PONTIFÍCIA UNIVERSIDADE CATÓLICA DE SÃO PAULO

CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

PAULO VITOR DA SILVA RENDEIRO

CROSS-ORIGIN RESOURCE SHARING, UM ESTUDO PRÁTICO SOBRE O NOVO E REVOLUCIONÁRIO MECANISMO DE COMPARTILHAMENTO DE RECURSOS ENTRE APLICAÇÕES WEB

SÃO PAULO, SETEMBRO DE 2011

PAULO VITOR DA SILVA RENDEIRO

CROSS-ORIGIN RESOURCE SHARING, UM ESTUDO PRÁTICO SOBRE O NOVO E REVOLUCIONÁRIO MECANISMO DE COMPARTILHAMENTO DE RECURSOS ENTRE APLICAÇÕES WEB

Monografia apresentada ao Curso de Especialização em Engenharia de Software da Pontifícia Universidade Católica de São Paulo, como requisito parcial para obtenção do título de Especialista em Engenharia de Software, orientado pelo Prof. Dr. Carlos Eduardo de Barros Paes.

SÃO PAULO, SETEMBRO DE 2011

Dedico este trabalho a todos os professores que acreditam na educação como uma possibilidade de construir um mundo melhor e mais humano.

Agradeço primeiramente à minha família, sobretudo aos meus pais por estarem sempre presentes me apoiando e incentivando a alcançar meus objetivos.
Agradeço também a todos os mestres que de alguma forma colocaram um tijolo na minha parede de conhecimento. Prometo a vocês tornar essa parede uma muralha.

"Entre as recordações que cada um de nós guarda, algumas há que só contamos aos amigos. Há ainda outras que nem sequer aos amigos confessamos, que só a nós próprios dizemos e, mesmo assim, no máximo segredo. Finalmente, há coisas que o homem nem sequer se permite confessar a si mesmo. Ao longo da existência, toda a pessoa honesta acumulou não poucas destas recordações. Diria mesmo que a quantidade é tanto maior quanto mais honesto o homem. Eu, em todo o caso, não foi há muito que me decidi a recordar algumas das minhas antigas aventuras; até agora evitava fazê-lo, aliás com um certo desassossego. Porém agora, quando as evoco e desejo mesmo anotá-las, agora vou tirar a prova: será possível sermos francos e sinceros, pelo menos com nós próprios, e dizermo-nos toda a verdade?"

Fiodor Dostoievski

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

RESUMO

O compartilhamento de recursos entre aplicações web, como web services, é uma prática de mercado cada vez mais adotada, pois, além de permitir o reuso de serviços já construídos e consolidados, proporciona uma maior agilidade na criação de novas aplicações, conhecidas por aplicações Mashup. Até um passado bem recente, devido a uma limitação de segurança dos navegadores web, conhecida por Same-Origin Police, as aplicações Mashup necessitavam fazer o consumo de serviços externos na sua camada servidor, concentrando todo o trabalho de agragação de conteúdo em uma única arquitetura física. O presente trabalho tem como objetivo apresentar, de forma abrangente e prática, a especificação do mecanismo de compartilhamento, proposto pelo Consórcio W3C, denominado Cross-Origin Resource Sharing. Como resultado deste trabalho é esperado o desenvolvimento de um estudo de caso, capaz de comprovar o funcionamento deste novo mecanismo de compartilhamento. Por fim, com base nestes experimentos práticos, são elencados possíveis pontos de estudo futuro.

Palavras-chave: Compartilhamento, Cross-Origin Resource Sharing, Same-Origin-Police.

ABSTRACT

The sharing of resources among web applications is an increasingly accepted market practice, not just for allowing the reuse of already constructed and consolidated services, but also for providing greater productivity for the creation of new applications, known as Mashup applications. Even in a recent past, due to web browsers security limitation (Same-Origin Police) Mashup applications had to make use of external services on their server side, concentrating all the work of content aggregation into a single physic architecture. The present work has as objective to show, in a comprehensive and practice way, the specification of the sharing mechanism proposed by the W3C Consortium, called Cross-Origin Resource Sharing. A case study, able to demonstrate the operation of this new sharing mechanism, is expected as a result of the present work. Finally a list of some possible future studies is listed.

Keywords: Sharing, Cross-Origin Resource Sharing, Same-Origin-Police.

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

Com o crescimento e a popularização de APIs para o desenvolvimento de aplicações baseadas em Asynchronous Javascript and XML (AJAX), como jQuery, Prototype e ExtJS, é cada vez maior o número de aplicações Mashup, isto é, aplicações que combinam dados e códigos provenientes de diversas fontes espalhadas pela Internet, proporcionando a criação de inúmeros novos serviços e funcionalidades.

Quando uma aplicação Mashup é desenvolvida, encontram-se diversas dificuldades inerentes a realização de comunicações assíncronas, disparadas por um navegador web, contra serviços de outras origens. Neste contexto, as comunicações assíncronas devem ser inferidas como parte do estilo arquitetônico AJAX, e, uma origem, como a combinação de um protocolo, um domínio e uma porta [FLANAGAN, 2011].

O principal obstáculo na execução de uma chamada AJAX para um recurso de outra origem, técnica também conhecida por AJAX Cross-Domain, é o conceito de segurança Same-Origin Policy (SOP), introduzido pelo navegador Netscape 2.0 em Março de 1996 [PRATES, 2010]. Este conceito de segurança é aplicado a linguagens de programação que são executadas no navegador, como JavaScript, e limita o acesso à maioria das propriedades entre páginas de domínios diferentes, como funções JavaScript, cookies e dados de formulários. Tais restrições foram criadas para mitigar ações maliciosas contra usuários enquanto navegam por aplicações web. Alguns dos ataques mais conhecidos são o Cross-site scripting (XSS) e o Cross-site request forgery (CSRF/XSRF) [FLANAGAN, 2011].

A política de mesma origem, devido a sua característica de forte restrição, maximizou consideravelmente a segurança dos navegadores e é adotada em todos os navegadores web

modernos. Porém, a comunicação AJAX entre origens distintas não é por natureza insegura ou mal, é de fato essencial para muitas das aplicações mais populares e utilizadas do mundo [JSON-P, 2008].

Diante da emergente necessidade da troca de informações entre aplicações de diferentes origens via AJAX, foram desenvolvidos diversos mecanismos para contornar as limitações impostas pelo Same-Origin Policy. Alguns dos mais conhecidos são:

A. Equiparação de domínio: este método iguala os domínios das aplicações através da propriedade "document.domain", permitindo assim a execução de requisições AJAX. Esta técnica, porém, só funciona para cenários de cross-subdomain.

B. Cross-Origin Messaging: especificada pelo HTML 5, essa é uma nova forma de comunicação assíncrona entre janelas do navegador, inclusive para documentos carregados em iframes. Sua utilização se dá pela invocação da função "window.postMessage". Pela sua finalidade, entretanto, esta técnica não prevê suporte para comunicação AJAX.

C. Server-Side Proxy: consiste em efetuar as chamadas AJAX para uma URL da sua própria aplicação e, no lado servidor, fazer a chamada para o serviço cross-domain. Este mecanismo atende ao cenário de cross-domain para requisições que utilizem os mais variados verbos HTTP, porém sua implementação, além de intrusiva, demanda muito esforço de configuração e aumenta a complexidade de implantação da aplicação.

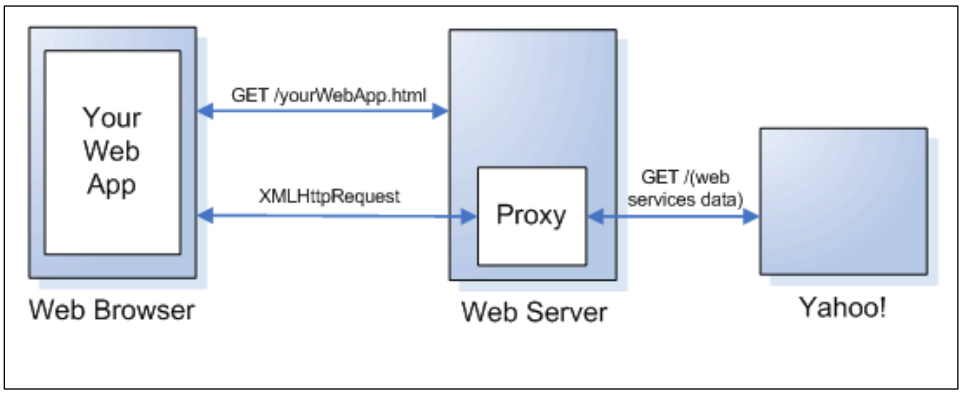


Figura 1: Exemplo de Server-Side Proxy [YAHOO DEVELOPER, 2011]

A Figura 1: Exemplo de Server-Side Proxy apresenta o fluxo de uma comunicação assíncrona entre origens distintas através do emprego da técnica Server-Side Proxy.

D. JavaScript Object Notation with Padding (JSONP): esta técnica utiliza a simples técnica do servidor retornar um código Javascript que passa um JSON para uma função de callback provida pelo lado cliente via parâmetro da requisição. Este método funciona para cenários cross-domain, porém apenas com requisições que utilizam GET como verbo HTTP, além de não ser aderente ao modelo de comunicação AJAX.

1.2 Problema

Atualmente para implementarmos um diálogo cross-domain via AJAX temos apenas o Server-Side Proxy como alternativa técnica disponível, sendo que sua adoção implica em uma importante consequência arquitetural, a centralização do processamento das solicitações e respostas em uma única arquitetura física. A falta de um modelo robusto, padronizado e distribuído de comunicação entre origens dissemelhantes gera uma lacuna que muitas vezes é preenchida por soluções muito complexas, intrusivas, não escaláveis e que subutilizam o poder computacional disponível nas máquinas clientes da aplicação web.

1.3 Problematização

Com o contexto mencionado, questiona-se:

A. Como descentralizar a comunicação entre aplicações de diferentes origens potencializando sua escalabilidade?

B. Como abstrair a complexidade no desenvolvimento de sistemas que necessitem consumir, através de requisições AJAX, serviços expostos em diversos domínios?

C. Como expor serviços na internet que possam ser consumidos apenas por origens autorizadas?

D. Como fazer o controle do nível de acesso de acordo com o recurso solicitado e origem da requisição?

E. Como estender a API de chamadas AJAX, disponibilizada pelos navegadores web, para saber trabalhar requisições cross-domain mantendo sua interface original?

F. Como tornar possível o uso dos mais variados verbos HTTP, como: GET, POST, PUT, DELETE e OPTIONS, na realização de requisições AJAX cross-domain?

1.4 Objetivos

1.4.1 Objetivos gerais

O objetivo deste projeto é apresentar a especificação Cross-Origin Resource Sharing (CORS), desenvolvida pelo pelo consórcio W3C para o compartilhamento de recursos consumíveis por aplicações de outros domínios através de chamadas AJAX, estendendo o conceito de segurança Same-Origin Policy.

1.4.2 Objetivos específicos

Os objetivos específicos deste projeto, que visa abordar a especificação CORS, são:

A. Detalhar o funcionamento teórico da especificação CORS, apresentando toda a mecânica envolvida no diálogo entre as aplicações envolvidas em uma comunicação cross-domain;

B. Desenvolver um estudo de caso para demonstrar a aplicabilidade prática do CORS. Este estudo de caso deverá abranger o desenvolvimento passo a passo de duas aplicações:

- Aplicação api-provider, cuja finalidade será disponibilizar serviços RESTful consumíveis por diversas origens;
- Aplicação api-consumer, cujo objetivo será fazer o consumo, via AJAX, dos serviços disponibilizados pela aplicação api-provider.

C. Informar a atual aderência dos principais navegadores web para com a especificação CORS, indicando quais características já se tornaram realidade e quais ainda são tendência.

1.5 Relevância

Este projeto visa simplificar e melhorar a qualidade do desenvolvimento de aplicações que efetuem interações com aplicações externas as suas fronteiras de domínio utilizando o conceito de requisições assíncronas AJAX.

A simplificação deste desenvolvimento é dada através da minimização do esforço de configurações externas ao escopo da aplicação, como intervenções em Web Servers. A

remoção desta dependência arquitetural é de grande valia em ambientes corporativos pois, como consequência, diminuímos os esforços delegados a área de infra-estrutura.

A forma semi transparente com que a solução apresentada é aplicada também torna os elementos participantes da comunicação cross-domain mais coesos, ou seja, eles não precisam saber nada além do que solicitar e do que responder. Toda negociação de acesso aos recursos fica apartada e abstraída, tornando o código do cliente e do servidor mais limpo. Isso maximiza a legibilidade do código e o seu nível de manutenabilidade.

Como resultado das vantagens apresentadas previamente, este projeto tem como finalidade contribuir com a disseminação de conhecimento da especificação CORS, proporcionando melhorias de qualidade do código, aumento na segurança das informações trafegadas e redução de complexidade e de recursos necessários para o desenvolvimento, como tempo e pessoas.

2 REVISÃO BIBLIOGRÁFICA

2.1 Introdução

Este capítulo apresenta uma revisão bibliográfica acerca dos principais conceitos e fundamentos necessários para plena compreensão da solução técnica exposta nos capítulos subsequentes.

2.2 Hypertext Transfer Protocol (HTTP)

A. Definição

Hypertext Transfer Protocol é um protocolo de camada de aplicação para sistemas de informação distribuídos, colaborativos e de hipermedia. Este protocolo está em uso desde 1990, quando foi incorporado a iniciativa de informação global World Wide Web, que em português significa Rede de alcance mundial.

O HTTP é genérico e não mantem estado, tornando-o aplicável para as mais diversas tarefas além da sua utilização para hipertexto, tais como serviços de nomes e sistemas de gerenciamento de objetos distribuídos, através da extensão dos seus métodos de requisição, códigos de erro e cabeçalhos. Suas principais características, no entanto, são a tipagem e a negociação de representação de dados, permitindo que os sistemas possam ser construídos independentemente dos dados que serão trafegados [W3C - HTTP/1.1, 2011].

B. Como funciona uma transação HTTP?

O HTTP define uma forma de conversação no estilo pedido-resposta. Por exemplo, um navegador web realiza uma requisição para um servidor, tipicamente abrindo uma conexão TCP/IP. Assim que a conexão TPC é estabelecida, o navegador envia a requisição HTTP através da interface de Socket desta conexão. Uma vez que esta requisição chega a esta interface de Socket, a mensagem sai da jurisdição do cliente e passa a ser gerenciada pelo protocolo de transporte TCP, que neste contexto, tem como principal responsabilidade garantir que a mensagem chegue intacta ao seu destinatário. O servidor recebe está mensagem também pela interface de Socket da conexão TCP, realiza os processamentos internos inerentes a requisição recebida, retorna o resultado deste processamento para a interface de Socket e fecha a conexão TCP assim que o cliente termina de receber a mensagem enviada [GOURLEY et al., 2002].

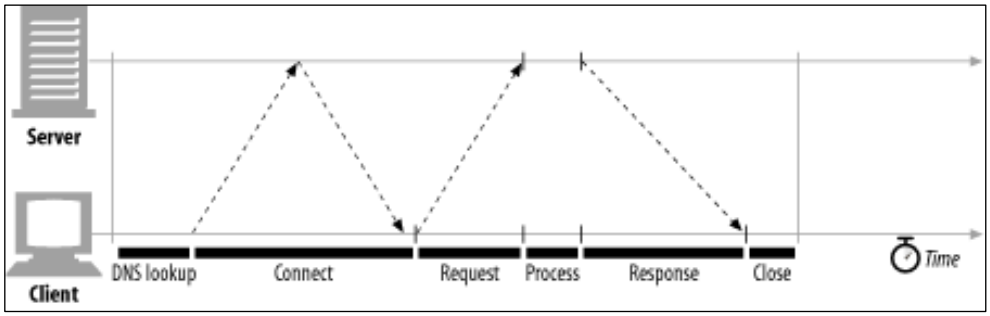


Figura 2: Linha do tempo de uma transação HTTP [GOURLEY et al., 2002]

C. Histórico

O HTTP, que hoje é robusto e predominante no mundo da Internet, iniciou com definições extremamente simples. Inicialmente, existia apenas um método (GET) de requisição, e não haviam cabeçalhos ou códigos de status. O servidor simplesmente retornava um documento HTML. Esta era a única operação passível de ser realizada na versão 0.9 do protocolo HTTP, que foi definida por Tim Berners-Lee em 1991.

Em 1995, Dave Raggett liderou o Grupo de Trabalho da especificação HTTP, com o intuito de expandir o protocolo estendendo suas operações e negociações, enriquecendo suas meta-informações, e adicionando campos de cabeçalho. Como resultado deste trabalho, a RFC 1945 oficialmente introduziu e reconheceu, no ano de 1996, a versão 1.0. Esta versão, porém, não permitia o uso de conexões persistentes ou configuração de hosts virtuais em servidores web. Consequentemente, em 1997, foi lançada a versão 1.1 do protocolo HTTP. Esta versão continua sendo usada até os dias de hoje.

Method	HTTP/0.9	HTTP/1.0	HTTP/1.1
CONNECT			•
DELETE		•	•
GET	•	•	•
HEAD		•	•
LINK		•	
POST		•	•
PUT		•	•
OPTIONS			•
TRACE			•
UNLINK		•	

Figura 3: Métodos por versão do protocolo HTTP [THOMAS, 2001]

2.3 Navegadores web

A. Definição

Navegador web, em inglês conhecido como Web browser ou apenas browser, é um programa que permite a seus usuários a interagirem com documentos eletrônicos de hipertexto, como as páginas HTML e que estão armazenados em algum endereço eletrônico da internet.

Tendo como premissa a arquitetura do protocolo HTTP, a finalidade principal de um navegador web é ser um cliente capaz de abstrair pedidos de recursos e trabalhar respostas provenientes de servidores web. Vale ressaltar que os navegadores atuais trabalham com vários outros protocolos de aplicação, como FTP, HTTPS etc.

B. Histórico

Até a recente ascensão de alternativas como Firefox, Chrome, Safari e Opera, a grande maioria das pessoas associavam o navegador web ao Internet Explorer. Mas ao contrário do que muitos imaginam, o Internet Explorer não foi nem de perto o primeiro navegador web da história. O primeiro navegador web, inicialmente chamado de WorldWideWeb e depois renomeado para Nexus, foi escrito na linguagem Objective-C por Tim Berners-Lee, também criador da primeira versão do protocolo HTTP, no ano de 1990 [ASLESON; SCHUTTA, 2006a].

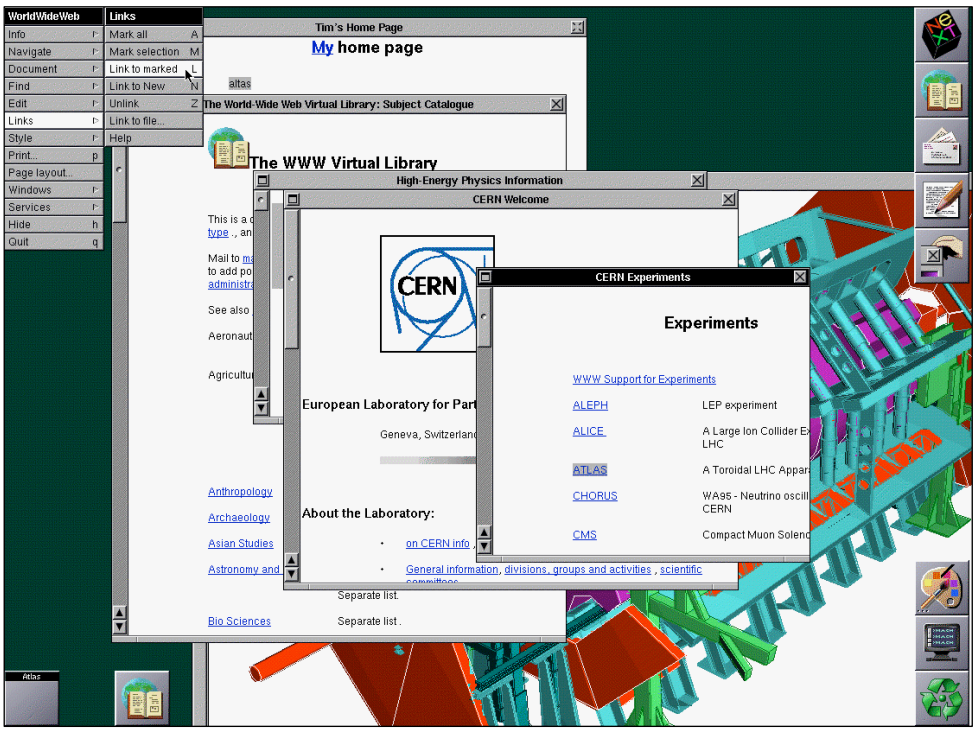


Figura 4: Interface do navegador Nexus em 1993 [BERNERS-LEE, 2011].

Ainda no ano de 1990, Berners-Lee e Jean-Francois Groff, portaram o Nexus para a linguagem C, renomeando o navegador para libwww.

Em Fevereiro de 1993, Marc Andreessen e Eric Bina do Centro Nacional para aplicações de supercomputação da Universidade de Illinois lançaram o Mosaic para Unix. Alguns meses depois, o Mosaic se tornou o primeiro navegador multi-plataforma quando Aleks Totic apresentou a versão para Macintosh. Rapidamente o Mosaic se tornou o navegador web mais popular. Sua tecnologia foi então licenciada para uma empresa, também de Illinois, chamada Spyglass que, posteriormente, licenciou o Mosaic para a Microsoft.

Desenvolvedores da Universidade de Kansas escreveram no ano de 1993 um navegador modo texto chamado Lynx que tornou-se um padrão nos terminais de servidores e mainframes. No ano de 1994, na cidade de Oslo, Noruega, um time desenvolveu a primeira versão do Opera, que foi disponibilizado no ano de 1996. Em Dezembro de 1994, a Netscape lança a versão 1.0 do Mozilla, e do primeiro navegador com fins lucrativos. Apenas em 2002 uma versão de código aberto foi lançada, se tornando o ponto de partida para o popular Firefox, lançado em Novembro de 2004 [ASLESON; SCHUTTA, 2006a].

Quando a Microsoft lançou o Windows 95, o Internet Explorer 1.0 vem como parte do pacote Microsoft Plus!. Na versão 2.0, lançada também no ano de 1995, houveram melhorias

significativas como suporte para cookies, Secure Socket Layer (SSL), e outros padrões emergentes. Esta versão foi disponibilizada também para Macintosh, tornando-se o primeiro navegador multi-plataforma da Microsoft.

No ano de 1996, a Microsoft lança a versão 3.0 do Internet Explorer e, virtualmente da noite para o dia, as pessoas migraram em bando para o Internet Explorer, uma vez que a Netscape cobrava pelo seu navegador. Finalmente, em 1999, a versão 5 do Internet Explorer é lançada, se tornando o navegador mais usado do mundo.

2.4 Javascript

A. Definição

Javascript é uma linguagem de script de fácil aplicabilidade, alta expressividade e desenvolvida para possibilitar o dinamismo de aplicações web. Javascript foi projetado para ser usado por autores de páginas HTML e desenvolvedores de aplicativos corporativos para, em tempo de execução, definir comportamentos para objetos no lado cliente ou servidor. Sua relação direta com o navegador a torna esta linguagem muito importante e, consequentemente, uma das linguagens de programação mais populares do mundo [CROCKFORD, 2008].

B. Histórico

O Javascript foi originado pela Netscape com o nome de Livescript, desenvolvido paralelamente com o software de servidor Livewire. Seu desenvolvimento foi realizado com algumas finalidades. A primeira era enriquecer páginas da web de uma forma que o HTML não era capaz. O exemplo clássico desta necessidade era a validação das entradas realizadas por um usuário em um formulário. Era importante que uma validação acontecesse ainda no lado cliente, para que só posteriormente esses dados fossem enviados para o servidor. Outra finalidade do Livescript foi criar uma forma de comunicação entre documentos HTML e applets Java.

Em Dezembro de 1995, a Sun Microsystems assumiu o desenvolvimento do Livescript e mudou o seu nome para Javascript. No ano seguinte, em 1996, a Microsoft introduziu navegador web capaz de interpretar Javascript, o Internet Explorer 3. Uma semana após este lançamento, a Netscape lançou o seu navegador web com suporte a Javascript, o Netscape

Navigator 3. O Internet Explorer estava longe de alcançar o nível de suporte a Javascript que o Navigator 3 possuia, ficando atrás em muitas características e capacidades. No primeiro semestre de 1997, a Netscape lança o Navigator 4, versão conhecida como “Communicator”. Neste ponto a Microsoft ainda estava longe de atingir a maturidade do Navigator 3.

A linguagem Javascript passou a ser totalmente suportada pelo Internet Explorer e pelo Netscape Navigator pelas respectivas versões 5.5 e 6.0 [EASTTOM, 2008].

2.4.1 XMLHttpRequest (XHR)

A. Definição

O objeto XMLHttpRequest é a implementação de uma interface exposta por um mecanismo de script para realizar funcionalidades inerentes a um cliente HTTP, como enviar dados de um formulário ou recuperar informações de um servidor.

O nome deste objeto é XMLHttpRequest por questões de compatibilidade com antigas implementações deste componente implementadas por diversos fabricantes de navegadores web. Ainda sobre seu nome, apesar do objeto XMLHttpRequest iniciar com o acrônimo XML, ele suporta qualquer formato baseado em texto, como por exemplo uma String, um JSON, ou mesmo um XML.

Por fim, este componente é capaz de realizar apenas requisições HTTP e HTTPS, sendo assim, qualquer outro protocolo não é coberto por sua especificação [W3C - XHR, 2011].

B. Histórico

O XMLHttpRequest foi implementado originalmente no Internet Explorer 5 como um componente do ActiveX, o que tornou sua adoção tímida por parte dos desenvolvedores. Sua utilização apenas ganhou espaço quando outros navegadores, como Mozilla 1.0 e Safari 1.2, o incorporaram entre os anos de 2003 e 2004. É importante destacar que o componente XMLHttpRequest não nasceu como um padrão do W3C, logo, até poucos anos atrás sua implementação possuía comportamentos muito variados entre os diversos navegadores. Atualmente, Firefox, Safari, Opera, Chrome, Konqueror e Internet Explorer possuem implementações muito similares e já existe um trabalho em andamento, iniciado em 2006, de especificação deste componente pelo W3C [ASLESON; SCHUTTA, 2006b].

Para ser utilizado, primeiramente devemos criar um objeto XMLHttpRequest, para então ser possível enviar requisições e processar respostas. Por ainda não ser um componente devidamente padronizado, sua criação ainda depende de uma verificação do navegador. No demais, com relação a sua interface de utilização, não existem grandes diferenças entre os navegadores mais populares do mercado.

2.4.2 Same-Origin Policy (SOP)

A. Definição

O Same-Origin Policy, ou Política de mesma origem, é uma restrição de segurança que define quais conteúdos web podem ser manuseados com código Javascript. Um código Javascript tem acesso apenas a propriedades de janelas e documentos que possuam a mesma origem que o documento que contem este script. A origem de um documento é definida através da junção do protocolo, domínio e porta da URL de onde o documento foi carregado. Documentos carregados de diferentes servidores web possuem origens diferentes. Documentos carregados através de diferentes portas do mesmo domínio possuem origens diferentes. E, por fim, um documento carregado com o protocolo HTTP possui uma origem diferente de um carregado via protocolo HTTPS, mesmo que estes sejam carregados do mesmo servidor web [FLANAGAN, 2011].

URL	Outcome	Reason
http://store.company.com/dir2/other.html	Success	
http://store.company.com/dir/inner/another.html	Success	
https://store.company.com/secure.html	Failure	Different protocol
http://store.company.com:81/dir/etc.html	Failure	Different port
http://news.company.com/dir/other.html	Failure	Different host

Tabela 1: Validação de SOP partindo da origem http://store.company.com [MOZILLA, 2011].

B. Como funciona?

É importante entender que a origem de um script, por si só, não é relevante para o SOP. O que importa é a origem do documento no qual o script está inserido. Suponha, por exemplo, que um script hospedado no servidor X seja incluído, através da propriedade “src” do elemento “<script>”, em uma página hospedada no servidor Y. A origem do script é o servidor Y e o script possui acesso integral ao conteúdo do documento que o contem. Se o script abrir uma nova janela e carregar um segundo documento do servidor Y, o script

também terá acesso completo ao conteúdo do segundo documento. Porém, se o script abrir uma outra janela e carregar um documento de um servidor Z, ou mesmo do servidor X, o SOP irá barrar o script de acessar as propriedades deste documento.

O SOP não é aplicado a todas as propriedades de todos os objetos em uma janela ou documento de outra origem. Porém, ele é aplicado para a grande maioria destas propriedades, e, em particular, aplicado a praticamente todas as propriedades do objeto "Document". É importante considerar qualquer janela que contenha um documento carregado de outro servidor fora da fronteira de acesso de seus scritps. No máximo, caso o seu script tenha aberto a janela, ele será capaz de fechá-la, mas não será capaz de observar nada dentro dela.

O SOP é necessário para prevenir scripts de roubarem informações proprietárias. Sem esta restrição, um script malicioso poderia abrir uma janela vazia, esperar que o usuário acessasse algum site bancário, ou qualquer outro, e então seria capaz de ler o conteúdo carregado e enviar estas informações para seu servidor de origem. A grande importância da Política de mesma origem é prevenir este tipo de comportamento.

C. Same-Origin Policy para XMLHttpRequest

Como dito anteriormente, o XMLHttpRequest serve para que códigos em Javascript possam realizar requisições HTTP ou HTTPS para os seus servidores de origem e ler suas respostas. Com essa premissa em mente, vamos ressaltar o conjunto de características relevantes, em termos de segurança, pelo objeto XMLHttpRequest:

- Habilidade de especificar um verbo HTTP arbitrário em uma requisição (via método "open");
- Habilidade de definir cabeçalhos customizados em uma requisição (via método "setRequestHeader");
- Habilidade de leitura completa dos cabeçalhos recebidos em uma resposta (via métodos "getResponseHeader" e "getAllResponseHeaders");
- Habilidade de leitura completa do corpo da resposta (via propriedade "responseText").

Uma vez que todos as requisições realizadas pelo objeto XMLHttpRequest possuem a capacidade de envio de cookies e são extremamente flexíveis na interação com componentes do lado servidor, é extremamente importante existirem mecanismos de segurança que limitem este componente. O conjunto de verificações implementadas por todos os navegadores web

para o objeto XMLHttpRequest é uma pequena variação do Same-Origin Policy aplicado a componentes DOM. Essas variações são:

- Verificação do destino da requisição, independente da propriedade "document.domain", tornando impossível para sites terceiros mutuamente concordarem em realizar requisições cross-domain entre eles;
- Em algumas implementações, existem restrições adicionais sobre protocolos, cabeçalhos e verbos HTTP disponíveis;
- No Internet Explorer, embora a porta não seja considerada para verificação do Same-Origin Policy no acesso a propriedades DOM, a porta é considerada para requisições XMLHttpRequest [ZALEWSKI, 2011].

2.5 JavaScript Object Notation (JSON)

A. Definição

JSON é um formato de texto leve para o intercâmbio de dados. Este formato possui como características a simplicidade, tornando sua leitura e escrita fáceis, e a sua independência das linguagens de programação, uma vez que sua composição é baseada em duas estruturas de dados suportadas, virtualmente, por todas as linguagens de programação modernas:

- Uma coleção de pares chave/valor. Representado, em linguagens de programação modernas, por objetos, estruturas, dicionários, tabelas de hash, listas chaveadas ou arrays associativos.
- Uma lista ordenada de valores. Representada, na maioria das linguagens, por arrays, vetores, listas ou sequências.

Uma vez que estas estruturas são disponibilizadas por muitas linguagens de programação, JSON se torna um formato ideal para integração de sistemas. Adicionalmente, uma vez que o JSON é baseado em um subconjunto do padrão Javascript, ele é compatível com praticamente todos os navegadores web.

```
{"planet": {
  "name": "earth",
  "type": "small",
  "info": [
    "Earth is a small planet, third from the sun",
    "Surface coverage of water is roughly two-thirds",
    "Exhibits a remarkable diversity of climates and landscapes"
  ]
}}
```

Figura 5: Exemplo de representação JSON [CRANE; PASCARELLO; JAMES, 2006].

B. Estrutura de um objeto JSON

Um objeto JSON é um conjunto não ordenado de pares chave/valor. Estes objetos são iniciados com { (chave esquerda) e termina com } (chave direita). Uma vírgula separa os pares chave/valor. Um array JSON é uma coleção ordenada de valores iniciada com [(colchete esquerdo) e terminada com] (colchete direito). Uma vírgula separa os valores do array. Um valor por ser uma String, encapsulada por aspas duplas, um número, valores booleanos “true”ou “false”, um objeto ou um array. Isto permite a criação de estruturas aninhadas.

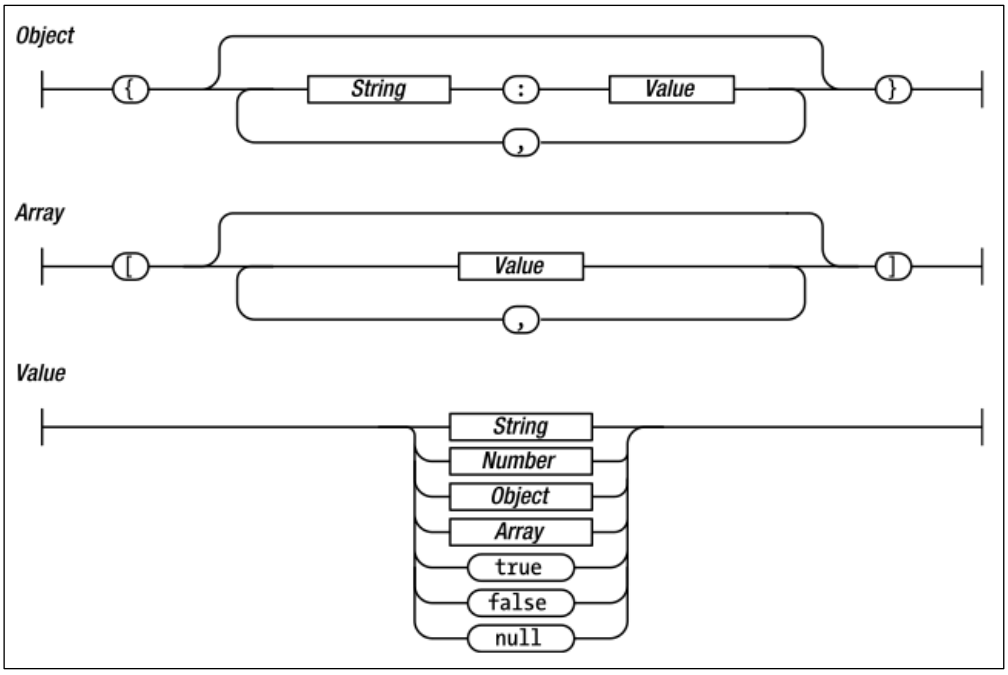


Figura 6: Estrutura de objetos JSON [ASLESON; SCHUTTA, 2006a].

2.6 JSON with Padding (JSONP)

A. Definição

JSONP é uma forma largamente utilizada para realização de requisições cross-domain. Diferentemente do objeto XMLHttpRequest, o JSONP não é restrito ao Same-Origin Policy, porém está técnica é restrita ao uso do verbo HTTP GET, o que implicitamente limita os dados trafegados a 4KB e impossibilita o seu uso no tráfego de dados sensíveis.

B. Como funciona?

O JSONP funciona baseado na recepção de um fonte Javascript que passa um objeto JSON para uma função de callback cujo nome é provido como parâmetro na requisição. Isto significa que a sua função de callback deve existir em contexto global antes da realização da chamada ao servidor.

Caso o servidor destino da requisição esteja sob seu domínio, você está em um ótimo cenário. Porém, se o destino da sua requisição for um fornecedor externo que você não confie totalmente, podem ocorrer problemas. Isto porque você não pode fazer uma verificação antes da invocação da função de callback, ou seja, você terá que confiar integralmente no conteúdo recebido [PORTENEUVE, 2010].

2.7 Extensible Markup Language (XML)

A. Definição

Criado pelo W3C, o XML é um formato, flexível e verboso, para dados e documentos. A palavra Markup, em português marcação, é o termo que denota a forma como é composta a estrutura do documento. XML tem suas raízes na Standard Generalized Markup Language (SGML), metalinguagem através da qual se pode definir linguagens de marcação para documentos. Uma outra importante linguagem derivada do SGML é o HTML [HOLDENER, 2008].

```
<?xml version="1.0" ?>
<!- Recipes from thehelpfulcook.com -->
<recipes>
   <recipe>
      <title>Grilled Steak Sauce</title>
      <author>Tanya Evans</author>
      <ingredients>
         <ingredient>1/2 cup soy sauce</ingredient>
         <ingredient>1/2 cup plum sauce</ingredient>
         <ingredient>1/2 cup pineapple juice</ingredient>
      </ingredients>
  </recipe>
</recipes>
```

Figura 7: Exemplo de documento XML [CEPONKUS; HOODBHOY, 1999].

B. Histórico

O XML foi criado para que documentos pudessem ser lidos por máquinas da mesma forma que um documento HTML é lido por seres humanos, ou seja, deveria prover uma sintaxe comum para que documentos pudessem ser acessíveis por todos os usuário. Sua versão corrente nas recomendações do W3C é a 1.1 (segunda edição), publicada em Setembro de 2006 e, até os dias correntes, a última versão lançada. Apesar disso, a grande maioria dos documentos XML são da versão 1.0. Isto se deve ao fato da versão 1.1 ter feito pequenas alterações na especificação.

Diferentemente do HTML, o XML vem com um conjunto muito pequeno de predefinições. Desenvolvedores HTML estão habituados com a noção de utilização dos caracteres < (menor) e > (maior) para definição de elementos, e com o subconjunto de nomes dos elementos, como body, head, etc. O XML compartilha apenas o conceito de definição de elementos utilizando os caracteres de menor e maior, não possuindo elementos predefinidos. Desta forma, o XML é meramente um conjunto de regras que nos permite a escrita de outras linguagens, como o próprio HTML.

O fato do XML especificar poucas restrições torna a sua aceitação menos áspera. Sua adoção é similar a aceitação de um alfabeto, que contém o seu conjunto de acentuações, símbolos e outros, porém, em nenhum momento especifica-se que linguagem utilizar. Esta característica proporciona uma imensa flexibilidade para comunicação entre sistemas [HOLDENER, 2008].

2.8 Asynchronous Javascript And XML (AJAX)

A. Definição

AJAX é uma forma de desenhar e construir páginas web de forma que estas sejam tão interativas e responsivas quanto uma aplicação desktop. Isto significa que são realizadas diversas alterações no estado da página, sendo que estas alterações são administradas, quase que na sua totalidade, no navegador. Isto é possível em virtude das requisições assíncronas que permitem que o usuário continue trabalhando ao invés de permanecer esperando uma resposta. Outro benefício desta técnica é a atualização sob demanda da interface, tornando esta tarefa menos custosa [RIORDAN, 2008].

B. Princípios

O AJAX, apesar de ser um conceito que utiliza diversas tecnologias já existentes, alterou o modelo tradicional de construção de aplicações web. Esta transformação presume uma remodelagem nas nossas premissas e pensamentos para que possamos extrair o melhor do AJAX. Existem quatro princípios fundamentais que devemos reaprender, são eles:

1. O navegador web hospeda uma aplicação, não conteúdo;

   No clássico modelo de aplicação web baseado em páginas, a cada interação do usuário com o site, outro documento é enviado ao navegador, contendo a mesma mistura de conteúdo imutável e dados. O navegador simplesmente jogava o antigo documento fora e apresentava o novo, ou seja, não utiliza-se o poder computacional disponível no cliente.

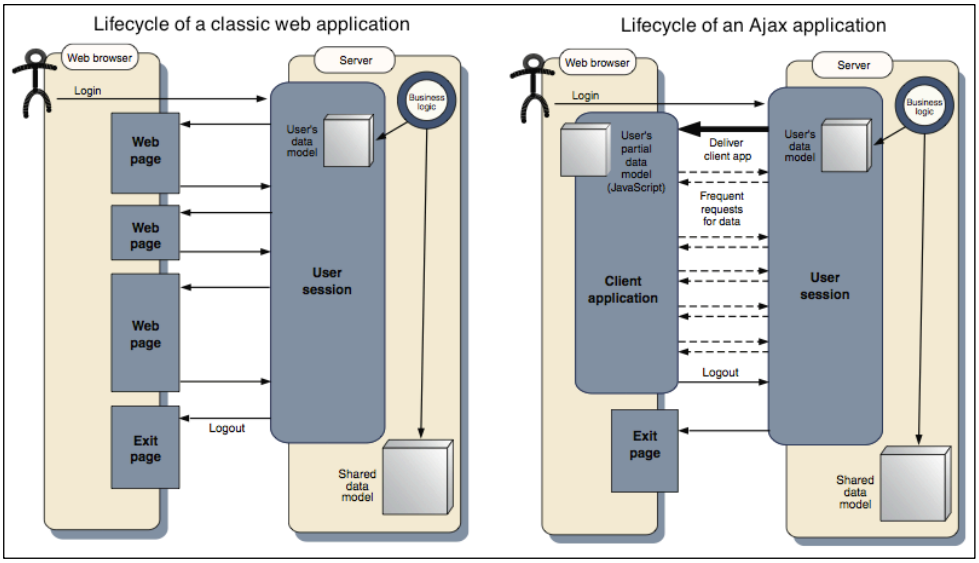


Figura 8: Ciclo de vida de aplicações web tradicionais e AJAX [CRANE; PASCARELLO; JAMES, 2006].

No modelo de aplicação AJAX, partes da lógica da aplicação são movidos para o navegador web. Desta forma, o documento irá ser carregado apenas uma vez e perdurará durante toda sessão do usuário, sendo alterado continuamente, e de forma assíncrona, de acordo com as interações que o usuário for realizando com a aplicação.

2. O servidor entrega dados, não conteúdo;

Como observado no primeiro princípio, no modelo de aplicação web AJAX o documento é carregado na primeira requisição e, nas requisições subsequentes, apenas dados são trafegados. Esta característica reduz drasticamente o consumo de banda necessário para o funcionamento pleno da aplicação, uma vez que a maior parte da banda consumida nesta tarefa é dedicada a elementos de interface.

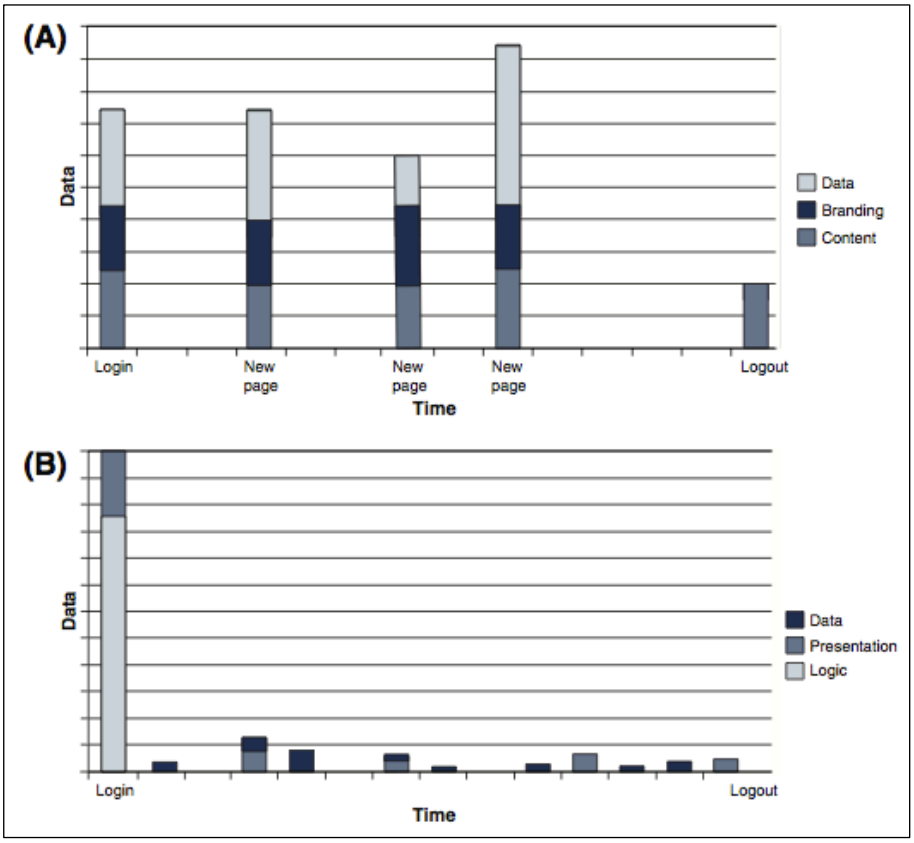


Figura 9: Consumo de banda entre aplicações web tradicionais e AJAX [CRANE; PASCARELLO; JAMES, 2006].

Em aplicações AJAX, existe um esforço inicial para o carregamento maciço dos componentes de interface no navegador do usuário, porém, em todas as demais interações da aplicação com o servidor apenas dados são intercambiados. Este

mecanismo agiliza a comunicação entre cliente e servidor, tornando a aplicação mais responsiva, melhorando a experiência do usuário.

3. A interação do usuário com a aplicação pode ser flexível e contínua;

Este princípio sugere o apontamento de formulários e links para funções Javascript aderentes a comunicação assíncrona. Esta modificação evita que o navegador realize uma requisição síncrona onde todo o documento tenha que ser renderizado novamente após o recebimento da resposta do servidor. A consequência do uso consistente deste princípio é uma navegação contínua por parte do usuário. A idéia principal é não tirar o usuário do seu fluxo de trabalho devido ao processamento de um evento.

4. A real codificação requer disciplina.

O código que é entregue quando o usuário carrega a aplicação deve ser capaz de rodar até que o usuário feche-a, sem quebra, sem ficar lenta e sem gerar vazamento de memória. Fazer esta filosofia funcionar demanda esforço e disciplina para absorção e entendimento da divisão das responsabilidades que cada camada da aplicação deve possuir.

C. Como funciona?

Diferentemente da abordagem pedido/resposta que encontramos em um cliente web padrão, uma aplicação AJAX possui uma mecânica um pouco diferente.

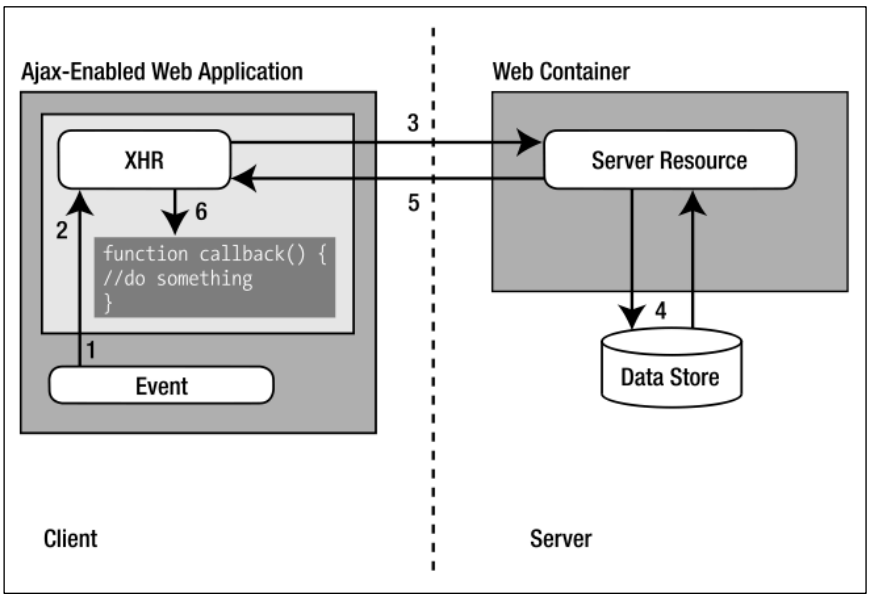


Figura 10: Modelo de interação AJAX [ASLESON; SCHUTTA, 2006a].

Podemos definir uma interação AJAX através dos seguintes passos:

1. Um evento ocorre na aplicação, neste momento hospedada no navegador do usuário, o que dispara um evento AJAX. Podemos exemplificar esse disparo através de eventos de elementos HTML como onchange, onblur etc;
2. A função Javascript, invocada pelo evento do passo 1, cria uma instância do objeto XMLHttpRequest e, usando o método open() define a URL e o verbo HTTP que serão utilizados na requisição. Na sequência, é feita a invocação do método send();
3. A requisição é disparada para o servidor, que pode ser uma Servlet, um script CGI, ou qualquer outra técnica semelhante;
4. O servidor processa a requisição, muitas vezes fazendo acesso a fontes de dados ou mesmo outros sistemas;
5. Uma resposta é enviada para navegador. Esta resposta pode conter dados em qualquer variação de formatos de texto, como por exemplo “text/xml” e “text/html”, deste modo a resposta pode ser interpretada pelo objeto XMLHttpRequest. Neste passo é importante definir cabeçalhos na resposta do servidor para que o navegador não faça cache da resposta obtida.
6. Uma vez que o objeto XMLHttpRequest tenha a resposta em mãos, ele irá invocar a função de callback definida no momento de sua criação. Esta função tem a

responsabilidade de saber trabalhar os dados enviados pelo servidor e, caso necessário, de atualizar elementos da interface da aplicação.

2.9 Cross-site scripting (XSS)

A. Definição

Cross-site scripting (XSS) é uma forma comum de ataque na web que consiste na inclusão de scripts maliciosos, ou outro tipo de código, na resposta HTTP que são executados involuntariamente pelo navegador do usuário. Estes tipos de ataque podem ter diversas variações e podem ser extremamente perigosos. Uma das consequências deste ataque pode ser o roubo de dados privados, como cookies. Este procedimento pode ser feito redirecionando o navegador do usuário para um web site controlado pela pessoa que gerou o ataque [WELLS, 2007].

B. Como funciona?

Tipicamente o ataque possui uma das seguintes formas: persistido ou refletido. Em um ataque persistido, o agressor armazena seu script no servidor, geralmente em um banco de dados. Posteriormente, quando a vítima navegar pele web site, o código malicioso é enviado ao navegador deste usuário e então o ataque é realizado. Em um ataque refletido, o agressor insere o script malicioso em um parâmetro da requisição ou na QueryString e passa o link para a vítima.

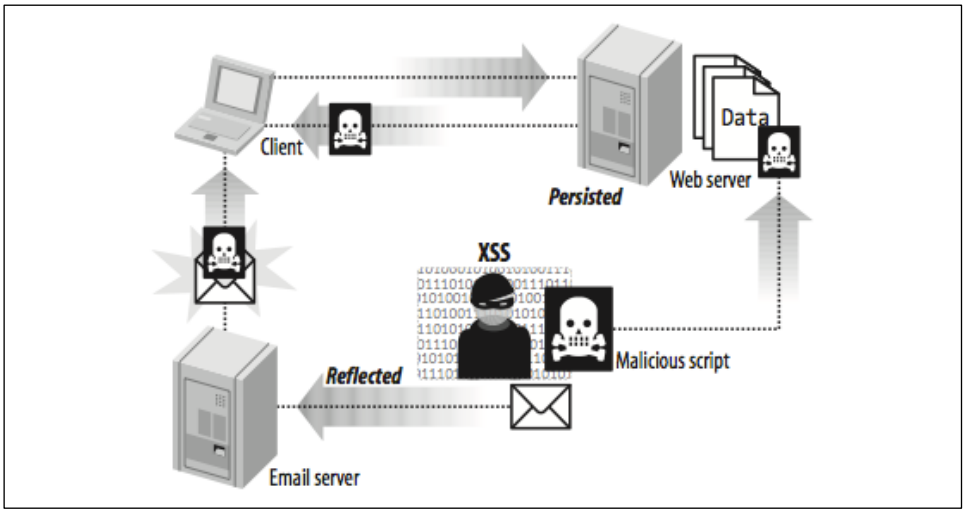


Figura 11: Cross-site scripting persistido e refletido [WELLS, 2007].

O XSS permite a apresentação de conteúdo HTML indesejado, executa arbitrariamente scripts (como Javascript ou VB Script) ou códigos maliciosos embarcados (em Applets, ActiveX ou Flash). Ataques XSS podem resultar em exposição de dados, deformação de web sites, roubo de identificadores de sessão, roubo de identidade etc.

Outra forma de se explorar a técnica XSS é utilizando o servidor e suas páginas web padrão para mecanismos de tratamento de erro. Geralmente servidores web e de aplicação apresentam em suas páginas de erro os dados enviados na requisição. Sendo assim, pode-se injetar XSS em uma requisição, pois quando o servidor apresentar a página padrão de tratamento de erro o script malicioso será executado.

2.10 Cross-site request forgery (CSRF/XSRF)

A. Definição

CSRF é uma forma de ataque na internet também conhecida por ataque de um clique. Apesar do nome desta técnica ser similar ao Cross-site scripting, estes ataques possuem essências bem distintas. Enquanto o XSS explora a confiança de um usuário em um determinado site, o CSRF se aproveita da confiança do servidor na ação de um usuário [POWESKI; RAPHAEL, 2009].

B. Como funciona?

Imagine que você esteja autenticado em um internet banking visualizando suas economias e receba um e-mail lhe oferendo a oportunidade de participar de uma promoção. Quando você clica no link contido no e-mail você está em um cenário de alta vulnerabilidade a um ataque CSRF. Como resultado, suas economias são roubadas.

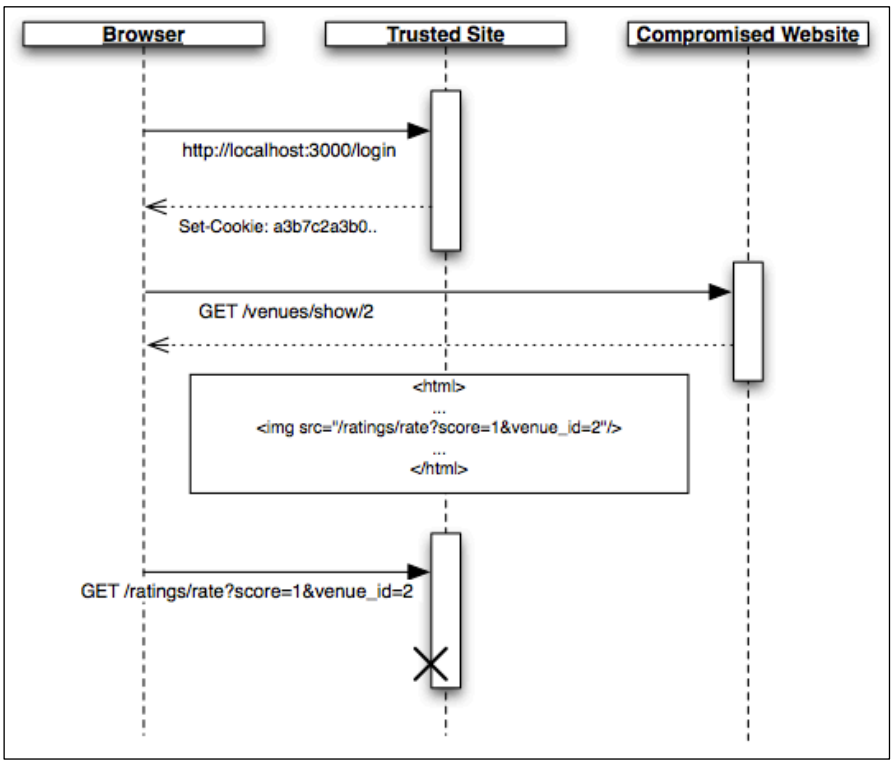


Figura 12: Exemplo de ataque CSRF [POWESKI; RAPHAEL, 2009].

O Ataque CSRF parte da premissa que o usuário esteja autenticado em um web site desejável (como por exemplo um site bancário) e caia na armadilha criada. Se o criador do ataque conseguir persuadir o usuário a clicar em um link, o atacante pode criar a sua própria requisição para o site bancário através deste link. Esta requisição pode conter, por exemplo, informações para transferência de valores entre contas.

2.11 Java

A. Definição

Java é uma linguagem de programação lançada pela Sun Microsystems em 1995, sendo que seus principais mentores foram James Gosling e Bill Joy. A linguagem grande parte de sua sintaxe de C e C++, no entanto, possui um modelo de objetos mais simples e muitas abstrações de estruturas de baixo nível. Aplicações Java são compiladas para bytecode e então podem ser executadas em qualquer Java Virtual Machine (JVM) independentemente da arquitetura do computador. Esta característica denota a principal filosofia desta linguagem: “escreva uma vez, execute em qualquer lugar”. Java é uma linguagem de propósito genérico, concorrente, orientada a objetos e especialmente desenhada para possuir o mínimo de

dependências possível. Atualmente, Java é a linguagem de programação mais utilizada no mundo, principalmente no cenário de aplicações cliente-servidor [NIEMEYER; KNUDSEN, 2000].

2.11.1 Java Enterprise Edition (JEE)

A. Definição

JEE é uma arquitetura de referência para desenvolvimento web em Java. Esta arquitetura é composta por um conjunto de componentes tecnológicos e serviços que são comumente utilizados por aplicações corporativas. Isto inclui componentes de construção de interfaces gráficas e lógica de negócio, API's para o controle de transações, segurança e ferramentas de infraestrutura para suportar o ambiente de operação da aplicação, além de ferramentas para integrações internas e externas.

2.11.2 Filtros

A. Definição

Filtros são componentes Java, muito similares a Servlets, que podem ser utilizados para interceptar e processar requisições antes que elas sejam recebida por um Servlet, ou processar respostas após a Servlet ter concluído sua execução, mas antes que esta resposta seja enviada para o cliente.

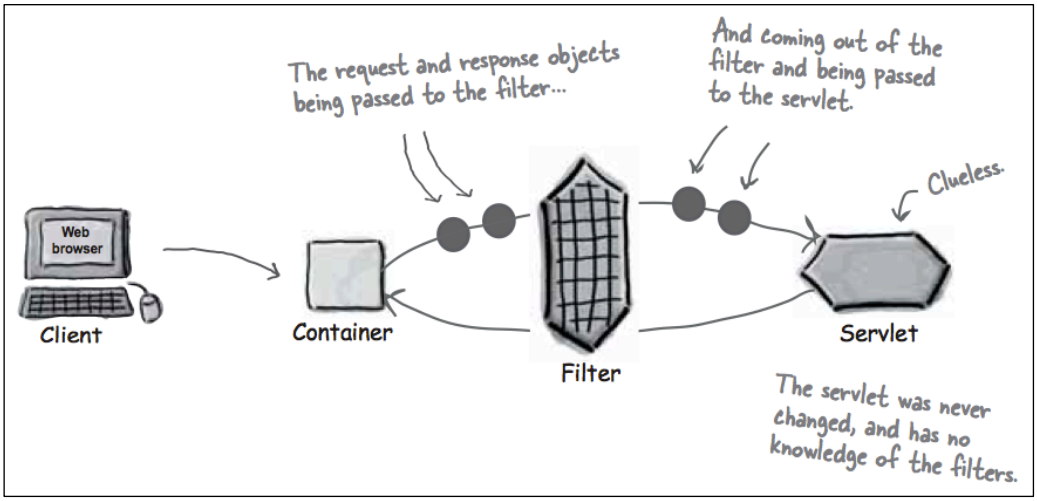


Figura 13: Funcionamento de um Filtro Java [BASHAM; SIERRA; BATES, 2004].

O servidor de aplicações decide quando invocar um filtro baseado nas declarações feitas no arquivo web.xml ou através de classes anotadas com @WebFilter.

B. Aplicabilidade

Abaixo são apresentadas algumas possíveis aplicabilidades para os componentes de filtro em uma aplicação Java web.

Filtros na requisição:

- Verificações de segurança;
- Logs de auditoria;
- Alterações de cabeçalhos e do corpo da requisição.

Filtros na resposta:

- Compressão da mensagem de resposta;
- Alteração ou modificação da mensagem de resposta;
- Log das mensagens de resposta.

3 CROSS-ORIGIN RESOURCE SHARING (CORS)

3.1 Introdução

Este capítulo tem como finalidade apresentar detalhadamente a especificação Cross-Origin Resource Sharing. Este detalhamento engloba sua definição, premissas arquiteturais, modo de funcionamento e impactos nas implementações dos atuais navegadores web e das aplicações fornecedoras de serviços. Ainda neste capítulo, são demonstradas novas abordagens arquiteturais no desenvolvimento de aplicações web baseadas neste novo mecanismo de compartilhamento.

3.2 Definição

O Cross-Origin Resource Sharing é a especificação que define um modelo, robusto e padronizado, de para compartilhamento de recursos entre origens dissemelhantes. Este modelo de compartilhamento compreende o consumo dos recursos compartilhados através de chamadas assíncronas e também provê mecanismos de segurança que permitem a manutenção da segurança propiciada pelo Same-Origin Policy [W3C - CORS, 2011].

O CORS consiste na adição de alguns cabeçalhos na conversação HTTP entre cliente e servidor, além da alteração do objeto XMLHttpRequest disponibilizado pelos navegadores web.

3.3 Requisitos

O mecanismo Cross-Origin Resource Sharing teve a sua modelagem influenciada por premissas e requisitos dos mais diversos aspectos arquiteturais. Abaixo são enumerados os requisitos desta especificação:

1. Não deve introduzir possibilidades de ataques para servidores que são protegidos apenas por um firewall;
2. Não deve ser possível realizar operações criativas ou destrutivas, ou seja, qualquer operação diferente de GET, HEAD e OPTIONS, sem antes realizar uma verificação de autorização;
3. Deve tentar prevenir, na medida do possível, ataques de força bruta distribuídos com a finalidade de obtenção de contas de usuário para servidores externos;
4. Não deve permitir o carregamento e exposição de conteúdos de servidores terceiros sem explícito consentimento destes servidores, uma vez que estes conteúdos podem conter informações sensíveis;
5. Não deve ser necessário que administradores de sistemas façam alterações para manter o atual nível de segurança;
6. Deve ser possível configurar políticas de autorização distintas entre domínios distintos para diferentes recursos hospedados em uma mesma origem;
7. Deve ser possível disponibilizar conteúdos de todos os tipos. Desta forma, deve ser possível transmitir conteúdos de qualquer “Content-Type” se o servidor assim permitir;
8. Deve ser possível definir servidores específicos ou um conjunto de servidores para fazer o consumo de determinado recurso;
9. Não deve ser necessário que o servidor filtre o corpo das requisições para poder negar ou permitir o acesso das origens solicitantes;
10. Não deve ser necessário alterar a interface do objeto XMLHttpRequest para ser possível realizar requisições cross-domain;
11. Deve ser possível realizar requisições com os mais diversos verbos HTTP;
12. Deve ser compatível com os mecanismos de autenticação HTTP e gerenciamento de sessão mais usados;

13. Deve ser claro para o provedor de recursos quando o acesso é concedido e quando é negado;

3.4 O objeto XMLHttpRequest nível 2 (XHR2)

XMLHttpRequest nível 2 é o resultado da adequação do objeto XMLHttpRequest as modificações estipuladas pela especificação CORS e da inclusão de respostas progressivas.

Com relação a conformidade do XMLHttpRequest com a especificação CORS, foram adicionados mecanismos para trabalhar novos cabeçalhos HTTP. Com isso, requisições HTTP cross-domain realizadas pelo componente XMLHttpRequest passam a enviar o cabeçalho Origin. Este cabeçalho, por questões de segurança, é protegido pelos navegadores de forma que não pode ser alterado por scripts ou códigos de aplicação. Seu conteúdo é similar ao do cabeçalho Referer, porém, não contém uma URL completa, e sim, somente a URL base da aplicação cliente. Como o Referer pode conter dados sensíveis, muitas vezes é omitido pelo navegador, o que tornou a criação do cabeçalho Origin necessária.

A outra modificação importante no XMLHttpRequest foi adição da funcionalidade de respostas progressivas. Nas versões anteriores, havia apenas um evento “readystatechange”, implementado com muita inconsistência pelos diversos navegadores web. Nesta sua nova versão, o XMLHttpRequest possui seis eventos para a monitoração do progresso de eventos, são eles: “loadstart”, “progress”, “abort”, “error”, “load” e “loadend”.

3.5 Como funciona?

Os navegadores modernos, como o Firefox 7, Chrome 14 e Safari 5, utilizam o objeto XMLHttpRequest como um container da API que implementa as funcionalidades de um “cliente HTTP”. O Internet Explorer, nas versões 8 e 9, implementa parte desta API através do objeto XDomainRequest, que é similar ao XMLHttpRequest, e também funciona como um container. Vale ressaltar que, uma boa alternativa ao uso nativo destes objetos é a biblioteca JavaScript JQuery 1.5.1, ou superior, que encapsula a lógica de criação dos objetos XMLHttpRequest e XDomainRequest de acordo com o navegador utilizado e fornece uma interface amigável para utilização.

Uma requisição cross-domain, que utilize o XMLHttpRequest nível 2 e os métodos HTTP “GET” ou “POST simples”, funciona da seguinte forma:

1. O cliente envia um cabeçalho chamado “Origin”, que contém a origem da requisição, ou seja, protocolo, domínio e porta da página solicitante;
2. O servidor, adequadamente preparado para receber esta requisição, aceita ou não a origem informada, e em caso positivo, responde com o cabeçalho “Access-Control-Allow-Origin”, que indica os domínios com permissão de acesso ao recurso, ou o valor "*" (asterisco) se for um recurso público.

O padrão CORS trabalha adicionando novos cabeçalhos HTTP, nas requisições do cliente e nas respostas do servidor, que permitem aos servidores efetuarem um processo de “Handshake” , como podemos observar na **Error! Reference source not found.**

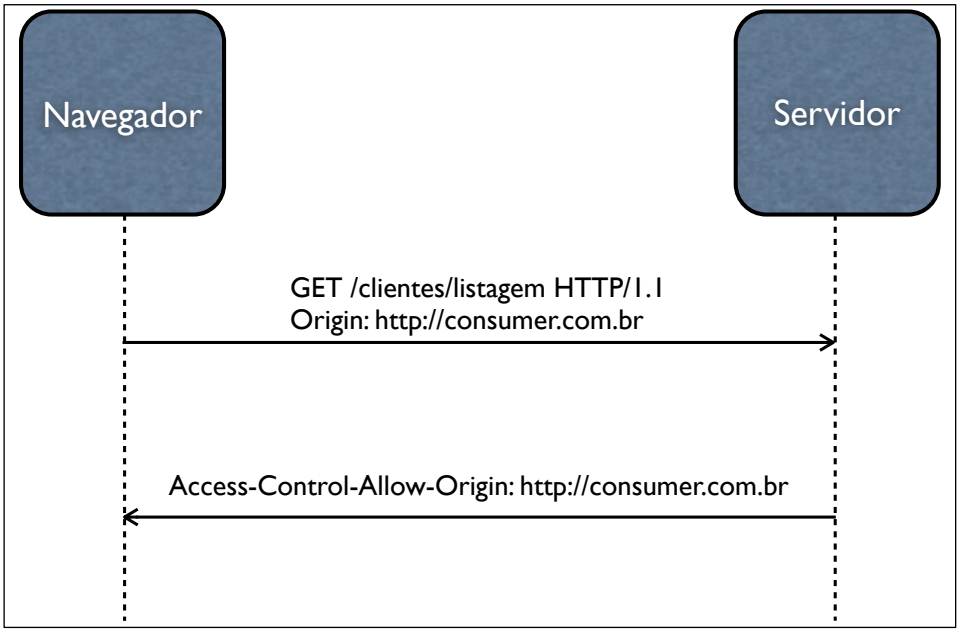


Figura 14: Fluxo de requisições GET e POST simples.

A especificação CORS obriga que requisições que não utilizem os verbos GET ou POST, ou mesmo requisições que utilizem POST para enviar dados com tipo de conteúdo diferente de “application/x-www-form-urlencoded”, “multipart/form-data”, ou “text/plain”, ou mesmo que possuam algum cabeçalho customizado façam um “pré-voo”, que em inglês é conhecido como “preflight”. Esta determinação serve para mitigar possíveis ataques contra os recursos disponibilizados, evitando assim que dados sensíveis possam ser alterados facilmente.

O pré-voo é uma requisição, com o verbo HTTP OPTIONS e com o cabeçalho Origin informando a origem solicitante, que acontece antes da requisição original. Com isto, o navegador descobre se o recurso solicitado está preparado para receber requisições de outras

origens e se a origem solicitante tem acesso ao recurso desejado. Após este processo, caso a validação seja bem sucedida, o navegador dispara a requisição original.

O mecanismo de pré-voo é transparente para o desenvolvedor web pois ele acontece automaticamente, sendo o navegador responsável por este trabalho.

Para ilustrar este mecanismo, a Listagem 1: Chamada AJAX cross-domain para um Web Service apresenta um código JavaScript carregado na página web http://consumer.com.br acessando um Web Service disponibilizado em http://provider.com.br.

```
01. function callOtherDomain(){
02.    var url = 'http://provider.com.br/clientes/adiciona';
03.    var invocation = new XMLHttpRequest();
04.    invocation.open('POST', url, true);
05.    invocation.setRequestHeader('Content-Type', 'application/xml');
06.    invocation.send('nome=PauloVitor');
07. }
```

Listagem 1: Chamada AJAX cross-domain para um Web Service.

Em seguida, na Figura 15: Cabeçalhos trafegados no mecanismo de pré-voo., são apresentados os cabeçalhos HTTP trafegados durante um pré-voo. É importante ressaltar os cabeçalhos "Access-Control-Allow-Origin" e "Access-Control-Allow-Methods" na resposta do servidor. O primeiro indica quais são as origens que podem acessar o recurso e o segundo indica quais são os verbos HTTP permitidos.

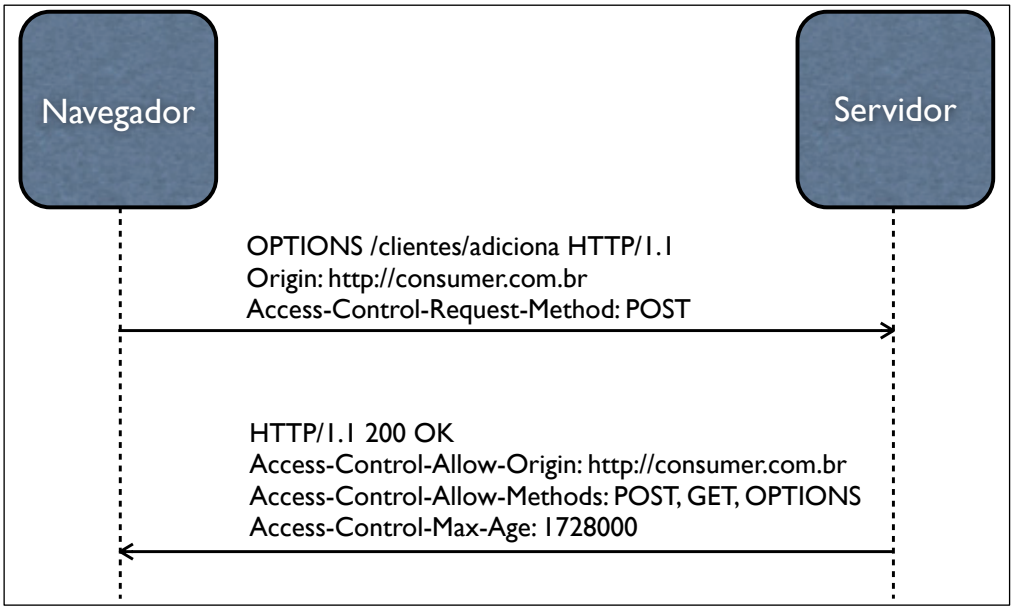


Figura 15: Cabeçalhos trafegados no mecanismo de pré-voo.

Uma vez que a origem solicitante recebe a resposta do mecanismo de pré-voo, o navegador avalia se a origem está habilitada a acessar o recurso solicitado e, também, se o verbo HTTP, com o qual a requisição original deve ser realizada, encontra-se na lista de métodos permitidos. Caso esta validação seja bem sucedida, a requisição original é enviada pelo navegador, como podemos observar na Figura 16: Execução da requisição original após pré-voo.. Caso contrario, o navegador aborta o procedimento, não realizando a requisição original.

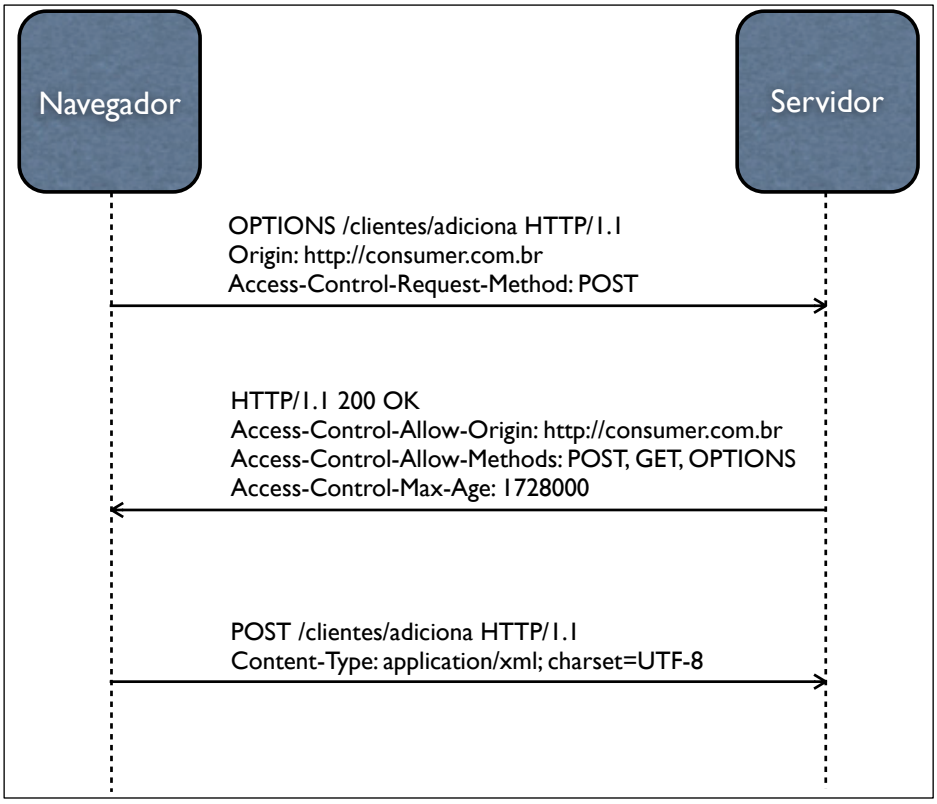


Figura 16: Execução da requisição original após pré-voo.

3.6 O uso de credenciais

O termo credenciais proposto pela especificação representa os cookies, autenticação HTTP e certificados SSL do lado do cliente. Esta recurso, sem dúvida, é um dos recursos mais interessantes da especificação CORS. Por padrão, este recurso é desabilitado e o XMLHttpRequest ignora as credenciais. Para habilitar esta opção, uma flag deve ser definida no objeto XMLHttpRequest após sua inicialização. É importante destacar que o uso de credenciais ainda não é suportado pelo objeto XDomainRequest utilizado pelo Internet Explorer 8 e 9.

O trecho de código apresentado na Listagem 2: Uso do objeto XMLHttpRequest em requisições credenciadas. mostra, na linha 5, como habilitar a flag “withCredentials” do objeto XMLHttpRequest para realizar requisições com cookie.

```
01. function callOtherDomain(){
02.     var url = 'http://provider.com.br/clientes/add?nome=';
03.     var invocation = new XMLHttpRequest();
04.     invocation.open('POST', url, true);
05.     invocation.withCredentials = true;
06.     invocation.setRequestHeader('Content-Type', 'application/xml');
07.     invocation.send('nome=PauloVitor');
08. }
```

Listagem 2: Uso do objeto XMLHttpRequest em requisições credenciadas.

Uma vez que o cliente, através da flag “withCredentials”, defina sua intenção em usar requisições credenciadas cabe ao servidor informar se determinado recurso aceita trabalhar este tipo de requisição. Mais especificamente, para que o navegador tenha acesso a estas informações, as respostas do servidor devem conter o cabeçalho “Access-Control-Allow-Credentials” com o valor “true”. Por padrão, esta opção é definida como “false”, inibindo o tráfego de cookies e inviabilizando o consumo de serviços autenticados.

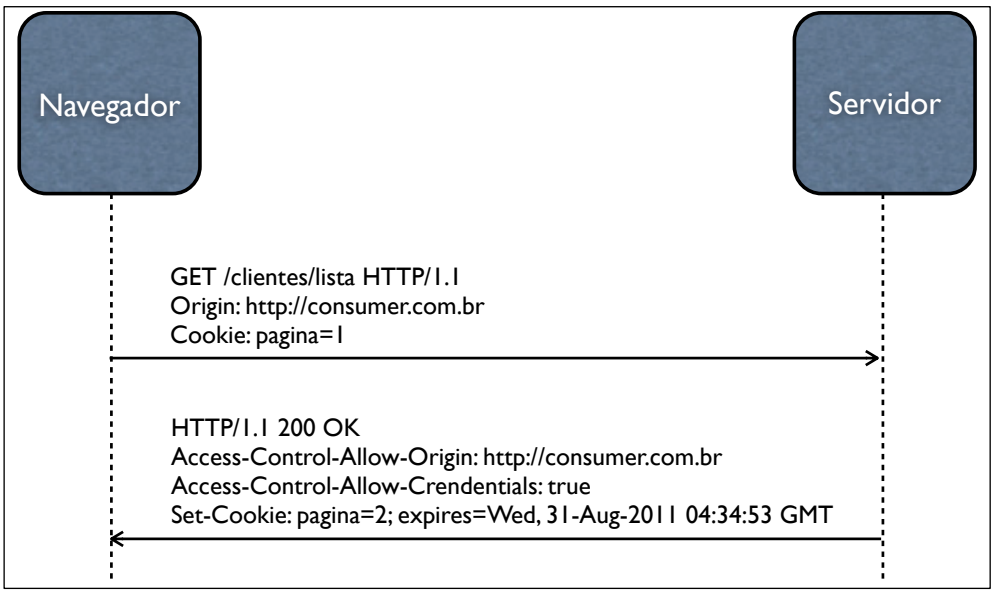


Figura 17: Cabeçalhos trafegados em uma requisição credenciada.

A Figura 17: Cabeçalhos trafegados em uma requisição credenciada. demonstra os cabeçalhos trafegados, entre cliente e servidor, com base no código ilustrado na Listagem 2: Uso do objeto XMLHttpRequest em requisições credenciadas..

3.7 Impacto arquitetural

Todas as mudanças advindas com a implementação desta nova forma de compartilhamento de recursos permite cria inúmeras possibilidades arquiteturais no desenvolvimento de aplicações web.

Por exemplo, é possível se desenvolver aplicações que utilizam serviços hospedados em inúmeras origens. Pode-se hospedar uma aplicação web que utiliza conteúdo estático e outra apenas para receber as requisições AJAX, sendo que toda a comunicação passa a ser feita via XMLHttpRequest. Esta possibilidade amplia as opções de implantação de uma aplicação web.

Como podemos observar na Figura 18: Diferença entre agregação no lado servidor podemos realizar a agregação de conteúdo de diferentes fontes no lado cliente, mais especificamente no navegador do usuário. Podemos ainda, caso o servidor de destino permita, acessar conteúdos protegidos utilizando as credenciais do usuário, provendo uma experiência personalizada para este usuário. A agregação de conteúdo no lado servidor, por outro lado, afunila todo o conteúdo disponibilizado por diversos fornecedores em apenas uma infra-estrutura física, muitas vezes causando gargalos.

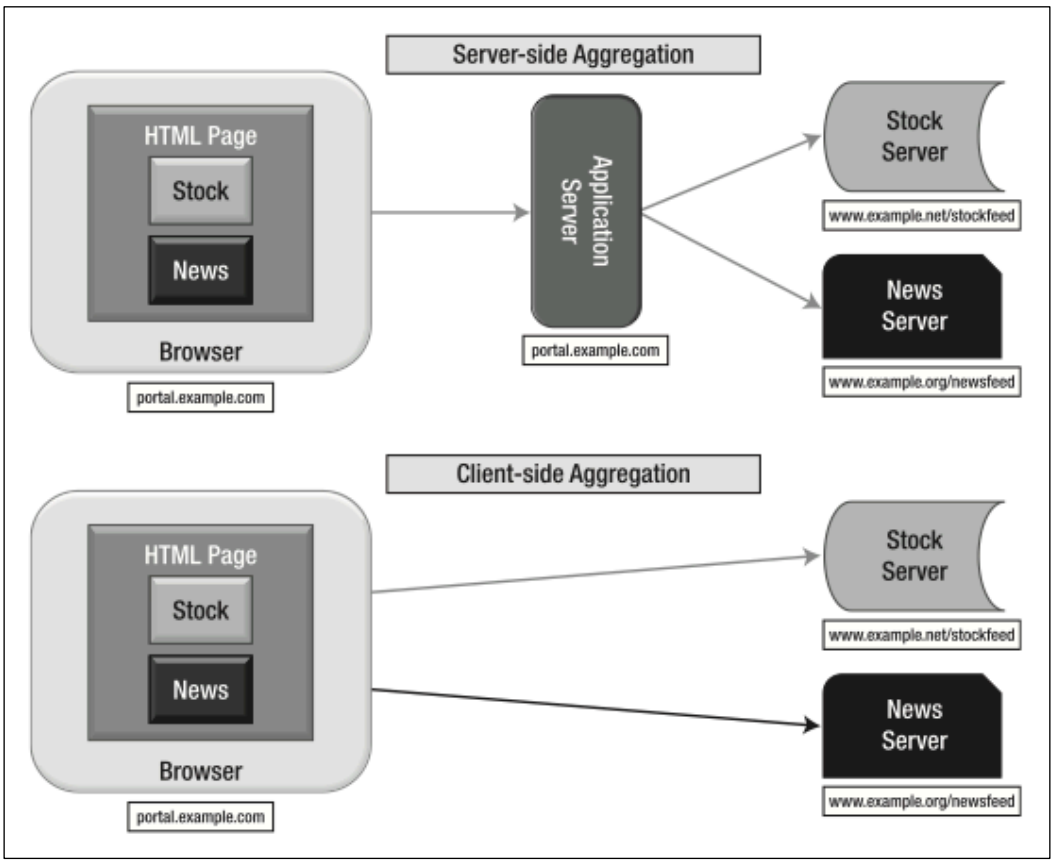
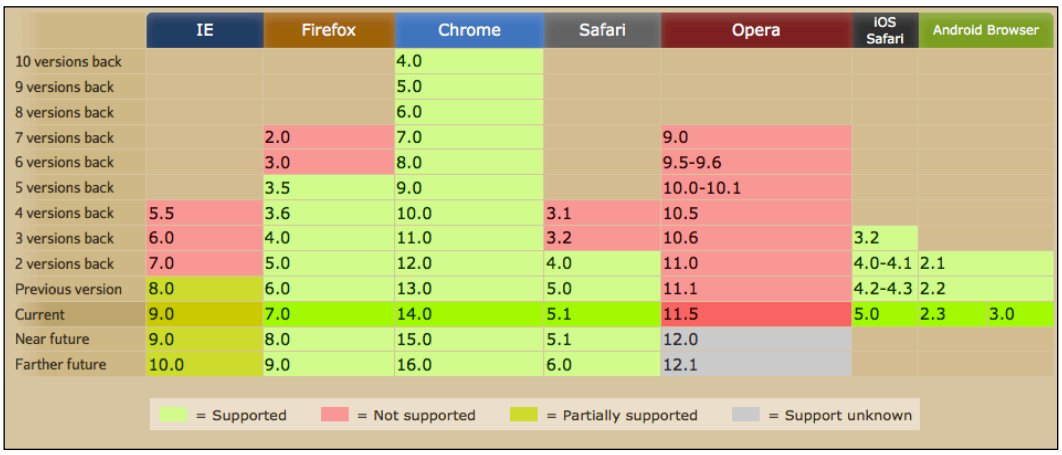


Figura 18: Diferença entre agregação no lado servidor e cliente [LUBBERS; ALBERS; SALIM, 2010].

Uma vez que passamos a agregar conteúdos, espalhados pela internet, no navegador do usuário final de uma aplicação web estamos implicitamente distribuindo grande parte do processamento desta aplicação. Essa distribuição de processamento computacional propicia altos níveis de escalabilidade.

3.8 Suporte dos navegadores atuais

Atualmente, grande parte dos navegadores modernos possuem suporte integral a especificação CORS, como podemos aferir através da Figura 19: Matriz de compatibilidade dos navegadores web com o padrão CORS [DEVERIA, 2011]..



	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Android Browser	
10 versions back			4.0					
9 versions back			5.0					
8 versions back			6.0					
7 versions back		2.0	7.0		9.0			
6 versions back		3.0	8.0		9.5-9.6			
5 versions back		3.5	9.0		10.0-10.1			
4 versions back	5.5	3.6	10.0	3.1	10.5			
3 versions back	6.0	4.0	11.0	3.2	10.6	3.2		
2 versions back	7.0	5.0	12.0	4.0	11.0	4.0-4.1	2.1	
Previous version	8.0	6.0	13.0	5.0	11.1	4.2-4.3	2.2	
Current	9.0	7.0	14.0	5.1	11.5	5.0	2.3	3.0
Near future	9.0	8.0	15.0	5.1	12.0			
Farther future	10.0	9.0	16.0	6.0	12.1			

= Supported = Not supported = Partially supported = Support unknown

Figura 19: Matriz de compatibilidade dos navegadores web com o padrão CORS [DEVERIA, 2011].

No contexto dos computadores de mesa, mais conhecidos como Desktop, os navegadores do Google, Mozilla e Apple, respectivamente, Chrome, Firefox e Safari são absolutamente aderentes ao CORS. O navegador Opera, atualmente na versão 11.51, não possui suporte, porém o assunto já é tratado com grande importância nos fóruns de sua comunidade e, acredita-se que, em sua próxima versão este navegador já compreenderá o novo mecanismo de compartilhamento. A Microsoft, mantenedora do Internet Explorer, implementa parcialmente o comportamento definido pela especificação, não suportando requisições credenciadas e o mecanismo de pré-voo. Outro ponto a ser considerado com relação a implementação da Microsoft é sua opção por não seguir a convenção, adotada pelos demais navegadores, de adicionar os novos comportamentos definidos pelo CORS no objeto XMLHttpRequest já existente, tendo criado um novo objeto denominado XDomainRequest.

No cenário do mundo mobile, praticamente todo o trefego de informações realizado é feito através dos navegadores iOS Safari, Android Browser, Opera Mini e Opera Mobile, sendo que, destes, apenas os dois últimos ainda não possuem suporte ao CORS, porém, como são mantidos pela mesma empresa do navegador Opera, suas aderências são consideradas eminentes.

4 ESTUDO DE CASO

4.1 Introdução

Este capítulo visa elucidar, através de uma abordagem prática, o funcionamento da especificação CORS em conjunto com a linguagem de programação Java.

4.2 Cenário

Este estudo de caso compreende o desenvolvimento de duas aplicações web em Java: a aplicação api-provider, que conterá alguns Web Services simulando a disponibilização de uma API; e a aplicação api-consumer, que será um site consumidor dos Web Services, da primeira aplicação, através de chamadas AJAX.

A princípio, mostraremos como as requisições AJAX, disparadas pela aplicação api-consumer, serão barradas pelo navegador em virtude do SOP. Na sequência, implementaremos um filtro em api-provider com o objetivo de interceptar as requisições recebidas e trabalhar os cabeçalhos HTTP, de acordo com a especificação CORS. Esta sutil e não intrusiva alteração, tornará a aplicação consumidora dos web services apta a fazer o seu trabalho.

4.3 A aplicação api-provider

A Figura 20: Estrutura inicial da aplicação api-provider. mostra a estrutura inicial da aplicação Java web api-provider. Nesta aplicação, duas classes são importantes para o nosso contexto: MessageBean e Resource. A função de MessageBean é encapsular as mensagens de

resposta dos web services e facilitar sua serialização para o formato JSON, e a função de Resource é disponibilizar os web services do recurso.

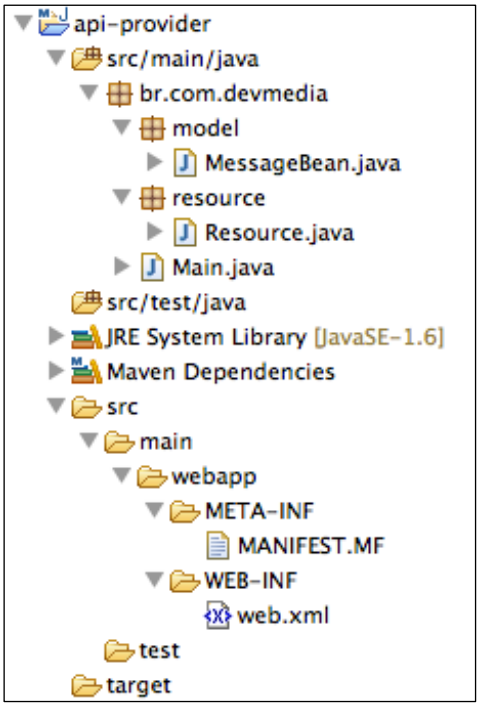


Figura 20: Estrutura inicial da aplicação api-provider.

A implementação da classe MessageBean pode ser visualizada na Listagem 3: Implementação da classe MessageBean.. Esta classe possui um design simples, contendo apenas o atributo mensagem, seus respectivos métodos de acesso, e dois construtores, sendo que um recebe o atributo mensagem como parâmetro.

```
01. package br.com.devmedia.model;
02. import javax.xml.bind.annotation.XmlRootElement;
03.
04. @XmlRootElement
05. public class MessageBean {
06.   private String mensagem;
07.
08.   public MessageBean() {}
09.   public MessageBean(String mensagem) {
10.     this.mensagem = mensagem;
11.   }
12.   public String getMensagem() {
13.     return mensagem;
14.   }
15.   public void setMensagem(String mensagem) {
16.     this.mensagem = mensagem;
17.   }
18. }
```

Listagem 3: Implementação da classe MessageBean.

A seguir, apresentada na Listagem 4: Implementação da classe Resource., temos a implementação da classe Resource. Esta classe é um POJO (Plain Old Java Object) que possui

algumas anotações da especificação JAX-RS, disponibilizadas pelo framework Jersey. Através da anotação @Path, informamos a URL relativa pela qual o recurso será disponibilizado. Na sequência, anotamos a classe com @Consumes e @Produces, indicando quais os media types serão aceitos e produzidos respectivamente. Em nosso caso, aceitaremos qualquer media type na requisição, e na resposta, enviaremos um JSON.

Por fim, as anotações @GET, @PUT, @POST e @DELETE, correspondem, similarmente, aos métodos HTTP, ou seja, a anotação @GET indica que o método, por ela anotado, responderá a requisições HTTP GET e assim por diante.

```
01. package br.com.devmedia.resource;
02.
03. import javax.ws.rs.*;
04. import javax.ws.rs.core.*;
05. import br.com.devmedia.model.MessageBean;
06.
07. @Path("/recurso")
08. @Consumes(MediaType.WILDCARD)
09. @Produces(MediaType.APPLICATION_JSON)
10. public class Resource {
11.   @GET
12.   public MessageBean get() {
13.     return new MessageBean("GET /recurso executado com sucesso!");
14.   }
15.   @POST
16.   public MessageBean post() {
17.     return new MessageBean("POST /recurso executado com sucesso!");
18.   }
19.   @PUT
20.   public MessageBean put() {
21.     return new MessageBean("PUT /recurso executado com sucesso!");
22.   }
23.   @DELETE
24.   public MessageBean delete() {
25.    return new MessageBean("DELETE /recurso executado com sucesso!");
26.   }
27. }
```

Listagem 4: Implementação da classe Resource.

4.4 A aplicação api-consumer

Na Figura 21: Estrutura da aplicação api-consumer., podemos observar a composição da aplicação api-consumer, uma aplicação Java web simples, que fará o consumo dos serviços disponibilizados pela aplicação api-provider, simulando o comportamento de uma aplicação cliente.

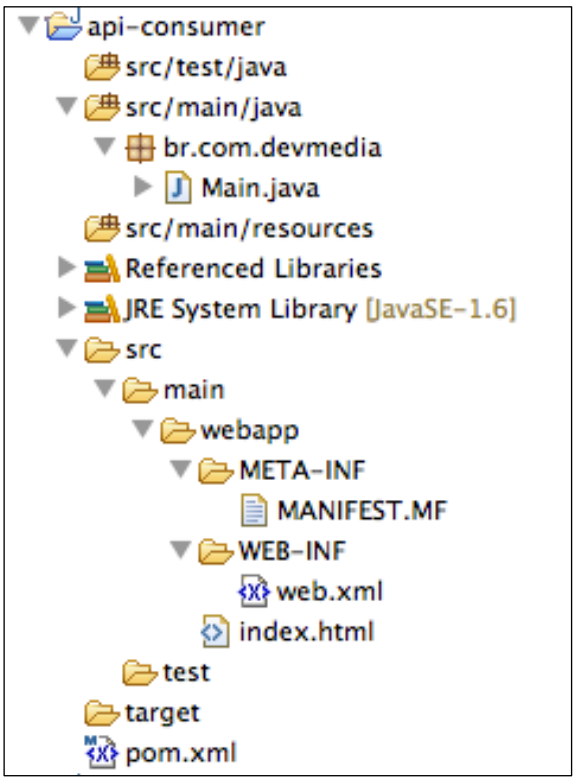


Figura 21: Estrutura da aplicação api-consumer.

Esta aplicação possui, como elemento significativo para o estudo de caso, a página index.html.

Repare que os elementos que compõem a aplicação são estáticos, pois estamos simulando um site ou aplicação onde a lógica de consumo dos web services acontece no navegador do usuário final através de scripts escritos em JavaScript.

A página index.html, cujo código-fonte é apresentado na Listagem 5, será responsável por disparar as requisições AJAX na tentativa de fazer o consumo dos web services.

```
01. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
02. <html xmlns="http://www.w3.org/1999/xhtml">
03. <head>
04.   <meta http-equiv="Content-type" content="text/html; charset=UTF-8" />
05.
06.   <script type="text/javascript">
07.   function consumirWebService(){
08.     var destino = document.getElementById('destino_requisicao').value;
09.     var metodoHttp = document.getElementById('metodo_http').value;
10.     var status = document.getElementById('status');
11.     var resposta = document.getElementById('resposta');
12.
13.     status.innerHTML = "Chamando Web Service: "+destino+" utilizando o método: "+metodoHttp;
14.
15.     var xhr = new XMLHttpRequest();
16.     xhr.onreadystatechange = function(){
17.       if(xhr.readyState != 4) return;
18.       resposta.innerHTML = xhr.responseText;
19.       resposta.style.backgroundColor = "#99FF33";
20.     };
21.     xhr.onerror = function(){
22.       resposta.innerHTML = "Erro ao consumir serviços.";
23.       resposta.style.backgroundColor = "#FFCC00";
24.     };
25.     xhr.open(metodoHttp, destino);
26.     xhr.send();
27.   };
28.   </script>
29. </head>
30. <body style="font:12px verdana">
31.   <h3>Exemplo de aplicação consumidora de Web Services com origem distinta</h3>
32.
33.   <h3>URL do recurso a ser consumido:</h3>
34.   <input type="text" id="destino_requisicao"
35.         value="http://provider.com.br:9999/recurso" style="width:300px" /> <br />
36.
37.   <h3>Método HTTP:</h3>
38.   <select id="metodo_http">
39.     <option value="get" selected="selected">get</option>
40.     <option value="post">post</option>
41.     <option value="put">put</option>
42.     <option value="delete">delete</option>
43.   </select> <br /><br />
44.
45.   <input type="button" onclick="javascript:consumirWebService();" value="Realizar requisição" />
46.   <br />
47.   <h3>Status da requisição:</h3> <div id="status"></div> <br />
48.   <h3>Resposta recebida:</h3> <div id="resposta"></div>
49. </body>
50. </html>
```

Listagem 5: Implementação da página index.html.

Ainda analisando a Listagem 5: Implementação da página index.html., podemos observar a declaração da função consumirWebService(), que será utilizada no disparo das requisições para a aplicação api-provider. Essas requisições serão efetuadas contra a URL estabelecida no campo destino_requisicao, utilizando o verbo HTTP definido no campo metodo_http.

4.5 Configurações de ambiente

Para tornar ainda mais real a simulação de comunicação cross-domain, além de subirmos as aplicações em portas diferentes, adicionaremos duas entradas no nosso arquivo de hosts para que possamos acessar as aplicações através de domínios diferentes. Vale lembrar

que em Sistemas Operacionais baseados em UNIX, esse arquivo encontra-se em /etc/hosts, e no Windows, em C:\WINDOWS\system32\drivers\etc\hosts.

As linhas que devem ser adicionadas ao arquivo de hosts são:

127.0.0.1 provider.com.br

127.0.0.1 consumer.com.br

4.6 Testes sem o filtro CORS

Para a realização dos testes, subiremos ambas as aplicações utilizando o servidor de aplicações Jetty de forma embarcada. Faremos isso clicando com o botão direito na classe Main.java e selecionando a opção Run As > Java Application do IDE Eclipse. A aplicação api-provider responderá na porta 9999, enquanto api-consumer ficará disponível na porta 8080.

Após a inicialização das instâncias do Jetty, vamos acessar a aplicação api-consumer através da URL http://consumer.com.br:8080/index.html, onde visualizaremos a página index.html. Nesta página faremos a parametrização da requisição selecionando o método HTTP a ser utilizado. Na sequência clicaremos no botão “Realizar requisição”, que por sua vez invocará a função JavaScript consumirWebService().

Nossa primeira tentativa de consumo utiliza o método HTTP GET. Porém, a tentativa não obtém sucesso e uma mensagem de erro é apresentada na interface, como mostra a Figura 22: Tentativa falha de consumo de web services de outra origem..

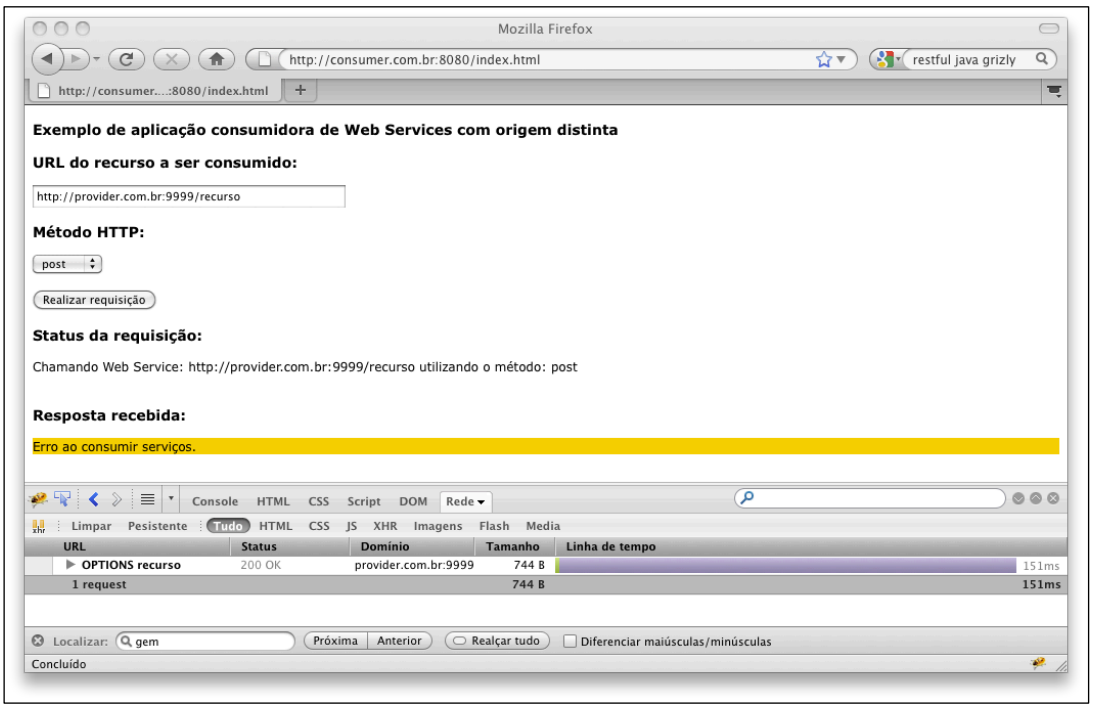


Figura 22: Tentativa falha de consumo de web services de outra origem.

Os demais resultados obtidos, para cada verbo HTTP disponível na interface de parametrização, são apresentados na Tabela 2: Resultados obtidos sem o filtro CORS..

Recurso	Método HTTP	Resultado Obtido
http://provider.com.br:9999/recurso	GET	Falha
http://provider.com.br:9999/recurso	POST	Falha
http://provider.com.br:9999/recurso	PUT	Falha
http://provider.com.br:9999/recurso	DELETE	Falha

Tabela 2: Resultados obtidos sem o filtro CORS.

As falhas obtidas nas requisições do nosso estudo de caso comprovam a aderência dos navegadores ao conceito de segurança SOP, uma vez que nossas requisições partiram de uma origem com domínio e porta diferentes da origem de onde estão os web services.

4.7 Aplicando o filtro CORS

Seguindo nosso estudo de caso, alteraremos a aplicação api-provider incluindo a codificação da especificação CORS. A nova estrutura pode ser visualizada na Figura 23: Estrutura da aplicação api-provider com o filtro CORS.. Comparando com a estrutura anteriormente apresentada, apenas o pacote cors foi adicionado. Neste novo pacote estão

contidas algumas exceções, a classe CORSFilter, responsável por trabalhar as requisições de acordo com a especificação CORS, e a classe CORSHelper, que contém alguns métodos auxiliares que fazem a parte mais pesada do trabalho, deixando a implementação do filtro mais enxuta e legível.

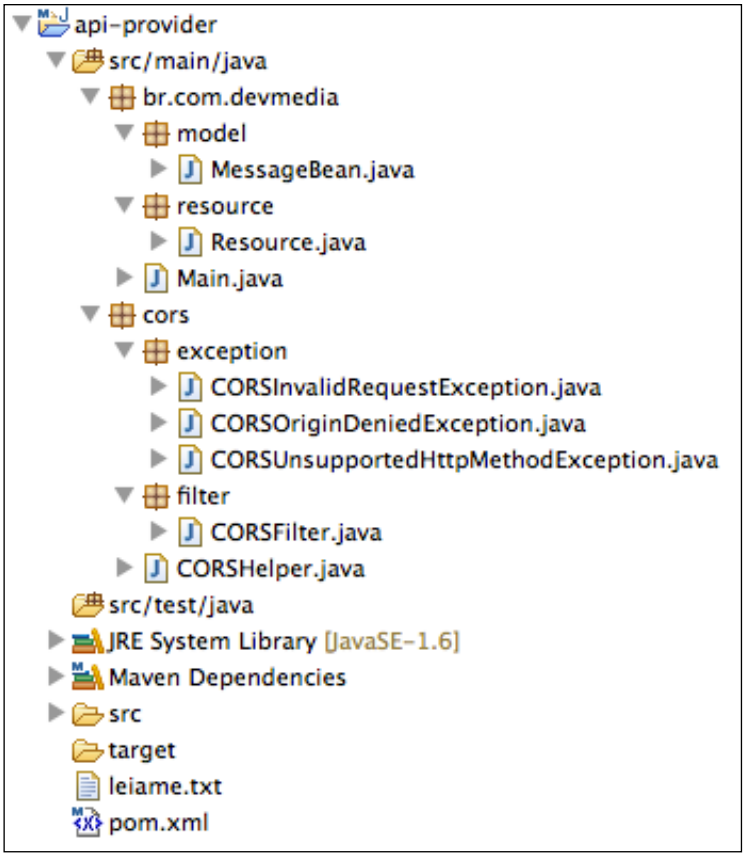


Figura 23: Estrutura da aplicação api-provider com o filtro CORS.

A Listagem 6: Implementação da classe CORSFilter. exibe o código da classe CORSFilter. Esta classe implementa a interface javax.servlet.Filter para que possamos interceptar as requisições e adicionar os comportamentos previstos na especificação CORS. O método doFilter(), que será invocado a cada requisição recebida, tem como responsabilidade principal tratar três possíveis tipos de requisição, são eles: requisições de pré-voo, requisições originais (aquelas subsequentes ao pré-voo) e requisições provenientes do mesmo domínio, ou seja, que não enviam os cabeçalhos estabelecidos pela especificação.

```
01. package cors.filter;
02.
03. import java.io.IOException;
04. import java.io.PrintWriter;
05. import javax.servlet.*;
06. import javax.servlet.http.*;
07. import cors.CORSHelper;
08. import cors.exception.*;
09.
10. public class CORSFilter implements Filter {
11.
12.   public CORSHelper corsHelper = new CORSHelper();
13.
14.   @Override
15.    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
       FilterChain filterChain) throws IOException, ServletException {
17.     HttpServletRequest request = (HttpServletRequest) servletRequest;
18.     HttpServletResponse response = (HttpServletResponse) servletResponse;
19.     try {
20.       //trata requisições preflighted
21.       if (corsHelper.isPreflightRequest(request)) {
22.         corsHelper.handlePreflightRequest(request, response);
23.         filterChain.doFilter(request, response);
24.         return;
25.       }
26.       //trata requisições originais (subsequentes às requisições preflighted)
27.       if (corsHelper.isOriginalRequest(request)) {
28.         corsHelper.handleOriginalRequest(request, response);
29.         filterChain.doFilter(request, response);
30.         return;
31.       }
32.       // trata requisições do mesmo domínio
33.       filterChain.doFilter(request, response);
34.
35.     } catch (CORSInvalidRequestException e) {
36.       printMessage(response, HttpServletResponse.SC_BAD_REQUEST, e.getMessage());
37.     } catch (CORSOriginDeniedException e) {
38.       printMessage(response, HttpServletResponse.SC_FORBIDDEN, e.getMessage());
39.     } catch (CORSUnsupportedHttpMethodException e) {
40.       printMessage(response, HttpServletResponse.SC_METHOD_NOT_ALLOWED, e.getMessage());
41.     }
42.   }
43.
44.   @Override
45.   public void init(FilterConfig filterConfig) throws ServletException {}
46.
47.   @Override
48.   public void destroy() {}
49.
50.   private void printMessage(final HttpServletResponse response, final int sc, final
       String msg) throws IOException, ServletException {
51.     response.setStatus(sc);
52.     response.resetBuffer();
53.     response.setContentType("text/plain");
54.     PrintWriter out = response.getWriter();
55.     out.println("Filtro CORS: " + msg);
56.   }
57.
58. }
```

Listagem 6: Implementação da classe CORSFilter.

A classe CORSHelper, indicada na Listagem 7: Implementação da classe CORSHelper., tem como finalidade auxiliar a classe CORSFilter criando uma camada de abstração para a verificação e manipulação dos cabeçalhos negociados entre cliente e servidor. Os métodos

isPreflightRequest() e isOriginalRequest() são responsáveis por identificarem, respectivamente, requisições preflighted e originais. Já os métodos handlePreflightRequest() e handleOriginalRequest(), têm a função de verificar se o método HTTP solicitado está autorizado a ser executado pelo cliente, e adicionar alguns cabeçalhos na resposta através do método addControlHeaders().

```
01. package cors;
02.
03. import javax.servlet.http.*;
04. import cors.exception.*;
05.
06. public class CORSHelper {
07.
08.   private static final String ALLOW_METHODS = "GET, POST, PUT, DELETE";
09.   private static final String ALLOW_CREDENTIALS = "false";
10.   private static final String ALLOW_HEADERS = "X-Requested-With, X-Prototype-Version,
XDomainRequest";
11.   private static final String MAX_AGE = "5";
12.
13.   public boolean isPreflightRequest(HttpServletRequest request){
14.     return (request.getHeader("Origin") != null &&
15.         request.getHeader("Access-Control-Request-Method") != null &&
16.         request.getMethod() != null &&
17.           request.getMethod().equals("OPTIONS"));
18.   }
19.
20.   public void handlePreflightRequest(HttpServletRequest request, HttpServletResponse response)
21.   throws CORSInvalidRequestException, CORSOriginDeniedException,
CORSUnsupportedHttpMethodException  {
22.     String originHeader = request.getHeader("Origin");
23.     String requestMethodHeader = request.getHeader("Access-Control-Request-Method");
24.
25.     if(! ALLOW_METHODS.contains(requestMethodHeader))
26.       throw new CORSUnsupportedHttpMethodException();
27.
28.     this.addControlHeaders(response, originHeader);
29.   }
30.
31.   public boolean isOriginalRequest(HttpServletRequest request){
32.     return (request.getHeader("Origin") != null &&
33.           request.getMethod() != null &&
34.         !request.getMethod().equals("OPTIONS"));
35.   }
36.
37.   public void handleOriginalRequest(HttpServletRequest request, HttpServletResponse response)
38.   throws CORSInvalidRequestException, CORSOriginDeniedException,
CORSUnsupportedHttpMethodException  {
39.     String originHeader = request.getHeader("Origin");
40.     String httpMethod = request.getMethod();
41.
42.     if(! ALLOW_METHODS.contains(httpMethod))
43.       throw new CORSUnsupportedHttpMethodException();
44.
45.     this.addControlHeaders(response, originHeader);
46.   }
47.
48.   private void addControlHeaders(HttpServletResponse response, String origin) {
49.     response.addHeader("Access-Control-Allow-Origin", origin);
50.     response.addHeader("Access-Control-Allow-Credentials", ALLOW_CREDENTIALS);
51.     response.addHeader("Access-Control-Allow-Methods", ALLOW_METHODS);
52.     response.addHeader("Access-Control-Allow-Headers", ALLOW_HEADERS);
53.     response.addHeader("Access-Control-Max-Age", MAX_AGE);
54.   }
55.
56. }
```

Listagem 7: Implementação da classe CORSHelper.

Precisamos ainda alterar o nosso arquivo web.xml para informar qual será o padrão de URL que, quando acessado, será filtrado pelo filtro CORSFilter. Para o nosso estudo de caso,

faremos com que todos os acessos à aplicação sejam filtrados. Esta configuração pode ser visualizada na Listagem 8: Declaração do filtro CORSFilter no arquivo web.xml..

```
  <?xml version="1.0" encoding="UTF-8"?>
  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd"
    version="2.5">

    <filter>
      <filter-name>Filtro CORS</filter-name>
      <filter-class>cors.filter.CORSFilter</filter-class>
    </filter>
    <filter-mapping>
      <filter-name>Filtro CORS</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
      <servlet-name>Jersey REST Service</servlet-name>
      <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
      <init-param>
        <param-name>com.sun.jersey.config.property.packages</param-name>
        <param-value>br.com.devmedia.resource</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
      <servlet-name>Jersey REST Service</servlet-name>
      <url-pattern>/*</url-pattern>
    </servlet-mapping>

  </web-app>
```

Listagem 8: Declaração do filtro CORSFilter no arquivo web.xml.

4.8 Testes com o filtro CORS

Assim como fizemos anteriormente, subiremos as aplicações api-provider e api-consumer respectivamente nas portas 9999 e 8080.

Na sequência, realizaremos os mesmos passos realizados nos testes anteriores. Acessaremos a URL http://consumer.com.br:8080/index.html, selecionaremos o método HTTP a ser utilizado e dispararemos a requisição AJAX através de um clique no botão “Realizar requisição”.

No entanto, diferentemente do comportamento observado nos testes sem o filtro CORS, os web services serão consumidos sem nenhuma limitação, como podemos observar na Figura 24: Consumo de web services de origem distinta realizado com sucesso..

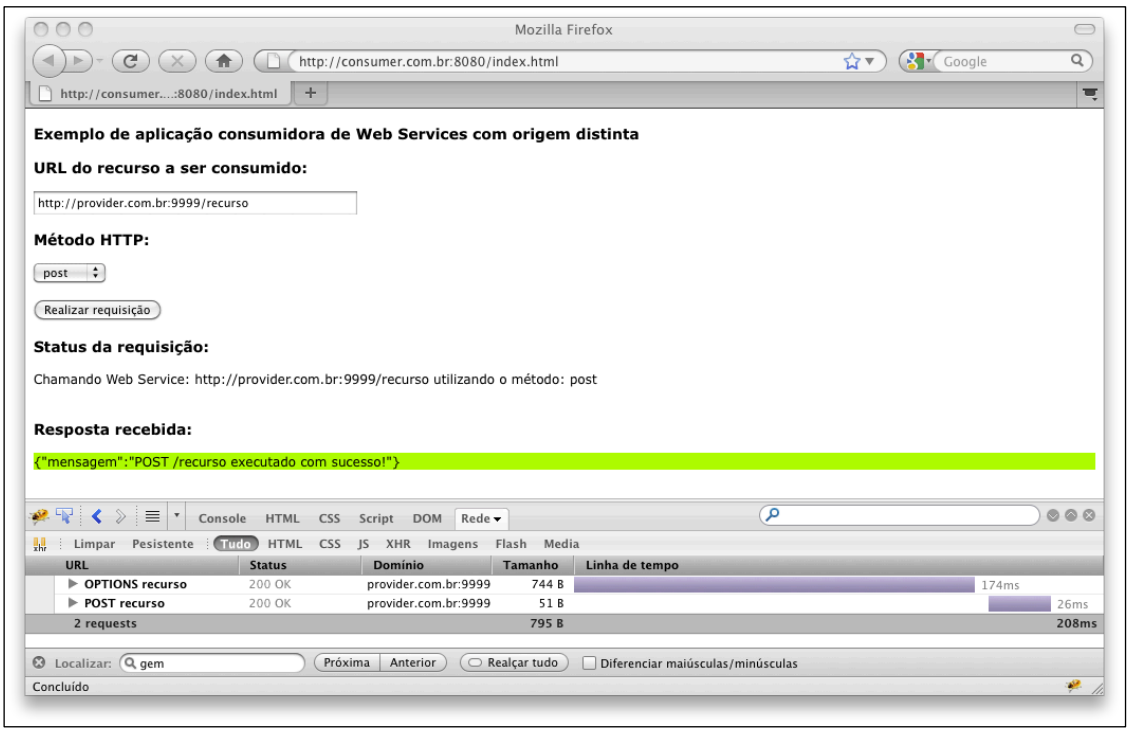


Figura 24: Consumo de web services de origem distinta realizado com sucesso.

Os demais resultados obtidos são apresentados na Tabela 3: Resultados obtidos sem o filtro CORS..

Recurso	Método HTTP	Resultado Obtido
http://provider.com.br:9999/recurso	GET	Sucesso
http://provider.com.br:9999/recurso	POST	Sucesso
http://provider.com.br:9999/recurso	PUT	Sucesso
http://provider.com.br:9999/recurso	DELETE	Sucesso

Tabela 3: Resultados obtidos sem o filtro CORS.

4.9 Considerações

Este estudo de caso demonstrou o uso do mecanismo de compartilhamento de recursos CORS combinado com a tecnologia de filtros da linguagem Java, criando assim uma camada desacoplada para tratamento dos cabeçalhos específicos do CORS. Esta abordagem mantém o código da aplicação limpo e coeso.

Conforme apresentado, o uso desta técnica requer um pouco de trabalho Javascript no lado cliente e um pouco de trabalho do lado servidor para garantir que os cabeçalhos sejam trabalhados adequadamente.

5 CONCLUSÕES E TRABALHOS FUTUROS

5.1 Introdução

Este capítulo apresenta os principais tópicos discutidos nesse trabalho, relaciona os possíveis trabalhos futuros advindos dessa pesquisa e avalia a principal contribuição da mesma.

5.2 Conclusões

Neste trabalho foi apresentado um mecanismo de compartilhamento de recursos entre origens distintas, recentemente criado pelo W3C, que permite a realização de chamadas AJAX cross-domain. Para isso, além da fundamentação do problema e apresentação de todo embasamento teórico, necessário para compreensão do tema, foi elaborado um estudo de caso onde é possível observar, passo a passo, as mudanças de implementação para o emprego do CORS.

Os resultados dos estudos sobre a especificação CORS mostraram que este novo mecanismo suprimi a complexidade das soluções de comunicação cross-domain, uma vez que seu princípio de adição de cabeçalhos HTTP é trabalhado de forma transparente pelos navegadores web e, no lado servidor, pode ser trabalhado de forma desacoplada da lógica da aplicação com o uso de módulos dos diversos servidores web ou componentes de filtro, no caso de uma aplicação web escrita em Java. Comprovou-se que a solução proposta permite que aplicações possam expor seus recursos com diferentes níveis de autorização, de acordo com a origem solicitante. Também foi provado que a extensão do objeto XMLHttpRequest, denominada nível 2, agregou os comportamentos de pré-voo e de manuseio dos novos

cabeçalhos definidos pela especificação, sem que houvessem modificações em sua interface, mantendo a retrocompatibilidade deste componente. Por fim, foram exposta uma nova possibilidade arquitetural mais distribuída e escalável para o desenvolvimento de aplicações web.

5.3 Trabalhos futuros

Nessa seção estão listados alguns pontos que foram identificados como oportunidades de evolução dos estudos realizados neste trabalho. Alguns dos tópicos listados têm como objetivo tornar esse trabalho mais completo, explorando aspectos que fugiram do escopo inicial proposto e outros pontos, que serão deixados como sugestões para extensões que visam atacar outros problemas existentes na comunicação de aplicações de origens dissemelhantes.

- Exploração de novos padrões de desenvolvimento web baseados em conteúdos provenientes de diversos fornecedores, como agregação de conteúdo no lado cliente.
- Análise das possibilidades de evolução do atual conceito de aplicações Mashup.
- Investigação do aumento de escalabilidade proveniente da mudança na forma de agregação de conteúdo, antes centralizada no lado servidor e agora disseminada nos clientes.
- Estudo sobre os possíveis ganhos de segurança do mecanismo de compartilhamento CORS em relação a ataques conhecidos, como XSS e CRFS.

6 REFERÊNCIAS BIBLIOGRÁFICAS

[FLANAGAN, 2011] FLANAGAN, David. **JavaScript: The Definitive Guide, 6ª edição**. Sebastopol, EUA: O'Reilly Media, 2011.

[PRATES, 2010] PRATES, Rubens. **JavaScript: Guia do Programador, 1ª edição**. São Paulo, BRA: Editora Novatec, 2010.

[GOURLEY et al., 2002] GOURLEY, David; Totty, Brian; Sayer, Marjorie; Aggarwal, Anshu; Reddy, Sailu. **HTTP: The Definitive Guide, 1ª edição**. Sebastopol, USA: O'Reilly Media, 2002.

[THOMAS, 2001] THOMAS, Stephen. **HTTP Essentials: Protocols for Secure, Scaleable Web Sites, 1ª edição**. Nova Iorque, EUA: John Wiley & Sons, 2001.

[ASLESON; SCHUTTA, 2006a] ASLESON, Ryan; SCHUTTA, Nathaniel. **Foundations of Ajax (Books for Professionals by Professionals), 1ª edição**. Berkeley, USA: Apress, 2006.

[CROCKFORD, 2008] CROCKFORD, Douglas. **JavaScript: The Good Parts, 1ª edição**. Sebastopol, EUA: O'Reilly Media, 2008.

[EASTTOM, 2008] EASTTOM, Chuck. **Advanced JavaScript, 3ª edição**. Plano, USA: Wordware Publishing, 2008.

[ASLESON; SCHUTTA, 2006b] ASLESON, Ryan; SCHUTTA, Nathaniel. **Pro Ajax and Java, 1ª edição**. Berkeley, USA: Apress, 2006.

[CRANE; PASCARELLO; JAMES, 2006] CRANE, Dave; PASCARELLO, Eric; JAMES, Darren. **Ajax in Action, 1ª edição**. Greenwich, GBR: Manning Publications, 2006.

[PORTENEUVE, 2010] PORTENEUVE, Christophe. **Pragmatic Guide to JavaScript, 1ª edição**. Dallas, EUA: Pragmatic Bookshelf, 2010.

[HOLDENER, 2008] HOLDENER, Anthony. **Ajax: The Definitive Guide, 1ª edição**. Sebastopol, EUA: O'Reilly Media, 2008.

[CEPONKUS; HOODBHOY, 1999] CEPONKUS, Alex; HOODBHOY, Faraz. **Applied XML: A Toolkit for Programmers, 1ª edição**. Nova Iorque, EUA: John Wiley & Sons, 1999.

[RIORDAN, 2008] RIORDAN, Rebecca. **Head First Ajax, 1ª edição**. Sebastopol, EUA: O'Reilly Media, 2008.

[NIEMEYER; KNUDSEN, 2000] NIEMEYER, Pat; KNUDSEN, Jonathan. **Learning Java, 1ª edição**. Sebastopol, EUA: O'Reilly Media, 2000.

[BASHAM; SIERRA; BATES, 2004] BASHAM, Bryan; SIERRA, Kathy; BATES, Bert. **Head First Servlets and JSP, 1ª edição**. Sebastopol, EUA: O'Reilly Media, 2004.

[WELLS, 2007] WELLS, Christopher. **Securing Ajax Applications, 1ª edição**. Sebastopol, EUA: O'Reilly Media, 2007.

[POWESKI; RAPHAEL, 2009] POWESKI, Ben; RAPHAEL, David. **Security on Rails, 1ª edição**. Dallas, EUA: Pragmatic Bookshelf, 2009.

[LUBBERS; ALBERS; SALIM, 2010] LUBBERS, peter; ALBERS, brian; SALIM, frank. **Pro HTML 5 Programming: Powerful APIs for Richer Internet Application Development, 1ª edição**. Nova Iorque, EUA: Apress, 2010.

[JSON-P, 2011] **JSON-P: Safer cross-domain Ajax with JSON-P/JSONP**. Disponível em: <http://json-p.org/>. Acesso em: Oububro de 2011.

[W3C - HTTP/1.1, 2011] **Especificação W3C da versão 1.1 do protocolo HTTP**. Disponível em: < http://www.w3.org/Protocols/rfc2616/rfc2616.html>. Acesso em: 2011.

[W3C - XHR, 2011] **Especificação W3C do objeto XMLHttpRequest**. Disponível em: <http://www.w3.org/TR/XMLHttpRequest/>. Acesso em: 2011.

[MOZILLA, 2011] **MOZILLA Developer Network**. Disponível em: <https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript>. Acesso em: 2011.

[BERNERS-LEE, 2011] **Página de Tim Berners-Lee no site do W3C**. Disponível em: <http://www.w3.org/People/Berners-Lee/WorldWideWeb.html>. Acesso em: 2011.

[ZALEWSKI, 2011] ZALEWSKI, Michal. **Browser Security Handbook**. Disponível em: <http://code.google.com/p/browsersec>. Acesso em: 2011.

[W3C - CORS, 2011] **Especificação W3C do Cross-Origin Resource Sharing**. Disponível em: < http://www.w3.org/TR/cors/>. Acesso em: 2011.

[DEVERIA, 2011] DEVERIA, Alexis. **Compatibility table for support of Cross-Origin Resource Sharing**. Disponível em: <http://caniuse.com/cors>. Acesso em: 2011.

[YAHOO DEVELOPER, 2011] **JavaScript: Use a Web Proxy for Cross-Domain XMLHttpRequest Calls**. Disponível em: <http://developer.yahoo.com/javascript/howto-proxy.html>. Acesso em: 2011.