Styling React Components

Introduction

Traditionally, we have split web pages up into markup (HTML), styling (CSS), and logic (JavaScript).

Now that we are working with React, we have already seen how markup and logic are appearing in the same file (using JSX). Pete Hunt (one of the early developers working on React) famously said while defending the initially controversial nature of JSX, *"we've long been forced to separate our technologies rather than our concerns"*.

Pete's talk also touches on the following points:

- Often display logic and markup are inevitably tightly **coupled**.
- This also means that display logic and markup are highly **cohesive**.
- The JavaScript code that drives the UI and the markup that displays it to the user are both doing basically the same thing - they are handling user events and are rendering data to the user.
- It's a seperation of technologies that you're using to implement the same concern.

Whilst he is referring to the relationship between markup and logic, we can also add styling to this conversation. The idea of building **reusable** and **composable** components with React has lead to new ways to think about styling. While best practices are still being figured out, some early patterns have begun to emerge. Let's have a look at some solutions that are currently on the cards.

Traditional Approaches

1. Old School Styling

We have already had a look at how to link a stylesheet into our React applications using the `import` syntax:

```
import './scss/style.scss';
```

This allows us to do the type of styling that we are familiar with - adding ids and classes, setting up variables and mixins if we're using SCSS, and hoping for the best. In CSS everything is global by default, which means that often we need to be wary of overriding styles and making updates to our CSS that will alter other areas of the app unexpectedly.

This approach is acceptable when starting out, but as your app grows in complexity, it might be a good idea to move towards a more structured solution.

2. CSS Methodologies

What happens when your application starts to expand and new concepts get added? Broad CSS selectors are like globals. The problem gets even worse if you have to deal with loading order. If selectors end up in a tie, the last declaration wins, unless there's !important somewhere. It gets complex very fast.

We could battle this problem by making the selectors more specific, using some naming rules, and so on. That just delays the inevitable. As people have battled with this problem for a while, various methodologies have emerged.

There are plenty of methodologies out there aiming to reduce the CSS footprint, organize cooperation among programmers and maintain large CSS codebases.

Particularly, OOCSS (Object-Oriented CSS), SMACSS (Scalable and Modular Approach for CSS), and BEM (Block Element Modifier) are well known. Each of them solves problems of vanilla CSS in their own way.

Check out the BEM docs to see what it's all about.

As they state, *"No matter what methodology you choose to use in your projects, you will benefit from the advantages of more structured CSS and UI"*, however maintaining long class names can be arduous.

React Based Approaches

With React we have some additional alternatives. What if the way we've been thinking about styling has been misguided? CSS is powerful, but it can become an unmaintainable mess without some discipline. There are various approaches for React that allow us to push styling to the component level.

3. Inline Styles

Adding inline styles to HTML using the `style` attribute has been far from best practice for a long time, but React developers are known to question best practices (think JSX!). It may feel counter intuitive, but foundational features of React fall into this same category. Who would have thought you would put your entire view hierarchy into a render function?

There are a few things to note when writing inline styles inside a React component:

- When we code inline styles with React, all of our styles are actually written in Javascript, not CSS
- CSS attributes should be camel case

- Values are normally wrapped in a string
- Having styling at the component level means we can implement logic that alters those styles easily

```
import React from 'react'

const Button = () => {
  const styles = {
    backgroundColor: 'dodgerblue',
    color: 'white',
    border: 'none',
    borderRadius: '4px',
    padding: '8px 12px'
  };

  return (
    <button style={styles}>Click</button>
  );
};

export default Button;
```

If you are concerned about cluttering your component file with CSS, you can always move the styles into a seperate file and import them into them, but note that this makes it less straightforward to use logic inside the styles object.

```
// button-styles.js

export default {
  backgroundColor: 'dodgerblue',
  color: 'white',
  border: 'none',
  borderRadius: '4px',
  padding: '8px 12px'
};
```

```
// button.js

import React from 'react'

import styles from './button-styles.js';

const Button = () => {
  return (
    <button style={styles}>Click</button>
  );
};

export default Button;
```

The idea would be that each component has their own `-styles.js` file that is imported into the component file, ensuring that there are no global styles, and one component's styles cannot affect another component.

Pros	Cons
No global styles, everything is a local style object	Since everything is in JS, there is no CSS media query syntax
Explicit dependecies using the import syntax	No native support for pseudo selectors, such as hover or last-child
Easy to share values between components when necessary	No good option for reuable animations such as CSS keyframe animation when we use inline styles
Isolation ensures no components styling influences any other component styling	Difficult to perform large, sweeping changes to our codebase

At the moment it is very common to see inline styles being used in React libraries.

There are packages that you can use to tackle some of the problems that inline styles have, such as keyframe animations, pseduo selectors and media queries. A popular one is Radium.

💻 Inline Styles Codealong

Open up the `inline-styles` starter code, `yarn install` and `yarn start`. We are going to style a progress bar that a user can increase using buttons. We have three components:

- `Donations` - this is a classical component that holds the logic, and renders child components
- `ProgressBar` - this is a functional component that will take the `percentageComplete` value and add styles accordingly

- `Button` - a functional component that will take a function to fire on click

Click on the buttons, and notice that the percentage increases each time. Let's style these `Button` components first.

Inside `Button.js` create an object called `buttonStyle` :

```
const buttonStyle = {
  border: '2px solid black',
  backgroundColor: 'white',
  marginRight: '5px',
  fontSize: '14px',
  padding: '8px 10px',
  cursor: 'pointer'
};
```

Then we can add those styles to the `<button>` element using the `style` attribute:

```
<button style={buttonStyle} value={value} onClick={handleClick}>{value}</button>
```

Make sure that you can see those styles in the browser. What if we wanted to add some styles to buttons that are disabled? If the value on the button will take the percentage over 100% we want to disable the button.

Let's add a `disabled` prop to the `Button` component, which checks for this condition and returns true or false. Inside `Donations.js` :

```
render() {
  const { percentageComplete } = this.state;

  return (
    <div>
      <h1>Percentage Complete: {percentageComplete}%</h1>
      <ProgressBar percentageComplete={percentageComplete} />
      {[5, 10, 15].map((number, i) => (
        <Button
          key={i}
          handleClick={this.handleClick}
          value={number}
          disabled={percentageComplete + number > 100}/>
      ))}
    </div>
  );
}
```

Then inside `Button.js` we can check for the disabled prop. Add it to the props object.

```
const Button = ({ value, handleClick, disabled }) => {
  ...
};
```

We can't style using pseduo selectors (`:disabled`) using inline styles, unless we use Radium or another library. Instead we can use this boolean value inside the styles object to determine certain CSS properties.

```
const buttonStyle = {
  border: '2px solid black',
  backgroundColor: 'white',
  marginRight: '5px',
  fontSize: '14px',
  padding: '8px 10px',
  cursor: disabled ? 'default': 'pointer',
  opacity: disabled ? 0.5 : 1
};
```

Here we are saying that disabled buttons should have a `default` cursor, and they should be `0.5` opacity.

We want to send that percentage into the `ProgressBar` component as a prop, to allow us to add a percentage based width to the inner div.

```
<ProgressBar percentageComplete={percentageComplete}/>
```

Desconstruct this out of props inside the `ProgressBar` component:

```
const ProgressBar = ({ percentageComplete }) => {
  ...
}
```

At this point we could console log `percentageComplete` to ensure that we are getting this value each time a button is clicked.

The progress bar is going to made of two divs. The outer div will have a border and a fixed height and width. The inner div will be 100% of the height of the outer div, and it's width will be based on the `percentageComplete` value.

Create two objects for these styles.

```
const outerStyle = {
  width: '500px',
  height: '20px',
  border: '2px solid black',
  padding: '3px',
  marginBottom: '20px'
};

const innerStyle = {
  height: '100%',
  backgroundColor: 'lightgrey',
  width: percentageComplete + '%',
  transition: 'all 0.3s ease'
};
```

Notice how we can use the `percentageComplete` value to create the `width` property by adding a `%` sign.

Add these styles to the divs inside the `return`.

```
return (
  <div style={outerStyle}>
    <div style={innerStyle}></div>
  </div>
);
```

We could also use the `percentageComplete` value to change the background colour of the progress bar once it reaches 100%.

```
const innerStyle = {
  height: '100%',
  backgroundColor: percentageComplete === 100 ? 'dodgerblue' : 'lightgrey'
  width: percentageComplete + '%',
  transition: 'all 0.3s ease'
};
```

4. CSS Modules

The premise of CSS Modules is simple - each React component gets its own CSS file, which is scoped to that file and component. The magic happens at build time, when local class names – which can be super simple without risking collisions – are mapped to automatically-generated (hashed) ones and exported as a JavaScript object to use within React components.

For example, two components could use the same `.button` class, but thanks to CSS modules, the class names will be uniquely hashed, meaning that the styles will not interfer with/be overrided by each other.

There is a little bit of setup that we have to do in order to use CSS modules. Inside `webpack.config.js` update the `style-loader` to be the following:

```
{ test: /\.scss$/, loader: ['style-loader', 'css-loader?modules&localIdentName=[lo
cal]---[hash:base64:5]', 'sass-loader'], exclude: /node_modules/ }
```

Create a `button-styles.scss` file inside the `scss` directory.

```
.button {
  background-color: dodgerblue;
  color: white;
  border: none;
  border-radius: 4px;
  padding: 8px 12px;
}
```

The Webpack CSS loader then generates a hash for each selector, known in Webpack as the local ident name. This hashed selector becomes unique on the page.

After hashing the class name will look something like this:

```
.button--99a0f {
  background-color: dodgerblue;
  color: white;
  border: none;
  border-radius: 4px;
  padding: 8px 12px;
}
```

Even if two files use the same selector name, the hashing will make it unique.

Import this file into the `Button.js` component:

```
import css from './scss/button-styles.scss';
```

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. We can then add this hashed class name to the `Button` component like this:

```
const Button = () => {
  return (
    <button className={css.button}>Regular Component</button>
  );
};
```

When specifiying the `className` attribute, you use the exports object key, but the value that is subsituted after build time and at run time will be the hashed selector. Simple, and powerful.

```
<button class="button--99a0f">Push</button>
```

Pros	Cons
One step towards modular and reusable components that will not have side effects	Not as human-readable DOM
Shorter and more semantic class names	Some Webpack setup to get started
Smaller CSS files	

For a lot of React developers, CSS modules are awesome. They are a great mix of the worlds of inline and external CSS. You get much of the local modular *inline styles* feeling, and you also get to write your styles using the CSS language. This means that we can use pseudo selectors, media queries and keyframe animations as usual without needed to bring in any additional packages.

💻 CSS Modules Codealong

Open up the `css-modules` starter code, `yarn install` and `yarn start`.

To use CSS modules we need the following Webpack loaders:

- style-loader
- css-loader

We have both of these in our Webpack setup, so we just need to add a little extra configuration. Replace the `.scss` loader with the following:

```
{ test: /\.scss$/, loader: ['style-loader', 'css-loader?modules&localIdentName=[lo
cal]---[hash:base64:5]', 'sass-loader'], exclude: /node_modules/ }
```

Here we are saying that it should use `module` mode, and that the class names be in the following format: original class name, followed by `---`, followed by 5 random characters.

Let's create a stylesheet for each component inside a `scss/components` directory.

```
mkdir src/scss/components
touch src/scss/components/button.scss
touch src/scss/components/progress-bar.scss
```

Inside `progress-bar.scss` add the following styles:

```
.outer {
  width: 500px;
  height: 20px;
  border: 2px solid black;
  padding: 2px;
  margin-bottom: 20px;
}

.inner {
  height: 100%;
  width: 0;
  background-color: lightgrey;
  transition: all 0.3s ease;
}
```

Now let's import this stylesheet into the component file.

```
import css from '../scss/components/progress-bar.scss';
```

This `css` variable will be an object with the original class names as keys and the hashed class names as variables. In this instance it might look something like this:

```
{
  inner: "inner---fFTE8"
  outer: "outer---3uaew"
}
```

This allows us to add the class names to the JSX elements in the following way:

```
return (
  <div className={css.outer}>
    <div className={css.inner}></div>
  </div>
);
```

This is really nice, however we do still need to do a little bit of inline styling in order to change the width as the `percentageComplete` changes.

Add the following styles object to the `ProgressBar` component:

```
const innerStyle = {
  width: percentageComplete + '%'
};
```

And then add these inline styles to the child div:

```
return (
  <div className={css.outer}>
    <div className={css.inner} style={innerStyle}></div>
  </div>
);
```

Neat! Next let's style the button component. Inside `button.scss` add the following styles:

```
.button {
  border: 2px solid black;
  background-color: white;
  margin-right: 5px;
  font-size: 14px;
  padding: 8px 10px;
  cursor: pointer;
  &:disabled {
    cursor: default;
    opacity: 0.5;
  }
}
```

Import the `.scss` file into the `Button` component file:

```
import css from '../scss/components/button.scss';
```

Now we can use the `css.button` value as the class name.

```
const Button = ({ value, handleClick, disabled}) => {
  return (
    <button className={css.button} value={value} onClick={handleClick} disabled={d
isabled}>{value}</button>
  );
};
```

The thing that's really cool is that even though we've used the `.button` class once, we could use it again inside another `.scss` file and the two *would not conflict*. Let's see this in action. Create a new component file called `ResetButton.js` and add the following code:

```
import React from 'react';

import css from '../scss/components/reset-button.scss';

const ResetButton = ({ handleClick }) => {
  return (
    <button className={css.button} onClick={handleClick}>Reset</button>
  );
};

export default ResetButton;
```

Create a `.scss` file called `reset-button` with the following SCSS inside:

```
.button {
  border: 2px solid dodgerblue;
  color: dodgerblue;
  background-color: white;
  margin-right: 5px;
  font-size: 14px;
  padding: 8px 10px;
  cursor: pointer;
}
```

Notice that we can use the same class (`.button`) here, and thanks to the hashing, it will end up being unique.

Now let's add this button inside the `Donations` component and pass in the `resetPercentage` method.

```
return (
  <div>
    <h1>Percentage Complete: {percentageComplete}%</h1>
    <ProgressBar percentageComplete={percentageComplete} />
    {[5, 10, 15].map((number, i) => (
      <Button
        key={i}
        handleClick={this.handleClick}
        value={number}
        disabled={percentageComplete + number > 100}/>
    ))}
    <ResetButton handleClick={this.resetPercentage}/>
  </div>
);
```

Check out the elements in Chrome and have a look at the hashed class names. This is cool as our components have CSS that is scoped to just that one component, meaning we can avoid long names and accidentally using the same class name.

Note: If you are going to use this method it's worth looking at the `classnames` package - which is a *"simple javascript utility for conditionally joining classNames together"*.

5. Styled Components 💅

Styled-Components is a new CSS tool, created by Max Stoiber and Glen Maddern, which helps you organize CSS in your React project. There is a talk that Max gave at a React conference about Style Components that is worth a watch.

The main thing you need to understand about Styled Components is that its name should be taken quite literally. You are no longer styling HTML elements or components based on their class or HTML element: Instead, you're defining styled components that possesses their own encapsulated styles. Then you're using these freely throughout your codebase.

Thee goals of styled components:

1. Getting rid of the mapping between styles and components — Most of the time, a dumb component always has its own small style.css file related. So, you need to create two files every time you want to create the dumb component. This seems to be fine at the beginning, however, when your project is getting bigger, you will end-up with a whole bunch of files. Styled-Components allows you to write CSS directly inside your component, which perfectly solved this problem.
2. Building small and reusable components — Small components can easily be reused and tested. By using Styled-Components, you can easily build a small component and extend its capability with props.
3. Reducing the risk of specificity clash — Everyone might have encounter the specificity clash problem before. For example, you just wanted to add a margin to a specific paragraph, but it unintentionally impacts the other paragraphs. You can easily solve this problem by applying a CSS class only once. Styled-Components is actually doing this for us. It automatically generates a unique class name and pass it to our component.

To use it in your project first install it using npm or yarn:

```
yarn add styled-components
```

Create a file called to hold your component, such as `StyledButton.js`:

```
import styled from 'styled-components';

const StyledButton = styled.button`
  background-color: tomato;
  color: white;
  border: none;
  border-radius: 4px;
  padding: 8px 12px;
`;

export default StyledButton;
```

First we need to import `styled-components`, and then we can use the Styled Components syntax to create a new button with set styles.

Import this button into another file:

```
import StyledButton from './styledButton';
```

Add it to a `render()` method:

```
render() {
  return (
    <main>
      <StyledButton>Styled Component</StyledButton>
    </main>
  );
}
```

> **Note:** The CSS rules are automatically vendor prefixed, so you don't have to think about it.

You can nest styled components inside other styled components. Here is the basic example from the docs.

```
// Create a Title component that'll render an <h1> tag with some styles
const Title = styled.h1`
    font-size: 1.5em;
    text-align: center;
    color: palevioletred;
`;

// Create a Wrapper component that'll render a <section> tag with some styles
const Wrapper = styled.section`
    padding: 4em;
    background: papayawhip;
`;

// Use Title and Wrapper like any other React component — except they're styled!
render(
    <Wrapper>
        <Title>
            Hello World, this is my first styled component!
        </Title>
    </Wrapper>
);
```

Another nice feature of the Styled Components library is the ability to adapt styles based on props. From the docs:

```
const Button = styled.button`
    /* Adapt the colours based on primary prop */
    background: ${props => props.primary ? 'palevioletred' : 'white'};
    color: ${props => props.primary ? 'white' : 'palevioletred'};

    font-size: 1em;
    margin: 1em;
    padding: 0.25em 1em;
    border: 2px solid palevioletred;
    border-radius: 3px;
`;

render(
    <div>
        <Button>Normal</Button>
        <Button primary>Primary</Button>
    </div>
);
```

💻 Styled Components Codealong

Open up the `styled-components` starter code, `yarn install` and `yarn start`.

Install Styled Components using yarn:

```
yarn add styled-components
```

We are going to empty out the `Button.js` file and replace it with the following:

```
import styled from 'styled-components';

const Button = styled.button`
  border: 2px solid black;
  background-color: white;
  margin-right: 5px;
  font-size: 14px;
  padding: 8px 10px;
  cursor: pointer;
  &:disabled {
    cursor: default;
    opacity: 0.5;
  }
`;

export default Button;
```

You can read more about the syntax in the docs. Note that we don't need to import React anymore. Also note that we don't need to use camel case and we can use pseudo selectors (and media queries).

We don't add event listeners or additional props to styled components. Instead we are going to use this component in the following way:

```
<Button>Click!</Button>
```

This means that we need to update the way that we are using the `Button` component inside the `Donations` directive to be the following:

```
return (
  <div>
    <h1>Percentage Complete: {percentageComplete}%</h1>
    <ProgressBar percentageComplete={percentageComplete}/>
    {[5, 10, 15].map((number, i) => (
      <Button
        value={number}
        key={i}
        onClick={this.handleClick}
        disabled={percentageComplete + number > 100}
      >
        {number}
      </Button>
    ))}
    <ResetButton onClick={this.resetPercentage}>Reset</ResetButton>
  </div>
);
```

Cool. Anytime we want a button with that default styling in our app we can just use the new `<Button>` component.

Let's use Styled Components to create the progress bar. Inside `ProgressBar.js` replace the code with the following:

```
import React from 'react';
import styled from 'styled-components';

const Outer = styled.div`
  width: 500px;
  height: 20px;
  border: 2px solid black;
  padding: 2px;
  margin-bottom: 20px;
`;

const Inner = styled.div`
  height: 100%;
  width: ${props => props.percentageComplete + '%'};
  background-color: ${props => props.percentageComplete === 100 ? 'dodgerblue' : '
lightgrey'};
  transition: all 0.3s ease;
`;

const ProgressBar = ({ percentageComplete }) => {

  return (
    <Outer>
      <Inner percentageComplete={percentageComplete}></Inner>
    </Outer>
  );
};

export default ProgressBar;
```

Note how we can use the props that were passed in to change the values of the `width` and `background-color` CSS properties.

Independent Practice

Create a styled component called `ResetButton.js` in a new file. The reset button should have the styles of the `Button` component, but should be extended to include the following:

```
border-color: dodgerblue;
color: dodgerblue;
```

Import this new Styled Component into the `Donations` component, and add the `resetPercentage` method to an `onClick` event listener.

Solution

```
// src/components/ResetButton.js

import Button from './Button';

const ResetButton = Button.extend`
    border-color: dodgerblue;
    color: dodgerblue;
`;

export default ResetButton;
```

```
// src/components/Donations.js

import ResetButton from './ResetButton';

...

class Donations extends React.Component {

  ...

  render() {
    const { percentageComplete } = this.state;

    return (
      <div>
        <h1>Percentage Complete: {percentageComplete}%</h1>
        <ProgressBar percentageComplete={percentageComplete}/>
        {[5, 10, 15].map((number, i) => (
          <Button
            value={number}
            key={i}
            onClick={this.handleClick}
            disabled={percentageComplete + number > 100}
          >
            {number}
          </Button>
        ))}
        <ResetButton onClick={this.resetPercentage}>Reset</ResetButton>
      </div>
    );
  }
}
```

Conclusion

There are so many blog posts out there that document the "best ways" to style React components - you should be doing your own research on these methods before picking one for a project. You might end up using SCSS as usual - this is totally fine!

Further reading

- Modular CSS with React
- Practical Guide to React and CSS Modules
- What to use for React styling?
- What is the best way to style React components these days?
- Styled Components: Enforcing Best Practices In Component-Based Systems
- Why you shouldn't style with JavaScript