Styling React Components

by Jake Trent

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Learnin

Course Overview

Course Overview

Hello. I'm Jake Trent, and I'm very happy to welcome you to my course, Styling React Components. React is simply a fantastic library with which to program your UIs. When you're programming a user interface, you're concerned with everything that a user can see. So, obviously, you're going to want it to look awesome. In this course, we're going to explore together some of the most popular or promising techniques for creating great-looking styles and in ways that are a joy to code. We'll investigate the use of inline styles and try out a library for inline styles called Radium. Who would've thought that that was an option, right? But you'll be surprised the problems it solves. We'll have a look at how to write a more traditional CSS stylesheet. And, finally, I'll show you something that I bet you never thought you'd see in CSS, real isolated CSS Modules, no more global namespace for you. By the end of this course, you'll have some great tools for styling in your toolbox. Your next React project will look better than ever. Before beginning the course, you should be familiar with the React basics. If you know a touch of CSS, that will be helpful as well. Let's jump right in and try this out. These techniques will be fun to learn together, and I'm excited to share them with you.

UIs in React

Welcome and Prereqs

Welcome to Styling React Components with Jake Trent. Thank you so much for being here to learn this exciting subject with me. I've had great experiences using React, and I hope you do as well. React is a fantastic library for building rich user interfaces. It provides a great component abstraction for organizing your interfaces into well-functioning code, but what about the look of your application? In the browser, the look and feel is mostly determined by the use of CSS. This course will be focused on the strategies one can use to apply CSS to React-based applications and to great effect. The good news is that there are some fun and solid options for us to try together. Note that to be successful in this course, you don't need to be a whiz with layout or design. We'll be styling a fairly simple demo component together to try out some different styling options. In order to understand the demo code that we will work off of, however, at least a beginner's grasp of how to write React code will be helpful. A certain amount of CSS knowledge will be helpful as well, but since the level of CSS that we'll be using in the demo is very basic, this is less of a concern. So get ready to move beyond simply writing the internal gut severe React components. Paired with a knowledge of good styling options, you will have full control to create fantastic React-based applications. Let's get started.

Why Style?

The style of an application is usually not as important as the function of an application. If it works but does not look good, it still works. Presumably, you can still complete the task that an application should enable you to complete. If the application does not function but looks good, it's usually of little use to its essential purpose. But there is a value to having well-styled applications or a component. Proper styling can enhance and complement the other features of your application. It can provide clarity to the purpose of the application. Proper layout and readability can give a better picture of the task to be completed or the help being given by an application. Great styling can help guide users to be more effective and efficient. Ease of use will increase user satisfaction. Great design and styling can create a simply more pleasing experience. You will attract more users to your application if it is a joy to use, not just purely utilitarian. When you delight the eyes through a better experience, you'll bring people back for more. Great styling will help you promote your brand. What users see and experience in your application becomes a part of what they think of you and what you represent. The desirability of your brand will be upheld by a presentation of the obvious care and love that you put into the experiences you provide. There's also a cost for a lack of good styling. Remember that you don't want to be the app in which your users can't complete tasks, don't enjoy coming back, and from which they gain a bad taste of your brand. Styling will take time and effort, and it's a skill, an investment that will pay off. As an example, here's an unstyled version of the demo component that we'll be styling together. It's fully functional, or is it? It's hard to tell what it is or if what I perceive it to be is what's intended. It's intended to be an image slider or gallery component. When I press the navigation buttons, it's supposed to advance between images and show associated captions. But it's really unclear what image I'm on. I can't find the navigation easily, and it does not function to put many images onto a small consumable space in my application. We can do better by styling it, and we will.

Prepare for React Land

Now as we begin to style actual components in React, you will want to compare and contrast it with your other experiences in UI development. We may try some things that are unexpected. We may show some options that you may have seen a long time ago that are now making a resurgence in a slightly different format inside of React. Let's point out a few key differences between what you've probably observed in web development outside React and what things may feel like when we do our styling from the context of React. For instance, when coding HTML in the past, you've probably experienced having to write and think about the entire document of a page at a time. In the world of React, we will be working inside of well-encapsulated components. They may be at a higher low point in the page hierarchy, but it is great to be able to put structural and cognitive boundaries around single components at a time and focus on doing one thing well. Extending from and supporting this idea, we have the idea of modules. In current day browsers and in development experience that we've had in the past, there have been no native modules. Often code would live in a single namespace. When coding in React, we will take advantage of complementary tooling that will allow us to use modules by default. This will enhance our organization. And some of the styling strategies we explore will even experience this good code organization technique for our styles. It is so, so good. Generally, what you've been doing in the past in regular HTML land may feel like old hat by now. But be ready to possibly be surprised a bit by the strategies and practices that we explore in this comparatively newer and younger land of React. This community has a penchant for questioning the status quo. You may be led to a refreshing place where you question the best practices of the day and are led to think and craft your own solutions that will best fit the problems you face. So we're going to try a few things.

We'll solve the same problem in a couple different ways to give you taste of each option that we explore. Some may better suit your tastes and preferences. Of course, that's why there are multiple options. Your experiences and exposure may lead you to prefer one of these options over the other. Know that smart and able people have invented or are using all of these options that we'll see. There are also other options that we aren't covering, and, doubtless, many yet to be discovered options that will well be worth your while. We're having a look at some of the most popular and promising ways to style your React components today. After you try them all, I'm eager to see which you like best and what you learned from all the other options. Let's jump in and get started with the first one.

Inline Styles

Forget What You Know

In this section, we'll explore our first strategy for styling React components, inline styles. The very idea of this is repulsive to some people. The fact that we're talking about inline styles and that many people have found that they fit well in their React styling strategy is a testament to the willingness of the React community to question best practices. You probably haven't heard anyone suggest using inline styles for a long time. It hasn't been considered a best practice. Of course, some of the very foundational features of React fall into that same category. Who would have thought that you should put your entire view hierarchy into a render function and call that render function to re-paint your view every time data within the view changes? The idea is counterintuitive at first. One would probably assume that this would have terrible performance characteristics. In the case of React, this top-down rendering feature becomes one of its most powerful properties. I've been to numerous talks on React where a slide is presented cautioning you to forget all you know about blank, being React, CSS, etc., before this next slide. This may be one of those moments for you.

Current Origins of Inline Styles

The widespread adoption of inline CSS within the React community was mostly spurred on by a talk by Christopher Chedeau at NationJS 2014. In speaking with Christopher, he indicates that the talk was purely meant to entice people to think about the problem. He was not prescribing the use of inline styles for everyone or every problem. In his talk, Christopher outlines seven types of problems that he and his teams at Facebook faced in styling with vanilla CSS. His framing of these issues has led him to find solutions and has influenced others as they have formed their own independent solutions. His outline will make it clear what issues with vanilla CSS you may run into. In CSS, everything is global by default. This makes it hard to encapsulate any styles that should be local to some specific UI. At Facebook, they made a custom CSS language extension and tooling around it that handle local versus public CSS selectors. They essentially made a new CSS language, not something that everyone will want to do. As of this writing, there is no current CSS language feature that lets a programmer describe what styles are depended on in his stylesheet. Dependencies are not explicitly stated. Facebook's answer was to create a class name's function that was wrapping the usage of any CSS selector in their code. This was the only way to generate the correct class name because of their local selector extensions. It also served the purpose of requiring the stylesheet for you allowing analysis of what programs depended on what styles. Without specific mapping between application code and styles used, it is not possible to confidently remove unused styles in a final stylesheet build in order to optimize performance. Again, Facebook solution was that of the classname function, allowing usage of selectors and CSS to become clear, and unused selectors could be removed. In an effort to shrink the size of stylesheet payloads for clients, minification is desired. This is possible in vanilla CSS to a point but became easier with the use of the Facebook classname's function. This is because it was used in all selector references. Thus, new smaller selector names could be generated for everything. It is usually hard to share variables between JavaScript and CSS. This is usually done via comments only. Facebook created its own functions to make variables from a single data source available to both JavaScript and CSS. Non-deterministic means that for the same input, the output might be different. It's unpredictable. CSS is built such that all the selectors are in the global namespace. The cascade of CSS will make the last matching selector win. If separate stylesheets are generated with asynchronous loading, styles can change simply when new styles show up in the global namespace. Facebook didn't easily resolve this at the time. It's easy to write global selectors that modify a UI component's look and feel. At times, this may feel very convenient, but it violates any public API contracts that the component may have. Now the outside client depends on the internals of the component being a certain way. Again, this is not easily solvable in traditional CSS. So, overall, Chris observes that Facebook compiled a bunch of hacks to solve most of these problems in CSS. It led him to believe that there might be a good reason to defy the conventional wisdom around writing CSS. Perhaps they would be better able to solve more of their problems and, hopefully, in a better way. Let's do some inline styling and see if we agree.

Demo Introduction

Let's introduce you to the demo project that we'll code together. It's an image slider or a gallery component. It's interesting enough that we can contrast the differences between each styling method. It's small enough that we'll be able to fully style it multiple times using the different methods. The starting point for the code will be a fully functional React component. The HTML markup via JSX is already there. All the logic is already there. It's up to us to layer in the styles, which are all absent to start. In its starting form, it looks like this, clearly unstyled. We're trying to apply styles to the components so it'll eventually look like this. Much better, right? Feel free to make your own variations on the style choices as we go through this together. In the code that is prewritten, you'll see a number of ES.next features such as classes, module imports, decorators, and more. These new language features are used in the source code and compiled into browser-ready code via the awesome Babel compiler. These are interesting, and I encourage you to learn more in other courses on how you might use them to your benefit, but don't get too hung up here on the syntax you may be unfamiliar with. We'll explore anything that's essential to our learning styling. This project's assets JavaScript and CSS will be built using a great tool called Webpack. For the first couple styling strategies, Webpack will be doing work for the JavaScript only. Since this is not the focus of the course, we'll learn some basic commands to get it up and running and trust that it's doing its job. For a couple of the later strategies, we'll have a quick introduction to some of the particulars of Webpack because at that point, Webpack will actually help us build our stylesheets. It's a fun little component in all. Please have a look through the internals and let it peak your interest.

Inline Styles Demo: Install

Let's get our environment in a state where we can run this demo application. First, you'll need Node.js and npm installed on your machine. We've identified a number of libraries, such as React, and tools, such as Webpack, that we're going to need to make this project run. Once you've downloaded the project demo available in the course material or online via GitHub, open the shell and cd into the project directory and install the third-party libraries via npm install. This will take a moment and must be completed successfully in order for us to be able to run our application. After install, you can run the app by typing npm start. Then you should see an indication that your app is available to view in the browser. By default, I have the app set to run on localhost port 3000. Address that in your browser, and you should be golden. At this point, you should see the unstyled version of our component where there are five images displayed with text captions beneath each.

Inline Styles Demo: Styling

Now our app is running and we're ready to start styling our components. We're going to start at the root or top of our view hierarchy and work our way down to more and more detailed components. There are five components in all that work together to make this thing work, so it'll go pretty quickly. Note that when coding inline styles, all our styles are actually written in JavaScript Also notice that we'll usually have a one-to-one mapping from our component's module file to that module's stylesheet file usually named in the format of component-- styles.js. Here are the files for the demo project. All the files that we'll be editing are here in the source directory. There you'll find the images shown in the image slider and all the functioning React components. In the index.js file, you'll find the starting point for our application. Here we call React's render function to attach our application to the DOM. The DriftApp component is rendered. It's found in app.js. This is the brain of our little component. In its render function, you'll see the view hierarchy including all subcomponents that we'll be styling. Here we have a hardcoded value in the JavaScript code that's a magic number describing the image width. Already we want to share style constants between our code and styles. So let's make some config-styles that we'll use in multiple components. Notice the --styles suffix in this JavaScript file name. That will be a convention that we use throughout the project. It is also a module that is exporting a single object. Within the object, we specify two dimension variables for how large the slider will appear. Now we want to use those config-styles in App.js. We can reference the attributes on the object and must remember to import the module in order to use it. Checking the application in the browser, it looks the same as we'd expect, and there are no new JavaScript errors, thank goodness. Let's go to the next component in the tree view. This is the frame component. If you ever need to orient yourself on the hierarchy of the view, look at app.js again. Let's make another JavaScript-based styles module for the frame component specifically. First, import the config-styles we made earlier. We'll use the root name to describe the topmost container in each component. Remember, this is not the root of the whole app per se. We're choosing names within the boundaries of this single component. Now that we finished the stylesheet, let us look at the format of these inline styles. The module exports a single object. Within this object, there may be several attributes which represent the elements within this component that will be styled. The value of each is another object. This object is what you put into the style prop as React requires. Within the object, you will see CSS attributes. These are all in CamelCase. This is different than normal CSS because in JavaScript objects, dashes in key names are harder to deal with and must be quoted. Instead, we use CamelCase only. The values, again, because this is JavaScript must be JavaScript datatypes. You'll see both strings, which are quoted, and numbers, which don't have to be. Numbers for attributes like width will be interpreted as pixel values. Notice as well that we're using a JavaScript variable from the config-styles module as the value for width. It's just code after all. Now in order to use the styles module, it's imported into the frame module, and we can find the root attribute hanging off the styles object. Previewing the app, we see a slight change. Now that frame has had margin 0 auto applied to it, the widget is in the center of the page. Onto the next component with the carousel. This component wraps the slides and navigation that we'll see next. We'll need a new style module to go with it. Follow our normal naming convention here. Again, we'll need the config-styles, and as usual, we'll have a single object exported. Here, again, there is one element, root, that we will style. When we use it back in the carousel component in the style prop, it's already looking better and more compact. Let's go right on to the slide component. Again make a one-to-one style module. There are a few more elements in this component to style, so we'll list them all here, root, footer, and title. Without entering the CSS yet, let's go back and wire up the style props. The first is the trickiest. Notice that in order to make sliding work, I've already anticipated needing props.style on the root article element. But now that we're making a stylesheet that needs to also provide inline styles, we need to merge these two objects. We could use something like Lodash extend or object design, but since we have the Babel compiler, we can confidently use many of the ES.next features such as the spread operator seen as a triple dot before your variable. The spread operator essentially de-structures all the attributes from the object in the variable into a new object, in this case, the object that we pass to the style prop. If we spread the style's root object in there as well, all the attributes from each variable will end up in the new object merging them together. Here, order matters, and that's why I'm letting props.style go last. Finally, specify the footer and title styles. Now back in slide- styles.js, we'll get the slides ready to stack on top of each other and have a proper-looking caption. Here note that we can create all the JavaScript code we need, in this case, just a variable within our styles module that we will need in order to generate the correct styles. Note that as I save the styles file periodically, the browser on the right is updating, looking better all the time. These tools really help style and experiment faster. (Working) Let's move to the nav.js file. This is going to be the last stub component that we style. It's also going to have a few fun variations of its own. Let's start as before and make a nav- styles.js module. Note that we're still making our dependency on config-styles explicit. Let's do as before and write out the keys of each element will style, and then reference them in the component's style props. Lastly, let's define the look of the navigation in a few CSS attributes. In the case of the prev and next buttons, I sense that they're both just buttons. Perhaps there's some common styling that we could store in a single variable. Once that's established, let's use the button variable in both the prev and next. Let's move the declaration for both of these elements up to make some changes. But make sure to leave them in the exported object. In ES.next, when we specify just a key with no value, the variable name becomes the key, and the variable value becomes the value. It's a convenient shorthand that helps us not have to write something like next: next. Let's spread button into each and then apply the variations that set each apart. Hey, it looks pretty good. Check out the nav arrows in our running app. Now there's a bit more we could do to make the nav component's styles match the level of functionality that's already there. What we want to do is hide the arrow buttons if there are no slides to advance to. In order to do that, we'll have to put in some logic to change the inline styles based on that state. The hasPrevious and hasNext variables in the nav component should help us have the flag we need to switch on. By writing a couple of functions that look at these values, we can switch easily between providing the component with the normal styles or the hidden styles in the case of no arrow for navigation being needed. Now we need to make those hidden styles that we were referencing. Much like the previous example of using the spread operator, we'll compile CSS attributes from different objects to make the combination we want. To make the prevHidden class, we'll combine the prev and the hidden attributes. For the nextHidden class, we'll combine the next and the prev attributes. Looks like it's working just fine.

Evaluate Inline Styles

Now you've experienced styling a component fully with inline styles. Do you feel like it helps solve the problems that you've possibly felt and that Christopher Chedeau outlined in his talk? Let's make a quick mental checklist. Global styles? Definitely not. Everything is a local style object. Dependencies explicit? Yes, very clearly we set our dependencies using imports for each. This makes it statically analyzable. Dead code elimination? Yes, it's possible. All usages are explicitly coded. Minification possible? Yes, it's just JavaScript, and there are no selector names here. Constant shareable? Yes, they're all JavaScript variables. Will resolutions be predictable? Yes, as long as new stylesheets don't attach to the global namespace. Isolated? Yes, no component styling influences any other component styling. Inline styles look to have solved this very well for us. There are some things that are harder for us to accomplish in inline styles. Since everything is specified in JavaScript, the media query syntax from CSS is unavailable. You can use the matchMedia API from JavaScript instead. It doesn't feel quite as convenient. Since all styles are inlines using the style attribute of HTML, there isn't native support for pseudo-selectors either. Things like hover or last child are left to be handled in JavaScript proper, and that's definitely more work to maintain logic for hover in something like React's state. Finally, there's really not a good option for reusable animations like those available using CSS keyframe animations when we're using inline styles. But, overall, it feels pretty good. A lot of people are using inline styles to great effect. It's very common to see this in React component libraries. Now let's look at the next styling strategy together from a library called Radium and see how it compares to inline styles and what it adds to the equation.

Radium

Radium Intro

Having just seen the inline style strategy, you'll feel quite comfortable with Radium. Radium is a third-party library developed by Formidable Labs that is essentially inlineStyles++. It gives you everything you had previously and more. There are several other libraries in this category. Where mediaQueries, pseudoSelectors, and keyframe animations were a no-go in your JavaScript with regular inline styles, Radium gives you an easy-to-use JavaScript API to make those happen. Radium provides a React component that wraps all your Radium-styled components. This component takes care of interesting things like tracking hover states, etc. Beyond that, the definition of styles will feel very close to inline styles proper. Let's jump right in.

Radium Demo

We'll start fresh at the unstyled point in our app that we started with at the beginning of the last section. We'll code the short exercise from the beginning. If you'd rather do a copy/paste from your similar previous work, feel free. Since we're using a new library to help us style, we'll first need to install it. In your shell, run npm install radium. If you've skipped previous sections and need to install all the other third-party dependencies, make sure that you run a full npm install as well. Once installed, we're ready to code our styles again. Make sure you start your app with npm start, and let's get started. As before, we'll work from the outside in styling the most generic components first and moving to the most specific. The first most generic bit of styling that we'll have to do is establishing some configuration styles that we'll reuse several times to control dimensions of the slider. Back in app.js, let's use one of our style constants within the component. Sharing constants has never been so easy. Let's make some application-level styles. These are actually global styles, but we're going to write them in JavaScript. To get global styles on the page, Radium defines a Style React component, and we need to import that and feed it our app-level stylesheet. It feels like inline styles, but it's not. Sometimes that's still useful. In this case, we've decided that we want to establish some things globally, such as boxSizing and fonts. When we preview our application, the global styles have been applied. We have a rich dark background, and our fonts have been updated, though they're hard to see at this point. Let's go on to style our first subcomponent, Frame. The first difference you'll notice when styling with Radium is that we import Radium into every component module that we style, and we wrap our component in the higher order Radium component. This Radium component keeps track of things that inline styles proper don't, such as hover states and media queries. Beyond that, however, the usage patterns for your inline stylesheets will feel very similar from the component side. Just pull attributes off your styles object and apply them to the appropriate elements in your component via the style prop. Note that the format of the actual style modules are just the same as the inline styles. They're just JavaScript files. We're naming them with the --style suffix as a project convention. They export a single object. Within that object are attributes, in this case root, that will map back to the elements that we're trying to style in our components. In the next component, the carousel, we'll import Radium as before. This is usually the first thing we do. Then we import our stylesheet. If it doesn't exist, as it doesn't now, and I save, there, we get a No such file error shown. No problem there. Let's just create it, and we're on our way again. Again, in this component, there is just a root element that will get our style with the styles object. Voilà! What used to be a list suddenly looks like a single image. Now let's get the slide component ready to look its part as we start interacting with the slider. Import Radium as always. Then create a new styles module. We'll list out all the elements that we'll style and the exported object first, leaving the actual CSS for a later moment. Back in the component, we make sure to wrap our exported component in the Radium component. Now we have another difference in the Radium API versus regular inline styles. When mixing multiple style objects together, where we previously merged objects into a single style object via the spread operator, here Radium just wants an array of style objects. But, again, order matters, and the most specific style object should come last. Finally, map your style object onto each element to receive the styles. Now we're ready to hammer out some CSS. These are all the bits that will help our slide look a little better. (Working) Now to our most interesting component to style, the nav component. Let's create our nav styles module and establish the three main elements that we'll style--root, prev, and next buttons. Remember as before, we're going to put some smarts into the nav styles where, if there are slides to advance to, we will show the appropriate nav arrows. If there are not, they'll be hidden. To accomplish this, we'll write two functions that control this logic based on the hasPrevious and hasNext props. Oh, and, again, don't forget to wrap your component in the Radium function call. This will be essential for later features on this component. Back in the stylesheet, we realize that prev and next are just buttons, so we want to make a common button style object to spread into each prev and next variable.

You'll note that at this point, the next button looks pretty good in the browser preview, but the prev button looks terrible. This is because the prev button is currently looking for the prevHidden style object, which we haven't created, so it's getting some of the browser default styles. Let's fix that by coding the hidden states making it look much better. Looks like it works as designed. Now let's pull some more Radium goodness out. Pseudo-selectors are easy with Radium. So let's put a hover state on our nav buttons indicating they're ready use. Simply nest a :hover selector in the style object you want a hover state on. In this case, all nav buttons should hover. At first glance, it doesn't seem to work. When we pull up the browser console and Chrome dev tools, in this case using Cmd+Option+I, we see this error, Radium requires each element with interaction styles to have a unique key, set using either the ref or key prop. Well, we can do that much. Back to the nav component. Let's put a key prop on each of those now-hoverable buttons uniquely identifying them for Radium. Works like a charm. Let's try out one more feature that Radium gives us here. There's not a super-strong case for a keyframe animation, but let's try it to see how it works. Back in her nav styles, the first thing we'll do is import Radium. Radium has a JavaScript API for keyframes. It looks and feels very similar to the native CSS API for keyframes. The second parameter to keyframes is optional, purely for better debugging messages. Once we adjust the keyframe to our liking, it's time to apply it as an animation to our button. When applying the animation attribute, it will be a little bit different than the native CSS attribute. Instead of the animation name, we will apply a placeholder so that, separately, we might specify animation name to this pulse variable. Saving that, we see an error in the browser. To use plug-ins requiring addCSS, please wrap your application in the StyleRoot component. StyleRoot is the component that Radium will use to collect all of its inserted styles. At this point using this feature, it is required. Back to app.js, we can modify our render to include StyleRoot at the root, and we are golden.

Radium Evaluation

And there you have it. Radium felt quite nice. If you look at our original problem checklist, it checks the boxes just as inline styles did before it. Global styles are not the default. Most everything is local. We did observe that where we wanted to break into global styles, we had a JavaScript API to do so. When you use Radium, spend a little time in the app-level styles file and you'll be fine. All dependencies were explicitly imported, thus, dead code elimination is possible. Since it's just JavaScript, minification is possible. We were able to share constants using JavaScript variables. Resolution is predictable if we continue to eschew global styles. And each component is self-contained. Beautiful! Even better than inline styles proper, Radium gave us hover state and keyframes very easily. It can provide media queries in the same way. I'm glad that Radium takes care of all that. Now let's look forward to the next sections. We'll be looking at styling strategies that may fill a little bit more familiar. Anyway, they're not inline styles. But be ready for some powerful variations on current day best practices. Let's go!

A Webpack Intro for CSS

Some Webpack Config Highlights

For our next couple of styling strategies, we're going to need to adjust our Webpack configuration. So let's learn a few of the basics about this powerful tool. Webpack specializes in asset building. In our app so far, it has been in charge of compiling our JavaScript. In a moment, we're going to teach it how to compile our stylesheets as well. The details controlling how Webpack works are stored in a file called webpack.config. There's no configuration included in the demo project for our production build. Though the difference to make that happen would not be large, that's a course for another day. The webpack.config is itself a JavaScript module that exports a single object. Let's take a look at a couple key attributes in this configuration object. In the config, the entry attribute will contain one or more entry points to your app. These are the places the execution of your program begins. From these starting points, modules are imported defining dependencies. When building your application's code bundle, Webpack will crawl from your entry points following each dependency until all modules are included in the bundle. Webpack is module-centric. It loves modules, and so do I. So you've probably noticed, and it will continue to become more apparent, that every piece of code we need is written as a module and then imported into our application. We will do this with our CSS-based stylesheets in the coming sections. When bundles are produced, they are put in the place specified by the config's output attribute. Path and filename indicate where the bundle will live on the file system. PublicPath indicates the path that should be used within your code when referencing the bundle, as in a script import on an HTML page. If you check out the index.html page in our project root directory, you'll see that the script tag source matches the output parameters of the webpack.config. Now the real magic of Webpack is the module loaders. In this part of the config, Webpack is told to be aware of certain modules that you'll be writing in your project. Usually module types are determined by their file extensions. In this project's webpack.config, Webpack is already knowledgeable about how to handle two types of modules, .js files and .jpg files, one for

JavaScript, the other for images. Now image files aren't usually interpreted as modules, but to Webpack everything will feel like a module. What Webpack does with each module type is specified by your loader's attribute. For instance, we are currently telling Webpack that all .js files will be loaded by the babel-loader. If you check in our package.json file, this is another third-party plug-in that we're using to make Webpack work. Likewise, the file-loader is used for getting images into the app. Again, everything's a module. If you go into our app source code to the app.js file, you'll find that the image property supplied to the slide component is actually an import of a module that ends in .jpg. Thus, in that case, the file-loader is used to make that module available to the app. You can do powerful things with this module abstraction. Loaders function almost like UNIX pipes. The loader provides a transform function that expects one form of input and always gives one form of output. Because of this, loaders are usually very specific and have a fairly thin transform, thus are easily composable together to do really interesting things. We'll demonstrate this when we specify how to load our CSS in our webpack.config in a moment. There are many other options and plug-ins to allow Webpack to do other interesting and helpful things for your applications. I encourage you to seek out the Webpack documentation and experiment. In this, as in all things, make sure that you commit good working baselines of your webpack.config to source control before making new experiments on your configuration. You can make very slight changes that change how code is loaded or not in your application very easily. This webpack.config .js is used by Webpack-develop-server, which we start at a local web server for development. This server is run when we run npm start. This brief overview should allow you some basic information on the highlights of what will allow Webpack to support your impending use of CSS stylesheets.

Adjust Webpack for CSS

Now in preparation for the next two styling strategy sections, let's adjust our webpack.config. There are a number of ways to get CSS stylesheets on the page via Webpack. In this case, we'll choose the easiest option, that is, the style-loader. The style-loader generates code that takes CSS input and appends style element children into the actual DOM of a web page. How awesome is that? You can write JavaScript all day living in the land of modules, yet what our app needs ends up on the HTML page consumable by the browser. Let's install the style-loader. Go to your shell, and type npm install style-loader. While we're here, let's grab two additional loaders we'll use later, and I'll explain them in a moment. So npm install css-loader and postcss-loader as well. When specifying your loader chain, order matters. The first or leftmost loader is the last in the chain. So this style-loader is the loader that will put our CSS on the web page. But where does the

CSS come from? Well, we'll need to add the css-loader that will grab CSS as it normally appears in the stylesheet and prep it for the style-loader. But we're going to add one more loader still. Finally, let's add this postcss-loader as the initial loader in our chain. We're going to be writing our CSS in a format that's currently called CSSNext. At this point, that represents a flavor of CSS that is very close to the upcoming css-form. PostCSS will know how to interpret and transform those new CSS features. We can load various PostCSS plug-ins much like we've done with loaders in Webpack that will be utilized by the postcss-loader. We'll do that now and install postcss-import and postcss-preset-env. Postcss-import will support resolving local CSS file imports. Postcss-preset-env will allow us to consume various CSSNext features such as custom variables and selector composition. With these installed, let's specify each to be used by PostCSS in a new file in the project root called postcss.config. Now to apply the concepts we've learned about how Webpack loads CSS files, we are going to change the webpack.config. js file. In the module rules section, we'll add a new rule. This rule will test for the existence of .css files. We will apply a set of loaders to those files, and we will apply this test to all files that are included in this project. The loaders will follow the order that we have specified, style, css, and postcss. The postcss-loader will use the postcss.config file that we have just created. At this point, we want to allow Webpack to pick up this new config and use it as we move into the next sections. To do so, we stop and restart our Webpack by pressing Ctrl+C in the shell in which we had previously run npm start. Restart, and we are ready for a couple of great styling sections to come. Here is to bringing on the CSS Modules. I can't wait for that section.

CSS Stylesheet

CSS Stylesheet Demo Intro

Now we will explore what it feels like to style a React component with external CSS stylesheets. Of this handful of options that we're trying out, this likely will be the most familiar feeling possibly from your days before React. We do have the added benefit of Webpack in this case. So perhaps the mechanic for getting styles on the page will be slightly different. For instance, we'll import CSS files using the same syntax as we import JavaScript files. That's pretty cool! But inside the stylesheet file, the CSS will feel just like regular CSS because it is. That makes this option not the most progressive or noteworthy of the bunch, but often a welcome addition to the toolbox, if nothing else, because of its ubiquity. Let's jump right in. In the previous section, we already made the requisite modifications to our webpack.config, so we should be ready to start styling. Let's double-check by noting that our package.json includes all the CSS related loaders that our Webpack will need. And in the webpack.config, they're set up to use on all of our .css files. Again, when we import a file with the extension of .css, Webpack will run our original CSS stylesheets through a set of transform functions called Webpack loaders so that they're eventually built and available to our app running in the browser. Let's begin styling.

CSS Stylesheet Demo

We're going to start with an unstyled image slider component again. This time we'll write all the styles in vanilla CSS with one new difference being that Webpack will take care of getting our stylesheet on the page. First, we think about some of the configuration styles like those we've established in previous sections. Let's specify the height and width of our slider images. Now, however, unlike in the inline style scenarios we've covered previously, there's no native way to get our CSS variables available in JavaScript. So, we end up doing what many developers do. If we want to share constants like this for styling purposes, we'll just redefine it on the JavaScript side. Here I define imageWidth, and I'm just going to have to make another variable to show some hint of what this magic number means. Now I have to remember to manually keep the stylesheet and the code with the style information in it in sync if this ever changes. Now let's write some base styles for our application. We need to make the all-important link from our JavaScript code to the styles used. We import a file with a .css extension. This may seem quite odd at first. Why would we have our JavaScript code import CSS stylesheets? We need to make our CSS available for Webpack to build into our final application bundle. Thus, from our entry point, source index.js, Webpack will crawl all dependencies including CSS files. We need to make this import explicit. We're going to create our index.css file in the form of a vanilla CSS file. We're going to use the @import style imports from within the CSS file to create some file boundaries while we're working. First, we'll import config.css. There are a few important points to note. One, these aren't actual modules. We're essentially concatting files when we use the CSS feature. The other point is that the import still needs to be followable by Webpack. Thus, we use a. / to prefix the import path because it's a relative file path that Webpack needs to follow. For the same reason, we're always going to include the .css file extension. Let's also make a file for the global styles called base.css. The name is just part of a convention drawn from Harry Roberts' concept of the Inverted Triangle CSS. Here let's lay down a few global styles that will help us create the look we want on the whole page. It's worth noting that these kind of selectors should only be included if you're the application author. If you're just the author a single React component, especially one that might not be reused in other projects, it would be irresponsible to style HTML or star in your component styles. But in the right situations, this is how you do it. And for now, this is a good example of how we would break out into global styles. Now we can start styling the first component of our slider, specifically, the frame. Now we have to choose a class name. We're not in the land of inline styles anymore. We're not really in the land of isolated component styles anymore. We're stepping back into the world where our styles live together in a global namespace. We get a lot of our old problems back. Thus, we will find many of our old solutions applicable. BEM, or block, element, modifier, is a selector naming convention that is popular for organizing vanilla CSS. By creating block, element, modifier hierarchies, we create useful categorizations for our selectors increasing the organization of our styles. Since we're working in a global namespace, we also want to do no harm to the other styles that might live around us. Thus manual namespacing of selectors can be helpful. In our case, we'll prefix our selectors with the .dft__ representing the project codename React drift. Once we have the class name specified, we can go on to write the actual styles. Finally, don't forget to import frame.css from the index.css file so that the styles actually get included on the page. As we preview our changes in the browser, it looks like things are getting picked up nicely. Going on to the next component, we'll follow the same process. We'll choose a namespace to select our name. We'll create the new CSS file. Here we'll create a few attributes for the selector. Here note a difference from before. We're able to use the imageHeight variable without explicitly importing config.css. That's because we're not really functioning in modules at this point. Of course, Webpack deals wholly in modules, so we import index.css, but within that file, we're just doing CSS imports. The contents of the module essentially represent what you've probably used in the past, a single giant stylesheet in a single giant namespace. Since all the .css files are essentially concatted together, we can use selectors or variables defined earlier in the file. Include carousel.css in the index.css file, and the changes take effect. Let's go on to the slide component. Here we get more of a test of BEM as the selector names start to describe parent-child hierarchy. Selector names that live in a global namespace tend to get rather long. Here in the slide.css file, we'll define another variable. Note that variables go in the :root block, and variable names are prefixed with a --. As we create the footer selector, we can use footerHeight variable, as well as the imageHeight variable defined earlier in our stylesheet in config.css. As I make the final saves here, the changes look great. For the nav component, let's start with the most basic styling. We'll choose our selector names and then create a new nav.css file. Within this file, let's code out the styles that will allow the buttons to appear on top of the slider. Now let's add an additional selector for the previous and next buttons. Here let's use a tried-and-true strategy and compose multiple selectors in a single classname prop, and then create the corresponding styles. Now as in previous sections, let's make the nav button arrows appear only when there are slides to advance to. Let's make two functions to encapsulate that logic. Let's re-factor it to return our original class names from these functions. Save, and it still works. Now let's make this a bit fancier. For taking care of multiple class names conditionally, we're going to go to our shell and npm install classnames. Rerun npm start if you'd killed it in order to do the install, and let's jump back into the code. Let's import the classnames module and wrap our previous button ClassName string in the classnames function. When passed as a string, these classes will always appear in the resulting ClassName string. Then as a second argument, let's pass an object where the keys are the class names, and the value is a Boolean that determines whether or not to include that class name in the result. This is just a cleaner way to do dynamic class names, then string concatenation, handling spaces between selectors and that sort of thing. Finally, let's import the hidden class, save, and looks like it works. Now just to round out the demonstration and compare it to others, let's implement a hover state on our nav buttons. This is as vanilla CSS as it comes, so all the syntax you're used to is applicable here. Nice gray hovers. And then let's put our keyframe animation in for pulsing nav buttons as well. These scale transformations will cause the buttons to grow, then contract. As we specify the use of the animation on the button selector, it'll loop the animation infinitely. Alright, animations!

Evaluate CSS Stylesheet

If you've done any styling recently, that section probably felt really familiar. It was really just an okay styling technique, good for a tight spot if you had to. It really suffers from a flavor of all the seven original problems posed by Christopher Chedeau. Global styles? Yes, it feels terrible, doesn't it? We have to consider the world that our app will live in and try to do no harm by prefixing our selectors with some project-specific string and hope that other application developers have done the same. Dependencies explicit? No, the dependencies are not all explicit. Yes, we import a module, but the boundaries within the module are hazy. Different files actually don't represent individual modules with exported values. Dead code elimination? It could be attempted, but there is not a strong programmatic use of our selectors, again, because there is no explicit export from CSS, then an import and usage in JS. Is minification possible? Just a bit, mostly just whitespace removal. Selector names and CSS attributes must remain unchanged. Are constants shareable? No, not easily. Out come the code comments with us as developers on the hook for keeping the two sides in sync. Will resolution be predictable? No, if other components or applications on the page asynchronously load new styles, our component styling could be affected in unpredictable ways that are hard to defend from. Isolated? No, we saw that our style definitions were not contained within real component-centric boundaries. Now don't be sad.

Really, this failure and this feeling are really just prepping us for the glorious technology ahead. Get ready for an option that will challenge what you thought CSS could do--true CSS Modules.

CSS Modules

A CSS Loader Adjustment

We've just finished a section where we implemented the tried-and-true external CSS stylesheet method of styling. It was slightly different with one module imported so that Webpack could consume it and make it available for our application. But now we're going to embrace true modularity in the form of CSS Modules. We'll get into how this works. For the moment, let's make an adjustment to our webpack.config in order to support this awesome feature. The css-loader is the piece of code that knows how to interpret CSS Modules. In order to activate this feature, we must adjust our loader. It needs additional options passed to it. We'll convert it from a string to an object. Then we add a new options property. It importantly includes the modules: true flag. This will tell the css-loader to output CSS Modules. We specify another more cryptic property as well, localIdentName. Add this code verbatim. It is weird, and I'll explain it in a moment. Finally, the option importLoaders will tell the loader how many loaders before css-loader should be run to transform CSS imported using the @import syntax in our stylesheets. Because there is just postcss-loader before css-loader, the number is 1. Our final adjustment is to change the test for the loader. We want to load CSS modules for files that follow the naming convention .module .css. The previous test would still have worked, but we're making this adjustment to point out that React CSS Modules community has converged on this file naming convention to denote CSS Modules. If you get used to naming your files in this manner, you'll even get automatic CSS module support in some tooling such as Create React App. Also, in preparation for next steps, we'll install another third-party library called react-styleable by going to our shell and running an npm install react-styleable. Finally, remember that in order to update your running apps with the latest webpack.config, you'll need to stop and restart your Webpack process.

How CSS Modules Work

CSS Modules are exciting, at least they make me really excited. They have changed the way that I write styles and I think for the better. This strategy feels like a great mix of the worlds of inline and external CSS. I get much of the local modular feeling that used to be only available when inlining, and I get to write my styles using the CSS language. Using the CSS syntax means that everything it could do before it can still do. Pseudo-selectors work as expected. So do media queries and keyframe animation definitions. There are no new APIs for these things. If we wanted, we could use SASS, or LESS, or another style-specific language using this method as well. And yet we get real modules with real isolation. In addition, powerful adjacent features around the CSS language are available to us in our toolchain. We can mix in tools like PostCSS, which can provide a whole community of features like autoprefixer for automatic vendor prefix application, nested selectors, and other things that don't exist in CSS proper or in CSSNext. So how do CSS Modules work? Similar to JavaScript modules, the exports object is where the magic happens. I write a CSS file using the syntax I usually would. In it, there are one or many selectors. At build time, our build tool, in this case Webpack's css-loader, will generate a hash for each of these selectors known in our webpack.config as the localIdentName. This hashed selector becomes unique on the page. Because it's unique per module, that means that no other module even if it uses the same selector name will have the same hashed selector name. Thus, when Webpack puts all the styles in the resulting stylesheet, there are multiple modules simulated within the stylesheet. Then as a React component author, when you import the CSS module, what you really import is the CSS exports object. This object contains each of your original selector names as keys in the object. The values are the associated hashed selectors. Then when specifying your className attribute, you use the exports object key, but the value that is substituted after build time and at runtime will be the hashed selector. It is a simple and powerful mechanism. There are a few levers you can pull to customize the implementation of this concept. One of them is by use of the localIdentName parameter in the css-loader. This controls what the hashed selector looks like. When adjusting this, we're usually balancing readability in a local dev environment with size optimization in a prod environment. If you desire, you could make your webpack.config work differently based on something like the NODE¬_ENV variable value. Another great feature of CSS Modules is that composing styles now happens in the stylesheets themselves. This is where style definition belongs, so it feels very natural. Since we have real modules, now explicit dependencies can be made even at a selector level. For instance, if there is a base style for a selector such as .link, it may contain certain style attributes, but there may be components in which a link component with the .link class name is used, but the style should look differently. That specific usage can define its own style attributes and use the composes attribute to effectively import the original styles as a starting point in this new selector. From there, due to the cascade, overrides or just new attributes can be added. The fact that these things are possible and that the tooling exists to support this is very exciting. I think you'll be happy and surprised at how you're able to think about styles using this strategy.

react-styleable

In addition to using CSS Modules themselves, we'll be implementing our solution using an additional library called react-styleable. When I helped write this library, I had been considering how we would create a consistent pattern for components and application authors to consistently feed style information into components. If the way could be consistent, the interoperability would be easier, and all styles would be easily overridable. React-styleable simply sets up the convention that your CSS module exports object will be passed into your component via the CSS prop. If more people did this much, it would be a big win. Secondly, it allows a component to specify its own default styles while allowing every one of those styles to be overridden by the client application that imports the component. This is often desirable and will allow application devs to be considerate of their application style or what state the JavaScript in the application is in. As a bonus, it can make testing easier as well since your style object is passed in as a prop to the component. You can provide your own style object as a test fixture making assertions on what to expect. Very straightforward.

CSS Modules Demo

Okay, let's get down to it and style our demo component using CSS Modules. As before, we'll start with our unstyled image slider project. This time, we'll style in what looks like just a regular CSS stylesheet, but they'll be true CSS Modules, and we'll feel that as we import and use them. Let's import our first stylesheet. It's slightly different than the rest and doesn't exemplify the real power of CSS Modules. Rather, it will provide some application-level styles. We'll go over how you can do that, and then it'll be out of the way. In app.css, we'll define a couple selectors that are by default global. If you target native tags such as HTML or a selector like star, they're global by default. If you ever want to go back to the global namespace beyond that, you can use :global. There, that provides us some basic font control and background color for our application. Let's get one more thing out of the way. As we've done in previous sections, let's create a config.css stylesheet to hold the image dimension variables for our slider. Again, note that CSSNext puts our variable declarations in :root, and variable names are prefixed with --. Now let's fill out our first usage of a CSS module with truly isolated selectors. We'll start with our most general subcomponent and work our way in. The frame is up first. First, we'll import our CSS file. Note that the import is happening in JavaScript, but the file we're importing looks to be a CSS file. Webpack makes this possible crawling all our dependencies including .css files, which it handles with the loader specified in our webpack.config file. If we take a quick look at our frame, remember that it has but one element. It's just a frame, after all, and it has a root div. We're going to style that div in frame.css. As a convention in this project, we're going to give this root or parent element the class name of root. I fill in the styles. The difference here is that this file now represents a real CSS module. In order to get access to the variables in config.css, we must be explicit by importing it here as a dependency. Now back to our component file, we'll make our CSS module exports object available in our component via react-styleable. In order to do that, we'll wrap our component in react-styleable's higher-order component that receives our CSS module and passes it on as props.css to our component. Thus, the root class is available as props.css .root. Let's practice on the next component, carousel. We'll do as before and create a carousel.css module. We know we need config variables, so we'll make sure to explicitly import that. In this component, we'll similarly have just one container element, so we'll call it root. Then as before, we'll import styleable and wrap our component in its higher-order component. Use the props.css .root in the class name. Save, and we see our styles applied. Looking at this, you might wonder how we can use the same selector name root on both components, but the styles don't clash or overwrite one another. Remember that when the css-loader using the modules flag transforms our CSS Modules, it produces an exports object that hashes the selector names. That's why our selectors look the same in code, but in the browser, they actually have different selector names. To prove it to yourself, open up Chrome dev tools and take a look. The outermost div, frame, and carousel beneath it have the local selector name of root, but then their hashes make them unique. Pretty simple and awesome. Let's go on to the slide component. Get all this set up for importing slide.css, and save. We're stopped and told that slide.css doesn't exist yet. So let's create it. Now let's style the same elements that we have in previous sections on the slide component. We'll style the outer container, the footer, and the title element. Did you see what we called the selector names for these elements, root, footer, and title. Exactly what they are. We give no thought for anything except this small context of this little component. We don't have to keep the world of our CSS setup in our mind, add our project prefixes, or remember our BEM hierarchies because our little world here is isolated from all of that. We can call these things what our heart has wanted to call these things all along. We'll just name these elements' selectors using simple nouns and be on our way, confident that we won't be clobbered by outside styles and that we won't accidentally harm other component styles. Once we've filled in all the styles we need for this component and save, the styles take effect and make our footer look quite nice. Let's go to the next component, nav, which is usually quite interesting. We first create the nav.css module. We set up our component to consume the CSS module using react-styleable, and we establish that we want two selectors, one for each button prev and next. In the stylesheet, we observed that they're both just buttons, so we make a general selector for them to receive shared attributes from. To set up for this, we'll use the composes keyword on both prev and next. These two buttons will look generally the same receiving all of the attributes of button and then adding their own specific attributes for text alignment. We'll also set up and style the root container element that we originally passed by and add that class name to our component. Now let's fill in all those button class details. Once we get to the end and press save, our buttons appear styled onscreen, aligned as we'd expect. Now we want to make nav smarter. As before, we create two functions to take care of the logic of hiding buttons when there are no slides to advance to in that direction. This logic will be determined by the hasPrevious and hasNext props to the component. In the case of wanting the button hidden, we'll use the prevHidden or nextHidden selectors. When I save, notice that the previous button now looks unstyled. This is because it's trying to use the prevHidden selector, which we have not implemented. Jumping into the stylesheet, we create both hidden selectors composing them of their prev and next counterparts, additionally mixing in the hidden class, which will provide the hidden visibility. Trying it out, it works like a charm. Now let's add a button hover to our navigation buttons. CSS Modules feel like the best of both worlds in this regard. We get real isolated modules while still getting to write CSS when specifying style information. Gray hover seems to work. Finally, let's add a keyframe animation. This animation will scale the element to be larger and then shrink it back down again. We'll use it on the button selector and loop it infinitely. Hey, dancing buttons. We did it! CSS Modules are ours.

Evaluate CSS Modules

Out of the strategies that I personally use, this is currently my favorite and really represents the pinnacle option that we've explored here. It is very powerful and really pleasant to use. It's changed how I've thought about long-lived CSS code bases and re-factoring of them. Styles are truly isolated. You can still create long or complicated dependency chains, so be careful. At the very least, now they're explicit and tractable. Coding in CSS Modules has changed how I think about selector naming. Now my thoughts can be isolated to just the component that I'm constructing. They can be more semantic and specific within just that context. There is no need to prefix terribly long namespaces into selector names. And if you catch yourself doing so, it's likely a sign that your component could be decomposed into subcomponents. Let's go through a quick mental checklist on how this strategy fares according to our original problem outline, in this case, as a strategy that is both modular and uses the CSS language. Are there global styles? No, at least, there aren't by default. It is easier to break out and go global again, so we must be disciplined to avoid that. Are the dependencies explicit. Yes, they must be. They're all local modules. Is there dead code elimination? Yes, it's possible. All module usages are explicit. Is minification possible? Yes, in the CSS attributes and selectors, and selectors are local and generated programmatically. Are constants shareable? No, and this is too bad. Inline styles based in JavaScript still shine here. I would love a great library or language-based option here. Really, some CSS to JavaScript option or vice versa must exist already. Are resolutions predictable? Yes, these are local modules, and tools like react-styleable help incoming styles be added even more predictably by merging existing styles and pushing style information through React props. Are styles isolated? Yes, so long global namespace. I told you. Mmm-mmm, good.

Thank You

Now you have a handful of styling options at your fingertips for use the next time you need to make a React component look good. These are only a few, so keep your eyes open for more that will doubtless appear and be great. For now, you should have a good flavor and a good couple references and some good ideas for styling the React apps on your plate today. Thank you so much for taking the time to learn with me. I hope you've enjoyed the materials and feel empowered to make good-looking and well-organized React components. Enjoy the course materials. Do some experiments with them, change, expand, and improve on the ideas there. Thank you and happy styling.

Course author

Jake Trent

He's a creator at heart, making art, music and software. He's been privileged to work with talented teams and contribute to great causes and useful products and hopes to do more of it.

Course info

Level	Intermediate
Rating	★★★★★ (235)
My rating	★★★★★
Duration	1h 23m

Updated	13 Mar 2019

Share course