de outro se usa um elemento *Plugin*. Essa decisão foi adotada para garantir o mínimo acoplamento entre documentos.

Uma vez apresentada a interface do modelo, é mostrado, na Figura 32, o diagrama de seqüência para os três tipos de requisição. O pedido síncrono de um novo documento é o tipo de requisição mais simples. O controlador obtém ou cria o documento dependendo do endereço enviado para o servidor. Enquanto o documento é criado, ele faz chamadas aos métodos *On..(OnAttributeChanged, OnElementAdded, etc)* do controlador. Caso haja adição ou remoção de filhos, o controlador chama os eventos de servidor correspondentes. Após a criação, o controlador chama o evento de servidor *SyncRequestArrival* em quem o tiver

registrado. Por fim, o controlador obtém o *DocumentSerializer* e pede a ele para escrever o conteúdo do documento na resposta.

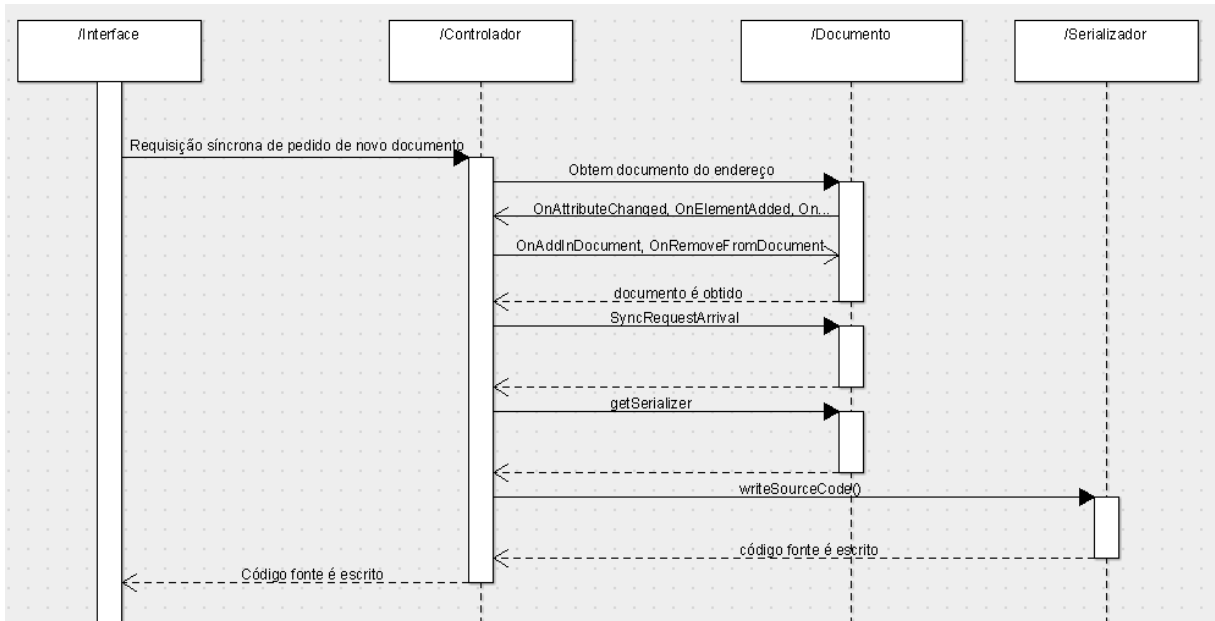


Figura 32 - Requisição síncrona de pedido de novo documento

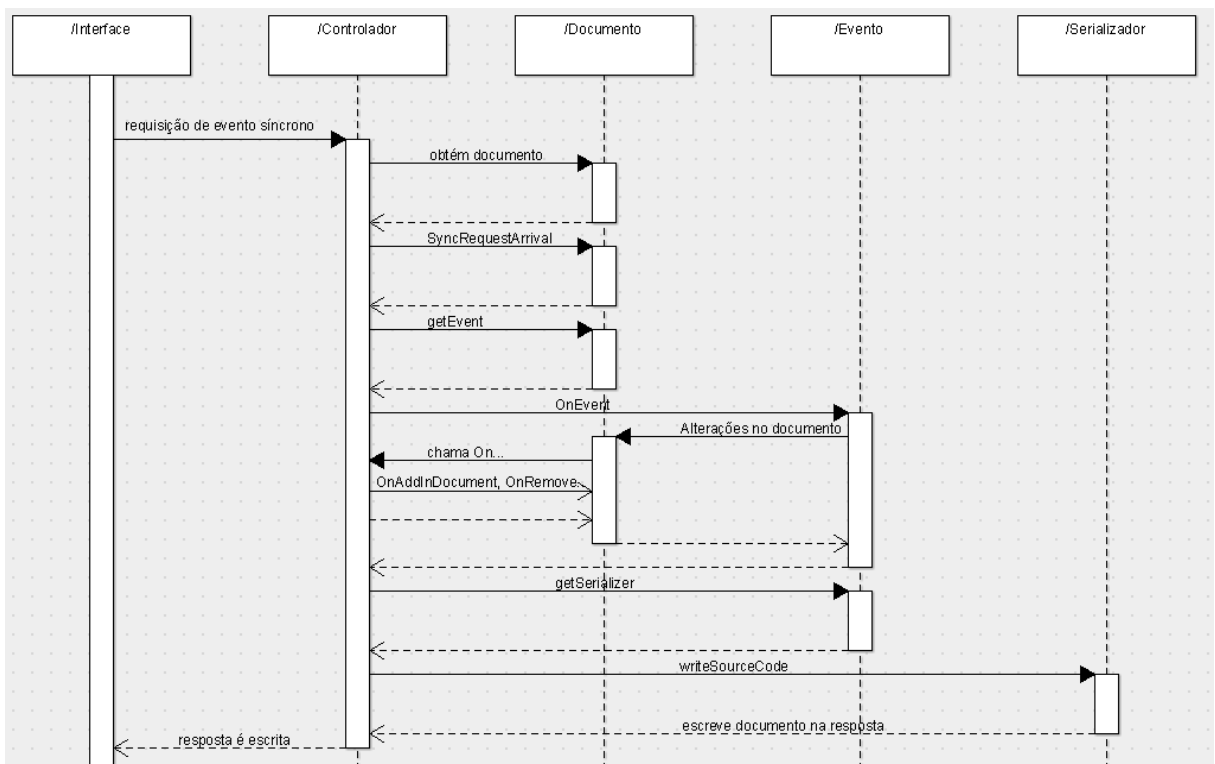


Figura 33 - Requisição de evento síncrono

A Figura 33 mostra o caso de uma requisição de evento síncrono. O controlador chama o evento de servidor *SyncRequestArrival*. Depois ele obtém o evento gerador da requisição usando os parâmetros enviados pelo cliente. O controlador obtém o documento e procura o evento. Uma vez obtido o evento, o método *OnEvent* é chamado usando-se como argumentos os parâmetros registrados no evento quando da sua criação. Esse evento pode alterar o estado da página e, se isso acontecer, o documento chamará os métodos *On..* (*OnAttributeChanged*, *OnElementAdded*, etc) do controlador. Caso haja adição ou remoção de filhos, o controlador chama os eventos de servidor correspondentes. Quando o evento terminar, o controlador pedirá à página o seu serializador. Por fim, o controlador pedirá ao serializador que escreva o código fonte do documento na resposta.

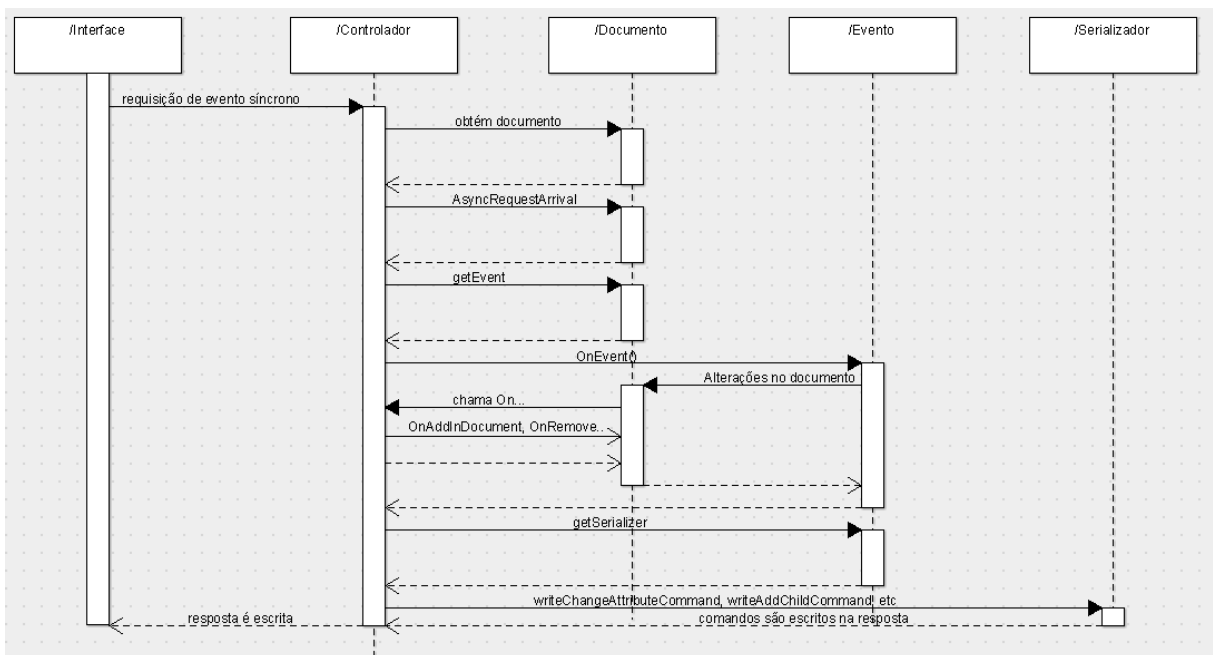


Figura 34 - Requisição de evento assíncrono

A requisição de evento assíncrono (Figura 34) possui tratamento semelhante à requisição de evento síncrono. A diferença ocorre em três pontos. Ao invés de chamar o evento *SyncRequestArrival*, ele chama o *AsyncRequestArrival*. Quando o documento chama os métodos *On..* (*OnAttributeChanged*, *OnElementAdded*, etc) do Controlador, este registra cada comando e seus respectivos parâmetros. E no instante de pedir ao serializador que escreva a resposta, ele chama o método

correspondente a cada comando. É importante que os métodos sejam chamados na mesma ordem em que os comandos ocorreram para que haja a sincronia do estado do documento no servidor e no cliente.

4.2 CÓDIGO INCLUÍDO NO CLIENTE

Para entender qual código deve ser incluído no cliente, voltemos ao modelo de um cliente AJAX. A Figura 35 mostra as três partes fundamentais de um cliente AJAX. O UI, que é a interface visual apresentada ao usuário, o DOM que é implementado internamente pelo navegador e o mecanismo AJAX.

O que deve ser incluído no cliente consiste justamente nas funções de JavaScript que irão fornecer o suporte ao código gerado pelo servidor. A comunicação com o servidor se dá em dois momentos, numerados como 1 e 2 na figura citada.

O número 1 mostra a geração do evento síncrono ou assíncrono. Deve ser criada no mecanismo AJAX uma função capaz de gerar um evento síncrono ou assíncrono.

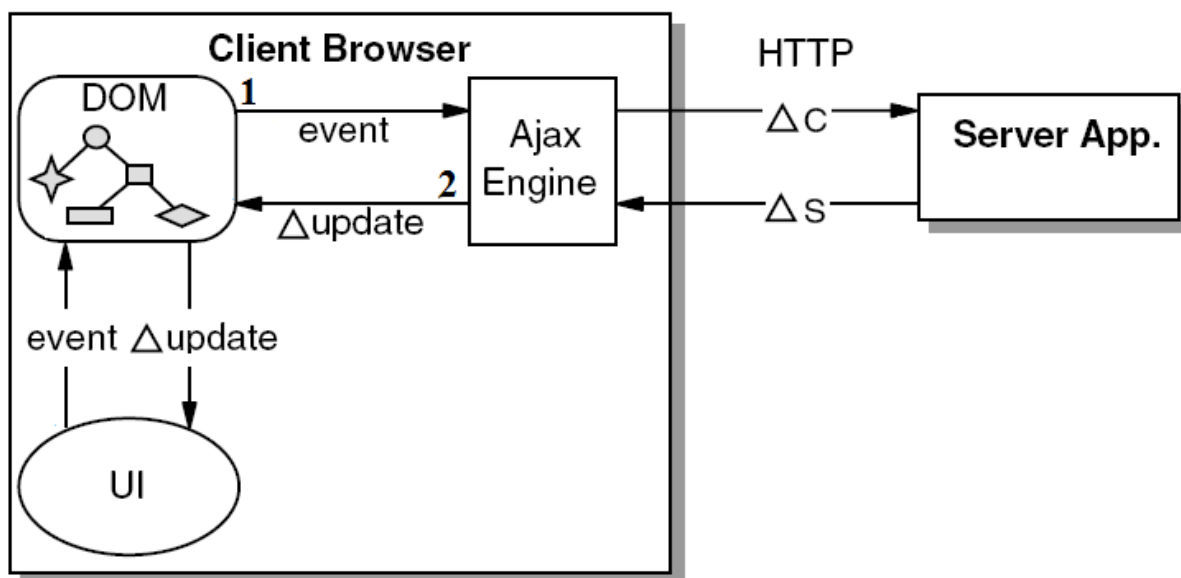


Figura 35 - Modelo do cliente de um aplicativo AJAX

O número 2 mostra a tradução de uma resposta assíncrona em comandos JavaScript (solicitações síncronas são tratadas automaticamente pelo navegador). Na arquitetura aqui apresentada, essa resposta contém um script que deve ser executado. Deve-se, no entanto, garantir que as funções chamadas existem. Essas funções são aquelas que irão executar os comandos escritos pelo *DocumentSerializer* na resposta. Referências para a implementação dessas funções para o Silverlight e para o SVG podem ser encontradas em [37] e [29].

Outro requisito desse cliente AJAX é a geração de requisições síncronas. Uma estratégia para se enviar uma requisição síncrona é a de incluir no formulário (supondo que haja só um formulário) caixas de textos escondidas com os parâmetros e depois executar o método *submit* desse formulário.

Conforme mencionado anteriormente, a inclusão de suporte no cliente consiste em incluir na página os scripts capazes de gerar e tratar requisições síncronas e assíncronas. É o suporte JavaScript para o código gerado no servidor. Exemplos não estão incluídos neste texto pelo fato do código correspondente ser muito grande. Porém, o leitor interessado pode consultar o site de implementação do framework KIS [38].

4.3 INCLUSÃO DE SUPORTE A NOVAS TECNOLOGIAS

A arquitetura proposta é baseada no modelo DOM e não em uma tecnologia específica e, por isso, é expansível. O suporte a qualquer tecnologia baseada em DOM pode ser inserido.

A inclusão de novas tecnologias segue os seguintes passos:

1. Inclusão do documento e dos elementos dessa nova tecnologia. Se for necessário incluir métodos nas interfaces *Element* ou *Document* deve-se criar especializações dessas interfaces com os novos métodos;
2. Se os métodos de registro de modificações (*OnAttributeChanged*, etc.) não contemplarem todas as mudanças do novo tipo de tecnologia, o Controlador deverá ser alterado para incluir em sua interface os novos

tipos de alterações. O registro desses comandos será usado em conjunto com o *DocumentSerializer*, e por esse motivo deve-se criar uma nova especialização dessa interface para incluir a escrita dos novos comandos. Não será preciso alterar a implementação das tecnologias existentes, de vez que esses novos comandos só serão usados pelos novos serializadores. A idéia é fazer conversão de tipos no controlador quando este for chamar serializadores através dos novos métodos;

3. Criação de um *DocumentSerializer*. Cada tecnologia tem suas peculiaridades de serialização que devem ser implementadas neste objeto. A outra parte do sistema que consome as interfaces *Element* e *Document* é o serializador. Caso essas interfaces tenha sido alteradas, será necessário fazer a conversão de tipos dentro dos métodos do serializador. Como cada serializador trata de um documento específico, é seguro fazer essa conversão;
4. Inclusão de suporte JavaScript aos comandos chamados pelo *Serializador*.

5 KIS, UMA IMPLEMENTAÇÃO DA ARQUITETURA PROPOSTA

Para demonstrar a viabilidade da arquitetura proposta no capítulo anterior, ela foi implementada em um framework de código aberto. O objetivo deste capítulo é apresentar brevemente esse framework.

O framework implementado foi batizado de KIS, nome que significa **Keep It Simple**. A idéia por trás do desenvolvimento dessa biblioteca foi a de se fazer um framework leve, mas funcional. É uma implementação tão simples quanto possível da arquitetura apresentada neste trabalho.

A linguagem escolhida para a implementação foi o C# da plataforma .Net. O principal motivo dessa escolha foi o pequeno número de frameworks web disponibilizados para essa plataforma. Assim, o KIS é um framework que funciona usando o IIS (Internet Information Services), que é o servidor de aplicações Web da Microsoft. O KIS está disponível para download em [38].

5.1 IMPLEMENTAÇÃO DA PARTE DO SERVIDOR

5.1.1 Interface

A interface foi implementada registrando-se um *HttpHandler*. Esse registro é feito no arquivo *Web.config* presente nas aplicações Web do C#. O *HttpHandler* deve implementar um método chamado *ProcessRequest*. Esse método será chamado toda vez que o servidor receber uma requisição. A Figura 36 mostra a implementação da interface.

A parte superior dessa figura mostra o registro do *HttpHandler*. O atributo *path* indica que todas as requisições terminadas com *.kis.aspx* serão tratadas por esse handler. O atributo *type* indica qual classe implementará o *HttpHandler* configurado.

A parte inferior da figura mostra o processamento da requisição. Se a requisição contiver o parâmetro que indica que a requisição é assíncrona o controlador assíncrono é instanciado. Caso contrário o controlador síncrono é que o

será. A partir daí o controlador será o responsável por tratar a requisição. O *HttpContext* passado para o controlador contém os objetos que representam a requisição e a resposta.

```
<httpHandlers>
...
  <add verb="*" path="*.kis.aspx" type="Kis.Web.View.HttpHandler, KisWeb" validate="false"/>
</httpHandlers>

public void ProcessRequest(HttpContext p_context)
{
    IHttpController v_currentController;
    string v_requestType = p_context.Request.Params[KisConstants.REQUEST_TYPE];
    if (v_requestType == KisConstants.REQUEST_TYPE_ASYNC)
    {
        v_currentController = new AsyncController(p_context);
    }
    else
    {
        v_currentController = new SyncController(p_context);
    }
    v_currentController.ProcessRequest();
}
```

Figura 36 - Implementação da interface em C#

5.1.2 Controlador

O controlador implementa exatamente o fluxo descrito na seção 4.1.2. As figuras abaixo mostram os métodos para processamento de requisições síncronas e assíncronas.

O método da Figura 37 mostra o tratamento de uma requisição síncrona. Se for uma requisição síncrona de uma página que já está sendo mostrada no cliente (*IsEvent*) o controlador executa o código do evento que gerou a requisição. Logo após o controlador pede ao serializador do documento que o escreva na resposta. O fluxo completo de processamento síncrono já foi mostrado no capítulo anterior.

O método da Figura 38 mostra o tratamento da requisição assíncrona. Primeiramente o evento gerador da requisição é chamado. Durante esse evento, modificações na estrutura da página serão reportadas ao controlador à medida que forem ocorrendo. O controlador armazena essas modificações em objetos *CommandLogger*. Existirá um *CommandLogger* para cada documento alterado. Ao fim do evento o controlador pede aos *CommandLoggers* que chamem os métodos

correspondentes para a escrita de cada comando catalogado. O fluxo completo de processamento assíncrono já foi mostrado no capítulo anterior.

```
protected void ProcessRequest ()
{
    if (IsEvent)
    {
        CallRequestEventListener ();
    }
    CurrentDocument.Serializer.WriteSourceCode (Context.Response.Output);
}
```

Figura 37 - Processamento de requisição síncrona

```
protected override void ProcessRequest ()
{
    CallRequestEventListener ();
    foreach (CommandLogger v_commandLogger in CurrentCommandLoggers)
    {
        v_commandLogger.WriteCommands (Context.Response.Output);
    }
}
```

Figura 38 - Processamento de requisição assíncrona

A Figura 39 mostra a implementação do método *WriteCommands* do *CommandLogger*. O método *WriteCommands* do *Serializer* é a implementação da interface *DocumentSerializer* mostrada na Figura 31. A diferença é que na interface citada existe um método para escrever cada comando e aqui existe um método que escreve todos os comandos.

```
public void WriteCommands (KisTextWriter p_output)
{
    f_document.Serializer.WriteCommands (p_output, |
        RemovedElements, CreateElementCommands,
        AddElementsCommand, AttributeChangeCommands,
        InnextTextChangeCommands, EventListenerCommands);
}
```

Figura 39 - Método *WriteCommands* do *CommandLogger*

5.1.3 Modelo

A interface das classes do modelo já foi mostrada com detalhes no item 4.1.3. No KIS foram implementados classes do modelo para o SVG, HTML e Silverlight. Foi criada uma classe básica *Element* que contém as estruturas que armazenarão o mapa de atributos, o mapa de eventos, o vetor de filhos, o nome da tag, documento pai, nó pai, etc. A partir daí, essa classe foi especializada para a criação dos documentos e elementos específicos de cada tecnologia. Essas classes não serão mostradas aqui já que são muitas. Toda a implementação está disponível para download em [38].

Para exemplificar a interação do modelo com o controlador, é mostrada na Figura 40 a implementação dos métodos *GetInnerText* e *SetInnerText* da interface *Element*. Os métodos *GetInnerText* e *SetInnerText* foram implementados na forma de propriedade do C#. A lógica de implementação já foi descrita no capítulo anterior. O que queremos mostrar é como o elemento reporta modificações ao controlador.

```
public string InnerText
{
    get { return f_innerText; }
    set
    {
        if (f_innerText != value)
        {
            if (value != null && Count != 0)
            {
                throw new InvalidOperationException();
            }
            IDocument v_doc = CurrentOrLastOwnerDocument;
            if (v_doc != null)
            {
                v_doc.CurrentController.OnInnerTextChange(v_doc, this, value);
            }
            f_innerText = value;
        }
    }
}
```

Figura 40 - Implementação dos métodos *GetInnerText* e *SetInnerText*

O elemento verifica se o valor do texto interno mudou e, se mudou, obtém o documento atual ou anterior. A partir do documento ele obtém o controlador responsável por aquela requisição e reporta a modificação. Esse procedimento é

feito sempre que o valor interno de um elemento for alterado. É de responsabilidade de cada controlador saber como tratar a notificação de alterações. Este método reporta modificações para o documento atual ou anterior para contemplar o caso no qual ele foi temporariamente removido de um documento.

5.2 IMPLEMENTAÇÃO DA PARTE DO CLIENTE

A implementação da parte do cliente consiste no suporte JavaScript incluído nas páginas. Toda página criada usando-se o KIS declara um conjunto de funções utilitárias JavaScript que serão usadas internamente pelo mecanismo AJAX.

```
265 GenerateSyncRequest: function (eventId, eventTarget, evaluatedParams) {
266     jqHiddenInputs = $("" +
267         "<input type='hidden' name='" +
268             this.RequestParams.DOCUMENT_NAME + "' value='" +
269             this.DocumentName + "'>" +
270         "<input type='hidden' name='" +
271             this.RequestParams.REQUEST_PARAMS + "' value='" +
272             evaluatedParams + "'>" +
273         "<input type='hidden' name='" +
274             this.RequestParams.REQUEST_TARGET_NODE + "' value='" +
275             this.GetId(eventTarget) + "'>" +
276         "<input type='hidden' name='" +
277             this.RequestParams.REQUEST_TARGET_LISTENER + "' value='" +
278             eventId + "'>" +
279         "<input type='hidden' name='" +
280             this.RequestParams.REQUEST_TYPE + "' value='" +
281             this.RequestParams.REQUEST_TYPE_SYNC + "'>");
282
283     jqHiddenInputs.appendTo(document.forms[0])
284     document.forms[0].submit();
285     jqHiddenInputs.remove();
286 },
287
288 GenerateAsyncRequest: function (eventId, eventTarget, evaluatedParams) {
289     var params = new Object();
290     params[this.RequestParams.DOCUMENT_NAME] = this.DocumentName;
291     params[this.RequestParams.REQUEST_PARAMS] = evaluatedParams;
292     params[this.RequestParams.REQUEST_TARGET_NODE] = this.GetId(eventTarget);
293     params[this.RequestParams.REQUEST_TARGET_LISTENER] = eventId;
294     params[this.RequestParams.REQUEST_TYPE] = this.RequestParams.REQUEST_TYPE_ASYNC;
295
296     $.get(
297         this.GetAsyncUrl(),
298         params,
299         function(data, textStatus) {
300             if (textStatus == "success") {
301                 eval(data);
302             }
303         },
304         "text"
305     );
306 },
```

Figura 41 - Principais funções do mecanismo AJAX

Para implementar essas funções de uma maneira *cross-browser* foi utilizada uma biblioteca JavaScript gratuita de código aberto chamada JQuery [39].

As principais funções do mecanismo AJAX do KIS são mostradas na Figura 41. Elas usam o JQuery para facilitar a escrita de um código *cross-browser*.

A função *GenerateSyncRequest* (linhas 265 a 286) gera uma requisição síncrona. Primeiramente é usado o JQuery para criar caixas de texto escondidas para a passagem de parâmetros (linhas 266 a 281). O nome de cada caixa de texto é o nome do parâmetro e o valor é o valor do parâmetro. Essas caixas de texto são adicionadas ao primeiro formulário da página (linha 283) e esse formulário é submetido via POST. Após a submissão, as caixas de texto são removidas.

A função *GenerateAsyncRequest* (linhas 288 a 306) gera uma requisição assíncrona. Primeiramente é criado um mapa de parâmetros (linhas 289 a 294). Logo após o JQuery é usado para fazer a requisição assíncrona via protocolo GET (linhas 296 a 305). Os parâmetros informados ao JQuery são:

1. Endereço da requisição. Neste caso é o mesmo endereço da página atual;
2. Mapa de parâmetros que serão enviados na requisição;
3. Função callback que será chamada quando a requisição retornar. Os parâmetros dessa função são: dados retornados e status da requisição. Nesta implementação os dados serão interpretados como comandos JavaScript e serão executados(função *eval*);
4. Codificação dos dados retornados. Aqui é usado texto puro.

Conforme já mencionado, a resposta de uma chamada assíncrona será sempre um script contendo chamadas a métodos que, quando executados, refletirão as alterações do modelo no servidor. Por exemplo, se no modelo do servidor o texto interno de um nó HTML for alterado, uma chamada ao método *Kis.SetInnerText* será incluída na script.

```

1  KIS, {
2
3  SetInnerText : function (node, text) {
4      if (typeof(node) === "string") {
5          node = document.getElementById(node);
6      }
7      if (!node) return;
8
9      $(node).text(text || "");
10
11  },
12 };

```

Figura 42 - Metodo *SetInnerText*

A implementação desse método pode ser vista na Figura 42. A lógica é simples. Se for passada uma string como parâmetro ele obterá o nó correspondente. Se o nó for encontrado ele usa o JQuery para alterar o valor interno de um nó.

Esse método também poderia ser usado para o SVG, desde que o método *getSvgElementById* fosse chamado para o obter o nó. É importante notar que esse método não é nativo do HTML e deveria ser criado antes.

5.3 EXEMPLO DE USO

Aqui serão mostrados dois exemplos de páginas criadas usando-se o KIS. O primeiro é uma página bem simples sem eventos e o segundo é um pequeno sinótico com eventos e uso de AJAX.

5.3.1 Hello World!

O primeiro exemplo de página criada usando-se o KIS é uma página de Hello World. O objetivo desse exemplo é o de mostrar a estrutura de uma página criada a partir do KIS.

A interface gráfica dessa página pode ser vista na Figura 43. Ela mostra textos “*Hello world!*” escritos usando-se HTML, SVG e Silverlight. O código que a criou é mostrado na Figura 44.

A classe *WebPage* (linha 16 da Figura 44) é a classe que implementa um documento HTML. Primeiramente é criado um elemento *Label*. O texto interno desse

elemento é alterado e ele é adicionado na página, juntamente com dois elementos de quebra de linha (linhas 20 a 22).

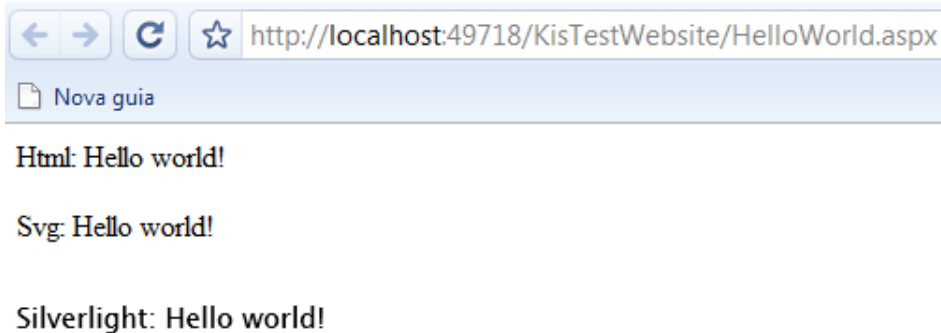


Figura 43 – Interface gráfica da página Hello world

```
16 public class HelloWorld: WebPage
17 {
18     public HelloWorld()
19     {
20         Label v_Label = new Label();
21         v_Label.Text = "Html: Hello world!";
22         AppendChildren(v_Label, new BreakLine(), new BreakLine());
23
24         SvgDocument v_svgDoc = new SvgDocument();
25         v_svgDoc.WidthPixels = 200;
26         v_svgDoc.HeightPixels = 20;
27         v_svgDoc.AppendChild(new SvgText("Svg: Hello world!", 0, 12.66));
28         AppendChildren(v_svgDoc.Plugin, new BreakLine(), new BreakLine());
29
30         SvlDocument v_svlDoc = new SvlDocument();
31         v_svlDoc.WidthPixels = 200;
32         v_svlDoc.HeightPixels = 20;
33         v_svlDoc.AppendChild(new SvlText("Silverlight: Hello world!"));
34         AppendChild(v_svlDoc.Plugin);
35
36     }
37 }
38 }
```

Figura 44 - Código da página de Hello world

Em seguida um documento SVG é criado (linha 24). A largura e altura desse documento são configuradas (linhas 25 e 26). É adicionado nele um elemento *SvgText* com o texto interno “*Svg: Hello world!*” na coordenada x=0 e y=12.66 (linha 27). A posição y de um texto SVG é definida com relação ao canto inferior direito do texto. Após a construção desse documento, o seu plugin é adicionado na página junto com duas quebras de linha (linha 28).

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
2   'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd' >
3 <html id='Kis_5' xmlns='http://www.w3.org/1999/xhtml' >
4   <head id='Kis_2' >
5     <title id='Kis_1' ></title>
6     <script id='Kis_6' type='text/javascript'
7       src='/KisTestWebsite/Kis.res?t=js&a=KisWeb, Version=1.0.0.0, Culture=neutral,
8       PublicKeyToken=null&f0=jquery.1.3.2.min.js&h0=5ca91c4a&f1=Kis.js&h1=1b434342&f2=Kis.svg.js&h2=6572eb28
9       &f3=Kis.svl.js&h3=ff149317' >
10
11     </script>
12     <script type='text/javascript' >
13       $(document).ready(function() {
14       });
15     </script>
16   </head>
17   <body id='Kis_4' >
18     <form id='Kis_3' name='Kis_3' action='' method='post' >
19       <label id='Kis_7' >Html: Hello world!</label>
20       <br id='Kis_8' />
21       <br id='Kis_9' />
22       <embed id='Kis_b' wmode='transparent'
23         type='image/svg+xml' width='200px' height='20px'
24         src='/KisTestWebsite/HelloWorld.aspx?DOC_NAME=Kis_5&CHILD_DOC=Kis_a' />
25       <br id='Kis_d' />
26       <br id='Kis_e' />
27       <embed id='Kis_11' windowless='true'
28         type='application/x-silverlight-2' width='200px' height='20px'
29         source='/KisTestWebsite/HelloWorld.aspx?DOC_NAME=Kis_5&CHILD_DOC=Kis_f' />
30       <script id='Kis_10' type='text/javascript'
31         src='/KisTestWebsite/HelloWorld.aspx?DOC_NAME=Kis_5
32         &CHILD_DOC=Kis_f&source_type=scripts' >
33
34       </script>
35     </form>
36   </body>
</html>

```

Figura 45 - Código HTML do Hello world

Por fim é criado um documento Silverlight (linha 30). A altura e largura são configuradas (linhas 31 e 32) e um elemento *Sv/Text* com o texto “*Silverlight: Hello world!*” é adicionado (linha 33). Como nenhuma posição foi explicitada, o valor considerado foi $x=0$, $y=0$. A posição de um texto no Silverlight é definida com relação ao seu canto superior direito. Após a construção desse documento, o seu plugin é adicionado na página (linha 34). O código HTML gerado a partir dessa página é mostrado na Figura 45.

Esse código HTML tem duas partes principais, o conteúdo da tag *head* (linhas 5 a 15) e o conteúdo da tag *form* (linhas 19 a 33).

O *head* tem duas partes. A importação dos scripts do KIS (linhas 6 a 11) e a adição dos event listeners (linhas 13 e 14). Neste exemplo não foram adicionados event listeners.

O *form* contém os elementos que foram adicionados. O label (linha 19), o plugin do documento SVG (linhas 22 a 24) e o plugin do documento Silverlight (linhas 27 a 33). As quebras de linhas são as tags *br*.

Quando o navegador encontra um *embed* SVG ele solicita ao navegador o seu código fonte e pede ao plugin para renderizá-lo. O código do documento SVG, mostrado na Figura 46, é bem simples. Contém apenas o nó raiz, o texto e os scripts de inicialização do documento. A inicialização consiste na criação do método *getSvgElementById*, feita internamente pelo método *AddDocument*, e na adição dos event listeners declarados. Neste exemplo nenhum event listener foi registrado.

```
1 <svg id='Kis_a'
2   xmlns='http://www.w3.org/2000/svg'
3   version='1.1' width='100%' height='100%' >
4
5   <text id='Kis_c' x='0' y='12.66' >Svg: Hello world!</text>
6   <script type='text/ecmascript' >
7     var Kis = parent.Kis;
8     Kis.Svg.AddDocument(document);
9   </script>
10 </svg>
```

Figura 46 – Código XML do documento SVG

O documento Silverlight foi dividido em dois elementos, mostrados na Figura 47 e na Figura 48. Um deles é o documento em si e o outro são os scripts de inicialização desse documento. Ao contrário do SVG, o Silverlight não permite a declaração de scripts dentro do seu código fonte. A estrutura desse documento é bem similar à do SVG, com a diferença que os scripts foram importados à parte. A criação do método *getSilverghtElementById* é feita através do identificador do plugin e não do documento Silverlight, pois o script não tem acesso direto ao documento.

```
1 <?xml version='1.0' encoding='utf-8' ?>
2 <Canvas Name='Kis_f'
3   xmlns='http://schemas.microsoft.com/client/2007'
4   xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml' >
5
6   <TextBlock Name='Kis_12' Text='Silverlight: Hello world!' ></TextBlock>
7 </Canvas>
```

Figura 47 - Código XML do documento Silverlight

```
1 $(document).ready(function() {  
2     Kis.Svl.AddDocument('Kis_11');  
3 });
```

Figura 48 - Scripts de inicialização do documento Silverlight

5.3.2 Sinótico

O segundo exemplo é a página de um sinótico. O objetivo desse exemplo é mostrar o AJAX em ação e fazer uma comparação entre a comunicação síncrona e a comunicação assíncrona. A comparação será feita com relação à velocidade e à quantidade de dados trafegados.

A Figura 49 mostra a página do sinótico de um tanque. Esse sinótico mostra graficamente duas variáveis: nível e temperatura do tanque. O sinótico do tanque 1 foi feito em SVG e o do tanque 2 em Silverlight. O código fonte desse exemplo também pode ser encontrado em [38].

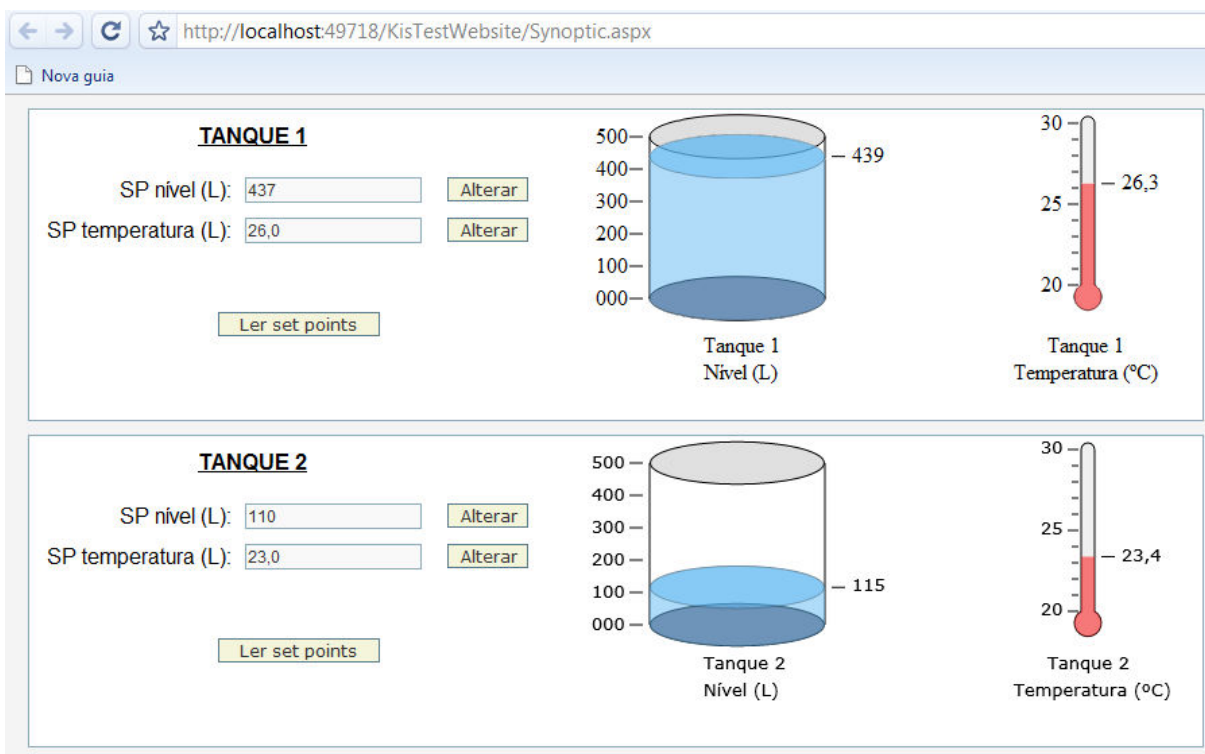


Figura 49 - Página com um sinótico SVG e Silverlight

O funcionamento dessa página é o seguinte: no servidor existe uma thread que simula um processo controlado. A thread simula o processo usando os valores dos set points de nível e temperatura. Se esses valores forem alterados, o nível e/ou a temperatura serão ajustados para acompanhá-los. O modelo da página no servidor acessa esses valores através de uma interface e atualiza os documentos que representam os sinóticos. A página atualiza esses valores em ciclos regulares de um segundo.

À esquerda da figura são mostrados os set points. Para se alterar um set point, basta escrever o seu valor na caixa de texto e clicar em alterar. Quando isso for feito será gerada uma requisição para o servidor que atualizará o set point correspondente. Ao se clicar em ler set points, a página gerará uma requisição que atualizará as caixas de texto correspondentes.

O nível do tanque é mostrado na figura situada no meio. O nível da água mostrado é proporcional ao nível medido do tanque. Cada tanque possui capacidade de 500 l e as marcas de nível são mostrados à sua esquerda. À direita é mostrado o nível atual.

O valor da temperatura é mostrado no termômetro à direita. O intervalo de medição do termômetro apresentado é de 20 °C a 30 °C. Assim como no tanque, as marcas com as temperaturas ficam à esquerda e o valor medido à direita.

A estrutura de uma página criada usando-se o KIS foi mostrada no exemplo anterior e não será mostrada novamente aqui. Iremos focar apenas a criação de eventos que geram requisição.

5.3.2.1 Eventos do sinótico

O sinótico gera eventos em quatro situações:

1. Clique no botão de alteração de valores dos set points;
2. Clique no botão de leitura dos valores dos set points;
3. Leitura cíclica dos valores do tanque;

4. Alteração do valor de uma caixa de texto;

A criação de um evento usando-se o KIS é mostrada na Figura 50. O evento é o de clique no botão de ler os valores dos set points de um tanque. A parte superior da figura mostra a criação do botão e o registro do evento. O objeto usado para registrar o evento é do tipo *RequestEventListener*. O *RequestEventListener* recebe dois parâmetros em sua construção: o tipo de requisição que será gerada (síncrona ou assíncrona) e um ponteiro para o método que será chamado quando o evento for executado.

```
Button v_btnReadSetPoints = new Button("Ler set points");
v_btnReadSetPoints.AddEventListener(
    new RequestEventListener(RequestEventType.Async, onSetPointRead));
f_tankByButton[v_btnReadSetPoints] = v_tank;

private void onSetPointRead(IElementBridge p_sender)
{
    Tank v_tank = TankController.GetTank(f_tankByButton[(Button)p_sender].TankCode);
    f_temperatureSetPointByTank[v_tank].Value = v_tank.TemperatureSetPoint.ToString("0.0");
    f_levelSetPointByTank[v_tank].Value = v_tank.LevelSetPoint.ToString("000");
}
```

Figura 50 - Criação de evento usando-se o KIS

Quando esse botão for clicado o método *onSetPointRead* será executado. O argumento *p_sender* dessa função é o botão que foi clicado. A lógica implementada nesse método obtém o código do tanque relacionado com aquele botão. Com o código do tanque, o método obtém o valor atual da suas variáveis e usa-os para atualizar os valores das caixas de texto correspondentes. Internamente essas modificações serão reportadas ao controlador que as escreverá na resposta.

Na Figura 51 o conteúdo de uma requisição e da consequente resposta assíncrona geradas por esse evento são mostrados. A requisição envia para o servidor os parâmetros necessários para identificar a solicitação. O parâmetro *DOC_NAME* contém o nome da página, *REQ_TARGET_LISTENER* é o identificador do event listener, *REQ_TARGET_NODE* é o dono do event listener e *REQ_TYPE* é o tipo de requisição.

Se o evento fosse gerado devido à alteração do valor de uma caixa de texto, seria enviado um parâmetro a mais, o *REQ_PARAMS*, que conteria o valor atual

dessa caixa de texto. Esse valor seria usado pelo controlador para atualizar o objeto do modelo que representa essa caixa de texto no servidor.

Cabeçalhos de Solicitação

```
Host localhost:49718
User-Agent Mozilla/5.0 (Windows; U; Windows NT 6.0; pt-BR; rv:1.9.0.10)
Accept text/plain, */*
Accept-Language pt-br,en-us;q=0.5
Accept-Encoding gzip,deflate
Accept-Charset ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive 300
Connection keep-alive
X-Requested-With XMLHttpRequest
Referer http://localhost:49718/KisTestWebsite/Synoptic.aspx
Cookie ASP.NET_SessionId=jvvggjc45dpedylvrclbdpj4
```

Parâmetros

```
DOC NAME Kis_11f
REQ_TARGET_LISTENER Kis_188
REQ_TARGET_NODE Kis_189
REQ_TYPE ASYNC
```

Figura 51 - Solicitação assíncrona do evento de atualizar os valores de set point

A resposta gerada pode ser vista na Figura 52 e contém o script de atualização da página. Quando executado, atualizará o valor das caixas de texto dos set points daquele tanque. É a representação em JavaScript das alterações efetuadas na página pelo método *onSetPointRead*.

Se a requisição acima fosse síncrona, ao invés de assíncrona, a resposta seria todo o código HTML da página.

Cabeçalhos de Resposta

```
Server ASP.NET Development Server/9.0.0.0
Date Sat, 16 May 2009 18:52:26 GMT
X-AspNet-Version 2.0.50727
Pragma no-cache,no-store
Cache-Control private
Expires Sat, 16 May 2009 18:51:26 GMT
Content-Type text/html; charset=utf-8
Content-Length 93
Connection Close
```

Resposta

```
Kis.SetAttribute('Kis_18f', 'value', '23,0');
Kis.SetAttribute('Kis_18b', 'value', '110');
```

Figura 52 - Resposta à solicitação assíncrona da figura anterior

Outro evento gerado por essa página é o evento de atualização cíclica dos valores do tanque. O diferencial nesse evento é que ele altera mais de um documento. O listener que responde a esse evento obtém o documento de cada sinótico e atualiza os objetos que representam o nível e a temperatura.

```
private void UpdateSynoptic(IElementBridge p_sender)
{
    foreach (Tank v_tank in TankController.GetAllTanks())
    {
        f_synopticByTank[v_tank].UpdateData(v_tank);
    }
}
```

Figura 53 – Listener do evento de atualização dos valores do tanque

O método *UpdateSynoptic* (Figura 53) é o event listener desse evento cíclico. Sua função é obter o sinótico de cada tanque e atualizar seus valores. Aqui cada sinótico é um documento SVG ou Silverlight.

O event listener pode alterar o estado de qualquer documento pertencente àquela página. Isso acontece porque um objeto informa a qual documento pertence quando reporta ao controlador as alterações sofridas. O controlador cria, então, históricos por documento e assim consegue escrever corretamente a resposta. Isso é necessário, pois cada tipo de documento gera comandos específicos para cada tipo de alteração.

A Figura 54 mostra um exemplo de resposta ao evento cíclico. Os comandos que começam por *Kis.Svg* são comandos que alteram um documento SVG e os que começam por *Kis.Svl* são comandos que alteram um documento Silverlight. O Kis atribui um identificador único para cada objeto e por isso, para obtê-lo, ele não precisa saber a qual documento pertence. Ele simplesmente pergunta a todos os documentos até achá-lo. Neste exemplo foram gerados comandos de alteração de atributo e mudança de texto interno. Os argumentos para o comando de mudança de atributo são: identificador, atributo e valor. Os argumentos do comando de mudança de texto interno são o identificador e o novo valor do texto interno. Em resumo, o que esse comando está fazendo é alterar o nível de água dos tanques e o nível de temperatura dos termômetros.


```

Kis.Svg.SetAttribute('Kis_126', 'y1', '34.4');
Kis.Svg.SetAttribute('Kis_126', 'y2', '34.4');
Kis.Svg.SetAttribute('Kis_125', 'y', '39.4');
Kis.Svg.SetAttribute('Kis_124', 'cy', '34.4');
Kis.Svg.SetAttribute('Kis_123', 'd', 'M80,34.4 v105.6 a65,16.25 0 1,0 130,0 v-105.6 a65,16.25 0 1,0 -130,0');
Kis.Svg.SetAttribute('Kis_129', 'y1', '57.8767816993776');
Kis.Svg.SetAttribute('Kis_129', 'y2', '57.8767816993776');
Kis.Svg.SetAttribute('Kis_128', 'y', '62.8767816993776');
Kis.Svg.SetAttribute('Kis_127', 'd', 'M400,57.8767816993776 v72.1232183006224 a10,10 0 1,0 10,0 v-72.1232183006224 h-10');
Kis.Svg.SetInnerText('Kis_125', '440');
Kis.Svg.SetInnerText('Kis_128', '26,0');
Kis.Svl.SetAttribute('Kis_15e', 'Y1', '113.6');
Kis.Svl.SetAttribute('Kis_15e', 'Y2', '113.6');
Kis.Svl.SetAttribute('Kis_15d', 'Canvas.Top', '103.6');
Kis.Svl.SetAttribute('Kis_15d', 'Text', '110');
Kis.Svl.SetAttribute('Kis_15c', 'Canvas.Top', '97.35');
Kis.Svl.SetAttribute('Kis_15b', 'Data', 'M80,113.6 v26.4 a65,16.25 0 1,0 130,0 v-26.4 a65,16.25 0 1,0 -130,0');
Kis.Svl.SetAttribute('Kis_161', 'Y1', '90.5018228205395');
Kis.Svl.SetAttribute('Kis_161', 'Y2', '90.5018228205395');
Kis.Svl.SetAttribute('Kis_160', 'Canvas.Top', '80.5018228205395');
Kis.Svl.SetAttribute('Kis_15f', 'Data', 'M400,90.5018228205395 v39.4981771794605 a10,10 0 1,0 10,0 v-39.4981771794605 h-10');

```

Figura 54 - Resposta ao evento de atualização dos sinóticos

Os conceitos envolvidos na geração dos outros eventos pela página já foram discutidos e por isso eles não serão particularmente comentados.

5.3.2.2 Comparação de desempenho: Síncrono x Assíncrono

O desempenho dos dois tipos de requisição foi comparado em dois quesitos: quantidade de dados trafegados e tempo de processamento de cada tipo de requisição.

O primeiro teste feito foi a medição da quantidade de dados trafegados. Para realizar essa medição foi usado o plugin *Firebug* [40] do Firefox. Com esse plugin é possível monitorar as requisições e as respostas feitas por uma página. A metodologia do teste foi a seguinte:

1. O evento cíclico do sinótico foi configurado para gerar requisições síncronas. Foi anotada a quantidade de dados trafegados para se fazer duzentas atualizações;
2. O evento cíclico do sinótico foi reconfigurado para gerar requisições assíncronas. O mesmo procedimento do item anterior foi adotado para medir os dados trafegados.

Os resultados obtidos são apresentados na Tabela 7.

| Requisição | Total dados (Kb) | Nº requisições | Dados por req. (Kb) |
|------------|------------------|----------------|---------------------|
| Síncrona | 2641 | 200 | 13,20 |
| Assíncrona | 283 | 200 | 1,41 |

Tabela 7 - Comparação de dados trafegados por tipo de requisição

Conforme pode ser observado, a requisição AJAX é bem mais leve para o caso observado. A quantidade de dados trafegada é quase dez vezes menor em relação à requisição síncrona. Esse era o resultado esperado, afinal a requisição AJAX contém apenas o que deve ser modificado e são poucas as modificações que cada requisição fez na página. Já a requisição síncrona sempre contém a página inteira e neste caso não compensa usá-la.

O segundo teste feito foi a medição do tempo gasto para executar cada tipo de requisição. Aqui o *Firebug* do Firefox também foi usado. Para efetuar esse teste a seguinte abordagem foi usada:

1. O evento cíclico do sinótico foi configurado para gerar requisições síncronas. O intervalo do evento cíclico foi alterado para um milissegundo. Dessa forma o tempo de cada requisição foi o tempo gasto para carregar o documento e gerar outra requisição. Foram feitas duzentas requisições nesse teste;
2. O evento cíclico do sinótico foi reconfigurado para gerar requisições assíncronas. O tempo de intervalo entre eventos foi diminuído iterativamente até o ponto em que a fila de requisições não aumentava indefinidamente (em torno de 50 milissegundos). Foram feitas duzentas requisições.

Os resultados obtidos são apresentados na Tabela 8.

| Requisição | Tempo total (s) | Nº requisições | Tempo por req. (s) |
|------------|-----------------|----------------|--------------------|
| Síncrona | 207 | 200 | 1,04 |
| Assíncrona | 111 | 200 | 0,56 |

Tabela 8 - Comparação de tempo gasto por tipo de requisição

Aqui a requisição assíncrona obteve melhor desempenho. A diferença verificada no tempo se deve à relação do tempo levado pelo navegador para renderizar a página com o tempo levado para renderizar as modificações dos vários comandos.

Um fator que não foi levado em conta em nenhum dos dois testes foi o conforto para o usuário. Quando a requisição é assíncrona, o fluxo de uso da página não é constantemente interrompido como na requisição síncrona. Essa questão já foi abordada nos capítulos iniciais dessa dissertação.

6 CONCLUSÕES E POSSIBILIDADES FUTURAS

Nos últimos anos os aplicativos web foram se tornando cada vez mais difundidos. Contudo, o modelo web clássico ainda apresentava interação muito limitada com o usuário, e isso era devido, em grande parte, ao modelo síncrono da requisição HTTP.

Com a evolução das aplicações e navegadores web, foram surgindo novas tecnologias e os desenvolvedores começaram a lançar mão delas para melhorar a experiência da navegação web. Foi a partir da combinação dessas tecnologias que surgiu o AJAX.

O AJAX se baseou em um modelo assíncrono de comunicação e melhorou significativamente o nível de interação que um aplicativo web poderia oferecer. Entretanto, a implementação do AJAX é bem trabalhosa e isso aumentou bastante o nível de complexidade do desenvolvimento web. Para mitigar esse problema, surgiram vários frameworks web com suporte a AJAX e foi neste ponto que este trabalho se inseriu.

O objetivo desse trabalho foi o de se produzir uma arquitetura expansível para frameworks web AJAX, mais especificamente uma arquitetura que suportasse no mínimo SVG e Silverlight.

O SVG e o Silverlight são tecnologias voltadas para a produção de desenhos bi ou tridimensionais e são usados para a criação de aplicações web RIA (**R**ich **I**nteractive **A**pplications), tais como visualização de vídeos dinâmicos e iterativos, jogos, gadgets, sinóticos, gráficos, banners de propaganda etc. O principal ganho de se usar SVG ou Silverlight com AJAX é o aumento de interatividade com o usuário que essa combinação proporciona.

Para atingir esse objetivo, primeiramente foi apresentado um panorama geral da implementação de AJAX em aplicativos web. Foram discutidos temas como macro-arquitetura de aplicativos web, mecanismos AJAX, uso do conteúdo de uma

resposta assíncrona para atualizar uma página já carregada, interface DOM e arquiteturas comuns para a inclusão de suporte no servidor.

Com base nessas referências, foi proposta a arquitetura apresentada no capítulo 4. Essa arquitetura usou o padrão MVC para gerenciar os dois tipos de requisição possíveis (síncrono e assíncrono).

O padrão MVC é usado para promover o desacoplamento entre a parte que interage com o usuário e a parte que faz o trabalho pesado. Esse padrão define três papéis principais: a interface, que interage com o usuário; o modelo que é a representação do problema; e o controlador que serve de intermediário entre as duas partes anteriores. Na arquitetura proposta, a interface representou a resposta enviada para o navegador; o modelo foi concebido como uma representação parcial da interface DOM do cliente no servidor; e o controlador fez o papel de intermediário, reportando ao modelo as modificações trazidas pela interface e pedindo ao modelo que escrevesse a resposta adequada (síncrona ou assíncrona) na interface.

Uma das características principais da arquitetura proposta é a expansibilidade (abertura à inclusão de novas tecnologias). O que proporcionou essa característica foi principalmente a concepção do modelo. O modelo não foi concebido para trabalhar com uma tecnologia específica como é o caso de muitos frameworks existentes no mercado [41], e sim para gerenciar tecnologias que implementam a interface DOM. Em geral os frameworks baseados em MVC trabalham apenas com um tipo de documento, o documento HTML. A principal inovação da arquitetura aqui apresentada é a possibilidade de tratar múltiplos documentos. Assim pode-se ter em uma mesma página web documentos HTML, SVG e Silverlight.

Outra característica essencial é o suporte a requisições AJAX. Embora existam muitos frameworks que também tenham essa característica, ela é aqui apresentada com dois aspectos diferenciados: a possibilidade de escolha entre a geração de eventos síncronos ou assíncronos e o suporte a outras tecnologias que não o HTML (SVG e Silverlight).

Existem situações em que o uso da requisição síncrona é mais recomendada que o uso da requisição assíncrona, como foi discutido nos itens 4.1.2.1 e 4.1.2.3, e

é interessante possibilitar ao desenvolvedor escolher qual das duas opções melhor atende a cada evento. Essa opção não é contemplada pela maioria das arquiteturas. Nelas uma requisição é sempre síncrona ou assíncrona, sem possibilidade de mudança.

A inclusão de suporte AJAX a outras tecnologias decorreu diretamente da possibilidade de tratar vários tipos de documentos em uma mesma requisição. Essa característica faz com que cada tipo de documento tenha o seu histórico de comandos e saiba serializá-los, possibilitando a inclusão do suporte mencionado. Dentre os frameworks pesquisados não existe nenhum que ofereça suporte AJAX ao SVG e somente o .Net da Microsoft oferece suporte AJAX ao Silverlight.

A validação da arquitetura foi feita no capítulo 5 com a sua implementação em um framework open source, o KIS [38]. Nessa implementação o suporte AJAX a SVG e Silverlight foi incluído com sucesso. Esse capítulo mostrou a viabilidade da arquitetura proposta e apresentou um exemplo de aplicativo que utiliza o AJAX juntamente com SVG e o Silverlight. Alguns testes feitos com base nesse exemplo mostraram a vantagem de se usar o AJAX com essas tecnologias. Estes testes compararam o tamanho e o tempo de processamento de requisições síncronas com as requisições assíncronas equivalentes (AJAX). Em ambos os testes a requisição assíncrona obteve melhor resultado, isto é, foi mais rápida e consumiu menor banda.

Este trabalho apresentou duas contribuições principais. A primeira delas é a criação do KIS, o primeiro framework de código aberto a oferecer suporte AJAX ao Silverlight e ao SVG. A segunda contribuição foi a discussão em torno dos problemas relacionados à implementação do AJAX nos capítulos 3, 4 e 5. Ao longo desses capítulos foram discutidos problemas e apresentadas soluções para temas como envio de parâmetros via requisições síncronas e assíncronas, inclusão de suporte JavaScript no cliente, arquiteturas de suporte AJAX no servidor, geração de uma resposta AJAX, lógica de controladores síncronos e assíncronos no servidor, criação de um modelo expansível, etc. É uma compilação dos problemas de implementação e das soluções adotadas.

Com relação às possibilidades futuras deste trabalho, podemos citar algumas áreas a serem desenvolvidas.

A primeira delas é a inserção no KIS de suporte a outras tecnologias. Existe hoje outra tecnologia, o Adobe Flex [42], não abordada nesse trabalho e que concorre com o Silverlight e SVG. O Flex é tido pela Adobe como a evolução do atual Flash (ambos são da mesma empresa).

Outra linha de desenvolvimento é o aumento de eficiência no registro de alterações ocorridas no Modelo do servidor (documento). Neste trabalho foi proposto o monitoramento dos seguintes tipos de modificações: alteração de texto interno, inclusão e remoção de elementos, inclusão e remoção de eventos. Pode ser incluído o monitoramento de outras operações para diminuir tráfego de dados, como por exemplo, a remoção de todos os filhos. O algoritmo de registro dos comandos também pode ser melhorado. Registrar todos os comandos na ordem em que aconteceram é ineficiente. Se o valor interno de um elemento for alterado e na mesma requisição esse elemento for retirado do documento, será gerado um comando inútil, e isso ocorre em várias outras situações. A tradução desses comandos em JavaScripts mais eficientes também é outro tema interessante que pode melhorar sensivelmente o desempenho de uma aplicação.

Uma questão que não foi abordada aqui é o gerenciamento de memória do servidor. A arquitetura proposta armazena todos os documentos mostrados em memória. Não foi discutido nesse trabalho um meio eficaz de gerenciamento desse armazenamento. É basicamente dar resposta à pergunta: Quando um documento deve ser removido da memória do servidor? Uma abordagem inicial poderia ser a criação de um mapa temporizado. Se um documento não for acessado por um tempo maior ou igual ao tempo da sessão, este documento deve ser removido.

Atualmente uma nova linha de frameworks web tem aparecido. São frameworks que funcionam quase que inteiramente no cliente. Superficialmente, podemos dizer que esses frameworks funcionam com o desenvolvedor usando, durante seu trabalho, uma linguagem de servidor, como o Java ou C#. Essa parte de desenvolvimento é a mesma que seria feita no KIS. Antes de publicar sua aplicação

ele roda um compilador que transforma esse código em HTML + JavaScript. As páginas não são geradas em tempo de execução, e sim em tempo de compilação. As chamadas a funções que necessitem de dados no servidor são feitas via RPC (**R**emote **P**rocedure **C**all). Uma área de desenvolvimento interessante aqui seria criar um compilador dessa natureza para ser usado em conjunto com o KIS. Seria o primeiro framework em C# a usar essa filosofia.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] GARRET, J. Ajax: a New Approach to Web Applications. fev. 2005. Disponível em: <http://www.adaptivepath.com/publications/essays/archives/000385.php>. Acesso em: jan 2009.
- [2] SOARES, W. AJAX (Asynchronous Javascript And XML): Guia prático para Windows. São Paulo: Érica, 2006.
- [3] GOOGLE MAPS, aplicação para consulta de mapas de vários lugares do mundo. Disponível em: <http://maps.google.com/>. Acesso em: dez 2009.
- [4] GOOGLE GMAIL, aplicação de e-mail on-line do google. Disponível em: <http://www.gmail.com/>. Acesso em: jan 2009.
- [5] R. P. Gabriel, **Patterns of Software - Tales from the Software Community**, Oxford University Press, New York, 1996.
- [6] World Wide Web Consortium. Disponível em: <http://www.w3.org/>. Acesso em jan 2009.
- [7] CRANE, D.; PASCARELLO, E, JAMES, D. **Ajax em Ação**. Rio de Janeiro: Pearson Prentice Hall, p. 62-70, 2007.
- [8] HEWITT, Joe. Ajax Debugging with Firebug. Dr. Dobb 's Journal, no.393, pp. 22-26, fev. 2007.
- [9] SMITH, Keith. Simplifying Ajax Style Web Development. Computer Vol. 39, nº 5, pag 98-102, maio 2006.
- [10] Silverlight chart designer, demonstração de componente do Silverlight. Disponível em: http://www.visifire.com/silverlight_chart_designer.php. Acesso em jan 2009.

- [11] ZEPEDA, Sergio, CHAPA, Sergio V. From Desktop Applications Towards Ajax Web Applications. In: 2007 4th International Conference on Electrical and Electronics Engineering (ICEEE 2007), Mexico City, pp. 193-196, Sep. 2007.
- [12] W3C Document Object Model main page. <http://www.w3.org/DOM/>. Acesso em: fev. 2009.
- [13] ERIC J. Bruno. Ajax: Asynchronous JavaScript and XML. Dr. Dobb's Journal, Vol.31 (2), pp. 32-35. Feb. 2006.
- [14] PAULSON, Linda Dailey. Building Rich Applications With AJAX. Industrial Trends Journal, pp. 15-17, Out. 2005.
- [15] W3C Document Object Model main page. <http://www.w3.org/DOM/> Acesso em: jan. 2009.
- [16] DOM level 1 home. Disponível em: <http://www.w3.org/TR/REC-DOM-Level-1/introduction.html>. Acesso em: Fev 2009.
- [17] LIMEIRA, José Luiz Silveira, Utilização de AJAX no desenvolvimento de sistemas Web, Monografia de especialização. Universidade Federal do Rio Grande do Sul, 2006.
- [18] W3C eXtensive Markup Language definition. <http://www.w3.org/XML/>. Acesso em: fev. 2009.
- [19] GOOGLE SEARCH, Mecanismo de busca de página web. Disponível em: <http://www.google.com/>. Acesso em: jan 2009.
- [20] ASLESON, R.; SCHUTTA, N. **Fundamentos do AJAX**. Rio de Janeiro: Alta Books, 2006.
- [21] Why use SVG? Disponível em <http://www.adobe.com/svg/overview/whyuse.html>. Acessado em 28 de abril de 2006.

- [22] SVG Tutorial. Disponível em <http://www.w3schools.com/svg/default.asp>. Acessado em 28 de abril de 2006.
- [23] Web Center Features. Disponível em <http://www.adobe.com/svg/viewer/install/main.html>. Acessado em 28 de abril de 2006.
- [24] Scalable Vector Graphics specification. Disponível em <http://www.w3.org/Graphics/SVG/>. Acesso em: 28 de abril de 2006.
- [25] Silverlight Technical Articles. Disponível em: <http://msdn2.microsoft.com/en-us/library/bb871519.aspx>. Acesso em dez 2007.
- [26] MESBAH, A.; DEURSEN; A. V. An Architectural Style for AJAX. In (WICSA'07) Proceedings of the Working IEEE/IFIP Conference on Software Architecture, pp. 181-190, March 2007.
- [27] Document Object Model Core. Disponível em: <http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-1590626202>. Acesso em Fev 2009.
- [28] Document Object Model Events. Disponível em: <http://www.w3.org/TR/DOM-Level-2-Events/events.html>. Acesso em Fev 2009. Acesso em mar. 2009.
- [29] Microsoft Silverlight Documentation. Disponível em: <http://msdn.microsoft.com/en-us/library/bb871518.aspx>. Acesso em mar 2009.
- [30] SAJAX: Simple AJAX Toolkit. Disponível em: <http://www.modernmethod.com/sajax/>. Acesso em abr 2009.
- [31] DWR: Direct Web Remoting. Disponível em <http://directwebremoting.org/>. Acesso em abr 2009.
- [32] GWT Home. Disponível em: <http://code.google.com/webtoolkit/>. Acesso em: nov. 2007.
- [33] ECHO2 Home. Disponível em: <http://www.nextapp.com/products/echo2>. Acesso em: nov. 2007.

- [34] Java Server Faces Home. Disponível em <http://java.sun.com/javaee/javaserverfaces/>. Acesso em fev. 2009.
- [35] Struts Home. Disponível em <http://struts.apache.org/>. Acesso em: fev. 2009.
- [36] The Official Microsoft Asp.Net Portal. Disponível em: <http://www.asp.net/>. Acesso em abr 2009.
- [37] SVG Document Object Model. Disponível em: <http://www.w3.org/TR/SVG11/svgdom.html>. Acesso em mar. 2009.
- [38] Keep It Simple Home. Disponível em [:https://sourceforge.net/projects/keepitsimple/](https://sourceforge.net/projects/keepitsimple/). Acesso em maio 2009.
- [39] JQuery: A JavaScript library home. Disponível em: <http://jquery.com/>. Acesso em maio 2009.
- [40] Firebug: Firefox Add-on. Disponível em <http://getfirebug.com/>. Acesso em maio 2009.
- [41] Frameworks and AJAX patterns. Disponível em: <http://ajaxpatterns.org/wiki/index.php?title=AJAXFrameworks>. Acesso em: fev 2009.
- [42] Adobe Flex home. Disponível em: <http://www.adobe.com/products/flex/>. Acesso em maio 2009.