React.js

> Course: Essential React.js by Eve Porcello on LinkedIn Learning
> Notes by Nolan Zurek

Introduction: What is react

What is react?

- Very popular library for building UIs
- React is a JS library
- Originally created at facebook, now open-source
- React Native lets you create native mobile applications
- Documentation can be found at react docs

React Developer Tools

- Installed as an extension from the chrome web store
- Command to open: control-shift-j (opens console) → components
- If react is running on the page, a react icon will appear in the address bar

Introduction to React elements

Creating a react app

- Building a project requires starting with a number of files
- We can build it using the following command in the terminal (requires Node.js and NPM)

```
npx create-react-app app-name
```

- This will create a react app at whichever folder we are currently in
- Once we are in the react project folder, running `npm start` will launch the app in the default browser
- Alternative: using codesandbox (an online IDE)
    - Simply navigating to react.new will create a new react project

What did `create-react-app` make?

- `package.json`: contains information about versions, dependencies, etc
- `src` (source) folder: contains the actual code

Creating react elements

- We want to write code to add the elements we create to our page
- The `ReactDOM.render` function is what will add elements to the page

```
ReactDOM.render(
        React.createElement("h1", {style: {color: "blue"}}, "Message..."),
        document.getElementById("root");
);
```

- Here, we have added an `h1` element to the page with a style property and a message `"Message..."`
    - Here, the style property is a nested object.
    - We can add any HTML property: href, src, class, id, etc.
- The render function takes two arguments
    - First argument: the `createElement` call (the element we want to add)
    - Second argument: the parent element on the page that we want to append it to
- We have just used JavaScript to add HTML to the page

Refactoring elements using JSX

- What if we wanted to add a large number of elements to the page (for example, an unordered list)?
- We would need a lot of `createElement` calls
- Not only that, but we would need to nest the `createElement` calls

- JSX (JavaScript as XML) is an extension that lets you write HTML-style tags directly in JavaScript
    - We can this to create complicated elements

```
ReactDOM.render(
        <ul>
                <li>One</li>
                <li>Two</li>
                <li>Three</li>
        </ul>,
        document.getElementById("root");
);
```

- This syntax is not natively supported by the browser. However, the project folder (created by the setup process) has a tool called Babel, which compiles (or rather, transpiles) JSX into the regular nested `createElement` calls that would need to be made

React Components

What are components?

- Components are building blocks of the UI (small pieces)
- There is already and `app` component in the project
- We create a component by defining a function that returns JSX
- A component must be exported, then imported into the main JS file in order to be used

```
//component code
export default componentName

//main page code
import componentName from "path/to/component/file"
```

- We can draw this component to a page using the following code

```
ReactDOM.render(<componentName />, /* parent element */)
```

- The complete code for a component may look something like this
    - In this example, we have two components, where one uses the other

```
import React from "react"
import "./myComponent.css";

function coolHeader() {
        return (
                <h1>I'm a cool header</h1>
        )
}

function myComponent() {
        return (
                //note that react uses className instead of class
                <div className="myComponent">
                        <coolHeader />
                        <p>Some text</p>
                        // etc.
                </div>
        );
}

export default myComponent
```

- `<coolHeader />` is a self-closing tag: we don't need to nest anything inside of it, so we can open and close it in the same declaration

- Nesting components together can be used to create a larger application
    - Usually, the entire application will be nested into a `Main` component

Adding properties to components

- We can pass a `props` element into a component function in order to parameterize properties to the function
- In our definition of the component, we can access the `props` object directly in the JSX
    - Parameters must be enclosed in curly braces `{}` to be accessible in the JSX
- When we use a component tag, we can specify properties as we would normally in HTML

```
//definition for coolHeader component
function coolHeader(props) {
 return (<h1>{props.message}</h1>);
}

//use of coolHeader component
<coolHeader message="Hi, I'm a cool header!"/>

//page contains an h1 element with inner HTML "Hi, I'm a cool header!"
```

- We can define whatever properties we want into a component call. The component may or may not use them
- A property can be of any type

```
//definition of dateDisplay component

<dateDisplay year={new Date().getFullYear()} />
//here, we are using a javascript class to get the current date
```

Working with Lists

- We can display an array (or any collection) of elements by mapping over it and adding each item as a JSX element

```
//if you know, you know
const skillsArray = ["8-2<". "12---o", "834/"];

//skillList component definition
//notice that the style property is in camelCase, not regular css "dash-case"
<ul style={{textAlign: "left"}}>
        {props.skills.map((skill) => (<li>{skills}</li>))}
</ul>
//functional programming <3 <3 <3

//using the skillList component
<skillList skills={skillsArray}>
```

- This is dynamic rendering: if we add another item to the array, the UI will update to include it

Adding keys to list items

- Adding (unique) keys to list items is recommended so that array indices to not go out of sync (if items are added to the list, for example)
- One way to do this is to define the key as the position of the item in the list

```
<ul style={{textAlign: "left"}}>
        {props.skills.map((skill, i) => (<li key={i}>{skills}</li>))}
</ul>
```

- However, this is recommended against because it can lead to rendering problems
- Solution: instead of having the array contain strings (or whichever primitive type), we have it contain objects with fields (ex. `title` for the string we want to store, `id` for the key, etc)

```
const skillsArray = ["8-2<". "12---o", "834/"];
const skillsObject = skillsArray.map((dish, i) => ({id: i, title: dish}));

//component maps from object to <li> instead
```

Displaying Images with react

- Now that we have experience with text, we are ready for more
- We will have to import the image

```
//other import statements
import myImage from "./myImage.png";
```

- Then, we can simply use an `img` tag like any other HTML element

```
<img src={myImage} height={200} />
```

- We can also use a url instead of a local image
- React (and web) best practices suggest that images have alt text in order to be accessible, so it should be added as a property in the JSX

Using Fragments

- If we try two render two different components or elements, we will get an error
    - Any adjacent JSX elements must be enclosed in a parent tag

    - We can solve the error (and render what we intend to) by wrapping the two components in a `div` or `section` (or whatever)
- However, this can lead to a lot of unnecessary nesting and tags to the DOM
- There is a specific tag we can use: `React.Fragment`

```
ReactDOM.render(
        <React.Fragment>
                <App />
                <App2 />
        </React.Fragment>,
        //parent element
);
```

- This encloses the components `App` and `App2` without adding anything to the DOM
- The empty tag `<>...</>` is a shorthand for `React.Fragment`

```
ReactDOM.render(
        <>
                <App />
                <App2 />
        </>,
        //parent element
);
```

React state in the component tree

Rendering elements conditionally

- We can use the conditional features of JS (i.e. if-statements, etc.)

```
//defining some components to render conditionally

function secretComponent() {
        return <h1>Secret title!</h1>;
}

function regularComponent() {
        return <h1>Regular title</h1>;
}

//App component definition
//authorized <==> secretComponent is visible

function App(props) {
        if(props.authorized) {
                return <secretComponent />;
        } else {
                return <regularComponent />;
        }
}

//rendering

ReactDOM.render(
        <App authorized={false} />, /* parent element */
)
```

- We could also write the `App` component using a ternary expression

```
function App(props) {
        return (
                <>
                {props.authorized ? <secretComponent /> : <regularComponent />}
                </>
        )
}
```

Destructuring Arrays and Objects

- We can essentially create keys for an array (or, depending how you think about it, create a bunch of variables at once) by specifying them in the declaration

```
const [fullOutPike, tripleTuck, Vachon] = ["8-2<". "12---o", "834/"];

console.log(tripleTuck);
//result: 12---o
```

- We do not need to create a name for every variable

```
const [fullOutPike, , Vachon] = ["8-2<". "12---o", "834/"];

console.log(Vachon);
//result: 834/
```

- In react, destructing is often used with the props object
- If we pass the name of the attribute(s) that we want (enclosed in curly braces) as an argument, the object is destructured and we can access that value directly
    - We can avoid the clunky dot notation

```
function App ({ myAttribute }) {
        return <h1>{myAttribute}<h1>;
}
```

The `useState` hook

- The best way to manage the state of a react application is the `useState` function
- First, we need to import it

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

- Calling `useState` returns an array containing two items
    - First item: state variable
    - Second item: function that can be used to update the state

```
const what = useState();
```

- We can pass a state into the `useState` function
    - This will be the initial state

```
const what = useState("happy");
```

- We can use array destructuring to grab both elements (current state and state altering function) at the same time

```
const [curState, setState] = useState("happy");
```

- We can use this function to alter the state inside of a component

```
//we have a button that can change the state from happy to frustrated
function App() {

        const [curState, setState] = useState("happy");

        return (
                <>
                <h1>Current state is {curState}</h1>
                <button onClick={() => setState("frustrated")}>Frustrate</button>
                </>
        );
}
```

- We can declare as many state variables as we need

```
//setting curState1 to "happy"
const [curState1, setState1] = useState("happy");

//setting curState2 to "tired"
const [curState2, setState2] = useState("tired");
```

The `useEffect` hook

- Used to manage side-effects that don't directly affect rendering
- As always, we must import the function from the react library

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

- `useEffect` takes a callback function (the function with the side effect)

```
useEffect(() => {
        console.log(`The current state is ${state}`);
});
```

- It also takes in a second argument: the dependency array
- If this array is empty, the props and state inside the effect will retain their initial values (i.e. it will only be called during the first render, not after)

- If the array is not empty, it should contain state values
    - Every time one of these values changes, the `useEffect` will execute the callback function

The `useReducer` hook

- The reducer takes in a current state and returns a new state
- `useReducer` takes in two arguments: the function used to change the state, then the initial state
- The following code updates text so that it remains consistent with a checkbox

```
function App() {

        //the toggle reducer toggles the state variable checked
        const [checked, toggle] = useReducer(
                (checked) => !checked,
                false
        );

        return (
                <>
                        <input
                                type="checkbox"
                                value={checked}
                                onChange={toggle}
                        />
                        <p>{checked ? "checked" : "not checked"}</p>
                </>
        );
}
```

Asynchronous React

Fetching Data with hooks

- Fetching data from external sources is a very common task
- Example: fetching a JSON object from the GitHub API
    - JSON file of user data at https://api.github.com/users/username

```
function App({ login }) {

        //initial state is null because there is no data initially
        const[data, setData] = useState(null);
        useEffect(() => {
                //.then executes functions in sequence if the promise is fulfilled
                fetch(`https://api.github.com/users/${login}`)
                //data is converted from text to JSON
                .then((response) => response.json())
                //state is updated; state variable "data" will hold the JSON object
                .then(setData);
                }, []);

        if(data) {
                //JSON.stringify turns the JSON data back into a string
                return <div>{JSON.stringify(data)}</div>;
        } else {
                return <div>No User Found</div>
        }

}
```

Displaying data from an API

- We can display something nicer than a string version of a JSON file
- We can access properties of the JSON file in the same way we access properties of objects (using dot syntax)

```
//rest of code...
if(data) {
        return (
                <div>
                        <h1>{data.name}</h1>
                        <p>{data.location}</p>
                        <img alt={data.login} src={data.avatar_url} />
                </div>
        );
}
```

Handling Loading States

- When we make an API https request, there are three possible states
  - Loading
  - Sucsess
  - Failed (ex. broken url, etc)
- We should be able to handle each of these states on our page

```
function App({ login }) {

        const[data, setData] = useState(null);
        const[loading, setLoading] = useState(false);
        const[error, setError] = useState(null);

        useEffect(() => {

                if(!login) return;  //login was not specified, so we can't do anything
                setLoading(true);   //if it is specified, we are now loading
                //.then executes functions in sequence if the promise is fulfilled
                fetch(`https://api.github.com/users/${login}`)
                //data is converted from text to JSON
                .then((response) => response.json())
                //state is updated; state variable "data" will hold the JSON object
                .then(setData);
                //we have the data, so we are no longer loading
                .then(() => setLoading(false));
                //if an error is thrown here somewhere, we will set the error state
                .catch(setError)
                //this gets called every time the login changes
                }, [login]);

        //possible things we can return

        if(loading) return <h1>Loading...</h1>;
        //if there's some error, it will be displayed on the page
        if(error) return <pre>{JSON.stringify(error, null, 2)}</pre>;
        if(!data) return null;

        if(data) {
                //JSON.stringify turns the JSON data back into a string
                return <div>{JSON.stringify(data)}</div>;
        } else {
                return <div>No User Found</div>
        }

}
```

React Testing

Using `create-react-app` for testing

- The `create-react-app` packages includes testing features
- If a file ends in `test.js`, it will be treated as a test
- Running `npm test` will run all of the tests

Testing small functions with Jest

- We can use a function called `test` (which comes from the Jest library, which gets included automatically by cra)
- There is also an assertion function called `expect`

```
//in functions file

export function myFunction(a) {return a*2;}

//functions.test.js

import {myFunction} from "./functions"

test("Name of test", () => {
        //assertion that we expect to be true
        expect(myFunction(4)).toBe(8);
});
//test passes!
```

React testing library

- Another testing suite
- Lets us render the output so that we can make sure it looks like what we expect it to

```
import { render } from "@testing-library/react";
import React from "react";
import App from "./App"

test("renders an h1", () => {
        //we have destructured the render function to get a function that
        //searches by text (getByText)
        const { getByText } = render(<App />);
        //we are searching for the text "Hello React Testing Library"
        //this uses a regular expression
        const h1 = getByText(/Hello React Testing Library/);
        //we expect our result to have this text in it
        expect(h1).toHaveTextContent("Hello React Testing Library");
});
```

- `getbyText` is a query, which returns information about some sort of element using destructuring
  - This is part of react testing library

Testing hooks with react testing library

- We will check the checkbox component that we wrote before

```
import { render, fireEvent } from "@testing-library/react";
import React from "react";
import App from "./App"

test("Selecting checkbox", () => {
        //destructuring to get getByLabelText function
        const { getByLabelText } = render(<Checkbox />);
        //getting the HTML element by searching text
        const checkbox = getByLabelText(/not checked/);
        //simulates an event happening (i.e. clicking the checkbox)
        fireEvent.click(checkbox);
        //we expect the checkbox to now be checked
        expect(checkbox.checked).toEqual(true);
});
```

React Router

Installing React Router 6

- When we created react apps, we were creating single page applications
    - Instead of having multiple pages, JS just changes the current page when we interact with it
    - But how to we get from page to page?
- React Router is a tool that can help us
- We can install it using npm

```
npm install react-router@next react-router-dom@next
```

- We will create a new file called `pages.js`, which will hold all of the pages in our app

```
import React from "react"

export function Home() {
        return (
                <div>
                        <h1>[My Website]</h1>
                </div>
        );
}

export function About() {
        return (
                <div>
                        <h1>[About]</h1>
                </div>
        );
}

//etc
```

Configuring the Router

- The router lives in the `index.js` file
- This is where we will pass the information from the router to any nested components

```
//index.js

//imports
import { BrowserRouter as Router } from "react-router-dom";

ReactDOM.render(
```

```
        <Router>
                <App />
        </Router>,
        document.getElementById("root");
);

//App.js

//imports
import { Routes, Route } from "react-router-dom";
import {
        Home, About, Events, Conact   //these were all defined in pages.js
} from "./pages"

function App() {
        return(
                <div>
                        <Routes>
                                <Route path="/" element={<Home />} />
                                <Route path="/about" element={<About />} />
                                <Route path="/events" element={<Events />} />
                                //etc
                        </Routes>
                </div>
        );
}
```

- The App is wrapped in a `Router` tag
- When we specify a path, this is the path after the domain name
    - For example, /events would be at mySite.com/events

Incorporating Links

- Navigating by typing URLs is not user friendly
- We can use a `Link` component (inbuilt) which will create a link to a path we have set

```
<Link to="about">About</Link>
<Link to="events">Events</Link>
<Link to="contact">Contact</Link>
//etc
```

- The `to` property specifies which page the link connects to
- We can also use links to make a 404 page that will be displayed if the user attempts to view a page that is not in our routes

```
//pages.js
export function Whoops404() { return(/* page content here*/); }
```

```
//App.js
//imports
import { ...Whoops404... } from "./pages"
//Routes
//other Route elements
<Route path="*" element={<Whoops404 />} />
```

- We can use the `useLocation` hook, which gives us our current location, to return a 404 message that mentions the erroneous page specifically

```
import { ...useLocation... } from "react-router-dom";
//...
let location = useLocation();
<h1>Resource not found at {location.pathname}!</h1>
```

Nesting Links with React Router

- We can nest routes, which adds another layer of subpages
  - I.e. we have URLs in the form mySite.com/page1/page2
- This is done by nesting `Route` tags

```
<Routes>
        <!-- mySite.com/ --->
        <Route path="/" element={<Home />} />
        <!-- mySite.com/about --->
        <Route path="/about" element={<About />} >
                <!-- mySite.com/about/services --->
                <Route path="services" element={<Services />} />
                <!-- mySite.com/about/history --->
                <Route path="history" element={<History />} />
        </Route>
</Routes>
```