# React Notes

Kamal Sacranie

2021-07-13

# Contents

# Preface

These are my react notes.

## 0.1 What is React?

React is a library for building UI:

- React runs on the client as a single page application abut can be used to build full stack apps by communicating with an API
- React is a frontend framework
- React has a whole bunch of plugins that allow you to do a lote more

## 0.2 Why use react

React allows the best way to build your UI using different components which allows for very dynamic HTML within JavaScript. It does this using JSX (which is extended JS). React uses what is called the Virtual DOM which allows us to have a more dymic environment etc.

With react, you'll look at your UI as a bunch of components which you will write code for.

Chapter 1

# Creating a React app

To create our boilerplate code, we want to use something called 'Create React App'.

To install this we want to run `npx create-react-app <app-name>`. This uses the node package manager so **Node must be installed**. This sets up our frontend.

In these created files we have `package.json` which tells us exactly what dependencies/addons are installed in React and we **change this when we add an extension** (this wil ususally be in the docs of the extension)

### 1.0.1 What we get in our boilerplate

We get an `index.html` which basically just has a body tag with `<div id="root"></div>` which is basically the crux of how our React app gets rendered.

In the `src` folder, inside the `index.js` which is our front-most part of our React app, we grab the `root` elemtent from our `index.html` and render our `<App />`. This `<App />` tag comes from importing our root app from `App.js` which houses all of our components. Basically:

```
graph LR:
    A[Components] -> B[App.js]
    B -> C[index.js]
    C -> D[index.html]
```

### 1.0.2 Starting up our dev server

You must change directory into your React app folder and then we can run `npm start` which will start our server on **localhost:3000**.

## 1.1 Components

In react, you can create components with funcitons and classes. The more common way these days is to **use functions**. Class components extend other classes.

Your basic functional component looks as follows:

```
let myVar;

export const Header = () => {
    return (
        <some HTML>
```

```
            {myVar}
            {1 + 2} // This will render as 3
        </some HTML>
    )
}
```

- Because we use the component in other `.js` files, we use the `export` keyword to denote that we are exporting the component for use elsewhere.
- We are also using arrow funciton here
- Note that the return is returning HTML. This isn't really HTML but rather JSX (JavaScript Syntax Extension) which allows us to use HTML and any JS expression to make our pseudo-HTML dynamic.
- We also see that we can have JS varaibels inside curly braces
- You must also only return one single top level element

You can also pass in 'props' to your components which allows you to pass data into your components which can be used.

Components have what is known as 'state' which is an object that determines how a component renders and behaves. This could be a value like 'opne' or 'close'. Sometimes you want App level state which referes to state that is available to the entire UI and not just a single component. This is done through Redux.

> In React and JSX, we **have clashes** of HTML tag attributes like `class` and `for`, to get around this we use `className` and `htmlFor` respectively.

### 1.1.1 Creating components

We create our components inside our `src` file inside a `components` file as `.js` files.

You can use VSCode snippets addon to create the boilerplate for a component which will look as follows:

```
const MyComponent = () => {
    return (
        <html>
        </html>
    )
}

export default MyComponent
```

> Note that we name our components in proper case.

We must then import it into our `App.js`:

```
import MyComponent from './componennets/MyComponent'

function App() {
    return (
        <div>
            <MyComponent /> // calling our component here
        </div>
    )
}
```

- Notice here that when we import we use quotes for te path name
- We **call our component with `<ComponentName />`**

## 1.2 Props

Props are basically parameters for your functional componenets which are **passed in through the tag**. We can access it in our component as an object.

```
const MyComponent = (props) => {
    return (
        <html>
            <h1></h1>
        </html>
    )
}

// ====== In the file where we call the component ====

<MyComponent title='Hello'>
```

- As you can see, you do not need to expliclity define `title` but can pass it in and access it from a dictionary.

### 1.2.1 Default props

We can have default props for our components by doing the following:

```
const MyComponent = ({ title }) => {
    return (
        <html>
            <h1>title</h1>
        </html>
    )
}

Header.defaultProps = {
    title: 'My Title'
}
```

- Note that here we have descructured the object by using the {} which allows us to just directly access title.

We can also `import PropTypes form 'prop-types'` to typeset your props using KVPs in the object `MyComponent.propTypes = {title: PropType.string}`

### 1.2.2 Componenets in components

It is simple to have components within components, all you have to do is perform the relative import in your current component and the component will be in the same directory so its just `'./Component`.

# Chapter 2

# Styles

## 2.1 In-line styles

If we are going to add styles inline in one of our React componenets, we can use the fact that JSX allows for HTML/CSS. This is donw within a tag using double curly braces:

```
const Component = () => {
    return (
        <h1 style={{ backgroundColor: 'black', color: 'red' }}>
        </h1>
    )
```

- Note here that we use double curly braces
- We also have CSS elements in camelCase
- Style elements are also seperated by `,` not `;`

## 2.2 Via a variable in the same JS file

This is pretty self explanitary. We can just create an object somewhere else in our JS document and add the style inline using the constant keyword that we created.

## 2.3 Using a CSS file

Using a CSS file is pretty simple, you just relate the link in your `index.html` file.

# Chapter 3

# Create state without Redux

To create state without Redux, we can create a component containing the data we want to use and import the `useState` function from React:

```
import { useState } from 'react'

const Data = () => {
    const [data, setData] = useState(<myObjectOfData>)

    return (
        <>
            {data.map((task) => (
                <li key={data.id}>{data.text}</li>
            ))}
        </>
    )
}
```

- Here we are importing `useState` and then using it to properly assign data to a variable
- We also use the map function to basically perform a for each loop on our data and then pass in the function we want to perform with each entry of data
- We also pass in a `key` attribute which is required for React

We don't typically want our data in to be in one component however, so we typically use Redux to create a state which is avaialbe to all of our components. For small projects which we don't want to use Redux for, we can put the data in our `App.js` file to emulate a global state and pass it through as props to our components.

To update our immutable state we use `setData` which we defined in **??** which is a funciton that we can use with `filter()` to change our state. Basically if we want to do something like this we can Google it on the spot.

> If you set the state within your functional component using `useState()` it is what is known as component level state.

## 3.1 The `useState` function

What the `useState` function actually does is:

1. It takes in our default state, and returns an array with 2 entries.

2. The first element is **our state** which we passed into `useState` and the second element is **a function which allows us to update our state later**

An example of how we would update our sate would be if we did the following in `App.js`:

```
import { useState } from 'react'

function App() {
    const [count, setCount] = useStata(4)

    const decrementCount = () => {
        setCount(previousCount => prevCount - 1)
    }

    return (
        <>
            <button onClick={decrementCount}>-</button>
            <span>{count}</span>
            <button>+</button>
        </>
    )
}
```

- Here we **desructure** our array response from `useState` into `count` and `setCount`.
- We then create a function which runs on the click of the button which decreases our count
- When the page loads, we load in our default state which is `4`
- Every time we call the `decrementCount` our component rerenders our component
- Any time you are modifying your state incrementally you must pass in a function to our `setCount` function which is what the react docs say

Use state can also take a function input which atually **only runs once** which is good for the performance of our webpage. For this to happen you have to pass the function directly in and not reference the function.

## 3.2 The `useEffect` function

Use effect basically handles side effects and it takes a function as an argument. `useEffect` is commonly used with asyn await to get a response from a backend like for example when using Django we would do the fillowing:

```
const App = () => {

// code

    useEffect(() => {
        const fetchData = async () => {
            const data = await axios.get('path/to/endpoint')
        }
        fetchData()
    }, [])
}
```

- Basically we've fetched the data from our backend to use in our React component. We have then called the `fetchData` function to got our data
- We then passin an empty array as a second argument which usually takes values which we want useEffect to run when the value changes

# Chapter 4

# Static assets

If we are ready to deploy, we can run `npm run build` which creates a build in a folder called build. Basically all our react gets converted to vanilla JS which we can deploy. So basically we would deploy the files in the build folder.

You can go about deploying your files in many different ways. Anything that hosts JS, HTML, and CSS static files can host regular react apps. If you have a database, it becomes a bit harder to host because you need a server to run your backend.

# Chapter 5

# Routing in React

Natively we cannot rout with react and I'm sure there is some way to use Django for routing but it's easier to just use react router dom. We just need to `npm install react-router-dom` which will automatically show up in our `package.json`which enables us to route.

When we want to create a route we need to `import { BrowserRouter as Router, Route} from 'react-router-dom`. Then we will wrap our whole return statement in `App.js` in the `<Router>` tag which allows us to then use `Route`:

```
cont App = () => {
    return (
        <Router>
            // HTML
            <Route
                path='/'
                exact
                render={(props) => (// what you want to render)}
            />
            <Route path='/about' components={About} />
        </Router>
    )
}
```

This basically allows us to selectively render stuff depending on the path so in essence your webapp is a single page application. This would render something when we match the **exact** route of `/` and then when the route is `/about` we would get the about component rendered.

## 5.1 React links

This allows us to not reload the page when routing with React. We do this directly in the component concerned like if we had an about component which directed us to the home page, we would:

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

const About = () => {
    return (
        <>
            <Link to='/'>Home</Link>
        </>
    )
}
```

This would just take us home and not force us to reload the page like we would with an anchor tag

## 5.2 `useLocation`

This allows us to use the current URL to selectively render elements depending on our route. We would `import { useLocation } from 'react-router-dom'` and then define a variable which just runs the function which gives us access to the path name and we can use a conditional to show something depending on the path.