Ryan Chenkie

Follow 2.7K Followers About

React Authentication: How to Store JWT in a Cookie

 Ryan Chenkie Apr 30, 2020 · 12 min read ★

👋 **Say hi to me on Twitter!**

If you have a React app that needs to access data, perhaps your setup looks like this:

If that's the case, there's a decent chance that your API is secured somehow. Maybe you're making authentication and authorization happen with JSON Web Tokens. If so, there's also a decent chance you're keeping your JWTs in local storage.

You should strongly consider **not** storing them there anymore.

Where Should JSON Web Tokens be Stored?

This question drums up a lot of controversy around the internet. Perhaps even more controversial is whether you should be using JSON Web Tokens at all. For many applications that are as simple as the diagram above, cookies and sessions would be a sufficient form of authentication and authorization and would offer a lot of benefits.

However, if you are in a position where you really need to use JWTs (or just really want to), there are some things you can do to harden the security posture for your React apps. In this article, we'll look specifically at where JWTs should be stored.

What are the options?

When moving your JWTs out of local storage, there are two options I recommend:

1. Browser memory (React state)
2. HttpOnly cookie

The first option is the more secure one because putting the JWT in a cookie doesn't completely remove the risk of token theft. Even with an HttpOnly cookie, sophisticated attackers can still use XSS and CSRF to steal tokens or make requests on the user's behalf.

However, the first option isn't always very practical. That's because storing a JWT in your React state will cause it to be lost any time the page is refreshed or closed and then opened again. This leads to a poor user experience—you don't want your users to need to log back in every time they refresh the page.

If you're using a third-party authentication service like Auth0 or Okta, this isn't a big deal because you can just call for another token behind the scenes (using a prompt=none call) to get a new token on refresh. However, this relies on a central auth server that is storing a session for your users.

The same isn't true if you're rolling your own auth. In that case, you most likely have a completely stateless backend that just signs tokens and validates them at your API. If you're using refresh tokens, Hasura has a great guide on how you can keep your access tokens in app state and refresh tokens in a cookie.

If you aren't able to keep your JWTs in app state, then the second option still offers some benefits. Most notably, if your app has any XSS vulnerabilities, attackers will not be able to steal your users' tokens as easily.

Putting your tokens in HttpOnly cookies is not a silver bullet though. Like any secure app, you need to effectively guard against both XSS and CSRF vulnerabilities. Ben Awad as a great video going into more detail.

An App that Uses Local Storage

Let's start by building out a small node API with **express** and a small React app. We'll start by having the app store tokens in local storage and we'll then move them to an HttpOnly cookie.

While we're using **express** for the backend in this tutorial, the same concepts map to pretty much any backend you might be using.

Create the API

```
npm i express express-jwt jsonwebtoken cors
```

In the entry file for the **express** API, add two routes: one for getting a JWT and the other for serving up some food data.

```
// server.js

const express = require('express');
const jwt = require('express-jwt');
const jsonwebtoken = require('jsonwebtoken');
const cors = require('cors');

const app = express();
```

```
app.use(cors());

const jwtSecret = 'secret123';

app.get('/jwt', (req, res) => {
  res.json({
    token: jsonwebtoken.sign({ user: 'johndoe' }, jwtSecret)
  });
});

app.use(jwt({ secret: jwtSecret, algorithms: ['HS256'] }));

const foods = [
  { id: 1, description: 'burritos' },
  { id: 2, description: 'quesadillas' },
  { id: 3, description: 'churos' }
];

app.get('/foods', (req, res) => {
  res.json(foods);
});

app.listen(3001);
console.log('App running on localhost:3001');
```

A few notes on the above code:

- The `jwtSecret` in this example is super weak. Don't use this kind of secret in real life. Use a long, complex, unguessable secret key.
- The route to get a JWT isn't checking any credentials, its just serving up a JWT when asked for one. This is just for simplicity. I assume you've got a proper mechanism for checking credentials (and if you don't, you can check out ReactSecurity for guides on how to do so).

Create the React App

Next, create a simple React app that makes calls for the food data.

```
npx create-react-app food-app
```

We'll use **axios** for this example but the same concepts apply to things like the browser's built-in Fetch API or any other HTTP library.

```
npm i axios
```

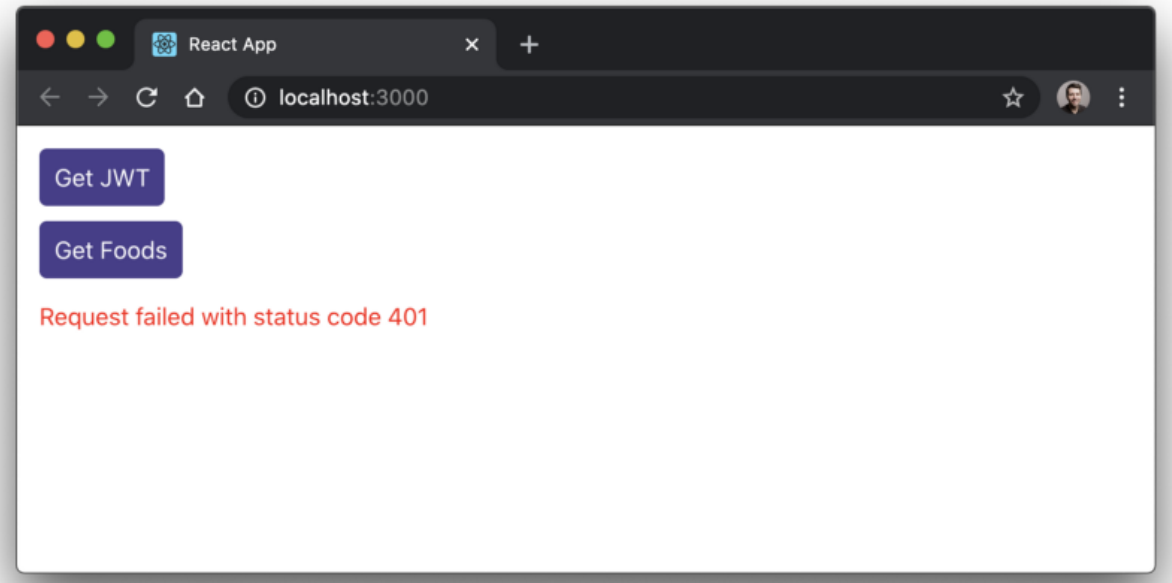
We can do all our work in `App.js` for this sample app.

```
// App.js

import React, { useState } from 'react';
import axios from 'axios';
import './App.css';

const apiUrl = 'http://localhost:3001';

axios.interceptors.request.use(
  config => {
    const { origin } = new URL(config.url);
    const allowedOrigins = [apiUrl];
    const token = localStorage.getItem('token');

    if (allowedOrigins.includes(origin)) {
      config.headers.authorization = `Bearer ${token}`;
    }
    return config;
  },
  error => {
    return Promise.reject(error);
  }
);

function App() {
  const storedJwt = localStorage.getItem('token');
  const [jwt, setJwt] = useState(storedJwt || null);
  const [foods, setFoods] = useState([]);
  const [fetchError, setFetchError] = useState(null);

const getJwt = async () => {
    const { data } = await axios.get(`${apiUrl}/jwt`);
    localStorage.setItem('token', data.token);
    setJwt(data.token);
  };

const getFoods = async () => {
    try {
      const { data } = await axios.get(`${apiUrl}/foods`);
      setFoods(data);
      setFetchError(null);
    } catch (err) {
      setFetchError(err.message);
    }
  };

return (
    <>
      <section style={{ marginBottom: '10px' }}>
        <button onClick={() => getJwt()}>Get JWT</button>
        {jwt && (
          <pre>
            <code>{jwt}</code>
          </pre>
        )}
```

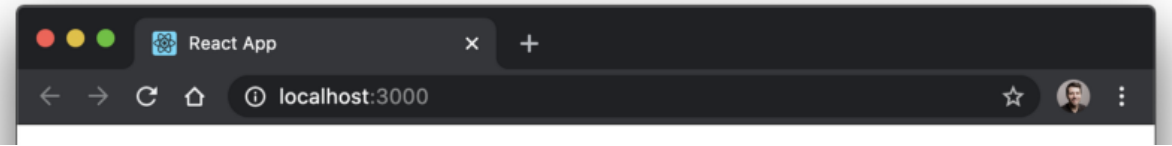
```
      </section>
      <section>
        <button onClick={() => getFoods()}>
          Get Foods
        </button>
        <ul>
          {foods.map((food, i) => (
            <li>{food.description}</li>
          ))}
        </ul>
        {fetchError && (
          <p style={{ color: 'red' }}>{fetchError}</p>
        )}
      </section>
    </>
  );
}

export default App;
```

The app has two buttons: one for getting a JWT and the other for getting a list of foods.

If we make a call for the foods before getting a JWT, we get an error.



But if we call for our JWT first, it gets stored in local storage and in our local component state. We are then able to make the request.



The token is being attached to the request by setting up an HTTP-interceptor with **axios**. It looks for whether the outgoing request is to an origin that we have pre-defined as being allowed and then attaches the user's JWT to the `Authorization` header if so.

Refactor to Store JWT in a Cookie

The first step to switching out to use cookies is to have our API set a cookie in the user's browser after they successfully log in. Cookies get set in the browser if the response to an HTTP call contains a `Set-Cookie` header. This header will have a string of cookie names and values, plus any additional settings for the cookies (like whether they should be HttpOnly or not).

In your express API, start by installing `cookie-parser`. It's an express middleware that allows us to parse cookies on incoming requests. This will help us later when we need to read the cookie value to grant access to the `foods` route.

```
npm i cookie-parser
```

Next, modify the route that sends back a JWT to set a cookie with a name of `token` and a value of the JWT itself.

```
// server.js

app.get('/jwt', (req, res) => {
  const token = jsonwebtoken.sign({ user: 'johndoe' }, jwtSecret);

  res.cookie('token', token, { httpOnly: true });

  res.json({ token });
});
```

The `httpOnly: true` setting means that the cookie can't be read using JavaScript but can still be sent back to the server in HTTP requests. Without this setting, an XSS attack could use `document.cookie` to get a list of stored cookies and their values.

Use an HTTP Proxy

So far, the React app has been running on port `3000` and the API on `3001`. This is fine if you're sending a JWT in the `Authorization` header of your API calls, but since we now want to send it in a cookie, we need to run the two apps on the same port. This is because cookies can only go to origins from which they came.

Since we used **create-react-app**, we can do this pretty easily in development mode. We just need to set the API URL as a `proxy` value in our `package.json` file.

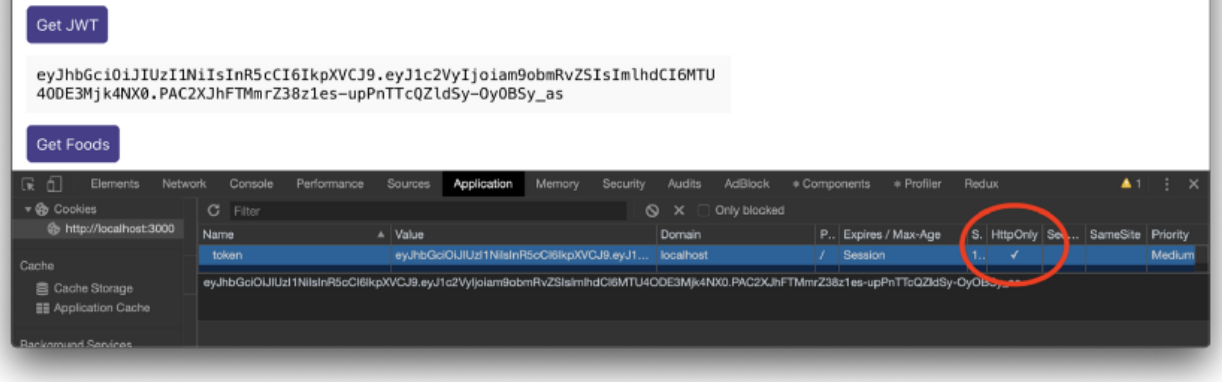
```
{
  ...
  "proxy": "http://localhost:3001"
}
```

Now in the React app, we can make API calls to a relative path instead of prefixing the calls with our API URL.

Refactor the call to the `/jwt` endpoint to no longer set the returned JWT in local storage. Instead, it will now be set as a cookie. We can keep the `setJwt` call so we can see the JWT on the screen.

```
// App.js

const getJwt = async () => {
  const { data } = await axios.get(`/jwt`);
  setJwt(data.token);
}
```

Clicking "Get JWT" will now return the JWT in the HTTP response as per usual, but will also set it as a cookie in the user's browser. If we inspect the cookies tab, we can see it in there as an HttpOnly cookie.



Validate the JWT from the Cookie

Now that the JWT is in a cookie, it will automatically be sent to the API in any calls we make to it. This is how the browser behaves by default. But again, we need to have our front end and backend served over the same origin to make this happen.

The JWT validation middleware supplied by **express-jwt** looks for a JWT on the `Authorization` header of requests by default. Let's update it to use a custom `getToken` function which will look for the token on an incoming cookie instead.

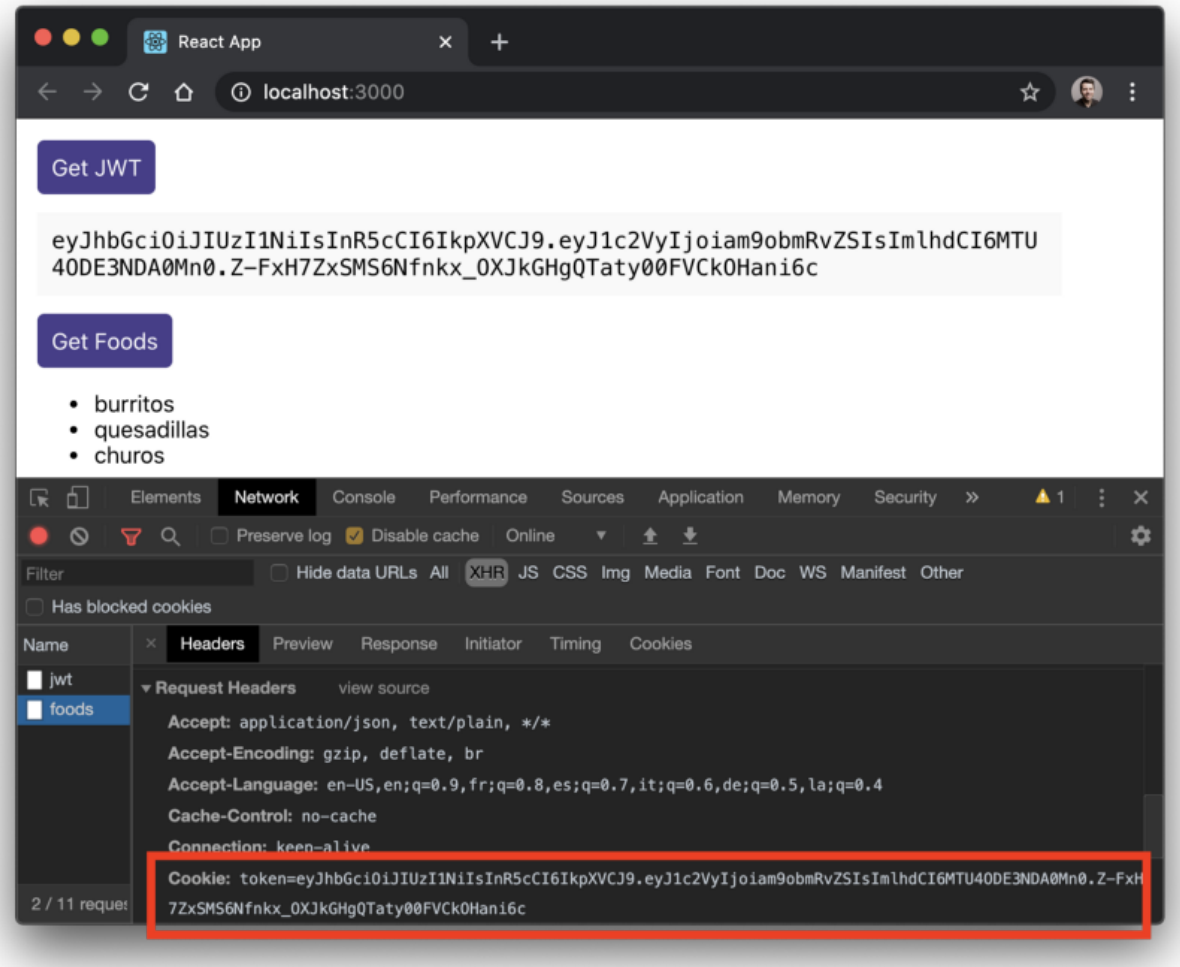
```
// server.js

app.use(cookieParser());

app.use(
  jwt({
    secret: 'secret123',
    getToken: req => req.cookies.token
  })
);
```

Not much needs to change in this case. Since we're using **cookie-parser**, we can just read the token right off of the cookies on the incoming request.

We can now adjust our `getFoods` call to go straight to `/foods` since we're using the proxy. In this call, we should now see that we're no longer sending the JWT as an `Authorization` header. Instead, it will be in a cookie.

```
// App.js

const getFoods = async () => {
  try {
    const { data } = await axios.get(`/foods`);
    setFoods(data);
    setFetchError(null);
```

```
  } catch (err) {
    setFetchError(err.message);
  }
};
```



Adding CSRF Protection

In the debate around where cookies should be stored, it's often brought up that local storage is susceptible to cross-site scripting (XSS) attacks, whereas cookies are susceptible to cross-site request forgery attacks (CSRF) attacks.

A CSRF attack is one in which a user is duped into performing some action in an app that they are currently logged into. If an attacker is able to get the user to make a request to that app (often without the user knowing it), the browser will automatically send its cookies, and thus the attack will be very possible.

There are numerous ways to make this happen, such as having the user load an image with a `src` point to the app in question along with some params that can perform some action.

This is really only an issue when it comes to mutating data on the server. Attackers don't have any way to see the results that a call to the server might provide, so a CSRF attack to retrieve data isn't useful.

One of the most common ways to do CSRF protection is to use an anti-CSRF token. Let's use a library called **csurf** to the **express** API to help.

```
npm i csurf
```

Now let's set up the **csurf** middleware. We'll want a new endpoint that accepts `GET` requests and sends back a new anti-CSRF token. This will later be called from our React app when it initializes.

```
// server.js

const csrfProtection = csrf({
  cookie: true
});

app.use(csrfProtection);

app.get('/csrf-token', (req, res) => {
  res.json({ csrfToken: req.csrfToken() });
});
```

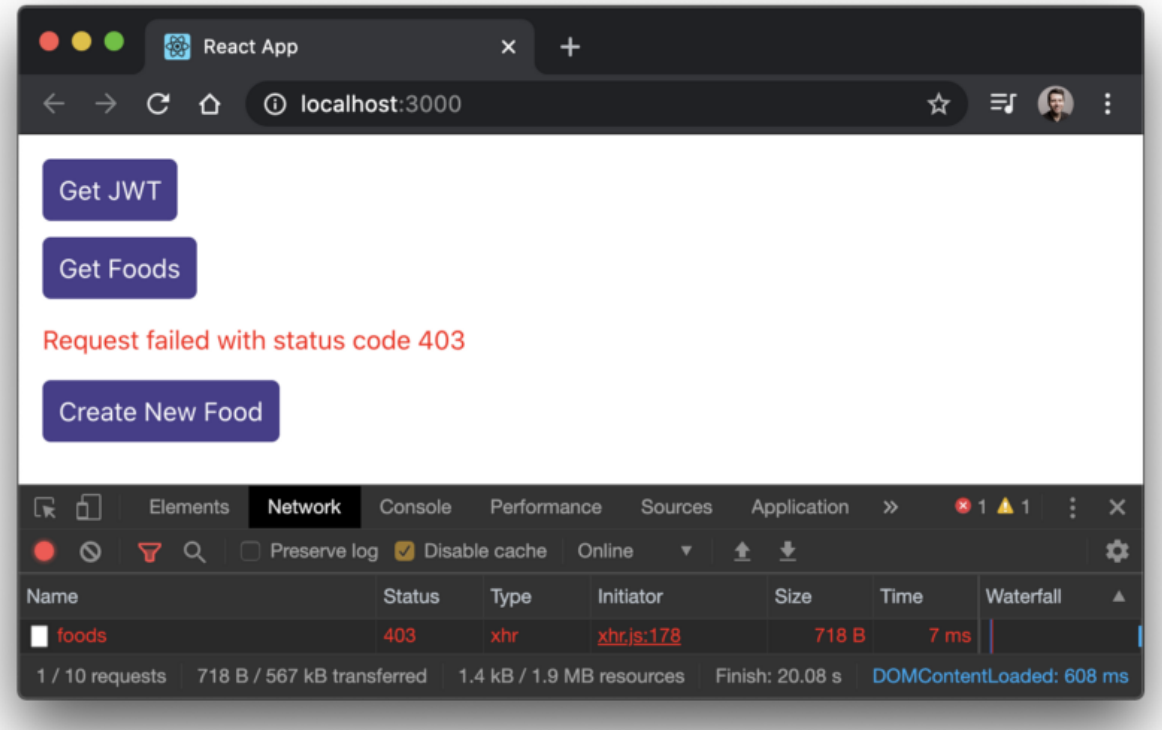
Let's now add a new route that accepts `POST` requests so we can see the middleware in action. It won't do anything for the existing `GET` request that we already have defined.

```
// server.js

app.post('/foods', (req, res) => {
  foods.push({
    id: foods.length + 1,
    description: 'new food'
  });
  res.json({
    message: 'Food created!'
  });
});
```

And we can add a call from the React app.

```
// App.js

function App() {
  const [newFoodMessage, setNewFoodMessage] = useState(null);

  const createFood = async () => {
    try {
      const { data } = await axios.post('/foods');
      setNewFoodMessage(data.message);
      setFetchError(null);
    } catch (err) {
      setFetchError(err.message);
    }
  };

  return (
    <>
      ...
      <section>
        <button onClick={() => createFood()}>
          Create New Food
        </button>
        {newFoodMessage && <p>{newFoodMessage}</p>}
      </section>
    </>
  );

}
```

In its current state, a `POST` request to create a new food item will result in a `403 Forbidden` error. This is because on the server we are requiring that a anti-CSRF token be present in the request, but we are not providing one.



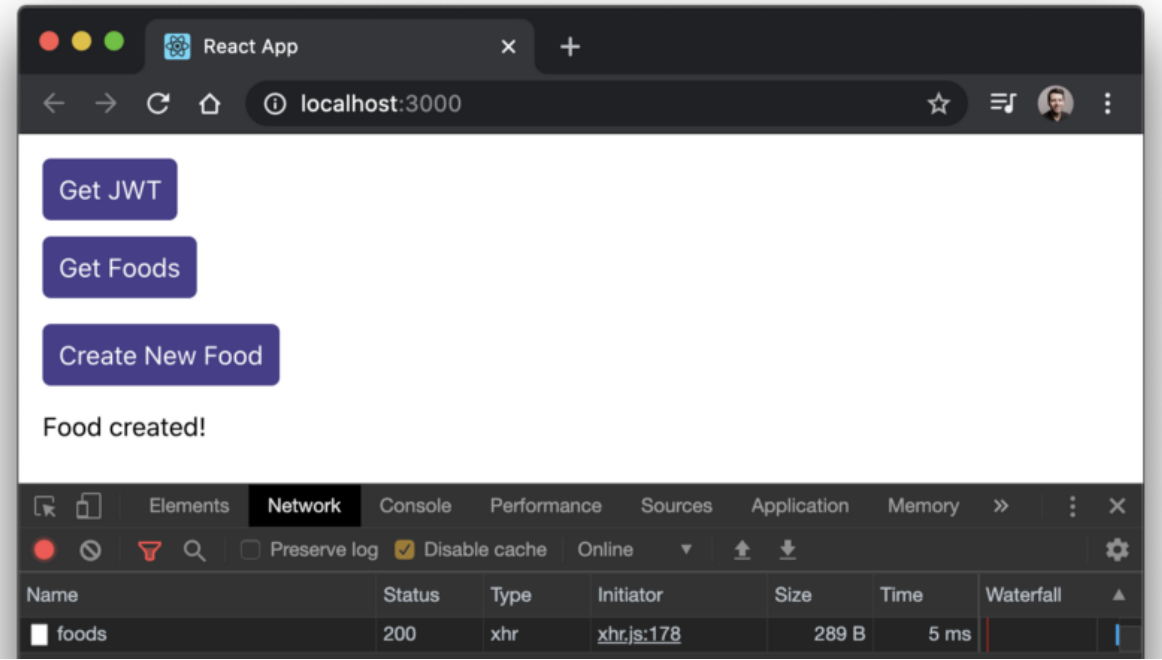
Getting and Setting the CSRF Token

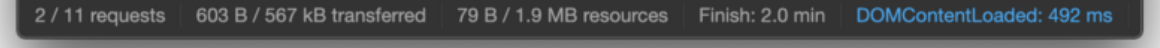
There are a number of different ways we can get the CSRF token and set it for later use. One common method is to put it in a `meta` tag when the app loads. It can then be set as a header in later requests as needed.

Since we're working from the root of our React app, we can send a request when the app loads and keep the resulting CSRF token in app state. It can then be set as a default header on `POST` request made by **axios**. This may or may not be suitable for your particular application.

```
// App.js

useEffect(() => {
  const getCsrfToken = async () => {
    const { data } = await axios.get('/csrf-token');
    axios.defaults.headers.post['X-CSRF-Token'] = data.csrfToken;
   };

  getCsrfToken();
}, []);
```

This isn't the only way to set the CSRF token in our requests. We could also set it as a header in individual outgoing requests. This setup might not be ideal for your own app but should give you an idea of how you can make it happen.





Other Considerations for Cookies

There are some cookie options other than HttpOnly that are useful for security. These include Max-Age (when the cookie expires) and Secure (connection needs to be over HTTPS).

In general, the cookie should expire when the JWT expires. This is a calculation that can be added easily in your backend when you set the cookie.

It's a good idea to set the cookie to be Secure. You don't want your JWTs going over the wire in the clear and this setting will allow you to be sure they don't.

Wrapping Up

If you have any XSS vulnerabilities in your app, you will be susceptible to token theft no matter where you store them. At the end of the day, keeping your JWT in a cookie can carry the same dangers as storing them in local storage. That means you really need to be sure that your app is free of XSS vulnerabilities in the first place.

That being said, many people prefer to use cookies for JWT storage as theft does arguably become somewhat more difficult.

If you can, store your JWTs in your app state and refresh them either through a central auth server or using a refresh token in a cookie, as outlined in this post by Hasura.

React Reactjs Authentication Authorization JavaScript