Using Fetch

The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global `fetch()` method that provides an easy, logical way to fetch resources asynchronously across the network.

This kind of functionality was previously achieved using `XMLHttpRequest`. Fetch provides a better alternative that can be easily used by other technologies such as `Service Workers`. Fetch also provides a single logical place to define other HTTP-related concepts such as CORS and extensions to HTTP.

The `fetch` specification differs from `jQuery.ajax()` in two main ways:

- The Promise returned from `fetch()` **won’t reject on HTTP error status** even if the response is an HTTP 404 or 500. Instead, it will resolve normally (with `ok` status set to false), and it will only reject on network failure or if anything prevented the request from completing.
- By default, `fetch` **won't send or receive any cookies** from the server, resulting in unauthenticated requests if the site relies on maintaining a user session (to send cookies, the *credentials* init option must be set).

Since Aug 25, 2017. The spec changed the default credentials policy to `same-origin`. Firefox changed since 61.0b13.

A basic fetch request is really simple to set up. Have a look at the following code:

```
fetch('http://example.com/movies.json')
  .then(function(response) {
    return response.json();
  })
  .then(function(myJson) {
    console.log(myJson);
  });
```

Here we are fetching a JSON file across the network and print it to the console. The simplest use of `fetch()` takes one argument — the path to the resource you want to fetch — and returns a promise containing the response (a `Response` object).

This is just an HTTP response of course, not the actual JSON. To extract the JSON body content from the response, we use the `json()` method (defined on the `Body` mixin, which is implemented by both the `Request` and `Response` objects.)

Note: The Body mixin also has similar methods to extract other types of body content; see the Body section for more.

Fetch requests are controlled by the `connect-src` directive of Content Security Policy rather than the directive of the resources it's retrieving.

Supplying request options

The `fetch()` method can optionally accept a second parameter, an `init` object that allows you to control a number of different settings:

See `fetch()` for the full options available, and more details.

```
// Example POST method implementation:

postData(`http://example.com/answer`, {answer: 42})
```

```
  .then(data => console.log(data)) // JSON from `response.json()` call
  .catch(error => console.error(error));

function postData(url = ``, data = {}) {
  // Default options are marked with *
    return fetch(url, {
        method: "POST", // *GET, POST, PUT, DELETE, etc.
        mode: "cors", // no-cors, cors, *same-origin
        cache: "no-cache", // *default, no-cache, reload, force-cache, only-i
        credentials: "same-origin", // include, same-origin, *omit
        headers: {
            "Content-Type": "application/json; charset=utf-8",
            // "Content-Type": "application/x-www-form-urlencoded",
        },
        redirect: "follow", // manual, *follow, error
        referrer: "no-referrer", // no-referrer, *client
        body: JSON.stringify(data), // body data type must match "Content-Typ
    })
    .then(response => response.json()); // parses response to JSON
}
```

Sending a request with credentials included

To cause browsers to send a request with credentials included, even for a cross-origin call, add `credentials: 'include'` to the `init` object you pass to the `fetch()` method.

```
fetch('https://example.com', {
  credentials: 'include'
})
```

If you only want to send credentials if the request URL is on the same origin as the calling script, add `credentials: 'same-origin'`.

```
// The calling script is on the origin 'https://example.com'

fetch('https://example.com', {
  credentials: 'same-origin'
})
```

To instead ensure browsers don’t include credentials in the request, use `credentials: 'omit'`.

```
fetch('https://example.com', {
  credentials: 'omit'
})
```

Uploading JSON data

Use `fetch()` to POST JSON-encoded data.

```
var url = 'https://example.com/profile';
var data = {username: 'example'};

fetch(url, {
  method: 'POST', // or 'PUT'
  body: JSON.stringify(data), // data can be `string` or {object}!
  headers:{
    'Content-Type': 'application/json'
  }
}).then(res => res.json())
.catch(error => console.error('Error:', error))
.then(response => console.log('Success:', response));
```

Uploading a file

Files can be uploaded using an HTML `<input type="file" />` input element, `FormData()` and `fetch()`.

```
var formData = new FormData();
var fileField = document.querySelector("input[type='file']");

formData.append('username', 'abc123');
formData.append('avatar', fileField.files[0]);

fetch('https://example.com/profile/avatar', {
  method: 'PUT'
  method: 'PUT',
  body: formData
})
.then(response => response.json())
.catch(error => console.error('Error:', error))
.then(response => console.log('Success:', response));
```

Uploading multiple files

Files can be uploaded using an HTML `<input type="file" />` input element, `FormData()` and `fetch()`.

```
var formData = new FormData();
var photos = document.querySelector("input[type='file'][multiple]");

formData.append('title', 'My Vegas Vacation');
formData.append('photos', photos.files);

fetch('https://example.com/posts', {
  method: 'POST',
  body: formData
})
.then(response => response.json())
.then(response => console.log('Success:', response))
.catch(error => console.error('Error:', error));
```

Checking that the fetch was successful

A `fetch()` promise will reject with a `TypeError` when a network error is encountered or CORS is misconfigured on the server side, although this usually means permission issues or similar — a 404 does not constitute a network error, for example. An accurate check for a successful `fetch()` would include checking that the promise resolved, then checking that the `Response.ok` property has a value of true. The code would look something like this:

```
fetch('flowers.jpg').then(function(response) {
  if(response.ok) {
    return response.blob();
  }
  throw new Error('Network response was not ok.');
}).then(function(myBlob) {
  var objectURL = URL.createObjectURL(myBlob);
  myImage.src = objectURL;
}).catch(function(error) {
  console.log('There has been a problem with your fetch operation: ', error.m
});
```

Supplying your own request object

Instead of passing a path to the resource you want to request into the `fetch()` call, you can create a request object using the `Request()` constructor, and pass that in as a `fetch()` method argument:

```
var myHeaders = new Headers();

var myInit = { method: 'GET',
               headers: myHeaders,
               mode: 'cors',
               cache: 'default' };

var myRequest = new Request('flowers.jpg', myInit);

fetch(myRequest).then(function(response) {
  return response.blob();
}).then(function(myBlob) {
  var objectURL = URL.createObjectURL(myBlob);
  myImage.src = objectURL;
});
```

`Request()` accepts exactly the same parameters as the `fetch()` method. You can even pass in an existing request object to create a copy of it:

```
var anotherRequest = new Request(myRequest, myInit);
```

This is pretty useful, as request and response bodies are one use only. Making a copy like this allows you to make use of the request/response again, while varying the `init` options if desired. The copy must be made before the body is read, and reading the body in the copy will also mark it as read in the original request.

Note: There is also a `clone()` method that creates a copy. Both methods of creating a copy will fail if the body of the original request or response has already been read, but reading the body of a cloned response or request will not cause it to be marked as read in the original.

Headers

The `Headers` interface allows you to create your own headers object via the `Headers()` constructor. A headers object is a simple multi-map of names to values:

```
var content = "Hello World";
var myHeaders = new Headers();
myHeaders.append("Content-Type", "text/plain");
myHeaders.append("Content-Length", content.length.toString());
myHeaders.append("X-Custom-Header", "ProcessThisImmediately");
```

The same can be achieved by passing an array of arrays or an object literal to the constructor:

```
myHeaders = new Headers({
  "Content-Type": "text/plain",
  "Content-Length": content.length.toString(),
  "X-Custom-Header": "ProcessThisImmediately",
});
```

The contents can be queried and retrieved:

```
console.log(myHeaders.has("Content-Type")); // true
console.log(myHeaders.has("Set-Cookie")); // false
myHeaders.set("Content-Type", "text/html");
myHeaders.append("X-Custom-Header", "AnotherValue");

console.log(myHeaders.get("Content-Length")); // 11
console.log(myHeaders.get("X-Custom-Header")); // ["ProcessThisImmediately",

myHeaders.delete("X-Custom-Header");
console.log(myHeaders.get("X-Custom-Header")); // [ ]
```

Some of these operations are only useful in `ServiceWorkers`, but they provide a much nicer API for manipulating headers.

All of the Headers methods throw a `TypeError` if a header name is used that is not a valid HTTP Header name. The mutation operations will throw a `TypeError` if there is an immutable guard (see below). Otherwise they fail silently. For example:

```
var myResponse = Response.error();
try {
  myResponse.headers.set("Origin", "http://mybank.com");
} catch(e) {
  console.log("Cannot pretend to be a bank!");
}
```

A good use case for headers is checking whether the content type is correct before you process it further. For example:

```
fetch(myRequest).then(function(response) {
    var contentType = response.headers.get("content-type");
    if(contentType && contentType.includes("application/json")) {
      return response.json();
    }
    throw new TypeError("Oops, we haven't got JSON!");
  })
  .then(function(json) { /* process your JSON further */ })
  .catch(function(error) { console.log(error); });
```

Guard

Since headers can be sent in requests and received in responses, and have various limitations about what information can and should be mutable, headers objects have a guard property. This is not exposed to the Web, but it affects which mutation operations are allowed on the headers object.

Possible guard values are:

- `none`: default.
- `request`: guard for a headers object obtained from a request (`Request.headers`).
- `request-no-cors`: guard for a headers object obtained from a request created with `Request.mode` `no-cors`.
- `response`: guard for a Headers obtained from a response (`Response.headers`).
- `immutable`: Mostly used for ServiceWorkers; renders a headers object read-only.

Note: You may not append or set a `request` guarded Headers' `Content-Length` header. Similarly, inserting `Set-Cookie` into a response header is not allowed: ServiceWorkers are not allowed to set cookies via synthesized responses.

Response objects

As you have seen above, `Response` instances are returned when `fetch()` promises are resolved.

The most common response properties you'll use are:

- `Response.status` — An integer (default value 200) containing the response status code.
- `Response.statusText` — A string (default value "OK"), which corresponds to the HTTP status code message.
- `Response.ok` — seen in use above, this is a shorthand for checking that status is in the range 200-299 inclusive. This returns a `Boolean`.

They can also be created programmatically via JavaScript, but this is only really useful in `ServiceWorkers`, when you are providing a custom response to a received request using a `respondWith()` method:

```
var myBody = new Blob();

addEventListener('fetch', function(event) { // ServiceWorker intercepting a f
  event.respondWith(
    new Response(myBody, {
      headers: { "Content-Type" : "text/plain" }
    })
  );
});
```

The `Response()` constructor takes two optional arguments — a body for the response, and an init object (similar to the one that `Request()` accepts.)

Note: The static method `error()` simply returns an error response. Similarly, `redirect()` returns a response resulting in a redirect to a specified URL. These are also only relevant to Service Workers.

Body

Both requests and responses may contain body data. A body is an instance of any of the following types:

- `ArrayBuffer`
- `ArrayBufferView` (Uint8Array and friends)
- `Blob`/File
- string
- `URLSearchParams`
- `FormData`

The `Body` mixin defines the following methods to extract a body (implemented by both `Request` and `Response`). These all return a promise that is eventually resolved with the actual content.

- `arrayBuffer()`
- `blob()`
- `json()`
- `text()`
- `formData()`

This makes usage of non-textual data much easier than it was with XHR.

Request bodies can be set by passing body parameters:

```
var form = new FormData(document.getElementById('login-form'));
fetch("/login", {
  method: "POST",
  body: form
});
```

Both request and response (and by extension the `fetch()` function), will try to intelligently determine the content type. A request will also automatically set a `Content-Type` header if none is set in the dictionary.

Feature detection

Fetch API support can be detected by checking for the existence of `Headers`, `Request`, `Response` or `fetch()` on the `Window` or `Worker` scope. For example:

```
if (self.fetch) {
    // run my fetch request here
} else {

    // do something with XMLHttpRequest?
}
```

Polyfill

To use Fetch in unsupported browsers, there is a Fetch Polyfill available that recreates the functionality for non-supporting browsers.

Specifications

Specification	Status	Comment
Fetch	LS Living Standard	Initial definition

Browser compatibility

We're converting our compatibility data into a machine-readable JSON format. This compatibility table still uses the old format, because we haven't yet converted the data it contains. **Find out how you can help!**

Desktop Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Basic support	42	14	39 (39) 34 (34)[1] 52 (52)[2]	No support	29 28[1]	10.1

[1] This API is implemented behind a preference.

[2] Prior to Firefox 52, `get()` only returned the first value in the specified header, with `getAll()` returning all values. From 52 onwards, `get()` now returns all values and `getAll()` has been deleted.

See also

- ServiceWorker API
- HTTP access control (CORS)
- HTTP
- Fetch polyfill
- Fetch examples on Github