Intro to ReactJS

ReactJS is a Javascript library which allows you to build a complex front-end using reusable pieces of code called components. You can render these components in a page using JSX (it looks like HTML but it's not). JSX allows us to write React "elements" like we would usually do our markup.

We can also pass data values using props (properties) and/or states.

Before we can get into the code, we need to understand these core concepts.

Is it a framework or a library?

At its core, although React is used like a framework in many cases, React is a Javascript **library**. It doesn't need Node.js to run. You just need to include the scripts and it can be used with just HTML, CSS and JS.

We will be building React apps using React with Node.js because of the building tools available on Node.

Start a React app

The quickest way to get started when learning React is to use the Vite bundler. You can run a command which will add all of the requisite modules in a *package.json* file to start. From there you just need to install and you'll get some boilerplate code. This is what we'll use (you'll need Node.js).

1. In your CLI/Terminal, type:
   - `npm init vite@latest`
2. You can change the project name by just starting to type. This will create a subfolder with that project name as the folder name, so **do not use spaces**.
3. When prompted choose a "`react`" project. (For the variant, choose "`Javascript`".)
4. Node modules do not come pre-installed, so **navigate to the project folder in your terminal**.
5. Once in your project folder, run: **`npm i`**.
6. **For Firefox**, modify the "dev" line in ***package.json*** to say "`vite --host`" instead of "`vite`" alone.
7. Run `npm run dev` to run the project.

Create React App

If you read older React tutorials you may come across Create React App as a build tool for React web apps. Currently, it's no longer recommended because it's very slow to install, build and run. Vite is much quicker. Overall, the folder structure of the boilerplate code is not too different. The only differences are that the *index.html* file is in ***/public*** and the React files are using .js file extensions (instead of .jsx).

Note

`vite` (like Create React App) is pretty barebones and you'll probably see suggestions for using frameworks, but since we're just starting introducing React, using Vite will allow us to focus on pure React concepts syntax initially. Frameworks have more features, but are a bit too complex to start because you'll be dealing with figuring out React *and* a framework.

Other boilerplate/frameworks can be looked at later (e.g. `create-next-app`).

Understanding the directory structure

Notice that the installation will create an app structure with the following folders:

- node_modules (after running **npm i**)
- public
- src

For us, the important stuff is in the ***src*** folder. This is where we'll have our React code. The *index.html* page in the root is where you can modify the favicon if you wish.

Components

As mentioned above, components are reusable chunks of webpages. This means that when we're building a React app, what we're actually doing is building pieces of web pages for display.

You can build a component using a function (this is the functional component style).

```
import React from "react"; //import React library (though in newer versions you can probably omit this
function SiteName() {
  return (
    <h2 id="site-name"><a href="/">My Site</a></h2>
  );
}
export default SiteName;
```

When you create a component you can then render in a parent component using JSX:

```
import SiteName from "<path-to-component-file>";
…
<SiteName />
```

Notice that the component can be written like an element (JSX) but the "element" name is the name of the object as imported. Typically, the JSX will be used in a method or render function.

JSX

Resources

- https://react.dev/learn#writing-markup-with-jsx

JSX is React's form of markup. Although you can include some HTML elements, some of the attributes are different because they are **NOT HTML**. For example, instead of `class`, use `className`; instead of the `for` attribute, use `htmlFor`. When using event handlers, use `onClick` instead of `onclick`.

`<SiteName />` will render the SiteName component (whatever's in the return() statement in the component's code).

In JSX, sometimes you need to use Javascript or write comments (Javascript comments). To do this, you enclose the Javascript in curly brackets.

```
<Movie title={myTitleVariable} />
```

The above line renders a Movie component which has a `title` property set to the value of the Javascript variable `myTitleVariable`.

If you need to add comments within JSX, you would add:

```
{/* my comment */}
```

Notice that the comment is a JS comment so it's enclosed in curly brackets. This is only for the return() code where JSX is expected (this will make more sense when we actually start writing code).

Props and States

Resources

- https://react.dev/learn/passing-props-to-a-component
- https://react.dev/learn#using-hooks
- https://react.dev/learn/state-a-components-memory

Props

Props (properties) allow you to pass values to a component. Typically, your React site app will be made up of many components. Some of these components may be using other components. If you need to pass data to a component to be used, you can pass a prop. Props can be passed in a JSX element (they look like attributes).

Expanding on the Movie component example, in the following line, we're passing two properties (in green) to the Movie component: `title` and `year`.

```
<Movie title="Turning Red" year="2022" />
```

Properties are one-way only. Property values are only passed from the parent component (the one *using* the component that is receiving a prop). **A component can never modify its props.** In the above example, this means that the Movie component code cannot modify the `title` or `year` props.

Within the **component code**, you need to have `props` as a parameter for the Movie function. Inside the Movie component function, you can access the prop by `props.title` (for the `title` prop).

Using the above example, if we have a parent component named MovieList using the Movie component, MovieList can pass props to Movie, but not the other way around.

States

States also carry data for components, but unlike props, states are managed by its component. This means that the state value can be updated within the component it belongs to.

When you set a state, you can set it in the component it belongs to using:

```
const [myStateVar, setMyStateVar] = useState();
```

This creates a state variable named `myStateVar` and also creates a setter function for `myStateVar` named `setMyStateVar()`. The value in `useState()` is the default value you're setting for `myStateVar`.

In your code, you can use the state variable by using `myStateVar` (in this case).

`useState()` is a hook function and you need to import it from React in order to be able to use it.

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

Hook functions

Hook functions are special functions which allow functional components to "hook" into certain functionality such as using states or executing code upon component render.

Previously, the state could only be accessed by class components (components using JS class syntax). For example (older syntax):

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      myStateVar: ""
    };
  }
  //to change state, this.setState({ myStateVar: "newValue" });
  //to use state, use this.state.myStateVar
  render() {
    return ( … );
  }
}
```

Note that functional components are the way to go now as it's shorter and the syntax is easier to follow.

When you change a state value, you need to use the setter function.

```
setMyStateVar(<new_value>);
```

The reason why you need to use the setter function and not just set using the state variable directly is because of the way React works. React is fast because it only re-renders components when needed. It can do this by monitoring for changes to the state. If a state changes, the component needs to be re-rendered. Using the setter function will update the state and trigger re-rendering.

Build some components

Let's try putting it all together. First, we'll add some common components (site header and footer).

In the boilerplate code, first take a look at ***src/main.jsx***. The index.js file is the starting file for your site. It contains some lines of code to render the App component in the #root of the page.

You can leave the default code. The strict mode is useful for development by activating extra tests and warnings (though you can remove strict mode and just use <App /> if you wish).

main.jsx:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

Notice that the App component is imported from the ***App.jsx*** file. The `import` statement refers to the relative path to the ***App.jsx*** (note that you can omit the *.jsx* file extension).

There is also an ***index.css*** imported. This is the styling for this page. You can put all your styles in this file or have a CSS file per component.

Now let's clear out the default stuff in App.jsx so that it's ready for us to add to it. Change the App.jsx code to get rid of the stuff in the return() and get rid of the extra import lines and useState() stuff.

App.jsx:

```
import './App.css';
function App() {
  return (
    <div className="page">
    </div>
  );
}
export default App;
```

Right now, there's only an empty div with the class "page" (I changed this), but let's take a closer look at the code.

The App function only makes a return(). This return() is where you write your JSX. **The key thing is that whenever you do a return, you should have a single root element**. This is what allows you to use the element syntax (i.e. one component element means one element must be rendered—the root element of the component). Every component should return something since components are small "pieces" of your web application. In the original code, the root element was a "virtual" element (a fragment) denoted by <> as the opening tag and </> as the closing tag. This fragment serves as the root element but is not rendered as markup when the app is built.

The `export default App` is what allows you to do an `import` in the ***main.jsx*** file. Without the export, an import would throw an error.

Now we're ready to start adding a few common site components.

Add a Header component

1. To keep things a little neater, create a ***components*** folder within ***/src***. This is where we'll put our component code files.
2. Create a ***Header.jsx*** file within ***/src/components***.
   Note: By convention, React components should be named in capitalized naming format!
3. Since we're not dealing with routing yet, we'll keep it simple and just print a site name link in the Header. Add the following code to ***Header.jsx*** and save it.

   ```
   export default function Header() {
     return (
       <header id="header">
         <h2 id="site-name"><a href="/">Test React Site</a></h2>
       </header>
     );
   }
   ```

4. Now, let's add use the `<Header />` component in the ***App.jsx*** file. In ***App.jsx*** (added lines in yellow):

   ```
   import './App.css';
   import Header from "./components/Header.jsx";
   function App() {
     return (
       <div className="page">
         <Header />
       </div>
     );
   }
   ```

```
export default App;
```

5. Check the page in the browser. If you have not run `npm run dev`, do so now. Running this script does monitor for changes and auto-compiles along with auto-refreshing so you can leave it running while developing.

Add a Footer component

1. Create a ***Footer.jsx*** file in ***/src/components/***.
2. Add the following code within ***Footer.jsx***:

```
export default function Footer() {
  return (
    <footer id="footer">
      <div>&copy; HTTP5211, 2023.</div>
    </footer>
  );
}
```

3. Modify ***App.jsx*** to import Footer and print the `<Footer />` component after `<Header />`.
4. Check it in the browser.

Add a Movie component

We'll be using a simple Movie component to print a movie title and year in a paragraph.

1. Create a ***Movie.jsx*** file within ***/src/components/***.
2. Add the following code:

```
export default function Movie(props) {
  //Props are read-only.
  //Prop values are passed from parent component and
  //are not set by the child component (this).
  //A state is initialized and managed by component.
  //If something needs to be changed in the component
  //use a state rather than a prop.
  return (
    <p>{props.title} ({props.year})</p>
  );
}
```

   The code above simply prints a prop named title and a prop named year (within parentheses) in a paragraph.
3. Try adding the following between the `<Header />` and `<Footer />` in ***App.jsx***.
   `<Movie title="The Dark Knight" year="2008" />`
4. Save and view in the browser. You should now see the movie title and year printed between the site name link and copyright text.

Add a MovieList component with a state

Next, we'll add a MovieList component which will print out a list of <Movie> components. The movie list will be stored in a state. Note that since we will be using Movie components, the MovieList component will need to import Movie.

1. Create a ***MovieList.jsx*** within ***/src/components/***.
2. Add the following code and save:

```
import {useState} from "react";
import Movie from "./Movie.jsx";

var moviesArray = [
  {
    title: "The King's Man",
    year: "2021"
  },
  {
    title: "The Dark Knight",
    year: "2008"
  }
];

export default function MovieList() {
  const [moviesList, setMoviesList] = useState(moviesArray);

  return (
    <div>
      {
        moviesList.map((m) => (
          <Movie
            key={m.title + m.year}
            title={m.title}
            year={m.year}
          />
        ))
      }
    </div>
  );
}
```

- In the `return()` code, there is a root <div> element. We need this because the component root must be only **one** element. Alternatively, you can add <> and </> instead of <div> and </div>. This indicates a fragment.

- The `moviesList` state variable is initially set to the value of `moviesArray`.
- In this particular component, we're not using props. If you were using props, you would need to include `props` as a parameter in `function MovieList()`.
- The code is using the `Array.map()` to loop over the array (we are storing the list of movie objects in an array under the `moviesList` state variable).
- Since the array-looping is JavaScript, the code is enclosed in curly brackets (yellow for emphasis).
- As we have multiple Movie components, React requires a unique identifier (`key`) to ensure it can keep track of any state changes accurately (if check the console in the browser, you'll see an. In our case, I'm assuming that the movie name plus year will be unique (i.e. you won't have the same movie name released within the same year), so I'm concatenating the value and using it as my key. If you are using index numbers in your array, **do not use index numbers as key values**. This is a bad idea because if the array changes (e.g. items removed), the index may no longer point to the same item anymore. This would affect how accurately React can track state changes.

3. In ***App.jsx***, import the MovieList component.
4. Now, in ***App.jsx*** replace the <Movie> component with the following to render the MovieList component instead:
   `<MovieList />`
5. View in a browser. You should see two paragraphs with the movie data printed out.

Add a form

To demonstrate how to modify a state using the setter, we'll add a form to the MovieList component.

1. Modify the MovieList return() block to add a form **before** the { moviesList.map()… } stuff.

```
return (
  <div>
    <form onSubmit={handleForm}>
      <label htmlFor="title">Movie title:</label>
      <input type="text" id="title" name="title" />
      <label htmlFor="year">Year:</label>
      <input type="text" id="year" name="year" />
      <button type="submit">Add</button>
    </form>
    {
      moviesList.map((m) => (
        <Movie
          key={m.title + m.year}
          title={m.title}
          year={m.year}
        />
      ))
    }
  </div>
);
```

- Upon submit, the form will call a function (for the current component) named `handleForm()`.
- The label's `for` attribute is written as `htmlFor` in JSX.

2. Above the `return()`, add the `handleForm()` function.

```
function handleForm(e) {
  e.preventDefault(); //prevent form submission from going through

  //get values from form and make a new movie object
  let newMovie = {
    title: e.target.title.value,
    year: e.target.year.value
  };
  //update the moviesList state using the setter function
  //use the array spread syntax
  setMoviesList([
    ...moviesList,
    newMovie
  ]);
}
```

- The `preventDefault()` prevents the form from submitting. We could use the form without a submit button but this breaks typical form behaviour (submit on hitting **Enter**) so we're sticking with a submit but prevent default behaviour.
- In React you cannot change the state variable directly (i.e. it's immutable). In this case, instead of directly changing the original array, we're using the array spread syntax to create a new array containing the existing items in moviesList and add the new movie data to end of the new array. (This is an immutable change because a new array was created.)
    - https://react.dev/learn/updating-arrays-in-state
    - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_array_literals
- `setMoviesList()` sets the `moviesList` state value to the new array (it's the setter function we named earlier). As the state variable has changed, this should trigger re-rendering.

3. Save and test out the new form in the browser.

More resources

If you want to try more, check out some of these tutorials:

- https://react.dev/learn/tutorial-tic-tac-toe: The official ReactJS tutorial (tic-tac-toe)
- https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started: A todo list tutorial on MDN.

You may also find the following resources useful:

- https://react.dev/learn: This is React's quick start guide. It has an overview of most-commonly-used syntax.
- https://nextjs.org/learn/foundations/from-javascript-to-react/react-core-concepts