**Exercice 1: Installing React on your Local Browser**

**1.1. Intro:**

The first thing to install is a good installer!

You need a way to download and install software packages easily, without having to worry about dependencies. We’ll be using a popular package manager named npm.

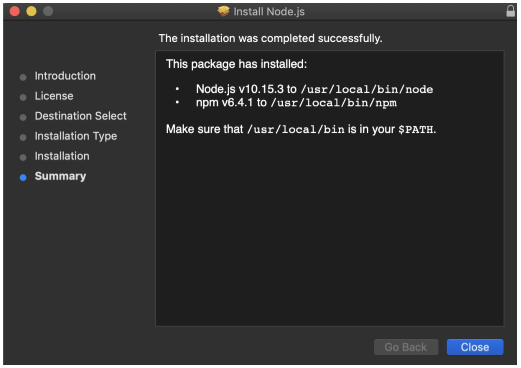
npm is a great way to download, install, and keep track of JavaScript software.

You can install npm by installing Node.js. Node.js is an environment for developing server-side applications. When you install Node.js, npm and npx will install automatically.

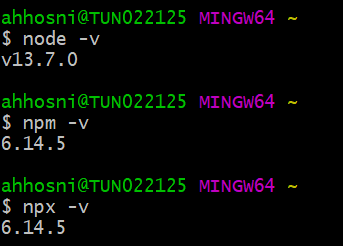
**1.** <https://nodejs.org/en/>

**2.** Download link of your choice.

**3.** Follow subsequent instructions to install Node.js and npm. Already installed Node.js? that's okay, do it anyway.



To check if node, npm and npx is installed go to your console or terminal and type:



You must be wondering, what is npx. Well, npx is what will help us set up a modern React web app by running just one command with no build configuration. This saves you a lot of time, I’d recommend you review how React application is configured to understand how helpful this is. [Click ​here](https://medium.com/@vikasharry03/react-setup-on-local-computer-912f9a551af3)​ to check it out.

**1.2. Installing React**

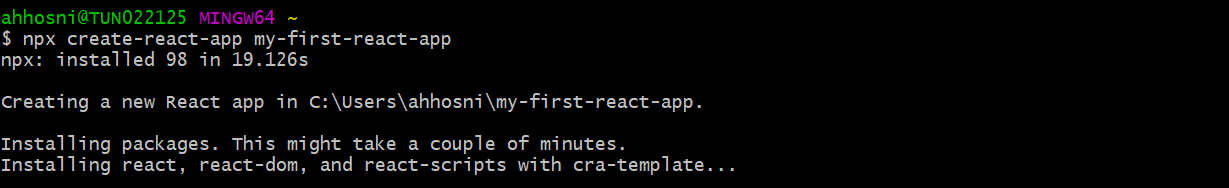
We are going to be using the **create-react-app** to install React, this enables you to create a new React project.

This program creates a directory with a React project in it, to do this, choose a directory of choice, for this project I have a directory called react projects in my documents directory.

Using terminal/console, cd into your projects directory and type the command:

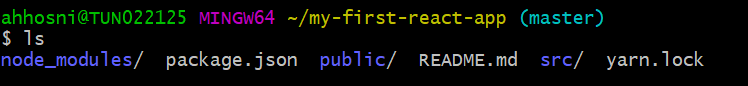
**$npx create-react-app my-first-react-app**

Where ​my-first-react-app​ is the name of my project.

****

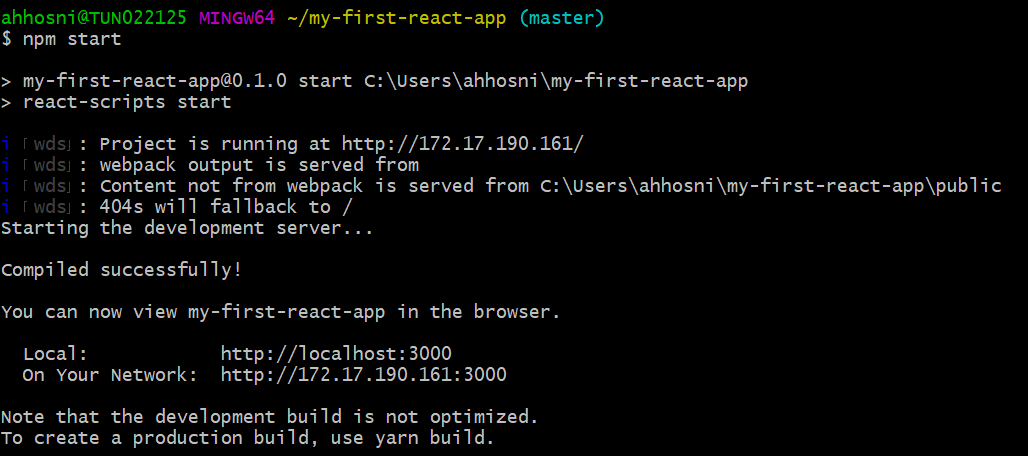
This installs all the dependencies you need to setup react, and how to startup and run your react application.

Once complete, you’ll have a new directory called my-first-react-application, cd into this directory to view your project directory.

****

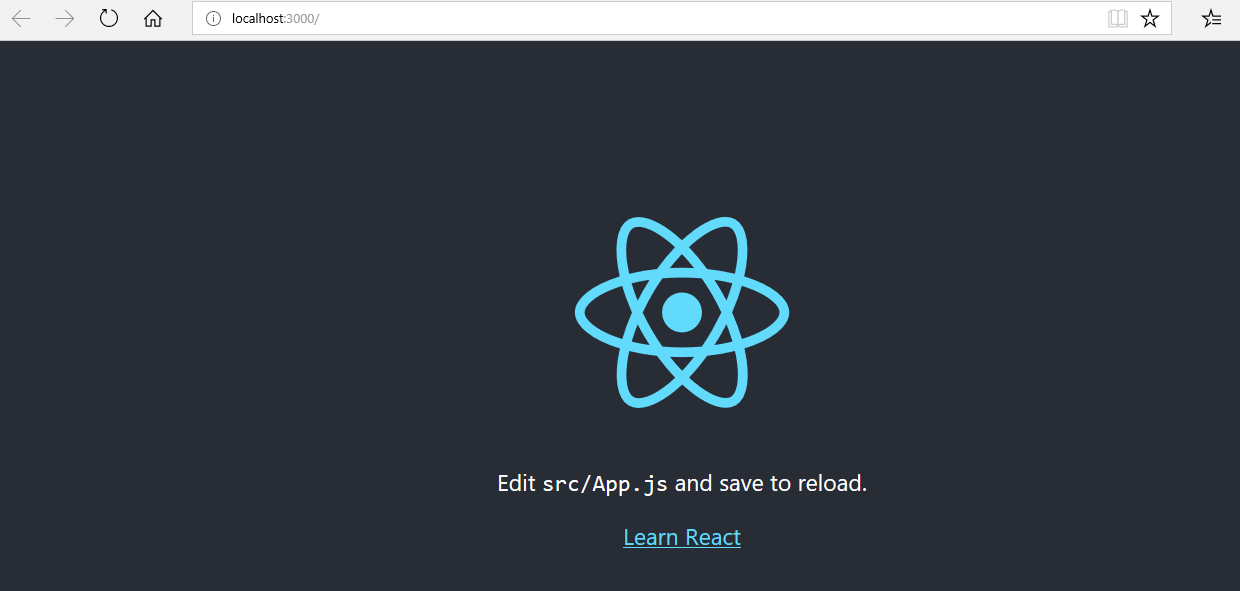
Let’s then open this in an editor, like ​Visual Studio Code​ or ​Atom.

To start your React application on your browser, go to your terminal/console and type: npm start, that’s it.



You can now view your application on your browser by going to:

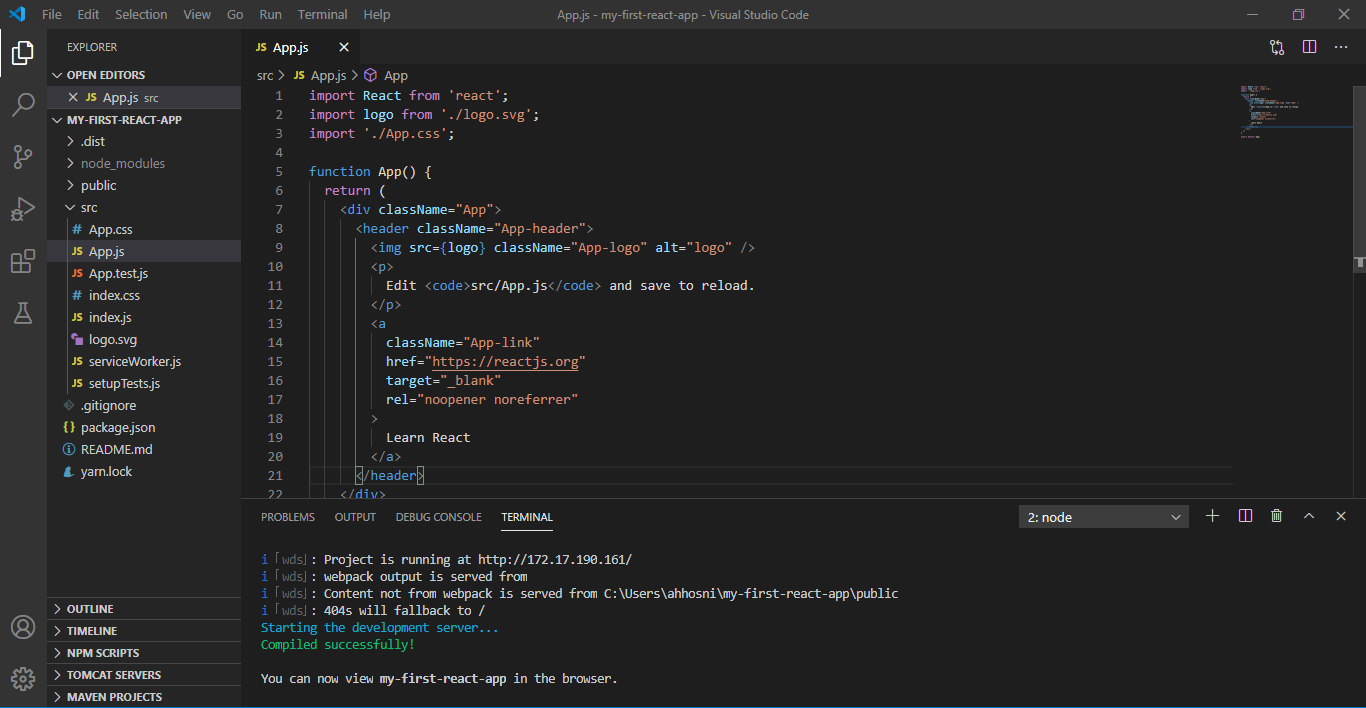
<http://localhost:3000/>



**1.3. Introducing JSX**

**1.3.1.Quick Overview before we get into details:**

Open your React project in your favourite editor, and in your **src** directory, you’ll find **App.js**. Open src/App.js.

****

Notice the function called App?

That function contains code that looks like HTML, this code is known as JSX. They’re components that we can play around with. We’ll be working mostly on this App.js file. Now let’s get started.

**Let’s get our hands a little dirty:**

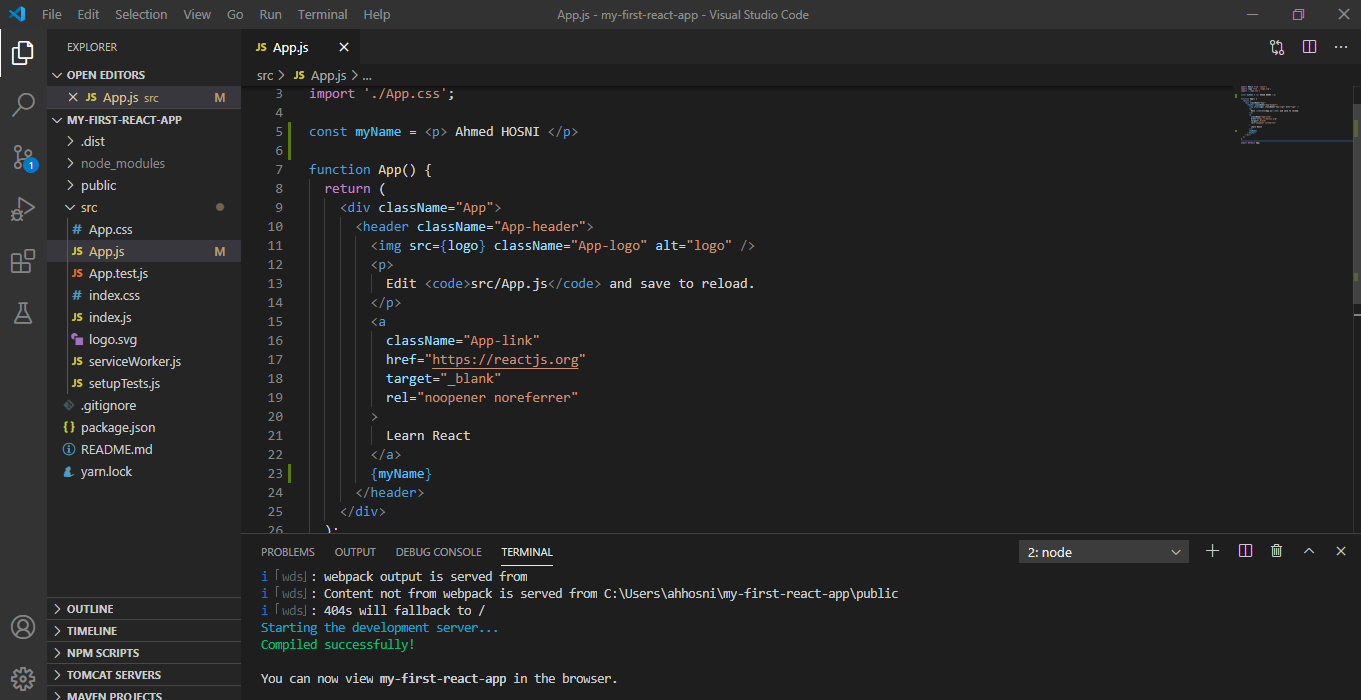
**1.** Open src/App.js in your favourite editor

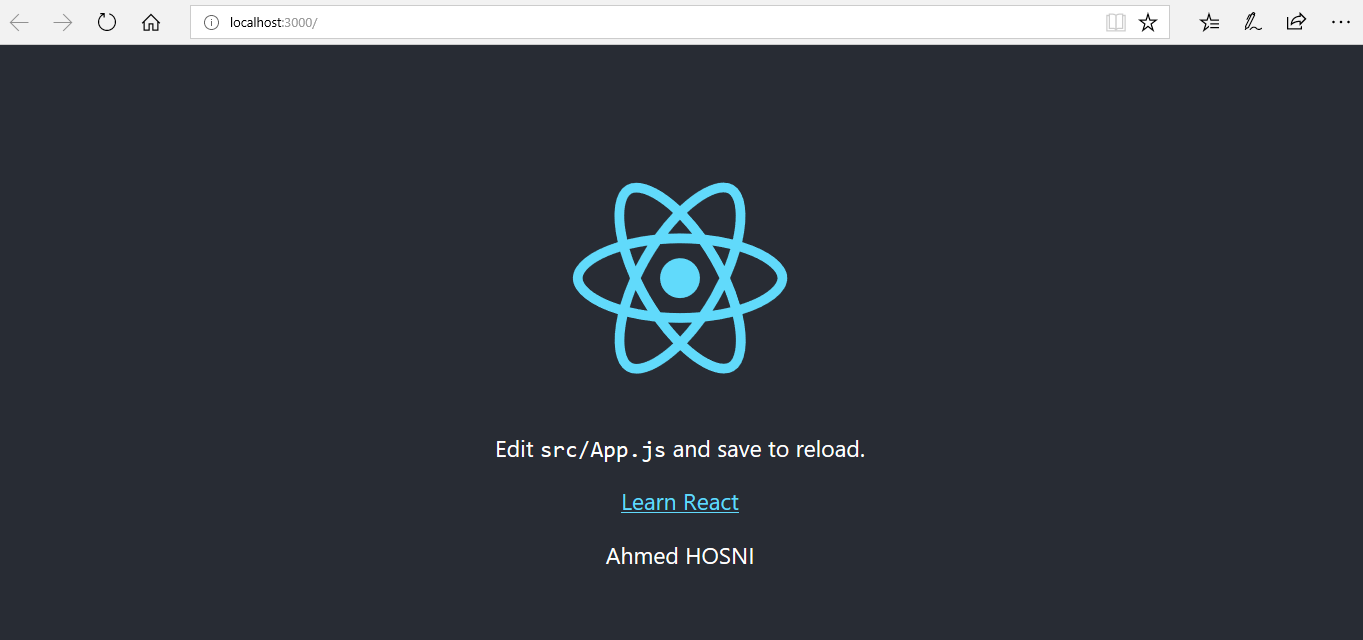
**2.** Add this simple paragraph to a variable myName:

const myName = <p> Ahmed HOSNI </p>

**3.** Just below Learn React link, add this variable ​{myName}

**4**. Save your changes and see the magic in your browser!

****

****

**1.3.2. Attributes In JSX**

JSX elements can also use attributes, just like HTML elements do. The syntax is similar, a name followed by an equals sign and followed by a value wrapped in quotes.

**Example:**

const sub\_title = <h2 id=”sub\_title”> Attributes In JSX </h2>

const puppy = <img src=”images/puppy.jpg” alt=”puppy” width=”400px” height=”400px”/>;

**1.3.3. Nested JSX**

JSX elements can be nested inside other JSX elements, just like you have it in HTML. To make your expressions more readable, you can use HTML-style line breaks and indentation. When a JSX expression takes more than one line, you must wrap the multi-line JSX expression in parentheses.

(

<a href=”​https://abc.com​”>

<p>

Abc

</p>

</a>

)

You can save nested JSX expressions as variables or even pass them into functions. E.g.

const abc = (

<a href=”​https://abc.com​”>

<p>

Abc

</p>

</a>

)

**1.3.4. JSX Outer Elements**

A JSX expression must have exactly one outermost element, that means the below code will work:

const paragraphs = (

<div id=”outermost-element”>

<p>first paragraph</p>

<p>second paragraph</p>

</div>

)

But, this will not work:

const paragraphs = (

<p>first paragraph</p>

<p>second paragraph</p>

)

It is easy to forget this rule, a simple solution is to wrap your JSX in a **<div></div>**

**1.3.5. Rendering JSX - ReactDOM.render()**

Making your JSX appear on your browser screen is also known as rendering your JSX.

ReactDOM.render(<h1>Hello world</h1>, document.getElementById('app'));

What is ReactDOM, this is a JavaScript library containing specific React methods which deal with the DOM.

ReactDOM’s render method is the most common way of rendering JSX, It takes a JSX expression, creates a corresponding tree of DOM nodes, and adds that tree to the DOM. That is the way to make a JSX expression appear on screen.

The first argument passed to ReactDOM.render() should be a JSX expression, and it will be rendered to the screen. Where on your screen should your JSX be displayed? The second argument specifies which part of your screen to display your JSX element.

document.getElementById('app')

The above code selects an element in your HTML with *id = “app”*, e.g.

<div id="app">

<h1>Render me!</h1>

</div>

**1.3.6. Rendering JSX - Passing a Variable to ReactDOM.render()**

The first argument of ReactDOM.render() evaluates to a JSX expression, which means it could be a variable, so long as that variable evaluates to a JSX expression.

const todo = (

<ol>

<li>Go Shopping</li>

<li>Dinner</li>

</ol>

);

ReactDOM.render(todo, document.getElementById('app'));

Here’s a complete JSX expression. We’ll talk about import statements later.

import React from 'react';

import ReactDOM from 'react-dom';

const myList = (

<ul>

<li>center of God's plan</li>

<li>love</li>

<li>health</li>

<li>wealth</li>

<li>Intelligence</li>

<li>global impact</li>

</ul>

)

ReactDOM.render(myList, document.getElementById('app'));

**1.3.6. The Virtual DOM**

ReactDOM.render only updates the DOM elements that have changed, if you render the exact thing twice in a row, the second render will not be displayed.

const hello = <h1>Hello world</h1>;

// This adds "Hello world" to the screen:

ReactDOM.render(hello, document.getElementById('app'));

// This does nothing:

ReactDOM.render(hello, document.getElementById('app'));

This is a very important concept, updating only the necessary DOM elements is a major reason why React is very successful. This is accomplished as a result of the virtual DOM.

**1.4. Advanced JSX**

**1.4.1 class vs className**

JSX language is very similar to HTML, but there are some differences to be mindful about. One of the most frequent differences is the use of the word class.

We use class commonly as a HTML attribute:

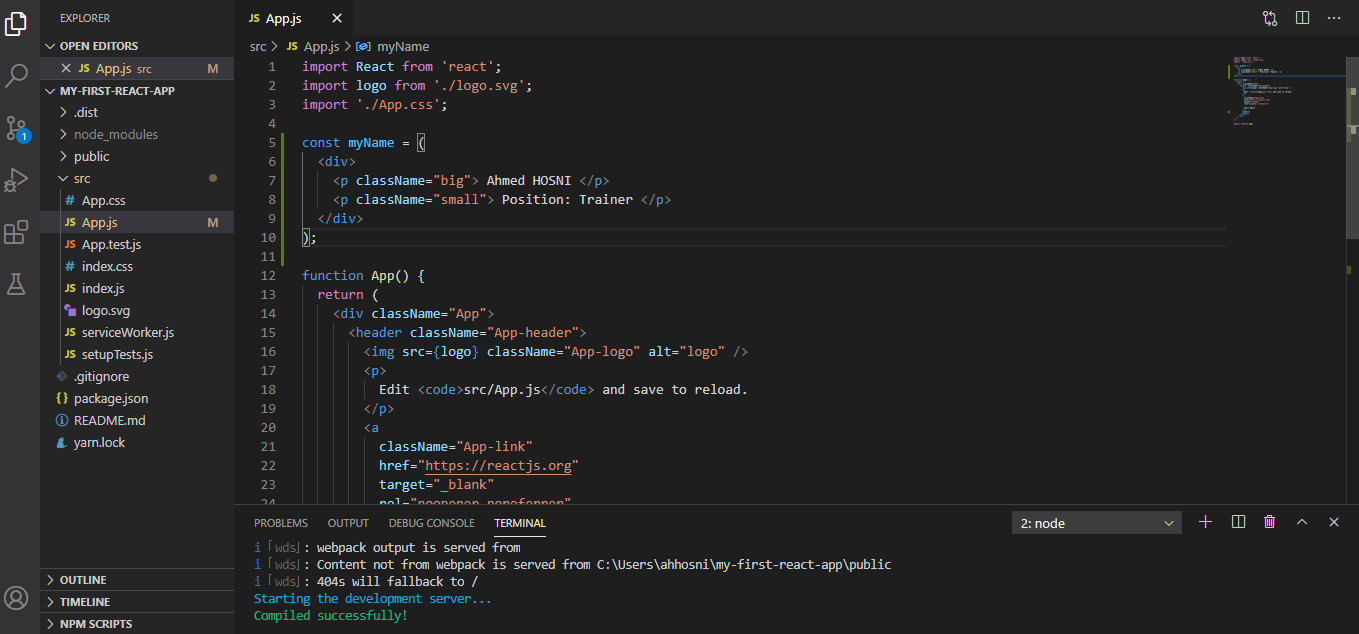
<span class=”small”> hello world </span>

Well, with JSX you cannot use the word class, instead we use className:

<span className=”small”> hello world </span>

JSX is translated to JavaScript, and in JavaScript class is a reserved word, hence we cannot use the word class in our JSX. When JSX is rendered, className will be translated to class.

Let’s update our **myName** variable:



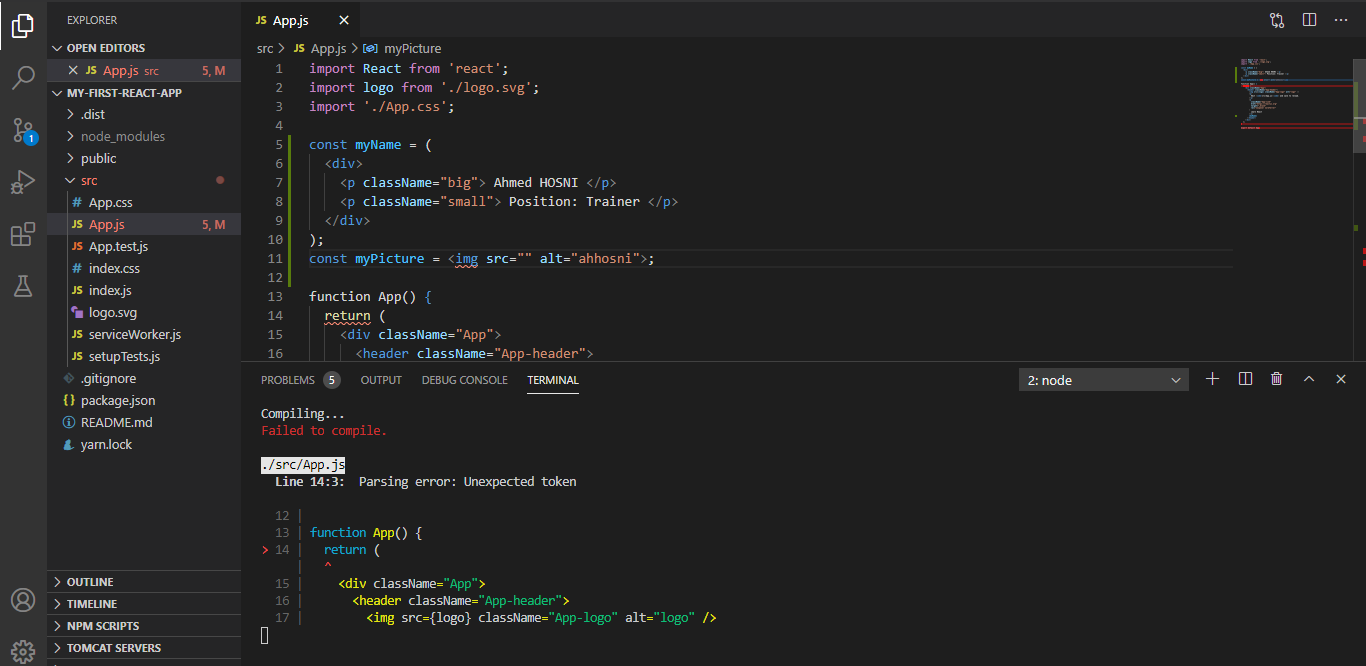
Save and see the magic! We have not defined CSS “small” and “big”!

**1.4.2 Self-Closing Tags**

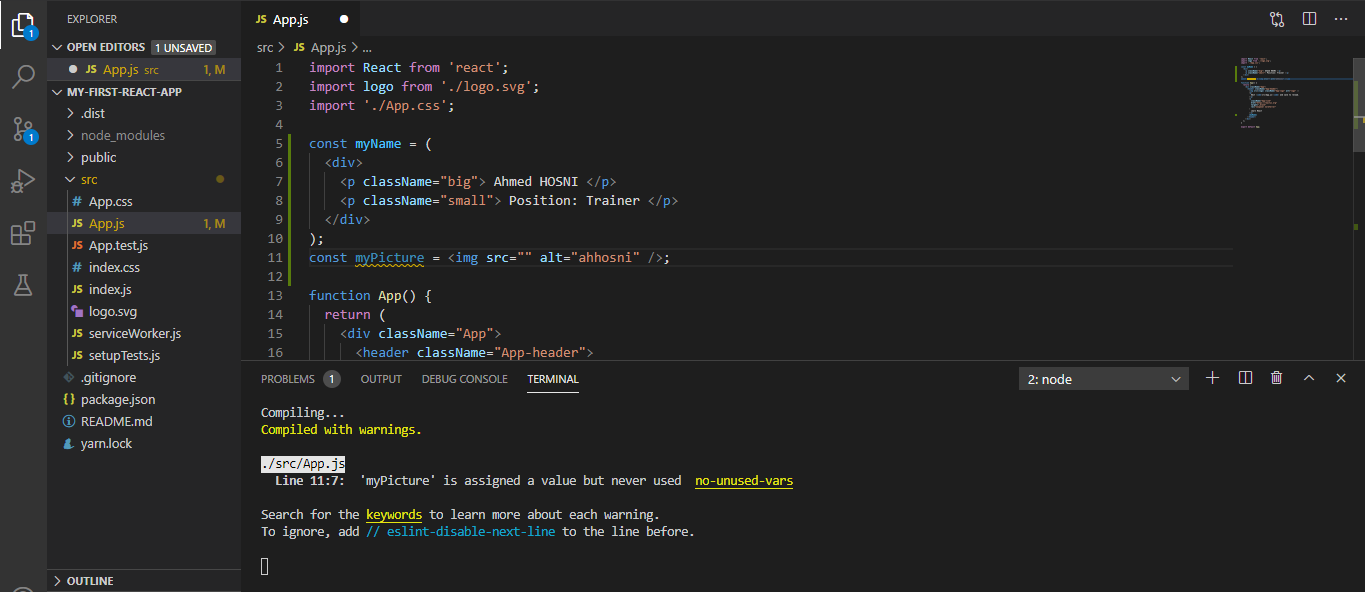
In HTML there is always an opening and a closing tag, **<div> </div>**. Some HTML elements such as **<br>** and **<input>** use only a single tag. This is called a self-closing tag. In HTML it’s optional to include the forward-slash right before the final angle-bracket, but in JSX it is mandatory to write the **forward-slash** else you raise an error.

***Let’s get our hands dirty:***

Create another constant called **myPicture** and assign an image element without closing the element with the forward-slash.

****

Now, add the forward-slash and note the difference:

****

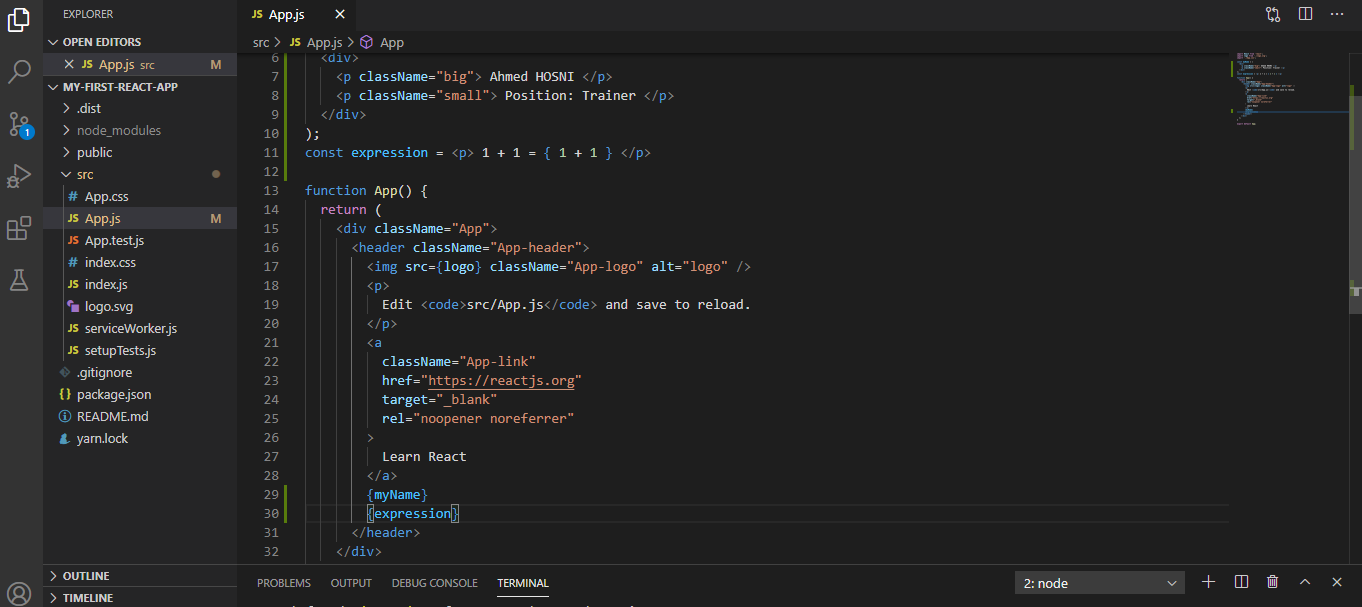
Compiles successfully, don’t worry about the warning.

**1.4.3. JavaScript In Your JSX**

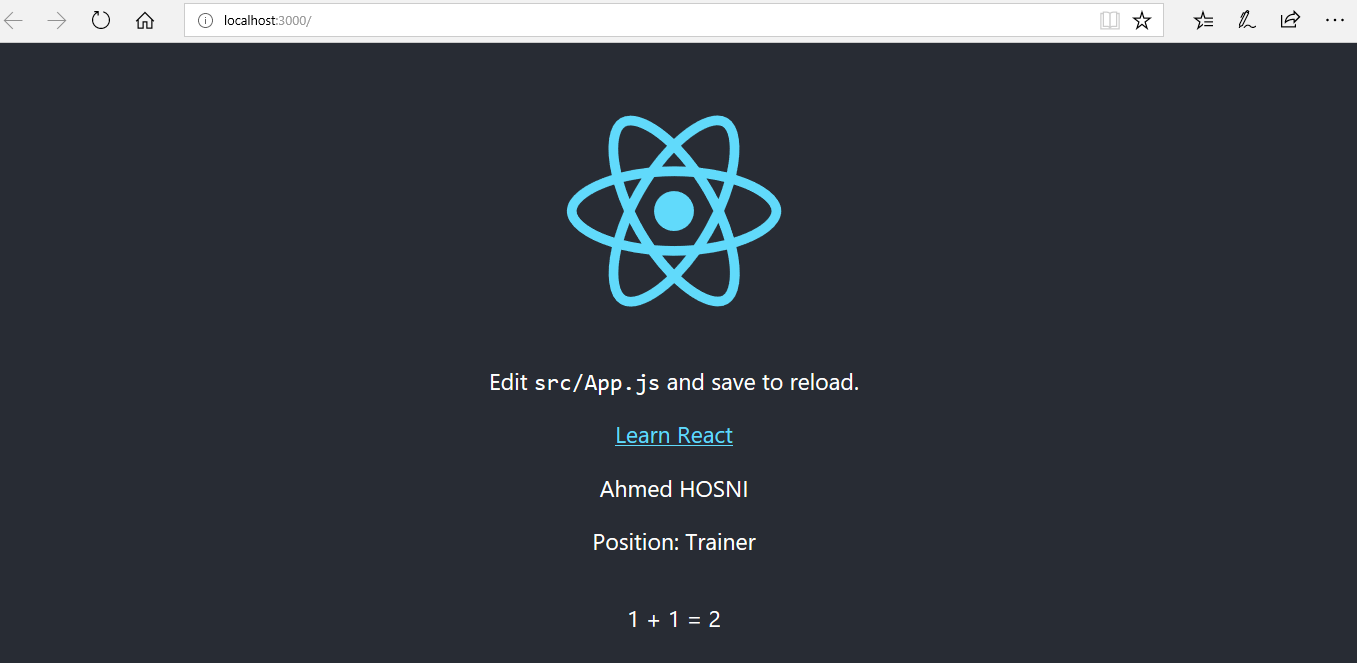
We can write javascript expressions inside our JSX expression, we’ve already seen this earlier when we used the curly braces and variable name, {myName}. Let’s try something simple, a mathematical expression:

const expression = <p> 1 + 1 = { 1 + 1 } </p>

Then, we can add this expression within our App():

****

Let’s take a careful look at the result on our browser:

****

Using the curly braces lets us add javascript expresssions to our JSX.

**1.4.4. Variable Attributes in JSX**

Attributes are used to set HTML behaviors, we can do this in JSX. Let’s do something fun, we’ll create a variable that holds the URL to our LinkedIn profile picture. Then pass this variable into a JSX element, ready?

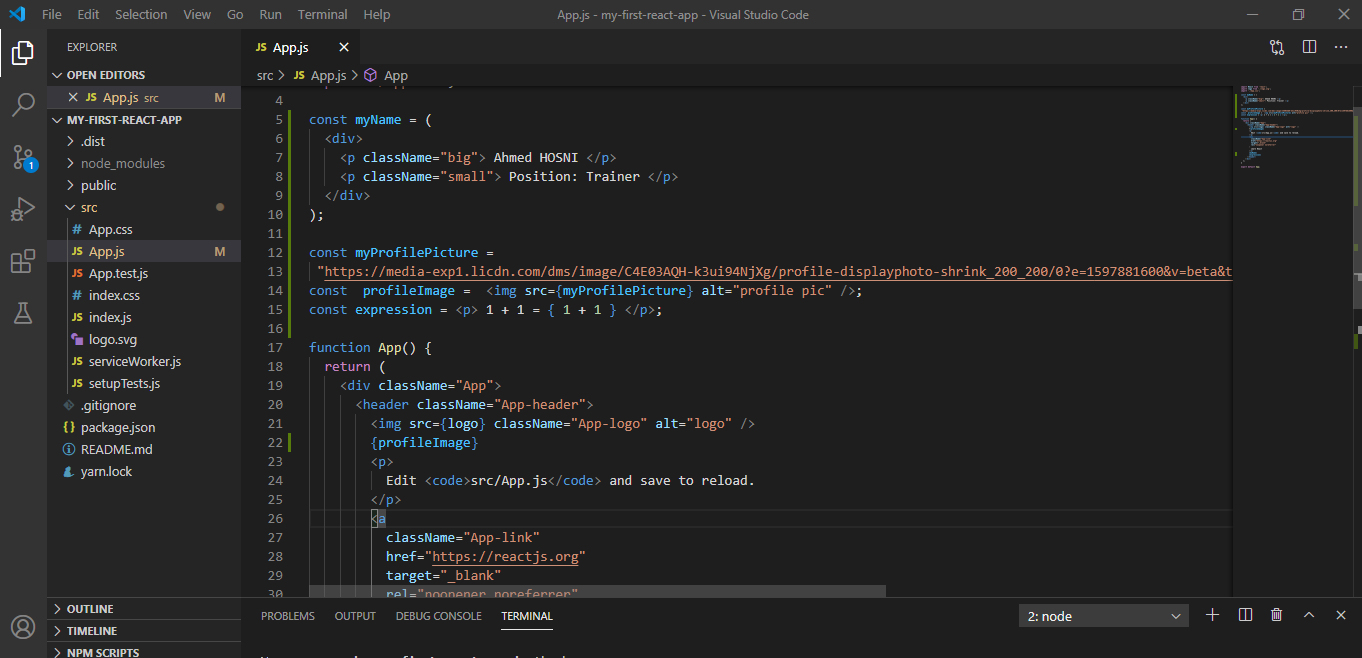
Go to your LinkedIn page, right click and copy your profile picture image address.

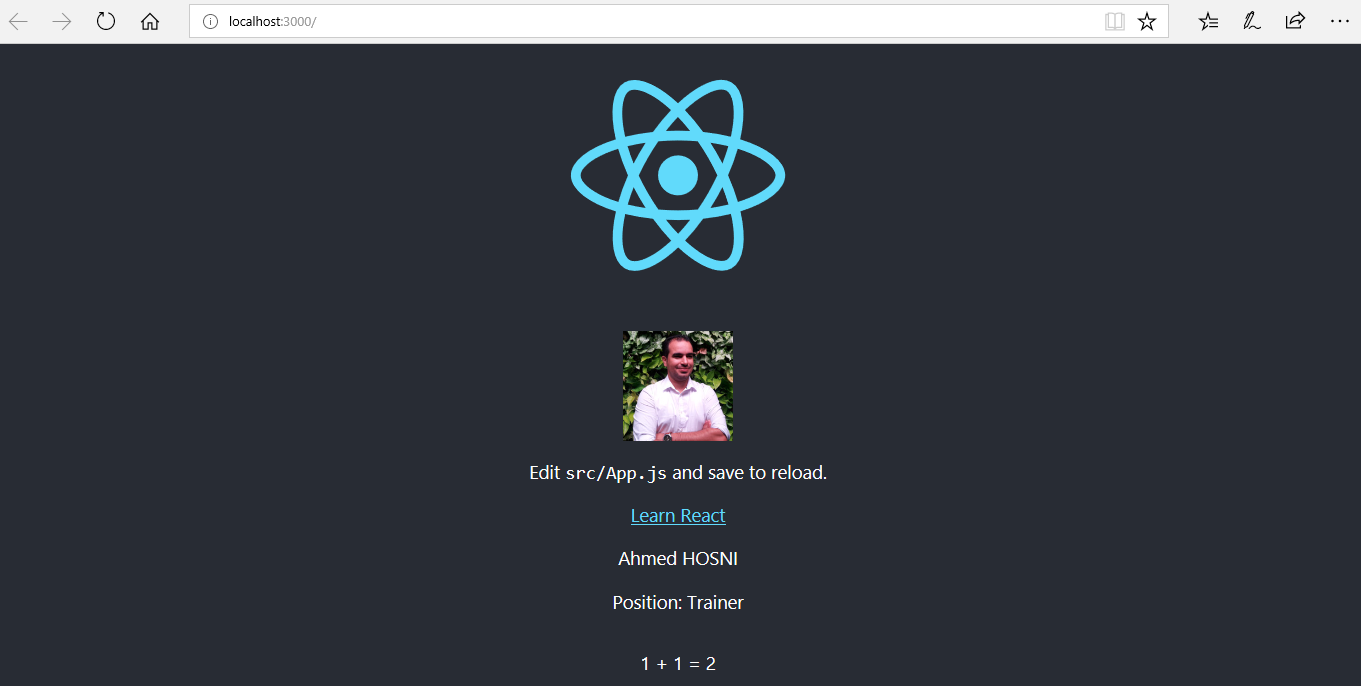
Now let’s update our **App.js** code with two variables: **myProfilePicture** and **profileImage**. myProfilePicture holds the URL to your LinkedIn profile picture, and profileImage is a JSX image element that references myProfilePicture in it’s src attribute.

const **​myProfilePicture​** = "https://media-exp1.licdn.com/dms/image/C4E03AQH-k3ui94NjXg/profile-displayphoto-shrink\_200\_200/0?e=1597881600&v=beta&t=oqWjiAecfjn-\_-PvN1TsgHXR6izpjTux4aKvIOnGp1w";

const **​profileImage** = <img src={myProfilePicture} alt="profile pic" />;

Then, add {profileImage} just below the default logo, see code below:

****

****

**1.4.5. Event Listeners in JSX**

JSX elements can contain listeners, just like in HTML. You can create an event listener the same way you do in HTML, with special attributes. E.g.:

<img onClick={callFunction} />

Event listeners start with a small case followed by an upper case for every word contained in the event listener. E.g. **onClick**, **onMouseOver**, etc, you can find more [​here​](https://reactjs.org/docs/events.html#supported-events). The value passed to the event listener’s attribute should be a function, e.g:

function callFunction(){

alert(“Hello World!”);

}

**Let’s get our hands dirty:**

Remember our LinkedIn profile picture? onClick, we’ll transform it into our LinkedIn cover photo! Go to LinkedIn, ***right click on your cover photo*** and ***copy the image address URL***.

Now, update the JSX profileImage to contain an event listener **onClick**, the value is a function called **changePicture**. We’ll create this function next:

const profileImage = (

<img onClick={changePicture} src={myProfilePicture} alt="profile pic" />

);

Now, let’s define the **changePicture** function. This function will basically change the src attribute of your profileImage JSX.

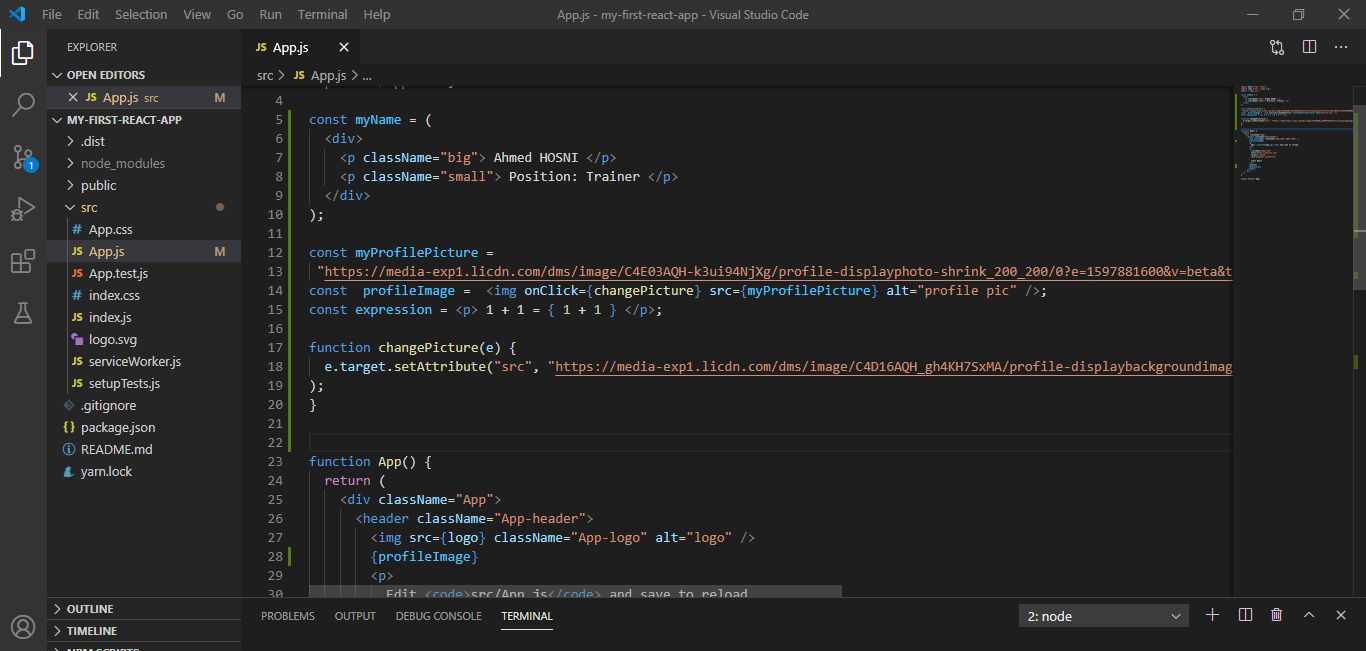
function changePicture(e) {

e.target.setAttribute("src", "https://media-exp1.licdn.com/dms/image/C4D16AQH\_gh4KH7SxMA/profile-displaybackgroundimage-shrink\_200\_800/0?e=1597881600&v=beta&t=z9Y2lkmj-6-c9Czk-KBdUkLIJiUe1uu8UcExSXK4LWs"

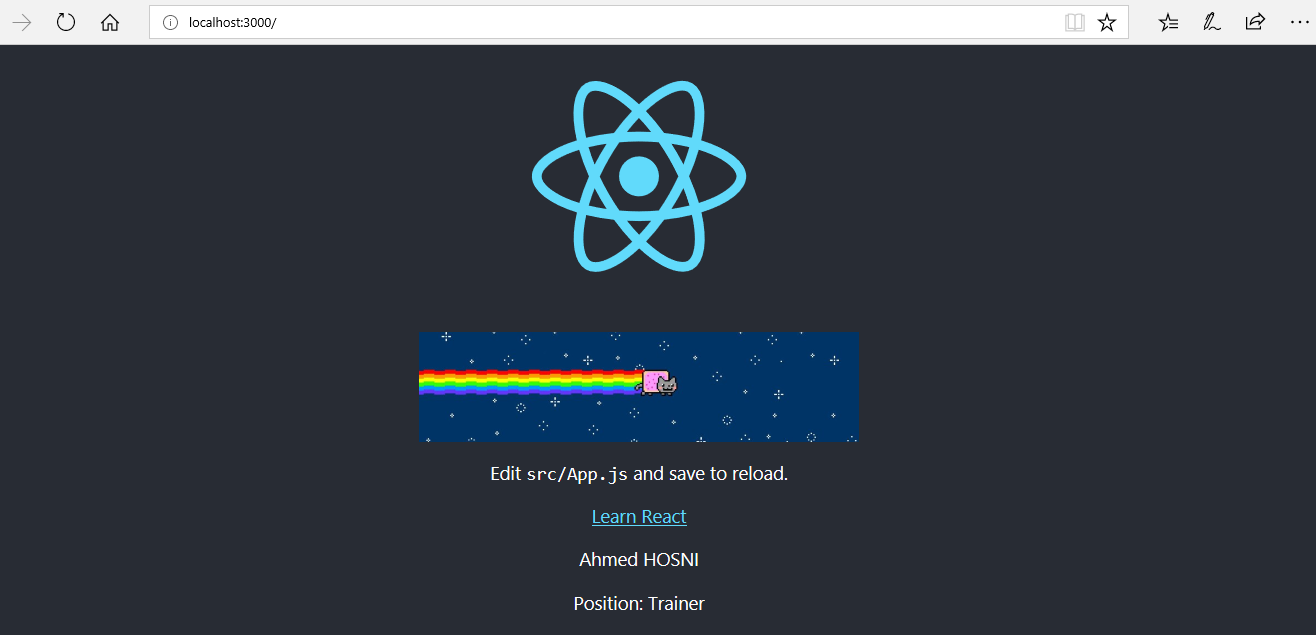
);

}

Let’s see the complete code in action!

****

Click on your profile image and see it transform into your cover photo.

****

**1.4.6. JSX Conditionals: if**

You have learned how to use curly braces to inject JavaScript expressions into a JSX expression. Here is an important rule, *you cannot inject an if statement into a JSX expression*.

So the question is, how do we inject a conditional statement into JSX, the first option is to write an if statement and not inject it into JSX. Here is an example:

let age = 17;

let drinkingAge = 21;

if (age >= drinkingAge) {

​const message = (

<h1>

You are allowed to get drunk tonight!

</h1>

);

}

The “if statement” isn’t injected between the JSX tags, it is on the outside. This is commonly used to express conditionals in JSX>

**1.4.7. JSX Conditionals: The Ternary Operator**

There’s an easier way to write conditional statements in JSX, the ternary operator. Recall how it works? x ? y : z, where x, y and z are JavaScript expressions. X is checked for it’s truthiness or falsiness, if x is true then the operation returns y, else if x is false the ternary operator returns z.

Let’s use the ternary operator to check drinking age like we did in our previous example:

**let age = 18;**

**const headline = <h1>{age >= 17 ? "Yes Drinks" : "No Drinks"}</h1>;**

Let’s do something fun with our code, remember our profile picture?

We will create a function that flips a coin, if the head displays your LinkedIn profile picture and if tail displays your LinkedIn cover photo.

**Step 1:** Add function called **tossCoin()**:

function coinToss() {

return Math.random() < 0.5 ? "heads" : "tail";

}

**Step 2:** Define your profile picture and cover photo in a **dictionary**:

const pics = {

profilePicture: 'your profile URL here',

coverPhoto: 'your cover photo URL here'

}

**Step 3:** Finally, update profileImage mini component (that’s what I call these variables) to show **profilePicture** when heads otherwise **coverPhoto**:

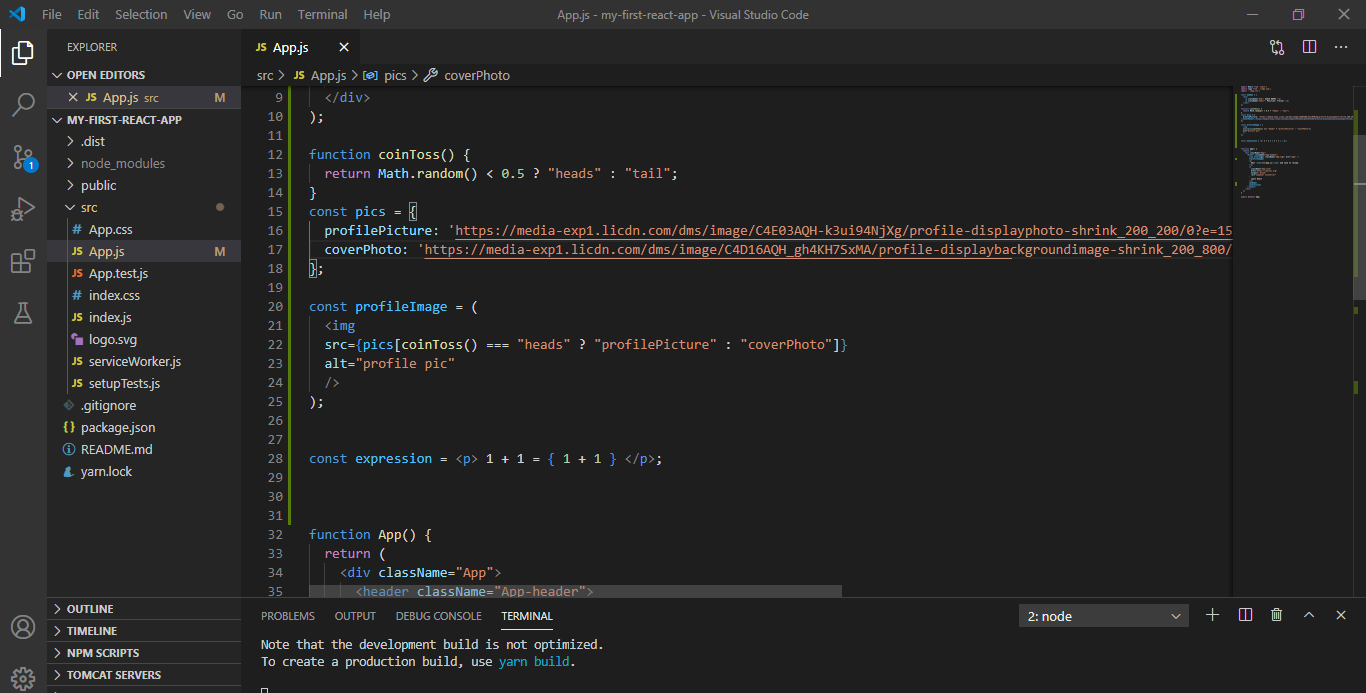
const profileImage = (

<img

src={pics[coinToss() === "heads" ? "profilePicture" : "coverPhoto"]} alt="profile pic"

/>

);

****

Ok, refresh your browser as many times as you like and see the magic!

**1.4.8. JSX Conditionals: &&**

Let’s look at another way of using conditionals, **the && operator**. In our last lesson we rendered our linkedIn profile picture and cover photo randomly, based on chance (coin toss). The && operator works best in conditions that will trigger an action sometimes, but other times do nothing at all.

Let’s redesign our **App.js**, we’ll display a simple profile link tree.

**Step 1: Add a profile picture mini component:**

const profilePicture = (

<img

src="your profile URL here"

alt="profile"

/>

);

**Step 2: Define a list**

const links = (

<ul>

<li>my book</li>

<li>book appointment</li>

<li>my video</li>

</ul>

);

**Step 3: Replace the App() function with:**

function App() {

return (

<div className="App">

<header className="App-header">

{profilePicture}

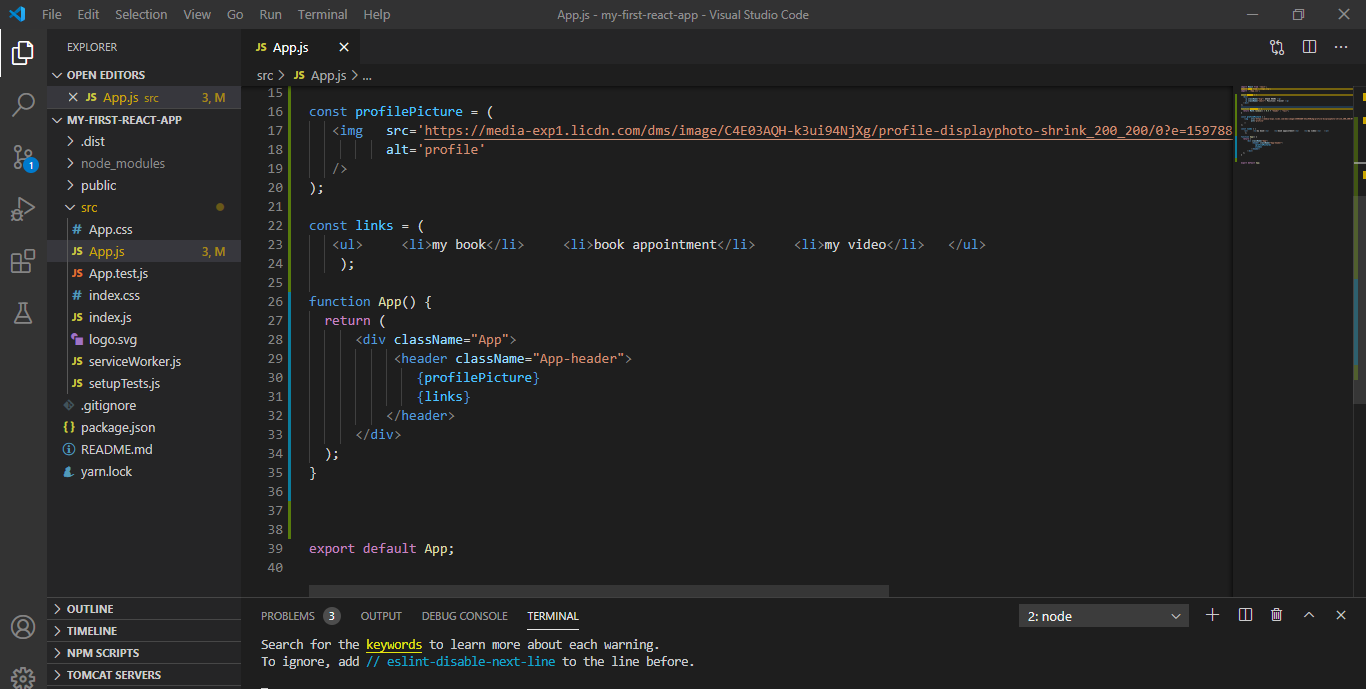
{links}

</header>

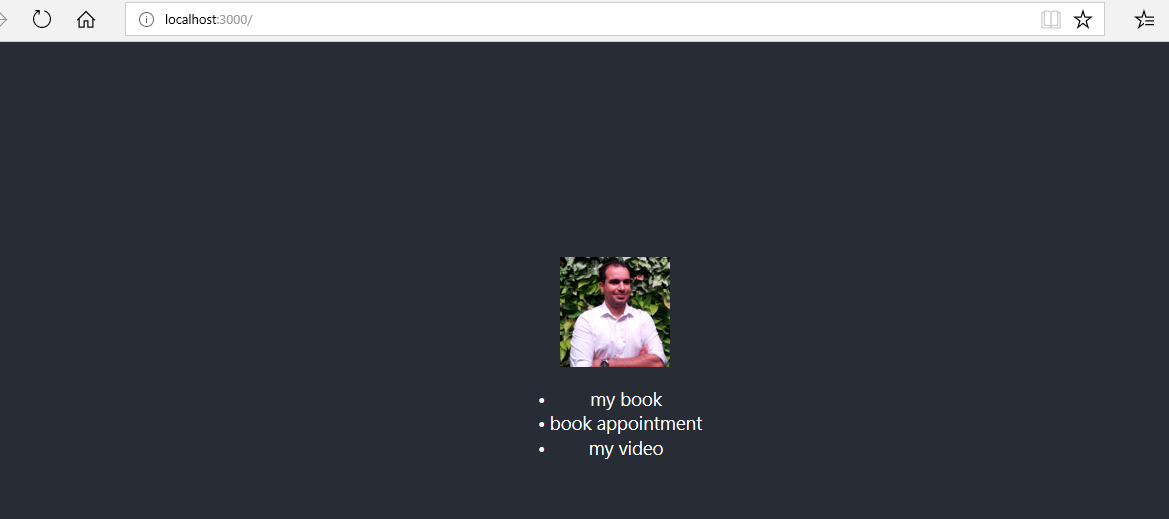
</div>

);

}

****

Refresh your browser and see the magic, please ignore the styling, we’ve not added any styling.

****

**1.4.9. .map in JSX**

.map() method is used in creating a list of JSX elements:

const links = ["my book", "book appointment", "my video"];

const linkItems = links.map(link => <li> {link} </li>);

We start by defining our list in an array, then call .map() method on this array to return a new array of <li>s**.**

function App() {

return (

<div className="App">

{linkItems}

</div>

);

}

Now in the above example we call {linkItems}, this will evaluate to an array of <li> items.

* **keys:**

In JSX sometimes your list needs to include keys:

<ul>

<li key="li-01">Example1</li>

<li key="li-02">Example2</li>

<li key="li-03">Example3</li>

</ul>

A key is a JSX attribute, it’s value has to be unique, just like the id attribute. React uses keys to internally keep track of lists so as to avoid scrambling of your list-items into a wrong order.

You’ll need keys if:

1. List items have memory from one render to the next.

2. The order of your list might be shuffled

Here is what an updated version of our previous mapping would look like with a key.

const links = ["my book", "book appointment", "my video"];

const linkItems = links.map((link, i) => (

<li key={"li-" + i}>

<a href="#">{link}</a>

</li>

));

Now let’s update our App.js to use the .map() method.

**Step 1: Add a profile picture mini component:**

const profilePicture = (

<img

src="your profile URL here"

alt="profile"

/>

);

**Step 2: Define a list**

const links = ["my book", "book appointment", "my video"];

const linkItems = links.map((link, i) => (

<li key={"li-" + i}>

<a href="#">{link}</a>

</li>

));

**Step 3: Replace the App() function with:**

function App() {

return (

<div className="App">

<header className="App-header">

{profilePicture}

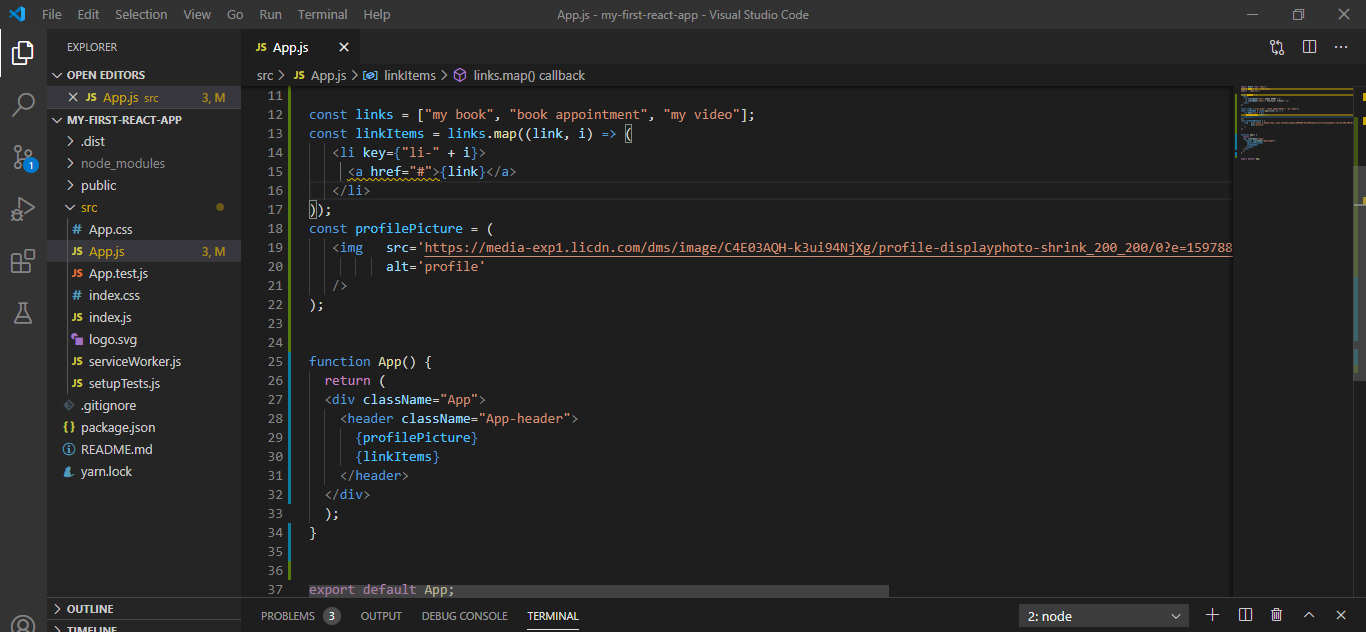
{linkItems}

</header>

</div>

);

}

****

**1.4.10. React.createElement**

Well, let’s see how JSX codes are made. Yes, you can write React code without JSX. I mean, most React developers use JSX, I just want you to understand how it works.

This simple JSX expression:

const myName = <p className="big">Ahmed HOSNI</p>

Can be re-written without JSX as:

const myName = React.createElement(

"p",

null,

"Ahmed HOSNI"

);

When a JSX element is compiled, it is converted into the above method (React.createElement()).

**Exercice 2: Your First React Component**

**1. The Component**

React applications contain lots of components, a component is a small, reusable code responsible for rendering some HTML. Let’s see what a component looks like:

import React from 'react';

import ReactDOM from 'react-dom';

class MyComponent extends React.Component {

render() {

return <h1>Hello world</h1>;

}

};

ReactDOM.render(

<MyComponent />,

document.getElementById('app')

);

Open your App.js and replace all the code you see with the below:

import React from "react";

class MyComponent extends React.Component {

render() {

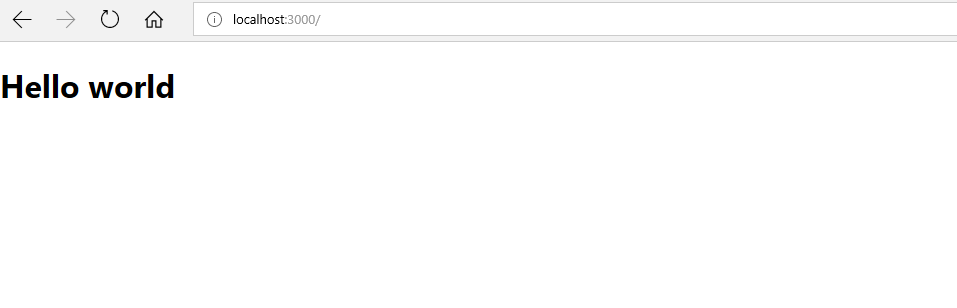
return <h1>Hello world</h1>;

}

}

export default MyComponent;

Refresh your browser!



Notice the difference? We’ve separated ReactDOM from this file, rather than have the render in our App.js:

ReactDOM.render( <MyComponent />, document.getElementById('app') );

We export MyComponent (​export default MyComponent;) to be used in index.js where we call **ReactDOM.render** on our App.js

ReactDOM.render(<App />, document.getElementById('root'));

Now we have a basic understanding of our file structure, let’s unpack the code above.

**1.1. Import React**

The first line of code creates a new variable called React whose value is an imported JavaScript object.

import React from 'react';

This object contains methods for you to be able to use React, this is basically the React library.

Remember *React.createElement()?* This is a method from the React library. You must import the React library in order to use any JSX.

**1.2. Import ReactDOM**

Take a look at index.js,

import ReactDOM from 'react-dom';

Looks similar to the previous import statement, both objects contain React-related methods, however, the methods imported from ‘react-dom’ are meant for DOM interaction only.

The DOM is used in React applications, but is not a part of React. You’re already familiar with the render method from *ReactDOM.render()*.

**1.3. Create a Component Class**

A React component is a reusable code responsible for rendering some HTML. Every component is created from a *component class*.

A component class is like a blueprint used to create components, with a component class, you can reproduce as many components as you want. React.Component is the base class used in creating React components.

React.Component is used to create your own component class by subclassing React.Component.

class MyComponent extends React.Component

Component class variable names must begin with capital letters, this is standard programming convention to use Capital CamelCasing when naming classes.

class MyComponent extends React.Component {

render() {

return <h1>Hello world</h1>;

}

}

Let’s get into the **render()** method inside our **MyComponent** React.Component class. The render() method is used to create a set of instructions that explain what your component class will render.

render() {

return <h1>Hello world</h1>;

}

The render method contains a return statement, that returns a JSX expression for rendering in your HTML.

**1.4. Create a Component Instance**

Now we’ve created a component class MyComponent, we have a working component class ready to be exported as a React component. It takes just a simple line to make a component instance:

<MyComponent />

If you are calling the *ReactDOM.render* method in the same file as your component, then:

ReactDOM.render(

<MyComponent />,

document.getElementById('app')

);

Otherwise, like we did, we’ll export our component to be used in another file using:

export default MyComponent;

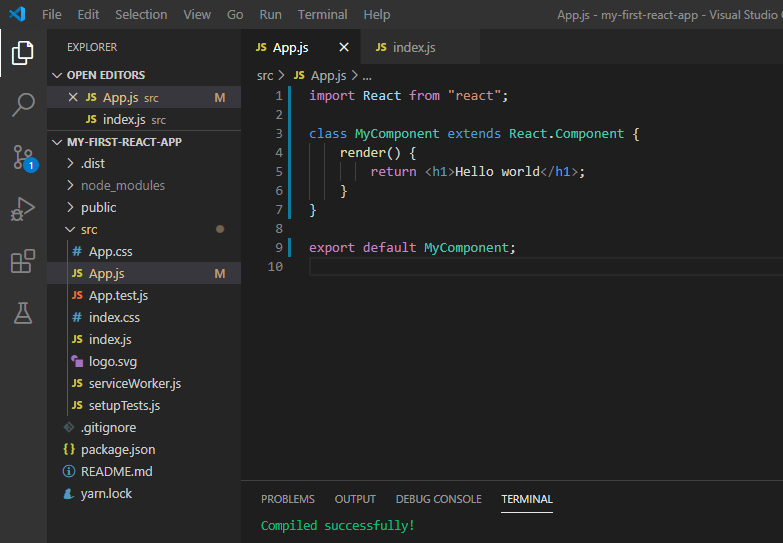
Then in the file (index.js) we intend to use this component, we import App.js

import App from './App';

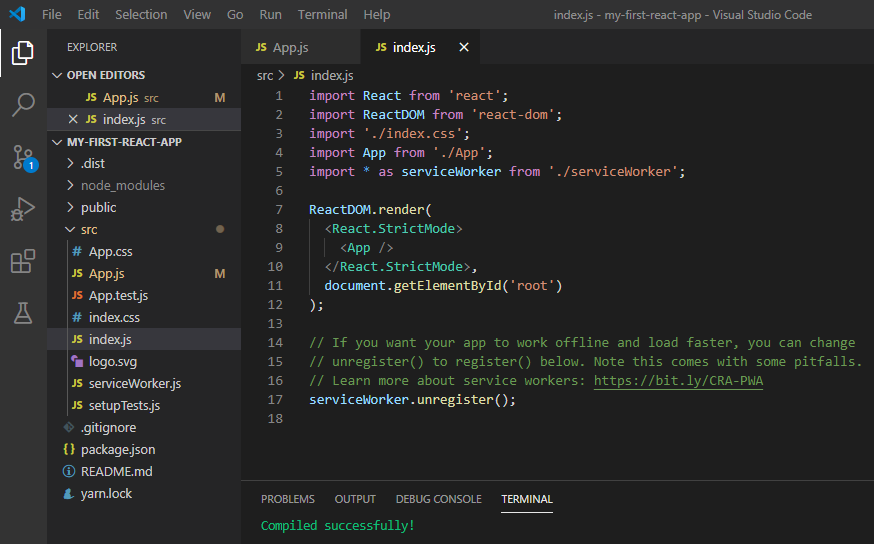
Then ReactDOM.render() your App

ReactDOM.render(<App />, document.getElementById('root'));

So our App.js would look like this:



While our index.js looks like this:



**2. Components and Advanced JSX**

**2.1. Use Multiline JSX in a Component**

We will learn how JSX and React components work together, checkout this HTML:

<blockquote>

<p>

My LinkedIn Profile

</p>

<cite>

<a href="https://www.linkedin.com/in/ahmed-hosni-18a07526/">

Ahmed Hosni

</a>

</cite>

</blockquote>

Let’s make a React component to render this HTML:

class LinkedInProfile extends React.Component {

render() {

return (

<blockquote>

<p> My LinkedIn Profile </p>

<cite>

<a href="https://www.linkedin.com/in/ahmed-hosni-18a07526/">

Ahmed Hosni

</a>

</cite>

</blockquote>

);

}

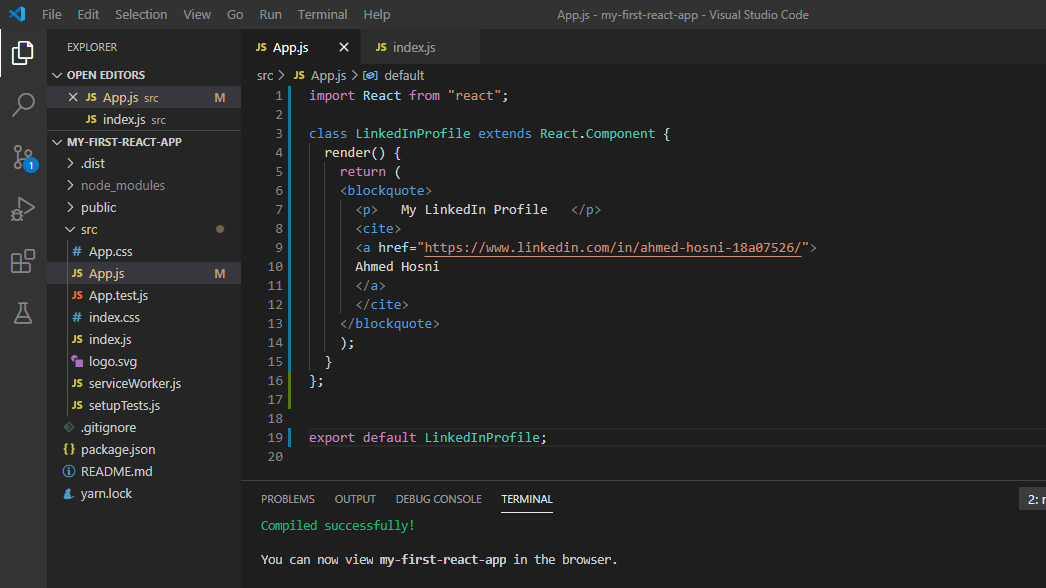
};

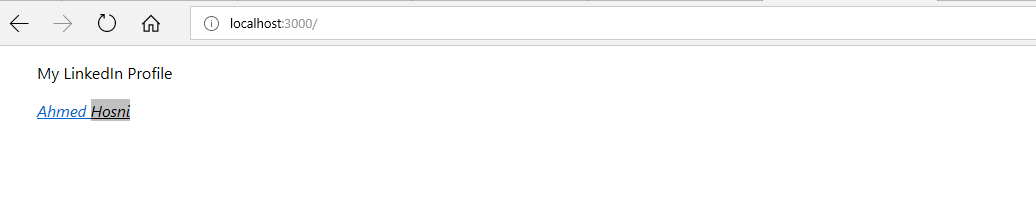
The important thing to notice in LinkedInProfile is the use of parentheses in the return statement. We’ve been rendering single line JSX without parentheses:

return <h1> Hello World! </h1>

In multi-line JSX expressions, the return JSX should always be wrapped in parentheses.

Now, in App.js, replace *MyComponent* component class with the above *LinkedInProfile* component, and export the same.





**2.2. Use a Variable Attribute in a Component**

Let’s checkout this JavaScript object named myLinkedInBasicInfo:

const myLinkedInProfileBasicInfo = {

url: "your linkedIn url here",

profile\_picture\_url: "your profile picture url here",

cover\_photo\_url: "your cover photo url here",

full\_name: "your full name"

};

We’ll Render our LinkedInProfile component using properties from the above JavaScript object.

class LinkedInProfile extends React.Component {

render() {

return (

<blockquote>

<p>My LinkedIn Profile</p>

<div>

<img

src={myLinkedInProfileBasicInfo.profile\_picture\_url}

alt={myLinkedInProfileBasicInfo.full\_name}

/>

</div>

<cite>

<a href={myLinkedInProfileBasicInfo.url}>

{myLinkedInProfileBasicInfo.full\_name}

</a>

</cite>

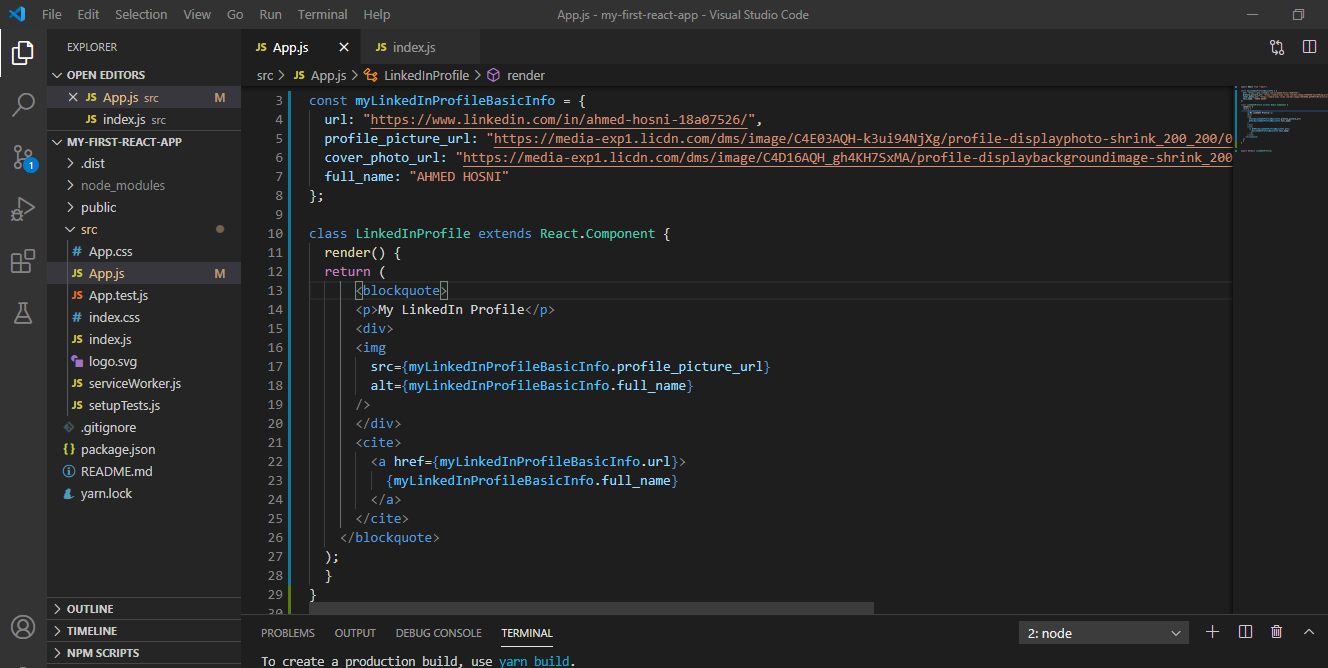
</blockquote>

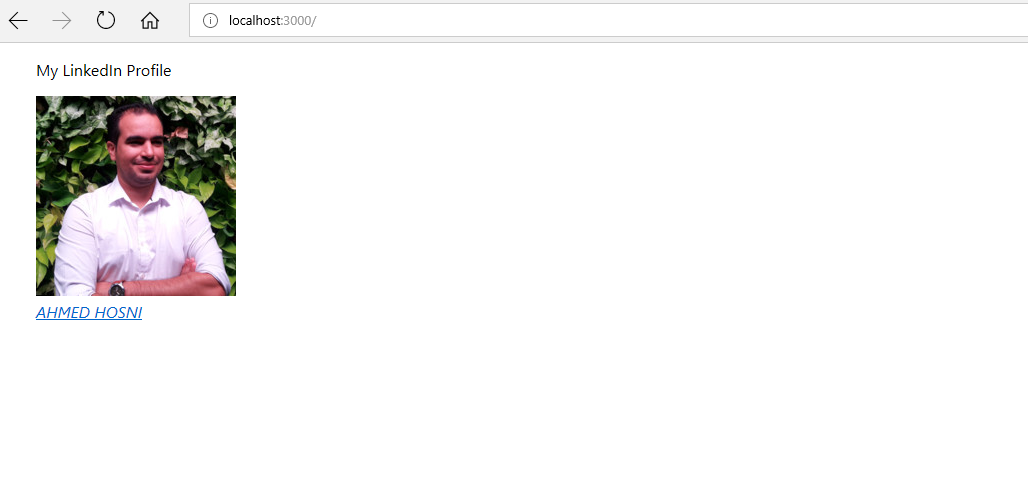
);

}

}

Have you noticed all the curly-brace JavaScript injections used inside of the component class? You will most often inject JavaScript into JSX within your render function.





**2.3. Using Logic in a Render Function**

Apart from a return statement in a render() function, you can have logic also. Logic like calculations, etc, right before you render your component.

class Age extends React.Component {

render() {

// First, some logic that must happen before rendering:

// Next, a return statement using that logic:

}

}

Let’s add an age property to myLinkedInProfileBasicInfo

const myLinkedInProfileBasicInfo = {

age: 30,

..

}​;

Now in the *LinkedInProfile* component, just before the render is called, add this logic:

let age\_verified = "Age not approved";

if (myLinkedInProfileBasicInfo.age < 18) {

age\_verified = "Age not approved";

} else {

age\_verified = "Age approved";

}

Finally, add the age\_verified element to your JSX, within the cite tag:

<cite>

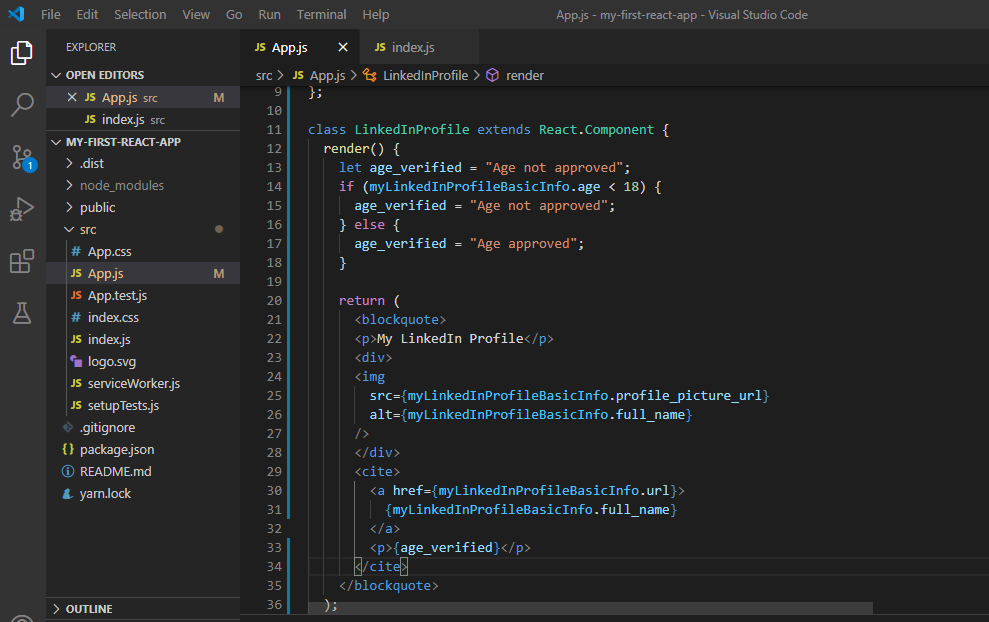
<a href={myLinkedInProfileBasicInfo.url}>

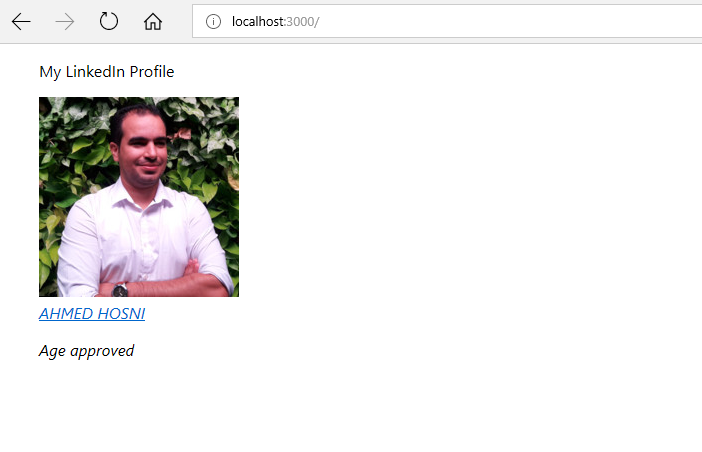
{myLinkedInProfileBasicInfo.full\_name}

</a>

<p>{age\_verified}</p>

</cite>





Don’t make a mistake of declaring your logic outside the render function, i.e. inside your component class. It will cause a syntax error, as it shouldn’t be a part of the class declaration, but only occur in your render().

**2.4. Use this in a Component**

class LinkedInProfile extends React.Component {

get fullName() {

return 'Ahmed HOSNI';

}

render() {

return <h1>I like {this.fullName}.</h1>;

}

}

Well, “this” refers to the object in which this’s enclosing method is called. It’s an instance of the LinkedInProfile. LinkedInProfile has two methods, **.fullName()** and **.render()**, a call to *this.fullName* will evaluate to a call of LinkedInProfile’s *.fullName* method, evaluating to the string “Ahmed HOSNI”.

Wondering why we don’t have it as this.food(). Well, shouldn’t we have a parentheses? You don’t need parentheses because .food is a getter method.

Let’s play:

Update **LinkedInProfile** component to u*se a getter method linkedInProfile*, whose value is equal to the **myLinkedInProfileBasicInfo** object.

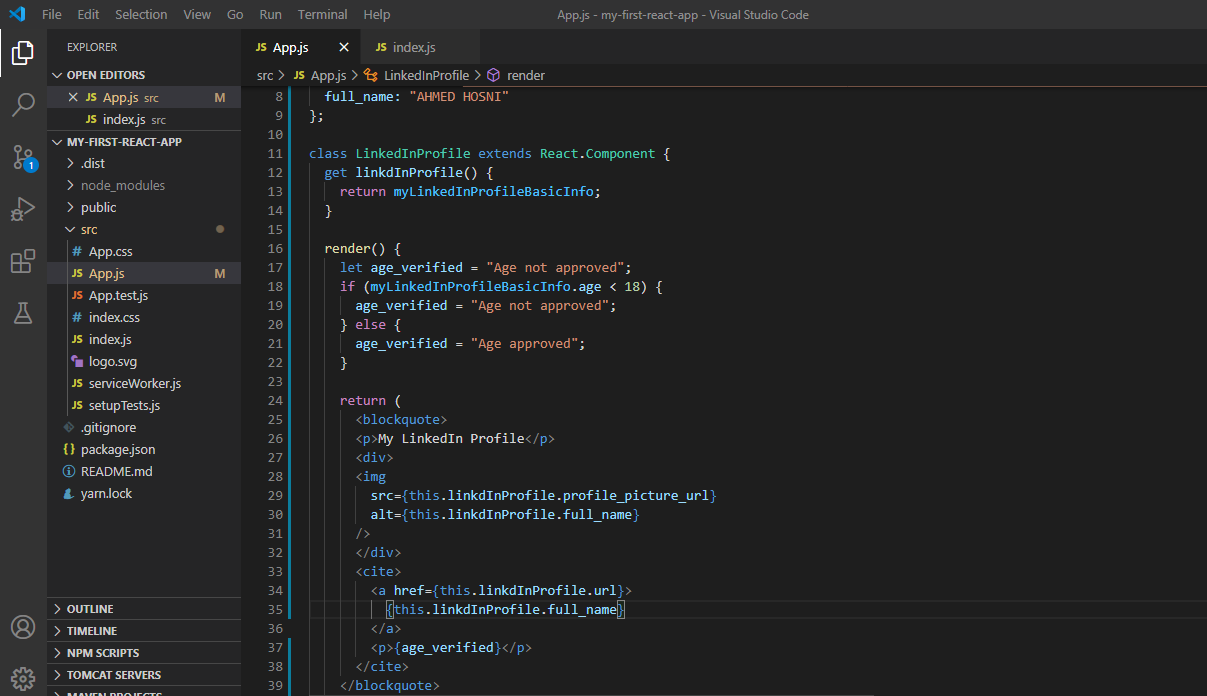
get linkdInProfile() {

return myLinkedInProfileBasicInfo;

}

Now, replace all the occurence of {myLinkedInProfileBasicInfo.property} with {this.linkdInProfile.property}

Your component should look like this:



**2.5. Event Listener in a Component**

I talked about event listeners in previous chapter, here’s an example of how event listeners are used in a render function:

render() {

return (

<div onClick={myFunc}> </div>

);

}

An event handler is a function that is called in response to an event, {myFunc} is the event handler in the above example. In React, event handlers are defined as methods on a component class, like this:

class MyClass extends React.Component {

myFunc() {

alert('Hello World');

}

render() {

return (

<div onClick={this.myFunc}>

</div>

);

}

}

Let’s add a button element to our linkedIn profile, on click we display a welcome message. We’ll add this welcome message to our myLinkedInProfileBasicInfo JavaScript object.

const myLinkedInProfileBasicInfo = {

welcome\_message: "Welcome to my linkedin profile component",

..

..

};

Now, let’s create a function to handle the event.

welcomeMessage() {

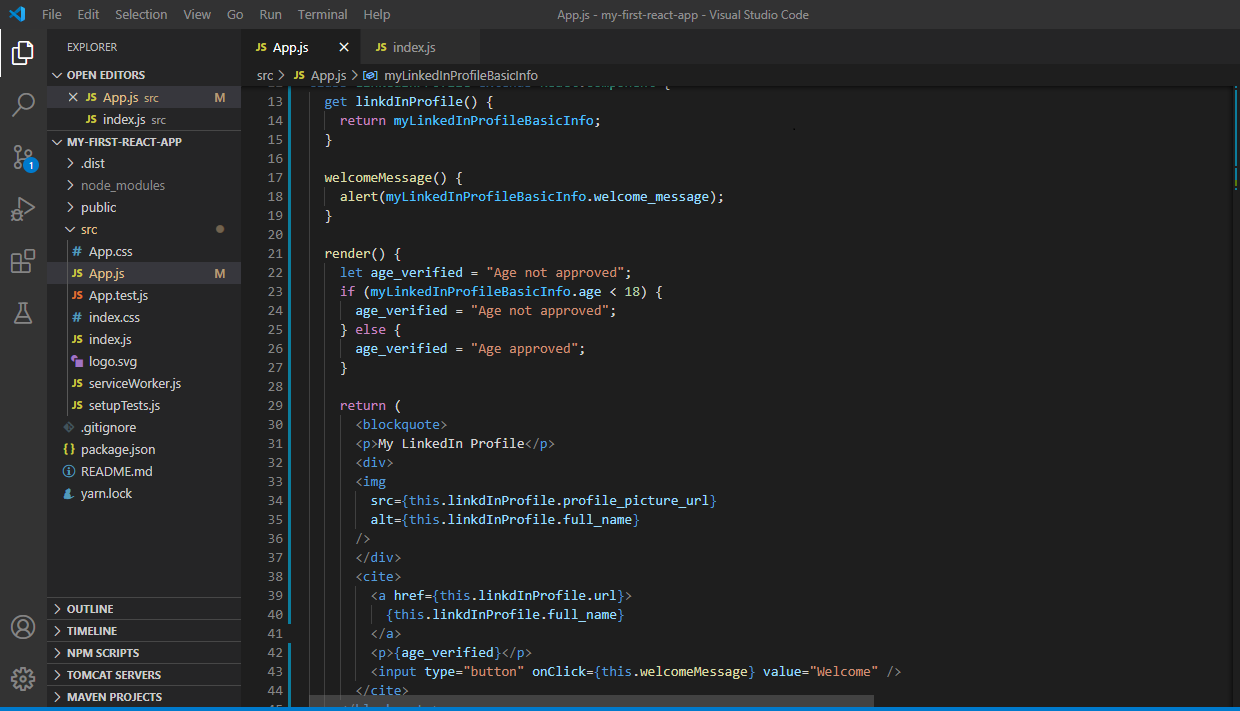
alert(myLinkedInProfileBasicInfo.welcome\_message);

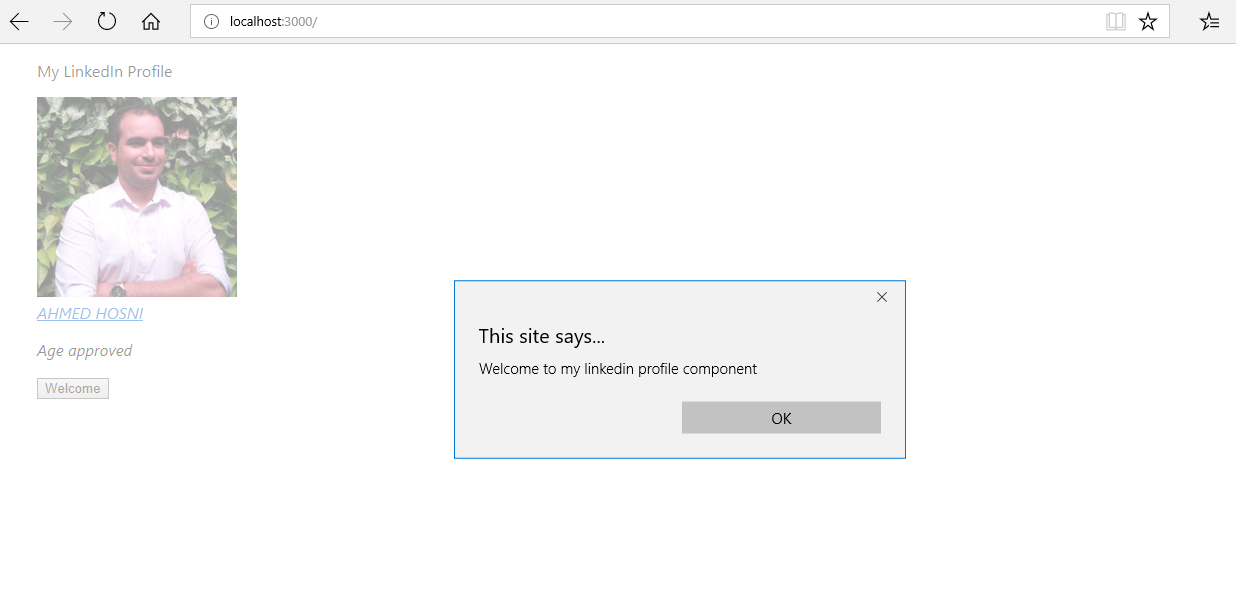
}

Finally, let’s add a button to our JSX with an **onClick** event that triggers the event handler, welcomeMessage()

<input type="button" onClick={this.welcomeMessage} value="Welcome" />

I added this button right below the {age.verified} element.





**3. Components Render Other Components**

**3.1 Components Interact**

React applications can contain as many components as you want, yes, hundreds of components. Each component might be small on its own, but when combined, wow, they create an amazing system. React is so cool in the way components interact with each other.

Remember how we render JSX?

class MyClass extends React.Component {

render() {

return (

<div>

<h1>Hello World</h1>

<p>Welcome to my React Component class</p>

</div>

);

}

}

Well, render can also return component instances, yes, you heard me.

class MessageSection extends React.Component {

render() {

return (

<div>

<p>Welcome to my Message</p>

<p>Here's what I have to say, nothing.</p>

</div>

);

}

}

class MyClass extends React.Component {

render() {

return (

<div>

<h1>Hello World</h1>

<p>Welcome to my React Component class</p>

​ <MessageSection />

</div>

);

}

}

In the example above, MyClass’s render method returns an instance of **MessageSection** component class.

**3.2. Apply a Component in a Render Function**

When a component renders another component, what happens is very similar to what happens when **ReactDOM.render()** renders a component.

We’ll create a react component for a profile card, which will be a part of our React application. We’ll learn how to import components and render components within another component, let’s go!

● Create a new file called ProfileCard.js

● Import React and create a component class called ProfileCard

ProfileCard Component:

export class ProfileCard extends React.Component {

   render() {

      return (

<div class="card">

<img

src="https://media-exp1.licdn.com/dms/image/C4E03AQH-k3ui94NjXg/profile-displayphoto-shrink\_200\_200/0?e=1597881600&v=beta&t=oqWjiAecfjn-\_-PvN1TsgHXR6izpjTux4aKvIOnGp1w"

alt="John"

/>

<h1>John Doe</h1>

<p class="title">CEO Founder, Example</p>

<p>Harvard University</p>

<p>

<button>Contact</button>

</p>

</div>

    );

  }

We need to make sure that ProfileCard is exportable because we’ll be using it in App.js

export default ProfileCard;

Now it’s time to add some styling, inside your src directory, create a new file called **style.css** add add style:

.card {

   box-shadow: 0 4px 8px 0 rgba(0, 0, 0, 0.2);

   max-width: 300px;

   margin: auto;

   text-align: center;

}

.title {

     color: grey;

     font-size: 18px;

}

button {

border: none;

outline: 0;

display: inline-block;

padding: 8px;

color: white;

background-color: #000;

text-align: center;

cursor: pointer;

width: 100%;

font-size: 18px;

}

a {

text-decoration: none;

font-size: 22px;

color: black;

}

button:hover, a:hover {

opacity: 0.7;

}

Now, make sure you’ve imported style.css in ProfileCard.js

import "./style.css";

Let’s get into our App.js file and import React and ProfileCard:

import React from "react";

import { ProfileCard } from "./ProfileCard";

Now, create your App React component and make it render ProfileCard:

class App extends React.Component {

render() {

return <ProfileCard />;

}

}

Okay, we’ve seen how to import a variable from another file, when you import a variable you need to make sure that variable is **exported**. Notice the error?

There are different ways to export variables, we’ll be using a style called named exports in this example. Place the keyword ​export before a *variable/class/function* you wish to export.

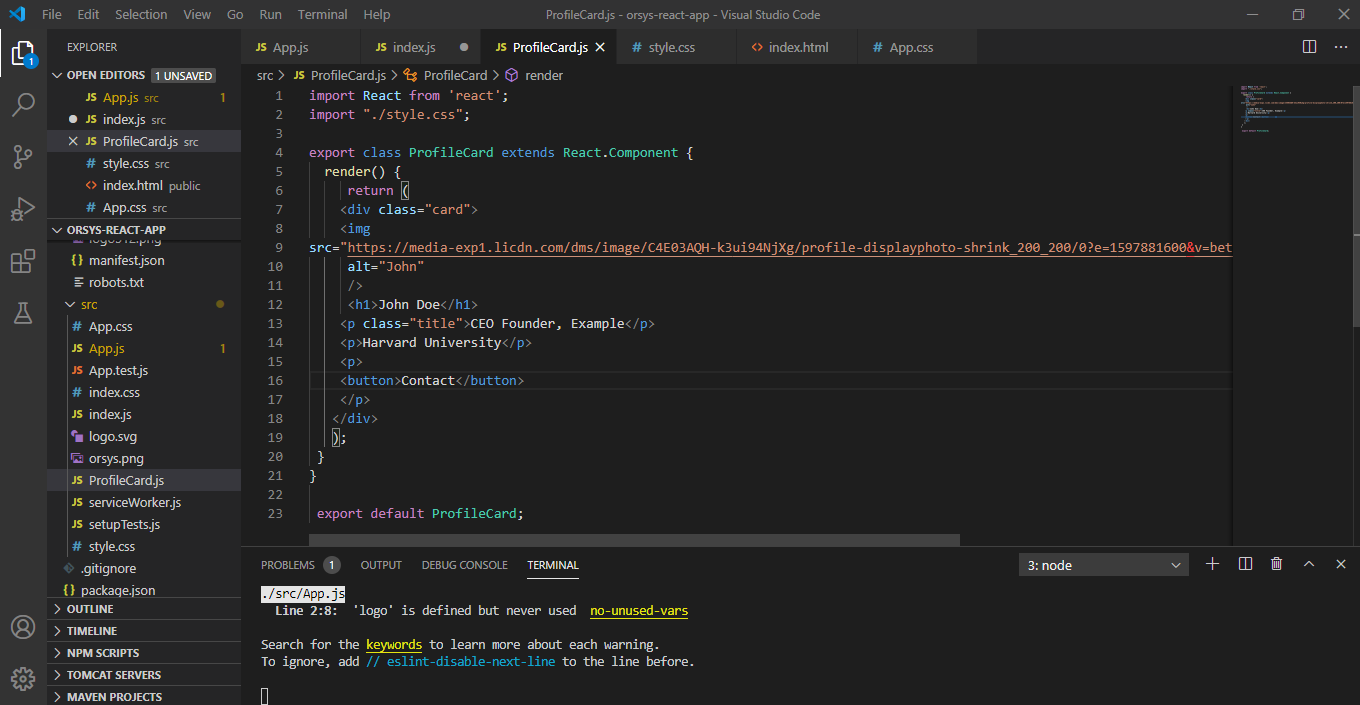
Now, add export keyword before the *ProfileCard* component class:

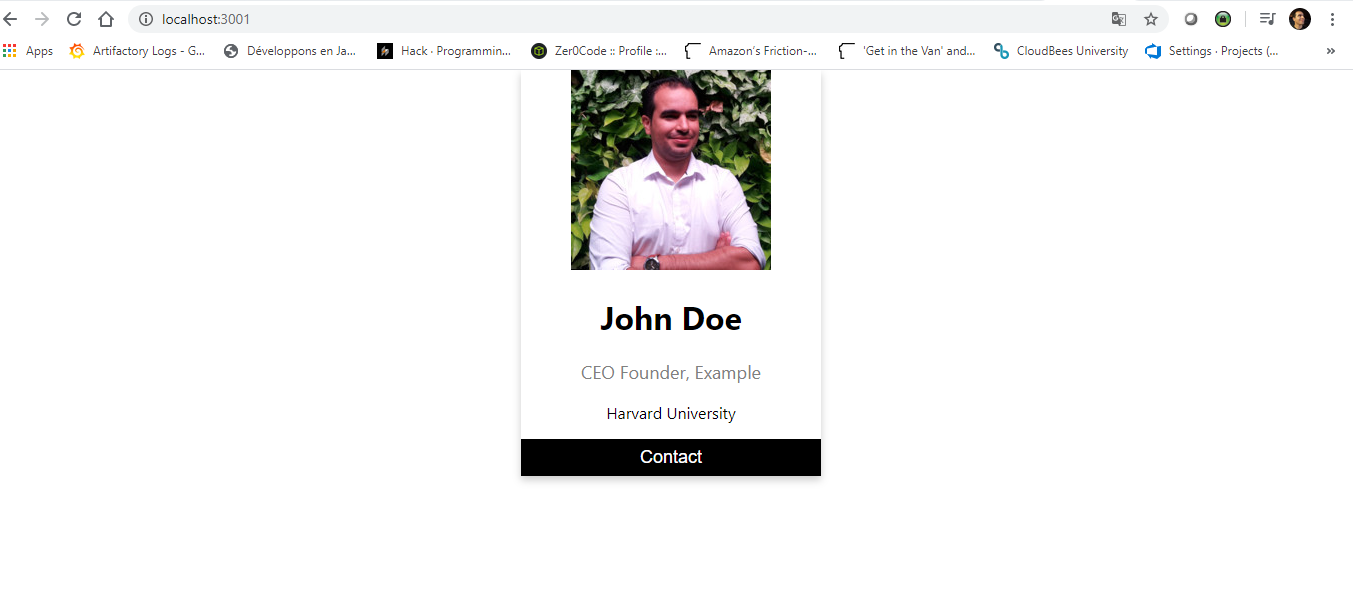
export class ProfileCard extends React.Component {

...

}

You’re all set, checkout your app in your browser. Feel free to create an object with your information and use same inside your profile card.





**4. this.props**

**4.1. Accessing a Component's props**

Let’s learn another way component can interact with each other by passing information known as props. Every component contains something known as props; it is an object that contains information about that component.

To access a component’s props object, we use the expression:

this.props

Let’s try out an example: We’ll create a new component called ***ShowProps***.

class ShowProps extends React.Component {

render() {

const myProps = JSON.stringify(this.props);

return (

<div>

<h1>My Props</h1>

<p>{myProps}</p>

</div>

);

}

}

class App extends React.Component {

render() {

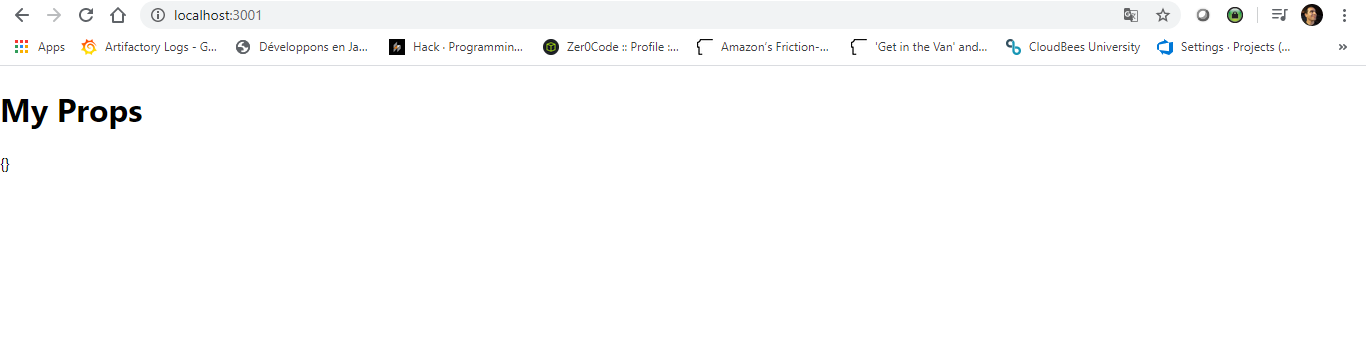
return <ShowProps />;

}

}

export default App;

In the above example, we’ve created a component with a variable **myProps** that holds a string version of the props object, **this.props**. We’ve added this component to our App to be rendered. Save and checkout your browser, notice that the object is empty?



Regardless of what you see in the browser, **<ShowProps />**’s props object is not really empty. It contains properties that **JSON.stringify** isn’t able to detect.

You can pass information to a React component by giving that component an attribute

<MyComponent name="Ahmed HOSNI" />

To pass information that isn’t a string you must wrap that information around a curly brace:

<ShowProps

name="ahmedhosni"

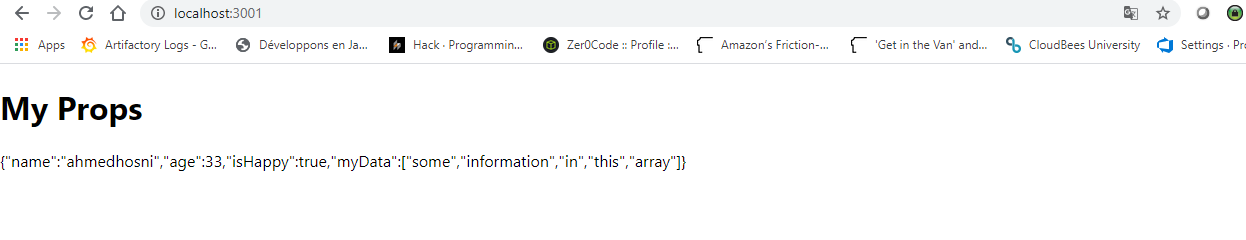
age={33}

isHappy={true}

myData={["some", "information", "in", "this", "array"]}

/>

Update your <ShowProps /> component with the above information, save and checkout your browser.



We have succeeded in passing some information to our component’s props object, we will most often want our component to display the information passed to its props object.

To do that is very easy, first find the component class that is going to receive that information and then use **this.props.name-of-information** inside that component class’s render method's return statement.

Let’s walk through an example with our **ShowProps** component:

class ShowProps extends React.Component {

render() {

const myProps = JSON.stringify(this.props);

return (

<div>

<h1>My Props</h1>

<p>{myProps}</p>

</div>

);

}

}

class App extends React.Component {

render() {

return (

<ShowProps

name="Ahmed HOSNI"

age={30}

isHappy={true}

myData={["some", "information", "in", "this", "array"]}

/>

);

}

}

You can see a piece of information being passed to ***<ShowProps />***. The information passed is name, age, isHappy and myData.

To make name show up in the screen, we need to include **this.props.name** in our **ShowProps** render method’s return statement.

Let’s update our react component to include this information.

class ShowProps extends React.Component {

render() {

return (

<div>

<h1>Show Props</h1>

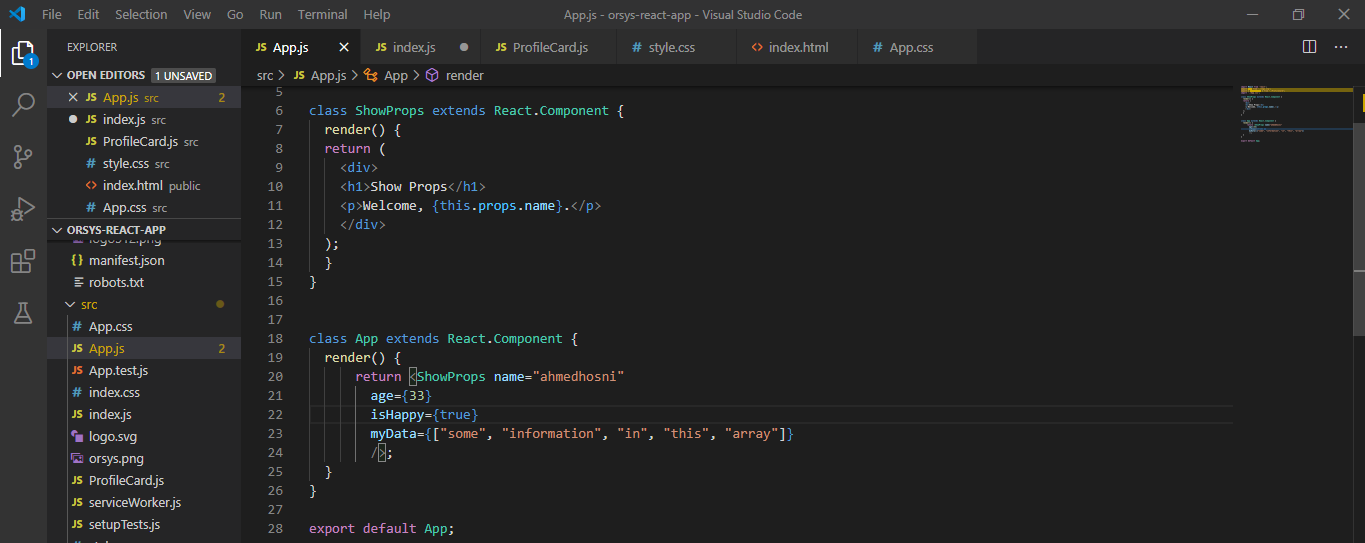
<p>Welcome, {this.props.name}.</p>

</div>

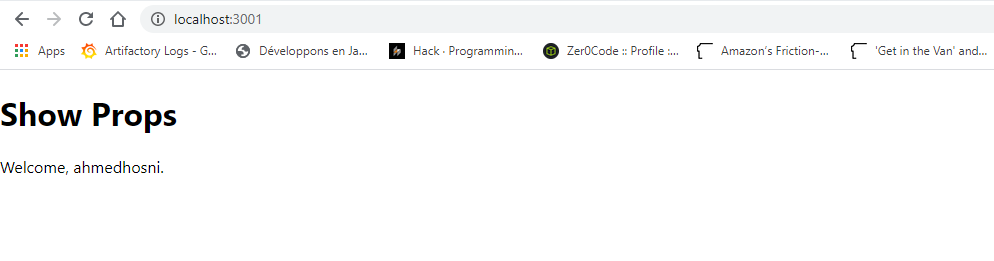
);

}

}



I’ve added a welcome message using ShowProp’s name props, now that’s easy. Save and checkout your browser.



**4.2. Pass props From Component To Component**

You now know how to pass a prop to a component:

<ShowProps age={30} />

You also know how to access and display passed-in prop:

render(){

return <h3>{this.props.age}</h3>

}

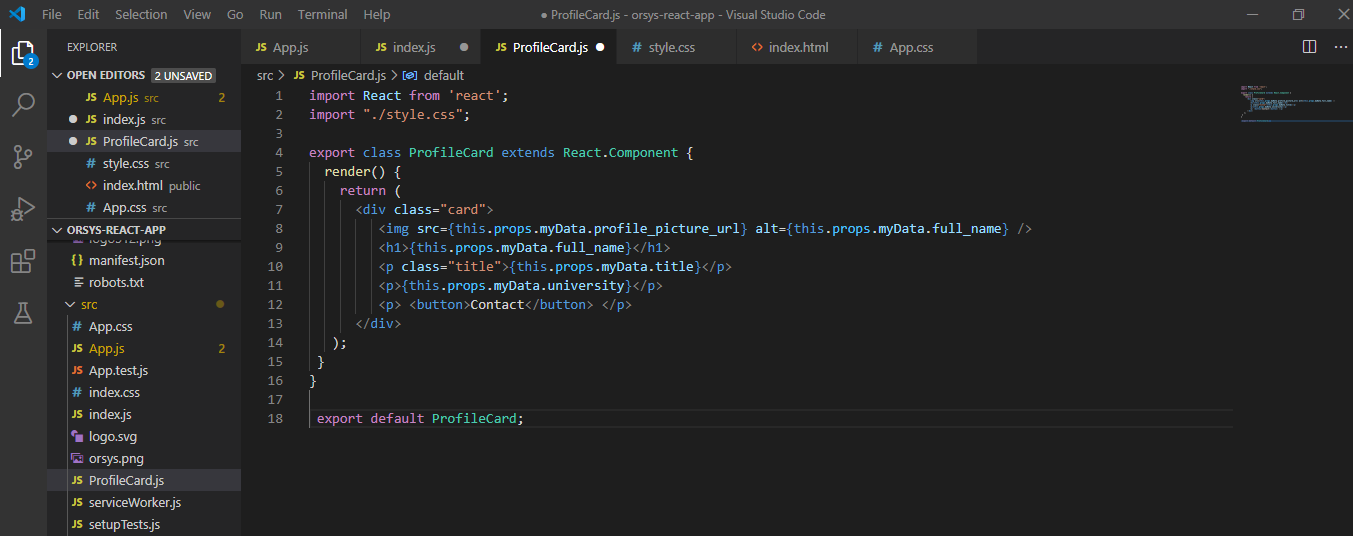
Now, we will learn how to pass props information to a different component entirely, this is the most common use of props, passing information to a component from a different component.

*Note: props is the name of the object that contains passed-in information, whereas each piece of information passed is called a prop.*

Let’s get our hands dirty, remember our **ProfileCard** component?

Well, let’s upgrade that component to pass all its information via props to our *App.js*

**Step 1: Open ProfileCard.js**

****

Well, it’s real easy. Since the object, **myData** is going to be passed using props object there’s no need of having **myData** object defined within **ProfileCard.js**.

Let’s start by deleting that entire object from **ProfileCard.js**, and replacing all calls to *{myData.property}* with *{this.props.property}*, where **property = the object’s attributes**.

import React from "react";

import "./style.css";

export class ProfileCard extends React.Component {

render() {

return (

<div class="card">

<img src={this.props.myData.profile\_picture\_url} alt={this.props.myData.full\_name} />

<h1>{this.props.myData.full\_name}</h1>

<p class="title">{this.props.myData.title}</p>

<p>{this.props.myData.university}</p>

<p> <button>Contact</button> </p>

</div>

);

}

}

Now, let’s get into our *App.js*. First, import **ProfileCard** component into *App.js*, and render **ProfileCard** while passing **myData** object as props.

import React from "react";

import { ProfileCard } from "./ProfileCard";

class App extends React.Component {

render() {

return (

<div>

<ProfileCard

myData={

{

age: 30,

full\_name: "Ahmed Hosni",

university: "TUNIS University",

title: "CEO",

linkedIn\_url: "https://www.linkedin.com/in/ahmed-hosni-18a07526/",

profile\_picture\_url: "https://media-exp1.licdn.com/dms/image/C4E03AQH-k3ui94NjXg/profile-displayphoto-shrink\_200\_200/0?e=1597881600&v=beta&t=oqWjiAecfjn-\_-PvN1TsgHXR6izpjTux4aKvIOnGp1w",

cover\_photo\_url: "https://media-exp1.licdn.com/dms/image/C4D16AQH\_gh4KH7SxMA/profile-displaybackgroundimage-shrink\_200\_800/0?e=1597881600&v=beta&t=z9Y2lkmj-6-c9Czk-KBdUkLIJiUe1uu8UcExSXK4LWs"

}

}

/>

</div>

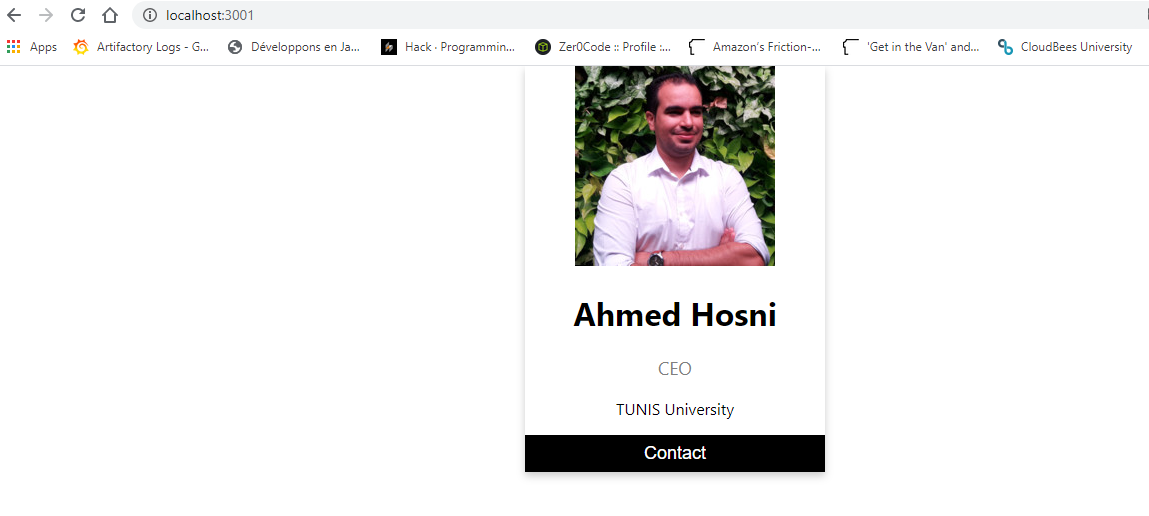
);

}

}

export default App;

Save your changes and checkout your browser, wow! You did it.



**4.3. Render Different UI Based on props**

Congratulations, so far you’ve passed a prop from one component to another, and accessed that prop from the receiver component, and render edit. We can do more than just displaying props, we can also use props to make decisions.

export class ProfileCard extends React.Component {

render() {

if (this.props.myData.age < 21) {

return (

<div class="card">

<p>Sorry, we cannot issue a profile card.</p>

</div>

);

} else {

return (

<div class="card">

<img src={this.props.myData.profile\_picture\_url}

alt={this.props.myData.full\_name}

/>

<h1>{this.props.myData.full\_name}</h1>

<p class="title">{this.props.myData.title}</p>

<p>{this.props.myData.university}</p>

<p> <button>Contact</button> </p>

</div>

);

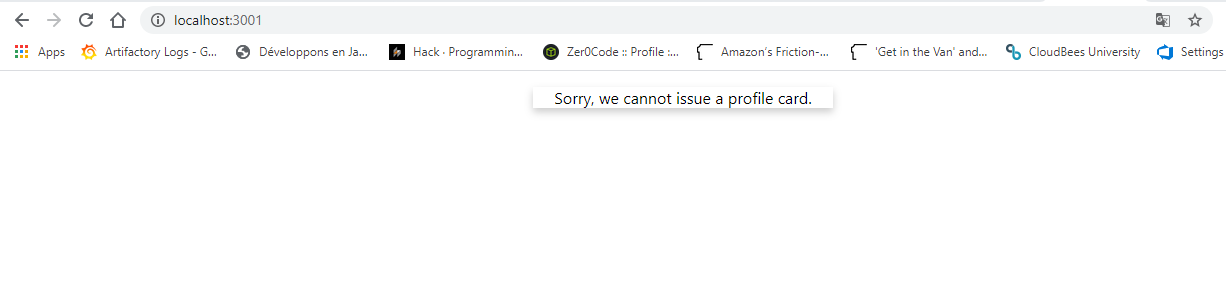
}

}

}

In the above example, we check if the age prop is below 21 as a criteria for issuing a profile card.

Update your **ProfileCard** component’s age prop **in App.js** to 20 and also try out 22, see the difference?



Setting the age prop to 20 will result in:

**4.4. Put an Event Handler in a Component Class**

**We can pass functions as props**; this is a very common way of passing event handler functions. We will learn how to do this, but first we must define an event handler.

You define an event handler as a method within the component class, like the render method.

import React from 'react';

class Example extends React.Component {

eventHandler() {

alert(`I am an event handler.`);

}

render() {

return (

<h1 onClick={this.eventHandler}>

Hello world

</h1>

);

}

}

The **eventHandler** method is defined and looks similar to your render method. In the render method, we call **eventHandler** whenever there’s a click registered on Hello world.

**4.5. Pass an Event Handler as a prop**

Yes, we can pass methods as prop objects to other components. Isn’t that cool? Methods can be passed in the exact same way as any other information.

Let’s create a simple button component:

import React from "react";

export class Button extends React.Component {

render() {

return <button>Click Here</button>;

}

}

Now, let’s use this button in our App, in App.js:

import React from "react";

import { Button } from "./Button";

class App extends React.Component {

render() {

return (

<div align="center">

<h1>My App</h1>

<Button />

</div>

);

}

}

export default App;

Let’s go ahead and create method called **screamHello()** in our App component:

class App extends React.Component {

screamHello() {

alert("Helllooo!!!!");

}

render() {

return (

<div align="center">

<h1>My App</h1>

<Button />

</div>

);

}

}

We want to pass **screamHello** method from our App into our **<Button />** component. To pass any prop to **<Button />,** *you need to give <Button /> an attribute*

<Button scream={this.screamHello} />

Amazing! You just passed a function from <App /> to <Button />. Now let’s attach screamHello() method to the <button></button> as an event handler. Let’s call screamHello() on click of our button:

export class Button extends React.Component {

render() {

return <button onClick={this.props.scream}>Click Here</button>;

}

}

Run your application and see the magic.

**4.6. handleEvent, onEvent, and this.props.onEvent**

It’s time to talk about some naming conventions, when we pass an event handler as a prop, there are two names we have to deal with. Both naming choices occur in the parent component class that defines the event handler and passes it.

First, the name of the event handler, in our previous example we called our event handler ​**screamHello**​.

Secondly, the name of the prop that we use to pass the event handler, that’s the attribute name. In our previous example we used ​scream​.

Well, both names can be anything. However, there’s a naming convention, I’d recommend you sticking to this convention.

For our event handler (screamHello), the naming convention is based on the type of event being listened for.

In **screamHello**, the event type was “click”. If event type is “click”, then the name of your event handler should be **handleClick**. If **mouseOver**, then event handler name should be **handleMouseOver**

class App extends React.Component {

​handleClick()​ {

alert("Helllooo!!!!");

}

render() {

return (

<div align="center">

<h1>My App</h1>

<Button scream={this.​handleClick​} />

</div>

);

}

}

Prop name should be prefixed with the word on, plus the event type. E.g. if you’re listening for a **mouseOver** event, then you name your prop **onMouseOver**, if it’s a click, then, **onClick**, etc.

class App extends React.Component {

​handleClick()​ {

alert("Helllooo!!!!");

}

render() {

return (

<div align="center">

<h1>My App</h1>

<Button ​onClick​={this.​handleClick​} />

</div>

);

}

}

Finally we can update the <Button /> component’s event

export class Button extends React.Component {

render() {

return <button onClick={this.props.onClick}>Click Here</button>;

}

}

I understand that it can be confusing with the **onClick attribute***, onClick* have special meaning only with HTML elements.

**When used with React components, they’re just arbitrary attribute name names.**

**4.7. this.props.children**

*All component’s props object has an attribute named children****, this.props.children*** *returns everything in between a component’s opening and closing JSX tags.*

We’ve been working with self-closing tags such as <Button />, you could also write this as <Button> </Button>, and it would still work.

**This.props.children** returns everything in between <Button> and <Button />.

Let’s define a <List /> component, see how we use **this.props.children** to call all the list items between the **<List> </List>** component.

export class List extends React.Component {

render() {

let titleText = `My Favorite ${this.props.type}`;

if (this.props.children instanceof Array) {

titleText += 's';

}

return (

<div>

<h1>{titleText}</h1>

<ul> {this.props.children} </ul>

</div>

);

}

}

Here’s where we use the List component in our App:

class App extends React.Component {

render() {

return (

<div>

<List type=’Nigerian Musician'>

<li>Davido</li>

<li>Burnaboy</li>

</List>

<List type='Legend Musician'>

<li>Burna Boy</li>

</List>

</div>

);

}

}

**4.8. defaultProps**

Let’s take a look at the Button component below:

class Button extends React.Component {

render() {

return (

<button> {this.props.text} </button>

);

}

}

The Button component expects to receive a prop named text to be displayed inside of the <button></button> element. if text is not passed to the Button component, a blank button will be rendered. It’s best if the button had a default message instead of displaying a blank button.

This is where **defaultProps** plays a role, you can give your props a default value.

class Example extends React.Component {

render() {

return <h1>{this.props.text}</h1>;

}

}

// Set defaultProps to equal an object:

Example.defaultProps = {text: 'Click Me'};

The **defaultProps** property should be equal to an object that contains key value pairs of all the default props you wish to set.

**5. this.state**

**5.1. Setting Initial State**

React components can access information dynamically using props and state. Unlike props, the state of a component is not passed in from another com outside of the component. A component decides its state.

We can initialize the state of a component by giving the component a state property, which is declared inside of a constructor:

class Example extends React.Component {

constructor(props) {

super(props);

this.state = { password: 'p123' };

}

render() {

return <div></div>;

}

}

<Example />

constructor method? super(props)? What’s happening here!

Let’s take a closer look at this unfamiliar code:

constructor(props) {

super(props);

this.state = { password: 'p123’' };

}

**this.state** must equal an object, this object represents the initial state of your component instance. React components always have to call super in their constructors in other to be properly setup.

**5.2. Access a Component's state**

In other to read the state of a component, use the expression below:

this.state.name-of-property

Let’s look at an example:

class BlogPost extends React.Component {

constructor(props) {

super(props);

this.state = { title: "Hello World" };

}

render() {

return <h1>{this.state.title}</h1>;

}

}

The **BlogPost** component class sets title to *“Hello World”* in its state, which is accessed within the component’s render function:​ **{this.state.title}**

**5.3. Update state with this.setState**

Apart from reading a component’s state, you can also change the state of a component by calling the function **this.setState():**

class BlogPost extends React.Component {

constructor(props) {

super(props);

this.state = { title: "Hello World" };

​this.changeState = this.changeState.bind(this)

}

changeState() {

​this.setState({ title: "New world order" });

}

render() {

return <button value="click to change title" onClick={this.changeState} />;

}

}

In the above example, the BlogPost component’s default state contains a title property whose value is set to “Hello World”.

The render function returns a button that when clicked changes the title to “New world order”.

You may have noticed a weird looking line:

​ this.changeState = this.changeState.bind(this)

This line is necessary because **changeState(**)‘s body contains the word this. It’s important to know that in React, whenever you create an event handler that uses ​this​, you need to add **this.methodName = this.methodName.bind(this)** to your constructor function.

**this.setState()** accepts an object, and merges this object with the component’s current state. If there are properties in the component’s current state that isn’t a part of the object, those properties will remain unchanged.

When you call **this.setState()** method, it automatically calls .**render()** as soon as the state has changed. This is the reason you cannot call **this.setState()** from within a .**render()** method, if you do, you get into an infinite loop.

**Exercise 2: Environment setup - Build a Reason React App**

This Learning Lab has a single prerequisite for you to be able to go through it: node. And also yarn.

If you have never used Node.js before, don't worry, we'll include all the necessary steps for you to take, and guide you along the way.

### **Step 0.0 — Install Node**

If you don't have node installed, you can normally get it here: <https://nodejs.org/en/>

But you may also be able to get it through your package manager of choice:

* on Debian-based Linuxes systems that would look like apt-get install node,
* on macOS with Homebrew it would look like brew install node,
* on Windows with **Choco** it looks like choco install nodejs.install

But either way, you can just go to the website above, download the installer, and get it set up in a few clicks.

### **Step 0.1 — Install Yarn**

Since this exercise makes some assumptions on the yarn tool, we will need to install it.

You can run npm install --global yarn to get the latest version of yarn globally available, and you can verify that you have yarn installed by running yarn --version.

### **Step 1.0 — Clone the Project**

1. Clone this repo to your computer

git clone <https://github.com/hosniah/my-reason-react-app.git>

NOTE: If you haven't done this before, we recommend github's guide on [Cloning Repositories](https://help.github.com/en/articles/cloning-a-repository). It is super short and will get you started

1. Go to the project folder in your computer and check out the setup branch.

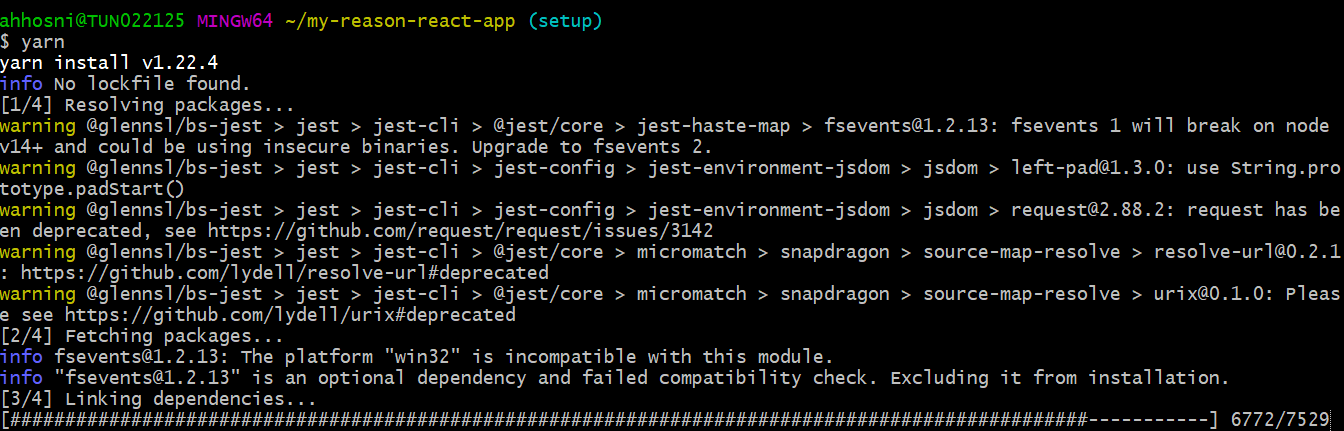
cd my-reason-react-app

git checkout setup

1. Run yarn to install all necessary dependencies:

Yarn

NOTE: installing dependencies sometimes can take a while and sometimes can fail because the internet is not perfect.



1. After the installation is completed, you will have 1 new file (yarn.lock) and 1 new folder (node\_modules). Make sure to add the yarn.lock file with git, commit it, and push your changes. You can do that by running:

git add yarn.lock

git commit --message "Add yarn.lock"

git push origin setup

Now that you have set up the repository with its dependencies, we should set up an editor for you to work in. We have chosen [Visual Studio Code](https://code.visualstudio.com/) because it has the most polished first-time setup experience.

So let's get it installed in your computer, then lets install the [Reason VSCode Extension](https://marketplace.visualstudio.com/items?itemName=jaredly.reason-vscode), and finally browse our code!

### **Step 1.1 — Install VS Code**

You can get VS Code from [here](https://code.visualstudio.com/) and it very likely has an installer for your operating system. After you're done with it, we can install the extension.

### **Step 1.2 — Install Reason Extension for VS Code**

### You can go to the extension page: [Reason VSCode](https://marketplace.visualstudio.com/items?itemName=jaredly.reason-vscode) and you should see a big green button that says *Install*.

After clicking on it, VS Code should show you that the extension is installed and ready to use. If this didn't happen, please check the [Trouble Installing? Page](https://aka.ms/vscode_extn_install).

### **Step 1.3 — Open Project in VS Code**

1. Now that VS Code is open, we can open our project. We can do this by clicking on the File menu, then on Open, and browsing until we find the folder with our package.json.
2. Open the src/model.re file.
3. If all the steps above worked, we should see on the left that a new lib folder appeared. This folder will have all of the compilation output; that is, all the Javascript code.
4. Add lib, commit it and push it

git add lib

git commit --message "Add built files from lib folder"

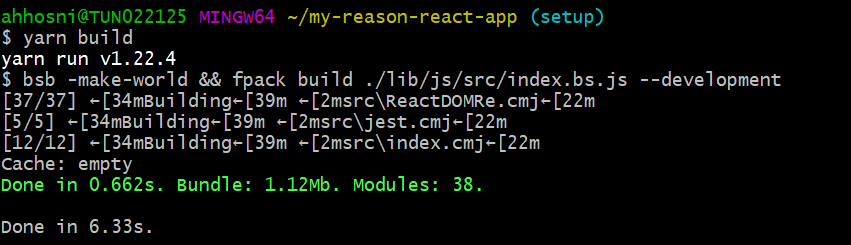
git push origin setup

Now we should be able to build our project bundle to serve it on the web.

NOTE: if you haven't worked with web stuff lately, a bundle is really just a single file that includes most if not all of the source code of the application. It's not too different from the linking process of languages that compile to native.

### **Step 1.4 — Build Project**

1. In your terminal, build the project by running yarn build.



1. You should see in VS Code that a new folder bundle has been created, with a single file: index.js. This is our compiled project.
2. Add bundle to git, commit it, and push it

git add bundle

git commit --message "Add the built bundle folder"

git push origin setup

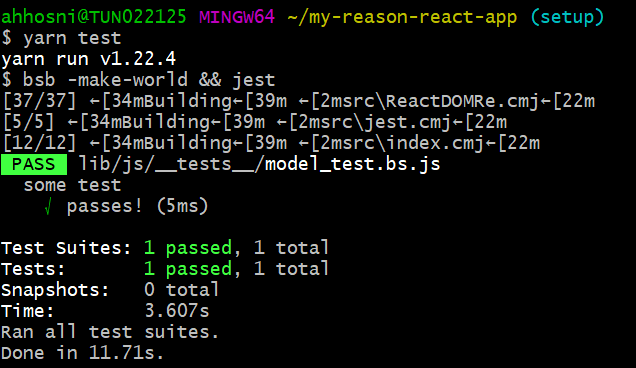
Now we should make sure that even if we change some code, our application is still working well. For that we have included some tests to run.

### **Step 1.5 — Test the Project**

1. In your terminal, run the tests with **yarn test**.



1. You should see a failing test. We need to fix it. This is a small test and it's intentionally failing but it should be a small change to fix it too! You can try removing the false, and adding true instead.



1. Add \_\_tests\_\_ to git, commit it, and push it

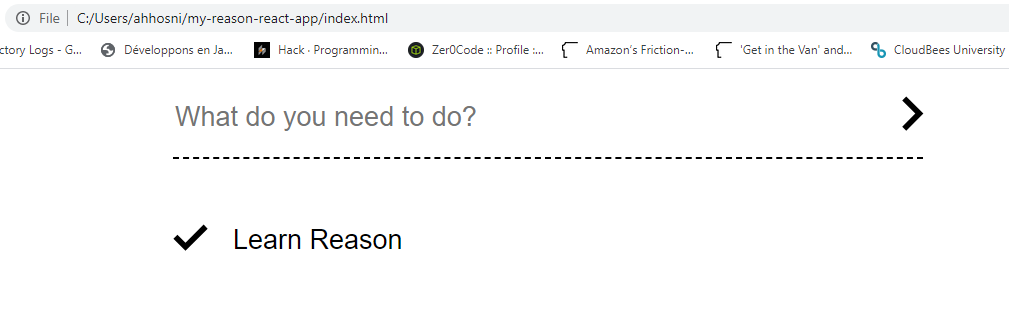
git add \_\_tests\_\_

git commit --message "Fix broken test"

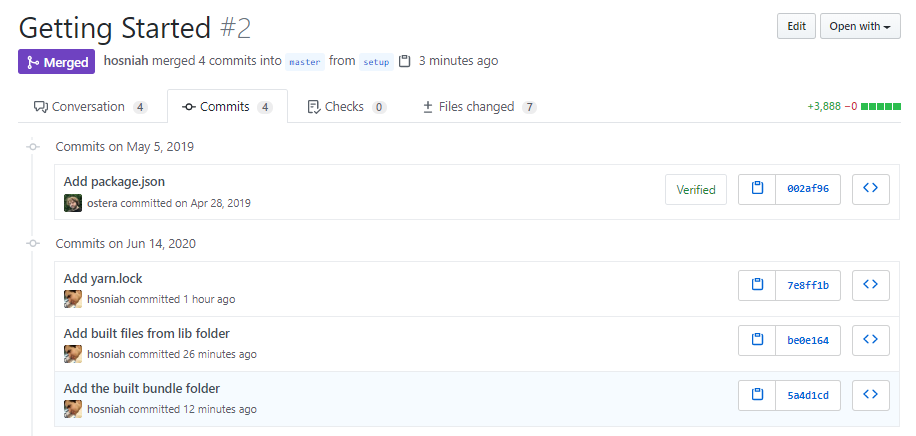
git push origin setup

Everything seems to be in place now, but we haven't verified that the application *runs*. Let's do that.

1. Verify app is running by opening index.html file locally. On macOS you can run open index.html, and on Windows and Linux you should be able to use your web browser to open it.
2. You should see something on the screen that *looks like*  :



You’ll need then to merge the Pull Request from [setup](https://github.com/hosniah/my-reason-react-app/tree/setup) branch into [master](https://github.com/hosniah/my-reason-react-app) branch



**Exercise 3: Understand the Basic Workflow in React**

**3.1. Intro:**

React is an excellent library for creating component-based user interfaces. It takes care of the developer experience by providing declarative APIs for creating components. The React ecosystem has an excellent global package called create-react-app that can easily set up a React project with a single command.

Starters like create-react-app are pretty great tools, but you must understand how it all works under the hood. In this guide, you'll learn how to set up a custom *webpack* config to create a React app from scratch and unpack the workflow behind it.

**3.2. Webpack:**

Before getting started, this section will give you a brief introduction to **webpack**, so that you can follow the next sections.

At its core, **webpack is a static module bundler** for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph that maps every module your project needs and generates one or more bundles.

That means webpack can grab all of your code, manage the dependencies, remove dead code that's not required, and spit out an optimized JavaScript file. It can preprocess or transpile your SASS to CSS, JSX to JS, and lots of other optimizations using webpack plugins.

Your source code might have many files, and you can also import dependencies based on your project structure; in the end, webpack bundles them together in a single .js file inside the dist folder by default. Another cool thing about webpack is that you can do code splitting using dynamic imports so that the users are only served the JavaScript required to run that particular page.

Next up, you are going to see the various plugins required to set up a React app.

## **3.3. Initialize the Project**

Start by creating a directory for the project and initialize npm.

$ mkdir react-app

$ cd react-app

$ npm init -y

After running these commands, you should see a package.json file inside your project.

## **3.4. Install Dependencies**

React has two parts:

the react library, which creates the Virtual DOM, and the react-dom library, which is a renderer for React on the browser. We require both of them to create a complete React web app. So to install these libraries, run the following command:

$ npm i --save react react-dom

Everything installed next are dev dependencies.

Install webpack, which will transpile all the JSX code in the components into JavaScript, and webpack-dev-server to enable hot reloading. That means whenever you make any changes, the browser will update the code automatically without having to be refreshed.

Another webpack dependency you'll require is webpack-cli, which will allow you to write webpack commands and build scripts.

$ npm i --save-dev webpack webpack-dev-server webpack-cli

So next you'll install all the babel dependencies. React uses ES6 class syntax and JSX. Babel will transpile the syntax to browser-friendly JavaScript depending on which browser is being used.

$ npm i --save-dev babel-core babel-loader babel-preset-react babel-preset-env html-webpack-plugin

**3.5. Set up Babel Preset**

Inside the root of your project, create a .babelrc file, which will specify all the babel presets to be used by webpack.

{

presets: ["env", "react"];

}

That's all you need to do for this file.

## **3.6. Set up Webpack Config**

Now that you have all the dependencies required to create a React project, configure webpack. Create a webpack.config.js file in your project directory.

There are a few things you need to do inside the webpack.config.js file. Require the path module to resolve the files in the project and the html-webpack-plugin to create the root HTML file.

const path = require("path");

const HtmlWebPackPlugin = require("html-webpack-plugin");

module.exports = {

// webpack config

};

It's best if you write all of your app code inside an src directory so that webpack can use this directory as source code for your project. Inside the src directory, create an index.js file to be the entry file for your project.

Once this is done, specify the entry and the output path inside webpack config file.

const path = require("path");

const HtmlWebPackPlugin = require("html-webpack-plugin");

module.exports = {

entry: "./src/index.js",

output: {

path: path.resolve(\_\_dirname, "dist"),

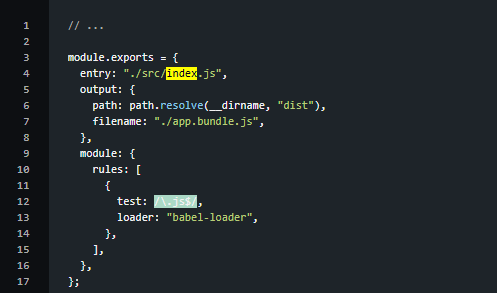
filename: "./app.bundle.js",

},

};

All the files in the src folder will get transpiled into the app.bundle.js file inside the dist folder.

Next, specify a module object inside the config to select the file extension that babel should transpile. In this case, the extension would be .js.



The next step is to create an HTML template inside the src directory into which webpack will automatically inject the final js bundle file in the <script> tag. The benefit of using a template is that you can add in additional tags that you might need for your app.

So inside the webpack config file, specify the HTML template using the html-webpack-plugin.

// ...

module.exports = {

entry: "./src/index.js",

output: {

path: path.resolve(\_\_dirname, "dist"),

filename: "./app.bundle.js",

},

module: {

// ...

},

plugins: [

new HtmlWebPackPlugin({

template: "./src/index.html",

inject: "body",

}),

],

};

That's pretty much it. Now you can write some React code to test this out.

**3.7.Write Build Scripts**

The last step is to include the webpack commands in the scripts object of the package.json file to run the webpack server and transpile the files in the dist folder.

{

// ...

"scripts" : {

"start": "./node\_modules/.bin/webpack-dev-server --mode development --open --hot",

"build": "./node\_modules/.bin/webpack --mode production"

},

// ...

}

The start script will load the development version of the app, open up the browser tab immediately at localhost:8000, and enable hot reloading for you out of the box.

The build script will spit out the production version of your app inside the dist folder. Typically, you would want to serve the dist folder as your root in a production server.

## **3.8.Conclusion**

It's important to understand the tools you use daily so if you hit a bump, you can go back to setting up the project on your own. The great thing is that you don't have to write the webpack config from scratch for each project. You can make a repo, clone the repo, and install the dependencies by running npm install.

Hopefully, this guide clarified the magic that create-react-app does under the hood.

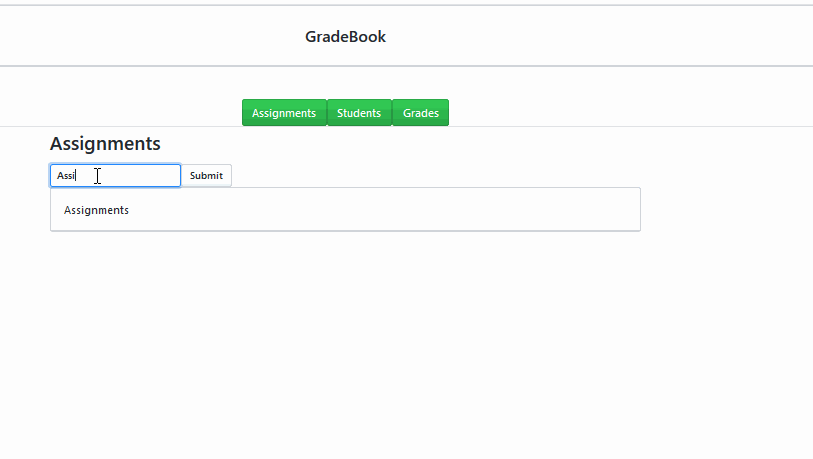
**Exercise 3: Introduction to react – Build a TODO Application**

We'll answer common questions like:

* How do you setup a React project?
* What is a component and how are they used?
* How do you add components?
* How is data passed to child components?
* How are state variables created and used?
* What is a callback function and how is it used to communicate data?

And when you're done you'll be able to:

* Setup a React project
* Add and use components
* Pass data to child components
* Use state variables
* Use callback functions to communicate data



Repositories:

<https://github.com/m2i-training/react-course>

<https://github.com/hosniah/intro-react>