Exploration — Routing & Forms

Introduction

In this exploration, we will study how to write Single Page Applications (SPAs) using the React Router. A second topic we will study is how to build forms in React apps.

SPAs and React

A major use for React is building SPAs. Recall that with SPAs the HTML, CSS, and JavaScript for a web app are sent from the server to the browser exactly once. After that when the user navigates to another page in the app, the client-side code makes changes to the DOM so that the user feels that a new page has been loaded in the browser. When writing React apps, we can use the React Router library to create SPAs. React Router supports defining routes in our React app, i.e., we can define how to map URLs to resources. When the user clicks a link in the app, the React app already loaded in the browser uses the React Router library to map this request to a new component and renders a different DOM tree completely within the browser, without sending the request to a server.

Versions of React Router

The example code in this exploration uses version 5.x for React Router and we recommend that you also use version 5.x. The newer version 6.x of React Router includes some breaking changes from version 5.x. If you use version 6.x, you will need to make changes to the code examples. See the page **Upgrading from V5 (https://reactrouter.com/docs/en/v6/upgrading/v5)** for more information about these updates and see **React Router Overview (https://reactrouterdotcom.fly.dev/docs/en/v6/getting-started/overview)** to see sample code that uses version 6.

What happens if you use create a new React app using the command `npx create-react-app` and then install React Router by running the command `npm install react-router-dom`? In this case, React Router version 6.x will be installed. However, if you instead run the command `npm install react-router-dom@5` then version 5 will be installed. See the **docs for npm install (https://docs.npmjs.com/cli/v8/commands/npm-install)** for more details about this.

How to Determine Version of a Package?

You can verify the (semantic) version of a package your app is using by examining the file `package.json` and looking at `dependencies`. As an example, the following snippet from a `package.json` file shows that the app is using version 6 of `react-router-dom`.

```
  "dependencies": {
    "@testing-library/jest-dom": "^5.12.0",
    "@testing-library/react": "^11.2.6",
    "@testing-library/user-event": "^12.8.3",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-icons": "^4.2.0",
    "react-router-dom": "^6.2.1",
    "react-scripts": "4.0.3",
    "web-vitals": "^1.1.2"
  },
```

You can change the version of a package used by your app by using `npm install package-name@version`. For example, we can update the app with the `package.json` file shown above to use version 5 of React Router by

- Shutting down the app,
- Running `npm install react-router-dom@5`.

Example React App

> The SPA React app is available in this file: **spa.zip (https://canvas.oregonstate.edu/courses/1879154/files/94331856?wrap=1)** (https://canvas.oregonstate.edu/courses/1879154/files/94331856/download?download_frd=1) . This app uses version 5 of the React Router version 5 and version 17 of React and React Dom.

To understand the use of React Router to build SPAs, we will build a website that contains the following 3 pages:

1. Home Page at the URL `http://localhost:3000/`
2. Blog Page at the URL `http://localhost:3000/blog`
3. Contact Page at the URL `http://localhost:3000/contact`

Each page in the website is a simple component that includes an `h2` heading with the name of the page. For example, here is the code for the Home Page which we will store in the file `src/pages/HomePage.js`.

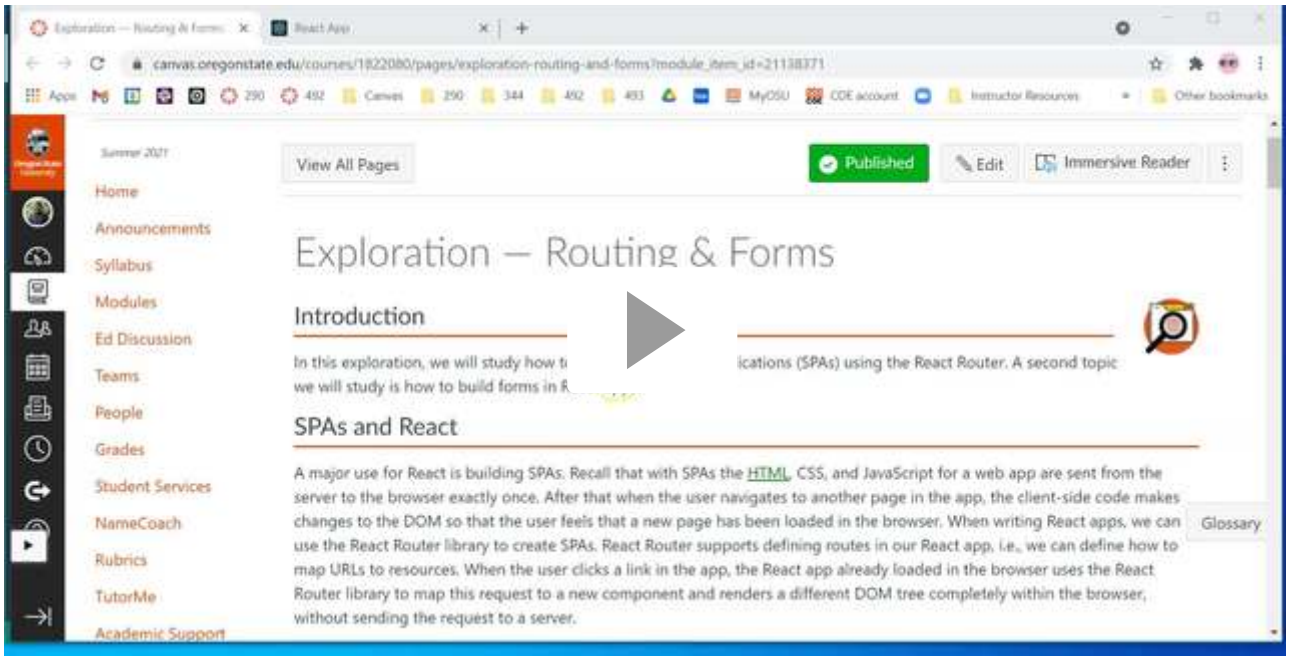
```
import React from 'react';

function HomePage() {
  return (
    <>
      <h2>HomePage</h2>
      <p>This app is all about...</p>
    </>
  );
}

export default HomePage;
```

We create similar files for the Blog Page and Contact Page with names `BlogPage.js` and `ContactPage.js`. We place those two files in the directory `src/pages` as well.

Using BrowserRouter and Route Components



To use the React Router, we install the library `react-router-dom` using `npm install`. We use the components `BrowserRouter` and `Route` from this library. Here is our `App.js` file with the routing set up for our website.

```
import './App.css';
import { BrowserRouter as Router, Route } from 'react-router-dom';
import HomePage from './pages/HomePage';
function App() {
  return (
    <>
      <Router>
          <main>
            <Route path="/" exact>
              <HomePage />
            </Route>
            <Route path="/blog">
              <BlogPage />
            </Route>
            <Route path="/contact">
              <ContactPage />
            </Route>
          </main>
        </Router>
    </>
  );
}
```

```
export default App;
```

We highlight a few things in our `App` component related to routing:

- We have imported the component `BrowseRouter` and use the name `Router` for it. This is a typical convention in React apps.
- We place the `Router` component near the root of the component tree.
- For each page in our website, we include one `Route` component within the `Router`.
- The component we want displayed for a URL is placed inside the `Route` component with that URL as the value of the `path` component.
  - E.g., to display the Contact Page at `/contact`, we specify `/contact` as the value of `path` of the `Route` component and nest `ContactPage` component inside this `Route` component.
- By default, the path in a route matches any URL that starts with that path.
  - E.g., a route with `path="/"` will match the URL `/blog` and `/contact` because both these URLs start with a `/`.
- However, if a route has the property `exact`, then the route will match a URL only if the URL is an exact match for the value of `path`.
  - In our `App`, we use the property `exact` for the route for the path `/`, i.e., `<Route path="/" exact>`. If we had not done this, all the URLs would have been routed to the Home Page.

If we now run an app created with the `App.js` file and the 3 files for our 3 pages from the previous section, it will route the pages of our website as we wanted, i.e.,

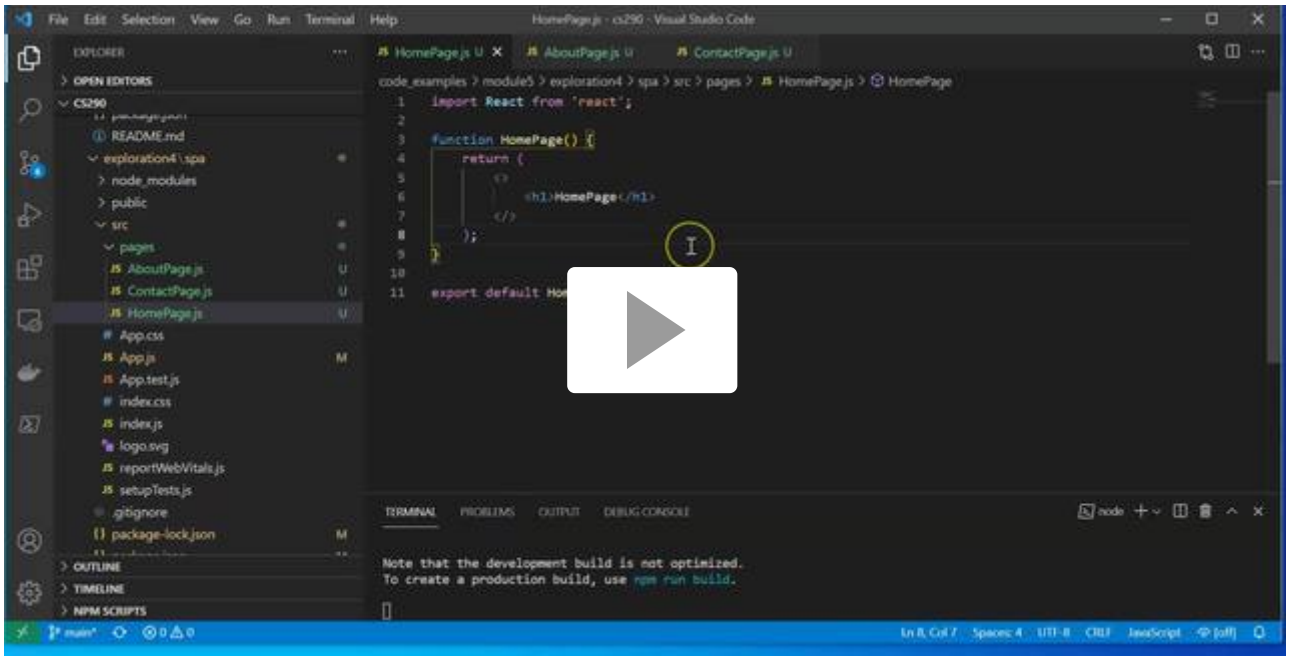
- The URL `http://localhost:3000/` will display the HomePage.
- The URL `http://localhost:3000/blog` will display the BlogPage.
- The URL `http://localhost:3000/contact` will display the ContactPage.

Here is how our app works:

- When we enter the URL for any page of our website in the browser, a request is sent to the server running on `localhost:3000` for this page.
- The routing for these pages is being done in the React code, not in any code running on the server-side.
- So the response sent by the server includes the files for our React app, rather than the specific page that was requested.
- When the app is rendered, the React Router in the `App` component matches the URL in the browser to the `Route` component with the matching path and renders the component nested in this route.

Our app is now doing the routing on the client-side, i.e., in the browser. However, this app is not a SPA. This is because whenever the user wants to view a different page, a request is still being sent to the server and the page is displayed after downloading the files for the React app from the server.

Using the Link Component



To create a SPA, we must also use the `Link` component provided by the React Router. The `Link` component has a property named `to` whose value is a URL. When we click that URL, the browser navigates to the page corresponding to that URL. This behavior may seem exactly the same as the behavior of the `a` anchor element in HTML. However, there is one important difference between the `Link` component and the `a` anchor element.

- When we click the URL in an `a` element, if the URL corresponds to a new page (rather to an anchor in the same page), then a request is sent to the *server*, the new page is *received* in the response and is then *rendered* in the browser.
- When we click the URL in a `Link` component, the React app *changes the DOM tree* entirely on the *client-side*, i.e., in the browser, without needing to talk to the server. The browser shows the new URL as the address of the page...but the page has not been loaded from the server. The React app has simply updated the DOM tree in the page!

Example: Using Link Component

In the following example, we show the code of a global Navigation component which has three `Link` components; one each for the HomePage, BlogPage, and ContactPage:

```
import React from 'react';
import { Link } from 'react-router-dom';

function Navigation() {
```

```
  return (
    <>
      <nav>
         <Link to="/">Home</Link>
         <Link to="/blog">Blog</Link>
         <Link to="/contact">Contact</Link>
      </nav>
    </>
  );
}

export default Navigation;
```

The first `Link` component has a hyperlink to the HomePage, which is the default index file, represented by /. The second `Link` has a hyperlink to the BlogPage with the URL `/about`, while the third `Link` component has a hyperlink to the ContactPage with the URL `/contact`. To verify that our app is now really working as a SPA, start the app and go to the Home Page. Now open the browser's Developer Tools and go to the Network tab. If we click on either of the three links, we will see that:

- The URL in the browser is updated based on the hyperlink we clicked.
- The page corresponding to that URL is rendered in the browser.
- There is no network traffic, i.e., the browser does not send any request in order to display the new page.

When we add the correlating <Navigation /> component to the App.js file, then users can link from page to page of the app.

Exercise: Link Component vs. the anchor element

Change the component `HomePage` and replace each `Link` component with an HTML <`a`> element that hyperlinks to the same URL as the corresponding `Link` element.

1. If you click either of the two hyperlinks in the `<a>` anchor elements, do you think any requests and responses will be sent over the network?
2. Now open the browser's Developer Tools and go to the Network tab.
3. Click the hyperlinks in the `<a>` elements and observe the Network traffic.
4. Explain the behavior that you observe and verify whether your answer to the question asked in 1st item of this list was correct or not.

Forms

Building forms in React apps is somewhat similar to building forms using pure HTML. All the HTML tags related to forms are available for use in JSX. However, we need to use **React state variables** to manage the values entered in the form by the user. This means that we need to use the `useState` Hook.

Example: Form in a React App Page

As an example, we can create a form in the ContactPage of the app using the typical HTML elements such as `fieldset, legend, label, input`, and `button`. The following example takes name `input` from the user and submits it with a button to a JavaScript alert:

```
import React, { useState } from 'react';

function ContactPage() {
  const [name, setName] = useState('');
  return (
    <>
      <h2>Contact Us</h2>
      <p>Contact using this secure form below:</p>
        <form>
          <fieldset>
            <legend>Your Details</legend>
            <label>Please enter your name
              <input type="text" value={name}
                onChange={e => setName(e.target.value)} />
            </label>
          </fieldset>
          <button onClick={e => {
            setName(e.target.value);
            alert(`Your name is ${name}`);
            e.preventDefault();
          }}>Submit</button>
        </form>
    </>
  );
}

export default ContactPage;
```

To explain:

- In order to manage the value the user enters in the `input` element, we define a state variable `name`.
- To set the value of `name`, we use `setName` which is the function at index 1 in the array returned by `useState`.
- Whenever the text in the `input` element changes, the `change` event is raised.
- In the handler for this event, we call `setName` with the current value of the `input` element.
  - We obtain the current value via the property `e.target.value`.
  - Here `e` is the event raised when the text changes and it is the argument to the event handler.
  - `e.target` is the element on which the event was raised, i.e., the `input` element.
  - `e.target.value` is the current value of this element.
- When the user hits the “Submit” button, we create an alert and display the value entered by the user.
- Note the use of `e.preventDefault()` in the handler for the submit button.
  - The default behavior for form submission is sending a GET request to the current URL in the browser.

- If we do not prevent the default behavior an HTTP request would have been sent to this URL.

Summary

In this exploration, we learned how to can create React SPAs using the React Router. All the client-side rendering in an SPA does not involve the server beyond the initial download of the app. Later in the course, we will study how to connect our SPAs with server-side programs.

Using forms in React is the other topic we studied in this exploration. We looked at how the `useState` Hook is used to manage the input values in the form. This pattern of managing the input via `useState` in React apps is called **Controlled Forms** (or Controlled Components). Some React apps use a different pattern for forms which is called **Uncontrolled Forms** (or Uncontrolled Components). In Uncontrolled Forms, the React app directly accesses the DOM nodes. The use of Controlled Forms is the recommended practice for modern React apps.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- **The React Router on npm** (https://www.npmjs.com/package/react-router-dom).
- For more info on the React Router **see the guide** (https://reactrouter.com/web/guides/quick-start).
- Discussion of **controlled** (https://reactjs.org/docs/forms.html#controlled-components) and **uncontrolled** (https://reactjs.org/docs/uncontrolled-components.html) forms on the React website. However, be aware that the examples in these docs use old-style class-based components, rather than the newer function-based components that we are using in this course.