The Pragmatic Programmers

Test-Driven React 2

Find Problems Early, Fix Them Quickly, Code with Confidence

Trevor Burnham
edited by Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Copyright © The Pragmatic Programmers, LLC.

Getting Started with Styled-Components

As apps grow, they can contain thousands of style rules—too many for a single person to keep track of. This leads to unintended conflicts. For example, which of the styles below will apply to a disabled button with the `white` class?

```
// StylesheetA.css
button.white {
  background-color: white;
  color: black;
}
```

```
// StylesheetB.css
button:disabled {
  background-color: grey;
  color: darkgrey;
}
```

The answer is that it depends on the order the stylesheets are loaded in, as both selectors have the same level of specificity. This is a very fragile state of affairs for a complex app. Worse, removing style rules becomes a risky endeavor. Let's say that your app has this style rule:

```
p.alert-message {
  color: red;
}
```

You search your codebase for `alert-message`, find no results, and so you remove the style. But your search didn't match this React code:

```
<p className={`${urgency}-message`}>This is an alert!</p>
```

The CSS-in-JS paradigm, exemplified by a library called *styled-components*, greatly alleviates these problems by allowing a component's style rules to be written as a function of its props. This offers a number of advantages:

- No need to search your codebase to find out which styles are associated with a component. Its styles are either in the same module, or imported like any other dependency.
- Styles are generated as a function of their component's props and state, just like markup.
- Styles can be subjected to unit tests.

And unlike the `style` prop, style rules generated by styled-components have the full range of functionality of ordinary CSS, including support for media queries, keyframe animations, and pseudo-classes.

Let's start adding some style to test-driven-carousel. Install the styled-components package as a dependency:

```
$ npm install --save styled-components@6.0.5
```

So far, this book's modus operandi has been to present tests first, then the code to satisfy these tests. This is, after all, a book about TDD, and TDD is usually taken to mean "writing tests first." But on a deeper level, TDD is about seeking useful feedback for your code as quickly as possible. Tests are just one possible source of feedback. And when it comes to styles, the most useful source of feedback is usually *seeing* those styles.

So set tests aside for now. All you'll need for this section is a live dev server.

Creating a Styled Component

Currently, the <img> created by CarouselSlide is unstyled, which means that it scales to whatever the original size of the image is. That means that the carousel jarringly expands and contracts as users move from slide to slide. Worse, it'll push other page content around in the process. Clearly, this needs to be fixed!

To do that, replace the unstyled <img> element with a component generated by styled-components:

```
// src/CarouselSlide.tsx
import { ComponentPropsWithRef, ReactNode } from "react";
import styled from "styled-components";

const ScaledImg = styled.img`
  object-fit: cover;
  width: 100%;
  height: 500px;
`;

const CarouselSlide = ({
  imgUrl,
  description,
  attribution,
  ...rest
}: {
  imgUrl?: string;
  description?: ReactNode;
  attribution?: ReactNode;
} & ComponentPropsWithRef<"figure">) => (
  <figure {...rest}>
    <ScaledImg src={imgUrl} />
    <figcaption>
      <strong>{description}</strong> {attribution}
    </figcaption>
```

```
  </figure>
);

export default CarouselSlide;
```

styled.img is a function that generates a component that renders an <img> tag with the given styles. When an instance of that ScaledImg component is mounted, styled-components will dynamically insert a style rule with the styles you provided, using a class name selector based on the hash of those styles.

There's some syntactic fanciness here in the form of an ES6 feature called *tagged templates:*[1] If you put a function directly in front of a template string (the kind delimited by backticks), that function is called with the template string as an argument.

In the case of ScaledImg, you could use the normal function call syntax, since the string with the styles is a constant. Where the tagged template syntax unlocks new possibilities is when the string has interpolations (the ${...} syntax): each piece of the interpolated string is passed in to the function as a separate argument. That gives the tag function the chance to process interpolated variables. As we'll soon see, styled-components takes advantage of this power.

As soon as you hit save, you should see the difference in your browser. Before, the size of the <img> tag was determined by the image file it loaded. Now, it takes up the full width of its container and has 500px of height. The object-fit: cover rule means that the image keeps its aspect ratio as it expands or contracts to those dimensions, getting clipped as needed.

Why 500px? Really, the height of the image should be determined by the app rendering the carousel component. So let's add an imgHeight prop to CarouselSlide to allow the component consumer to override the default image height:

```
// src/CarouselSlide.tsx
import { ComponentPropsWithRef, ReactNode } from "react";
import styled from "styled-components";

➤ const DEFAULT_IMG_HEIGHT = "500px";
➤
➤ export type CarouselSlideProps = {
➤   imgUrl?: string;
①   /** @default "500px" */
➤   imgHeight?: string | number;
➤   description?: ReactNode;
➤   attribution?: ReactNode;
```

1. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#Tagged_templates

```
➤ } & ComponentPropsWithRef<"figure">;
➤
❷ type ImgComponentProps = {
➤   $height?: CarouselSlideProps["imgHeight"];
➤ };

➤ const ScaledImg = styled.img<ImgComponentProps>`
➤   object-fit: cover;
➤   width: 100%;
➤   height: ${(
❸     props
➤   ) =>
➤     typeof props.$height === "number" ? `${props.$height}px` : props.$height};
➤ `;

  const CarouselSlide = ({
    imgUrl,
➤   imgHeight = DEFAULT_IMG_HEIGHT,
    description,
    attribution,
    ...rest
  }: CarouselSlideProps) => (
    <figure {...rest}>
      {}
➤     <ScaledImg src={imgUrl} $height={imgHeight} />
➤     {}
      <figcaption>
        <strong>{description}</strong> {attribution}
      </figcaption>
    </figure>
  );

  export default CarouselSlide;
```

❶ The /** ... */ syntax denotes a JSDoc comment. These comments don't affect the behavior of the code or the typechecker, but they can provide useful contextual information to developers who are using your code, complementing the TypeScript types.

For example, in VS Code if you hover over the imgHeight in <CarouselSlide imgHeight={}>, the editor will show you both its TypeScript type and the associated JSDoc. Here the JSDoc comment indicates that the default value used if the imgHeight prop is not set is "500px".

❷ The ImgComponentProps type is used as a *type parameter* for the styled.img function. In this case, the type parameter indicates the set of props that the ScaledImg component returned by that function will be able to accept, in addition to everything the native img tag can accept. The $ prefix in $height is a styled-components convention that indicates a *transient prop*,

which is used for interpolated styles but not passed through to the <img> element in the DOM.

3. This is where styled-components really gets exciting: interpolated values in the style template can be a function of the component's props! Whereas ordinary CSS is static, these styles are completely dynamic. If the imgHeight prop changes, the styles update automatically.

Right now, imgHeight can be overridden on a slide-by-slide basis, since Carousel passes the whole slide data object down to CarouselSlide as props. But in most cases, the Carousel consumer will want it to have a consistent height. So let's add a prop to Carousel that can override the default imgHeight on CarouselSlide:

```
// src/Carousel.tsx
import { ReactNode, useState } from "react";
import CarouselButton from "./CarouselButton";
import CarouselSlide, { CarouselSlideProps } from "./CarouselSlide";

type Slide = {
  imgUrl?: string;
  description?: ReactNode;
  attribution?: ReactNode;
};

export type CarouselProps = {
  slides: Slide[];
  defaultImgHeight?: CarouselSlideProps["imgHeight"];   // (1)
};

const Carousel = ({ slides, defaultImgHeight }: CarouselProps) => {
  const [slideIndex, setSlideIndex] = useState(0);
  return (
    <div data-testid="carousel">
      <CarouselSlide imgHeight={defaultImgHeight} {...slides?.[slideIndex]} />   // (2)
      {/* ... */}
    </div>
  );
};

export default Carousel;
```

1. The CarouselSlideProps["imgHeight"] syntax references the type of the imgHeight field in CarouselSlideProps.

2. Note that the order here is significant: Because the ...slides?.[slideIndex] spread comes after imgHeight is set to defaultImgHeight, the imgHeight value from the slide will take precedence (if it's defined).

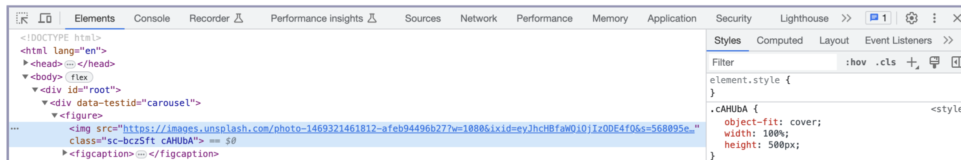
Try setting defaultImgHeight on ExampleCarousel, and you should see the carousel's height change in the browser.

• Click HERE to purchase this book now. discuss

Commit your work on this new feature:

```
:sparkles: Add image height styling
```

You may be wondering: How did styled-components apply those styles to the <img> tag? If you inspect one of the <img> tags in the browser, as in the next screenshot, you'll see that its class attribute is full of gobbledigook. Something like class="sc-bdVaJa hhfYDU". The styled-components library generated these class names for you, and injected a corresponding style rule into a <style> tag in the <head> of the page.



In fact, the <img> element has two classnames generated by styled-components. One of these, the one with the sc- prefix, is a stable class name that styled-components uses for selectors. The other, the one the styles are applied to, is generated from a hash of the styles. In practice, the distinction is just an implementation detail. Just remember: You should *never, ever* copy any class names generated by styled-components in your code! All generated class names are subject to change.

Having unreadable class names is an unfortunate drawback of styled-components. Luckily, it can be mitigated with help of a popular preprocessor, Babel.