7 CSS in JavaScript

This chapter covers

- Making your applications look great
- Evaluating the many ways to add styling
- Exploring three ways to style a component

Creating functional apps is a good start, but ensuring that they look gorgeous as well is going to make users love using your apps. Making HTML websites look good is the job of CSS, which is notoriously different from HTML and JavaScript, which we are using in the world of React.

We will be examining three different ways of styling React applications and evaluating their strengths and weaknesses:

- CSS files and class names
- CSS Modules
- Styled-components

I will cover them in this order, which is their order of complexity and learning curve. The first option requires almost no new knowledge, so you can start right away even if you know only React and CSS. I've excluded the most complex options because you would have to read a lot of documentation to start; they use methods and concepts far from regular CSS to achieve a similar result.

But why are there so many ways of styling React and web applications in general? Why don't all of us developers agree on the best way and stick with that? Well, the "best way" to style a React application is an unsolved problem. A styling solution has many requirements, depending on what you're building, who's building it, who your target audience is, and (of course) personal preference and professional sentiments.

In this chapter, I aim to demonstrate, compare, and evaluate three approaches so you will be equipped to select the best one for your project.

After some preliminary discussion of CSS in general, I'll show you how to implement a simple React application with each of the three methods so you can learn not just how to implement each method but also when and why to use one over another.

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch07**.

7.1 Styling with concerns

CSS has existed in its current form since the mid-1990s, so its use and incorporation in web development are well established at this point. In older web development, HTML, CSS, and JavaScript were often implemented independently, and JavaScript was more an add-on than a key component.

Modern web development, however, relies heavily on JavaScript, and with React and many other frameworks, the output HTML is defined directly inside the JavaScript application. In React, we write HTML inside JavaScript via JSX. Moving HTML into the JavaScript application allows us to closely link the script and the underlying document that it works on, making it much easier to keep them in sync and update one when the other is rewritten.

But CSS and HTML have the same problem of interconnectedness. If you update your HTML without the new structure's being reflected in CSS, your application might at best look wrong or at worst be unusable. So when we can move our HTML into our JavaScript components, why not also move the CSS in there? Compare the three scenarios in figure 7.1. The goal of CSS-in-JS technologies is to allow your web developers to define all, or almost all, of the parts of a component in a single place.

CSS-in-JS is a controversial topic. Many developers have a strong opinion about it, and some developers swear by certain paradigms and hate others. I will try my best to give you an unbiased overview of the various concepts and technologies, though of course I have my own preferences as well. (See whether you can guess which way I lean.)

If you and your team love using CSS, you're good to go. Keep using it by all means. But if you experience friction, bugs, and/or headaches, which cause velocity to go down or deadlines to slip, you may want to look for an alternative, depending on your goals and preferences.

People have many concerns about their styling solutions—and by *many*, I mean *many*! Not all the concerns apply to all projects, all teams, or all developers, and different people may have different desired outcomes for the same concern.

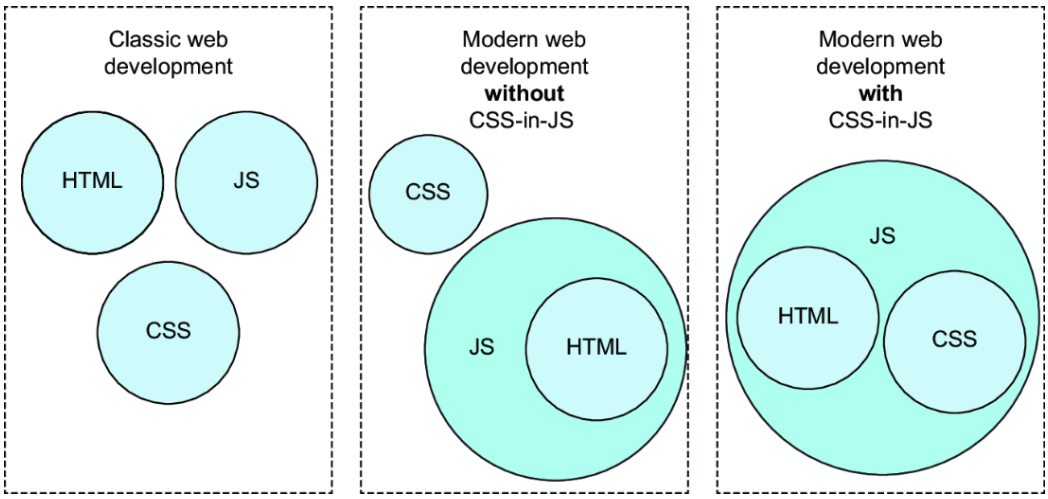

Figure 7.1 In classic web development, the three technologies are developed more or less independently. But in modern web development, either HTML alone or HTML and CSS gets bundled with the JavaScript, as the JavaScript is the main protagonist of our web application narratives.

In this chapter, I've grouped the different concerns into three main categories: CSS language features, developer experience, and web application development. First, I'll talk a bit about each category.

7.1.1 CSS language features

This category includes CSS language features that affect how easy it is to implement a given design. All these concerns are also valid in regular CSS as written in CSS files, but often with complications:

- *Specificity* —Specificity means making sure that the right rules apply to the right elements with the correct priority. The specificity of a rule is its priority. If two rules apply to an element, the least specific rule applies first and the more specific rule applies second. If you have a button that is normally green, but you want it to be purple in a single place, you must make sure that the rule for the button's being purple in that single place is more specific that the regular green rule; otherwise, the button will still be green. The green color would override the purple one, as the more specific rule applies last.
- *Collision* —Collisions happen when applications grow large and developers accidentally give things the same names. Because CSS is not hi-

erarchical, all the rules live in the same global namespace. If you try to create two different rules for a `.wrapper` class, both rules will apply to both elements even though you wanted one rule to apply to one element and the other rule to apply to another element. You can prevent collisions through sensible naming, but if you have hundreds of components, checking whether you've already used a `.group` class on some other component can be a daunting task.

- *Composition/nesting* —This principle specifies the rules for only one element as it relates to another element. The styling of the body of an element in an accordion, for example, depends on whether that entire accordion element is expanded. Specifying nested rules is super elegant in some frameworks and annoyingly complex in others.
- *Pseudoclasses and psuedoelements* —Pseudoclasses and pseudoelements are a subset of CSS rules that relates directly to specific elements, but only to parts of those elements, or that deals with those elements in certain states. You can style a disabled input with a pseudoclass and the first word of a paragraph with a pseudoelement.
- *At-rules* —At-rules are a specific subset of CSS rules that start with the `@` sign (hence, the name). The reason why they're special in this context is that they cannot be applied inline and have to live in a CSS block somewhere in the document. In more practical terms, these rules often aren't specific to one or two elements; they work on groups of (possibly unrelated) elements, such as media queries, or describe features related to individual properties of other rules, such as keyframe rules for animations.
- *Conditional rules* —These rules apply to an element under some circumstances. You might have a push button that has one styling when it's not activated and another styling when it's activated.
- *Dynamic rules* —These rules are declarations applied to an element with a value derived from JavaScript. You might have a range slider in your application that controls the opacity of an element, so you need to be able to apply any dynamic opacity to an element from JavaScript.
- *Vendor prefixes* —These experimental features have to be enabled with a prefix to work in various browsers. They used to be common in web development, but browsers rarely use them anymore. But some new CSS features still must be used through vendor prefixes, so it's nice to use a styling solution that knows exactly when and where to apply the prefixes automatically. Don't rely on that approach too heavily, though. Many of the specialized features that require vendor pre-

fixes are pseudoelements (such as scrollbars), and as far as I'm aware, they're rarely automatically vendor prefixed by any existing solution.

7.1.2 Developer experience

The concerns that are related to developer experience (DX) are about making sure that development is smooth and efficient for all team members, both new and experienced, especially when teams grow large, with many people working on the same codebase. All of the following concerns are important for efficient collaboration:

- *Colocation*—This principle involves moving related bits of code close together. For React, specifically, it makes sure that everything related to a component is located as close to that component as possible. Optimally, styles and JSX are located in the same file close to each other or at least in files in the same folder. At the extreme other end of the spectrum are styles located in some central CSS file far from the components that use the rules. Colocation is important for DX, as it makes it much easier for a developer to understand and maintain components that they haven't worked on before.
- *Familiarity*—Familiarity is about the learning curve. Imagine a new hire on your team who knows web development, CSS, HTML, and React. How quickly will they be able to contribute to your project, including maintaining and expanding the styling of your components?
- *Complexity*—Complexity is bad. Unnecessary complexity is extra bad. Web applications are complex, so CSS probably has to be complex too. But if your styling solution has too many moving parts, it might easily break down, be misconfigured, lead to bad behaviors, or be circumvented for convenience. Your solution should make your team members want to use it, not try to get around it.
- *Scaling*—Scaling involves team and project size. Having all your styles in a single CSS file might be great when you're building your own small website, but if 50 developers all need to update lines in the same file, which has more than 10,000 lines, your pull requests will get complex quickly!
- *Readability*—Readability is important to being able to scan a component quickly and understand what's going on. Can you determine which styles apply to a given element after you've found the rules? This concern may seem trivial, but some popular libraries out there, such as Tailwind, challenge readability.

- *Debugging*—Debugging concerns being able to work your way back from what you see in the browser. If you notice that a specific element has the wrong color, are you able to (quickly) find the specific line of source code that made it that color?
- *Encapsulation*—This concern is important for code reuse. If you use the same UI library in multiple projects, it's helpful if your styles are encapsulated inside the UI library and don't leak out to the surrounding project, requiring duplication of setup.
- *Theming*—Theming involves creating reusable UI libraries and distributing and customizing them for each project in an organization or even as a public library. How difficult is it to make the primary accent color of all the components in the library green rather than the default blue? Can you change all the bits and pieces of the library without too much hassle as a library consumer without knowing anything about the library internals?

7.1.3 Web application development

Web applications are supposed to be fast and efficient to use. Modern web applications use a bunch of tricks to make them even faster and serve them to the target audience as fast as possible by bundling and reusing bits and pieces. This topic is a huge one (and a moving target as browsers get smarter), but your styling approach matters for several reasons:

- *Bundle size*—This concern is trivially important. If your styles are 30 KB instead of 10 KB, your page loads slower. Some approaches are a lot more expensive in terms of bundle size than others.
- *Performance*—Performance concerns both how fast your project compiles and how fast your styles are applied to your components. Some approaches have no compilation overhead, whereas others do. Similarly, some approaches have some runtime overhead when you view the page as a visitor, whereas others do not. You must weigh these choices as the project developer.
- *Cacheability*—This concern is important for speeding up revisits to your applications. If your styles can be cached, returning users will have a much smoother experience than they would otherwise.
- *Code splitting*—Code splitting is important for minimizing the number of bytes sent to a user when they visit each part of the website. If your stylesheet for the entire website is 100 KB, but you use only about 2 KB

of that on a specific page, you're sending 98 KB of wasted content to a user who sees only that page. It would be a lot better to split your styles so that only the specific styles used on a specific page are sent to the user.
- *Maintenance*—Maintenance is about (among other things) dead-code elimination. You want to make sure you're shipping only styles that are used in your application, not styles left in there from a feature you deleted a long time ago. Some approaches make style use hard to determine, whereas others have it built in.

7.1.4 Why not inline styles?

Many examples in this book so far have used inline styles. By *using inline styles*, I mean using the React feature of assigning styles to HTML elements by setting the `style` property of an element to some object, like this:

```
return <h1 style={{ color: 'hotpink' }}>Welcome!</h1>;
```

Why are we not even considering that option, then? Well, as you look through the lists of concerns in the preceding sections, you may start to see the problems.

Almost none of the CSS language feature concerns are options with inline styles, which have no pseudoclasses or pseudoelements, no at-rules, no nested rules, and so on. What's more, these features aren't merely nice to have but essential for any useful web application that you or anyone else would want to build.

For these reasons, inline CSS is all but useless except for tiny examples and early prototypes. Tiny prototypes happen to be exactly what we're building in this book, but if we ever wanted to expand any of those examples or deploy them to a real website, we'd find a better method. If you take only one thing away from this chapter, let it be this: you should always use a better approach to CSS than inline styles.

We implemented the example application by using inline styles, and you can check it out in the `ch07/inline` folder. When you do, you'll see that it falls short on many features. It has no animations (which require at-rules), no hover states (which require pseudoclasses), and a much larger

file size than all the other examples (because the CSS is duplicated in full for every single button instance). Even in this tiny example application, the problems start piling up fast.

EXAMPLE: INLINE

This example is in the `inline` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch07/inline
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch07-inline**.

7.1.5 What about existing UI libraries?

You may be considering another option: rather than build all your styles yourself, you can use an existing UI library. You could consider using three types of UI libraries for a React application:

- Full React component libraries ready to import and use. Material UI is an example (**https://mui.com**).
- Opinionated stylesheets, in which several component designs have already been implemented in CSS, waiting for us to apply the proper class names to our application. Bootstrap is an example (**https://getbootstrap.com**).
- Unopinionated (utilitarian) stylesheets, which allow you to create almost anything from base class names. Tailwind CSS is an example (**https://tailwindcss.com**).

Although all these options are great for various React applications, I'm not going to cover them in this chapter, as I don't feel that they're good solutions for this particular application. Feel more than free to investigate them yourself to see whether they might match your requirements.

The first two types of UI libraries are opinionated and already have designs for a button. These two libraries work best if you want your buttons to look they way they prescribe. You do have some wiggle room, but if you want a unique design, you'll rarely go for one of these types, which is why I feel that they come up short in this instance.

The last type—the utilitarian approach—is getting a lot of traction lately. Tailwind CSS is popular because you can get started fast and don't have to write any CSS, though you still have almost the full capabilities of CSS at your disposal.

The type of application we're building here, however, does not lend itself well to the Tailwind approach. Because we have a single component with a lot of variability and styles to apply, using Tailwind would get messy. I implemented the target application in Tailwind for reference, however, and you can see the result in the repository.

EXAMPLE: TAILWIND

This example is in the `tailwind` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch07/tailwind
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch07-tailwind**.

7.2 The example button application

We need a good example project to test the solutions. Because we want to demo a wide variety of the concerns mentioned in the preceding sections, we cannot create a simple application that contains all of them. If we wanted to go that way, we'd have to create an application with at least 15 to 20 components, to illustrate all the aspects that we want to highlight. Although React can handle that task, book pages are less flexible, so we'd need a lot of pages for the source code for each example application.

So rather than go for a single application, we're going to create a minimal UI library and some example uses of said library with a single UI component: a button. In essence, we're going to create five variants of our button, allowing the application to style the button slightly differently in different use cases (figure 7.2).

Figure 7.2 The target application that we are going to implement in three ways. The application is only five versions of a button, but it will give us a ton of information about and insight into the technology in question.

Imagine that we know about a target application in which our button will be used. The button must serve several purposes throughout the application, and we have been given some design specifications that our UI library and button must support.

I'll go over the variants in this section and explain which problems they deal with. Note that we're going to challenge mostly the CSS language feature problems, as those are the ones we can see directly. As we go about implementing those variants, we'll discuss which other problems the various solutions make easy or hard to solve. The five variants of a button that our UI library has to support are as follows:

- *The normal button*—This button has a pink background with white text and a forward-pointing chevron. When you hover over the but-

ton, the chevron bounces forward in a looping animation. The chevron also needs some styling to be sized and positioned correctly.

- *The outline button* —This button is used as an alternative, less important button. it might be used for a confirm box, in which the confirm button is the primary one and the cancel button is this outline button. This outline button has a pink outline, pink text, and a transparent background.
- *The disabled button* —We need the button to work as a submit button for forms as well, so we need the button to be shown in a disabled state. When it's disabled, the button does not receive any pointer events (including hover), and it's faded to 50% opacity.
- *The wide button* —In a few places, we're going to need a button that has a custom width rather than the fixed width of the normal button. We can use this button in a shopping cart sidebar, where we want the checkout button to be as wide as the sidebar. We'll implement this button by passing in a `width` property to the button and use that value as a pixel value for the button width.
- *The custom button* —On a few occasions, we need to be able to customize the button and break all the regular rules if we so desire. We may want the button to be taller, a different color from normal, to have rounded corners, or a box shadow. We want to be able to pass custom styles to the button and have them applied.

Note that all these variants can be applied together. Thus, we could use the UI library to create a disabled outline button with a custom width or an outline button with a border radius applied via the custom styling option.

We're going to create a simple application for all the examples, using the button from our simple UI library (refer to figure 7.2). We'll create the applications identically, following this general recipe:

```
import Button from "./Button";
function App() {
  return (
    <>
      <fieldset>
        <legend>Normal button</legend>


        <Button>Send</Button>     #1
      </fieldset>
      <fieldset>
        <legend>Outline button</legend>
        <Button outline>Send</Button>    #2
      </fieldset>
      <fieldset>
        <legend>Disabled button</legend>
        <Button disabled>Send</Button>    #3
      </fieldset>
      <fieldset>
        <legend>Wide button</legend>
        <Button width={400}>Send</Button>    #4
      </fieldset>
      <fieldset>
        <legend>Custom button</legend>
        <Button hasCustomStyle>Send</Button>    #5
      </fieldset>
    </>
  );
}
export default App;
```

#1 The normal button instance is passed zero properties.
#2 The outline button instance is passed only a Boolean outline property.
#3 The disabled button instance is passed only a Boolean disabled property.
#4 The wide button is passed a width property with the value 400.
#5 The custom button is going to be passed custom styles in some way or another. This implementation will vary with the choice of styling methodology. The hasCustomStyle property is only an example, of course; we would need to pass those custom styles to the component in some fashion for it to make sense.

7.3 Method 1: CSS files and class names

Using CSS files and class names is the old method of styling any web application, and it's the well-established norm that all the other CSS-in-JS technologies are rebelling against. This method can be considered the

nonmethod. It is how “classic” web applications are built, so we can build our React-based web applications that way too.
I include this method so you can better understand what the other approaches are rebelling against, when it still makes sense to use this classic approach, what it does well, and, in particular, what it doesn’t do well.

7.3.1 How class names work

I don’t need to explain this part, which is how CSS has been done for 25 years, so I trust you know how it works. Figure 7.3 illustrates the relationship between JSX and the applied style.

Do note one thing, however: you have no guarantee that `card.css` contains the class names used in `Card.js`. It could be referring to any class names defined in any CSS files imported anywhere in the project. Often, it makes sense to group JavaScript and CSS files as shown in figure 7.3, but grouping isn’t a requirement and isn’t enforced by React or the bundler. I’ll get back to that topic in section 7.3.5.

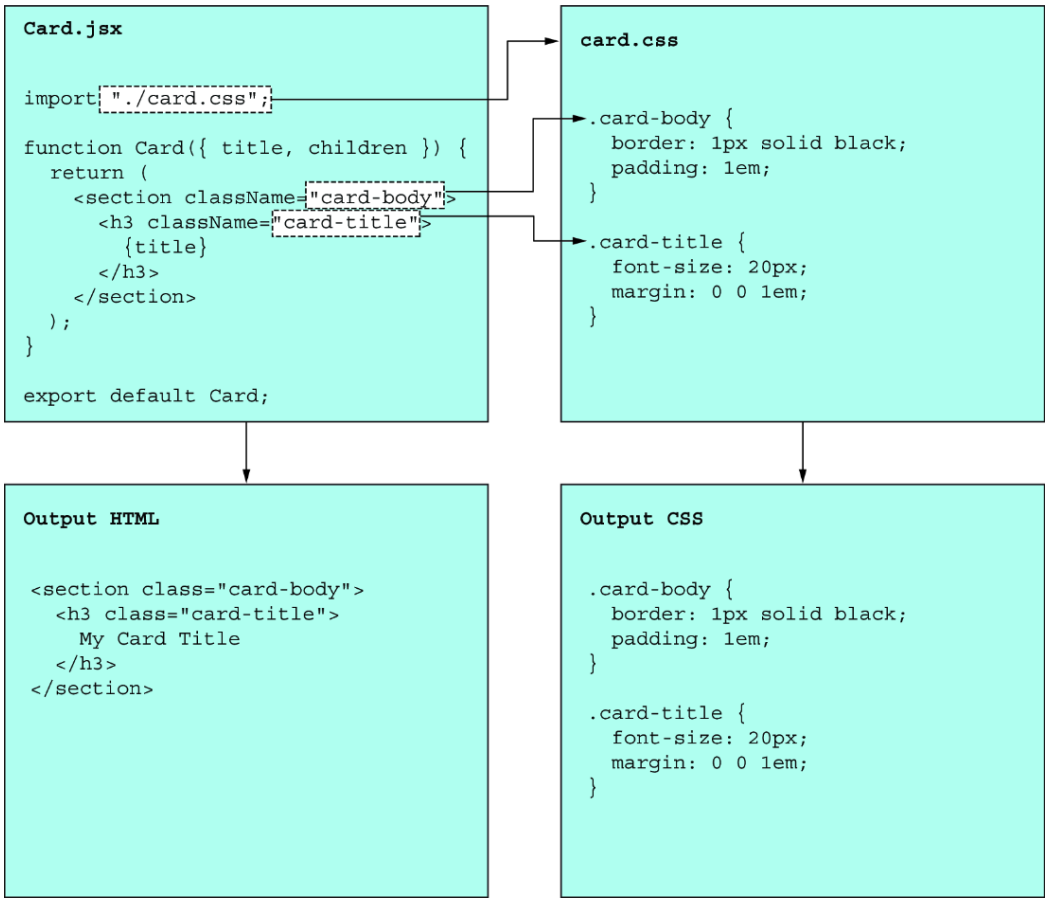

Figure 7.3 How JSX elements are styled when you use CSS files and class names. You can import a CSS file directly from a JavaScript file and then use CSS class names directly in your JSX. The resulting HTML and CSS looks exactly like what you type.

7.3.2 Setup for class name project

If you use Vite to initialize your project, you can use CSS files and class names right out of the box. The only slightly tricky thing about using CSS files in React development is that you can import the CSS files directly in the JavaScript files. Normally, you would have an HTML file with references to both CSS files and JavaScript files, but because we use a bundler (esbuild, built into the Vite setup), we can "import" the CSS files in the JavaScript files, and the bundler will turn the combined code into separate CSS and JavaScript files, correctly inserted into the document. Note that this process involves some technical details that your setup does in the background, so you don't have to worry about it.

CSS LOADING IN THE BUNDLER

If you don't use Vite and your React setup doesn't support automatic CSS bundling, you will have to figure out how to set up a CSS loader that works with your bundler. If you don't use a bundler, you can reference CSS files directly in your HTML file as usual.

7.3.3 Implementation with class names

Implementing the example application and button with class names is fairly straightforward. We simply create CSS files with the desired rules and apply the proper class names to the proper elements.

If we were building a real UI library, we would have many components, not just a button, and we'd have very clear separation of the library and the application. In this example, we have only two files, and we pretend that one of them is the UI library. To keep up the appearance of our button being inside a separate UI library, we are going to create two CSS files: one for the button, with all the normal button variant CSS, and another for the application itself, with the styles for the custom button that we're going to use for the fifth variant. All we have left to do is figure out which class names to apply to the button based on the relevant properties.

EXAMPLE: CLASSNAMES

This example is in the `classnames` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch07/classnames
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch07-classnames**.

APP.JSX

In the main application, we load the styles for our custom button and apply the class name to that custom button. Otherwise, we use the predetermined button variants. The source for `App.jsx` is in the following listing.

Listing 7.1 `src/App.jsx` with class names

```
import Button from "./Button";
import "./app.css";    #1
function App() {
  return (
    <>
      <fieldset>
        <legend>Normal button</legend>
        <Button>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Outline button</legend>
        <Button outline>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Disabled button</legend>
        <Button disabled>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Wide button</legend>
        <Button width={400}>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Custom button</legend>
        <Button className="custom-button">    #2
          Send
        </Button>
      </fieldset>
    </>
  );
}
export default App;
```

#1 First, we load the application stylesheet by importing the file.

#2 Then we add a custom class name to the custom button.

APP.CSS

The application stylesheet could contain other rules, but for this simple application, it contains only the rule for the custom button. Note that there's no direct synchronization between the rules defined in this CSS file and the rules used in the main application; we have to maintain that synchronization manually. If we misspell the class name in either file, no error will appear at either compile time or run time. The only indication

of an error would be the fact that the button isn't styled. The source for `app.css` is in the following listing.

Listing 7.2 `src/app.css` with class names

```
.custom-button {     #1
  background-color: purple;
  border-color: purple;
}
```

#1 The rule for a custom purple button

BUTTON.JSX

This section shows how to choose the styles to apply to the button element based on the properties of the component. We solve the normal button and the outline button by applying class names. We solve the disabled button in CSS only through pseudoclasses. Finally, we solve the wide button with inline styles. We could have used a CSS variable, but the result would have been comparable in terms of complexity. The source for `Button.jsx` is in the following listing.

Listing 7.3 `src/Button.jsx` **with class names**

```
import "./button.css";    #1
function Button({
  children,
  outline = false,
  disabled = false,
  className = "",
  width = null,
}) {
  const classNames = [    #2
    "button",     #3
    outline ? "button--outline" : "",    #4
    className,    #5
  ].filter(Boolean);
  const style = width     #6
    ? { width: `${width}px` }   #6
    : null;
  return (
    <button
      className={classNames.join(" ")}     #7
      style={style}    #7
      disabled={disabled}
    >
      {children}
      <svg
        className="button__icon"     #8
        version="1.1"
        xmlns="http://www.w3.org/2000/svg"
        xmlnsXlink="http://www.w3.org/1999/xlink"
        viewBox="0 0 490 490"
      >
        <polygon
          points="240.112,0 481.861,245.004 240.112,490
                  8.139,490 250.29,245.004 8.139,0"
          fill="currentColor"
        />
      </svg>
    </button>
  );
}
export default Button;
```

#1 We need to import the style sheet, and we check manually which class names are used in the stylesheet.

#2 We generate a final class name for the button element by creating an array of

class names and merging them.

#3 First, we always apply the button class name.

#4 Then we apply the button--outline only if the outline property is true.

#5 Finally, we append any custom class name to the component as well.

#6 We also have to create a style object if there is a custom width property.

#7 We apply the class name and style object to the button element, and we're all set.

#8 The inline Scalable Vector Graphics (SVG) element also needs some styles, which we apply with a simple class name.

BUTTON.CSS

This file is plain CSS, using all the rules and declarations we need to style the button. The source for `button.css` is in the following listing.

Listing 7.4 `src/button.css` with class names

```
.button {
  display: flex;
  background: hotpink;
  color: white;
  border: 2px solid hotpink;
  margin: 1em 2em;
  padding: 1em 2em;
  font-size: 120%;
  width: 250px;
  cursor: pointer;
  gap: 10px;
  justify-content: center;
  align-items: center;
}
.button__icon {
  width: 16px;
  height: 16px;
  position: relative;
}
.button:hover .button__icon {     #1
  animation: bounce .2s ease-in-out alternate infinite;
}
@keyframes bounce {     #2
  from {
    left: 0;
  }
  to {
    left: 10px;
  }
}
.button--outline {
  border-color: hotpink;
  background-color: transparent;
  color: hotpink;
}
.button:disabled {   #3
  opacity: 0.5;
  pointer-events: none;
}
```

#1 We can use pseudoclasses and composition.

#2 We can use at-rules.

#3 We can use pseudoclasses.

7.3.4 Strengths of the class names approach

The strengths of this approach are relatively obvious, but a few of them may come as a surprise due to the way the application is bundled:

- We can use all features of CSS because that's what we're writing—plain old CSS.
- This approach is familiar and understandable to all team members who have ever tried to use CSS. You don't have to do any onboarding; the team is up and running in seconds.
- Debugging is easy because you can see which classes apply to which elements in the browser and (ideally) find the same classes in your CSS files somewhere. Which CSS file includes a specific class may not be obvious, but a search will probably reveal it. If your bundler is set up correctly, source maps in your browser developer tools may tell you which file and line contained the original style (even though your bundler will have merged the files into a different CSS file).
- Because you can import stylesheets into the individual components that use them, you get some level of maintenance and code splitting for free. If you do not include a specific component in your application, the stylesheets included in that component will not be bundled with your application, so you won't be loading any unused styles. Also, if you lazy load (delay load) some components to optimize browser rendering, the styles for those components will also be lazy loaded.
- This approach is fast, with almost no overhead in compile time mostly because your CSS files aren't analyzed; they're included in the bundle as is. Also, this approach has zero runtime overhead because it is regular CSS files and class names. Nothing has to be calculated on the fly.

7.3.5 Weaknesses of the class names approach

The topic of this section is the entire reason why CSS-in-JS exists. This regular approach, although it's easy to work with and well understood, has several weaknesses. For many developers, those weaknesses are significant enough that they've created hundreds of alternative methods for styling complex web applications. Let's deal with the most important weaknesses first:

- *There is no synchronization between CSS files and components.* Yes, you might include a `button.css` file in your `Button.js` component file, but you have no guarantee or validation that the styles included in the

CSS file are the ones used in the component, and vice versa. If you misspell a class name, nothing helps you check it. If you have unused classes in your CSS file, no pruning occurs.

- *The split of the component between a CSS file and a JavaScript file requires a lot of context switching.* As you're developing the component, you have to switch back and forth between the two files and remember class names and HTML structure to make sure that you style your application as desired.
- *Collisions are likely.* As your application grows, many components might be named `Group` because the concept of a group can be used in many contexts. But you can have only one CSS class named `.group`. If you accidentally reuse that class name in a new component, errant styles will apply to your component (and you'll probably have broken the original). This problem can be tricky to track down. Even worse, if you embed your React application in a regular website (maybe you created a small calculator module that can be reused on many pages), the CSS classes used on the website might collide with the classes in the application, and vice versa. You have to apply some complex naming schemes to prevent collisions (one popular approach is BEM [**http://getbem.com**]), but even that precaution probably won't be enough.
- *If you need to use dynamic values, you need to use inline CSS.* This situation may occur if you have a `width` variable in JavaScript that you need to apply to an element.
- *Specificity can be a problem.* Rules defined in the same CSS file will be output in that order in the final document. But in what order will rules defined in different CSS files be output? A system exists, but it can be hard to understand. We happen to be lucky in this case because the rule for `.custom-button` apparently was defined after the rule for `.button`, but we can't depend on luck at scale.

Following are a few lesser but still important problems:

- *If you use basic CSS as I advocate in this section, you're not using a preprocessor.* That is, the CSS you write is the CSS that will be used in the browser. Preprocessors such as Less and Sass allow you to use fancy features that aren't supported in normal CSS. In this setup, none of those options is available, so you're limited to what CSS normally offers. To a CSS purist, that situation might be a plus, but to most other developers, it's a minus.

- *Theming is tricky.* You can use CSS variables, which all major browsers today support, but you have to manage your variables manually and remember the names of the variables that will be used throughout your application.
- *This approach scales acceptably, but only to a point.* Because of the single global namespace, collisions make collaboration difficult on a large scale.
- *Dead-code removal has to be done by hand.* It is impossible to tell automatically whether a given rule is used anywhere in your application.

7.3.6 When (not) to use CSS files and class names

This approach is good for small-to-medium-size projects that aren't too complex or dynamic. When your application, team, and ambitions grow, along with the need for more complex structures, you will probably have to look for a different solution. You will also find that most established teams use some other approach, for all the reasons stated in section 7.3.5.

7.4 Method 2: CSS Modules

This approach is a small extension of the regular CSS file and class name approach that addresses some of the problems I pointed out in section 7.3.5. This concept is called *CSS Modules*, and it primarily deals with the problems of colocation and collisions. CSS Modules still use CSS files, which you write as you normally would. But those files will be parsed, and the class names will be extracted and regenerated as new unique names. All other aspects stay the same.

7.4.1 How CSS Modules work

The syntax and setup for using CSS Modules is almost identical to using class names and CSS files. We are still using CSS files, but we're going to make sure that they use the `.module.css` extension rather than `.css`. The file extension `.module.css` instructs the bundler to handle them using the CSS Modules plugin instead of the regular CSS plugin. Figure 7.4 shows how we're going to be using this approach.

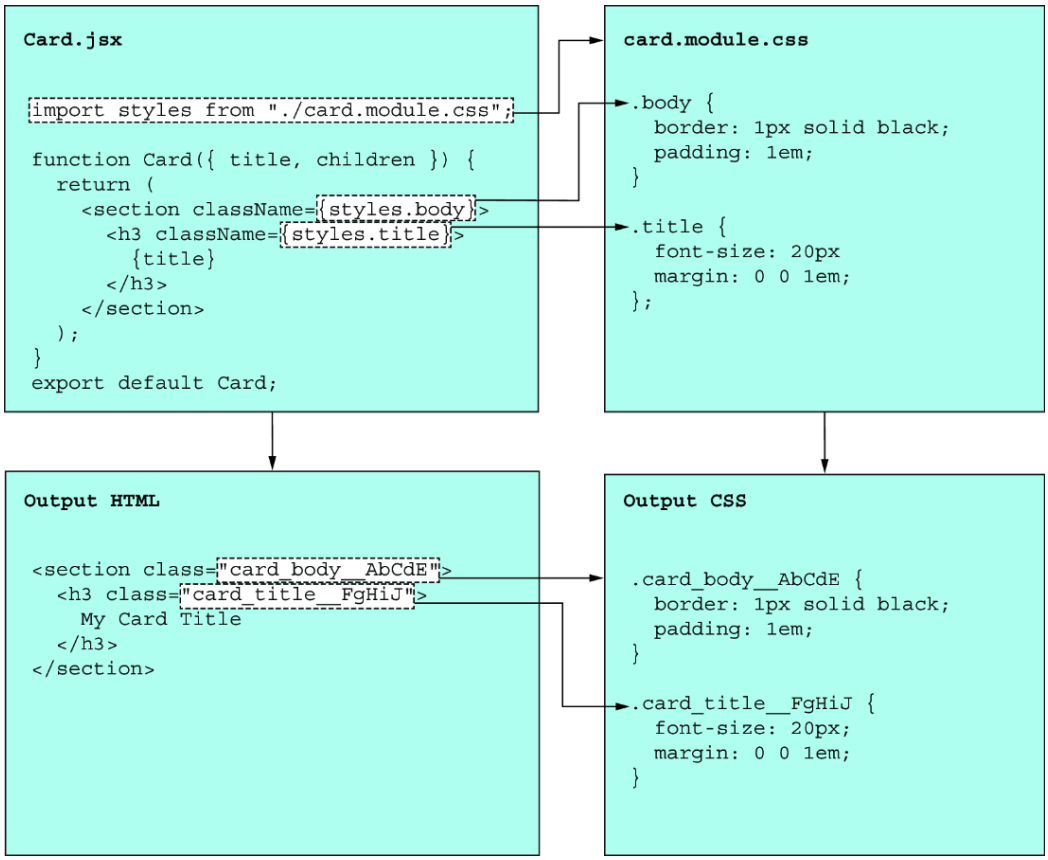

Figure 7.4 The relationship between the component and the CSS Module. We can use whatever class names we want in the module, and we refer to those class names in the component as direct variables. The resulting output will replace these class names in both the CSS file and the module with some autogenerated class names, partially based on the CSS Module filename, the class name, and some randomly generated padding to prevent collisions.

As you see in figure 7.4, CSS Modules is a clever library. When you import a CSS Module into a JavaScript file, the bundler converts the CSS files to an object with a key for each class defined in the module, and each key maps to a dynamic class name. This dynamic class name is generated based on the module filename, the class name in the module, and some random information.

7.4.2 Setup for CSS Modules project

Support for CSS Modules is built into any Vite setup by default. All you have to do is name your CSS files `*.module.css`, and you're good to go.

If you aren't using a Vite-based setup, you need to find a plugin for CSS Modules that works with your bundler. For webpack, you can use the `css-loader` plugin, and for everything else, you can use `postcss` and the `postcss-modules` plugin:

- `css-loader` —https://github.com/webpack-contrib/css-loader
- `postcss` —https://github.com/postcss/postcss
- `postcss-modules` —https://github.com/madyankin/postcss-modules

7.4.3 Source code with CSS Modules

Implementing the button application with CSS Modules is a straightforward evolution from the class name version. We simply rename the CSS files from `*.css` to `*.module.css`; we simplify the CSS class names because we no longer have to worry about collisions; and we replace the static filenames with the imported dynamic style names. In this section, I'll go over the files to show how we do all these things.

EXAMPLE: CSSMODULES

This example is in the `cssmodules` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch07/cssmodules
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: https://reactlikea.pro/ch07-cssmodules.

APP.JSX

We change the CSS import and the filename passed to the custom button. Otherwise, everything is standard. The source for `App.jsx` is in the following listing.

Listing 7.5 `src/App.jsx` with CSS Modules

```
import styles from "./app.module.css";    #1
import Button from "./Button";
function App() {
  return (
    <>
      <fieldset>
        <legend>Normal button</legend>
        <Button>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Outline button</legend>
        <Button outline>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Disabled button</legend>
        <Button disabled>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Wide button</legend>
        <Button width={400}>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Custom button</legend>
        <Button className={styles.customButton}>     #2
          Send
        </Button>
      </fieldset>
    </>
  );
}
export default App;
```

#1 We import the module to the named variable styles.
#2 We use the styles variable to apply the dynamic class name to the custom button.

APP.MODULE.CSS

The only change we make in the main CSS file of the application (besides the filename) is to use a class name that works better in JavaScript as an identifier. Before, the class name was `custom-button`, and although we can use it in JavaScript as `styles['custom-button']`, it is easier to type `styles.customButton` if we change the class name similarly.

We also need to change another thing. When we imported the CSS files in the previous application by using plain CSS files, we were lucky that the files were appended in the correct order, with the button CSS first and the app CSS second. That result meant that the app CSS applied last and thus had higher specificity. For this CSS Module application, the loader happens to load our rules in the opposite order, so the app CSS rules are loaded first, which means that they have lower specificity. To make sure that the custom button rule is applied to override the regular button styles, we need to increase the specificity of the selector or make the declarations important. We choose to do the former, using a tag name, so the final selector becomes `button.customButton`. The source for `app.module.css` is in the following listing.

Listing 7.6 `src/app.module.css` with CSS Modules

```
button.customButton {     #1
    background-color: purple;
    border-color: purple;
}
```

#1 Before, we had .custom-button. Now, we rename it .customButton to make it work better as a JavaScript identifier. Finally, we add the button tag name to raise the specificity of this rule; otherwise, it would not work correctly. We could also have solved this problem with !important.

BUTTON.JSX

The implementation of the button using CSS Modules is almost identical to the implementation using class names, except that we get the class names from the import rather than the strings. Changing the import and the class name is generally all it takes to switch to using CSS Modules. The source for `Button.jsx` is in the following listing.

Listing 7.7 `src/Button.jsx` with CSS Modules

```
import styles from "./button.module.css";     #1
function Button({
  children,
  outline = false,
  disabled = false,
  className = "",
  width = null,
}) {
  const classNames = [
    styles.normal,     #2
    outline ? styles.outline : "",    #2
    className,
  ].filter(Boolean);
  const style = width     #3
    ? { width: `${width}px` }    #3
    : null;
  return (
    <button
      className={classNames.join(" ")}
      disabled={disabled}
      style={style}
    >
      {children}
      <svg
        className={styles.icon}
        version="1.1"
        xmlns="http://www.w3.org/2000/svg"
        xmlnsXlink="http://www.w3.org/1999/xlink"
        viewBox="0 0 490 490"
      >
        <polygon
          points="240.112,0 481.861,245.004 240.112,490
                  8.139,490 250.29,245.004 8.139,0"
          fill="currentColor"
        />
      </svg>
    </button>
  );
}
export default Button;
```

#1 We import the class names from the CSS Module.

#2 We change all the instances of string class names to the class names returned

by the import.

#3 We still need inline styles for dynamic values.

BUTTON.MODULE.CSS

The button CSS is identical except that we can use less complex class names because we don't have to worry about collisions. We can use `.outline` for the outline button rather than `.button--outline` or `.buttonOutline`. The source for `button.module .css` is in the following listing.

Listing 7.8 `src/button.module.css` with CSS Modules

```
.normal {
    display: flex;
    background: hotpink;
    color: white;
    border: 2px solid hotpink;
    margin: 1em 2em;
    padding: 1em 2em;
    font-size: 120%;
    width: 250px;
    cursor: pointer;
    gap: 10px;
    justify-content: center;
    align-items: center;
}
.icon {
    width: 16px;
    height: 16px;
    position: relative;
}
.normal:hover .icon {    #1
    animation: bounce .2s ease-in-out alternate infinite;
}
@keyframes bounce {
    from { left: 0; }
    to { left: 10px; }
}
.outline {
    border-color: hotpink;
    background-color: transparent;
    color: hotpink;
}
.normal[disabled] {
    opacity: 0.5;
    pointer-events: none;
}
```

#1 Note that we can make selectors composed of multiple class names. CSS Modules will replace each class name with the correct output class name.

7.4.4 Strengths of CSS Modules

The strengths of using CSS Modules are the same as the strengths of using regular CSS and class names, but some of the weaknesses of the latter

method have been fixed. In particular, compared with the first method we tested, CSS Modules gives us the following benefits:

- *Synchronization between CSS files and components* —Because we're not betting on strings happening to match up but using actual variables, we can be sure that we use the correct class names in our components.
- *Look, Mom, no collisions* —Because the final class names will be auto-generated, we don't have to worry about coming up with unique names for our classes. We can have a `.wrapper` class in every component, and the bundler will make sure that they all have unique names in the final product.
- *Easy dead-code elimination* —Dead-code elimination is often easier, as you can be fairly certain that a given rule is used only by components that import the CSS Module file in which it exists. If a CSS Module contains a rule that is no longer used in any component importing that module, it can be safely removed (most of the time, at least).

7.4.5 Weaknesses of CSS Modules

Some of the weaknesses of the classic, class names method persist in CSS Modules:

- We still have our component split into two files. Even for a tiny component, we need to switch back and forth between the JavaScript file and the CSS file to get the full picture.
- We still need to use inline CSS for dynamic values.
- Specificity and source order are still problems, as we aren't in control of the order of the CSS rules in the final output.

Additional weaknesses include the following:

- Debugging is harder. If you are testing the application in the browser and find a component that's wrongly styled, it's trickier to track down the offending CSS, as you can't search for the rule that you see in the output. The original rule in your source code will be different. Browser source maps will get you closer, but the process can still be tricky.
- CSS Modules are slightly more complex than plain CSS files, so new developers need an introduction to this system if they are unfamiliar with it.

- Composition of rules is harder. You cannot refer directly to class names from other modules, as those class names will not remain the same in the final output. So you have to isolate your styles inside each component and cannot apply styles inside a different component without using some weird tricks that are specific to CSS Modules.

7.4.6 When (not) to use CSS Modules

Despite the previously mentioned weaknesses, CSS Modules are a stable method for larger applications. When CSS Modules came out, they quickly gained traction as the best way to work with CSS in larger codebases, and you will still find it used in many codebases. That being said, it does have some weaknesses that other systems have solved better.

Suppose that you're on a project using CSS Modules, and the team is doing fine. In that case, you don't need to change things just for the sake of change. If you have the opportunity to define the styling method for a new project, however, CSS Modules may be what you want if you can live with or at least work around the weaknesses. If not, newer alternatives are available.

7.5 Method 3: Styled-components

Styled-components is the natural extension of CSS Modules, taking the concept to the next step. CSS Modules have CSS files and class names, but those class names are local variables, not class names in the final CSS. So why not skip having class names, create elements directly with a given set of styles already applied, and use those elements directly in your component?

What do I mean? Well, imagine that we could import the HTML-element-with-class directly from the CSS file as shown in figure 7.5.

CSS Modules

card.module.css

```
.body {
  border: 1px solid black;
  padding: 1em;
}

.title {
  font-size: 20px;
  margin: 0 0 1em;
}
```

Card.js

```
import styles from "./card.module.css";


function Card({ title, children }) {
  return (
    <section className={styles.body}>
      <h3 className={styles.title}>
        {title}
      </h3>
    </section>
  );
}
```

New fancy idea?

card.fancy.css

```
section.body {
  border: 1px solid black;
  padding: 1em;
}

h3.title {
  font-size: 20px;
  margin: 0 0 1em;
}
```

Card.js

```
import {
  Body, Title
} from "./card.fancy.css";

function Card({ title, children }) {
  return (
    <Body>
      <Title>
        {title}
      </Title>
    </Body>
  );
}
```

Figure 7.5 The structure on the left is what CSS Modules give us. But what if we could skip some of that extra work and go to the version on the right, where we don't have to type the class names and can import the CSS directly as components?

Figure 7.5 illustrates the general idea behind styled-components. Additionally, styled-components eliminates the need for CSS files and allows you to specify the components with CSS directly in the main JavaScript file.

7.5.1 How styled-components works

Styled-components allows you to define HTML JSX elements with styles applied directly in your component JavaScript files. See the example in figure 7.6.

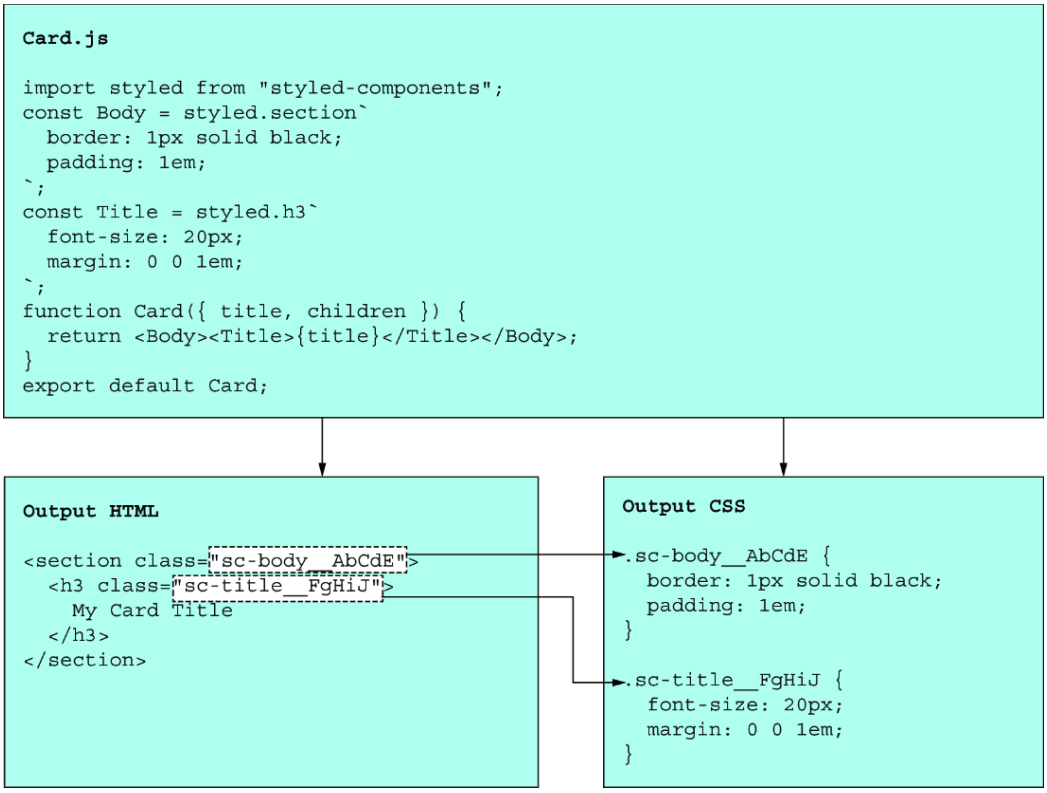

Figure 7.6 We have a single source file and can define CSS directly applied to the relevant JSX elements. The file is a bit more complex overall, but the resulting JSX is clean.

You might say, “Well, that’s good for these examples, but what about pseudoclasses, composition, and so on?” Don’t worry. Styled-components has got you covered with nested syntax from tools such as Less and Sass.

STRING TEMPLATE LITERALS

How does this backtick syntax in `styled.section``` work?

Backticks are used in JavaScript to create *string template literals*, a great way to create complex multiline strings, especially with interpolation. You can interpolate regular JavaScript variables into a string template literal with the `${}` syntax like so:

```
const name = 'World';
const message = `Hello ${name}!`;
```

What's this extra notion of adding the backticks right after another variable, as in `styled.section``` ? This syntax is a *tagged template literal*. It allows you to call a function with the literal parts of the string and the interpolated values passed as arguments. It's complicated to explain and implement, but the result is a smooth experience, as you will see in this section. Read more about the concept at **https://mdn.io/template-literals**.

We can pass any type of JavaScript inside the template by using interpolation, even functions, as long as the template tag function knows how to interpret it:

```
const message = messageBuilder`Hello ${person => person.name}!`;
```

This example doesn't do anything on its own, as it requires the `messageBuilder` function to call the passed function, but it illustrates the process, which we'll be using for styled-components.

If you need to add a pseudoclass such as `:hover` or `:disabled`, you do it like so:

```
const Submit = styled.button`
  background-color: red:
  &:hover {
    background-color: blue;
  }
`;
```

When you want to create nested rules, you use the interpolation syntax for the template literal like so:

```
const Icon = styled.img`
  width: 32px;
`
const Submit = styled.button`
  background-color: red:
  & ${Icon} {
    width: 24px;
  }
`;
```

You can combine all these tricks to create all the rules you need. You can even create dynamic rules that use properties passed to the components to generate the CSS on the fly, using interpolation with a function receiving all components passed:

```
const Submit = styled.button`
  background-color: ${({ color }) => color};
`;
function MyForm() {
  return (
    <Submit color="red">Red submit!</Submit>
  );
}
```

You can also extend a definition by wrapping another component in the `styled` function:

```
const Submit = styled.button`
  background-color: green;
`;
const AngrySubmit = styled(Submit)`
  background-color: red;
`;
```

Here, we create a component with `styled``` and then wrap that component in `styled()`. Styled-components will automatically stack the two rules on top of each other with the proper specificity and source order. But if you wrap a custom component that you create yourself with

`styled()`, the library will generate a class name for you and pass it into the component in question as a `className` property. We will use this trick in listing 7.9.
When it comes to at-rules, even more tricks are available. If you need to create keyframes for an animation, you can use a specific function (which we'll look at in listing 7.10). For media queries, you can create those inline as well like pseudoclasses. Finally, if you still need to create global rules (maybe you need to style the `html` or `body` element), another function for adding simple global CSS applies everywhere.

That's a lot of information in a short time. I'm only trying to give you a sense of what you can do and how you can do it, and I'm still covering only the bare minimum capabilities. There's a lot to learn about styled-components, and the best way is to read the documentation at **https://styled-components.com**.

7.5.2 Setup for styled-components project

Support for styled-components is not baked into Vite, but it is still easy to install. You simply install the module by using npm:

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

Then you can start importing from the package and using it as we saw in section 7.5.1.

7.5.3 Source code with styled-components

React components implemented with styled-components tend to be lengthy because the CSS is included in the same file as the React component code. This length is a feature of the library, however; related bits of code are placed as close together as possible to make updating the component easier. If you are restyling a component, you'll likely be changing both the CSS and the HTML (JSX) structure. With this library, both CSS and HTML are conveniently located next to each other.

EXAMPLE: STYLED

This example is in the `styled` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch07/styled
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch07-styled**.

APP.JS

To apply a custom style to another component, we use the trick mentioned in section 7.5.1: wrapping a component in the `styled` function and applying the CSS by using the tagged template. Then our component will be invoked with a class name that we pass to the relevant JSX. The latter part goes inside the button. For now, we're worried only about how to style the button. The source for `App.js` is in the following listing.

Listing 7.9 `src/App.js` with styled-components

```
import styled from "styled-components";
import Button from "./Button";
const CustomButton = styled(Button)`     #1
  background-color: purple;    #2
  border-color: purple;    #2
`;
function App() {
  return (
    <>
      <fieldset>
        <legend>Normal button</legend>
        <Button>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Outline button</legend>
        <Button outline>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Disabled button</legend>
        <Button disabled>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Wide button</legend>
        <Button width={400}>Send</Button>
      </fieldset>
      <fieldset>
        <legend>Custom button</legend>
        <CustomButton>Send</CustomButton>    #3
      </fieldset>
    </>
  );
}
export default App;
```

#1 We add custom styles to an existing component by wrapping it in the styled function and get a new "enhanced" component back.
#2 We can include the CSS directly inline.
#3 Finally, we use this "enhanced" component like normal in JSX.

BUTTON.JSX

This part is where all the magic happens. There's no CSS file anymore; all the CSS exists inline in the button component JavaScript file.

Note that we add an animation to the CSS. We also specify that the component will still accept a class name because, under the hood, styled-components still generates CSS files with classes and applies those classes to the elements in question. The source for `Button.jsx` is in the following listing.

Listing 7.10 `src/Button.jsx` **with styled-components**

```
import styled, { keyframes } from "styled-components";
const Icon = styled.svg`
  width: 16px;
  height: 16px;
  position: relative;
`;
const bounce = keyframes`     #1
  from { left: 0; }    #1
  to { left: 10px; }    #1
`;   #1
const Normal = styled.button`
  display: flex;
  background: hotpink;
  color: white;
  border: 2px solid hotpink;
  margin: 1em 2em;
  padding: 1em 2em;
  font-size: 120%;
  width: 250px;
  cursor: pointer;
  gap: 10px;
  justify-content: center;
  align-items: center;
  ${({ $width }) =>
    $width && `width: ${$width}px`
  };     #2
  &:hover ${Icon} {     #3
    animation: ${bounce} 0.2s ease-in-out     #4
      alternate infinite;   #4
  }
  &[disabled] {   #3
    opacity: 0.5;
    pointer-events: none;
  }
`;
const Outline = styled(Normal)`    #5
  border-color: hotpink;
  background-color: transparent;
  color: hotpink;
`;
function Button({
  children,
  outline = false,
  disabled = false,
```

```
    className = null,
    width = null,
  }) {
    const Element = outline ? Outline : Normal;
    return (
      <Element #6
        disabled={disabled}     #6
        $width={width}   #6
        className={className}     #6
      >  #6
        {children}  #6
        <Icon  #6
          version="1.1"
          xmlns="http://www.w3.org/2000/svg"
          xmlnsXlink="http://www.w3.org/1999/xlink"
          viewBox="0 0 490 490"
        >
          <polygon
            points="240.112,0 481.861,245.004 240.112,490
                    8.139,490 250.29,245.004 8.139,0"
            fill="currentColor"
          />
        </Icon>
      </Element>
    );
  }
  export default Button;
```

#1 To add an animation with custom keyframes, we use the util function keyframes from the styled-components library.
#2 We can include custom properties from the JSX in the generated CSS by using interpolated functions, which have access to all the properties.
#3 We can use composed rules, nested rules, pseudoclasses, and all sorts of selectors directly in the CSS declaration.
#4 Uses the bounce animation by interpolating the value returned from the keyframes function
#5 We can even extend one class with the rules of another (so this element will have both classes applied) to avoid duplicating rules.
#6 When we use an element styled with styled-components, we can pass in HTML properties (disabled and className) as well as properties used to generate the resulting CSS ($width). Prefixing properties with $ is a way to make sure that they don’t go into the output HTML. $-prefixed properties are called transient properties.

7.5.4 Strengths of styled-components

Proponents of styled-components often promote several attributes as key benefits of this framework. Here are the highlights:

- Colocation is superb because you have the styles right next to the elements you're styling.
- Specificity is not a problem because the library takes care of it for you. If you extend one rule by another rule, the library makes sure to include them in the right order so that the rules apply correctly.
- You can use the full feature set of CSS seamlessly.
- No collisions occur, so you never have to worry about coming up with unique class names or meaningful selectors.
- Nesting is often eliminated or kept to two levels, making your selectors much shorter, which in turn means that the browser can apply the CSS much faster.
- Because you write the CSS inside JavaScript files, you can use JavaScript to do more complex calculations if you need to.

7.5.5 Weaknesses of styled-components

This library has a a couple of drawbacks, of course, and I want to highlight the following:

- Because the CSS is generated at compile time, extensive use of styled-components makes your application build slower. The overhead normally is not significant, but it is something to watch out for.
- There is an unclear separation of concerns, with CSS "logic" spread between both components and CSS rules. If you have conditional values, you can place the condition in either CSS or JavaScript; different developers on the same team might do things differently.
- Debugging is a bit harder because you just see a class name in the production HTML and can't immediately track it back to a CSS file, but you have to rely on source maps.
- Because you have CSS directly inside the component files, these files tend to get longer—sometimes a lot longer. Compare `Button.jsx` in the class names example (~40 lines) with the `styled` example (~70 lines). Generally, you can mitigate more lines of code by creating more, smaller components or even moving all the `styled` components to a separate file (such as `Component.styled.js`) and importing the elements from there.

- Because you are writing CSS-in-JS, it can be hard to revert to plain CSS files when you have a complex codebase, so evaluate your options carefully before committing to a library.
- It can be hard to see whether a JSX node is a component or a styled HTML element. You might see `<Heading>Expenses</Heading>` in a component and not know whether `Heading` is a styled HTML element or a different complex component. But maybe it's okay that you can't tell. Also, modern editors often give you inline hints about the element source.

7.5.6 When (not) to use styled-components

Styled-components is great for complex designs with many one-off components but is often considered less ideal for component libraries or web applications with unified and streamlined designs, such as dashboards and admin interfaces. The library is versatile and popular, so you can't go wrong with using it, but you may want to consider the alternatives before committing to it on a huge project.

7.6 One problem, infinite solutions

As I mentioned earlier, there is no one best way to style a React application because applications and developers are different. What works for one application and one team might be a terrible choice for another team.

In this section, I summarize five ways of styling an application:

- Inline CSS
- CSS files and class names
- CSS Modules
- Styled-components
- Tailwind CSS

I included the full source code for these methods in the online repository so you can check it out for yourself. I carefully dissected three of these approaches in this chapter; I implemented the other two (inline CSS and Tailwind CSS) but didn't describe them here to save space. Those implementations are linked in sections 7.1.4 and 7.1.5 if you want to check them out.

Table 7.1 is my admittedly subjective review of various attributes. I rate each attribute as being a feature of the approach (✓), a problem (✗), or something that the approach is neutral about or that has both ups and downs (!).

Table 7.1 Evaluating CSS methods on several attributes

Category and attribute	Inline CSS	CSS files	CSS Modules	Styled-components	Tailwind CSS
Folder	`inline`	`classnames`	`cssmodules`	`styled`	`tailwind`
CSS language features					
Specificity	!	✗	!	✓	✗
Collisions	✓	✗	✓	✓	✓
Composition	✗	!	✓	✓	!
Pseudoclasses	✗	✓	✓	✓	!
Pseudoelements	✗	✓	✓	✓	!
At-rules	✗	✓	✓	✓	!
Conditional rules	✗	✗	✗	✓	✓
Dynamic rules	✓	✗	✗	✓	✓
Vendor prefixes	✗	✗	✗	✓	✓
DX					
Colocation	✓	!	!	✓	✓
Familiarity	✓	✓	✓	!	✗
Simplicity	!	✓	!	!	✗
Scaling	✗	✗	!	✓	✓

Category and attribute	Inline CSS	CSS files	CSS Modules	Styled-components	Tailwind CSS
Readability	!	✓	✓	!	✗
Debugging	✓	✓	!	✓	!
Encapsulation	✗	!	!	✓	✓
Theming	✗	!	!	✓	✓
Web application development					
Bundle size	✗	✓	✓	!	!
Performance	✗	!	!	✓	✓
Cacheability	✗	✓	✓	!	✓
Code-splitting	!	!	!	✓	✓
Maintenance	!	✓	✓	✓	✗

You may notice that styled-components has the fewest drawbacks in table 7.1. I often use and strongly recommend this library for many projects, but I use different approaches in different projects, depending on other factors. Styled-components is like IBM: nobody gets fired for using it because it is safe. But check your alternatives.

One final note on this topic: these libraries are only a few of many. Dozens of other libraries are available, with all sorts of approaches developed by teams and developers who had all sorts of weird requirements or restrictions. New libraries pop up every month. By the time you read this book, a new approach might be the cool kid on the block. So although no one solution fits every project, at least some solution is likely to fit each individual project.

Summary

- You have many ways to style a web application in general and a React application in particular.
- CSS-in-JS refers to different ways of writing CSS inside your JavaScript files. It's a popular, easy way to write components and styles at the same time. But not all styling libraries use this approach.
- Using CSS files and class names is the old approach but still valid for smaller projects. Although it doesn't scale well, this tried-and-true approach doesn't require extra training for new team members.
- CSS Modules are a minor extension to regular CSS files. Rather than plain class names as written in the CSS file, the CSS declarations are preprocessed, and autogenerated unique class names are used instead with minimal effect on DX. It doesn't solve all the problems of regular class names, but it scales better.
- Styled-components is a modern CSS-in-JS approach to web application styling. With styled-components, you write your CSS directly inside your JavaScript files next to the components that need it. This approach leads to a much better overview of small components, as you can see JavaScript, HTML, and CSS at a glance. For larger components, however, the benefits diminish.
- Each individual library or approach has benefits and drawbacks. Although I can't recommend any single solution for every project, I recommend going with styled-components for medium-to-large-size projects because I feel that it has the most benefits and the fewest drawbacks. But remember that this recommendation is subjective and depends on a lot of things that I don't know about your project.