Link - <https://davidwalsh.name/fetch>

One of the worst kept secrets about AJAX on the web is that the underlying API for it, XMLHttpRequest, wasn't really made for what we've been using it for.  We've done well to create elegant APIs around XHR but we know we can do better.  Our effort to do better is the fetch API.  Let's have a basic look at the new window.fetch method, available now in Firefox and Chrome Canary.

## XMLHttpRequest

[XHR](https://davidwalsh.name/xmlhttprequest) is a bit overcomplicated in my opinion, and don't get me started on why "XML" is uppercase but "Http" is camel-cased.  Anyways, this is how you use XHR now:

// Just getting XHR is a mess!

if (window.XMLHttpRequest) { // Mozilla, Safari, ...

request = new XMLHttpRequest();

} else if (window.ActiveXObject) { // IE

try {

request = new ActiveXObject('Msxml2.XMLHTTP');

}

catch (e) {

try {

request = new ActiveXObject('Microsoft.XMLHTTP');

}

catch (e) {}

}

}

// Open, send.

request.open('GET', 'https://davidwalsh.name/ajax-endpoint', true);

request.send(null);

Of course our JavaScript frameworks make XHR more pleasant to work with, but what you see above is a simple example of the XHR mess.

## Basic fetch Usage

A fetch function is now provided in the global window scope, with the first argument being the URL:

// url (required), options (optional)

fetch('https://davidwalsh.name/some/url', {

method: 'get'

}).then(function(response) {

}).catch(function(err) {

// Error :(

});

Much like the updated [Battery API](https://davidwalsh.name/javascript-battery-api), the fetch API uses [JavaScript Promises](https://davidwalsh.name/promises) to handle results/callbacks:

// Simple response handling

fetch('https://davidwalsh.name/some/url').then(function(response) {

}).catch(function(err) {

// Error :(

});

// Chaining for more "advanced" handling

fetch('https://davidwalsh.name/some/url').then(function(response) {

return //...

}).then(function(returnedValue) {

// ...

}).catch(function(err) {

// Error :(

});

If you aren't used to then yet, get used to it -- it will soon be everywhere.

## Request Headers

The ability to set request headers is important in request flexibility. You can work with request headers by executing new Headers():

// Create an empty Headers instance

var headers = new Headers();

// Add a few headers

headers.append('Content-Type', 'text/plain');

headers.append('X-My-Custom-Header', 'CustomValue');

// Check, get, and set header values

headers.has('Content-Type'); // true

headers.get('Content-Type'); // "text/plain"

headers.set('Content-Type', 'application/json');

// Delete a header

headers.delete('X-My-Custom-Header');

// Add initial values

var headers = new Headers({

'Content-Type': 'text/plain',

'X-My-Custom-Header': 'CustomValue'

});

You can use the append, has, get, set, and delete methods to modify request headers. To use request headers, create a Request instance :

var request = new Request('https://davidwalsh.name/some-url', {

headers: new Headers({

'Content-Type': 'text/plain'

})

});

fetch(request).then(function() { /\* handle response \*/ });

Let's have a look at what Response and Request do!

## Request

A Request instance represents the request piece of a fetch call. By passing fetch aRequest you can make advanced and customized requests:

* method - GET, POST, PUT, DELETE, HEAD
* url - URL of the request
* headers - associated Headers object
* referrer - referrer of the request
* mode - cors, no-cors, same-origin
* credentials - should cookies go with the request? omit, same-origin
* redirect - follow, error, manual
* integrity - subresource integrity value
* cache - cache mode (default, reload, no-cache)

A sample Request usage may look like:

var request = new Request('https://davidwalsh.name/users.json', {

method: 'POST',

mode: 'cors',

redirect: 'follow',

headers: new Headers({

'Content-Type': 'text/plain'

})

});

// Now use it!

fetch(request).then(function() { /\* handle response \*/ });

Only the first parameter, the URL, is required. Each property becomes read only once the Request instance has been created. Also important to note that Request has a clonemethod which is important when using fetch within the Service Worker API -- a Request is a stream and thus must be cloned when passing to another fetch call.

The fetch signature, however, acts like Request so you could also do:

fetch('https://davidwalsh.name/users.json', {

method: 'POST',

mode: 'cors',

redirect: 'follow',

headers: new Headers({

'Content-Type': 'text/plain'

})

}).then(function() { /\* handle response \*/ });

You'll likely only use Request instances within Service Workers since the Request and fetch signatures can be the same. ServiceWorker post coming soon!

## Response

The fetch's then method is provided a Response instance but you can also manually create Response objects yourself -- another situation you may encounter when using service workers. With a Response you can configure:

* type - basic, cors
* url
* useFinalURL - Boolean for if url is the final URL
* status - status code (ex: 200, 404, etc.)
* ok - Boolean for successful response (status in the range 200-299)
* statusText - status code (ex: OK)
* headers - Headers object associated with the response.

// Create your own response for service worker testing

// new Response(BODY, OPTIONS)

var response = new Response('.....', {

ok: false,

status: 404,

url: '/'

});

// The fetch's `then` gets a Response instance back

fetch('https://davidwalsh.name/')

.then(function(responseObj) {

console.log('status: ', responseObj.status);

});

The Response also provides the following methods:

* clone() - Creates a clone of a Response object.
* error() - Returns a new Response object associated with a network error.
* redirect() - Creates a new response with a different URL.
* arrayBuffer() - Returns a promise that resolves with an ArrayBuffer.
* blob() - Returns a promise that resolves with a Blob.
* formData() - Returns a promise that resolves with a FormData object.
* json() - Returns a promise that resolves with a JSON object.
* text() - Returns a promise that resolves with a USVString (text).

## Handling JSON

Let's say you make a request for JSON -- the resulting callback data has a json method for converting the raw data to a JavaScript object:

fetch('https://davidwalsh.name/demo/arsenal.json').then(function(response) {

// Convert to JSON

return response.json();

}).then(function(j) {

// Yay, `j` is a JavaScript object

console.log(j);

});

Of course that's a simple JSON.parse(jsonString), but the json method is a handy shortcut.

## Handling Basic Text/HTML Responses

JSON isn't always the desired request response format so here's how you can work with an HTML or text response:

fetch('/next/page')

.then(function(response) {

return response.text();

}).then(function(text) {

// <!DOCTYPE ....

console.log(text);

});

You can get the response text via chaining the Promise's then method along with the text() method.

## Handling Blob Responses

If you want to load an image via fetch, for example, that will be a bit different:

fetch('https://davidwalsh.name/flowers.jpg')

.then(function(response) {

return response.blob();

})

.then(function(imageBlob) {

document.querySelector('img').src = URL.createObjectURL(imageBlob);

});

The blob() method of the Body mixin takes a Response stream and reads it to completion.

## Posting Form Data

Another common use case for AJAX is sending form data -- here's how you would use fetchto post form data:

fetch('https://davidwalsh.name/submit', {

method: 'post',

body: new FormData(document.getElementById('comment-form'))

});

And if you want to POST JSON to the server:

fetch('https://davidwalsh.name/submit-json', {

method: 'post',

body: JSON.stringify({

email: document.getElementById('email').value,

answer: document.getElementById('answer').value

})

});

Very easy, very eye-pleasing as well!

## Unwritten Story

While fetch is a nicer API to use, the API current doesn't allow for canceling a request, which makes it a non-starter for many developers.

The new fetch API seems much saner and simpler to use than XHR.  After all, it was created so that we could do AJAX the right way; fetch has the advantage of hindsight.  I can't wait until fetch is more broadly supported!