Build an Interface to Edit Nodes with React

Last updated March 31, 2020

Theming Module Development 8.9.x/9.0.x

Using React we can do more than just list content. By using the POST, PATCH, and PUT methods of Drupal core's JSON:API web service we can also add, update, and delete, content entities. To demonstrate how this works we'll create a small React application with a form that lets you add, edit, and delete article nodes.

In this tutorial we'll:

- Learn how to handle user authentication and CSRF tokens in a React application
- Create a single React component that outputs a form to add or edit content
- Create a wrapper around the JavaScript `fetch` API to assist in dealing with requests to Drupal's JSON:API

By the end of this tutorial you should know how to create, edit, and delete content in a Drupal site using a React application.

Goal

Build a React application that can create, edit, and delete node content.

Prerequisites

- Install JSON:API Module.
- This tutorial builds on the work started in the previous tutorials in this series. At a minimum you'll need to know how to Connect React to a Drupal Theme or Module. We'll be starting from the code written in Use React to List Content from Drupal.

Required JSON:API configuration

Drupal core's JSON:API module is set to read-only mode by default -- anything other than HTTP GET requests are denied. To make POST, PATCH, and DELETE requests the JSON:API configuration must be set to read-write mode.

In the Drupal *Manage* administration menu, navigate to *Configuration* > *Web services* > *JSON:API* (admin/config/services/jsonapi). Using the provided settings form, make sure the *Allowed operations* field is set to "Accept all JSON:API create, read, update, and delete operations." Press *Save*.

If you're seeing an HTTP 405 error when trying to POST or PATCH to JSON:API you might need to update this configuration to read-write.

Example code

Code for this example is in the Git repository: https://github.com/DrupalizeMe/react-and-drupal-examples.

Create the pieces of our CRUD application

1 Create a wrapper around the JavaScript `fetch` function

In Overview: Create, Update, and Delete Drupal Content with JavaScript we learned about the need to use CSRF tokens when making write requests to JSON:API that use the browser's existing session for authentication. To make it easier we created a new function `fetchWithCsrfToken()` that provides a wrapper around `fetch()`.

Add the file *js/src/utils/fetch.js*:

```
/**
 * Helper function wraps a normal fetch call with a fetch request that first
 * retrieves a CSRF token and then adds the token as an X-CSRF-Token header
 * to the options for desired fetch call.
 *
 * @param {string} csrfUrl
 *   URL where we can retrieve a CSRF token for the current user.
 * @param {string} fetchUrl
 *   URL to fetch with X-CSRF-Token header included.
 * @param {object} fetchOptions
 *   Options to pass to fetch for the call to fetchUrl.
 */
export const fetchWithCSRFToken = (csrfUrl, fetchUrl, fetchOptions) => {
  if (!fetchOptions.headers.get('X-CSRF-Token')) {
    return fetch(csrfUrl)
      .then(response => response.text())
      .then((csrfToken) => {
        fetchOptions.headers.append('X-CSRF-Token', csrfToken);
        return fetch(fetchUrl, fetchOptions);
      });
  }
  else {
    return fetch(fetchUrl, fetchOptions);
  }
};
```

This function provides a helpful wrapper around the `fetch()` API that will first make a request to `/session/token?_format=json` to get a CSRF token, and then include that token in the final request via the `X-CSRF-Token` header. We'll use it anytime we need to make a write request to Drupal's JSON:API from our application.

2 Create a `NodeForm` component

In order for a user to add or edit an article, they'll need a form they can use to modify the values. To keep things simple, our form will only allow editing the article's title and body fields for now. We can also make a single form component for both scenarios by adding a little extra logic to the code. Then, we'll wrap the form in a `NodeAdd` and `NodeEdit` component to provide a better developer experience.

Create the file *js/src/components/NodeForm.jsx* with the following code:

```
import React, {useState} from "react";
import {fetchWithCSRFToken} from "../utils/fetch";

const NodeForm = ({id, title, body, onSuccess}) => {
  const [isSubmitting, setSubmitting] = useState(false);

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

  const defaultValues = {
    title: title ? title : '',
    body: body ? body : '',
  };
  const [values, setValues] = useState(defaultValues);

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

  const handleSubmit = (event) => {
    setSubmitting(true);
    event.preventDefault();

    const csrfUrl = `/session/token?_format=json`;
    const fetchUrl = id ? `/jsonapi/node/article/${id}` : `/jsonapi/node/article`;

    let data = {
      "data": {
        "type": "node--article",
        "attributes": {
          "title": `${values.title}`,
          "body": {
            "value": `${values.body}`,
            "format": 'plain_text',
          }
        }
      }
    };

    // If we have an ID that means we're editing an existing node and not
    // creating a new one.
    if (id) {
      // Set the ID in the data we'll send to the API.
      data.data.id = id;
    }

    const fetchOptions = {
      // Use HTTP PATCH for edits, and HTTP POST to create new articles.
      method: id ? 'PATCH' : 'POST',
      credentials: 'same-origin',
      headers: new Headers({
        'Accept': 'application/vnd.api+json',
        'Content-Type': 'application/vnd.api+json',
        'Cache': 'no-cache'
      }),
      body: JSON.stringify(data),
    };

    try {
      fetchWithCSRFToken(csrfUrl, fetchUrl, fetchOptions)
        .then((response) => response.json())
        .then((data) => {
          // We're done processing.
          setSubmitting(false);

          // If there are any errors display the error and return early.
          if (data.errors && data.errors.length > 0) {
            setResult({
              success: false,
              error: true,
              message: <div className="messages messages--error">{data.errors[0].title}: {data.errors[0].detail}</div>,
            });
            return false;
          }

          // If the request was successful, remove existing form values and
          // display a success message.
          setValues(defaultValues);
          if (data.data.id) {
            setResult({
              success: true,
              message: <div className="messages messages--status">{(id ? 'Updated' : 'Added')}: <em>{data.data.attributes.title}
            });

            if (typeof onSuccess === 'function') {
              onSuccess(data.data);
            }
          }
        });
    } catch (error) {
      console.log('Error while contacting API', error);
      setSubmitting(false);
    }
  };

  // If the form is currently being processed display a spinner.
  if (isSubmitting) {
    return (
      <div>
        Processing ...
      </div>
    )
  }

  return (
    <div>
      {(result.success || result.error) &&
        <div>
          <h2>{(result.success ? 'Success!' : 'Error')}:</h2>
          {result.message}
        </div>
      }
      <form onSubmit={handleSubmit}>
        <input
```

```
          name="title"
          type="text"
          value={values.title}
          placeholder="Title"
          onChange={handleInputChange}
        />
        <br/>
        <textarea
          name="body"
          rows="4"
          cols="30"
          value={values.body}
          placeholder="Body"
          onChange={handleInputChange}
        />
        <br/>
        <input
          name="submit"
          type="submit"
          value={id ? 'Edit existing node' : 'Add new node'}
        />
      </form>
    </div>
  )
};

export default NodeForm;
```

There's a lot going on in this component. Let's walk through the important parts:

- The `NodeForm` component takes four props: `id`, `title`, and `body` are all optional. If present, they mean we're editing an existing article, and these props contain the current values. `onSuccess` is a function that gets called after the form gets submitted. For example, after we make a request to the API to save a new Article, call the `onSuccess` callback function.
- We establish three new state variables: `isSubmitting` keeps track of whether the form is currently processing or not and allows us to display a *loading* indicator; `result` is used to keep track of and display the results of any API requests; and `values` tracks the content of the various `<input>` fields in the form.
- `handleInputChange()` provides a generic onChange callback for updating the `values` state variable whenever a user modifies the content of a form item.
- The `handleSubmit()` function serves as a callback for the `onSubmit` event of the form. This is code that's called when the form gets submitted, and contains the bulk of the logic for adding and editing articles.
- Finally, there's a bit of logic that says, "if the form `isSubmitting`, output a *processing* message, otherwise display a `<form>` with the fields pre-populated if we have content for an existing article, as well as any success or error messages."

Let's also dig into the `handleSubmit()` function a bit more. At a high level this function gets called whenever the form gets submitted, takes the values from the form, and uses them to make either a `POST` or a `PATCH` request to the Drupal JSON:API -- depending on whether we are adding a new article or editing an existing one. This function:

- Starts by setting the `isSubmitting` state of the form to `true`, which causes the component to re-render and the *processing* message to display
- Then it figures out some configuration for `fetch`. Notably, if `id` is present we are editing an existing article, and we make a `PATCH` request to `jsonapi/node/article/{UUID}`. If `id` is empty we create a new article, and make a `POST` request to `jsonapi/node/article`
- Rather than call `fetch()` directly we use the `fetchWithCsrfToken()` function we created earlier
- If there's an error on the Drupal site, like the current user doesn't have permission, or a required field is missing, the response will contain `data.errors` which is an array of errors generated by the API. Note that this only works if you set the HTTP `Accept` header to `'application/vnd.api+json'`. If any errors are returned we update the `result` state to display them.
- Otherwise, we update `result` with a success message.
- If the API successfully saves the article it'll return a copy of the article (with any changes made) in the response as `data.data`. Our code calls the `onSuccess` function passed to the component, and passes in the new `data.data` object.

3 Create a `NodeAdd` component

To make our code a little more developer-friendly let's wrap the `NodeForm` component with `NodeAdd` and `NodeEdit`. While not required, this will make it clearer to anyone reading our code later on which version of the form gets shown without having to try and figure out if there's an existing article or not.

Create the file *js/src/components/NodeAdd.jsx* with the following code:

```
import React from "react";
import NodeForm from './NodeForm';

const NodeAdd = ({ onSuccess }) => (
  <NodeForm
    id={null}
    onSuccess={onSuccess}
  />
);

export default NodeAdd;
```

Here we set `id={null}` so that the `NodeForm` component will always operate using its *create a new article* logic.

4 Create a `NodeEdit` component

Create the file *js/src/components/NodeEdit.jsx* with the following code:

```
import React from "react";
import NodeForm from "./NodeForm";

const NodeEdit = ({ id, title, body, onSuccess }) => (
  <NodeForm
    id={id}
    title={title}
    body={body}
    onSuccess={onSuccess}
  />
);

export default NodeEdit;
```

Doing this makes it easier to see that the `id`, `title`, and `body` fields are necessary for editing an article. You could take this even further by adding type checking to the component using PropTypes so that these props are required, documented, and a developer's IDE could make intelligent suggestions.

5 Create a `NodeDelete` component

Delete requests are a little bit different from adding or editing an article. There's no form the user needs to fill out. We already have everything we need to present a button users can press to delete an article. But we should collect confirmation from them before doing so, since it's an irreversible action.

Create the file *js/src/components/NodeDelete.jsx* with the following code:

```
import React from "react";
import { fetchWithCSRFToken } from "../utils/fetch";

const NodeDelete = ({ id, title, onSuccess }) => {
  function doConfirm() {
    return window.confirm(`Are you sure you want to delete ${title}?`);
  }

  function doDelete() {
    const csrfUrl = `/session/token?_format=json`;
    const fetchUrl = `/jsonapi/node/article/${id}`;
    const fetchOptions = {
      method: 'DELETE',
      credentials: 'same-origin',
      headers: new Headers({
        'Accept': 'application/vnd.api+json',
        'Content-Type': 'application/vnd.api+json',
        'Cache': 'no-cache'
      }),
    };
    try {
      fetchWithCSRFToken(csrfUrl, fetchUrl, fetchOptions)
        .then((response) => {
          // Should be 204. If so, call the onSuccess callback.
          if (response.status === 204) {
            if (typeof onSuccess === 'function') {
              onSuccess(id);
            }
          }
        });
    } catch (error) {
      console.log('API error', error);
    }
  }

  return (
    <button onClick={event => doConfirm() && doDelete()}>
      delete
    </button>
  );
};

export default NodeDelete;
```

To understand what's happening here, start with the `<button>` this component renders.

- This JSX statement `<button onClick={event => doConfirm() && doDelete()}>` says: when the button gets pressed, first call the function `doConfirm()`. If that function returns a truthy value, call `doDelete()`, otherwise skip `doDelete()`. The `&&` operator evaluates conditions from left to right until it encounters a falsy value, at which point it stops execution and returns the value from the falsy condition.
- The `doConfirm()` function opens the Browser's native confirmation dialog. If the user clicks "Yes" it returns `true`.
- The `doDelete()` function uses `fetchWithCSRFToken()` to make a request to delete the article, and then calls the `onSuccess` function passed to the component if the API reports that the delete operation was a success.

We now have all the primitive pieces we need to start assembling an application.

A note about the use of `onSuccess` callbacks: by allowing the code that uses a component to pass in a callback function to trigger what happens when the operation is successful, our code is more reusable. This inversion of control will also make things like management of shared application state easier. We'll see this in action when we put all the pieces together.

Managing a list of content

In Use React to List Content from Drupal we demonstrated how you can retrieve a list of articles from Drupal's JSON:API and then store them in a state variable within a component using `useState`. Now that our application allows someone to add, edit, and delete items in the list, we need to account for this in our application's state management.

One option would be to force the component that displays the list to ping the API for a new copy of the list whenever an operation that changes its content occurs. This isn't the most efficient approach; it would require fetching the list quite frequently.

Instead, we can write logic into our application that updates the `content` state variable that corresponds with the changes that we make via the API. For example: make a `POST` request to add a new article, and shift a new item onto the array of articles we already have.

This has some performance benefits as it requires fewer requests to Drupal. We can make a single request to edit a node, and then update the node in the list we already have. Compare this to making a request to edit a node, and then a second request to retrieve the updated list.

We can also update the application state, while allowing the API request to happen asynchronously in the background. This makes the experience feel faster for the user. For example, when a user clicks the delete button we can remove the item from local state, and thus the UI, immediately, regardless of how long it takes to complete the roundtrip API request to Drupal.

Tie it all together

Let's tie all these components together into an actual application.

1

Create a `NodeReadWrite` component

Create new file *js/src/components/NodeReadWrite.jsx* with the following content:

```
import React, { useEffect, useState } from "react";
import NodeAdd from "./NodeAdd";
import NodeEdit from "./NodeEdit";
import NodeDelete from "./NodeDelete";

/**
 * Helper function to validate data retrieved from JSON:API.
 */
function isValidData(data) {
  if (data === null) {
    return false;
  }
  if (data.data === undefined ||
    data.data === null ||
    data.data.length === 0 ) {
    return false;
  }
  return true;
}

/**
 * Component for displaying an individual article, with optional admin features.
 */
const NodeItem = ({id, drupal_internal__nid, title, body, contentList, updateContent}) => {
  const [showAdminOptions, setShowAdminOptions] = useState(false);

  function handleClick(event) {
    event.preventDefault();
    setShowAdminOptions(!showAdminOptions)
  }

  function onEditSuccess(data) {
    // Replace the edited item in the list with updated values.
    const idx = contentList.findIndex(item => item.id === data.id);
    console.log('index', {idx, data, content: contentList});
    contentList[idx] = data;
    updateContent([...contentList]);
  }

  function onDeleteSuccess(id) {
    // Remove the deleted item from the list.
    const list = contentList.filter(item => item.id !== id);
    updateContent([...list]);
  }

  // Show the item with admin options.
  if (showAdminOptions) {
    return (
      <div>
        <hr/>
        Admin options for {title}
        <NodeEdit
          id={id}
          title={title}
          body={body.value}
          onSuccess={onEditSuccess}
        />
        <hr/>
        <button onClick={handleClick}>
          cancel
        </button>
        <NodeDelete
          id={id}
          title={title}
          onSuccess={onDeleteSuccess}
        />
        <hr/>
      </div>
    );
  }

  // Show just the item.
  return (
    <div>
      <a href={`/node/${drupal_internal__nid}`}>{title}</a>
      {" -- "}
      <button onClick={handleClick}>
        edit
      </button>
    </div>
  );
```

```
};

/**
 * Component to render when there are no articles to display.
 */
const NoData = () => (
  <div>No articles found.</div>
);

/**
 * Display a list of Drupal article nodes.
 *
 * Retrieves articles from Drupal's JSON:API and then displays them along with
 * admin features to create, update, and delete articles.
 */
const NodeReadWrite = () => {
  const [content, updateContent] = useState([]);
  const [filter, setFilter] = useState(null);
  const [showNodeAdd, setShowNodeAdd] = useState(false);

  useEffect(() => {
    // This should point to your local Drupal instance. Alternatively, for React
    // applications embedded in a Drupal theme or module this could also be set
    // to a relative path.
    const API_ROOT = '/jsonapi/';
    const url = `${API_ROOT}node/article?fields[node--article]=id,drupal_internal__nid,title,body&sort=-created&page[limit]=10`;

    const headers = new Headers({
      Accept: 'application/vnd.api+json',
    });

    fetch(url, {headers})
      .then((response) => response.json())
      .then((data) => {
        if (isValidData(data)) {
          // Initialize the list of content with data retrieved from Drupal.
          updateContent(data.data);
        }
      })
      .catch(err => console.log('There was an error accessing the API', err));
  }, []);

  // Handle updates to state when a node is added.
  function onNodeAddSuccess(data) {
    // Add the new item to the top of the list.
    content.unshift(data);
    // Note the use of [...content] here, this is because we're
    // computing new state based on previous state and need to use a
    // functional update. https://reactjs.org/docs/hooks-reference.html#functional-updates
    // [...content] syntax creates a new array with the values of
    // content, and updates the state to that new array.
    updateContent([...content]);
  }

  return (
    <div>
      <h2>Site content</h2>
      {content.length ? (
        <>
          <label htmlFor="filter">Type to filter:</label>
          <input
            type="text"
            name="filter"
            placeholder="Start typing ..."
            onChange={(event => setFilter(event.target.value.toLowerCase()))}
          />
          <hr/>
          {
            // If there's a `filter` apply it to the list of nodes.
            content.filter((item) => {
              if (!filter) {
                return item;
              }

              if (filter && (item.attributes.title.toLowerCase().includes(filter) || item.attributes.body.value.toLowerCase().in
                return item;
              }
            }).map((item) => (
              <NodeItem
                key={item.id}
                id={item.id}
                updateContent={updateContent}
                contentList={content}
                {...item.attributes}
              />
            ))
            }
          </>
        ) : (
          <NoData />
        )}
        <hr />
        {showNodeAdd ? (
          <>
            <h3>Add a new article</h3>
            <NodeAdd
              onSuccess={onNodeAddSuccess}
            />
          </>
        ) : (
          <p>
            Don't see what you're looking for?
            <button onClick={() => setShowNodeAdd(true)}>Add a node</button>
          </p>
        )}
      </div>
    );
};

export default NodeReadWrite;
```

To understand what's going on here it helps to first review the `NodeListOnly` component created in Use React to List Content from Drupal. `NodeListOnly` retrieves a list of articles nodes from Drupal's JSON:API, stores in them in the `content` state variable, and then loops over the list and outputs each one with `NodeItem`. The `NodeReadWrite` component above expands on that:

- Imports the new `NodeAdd`, `NodeEdit`, and `NodeDelete` components we created
- Adds a `showNodeAdd` state variable, and a `<button>` to toggle it on/off. When it's on, output the `NodeAdd` component and pass in an `onSuccess` callback that will add the new item to the existing `content` state using `updateContent`
- Passes the `updateContent` function as a prop to the `NodeItem` component so it can be used to update `content` when an individual item gets edited or deleted.
- Updates the `NodeItem` component and adds a `showAdminOptions` toggle that when `true` displays the `NodeEdit` form and the `NodeDelete` button. Both items have `onSuccess` callbacks that use the passed-in `updateContent` function to update the parent component (`NodeReadWrite`'s) `content` state variable to reflect changes to individual articles.

2 Output `NodeReadWrite` as the root of your application

Edit the *js/src/index.jsx* file, and update it so that it imports the `NodeReadWrite` component, and then outputs it:

```
ReactDOM.render(<NodeReadWrite />, document.getElementById('react-app'));
```

Finally, build the necessary JavaScript assets with `npm run build` or `npm run start`, and refresh the page in Drupal.

You should now see the interface we created above displayed on the page. Test it out by adding a new article, and editing or deleting existing ones.

useState or useReducer?

This example could use `useReducer` instead of `useState`.

If you've never used the `useReducer` hook before, we recommend reading Getting to Know the useReducer React Hook and learning about how `useReducer` differs from `useState`. As a rule of thumb, `useReducer` is beneficial when you need to compute new state based on existing state. For example, add a new item to a list.

Checkout the example code in `NodeReadWriteReducer.jsx`.

The biggest difference between this example and the one above is that you no longer need to pass the full `content` list down to the `NodeItem` component. In the `useState` example we need to access the list so we can update it. A reducer keeps track of the current content list internally. When the reducer gets called via the dispatcher there's no need to pass along the existing content list as we already know what it contains. This results in fewer props that need to be passed down the tree.

Try them both! There's no substitute for experience to help figure out which is right for your specific use-case.

Recap

In this tutorial, we created a React application that uses several components, and some vanilla JavaScript, to allow a user to add, edit, and delete nodes. This demonstrates how you might architect a more complex application by breaking UI elements up into individual components, as well as how to use HTTP `POST`, `PATCH`, and `DELETE` requests to interact with Drupal via the JSON:API module.

Further your understanding

- Can you add a pager to the `NodeReadWrite` to allow for displaying more than 10 items?
- Can you improve the error handling? For example, how would you handle the scenario where the Drupal backend is un-reachable?
- Rewrite the state handling logic to use `useReducer` instead of `useState`.

Additional resources

- Review the HTTP methods (developer.mozilla.org)
- Learn more about forms in React (reactjs.org)
- An imperative guide to forms in React (blog.logrocket.com)

Was this helpful? Yes No

◀ Previous tutorial

Next tutorial ▶

Get Started Using React and Drupal Together