codeinwp

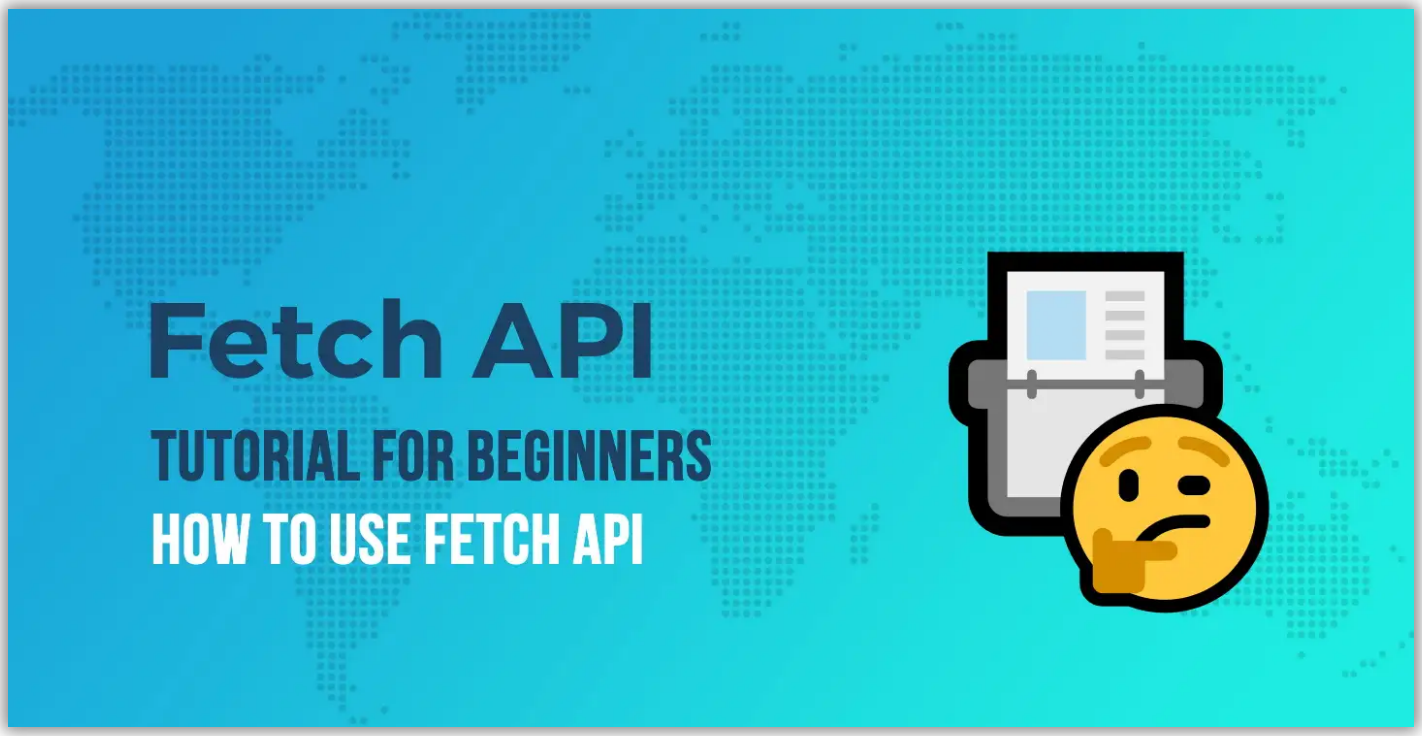
Fetch API Tutorial for Beginners: How to Use Fetch API

by Louis Lazaris / February 2, 2023 / web design & development

If you're looking for a gentle introduction to using the Fetch API, a modern replacement for XMLHttpRequest-based Ajax-driven pages, this is the place to start!

Today it's safe to say that most web apps (including PWAs) make requests for resources on-the-fly, or dynamically. This is often referred to as *asynchronous* resource loading. This is different from resources that load when the page initially loads. Asynchronous resources are requested on demand in specific circumstances and don't require a full page load.

As mentioned, in most cases, this type of loading has been done via Ajax, which utilizes a technology called `XMLHttpRequest` to make asynchronous resource calls. In this tutorial, I'm going to go through how to use the Fetch API (official), which now has excellent browser support and is slowly starting to replace XMLHttpRequest.



CLICK TO TWEET

Basic syntax for a Fetch API request

To get started, let's look at a simple Fetch API example so you can start to get familiar with the basic syntax:

```
fetch(url)
  .then((response) => {
    return response.text();
  })
  .then((data) => {
    // do something with 'data'
  });
```

The first line in that code uses the `fetch()` method, which is a method of the `Window` object in a browser environment. So I could write the first line like this instead:

```
window.fetch(url)
```

Fetch can also be used in other environments, but for the purposes of this Fetch API tutorial, I'm going to focus on using the Fetch API in the browser.

The `fetch()` method takes two arguments, but only one is mandatory: the location of the requested resource. In the example above, I'm assuming somewhere else in my code I've defined that resource's location in a variable called `url`.

Below is a CodePen demo that uses the Fetch API interactively to request an external file and display its contents on the page:

Use the button in the demo to display the content.

Notice the resource that I'm loading in the CodePen demo:

```
fetch('https://codepen.io/impressivewebs/pen/KKVopdL.html')
```

That's a separate CodePen file that I created. CodePen allows you to append .html, .css, or .js to any CodePen URL to request that pen's HTML, CSS, or JavaScript content, which is useful when using Ajax or Fetch API requests so you're not hampered by cross-origin limitations. So that's pretty useful! Try changing the URL to request the CSS or JavaScript (i.e. using KKVopdL.css or KKVopdL.js at the end of the URL) to see the results.

Now that you've seen a basic Fetch API example, I'll introduce you to the different parts of that request.

The resource location parameter

As already discussed, the only argument that's required when using the `fetch()` method is the *resource*. This will usually be a direct URL to the resource being requested, but it can also be a

Let's look at some examples using a direct URL to various free APIs as the requested resources (i.e. the resources I want to 'fetch').

Here's a request for some basketball data:

```
fetch('https://www.balldontlie.io/api/v1/teams/28')
```

Here's a request for JavaScript jobs in the New York area:

```
fetch('https://jobs.github.com/positions.json?description=javascript&locatio
```

And here's a request for a random beer:

```
fetch('https://api.punkapi.com/v2/beers/random')
```

Any of those URLs can be viewed directly in your browser and they all involve resources from various free APIs. You can try out other free APIs by visiting this GitHub repo. Just make sure you choose one that doesn't require authentication (i.e. it has "No" under the "Auth" column).

The returned Promises in a Fetch API request

Every `fetch()` method begins the process of requesting a resource and returns a Promise. A detailed discussion of JavaScript Promises is out of the scope of this tutorial, but I'll cover enough for you to be able to work with `fetch()`.

According to MDN, a Promise is an object that:

> [...] represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

> Do something. When that “something” is finished, do something else.

Of course, that’s an oversimplification, but the concept is more or less there. You’re basically ‘promising’ something then you’re responding when that ‘something’ is completed. And because this is done asynchronously, you can run other lines of JavaScript while the Promise is in the process of being fulfilled.

In the code I used earlier, there are two Promises being returned, one for each `.then()` method call:

```
.then((response) => {
  return response.text();
})
.then((data) => {
  // do something with 'data'
});
```

These `.then()` methods are doing what’s referred to as *chaining*, which is possible and quite common with Promises. As MDN explains:

> If the function passed as handler to `then` returns a `Promise`, an equivalent `Promise` will be exposed to the subsequent `then` in the method chain.

So to summarize that example code: I’m ‘fetching’ a resource; ‘then’ I’m returning a response; ‘then’ I’m returning the data after reading the response.

The Promise response in a Fetch API

Notice again in the example code the following syntax:

```
.then((response) => { }
```

the arrow function that's passed as the argument. This arrow function, in turn, is passing in what's referred to as the Response object.

When the Promise is fulfilled, the Response object is what's returned, and this is passed into the function from where it can be handled in a number of ways, which I'll discuss next.

Useful properties & methods of the response object

There are a number of different properties and methods available on the response object once it's received from the `fetch()` request. Here are a few useful ones:

- `Response.ok` – This property returns a Boolean indicating whether the request was successful.
- `Response.status` – This property returns the status code of the response (e.g. `200` for a successful request).
- `Response.url` – This property returns the URL of the request. Normally you'd have this when initially sending the request, but if you're creating the request dynamically, this might be useful.

View a full list of properties in this MDN article.

There are also some useful methods you'll want to be familiar with:

- `Response.clone()` – Creates a clone of the `Response` object
- `Response.text()` – Returns the data in the response as text, which is useful when retrieving HTML
- `Response.json()` – Returns the data in the response as a valid JSON object, which is useful when retrieving a JSON string.
- `Response.blob()` – Returns the data as a blob, which is useful for data that's in the form of a file reference (like an image URL).

Again, you can view a full list of valid methods in this MDN article.

Reading the response

You'll notice in my example that I'm including two chained `.then()` methods when dealing with the content being requested. The first `.then()` is where I'm dealing with the response object and the second `.then()` is where I'm dealing with the data after it's been read (usually via one of the methods mentioned in the previous section).

Promise is fulfilled, I'm able to manipulate the data however I want.

This is similar to how Ajax calls work. Whether I'm working with Ajax's `XMLHttpRequest` or the Fetch API, once I've received the data in a usable format on the current page, I can do with it what I like.

When is a Fetch API request considered rejected?

An important distinction to make when looking at successful and unsuccessful Fetch API requests is when considering network errors vs. HTTP errors.

When making a `fetch()` request, if permission is not granted to the document that initiated the fetch, then this is considered a rejected request. This occurs at the network level. However, if a fetch request is asking for content that doesn't exist on the server, this is still considered a successful request, so the Promise is viewed as successfully fulfilled.

To illustrate, take a look at the following CodePen:

This is the same code that I used earlier, but this time instead of requesting another CodePen document (which I'm allowed to do on CodePen), I'm attempting to request the CodeinWP

Now notice the different error message that appears when viewing the following demo:

Fetch API 404 Demo

Open up the browser console to see the messages displayed. Here's the JavaScript I'm using for that demo:

```
fetch('https://www.impressivewebs.com/page-doesnt-exist')
  .then((response) => {
    console.log('Promise was fulfilled');
    return response.status;
  })
  .then((status) => {
    console.log('Promise 2 was fulfilled');
    document.querySelector('.par').innerHTML = status;
    document.querySelector('.par').classList.add('box');
  });
```

The resource I'm requesting doesn't exist, but I have access to the server I'm requesting it from since it's on the same origin. To demonstrate that the Promises are fulfilled, I'm displaying a console message for each one. But you'll notice the browser console also displays a `GET` error message and the `Response` object returns a value of `404` for the `status` property. Similarly, if I logged `Response.ok` I would get a generic value of `false`, rather than a specific status message.

Knowing how this works, you can deal with situations where the content isn't found, for example by providing a custom error message inside the first `.then()` method:

```
.then((response) => {
  if (!response.ok) {
    throw new Error(`HTTP error! status: ${response.status}`);
  }
})
```

As mentioned, the only mandatory argument when using `fetch()` is the location of the resource, usually a string value in the form of a relative or absolute URL. But `fetch()` also allows you to provide a second optional argument, the `init` object.

Here's an example that includes some possible settings inside `init`:

```
fetch(url, {
  method: 'GET',
  mode: 'no-cors',
  cache: 'no-cache',
  referrerPolicy: 'same-origin'
})
```

You can see a full list of settings for the optional `init` parameter in the Syntax section of the MDN article on `fetch()`.

The ones I've included above are a few of the more common ones. For example, `method` lets me define whether the request is a `GET` or `POST` (it will usually be a `GET`, which is the default). The `mode` allows me to define the type of request. I could set this to "no-cors" which would allow me to request a resource from anywhere. But "no-cors" means I'm limited as to what I can do with the content. For example, I can't access it with JavaScript but it can be used in a Service Worker environment. Finally, I can define a referrer policy and the `cache` setting lets me define a cache mode for the requested resource.

These are just a few examples of the many settings available, but not all are going to be useful to you in most examples when using the Fetch API. You'll likely only work with one or two of these, or you you'll leave the `init` parameter out and work with the defaults.

Requesting JSON data via the Fetch API

As mentioned earlier, there are a number of different services that allow you to request data via their APIs. One such service is called Dog API, that claims to be "the internet's biggest collection of open source dog pictures". I'm going to use that API to demonstrate obtaining JSON data via the Fetch API.

Here's the code:

```
  .then((response) => {
    return response.json();
  })
  .then((myContent) => {
    myImage.src = myContent['message'];
  });
```

Notice the first line is the URL of the request. As mentioned on the Dog API documentation, you can use their API to grab a random dog image. The response that's returned is in JSON format that looks like this:

```
{
  "message": "https://images.dog.ceo/breeds/boxer/IMG_3394.jpg",
  "status": "success"
}
```

I'm only concered with the URL found in the "message" property, which is why I have the following line in my code:

```
myImage.src = myContent['message'];
```

The `myContent` variable is passed into the second `.then()` method, which happens after the response is converted to JSON format using the `.json()` method that I mentioned earlier. As long as I have access to the data in valid JSON format, from there I can do anything I would normally be able to do with JSON in JavaScript.

Here's a live CodePen demo so you can try it out:

Use the button in the demo to request a random dog image that gets displayed on the page. This uses an event listener to change the image each time the button is clicked.

Browser support shouldn't be a big problem in most cases, but if you still need to support Fetch in older browsers, there are some workarounds and polyfills.

You can check for the existence of Fetch using something like the following code:

```
if (!('fetch' in window)) {
  // fetch not available, use a polyfill
} else {
  // fetch is available
}
```

And here are a couple of links to Fetch API polyfills:

- unfetch – This is a minimal polyfill that's under 500 bytes that works in IE8+
- window.fetch polyfill – Implements a subset of the standard Fetch specification

Keep in mind that if you're bundling your app using something like Parcel.js along with its built-in support for Babel, you'll have to include a polyfill separately if your Babel settings go back to old version of IE or another browser that doesn't support the Fetch API.

Go to top

Conclusion

There's a lot more that could be covered when it comes to using the Fetch API, and I'm still researching the various features. But this should give you a nice starting point.

I hope the examples in this post have been useful to you to get a decent overall understanding of how the Fetch API works and how it can be useful in dynamic apps in place of the old Ajax syntax.

Check out more of our web development tutorials:

👉 Webpack tutorial for beginners
👉 CSS Grid tutorial for beginners
👉 CSS Flexbox tutorial for beginners
👉 Micro-Interactions Tutorial for Beginner Developers

codeinwp

CLICK TO TWEET

...

Don't forget to join our crash course on speeding up your WordPress site. With some simple fixes, you can reduce your loading time by even 50-80%:

Layout and presentation by Karol K.

Was this article helpful? Yes No

SHOW COMMENTS

Or start the conversation in our Facebook group for WordPress professionals. Find answers, share tips, and get help from other WordPress experts. Join now (it's free)!

RELATED ARTICLES

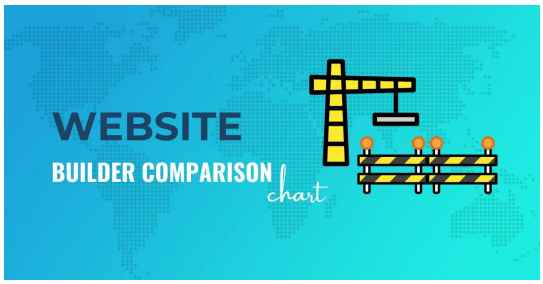
15 Best Elementor Templates for April 2023

codeinwp

10+ Best Themes for Thrive Architect in 2023 (Full Breakdown)

Best Website Design Software on the Market (15 Options Analyzed)

10+ Best Laravel Admin Templates for 2023 (Free and Premium)



Website Builder Comparison Chart: 7 Best Website Building Tools and ...

10 Best Free Shopify Themes of 2023 – Beautiful & ...

Wix Tutorial: A Step-by-Step Guide for Beginners on How to ...

codeinwp

Squarespace vs WordPress: Which is Best for Making a Website ...

10+ Best HTML/CSS Books for Beginners and Advanced Coders

FEATURED ON

YAHOO! WOOCOMMERCE CNBC Forbes Entrepreneur MAGAZINE

CodeinWP stands for all-things-WordPress. From web design to freelancing and from development to business, your questions are covered.

OUR NETWORK

EDITORS' PICKS