13 Project: Create a React UI library

This chapter covers

- Creating UI components from a product brief
- Composing a UI library by using Storybook
- Testing UI components exhaustively

> *"Hi, and welcome. It's great to see a new face in the office. Before we show you around, let's get right to the task we hired you for. We need a common UI library across all our React projects to make our brand look great and uniform and make it a lot easier to prototype new features across all our products quickly. We've started a library with some tools we'd like you to use, but it's all on you to expand it to all the components we need.*
>
> *"Our design team has created some visualizations for you, with all the pixels and hex codes you desire, and the product team has written up some specifications for each component so you know exactly what they're supposed to do and how they interface with the world and the user. Let me introduce you to the team. Come along!"*

With this fictional intro, pretend that you're a new hire at a company. Your job is to extend a budding new UI library using whatever tech stack this company happens to use.

Before we get to the tech stack, let's talk about UI libraries. There are already a ton of them out there, so why would you create your own? Why does every middle-size and larger company always create its own rather than repurpose something that's already available? The scenario in the chapter introduction is more common than you might think. It seems so wasteful to spend resources creating a huge component library from scratch when you can use an existing library. Teams do so anyway for several reasons. The two most important ones are

- *Ownership*—You want to own your stack, especially the most important bits of it. If you need a special variant of a component that's not supported by the external library you're using, you want to be able to fix it yourself, not depend on someone else to add it. Also, if you ever find a bug, you want to be able to fix it quickly without going through a third party.
- *Freedom*—When you build your own thing, you have complete freedom to do what you need. You may need a particular combination of components that no library currently supports, but when you roll your own, you can include what you need—nothing else.

A third, less important reason may be more personal. Not Invented Here syndrome is common mostly among junior developers who don't like to use an existing framework or library; they'd rather build one on their own. As you mature as a developer, however, you quickly realize that building a framework such as React or a library such as date-fns is immensely complex, and these projects have already seen and dealt with edge cases that you haven't even considered relevant.

Similar things happen to designers who want to create a unique style and make it their own. This wish is often at odds with how inflexible existing libraries are, so you end up having to create your own library. Luckily, this scenario is not one of the primary reasons for creating a custom UI library, but it's probably a common minor one, regardless.

That said, your job is to extend the UI library that your new employer created and follow the instructions you're given. In this chapter, I'll give you three challenges of increasing complexity. Your job is to create these components as designed and described, as well as follow the conventions and patterns of the existing stack. Before I get to what we're going to build, I'll show how I built the base library and, more importantly, how I'm testing it (exhaustively!).

With all that work ahead of us, let's hope that this new employer at least has a great Friday bar. I hear that they have karaoke!

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch13**.

13.1 The existing stack

The current stack is built on React. Phew! That's a load off your shoulders already because you already know how to use React. Right? You can check out the stack and the base library by using the following example in one of the usual ways.

EXAMPLE: BASE

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

```
$ npm run dev -w ch13/base
```

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/ch13-base**.

Here are all the technologies in the stack other than React itself:

- *Build tool*—esbuild
- *Bundler*—Vite
- *Icon library*—react-icons
- *Test runner*—Vitest
- *Test library*—Testing Library
- *Linter*—ESLint
- *Formatter*—Prettier
- *Visual test tool*—Storybook
- *Coverage reporter*—Istanbul

You should be very familiar with most of the components in this stack at this point. The first line is what every Vite setup is built on behind the scenes. You don't notice that esbuild is doing the work, but it has been working behind the scenes up until this point, so it shouldn't put you off.

The icon library is simply an icon library. It's a collection of collections of icons and easy to use, so I'm not going to cover it much more.

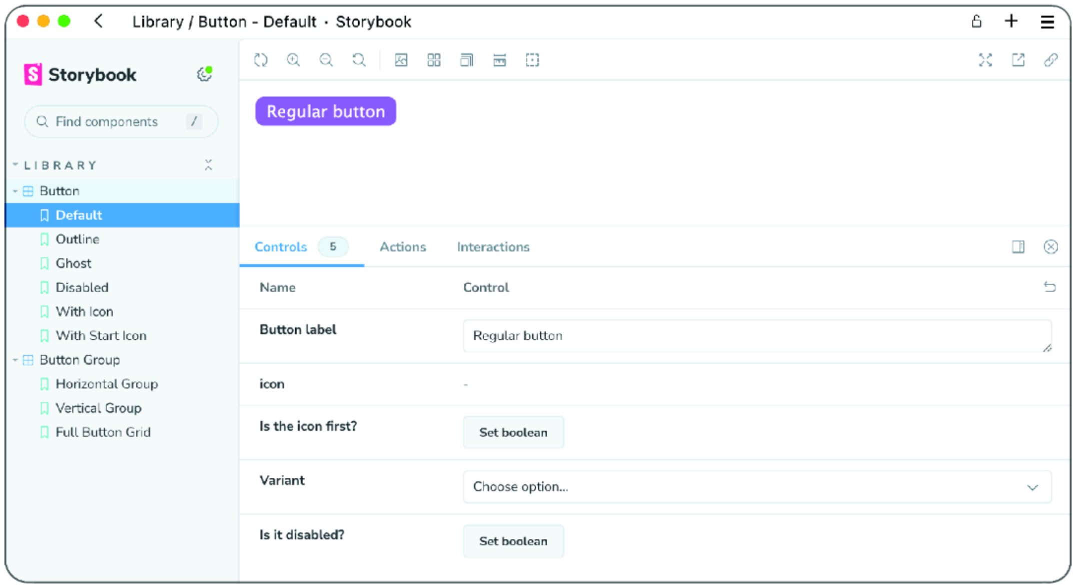
But what are the last two things? What's a *visual test tool*, and what does *coverage* mean in this context? I'll get to those topics in the next two sections.

13.1.1 Storybook: Visual testing

Suppose that you already have a UI library, and you need to use it to implement a new feature. You are given a design that features different types of buttons and different inputs. You know that all these components already exist in your library, but how would you go about finding them? How do you know which setting in which component gives you the result you see in the design?

This scenario is where a visual test tool comes in. Storybook is probably the only tool of its kind, and many teams have adopted it. It is perfect for this kind of UI library visualization.

In short, Storybook is a visual way to browse your codebase. You can see what every component looks like in every possible combination of properties and inputs that you might configure. If you set things up correctly, you can even play with the components live to see how they interact. Also, you see the code behind the given component visualization. For the custom UI library that we're going to work with in this chapter, the output of Storybook is what you see in figure 13.1.



Name	Control
Button label	Regular button
icon	-
Is the icon first?	Set boolean
Variant	Choose option...
Is it disabled?	Set boolean

Figure 13.1 Storybook has a menu with all your components (currently, only the button) and the variants for each to the left, with the main canvas and component display on the right. Below the component display are various knobs and controls that you can manipulate to interact with the displayed component. Here, you can change the button label, for example.

Storybook is already set up in the base UI library, and you can run it by running the command

```
$ npm run storybook -w ch13/base
```

Storybook starts running in the terminal and opens a browser window to the URL `localhost:6006`, which is the default URL. You can configure the port in `package.json` if you so desire. In fact, I recommend that you run Storybook now so you can see how it works.

Above the main component display in figure 13.1 are buttons that allow you to see the component in various configurations. You can do a lot with Storybook, so I recommend that you test it.

So, how do you add a component to Storybook? Does Storybook pick up the component automatically? Unfortunately, it's not that clever. You create a file named `*.stories.js` (or `.jsx`, `.ts`, or `.tsx`, depending on your setup), and it will be included if it's formatted correctly. For the button, I created `Button.stories.jsx`, which you see in the following listing.

Listing 13.1 Storybook configuration for a button

```
import { Button } from "./Button";
import {     #1
  FaPaperPlane,     #1
  FaUserCircle,     #1
} from "react-icons/fa";     #1
export default {
  title: "Library/Button",    #2
  component: Button,     #3
  argTypes: {     #4
    children: {    #4
      control: "text",    #4
      name: "Button label",    #4
    },     #4
    variant: {    #4
      control: "select",    #4
      options: ["regular", "outline", "ghost"],    #4
      name: "Variant",    #4
    },    #4
    disabled: {    #4
      control: "boolean",     #4
      name: "Is it disabled?",    #4
    },    #4
    isIconFirst: {    #4
      control: "boolean",     #4
      name: "Is the icon first?",    #4
    },     #4
    icon: {    #4
      control: "none",     #4
    },   #4
  },    #4
};
export const Default = {
  args: { children: "Regular button" }    #5
};
export const Outline = {
  args: {     #5
    variant: "outline",    #5
    children: "Fancy",  #5
  },
};
export const Ghost = {
  args: {     #5
```

```
      variant: "ghost",     #5
      children: "Ghost-like!",     #5
    },
  };
  export const Disabled = {
    args: {     #5
      disabled: true,     #5
      children: "I'm disabled",     #5
    },
  };
  export const WithIcon = {
    args: {    #5
      children: "Send",   #5
      icon: <FaPaperPlane />,    #5
    },
  };
  export const WithStartIcon = {
    args: {     #5
      children: "Profile",     #5
      isIconFirst: true,     #5
      icon: <FaUserCircle />,    #5
    },
  };
```

#1 We import some icons to test some variations
#2 The default export contains a title that matches the folder structure you see in the Storybook output.
#3 The default metadata export also contains a link to the component itself.
#4 Defines the control knobs that you see below the component in figure 13.1
#5 For each variant of the component we want to display as a separate page in Storybook, we pass in the args required to render that specific variant of the main component.
#6 For each variant of the component we want to display as a separate page in Storybook, we pass in the args required to render that specific variant of the main component.

The best way to understand Storybook is to play with `Button.stories.jsx` and see how the output in Storybook matches the code in the source file. Notice, for example, that the folder structure in the Storybook menu matches the titles and names of the named exports in the code file, and the controls below each component on the canvas match the definitions in `argTypes`.

As you see, I also created a Storybook file for the button-group component, which is a companion component to the button component. This new component stacks multiple buttons horizontally or vertically, as you can see in Storybook if you expand the Button Group folder. If you click the last story in the Button Group folder, you see a grid of various buttons in a single display (figure 13.2).

Such a grid is a great way to communicate easily to other developers (and even designers) that your components can be used in different ways. Storybook provides one extra benefit: you can use the stories and variants that you’ve already created in Storybook as the setup in your unit tests. Storybook comes with a coupler for most test libraries, including both Vitest and Testing Library, so you can use the various definitions of component variants to run tests.

The following listing shows a snippet of the current `Button.test.jsx`. In this code, I reuse the component variants from Storybook as the basis of the unit tests.

Listing 13.2 Unit test of button based on Storybook (excerpt)

```
import { render } from "@testing-library/react";
import { composeStories } from "@storybook/react";    #1
import userEvent from "@testing-library/user-event";
import "@testing-library/jest-dom";
import { vi } from "vitest";
import * as stories from "./Button.stories";     #2
const {     #3
  Default,   #3
  Outline,    #3
  Ghost,    #3
  WithIcon,   #3
  WithStartIcon,    #3
  Disabled,  #3
} = composeStories(stories);  #3
describe("Button", () => {
  test("should be clickable", async () => {
    // ARRANGE
    const mockOnClick = vi.fn();
    const { getByRole } =    #4
      render(<Default onClick={mockOnClick} />);   #4
    // ACT
    const user = userEvent.setup();
    await user.click(getByRole("button", { name: "Regular button" }));
    // ASSERT
    expect(mockOnClick).toHaveBeenCalledTimes(1);
  });
  ...
});
```

#1 Uses a Storybook utility to set up components for use with Testing Library
#2 Imports all the variants of the component we want to test directly from the story file
#3 Converts the story components to testable components through the composeStories utility
#4 Uses the composed components like any other component, but we have already configured it with various properties as required. We can still add more properties, of course, like this onClick handler here.

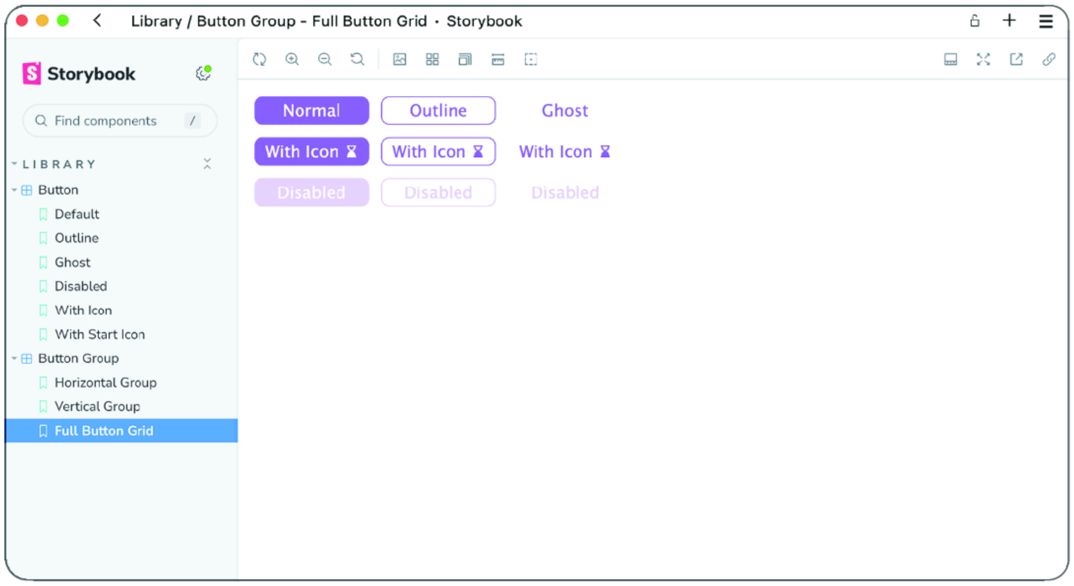


Figure 13.2 A grid of various button combinations for easy overview. I should note that the button's main color is purple, which you can't see in a grayscale print book, but you are free to use any other primary color you see fit. The color is defined in the storybook config file `.storybook/preview.jsx` .

For this particular button component, there isn't a lot of setup in the story file, so it would be easy to re-create all the variants by using the "raw" button component in the test file. But for more complex components, reusing Storybook definitions can save a lot of time, and it's a nice way of making sure that all the variants you're displaying in Storybook work correctly.

13.1.2 Istanbul: Code coverage reporting

The big question about unit tests is how much testing is enough testing. This question has many answers, including this simple one: when every bit of the code has been tested. How can you tell which lines and parts of lines have been covered by a test? The key term here is *code coverage*. A code coverage reporting tool can tell (through the magic of computer science) which bits of code have been executed how many times while the whole test is running. Istanbul is one such tool; it's one of the most popular tools for code coverage in JavaScript and TypeScript. Suppose that we have this function, which returns the display name for a person with some extra optional information:

```
function getName(person, prefix = "") {
  if (person.title) {
    return `${prefix}${person.title} ${person.name}`;
  }
  return `${prefix}${person.name}`;
}
```

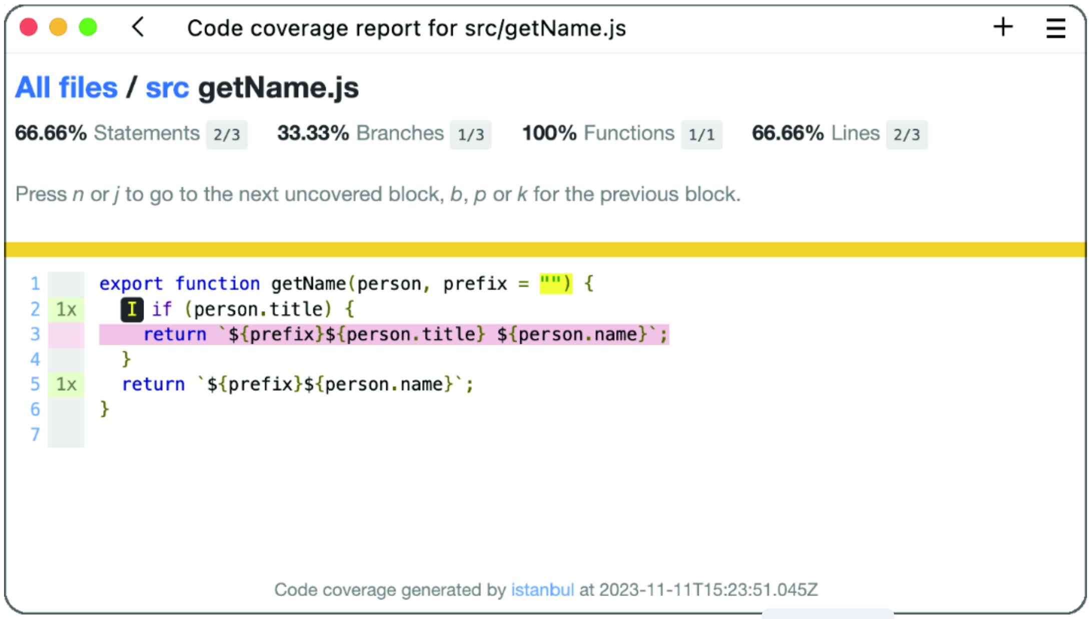
We create this test for the function:

```
test("getName returns the name", () => {
  const untitledPerson = { name: "John" };
  expect(getName(untitledPerson, "Sir ")).toBe("Sir John");
});
```

How much of the code have we covered? We haven't tested what happens if the `if` statement is `true` or what happens if the prefix is missing. If we run this test with code coverage enabled, Istanbul highlights those missing bits of code execution (figure 13.3).

At the top of the screenshot in figure 13.3, we see a summary of the report. Istanbul reports that only two out of three statements are executed, only one out of three branches is followed, and only two out of three lines are evaluated. All functions are run, however, so that's a small win. If we change the test to look like this,

```
test("getName returns the name", () => {
 const titledPerson = { title: "Dr.", name: "Jane" };
 expect(getName(titledPerson)).toBe("Dr. Jane");
});
```



Code coverage report for src/getName.js

All files / src getName.js

66.66% Statements 2/3 33.33% Branches 1/3 100% Functions 1/1 66.66% Lines 2/3

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

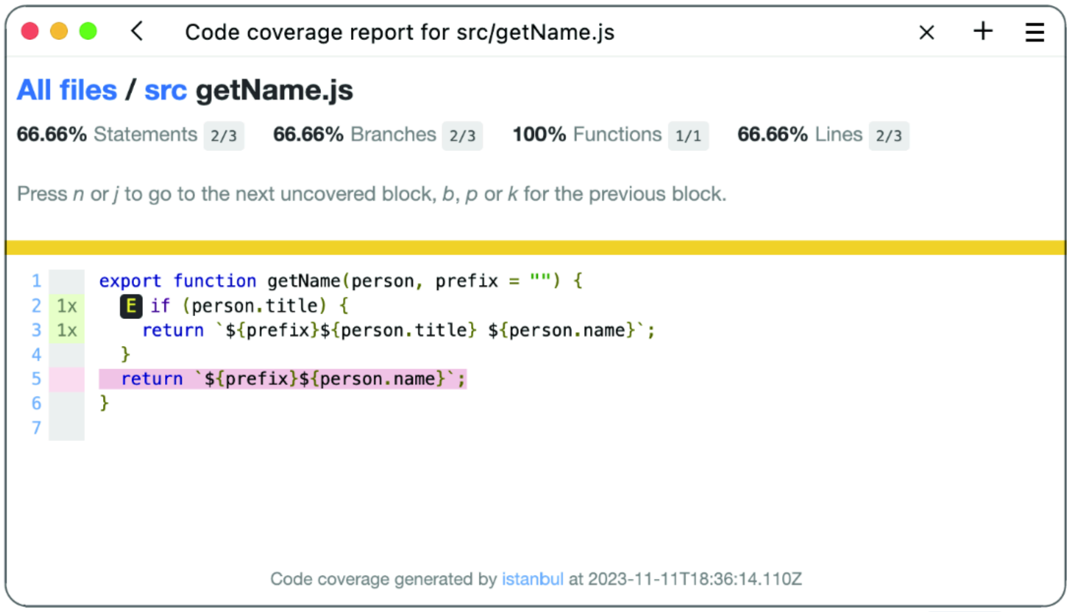
```
export function getName(person, prefix = "") {
  if (person.title) {
    return `${prefix}${person.title} ${person.name}`;
  }
  return `${prefix}${person.name}`;
}

```

Code coverage generated by istanbul at 2023-11-11T15:23:51.045Z

Figure 13.3 The Istanbul coverage report for our `getName` function contains three highlights: the `if` statement is never tested for being `true`, line 3 is never executed, and the prefix default value is never used.

we get a little closer to the goal but not all the way there, as you see in figure 13.4.



Code coverage report for src/getName.js

All files / src getName.js

66.66% Statements 2/3 66.66% Branches 2/3 100% Functions 1/1 66.66% Lines 2/3

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
export function getName(person, prefix = "") {
  if (person.title) {
    return `${prefix}${person.title} ${person.name}`;
  }
  return `${prefix}${person.name}`;
}

```

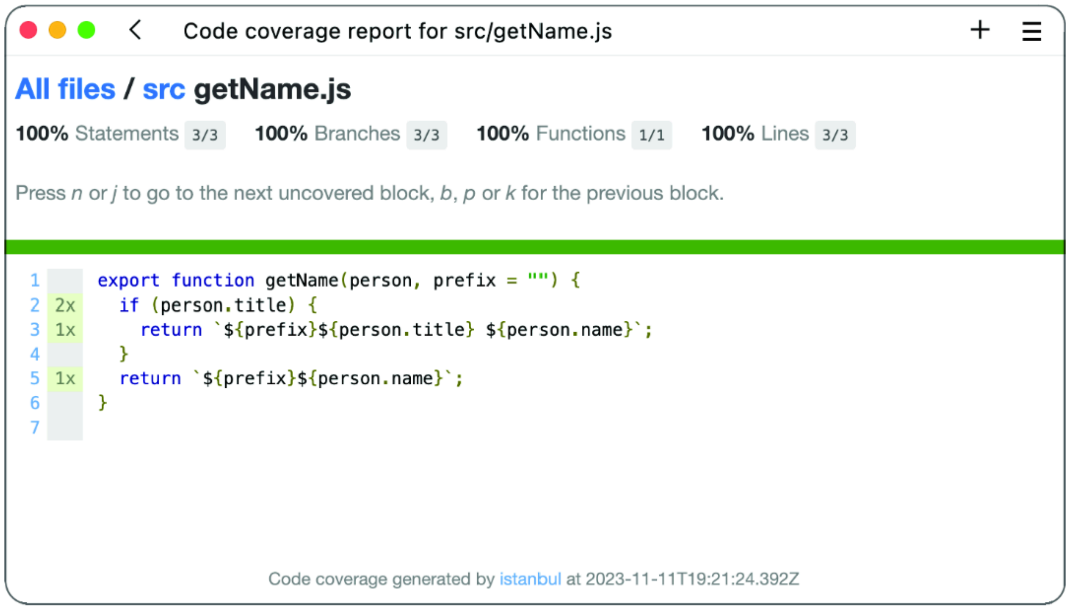
Code coverage generated by istanbul at 2023-11-11T18:36:14.110Z

Figure 13.4 Coverage is a little better but still a way off. The `else` branch of the `if` statement is not traversed, and the last return is never executed.

If, however, we run both tests in unison, like so,

```
test("getName returns the name for an untitled prefixed person", () => {
  const untitledPerson = { name: "John" };
  expect(getName(untitledPerson, "Sir ")).toBe("Sir John");
});
test("getName returns the name for a titled unprefixed person", () => {
  const titledPerson = { title: "Dr.", name: "Jane" };
  expect(getName(titledPerson)).toBe("Dr. Jane");
});
```

we have full coverage of this function, with all statements and lines executed and all branches followed (figure 13.5).



Code coverage report for src/getName.js

All files / src getName.js

100% Statements 3/3 100% Branches 3/3 100% Functions 1/1 100% Lines 3/3

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
    export function getName(person, prefix = "") {
2x    if (person.title) {
1x      return `${prefix}${person.title} ${person.name}`;
      }
1x    return `${prefix}${person.name}`;
    }
```

Code coverage generated by istanbul at 2023-11-11T19:21:24.392Z

Figure 13.5 We finally have full coverage. We can even see that the line with the `if` statement is executed twice (`2x` in the margin), but each return is executed only once (`1x` in the margin).

How Istanbul works is beyond me. I consider it pure magic. Somehow, Istanbul is able to hook into the compiler, and we can watch what it does in files. We don't need to know how it works. We only have to trust that it works. Where does the coverage report show up, and how is it generated? If it's enabled, Vitest does the work. Vitest supports Istanbul but doesn't bundle it directly, so setup is quick. We can add some configuration options, and we're good to go.

In the base-library example (section 13.1, in the sidebar "Example: base"), I added some configuration options for Vitest in `package.json` , as well as a new script called `test:coverage` . You can run that script in the terminal by running

```
$ npm run test:coverage -w ch13/base
```

If you do, you see the following output:

```
 PASS  src/library/button/ButtonGroup.test.jsx
 PASS  src/library/button/Button.test.jsx

============================= Coverage summary ==============================
Statements   : 100% ( 15/15 )     #1
Branches     : 100% ( 15/15 )    #1
Functions    : 100% ( 4/4 )    #1
Lines        : 100% ( 15/15 )   #1
=============================================================================

Test Suites: 2 passed, 2 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        2.173 s
```

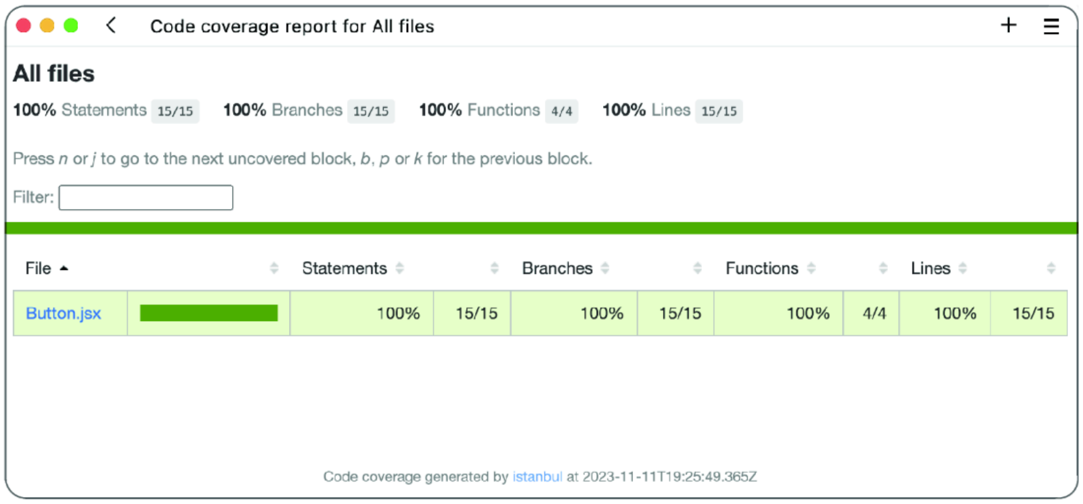
#1 Now the test output also contains a coverage summary.

You get the coverage summary directly in the output and also generate the full HTML report in a library named `coverage/`, so you can open the file `coverage/index.html` like so:

```
$ open ch13/base/coverage/index.html
```

Then you see the report shown in figure 13.6.

Chasing that magic 100% coverage is often futile, especially if you try to do it for the entire project. But getting 100% coverage on select files, folders, or packages might be worthwhile, especially if they contain generalized functions or utilities used throughout the application or are especially intensive or complex mathwise.



Code coverage report for All files

All files

100% Statements 15/15 100% Branches 15/15 100% Functions 4/4 100% Lines 15/15

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

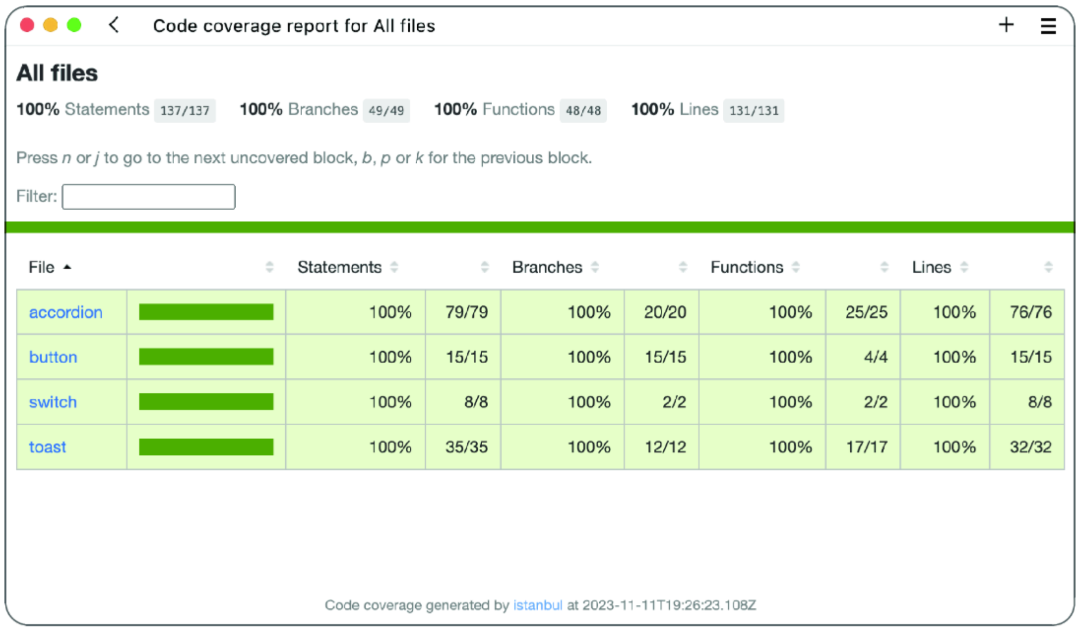
Filter:

File		Statements		Branches		Functions		Lines	
Button.jsx	🟩	100%	15/15	100%	15/15	100%	4/4	100%	15/15

Code coverage generated by istanbul at 2023-11-11T19:25:49.365Z

Figure 13.6 The coverage report for the initial base library shows 100% coverage across all components (one, in this case).

I achieved 100% test coverage for the initial button, as you see in figure 13.6. Similarly, I can reveal that in the final solution for all the components we'll build in this chapter, I achieved 100% coverage (figure 13.7).



Code coverage report for All files

All files

100% Statements 137/137 100% Branches 49/49 100% Functions 48/48 100% Lines 131/131

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File		Statements		Branches		Functions		Lines	
accordion	🟩	100%	79/79	100%	20/20	100%	25/25	100%	76/76
button	🟩	100%	15/15	100%	15/15	100%	4/4	100%	15/15
switch	🟩	100%	8/8	100%	2/2	100%	2/2	100%	8/8
toast	🟩	100%	35/35	100%	12/12	100%	17/17	100%	32/32

Code coverage generated by istanbul at 2023-11-11T19:26:23.108Z

Figure 13.7 In the example with the completed library for reference, I achieved full coverage across all four components. You can see by the pure numbers that the accordion and toast components are a lot more complex than the button and switch.

You are not obliged to chase 100% coverage, but it would be a good idea to keep an eye on the coverage as you test your components. Think about whether it makes sense to cover a particular line or branch.

13.2 Your new job: Extending the library

You have three components to complete:

1. First is the switch component, which is easy to set up but requires some fancy CSS to make it pretty. Functionally, it doesn't do a lot, so there isn't that much for you to do, and you should get a quick win. Just setting up the stories and adding the tests will prove to be a good challenge.
2. Next is the accordion component, which is easy to get going, but you have to take a lot of extra concerns into consideration to complete the task. Writing tests for this component is going to be extra tricky because you have to test some fairly complex keyboard-based interactions.
3. Finally, you have a toast component. This component isn't about making breakfast; it creates floating messages atop the application that can be dismissed or that go away by themselves. Such messages are used for asynchronous feedback, such as "Account created successfully" and "The message could not be delivered." Toasts are tricky to implement and involve a bit of React API that I haven't discussed: *portals*. Portals are a lot more boring than they sound, though—no interstellar travel or parallel universes. Sorry!

As I set you up for the tasks ahead, I'm going to cover three areas for each component that will guide you in your solution:

- *Design*—What will the component look like, and what will it look like during certain interactions? If something can be focused, for example, the design should include both a hover style and a focus style for that particular element. (The hover and the focus styles might be the same, though.)
- *List of acceptance criteria*—"Acceptance criteria" (AC) is the formal term for "What does this thing do?" Stakeholders use AC when evaluating whether a thing does what they expect it to do. If a button is expected to invoke its callback when it's clicked except when it's disabled, that expectation is an explicit AC.
- *Relevant standards to follow and/or reference implementations*—For some types of components, users often already have a well-defined expectation of how the component is supposed to work. If you have a

range slider, people expect to manipulate the slider thumb by pressing the arrow keys on the keyboard. For the components built in this chapter, I'll provide links to relevant standards for how these components are generally supposed to be built. Sometimes I will also provide relevant reference implementations either by standardizing bodies or by other UI libraries. I might even add some links to clever ways to implement a specific snippet or style.

13.2.1 A Switch component

We need a switch component. You may not recognize it by the name, but it's a modern version of a check box used in mobile UIs. Figure 13.8 shows what such a component looks like in various mobile operating systems.

iOS
Android

Figure 13.8 A switch is a popular mobile UI component that's making its way into nonmobile web interfaces.

The following sections go over the requirements for the component you're about to build, starting with the design.

DESIGN GUIDELINE

The switch component needs to look like figure 13.9. You may be reading this chapter in a grayscale paper book; in that case, imagine any color for the active state. In the original here, we use the same purple as in the button component, but in your implementation, you can use any other reasonably dark color.

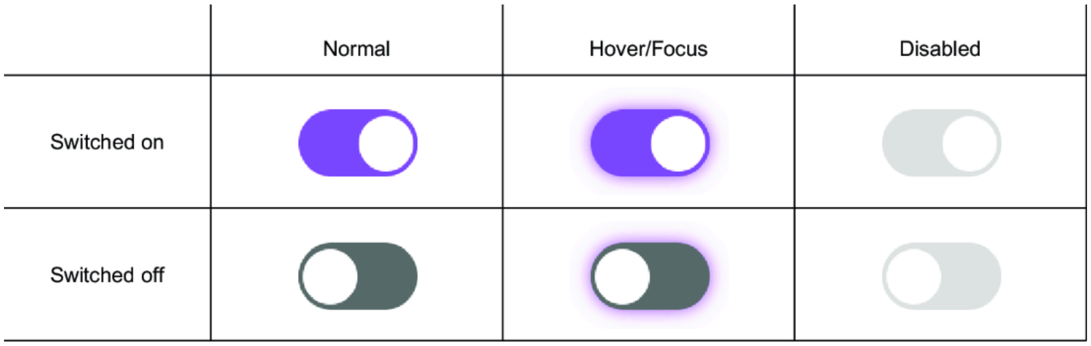


Figure 13.9 The switch component in various states. Note that the hover and focus states are identical and that no hover or focus states apply to the disabled component. Regarding the colors, both hover and normal on states should have a nongray color, and the focus/hover outline should be a shade of the active color.

ACCEPTANCE CRITERIA

The switch is correctly implemented if these criteria are true for your implementation:

- The component should semantically be a check box, as in an `<input type= "checkbox" />` element.
- Users should be able to toggle the component by clicking any part of the component itself, clicking the associated label, or focusing the component and pressing the Enter key, the spacebar, or the left- or right-arrow key on the keyboard.
  Most of these interactions are automatically supported if you use a proper semantic check box. But you have to add some interactions (hint: three of the keyboard interactions) manually.
- The component signature must be

```
function Switch({ label, ...props }) {
  ...
}
```

- This signature means that the component must take a label as an argument and that any other properties passed to it will be forwarded to the check box `<input />` element.
- The component must work in your daily browser of choice. Cross-browser support is not required.
- The component should follow best practices for accessible check boxes.

REFERENCE IMPLEMENTATIONS AND BEST PRACTICES

Feel free to look at the following resources for specifications and inspiration:

- *Web Accessibility Initiative-Accessible Rich Internet Applications (WAI-ARIA) standards for a two-state check box*—**https://www.w3.org/WAI/ARIA/apg/patterns/checkbox**
- *CSS inspiration*—**http://mng.bz/WV6X** (more comprehensive than necessary)
- *Reference implementations*
  - *MUI*—**https://mui.com/material-ui/react-switch**
  - *Ant Design*—**https://ant.design/components/switch**
  - *Chakra UI*—**https://chakra-ui.com/docs/components/switch**

13.2.2 An accordion component

The second UI element is an accordion component. These components are ubiquitous across the web in many forms. Figure 13.10 shows a few examples.

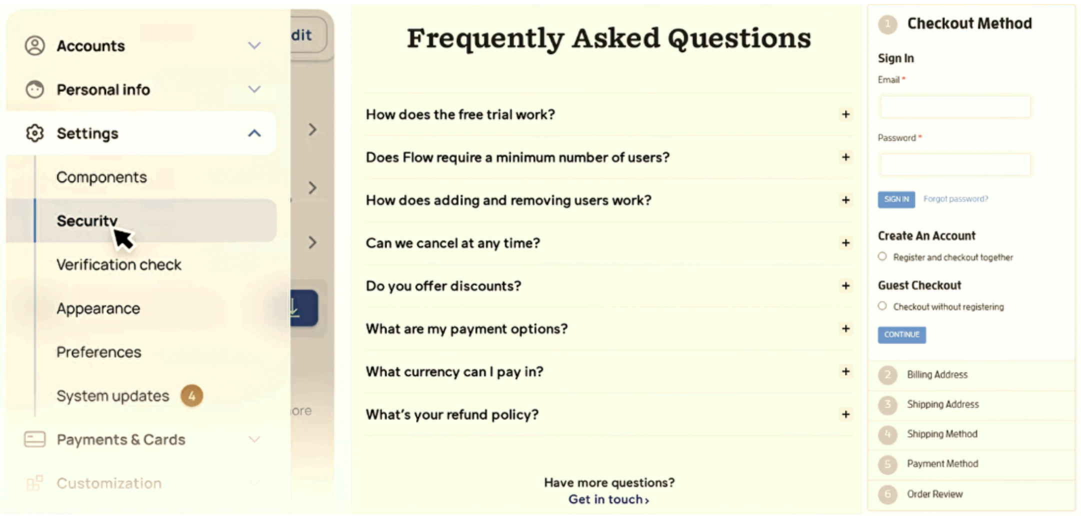


Figure 13.10 Three examples of an accordion component: a menu, an FAQ section, and a checkout flow

DESIGN GUIDELINE

The accordion component can have many appearances, depending on how you want to use it. For our use case, we'll go for a simple, clean component. Figure 13.11 shows various combinations of states, including focus and hover.

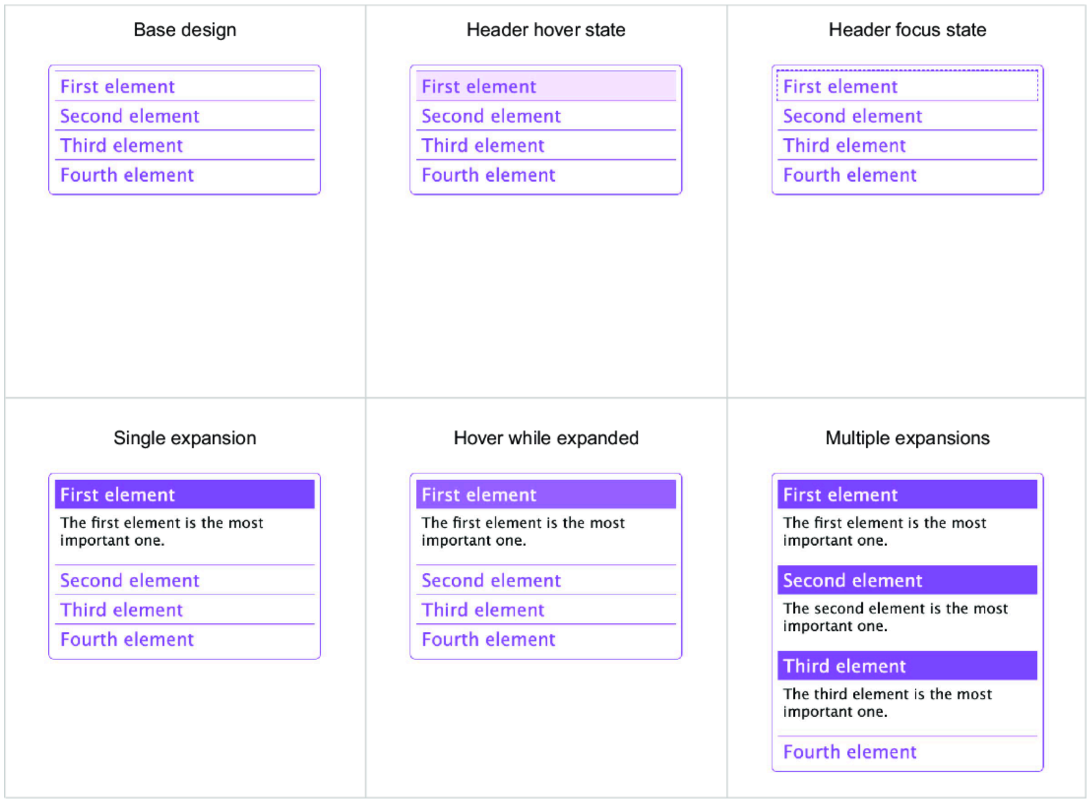


Figure 13.11 The accordion component in various states. It's important to pay attention to all the variants in your implementation. Again, the active color (used for the headline) in my implementation is purple, but you can choose any appropriate color. Refer to the component signature to see the different options.

ACCEPTANCE CRITERIA

- The accordion is correctly implemented if these criteria are true for your implementation: the component should semantically be a section with an article for each item, each consisting of a button for the header and an article for the item content.
- The component should follow WAI-ARIA guidelines for keyboard interaction. (Note that arrow-key interaction is optional but recommended.)
- The component should follow WAI-ARIA guidelines for `aria-*` properties. In particular, item headers should point to their associated content through `aria-controls`, and item content should point to their headers through `aria-labelledby`.
- The component signature must be

```
<Accordion
  activeIndex={0}    #1




  isCollapsible={false}     #2
  allowsMultiple={false}     #3
>
  <Accordion.Item>     #4
    <Accordion.Header>     #5
      First element
    </Accordion.Header>
    <Accordion.Content>
      The first element is the most important one.
    </Accordion.Content>
  </Accordion.Item>
  ...     #6
</Accordion>
```

#1 The component accepts three arguments. The first argument is the index of the initially active item, which defaults to 0. Set it to −1 to have all the arguments collapsed initially.
#2 Per default, an expanded accordion item cannot be collapsed unless another is expanded. If you want to allow collapsing an item again, set isCollapsed to true.
#3 If multiple items can be expanded at the same time, set allowsMultiple to true, which automatically allows items to be collapsible.
#4 Each item in the accordion must be wrapped in this component.
#5 The item consists of a header, which is always visible, and content, which is visible only when expanded.
#6 Add multiple items by adding more items inside the accordion component.

- It must be possible for the developer to put any content inside each accordion item.
- The component must work in your daily browser of choice. Cross-browser support is not required.

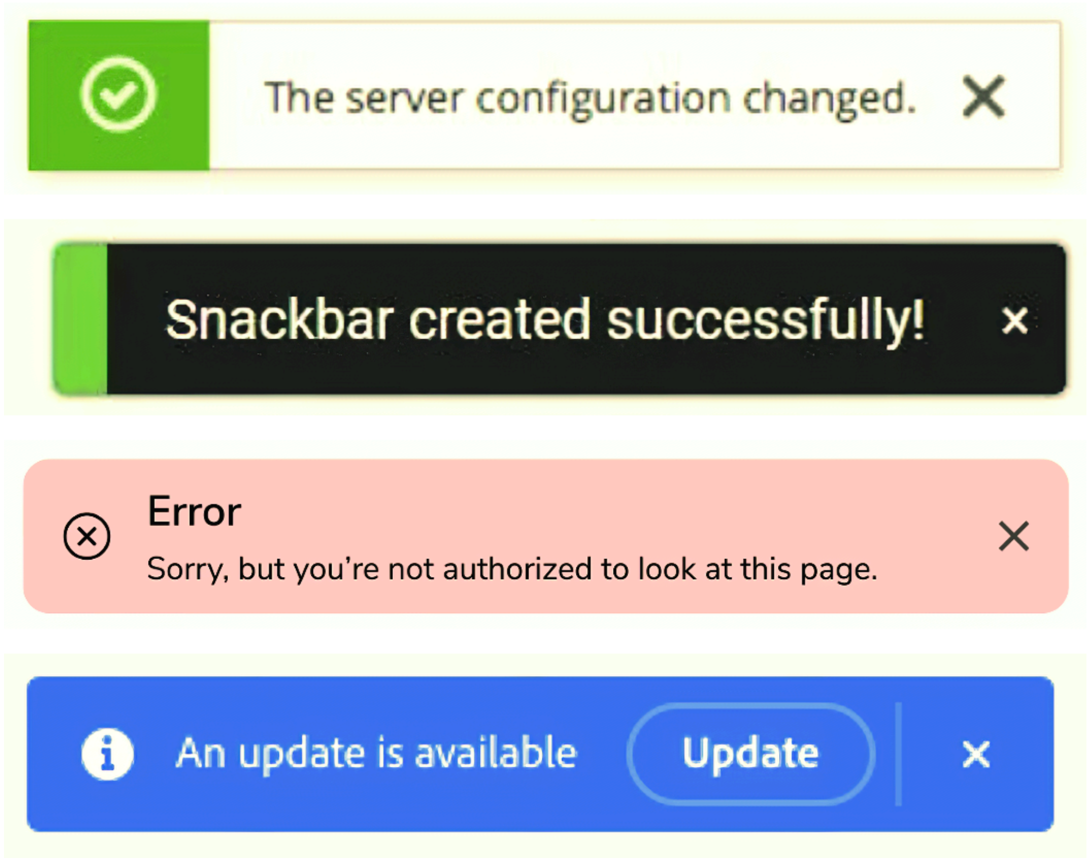
REFERENCE IMPLEMENTATIONS AND BEST PRACTICES

Feel free to look at the following resources for specifications and inspiration:

- *WAI-ARIA standards for an accordion*—
  https://www.w3.org/WAI/ARIA/apg/patterns/accordion
- *Reference implementations*
  - *MUI*—**https://mui.com/material-ui/react-accordion**
  - *Ant Design*—**https://ant.design/components/collapse**
  - *Semantic UI*—**https://react.semantic-ui.com/modules/accordion**
  - *Chakra UI*—**https://chakra-ui.com/docs/components/accordion/usage**

13.2.3 A toast component

The third UI element we need is a toast component. This component is known by many names, including notification, snack bar, alert, and message. The component informs a user about something happening in the application with a potentially dismissible notification. Figure 13.12 shows some real-world examples of toast messages.


The server configuration changed.

Snackbar created successfully!

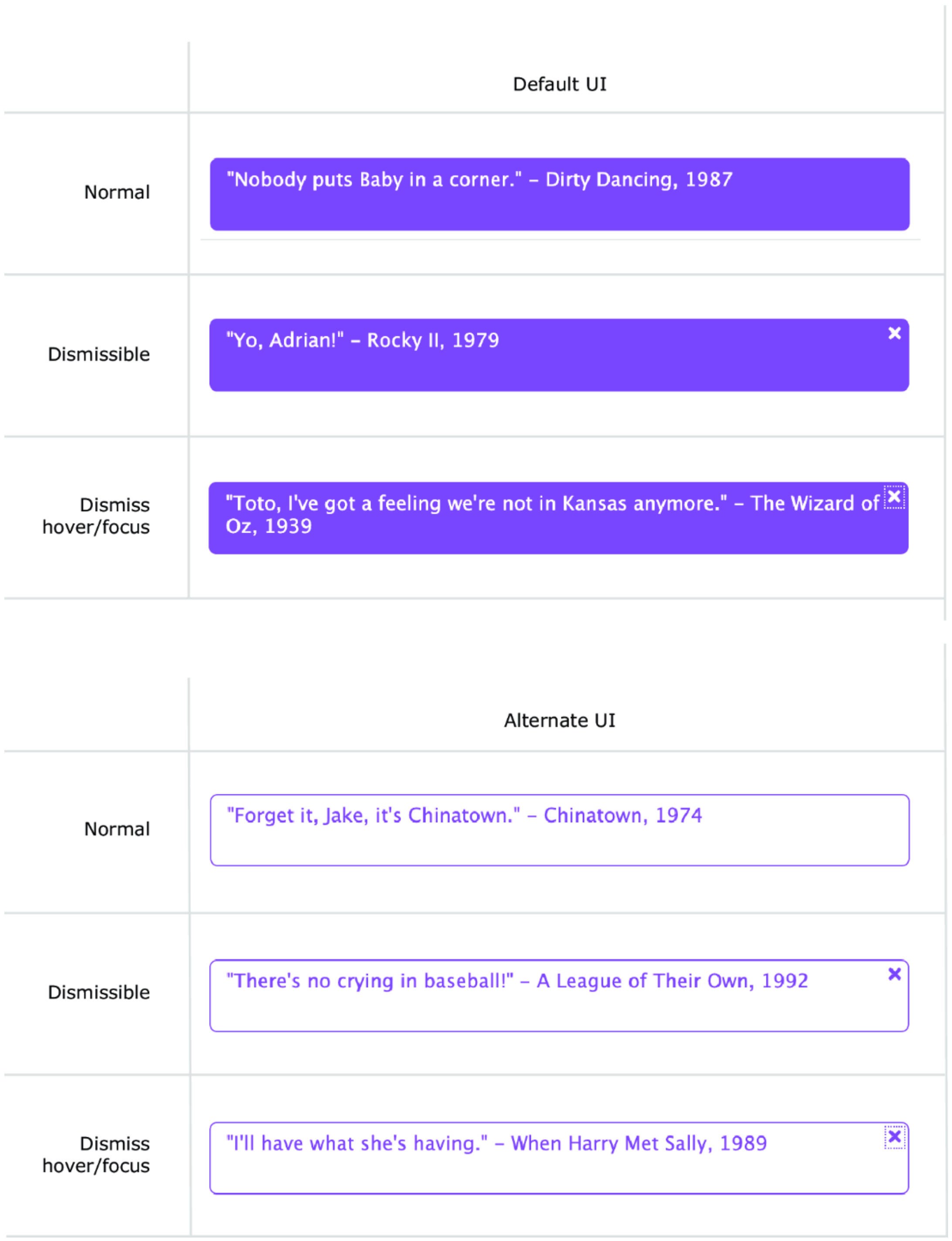
Error
Sorry, but you're not authorized to look at this page.

An update is available

Figure 13.12 Variants of toast messages used on the web. Note that these examples use different combinations of icons, buttons, and headlines.

DESIGN GUIDELINE

We use a simple toast message but with an alternate (inverted) UI option. The dismiss button is configurable (default: off). See figure 13.13 for details.



	Default UI
Normal	"Nobody puts Baby in a corner." – Dirty Dancing, 1987
Dismissible	"Yo, Adrian!" – Rocky II, 1979 ×
Dismiss hover/focus	"Toto, I've got a feeling we're not in Kansas anymore." – The Wizard of Oz, 1939 ×

	Alternate UI
Normal	"Forget it, Jake, it's Chinatown." – Chinatown, 1974
Dismissible	"There's no crying in baseball!" – A League of Their Own, 1992 ×
Dismiss hover/focus	"I'll have what she's having." – When Harry Met Sally, 1989 ×

Figure 13.13 The design for the toasts in our UI with the two variants and their states displayed. The button to dismiss a toast has the same style for both hover and focus, as shown in the bottom rows.

ACCEPTANCE CRITERIA

The toast is correctly implemented if these criteria are true for your implementation.

- To use toast messages in an application, two things must be added:
  - A toast provider wrapping the entire application.
  - Invocation of a `useToast` hook that displays toast messages.
- Toast messages must be displayed globally in the browser window at the bottom-center position, regardless of the position of the toast provider in the document object model (DOM) structure. (Hint: use a React portal.)
- Toast messages are displayed by using either the primary or the alternate (inverted) style, with the latter triggered by setting the `isOutline` property to `true`.
- Toast messages normally don't include a dismiss button but will if the `canDismiss` property is set to `true`.
- Toast messages disappear automatically after 3 seconds (even if they're dismissible) unless the `isPersistent` property is set to `true`. Note that if this property is `true`, the toast message is dismissible automatically, regardless of the `canDismiss` flag.
- The toast provider does not take any properties and is used as a wrapper around the relevant section of the application:

```
<ToastProvider>
  <App />
</ToastProvider>
```

- The `useToast` hook is used as follows with the relevant properties:

```
const addToast = useToast();    #1
addToast({      #1
  message: "This is the toast message",     #2
  canDismiss: false,     #3
  isPersistent: false,    #3
  isOutline: false,    #3
});
```

#1 The hook returns a function to display the toast.
#2 The only required parameter is the toast message.
#3 The other three options toggle the other variants, which default to false and can be combined freely except as noted.

- Toast messages should follow WAI-ARIA best practices for a polite live region.
- The component must work in your daily browser of choice. Cross-browser support is not required.

REFERENCE IMPLEMENTATIONS AND BEST PRACTICES

Feel free to look at the following resources for specifications and inspiration:

- *WAI-ARIA standards for live regions* —https://mng.bz/0GpE
- *React documentation on portals* —https://react.dev/reference/react-dom/createPortal
- *A concrete React portal example* —https://mng.bz/KZQg
- *Reference implementations*
  - *MUI (Snackbar)* —https://mui.com/material-ui/react-snackbar
  - *Ant Design (Message)* —https://ant.design/components/message
  - *Chakra UI (Toast)* —https://chakra-ui.com/docs/components/toast

13.2.4 My solution

I made my own implementation of this project. My solution is one possible solution but far from the only one, and it might not even be the best one. You can see my solution in the following example.

EXAMPLE: COMPLETE

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

```
$ npm run dev -w ch13/complete
```

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/ch13-complete**.

13.3 Future work

This UI library could be expanded in many ways. The most obvious way is to add other component types. More form elements are especially common in UI libraries, such as drop-down menus and input variants. I don't need to tell you what should go in this library; it's yours now, so feel free to expand it as you see fit.

I still need to address another issue with the existing library: accessibility. You may think that we've already looked at the WAI-ARIA recommendations for each of these types of components, followed the best practices, and addressed `aria-*` properties, screen readers, and keyboard accessibility. How much more can there be to do? There are quite a few accessibility concerns that we haven't considered:

- These components only support English and cannot be translated.
- If you could translate them, the components do not necessarily work correctly in right-to-left writing mode (as in Hebrew or Arabic writing).
- These components don't support dark mode because they don't respect the `prefers-color-scheme` browser setting.
- The components don't respect the `prefers-reduced-motion` browser setting, which allows people with vestibular motion disorders to ask pages to avoid unnecessary animations and motion.
- Some Microsoft devices use a high-contrast setting for Surface laptops. Imagine a Kindle reader with an all-black-and-white screen so that it's readable in direct sunlight. Microsoft Surface laptops use similar high

contrast in flip mode to be readable in direct sunlight, but this feature only works if the visited websites support it.
- The components do not scale with the browser font size, so people with reduced eyesight (or using a screen further away) are not able to increase their browser font size, which all UI elements should respect.

These and many other concerns are some of the features that the best UI libraries take into consideration (in addition to the features we’ve already implemented).