18. Fundamentals of React

GitHub Repository Branch | Vimeo Video Recording

- [X] React—what is it, and why use it?
- [X] JSX (JavaScript XML)
- [X] Components
- [X] Props
- [X] State
- [X] DOM Events

What Problem are we Alleviating?

Think to our time in the front-end with HTML, CSS, JavaScript and jQuery. One inconvenience we faced, in contrast to Express with EJS in the back-end, was there being no standard solution for re-usable chunks of code.

If we wanted to have the same navigation on every page of our Express application, we can use EJS partials to accomplish that. If we wanted the same navigation on every page of a static website... we were left repeating it on every single HTML file. There was little that was re-usable in terms of our user interface, once we let go of Express.

Using JavaScript, with or without jQuery, we have the opportunity to manipulate the DOM, generate styles, and more! This definitely helps with writing your user interface once, and having it run in many places... however, if you are writing your own functions and programs to accomplish this on a per-project basis there will be little consistency and it is hard to onboard new team members to your project.

Consider as well, the prospect of building an SPA (Single Page App.) How might you approach this using JavaScript and/or jQuery? If you turned to another student, or hobbiest, or professional, do you think they'd answer with the same solution? Probably not... While JavaScript and jQuery are incredible powerful and offer you unlimited freedom in your approach, again, sometimes too much choice can be difficult to maintain and work around.

Okay, so we're having trouble making things DRY in the front-end (especially in static web sites.) What can we use to DRY things out in a maintainable and easy-to-communicate way?

Front-end Libraries and Frameworks

To assist developers in achieving this idea of a more modular, DRY, front-end codebase, talented and passionate developers in the JavaScript community began trying to write their own libraries that would bring structure and convention to the table. A set way and set of function that a developer can learn to use, and implement more easily alongside their team. Of these, we've seen a variety mature over the years from various companies and developers:

- Angular (Google)
- Backbone (Jeremy Ashkenas)
- Ember (Yehuda Katz; Tilde Inc.)
- React (Jordan Walke; Facebook)
- Svelte (Rich Harris)
- Vue (Evan You)

Each of the above offers a potential solution for our problem. There is a level of rigidity added to the approach you'll take with any of these—you sacrifice some freedom for convention—that is to say: a correct, or preferred, way to organize your project to achieve this end.

If you play along with any of these frameworks, you'll find each and every one of them have convenient tools baked in to make working on your front-end code easier, cleaner, and faster! That boost in development speed, of course, is once you've gotten the hang of how it is they'd like you to use their library. Thankfully, most mature libraries in this ecosystem will have a solid documentation as well as there being a cornucopia of YouTube videos, forum posts, blog articles, books, and more at your fingertips for getting up to speed.

Choosing a Library to Cover in Lighthouse

Picking a framework or library to power your front-end user interface can be daunting. Each carries its own strengths and weaknesses, its own set of features that may suit one project better over another! Given this, how did Lighthouse decide which one should be covered during your time in our program?

One metric is the amount of market share each one holds. Essentially—how popular is each one? Which one might be most likely to get you a job?

Another is how long it might take to learn the basics of the library, as in a bootcamp, we have no time to lose.

The top three libraries, in terms of both use and employability (at the time of writing these notes) are:

1. React (by a *Significant* Margin) `Estimated ~40% of Web Developers`
2. Angular `Estimated ~22% of Web Developers`
3. Vue `Estimated ~18% of Web Developers`

Alright, it sounds like React is the clear winner on that front! How about in the "covering within a bootcamp" category?

Well, conveniently enough, React is a nice middleground.

Angular has a lot more features built-in, and there is a lot more to learn in order to take advantage of each of them. This makes it harder for us to squeeze Angular into a short time-frame without sacrificing content or understanding.

How about Vue? Vue is the most lightweight of the three, and the only of the top three to be created by an individual and not a global corporation. It is meant to run very quickly, be fast to learn, but does not incorporate as many features out-of-the-box as React or Angular. Because it is newer, you won't always find as many answers or add-ons as the alternatives.

React finds itself in a very nice place for us to adopt here at Lighthouse. It is the most popular, most likely to get you a job, has the biggest community, the most add-ons, is reasonable to learn, and will still cover core concepts that you can carry over to other popular libraries.

Browser Extensions

Please install React Developer Tools in any browsers you have installed, they'll provide a lot more insight during your development process, offering tips, warnings, and more detailed errors as you troubleshoot:

- Google Chrome
- Microsoft Edge
- Mozilla Firefox

You'll find they even offer you a few new tabs in the Web Developer Tools in these browsers; these will come in handy later!

First Steps in React

React projects can run in a page as long as the library is included in the page (note in most cases you'll want to have the core React library as well as its DOM counterpart):

In the `<body>` of your web page, you'll want to include an element for your React application to mount to. Your application will render inside of this element for the user to interact with:

Once those are added, you can begin writing your own React code. Let's look at a very basic example:

```
// Choose an element in your web page.
const domContainerElement = document.getElementById('your-container');

// Let React know you want your application to appear in this element.
const reactRoot = ReactDOM.createRoot(domContainerElement);

// Build a component for your application.
class HelloComponent extends React.Component {
  render() {
    // Notice we pass the HTML element, and the text inside.
    return React.createElement('p', {}, 'Hello, World!');
  }
}

// Get React to render your application in that root!
reactRoot.render(React.createElement(HelloComponent, null, null));
```

Have a look at your page in the web browser. Do you see the `<p>Hello, World!</p>` component displaying on your web page?

Congratulations! This is your first React application!

JavaScript XML (JSX)

You'll find in most examples, as you traverse the web, an alternative syntax for your output. A lot of developers do not like typing out: `React.createElement()`, and then pass in information. People were hoping for a way to write the output that would more closely resemble HTML. The answer to this is a syntax called JSX!

JSX has the benefit of looking *very* similar to HTML, but still running in JavaScript—offering us the unique ability to embed JavaScript code and (in this case) React features in the markup.

Note that JSX is not compatible with HTML or regular, vanilla, JavaScript. It is a syntax layer written into the Babel compiler. To run this special syntax we'll need to include the Babel source code in our web page:

Now that Babel can help us out with any JSX, we can *convert* our existing React component to utilize this syntax, we'll be able to see the difference quite quickly!

Here's our old code:

```
<script>
  const domContainerElement = document.getElementById('your-container');
  const reactRoot = ReactDOM.createRoot(domContainerElement);

  class HelloComponent extends React.Component {
    render() {
      return React.createElement(
        'p', // Element tagname.
        {onClick: () => {console.log('Clicked!');}}, // Events and
config.
        'Hello, World!' // Text.
      );
    }
  }

  reactRoot.render(React.createElement(HelloComponent, null, null));
</script>
```

Here's the JSX way to run it:

Consider an application, as it grows and grows. You'll have more and more components, and each one may have any number of elements that it'll need to contain. It is much easier to read code written in JSX, as it so closely represents the final output and is organized much more clearly.

There are some rules you should keep in mind when writing JSX code:

- There can only be one root element per JSX expression
- Any opening tags must have a corresponding closing tag
- Self-closing elements must end with `/>`, as per XHTML / XML syntax (example: `<img />`)
- Multi-line JSX expressions should be wrapped in parenthesis `()`
- Regular JavaScript expressions can be executed in elements with use of curly braces `{}`

- JavaScript reserved words (example: `for`, `class`) should not be used as element or attribute names (look up alternatives like: `htmlFor`, `className`)

At the end of the day, the JSX code is actually being "compiled" (transpiled) using Babel, into regular JavaScript. Consider JSX a syntax sugar that makes the development process a bit easier, as it isn't a real JavaScript feature.

So is this, then, the process React developers usually follow in creating their applications? It can be in a pinch, but there is a *better* way!

Create React App

There is a command-line tool we can make use of to quickly generate React projects called "Create React App." This is a very powerful and optimized tool that has become the defacto standard for starting work on a React application. It comes, out-of-the-box with amazing features like:

- Babel for transpiling JSX
- Easy inclusion of third-party libraries
- Debugging and lint tools for ease of troubleshooting
- Live-refresh of the project on file save
- Build tools for minifying and launching your project
- A test suite for writing application and component tests

To get started, navigate in your terminal to your projects directory. Consider the name you'd like to use for your project, this tool will create a new folder for us with a name we give it. Run the following, switching out `my-first-react-app` with a descriptive name for your project:

```
# Generate a React application.
npx create-react-app my-first-react-app

# Navigate in your terminal to this new application directory.
cd my-first-react-app
```

Open the project in your favourite editor, and let's have a look around!

The Create React App Directory Structure

There are a variety of files in the project to give you an idea of how and where to put things.

- `public` : Static assets that will not need to be run through Babel, Sass, etc.
- `src` : Working project files (before transpilation)

Note that `/public/index.html` represents the static page you'll see in the browser when running the application, and it can be customized to your needs.

For more information on `/public/robots.txt` , and how it can be used to let bots know where, or if, they should read parts of your site, check out `robotstxt.org` .

Try Running the App

Run your project to see if everything is configured and installed properly:

```
npm start
```

The default configuration in a Create React App application will have this command run through a number of steps and ultimately run `/src/index.js` as the application entry-point.

In a few seconds, you should see it go through the app, linting and checking for errors in the code. As it is fresh, it should run error-free and it will attempt to open the project in your default web browser with the default address: `http://localhost:3000` . Note that if port `3000` is in use, it will keep bumping the port up by `1` until it finds a free port (so `3001` would be next, if you already had another program taking up that port.)

This web page will automatically refresh as you make changes to the application! Try opening `/src/App.js` and changing some of the text content in the element(s), you'll see the text update in the browser in real-time once you save the file.

This set-up includes a fantastic toolchain for development, and comes with a lot of bells and whistles out of the box, as you are already seeing. Note that React itself *is* a front-end only library, but this environment comes with a server so you can easily see how it would look and behave if launched. Each time you hit save, it is transpiling your `/src/` files into regular JavaScript files that can be understood by the browser. Not all JavaScript libraries out there have a development system as robust as all of this!

Trimming Files you Might not Need

It is important to note that not all files that are placed in the initial project are necessary in order to run or build a React app, many are there as an example or in case you need them for your particular use-case. If you'd like to focus on only some basic React practice, consider removing the following files:

- `/src/App.css`
- `/src/App.js`
- `/src/App.test.js`
- `/src/logo.svg`
- `/src/reportWebVitals.js`
- `/src/setupTests.js`

If you remove the above, you'll also need to look inside the `/src/index.js` file and **remove** the following lines:

```
// REMOVE the following (if you got rid of files from the list:)

import App from './App'; // X
import reportWebVitals from './reportWebVitals'; // X

// ...

  <App /> // Replace this with your own components!

// ...

reportWebVitals(); // X
```

You would also, for real application, want to replace the React icon that appears in the browser tabs with one more fitting for your project:

- `/public/favicon.ico`
- `/public/logo182.png`
- `/public/logo512.png`

If you don't have software installed that can save an `.ico`, you can consider some web applications or software out there like Favicon.ico & App Icon Generator to bridge the gap.

Components

Components are just functions that return a "React DOM Element," compatible output. For cases like a `create-react-app` project, this usually simply means some JSX! Let's create our first component in a new file, `/src/components/Hello.jsx`. You can name the file using either `.js` or `.jsx` as an extension, but `.jsx` will let your editor know you might use JSX, and it may offer more suggestions to you.

We'll place the following in the file, make note of the use of ES6 Module syntax (`import` and `export` :)

```
// Components are usually named in PascalCase
function Hello() {
  return <p>Hello, World!</p>; // Return JSX, this is our "render."
}

export default Hello; // Allows us to import this function in other
files.
```

We can now call upon this component in other files. If you've slimmed down your project, we can use the `/src/index.js` file, adding an `import` statement and our component:

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

// Import your component, so you can use it in this file!
import Hello from './components/Hello';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Hello /> // Add your component to the render, or you won't see it in
the browser!
  </React.StrictMode>
);
```

Take a look in the browser... congratulations!!! Your very first component!

Props

Let's suppose we had a regular JavaScript function that said hello, like so:

```
function sayHello() {
  return `Hello!`;
}

console.log(
  sayHello()
);
```

How might we make this function more re-usable, so it could say hello to anyone? If we add a `name` parameter, all of a sudden, we have the ability to change that potential return value. Let's give it a shot:

```
function sayHello(name) {
  return `Hello ${name}!`;
}

console.log(
  sayHello('Sam')
);

console.log(
  sayHello('Quinn')
);
```

Much better, look at that! We can do the same thing in React, JSX offers us some awesome syntax sugar. In React, instead of parameters, we refer to these as "props."

Let's adapt the above example into a new component, `/src/components/SayHello.jsx`:

```
function SayHello(props) { // Components accept an object: props
  // We use curly braces in JSX to run a JS expression.
  return <p>Hello, {props.name}!</p>;
}

export default SayHello;
```

How do we populate this props object, though!? Observe their use in `/src/index.js`:

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

// import Hello from './components/Hello';
import SayHello from './components/SayHello'; // Don't forget to import!

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    // <Hello />
    // Now we can re-use this component, and customize its output with
props!
    <SayHello name="Sam" />
    <SayHello name="Quinn" />
  </React.StrictMode>
);
```

Have another look in your browser to see props in action!

Conditional Rendering

How might we make decisions in our component render (`return`)? Let's make another component to see one way this could be done.

Create a new file, `/src/components/Mood.jsx` :

```
function Mood(props) {
  return ( // Remember, multi-line JSX needs to be in parenthesis.
    <section> // You can only have one root element in the return.
      <h2>My name is {props.name}</h2>
      // We cannot easily use `if` (aside from ternary) in JSX...
      // This results in a pattern of `and` to determine if we...
      // see one output from an expression or another.
      // `false` will not result in any HTML showing in the browser.
      {props.mood && props.mood.toLowerCase() === 'happy' &&
        <p>I am feeling very happy about everything!</p>}
      {props.mood && props.mood.toLowerCase() === 'sad' &&
        <p>Today isn't a great day...</p>}
      {props.mood && props.mood.toLowerCase() === 'mad' &&
        <p>I just wanna' scream!</p>}
      {!['happy', 'sad', 'mad'].includes(props.mood &&
props.mood.toLowerCase()) &&
        <p>I don't know how I'm feeling...</p>}
    </section>
  );
}

export default Mood;
```

Note how the value returned in each expression is the last truthy value in an `&&` chain. If something results in a `false`, it simply is not rendered in the browser. Let's try using this component now, head to `/src/index.js`:

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

// import Hello from './components/Hello';
// import SayHello from './components/SayHello';
import Mood from './components/mood'; // Don't forget to import!

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    // <Hello />
    // Now we can re-use this component, and customize its output with
props!
    // <SayHello name="Sam" />
    // <SayHello name="Quinn" />
    <Mood name="Sam" mood="happy" />
    <Mood name="Quinn" mood="sad" />
    <Mood name="Ari" mood="mad" />
    <Mood name="Kade" mood="other" />
  </React.StrictMode>
);
```

This example happens to use multiple props, and conditional rendering! Have a look in the browser to confirm that it is working properly.

State

You might be tempted to begin creating variables in your components when a value you're showing to the user might change over time... but React has a special way we have to handle such a case. Recall that JSX is not actually what shows directly in the browser—there is a middle-step where React must convert that to real DOM elements and inject them into the browser. Because of this degree of separation, if we changed a variable we created, React wouldn't know to re-render the actual element in the browser.

How might we get around this? There is another concept in React called "state." If we set up a value carefully, React can pay attention to it and watch for changes. If a change in an displayed value is detected, it can re-render the component in the browser to show the user the new content! This is great for performance, as we get to decide what is important enough to watch, or not.

The classic example to experiment with this idea is a counter button. Essentially, a button we can click on that will tell us how many times it has been clicked.

Let's create a counter component, `/src/components/Counter.jsx`:

```
function Counter() {
  return (
    <button>
      Clicked 0 Times
    </button>
  );
}

export default Counter;
```

Let's check right away that we're on the right track, mention this in your `/src/index.js` file:

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

// import Hello from './components/Hello';
// import SayHello from './components/SayHello';
// import Mood from './components/Mood';
import Counter from './components/Counter'; // Don't forget to import!

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    // <Hello />
    // Now we can re-use this component, and customize its output with
props!
    // <SayHello name="Sam" />
    // <SayHello name="Quinn" />
    // <Mood name="Sam" mood="happy" />
    // <Mood name="Quinn" mood="sad" />
    // <Mood name="Ari" mood="mad" />
    // <Mood name="Kade" mood="other" />
    <Counter />
  </React.StrictMode>
);
```

Is it showing in the browser? Great! Okay, so, we don't want it to just say "Clicked 0 Times" forever. We want to be able to update that number! So, again, we have to avoid a regular variable... what does React expect us to type to mark our count as **state**? For this, we run a function (hook) they provide us called `useState()`. It takes an argument: *the default value that this state should have*.

In our case, the default we'd want would be the number `0`, so we'd want to run: `useState(0)`. This function also returns an array containing two important things:

1. `Index 0:` The current value of your state.
2. `Index 1:` A function you can use to update your state; it will notify React to re-render (`return`) the component.

Let's try keeping track of the count using the `useState` hook. Update your `/src/components/Counter.jsx` file accordingly:

```
import { useState } from 'react'; // Import the "useState" hook.

function Counter() {
  const countState = useState(0);
  const count = countState[0]; // Current count value; NEVER update
directly.
  const setCount = countState[1]; // Function to update count value.

  // Use curly braces in your return to display the current count.
  return (
    <button>
      Clicked {count} Times
    </button>
  );
}

export default Counter;
```

Have a look in the browser, ensure it is error-free so far. Okay, so now we see the real zero from our state, instead of the hard-coded string... we still don't have a way to update the count on click though. Can we somehow hook up element event listeners from React? You bet we can! Observe:

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

function Counter() {
  const countState = useState(0);
  const count = countState[0];
  const setCount = countState[1];

  // A function to help us update our count.
  const addOneToCount = () => {
    // Remember, we never modify our current state value directly.
    // We ALWAYS use the "setState" function we get back from...
    // the "useState" hook.

    // Set count can accept a direct value, or a callback.
    // The callback will receive the current/previous value...
    // for your convenience! This callback must return the...
    // new value you'd like stored in state.
    setCount((previousCountValue) => {
      return previousCountValue + 1;
    });
  };

  // We can add event listeners right in our JSX:
  return (
    <button onClick={addOneToCount}>
      Clicked {count} Times
    </button>
  );
}

export default Counter;
```

Voilà! There we have it, a component with state, and an event listener. Note how few lines we wrote to accomplish this (comments aside!) We *could* clean this up a bit to emphasize this point, you'll likely want to keep the comments for your first few experiments until you get the hang of it though:

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

function Counter() {
  const [count, setCount] = useState(0); // Array destructuring saves a
couple lines.

  return ( // We can include anonymous function definitions right in the
listener.
    <button onClick={() => setCount(prev => prev + 1)}>
      Clicked {count} Times
    </button>
  );
}

export default Counter;
```

Controlled Components

The browser keeps track of things like which values are in which form fields at any given time... React uses state to keep track of values that may change over time. This leaves us in a bit of a pickle, as we know we have a bit of a disconnect between what we write in our component and what ultimately gets printed in the browser. Thankfully, this has been taken into account. We are able to create, what is called, a controlled component.

The idea here is, we attach state to a form field and use its event listener to ensure we always have the most up-to-date value available for anything we need.

Let's explore this concept via another classic application: the to-do list! This will not be a very feature-complete one, but it will show us how to build a controlled component and iterate through a list for output.

Create a new file, `/src/components/ToDoList.jsx`:

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

function ToDoList(props) {
  // These "blank" elements are called fragments.
  // If we need to wrap some elements, but don't want any extra to...
  // show in our browser, we can use them! Handy in many cases, as...
  // JSX only lets us return one root element per component.
  return (
    <>
      <h2>{'To-Do List Component' || props.heading}</h2>
      <form>
        <label htmlFor="new-task">Enter Task:</label>
        <input
          id="new-task"
          type="text"
          />
        <input type="submit" value="Add Task to List" />
      </form>
      <p>There are no to-do items to display.</p>
    </>
  );
}

export default ToDoList;
```

We want to use state to keep track of what is in our form field. We can use the `onChange` input field event listener to update our state, and we can show the user the up-to-date state value in the `input`'s `value` attribute:

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

function ToDoList(props) {
  const [newTask, setNewTask] = useState(''); // Default is an empty
string.

  // Function for our input changed event.
  const newTaskFieldChanged = (event) => {
    // Get the new value entered by the user from the event element.
    const enteredValue = event.target.value;
    // Update the state.
    setNewTask(enteredValue);
  };

  return (
    <>
      <h2>{'To-Do List Component' || props.heading}</h2>
      <form>
        <label htmlFor="new-task">Enter Task:</label>
        <input
          id="new-task"
          type="text"
          onChange={newTaskFieldChanged} // Listen for change.
          value={newTask} // Show the user the latest input.
          />
        <input type="submit" value="Add Task to List" />
      </form>
      <p>There are no to-do items to display.</p>
    </>
  );
}

export default ToDoList;
```

Awesome!! Check the "Component" tab in your browser developer tools. Find your To-Do component, and see if the state updates as you type. If it does, awesome! We're on the right track. However, we are not yet making a *list* of anything... let's give that a shot!

We'll need to listen for `<form>` submissions... and while we're at it, we'll want to let our form know not to submit in the traditional way—or we'll end up having a pageload happen and lose our submission data to a non-existant back-end!

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

function ToDoList(props) {
  const [newTask, setNewTask] = useState('');
  const [toDos, setToDos] = useState([]); // Default is an empty array.

  const newTaskFieldChanged = (event) => {
    const enteredValue = event.target.value;
    setNewTask(enteredValue);
  };

  const toDosFormSubmission = (event) => {
    event.preventDefault(); // Prevent form from loading new page.

    // Return the array along with its new value.
    setToDos((prev) => {
      return [...prev, newTask];
    });

    // Clear out the text input so the user can type a new one nice and
easy!
    setNewTask(''); // Just an empty string.
  };

  return (
    <>
      <h2>{'To-Do List Component' || props.heading}</h2>
      <form>
        <label htmlFor="new-task">Enter Task:</label>
        <input
          id="new-task"
          type="text"
          onChange={newTaskFieldChanged}
          value={newTask}
          />
        <input type="submit" value="Add Task to List" />
      </form>
      {toDos.length === 0 && <p>There are no to-do items to display.</p>}
      {toDos.length > 0 && <ul>{toDos.map((toDo, index) => <li
key={index}>{toDo}</li>)</ul>}
    </>
  );
}

export default ToDoList;
```

Notice the use of the [array `map()` method]. It gives us a convenient way to return a new / updated array of JSX to be displayed in our render.

We now have a controlled component! Our rendered output, and our state, will remain in-sync. Congratulations! You've covered the core concepts and essentials for React. The rest is practice and becoming more familiar with React!

Resources

- Add React in One Minute
  - Try JSX
- JavaScript XML (JSX)
- Babel
- JavaScript ES6 Module Syntax
- Creating a Production Build
- React "StrictMode"
- Components and Props
- Using the State Hook
- Forms in React
- `Array.map()`
- Robots.txt
- Favicon.ico & App Icon Generator
- Oreilly's Learning React, 2nd Edition (Warren's Favourite Resource!)

View on Compass | Leave Feedback