Use React to List Content from Drupal

Last updated April 21, 2020

Theming Module Development 8.9.x/9.0.x

In order for our React code to list content from Drupal we'll need to enable the Drupal core JSON:API module, and then use `fetch()` in our React component to retrieve the desired data. This technique works for both React code embedded in a Drupal theme or module, and React code that is part of a fully decoupled application. We'll discuss the differences between those styles as well.

In this tutorial we'll:

- Use `fetch()` to bring data from Drupal into React
- Parse the data using ES6 array functions to find just the bits of data we need
- Combine multiple React components together to render a list of articles retrieved from Drupal

By the end of this tutorial, you should have a better understanding of how data from a Drupal API gets incorporated into a React application.

Goal

Use React to retrieve a list of nodes from Drupal's API and display them in a React component.

Prerequisites

- We'll build on the code started in Connect React to a Drupal Theme
- Create a React Component
- A Drupal site with JSON:API module installed and *article* nodes created
- Retrieve Data from an API with React

Enable JSON:API

To make use of content provided by Drupal in React, first enable the core JSON:API module, and then make a request to the API to retrieve the data. The JSON:API module requires zero configuration to get started, all you need to do is make sure that it's enabled. From here on we'll assume that it's enabled.

For a much more in-depth look at the Drupal core JSON:API module and its features see our Web Services in Drupal 8 series.

Example code

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

Display a list of Drupal nodes with React

Let's write some React code that'll list *Article* nodes from our Drupal site in the sidebar and allow a user to filter that list via a text field.

This assumes that JSON:API is enabled, you've created some *Article* nodes, and you've already set up a toolchain and "Hello, World" component like we did in Connect React to a Drupal Theme or Module.

1 Scaffold the new React components

Create the file *react_example_theme/js/src/components/NodeListOnly.jsx* with the following content:

```
import React, { useEffect, useState } from "react";

const NodeItem = () => (
  <div>Node item placeholder</div>
);

const NoData = () => (
  <div>No articles found.</div>
);

const NodeListOnly = () => {
  const [content, setContent] = useState(false);

  return (
    <div>
      <h2>Site content</h2>
      {content ? (<NodeItem />) : (<NoData />)}
    </div>
  );
};

export default NodeListOnly;
```

Then update the *react_example_theme/js/src/index.jsx* file to use our newly created `NodeListOnly` component:

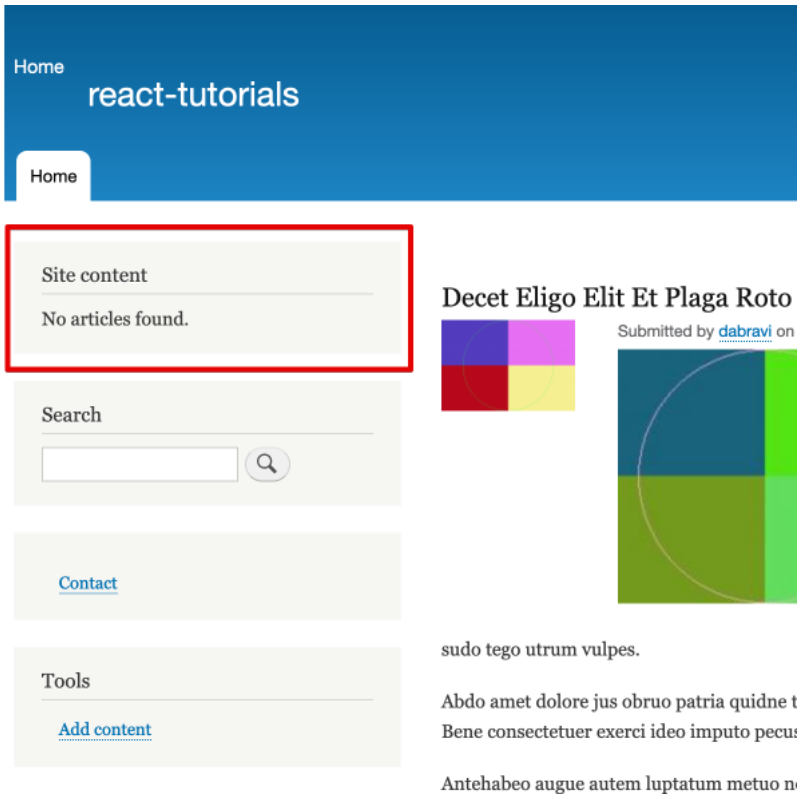
```
import React from 'react'
import ReactDOM from 'react-dom'
import NodeListOnly from "./components/NodeListOnly";

const Main = () => (
  <NodeListOnly />
);

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

This scaffolds a component to list node content `NodeListOnly`, and sets up 2 placeholder components `NodeItem` and `NoData`. It then outputs the new `NodeListOnly` component as the root of our application. The `NodeListOnly` component will use the `content` state variable to keep track of data loaded from Drupal.

We chose `NodeListOnly` for the component name because this one is read-only. Later we'll create a read-write list component.

2 Verify it worked

Run `npm run start` in the root directory of your theme to compile the new JavaScript and refresh the page in your browser to verify it's working. You should see something like the following:



3 Connect to JSON:API and get a list of nodes

Here's the code we'll use to request data from Drupal's JSON:API:

```
// Either the full path to your API endpoint, or a relative path for React
// applications embedded in a Drupal theme or module.
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) => console.log(data))
  .catch(err => console.log('There was an error accessing the API', err));
```

In most browsers you should be able to run this code directly in the console while viewing any page of your Drupal site.

This fetches content from Drupal's JSON:API. Specifically, it fetches the id, internal node id, title, and body of 10 article nodes in reverse chronological order based on creation date. Then it parses the response into a JSON object, and then logs that to the console. In the case of error, it catches any errors that occur and logs those. While the error handling could be more robust, this is all that's required to retrieve data about a list of Article nodes from Drupal.

Note the use of sparse fieldsets in our API query. This limits the content of the request to the data we're interested in rather than complete node objects. We also set the `Accept` header to `application/vnd.api+json` so that we can get better JSON:API error handling.

This will result in a response that looks something like the following:

```
{
  "jsonapi": { ... },
  "data": [
    {
      "type": "node--article",
      "id": "41b25a44-ddf9-4b7e-aeba-071dfa3beffb",
      "links": {
        "self": {
```

```
          "href": "https://localhost:8181/jsonapi/node/article/41b25a44-ddf9-4b7e-aeba-071dfa3beffb?resourceVersion=id%3A9"
        }
      },
      "attributes": {
        "drupal_internal__nid": 2,
        "title": "Abdo Plaga Rusticus Saepius Zelus",
        "body": {
          "value": "Abdo haero meus uxor ...",
          "format": "plain_text",
          "processed": "<p>Abdo haero meus uxor ...",
          "summary": "Abdo haero meus uxor ..."
        }
      }
    },
    {
      "type": "node--article",
      "id": "d0a5b576-1aa0-4461-9be3-6f7ea2690a37",
      "links": {
        "self": {
          "href": "https://localhost:8181/jsonapi/node/article/d0a5b576-1aa0-4461-9be3-6f7ea2690a37?resourceVersion=id%3A10"
        }
      },
      "attributes": {
        "drupal_internal__nid": 1,
        "title": "Hos Ideo Quidem Roto Turpis Voco",
        "body": {
          "value": "Cogo esse feugiat incassum  ...",
          "format": "plain_text",
          "processed": "<p>Cogo esse feugiat incassum ...",
          "summary": "Cogo esse feugiat incassum ..."
        }
      }
    },
    ...
  ],
  "links": { ... }
}
```

The content we're interested in is in the `data` key.

4 Fetch when the `NodeListOnly` component mounts

We can take the above code and add it to our `NodeListOnly` component so that when the component mounts (the React term for "is added to the page") it'll start fetching data from Drupal.

```
const NodeListOnly = () => {
  const [content, setContent] = useState(false);

  useEffect(() => {
    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) => setContent(data.data))
      .catch(err => console.log('There was an error accessing the API', err));
  }, []);

  return (
    <div>
      <h2>Site content</h2>
      {content ? (<NodeItem />) : (<NoData />)}
    </div>
  );
};
```

The main difference here is instead of logging the retrieved data to the console we're using the `setContent` function to update the component's `content` state variable with the results of our API query. That will cause a re-render of the component due to the state change, and in the JSX returned it'll now show `NodeItem` instead of `NoData`.

This is all wrapped in a `useEffect()` with an empty array as the second argument to ensure that the API request only happens the first time `NodeListOnly` renders -- not on any re-renders.

5 Iterate over the data and output a component

Let's modify the `return` value of `NodeListOnly` to iterate over the `content` state variable when it contains data to display, and use the `NoData` component if no data is found.

```
return (
  <div>
    <h2>Site content</h2>
    {content ? (
      content.map((item) => <NodeItem key={item.id} {...item.attributes}/>)
    ) : (
      <NoData />
      )}
  </div>
);
```

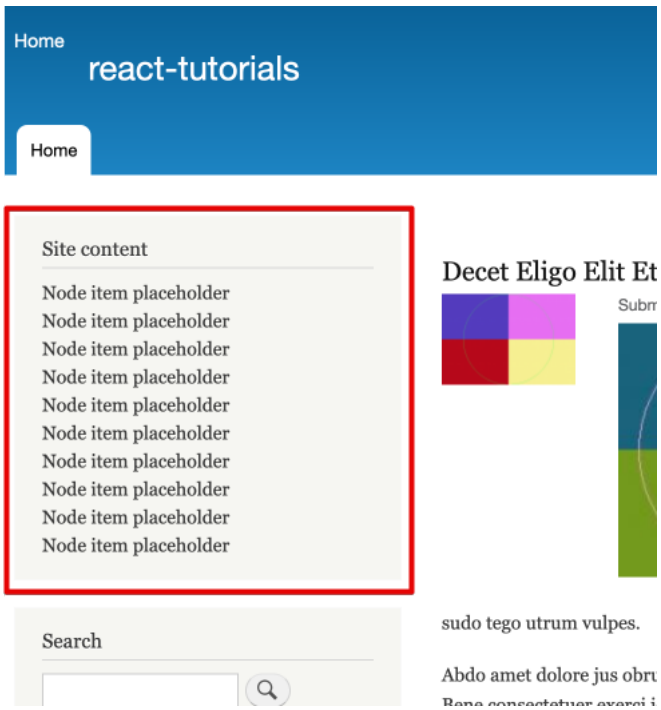
This says that if `content` is not `false` then use `.map` function to iterate over the array and pass each `item` into the `NodeItem` component.

The `...` operator takes the attributes of the `item`, and spreads them as props onto the `NodeItem` component. React also likes to have a unique `key` set for every item in a list, so we use the UUID of the article to do that.

Again, because our `fetch` logic put the data into the component's `content` state variable, the component will automatically re-render when `content` changes. (That's the magic of React!) Since `content` is no longer `false` it can map through our data and output a `NodeItem` component. If not, it will show `NoData`.

If you refresh the page in your browser after making this change you should see the `NodeItem` component rendered once for each article. It's not displaying the actual content yet.

Example:



6 Update `NodeItem` to display an article's title

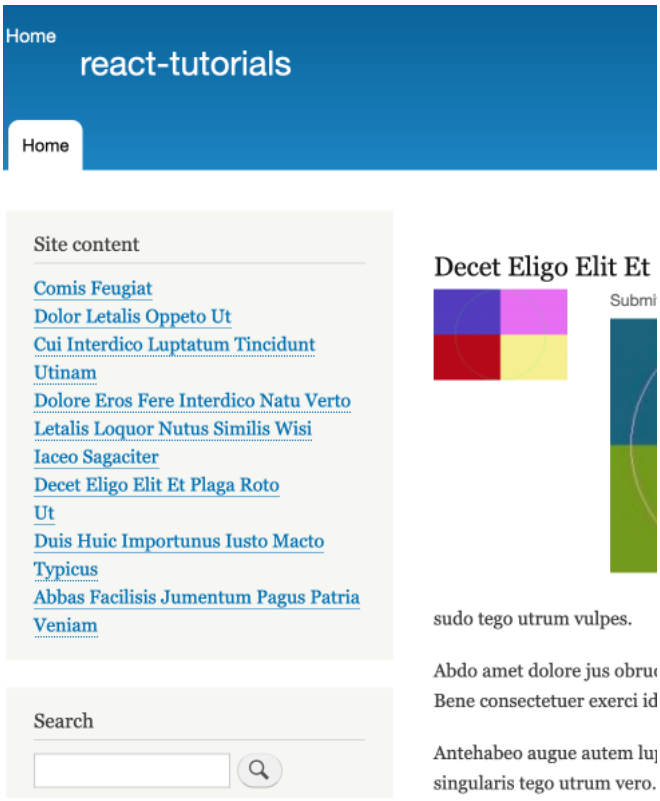
Our `NodeItem` component gets used once for each node returned from JSON:API and the Drupal article node's content gets passed in as props. Let's render the article's title as a link to the article.

```
const NodeItem = ({drupal_internal__nid, title}) => (
  <div>
    <a href={`/node/${drupal_internal__nid}`}>{title}</a>
  </div>
);
```

We have passed all the props, which in this case is the set of fields we specified in our JSON:API query, into `NodeItem` and now we can access them. In our case we need the `drupal_internal__nid`, and the `title` so we destructure those out of the props passed to the component.

Refresh your browser and you should see a list of your nodes. Clicking on an item in the list should take you to that article.

Example:



There's more we can do to make this code more robust, and add some interactivity.

Validate the API data before using it

Sometimes the API structure changes, or a field is missing or null. It helps to validate your data before you pass it to a component for rendering.

Here is a sample validator function. You can add this anywhere in your *NodeListOnly.jsx* file, and then add a conditional in the `fetch` logic to update the component's state if the data is valid. This simplified example verifies that `data.data` isn't empty. We recommend you check for the existence of specific values, like `title` or `body.summary`, and provide a default option or `null` value so that your component will not error out when trying to make use of data it expects to be present but is not.

```
function isValidData(data) {
  if (data === null) {
    return false;
  }
  if (data.data === undefined ||
    data.data === null ||
    data.data.length === 0 ) {
    return false;
  }
  return true;
}
```

Use this to validate the retrieved data by updating the `fetch` call in `NodeListOnly`:

```
fetch(url, {headers})
  .then((response) => response.json())
  .then((data) => {
    if (isValidData(data)) {
```

```
      setContent(data.data)
    }
  })
  .catch(err => console.log('There was an error accessing the API', err));
```

Add a text field to allow users to filter the list

We can add some interactivity to our `NodeListOnly` component, and demonstrate the magic of React's state system. We'll add an `<input>` field that you can type into, and in real-time use the value entered to filter the list of articles to those that have the input value in either their title or body.

Add another state variable to `NodeListOnly`:

```
const [filter, setFilter] = useState(null);
```

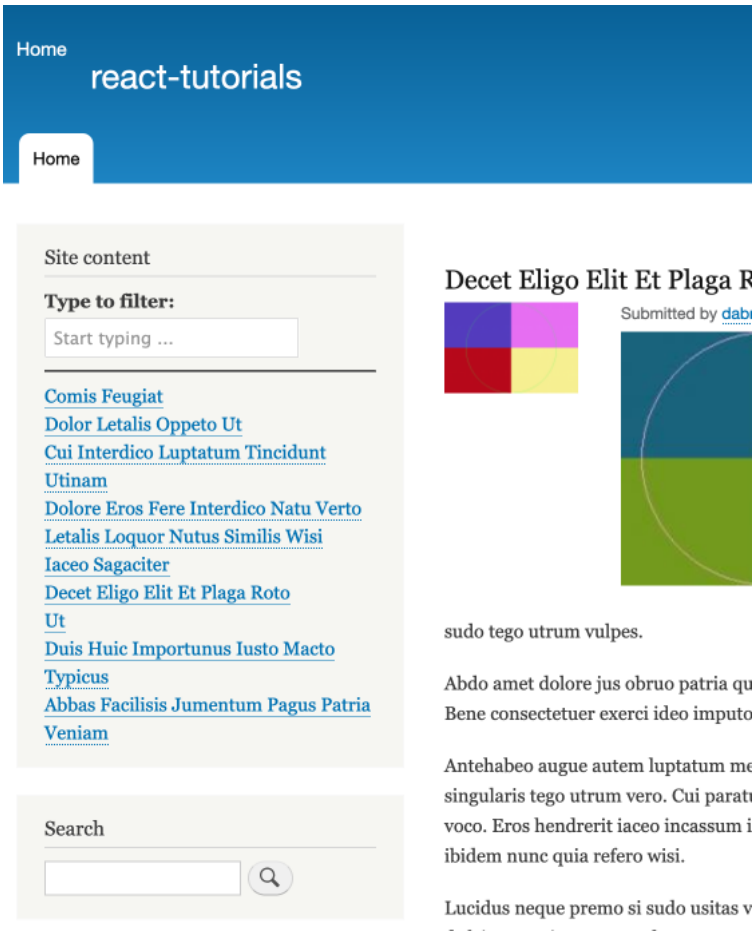
Then update the return statement to the following:

```
return (
  <div>
    <h2>Site content</h2>
    {content ? (
      <>
        <label htmlFor="filter">Type to filter:</label>
        <input
          type="text"
          name="filter"
          placeholder="Start typing ..."
          onChange={(event => setFilter(event.target.value.toLowerCase()))}
        />
        <hr/>
        {
          content.filter((item) => {
            if (!filter) {
              return item;
            }

            if (filter && (item.attributes.title.toLowerCase().includes(filter) || item.attributes.body.value.toLowerCase().includes(filter
              return item;
            }
          }).map((item) => <NodeItem key={item.id} {...item.attributes}/>)
        }
      </>
    ) : (
      <NoData />
      )}
  </div>
);
```

This adds a new `<input>` element, with an `onChange` callback that's called every time the value of the field changes. In this callback we update the new `filter` state variable with the content from the `<input>` field by calling `setFilter(event.target.value.toLowerCase())`. This will trigger a re-render, because we've updated the component's state.

Then, we use `Array.filter()` to filter the value of `content` to only those items whose title, or body, contain a match for the value in `filter`, making a quick and dirty real-time search.

The final result should look like this:



A note about permissions

Try this: create a new article node and leave it unpublished. Then refresh the page showing your `NodeListOnly` component from above. Is the unpublished article in the list or not? What if you log out and open the page again?

Because all the JavaScript we've written here is embedded in the page that Drupal is serving, it -- most importantly, the `fetch()` calls in it -- uses the same session/cookies our browser uses for other interactions with this site. Calling `fetch()` sends along the browser's cookies with the request. Drupal uses this to authenticate your session, and the response from JSON:API will contain data that whatever user you're logged in as has access to.

This is important to keep in mind, because this behavior differs when we get into fully decoupled React applications later.

Recap

In this tutorial, we used the JavaScript Fetch API to retrieve a list of nodes from Drupal's JSON:API and then displayed them with React.

Further your understanding

- Play around with changing your API or creating a new endpoint with more data and loading that data into React.
- Observe how changes in the interface between your API and front-end might crash your application. Can you add more error handling? Can documenting your API help? The issues that come up here are similar to what you will face in working with a team on a real-world project.
- Can you add a "Read more" link to the `NodeItem` component that, when pressed, toggles display of the article's summary?

Additional resources

- To understand the `{...}` syntax, learn about the destructuring assignment operator (developer.mozilla.org)
- Pretty component syntax (medium.com)

Was this helpful? Yes No

◀ Previous tutorial

Next tutorial ▶

Get Started Using React and Drupal Together

Contact us

Powered by: