Reverting to React 17

`npx create-react-app` builds React applications to the newest stable version of React, but sometimes we might have to work with an older version. Currently, the platform is written for React 17, which as of March 2022, is now an older version! We still want to use `npx create-react-app`, but we also still want version 17. Without it, we can't use React Router as described in the next tab!

Officially `npx create-react-app` is recommended for learning React, however it installs React 18 by unchangeable default, and many of the newest features of React 18 won't be relevant to us yet.

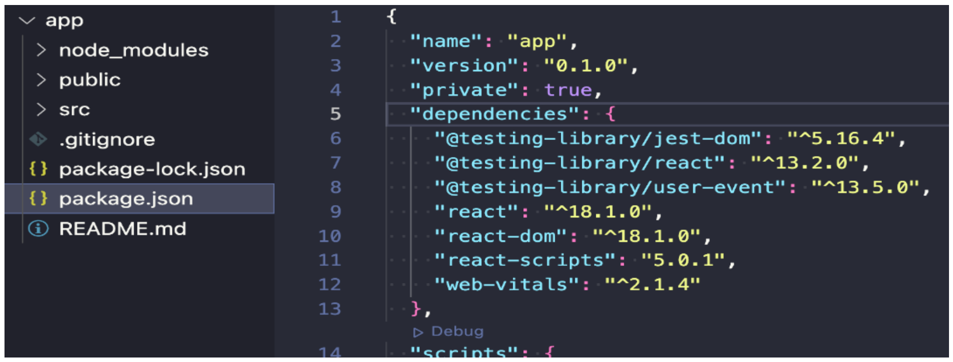
For this stack, we will use `npx create-react-app` and the below steps to ensure we work in React 17.

Step 1 – Update package.json

Start with the create command and name your project folder. For this example, our folder will be named app:

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

Navigate into our new app folder and investigate package.json. It should look something like this:



```
{
  "name": "app",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.16.4",
    "@testing-library/react": "^13.2.0",
    "@testing-library/user-event": "^13.5.0",
    "react": "^18.1.0",
    "react-dom": "^18.1.0",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
```

Replace

```
"react": "^18.1.0",

"react-dom": "^18.1.0",
```

with

```
"react": "^17.0.2",

"react-dom": "^17.0.2"
```

Note: Your original version number might be slightly different, just make sure to set ^17.0.2

Step 2 - Update index.js

Inside the folder labeled src you will find your index.js file. Replace

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

with

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

Additionally replace this below statement

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

root.render(

   <React.StrictMode>

            <App />

   </React.StrictMode>

);
```

with

```
ReactDOM.render(

   <React.StrictMode>

            <App />

   </React.StrictMode>,

   document.getElementById('root')
```

Step 3 - Reinstall node_modules

Reinstall your `node_modules` by navigating into `app` and running `npm install`. It should pick up the modified `package.json` from Step 1 and install React 17.

Notes

You can use this same method to install even earlier versions of React. It is not uncommon to have to go back to a prior version for a piece of software. This method will also allow you to use `react-router-dom@5`.

What is Routing

You should be familiar with routing at this point. When you go to https://www.codingdojo.com, you are visiting a website. If you then visit https://www.codingdojo.com/web-development-courses, you are going to another route. This follows the request/response cycle, where you make a request to the server, and it sends HTML, CSS, and Javascript back to you dynamically depending on what route you are visiting.

However, in a Single Page Application, routing does not make sense in the same way. We know that when you first visit a site, all the HTML, CSS and Javascript are all loaded. So, how would routing even work? It seems like it would be inefficient to reload everything with a new route, and it defeats the purpose of a Single Page Application, right?

Well, we need to change our understanding of routing when dealing with SPAs. In SPAs, we will use routing, but only in a very superficial sense. We will go to a new route, but we will actually not necessarily be making another request to the server. Instead, this pseudo-route will just tell our SPA which part of the page we want to see. So, if you are not making requests to an API, visiting a new route in an SPA is not actually making another request. It is just a way to tell our SPA the portion of the app we want to see. This gives the illusion to the user that they are visiting another route.

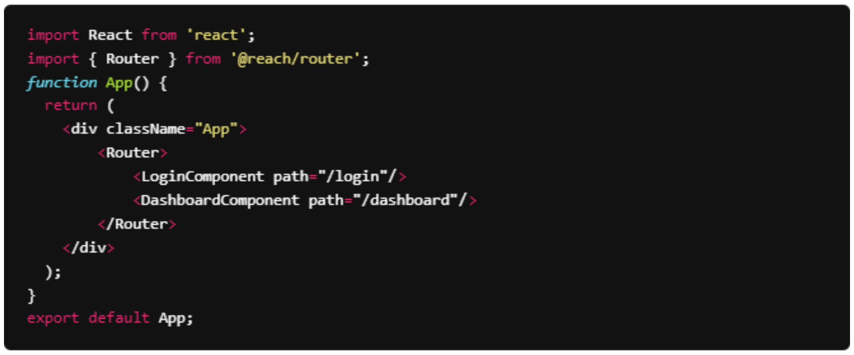
What is Routing

Let's consider a dynamic site setup. If you are on a website like Facebook, and it is using an SPA, going to a new page changes the UI (HTML, CSS and Javascript), but there is also data that is loaded. We know that if we refresh a page, and there has been a new Facebook post, that post will show up after refresh. That clues us in that the data is not sent in with the first response. This is because when we go to a new "pseudo-route" in which it grabs data from our database, we actually are going to make an asynchronous request (AJAX) to our server to grab the data, and just update the DOM with that data. Other than that, the HTML, CSS and Javascript will just be loaded based on what was sent in the first request.

Reach Router

Since we have covered the idea of routing in SPAs, let's look at a way to implement it in React. React has a variety of third party libraries we can use for this purpose, such as React Router and Reach Router. We will be covering Reach Router, but be aware that React Router is also a very popular solution that is widely used. Both follow similar naming conventions as well.

So, let's look at our App.js file that acts as an entry point into our React project. To use Reach Router in a project, first run `npm install @reach/router` to install it and add it to the project's dependencies list. We want to wrap the part of our website that relies on routing within the `<Router>` tag. This will then create a container that all of our routes will live in.



```
import React from 'react';
import { Router } from '@reach/router';
function App() {
  return (
    <div className="App">
        <Router>
            <LoginComponent path="/login"/>
            <DashboardComponent path="/dashboard"/>
        </Router>
    </div>
  );
}
export default App;
```

Within the `<Router>` component, we have two more components with a `path` prop. These tell React that when we go to the path `/login`, we want to update the DOM so that it shows the `LoginComponent` within the `<Router>` component wrapper. It looks like it is re-routing the page to a new url. In reality, it is just changing what we see on the webpage, without a true refresh of the page. AJAX requests may be firing in the background, but the DOM is changing based on front end Javascript.

Link

Now that we have a router set up, we want to be able to link to other pages. Traditionally, this is done with an html a tag and an href attribute. However, a <Link> component will not refresh the page. It will simply change the url and change the DOM.

So, you can have a navbar that looks something like this:

It is as simple as that. Now, we have the opportunity to add front end routing to our project.

```
import React from 'react';
import { Link } from '@reach/router';
const NavBar = (props) => {
  return (
      <div>
        <Link to = "/dashboard">Dashboard</Link>
        <Link to = "/login">Login</Link>
      </div>
  );
}
```

Navigate

What if we wanted to redirect a user to another page in our React project? This can be done with navigate (import { navigate } from '@reach/router'). After performing some action, we can run a method such as navigate('/success'), and this will programmatically redirect the user to that front end route.

Routing with Parameters

With Reach Router, we have an extremely easy way to pass parameters through our React project. Our front end url parameters will be passed down as props to our component. Let's say we have the following:

```
function App(){
    return (
        <div className="App">
            <Router>
                <ListOfDogsComponent path="/dogs" />
                <DetailDogComponent path="/dogs/:id" />
            </Router>
        </div>
    )
}
```

Then, within our Detail dog component:

DetailDogComponent.js

```
import React from 'react';
const DetailDogComponent = props => {
    return (
        <p>You are looking at the dog with id {props.id}</p>
    )
}
```

That is it!

Note: Regardless of what we send down in our url, the type of the parameter will always be a string. If you need it to be a number, you will need to convert the value into a number.