Create a React Component

Last updated March 25, 2020

Theming Module Development 8.9.x/9.0.x

Components are the fundamental building blocks of any React application. React uses components to represent different elements in the UI. To show this, we'll build a React widget that can query the Drupal.org REST API to retrieve usage statistics for a project and then display them. We'll create buttons that allow us to toggle between two different projects. In doing so we'll learn about creating components and using *props* and *state* in React.

In this tutorial we'll:

- Define two new React components
- Learn about using *props* to pass data to a component
- Learn about using *state*, and the *useState()* hook, to create interactivity

By the end of this tutorial you should have a basic understanding of how to write a React component that uses *props* and *state* to display data from a third party API.

Goal

Define two new React components, `DrupalProjectStats` and `StatsItem`, that when used together can display data about a Drupal.org project's usage.

Prerequisites

```
import React, { useState, useEffect } from "react";
import PropTypes from "prop-types";

const DrupalProjectStats = ({ projectName }) => {
  const [project, setProject] = useState(projectName);
  const [usage, setUsage] = useState(null);

  useEffect(() => {
    setUsage(false);
    const data = fetch(
      `https://www.drupal.org/api-d7/node.json?field_project_machine_name=${project}`
    )
      .then(response => response.json())
      .then(result => {
        if (result.list[0].project_usage) {
          setUsage(result.list[0].project_usage);
        }
      })
      .catch(error => console.log("error", error));
  }, [project]);

  return (
    <div>
      <div>
        Choose a project:
        <br/>
        <button onClick={() => setProject('drupal')}>Drupal core</button>
        <button onClick={() => setProject('marquee')}>Marquee</button>
      </div>
      <hr />
      <div className="project--name">
        Usage stats for <strong>{project}</strong> by version:
        {usage ? (
          <ul>
            {Object.keys(usage).map(key => (
              <StatsItem count={usage[key]} version={key} key={key} />
            ))}
          </ul>
        ) : (
          <p>fetching data ...</p>
        )}
      </div>
    </div>
  );
};

// Provide type checking for props. Think of this as documentation for what
// props a component accepts.
// https://reactjs.org/docs/typechecking-with-proptypes.html
DrupalProjectStats.propTypes = {
  projectName: PropTypes.string.required
};

// Set a default value for any required props.
DrupalProjectStats.defaultProps = {
  projectName: 'drupal',
};
```

- The code in this tutorial builds on the code from Connect React to a Drupal Theme

Create a component that displays Drupal project usage

The end result we're aiming for is a React widget that (when displayed) shows usage statistics for each version of a project on Drupal.org, as well as buttons to toggle between two different projects.

Example:

Create a React widget

Note: The following example code assumes you're using a JavaScript build toolchain as described in Connect React to a Drupal Theme or Module. If you want this to work without a build toolchain and are directly including React via the `react_example_theme/react_external` asset library, you'll need to write the code into the *js/index.js* file and comment out or remove any `import` statements.

Define a `DrupalProjectStats` component

Create a file called *react_example_theme/js/src/components/DrupalProjectStats.jsx* with the following content:

```
/**
 * Another component: this one displays usage statistics for a specific version
 * of Drupal. It's not exported, so it can only be used in this file's scope.
 * Breaking things up like this can help make your code easier to maintain.
 */
const StatsItem = ({ count, version }) => (
  <li>
    <strong>{version}</strong>: {count}
  </li>
);

StatsItem.propTypes = {
  count: PropTypes.number,
  version: PropTypes.string,
};

export default DrupalProjectStats;
```

Let's break down what the code in this file does:

`import React, { useState, useEffect } from "react";` Declares that this file contains React-specific code, and imports the `useState` and `useEffect` hooks from the React library. Since we're using a build toolchain these `import` statements will be resolved and the linked libraries bundled into our application.

`const DrupalProjectStats = ({ projectName }) => {}` Creates a new React component named `DrupalProjectStats`. A component can be either a function (like this one) or a class that extends `React.Component`. The function performs any internal logic, and then returns the JSX that we want React to render as HTML. Once defined, we can use our component in a JSX statement like so:

```
<DrupalProjectState projectName="drupal" />
```

When React encounters that JSX tag it'll trigger our `DrupalProjectState` function and pass the props (those things that looks like HTML tag attributes) as arguments to the function. The `projectName` variable in this case would have a value of `"drupal"`.

```
const [project, setProject] = useState(projectName);
const [usage, setUsage] = useState(null);
```

This defines two new state variables, and related functions for modifying their value. In the first statement we're setting the default value of the `project` state variable to whatever the `projectName` prop contains. By default, our code will display stats for the `"drupal"` project.

We use state for variables whose value can change over time, and that when changed should trigger the component to re-render with the new data. Essentially, we use state to make our components interactive. Learn more about state in React.

```
useEffect(() => {
  setUsage(false);
  const data = fetch(
    `https://www.drupal.org/api-d7/node.json?field_project_machine_name=${project}`
  )
    .then(response => response.json())
    .then(result => {
      if (result.list[0].project_usage) {
        setUsage(result.list[0].project_usage);
      }
    })
    .catch(error => console.log("error", error));
}, [project]);
```

This code uses the JavaScript `fetch()` API to retrieve JSON data from the given URL, and when successful extracts a subset of the returned data and uses the `setUsage()` function to update the value of the `usage` state variable. Note the use of `?field_project_machine_name=${project}` which makes the URL change depending on the value of the `project` state variable. Here's a truncated example of the data returned for `project="drupal"`:

```
{
  "list": [
    {
      // ... snip ...
      "title": "Drupal core",
      "created": "1064766675",
      "changed": "1571223050",
      "project_usage": {
        "6.x": "36391",
        "7.x": "714295",
        "8.x": "313748"
      }
    }
  ]
}
```

This code is wrapped by `useEffect(() => {}, [project])`. The `useEffect()` hook tells React that your component needs to do something that can cause side effects, for example changing the component's state. Think of side effects as anything that can cause the component to re-render. Rendering can happen when a component is first mounted, or when either the props or state change.

This could result in an infinite loop where the component is first mounted, then loads data via `fetch()` and updates the `data` state variable, which triggers a re-render, which causes the `fetch()` logic to execute again, which updates the `data` state variable again, and on, and on. To prevent this we use the `useEffect()` hook.

React remembers the function you passed to `useEffect()` and calls it after performing any DOM updates. That is, any time the component is re-rendered. The second argument to `useEffect()` is used to skip running the effect function unless the value of the passed in variable(s) changes. In this example, that means that no matter how many times the component is re-rendered the `fetch()` logic inside `useEffect()` won't be called a second time unless the value of `project` changes. In that case it'll call the function, get the updated data, and then display that.

We return a JSX statement from the function. The two interesting bits are:

```
<button onClick={() => setProject('drupal')}>Drupal core</button>
```

The two buttons allow toggling to stats for a different project. When you press this button the `onClick` handler calls the `setProject()` state manipulation function we created earlier and passes in a project name. This effectively changes the value of `project`, and more importantly, causes a rerender of the component. That is, `DrupalProjectState` function is called again; this time the `project` variable has a new value. That causes the `useEffect()` memoized logic to run again, which gets new data from the API and updates `usage`, which also triggers another rerender. This time, `project` hasn't changed so the `fetch` code isn't called again. This bit is:

```
{usage ? (
  <ul>
    {Object.keys(usage).map(key => (
      <StatsItem count={usage[key]} version={key} key={key} />
    ))}
  </ul>
) : (
  <p>fetching data ...</p>
)}
```

Assuming `usage` has a value, that code will loop through the results returned from the API and display them using the `StatsItem` component defined later in the file.

Learn more about using `fetch` with React.

Pull it all together

Edit the *react_example_theme/js/src/index.jsx* file so it looks like this:

```
import React from 'react'
import ReactDOM from 'react-dom'

/* Import Components */
import DrupalProjectStats from './components/DrupalProjectStats';

const Main = () => (
  <DrupalProjectStats projectName="drupal" />
);

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

This imports our new `DrupalProjectStats` component, and then uses it in a JSX expression inside the `Main` component. `ReactDOM.render()` is then used to bind our React code to the DOM element with the id of `react-app`.

Run `npm run start`, or `npm run build:dev` to compile the updated JavaScript code, and refresh the page in your browser to see it in action. Then open up your browser's network inspector and notice that when you press the buttons to toggle between projects it makes a new request. Also notice that `fetch` is smart about caching the data. Cool!

Recap

This is a quick introduction to React, and some fundamental concepts. Hopefully it demonstrates the power of React and its handy ability to rerender when an application's state changes.

In this tutorial, we created two new React components that, combined together, allow us to retrieve data about a project's usage statistics from the Drupal.org API and display them on the page. In doing so we learned how to define a new component, what props and state are in a React component, how to use hooks to modify state and create interactivity, and how to import components from one file into another to make use of them.

Further your understanding

- Create another component named `Button` that replaces the `<button>` elements currently used. Use props to allow the parent component to dictate what the button does, and what the text says.

- Replace the project selection buttons with a `<select>` list and a larger number of projects to choose from.
- Use an animation library like `react-spring` to animate the transition from showing stats for one project to showing stats for another.

Additional resources

- Conditional rendering (reactjs.org)
- Handling events (reactjs.org)

Was this helpful? Yes No

◀ Previous tutorial

Next tutorial ▶

Get Started Using React and Drupal Together

8 Use React to List Content from Drupal

9 Create, Update, and Delete Drupal Content with JavaScript

10 Build an Interface to Edit Nodes with React

11 Create a Fully Decoupled React Application Free

12 Use create-react-app to Start a Decoupled React Application

13 Make API Requests with OAuth

14 Use Fetch and OAuth to Make Authenticated Requests