React Intro

Prior Knowledge

- Understanding and awareness of basic HTML elements
- DOM tree structure
- DOM manipulation
- JavaScript

Learning Objectives

- Understand how a single page application is structured in React.
- Be able to use JSX to create html elements.
- Be able to use JSX to insert JS values into html.

React JS

"The library for web and native user interfaces" - React Homepage

Why React ?

Making changes through DOM manipulation can be slow as well as difficult.

- The DOM is relatively slow to update due to it's tree structure, which requires recursion to traverse.
- React gives developers the ability to work with a ***virtual DOM***, removing the need to use the browser's inbuilt DOM methods.
- A benefit of this is that it allows us to be ***declarative*** when building our User Interface (UI) using JavaScript. This means that rather than having to describe each step in the transition to an updated UI using DOM methods, we can simply write the final version of the page that should be displayed with React.
- The ***virtual DOM*** used by React is much faster and more efficient to update, as instead of manually traversing through the parent and child nodes of the DOM, the ***virtual DOM*** will check itself against the previous version and only update what is necessary, based on the updated UI that has been declared.
- As a result, React makes it easy to "hydrate" a page with data from API requests on the client side.

Basic React Set-up

React can be used as part of a larger framework to provide out of the box features, like routing. We'll focus on React itself and set up a standalone React project using Vite. This is an example of what's known as a `build tool` and there are several good options out there but Vite is very good at what it does and comes recommended by the React docs which is why we're using it.

The file structure of a typical React application will something like this:

```
├── public
│   ├── vite.svg (plus any other static assets)
└── src
    ├── App.css
    ├── App.jsx
    ├── index.css
    └── main.jsx
├── index.html
├── package-lock.json
├── package.json
```

In the majority of standalone React apps that you see and create, there will be a `div` with an `id` of `root` within an `index.html` file. This is the *Single html page* that the rest of our React code will be injected into.

```
<!-- the rest of the html file... -->
<div id="root"></div>
<!-- html file continued... -->
```

This is referred to as the "root" DOM node because everything inside it will be managed by React, and this is where our React apps will be rendered (displayed).

The `src/main.jsx` module is also added and this will serve as the entry point from which we will write all of our React code

```
<script type="module" src="/src/main.jsx"></script>
```

Within the `main.jsx` file, we can render a React element inside the root DOM node using `root.render()`:

```
import ReactDOM from 'react-dom/client';

const element = <h1>Hello, world</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(element);
```

The `element` that has been created above looks just like a HTML element, but the above example is a blend of JavaScript and HTML syntax, this is in fact a syntax extension of JavaScript called *JSX*.

JSX effectively allows us to write HTML syntax inside our JavaScript files. See Writing Markup with JSX for more info.

JSX

Notice the way that the `react-dom` package is required (or in this case ***imported***) into the file. This is module system called ES6 modules that is enabled by `type=module` in the script tag. These are supported by the browser whereas CommonJS(require) is not so we'll be using them going forward. See the section below for a list of equivalent patterns.

When we we write JSX in our React code it is compiled to *regular JS* using a tool called esBuild (again other tools are available. Babel being a notable option.) For example:

```
const element = <h1>Hello world</h1>;
```

Compiles down to:

```
import { jsx } from 'react/jsx-runtime';

const element = /*#__PURE__*/ jsx('h1', {
  children: 'Hello world',
});
```

Here we can see that the `jsx-runtime` has been imported automatically and the element is actually a call to this function. This is why it can be saved to a variable and treated as a *normal JS value*.

n.b. This behaviour of automatically importing the `jsx-runtime` is new to React 17. Previous versions of React compiled JSX to `React.createElement` instead. This required React to be in scope and as such we would have to add the following import to any file using JSX.

```
import React from 'react';

const element = <h1>Hello world</h1>;
```

You may still see this import in projects using React versions < 17

React Elements

JavaScript can be embedded within JSX using curly braces. E.g.

```
const sum = <p>1 plus 2 = {1 + 2}</p>; // {1 + 2} will be evaluated to 3

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(sum);
```

See Using curly braces: A window into the JavaScript world for more info.

We can embed several JS data types, notably `Strings`, `Numbers` and `Arrays`. `Objects` however are not valid and React will throw an error if we try to embed one. If we want to render the information from an object we should render the properties into appropriate tags.

```
const person = {
  name: 'Paul',
  age: 34,
};

const profile = (
  <section>
    <h1>{person.name}</h1>
    <p>{person.age} years old</p>
  </section>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(profile);
```

Displaying Lists and Keys

React allows us to render arrays. Each element of the array will be rendered to the DOM in order. Good semantically correct html should be made up of an `ul` or `ol` tag containing `li` tags as their children.

A common technique is to use `.map` to transform an array of JS values into an array of `li` tags. See the React docs for more info on Rendering Lists.

```
const listItems = ['One', 'Two', 'Three', "Four o'clock rock"];

const rockSchedule = (
  <ul>
    {listItems.map((item) => (
      <li key={item}>{item}</li>
    ))}
  </ul>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(rockSchedule);
```

Each item in the array must be given a key prop. This is for React to identify each element when adding / removing or re-ordering elements in the array. The key should be stable and unique. Best practice here is to use an id or some unique feature of each element.

See the React docs on keys for more detail on how they are used and why indexes should only be used as a last resort.

Dan Abramov (React.JS dev & co-creator of Redux and Create-React-App) also posted this fantastic twitter thread explaining ***Why React Needs Keys***.

ES6 Modules

In React we will use ES6 modules to import and export variables as opposed to the common JS modules that we used in node.

MDN has an excellent guide on how they work but some of the more common patterns are shown below:

Default exports - exporting a single value

To export a value from a file in node we would assign that value to the `module.exports`. `ES6 modules` come with the ability to export a single `default` value which can be imported in other files.

```
// sayHello.js
function sayHello() {
  console.log('hello world');
}

export default sayHello;

// another file
import sayHello from './sayHello.js';
```

is equivalent to

```
function sayHello() {
  console.log('hello world');
}

module.exports = sayHello;

// another file
const sayHello = require('./sayHello.js');
```

Named exports - exporting multiple values

When exporting multiple values in node we typically export an object containing those values and destructure them when importing. The same can be done with the `export` syntax put in front of a variable we want to export.

```
// functions.js
export const foo = () => {};
export const bar = () => {};
export const baz = () => {};

// another file
import { foo, bar, baz } from './functions';
```

is equivalent to

```
// functions.js
exports.foo = () => {};
exports.bar = () => {};
exports.baz = () => {};

// another file
const { foo, bar, baz } = require('./functions');
```

When importing named exports they can be renamed or collected into a single object using the `as` syntax.

```
import * as myFunctions from './functions';

console.log(myFunctions); // { foo: [function foo], bar: [function bar], baz: [function baz] }
```

Additional features

There are some additional features of `ES6 modules` that do not have equivalents in node.

For example, you can have both a `default` export as well as several `named` exports.

```
// myLibrary.js
function myLibrary() {}

export const helperFunction1 = () => {};
export const helperFunction2 = () => {};

export default myLibrary;

// another file
import myLibrary, { helperFunction1, helperFunction2 } from './myLibrary';
```

copy original markdown to clipboard

v.0.1