Tip 45: Make Simple AJAX Calls with Fetch

In this tip, you'll learn how to retrieve remote data using fetch().

We'll cover the following

- AJAX calls
- Using fetch()
  - GET requests
  - Handling the response body
  - Handling failed requests
- POST requests

AJAX calls

If you do any significant JavaScript app development, you'll have to interact with APIs. With APIs, you can get current information and update single elements without a page refresh. In short, you can create very fast applications that behave like native software.

Single-page web apps are part of the reason why JavaScript is so popular, but getting data with **AJAX—Asynchronous JavaScript And XML**—used to be a hassle. It was such a hassle that most developers used a library, usually **jQuery**, to to reduce the complexity. You can see the documentation on the Mozilla Developer Network. It's not easy stuff.

Using `fetch()`

Now, there's a much simpler tool for AJAX calls: `fetch()`. This tip is a little different than the others. `fetch()` isn't part of the JavaScript spec. The `fetch` spec is defined by the Web Hypertext Application Technology Working Group or WHATWG. That means you'll be able to find it in most major browsers, but it isn't natively supported in Node.js. If you want to use it in Node.js, you'll need to use the node-fetch package.

Enough trivia. *How does it work?*

Enough trivia. *How does it work?*

To start, you need an endpoint. The good folks at typicode have an API for fake blog data. They also make an amazing tool called JSON Server that enables you to mock APIs locally. JSON Server is a great way to mock APIs that are in development or slow, require authentication, or cost money for each call. You should use it.

Now that you have an endpoint, it's time to make some requests.

GET requests

The first request you'll make is a simple **GET** request. If all you're doing is asking for data, the `fetch()` call is simple. Call `fetch()` with the endpoint **URL** as the argument:

```
fetch('https://jsonplaceholder.typicode.com/posts/1');
```

The response body for this endpoint will be information about a blog post:

```
{
  userId: 1,
  id: 1,
  title: 'First Post',
  body: 'This is my first post...',
};
```

You can't get much easier than that. After you make the request, `fetch()` will return a *promise* that resolves with a *response*. The next thing you'll need to do is add a *callback* function to the `then()` method to handle the response.

Ultimately, you want to get the response body. But the response object contains quite a bit of information beyond the body, including the *status code, headers, and more*. You'll see more about the response in a moment.

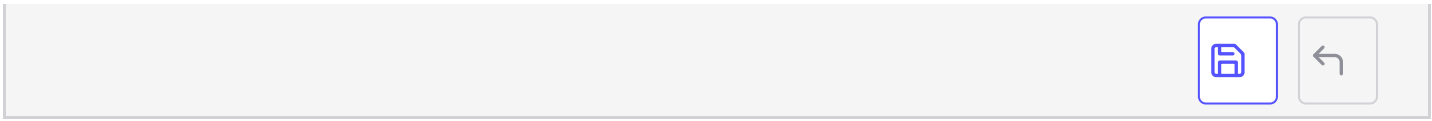
Handling the response body

The response body isn't always in a usable format. You may need to convert it to a format JavaScript can handle. Fortunately, `fetch()` contains a number of *mixins* that will automatically convert the response body data. In this case, because you know you're getting JSON, you can convert the body to **JSON** by calling `json()` on the response. The method also returns a *promise*, so you'll need another `then()` method. After that, you can do something with the parsed data. For example, if you want the `title` only, you can pull it out

want the `title` only, you can pull it out.

JavaScript

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(data => {
    return data.json();
  })
  .then(post => {
    console.log(post.title);
  });
```



Console

Clear

```
sunt aut facere repellat provident occaecati excepturi optio reprehenderit
```

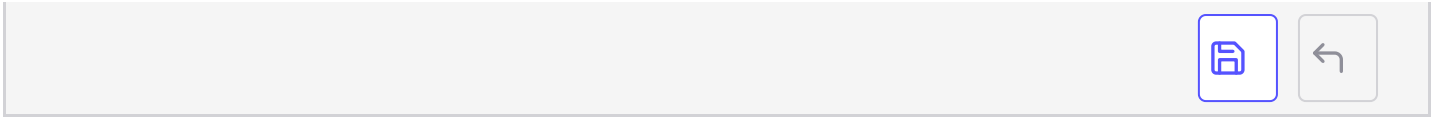
Handling failed requests

Of course, nothing is ever easy. The `fetch()` promise will resolve even if you get a *failing* status code, such as a `404` response. In other words, you can't rely on a `catch()` method on the promise to handle failed requests.

The response does include a field called `ok` that's set to `true` if the response code is in the `200-299` range. You can check for that response and throw an *error* if there's a problem. Unfortunately, Internet Explorer doesn't include `ok`, but Edge does. If you need to support older versions of Internet Explorer, you can check `response.status` to see if the value is between `200` and `299`.

JavaScript

```
fetch('https://jsonplaceholder.typicode.com/pots/1')
    .then(data => {
        if (!data.ok) {
            throw Error(data.status);
        }
        return data.json();
    })
    .then(post => {
        console.log(post.title);
    })
    .catch(e => {
        console.log(e);
    });
```



Console

Clear

```
404
```

POST requests

Most of your requests will be simple **GET** requests. But you'll eventually need to make more complex requests. What if you wanted to add a new blog post? Easy, make a **POST** request to https://jsonplaceholder.typicode.com/posts.

Once you move beyond **GET** requests, you'll need to set a few more options. So far, you've only supplied a single argument to `fetch()`—*the URL endpoint*. Now you'll need to pass an object of *configuration* options as a *second* argument. The optional object can take a lot of different details. In this case, include only the most necessary information.

Because you're sending a **POST** request, you'll need to declare that you're using the **POST** method. In addition, you'll need to pass some **JSON** data to actually create the *new* blog post. Because you're sending *JSON* data, you'll need to set a header of *Content-Type* set to application/json. Finally, you'll need the body, which will be a single string of JSON data.

The *final* request is nearly identical to your other request, except you pass in the special options as the *second* argument.

JavaScript

```
const update = {
  title: 'Clarence White Techniques',
  body: 'Amazing',
  userId: 1,
};
const options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(update),
};
fetch('https://jsonplaceholder.typicode.com/posts', options).then(data => {
  if (!data.ok) {
    throw Error(data.status);
  }
```

```
  return data.json();
}).then(update => {
  console.log(update);

}).catch(e => {
  console.log(e);
});
```

Console

Clear

```
{
   title: "Clarence White Techniques",
   body: "Amazing",
   userId: 1,
   id: 101
}
```

If your request is successful, you'll get a response body containing the blog post object along with a *new* ID. The response will vary depending on how the API is set up.

While JSON data is probably the most common request body, there are other options, such as *FormData*. Beyond that, there are even more methods for customizing your request. You can set a mode, a cache method, and so on. Most of these are specialized, but you'll need them at some point. As always, the best place to find out more is the Mozilla Developer Network.

Finally, you should be careful about where you place your AJAX requests in your code. Remember that `fetch` will most likely need an Internet connection, and endpoints may change during the project. It's good practice to keep all your `fetch` actions in one location. This will make them easier to update and easier to test. Check out the code for the course to see how you can create a services directory to store your `fetch` functions and how you can use them in other functions.

In the next tip, you'll learn how to preserve user data with `localStorage`.