Cross-Origin Resource Sharing

W3C Working Draft 3 April 2012

This Version:
http://www.w3.org/TR/2012/WD-cors-20120403/
Latest Version:
http://www.w3.org/TR/cors/
Latest Editor Draft:
http://dvcs.w3.org/hg/cors/raw-file/tip/Overview.html
Previous Versions:
http://www.w3.org/TR/2010/WD-cors-20100727/
http://www.w3.org/TR/2009/WD-cors-20090317/
http://www.w3.org/TR/2008/WD-access-control-20080912/
http://www.w3.org/TR/2008/WD-access-control-20080214/
http://www.w3.org/TR/2007/WD-access-control-20071126/
http://www.w3.org/TR/2007/WD-access-control-20071001/
http://www.w3.org/TR/2007/WD-access-control-20070618/
http://www.w3.org/TR/2007/WD-access-control-20070215/
http://www.w3.org/TR/2006/WD-access-control-20060517/
http://www.w3.org/TR/2005/NOTE-access-control-20050613/
Editor:
Anne van Kesteren (Opera Software ASA) <annevk@opera.com>

Abstract

This document defines a mechanism to enable client-side cross-origin requests. Specifications that enable an API to make cross-origin requests to resources can use the algorithms defined by this

specification. If such an API is used on `http://example.org` resources, a resource on `http://hello-world.example` can opt in using the mechanism described by this specification (e.g., specifying `Access-Control-Allow-Origin: http://example.org` as response header), which would allow that resource to be fetched cross-origin from `http://example.org`.

Status of this Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is the 3 April 2012 W3C *Last Call* Working Draft of Cross-Origin Resource Sharing. Please send comments to public-webapps@w3.org (archived) *by 1 May 2012* with `[CORS]` at the start of the subject line.

This document was produced jointly by the Web Applications (WebApps) and Web Application Security (WebAppSec) Working Groups, and published by the WebAppSec Working Group.

This document was produced by two groups operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Publication as a Working Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Table of Contents

1 Introduction

This section is non-normative.

User agents commonly apply same-origin restrictions to network requests. These restrictions prevent a client-side Web application running from one origin from obtaining data retrieved from another origin, and also limit unsafe HTTP requests that can be automatically launched toward destinations that differ

from the running application's origin.

In user agents that follow this pattern, network requests typically use ambient authentication and session management information, including HTTP authentication and cookie information.

This specification extends this model in several ways:

- A response can include an `Access-Control-Allow-Origin` header, with the origin of where the request originated from as the value, to allow access to the resource's contents.

  The user agent validates that the value and origin of where the request originated match.

- User agents can discover via a preflight request whether a cross-origin resource is prepared to accept requests, using a non-simple method, from a given origin.

  This is again validated by the user agent.

- Server-side applications are enabled to discover that an HTTP request was deemed a cross-origin request by the user agent, through the `Origin` header.

  This extension enables server-side applications to enforce limitations (e.g. returning nothing) on the cross-origin requests that they are willing to service.

This specification is a building block for other specifications, so-called CORS API specifications, which define how this specification is used. Examples are Server-Sent Events and XMLHttpRequest. [EVENTSOURCE] [XHR]

The design of this specification introduces is based on requirements and use cases, both included as appendix. A FAQ describing the design decisions is also available.

> If a resource author has a simple text resource residing at `http://example.com/hello` which contains the string "Hello World!" and would like `http://hello-world.example` to be able to access it, the response combined with a header introduced by this specification could look as follows:
>
> ```
> Access-Control-Allow-Origin: http://hello-world.example
>
> Hello World!
> ```
>
> Using `XMLHttpRequest` a client-side Web application on `http://hello-world.example` can access this resource as follows:
>
> ```
> var client = new XMLHttpRequest();
> ```

```
client.open("GET", "http://example.com/hello")
client.onreadystatechange = function() { /* do something */ }
client.send()
```

It gets slightly more complicated if the resource author wants to be able to handle cross-origin requests using methods other than simple methods. In that case the author needs to reply to a preflight request that uses the `OPTIONS` method and then needs to handle the actual request that uses the desired method (`DELETE` in this example) and give an appropriate response. The response to the preflight request could have the following headers specified:

```
Access-Control-Allow-Origin: http://hello-world.example
Access-Control-Max-Age: 3628800
Access-Control-Allow-Methods: PUT, DELETE
```

The `Access-Control-Max-Age` header indicates how long the response can be cached, so that for subsequent requests, within the specified time, no preflight request has to be made. The `Access-Control-Allow-Methods` header indicates the methods that can be used in the actual request. The response to the actual request can simply contain this header:

```
Access-Control-Allow-Origin: http://hello-world.example
```

The complexity of invoking the additional preflight request is the task of the user agent. Using `XMLHttpRequest` again and assuming the application were hosted at `http://calendar.example/app` the author could use the following ECMAScript snippet:

```
function deleteItem(itemId, updateUI) {
  var client = new XMLHttpRequest()
  client.open("DELETE", "http://calendar.example/app")
  client.onload = updateUI
  client.onerror = updateUI
  client.onabort = updateUI
  client.send("id=" + itemId)
}
```

2 Conformance

This specification is written for resource authors and user agents. It includes advice for specifications that define APIs that use the cross-origin request algorithm defined in this specification — CORS API specifications — and the general security considerations section includes some advice for client-side Web application authors.

As well as sections and appendices marked as non-normative, all diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

In this specification, The words must and may are to be interpreted as described in RFC 2119. [RFC2119]

Requirements phrased in the imperative as part of algorithms (e.g. "terminate the algorithm") are to be interpreted with the meaning of the key word (e.g. must) used in introducing the algorithm.

A conformant resource is one that implements all the requirements listed in this specification that are applicable to resources.

A conformant user agent is one that implements all the requirements listed in this specification that are applicable to user agents.

User agents and resource authors may employ any algorithm to implement this specification, so long as the end result is indistinguishable from the result that would be obtained by the specification's algorithms.

3 Terminology

Some terminology in this specification is from *The Web Origin Concept HTML*, *HTTP* and *URI*. [ORIGIN] [HTML] [HTTP] [URI]

Terminology is generally defined throughout the specification. However, the few definitions that did not really fit anywhere else are defined here instead.

Comparing two strings in a **case-sensitive** manner means comparing them exactly, codepoint for codepoint.

Comparing two strings in an **ASCII case-insensitive** manner means comparing them exactly, codepoint for codepoint, except that the characters in the range U+0041 LATIN CAPITAL LETTER A to U+005A LATIN CAPITAL LETTER Z and the corresponding characters in the range U+0061 LATIN SMALL LETTER A to U+007A LATIN SMALL LETTER Z are considered to also match.

Converting a string to ASCII lowercase means replacing all characters in the range U+0041 LATIN CAPITAL LETTER A to U+005A LATIN CAPITAL LETTER Z with the corresponding characters in the range U+0061 LATIN SMALL LETTER A to U+007A LATIN SMALL LETTER Z).

The term **user credentials** for the purposes of this specification means cookies, HTTP authentication,

and client-side SSL certificates. Specifically it does not refer to proxy authentication or the `Origin` header. [COOKIES]

The term **cross-origin** is used to mean non same origin.

A *method* is said to be a **simple method** if it is a case-sensitive match for one of the following:

- `GET`
- `HEAD`
- `POST`

A *header* is said to be a **simple header** if the header field name is an ASCII case-insensitive match for `Accept`, `Accept-Language`, or `Content-Language`, or if it is an ASCII case-insensitive match for `Content-Type` and the header field value media type (excluding parameters) is an ASCII case-insensitive match for `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`.

A *header* is said to be a **simple response header** if the header field name is an ASCII case-insensitive match for one of the following:

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

When **parsing a header** the header must be parsed per the corresponding ABNF production in the syntax section. If the header does not match the production it is said that **header parsing failed**.

4 Security Considerations

This section is non-normative.

Security requirements and considerations are listed throughout this specification. This section lists advice that did not fit anywhere else.

A simple cross-origin request has been defined as congruent with those which may be generated by currently deployed user agents that do not conform to this specification. Simple cross-origin requests generated outside this specification (such as cross-origin form submissions using `GET` or `POST` or

cross-origin `GET` requests resulting from `script` elements) typically include user credentials, so resources conforming to this specification must always be prepared to expect simple cross-origin requests with credentials.

Because of this, resources for which simple requests have significance other than retrieval must protect themselves from Cross-Site Request Forgery (CSRF) by requiring the inclusion of an unguessable token in the explicitly provided content of the request. [CSRF]

This specification defines how to authorize an instance of an application from a foreign origin, executing in the user agent, to access the representation of the resource in an HTTP response. Certain types of resources should not attempt to specify particular authorized origins, but instead either deny or allow all origins.

1. A resource that is not useful to applications from other origins, such as a login page, ought not to return an `Access-Control-Allow-Origin` header. The resource still must protect itself against CSRF attacks, such as by requiring the inclusion of an unguessable token in the explicitly provided content of the request. The security properties of such resources are unaffected by user-agents conformant to this specification.

2. A resource that is publicly accessible, with no access control checks, can always safely return an `Access-Control-Allow-Origin` header whose value is "`*`".

3. A `GET` response whose entity body happens to parse as ECMAScript can return an `Access-Control-Allow-Origin` header whose value is "`*`" provided there are no sensitive comments as it can be accessed cross-origin using an HTML `script` element. If needed, such resources can implement access control and CSRF protections as described above.

Care must always be taken by applications when making cross-origin requests with user credentials, and servers processing such requests must take care in the use of credentials, including the `Origin` header.

1. When requests have significance other than retrieval, and when relying on the `Origin` header as a credential, servers must be careful to distinguish between authorizing a request and authorizing access to the representation of that resource in the response.

    1. Authorization for a request should be performed using only the intersection of the authority of the user and the requesting origin(s). In the case of redirects, more than one value for `Origin` may be present and all must be authorized.

    2. Servers using the `Origin` header to authorize requests are encouraged to also verify that the `Host` header matches its expected value to prevent forwarding attacks. Consider two sites, `corp.example` and `corp.invalid`. A web application at `corp.example`

makes a cross-origin request to `corp.invalid`, and the user agent sends the `Origin` header `corp.example`. If `corp.invalid` or the network is malicious, it may cause the request to be delivered to `corp.example`, with the result that `corp.example` would receive a requst that appears to originate from itself. Verifying the `Host` header would reveal that the user agent intended the request for `corp.invalid` and it can be discarded. Even better would be to exclusively use secure connections for cross-origin requests to enable user agents to detect such misdirections.

3. It is often appropriate for servers to require an authorization ceremony asking a user to consent that cross-origin requests with credentials be honored from a given origin. In such cases, passing security tokens explicitly as part of the cross-origin request can remove any ambiguity as to the scope of authorization. OAuth is an example of this pattern. [OAUTH]

2. Use of user credentials in a cross-origin request is appropriate when:

    1. A cross-origin request with credentials as defined in this specification is used to substitute for alternate methods of authenticated resource sharing, such as server-to-server back channels, JSONP, or cross-document messaging. [JSONP] [HTML]

        This substitution can expose additional attack surface in some cases, as a cross-site scripting vulnerability in the requesting origin can allow elevation of privileges against the requested resource when compared to a server-to-server back channel.

        As a substitute for JSONP-style cross-origin credentialed requests, use of this specification significantly improves the security posture of the requesting application, as it provides cross-origin data access whereas JSONP operates via cross-origin code-injection. The requesting application has to validate that data received from origins that are not completely trusted conforms to expected formats and authorized values.

        As a substitute for cross-origin communication techniques relying on loading a resource, with credentials, into an HTML `iframe` element, and subsequently employing cross-document messaging or other cross-origin side channels, this specification provides a roughly equivalent security posture. Again, data received from origins that are not completely trusted has to be validated to conform to expected formats and authorized values.

    2. For resources that are safe and idempotent per HTTP, and where the credentials are used only to provide user-specific customization for otherwise publicly accessible information. In this case, restricting access to certain origins may protect user privacy by preventing customizations from being used to identify a user, except at authorized origins.

3. When this specification is used for requests which have significance other than retrieval and

which involve coordination between or data originating from more than two origins, (e.g. between resources enabling editing, printing and storage, each at distinct origins) requests ought to set the omit credentials flag and servers ought to perform authorization using security tokens explicitly provided in the content of the request, especially if the origins are not all mutually and completely trusted.

In such multi-origin scenarios, a malicious resource at one of the origins may be able to enlist the user-agent as a confused deputy and elevate its privileges by abusing the user's ambient authority. Avoiding such attacks requires that the coordinating applications have explicit knowledge of the scope of privilege for each origin and that all parameters and instructions received are carefully validated at each step in the coordination to ensure that effects implied do not exceed the authority of the originating principal. [CONFUSED]

Given the difficulty of avoiding such vulnerabilities in multi-origin interactions it is recommended that, instead of using implicit credentials, security tokens which specify the particular capabilities and resources authorized be passed explicitly as part of each request. OAuth again provides an example of such a pattern.

Authors of client-side Web applications are strongly encouraged to validate content retrieved from a cross-origin resource as it might be harmful.

Authors of client-side Web applications using a URL of the type `people.example.org/~author-name/` are to be aware that only cross-origin security is provided and that therefore using a distinct origin rather than distinct path is vital for secure client-side Web applications.

5 Syntax

This section defines the syntax of the new headers this specification introduces. It also provides a short description of the function of each header.

The resource processing model section details how resources are to use these headers in a response. Likewise, the user agent processing model section details how user agents are to use these headers.

The ABNF syntax used in this section is from HTTP/1.1. [HTTP]

Note: HTTP/1.1 is used as ABNF basis to ensure that the new headers have equivalent parsing rules to those introduced in that specification.

Issue: HTTP/1.1 currently does not make leading OWS implied in header value definitions so

please assume it is for now.

5.1 `Access-Control-Allow-Origin` Response Header

The **`Access-Control-Allow-Origin`** header indicates whether a resource can be shared based by returning the value of the `Origin` request header in the response. ABNF:

```
Access-Control-Allow-Origin = "Access-Control-Allow-Origin" ":" o
```

5.2 `Access-Control-Allow-Credentials` Response Header

The **`Access-Control-Allow-Credentials`** header indicates whether the response to request can be exposed when the omit credentials flag is unset. When part of the response to a preflight request it indicates that the actual request can include user credentials. ABNF:

```
Access-Control-Allow-Credentials: "Access-Control-Allow-Credentia
                                  true: %x74.72.75.65 ; "true", case-s
```

5.3 `Access-Control-Expose-Headers` Response Header

The **`Access-Control-Expose-Headers`** header indicates which headers are safe to expose to the API of a CORS API specification. ABNF:

```
Access-Control-Expose-Headers = "Access-Control-Expose-Headers"
```

5.4 `Access-Control-Max-Age` Response Header

The **`Access-Control-Max-Age`** header indicates how long the results of a preflight request can be cached in a preflight result cache. ABNF:

```
Access-Control-Max-Age = "Access-Control-Max-Age" ":" delta-seco
```

5.5 `Access-Control-Allow-Methods` Response Header

The **`Access-Control-Allow-Methods`** header indicates, as part of the response to a preflight request, which methods can be used during the actual request. ABNF:

```
Access-Control-Allow-Methods: "Access-Control-Allow-Methods" ":"
```

5.6 `Access-Control-Allow-Headers` Response Header

The **`Access-Control-Allow-Headers`** header indicates, as part of the response to a preflight request, which header field names can be used during the actual request. ABNF:

```
Access-Control-Allow-Headers: "Access-Control-Allow-Headers" ":"
```

5.7 `Origin` Request Header

The **`Origin`** header indicates where the cross-origin request or preflight request originates from. [ORIGIN]

5.8 `Access-Control-Request-Method` Request Header

The **`Access-Control-Request-Method`** header indicates which method will be used in the actual request as part of the preflight request. ABNF:

```
Access-Control-Request-Method: "Access-Control-Request-Method" "
```

5.9 `Access-Control-Request-Headers` Request Header

The **`Access-Control-Request-Headers`** header indicates which headers will be used in the actual request as part of the preflight request. ABNF:

```
Access-Control-Request-Headers: "Access-Control-Request-Headers"
```

6 Resource Processing Model

This section describes the processing models that resources have to implement. Each type of request a resource might have to deal with is described in its own subsection.

The resource sharing policy described by this specification is bound to a particular resource. For the purposes of this section each resource is bound to the following:

- A **list of origins** consisting of zero or more origins that are allowed access to the resource.

  Note: This can include the origin of the resource itself though be aware that requests to cross-origin resources can be redirected back to the resource.

- A **list of methods** consisting of zero or more methods that are supported by the resource.
- A **list of headers** consisting of zero or more header field names that are supported by the resource.
- A **list of exposed headers** consisting of zero or more header field names of headers other than the simple response headers that the resource might use and can be exposed.
- A **supports credentials** flag that indicates whether the resource supports user credentials in the request. It is true when the resource does and false otherwise.

6.1 Simple Cross-Origin Request, Actual Request, and Redirects

In response to a simple cross-origin request or actual request the resource indicates whether or not to share the response.

If the resource has been relocated, it indicates whether to share its new URL.

Resources must use the following set of steps to determine which additional headers to use in the response:

1. If the `Origin` header is not present terminate this set of steps. The request is outside the scope of this specification.
2. If the value of the `Origin` header is not a case-sensitive match for any of the values in list of origins, do not set any additional headers and terminate this set of steps.

   Note: Always matching is acceptable since the list of origins can be unbounded.
3. If the resource supports credentials add a single `Access-Control-Allow-Origin` header, with the value of the `Origin` header as value, and add a single `Access-Control-Allow-Credentials` header with the case-sensitive string "`true`" as value.

   Otherwise, add a single `Access-Control-Allow-Origin` header, with either the value of the `Origin` header or the string "`*`" as value.

   Note: The string "*" cannot be used for a resource that supports credentials.
4. If the list of exposed headers is not empty add one or more `Access-Control-Expose-Headers` headers, with as values the header field names given in the list of exposed headers.

   Note: By not adding the appropriate headers resource can also clear the preflight result

cache of all entries where origin is a case-sensitive match for the value of the `Origin` header and url is a case-sensitive match for the URL of the resource.

6.2 Preflight Request

In response to a preflight request the resource indicates which methods and headers (other than simple methods and simple headers) it is willing to handle and whether it supports credentials.

Resources must use the following set of steps to determine which additional headers to use in the response:

1. If the `Origin` header is not present terminate this set of steps. The request is outside the scope of this specification.

2. If the value of the `Origin` header is not a case-sensitive match for any of the values in list of origins do not set any additional headers and terminate this set of steps.

   Note: Always matching is acceptable since the list of origins can be unbounded.

   Note: The `Origin` header can only contain a single origin as the user agent will not follow redirects.

3. Let *method* be the value as result of parsing the `Access-Control-Request-Method` header.

   If there is no `Access-Control-Request-Method` header or if parsing failed, do not set any additional headers and terminate this set of steps. The request is outside the scope of this specification.

4. Let *header field-names* be the values as result of parsing the `Access-Control-Request-Headers` headers.

   If there are no `Access-Control-Request-Headers` headers let *header field-names* be the empty list.

   If parsing failed do not set any additional headers and terminate this set of steps. The request is outside the scope of this specification.

5. If *method* is not a case-sensitive match for any of the values in list of methods do not set any additional headers and terminate this set of steps.

   Note: Always matching is acceptable since the list of methods can be unbounded.

6. If any of the *header field-names* is not a ASCII case-insensitive match for any of the values in list of headers do not set any additional headers and terminate this set of steps.

    Note: Always matching is acceptable since the list of headers can be unbounded.

7. If the resource supports credentials add a single `Access-Control-Allow-Origin` header, with the value of the `Origin` header as value, and add a single `Access-Control-Allow-Credentials` header with the case-sensitive string "`true`" as value.

    Otherwise, add a single `Access-Control-Allow-Origin` header, with either the value of the `Origin` header or the string "`*`" as value.

    Note: The string "*" cannot be used for a resource that supports credentials.

8. Optionally add a single `Access-Control-Max-Age` header with as value the amount of seconds the user agent is allowed to cache the result of the request.

9. If *method* is a simple method this step may be skipped.

    Add one or more `Access-Control-Allow-Methods` headers consisting of (a subset of) the list of methods.

    Note: If a method is a simple method it does not need to be listed, but this is not prohibited.

    Note: Since the list of methods can be unbounded simply returning method can be enough.

10. If each of the *header field-names* is a simple header and none is `Content-Type`, than this step may be skipped.

    Add one or more `Access-Control-Allow-Headers` headers consisting of (a subset of) the list of headers.

    Note: If a header field name is a simple header and is not `Content-Type`, it is not required to be listed. `Content-Type` is to be listed as only a subset of its values makes it qualify as simple header.

    Note: Since the list of headers can be unbounded simply returning headers can be enough.

6.3 Security

This section is non-normative.

Resource authors are strongly encouraged to ensure that requests using safe methods, e.g. `GET` or `OPTIONS`, have no side effects so potential attackers cannot modify the user's data easily. If resources are set up like this attackers would effectively have to be on the list of origins to do harm.

In addition to checking the `Origin` header, resource authors are strongly encouraged to also check the `Host` header. That is, make sure that the host name provided by that header matches the host name of the server on which the resource resides. This will provide protection against DNS rebinding attacks.

To provide integrity protection of resource sharing policy statements usage of SSL/TLS is encouraged.

7 User Agent Processing Model

This section describes the processing models that user agents have to implement.

The processing models in this sections need to be referenced by a CORS API specification that defines when the algorithm is invoked and how the return values are to be handled. The processing models are not suitable for standalone use.

7.1 Cross-Origin Request

The **cross-origin request** algorithm takes the following parameters:

request URL

The URL to be fetched.

Note: The request URL is modified in face of redirects.

request method

The method for the request. `GET`, unless explicitly set.

author request headers

A list of headers set by authors for the request. Empty, unless explicitly set.

request entity body

The entity body for the request. Missing, unless explicitly set.

source origin

The origin of the request.

> ***Note: Due to the specifics of some APIs this cannot be defined in a generic way and therefore it has to be provided as argument.***

manual redirect flag

Set when redirects are *not* to be automatically followed.

omit credentials flag

Set when user credentials are to be excluded in the request and when cookies are to be ignored in its response.

force preflight flag

Set when a preflight request is required.

The cross-origin request algorithm can be used by CORS API specifications who wish to allow cross-origin requests for the network APIs they define.

> ***Note: CORS API specifications are free to limit the abilities of a cross-origin request. E.g., the omit credentials flag could always be set.***

When the cross-origin request algorithm is invoked, these steps must be followed:

1. If for some reason the user agent does not want to make the request terminate this algorithm and set the cross-origin request status to *network error*.

   > ***Note: The request URL could have been blacklisted by the user in some fashion.***

2. If the following conditions are true, follow the simple cross-origin request algorithm:

   - The request method is a simple method and the force preflight flag is unset.
   - Each of the author request headers is a simple header or author request headers is empty.

3. Otherwise, follow the cross-origin request with preflight algorithm.

Note: Cross-origin requests using a method that is simple with author request headers that are not simple will have a preflight request to ensure that the resource can handle those headers. (Similarly to requests using a method that is not a simple method.)

7.1.1 Handling a Response to a Cross-Origin Request

User agents must filter out all response headers other than those that are a simple response header or of which the field name is an ASCII case-insensitive match for one of the values of the `Access-Control-Expose-Headers` headers (if any), before exposing response headers to APIs defined in CORS API specifications.

Note: The `getResponseHeader()` method of `XMLHttpRequest` will therefore not expose any header not indicated above.

7.1.2 Cross-Origin Request Status

Each cross-origin request has an associated **cross-origin request status** that CORS API specifications that enable an API to make cross-origin requests can hook into. It can take at most two distinct values over the course of a cross-origin request. The values are:

preflight complete
: The user agent is about to make the actual request.

success
: The resource can be shared.

abort error
: The user aborted the request.

network error
: The resource cannot be shared. Also used when a DNS error, TLS negotiation failure, or other type of network error occurs. *This does not include HTTP responses that indicate some type of error, such as HTTP status code 410.*

7.1.3 Source Origin

The source origin is the initial origin that user agents must use for the `Origin` header. It can be modified during the redirect steps.

7.1.4 Simple Cross-Origin Request

The steps below describe what user agents must do for a **simple cross-origin request**:

1. Apply the make a request steps and observe the *request rules* below while making the request.

   ↪ **If the manual redirect flag is unset and the response has an HTTP status code of 301, 302, 303, or 307**

   Apply the redirect steps.

   ↪ **If the end user cancels the request**

   Apply the abort steps.

   ↪ **If there is a network error**

   In case of DNS errors, TLS negotiation failure, or other type of network errors, apply the network error steps. Do not request any kind of end user interaction.

   Note: This does not include HTTP responses that indicate some type of error, such as HTTP status code 410.

   ↪ **Otherwise**

   Perform a resource sharing check. If it returns fail, apply the network error steps. Otherwise, if it returns pass, terminate this algorithm and set the cross-origin request status to *success*. Do not actually terminate the request.

7.1.5 Cross-Origin Request with Preflight

To protect resources against cross-origin requests that could not originate from certain user agents before this specification existed a preflight request is made to ensure that the resource is aware of this specification. The result of this request is stored in a preflight result cache.

The steps below describe what user agents must do for a **cross-origin request with preflight**. This is a request to a non same-origin URL that first needs to be authorized using either a preflight result cache entry or a preflight request.

1. Go to the next step if the following conditions are true:

   - For request method there either is a method cache match or it is a simple method and the force preflight flag is unset.
   - For every header of author request headers there either is a header cache match for the field name or it is a simple header.

Otherwise, make a **preflight request**. Fetch the request URL from *origin* source origin with the *manual redirect flag* and the *block cookies flag* set, using the method `OPTIONS`, and with the following additional constraints:

- Include an `Access-Control-Request-Method` header with as header field value the request method (even when that is a simple method).
- If author request headers is not empty include an `Access-Control-Request-Headers` header with as header field value a comma-separated list of the header field names from author request headers in lexicographical order, each converted to ASCII lowercase (even when one or more are a simple header).
- Exclude the author request headers.
- Exclude user credentials.
- Exclude the `Referer` header if source origin is a globally unique identifier.
- Exclude the request entity body.

The following *request rules* are to be observed while making the preflight request:

↪ **If the end user cancels the request**

Apply the abort steps.

↪ **If the response has an HTTP status code that is not 200**

Apply the network error steps.

Note: The cache and network error steps are not used here as this is about an actual network error.

↪ **If there is a network error**

In case of DNS errors, TLS negotiation failure, or other type of network errors, apply the network error steps. Do not request any kind of end user interaction.

Note: This does not include HTTP responses that indicate some type of error, such as HTTP status code 410.

Note: The cache and network error steps are not used here as this is about an actual network error.

↪ **Otherwise (the HTTP status code is 200)**

1. If the resource sharing check returns fail, apply the cache and network error steps.

2. Let *methods* be the empty list.

3. If there are one or more `Access-Control-Allow-Methods` headers let *methods* be the values as result of parsing the headers.

   If parsing failed apply the cache and network error steps.

4. If *methods* is still the empty list and the force preflight flag is set, append the request method to *methods*.

   Note: This ensures that preflight requests that happened solely because of the force preflight flag are cached too.

5. Let *headers* be the empty list.

6. If there are one or more `Access-Control-Allow-Headers` headers let *headers* be the values as result of parsing the headers.

   If parsing failed apply the cache and network error steps.

7. If request method is not a case-sensitive match for any method in *methods* and is not a simple method, apply the cache and network error steps.

8. If the field name of each header in author request headers is not an ASCII case-insensitive match for one of the header field names in *headers* and the header is not a simple header, apply the cache and network error steps.

9. If for some reason the user agent is unable to provide a preflight result cache (e.g. because of limited disk space) go to the next step in the overall set of steps (i.e. the actual request).

10. If there is a single `Access-Control-Max-Age` header, parse it and let *max-age* be the resulting value.

    If there is no such header, there is more than one such header, or parsing failed, let *max-age* be a value at the discretion of the user agent (zero is allowed).

    If the user agent imposes a limit on the max-age field value and *max-age* is greater

than that limit let *max-age* be the limit.

11. For each method in *methods* for which there is a method cache match set the max-age field value of the matching entry to *max-age*.

    For each method in *methods* for which there is *no* method cache match create a new entry in the preflight result cache with the various fields set as follows:

    origin
    The source origin.
    url
    The request URL.
    max-age
    The *max-age*.
    credentials
    False if the omit credentials flag is set, or true otherwise.
    method
    The given method.
    header
    Empty.

12. For each header in *headers* for which there is a header cache match set the max-age field value of the matching entry to *max-age*.

    For each header in *headers* for which there is *no* header cache match create a new entry in the preflight result cache with the various fields set as follows:

    origin
    The source origin.
    url
    The request URL.
    max-age
    The *max-age*.
    credentials
    False if the omit credentials flag is set, or true otherwise.
    method
    Empty.
    header
    The given header.

2. Set the cross-origin request status to *preflight complete*.

3. This is the **actual request**. Apply the make a request steps and observe the *request rules* below while making the request.

   ↪ **If the response has an HTTP status code of 301, 302, 303, or 307**

   Apply the cache and network error steps.

   ↪ **If the end user cancels the request**

   Apply the abort steps.

   ↪ **If there is a network error**

   In case of DNS errors, TLS negotiation failure, or other type of network errors, apply the network error steps. Do not request any kind of end user interaction.

   Note: This does not include HTTP responses that indicate some type of error, such as HTTP status code 410.

   ↪ **Otherwise**

   Perform a resource sharing check. If it returns fail, apply the cache and network error steps. Otherwise, if it returns pass, terminate this algorithm and set the cross-origin request status to *success*. Do not actually terminate the request.

Consider the following scenario:

1. The user agent gets the request from an API, such as `XMLHttpRequest`, to perform a cross-origin request using the custom `XMODIFY` method from source origin `http://example.org` to `http://blog.example/entries/hello-world`.
2. The user agent performs a preflight request using the `OPTIONS` method to `http://blog.example/entries/hello-world` and includes the `Origin` and `Access-Control-Request-Method` headers with the appropriate values.
3. The response to that request includes the following headers:

   ```
   Access-Control-Allow-Origin: http://example.org
   Access-Control-Max-Age: 2520
   Access-Control-Allow-Methods: PUT, DELETE, XMODIFY
   ```

4. The user agent then performs the desired request using the `XMODIFY` method to `http://blog.example/entries/hello-world` as this was allowed by the

> resource. In addition, for the coming forty-two minutes, no preflight request will be needed.

7.1.6 Preflight Result Cache

As mentioned, a cross-origin request with preflight uses a **preflight result cache**. This cache consists of a set of entries. Each entry consists of the following fields:

origin
: Holds the source origin.

url
: Holds the request URL.

max-age
: Holds the `Access-Control-Max-Age` header value.

credentials
: False if the omit credentials flag is set, or true otherwise.

method
: Empty if header is not empty; otherwise one of the values from the `Access-Control-Allow-Methods` headers.

header
: Empty if method is not empty; otherwise one of the values from the `Access-Control-Allow-Headers` headers.

Note: To be clear, the method and header fields are mutually exclusive. When one of them is empty the other is non-empty.

Note: The primary key of an entry consists of all fields excluding the max-age field.

Entries must be removed when the time specified in the max-age field has passed since storing the entry. Entries can also be added and removed per the algorithms below. They are added and removed in such a way that there can never be duplicate items in the cache.

User agents may clear cache entries before the time specified in the max-age field has passed.

Note: Although this effectively makes the preflight result cache optional, user agents are strongly encouraged to support it.

7.1.7 Generic Cross-Origin Request Algorithms

The variables used in the generic set of steps are part of the algorithms that invoke these set of steps.

Whenever the **make a request steps** are applied, fetch the request URL from *origin* source origin with the *manual redirect flag* set, and the *block cookies flag* set if the omit credentials flag is set. Use method request method, entity body request entity body, including the author request headers, and include user credentials if the omit credentials flag is unset. Exclude the `Referer` header if source origin is a globally unique identifier.

Whenever the **redirect steps** are applied, follow this set of steps:

1. Let *original URL* be the request URL.
2. Let request URL be the URL conveyed by the `Location` header in the redirect response.
3. If the request URL <scheme> is not supported, infinite loop precautions are violated, or the user agent does not wish to make the new request for some other reason, apply the network error steps.
4. If the request URL contains the `userinfo` production apply the network error steps.
5. If the resource sharing check for the current resource returns fail, apply the network error steps.
6. If the request URL origin is not same origin with the *original URL* origin, set source origin to a globally unique identifier (becomes "`null`" when transmitted).
7. Transparently follow the redirect while observing the set of *request rules*.

Whenever the **abort steps** are applied, terminate the algorithm that invoked this set of steps and set the cross-origin request status to *abort error*.

Whenever the **network error steps** are applied, terminate the algorithm that invoked this set of steps and set the cross-origin request status to *network error*.

> ***Note: This has no effect on setting of user credentials. I.e. if the block cookies flag is unset, cookies will be set by the response.***

Whenever the **cache and network error steps** are applied, follow these steps:

1. Remove the entries in the preflight result cache where origin field value is a case-sensitive match for source origin and url field value is a case-sensitive match for request URL.
2. Apply the network error steps acting as if the algorithm that invoked the cache and network error

steps invoked the network error steps instead.

There is a **cache match** when there is a cache entry in the preflight result cache for which the following is true:

- The origin field value is a case-sensitive match for source origin.
- The url field value is a case-sensitive match for request URL.
- The credentials field value is true and the omit credentials flag is unset, or it is false and the omit credentials flag is set.

There is a **method cache match** when there is a cache entry for which there is a cache match and the method field value is a case-sensitive match for the given method.

There is a **header cache match** when there is a cache entry for which there is a cache match and the header field value is an ASCII case-insensitive match for the given header field name.

7.2 Resource Sharing Check

The **resource sharing check** algorithm for a given resource is as follows:

1. If the response includes zero or more than one `Access-Control-Allow-Origin` header values, return fail and terminate this algorithm.
2. If the `Access-Control-Allow-Origin` header value is the "`*`" character and the omit credentials flag is set, return pass and terminate this algorithm.
3. If the value of `Access-Control-Allow-Origin` is not a case-sensitive match for the value of the `Origin` header as defined by its specification, return fail and terminate this algorithm.
4. If the omit credentials flag is unset and the response includes zero or more than one `Access-Control-Allow-Credentials` header values, return fail and terminate this algorithm.
5. If the omit credentials flag is unset and the `Access-Control-Allow-Credentials` header value is not a case-sensitive match for "`true`", return fail and terminate this algorithm.
6. Return pass.

Note: The above algorithm also functions when the ASCII serialization of an origin is the

string `"null"`.

7.3 Security

This section is non-normative.

At various places user agents are allowed to take additional precautions. E.g. user agents are allowed to not store cache items, remove cache items before they reached their max-age, and not connect to certain URLs.

User agents are encouraged to impose a limit on max-age so items cannot stay in the preflight result cache for unreasonable amounts of time.

As indicated as the first step of the cross-origin request algorithm and in the redirect steps algorithm user agents are allowed to terminate the algorithm and not make a request. This could be done because e.g.:

- The server on which the resource resides is blacklisted.
- The server on which the resource resides is known to be part of an intranet.
- The URL <scheme> is not supported.
- `https` to `http` is not allowed.
- `https` to `https` is not allowed because e.g. the certificates differ.

User agents are encouraged to apply security decisions on a generic level and not just to the resource sharing policy. E.g. if a user agent disallows requests from the `https` to the `http` scheme for a cross-origin request it is encouraged to do the same for the HTML `img` element.

8 CORS API Specification Advice

This section is non-normative.

This specification defines a resource sharing policy that cannot be implemented without an API that utilizes it. The specification that defines the API that uses the policy is a CORS API specification.

In case a CORS API specification defines multiple APIs that utilize the policy the advice is to be considered separately for each API.

8.1 Constructing a Cross-Origin Request

For all cross-origin requests that APIs can make for which the resource sharing policy in this

specification is supposed to apply, the CORS API specification needs to reference the cross-origin request algorithm and set the following input variables appropriately: request URL, request method, author request headers, request entity body, source origin, manual redirect flag, omit credentials flag, and the force preflight flag.

CORS API specifications are allowed to let these input variables be controlled by the API, but can also set fixed values.

> A CORS API specification for an API that only allows requests using the `GET` method might set request method to `GET`, request entity body to empty, and source origin to some appropriate value and let the other variables be controlled by the API.

8.2 Dealing with Same Origin to Cross-Origin Redirects

Since browsers are based on a same origin security model and the policy outlined in this specification is intended for APIs used in browsers, it is expected that APIs that will utilize this policy will have to handle a same origin request that results in a redirect that is cross-origin in a special way.

For APIs that transparently handle redirects CORS API specifications are encouraged to handle this scenario transparently as well by "catching" the redirect and invoking the cross-origin request algorithm on the (cross-origin) redirect URL.

Note: The XMLHttpRequest specification does this. [XHR]

8.3 Dealing with the Cross-Origin Request Status

While a cross-origin request is progressing its associated cross-origin request status is updated. Depending on the value of the cross-origin request status the API is to react in a different way:

preflight complete

Features that can only be safely exposed after a preflight request can now be enabled.

Note: E.g. upload progress events for `XMLHttpRequest`.

success

The contents of the response can be shared with the API, including headers that have not been filtered out.

Note: The request itself can still be progressing. I.e. the cross-origin request status value

does not indicate that the request has completed.

abort error

Handle analogous to requests where the user aborted the request. This can be handled equivalently to how *network error* is handled. Ensure not to reveal any further information about the request.

network error

Handle analogous to requests where some kind of error occured. Ensure not the reveal any further information about the request.

8.4 Security

Similarly to same origin requests, CORS API specifications are encouraged to properly limit headers, methods, and user credentials the author can set and get for requests that are cross-origin.

Note: Reviewing the XMLHttpRequest specification provides a good start for the kind of limitations that are to be imposed. [XHR]

CORS API specifications also need to ensure not to reveal anything until the cross-origin request status is set to *preflight complete* or *success* to prevent e.g. port scanning.

Note: In XMLHttpRequest progress events are dispatched only after the cross-origin request status is set to success. Upload progress events are only dispatched once the cross-origin request status is preflight complete.

Requirements

This appendix is non-normative.

This appendix outlines the various requirements that influenced the design of the Cross-Origin Resource Sharing specification.

1. Must not introduce attack vectors to servers that are only protected only by a firewall.

2. The solution should not introduce additional attack vectors against services that are protected only by way of firewalls. This requirement addresses "intranet" style services authorize any requests that can be sent to the service.

Note that this requirement does not preclude `HEAD`, `OPTIONS`, or `GET` requests (even with ambient authentication and session information).

3. It should not be possible to perform cross-origin non-safe operations, i.e., HTTP operations except for `GET`, `HEAD`, and `OPTIONS`, without an authorization check being performed.

4. Should try to prevent dictionary-based, distributed, brute-force attacks that try to get login accounts to 3rd party servers, to the extent possible.

5. Should properly enforce security policy in the face of commonly deployed proxy servers sitting between the user agent and any of servers with whom the user agent is communicating.

6. Should not allow loading and exposing of resources from 3rd party servers without explicit consent of these servers as such resources can contain sensitive information.

7. Must not require content authors or site maintainers to implement new or additional security protections to preserve their existing level of security protection.

8. Must be deployable to IIS and Apache without requiring actions by the server administrator in a configuration where the user can upload static files, run serverside scripts (such as PHP, ASP, and CGI), control headers, and control authorization, but only do this for URLs under a given set of subdirectories on the server.

9. Must be able to deploy support for cross-origin `GET` requests without having to use server-side scripting (such as PHP, ASP, or CGI) on IIS and Apache.

10. The solution must be applicable to arbitrary media types. It must be deployable without requiring special packaging of resources, or changes to resources' content.

11. It should be possible to configure distinct cross-origin authorization policies for different target resources that reside within the same origin.

12. It should be possible to distribute content of any type. Likewise, it should be possible to transmit content of any type to the server if the API in use allows such functionality.

13. It should be possible to allow only specific servers, or sets of servers to fetch the resource.

14. Must not require that the server filters the entity body of the resource in order to deny cross-origin access to all resources on the server.

15. Cross-origin requests should not require API changes other than allowing cross-origin requests. This means that the following examples should work for resources residing on

`http://test.example` (modulo changes to the respective specifications to allow cross-origin requests):

- ```
  <?xml-stylesheet type="application/xslt+xml" href="http://e
  ```

- ```
  xhr = new XMLHttpRequest();
  xhr.open("GET", "http://example.org/data.text");
  xhr.send();
  ```

16. It should be possible to issue methods other than `GET` to the server, such as `POST` and `DELETE`.

17. Should be compatible with commonly used HTTP authentication and session management mechanisms. I.e. on an IIS server where authentication and session management is generally done by the server before ASP pages execute this should be doable also for requests coming from cross-origin requests. Same thing applies to PHP on Apache.

18. Should reduce the risk of inadvertently allowing access when it is not intended. This is, it should be clear to the content provider when access is granted and when it is not.

Use Cases

This appendix is non-normative.

The main motivation behind Cross-Origin Resource Sharing (CORS) was to remove the same origin restriction from various APIs so that resources can be shared among different origins (i.e. servers).

Here are some examples of how we envision APIs to be able to change with CORS.

`XMLHttpRequest` ([XHR])

Currently if you have an API on the server at `https://calendar.example/add` that accepts requests using the HTTP `PUT` method to add new appointments you can only issue such requests from within the browser environment on resources within the `https://calendar.example/` origin, as follows:

```
new client = new XMLHttpRequest()
client.open("PUT", "https://calendar.example/add")
client.onload = requestSuccess
client.onerror = requestError
client.onabort = requestError
client.send(apointment)
```

If the `https://calendar.example/add` resource implements CORS it can accept requests from other origins. To do this the server has to indicate it is willing to handle HTTP `PUT` methods for non same-origin requests in response to a preflight request. Further when the actual request is issued it has to indicate it is willing to share any response data.

Code Web application developers use to talk with this resource can however remain unmodified, even when put on another origin.

If there is an API on `http://foo.example.org/` that allows authenticated users to edit resources, CORS could be used to allow users to use `http://editor.example/` as editor without the need of proxies when communicating changes to resources (e.g. addition or removal).

Not tainting the `canvas` element ([HTML])

Currently if you have an image editor implemented using the `canvas` element at `http://unicornimages.example` and a clip art collection at `http://narwhalart.example` drawing the clip art on the `canvas` element will cause it to be tainted because the images are from a different origin. The effect of a tainted `canvas` element is that the `toDataURL()` method call in the following snippet will throw:

```
var canvas, context, clipart = []
function init() {
  canvas = document.getElementsByTagName("canvas")[0]
  context = canvas.getContext("2d")
}
function preload() {
  // populates clipart with five images from
  // http://narwhalart.example/archives/[0-9]
  // all represented as HTML <img> elements
  …
}
function draw(clipart) {
  context.drawImage(clipart, …)
}
function save() {
  // get data out of <canvas> and process it
  var data = canvas.toDataURL()
  …
}
```

Using CORS the maintainer of `http://narwhalart.example` can very easily indicate that all images can be used by `http://unicornimages.example` (or in fact all origins). To do so all that is required to change is that the server has to add the following HTTP headers

for the clip art resources:

```
access-control-allow-origin: http://unicornimages.example
access-control-allow-credentials: true
```

This would also make the `toDataURL()` method call no longer throw.

Getting metadata out of media elements ([HTML])

At some point in the future the HTML `video` and `audio` elements will give a programmatic API to access their metadata. This could be as simple as the following snippet shows:

> ***Note: The API itself is pure speculation and its specifics are not relevant for explaining how CORS can be used.***

```
var vid = document.querySelector("video"),
    vidAuthor = vid.meta.author
```

To prevent data theft this API will only work if the media resource is same origin with where the script is executed from. However, if the video were annotated with CORS, similarly to the image resource in the previous use case, this could work just fine.

Server-Sent Events ([EVENTSOURCE])

Currently if `http://example.org/news` exposes a stream of news events only resources on `http://example.org` can make use of it. With CORS it would be very easy to allow `http://international.example.org` to access the stream of news as well. If this news stream is personalized e.g. by the means of cookies it only requires one additional response header for `http://international.example.org` to be able to make use of it:

```
access-control-allow-origin: http://international.example.org
access-control-allow-credentials: true
```

The code used by Web authors would remain near identical (identical if they use an absolute URL):

```
stream = EventSource("http://example.org/news")
stream.onmessage = function(e) { … }
```

`xml-stylesheet` processing instruction ([XMLSS])

Currently cross-origin loads of XSLT resources are prohibited to prevent data theft (e.g. from an intranet). With CORS an XSLT resource `http://static.example.org/generic` can

easily be used by `http://example.org` resources by adding an additional HTTP header to the resource. Again, the code used by Web authors remains the same:

```
<?xml-stylesheet href="http://static.example.org/generic"?>
```

Design Decision FAQ

This appendix is non-normative.

This appendix documents several frequently asked questions and their corresponding response.

Why is there a preflight request?

For most type of requests two resource sharing checks are performed. Initially a "permission to make the request" check is done on the response to the preflight request. And then a "permission to read" check is done on the response to the actual request. Both of these checks need to succeed in order for success to be relayed to the API (e.g. `XMLHttpRequest`).

The "permission to make the request" check is performed because deployed servers do not expect such cross-origin requests. E.g., a request using the HTTP `DELETE` method. If they reply positively to the preflight request the client knows it can go ahead and perform the actual desired request.

Why is `POST` treated similarly to `GET`?

Cross-origin `POST` requests have long been possible using the HTML `form` element. However, this is only the case when `Content-Type` is set to one of the media types allowed by HTML forms.

Why can cookies and authentication be included in the request?

Sending cookies and authentication information enables user-specific cross-origin APIs.

Cookies and authentication information is already sent cross-origin for various HTML elements, such as `img`, `script`, and `form`.

Why can cookies and authentication information *not* be provided by the script author for the request?

This would allow dictionary based, distributed, cookies / user credentials search.

Why is the client the policy enforcement point?

The client already is the policy enforcement point for these requests. The mechanism allows the server to opt-in to let the client expose the data. Something clients currently not do and which servers rely upon.

Note however that the server is in full control. Based on the value of the `Origin` header in cross-origin requests it can decide to return no data at all or not provide the necessary handshake (the `Access-Control-Allow-Origin` header).

What about the `JSONRequest` proposal?

`JSONRequest` has been considered by the Web Applications Working Group and the group has concluded that it does not meet the documented requirements. `JSONRequest` is a specific API and cannot handle e.g. cross-origin XSLT through `<?xml-stylesheet?>` or the same scenarios same-origin `XMLHttpRequest` can handle today in cross-origin fashion, e.g. manipulating resources making use of the REST architectural style.

References

Normative references

[COOKIES]
HTTP State Management Mechanism, Adam Barth. IETF.

[HTML]
HTML, Ian Hickson. WHATWG.

[HTTP]
Hypertext Transfer Protocol -- HTTP/1.1, Roy Fielding, James Gettys, Jeffrey Mogul et al.. IETF.

[ORIGIN]
The Web Origin Concept, Adam Barth. IETF.

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels, Scott Bradner. IETF.

[URI]
Uniform Resource Identifier (URI): Generic Syntax, Tim Berners-Lee, Roy Fielding and Larry Masinter. IETF.

Informative references

[CONFUSED]
The Confused Deputy, Norm Hardy.

[CSRF]

Cross-Site Request Forgeries, Peter Watkins.

[EVENTSOURCE]

Server-Sent Events, Ian Hickson. W3C.

[JSONP]

JSONP, Bob Ippolito.

[OAUTH]

The OAuth 1.0 Protocol, Eran Hammer-Lahav. IETF.

[XHR]

XMLHttpRequest, Anne van Kesteren. W3C.

[XMLSS]

Associating Style Sheets with XML documents 1.0 (Second Edition), James Clark, Simon Pieters and Henry S. Thompson. W3C.

Acknowledgments

This appendix is non-normative.

The editor would like to thank Adam Barth, Alexey Proskuryakov, Arthur Barstow, Benjamin Hawkes-Lewis, Bert Bos, Björn Höhrmann, Boris Zbarsky, Brad Hill, Cameron McCormack, Collin Jackson, David Håsäther, David Orchard, Dean Jackson, Eric Lawrence, Frank Ellerman, Frederick Hirsch, Graham Klyne, Hal Lockhart, Henri Sivonen, Ian Hickson, Jesse M. Heines, Jonas Sicking, Lachlan Hunt, 呂康豪 (Kang-Hao Lu), Maciej Stachowiak, Marc Silbey, Marcos Caceres, Mark Nottingham, Mark S. Miller, Martin Dürst, Matt Womer, Mhano Harkness, Michael Smith, Mohamed Zergaoui, Nikunj Mehta, Odin Hørthe Omdal, Sharath Udupa, Simon Pieters, Sunava Dutta, Surya Ismail, Thomas Roessler, Tyler Close, Vladimir Dzhuvinov, Wayne Carr, and Zhenbin Xu for their contributions to this specification.

Special thanks to Brad Porter, Matt Oshry and R. Auburn, who all helped editing earlier versions of this document.