Comunicação AJAX Cross-Domain

Implementando a especificação Cross-Origin Resource Sharing (CORS)

Este artigo aborda a especificação CORS, uma alternativa ao padrão JSONP para comunicação cross-domain

Com o crescimento e a popularização de APIs para o desenvolvimento de aplicações baseadas em AJAX, como jQuery e Prototype, é cada vez maior o número de aplicações Mashup escritas em JavaScript espalhadas pela internet. Estas aplicações combinam dados e códigos de diversas fontes de forma a criar novos serviços e funcionalidades.

Quando desenvolvemos este tipo de aplicação, encontramos algumas dificuldades quando a aplicação tenta realizar alguma forma de comunicação entre domínios distintos. Isso acontece porque os navegadores implementam o conceito de segurança *Same Origin Policy* (SOP). O SOP 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, permitindo apenas que scripts rodando em páginas com o mesmo domínio, na mesma porta e através do mesmo protocolo acessem métodos e propriedades sem nenhum tipo de restrição.

Neste artigo abordaremos a utilização do padrão Cross-Origin Resource Sharing para permitir requisições “client-side” entre diferentes domínios, detalhando seu funcionamento teórico e demonstrando a sua aplicação prática através de um estudo de caso.

O conceito de segurança Same Origin Policy

O *Same Origin Policy* (SOP) é um conceito importante de segurança que surgiu no navegador Netscape 2.0 e define as restrições de segurança que um JavaScript pode ou não realizar em uma página web. O SOP previne que um documento ou script carregado em um site de uma determinada origem recupere ou altere as propriedades de um documento de uma outra origem. Essas propriedades, que podem ser cookies ou mesmo dados de um formulário, só podem ser acessadas por scripts providos pela mesma origem, protegendo a integridade e a confidencialidade destas informações.

O termo origem é definido pela combinação do protocolo, nome do domínio e, por último, a porta TCP do documento HTML que está rodando o script. Por exemplo, [*http://foo.com:80*](http://foo.com:80/) representa uma origem, onde HTTP é o protocolo utilizado, [*foo.com*](http://foo.com/) é o domínio da aplicação web e 80 é a porta utilizada para receber as requisições.

Imaginemos um código JavaScript carregado pela página *http://provider.com.br/pagina.html* tentando acessar propriedades de outras páginas. A **Tabela 1** mostra o comportamento do navegador ao tentar se comunicar com cada origem apresentada na coluna URL.

|  |  |  |
| --- | --- | --- |
| **URL** | **Resultado** | **Motivo** |
| http://provider.com.br/recursos2/pagina2.html | Sucesso | Mesmo protocolo e domínio |
| http://provider.com.br/recursos/dir/pagina3.html | Sucesso | Mesmo protocolo e domínio |
| https://provider.com.br/seguranca.html | Falha | Protocolo diferente |
| http://provider.com.br:81/recursos/etc.html | Falha | Porta diferente |
| http://devmedia.com.br/recursos/pagina.html | Falha | Domínio diferente |

**Tabela 1.** Resultados obtidos por requisições disparadas da origem http://provider. com.br/recursos/pagina.html.

A solução Cross-Origin Resource Sharing

O Cross-Origin Resource Sharing (CORS) é uma especificação que consiste na troca de cabeçalhos HTTP entre cliente e servidor permitindo o sucesso de requisições entre domínios diferentes. Desta forma os navegadores são capazes de realizar chamadas HTTP com o uso de scripts para outros domínios. Este tipo de mecanismo nos permite realizar o consumo de APIs de diversos provedores, não ficando restrito apenas ao uso do método HTTP Get, como no padrão JSONP.

Os navegadores que implementam esta especificação possuem recursos próprios para realizar tarefas inerentes ao processo de determinação e validação da origem das requisições, como autenticação HTTP, criação e manutenção de cookies e maneiras de comparar informações, especialmente na forma de cabeçalhos enviados por ambos os lados envolvidos na comunicação.

Como funciona

Os navegadores, como o Firefox 6 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.

Uma alternativa seria o uso da biblioteca JavaScript JQuery 1.5.1, ou superior, que encapsula a lógica de criação dos objetos **XMLHttpRequest** e **XDomainRequest** de acordo com o navegador e fornece uma interface amigável para utilização. O link da documentação encontra-se nas referências deste artigo.

Estes navegadores trabalham da seguinte maneira:

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 disponibilizarem recursos a origens autorizadas.

A **Figura 1** demonstra como os navegadores trabalham, conforme o fluxo descrito acima.



**Figura 1.** Fluxo de requisições GET e POST simples.

A requisição "Preflighted"

A especificação 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* e com cabeçalhos customizados sejam *preflighted*. Esta determinação serve para mitigar possíveis ataques contra os recursos disponibilizados, evitando assim que dados sensíveis possam ser alterados facilmente.

A requisição *preflighted* é uma requisição, com o verbo OPTIONS do HTTP 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. Esta funcionalidade não é suportada pelo objeto **XDomainRequest** do Internet Explorer 8 e 9, somente pelo objeto **XMLHttpRequest**. Esta limitação existe porque o Internet Explorer não implementa a especificação por completo.

O mecanismo de *preflighted* é transparente para o desenvolvedor web pois ele acontece automaticamente, sendo o navegador responsável por este tratamento.

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

**Listagem 1.** Exemplo de chamada AJAX para um Web Service de outro domínio.

**var** invocation = **new** XMLHttpRequest();

**var** url = 'http://provider.com.br/clientes/adiciona';

**var** body = 'nome=Italo';

**function** callOtherDomain(){

**if**(invocation){

invocation.open('POST', url, **true**);

invocation.setRequestHeader('Content-Type', 'application/xml');

invocation.send(body);

}

}

Os cabeçalhos trafegados na requisição *preflighted* podem ser visualizados na **Figura 2**.



**Figura 2.** Fluxo de requisições GET e POST preflighted.

Na **Figura 2**, 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.

Uma vez que a origem solicitante recebe a resposta da requisição *preflighted*, 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 3. Caso contrario, o navegador aborta o procedimento, não realizando a requisição original.



**Figura 3.** Fluxo de requisições GET e POST preflighted com validação bem sucedida.

Credenciais

Este sem dúvida é um dos recursos mais interessantes do CORS. O termo credenciais proposto pela especificação representa os cookies, autenticação HTTP e certificados SSL do lado do cliente. Por padrão, este recurso é desabilitado nas requisições entre domínios distintos e o **XMLHttpRequest** não enviará as credenciais. Para habilitar esta opção, um flag deve ser setado no objeto **XMLHttpRequest** após sua inicialização. Esta opção não é suportada pelo **XDomainRequest** utilizado pelo Internet Explorer 8.

Como dito anteriormente, o **XDomainRequest** implementa a especificação parcialmente.

O trecho de código apresentado na **Listagem 2** mostra como habilitar a flag **withCredentials** do objeto **XMLHttpRequest** para realizar requisições com cookie.

**Listagem 2.** Forma de utilizar o objeto XMLHttpRequest para requisições credenciadas.

**var** invocation = **new** XMLHttpRequest();

**var** url = 'http://provider.com.br/clientes/lista';

**function** callOtherDomain(){

**if**(invocation){

invocation.open('GET', url, **true**);

invocation.withCredentials = **true**;

invocation.send();

}

}

A linha 7 mostra a propriedade **withCredentials** setada com o valor **true**, indicando que a aplicação cliente deseja utilizar cookies ou serviços HTTP autenticados. Porém, para que o navegador tenha acesso a estas informações, as respostas do servidor devem conter o cabeçalho **Access-Control-Allow-Credentials: true**. Vale ressaltar que, por padrão, esta opção é setada como **false**, inibindo o tráfego de cookies e inviabilizando o consumo de serviços autenticados.

A Figura 4 demonstra os cabeçalhos trafegados, entre cliente e servidor, com base no código ilustrado na **Listagem 2**.

****

**Figura 4.** Fluxo de requisições com credencial.

Estudo de caso

Agora que temos em mente quais os pontos mais delicados na troca de informações “Cross-Domain”, onde o intermediário é o navegador do usuário final, criaremos um cenário onde possamos demonstrar na prática os conceitos apresentados até o momento.

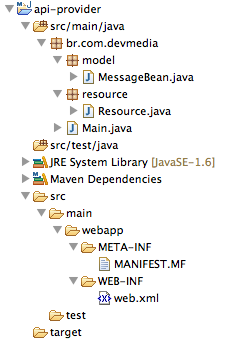
O cenário

Desenvolveremos duas aplicações: a aplicação api-provider, que conterá alguns Web Services simulando a disponibilização de uma API; e a aplicação api-consumer, que será um site que consumirá os Web Services de api-provider através de algumas 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.

A aplicação api-provider

A **Figura 5** 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 5.** Estrutura inicial da aplicação api-provider.

A implementação da classe MessageBean pode ser visualizada na **Listagem 3**. 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.

**Listagem 3.** Implementação da classe MessageBean.

**package** br.com.devmedia.model;

**import** javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement

**public** **class** MessageBean {

**private** String mensagem;

**public** MessageBean() {}

**public** MessageBean(String mensagem) {

**this**.mensagem = mensagem;

}

**public** String getMensagem() {

**return** mensagem;

}

**public** **void** setMensagem(String mensagem) {

**this**.mensagem = mensagem;

}

}

A seguir, apresentada na **Listagem 4**, 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.

**Listagem 4.** Implementação da classe Resource.

**package** br.com.devmedia.resource;

**import** javax.ws.rs.\*;

**import** javax.ws.rs.core.\*;

**import** br.com.devmedia.model.MessageBean;

@Path("/recurso")

@Consumes(MediaType.*WILDCARD*)

@Produces(MediaType.*APPLICATION\_JSON*)

**public** **class** Resource {

@GET

**public** MessageBean get() {

**return** **new** MessageBean("GET /recurso executado com sucesso!");

}

@POST

**public** MessageBean post() {

**return** **new** MessageBean("POST /recurso executado com sucesso!");

}

@PUT

**public** MessageBean put() {

**return** **new** MessageBean("PUT /recurso executado com sucesso!");

}

@DELETE

**public** MessageBean delete() {

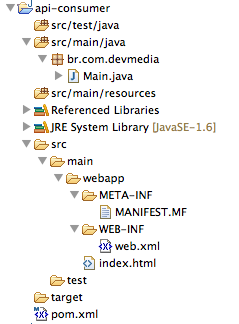
**return** **new** MessageBean("DELETE /recurso executado com sucesso!");

}

}

A aplicação api-consumer

Na **Figura 6**, 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 6.** 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.

**Listagem 5.** Implementação da página index.html.

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

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta http-equiv="Content-type" content="text/html; charset=UTF-8" />

<script type="text/javascript">

**function** consumirWebService(){

**var** destino = document.getElementById('destino\_requisicao').value;

**var** metodoHttp = document.getElementById('metodo\_http').value;

**var** status = document.getElementById('status');

**var** resposta = document.getElementById('resposta');

status.innerHTML = "Chamando Web Service: "+destino+" utilizando o método: "+metodoHttp;

**var** xhr = **new** XMLHttpRequest();

xhr.onreadystatechange = **function**(){

**if**(xhr.readyState != 4) **return**;

resposta.innerHTML = xhr.responseText;

resposta.style.backgroundColor = "#99FF33";

};

xhr.onerror = **function**(){

resposta.innerHTML = "Erro ao consumir serviços.";

resposta.style.backgroundColor = "#FFCC00";

};

xhr.open(metodoHttp, destino);

xhr.send();

};

</script>

</head>

<body style="font**:**12px verdana">

<h3>Exemplo de aplicação consumidora de Web Services com origem distinta</h3>

<h3>URL do recurso a ser consumido:</h3>

<input type="text" id="destino\_requisicao"

value="http://provider.com.br:9999/recurso" style="width**:**300px" /> <br />

<h3>Método HTTP:</h3>

<select id="metodo\_http">

<option value="get" selected="selected">get</option>

<option value="post">post</option>

<option value="put">put</option>

<option value="delete">delete</option>

</select> <br /><br />

<input type="button" onclick="javascript:consumirWebService();" value="Realizar requisição" /> <br />

<h3>Status da requisição:</h3> <div id="status"></div> <br />

<h3>Resposta recebida:</h3> <div id="resposta"></div>

</body>

</html>

Ainda analisando a **Listagem 5**, 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**.

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

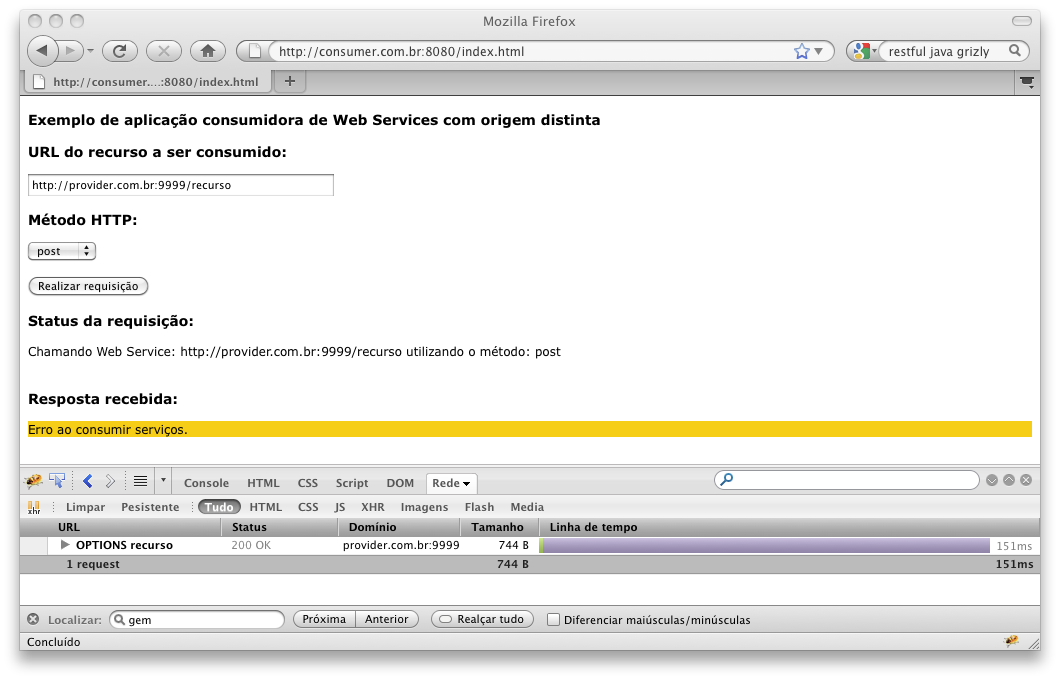
127.0.0.1 consumer.com.br

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 7**.

****

**Figura 7.** Tentativa falha de consumo de web services de origem distinta.

Os demais resultados obtidos, para cada verbo HTTP disponível na interface de parametrização, são apresentados na **Tabela 2**.

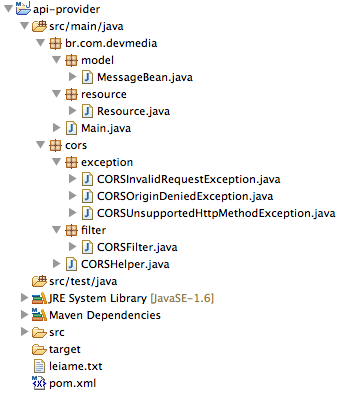
|  |  |  |
| --- | --- | --- |
| Recurso | Método HTTP | Resultado Obtido |
| http://provider.com.br:9999/recurso | GET | Falha |
| http://provider.com.br:9999/recurso | POST | Falha |
| http://provider.com.br:9999/recurso | PUT | Falha |
| http://provider.com.br:9999/recurso | DELETE | Falha |

**Tabela 2.** Resultados obtidos na tentativa de consumo dos web services "Cross-Domain".

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.

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 8**. 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 8.** Estrutura da aplicação api-provider com o filtro CORS.

A **Listagem 6** 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 *preflighted*, requisições originais (subsequentes às requisições *preflighted*) e requisições provenientes do mesmo domínio, ou seja, que não enviam os cabeçalhos estabelecidos pela especificação.

**Listagem 6.** Implementação da classe CORSFilter.

**package** cors.filter;

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**import** cors.CORSHelper;

**import** cors.exception.\*;

**public** **class** CORSFilter **implements** Filter {

**public** CORSHelper corsHelper = **new** CORSHelper();

@Override

**public** **void** doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)

**throws** IOException, ServletException {

HttpServletRequest request = (HttpServletRequest) servletRequest;

HttpServletResponse response = (HttpServletResponse) servletResponse;

**try** {

//trata requisições preflighted

**if** (corsHelper.isPreflightRequest(request)) {

corsHelper.handlePreflightRequest(request, response);

filterChain.doFilter(request, response);

**return**;

}

//trata requisições originais (subsequentes às requisições preflighted)

**if** (corsHelper.isOriginalRequest(request)) {

corsHelper.handleOriginalRequest(request, response);

filterChain.doFilter(request, response);

**return**;

}

// trata requisições do mesmo domínio

filterChain.doFilter(request, response);

} **catch** (CORSInvalidRequestException e) {

printMessage(response, HttpServletResponse.*SC\_BAD\_REQUEST*, e.getMessage());

} **catch** (CORSOriginDeniedException e) {

printMessage(response, HttpServletResponse.*SC\_FORBIDDEN*, e.getMessage());

} **catch** (CORSUnsupportedHttpMethodException e) {

printMessage(response, HttpServletResponse.*SC\_METHOD\_NOT\_ALLOWED*, e.getMessage());

}

}

@Override

**public** **void** init(FilterConfig filterConfig) **throws** ServletException {}

@Override

**public** **void** destroy() {}

**private** **void** printMessage(**final** HttpServletResponse response, **final** **int** sc, **final** String msg)

**throws** IOException, ServletException {

response.setStatus(sc);

response.resetBuffer();

response.setContentType("text/plain");

PrintWriter out = response.getWriter();

out.println("Filtro CORS: " + msg);

}

}

A classe CORSHelper, indicada na **Listagem 7**, 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()**.

**Listagem 7.** Implementação da classe CORSHelper.

**package** cors;

**import** javax.servlet.http.\*;

**import** cors.exception.\*;

**public** **class** CORSHelper {

**private** **static** **final** String *ALLOW\_METHODS* = "GET, POST, PUT, DELETE";

**private** **static** **final** String *ALLOW\_CREDENTIALS* = "false";

**private** **static** **final** String *ALLOW\_HEADERS* = "X-Requested-With, X-Prototype-Version, XDomainRequest";

**private** **static** **final** String *MAX\_AGE* = "5";

**public** **boolean** isPreflightRequest(HttpServletRequest request){

**return** (request.getHeader("Origin") != **null** &&

request.getHeader("Access-Control-Request-Method") != **null** &&

request.getMethod() != **null** &&

request.getMethod().equals("OPTIONS"));

}

**public** **void** handlePreflightRequest(HttpServletRequest request, HttpServletResponse response)

**throws** CORSInvalidRequestException, CORSOriginDeniedException, CORSUnsupportedHttpMethodException {

String originHeader = request.getHeader("Origin");

String requestMethodHeader = request.getHeader("Access-Control-Request-Method");

**if**(! *ALLOW\_METHODS*.contains(requestMethodHeader))

**throw** **new** CORSUnsupportedHttpMethodException();

**this**.addControlHeaders(response, originHeader);

}

**public** **boolean** isOriginalRequest(HttpServletRequest request){

**return** (request.getHeader("Origin") != **null** &&

request.getMethod() != **null** &&

!request.getMethod().equals("OPTIONS"));

}

**public** **void** handleOriginalRequest(HttpServletRequest request, HttpServletResponse response)

**throws** CORSInvalidRequestException, CORSOriginDeniedException, CORSUnsupportedHttpMethodException {

String originHeader = request.getHeader("Origin");

String httpMethod = request.getMethod();

**if**(! *ALLOW\_METHODS*.contains(httpMethod))

**throw** **new** CORSUnsupportedHttpMethodException();

**this**.addControlHeaders(response, originHeader);

}

**private** **void** addControlHeaders(HttpServletResponse response, String origin) {

response.addHeader("Access-Control-Allow-Origin", origin);

response.addHeader("Access-Control-Allow-Credentials", *ALLOW\_CREDENTIALS*);

response.addHeader("Access-Control-Allow-Methods", *ALLOW\_METHODS*);

response.addHeader("Access-Control-Allow-Headers", *ALLOW\_HEADERS*);

response.addHeader("Access-Control-Max-Age", *MAX\_AGE*);

}

}

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**.

**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>

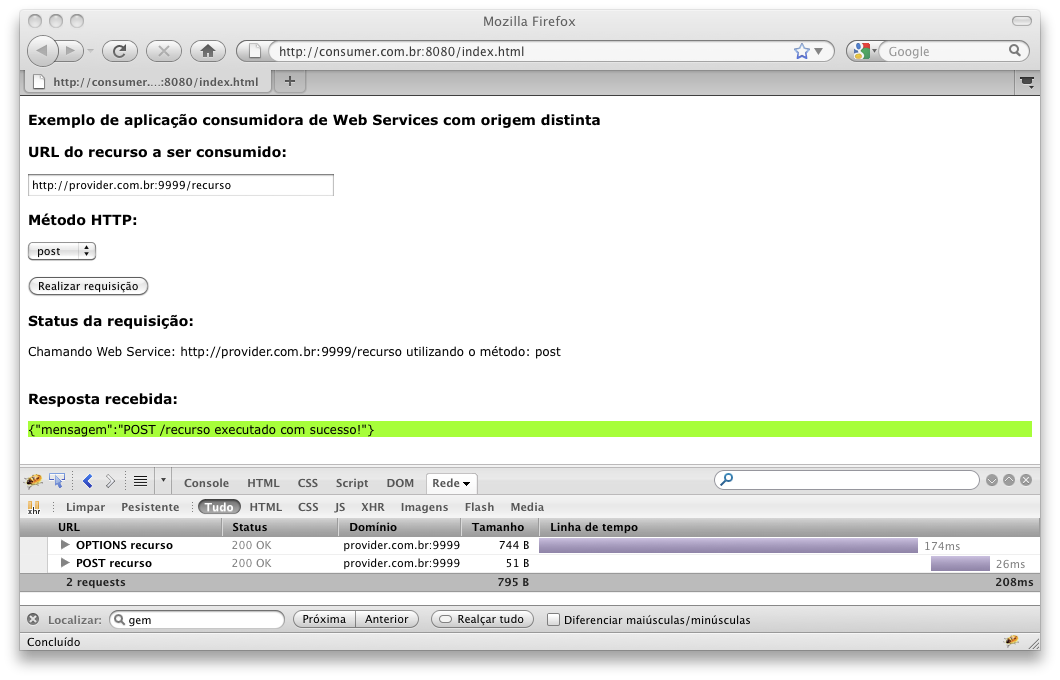
</web-app>

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 9**.



**Figura 9.** Consumo de web services de origem distinta realizado com sucesso.

Os demais resultados obtidos são apresentados na **Tabela 3**.

|  |  |  |
| --- | --- | --- |
| 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 na tentativa de consumo dos web services "Cross-Domain".

Conclusão

Este artigo demonstrou a especificação Cross-Origin Resource Sharing da W3C combinada com os recursos de filtro 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.

Os navegadores atuais dão um sólido suporte para cross-domain Ajax, mas a maioria dos desenvolvedores ainda não está consciente deste poderoso mecanismo, capaz de tornar recursos compartilhados entre diferentes origens com segurança, ou seja, podendo restringir os acessos e ações de cada origem.

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 corretos sejam enviados.

Vale ressaltar que a implementação do Internet Explorer 8 deixa um pouco a desejar em relação ao tráfego de credenciais e requisições *preflighted*, no entanto acreditamos que o seu suporte ao CORS continuará a melhorar.

De que se trata o artigo:

Aplicação do padrão W3C Cross-Origin Resource Sharing no desenvolvimento de sistemas que disponibilizem recursos consumíveis por aplicações de outros domínios através de chamadas AJAX, estendendo o conceito de segurança Same Origin Policy.

Em que situação o tema é útil:

Este tema é útil quando precisamos desenvolver aplicações que disponibilizam web services consumíveis por aplicações que não estejam no mesmo domínio ou sub-domínio de rede, pois permite a realização de comunicação Cross-Domain.

Resumo DevMan:

O artigo aborda os problemas e dificuldades, derivadas do conceito de segurança Same Origin Policy, na utilização de métodos AJAX para o consumo de recursos de outros domínios, bem como, apresenta como solução a especificação Cross-Origin Resource Sharing.

A especificação CORS é abordada de forma abrangente, detalhando o seu funcionamento teórico, comentando o atual suporte dos navegadores, e demonstrando sua aplicabilidade prática por meio de um estudo de caso, onde uma implementação é desenvolvida com base na tecnologia de filtros da linguagem Java.



**Paulo Vitor da Silva Rendeiro** (*paulovittor23@gmail.com*) é bacharel em Sistemas de Informação pela Universidade Católica de Santos, pós-graduado em Engenharia de Software pela PUC-SP e possui as certificações SCJP 5.0, SCWCD 1.4, SCBCD 5. Atualmente trabalha como desenvolvedor web na Editora Abril.



**Alexsandro Terto Santos da Silva** (*alexsandro.tertoss@gmail.com*) é desenvolvedor web senior na Editora Abril, formado em ciência da computação, java programmer certified, atuando há mais de 10 anos em desenvolvimento de sistemas e tunning de aplicações.

Links

**Cross Origin Resource Sharing - Referência do W3C:**

[**http://www.w3.org/TR/cors/**](http://www.w3.org/TR/cors/)

**Cross-site HTTP requests - Referência do Mozilla Dev:**

[**https://developer.mozilla.org/En/HTTP\_access\_control**](https://developer.mozilla.org/En/HTTP_access_control)

**Same origin policy for JavaScript - Referência do Mozilla Dev:**

[**https://developer.mozilla.org/en/Same\_origin\_policy\_for\_JavaScript**](https://developer.mozilla.org/en/Same_origin_policy_for_JavaScript)

API Java para web services RESTful

<http://jsr311.java.net/>

**Implementação de referência JAX-RS**

<http://jersey.java.net/>

**Documentação do método Ajax da API JQuery**

**http://api.jquery.com/jQuery.ajax/**