XMLHttpRequest

Living Standard — Last Updated 24 March 2019

Participate:

GitHub whatwg/xhr (new issue, open issues)
IRC: #whatwg on Freenode

Commits:

GitHub whatwg/xhr/commits
Snapshot as of this commit
@xhrstandard

Tests:

web-platform-tests xhr/ (ongoing work)

Translations (non-normative):

日本語

File an issue about the selected text

Abstract

The XMLHttpRequest Standard defines an API that provides scripted client functionality for transferring data between a client and a server.

Table of Contents

File an issue about the selected text

1. Introduction §

This section is non-normative.

The `XMLHttpRequest` object is an API for fetching resources.

The name `XMLHttpRequest` is historical and has no bearing on its functionality.

Example

Some simple code to do something with data from an XML document fetched over the network:

```
function processData(data) {
  // taking care of data
}

function handler() {
  if(this.status == 200 &&
    this.responseXML != null &&
    this.responseXML.getElementById('test').textContent) {
    // success!
    processData(this.responseXML.getElementById('test').textContent);
  } else {
    // something went wrong
    …
  }
}

var client = new XMLHttpRequest();
client.onload = handler;
client.open("GET", "unicorn.xml");
client.send();
```

If you just want to log a message to the server:

```
function log(message) {
  var client = new XMLHttpRequest();
  client.open("POST", "/log");
  client.setRequestHeader("Content-Type", "text/plain;charset=UTF-8");
  client.send(message);
}
```

Or if you want to check the status of a document on the server:

```
function fetchStatus(address) {
  var client = new XMLHttpRequest();
  client.onload = function() {
    // in case of network errors this might not give reliable results
    returnStatus(this.status);
  }
  client.open("HEAD", address);
  client.send();
}
```

1.1. Specification history §

The `XMLHttpRequest` object was initially defined as part of the WHATWG's HTML effort. (Based on Microsoft's implementation many years prior.) It moved to the W3C in 2006. Extensions (e.g. progress events and cross-origin requests) to `XMLHttpRequest` were developed in a separate draft

File an issue about the selected text

(XMLHttpRequest Level 2) until end of 2011, at which point the two drafts were merged and `XMLHttpRequest` became a single entity again from a standards perspective. End of 2012 it moved back to the WHATWG.

Discussion that led to the current draft can be found in the following mailing list archives:

- whatwg@whatwg.org
- public-webapps@w3.org
- public-webapi@w3.org
- public-appformats@w3.org

File an issue about the selected text

2. Conformance §

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

The key words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this specification are to be interpreted as described in RFC2119. For readability, these words do not appear in all uppercase letters in this specification. [RFC2119]

2.1. Extensibility §

User agents, Working Groups, and other interested parties are *strongly encouraged* to discuss new features with the WHATWG community.

File an issue about the selected text

3. Terminology §

This specification uses terminology, cross-linked throughout, from DOM, DOM Parsing and Serialization, Encoding, Feature Policy, Fetch, File API, HTML, HTTP, URL, Web IDL, and XML.

[DOM] [DOMPS] [ENCODING] [FEATURE-POLICY] [FETCH] [FILEAPI] [HTML] [HTTP] [URL] [WEBIDL] [XML] [XMLNS]

It uses the typographic conventions from HTML. [HTML]

File an issue about the selected text

4. Interface XMLHttpRequest §

IDL
```
[Exposed=(Window,DedicatedWorker,SharedWorker)]
interface XMLHttpRequestEventTarget : EventTarget {
  // event handlers
  attribute EventHandler onloadstart;
  attribute EventHandler onprogress;
  attribute EventHandler onabort;
  attribute EventHandler onerror;
  attribute EventHandler onload;
  attribute EventHandler ontimeout;
  attribute EventHandler onloadend;
};

[Exposed=(Window,DedicatedWorker,SharedWorker)]
interface XMLHttpRequestUpload : XMLHttpRequestEventTarget {
};

enum XMLHttpRequestResponseType {
  "",
  "arraybuffer",
  "blob",
  "document",
  "json",
  "text"
};

[Constructor,
 Exposed=(Window,DedicatedWorker,SharedWorker)]
interface XMLHttpRequest : XMLHttpRequestEventTarget {
  // event handler
  attribute EventHandler onreadystatechange;

  // states
  const unsigned short UNSENT = 0;
  const unsigned short OPENED = 1;
  const unsigned short HEADERS_RECEIVED = 2;
  const unsigned short LOADING = 3;
  const unsigned short DONE = 4;
  readonly attribute unsigned short readyState;

  // request
  void open(ByteString method, USVString url);
  void open(ByteString method, USVString url, boolean async, optional USVString? username = null,
optional USVString? password = null);
  void setRequestHeader(ByteString name, ByteString value);
           attribute unsigned long timeout;
           attribute boolean withCredentials;
  [SameObject] readonly attribute XMLHttpRequestUpload upload;
  void send(optional (Document or BodyInit)? body = null);
  void abort();

  // response
  readonly attribute USVString responseURL;
  readonly attribute unsigned short status;
  readonly attribute ByteString statusText;
  ByteString? getResponseHeader(ByteString name);
  ByteString getAllResponseHeaders();
  void overrideMimeType(DOMString mime);
           attribute XMLHttpRequestResponseType responseType;
  readonly attribute any response;
  readonly attribute USVString responseText;
```

File an issue about the selected text

```
  [Exposed=Window] readonly attribute Document? responseXML;
};
```

An `XMLHttpRequest` object has an associated `XMLHttpRequestUpload` object.

An `XMLHttpRequest` object has an associated **state**, which is one of *unsent*, *opened*, *headers received*, *loading*, and *done*. Unless stated otherwise it is *unsent*.

An `XMLHttpRequest` object has an associated **`send()` flag**. Unless stated otherwise it is unset.

4.1. Constructors §

For web developers (non-normative)

client **= new `XMLHttpRequest()`**

Returns a new `XMLHttpRequest` object.

The **`XMLHttpRequest()`** constructor, when invoked, must return a new `XMLHttpRequest` object.

4.2. Garbage collection §

An `XMLHttpRequest` object must not be garbage collected if its state is either *opened* with the `send()` flag set, *headers received*, or *loading*, and it has one or more event listeners registered whose **type** is one of `readystatechange`, `progress`, `abort`, `error`, `load`, `timeout`, and `loadend`.

If an `XMLHttpRequest` object is garbage collected while its connection is still open, the user agent must terminate the ongoing fetch operated by the `XMLHttpRequest` object.

4.3. Event handlers §

The following are the event handlers (and their corresponding event handler event types) that must be supported on objects implementing an interface that inherits from `XMLHttpRequestEventTarget` as attributes:

event handler	event handler event type
`onloadstart`	`loadstart`
`onprogress`	`progress`
`onabort`	`abort`
`onerror`	`error`
`onload`	`load`
`ontimeout`	`timeout`
`onloadend`	`loadend`

The following is the event handler (and its corresponding event handler event type) that must be supported as attribute solely by the `XMLHttpRequest` object:

event handler	event handler event type
`onreadystatechange`	`readystatechange`

4.4. States §

For web developers (non-normative)

client **. `readyState`**

Returns *client*'s state.

File an issue about the selected text

The **readyState** attribute's getter must return the value from the table below in the cell of the second column, from the row where the value in the cell in the first column is context object's state:

unsent	**UNSENT** (numeric value 0)	The object has been constructed.
opened	**OPENED** (numeric value 1)	The open() method has been successfully invoked. During this state request headers can be set using setRequestHeader() and the fetch can be initiated using the send() method.
headers received	**HEADERS_RECEIVED** (numeric value 2)	All redirects (if any) have been followed and all HTTP headers of the response have been received.
loading	**LOADING** (numeric value 3)	The response's body is being received.
done	**DONE** (numeric value 4)	The data transfer has been completed or something went wrong during the transfer (e.g. infinite redirects).

4.5. Request §

Each XMLHttpRequest object has the following request-associated concepts: **request method**, **request URL**, **author request headers**, **request body**, **synchronous flag**, **upload complete flag**, **upload listener flag**, and **timed out flag**.

The author request headers is an initially empty header list.

The request body is initially null.

The synchronous flag, upload complete flag, upload listener flag and timed out flag are initially unset.

Note

Registering one or more event listeners on an XMLHttpRequestUpload object will result in a CORS-preflight request. (That is because registering an event listener causes the upload listener flag to be set, which in turn causes the use-CORS-preflight flag to be set.)

4.5.1. The open() method §

For web developers (non-normative)

***client* . open(*method*, *url* [, *async* = true [, *username* = null [, *password* = null]]])**

Sets the request method, request URL, and synchronous flag.

Throws a "SyntaxError" DOMException if either *method* is not a valid HTTP method or *url* cannot be parsed.

Throws a "SecurityError" DOMException if *method* is a case-insensitive match for `CONNECT`, `TRACE`, or `TRACK`.

Throws an "InvalidAccessError" DOMException if *async* is false, current global object is a Window object, and the timeout attribute is not zero or the responseType attribute is not the empty string.

Synchronous XMLHttpRequest outside of workers is in the process of being removed from the web platform as it has detrimental effects to the end user's experience. (This is a long process that takes many years.) Developers must not pass false for the *async* argument when current global object is a Window object. User agents are strongly encouraged to warn about such usage in developer tools and may experiment with throwing an "InvalidAccessError" DOMException when it occurs.

The **open(*method*, *url*)** and **open(*method*, *url*, *async*, *username*, *password*)** methods, when invoked, must run these steps:

1. Let *settingsObject* be context object's relevant settings object.
2. If *settingsObject* has a responsible document and it is *not* fully active, then throw an "InvalidStateError" DOMException.
3. If *method* is not a method, then throw a "SyntaxError" DOMException.
4. If *method* is a forbidden method, then throw a "SecurityError" DOMException.
5. Normalize *method*.
6. Let *parsedURL* be the result of parsing *url* with *settingsObject*'s API base URL and *settingsObject*'s API URL character encoding.
7. If *parsedURL* is failure, then throw a "SyntaxError" DOMException.
8. If the *async* argument is omitted, set *async* to true, and set *username* and *password* to null.

Note

Unfortunately legacy content prevents treating the async argument being undefined identical from it being omitted.

File an issue about the selected text

9. If *parsedURL*'s host is non-null, then:

    1. If the *username* argument is not null, set the username given *parsedURL* and *username*.

    2. If the *password* argument is not null, set the password given *parsedURL* and *password*.

10. If *async* is false, current global object is a `Window` object, and the `timeout` attribute value is not zero or the `responseType` attribute value is not the empty string, then throw an "`InvalidAccessError`" `DOMException`.

11. Terminate the ongoing fetch operated by the `XMLHttpRequest` object.

    Note

    A fetch can be ongoing at this point.

12. Set variables associated with the object as follows:

    - Unset the `send()` flag and upload listener flag.
    - Set request method to *method*.
    - Set request URL to *parsedURL*.
    - Set the synchronous flag, if *async* is false, and unset the synchronous flag otherwise.
    - Empty author request headers.
    - Set response to a network error.
    - Set received bytes to the empty byte sequence.
    - Set response object to null.

    Note

    Override MIME type is not overridden here as the `overrideMimeType()` method can be invoked before the `open()` method.

13. If the state is not *opened*, then:

    1. Set state to *opened*.

    2. Fire an event named `readystatechange`.

Note

The reason there are two `open()` methods defined is due to a limitation of the editing software used to write the XMLHttpRequest Standard.

4.5.2. The `setRequestHeader()` method §

For web developers (non-normative)

***client* . `setRequestHeader(name, value)`**

Combines a header in author request headers.

Throws an "`InvalidStateError`" `DOMException` if either state is not *opened* or the `send()` flag is set.

Throws a "`SyntaxError`" `DOMException` if *name* is not a header name or if *value* is not a header value.

The **`setRequestHeader(name, value)`** method must run these steps:

1. If state is not *opened*, then throw an "`InvalidStateError`" `DOMException`.

2. If the `send()` flag is set, then throw an "`InvalidStateError`" `DOMException`.

3. Normalize *value*.

4. If *name* is not a name or *value* is not a value, then throw a "`SyntaxError`" `DOMException`.

    Note

    An empty byte sequence represents an empty header value.

5. Terminate these steps if *name* is a forbidden header name.

6. Combine *name*/*value* in author request headers.

File an issue about the selected text

Example

Some simple code demonstrating what happens when setting the same header twice:

```
// The following script:
var client = new XMLHttpRequest();
client.open('GET', 'demo.cgi');
client.setRequestHeader('X-Test', 'one');
client.setRequestHeader('X-Test', 'two');
client.send();

// …results in the following header being sent:
// X-Test: one, two
```

4.5.3. The `timeout` attribute §

For web developers (non-normative)

`client` **.** **`timeout`**

Can be set to a time in milliseconds. When set to a non-zero value will cause fetching to terminate after the given time has passed. When the time has passed, the request has not yet completed, and the synchronous flag is unset, a `timeout` event will then be dispatched, or a "`TimeoutError`" `DOMException` will be thrown otherwise (for the `send()` method).

When set: throws an "`InvalidAccessError`" `DOMException` if the synchronous flag is set and current global object is a `Window` object.

The **`timeout`** attribute must return its value. Initially its value must be zero.

Setting the `timeout` attribute must run these steps:

1. If current global object is a `Window` object and the synchronous flag is set, then throw an "`InvalidAccessError`" `DOMException`.

2. Set its value to the new value.

Note

This implies that the `timeout` attribute can be set while fetching is in progress. If that occurs it will still be measured relative to the start of fetching.

4.5.4. The `withCredentials` attribute §

For web developers (non-normative)

`client` **.** **`withCredentials`**

True when credentials are to be included in a cross-origin request. False when they are to be excluded in a cross-origin request and when cookies are to be ignored in its response. Initially false.

When set: throws an "`InvalidStateError`" `DOMException` if state is not *unsent* or *opened*, or if the `send()` flag is set.

The **`withCredentials`** attribute must return its value. Initially its value must be false.

Setting the `withCredentials` attribute must run these steps:

1. If state is not *unsent* or *opened*, then throw an "`InvalidStateError`" `DOMException`.

2. If the `send()` flag is set, then throw an "`InvalidStateError`" `DOMException`.

3. Set the `withCredentials` attribute's value to the given value.

Note

The `withCredentials` attribute has no effect when fetching same-origin resources.

File an issue about the selected text

4.5.5. The `upload` attribute §

For web developers (non-normative)

`client` **`. upload`**

Returns the associated `XMLHttpRequestUpload` object. It can be used to gather transmission information when data is transferred to a server.

The **`upload`** attribute must return the associated `XMLHttpRequestUpload` object.

Note

As indicated earlier, each `XMLHttpRequest` object has an associated `XMLHttpRequestUpload` object.

4.5.6. The `send()` method §

For web developers (non-normative)

`client` **`. send([body = null])`**

Initiates the request. The *body* argument provides the request body, if any, and is ignored if the request method is `GET` or `HEAD`.

Throws an "`InvalidStateError`" `DOMException` if either state is not *opened* or the `send()` flag is set.

The **`send(body)`** method must run these steps:

1. If state is not *opened*, then throw an "`InvalidStateError`" `DOMException`.
2. If the `send()` flag is set, then throw an "`InvalidStateError`" `DOMException`.
3. If the request method is `GET` or `HEAD`, set *body* to null.
4. If *body* is not null, then:
    1. Let *extractedContentType* be null.
    2. If *body* is a `Document`, then set request body to *body*, serialized, converted to Unicode, and UTF-8 encoded.
    3. Otherwise, set request body and *extractedContentType* to the result of extracting *body*.
    4. If author request headers contains `` `Content-Type` ``, then:
        1. If *body* is a `Document` or a `USVString`, then:
            1. Let *originalAuthorContentType* be the value of the header whose name is a byte-case-insensitive match for `` `Content-Type` `` in author request headers.
            2. Let *contentTypeRecord* be the result of parsing *originalAuthorContentType*.
            3. If *contentTypeRecord* is not failure, *contentTypeRecord*'s parameters["`charset`"] exists, and parameters["`charset`"] is not an ASCII case-insensitive match for "`UTF-8`", then:
                1. Set *contentTypeRecord*'s parameters["`charset`"] to "`UTF-8`".
                2. Let *newContentTypeSerialized* be the result of serializing *contentTypeRecord*.
                3. Set `` `Content-Type` ``/*newContentTypeSerialized* in author request headers.
    5. Otherwise:
        1. If *body* is a HTML document, set `` `Content-Type` ``/`` `text/html;charset=UTF-8` `` in author request headers.
        2. Otherwise, if *body* is an XML document, set `` `Content-Type` ``/`` `application/xml;charset=UTF-8` `` in author request headers.
        3. Otherwise, if *extractedContentType* is not null, set `` `Content-Type` ``/*extractedContentType* in author request headers.
5. If one or more event listeners are registered on the associated `XMLHttpRequestUpload` object, then set upload listener flag.
6. Let *req* be a new request, initialized as follows:

File an issue about the selected text

request method

url
request URL

header list
author request headers

unsafe-request flag
Set.

body
request body

client
context object's relevant settings object

synchronous flag
Set if the synchronous flag is set.

mode
"`cors`"

use-CORS-preflight flag
Set if upload listener flag is set.

credentials mode
If the `withCredentials` attribute value is true, "`include`", and "`same-origin`" otherwise.

use-URL-credentials flag
Set if either request URL's username is not the empty string or request URL's password is non-null.

7. Unset the upload complete flag.
8. Unset the timed out flag.
9. If *req*'s body is null, set the upload complete flag.
10. Set the `send()` flag.
11. If the synchronous flag is unset, then:
    1. Fire a progress event named `loadstart` with 0 and 0.
    2. If the upload complete flag is unset and upload listener flag is set, then fire a progress event named `loadstart` on the `XMLHttpRequestUpload` object with 0 and *req*'s body's total bytes.
    3. If state is not *opened* or the `send()` flag is unset, then return.
    4. Fetch *req*. Handle the tasks queued on the networking task source per below.

       Run these steps in parallel:

       1. Wait until either *req*'s done flag is set or
          1. the `timeout` attribute value number of milliseconds has passed since these steps started
          2. while `timeout` attribute value is not zero.
       2. If *req*'s done flag is unset, then set the timed out flag and terminate fetching.

       To process request body for *request*, run these steps:

       1. If not roughly 50ms have passed since these steps were last invoked, terminate these steps.
       2. If upload listener flag is set, then fire a progress event named `progress` on the `XMLHttpRequestUpload` object with *request*'s body's transmitted bytes and *request*'s body's total bytes.

       Note

       These steps are only invoked when new bytes are transmitted.

       To process request end-of-body for *request*, run these steps:

       1. Set the upload complete flag.
       2. If upload listener flag is unset, then terminate these steps.

File an issue about the selected text *smitted* be *request*'s body's transmitted bytes.

        4. Let *length* be *request*’s body’s total bytes.

        5. Fire a progress event named `progress` on the `XMLHttpRequestUpload` object with *transmitted* and *length*.

        6. Fire a progress event named `load` on the `XMLHttpRequestUpload` object with *transmitted* and *length*.

        7. Fire a progress event named `loadend` on the `XMLHttpRequestUpload` object with *transmitted* and *length*.

    To process response for *response*, run these steps:

    1. Set response to *response*.

    2. Handle errors for *response*.

    3. If response is a network error, return.

    4. Set state to *headers received*.

    5. Fire an event named `readystatechange`.

    6. If state is not *headers received*, then return.

    7. If *response*’s body is null, then run handle response end-of-body and return.

    8. Let *reader* be the result of getting a reader from *response*’s body’s stream.

       Note *This operation will not throw an exception.*

    9. Let *read* be the result of reading a chunk from *response*’s body’s stream with *reader*.

       When *read* is fulfilled with an object whose `done` property is false and whose `value` property is a `Uint8Array` object, run these steps and then run this step again:

       1. Append the `value` property to received bytes.

       2. If not roughly 50ms have passed since these steps were last invoked, then terminate these steps.

       3. If state is *headers received*, then set state to *loading*.

       4. Fire an event named `readystatechange`.

          Note

          Web compatibility is the reason `readystatechange` fires more often than state changes.

       5. Fire a progress event named `progress` with *response*’s body’s transmitted bytes and *response*’s body’s total bytes.

       Note

       These steps are only invoked when new bytes are transmitted.

       When *read* is fulfilled with an object whose `done` property is true, run handle response end-of-body for *response*.

       When *read* is rejected with an exception, run handle errors for *response*.

12. Otherwise, if the synchronous flag is set, run these steps:

    1. If context object’s relevant settings object has a responsible document which is *not* allowed to use the "`sync-xhr`" feature, then run handle response end-of-body for a network error and return.

    2. Let *response* be the result of fetching *req*.

       If the `timeout` attribute value is not zero, then set the timed out flag and terminate fetching if it has not returned within the amount of milliseconds from the `timeout`.

    3. If *response*’s body is null, then run handle response end-of-body and return.

    4. Let *reader* be the result of getting a reader from *response*’s body’s stream.

       Note *This operation will not throw an exception.*

    5. Let *promise* be the result of reading all bytes from *response*’s body’s stream with *reader*.

    6. Wait for *promise* to be fulfilled or rejected.

    7. If *promise* is fulfilled with *bytes*, then append *bytes* to received bytes.

    8. Run handle response end-of-body for *response*.

File an issue about the selected text

To **handle response end-of-body** for *response*, run these steps:

1. If the synchronous flag is set, set response to *response*.
2. Handle errors for *response*.
3. If response is a network error, return.
4. If the synchronous flag is unset, update response's body using *response*.
5. Let *transmitted* be *response*'s body's transmitted bytes.
6. Let *length* be *response*'s body's total bytes.
7. If the synchronous flag is unset, fire a progress event named `progress` with *transmitted* and *length*.
8. Set state to *done*.
9. Unset the `send()` flag.
10. Fire an event named `readystatechange`.
11. Fire a progress event named `load` with *transmitted* and *length*.
12. Fire a progress event named `loadend` with *transmitted* and *length*.

To **handle errors** for *response* run these steps:

1. If the `send()` flag is unset, return.
2. If the timed out flag is set, then run the request error steps for event `timeout` and exception "`TimeoutError`" `DOMException`.
3. If *response* is a network error, then run the request error steps for event `error` and exception "`NetworkError`" `DOMException`.
4. Otherwise, if *response*'s body's stream is errored, then:
    1. Set state to *done*.
    2. Unset the `send()` flag.
    3. Set response to a network error.
5. Otherwise, if *response*'s aborted flag is set, then run the request error steps for event `abort` and exception "`AbortError`" `DOMException`.

The **request error steps** for event *event* and optionally an exception *exception* are:

1. Set state to *done*.
2. Unset the `send()` flag.
3. Set response to a network error.
4. If the synchronous flag is set, throw an *exception* exception.
5. Fire an event named `readystatechange`.

   Note

   At this point it is clear that the synchronous flag is unset.

6. If the upload complete flag is unset, then:
    1. Set the upload complete flag.
    2. If the upload listener flag is set, then:
        1. Fire a progress event named *event* on the `XMLHttpRequestUpload` object with 0 and 0.
        2. Fire a progress event named `loadend` on the `XMLHttpRequestUpload` object with 0 and 0.
7. Fire a progress event named *event* with 0 and 0.
8. Fire a progress event named `loadend` with 0 and 0.

4.5.7. The `abort()` method §

File an issue about the selected text ve)

***client* . abort()**

Cancels any network activity.

The **abort()** method, when invoked, must run these steps:

1. Terminate the ongoing fetch with the *aborted* flag set.
2. If state is either *opened* with the send() flag set, *headers received*, or *loading*, run the request error steps for event abort.
3. If state is *done*, then set state to *unsent* and response to a network error.

Note

No readystatechange event is dispatched.

4.6. Response §

An XMLHttpRequest has an associated **response**. Unless stated otherwise it is a network error.

An XMLHttpRequest also has an associated **received bytes** (a byte sequence). Unless stated otherwise it is the empty byte sequence.

4.6.1. The responseURL attribute §

The **responseURL** attribute must return the empty string if response's url is null and its serialization with the *exclude fragment flag* set otherwise.

4.6.2. The status attribute §

The **status** attribute must return the response's status.

4.6.3. The statusText attribute §

The **statusText** attribute must return the response's status message.

4.6.4. The getResponseHeader() method §

The **getResponseHeader(*name*)** method, when invoked, must return the result of getting *name* from response's header list

Note

The Fetch Standard filters response's header list. [FETCH]

Example

For the following script:

```
var client = new XMLHttpRequest();
client.open("GET", "unicorns-are-teh-awesome.txt", true);
client.send();
client.onreadystatechange = function() {
  if(this.readyState == this.HEADERS_RECEIVED) {
    print(client.getResponseHeader("Content-Type"));
  }
}
```

The print() function will get to process something like:

File an issue about the selected text

```
et=UTF-8
```

4.6.5. The `getAllResponseHeaders()` method §

The `getAllResponseHeaders()` method, when invoked, must run these steps:

1. Let *output* be an empty byte sequence.
2. Let *headers* be the result of running sort and combine with response's header list.
3. For each *header* in *headers*, append *header*'s name, followed by a 0x3A 0x20 byte pair, followed by *header*'s value, followed by a 0x0D 0x0A byte pair, to *output*.
4. Return *output*.

Note

The Fetch Standard filters response's header list. [FETCH]

Example

For the following script:

```
var client = new XMLHttpRequest();
client.open("GET", "narwhals-too.txt", true);
client.send();
client.onreadystatechange = function() {
  if(this.readyState == this.HEADERS_RECEIVED) {
    print(this.getAllResponseHeaders());
  }
}
```

The `print()` function will get to process something like:

```
connection: Keep-Alive
content-type: text/plain; charset=utf-8
date: Sun, 24 Oct 2004 04:58:38 GMT
keep-alive: timeout=15, max=99
server: Apache/1.3.31 (Unix)
transfer-encoding: chunked
```

4.6.6. Response body §

The **response MIME type** is the result of running these steps:

1. Let *mimeType* be the result of extracting a MIME type from response's header list.
2. If *mimeType* is failure, then set *mimeType* to `text/xml`.
3. Return *mimeType*.

The **override MIME type** is initially null and can get a value when `overrideMimeType()` is invoked. The **final MIME type** is the override MIME type unless that is null in which case it is the response MIME type.

The **final charset** is the return value of these steps:

1. Let *label* be null.
2. If response MIME type's parameters["`charset`"] exists, then set *label* to it.
3. If override MIME type's parameters["`charset`"] exists, then set *label* to it.
4. If *label* is null, then return null.
5. Let *encoding* be the result of getting an encoding from *label*.
6. If *encoding* is failure, then return null.

File an issue about the selected text

7. Return *encoding*.

Note

The above steps intentionally do not use the final MIME type as it would yield the wrong result.

An `XMLHttpRequest` object has an associated **response object** (an object, failure, or null). Unless stated otherwise it is null.

An **arraybuffer response** is the return value of these steps:

1. Set response object to a new `ArrayBuffer` object representing received bytes. If this throws an exception, then set response object to failure and return null.

   Note

   Allocating an `ArrayBuffer` object is not guaranteed to succeed. [ECMASCRIPT]

2. Return response object.

A **blob response** is the return value of these steps:

1. Set response object to a new `Blob` object representing received bytes with `type` set to the final MIME type.

2. Return response object.

A **document response** is the return value of these steps:

1. If response's body is null, then return null.

2. If the final MIME type is not an HTML MIME type or an XML MIME type, then return null.

3. If `responseType` is the empty string and the final MIME type is an HTML MIME type, then return null.

   Note

   This is restricted to `responseType` being "`document`" in order to prevent breaking legacy content.

4. If the final MIME type is an HTML MIME type, then:

   1. Let *charset* be the final charset.

   2. If *charset* is null, prescan the first 1024 bytes of received bytes and if that does not terminate unsuccessfully then let *charset* be the return value.

   3. If *charset* is null, then set *charset* to UTF-8.

   4. Let *document* be a document that represents the result parsing received bytes following the rules set forth in the HTML Standard for an HTML parser with scripting disabled and a known definite encoding *charset*. [HTML]

   5. Flag *document* as an HTML document.

5. Otherwise, let *document* be a document that represents the result of running the XML parser with XML scripting support disabled on received bytes. If that fails (unsupported character encoding, namespace well-formedness error, etc.), then return null. [HTML]

   Note

   Resources referenced will not be loaded and no associated XSLT will be applied.

6. If *charset* is null, then set *charset* to UTF-8.

7. Set *document*'s encoding to *charset*.

8. Set *document*'s content type to the final MIME type.

9. Set *document*'s URL to response's url.

10. Set *document*'s origin to context object's relevant settings object's origin.

11. Set response object to *document* and return it.

A **JSON response** is the return value of these steps:

1. If response's body is null, then return null.

File an issue about the selected text sult of running parse JSON from bytes on received bytes. If that threw an exception, then return null.

3. Set response object to *jsonObject* and return it.

A **text response** is the return value of these steps:

1. If response's body is null, then return the empty string.
2. Let *charset* be the final charset.
3. If `responseType` is the empty string, *charset* is null, and the final MIME type is an XML MIME type, then use the rules set forth in the XML specifications to determine the encoding. Let *charset* be the determined encoding. [XML] [XMLNS]

   Note

   This is restricted to `responseType` being the empty string to keep the non-legacy `responseType` value "`text`" simple.

4. If *charset* is null, then set *charset* to UTF-8.
5. Return the result of running decode on received bytes using fallback encoding *charset*.

Note

Authors are strongly encouraged to always encode their resources using UTF-8.

4.6.7. The `overrideMimeType()` method §

For web developers (non-normative)

***client* . `overrideMimeType(`*mime*`)`**

Acts as if the `` `Content-Type` `` header value for response is *mime*. (It does not actually change the header though.)

Throws an "`InvalidStateError`" `DOMException` if state is *loading* or *done*.

The **`overrideMimeType(`*mime*`)`** method, when invoked, must run these steps:

1. If state is *loading* or *done*, then throw an "`InvalidStateError`" `DOMException`.
2. Set override MIME type to the result of parsing *mime*.
3. If override MIME type is failure, then set override MIME type to `application/octet-stream`.

4.6.8. The `responseType` attribute §

For web developers (non-normative)

***client* . `responseType` [= *value*]**

Returns the response type.

Can be set to change the response type. Values are: the empty string (default), "`arraybuffer`", "`blob`", "`document`", "`json`", and "`text`".

When set: setting to "`document`" is ignored if current global object is *not* a `Window` object.

When set: throws an "`InvalidStateError`" `DOMException` if state is *loading* or *done*.

When set: throws an "`InvalidAccessError`" `DOMException` if the synchronous flag is set and current global object is a `Window` object.

The **`responseType`** attribute must return its value. Initially its value must be the empty string.

Setting the `responseType` attribute must run these steps:

1. If current global object is *not* a `Window` object and the given value is "`document`", terminate these steps.
2. If state is *loading* or *done*, then throw an "`InvalidStateError`" `DOMException`.
3. If current global object is a `Window` object and the synchronous flag is set, then throw an "`InvalidAccessError`" `DOMException`.
4. Set the `responseType` attribute's value to the given value.

File an issue about the selected text

4.6.9. The `response` attribute §

For web developers (non-normative)

***client* . response**

Returns the response's body.

The **`response`** attribute must return the result of running these steps:

↪ **If `responseType` is the empty string or "`text`"**

1. If state is not *loading* or *done*, return the empty string.
2. Return the text response.

↪ **Otherwise**

1. If state is not *done*, return null.
2. If response object is failure, then return null.
3. If response object is non-null, then return it.
4. ↪ **If `responseType` is "`arraybuffer`"**
   Return the arraybuffer response.

   ↪ **If `responseType` is "`blob`"**
   Return the blob response.

   ↪ **If `responseType` is "`document`"**
   Return the document response.

   ↪ **If `responseType` is "`json`"**
   Return the JSON response.

4.6.10. The `responseText` attribute §

For web developers (non-normative)

***client* . responseText**

Returns the text response.

Throws an "`InvalidStateError`" `DOMException` if `responseType` is not the empty string or "`text`".

The **`responseText`** attribute must return the result of running these steps:

1. If `responseType` is not the empty string or "`text`", then throw an "`InvalidStateError`" `DOMException`.
2. If state is not *loading* or *done*, then return the empty string.
3. Return the text response.

4.6.11. The `responseXML` attribute §

For web developers (non-normative)

***client* . responseXML**

Returns the document response.

Throws an "`InvalidStateError`" `DOMException` if `responseType` is not the empty string or "`document`".

The **`responseXML`** attribute must return the result of running these steps:

1. If `responseType` is not the empty string or "`document`", then throw an "`InvalidStateError`" `DOMException`.

File an issue about the selected text

2. If state is not *done*, then return null.
3. Assert: response object is not failure.
4. If response object is non-null, then return it.
5. Return the document response.

4.7. Events summary §

This section is non-normative.

The following events are dispatched on `XMLHttpRequest` or `XMLHttpRequestUpload` objects:

Event name	Interface	Dispatched when…
`readystatechange`	`Event`	The `readyState` attribute changes value, except when it changes to UNSENT.
`loadstart`	`ProgressEvent`	The fetch initiates.
`progress`	`ProgressEvent`	Transmitting data.
`abort`	`ProgressEvent`	When the fetch has been aborted. For instance, by invoking the `abort()` method.
`error`	`ProgressEvent`	The fetch failed.
`load`	`ProgressEvent`	The fetch succeeded.
`timeout`	`ProgressEvent`	The author specified timeout has passed before the fetch completed.
`loadend`	`ProgressEvent`	The fetch completed (success or failure).

4.8. Feature Policy integration §

This specification defines a policy-controlled feature identified by the string "**`sync-xhr`**". Its default allowlist is `*`.

File an issue about the selected text

5. Interface FormData §

```
IDL
typedef (File or USVString) FormDataEntryValue;

[Constructor(optional HTMLFormElement form),
 Exposed=(Window,Worker)]
interface FormData {
  void append(USVString name, USVString value);
  void append(USVString name, Blob blobValue, optional USVString filename);
  void delete(USVString name);
  FormDataEntryValue? get(USVString name);
  sequence<FormDataEntryValue> getAll(USVString name);
  boolean has(USVString name);
  void set(USVString name, USVString value);
  void set(USVString name, Blob blobValue, optional USVString filename);
  iterable<USVString, FormDataEntryValue>;
};
```

Each `FormData` object has an associated **entry list** (a list of entries). It is initially the empty list.

An **entry** consists of a **name** and a **value**.

For the purposes of interaction with other algorithms, an entry's filename is the empty string if value is not a `File` object, and otherwise its filename is the value of entry's value's `name` attribute.

To **create an entry** for *name*, *value*, and optionally a *filename*, run these steps:

1. Let *entry* be a new entry.
2. Set *entry*'s name to *name*.
3. If *value* is a `Blob` object and not a `File` object, then set *value* to a new `File` object, representing the same bytes, whose `name` attribute value is "`blob`".
4. If *value* is (now) a `File` object and *filename* is given, then set *value* to a new `File` object, representing the same bytes, whose `name` attribute value is *filename*.
5. Set *entry*'s value to *value*.
6. Return *entry*.

The **`FormData(form)`** constructor must run these steps:

1. Let *fd* be a new `FormData` object.
2. If *form* is given, then:
    1. Let *list* be the result of constructing the entry list for *form*.
    2. If *list* is null, then throw an "`InvalidStateError`" `DOMException`.
    3. Set *fd*'s entry list to *list*.
3. Return *fd*.

The **`append(name, value)`** and **`append(name, blobValue, filename)`** methods, when invoked, must run these steps:

1. Let *value* be *value* if given, and *blobValue* otherwise.
2. Let *entry* be the result of creating an entry with *name*, *value*, and *filename* if given.
3. Append *entry* to the context object's entry list.

File an issue about the selected text

Note

The reason there is an argument named value *as well as* blobValue *is due to a limitation of the editing software used to write the XMLHttpRequest Standard.*

The **`delete(`*`name`*`)`** method, when invoked, must remove all entries whose name is *name* from the context object's entry list.

The **`get(`*`name`*`)`** method, when invoked, must return the value of the first entry whose name is *name* from the context object's entry list, and null otherwise.

The **`getAll(`*`name`*`)`** method, when invoked, must return the values of all entries whose name is *name*, in order, from the context object's entry list, and the empty list otherwise.

The **`has(`*`name`*`)`** method, when invoked, must return true if there is an entry whose name is *name* in the context object's entry list, and false otherwise.

The **`set(`*`name, value`*`)`** and **`set(`*`name, blobValue, filename`*`)`** methods, when invoked, must run these steps:

1. Let *value* be *value* if given, and *blobValue* otherwise.
2. Let *entry* be the result of creating an entry with *name*, *value*, and *filename* if given.
3. If there are any entries in the context object's entry list whose name is *name*, then replace the first such entry with *entry* and remove the others.
4. Otherwise, append *entry* to the context object's entry list.

Note

The reason there is an argument named value *as well as* blobValue *is due to a limitation of the editing software used to write the XMLHttpRequest Standard.*

The value pairs to iterate over are the context object's entry list's entries with the key being the name and the value being the value.

File an issue about the selected text

6. Interface ProgressEvent §

```
IDL [Constructor(DOMString type, optional ProgressEventInit eventInitDict),
     Exposed=(Window,DedicatedWorker,SharedWorker)]
    interface ProgressEvent : Event {
      readonly attribute boolean lengthComputable;
      readonly attribute unsigned long long loaded;
      readonly attribute unsigned long long total;
    };

    dictionary ProgressEventInit : EventInit {
      boolean lengthComputable = false;
      unsigned long long loaded = 0;
      unsigned long long total = 0;
    };
```

Events using the `ProgressEvent` interface indicate some kind of progression.

The **`lengthComputable`**, **`loaded`**, and **`total`** attributes must return the value they were initialized to.

6.1. Firing events using the ProgressEvent interface §

To **fire a progress event** named *e* at *target*, given *transmitted* and *length*, means to fire an event named *e* at *target*, using `ProgressEvent`, with the `loaded` attribute initialized to *transmitted*, and if *length* is not 0, with the `lengthComputable` attribute initialized to true and the `total` attribute initialized to *length*.

6.2. Suggested names for events using the ProgressEvent interface §

This section is non-normative.

The suggested `type` attribute values for use with events using the `ProgressEvent` interface are summarized in the table below. Specification editors are free to tune the details to their specific scenarios, though are strongly encouraged to discuss their usage with the WHATWG community to ensure input from people familiar with the subject.

`type` attribute value	Description	Times	When
`loadstart`	Progress has begun.	Once.	First.
`progress`	In progress.	Once or more.	After `loadstart` has been dispatched.
`error`	Progression failed.	Zero or once (mutually exclusive).	After the last `progress` has been dispatched.
`abort`	Progression is terminated.		
`timeout`	Progression is terminated due to preset time expiring.		
`load`	Progression is successful.		
`loadend`	Progress has stopped.	Once.	After one of `error`, `abort`, `timeout` or `load` has been dispatched.

The `error`, `abort`, `timeout`, and `load` event types are mutually exclusive.

Throughout the web platform the `error`, `abort`, `timeout` and `load` event types have their `bubbles` and `cancelable` attributes initialized to false, so it is suggested that for consistency all events using the `ProgressEvent` interface do the same.

6.3. Security considerations §

For cross-origin requests some kind of opt-in, e.g. the CORS protocol defined in the Fetch Standard, has to be used before events using the

File an issue about the selected text ispatched as information (e.g. size) would be revealed that cannot be obtained otherwise. [FETCH]

6.4. Example §

Example

In this example XMLHttpRequest, combined with concepts defined in the sections before, and the HTML progress element are used together to display the process of fetching a resource.

```
<!DOCTYPE html>
<title>Waiting for Magical Unicorns</title>
<progress id=p></progress>
<script>
  var progressBar = document.getElementById("p"),
      client = new XMLHttpRequest()
  client.open("GET", "magical-unicorns")
  client.onprogress = function(pe) {
    if(pe.lengthComputable) {
      progressBar.max = pe.total
      progressBar.value = pe.loaded
    }
  }
  client.onloadend = function(pe) {
    progressBar.value = pe.loaded
  }
  client.send()
</script>
```

Fully working code would of course be more elaborate and deal with more scenarios, such as network errors or the end user terminating the request.

File an issue about the selected text

Acknowledgments §

Thanks to Addison Phillips, Adrian Bateman, Ahmed Kamel, Alan Thomas, Alex Hopmann, Alex Vincent, Alexey Proskuryakov, Ali Alabbas, Andrea Marchesini, Asbjørn Ulsberg, Bertrand Guay-Paquet, Björn Höhrmann, Boris Zbarsky, Caitlin Potter, Cameron McCormack, 白丞祐 (Cheng-You Bai), Chris Marrin, Christophe Jolif, Charles McCathieNevile, Dan Winship, David Andersson, David Flanagan, David Håsäther, David Levin, Dean Jackson, Denis Sureau, Domenic Denicola, Dominik Röttsches, Doug Schepers, Douglas Livingstone, Elliott Sprehn, Elliotte Harold, Eric Lawrence, Eric Uhrhane, Erik Arvidsson, Erik Dahlström, Feras Moussa, Gideon Cohn, Glenn Adams, Gorm Haug Eriksen, Håkon Wium Lie, Hallvord R. M. Steen, Henri Sivonen, Hiroshige Hayashizaki, Huub Schaeks, Ian Clelland, Ian Davis, Ian Hickson, Ivan Herman, Jake Archibald, Jared Jacobs, Jarred Nicholls, Jeff Walden, Jens Lindström, Jim Deegan, Jim Ley, Joe Farro, Jonas Sicking, Julian Reschke, 송정기 (Jungkee Song), 呂康豪 (Kang-Hao Lu), Karl Dubost, Keith Yeung, 田村健人 (Kent TAMURA), Lachlan Hunt, Maciej Stachowiak, Magnus Kristiansen, Manish Goregaokar, Marc Hadley, Marcos Caceres, Mark Baker, Mark Birbeck, Mark Nottingham, Mark S. Miller, Martin Hassman, Mohamed Zergaoui, Ms2ger, Odin Hørthe Omdal, Olli Pettay, Pawel Glowacki, Peter Michaux, Philip Jägenstedt, Philip Taylor, Robin Berjon, Rune F. Halvorsen, Ruud Steltenpool, Ryo Onodera, Sam Sneddon, Sergiu Dumitriu, Shivakumar Jagalur Matt, Sigbjørn Finne, Simon Pieters, Stewart Brodie, Sunava Dutta, Takeshi Kurosawa, Takeshi Yoshino, Thomas Roessler, Thomas Wisniewski, Tom Magliery, Travis Leithead, triple-underscore, Yaron Tausky, Yehuda Katz, Youenn Fablet, and Zhenbin Xu for their contributions to this standard.

Special thanks to the Microsoft employees who first implemented the `XMLHttpRequest` interface, which was first widely deployed by the Windows Internet Explorer browser.

Special thanks to Ian Hickson for drafting an initial version of this specification in the HTML Standard (then Web Applications 1.0). [HTML]

Special thanks to the W3C SVG WG for drafting the original `ProgressEvent` class as part of the SVG Micro DOM.

This standard is written by Anne van Kesteren (Mozilla, annevk@annevk.nl).

File an issue about the selected text

Index §

Terms defined by this specification §

File an issue about the selected text

Terms defined by reference §

- [DOM] defines the following terms:
  - Document
  - Event
  - EventInit
  - EventTarget
  - bubbles
  - cancelable

File an issue about the selected text

  - content type
  - context object
  - dispatch
  - document
  - encoding
  - event
  - event listener
  - fire an event
  - html document
  - origin
  - type
  - url
  - xml document
- [DOMPS] defines the following terms:
  - fragment serializing algorithm
- [ENCODING] defines the following terms:
  - decode
  - getting an encoding
  - utf-8
  - utf-8 encode
- [FEATURE-POLICY] defines the following terms:
  - default allowlist
  - policy-controlled feature
- [FETCH] defines the following terms:
  - BodyInit
  - aborted flag
  - body (for response)
  - client
  - combine
  - contains
  - cors protocol
  - cors-preflight request
  - credentials
  - credentials mode
  - done flag
  - errored
  - extract
  - extracting a mime type
  - fetch
  - forbidden header name
  - forbidden method
  - get
  - get a reader
  - header
  - header list (for response)
  - method (for request)
  - mode
  - name
  - network error
  - normalize (for method)
  - process request body
  - process request end-of-body
  - process response
  - read a chunk
  - read all bytes
  - request
  - set
  - sort and combine
  - status
  - status message
  - stream
  - synchronous flag
  - terminated
  - total bytes
  - transmitted bytes
  - unsafe-request flag
  - url (for response)
  - use-cors-preflight flag
  - use-url-credentials flag

File an issue about the selected text

  - value
- [FILEAPI] defines the following terms:
  - Blob
  - File
  - name
  - type
- [HTML] defines the following terms:
  - EventHandler
  - HTMLFormElement
  - Window
  - a known definite encoding
  - allowed to use
  - api base url
  - api url character encoding
  - constructing the entry list
  - current global object
  - event handler
  - event handler event type
  - fully active
  - in parallel
  - networking task source
  - origin
  - prescan a byte stream to determine its encoding
  - progress
  - queue a task
  - relevant settings object
  - responsible document
  - same origin
  - task
  - xml parser
  - xml scripting support disabled
- [INFRA] defines the following terms:
  - append
  - ascii case-insensitive
  - byte-case-insensitive
  - exist
  - for each
  - list
  - parse json from bytes
  - remove
  - replace
  - set
- [MIMESNIFF] defines the following terms:
  - html mime type
  - parameters
  - parse a mime type
  - parse a mime type from bytes
  - serialize a mime type to bytes
  - xml mime type
- [URL] defines the following terms:
  - host
  - password
  - set the password
  - set the username
  - url parser
  - url serializer
  - username
- [WEBIDL] defines the following terms:
  - AbortError
  - ArrayBuffer
  - ByteString
  - DOMException
  - DOMString
  - Exposed
  - InvalidAccessError
  - InvalidStateError
  - NetworkError
  - SameObject
  - SecurityError

File an issue about the selected text

- SyntaxError
- TimeoutError
- USVString
- boolean
- obtain unicode
- throw
- unsigned long
- unsigned long long
- unsigned short
- value pairs to iterate over

File an issue about the selected text

References §

Normative References §

[DOM]

Anne van Kesteren. DOM Standard. Living Standard. URL: https://dom.spec.whatwg.org/

[DOMPS]

Travis Leithead. DOM Parsing and Serialization. URL: https://w3c.github.io/DOM-Parsing/

[ECMASCRIPT]

ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[ENCODING]

Anne van Kesteren. Encoding Standard. Living Standard. URL: https://encoding.spec.whatwg.org/

[FEATURE-POLICY]

Feature Policy. Living Standard. URL: https://wicg.github.io/feature-policy/

[FETCH]

Anne van Kesteren. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/

[FILEAPI]

Marijn Kruisselbrink; Arun Ranganathan. File API. URL: https://w3c.github.io/FileAPI/

[HTML]

Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[HTTP]

R. Fielding, Ed.; J. Reschke, Ed.. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7230

[INFRA]

Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[MIMESNIFF]

Gordon P. Hemsley. MIME Sniffing Standard. Living Standard. URL: https://mimesniff.spec.whatwg.org/

[RFC2119]

S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[URL]

Anne van Kesteren. URL Standard. Living Standard. URL: https://url.spec.whatwg.org/

[WEBIDL]

Boris Zbarsky. Web IDL. URL: https://heycam.github.io/webidl/

[XML]

Tim Bray; et al. Extensible Markup Language (XML) 1.0 (Fifth Edition). 26 November 2008. REC. URL: https://www.w3.org/TR/xml/

[XMLNS]

Tim Bray; et al. Namespaces in XML 1.0 (Third Edition). 8 December 2009. REC. URL: https://www.w3.org/TR/xml-names/

File an issue about the selected text

IDL Index §

```
IDL
[Exposed=(Window,DedicatedWorker,SharedWorker)]
interface XMLHttpRequestEventTarget : EventTarget {
  // event handlers
  attribute EventHandler onloadstart;
  attribute EventHandler onprogress;
  attribute EventHandler onabort;
  attribute EventHandler onerror;
  attribute EventHandler onload;
  attribute EventHandler ontimeout;
  attribute EventHandler onloadend;
};

[Exposed=(Window,DedicatedWorker,SharedWorker)]
interface XMLHttpRequestUpload : XMLHttpRequestEventTarget {
};

enum XMLHttpRequestResponseType {
  "",
  "arraybuffer",
  "blob",
  "document",
  "json",
  "text"
};

[Constructor,
 Exposed=(Window,DedicatedWorker,SharedWorker)]
interface XMLHttpRequest : XMLHttpRequestEventTarget {
  // event handler
  attribute EventHandler onreadystatechange;

  // states
  const unsigned short UNSENT = 0;
  const unsigned short OPENED = 1;
  const unsigned short HEADERS_RECEIVED = 2;
  const unsigned short LOADING = 3;
  const unsigned short DONE = 4;
  readonly attribute unsigned short readyState;

  // request
  void open(ByteString method, USVString url);
  void open(ByteString method, USVString url, boolean async, optional USVString? username = null,
optional USVString? password = null);
  void setRequestHeader(ByteString name, ByteString value);
           attribute unsigned long timeout;
           attribute boolean withCredentials;
  [SameObject] readonly attribute XMLHttpRequestUpload upload;
  void send(optional (Document or BodyInit)? body = null);
  void abort();

  // response
  readonly attribute USVString responseURL;
  readonly attribute unsigned short status;
  readonly attribute ByteString statusText;
  ByteString? getResponseHeader(ByteString name);
  ByteString getAllResponseHeaders();
  void overrideMimeType(DOMString mime);
           attribute XMLHttpRequestResponseType responseType;
  readonly attribute any response;
  readonly attribute USVString responseText;
File an issue about the selected text eadonly attribute Document? responseXML;
};
typedef (File or USVString) FormDataEntryValue;

[Constructor(optional HTMLFormElement form),
 Exposed=(Window,Worker)]
interface FormData {
  void append(USVString name, USVString value);
  void append(USVString name, Blob blobValue, optional USVString filename);
  void delete(USVString name);
  FormDataEntryValue? get(USVString name);
  sequence<FormDataEntryValue> getAll(USVString name);
  boolean has(USVString name);
  void set(USVString name, USVString value);
  void set(USVString name, Blob blobValue, optional USVString filename);
  iterable<USVString, FormDataEntryValue>;
};
[Constructor(DOMString type, optional ProgressEventInit eventInitDict),
 Exposed=(Window,DedicatedWorker,SharedWorker)]
interface ProgressEvent : Event {
  readonly attribute boolean lengthComputable;
  readonly attribute unsigned long long loaded;
  readonly attribute unsigned long long total;
};

dictionary ProgressEventInit : EventInit {
  boolean lengthComputable = false;
  unsigned long long loaded = 0;
  unsigned long long total = 0;
};
```

File an issue about the selected text