The fetch API allows us to create an HTTP request so we accomplish a number of things in JavaScript — like retrieving data from an API, posting data to a server, or even just get the entire contents of a web page. This HTTP request will asynchronously retrieve data from the URL provided, and generate some kind of HTTP response. Let's look at how it works.

The fetch() function is a global function, and it is most frequently used to interact with APIs. If you're new to it, you're not alone - so let's take a look at how fetch() works.

**Using fetch in JavaScript**

The most basic use of fetch takes one argument — the URL we want to fetch. Since fetch generates HTTP requests, we always have to provide a URL:

let fetchExample = fetch("https://fjolt.com").then((res) => {  
 // Do something with res  
});

Since the result of a fetch is asynchronous, we can use then() to catch the response, and do something with it. The cool thing about the returned res or response is it has a bunch of built in methods, which let us immediately parse the contents we get from fetch:

* res.text() - returns the text content of a URL. If it's a website, it returns the HTML.
* res.json() - returns formatted JSON data, if it exists.
* res.blob() - returns blob data, if any exists.
* res.arrayBuffer() - returns arrayBuffer data, if any exists.
* res.formData() - returns formData data, if any exists.

Since different URLs produce different types of content, the above methods allow us to parse that content in any way we like. To understand how that all works, let’s look at two very common examples.

**Example 1: Get HTML content of a website using JavaScript fetch**

As mentioned above, res.text() gives us the text content of a URL - so we can use it to get the entire HTML content of a URL. Once we catch our response using res.text(), we can catch the response with another then, allowing us to download and return the content of the URL provided:

let websiteData = fetch("https://fjolt.com").then(res => res.text()).then((data) => {  
 return data;  
});   
// Now contains our website's HTML.

If the link doesn’t exist, or an error occurs, our response object will contain an error. For example, a page not found will return 404, or a bad gateway error will return 502.

**Example 2: Get JSON via JavaScript Fetch**

If a URL’s contents consist of JSON, we can use the res.json(). The following code, for example, will return a JSON object from the URL, assuming the URL is sending valid JSON back:

let apiResponse = fetch("https://fjolt.com/api").then(res => res.json()).then((data) => {  
 return data;  
});  
// Now contains a JSON object - assuming one exists

**Options for JavaScript Fetch**

It’s also important to understand the options available in fetch,. They come after the URL, as an object — i.e. fetch(URL, { options }). If you've worked with HTTP requests before, some may be familiar. The fetch function displayed below contains all possible of the possible options you can use:

fetch("https://fjolt.com/", {  
 body: JSON.stringify({ someData: "value" })  
 method: 'POST'  
 mode: 'cors'  
 cache: 'no-cache'  
 credentials: 'same-origin'  
 headers: {  
 'Content-Type': 'application/json'  
 },  
 redirect: 'follow'  
 referrerPolicy: 'no-referrer'  
});

And here is a summary of what each of these mean:

* body contains the body of the text. In this example, we are sending some JSON, which needs to be stringified.
* method is a standard HTTP method. It can be POST/GET/DELETE/PUT/CONNECT/PATCH/TRACE/OPTIONS.
* mode refers to if cross origin requests are accepted. It can be cors/no-cors/same-origin.
* cache refers to how the browser will interact with the cache. It can be default/no-cache/reload/force-cache/only-if-cached.
* credentials refers to if cross origin cookies should be sent with the request. It can be include/same-origin/omit.
* headers contains any header associated with the request. It can contain any HTTP header - for example, here it shows Content-Type - but you can have custom HTTP headers too.
* redirect determines what happens if the fetched URL redirects. It can be follow/error/manual.
* referrerPolicy determines how much referrer information is passed with the request. It can be no-referrer/no-referrer-when-downgrade/origin/origin-when-cross-origin/same-origin/strict-origin/strict-origin-when-cross-origin/unsafe-url.

**Remember, JavaScript fetch is asynchronous**

When we use fetch, it goes to the URL, gathers the information, and provides a response back to us. This is not immediate, since loading the URL, downloading it and bringing it back takes time. If we simply run fetch alone, a console log immediately after it will only return a Promise, not the response from the URL we want:

let apiResponse = fetch("https://fjolt.com/api");console.log(apiResponse); // Returns Promise<Pending>

This happens because the fetch() function runs, but JavaScript doesn't wait for the response. As such, we have to explicitly tell JavaScript to wait for it, if we want to access the response.

There are two ways to wait for fetch():

* We can use then, and manipulate the response of our fetch() within then().
* We can use await, and wait for the fetch to return before using its contents.

**Using then to wait for a fetch in JavaScript**

Using then is frequently used to catch and process responses from fetch. The contents of fetch() can be manipulated within the then() callback function, but not outside of it. For example:

let apiResponse = fetch("https://fjolt.com/api").then(res => res.json()).then((data) => {  
 console.log(data);  
 // We can do anything with the data from our api here.   
 return data;  
});console.log(apiResponse); // This will return Promise<Pending>  
 // That means we can't use the apiResponse variable  
 // outside of the then() function.

If we want to use the contents from fetch() outside of then, we have to use await.

**Using await to wait for a fetch in JavaScript**

The other way to wait for a fetch is to use the await keyword. Most modern browsers support [Top level awaits](https://caniuse.com/?search=top%20level%20await), but if you are concerned about support, or using a version of Node.JS before 14.8, you'll want to wrap any await code within an async function.

If we use await, we can use the response from our API anywhere in our function or code, and use any response functions, like text() or json() on it. For example:

// Typically we wrap await in an async function  
// But most modern browsers and Node.JS support  
// await statements outside of async functions now.  
async getAPI() {  
 let apiResponse = await fetch("https://fjolt.com/api");  
 let response = apiResponse.json();  
 // Since we waited for our API to respond using await  
 // The response variable will return the response from the API  
 // And not a promise.  
 console.log(response);  
}getAPI();

If you want to learn more about async operations, [read our tutorial on asynchronous JavaScript here](https://fjolt.com/article/javascript-async).

**Conclusion**

In this guide, we’ve gone through how fetch works. We’ve shown the different options you can send with your fetch() requests, and how to wait for the response using asynchronous concepts in JavaScript. fetch() is an incredibly powerful tool in JavaScript, and is used frequently in big products all the time. I hope you've enjoyed this article.