Introduction to fetch()

By Matt Gaunt (/web/resources/contributors/mattgaunt)
Matt is a contributor to Web**Fundamentals**

So long XMLHttpRequest

`fetch()` allows you to make network requests similar to XMLHttpRequest (XHR). The main difference is that the Fetch API uses Promises, which enables a simpler and cleaner API, avoiding callback hell and having to remember the complex API of XMLHttpRequest.

The Fetch API (https://fetch.spec.whatwg.org/) has been available in the Service Worker (https://w3c.github.io/ServiceWorker/) global scope since Chrome 40, but it'll be enabled in the window scope in Chrome 42. There is also a rather fetching polyfill by GitHub (https://github.com/github/fetch) that you can use today.

If you've never used Promises (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) before, check out Introduction to JavaScript Promises (/web/fundamentals/getting-started/primers/promises).

Basic Fetch Request

Let's start by comparing a simple example implemented with an `XMLHttpRequest` and then with `fetch`. We just want to request a URL, get a response and parse it as JSON.

XMLHttpRequest

An `XMLHttpRequest` would need two listeners to be set to handle the success and error cases and a call to `open()` and `send()`. *Example from MDN docs* (https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest).

```
function reqListener() {
  var data = JSON.parse(this.responseText);
  console.log(data);
}

function reqError(err) {
  console.log('Fetch Error :-S', err);
}

var oReq = new XMLHttpRequest();
oReq.onload = reqListener;
oReq.onerror = reqError;
oReq.open('get', './api/some.json', true);
oReq.send();
```

Fetch

Our fetch request looks a little like this:

```
fetch('./api/some.json')
  .then(
    function(response) {
      if (response.status !== 200) {
        console.log('Looks like there was a problem. Status Code: ' +
          response.status);
        return;
      }

      // Examine the text in the response
      response.json().then(function(data) {
        console.log(data);
      });
    }
  )
  .catch(function(err) {
    console.log('Fetch Error :-S', err);
  });
```

We start by checking that the response status is 200 before parsing the response as JSON.

The response of a `fetch()` request is a Stream (https://streams.spec.whatwg.org/) object, which means that when we call the `json()` method, a Promise is returned since the reading

of the stream will happen asynchronously.

Response Metadata

In the previous example we looked at the status of the Response
(https://fetch.spec.whatwg.org/#responses) object as well as how to parse the response as
JSON. Other metadata we may want to access, like headers, are illustrated below.

```
fetch('users.json').then(function(response) {
    console.log(response.headers.get('Content-Type'));
    console.log(response.headers.get('Date'));

    console.log(response.status);
    console.log(response.statusText);
    console.log(response.type);
    console.log(response.url);
});
```

Response Types

When we make a fetch request, the response will be given a `response.type` of "basic
(https://fetch.spec.whatwg.org/#concept-filtered-response-basic)", "cors
(https://fetch.spec.whatwg.org/#concept-filtered-response-cors)" or "opaque
(https://fetch.spec.whatwg.org/#concept-filtered-response-opaque)". These `types` indicate where
the resource has come from and can be used to inform how you should treat the response
object.

When a request is made for a resource on the same origin, the response will have a `basic`
type and there aren't any restrictions on what you can view from the response.

If a request is made for a resource on another origin which returns the CORs headers
(https://enable-cors.org/), then the type is `cors`. `cors` and `basic` responses are almost
identical except that a `cors` response restricts the headers you can view to `Cache-Control`,
`Content-Language`, `Content-Type`, `Expires`, `Last-Modified`, and `Pragma`.

An `opaque` response is for a request made for a resource on a different origin that doesn't
return CORS headers. With an opaque response we won't be able to read the data returned

or view the status of the request, meaning we can't check if the request was successful or not.

You can define a mode for a fetch request such that only certain requests will resolve. The modes you can set are as follows:

- `same-origin` only succeeds for requests for assets on the same origin, all other requests will reject.

- `cors` will allow requests for assets on the same-origin and other origins which return the appropriate CORs headers.

- `cors-with-forced-preflight` will always perform a [preflight check (https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#Preflighted_requests)](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#Preflighted_requests) before making the actual request.

- `no-cors` is intended to make requests to other origins that do not have CORS headers and result in an `opaque` response, but as stated, this isn't possible in the window global scope at the moment.

To define the mode, add an options object as the second parameter in the `fetch` request and define the mode in that object:

```
fetch('http://some-site.com/cors-enabled/some.json', {mode: 'cors'})
  .then(function(response) {
    return response.text();
  })
  .then(function(text) {
    console.log('Request successful', text);
  })
  .catch(function(error) {
    log('Request failed', error)
  });
```

Chaining Promises

One of the great features of promises is the ability to chain them together. For fetch, this allows you to share logic across fetch requests.

If you are working with a JSON API, you'll need to check the status and parse the JSON for each response. You can simplify your code by defining the status and JSON parsing in

separate functions which return promises, freeing you to only worry about handling the final data and the error case.

```
function status(response) {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response)
  } else {
    return Promise.reject(new Error(response.statusText))
  }
}

function json(response) {
  return response.json()
}

fetch('users.json')
  .then(status)
  .then(json)
  .then(function(data) {
    console.log('Request succeeded with JSON response', data);
  }).catch(function(error) {
    console.log('Request failed', error);
  });
```

We define the `status` function which checks the **response.status** and returns the result of
Promise.resolve()
 (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve)
or Promise.reject()
 (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject),
which return a resolved or rejected Promise. This is the first method called in our `fetch()`
chain, if it resolves, we then call our `json()` method which again returns a Promise from the
`response.json()` call. After this we have an object of the parsed JSON. If the parsing fails
the Promise is rejected and the catch statement executes.

The great thing with this is that you can share the logic across all of your fetch requests, making code easier to maintain, read and test.

POST Request

It's not uncommon for web apps to want to call an API with a POST method and supply some parameters in the body of the request.

To do this we can set the `method` and `body` parameters in the `fetch()` options.

```
fetch(url, {
    method: 'post',
    headers: {
      "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"
    },
    body: 'foo=bar&lorem=ipsum'
})
.then(json)
.then(function (data) {
    console.log('Request succeeded with JSON response', data);
})
.catch(function (error) {
    console.log('Request failed', error);
});
```

Sending Credentials with a Fetch Request

Should you want to make a fetch request with credentials such as cookies, you should set the `credentials` of the request to `"include"`.

```
fetch(url, {
  credentials: 'include'
})
```

FAQ

How do I cancel a fetch() request?

At the moment there is no way to cancel a fetch, but this is being discussed on GitHub (https://github.com/whatwg/fetch/issues/20). H/T @jaffathecake (https://twitter.com/jaffathecake) for this link.

Is there a polyfill?

GitHub has a polyfill for fetch (https://github.com/github/fetch). H/T @Nexii
 (https://twitter.com/Nexii) for pointing this out.

Why is "no-cors" supported in service workers but not the window?

This is due to a security concern, you can learn more here
 (https://bugs.chromium.org/p/chromium/issues/detail?id=457157&q=fetch%20no-cors&colspec=ID%20Pri%20M%20Week%20ReleaseBlock%20Cr%20Status%20Owner%20Summary%20OS%20Modified)
.

Last updated 2019-03-20 UTC.