

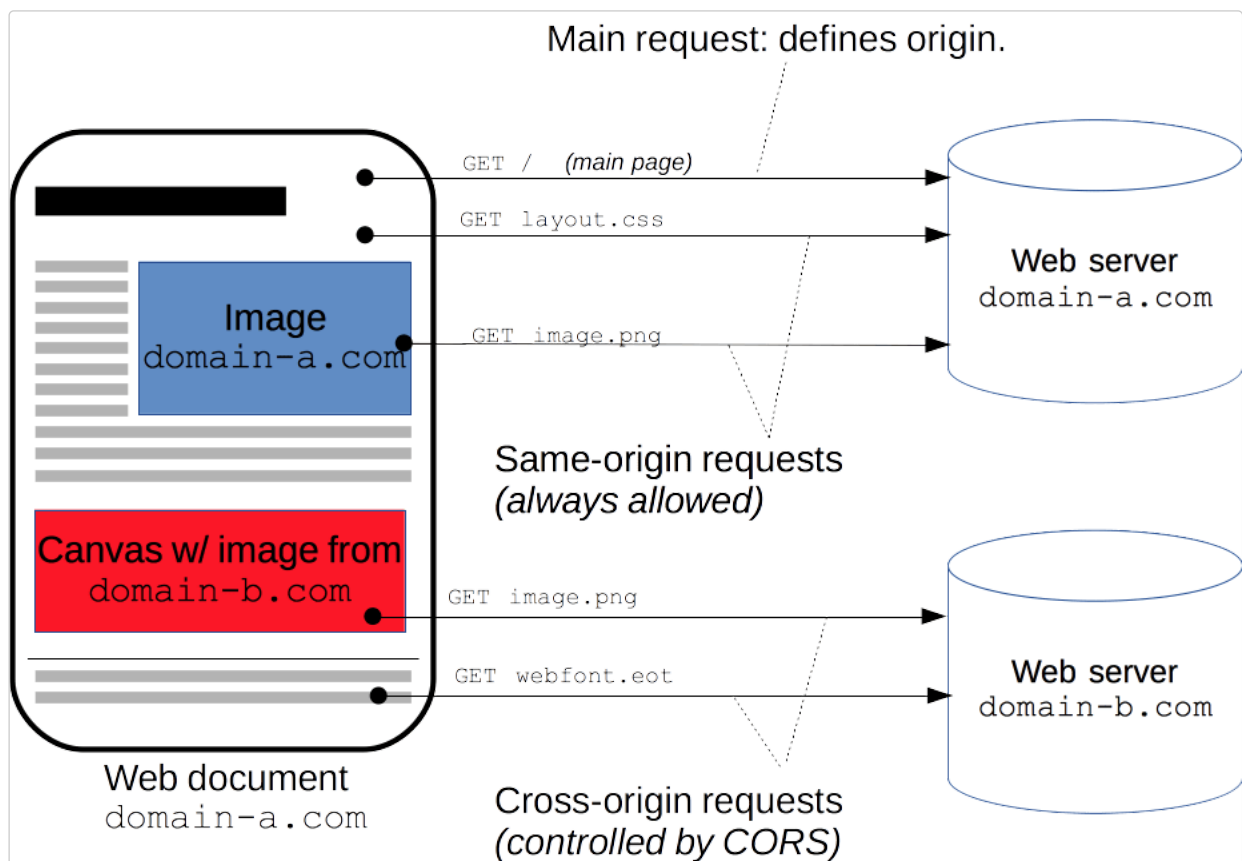
 This page was translated from English by the community. Learn more and join the MDN Web Docs community.

## Cross-Origin Resource Sharing (CORS)

[CORS](#) - Cross-Origin Resource Sharing (Compartilhamento de recursos com origens diferentes) é um mecanismo que usa cabeçalhos adicionais [HTTP](#) para informar a um navegador que permita que um aplicativo Web seja executado em uma origem (domínio) com permissão para acessar recursos selecionados de um servidor em uma origem distinta. Um aplicativo Web executa uma **requisição *cross-origin* HTTP** ao solicitar um recurso que tenha uma origem diferente (domínio, protocolo e porta) da sua própria origem.

Um exemplo de requisição *cross-origin*: o código JavaScript *frontend* de um aplicativo web disponível em `http://domain-a.com` usa [XMLHttpRequest](#) para fazer uma requisição para `http://api.domain-b.com/data.json`.

Por motivos de segurança, navegadores restringem requisições *cross-origin* HTTP iniciadas por scripts. Por exemplo, `XMLHttpRequest` e [Fetch API](#) seguem a [política de mesma origem \(en-US\)](#) (*same-origin policy*). Isso significa que um aplicativo web que faz uso dessas APIs só poderá fazer solicitações para recursos de mesma origem da qual o aplicativo foi carregado, a menos que a resposta da outra origem inclua os cabeçalhos CORS corretos.



O mecanismo CORS suporta requisições seguras do tipo *cross-origin* e transferências de dados entre navegadores e servidores web. Navegadores modernos usam o CORS em uma API contêiner, como `XMLHttpRequest` ou [Fetch](#), para ajudar a reduzir os riscos de requisições *cross-origin* HTTP.

### Quem deve ler este artigo?

Todos, realmente.

Este artigo destina-se a administradores da Web, desenvolvedores de servidores e desenvolvedores front-end. Os navegadores modernos lidam com os componentes do lado cliente em compartilhamento entre origens, incluindo cabeçalhos e aplicação de políticas. Mas esse novo padrão significa que os servidores precisam lidar com novos cabeçalhos de requisição e resposta. Outro artigo para desenvolvedores de servidores que discutem [compartilhamento cross-origin a partir de uma perspectiva de servidor \(com fragmentos de código PHP\)](#), pode ser uma leitura complementar.

## Quais solicitações usam o CORS?

Esse [padrão de compartilhamento cross-origin](#) <sup>↗</sup> é usado para habilitar solicitações HTTP entre sites para:

- Chamadas [XMLHttpRequest](#) ou [Fetch API](#) pela comunicação entre origens diferentes, tal como discutido acima.
- Web Fonts (para o uso de fontes pelo *cross-domain* em `@font` do CSS), [para que os servidores possam implantar fontes TrueType que só podem ser carregadas em origens diferentes e usadas por sites com autorização para isso](#) <sup>↗</sup>.
- [Texturas WebGL](#) <sup>(en-US)</sup>.
- *Frames* de Imagens/vídeos desenhados em uma tela usando [drawImage\(\)](#) <sup>(en-US)</sup>.

Este artigo é uma discussão geral sobre *Cross-Origin Resource Sharing* (Compartilhamento de recursos com origens diferentes) e inclui uma discussão de cabeçalhos HTTP necessários.

## Visão Geral

O padrão *Cross-Origin Resource Sharing* trabalha adicionando novos [cabeçalhos HTTP](#) que permitem que os servidores descrevam um conjunto de origens que possuem permissão a ler uma informação usando o navegador. Além disso, para métodos de requisição HTTP que podem causar efeitos colaterais nos dados do servidor (em particular, para métodos HTTP diferentes de [GET](#) ou para uso de [POST](#) com certos [MIME types](#)), a especificação exige que navegadores "pré-enviem" a requisição, solicitando os métodos suportados pelo servidor com um método de requisição HTTP [OPTIONS](#) e, após a "aprovação", o servidor envia a requisição verdadeira com o método de requisição HTTP correto. Servidores também podem notificar clientes se "credenciais" (incluindo [Cookies](#) e dados de autenticação HTTP) devem ser enviadas com as requisições.

Falhas no CORS resultam em erros, mas por questões de segurança, detalhes sobre erros não estão disponíveis no código JavaScript. O código tem apenas conhecimento de que ocorreu um erro. A única maneira para determinar especificamente o que ocorreu de errado é procurar no console do navegador por mais detalhes.

Seções subsequentes discutem cenários, assim como fornecem um detalhamento dos cabeçalhos HTTP utilizados.

## Exemplos de cenários com controle de acesso

Aqui, apresentamos três cenários que ilustram como *Cross-Origin Resource Sharing* funciona. Todos estes exemplos usam o objeto [XMLHttpRequest](#), que pode ser utilizado para fazer requisições entre origens em qualquer navegador compatível.

Os snippets JavaScript inclusos nessas seções (e instâncias executáveis de código do lado servidor que tratam corretamente essas requisições entre origens) podem ser encontrados "em ação" aqui: <http://arunranga.com/examples/access-control/> <sup>↗</sup>, e irão funcionar em navegadores que suportam `XMLHttpRequest` entre origens.

Uma discussão sobre *Cross-Origin Resource Sharing* a partir da perspectiva do servidor (incluindo snippets de código PHP) pode ser encontrada no artigo [Server-Side Access Control \(CORS\)](#) <sup>(en-US)</sup>.

## Requisições simples

Algumas requisições não acionam um [pré-envio CORS](#) <sup>(en-US)</sup>. Essas são denominadas neste artigo como "requisições simples" (*simple request*), embora a especificação [Fetch](#) <sup>↗</sup> (que define CORS) não utilize esse termo. Uma requisição que não aciona um [pré-envio CORS](#) <sup>(en-US)</sup> — denominada "requisição simples" — é uma que **atende todas as seguintes condições**:

- Os únicos métodos permitidos são:

- [GET](#)
- [HEAD](#)
- [POST](#)
- Além dos cabeçalhos definidos automaticamente pelo agente do usuário (por exemplo, [Connection](#), [User-Agent](#) ou [qualquer um dos outros cabeçalhos com nomes definidos na especificação Fetch como "forbidden header name"](#) <sup>↗</sup>), os únicos cabeçalhos que podem ser definidos manualmente são [aqueles cujo a especificação Fetch define como sendo um "CORS-safelisted request-header"](#) <sup>↗</sup>, que são:
  - [Accept](#)
  - [Accept-Language](#)
  - [Content-Language](#)
  - [Content-Type](#) (porém observe os requisitos adicionais abaixo)
- Os únicos valores permitidos para o [Content-Type](#) do cabeçalho são:
  - `application/x-www-form-urlencoded`
  - `multipart/form-data`
  - `text/plain`
- Nenhum *event listener* é registrado em qualquer objeto `XMLHttpRequestUpload` usado na requisição, estes são acessados usando a propriedade [XMLHttpRequest.upload](#) <sup>(en-US)</sup>.
- Nenhum objeto [ReadableStream](#) <sup>(en-US)</sup> é usado na requisição.

**Nota:** Esses são os mesmos tipos de requisições entre origens distintas que o conteúdo da web já pode realizar e nenhum dado de resposta é liberado ao solicitante, a menos que o servidor envie um cabeçalho adequado. Portanto, sites que impedem a falsificação de requisições entre origens não tem nada a temer em relação ao controle de acesso HTTP.

**Nota:** O WebKit Nightly e Safari Technology Preview impõem restrições adicionais nos valores permitidos nos cabeçalhos [Accept](#), [Accept-Language](#) e [Content-Language](#). Caso algum destes cabeçalhos tenham valores "não-padronizados", o WebKit/Safari não considera que a requisição atenda as condições para uma "requisição simples". O que o WebKit/Safari considera valores "não-padronizados" para estes cabeçalhos não é documentado exceto nos seguintes bugs do WebKit: [Require preflight for non-standard CORS-safelisted request headers Accept, Accept-Language, and Content-Language](#) <sup>↗</sup>, [Allow commas in Accept, Accept-Language, and Content-Language request headers for simple CORS](#) <sup>↗</sup> e [Switch to a blacklist model for restricted Accept headers in simple CORS requests](#) <sup>↗</sup>. Nenhum outro navegador implementa estas restrições adicionais, pois elas não são parte da especificação.

Por exemplo, suponha que o conteúdo web no domínio `http://foo.example` deseje chamar ( *invocation* do exemplo abaixo) um outro conteúdo no domínio `http://bar.other`. Esse código Javascript pode estar hospedado em `foo.example`:

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/public-data/';

function callOtherDomain() {
  if(invocation) {
    invocation.open('GET', url, true);
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}
```

Isso fará uma troca simples entre cliente e servidor, utilizando o cabeçalho CORS para tratar os privilégios.



Neste caso, vamos ver o que o navegador enviará ao servidor e vamos olhar como o servidor responde:

```

GET /resources/public-data/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://foo.example/examples/access-control/simpleXSIInvocation.html
Origin: http://foo.example

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 00:23:53 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: *
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: application/xml

[XML Data]

```

As linhas de 1 a 10 são enviadas no header. Note que o cabeçalho principal da requisição HTTP aqui é [Origin](#) na linha 10, o qual revela que a chamada é proveniente de um conteúdo no domínio `http://foo.example`.

As linhas de 13 a 22 mostram a resposta HTTP do servidor no domínio `http://bar.other`. Nesta resposta, o servidor envia de volta um cabeçalho [Access-Control-Allow-Origin](#) exibido na linha 16. O uso dos cabeçalhos [Origin](#) e [Access-Control-Allow-Origin](#) mostram o protocolo de controle de acesso em seu uso mais simples. Neste caso, o servidor responde com `Access-Control-Allow-Origin: *`, o que significa que o recurso pode ser acessado por **qualquer** domínio pela comunicação entre origens. Se os proprietários dos recursos em `http://bar.other` desejarem restringir o acesso ao conteúdo para o mesmo ser apenas de `http://foo.example`, eles retornaram:

```
Access-Control-Allow-Origin: http://foo.example
```

Observe que, agora, nenhum domínio além de `http://foo.example` (identificado na requisição pelo cabeçalho `ORIGIN`: como na linha 10) pode acessar o recurso pela comunicação entre origens. O cabeçalho [Access-Control-Allow-Origin](#) deve conter o valor que foi enviado no cabeçalho [origin](#) da requisição.


## Requisições com pré-envio

Ao contrário de ["requisições simples" \(discutido acima\) \(en-US\)](#), requisições com "pré-envio" (*Preflighted requests*) primeiramente enviam uma requisição HTTP através do método [OPTIONS](#) para obter um recurso em outro domínio, a fim de determinar se de fato a requisição atual é segura para envio. Requisições entre sites possuem pré-envio, já que podem interferir em dados do usuário.

Em particular, uma requisição tem um pré-envio **se qualquer das seguintes condições** for verdadeira:

- Se a requisição usa algum dos seguintes métodos:

- [PUT](#)
- [DELETE](#)
- [CONNECT](#)
- [OPTIONS](#)
- [TRACE](#)
- [PATCH](#)
- **Ou se**, além dos cabeçalhos definidos automaticamente pelo agente do usuário (por exemplo, [Connection](#), {HTTPHeader("User-Agent")}) ou qualquer OUTRO cabeçalho com um nome definido na especificação Fetch como “forbidden header name” [↗](#), a requisição inclui quaisquer cabeçalhos **além** daqueles que a especificação Fetch define como sendo um “CORS-safelisted request-header” [↗](#), que são:
  - [Accept](#)
  - [Accept-Language](#)
  - [Content-Language](#)
  - [Content-Type](#) (porém observe os requisitos adicionais abaixo)
  - [DPR](#) [↗](#)
  - [Downlink](#) [\(en-US\)](#)
  - [Save-Data](#) [↗](#)
  - [Viewport-Width](#) [↗](#)
  - [Width](#) [↗](#)
- **Ou se** o [Content-Type](#) do cabeçalho **tem outro** valor que:
  - `application/x-www-form-urlencoded`
  - `multipart/form-data`
  - `text/plain`

 [mdn web docs](#)

- **Ou se** um objeto [ReadableStream](#) [\(en-US\)](#) é usado nessa requisição.

**Nota:** WebKit Nightly e Safari Technology Preview colocam restrições adicionais nos valores permitidos dos cabeçalhos [Accept](#), [Accept-Language](#) e [Content-Language](#). Caso qualquer um desses cabeçalhos tenha algum valor fora do padrão (non-standard), o WebKit/Safari faz o pré-envio da requisição. O que o WebKit/Safari considera como valor “non-standard” para tais cabeçalhos não está documentado, exceto nos seguintes bugs do WebKit: [Require preflight for non-standard CORS-safelisted request headers Accept, Accept-Language, and Content-Language](#) [↗](#), [Allow commas in Accept, Accept-Language, e Content-Language request headers for simple CORS](#) [↗](#) e [Switch to a blacklist model for restricted Accept headers in simple CORS requests](#) [↗](#). Nenhum outro navegador implementa estas restrições adicionais, pois elas não são parte da especificação.

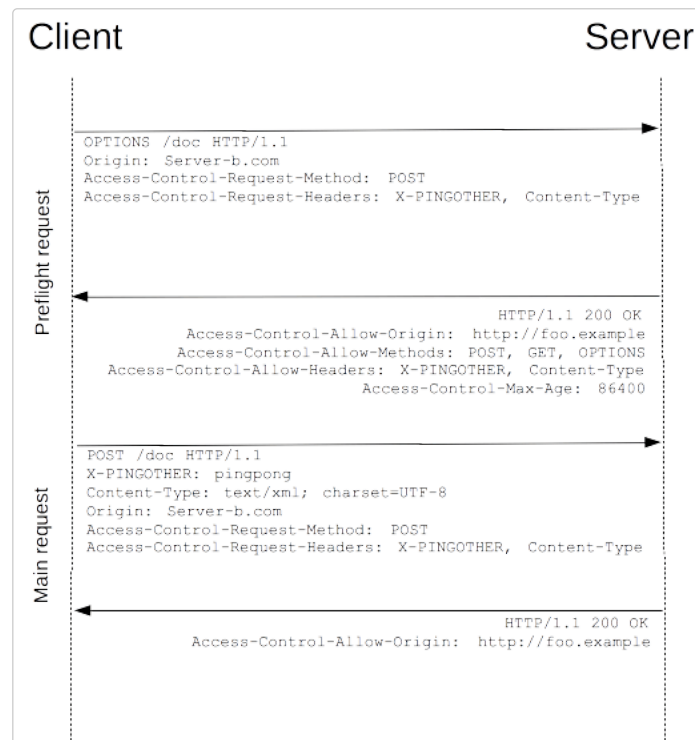
O exemplo a seguir é de uma requisição com pré-envio.

```
var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/post-here/';
var body = '<?xml version="1.0"?><person><name>Arun</name></person>';

function callOtherDomain(){
  if(invocation)
  {
    invocation.open('POST', url, true);
    invocation.setRequestHeader('X-PINGOTHER', 'pingpong');
    invocation.setRequestHeader('Content-Type', 'application/xml');
    invocation.onreadystatechange = handler;
    invocation.send(body);
  }
}
```

.....

No exemplo acima, a linha 3 cria um XML para enviar com a requisição `POST` da linha 8. Também, na linha 9, é definido um cabeçalho de uma requisição HTTP "personalizada" (non-standard) com (`X-PINGOTHER: pingpong`). Tais cabeçalhos não fazem parte do protocolo HTTP/1.1, mas podem ser usados para aplicações web. Já que a requisição usa um Content-Type do tipo `application/xml` e como é uma requisição personalizada, esta requisição faz um pré-envio.



(Observação: conforme descrito abaixo, a requisição POST real não inclui os cabeçalhos `Access-Control-Request-*`; eles são necessários apenas para a requisição `OPTIONS`.)

Vamos conferir a comunicação completa que ocorre entre cliente e servidor. A primeira comunicação é a *requisição com pré-envio/resposta*:

```

OPTIONS /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Origin: http://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
  
```

```
Connection: Keep-Alive
Content-Type: text/plain
```

Uma vez que a requisição com pré-envio é completa, a requisição efetiva será enviada:

```
POST /resources/post-here/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
X-PINGOTHER: pingpong
Content-Type: text/xml; charset=UTF-8
Referer: http://foo.example/examples/preflightInvocation.html
Content-Length: 55
Origin: http://foo.example
Pragma: no-cache
Cache-Control: no-cache

<?xml version="1.0"?><person><name>Arun</name></person>

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:40 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://foo.example
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 235
Keep-Alive: timeout=2, max=99
Connection: Keep-Alive
Content-Type: text/plain

[Some GZIP'd payload]
```

As linhas de 1 a 12 acima representam a requisição com pré-envio tendo o método [OPTIONS](#). O navegador determina que precisa fazer este envio baseado nos parâmetros da requisição do código JavaScript acima utilizado, para que o servidor possa responder caso seja aceitável o envio da requisição com os dados parâmetros da mesma. OPTIONS é um método HTTP/1.1 usado para determinar informações complementares dos servidores, sendo o mesmo um método [safe](#), o que significa que não pode ser utilizado para troca de recurso. Note que junto da requisição OPTIONS, outros dois cabeçalhos são enviados (linhas 10 e 11, respectivamente):

```
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-Type
```

O cabeçalho [Access-Control-Request-Method](#) notifica o servidor como sendo uma parte da requisição com pré-envio que, quando a requisição efetiva é enviada, será enviada com uma requisição de método `POST`. O cabeçalho [Access-Control-Request-Headers](#) notifica o servidor que quando a requisição efetiva for enviada, será enviada com os seguintes cabeçalhos personalizados `X-PINGOTHER` e `Content-Type`. O servidor agora tem a oportunidade para definir se deseja aceitar uma requisição sob estas condições.

As linhas 14 a 26 acima são as respostas que o servidor devolve, indicando que o método (`POST`) e os cabeçalhos (`X-PINGOTHER`) da requisição são aceitáveis. Em particular, vejamos as linhas 17 a 20:

```
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
```



O servidor responde com `Access-Control-Allow-Methods` e diz que `POST`, `GET`, e `OPTIONS` são métodos viáveis para requerir o recurso em questão. Perceba que este cabeçalho é similar ao cabeçalho da resposta `Allow`, mas usado estritamente dentro do contexto do controle de acesso.

O servidor envia também `Access-Control-Allow-Headers` com um valor de `"X-PINGOTHER, Content-Type"`, confirmando estes são cabeçalhos permitidos a serem usados com a requisição efetiva. Assim como `Access-Control-Allow-Methods`, `Access-Control-Allow-Headers` é uma lista de cabeçalhos aceitáveis, separados por vírgula.

Por fim, `Access-Control-Max-Age` traz o valor em segundos de quão longo pode ser mantida em cache a resposta da requisição pré-envio sem o envio de outra requisição pré-envio. Neste caso, 86400 segundos são 24 horas. Note que cada browser tem um [valor interno máximo](#) que toma precedência quando `Access-Control-Max-Age` for maior.

## Requisições com pré-envio e redirecionamento

Not all browsers currently support following redirects after a preflighted request. If a redirect occurs after a preflighted request, some browsers currently will report an error message such as the following.

The request was redirected to '<https://example.com/foo>', which is disallowed for cross-origin requests that require preflight

Request requires preflight, which is disallowed to follow cross-origin redirect

The CORS protocol originally required that behavior but [was subsequently changed to no longer require it](#). However, not all browsers have implemented the change, and so still exhibit the behavior that was originally required.

So until all browsers catch up with the spec, you may be able to work around this limitation by doing one or both of the following:

- change the server-side behavior to avoid the preflight and/or to avoid the redirect—if you have control over the server the request is being made to
- change the request such that it is a [simple request]#simple\_requests) that doesn't cause a preflight

But if it's not possible to make those changes, then another way that may be possible is to this:

1. Make a [simple request](#) (using `Response.url` [\(en-US\)](#) for the Fetch API, or `XMLHttpRequest.responseURL` [\(en-US\)](#)) to determine what URL the real preflighted request would end up at.
2. Make another request (the "real" request) using the URL you obtained from `Response.url` or `XMLHttpRequest.responseURL` in the first step.

However, if the request is one that triggers a preflight due to the presence of the `Authorization` header in the request, you won't be able to work around the limitation using the steps above. And you won't be able to work around it at all unless you have control over the server the request is being made to.

## Requisições com credenciais

The most interesting capability exposed by both `XMLHttpRequest` or `Fetch` and CORS is the ability to make "credentialed" requests that are aware of [HTTP cookies](#) and HTTP Authentication information. By default, in cross-site `XMLHttpRequest` or `Fetch` invocations, browsers will **not** send credentials. A specific flag has to be set on the `XMLHttpRequest` object or the `Request` constructor when it is invoked.

In this example, content originally loaded from `http://foo.example` makes a simple GET request to a resource on `http://bar.other` which sets Cookies. Content on `foo.example` might contain JavaScript like this:



```

var invocation = new XMLHttpRequest();
var url = 'http://bar.other/resources/credentialed-content/';

function callOtherDomain(){
  if(invocation) {
    invocation.open('GET', url, true);
    invocation.withCredentials = true;
    invocation.onreadystatechange = handler;
    invocation.send();
  }
}

```

Line 7 shows the flag on `XMLHttpRequest` that has to be set in order to make the invocation with Cookies, namely the `withCredentials` boolean value. By default, the invocation is made without Cookies. Since this is a simple `GET` request, it is not preflighted, but the browser will **reject** any response that does not have the `Access-Control-Allow-Credentials: true` header, and **not** make the response available to the invoking web content.



Here is a sample exchange between client and server:

```

GET /resources/access-control-with-credentials/ HTTP/1.1
Host: bar.other
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.1b3pre) Gecko/20081130 Minefield/3.1b3pre
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://foo.example/examples/credential.html
Origin: http://foo.example
Cookie: pageAccess=2

HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:34:52 GMT
Server: Apache/2.0.61 (Unix) PHP/4.4.7 mod_ssl/2.0.61 OpenSSL/0.9.7e mod_fastcgi/2.4.2 DAV/2 SVN/1.4.2
X-Powered-By: PHP/5.2.6
Access-Control-Allow-Origin: http://foo.example
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Pragma: no-cache
Set-Cookie: pageAccess=3; expires=Wed, 31-Dec-2008 01:34:53 GMT
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 106
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain

[text/plain payload]

```

Although line 11 contains the Cookie destined for the content on `http://bar.other`, if `bar.other` did not respond with an `Access-Control-Allow-Credentials : true` (line 19) the response would be ignored and not made available to web content.

## Solicitações credenciadas e curingas (*wildcards*)

When responding to a credentialed request, the server **must** specify an origin in the value of the `Access-Control-Allow-Origin` header, instead of specifying the `"*"` wildcard.

Because the request headers in the above example include a `Cookie` header, the request would fail if the value of the `Access-Control-Allow-Origin` header were `"*"`. But it does not fail: Because the value of the `Access-Control-Allow-Origin` header is `"http://foo.example"` (an actual origin) rather than the `"*"` wildcard, the credential-cognizant content is returned to the invoking web content.

Note that the `Set-Cookie` response header in the example above also sets a further cookie. In case of failure, an exception—depending on the API used—is raised.

All of these examples can be [seen working here](#). The next section deals with the actual HTTP headers.

## Os cabeçalhos de resposta HTTP

This section lists the HTTP response headers that servers send back for access control requests as defined by the Cross-Origin Resource Sharing specification. The previous section gives an overview of these in action.

### Access-Control-Allow-Origin

A returned resource may have one `Access-Control-Allow-Origin` header, with the following syntax:

```
Access-Control-Allow-Origin: <origin> | *
```

The `origin` parameter specifies a URI that may access the resource. The browser must enforce this. For requests **without** credentials, the server may specify `"*"` as a wildcard, thereby allowing any origin to access the resource.

For example, to allow `http://mozilla.org` to access the resource, you can specify:

```
Access-Control-Allow-Origin: http://mozilla.org
```

If the server specifies an origin host rather than `"*"`, then it could also include `Origin` in the `Vary` response header to indicate to clients that server responses will differ based on the value of the `Origin` request header.

### Access-Control-Expose-Headers

The `Access-Control-Expose-Headers` header lets a server whitelist headers that browsers are allowed to access. For example:

```
Access-Control-Expose-Headers: X-My-Custom-Header, X-Another-Custom-Header
```

This allows the `X-My-Custom-Header` and `X-Another-Custom-Header` headers to be exposed to the browser.

### Access-Control-Max-Age

The `Access-Control-Max-Age` header indicates how long the results of a preflight request can be cached. For an example of a preflight request, see the above examples.

```
Access-Control-Max-Age: <delta-seconds>
```

The `delta-seconds` parameter indicates the number of seconds the results can be cached.

## Access-Control-Allow-Credentials

The [Access-Control-Allow-Credentials](#) header indicates whether or not the response to the request can be exposed when the `credentials` flag is true. When used as part of a response to a preflight request, this indicates whether or not the actual request can be made using credentials. Note that simple `GET` requests are not preflighted, and so if a request is made for a resource with credentials, if this header is not returned with the resource, the response is ignored by the browser and not returned to web content.

```
Access-Control-Allow-Credentials: true
```

[Credentialed requests](#) are discussed above.

## Access-Control-Allow-Methods

O [Access-Control-Allow-Methods](#) cabeçalho especifica o método ou os métodos permitidos ao acessar o recurso. Isso é usado em resposta há uma requisição preflight. As condições na qual a requisição é preflight são discutidas à seguir.

```
Access-Control-Allow-Methods: <method>[, <method>]*
```

An example of a [preflight request is given above](#), including an example which sends this header to the browser.

## Access-Control-Allow-Headers

The [Access-Control-Allow-Headers](#) header is used in response to a [preflight request](#) to indicate which HTTP headers can be used when making the actual request.

```
Access-Control-Allow-Headers: <field-name>[, <field-name>]*
```

## Os cabeçalhos de solicitação HTTP

This section lists headers that clients may use when issuing HTTP requests in order to make use of the cross-origin sharing feature. Note that these headers are set for you when making invocations to servers. Developers using cross-site [XMLHttpRequest](#) capability do not have to set any cross-origin sharing request headers programmatically.

### Origin

The [origin](#) header indicates the origin of the cross-site access request or preflight request.

```
Origin: <origin>
```

The origin is a URI indicating the server from which the request initiated. It does not include any path information, but only the server name.

**Nota:** The `origin` can be the empty string; this is useful, for example, if the source is a `data` URL.

Note that in any access control request, the [origin](#) header is **always** sent.

## Access-Control-Request-Method

The [Access-Control-Request-Method](#) is used when issuing a preflight request to let the server know what HTTP method will be used when the actual request is made.

```
Access-Control-Request-Method: <method>
```

Examples of this usage can be [found above](#).

## Access-Control-Request-Headers

The [Access-Control-Request-Headers](#) header is used when issuing a preflight request to let the server know what HTTP headers will be used when the actual request is made.

```
Access-Control-Request-Headers: <field-name>[, <field-name>]*
```

Examples of this usage can be [found above](#).

## Especificações

Specification	Status	Comment
<a href="#">Fetch</a> <a href="#">The definition of 'CORS' in that specification.</a>	Padrão em tempo real	New definition; supplants <a href="#">W3C CORS</a> specification.

## Compatibilidade com navegadores

[Report problems with this compatibility data on GitHub](#)

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	
Access-Control-Allow-Origin	✓ Chrome 4	✓ Edge 12	✓ Firefox 3.5	✓ Opera 12	✓ Safari 4	✓ Chrome Yes Android	✓ Firefox 4 for Android	✓ Opera 12 Android	✓ Sa o iC

Tip: you can click/tap on a cell for more information.

✓ Full support

## Notas de compatibilidade

- Internet Explorer 8 and 9 expose CORS via the `XDomainRequest` object, but have a full implementation in IE 10.
- While Firefox 3.5 introduced support for cross-site XMLHttpRequests and Web Fonts, certain requests were limited until later versions. Specifically, Firefox 7 introduced the ability for cross-site HTTP requests for WebGL Textures, and Firefox 9 added support for Images drawn on a canvas using `drawImage`.

## Veja também

- [CORS errors](#)
- [Enable CORS: I want to add CORS support to my server](#)
- [XMLHttpRequest](#)
- [Fetch API](#)
- [Using CORS with All \(Modern\) Browsers](#)
- [Using CORS - HTML5 Rocks](#)
- [Code Samples Showing XMLHttpRequest and Cross-Origin Resource Sharing](#)
- [Client-Side & Server-Side \(Java\) sample for Cross-Origin Resource Sharing \(CORS\)](#)
- [Cross-Origin Resource Sharing From a Server-Side Perspective \(PHP, etc.\) \(en-US\)](#)
- [Stack Overflow answer with “how to” info for dealing with common problems](#)

- How to avoid the CORS preflight
- How to use a CORS proxy to get around *"No Access-Control-Allow-Origin header"*
- How to fix *"Access-Control-Allow-Origin header must not be the wildcard"*

**Last modified:** 12 de dez. de 2022, [by MDN contributors](#)