Use Fetch and OAuth to Make Authenticated Requests

Last updated March 31, 2020

Theming Module Development 8.9.x/9.0.x

To perform POST, PUT, and DELETE operations to Drupal's JSON:API via a decoupled React application we need to use an OAuth access token. This requires first fetching the access token from Drupal, and then including it in the HTTP `Authorization` header of all future requests. We'll also need to handle the situation where our access token has expired, and we need to get a new one using refresh token.

To update our example application we need to first write some JavaScript code to manage the OAuth tokens. Then we'll update our existing React components to use that new code.

In this tutorial we'll:

- Write code to exchange a username and password for an OAuth access token using the password grant flow
- Create a wrapper for the JavaScript `fetch()` that handles inserting the appropriate HTTP `Authorization` headers
- Update existing code that calls the Drupal JSON:API to use the new code

By the end of this tutorial you'll be able to use OAuth access tokens to make authenticated requests in a React application using `fetch`.

Goal

Modify our `NodeReadWrite` application's code to get and use an OAuth access token.

Prerequisites

- `NodeReadWrite` code from previous tutorials in this series
- Simple OAuth installed
- Make API requests with OAuth
- Access an API from the Browser with Cross-Origin Resource Sharing

What we're building

Here's an example of what your application should look like after completing this tutorial:

A note about CSRF tokens

If you've been following along with this series, you'll recall in the Build an Interface to Edit Nodes with React tutorial we added a `fetchWithCSRFToken()` helper function. It's used to retrieve a CSRF token from Drupal and include that in POST/PATCH/DELETE requests. Because we're switching to OAuth for user authentication and no longer riding on the browser's active session, we don't need to add an `X-CSRF-Token` to our HTTP request's headers. This code can be removed.

Write the code to make authenticated requests

This assumes you've already started writing an application following the steps in Use create-react-app to Start a Decoupled React Application. Though, the code examples should be helpful for any JavaScript application integrating with Drupal's OAuth features.

Start by creating a set of helper functions that we can re-use throughout our application. At a high level we'll need to be able to do the following:

- Log a user in by getting a new OAuth access token from Drupal using a username and password
- Allow a user to logout
- Store the OAuth token retrieved from Drupal in our application
- Make a `fetch()` request to Drupal that includes an OAuth access token
- Refresh a locally stored access token when it expires using a refresh token

Add the file *src/utils/auth.js* with the following content. We'll explain it in detail below:

```
/**
 * @file
```

```
 *
 * Wrapper around fetch(), and OAuth access token handling operations.
 *
 * To use import getAuthClient, and initialize a client:
 * const auth = getAuthClient(optionalConfig);
 */

const refreshPromises = [];

/**
 * OAuth client factory.
 *
 * @param {object} config
 *
 * @returns {object}
 *   Returns an object of functions with $config injected.
 */
export function getAuthClient(config = {}) {
  const defaultConfig = {
    // Base URL of your Drupal site.
    base: 'https://react-tutorials-2.ddev.site',
    // Name to use when storing the token in localStorage.
    token_name: 'drupal-oauth-token',
    // OAuth client ID - get from Drupal.
    client_id: 'cb0379f2-7c9f-48bb-8973-f0f11b6064d5',
    // OAuth client secret - set in Drupal.
    client_secret: 'app',
    // Drupal user role related to this OAuth client.
    scope: 'oauth',
    // Margin of time before the current token expires that we should force a
    // token refresh.
    expire_margin: 0,
  };

  config =  {...defaultConfig, ...config}

  /**
   * Exchange a username & password for an OAuth token.
   *
   * @param {string} username
   * @param {string} password
   */
  async function login(username, password) {
    let formData = new FormData();
    formData.append('grant_type', 'password');
    formData.append('client_id', config.client_id);
    formData.append('client_secret', config.client_secret);
    formData.append('scope', config.scope);
    formData.append('username', username);
    formData.append('password', password);
    try {
      const response = await fetch(`${config.base}/oauth/token`, {
        method: 'post',
        headers: new Headers({
          'Accept': 'application/json',
        }),
        body: formData,
      });
      const data = await response.json();
      if (data.error) {
        console.log('Error retrieving token', data);
        return false;
      }
      return saveToken(data);
    }
    catch (err) {
      return console.log('API got an error', err);
    }
  };

  /**
   * Delete the stored OAuth token, effectively ending the user's session.
   */
  function logout() {
    localStorage.removeItem(config.token_name);
    return Promise.resolve(true);
  };

  /**
   * Wrapper for fetch() that will attempt to add a Bearer token if present.
   *
   * If there's a valid token, or one can be obtained via a refresh token, then
   * add it to the request headers. If not, issue the request without adding an
```

```
 * Authorization header.
 *
 * @param {string} url URL to fetch.
 * @param {object} options Options for fetch().
 */
async function fetchWithAuthentication(url, options) {
  if (!options.headers.get('Authorization')) {
    const oauth_token = await token();
    if (oauth_token) {
      console.log('using token', oauth_token);
      options.headers.append('Authorization', `Bearer ${oauth_token.access_token}`);
    }
    return fetch(`${config.base}${url}`, options);
  }
}

/**
 * Get the current OAuth token if there is one.
 *
 * Get the OAuth token form localStorage, and refresh it if necessary using
 * the included refresh_token.
 *
 * @returns {Promise}
 *   Returns a Promise that resolves to the current token, or false.
 */
async function token() {
  const token = localStorage.getItem(config.token_name) !== null
    ? JSON.parse(localStorage.getItem(config.token_name))
    : false;

  if (!token) {
    Promise.reject();
  }

  const { expires_at, refresh_token } = token;
  if (expires_at - config.expire_margin < Date.now()/1000) {
    return refreshToken(refresh_token);
  }
  return Promise.resolve(token);
};

/**
 * Request a new token using a refresh_token.
 *
 * This function is smart about reusing requests for a refresh token. So it is
 * safe to call it multiple times in succession without having to worry about
 * whether a previous request is still processing.
 */
function refreshToken(refresh_token) {
  console.log("getting refresh token");
  if (refreshPromises[refresh_token]) {
    return refreshPromises[refresh_token];
  }

  // Note that the data in the request is different when getting a new token
  // via a refresh_token. grant_type = refresh_token, and do NOT include the
  // scope parameter in the request as it'll cause issues if you do.
  let formData = new FormData();
  formData.append('grant_type', 'refresh_token');
  formData.append('client_id', config.client_id);
  formData.append('client_secret', config.client_secret);
  formData.append('refresh_token', refresh_token);

  return(refreshPromises[refresh_token] = fetch(`${config.base}/oauth/token`, {
    method: 'post',
    headers: new Headers({
      'Accept': 'application/json',
    }),
    body: formData,
    })
    .then(function(response) {
      return response.json();
    })
    .then((data) => {
      delete refreshPromises[refresh_token];

      if (data.error) {
        console.log('Error refreshing token', data);
        return false;
      }
      return saveToken(data);
    })
    .catch(err => {
```

```
      delete refreshPromises[refresh_token];
      console.log('API got an error', err)
      return Promise.reject(err);
    })
  );
}

/**
 * Store an OAuth token retrieved from Drupal in localStorage.
 *
 * @param {object} data
 * @returns {object}
 *   Returns the token with an additional expires_at property added.
 */
function saveToken(data) {
  let token = Object.assign({}, data); // Make a copy of data object.
  token.date = Math.floor(Date.now() / 1000);
  token.expires_at = token.date + token.expires_in;
  localStorage.setItem(config.token_name, JSON.stringify(token));
  return token;
}

/**
 * Check if the current user is logged in or not.
 *
 * @returns {Promise}
 */
async function isLoggedIn() {
  const oauth_token = await token();
  if (oauth_token) {
    return Promise.resolve(true);
  }
  return Promise.reject(false);;
};

/**
 * Run a query to /oauth/debug and output the results to the console.
 */
function debug() {
  const headers = new Headers({
    Accept: 'application/vnd.api+json',
  });

  fetchWithAuthentication('/oauth/debug?_format=json', {headers})
    .then((response) => response.json())
    .then((data) => {
      console.log('debug', data);
    });
}

return {debug, login, logout, isLoggedIn, fetchWithAuthentication, token, refreshToken};
}
```

The above code is a lightweight library for managing OAuth tokens. You could also use an existing library; many HTTP request libraries like Axios or request have middleware to assist with OAuth requests already. We wanted to create our own to help better explain how this all works.

The module exports a `getAuthClient()` function, which returns an object populated with the libraries functions, `return {debug, login, logout, isLoggedIn, fetchWithAuthentication, token, refreshToken};`. This is a technique called currying, which allows us to have a set of functions that all share the same configuration, without having to pass that configuration to each function individually. It's somewhat akin to dependency injection.

The functions it returns include:

- `login()`: This function takes a username and password, and uses them to make a request to the Drupal /oauth/token endpoint attempting to retrieve a new OAuth token. Then it calls `saveToken()`: to store a copy of the new token in localStorage. This function gets used as part of a submit handler for a login form.
- `fetchWithAuthentication()`: Wrapper around the JavaScript `fetch()` function that will insert an HTTP `Authorization` header with the current access token if one is found. This uses `token()`, so if the access token is expired but there is a refresh token it will first attempt to refresh the access token before using it. This gets used any time you want to make a request to the Drupal API that requires authorization.

- `logout()`: Delete the access token from localStorage, effectively logging the user out of their current session. This gets used as part of the callback for a logout link or button.
- `isLoggedIn()`: Check to see if the current user is logged in or not. Intended to get used as a helper when a component or some other code needs to know if the user is logged in. To figure this out, the function calls `token()` and returns true if there's a valid token to use in the request.
- `token()`: Retrieve the current token, optionally refreshing it via the included refresh token if the access token is expired. OAuth access tokens typically have a short expiration time for security reasons. To keep a user logged in without requiring them to re-enter their username and password every couple of minutes we use a refresh token (which has a much longer expiration time) to get a new access token. To make this as seamless as possible we try and perform this operation automatically any time the access token gets used, keeping the burden on our OAuth client library and not the implementing code.
- `refreshToken()`: This function makes the actual request to Drupal that uses a refresh token to obtain a new access token. Because it's possible that this function gets called multiple times in rapid succession it has some debouncing logic: if a previous request to refresh the token is already in progress it will return that one instead of starting a new one. This prevents a scenario where the access token is expired, and two or more components use the `fetchWithAuthentication()` function to make a request to Drupal. The first one will trigger an attempt to refresh the access token, but since that operation is asynchronous it's possible that while the request is still being processed `fetchWithAuthentication()` will be called a second time which would generate a 2nd token and invalidating the 1st one before it's even used.
- `saveToken()`: Helper function to store a token retrieved from Drupal in localStorage. Also calculates the expiration time of the token before storing it so we only need to do this calculation once.
- `debug()`: This makes a request to the Drupal OAuth debugging endpoint at /oauth/debug. Which, if you include an Authorization header, will include information about the token like what roles it's associated with, what permissions it allows, and more. This is useful if you're trying to figure out why a request with a token included doesn't have permission to perform one or more operations.

Note the `defaultConfig` variable. **This needs to be updated to contain relevant values for your application.** Or, you can pass in the necessary config for your environment when you call `getAuthClient()` and it will merge with the defaults.

Add a UI so users can login and logout

Let's create a new `LoginForm` component that displays a form that users can fill out with a username and password to login. If the current user is already logged in, display a button that allows them to logout.

Add a new file, *src/components/LoginForm.jsx* with the following content:

```
import React, { useState, useEffect } from 'react';
import { getAuthClient } from '../utils/auth';

const auth = getAuthClient();

const LoginForm = () => {
  const [isSubmitting, setSubmitting] = useState(false);

  const [result, setResult] = useState({
    success: null,
    error: null,
    message: '',
  });

  const defaultValues = {name: '', pass: ''};
  const [values, setValues] = useState(defaultValues);

  const [isLoggedIn, setLoggedIn] = useState(false);
  // Only need to do this on first mount.
  useEffect(() => {
    auth.isLoggedIn().then((res) => {
      setLoggedIn(true);
    })
  }, []);

  const handleInputChange = (event) => {
    const {name, value} = event.target;
    setValues({...values, [name]: value});
  };

  const handleSubmit = (event) => {
```

```
    event.preventDefault();
    setSubmitting(true);

    auth.login(values.name, values.pass)
      .then(() => {
        setSubmitting(false);
        setLoggedIn(true);
        setResult({ success: true, message: 'Login success' });
      })
      .catch((error) => {
        setSubmitting(false);
        setLoggedIn(false);
        setResult({ error: true, message: 'Login error' });
        console.log('Login error', error);
      });
  };

  if (isLoggedIn) {
    return(
      <div>
        <p>You're currently logged in.</p>
        <button onClick={() => auth.logout().then(setLoggedIn(false))}>
          Logout
        </button>
      </div>
    );
  }

  if (isSubmitting) {
    return (
      <div>
        <p>Logging in, hold tight ...</p>
      </div>
    )
  }

  return (
    <div>
      {(result.success || result.error) &&
        <div>
          <h2>{(result.success ? 'Success!' : 'Error')}:</h2>
          {result.message}
        </div>
      }
      <form onSubmit={handleSubmit}>
        <input
          name="name"
          type="text"
          value={values.name}
          placeholder="Username"
          onChange={handleInputChange}
        />
        <br/>
        <input
          name="pass"
          type="text"
          value={values.pass}
          placeholder="Password"
          onChange={handleInputChange}
        />
        <br/>
        <input
          name="submit"
          type="submit"
          value="Login"
        />
      </form>
    </div>
  );
};

export default LoginForm;
```

This provides a new form where a user can input their username and password. Then, when the form gets submitted, it uses the `login()` function from our OAuth library to request a new access token from Drupal. The important parts are:

- Import `import { getAuthClient } from '../utils/auth';` the OAuth, and initialize `const auth = getAuthClient();` an OAuth client.

- Check to see if the current user is already logged in or not. If they are, display a logout button; if they are not, display a login form. The call to `auth.isLoggedIn()` returns a Promise that will resolve to true if the current user already has an active access token. It's wrapped in `useEffect` to ensure the check is only performed once when the component is first loaded. After that we keep track of logged in state in the component itself.
- In the `handleSubmit` function call `auth.login(username, password)` to retrieve a token. It returns a Promise, which (if it resolves) we can assume it retrieved a token and added it to localStorage. We can now treat the current user as authenticated.
- The `onClick` handler for the logout button calls `auth.logout()` which returns a Promise that resolves after the current user's access token gets deleted and they are logged out.

Make the `LoginForm` component active by importing it into *src/App.js* and outputting it as part of the main `App` component.

Example:

```
function App() {
  return (
    <>
      <LoginForm />
      <NodeReadWrite />
    </>
  );
}
```

Update existing components to use new OAuth library

Next, update any existing components and replace calls to either `fetch()` or `fetchWithCsrfToken()` to use the new `fetchWithAuthentication()` function from above.

1 Update `NodeReadWrite`

Edit the file *src/components/NodeReadWrite.jsx*, and at the top import and intialize the OAuth client:

```
import { getAuthClient } from "../utils/auth";
const auth = getAuthClient();
```

Then replace the existing call to `fetchWithCsrfToken()` with `auth.fetchWithAuthentication(url, {headers})`. You may also need to update the `url` variable. It should be a root relative path, and the OAuth client library will prefix it with the Drupal base URL.

The `fetchWithAuthentication` function can is a drop in replacement for `fetch()`. If there's no OAuth token present it'll skip adding the `Authorization` header and make an anonymous HTTP request. The function gets the list of content regardless of whether the user is logged in or not.

Example:

```
const url = `/jsonapi/node/article?fields[node--article]=id,drupal_internal__nid,title,body&sort=-created&page[limit]=10`;
```

2 Update `NodeForm`, and `NodeDelete`

Edit the *src/components/NodeForm.jsx*, *src/components/NodeDelete.jsx*, files and import and initialize the OAuth client code (same as above). Then replace calls to `fetchWithCsrfToken()` with `auth.fetchWithAuthentication()`.

3 Add a bit of style

Let's add a bit of CSS to make our app easier to use. Since we started with `create-react-app` our project has a *src/App.css* file which we can edit to add global styles. Webpack is already configured to process it, add vendor prefixes, and compress it.

This works because the *src/App.js* file contains an import like this:

```
import './App.css';
```

Webpack will recognize that the import is actually for a .css file, and take appropriate actions.

Edit *src/App.css* and add the following, along with any styling you want to apply:

```
#root {
  border: 1px solid #000;
  margin: 1em auto;
  max-width: 600px;
  padding: 1em;
}

hr {
  border: 1px solid palevioletred;
}

input, textarea, button {
  border: 1px solid #444;
  margin: 0.5em 0.5em 0.5em 0;
  padding: 0.5em;
}

input[type="submit"] {
  padding: 0.5em;
}

button:hover,
input[type="submit"]:hover {
  border-color: yellowgreen;
  cursor: pointer;
}
```

Read more about the various ways you can add CSS styles to a create-react-app project.

With this code in place, if someone visits the page and there is no token in the localStorage, then they will see a login form. When they login, an OAuth token that contains both an access token and a refresh token gets retrieved from Drupal. The login form's state gets updated and changes to display a logout button. All future requests to Drupal, assuming they're made via the `auth.fetchWithAuthentication()` function, will include the access token in the HTTP `Authorization` header if it's present. Drupal will treat those as authenticated requests, and allow operations like `POST`, and `DELETE` if the user represented by the access token is authorized to do those things.

Additional code required

At this point our application is a fully decoupled version of the same application we built inside the Drupal theme earlier. If you start to play around you'll notice some things that don't work. For example, you can't navigate to an article by pressing the link.

When our application is contained within Drupal this works because Drupal handled responding to a path like /node/42 with the appropriate content. In our decoupled application there's nothing in place to handle those paths. We'll add this feature in a future tutorial.

Recap

In this tutorial, we added code to our application to retrieve and refresh OAuth access tokens. Then we updated all the code in our existing components that makes API requests to allow it to use the new access token to make an authenticated request. Finally, we added a login component to allow our application to collect a username and password from a user and exchange it for an OAuth access token.

Further your understanding

- What is the difference between an access token and a refresh token? What are they each used for?
- Update the code so that the *edit* and *Add a node* buttons only show up for users who are logged in.
- Can you update this codebase so that when a user transitions from logged out to logged in, or vice versa, the `NodeReadWrite` component updates without a refresh? Hint: you can do this with React's `useContext` to manage global state variables.

Additional resources

- How to Implement Authentication For Your React App (medium.appbase.io) This is a similar implementation of authentication, which also includes some integration with a router, for page redirects.

Was this helpful? Yes No

◀ Previous tutorial

Get Started Using React and Drupal Together

Subscribe

STAY CONNECTED

Powered by: