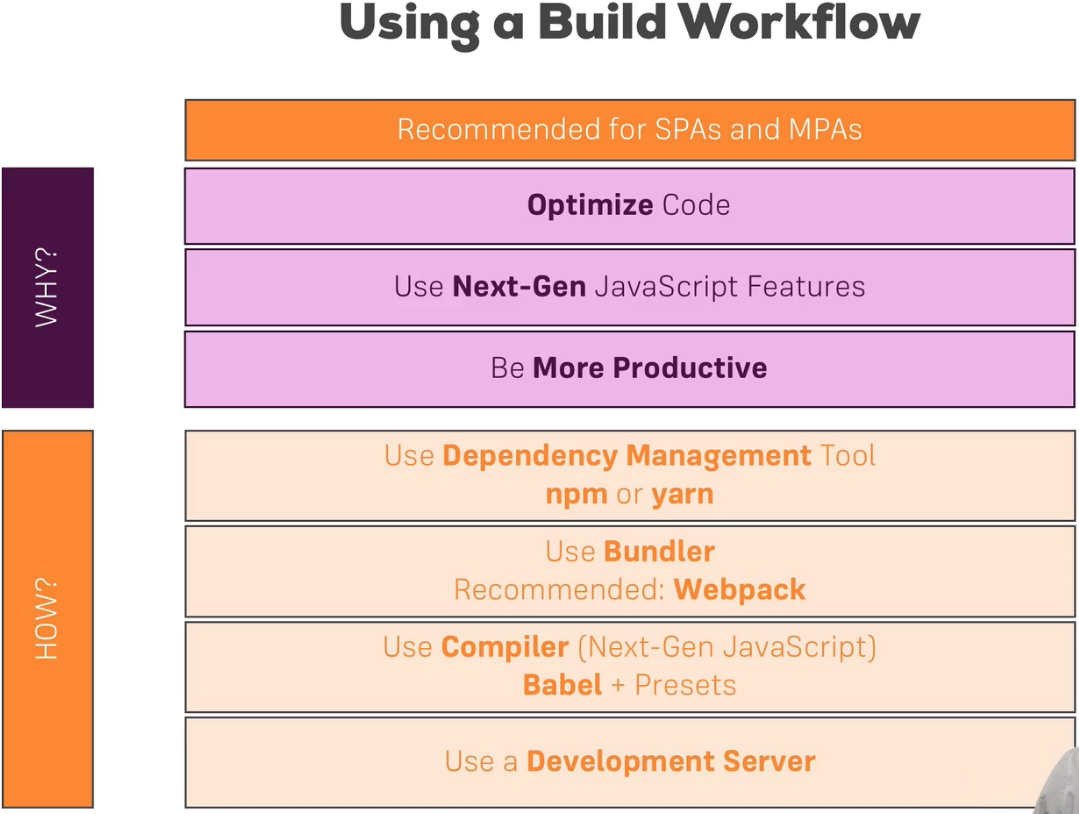
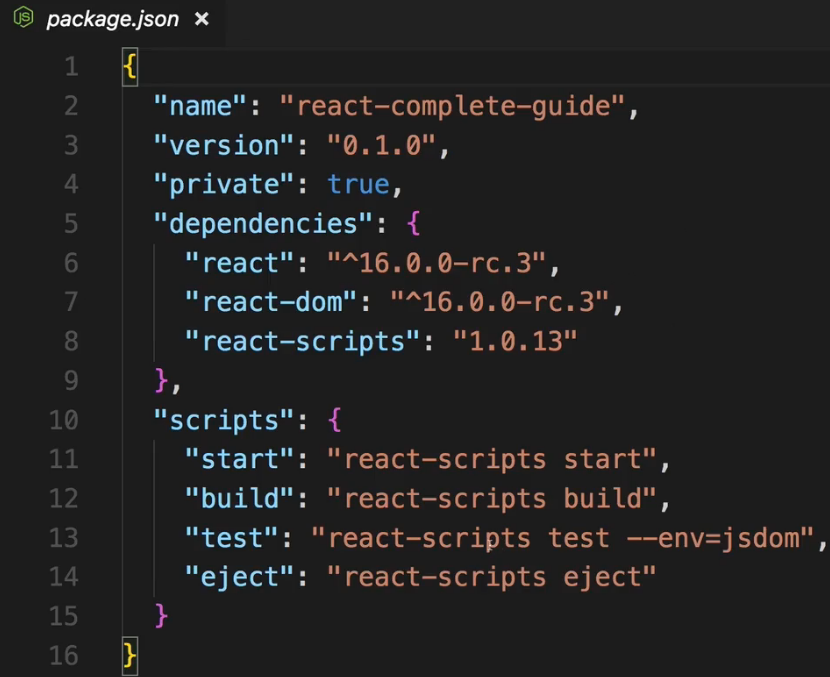
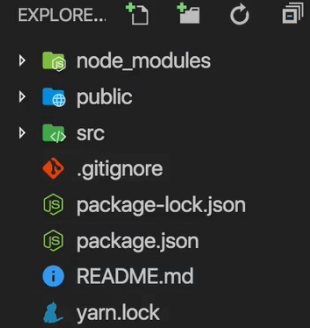
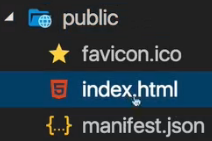
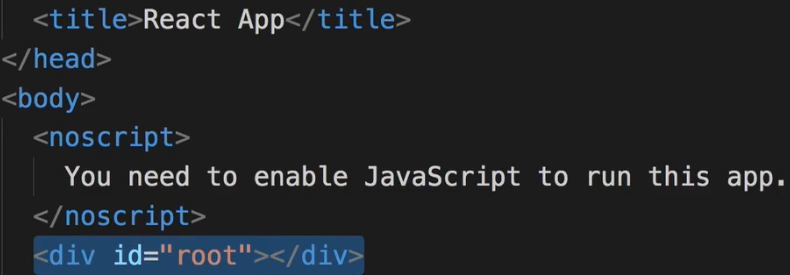
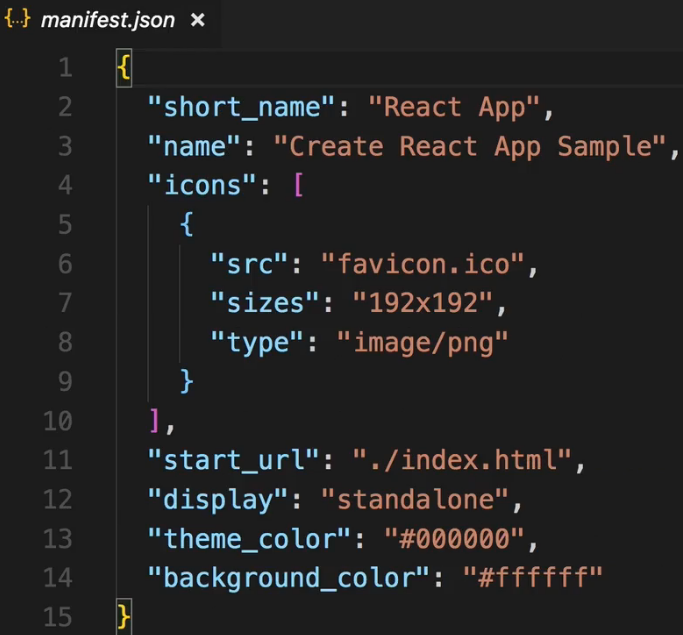
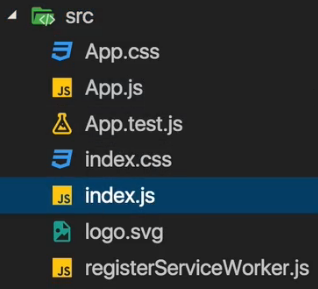
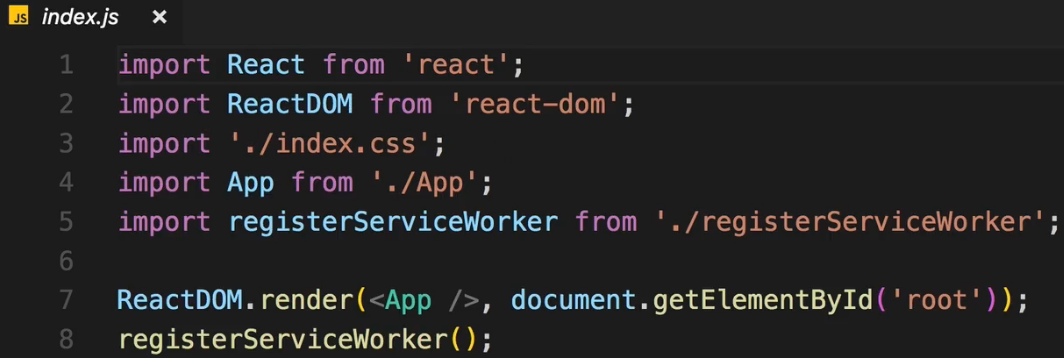
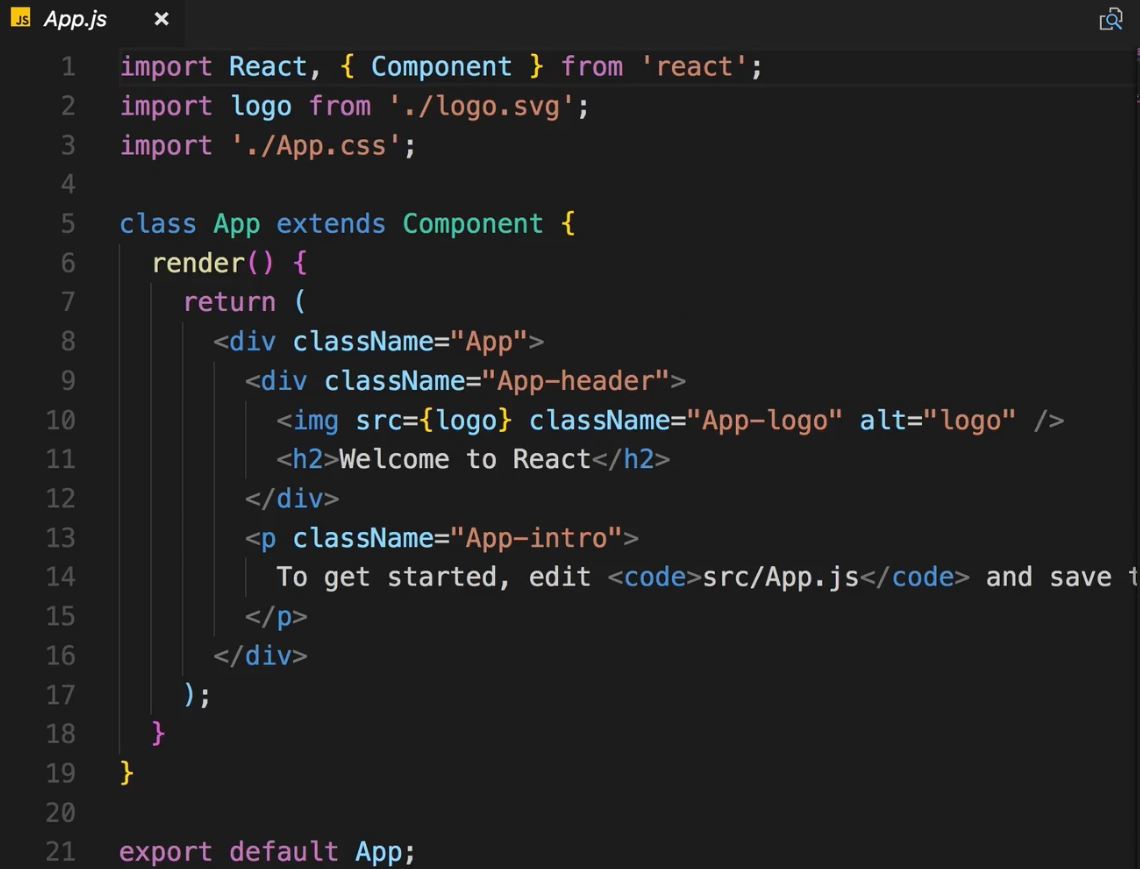
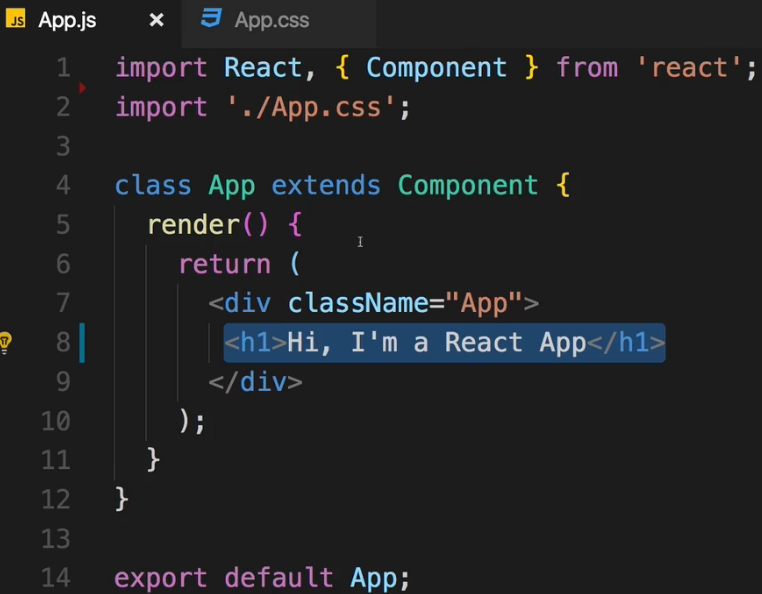
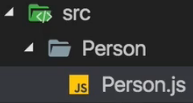
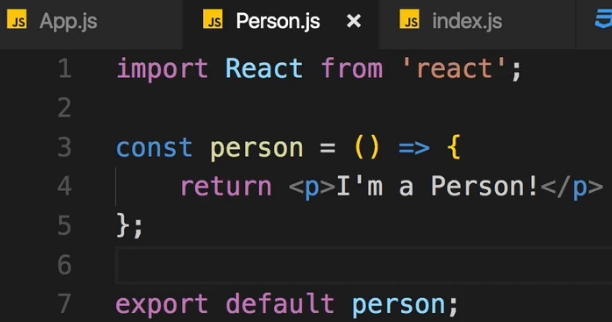
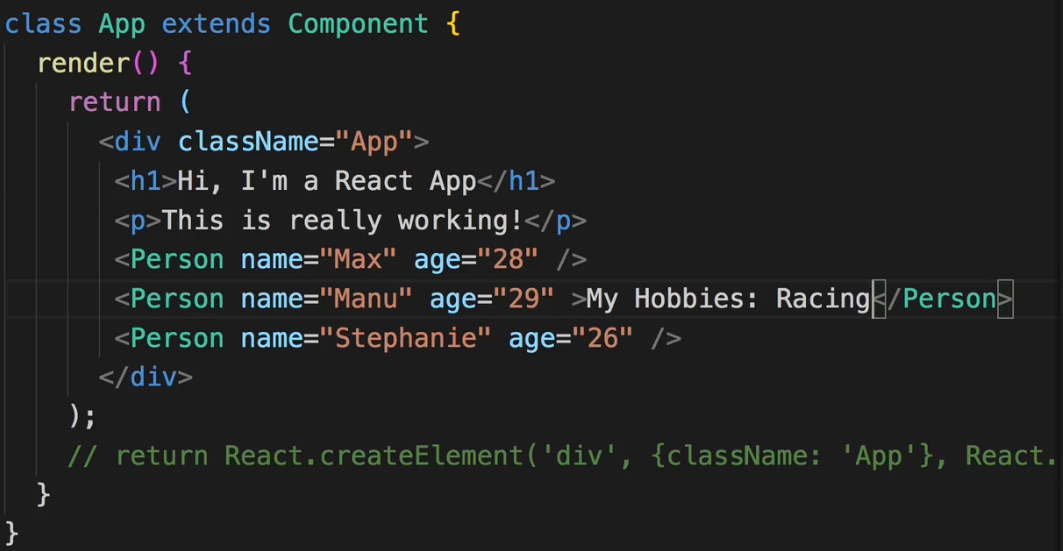
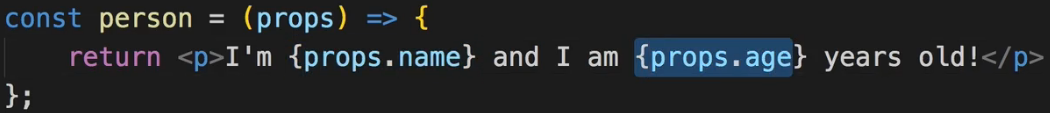
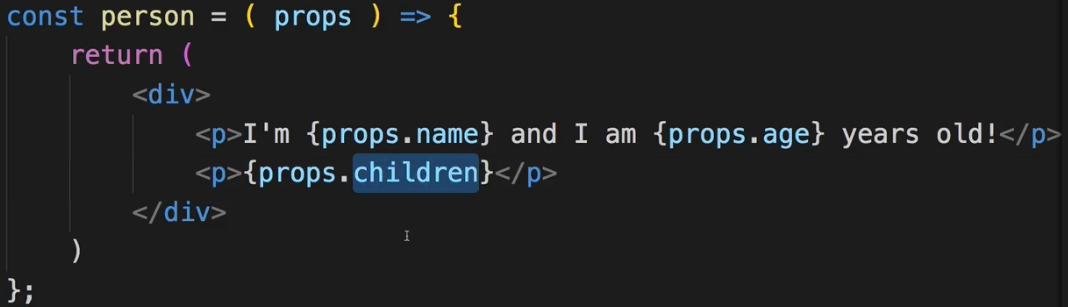
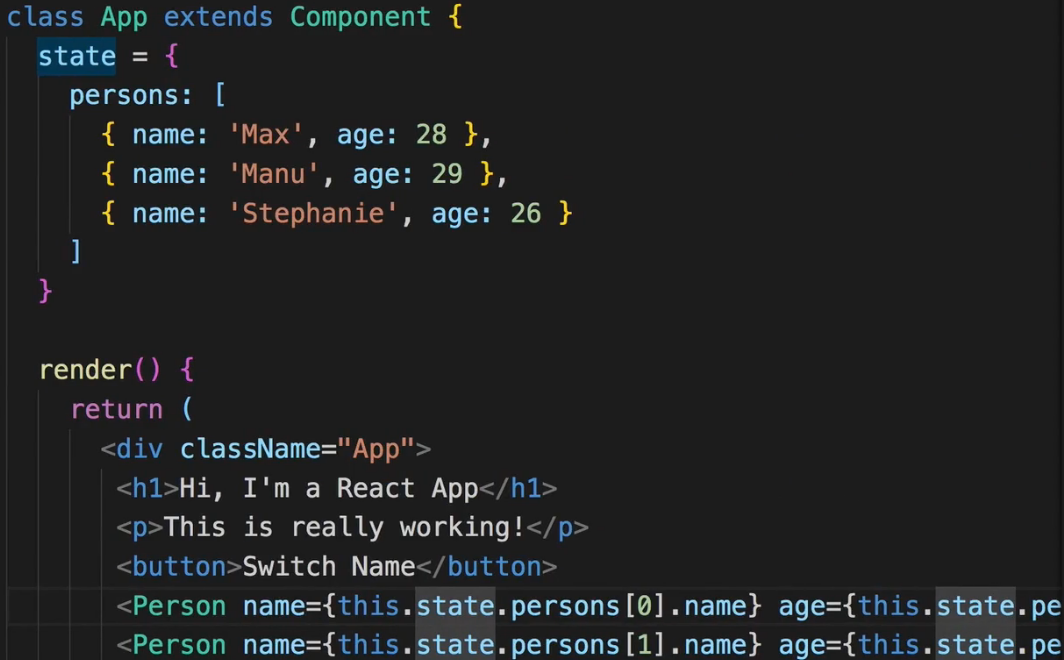
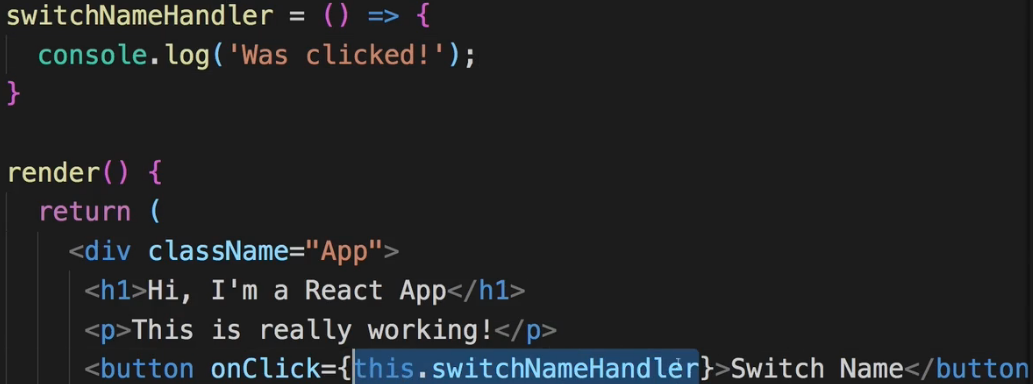
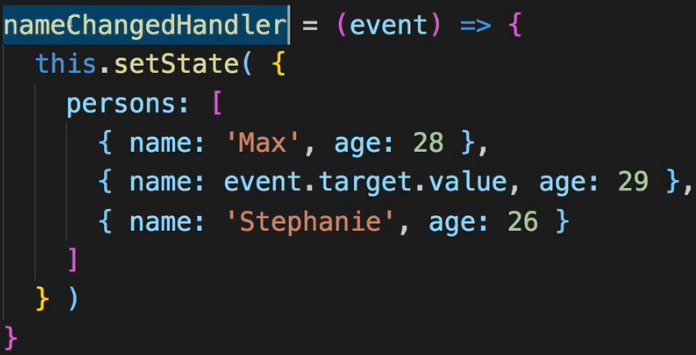
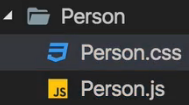
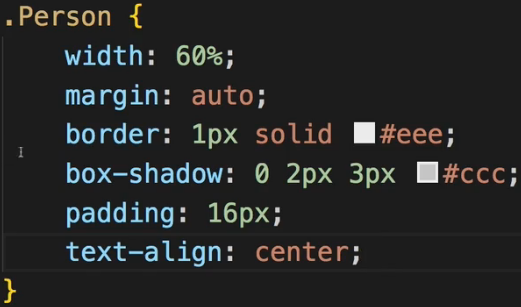
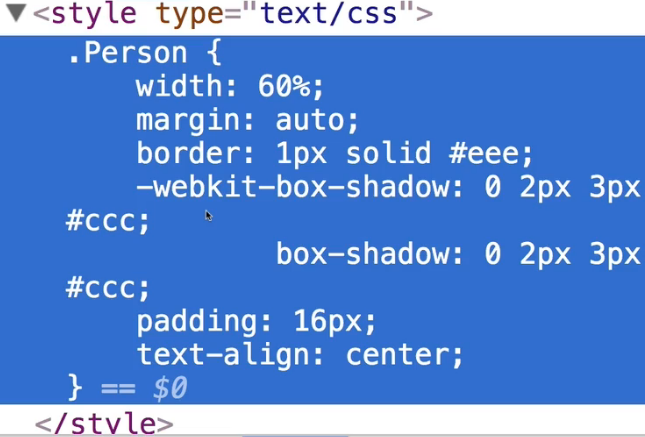
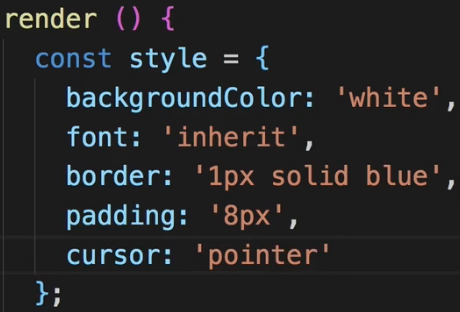
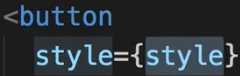
**Section 3 done: 24/25 Understanding the Base Features & Syntax**  
\* npm - Node’s Package Manager  
**download and install npm  
npm install create-react-app -g  
create-react-app some-name**  
**npm start**  
  
\* the `public` folder is basically the root folder which gets served by the web server in the end though here it only holds the files we can edit, the script files are editted in the `src` folder  
  
=> here we got 1 important file: index.html  
\* this is a normal HTML page and it is the single page we have here, we will never add more HTML pages in this project  
\* if you were creating a multi-page project, you would create multiple such projects here with create-react-app, you wouldn’t add more HTML files here - or you need your own workflow if you want to do that  
\* so this is the single page where in the end our script files will get injected by that build workflow - which is why you don’t see a script import here and you can edit this file but we won’t write any HTML code here  
  
=> this will be where we actually mount our React application later  
=> if you need to add any imports to let’s say libraries - css libraries or want to add some meta-tags, you can do that here  
  
\* the manifest.json file is there because create-react-app gives us a progressive web app out of the box or a very basic one at least and here we can define some metadata about our application  


  
=> the index.js file gets access to this `root` element in our DOM, in our HTML file and there it renders our React Application with the ReactDOM.render() method, now here it references some App object/element which we `import` from an `App.js` file  
  
=> this is where we see our first and only React Component we have in this starting project right now   
=> here we see some JSX, for now let’s remove all the content in this wrapping div  
\* we can remove the logo.svg file  
\* it also gave us this `**App.css**` file which basically defines some stylings we use in this App.js file, though I will say that these are not scoped to this file, these are still global stylings and I will actually delete everything but this first .App {text-align: center;} class definition on that file   
\* we also get an `**index.css**` file which also applies styles globally and which should be used for some general set up as here for our `body` of our application (margin: 0, padding: 0, font-family: sans-serif)  
\* the `**registerServiceWorker.js**` is important for registering a service worker which is generated automatically, that’s related to this progressive web app we get out of the box, it will basically pre-catch our script files, we don’t need to configure anything there  
\* and the `**App.test.js**` file - we’ll dive into testing later in this course, it basically allows us to create unit tests for different units for example components in our application  
\* this is the general setup and for the majority of this course, we’ll work in App.js or other new components we create

\* We see a React component, typocally we render 1 root component, in our case <App /> and in there you would nest all the other components your application might need and of course these can be nested into each other but all the way up to the top, you only have 1 root component.  
\* You could reach out to multiple nodes in your HTML file and mount different root components for different React app all in the same project - that would be possible - but it’s not what we do here.  
\* Here we see 1 of 2 ways of creating a React component, it’s a class and it inherits from this `Component` class which is imported up here from the `react` library.  
\* `React` is responsible for rendering anything, anything to the DOM.  
\* Every class needs to have the render() {} method because React will call this method to render something to the screen. There’s 1 important job every React component has to do - it has to return or render some JSX code which can be rendered to the DOM to the screen. You can do other things there too - reach out to the internet, do some calculations, listen to events, but you always also need to render some JSX to the DOM.  
\* We then export this `App` class as the default export of this file - this is an ES6 feature and simply means: if you import this whole file, you simply import this class because it’s the default export.  
\* You might also see these components with .jsx as a file extension instead of .js. The convention is nowadays to just use .js file extension.  
  
=> These 2 are exactly the same. The second one is to what the first one will eventually get compiled.  
=> It is the reason why we need to import `React` even though we’re not using it at all in the first syntax - because behind the scenes we will use it once it is compiled. It’s much easier and that’s why we use the JSX but it’s super important to understand the internals and understand what this compiles to and also most important of all - understand that while it does look like HTML, it isn’t, this is JavaScript in the end, it gets compiled to the second code.  
**JSX Restrictions**  
> className => class  
> our JSX expression must have 1 root element, with React 16 it’s kind of loosened and we can actually return a JSON element in the end but it is a typical best practice to wrap everything into 1 root element per component though, this also makes sense since we typically want to style our components and want to add the css class which is responsible for styling, to the root element as we do here  
> the return (); parenthesis are simply used to that we can write the JSX nicely structured across multiple lines without receiving errors  
  
=> You give your components, the files where you create them, Capital starting characters  
  
\* Most of the time you should use a different way of creating components => a bare function, a simple JavaScript function. Because in its simplest form, component is just a function which returns some JSX.  
\* Inside the `Person.js` we name the function `person` with lower case - this is also a convention, it should otherwise be the same name as your file name though or as the component name you wanna use.  
\* The ES6 arrow function holds some advantages, especially when it comes to the `this` keyword.  
\* We need to import the `React`.  
\* We don’t need the `Component` because here we’re not using a class which extends Component, instead we’re creating a function.  
\* We still need to export that function though as default of this file, we export this `person` constant which holds the function.  
\* When importing the name should be Capital because in React and JSX all elements starting with lowercase characters like div, h1, are reserved for the native HTML elements.  
=> So you could for example create your own element call Div with Capital D and it would work.   
\* We can use <Person></Person> or if you’re not nesting anything then simply <Person />  
**Components & JSX Cheat Sheet**  
<https://www.udemy.com/react-the-complete-guide-incl-redux/learn/v4/t/lecture/8124172?start=0>   
**Re-Using Components**  
\* Advantages of components - modular easier to maintain, reusable, configurable  
**Outputting Dynamic Content**  
\* If we want to run something as a JavaScript code, we wrap it in single curly braces {}  
=> We can execute 1-line expressions. Like simple calculations or function calls. We could call a function there and that function might then do more complex stuff.  
**Props**  
=> **We can also nest something inside our custom elements**

\* It actually is able to take these attributes and gives us access inside our receiving component on object named `props` - the name here is actually up to you. The properties means the attributes you add on your component.  
\* **Functional Components** => **props.name**  
\* **Class-Based Components** => **this.props.name**  
 **props.children**  
=> Accessing whatever we pass inside <></> custom component.  
=> You could nest complex code there too. Could be an unordered list with multiple items.  
=> Could be other React components.  
  
\* You can put your content into your component from outside not only by passing props like name, age, but you can pass some structured HTML content and accessing it with props.children.  
**State**  
\* Sometimes you don’t want to get some information from outside, but you want to have it inside a component and change it from inside there too.  
\* A `class` has properties. You can kind of think of a property as a variable of a class.  
\* Inside a class you can just write **something = someValue**  
\* **There is 1 special property you can define in any component which extends Component**  
=> We can define a special property named `state` = {}.  
=> Whereas `props` are set and passed from outside, state is managed from inside a component.  
=> And `state` is only available in components which are used by extending Component.  
\* Still you should use Function Components as often as possible.  
\* **`this` refers to `class`** - due to our ES6 syntax we’re using.  
=> **this.state.persons[0].name**  
\* State can be changed and if it changes - and that’s the special thing about it and it only works on the state property - if it changes, it will lead React to re-render our DOM - to UPDATE the DOM.  
**Props & State Notes**  
<https://www.udemy.com/react-the-complete-guide-incl-redux/learn/v4/t/lecture/8124208?start=0>   
**Handling Events with Methods**  
\* onClick => onclick (JSX vs HTML)  
**onClick={this.switchNameHandler}**  
\* You typically use `Handler` in the name of a method to indicate that this is a method you’re not actively calling but you’re assigning as an event handler. It is a good practice to name it like this.  
\* We can assign a function as a value here and it becomes a method basically - it’s still a property you could say but a property which holds a function which can be executed.  
\* Don’t add parenthesis => this would execute it immediatelly once React renders this to the DOM. We only want to pass a reference.  
  
**Events Notes**  
<https://www.udemy.com/react-the-complete-guide-incl-redux/learn/v4/t/lecture/8124210?start=0>   
**Manipulating the State**  
\* We shouldn’t mutate (change) the state directly like this:  
  
=> React will not recognize that and will not pick up this change.  
\* Instead we use a special method that comes from the Component  
=> this.setState({})  
=> This is a method which allows us to update this special state property and it will then ensure that React gets to know about this update and updates the DOM.  
=> It will merge whatever we define in it with our existing state. It will not discard other state. We can overwrite something but leave other state untouched if we don’t change it.  
\* There aren’t many things which lead React to update the DOM. There actually only are 2.  
 => **1. changing state**  
 => **2. props** - what we actually output for each person is defined in this Person component and there we use props and that’s the other thing React watches out for. If state changes or props changes it basically analyzes the code it already rendered to the DOM and the code it would now render if it were to re-render everything. And then it updates the existing DOM in all the places where it needs to update it to reflect your new state and props.  
  
\* In Function Components we can’t use state.  
\* But we could run other code before returning some JSX - and you often do that if you need to transform your props first or something like that.  
\* Use them as much as possible because these simple components which are just functions receiving props are very clear about what they do. They only render something to the DOM, they are dynamic because of props and you can add some additional logic prior to calling return(); but - and that is super important - they don’t manipulate your application’s state. As your application grows this is very important. Most parts of your application shouldn’t state the application’s state. They should just render something to the DOM.  
=> Your application’s **state should only be changed and handled in a few selected components** also referred to as **containers**. We call it a container because it contains some part of our application’s state.  
\* You might have cases where maybe you also want to listen to an event in the Person component, now you could turn this into a component which extends Component so that you can define methods which you execute, but maybe you want to listen to an event here (in Person.js) but execute some method in App.js so that you can keep the pattern of changing the name in App.js but actually listening to the event in another component. We can handle this and change the state from another component.  
**Passing Method References Between Components**  
\* We can pass a reference to it as a property to a component.  
\* We can pass it to 1/3 Person components.  
\* While passing we use for example: click={this.switchNameHandler}  
\* And then using it inside that component: onClick={props.click}  
=> You can pass methods also as props so that you can call a method which might change the state, in another component, which doesn’t have direct access to the state.  
  
\* **There are 2 ways of passing an argument to the method**:  
1.  
  
2.   
  
=> it can receive an event argument  
=> on 1 line there is a hidden `return` keyword, it returns the function call  
 => earlier I said you should use the () but here it’s not executed immediatelly, instead what we pass here is an anonymous function which will be executed on a click and which then returns the result of this function getting executed  
=> we could wrap it in {} and write a regular code  
=> now this is a very convenient syntax but it can be inefficient - React can re-render certain things in your app too often - so I don’t necessarily recommend using this if you don’t have to - use the `bind syntax` instead if you can  
**Adding Two Way Binding**  
\* For now we always change a specific name - later we’ll see how to change the one in which we type  


\* Then we pass it to the Person component as a property.  
\* And we call onChange={props.changed}  
  
\* This shows us 2 things  
 => How we can dynamically update something, dynamically call an event and use the things we learned before like passing down method references   
 => But it also shows us how we can handle inputs. Now it would also be nice if we would see the current value of the name in the input right from the start.   
 \* So we basically wanna set up **two-way binding**   
 => When we change it, we wanna propagate that change so that we can update the state but we also want to see the current state right from the start.  
=> To do this I can set **value={props.name}**  
=> And now we have our own two-way binding set up.  
 => We listen to changes, call the **onChange={props.changed}** method in the end which refers to the nameChangedHandler which updates the state and we pass down the state to the person with **name={this.state.persons[1].name}** and we output the name as the value of the input. This now allows us to show that value right from the start  
**Adding Styling with Stylesheets**  
  
\* One important thing - whichever CSS code I write in here is NOT scoped to this Person.js component - it is global CSS code.  
  
\* I will name this class .Person{} - still I could add this anywhere in our application since it is global but by using our component name we rule out the danger of accidentally using it somewhere else.  
\* By default no file is included into your project, into the code which gets created by the build workflow - you always have to import files to add them to the game.  
  
\* Now it might look strange to import a CSS file into a JavaScript file like this - but thanks to Webpack which is the build tool which is in the end used by this React scripts package we’re using implicitly here, we can actually import CSS into JavaScript though it will not really merge the two files or anything like that - it is just made aware of that CSS file and will basically import it into our HTML file. It shows in the final HTML file in the head tag as style tags. So they are injected dynamically by webpack. You can also see that automatic pre-fixing which is very convenient because it makes sure that we don’t have to do that - we can write the shortest CSS code possible/needed.  


\* Let’s now see the 2nd way of styling.  
**Working with Inline Styles**  
\* There is nothing wrong with using CSS files and classes and that might even be the preferred way.   
\* But what you also often see in React apps is that you actually design your styles in JavaScript.  
\* JavaScript CSS representation - you can use “background-color” or just backgroundColor.  
  
  
\* Now it doesn’t change in hovering and actually styling that hover effect is pretty hard when using inline styles. This is one restriction already but I got a whole module in this course where I show you some clever solutions to styling React components which really can be something you can put a lot of thought into.  
\* Always be aware you can use a CSS file though then the styles defined here are global so if we change button element here, all buttons in our whole app will be changed so this might not be what you want but on the other hand you can use the normal CSS syntax.  
\* Or you use inline styles, then the styling is scoped to the component or to the element you actually add it to but you have some restrictions of not being able to leverage the full power of CSS.  
\* Later - there is a golden way in the middle where you can scope styles and still use all the CSS features.  
create-react-app: <https://github.com/facebookincubator/create-react-app>  
Introducing JSX: <https://reactjs.org/docs/introducing-jsx.html>  
Rendering Elements: <https://reactjs.org/docs/rendering-elements.html>  
Components & Props: <https://reactjs.org/docs/components-and-props.html>  
Listenable Events: <https://reactjs.org/docs/events.html>

**Assignment 1: Time to Practice - The Base Syntax**  
x

Components are the **core building block of React apps**. Actually, React really is just a library for creating components in its core.

A typical React app therefore could be depicted as a **component tree** - having one root component ("App") and then an potentially infinite amount of nested child components.

Each component needs to return/ render some **JSX** code - it defines which HTML code React should render to the real DOM in the end.

**JSX is NOT HTML** but it looks a lot like it. Differences can be seen when looking closely though (for example className in JSX vs class in "normal HTML"). JSX is just syntactic sugar for JavaScript, allowing you to write HTMLish code instead of nested React.createElement(...) calls.

When creating components, you have the choice between**two different ways:**

1. **Functional components** (also referred to as "presentational", "dumb" or "stateless" components - more about this later in the course) => const cmp = () => { return <div>some JSX</div> } (using ES6 arrow functions as shown here is recommended but optional)
2. **class-based components** (also referred to as "containers", "smart" or "stateful" components) => class Cmp extends Component { render () { return <div>some JSX</div> } }

We'll of course dive into the difference throughout this course, you can already note that you should use 1) as often as possible though. It's the best-practice.

props  and state  are **CORE concepts** of React. Actually, only changes in props  and/ or state  trigger React to re-render your components and potentially update the DOM in the browser (a detailed look at how React checks whether to really touch the real DOM is provided in section 6).

**Props**

props  allow you to pass data from a parent (wrapping) component to a child (embedded) component.

**Example:**

**AllPosts Component:**

1. const posts = () => {
2. return (
3. <div>
4. <Post title="My first Post" />
5. </div>
6. );
7. }

Here, title  is the custom property (prop ) set up on the custom Post  component. We basically replicate the default HTML attribute behavior we already know (e.g. <input type="text">  informs the browser about how to handle that input).

**Post Component:**

1. const post = (props) => {
2. return (
3. <div>
4. <h1>{props.title}</h1>
5. </div>
6. );
7. }

The Post  component receives the props  argument. You can of course name this argument whatever you want - it's your function definition, React doesn't care! But React will pass one argument to your component function => An object, which contains all properties you set up on <Post ... /> .

{props.title}  then dynamically outputs the title  property of the props  object - which is available since we set the title  property inside AllPosts  component (see above).

**State**

Whilst props allow you to pass data down the component tree (and hence trigger an UI update), state is used to change the component, well, state from within. Changes to state also trigger an UI update.

**Example:**

**NewPost Component:**

1. class NewPost extends Component { // state can only be accessed in class-based components!
2. state = {
3. counter: 1
4. };
6. render () { // Needs to be implemented in class-based components! Needs to return some JSX!
7. return (
8. <div>{this.state.counter}</div>
9. );
10. }
11. }

Here, the NewPost  component contains state . Only class-based components can define and use state . You can of course pass the state  down to functional components, but these then can't directly edit it.

state  simply is a property of the component class, you have to call it state  though - the name is not optional. You can then access it via this.state  in your class JSX code (which you return in the required render()  method).

Whenever state  changes (taught over the next lectures), the component will re-render and reflect the new state. The difference to props  is, that this happens within one and the same component - you don't receive new data (props ) from outside!

In the last lecture, we saw that you can react to the onClick event - but to which other events can you listen? You can find a list of supported events here: <https://reactjs.org/docs/events.html#supported-events>

Clipboard Events

Event names:

1. onCopy onCut onPaste

Properties:

1. DOMDataTransfer clipboardData

Composition Events

Event names:

1. onCompositionEnd onCompositionStart onCompositionUpdate

Properties:

1. string data

Keyboard Events

Event names:

1. onKeyDown onKeyPress onKeyUp

Properties:

1. boolean altKey
2. number charCode
3. boolean ctrlKey
4. boolean getModifierState(key)
5. string key
6. number keyCode
7. string locale
8. number location
9. boolean metaKey
10. boolean repeat
11. boolean shiftKey
12. number which

Focus Events

Event names:

1. onFocus onBlur

These focus events work on all elements in the React DOM, not just form elements.

Properties:

1. DOMEventTarget relatedTarget

Form Events

Event names:

1. onChange onInput onInvalid onSubmit

For more information about the onChange event, see [Forms](https://reactjs.org/docs/forms.html).

Mouse Events

Event names:

1. onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
2. onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
3. onMouseMove onMouseOut onMouseOver onMouseUp

The onMouseEnter and onMouseLeave events propagate from the element being left to the one being entered instead of ordinary bubbling and do not have a capture phase.

Properties:

1. boolean altKey
2. number button
3. number buttons
4. number clientX
5. number clientY
6. boolean ctrlKey
7. boolean getModifierState(key)
8. boolean metaKey
9. number pageX
10. number pageY
11. DOMEventTarget relatedTarget
12. number screenX
13. number screenY
14. boolean shiftKey

Selection Events

Event names:

1. onSelect

Touch Events

Event names:

1. onTouchCancel onTouchEnd onTouchMove onTouchStart

Properties:

1. boolean altKey
2. DOMTouchList changedTouches
3. boolean ctrlKey
4. boolean getModifierState(key)
5. boolean metaKey
6. boolean shiftKey
7. DOMTouchList targetTouches
8. DOMTouchList touches

UI Events

Event names:

1. onScroll

Properties:

1. number detail
2. DOMAbstractView view

Wheel Events

Event names:

1. onWheel

Properties:

1. number deltaMode
2. number deltaX
3. number deltaY
4. number deltaZ

Media Events

Event names:

1. onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted
2. onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay
3. onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend
4. onTimeUpdate onVolumeChange onWaiting

Image Events

Event names:

1. onLoad onError

Animation Events

Event names:

1. onAnimationStart onAnimationEnd onAnimationIteration

Properties:

1. string animationName
2. string pseudoElement
3. float elapsedTime

Transition Events

Event names:

1. onTransitionEnd

Properties:

1. string propertyName
2. string pseudoElement
3. float elapsedTime

Other Events

Event names:

1. onToggle
2. create-react-app: <https://github.com/facebookincubator/create-react-app>
3. Introducing JSX: <https://reactjs.org/docs/introducing-jsx.html>
4. Rendering Elements: <https://reactjs.org/docs/rendering-elements.html>
5. Components & Props: <https://reactjs.org/docs/components-and-props.html>
6. Listenable Events: <https://reactjs.org/docs/events.html>