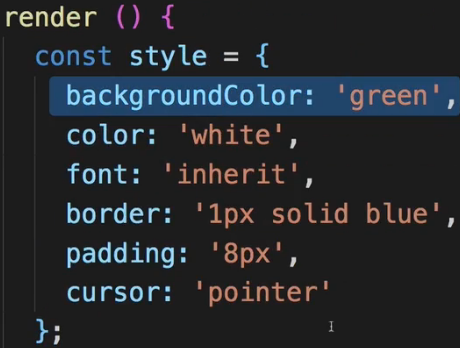
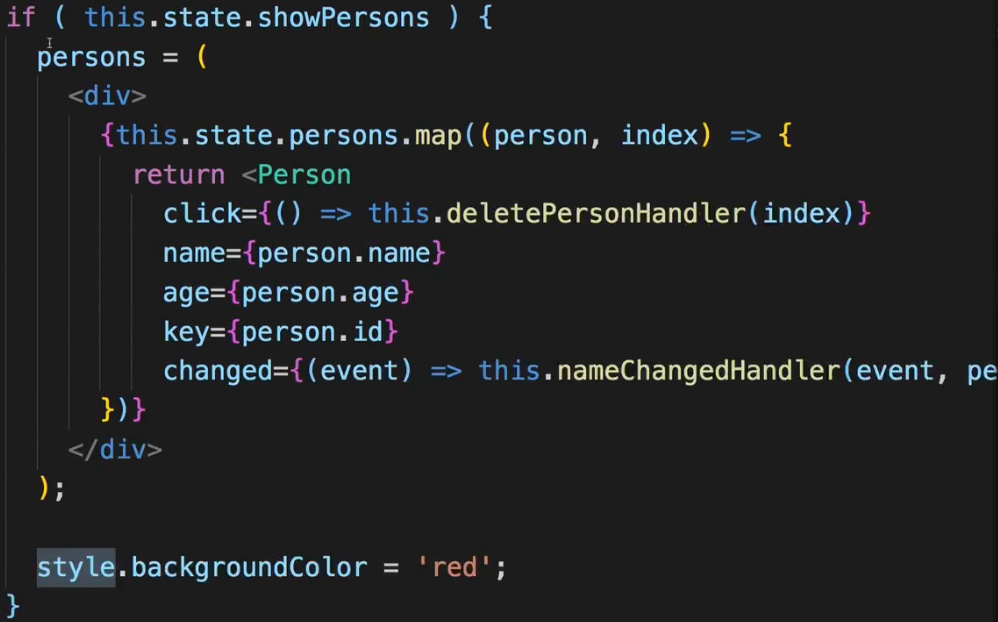
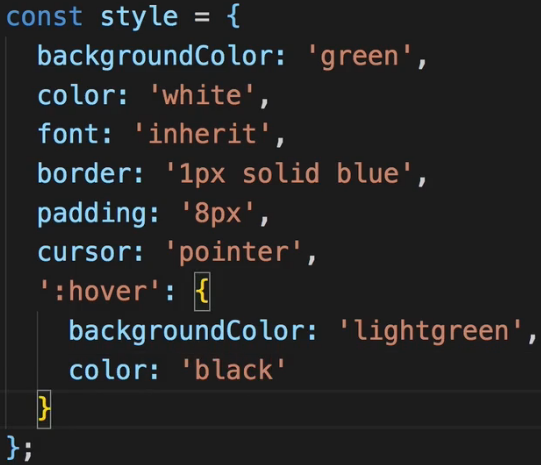
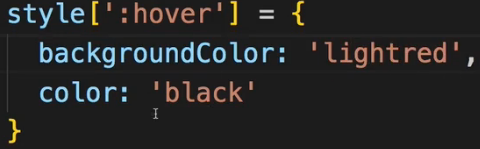
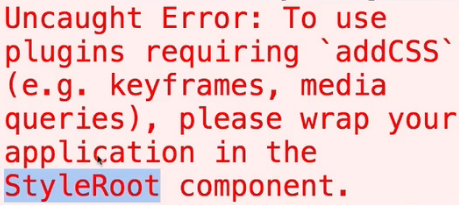
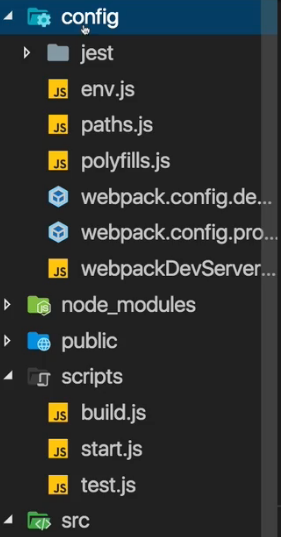
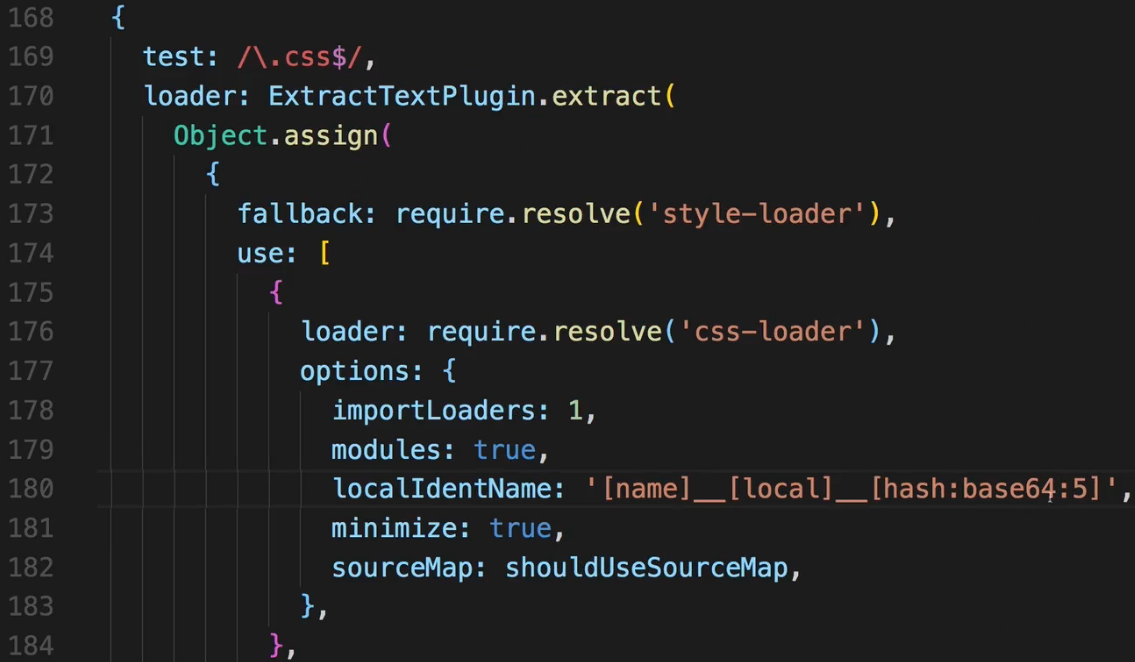
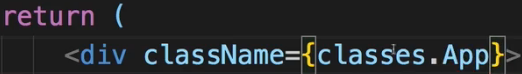
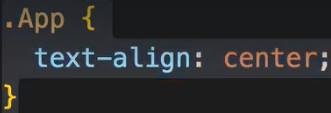
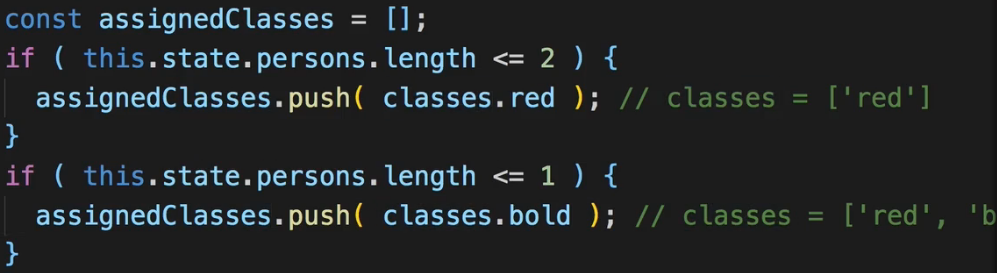
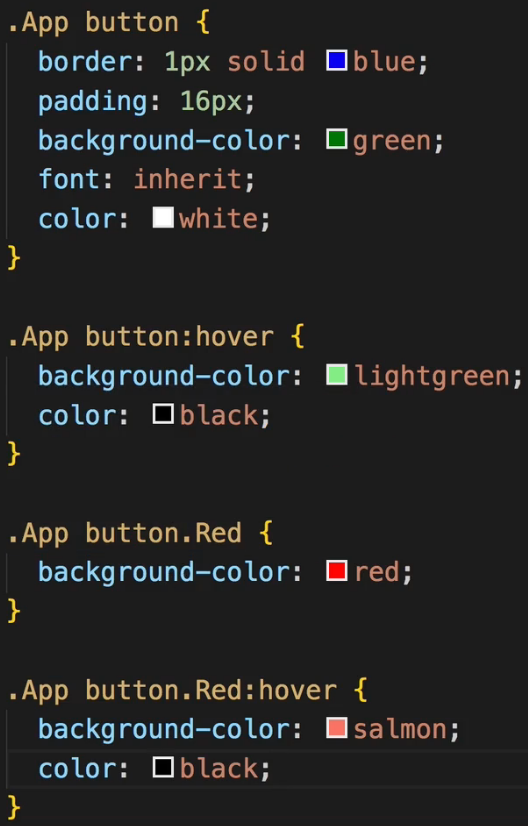
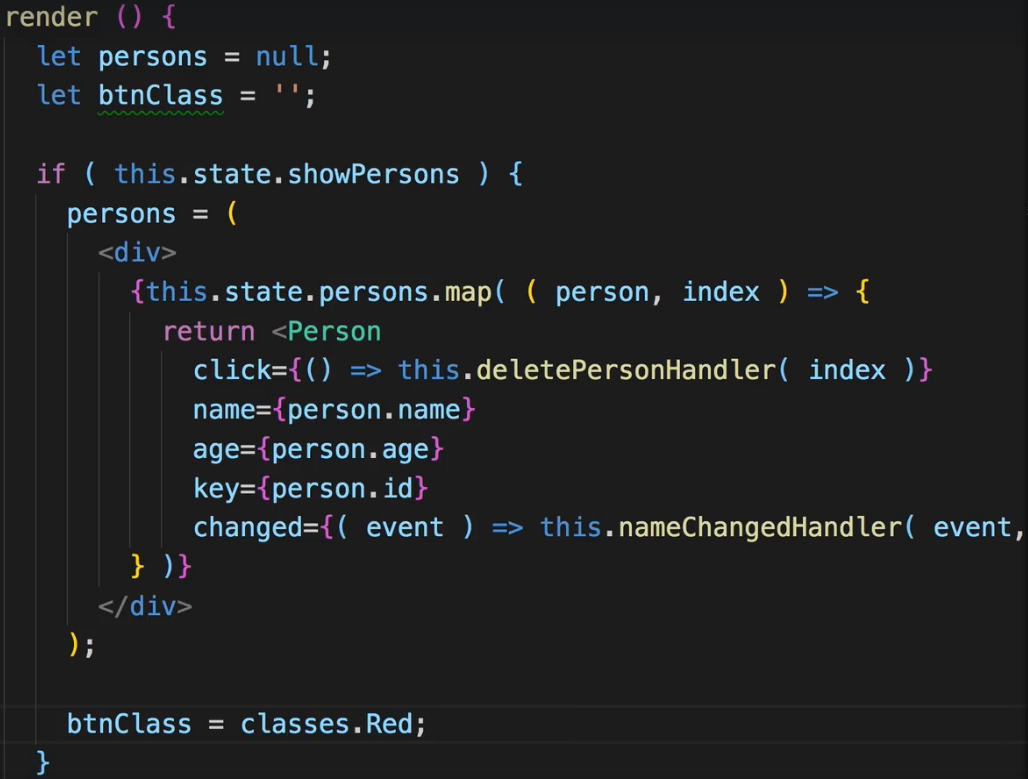
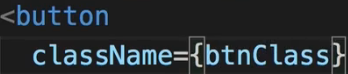
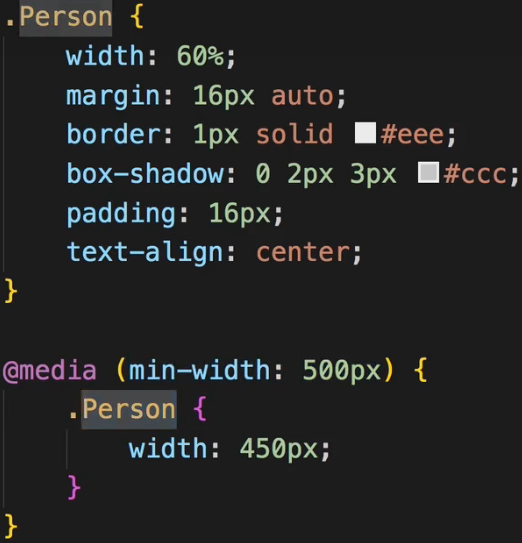
**Section 5 done: 11/11 Styling React Components & Elements**\* How you can dynamically adjust styles or classNames and how to work around this restriction of either using inline styles and hence having scoped styles but having limitations like not being able to use media queries or pseudo-selectors or using CSS files and then having global styles. There are a couple of ways of handling this and we’ll dive into all of these ways of working with styles in this module.  
**Setting Styles Dynamically**  
\* style={style} - this part between the {} can be any JavaScript expression. We should pass in an object with the styles or like in this case a variable that holds the object with the styles. And we could adjust this style variable dynamically therefore.  
  


\* We don’t assign it a new value so it’s still a constant but we assign a new value to one of its properties.  
=> And with this tiny change in place we already got dynamic styling.  
=> This shows 1 crucial thing which you really have to wrap your head around if you haven’t already: everything is JavaScript. You can manipulate the style object with any JavaScript code you want and in the end when you then bind it here, it will simply apply this - no matter how you edited it.  
\* What about classNames though? Can we also set these dynamically? Can we pass a list of classNames? Yes we can.  
**Setting Class Names Dynamically**  
\* Now adding some conditions  
  
  
\* We could’ve also used the .join prior to this and store it in a new variable.  
\* And of course you could have whatever logic you need - to construct an array of classes, or get a string with CSS classes by other means without joining an array - whatever you have, maybe take some user input which is a string by default. You can assign anything which is a valid string to className and it will then be handled as a list of CSS classes.  
**Adding and Using Radium - for pseudo-selectors**   
\* Let’s go back to our limitations such as not being able to assign a hover style for our button.  
\* Hover is a pseudo-selector which means it’s a selector based on some other selector indicated by that colon :  
\* We could use a CSS file but then it would not be scoped to this component only. The cool thing about using inline styles is the ability to edit them using JavaScript.  
\* So it would be nice if we could use pseudo-selectors and media queries in our normal JavaScript inline styles.  
=> We can add a Third-Party Package which is pretty popular, to add this functionality to our app.  
**npm install --save radium**  
=> I’ll add --save to also save an entry in package.json so that we also fix the version number and can easily share our project.  
  
  
=> This is called a **higher order component** - something we will also create later.  
=> It’s just a component wrapping our component, kind of injecting some extra functionality, in this case some extra syntax which will parse our styles and understand some extra features we can now start using. You can use it on both stateful and stateless components.  


  
\* You can also add media queries.  
**Using Radium for Media Queries**  
\* You can do it in a CSS file but you might need to use Radium if you want it to be scoped or if you want to change it dynamically.  
  
=> the StyleRoot is a component made available by Radium.  
=> Wrapping the export with Radium() is enough for pseudo-selectors.  
=> For basically transforming a selector or media queries or also animations with keyframes we need to wrap the entire application in a special component provided by Radium.  
\* We import a **named export**.  
  
  
**Enabling & Using CSS Modules**  
\* Wouldn’t it be great if a CSS file like Person.css could be scoped to the Person.js component file? So that whichever styles we define in Person.css could be imported and assigned to elements in this component and wouldn’t overwrite styles in other components or other parts of the app even if we were to share the names of the CSS classes.  
=> We can use a feature called CSS Modules.  
=> Now we want to handle everything through scoped CSS files.  
=> And to do that we actually need to tweak the build configuration of our project.  
=> I mentioned earlier that we’re using these `react-scripts` - that this is a package which exposes this whole build workflow to us and we can’t really edit the configuration. We could dig into the node\_modules folder, search for this package and edit it there but this is something you should never do because everytime you re-run npm install, your changes will be overwritten.  
=> There is a command which gives us access to the configuration - **eject**  
**npm run eject**  
=> To basically convert the “everything is managed for me project” to “everything is managed for me but I can edit the configuration of the project”.  
\* **One important note - if you’re using GIT to track your changes, make sure you add and commit all changes before you run the eject command - it will not work otherwise**.  
\* Now we see new folders: `**scripts**` and `**config**`  
  
\* This `scripts` folder has 1 script for each command we have had in the package.json file. Now we see all the development dependencies which are used under the hood. Babel for example.  
\* For us interesting is in the `config` folder the `**webpack.config.dev.js**` and `**webpack.config.prod.js**`  
webpack is basically the build, the bundling tool which gets used which applies all kinds of transformations and optimizations on our files and bundles them together. And as part of this process it also takes care of our CSS files. It’s webpack which is able to parse this CSS import in our JavaScript file.  
=> Webpack is the place where we can now also adjust the way we handle CSS files and where we can unlock this extra feature I was referring to. Using CSS Modules.  
=> In `webpack.config.dev.js` scroll down to `module` and then down to the module where you can spot this CSS extension here, where you test for CSS files. There we then use this `css-loader` and we will tweak the options here for the css-loader, we will set modules to true.  
=> `**css-loader**` is a tool which simply parses and converts our CSS, extracts it and stores it.  
=> The `localIdentName` - is important for your styles, your CSS classes getting unique names per component so that they don’t overwrite each other application-wide.  
  
=> Now copy that setup and go to the `webpack.config.prod.js ` - there you should also search for the `css-loader` and add it to the options.  
=> So that we use CSS Modules for both our development workflow and the production workflow, because we don’t wanna lose that feature when we ship the code on our real server.  
  
=> That’s everything we need.  
\* With that 1 important thing changed - when we import ‘./App.css’ it will now actually scope these CSS classes in this CSS file to this component where we import it.  
\* **We also need to tweak that import**. It will be a JavaScript object containing the CSS classes as properties.  
  
   
\* So it does what we did before but now using a totally different technique behind the scenes.   
\* Now what’s happening here is that the `css-loader` transforms these CSS class names we set up in the CSS file into unique ones. Using this localIdentName pattern we set up in the webpack config file. Where it essentially takes the class name we defined, the file name of the JavaScript file where you import the class and some random hash to generate a unique CSS class name. And then it simply stores all these unique class names it generated in this JavaScript object you import. And it stores them in the properties where the property names are your original CSS class names so the values of these properties like here for classes.App are simply the unique class names generated by `css-loader`.  
\* Of course we need to adjust our classes down there where we .join our strings.  
=> This .join can stay but up here we need to .push classes.something instead of the name itself.  
  
  
=> Referring to the CSS classes not as strings but as properties on this `classes` object which we’re importing from App.css.  
\* Now we got a name issue here because we have constant named `classes` here so we rename it.  
\* We need to restart our development server if we change the configuration as we did.  
\* Here you see the strange name.  
  
\* **It’s scoped to the component where we import it**. So the name will be different in a different component.  
\* Our Person.css classes aren’t working because the old import no longer works.  
\* Now let’s **make that Media Query and that :hover state work too with CSS Modules**.  
**More on CSS Modules**  
<https://www.udemy.com/react-the-complete-guide-incl-redux/learn/v4/t/lecture/8111608?start=0>   
<https://github.com/css-modules/css-modules>   
**Adding Pseudo Selectors using CSS Modules**  
  
=> The .App button already took effect because .App is used on the div and button is just a selector.   
\* Now all we need is to conditionally add the .Red class to this button if we need it. So we simply need to bind the class name of the button. So we will do this in the render() function of course.  
  
  
\* Now 1 important note: the btnClass always needs to be a string if we assign it to className. And the classes.Red in the end is just a string created by the `css-loader` which gives us still access to the class like this but in the end returns a string with the valid CSS. Or it is an empty string if we’re outside of this `if` condition.  
\* All the classes in there even if they only apply to a special sub-selector, are available on the `classes` object where importing from that CSS file.   
**Working with Media Queries in CSS Modules**  
\* Just like the pseudo-selectors you can simply define it in the CSS file and it works.  
  
\* Simply think of this `classes` object as a JavaScript object which gives you access to a string version of your CSS styles, of your CSS class but of a CSS class which was adjusted to be unique. So it will simply just translate your .Person class to something unique. So that you can be safe and sure that when you use that class in a given component, that’s the only place where the CSS styles apply. Nowhere else in your application can these styles have any effect. That’s the idea behind CSS Modules.  
**Resources:**  
Using CSS Modules in create-react-app Projects: <https://medium.com/nulogy/how-to-use-css-modules-with-create-react-app-9e44bec2b5c2>  
More information about CSS Modules: <https://github.com/css-modules/css-modules>

More on CSS Modules

Section 5, Lecture 67

**CSS Modules** are a relatively new concept (you can dive super-deep into them here: <https://github.com/css-modules/css-modules>). With CSS modules, you can write normal CSS code and make sure, that it only applies to a given component.

It's not using magic for that, instead it'll simply **automatically generate unique CSS class names** for you. And by importing a JS object and assigning classes from there, you use these dynamically generated, unique names. So the imported JS object simply exposes some properties which hold the generated CSS class names as values.

**Example:**

**In Post.css File**

1. .Post {
2. color: red;
3. }

**In Post Component File**

1. import classes from './Post.css';
3. const post = () => (
4. <div className={classes.Post}>...</div>
5. );

Here, classes.Post  refers to an automatically generated Post  property on the imported classes  object. That property will in the end simply hold a value like Post\_\_Post\_\_ah5\_1 .

So your .Post  class was automatically transformed to a different class (Post\_\_Post\_\_ah5\_1 ) which is unique across the application. You also can't use it accidentally in other components because you don't know the generated string! You can only access it through the classes  object. And if you import the CSS file (in the same way) in another component, the classes  object there will hold a Post  property which yields a **different**(!) CSS class name. Hence it's scoped to a given component.

By the way, if you somehow also want to define a global (i.e. un-transformed) CSS class in such a .css  file, you can prefix the selector with :global .

**Example:**

:global .Post { ... }

Now you can use className="Post"  anywhere in your app and receive that styling.