10 Unit-testing React

This chapter covers

- Using common testing terminology
- Testing React components correctly
- Adding tests for custom hooks

How do we know that our React applications work as intended? Well, we can click around and see whether everything works the way we want it to work, right? We might forget to click some button or straight up forget some unlikely but possible combination of events. Also, if we're testing our application manually, we have to remember to do all the tests every time we change something to make sure that we didn't break something else.

That approach, of course, is a terrible way to ensure persistent quality in your application. Instead, quality assurance is a prime candidate for automation. With automated tests, you write scripts that emulate how a user would interact with your application (at various scopes) and verify automatically that the application acts the way it is supposed to act.

We'll jump right into testing React components by using React Testing Library (yes, that is the name of the package; it's not all that inventive). This library has many interesting aspects and variations, so we'll discuss a few types of components and how to test them properly. After that, we'll test some custom hooks, setting up some hypothetical situations based on real-world examples.

Note that, in this chapter, we won't create one large application, as we've done in the past few chapters. We don't need a large codebase to discuss proper testing, and the final three project chapters will give us a great chance for testing. All three projects include optional testing homework, if you want an extra challenge.

You can use a ton of tools for testing with different goals and at different levels. In this chapter, we'll discuss how to use Vitest as a test runner and React Testing Library as the component black-box testing framework. Vitest is a new library that's often used with Vite, which we use in our examples. Jest is another testing library that's probably more commonly used in React testing, but Vitest is almost identical in its API and is gaining a lot of traction because of its tie-in with Vite. Both Vitest and Jest can be used on their own without React Testing Library, but React Testing Library has quickly become the standard, as it comes with a better set of tools and utility functions and is built on a better set of principles. We'll get back to those principles in section 10.1.3. So 1-2-3 testing!

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch10**.

10.1 Testing a static component

We will start our journey with a simple functional component that could be used in any website's main navigation, `MenuItem`:

```
export function MenuItem({ href, label }) {
  return (
    <li>
      <a href={href} title={label}>
        {label}
      </a>
    </li>
  );
}
```

Before we consider testing this component, let's go up one level and think about what makes this component work. We want to make sure that this component always works, but what does that mean? What does the component do? I'll try to rephrase the responsibility of the component in English rather than in JavaScript:

> *The `MenuItem` component renders a list item with a link. The link points to the given `href` property and uses the given `label` property as both title and link text.*

That text seems like a fair description of what we expect from this component, no? In pseudocode, it would be good if we could test with something like this script:

```
// PSEUDOCODE - DOESN'T WORK
const component =     #1
  <MenuItem href="/blog" label="Blog" />;    #1
const listItem =     #2
  findElement(component, "listitem");    #2
const link = findElement(listItem, "link");    #2
expect(link).toHaveProperty("href", "/blog");     #3
expect(link).toHaveProperty("title", "Blog");   #3
expect(link).toHaveTextContent("Blog");     #4
```

#1 First, we set up the test with an instance of the component.
#2 We want to find elements with a given role inside some other element.
#3 Then we want to check whether an element has a specific property . . .
#4 . . . as well as specific text content.

Luckily, modern test tools are fairly intuitive. This fictitious pseudocode is close to what we actually do. The following listing shows how we can implement this test correctly with Vitest and React Testing Library.

Listing 10.1 Unit-testing `MenuItem`

```
import {    #1
  render,    #1
  screen,   #1
  getByRole,  #1
} from "@testing-library/react";   #1
import "@testing-library/jest-dom";    #2
import MenuItem from "./MenuItem";     #3
test(    #4
  "MenuItem renders a link in a list item",   #4
  () => {   #4
  // ARRANGE
  render(<MenuItem href="/blog" label="Blog" />);    #5
  // ASSERT
  const listItem = screen.getByRole("listitem");  #6
  const link = getByRole(listItem, "link");    #6
  expect(link).toHaveAttribute("href", "/blog");     #7
  expect(link).toHaveAttribute("title", "Blog");   #7
  expect(link).toHaveTextContent("Blog");   #7
});
```

#1 We import the render() function and two utility functions from React Testing Library.
#2 We need to make sure the jest-dom extension is installed as well; we'll get back to what it does. Also, even though the extension says Jest, it works the same way for Vitest.
#3 We also have to import the component we want to test.
#4 We wrap the test body in the function test() with a descriptive name. This function is provided by Vitest and need not be imported.
#5 We render the component into a JavaScript structure like a document object model (DOM) so we can run tests on it.
#6 Via screen.getByRole, we can get any element by role in the entire DOM. getByRole as a function allows you to get elements inside other elements by role.
#7 The Jest DOM extension allows us to test elements for properties and text content in a meaningful way.

Listing 10.1 contains a lot to unpack because it has some extra content we didn't know we needed—mostly imports of the relevant utilities and some scaffolding that we always need to run tests.

BUT AREN'T TESTS MORE CODE?

If you are worried about adding extra code, which means that the whole application will grow larger, (a) you have your head in the right place because reducing bundle size is always a concern, but (b) don't worry about it because tests aren't included in the production bundle.

Your test files are run only when you specifically ask to test your project. When you run your project in development mode or build the project for production, the test files are ignored. So don't worry that they'll increase your application's footprint.

If you wonder what `"listitem"` and `"link"` mean in listing 10.1, I hear you; those items are confusing. I'll get to that topic shortly. For now, just know that `<li>` elements have the role `"listitem"` (because that's their role) and `<a>` elements have the role `"link"` (because that's their role).

Where do we put this test, though? Suppose that we have the component defined in `MenuItem.jsx`. We simply put the test in `MenuItem.test.jsx` and place it next to the `MenuItem` component in the same folder, which we've done in the `menuitem` example.

EXAMPLE: MENUITEM

This example is in the `menuitem` folder. You can run the tests in that example by running this command in the source folder:

```
$ npm test -w ch10/menuitem
```

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/ch10-menuitem**.

The files inside the `src` folder are now

```
src/
  App.jsx
  index.jsx
  MenuItem.jsx
  MenuItem.test.jsx
```

We're now ready to run the test. So let's learn how.

ROLES AND NAMES

What does the `getByRole` function mean? It refers to ARIA roles. *ARIA* is an acronym for *Accessible Rich Internet Applications*, and *roles* pertain to all elements in a web document. Because there are many types of users of web pages with many wants and needs, web documents must be accessible not only visually but also programmatically.

Users with various disabilities may be using a screen reader to interact with your web page, or a search engine bot or an information service such as Siri or Alexa might try to index your information. Such services can tell you the height of the Eiffel Tower

only if they can find that information on some website that is marked up in a way to make that information easily accessible by a computer.

ARIA's *Web Accessibility Initiative* (often called *WAI-ARIA*) is a specification concerning how elements in a web page can be structured, understood, and interacted with in a nonvisual manner that mirrors visual interaction with the same web page.

WAI-ARIA defines the possible roles that elements can take. The specification defines how each role should be understood and, if relevant, how it can be interacted with. Most interesting elements come with an implicit role, but you can override the role of any given element. Some roles are applied only manually, never implicitly.

Most of this discussion is beyond what we need to know here. All we care about is being able to test our components by accessing elements by role rather than by HTML tag name, class name, or other attribute, as roles are similar to how users interact with an application.

Let's get back to practical matters. Links have the role `"link"`. List items have the role `"listitem"`. Headlines (`h1` – `h6`) are `"heading"`, buttons are `"button"`, images are `"image"`, and so on. With that information, this line suddenly makes a lot more sense:

```
const listItem = screen.getByRole("listitem");
```

This concept has another important aspect: every element also has an automatically generated *name*. This name is the *accessible name* and has a complicated algorithm behind it, but basically, the name of an element is the text inside it. Consider this snippet:

```
<a href="/blog">Blog</a>
```

This link has the name `"Blog"`. That's obvious. But we also have names for elements with nested nodes, elements with images, Scalable Vector Graphics (SVGs), or other complex HTML elements inside them. We could create another link element with the name `"Blog"` by including an image with the `alt` attribute `"Blog"`:

```
<a href="/blog">
  <img src="/images/blog.png" alt="Blog" />
</a>
```

Both of these links work well visually because the blog image is probably something that a visual user would immediately recognize as a blog icon. A nonvisual user would also know that the link is a link to the blog because its accessible name is `"Blog"` regardless of how it is communicated visually. This example is the core of the concept of an accessible name. You can arrange your document so that your elements have the correct accessible name regardless of how your element appears to sighted users.

We can get an element by role *and* name by passing the name as an option to the `getByRole` function:

```
const listItem = screen.getByRole("listitem", { name: "Blog" });
```

We will be using roles and names throughout our tests.

10.1.1 Running tests

Vitest doesn't come bundled with Vite (to reduce the initial download), so we need to do four things to make tests work in a standard Vite setup. I've already done the work for you in the `menuitem` example, but these are the four steps:

1. Install new packages.
2. Update ESLint configuration.
3. Amend Vite configuration with test setup.
4. Add test scripts to `package.json`.

INSTALLING NEW PACKAGES

This step is the easiest one. We simply install four packages and save them in `devDependencies`:

```
$ npm -w ch10/menuitem install --save-dev vitest jsdom
↪@testing-library/react @testing-library/jest-dom
```

These packages are

- `vitest`—The test runner
- `jsdom`—The simulated DOM environment in which we run the tests
- `@testing-library/react`—The testing framework
- `@testing-library/jest-dom`—Some utility methods that make writing tests more elegant

Note the order of the command-line interface (CLI) arguments. If you use `npm install -w <workspace> --save-dev <package>`, that command will *not* save the dependencies to `package.json`. You have to use `npm -w <workspace> install --save-dev <package>` for the installation to work correctly, because we use npm workspaces.

UPDATING ESLINT

ESLint is set up to assume that all your source files are regular React files. But for our tests, we commonly use globally available functions such as `test`, `it`, `describe`, and `expect`. We need to tell ESLint that all files that end with `*.test.js` or `*.test.jsx` are to be run in a Vitest environment. For that purpose, we add this block to `.eslintrc.js`:

```
module.exports = {
  ...
  overrides: [
    {
      files: ["**/*.test.js", "**/*.test.jsx"],
      env: { vitest: true },
    },
  ],
};
```

AMENDING VITE CONFIGURATION

Next, we need to tell Vite that we're using Vitest as the test runner. This process is simple. We add a single line of config to the default:

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";
// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  test: { globals: true, environment: "jsdom" },     #1
});
```

#1 The new line we're adding to set up Vitest

ADDING TEST SCRIPTS

The last step is adding new scripts to `package.json` so we can run them by using the `npm run` command. First, let's add the basic test script:

```
{
  ...
  "scripts": {
    ...
    "test": "vitest",     #1
    ...
  },
  ...
}
```

#1 We add an extra script that invokes Vitest with the default parameters.

Now we can run the script by using `npm run test`, and when we do so, we get this output:

```
$ npm test -w ch10/menuitem
> @jrr/ch10-menuitem@0.1.0 test
> vitest
 DEV  v0.34.6 <path>/ch10/menuitem
 ✓ src/MenuItem.test.jsx (1)    #1
   ✓ MenuItem renders a link in a list item    #1
 Test Files  1 passed (1)     #2
      Tests  1 passed (1)   #2
   Start at  23:35:28
   Duration  1.12s (transform 47ms, setup 0ms, collect 218ms,
     tests 63ms, environment 537ms, prepare 101ms)
 PASS  Waiting for file changes...    #3
       press h to show help, press q to quit   #3
```

#1 When running, the script prints out every test suite (test file) and test case it comes across and shows it as passing or failing.
#2 The script sums up the results.
#3 The script doesn't exit after completion but keeps listening for file changes to rerun the tests on any change.

This output looks great. The test passes, which is a good start. As a bonus, `npm` allows us to run the test script by typing `npm test` rather than `npm run test`, so we've already saved a couple of keystrokes.

Notice that in the preceding output, the test runner keeps listening after a full run of all tests. That setting is the default; the test runner monitors your test files continuously. But we can pass a flag to have the test run only once:

```
$ npm test -w ch10/menuitem -- run
```

When we run the command with the `run` flag, the test runs only once, which is nice for this demonstration. For development purposes, it's probably better to run the test without the flag to keep it running as you write your application and observe that all tests always pass.

In the `menuitem` example, I added an extra script called `test:once` that will run the test with the `run` flag. You can run that test by typing `npm run test:once` if is configured like so:

```
{
  ...
  "scripts": {
    ...
    "test": "vitest",
    "test:once": "vitest run",
    ...
  },
  ...
}
```

10.1.2 Test file location

Note that not everybody puts test files next to the components that they're testing, and not everyone follows the same naming patterns. Here are a few other possible patterns:

- Tests go in a separate folder inside each source folder. If you have `/src/MenuItem.jsx` and `/src/library/Button.jsx`, the tests will reside in `/src/test/MenuItem.jsx` and `/src/library/test/Button.jsx`, respectively.
- Tests go in a folder other than the source folder. If you have `/src/MenuItem .jsx` and `/src/library/Button.jsx`, the tests will reside in `/tests/MenuItem .jsx` and `/tests/library/Button.jsx`, respectively.
- Some people use the name fragment `spec` instead of `test`, as in `Button .spec.jsx`.

All these approaches have pros and cons. Make sure that you're consistent, at least.

10.1.3 Test resilience

The menu item test didn't test for several things. For one, it didn't test whether the list item or link had any specific styling. Also, the list item could have content other than the link (maybe an icon), which wouldn't break the functionality of this component. But if you remember the text description of our component, we didn't mention those things, so I argue that those behaviors are not part of what the component *does*.

Is that fact a feature or a bug? The answer probably depends on your application. But many people agree that testing only the important parts and allowing other parts to change makes your application and tests more resilient to change.

Let's see how this process works. Suppose that several months later, some other developer has to update the `MenuItem` component to include a small icon before every element. The following listing shows the new component.

Listing 10.2 Unit-testing `MenuItem` with an icon

```
export function MenuItem({ href, label }) {
  return (
    <li>
      <a href={href} title={label}>
        <img     #1
          src="/images/link.png"    #1
          width="20"   #1
          alt=""    #1
        />    #1
        {label}
      </a>
    </li>
  );
}
```

#1 We've put a small link icon in every link component.

EXAMPLE: MENUITEM-ICON

This example is in the `menuitem-icon` folder. You can run the tests in that example by running this command in the source folder:

```
$ npm test -w ch10/menuitem-icon
```

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/ch10-menuitem-icon**.

Let's try to run the tests again by using `npm run test:once -w ch10/menuitem-icon`:

```
npm run test:once -w ch10/menuitem-icon    #1
> @jrr/ch10-menuitem-icon@0.0.0 test:once
> vitest run
 RUN  v0.34.6 <path>/ch10/menuitem-icon
 ✓ src/MenuItem.test.jsx (1)
   ✓ MenuItem renders a link in a list item
 Test Files  1 passed (1)    #2
      Tests  1 passed (1)    #2
   Start at  00:20:45
   Duration  1.03s (transform 47ms, setup 0ms, collect 213ms,
     tests 65ms, environment 377ms, prepare 100ms)
```

#1 We run the test:once script in the new example workspace.
#2 The test still passes!

This result is called *test resiliency*. We want our tests to pass when we make nonbreaking changes and fail only when we make significant changes to functionality. Suppose that instead of adding an icon, we added a class name or two to the elements. These things shouldn't cause the test to fail because we're not changing any functionality.

Other test paradigms would break if we made these changes. We could decide to match the exact JSX output of a component to a statically defined one, for example. If we took this approach, the test would break if the JSX changed in even in the slightest way, including changes to class names or other things that may not affect the functionality of the component.

Different schools of thought exist, but I believe in the former approach: testing functionality only at a user-interaction level. This approach is one of the governing principles behind the Testing Library package, which is why I recommend it. Next, let's see what happens when we add stateful components and other interactivity to the mix.

10.2 Testing interactive components

In section 10.1, we made a simple test for a simple static component. The component never changes unless the properties change, so we don't have a lot to test. When we start testing more interactive components, however, the tests get a lot more interesting.

If a button activates something, we want our test to simulate a user clicking that button and then verifying that the result occurs. If we have input fields, the interaction sequence can get even more complex, with the input field receiving focus, receiving keyboard inputs, losing focus, and so on.

10.2.1 Testing a stateful component

Let's go back to Old Faithful: the click counter. This counter is one of the most basic components to create in React, and you probably made several variants when you were learning how stateful components work. The following listing shows a two-button variant.

Listing 10.3 The click counter

```
import { useState } from "react";
export function Counter({ start = 0 }) {    #1
  const [counter, setCounter] = useState(start);     #2
  const update = (delta) => () =>
    setCounter((value) => value + delta);
  return (
    <main>
      <h1>Counter: {counter}</h1>     #3
      <button onClick={update(1)}>     #4
        Increment     #4
      </button>   #4
      <button onClick={update(-1)}>     #4
        Decrement     #4
      </button>    #4
    </main>   #4
  );
}
```

#1 This component has four interesting bits. First is the initial value passed as a property, which defaults to 0.
#2 Then we have the stateful variable counter and its update function.
#3 We display the counter in a heading here.
#4 We update the counter in the two buttons for up and down, respectively.

Before we start testing anything, it might make sense to formulate in plain English what we want to test:

- The counter component should display the current value in a heading.
- The counter component should initialize the counter to the given `start` property.
- The counter component should initialize the counter to `0` if no `start` property is given.
- When you click the increment button, the counter should increment and the heading should update.
- When you click the decrement button, the counter should decrement and the heading should update.
- If you instantiate the counter and later pass a different `start` property to the component, the counter does not change to reflect the value of the `start` property and can be changed only via the buttons.

That list seems to be exhaustive. First, let's start the test runner in the terminal. The test runner will run the tests automatically as we write and save them, checking them live. To enable that feature, execute `npm test -w ch10/click-counter` in the terminal.

We can implement the first three tests quickly (and we'll test only the first one implicitly via the two others). The following listing shows how.

Listing 10.4 Testing the static parts of the click counter

```
import { render, screen } from "@testing-library/react";
import "@testing-library/jest-dom";
import { Counter } from "./Counter";
test(     #1
  "Counter should start at the given value",    #1
  () => {   #1
  // ARRANGE    #1
  render(<Counter start={10} />);    #1
  // ASSERT    #1
    const heading = screen.getByRole(    #1
      "heading",    #1
      { name: "Counter: 10" },  #1
    );    #1
  expect(heading).toBeInTheDocument();    #1
});
test(    #2
  "Counter should start at 0 if no value is given",   #2
  () => {   #2
  // ARRANGE    #2
  render(<Counter />);   #2
  // ASSERT    #2
    const heading = screen.getByRole(    #2
      "heading",    #2
      { name: "Counter: 0" },   #2
    );    #2
  expect(heading).toBeInTheDocument();   #2
});
```

#1 In the first test, we start the counter at 10 and verify that it displays a heading with that value.
#2 In the second test, we start the counter without a start value and verify that it displays a heading with 0.

The test runner should pick up this new test automatically and display the following:

```
✓ src/Counter.test.jsx (2)
   ✓ Counter should start at the given value
   ✓ Counter should start at 0 if no value is given
 Test Files  1 passed (1)
      Tests  2 passed (2)
   Start at  20:33:07
   Duration  381ms
```

Great start! Next, we need to test the buttons. How do we click a button? Do we find the button and call its `click()` method? Do we send an event to the button by using `dispatchEvent`? Do we hover the button first (by using an event) because that's what a real user does?

We don't have to worry about any of those things because we will use another library (a package that is also part of Testing Library): `user-event`. This library has great methods for easily interacting with elements as a user would. Do you need to click a button, drag a slider, hover over an image, or type in an input field? All those things and many more are neatly solved by the `user-event` library. Let's add the test for clicking the increment button.

Listing 10.5 Clicking a button

```
import { render, screen } from "@testing-library/react";
import userEvent
  from "@testing-library/user-event";     #1
import "@testing-library/jest-dom";
import { Counter } from "./Counter";
test(     #2
  "Counter should start at the given value",    #2
  () => {...},   #2
);    #2
test(    #2
  "Counter should start at 0 if no value is given",   #2
  () => {...},    #2
);    #2
test(    #3
  "Counter should increment when button is clicked",   #3
  async () => {    #3
  // ARRANGE
  render(<Counter />);
  // ACT
  const user = userEvent.setup();    #4
  const increment = screen.getByRole(     #5
    "button",    #5
    { name: "Increment" },    #5
  );    #5
  await user.click(increment);     #6
  // ASSERT
  const heading = screen.getByRole(    #7
    "heading",    #7
    { name: "Counter: 1" },    #7
  );    #7
  expect(heading).toBeInTheDocument();
});
```

#1 We import the userEvent library.
#2 We collapsed the old tests to reduce clutter, but they're still there.
#3 Notice that we added async to the function definition because we'll have to wait for something to complete before progressing.
#4 To use the user-event library, we start the session by calling the setup method and getting a user session back.
#5 We find the button we need to click by role and name (because we have two buttons now, so the name is required).
#6 We click the button via the user session. This line of code is the one we have to wait for—hence, the await in front. This line works because it returns a promise that resolves when everything that happens from the click has been executed.
#7 Finally, we validate that the counter has updated.

If you still have the test runner running in the background, check the terminal to see whether the new test also passes. Ideally, you'll see something like this:

```
✓ src/Counter.test.jsx (3)
   ✓ Counter should start at the given value
   ✓ Counter should start at 0 if no value is given
   ✓ Counter should increment when button is clicked
 Test Files  1 passed (1)
      Tests  3 passed (3)
   Start at  20:34:45
   Duration  413ms
```

Woohoo! That's awesome. We can easily copy that last test to test the decrement button. For that final test, what happens if we update the property passed to a component?

I've hidden one thing from you (probably a ton of things, but this one is relevant now). The `render()` function from React Testing Library returns something useful: an object with some convenient properties. One of those properties is the `rerender` function, which does exactly what it says on the tin: it allows you to render the component with potentially different properties but in the same instance, so it will update the existing component. The test for the last item in our plain-English wish list becomes what you see in the following listing.

Listing 10.6 Re-rendering a component

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import "@testing-library/jest-dom";
import { Counter } from "./Counter";
test("Counter should start at the given value", () => {...});
test("Counter should start at 0 if no value is given", () => {...});
test("Counter should increment when button is pressed", async () => {...});
test("Counter should decrement when button is pressed", async () => {...});
test(
  "Counter should not update value if passed start property changes",
  () => {
  // ARRANGE
    const { rerender } =    #1
      render(<Counter start={10} />);    #1
  // ACT
  rerender(<Counter start={20} />);    #2
  // ASSERT
  const heading = screen.getByRole(    #3
    "heading",   #3
    { name: "Counter: 10" },   #3
  );   #3
  expect(heading).toBeInTheDocument();   #3
});
```

#1 We destructure the rerender function from the return value of the render call.
#2 We rerender the component with a new property passed in. Note that this happens synchronously, so there's no need for await here (or for async in the test in general).
#3 We test the heading as usual.

Again, you can check the test runner terminal window and see that all five tests are passing.

ARRANGE, ACT, ASSERT

You may have noticed the `ARRANGE`, `ACT`, `ASSERT` comments in the tests. (Sometimes, the `ACT` part is skipped if it's not relevant.) What are those comments about?

These comments are homages to the classic pattern of organizing your tests this way for a more structured approach. That pattern is unsurprisingly called *Arrange-Act-Assert*. (Programmers are terrible at creative writing, though they do get a few points for alliteration!)

It is not uncommon to see people adding these comments to indicate clearly what happens on which line. Note that, sometimes, multiple rounds of `ACT` and `ASSERT` take place, as we will see in listing 10.11.

While we're here, let's make the tests a bit nicer to look at. We are often looking for the heading to check the current value, so why not make that check a bit easier? We also need a reference to the buttons, which we could make easier to access.

It is common practice to create a custom setup function at the top of a test file that initializes the test case for use throughout the component. I included this final test file in the following listing.

Listing 10.7 The full test file with a convenient setup function

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import "@testing-library/jest-dom";
import { Counter } from "./Counter";
function setup(start) {    #1
  const { rerender } = render(<Counter start={start} />);
  const getCounter = (val) => screen.getByRole(     #2
    "heading",    #2
    { name: `Counter: ${val}` },    #2
  );    #2
  const increment = screen.getByRole("button", { name: "Increment" });
  const decrement = screen.getByRole("button", { name: "Decrement" });
  const user = userEvent.setup();
  return {    #3
    getCounter,
    increment,
    decrement,
    user,
    rerender: (newStart) => rerender(<Counter start={newStart} />),
  };
}
test("Counter should start at the given value", () => {
  // ARRANGE
  const { getCounter } = setup(10);     #4
  // ASSERT
  expect(getCounter(10)).toBeInTheDocument();    #5
});
test("Counter should start at 0 if no value is given", () => {
  // ARRANGE
  const { getCounter } = setup();
  // ASSERT
  expect(getCounter(0)).toBeInTheDocument();
});
test("Counter should increment when button is pressed", async () => {
  // ARRANGE
  const { getCounter, increment, user } = setup();
  // ACT
  await user.click(increment);     #6
  // ASSERT
  expect(getCounter(1)).toBeInTheDocument();
});
test("Counter should decrement when button is clicked", async () => {
  // ARRANGE
  const { getCounter, decrement, user } = setup();
  // ACT
  await user.click(decrement);
  // ASSERT
  expect(getCounter(-1)).toBeInTheDocument();
});
test("Counter should not update value if passed start property changes", () => {
  // ARRANGE
  const { getCounter, rerender } = setup(10);
  // ACT
  rerender(20);     #7
  // ASSERT
  expect(getCounter(10)).toBeInTheDocument();
});
```

#1 We define the setup function outside the test cases.
#2 This setup function defines a getCounter function that allows us to query for the heading with a given counter value.
#3 We return an object with all the things we're going to need for all the tests.
#4 Here, we set up a counter with an initial value of 10 and have easy access to the getCounter function as a result.
#5 We invoke the getCounter function with the expected counter value.
#6 We can click the increment button in a single line because we get both the button reference and the user session instance from the setup function.
#7 We can even re-render the component with a new start value via the rerender property.

EXAMPLE: CLICK-COUNTER

This example is in the `click-counter` folder. You can run the tests in that example by running this command in the source folder:

```
$ npm test -w ch10/click-counter
```

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/ch10-click-counter**.

Now, that's a clean-looking test suite (**chef's kiss**)! All the tests have one line of arrangement, optionally one line of acting, and, finally, one line of assertion. It's immediately clear what every test does both from the description and from reading the code. This result almost brings a tear to one's eye. Almost.

10.2.2 Testing callbacks

How do we test a component when part of its function is invoking a callback passed as a parameter? This may not seem like a big deal, but this pattern is common, so it's very important to test such a component properly so that it doesn't break down due to a typo and end up being pushed to production while broken.

Consider another standard application that every React developer created *ad infinitum* in their early days: the to-do application. For our example application, we have a component that shows all the items and a delete button next to each item. When the delete button is clicked, the `onDelete` callback property is invoked with the to-do item in question. The following listing shows the source code for the `Items` component.

Listing 10.8 The items component from a to-do app

```
export function Items({ items, onDelete }) {     #1
  return (
    <>
      <h2>Todo items</h2>
      <ul>
        {items.map((todo) => (
          <li key={todo}>
            {todo}
            <button
              title={`Delete '${todo}'`}    #2
              onClick={() => onDelete(todo)}    #2
            >    #2
              <span aria-hidden>×</span>    #3
            </button>
          </li>
        ))}
      </ul>
    </>
  );
}
```

#1 The component accepts a property that is a function to be invoked when the user interacts with the component.
#2 We add a descriptive title to the button to serve as its accessible name for various textual interfaces.
#3 The title is used as the accessible name only if the button does not have text content, so we hide the X icon inside the button from screen readers by using aria-hidden.

Let's see how we test this component. We want to pass in some callback that we can test if it's invoked later, and Vitest has a built-in method for that purpose: `vi.fn()`. The following listing shows how to use this method.

Listing 10.9 Testing the items component from a `Todo` app

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import "@testing-library/jest-dom";
import { vi } from "vitest";
import { Items } from "./Items";
test("Items should call onDelete callback
↪when item is deleted", async () => {
  // ARRANGE
  const items = ["Item A", "Item B"];
  const mockDelete = vi.fn();    #1
  render(     #2
    <Items items={items} onDelete={mockDelete} />,    #2
  );
  // ACT
  const user = userEvent.setup();
  const secondItemDelete = screen.getByRole("button", {
    name: "Delete 'Item B'",
  });
  await user.click(secondItemDelete);     #3

  // ASSERT
  expect(mockDelete)    #4
    .toHaveBeenCalledWith("Item B");  #4
});
```

#1 We create a variable with vi.fn. This method returns a mock function, which we can later ask whether it has been called, how many times, with which arguments, and so on.
#2 We pass in this mock function like any other property.
#3 We click the button as we now know how to do (but remember to await the click).
#4 Finally, we expect the mock function to have been called with the string "Item B" because we clicked the button next to the second item.

EXAMPLE: TODO

This example is in the `todo` folder. You can run the tests in that example by running this command in the source folder:

```
$ npm test -w ch10/todo
```

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/ch10-todo**.

That code is fairly easy to read (except for the slightly cryptic `vi.fn` call). A mock function is versatile and can be used every time you have to pass in a function and control what it does or what it returns, or to validate its invocation. Please check the documentation at **https://vitest.dev/guide/mocking.html#functions** to find out what mocking functions can do for you.

10.2.3 Testing a form

Now we get to one of the more complex types of interactive components: the form. As you will see, it's not too complex to test when we apply what we know.

For this example, I'm using a slightly more complex application to fully illustrate how to integrate a test into a somewhat larger project. The application is a timer; the user can define multiple timers and start and stop each one individually. When you want to add a timer, you input the minutes and seconds in a simple form rendered by the `AddTimer` component.

Listing 10.10 A form to add a new timer

```
import { useState } from "react";
import { Button } from "./Button";
import { Input } from "./Input";
const EMPTY = { minutes: 0, seconds: 0 };
export function AddTimer({ onAdd }) {     #1
  const [data, setData] = useState(EMPTY);
  const onChange = (evt) => {     #2
    setData((oldData) => ({    #2
      ...oldData,    #2
      [evt.target.name]:    #2
        evt.target.valueAsNumber,   #2
    }));   #2
  };   #2
  const onSubmit = (evt) => {     #3
    evt.preventDefault();    #3
    onAdd(data.minutes * 60 + data.seconds);   #3
    setData(EMPTY);   #3
  };   #3
  return (
    <form onSubmit={onSubmit} className="timer timer-new">
      <ul className="parts">
        <Input
          name="minutes"     #4
          value={data.minutes}
          onChange={onChange}
        />
        <li className="colon">:</li>
        <Input
          name="seconds"    #4
          value={data.seconds}
          onChange={onChange}
        />
      </ul>
      <Button icon="play" label="Start" />     #5
    </form>
  );
}
```

#1 The only property is a callback.
#2 When anything in the form changes, we update local state data.
#3 When the form is submitted, we invoke the callback with the form data and reset it.
#4 The two inputs have labels—"minutes" and "seconds"—that also serve as their accessible names.
#5 We also have a submit button in the form labeled "Start".

We don't do any error handling here. Also, we don't handle the case in which the user submits the form but the time is zero, so we don't want to bother testing for that. We want to test whether the callback is invoked correctly when the user inputs values and clicks the submit button. To do this test, we need to be able to find the relevant elements and interact with them appropriately.

We already know how to find the button (`getByRole("button", { name: "Start" })`) and interact with it (`user.click(button)`), but how do we find the inputs? The default role for a regular input field is `textbox`, but for a number input field, the implicit role is `spinbutton`

because you can use the arrow keys to update the value as well as type it. We interact with it through the user event method `.type()`, simply passing in what we want to type in the input field. Then the `user-event` library correctly submits all the events involved in typing: focusing the fields, inputting the characters, triggering the change handler after each character, and so on. Putting everything together, we can test this component by using the test suite you see in the following listing.

Listing 10.11 Testing a form

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import "@testing-library/jest-dom";
import { vi } from "vitest";
import { AddTimer } from "./AddTimer";
function setup() {     #1


  const onAdd = vi.fn();    #2
  render(<AddTimer onAdd={onAdd} />);
  const minutes = screen.getByRole(   #3
    "spinbutton",    #3
    {name: "minutes"},   #3
  );    #3
  const seconds = screen.getByRole( #3
    "spinbutton",   #3
    {name: "seconds"},   #3
  );    #3
  const start = screen.getByRole(    #3
    "button",    #3
    { name: "Start" },    #3
  );    #3
  const user = userEvent.setup();
  return { onAdd, minutes, seconds, start, user };
}
test("AddTimer should invoke callback when submitted", async () => {
  // ARRANGE
  const { onAdd, user, minutes, seconds, start } = setup();
  // ASSERT
  expect(minutes).toHaveValue(0);    #4
  expect(seconds).toHaveValue(0);   #4
  // ACT
  await user.type(minutes, "5");    #5
  await user.type(seconds, "30");   #5
  // ASSERT    #5
  expect(minutes).toHaveValue(5);  #5
  expect(seconds).toHaveValue(30);   #5
  // ACT
  await user.click(start);     #6
  // ASSERT  #6
  const expectedNumber = 5 * 60 + 30;    #6
  expect(onAdd)   #6
    .toHaveBeenCalledWith(expectedNumber);   #6
  expect(minutes).toHaveValue(0);    #6
  expect(seconds).toHaveValue(0);   #6
});
```

#1 We start with a setup function to make things a bit easier even though we have only a single test case (so far). If we add some error handling later, this function will make adding a second test case much quicker.
#2 The submit callback is a Vitest mock function.
#3 We get references to all the elements we care about.
#4 First, we validate that the form inputs are initially empty.
#5 Then, we type in both fields and validate that the fields contain what we typed.
#6 After we click the start button (which submits the form), we validate that the callback has been invoked with the correct number and that the form inputs have reset to 0.

EXAMPLE: TIMER

This example is in the `timer` folder. You can run the tests in that example by running this command in the source folder:

```
$ npm test -w ch10/timer
```

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/ch10-timer**.

That's it. In general, if you need to interact with a component in any way, the `user-event` library almost always has you covered. In a few specialized cases, that library won't cover your needs, but you can use the more low-level `fireEvent` from the React Testing Library itself.

DON'T TEST IMPLEMENTATION DETAILS

Note that we specifically *don't* test several things. When we enter data in input fields, for example, we don't test whether the internal state variable data is updated correctly because we don't care. If someone changes the component to store the data in some other way, but the component works otherwise, our tests should happily carry on passing and not report anything as being broken (because nothing broke). We call such a thing an *implementation detail*, which you should never test. We test the component as though it is a black box. We can give it some inputs (properties and events) and see what it outputs (callbacks and DOM), but we can and should never look inside it to see how it works (effects, stateful variables, and so on). Those internals are implementation details that may change.

You can implement a specific piece of functionality in hundreds of ways, and your tests should accept all of them, not just the one that you happened to have used.

10.2.4 Testing a hook

Custom hooks in React are used only inside components, so if you thoroughly test all your components, by inference you invariably also test all your custom hooks. Sometimes, however, it's hard to test every combination of inputs for every hook in a big, complicated component that uses many custom hooks. If you have a general hook that's used in multiple places, you want to make sure it works correctly. In such a case, it can be helpful to set up a test case for the custom hook itself. But because hooks are special functions, you can't call a hook like a regular function, so you can't test it like any other function.

For that purpose, React Testing Library comes with a utility for testing hooks specifically. In this example, which involves a game, we need to tell whether keys are pressed to move the correct elements around. We'll create a small hook, `useIsKeyPressed`, that takes the name of a keyboard key and returns `true` or `false` depending on whether the key is pressed. Most important, the hook causes the component to re-render, as

the key is pressed and released throughout the application lifetime. You can see the hook in the following listing.

Listing 10.12 A custom hook

```
import { useState, useEffect } from "react";
export function useIsKeyPressed(target) {     #1
  const [pressed, setPressed] = useState(false);
  useEffect(() => {
    const getHandler =     #2
      (isPressed) =>    #2
      ({ key }) =>    #2
        key === target && setPressed(isPressed);   #2
    const downHandler = getHandler(true);
    const upHandler = getHandler(false);   #2
    window    #2
      .addEventListener("keydown", downHandler);    #2
    window.addEventListener("keyup", upHandler);    #2
    return () => {
      window.removeEventListener("keydown", downHandler);
      window.removeEventListener("keyup", upHandler);
    };
  }, [target]);
  return pressed;
}
```

#1 The hook accepts the name of a key, such as "ArrowDown" or "g".
#2 When the particular key is pressed anywhere inside the window, the Boolean state flag is set to the appropriate state.
#3 When the particular key is pressed anywhere inside the window, the Boolean state flag is set to the appropriate state.

We can test this hook with the `renderHook` function from the React Testing Library. This function returns (among other things) a reference with the current return value from the hook, which we need to check at various times to see whether it has the right value. This test is implemented in the following listing. Beware—the syntax for pressing and releasing a key using the `user-event` library is a bit weird.

TIP For help with the syntax for pressing, holding, and releasing keys, check the full documentation on the React Testing Library website at **https://mng.bz/gv9n**.

Listing 10.13 Testing a custom hook

```
import { renderHook } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { useIsKeyPressed } from "./useIsKeyPressed";
test(
  "useIsKeyPressed should react to the target key and only that key",
  async () => {
  // ARRANGE
    const { result } =    #1
      renderHook(() => useIsKeyPressed("h"));    #1
  const user = userEvent.setup();
  // ASSERT
  expect(result.current).toBe(false);     #2
  // ACT
  await user.keyboard("{f>}");     #3
  // ASSERT    #3
  expect(result.current).toBe(false);   #3
  // ACT
  await user.keyboard("{h>}");    #4
  // ASSERT  #4
  expect(result.current).toBe(true);    #4
  // ACT
  await user.keyboard("{/f}");     #5
  // ASSERT    #5
  expect(result.current).toBe(true);    #5
  // ACT
  await user.keyboard("{/h}");     #6
  // ASSERT    #6
  expect(result.current).toBe(false);   #6
});
```

#1 We render the hook and destruct the result reference from the response. We set the hook up to look for the "h" key.
#2 First, we verify that the key is not pressed at the beginning.
#3 Then we press and hold the "f" key, which should not change anything.
#4 We press and hold the "h" key, which should do something. Now the hook returns true.
#5 We release the "f" key. The hook still returns true.
#6 Finally, we release the "h" key. Now the hook correctly returns false.

EXAMPLE: KEYPRESS

This example is in the `keypress` folder. You can run the tests in that example by running this command in the source folder:

```
$ npm test -w ch10/keypress
```

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/ch10-keypress**.

TIP `renderHook` has a bunch of other capabilities for testing more complex hooks, but this example demonstrates the basic way it works and how and when it can be used. If you need to test a hook for some application, it's always a good idea to check the documentation, located at **https://mng.bz/eo0Z**.

10.3 Testing a component with dependencies

Sometimes, you have a component that depends on some external or even some complex internal functionality for your component to work, but that functionality is undesirable or improbable to include in the test of said component. You might have a component that displays the user's current coordinates as reported by the browser's geolocation API. You want to test what your component does when the geo- location functionality is turned off or when the user refuses to give you the data, as well as whether you receive the data. How would you go about automating this test? Moving your computer around to test it from different locations seems to be counterproductive.

Similarly, you might have a component that depends on an external library and the functionality it provides. Let's say that your component loads data from a web server by using Axios, which is a popular data-fetching library—an alternative to the built-in `fetch()` method. We don't want to query our server every time, and we also want to control whether the server responds with an error message or the correct data. Again, we need to figure out how to automate this test.

Finally, we might have some complex internal functionality, such as an app-wide context, that is updated in many ways, but we want to test our component in isolation for various configurations. We need to figure out how we can use our context in a controlled fashion so it can deliver the right inputs to the component we want to test.

All these examples involve mocking. When you *mock* something, you replace it with a tiny bit of pseudocode that doesn't do what the thing normally does; it's a drop-in replacement that you fully control. Let's see how to go about this process.

10.3.1 Mocking the browser API

Suppose that you have a simple component that, at the click of a button, asks the user for their current position, using the browser geolocation API. If the user chooses to divulge this information, their coordinates are displayed; if not, an error message is displayed. Figure 10.1 shows this simple component flow.

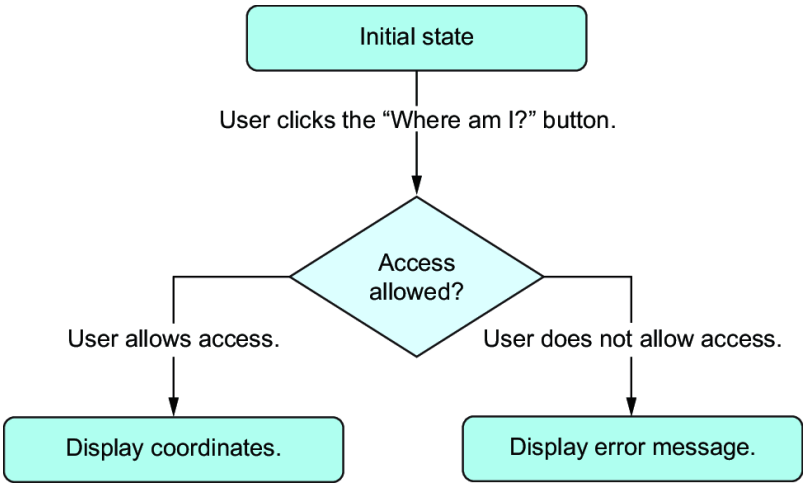

Figure 10.1 When the user clicks the “Where Am I?” button, we ask the browser for the geolocation, using the `navigator.geolocation` API. If we’re allowed to, we show the coordinates; otherwise, we show an error.

We will implement this component with a simple `useState` hook because it’s so small. The following listing shows the implementation.

Listing 10.14 Where am I? No, really, I'm not sure!

```
import { useState } from "react";
const initialState = { status: "initial" };
export function WhereAmI() {
  const [state, setState] = useState(initialState);
  if (state.status === "initial") {
    const onSuccess = ({ coords }) =>    #1
      setState({ status: "success", coords });    #1
    const onError = ({ message }) =>    #1
      setState({ status: "error", message });    #1
    const onClick = () =>   #1
      navigator.geolocation   #1
        .getCurrentPosition(onSuccess, onError);    #1
    return <button onClick={onClick}>Where am I?</button>;
  }
  if (state.status === "error") {
    return <h1>Error: {state.message}</h1>;
  }
  const { latitude, longitude } = state.coords;
  return (
    <h1>
      Coordinates: {latitude}, {longitude}
    </h1>
  );
}
```

#1 We invoke the geolocation API, and depending on which callback is invoked, we transition the state to the success or the error state.

To test this component, we need to control what the geolocation API returns, but we can’t. We can replace the API with one that we do control, however. We mock the global `navigator.geolocation` object and replace the `getCurrentPosition` function with a mock function over which we have full control.

Listing 10.15 Mocking the browser API for test purposes

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import "@testing-library/jest-dom";
import { vi } from "vitest";
import { WhereAmI } from "./WhereAmI";
function setup() {
  const mockAPI = vi.fn();    #1
  global.navigator.geolocation =    #1
    { getCurrentPosition: mockAPI };    #1
  render(<WhereAmI />);
  const button = () => screen.getByRole("button");
  const heading = () => screen.getByRole("heading");
  const user = userEvent.setup();
  return { mockAPI, button, heading, user };
}
describe("WhereAmI component", () => {     #2
  test("should show the coordinates if the user allows it", async () => {
    // ARRANGE
    const { mockAPI, button, heading, user } = setup();
    mockAPI.mockImplementationOnce(     #3
      (success, error) =>    #3
        success({    #3
          coords: { latitude: 55, longitude: 12 }    #3
        })
    );
    // ACT
    await user.click(button());
    // ASSERT
    expect(heading()).toHaveTextContent("Coordinates: 55, 12");
  });
  test(
    "should show an error if the user does not allow access",
    async () => {
    // ARRANGE
    const { mockAPI, button, heading, user } = setup();
      mockAPI.mockImplementationOnce(     #4
        (success, error) =>    #4
      error({ message: "User denied access" })    #4
    );
    // ACT
    await user.click(button());
    // ASSERT
    expect(heading()).toHaveTextContent("Error: User denied access");
  });
});
```

#1 In the setup function, we mock the global navigator geolocation object with our mock function.
#2 This time, we group the test cases in a describe block, which is a way to group related tests. You'll see it in the output.
#3 When we want the API lookup to succeed, we set the implementation of the mock function to call the success callback with a properly formatted response.
#4 When we want the API lookup to fail, we call the error callback in the mock function implementation.

EXAMPLE: GEO

This example is in the `geo` folder. You can run the tests in that example by running this command in the source folder:

```
$ npm test -w ch10/geo
```

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/ch10-geo**.

If we run this code, we get the following output:

```
✓ src/WhereAmI.test.jsx (2)
   ✓ WhereAmI component (2)     #1
     ✓ should show the coordinates if the user     #2
       allows it #2
     ✓ should show an error if the user does    #2
       not allow access    #2
 Test Files  1 passed (1)
      Tests  2 passed (2)
   Start at  21:16:51
   Duration  400ms
```

#1 The describe block is used to group related tests. Describe blocks can be nested as much as necessary.
#2 Each test inside a describe block is indented a bit further to show the hierarchy.
Using this method, you can mock any API that your components might be using from the browser, including local storage, network requests (`fetch`), battery status, and screen capture.

10.3.2 Mocking a library

In this example, we want to make a list of the first 10 starships in some famous movie series about starships from a galaxy far, far away. We will use the convenient `swapi.dev` resource; feel free to guess what the `sw` part of that URL stands for. That web service has a (paginated) list of starships available at this URL: **https://swapi.dev/api/starships/**. We'll request only the first page of results for this exercise, but you can add pagination if you're so inclined.

To make this network request, we could use the built-in `fetch()` function, but for this exercise, we will use the Axios library, which is a popular library for making network requests. When we use Axios, we can implement this `StarshipList` component as in the following listing.

Listing 10.16 A component with a network request using Axios

```
import { useState, useEffect } from "react";
import axios from "axios";
const URL = "https://swapi.dev/api/starships/";
export function StarshipList() {
  const [state, setState] = useState({ status: "initial" });
  useEffect(() => {
    setState({ status: "loading" });
    axios
      .get(URL)     #1
      .then(({ data }) =>     #2
        setState({    #2
          status: "success",   #2
          list: data.results },   #2
        )   #2
      )
      .catch(({ message }) =>     #3
        setState({   #3
          status: "error",    #3
          error: message },   #3
        )    #3
      );    #3
  }, []);
  if (["loading", "initial"].includes(state.status)) {
    return <h1>Loading...</h1>;
  }
  if (state.status === "error") {
    return <h1>Error: {state.error}</h1>;
  }
  return (
    <>
      <h1>List of Starships:</h1>
      <table border="1">
        <thead>
          <tr>
            <th>Name</th>
            <th>Model</th>
            <th>Class</th>
          </tr>
        </thead>
        <tbody>
          {state.list.map(({ url, name, model, starship_class }) => (
            <tr key={url}>
              <td>{name}</td>
              <td>{model}</td>
              <td>{starship_class}</td>
            </tr>
          ))}
        </tbody>
      </table>
    </>
  );
}
```

#1 Using Axios, we make a request for the given URL.

#2 If the request succeeds, we update the state accordingly.

#3 If the request fails, we also update the state, but to a slightly different status.

This component requests that URL every time it is loaded because that's what we want it to do. We don't want our tests to do the same thing, however. For one thing, network requests are slow; more important, this URL is an external service, and we don't want to bombard it with requests or hit any rate limits on requests from the same IP address. Finally, we would have a hard time testing the request failure state because how can we ensure that the request fails when we don't control the backend?

To avoid making the request to the server but still test the component, we can mock out the entire Axios library and replace it with a mock function that we can control within our tests. We can set that function to return either a positive or negative result, and we can validate that the component works correctly. The following listing implements this test suite.

Listing 10.17 Mocking a library when testing a component

```
import { render, screen } from "@testing-library/react";
import "@testing-library/jest-dom";
import axios from "axios";     #1
import { vi } from "vitest";
import { StarshipList } from "./StarshipList";
vi.mock("axios", () => ({     #2
  default: { get: vi.fn() },   #2
}));    #2
describe("StarshipList component", () => {
  test("should initially be in a loading state while fetching", () => {
    // ARRANGE
    axios.get.mockImplementationOnce(     #3
     () => new Promise(() => {})   #3
    );    #3
    render(<StarshipList />);
    // ASSERT
    const heading = screen.getByRole("heading", { name: "Loading..." });
    expect(heading).toBeInTheDocument();
  });
  test("should show an error message on failure", async () => {
    // ARRANGE
    axios.get.mockImplementationOnce(() =>     #4
      Promise    #4
        .reject({ message: "Request failed" })   #4
    );   #4
    render(<StarshipList />);
    // ASSERT
    const title =    #5
      await screen.findByRole("heading",   #5
      { name: "Error: Request failed" },    #5
    );   #5
    expect(title).toBeInTheDocument();
  });
  test("should show a list of ships when all goes well", async () => {
    // ARRANGE
    axios.get.mockImplementationOnce(() =>    #6
      Promise.resolve({    #6
        data: {results: [    #6
          { name: "Tardis", url: "/tardis" },   #6
        ] },    #6
      })   #6
    );
    render(<StarshipList />);
    // ASSERT
    const heading = await screen.findByRole("heading", {
      name: "List of Starships:",
    });
    expect(heading).toBeInTheDocument();
    const firstName = screen.getByRole(    #7
      "cell",    #7
      { name: "Tardis" },    #7
    );    #7
    expect(firstName).toBeInTheDocument();   #7
  });
});
```

#1 We import the Axios library here, but because we mock it (later), the mock implementation is imported, not the real library.
#2 The library mock is created here, and even though it occurs after all the imports, Vitest runs all mocks before any imports are resolved, so the library will be mocked everywhere it is used.
#3 When we want to test the loading state, we make sure that the promise returned from the Axios get method never settles.
#4 To test the error state, we return a rejected promise with an appropriate error message.
#5 Because the promise is resolved inside an effect inside the component, the error message won't be visible on first render of the component. If we wait a while, however, that message will appear. We can use the findByRole method for that purpose, as it waits for the given node to be in the document and throws an error if that doesn't happen within a reasonable time (default timeout: 5 seconds).
#6 To test the successful case, we return a resolved promise, and we must be careful to format the response as the real response would be formatted.
#7 We test not only the headline but also the rendered table rows to check whether they match the mock data.

EXAMPLE: AXIOS

This example is in the `axios` folder. You can run the tests in that example by running this command in the source folder:

```
$ npm test -w ch10/axios
```

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/ch10-axios**.

Although this component is extremely simple (and a bit naive), it does show the power of mocking a whole library. You can use the same method to mock internal files by using a relative mock path such as `vi.mock("../someLibrary", () => { ... })`. This approach is a handy way to test only the component in question—not all sorts of dependencies, which might have weird side effects that you don't want to trigger.

NOTE For networking libraries, I recommend using Mock Service Worker (MSW) instead to mock out the entire backend rather than the network library itself, which is a much more reliable and future-proof method of mocking the API layer. Feel free to go back to chapter 9, which discusses how MSW works. The method described in this section is still valid and useful, however.

10.3.3 Mocking a context

We're going to go back to an application that we've already implemented in many iterations through the book: the dark mode application. We first implemented it in chapter 2 but have since amended it in several iterations. This application has several components, several of which require access to different parts of a context provided by an app-wide provider.

We use a selectable context from the `use-context-selector` library rather than the built-in context from React itself in this application. The principles used in this section are identical to using a React context, how-

ever. Nothing changes except that we would import `createContext` and `useContext` from the React package rather than from the `use-context-selector` package (and we couldn't use selectors, of course).
In this application, components are not individually wrapped by a provider (which would serve no purpose), so if we test a component in isolation, it can't access that context and will have access only to the default context defined in the call to `createContext()`. The following listing shows one of those components—the button—from the dark mode application.

Listing 10.18 A context-dependent button component

```
import { useContextSelector }     #1
  from "use-context-selector";   #1
import { DarkModeContext }   #1
  from "./DarkModeContext";   #1
function Button({ children, ...rest }) {
  const isDarkMode = useContextSelector(     #2
    DarkModeContext,    #2
    (contextValue) => contextValue.isDarkMode    #2
  );    #2
  const style = {
    backgroundColor:
      isDarkMode ? "#333" : "#CCC",     #3
    border: "1px solid",
    color: "inherit",
  };
  return (
    <button style={style} {...rest}>
      {children}
    </button>
  );
}
```

#1 We import the relevant parts from outside this component.
#2 Then we select the isDarkMode flag from the context.
#3 Finally, we style the button based on the flag, using inline styles. (How naive we were back then).

This component works well in the context of the full application, but when we test it in isolation, we need to surround the button with the relevant values in the proper context.

Listing 10.19 Wrapping the button in a mock context for testing

```
import { render, screen } from "@testing-library/react";
import "@testing-library/jest-dom";
import { Button } from "./Button";
import { DarkModeContext }     #1
  from "./DarkModeContext";    #1
function setup(text, isDarkMode = false) {     #2
  const value = { isDarkMode };    #2
  render(     #3
    <DarkModeContext.Provider value={value}>    #3
      <Button>{text}</Button>    #3
    </DarkModeContext.Provider>    #3
  );    #3
  return screen.getByRole("button");
}
describe("Button component", () => {
  test("should render in light mode", () => {
    // ARRANGE
    const button = setup("Click me");    #4
    // ASSERT
    expect(button).toHaveTextContent("Click me");
    expect(button.style.backgroundColor)     #5
      .toBe("rgb(204, 204, 204)");   #5
  });
  test("should render in dark mode if enabled", () => {
    // ARRANGE
    const button = setup("Click me", true); #6
    // ASSERT
    expect(button.style.backgroundColor)    #7
      .toBe("rgb(51, 51, 51)");   #7
  });
});
```

#1 We have to import the raw context because we need to provide it.
#2 We create the context value dynamically.
#3 We render the button wrapped in a context provider with the dynamic value.
#4 When we want to test light mode, we use the default value (false).
#5 We have to validate the style this way, which we can do via button.style because we used inline styles. Had we used some CSS rule, we would have to use getComputedStyle.
#6 When we test the dark mode variant of the button, we make sure to set the flag.
#7 We test the button color in the same way.

That code is relatively straightforward. Let's see how we test using a function from the context. In the toggle button component, we use the `toggleDarkMode` function from the context.

Listing 10.20 The toggle button using a context function

```
import { useContextSelector } from "use-context-selector";
import { Button } from "./Button";
import { DarkModeContext } from "./DarkModeContext";
export function ToggleButton() {
  const toggleDarkMode = useContextSelector(     #1
    DarkModeContext,   #1
    (contextValue) => contextValue.toggleDarkMode   #1
  );    #1
  return (
    <Button onClick={toggleDarkMode}>    #2
      Toggle mode
    </Button>
  );
}
```

#1 Retrieves the toggleDarkMode callback from the context
#2 Assigns the callback to the onClick event handler on the button

We already know how to test this component. We know how to mock a context (from listing 10.19), and we know how to test a callback (from listing 10.9), so we need to put these two things together, as I've done in the following listing.

Listing 10.21 Wrapping the button in a mock context for testing

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import "@testing-library/jest-dom";
import { vi } from "vitest";
import { ToggleButton } from "./ToggleButton";
import { DarkModeContext } from "./DarkModeContext";
function setup() {
  const toggleDarkMode = vi.fn();    #1
  const value = { toggleDarkMode };    #1
  render(
    <DarkModeContext.Provider value={value}>
      <ToggleButton />
    </DarkModeContext.Provider>
  );
  return { button: screen.getByRole("button"), toggleDarkMode };
}
describe("ToggleButton component", () => {
  test("should invoke the toggle on click", async () => {
    // ARRANGE
    const { button, toggleDarkMode } = setup();
    // ACT
    const user = userEvent.setup();
    await user.click(button);
    // ASSERT
    expect(toggleDarkMode)     #2
      .toHaveBeenCalledTimes(1);    #2
  });
});
```

#1 Again, we set up the button wrapped in a context provider with a custom value containing a mock function.
#2 We check whether the mock function is invoked once when we click the button.

EXAMPLE: DARK-MODE

This example is in the `dark-mode` folder. You can run the tests in that example by running this command in the source folder:

```
$ npm test -w ch10/dark-mode
```

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/ch10-dark-mode**.

Running this code gives us slightly different output because now our project has multiple test files. The output from the example with the two tests looks like this:

```
✓ src/Button.test.jsx (2)
 ✓ src/ToggleButton.test.jsx (1)
 Test Files  2 passed (2)     #1
      Tests  3 passed (3)   #1
   Start at  21:41:44
   Duration  1.09s (transform 67ms, setup 1ms, collect 565ms, tests 168ms, environment 775ms,
```

#1 Both files, both suites, and all three tests pass. Looking good!

Again, this context is simple. Testing a more complex context requires doing these same tricks with more variables. If you have multiple contexts, you can wrap your component in all of them.

Summary

- Testing your components is a great way to future-proof your application against bugs and regressions when you add new functionality or change existing functionality down the line.
- An application created with Vite comes without testing infrastructure, but we can easily add it in with Vitest. You may want to extend the testing infrastructure with some additional packages from Testing Library, including (most importantly) the React package.
- You can test both static and interactive components in a fairly straightforward and easy-to-understand manner if you follow the Arrange-Act-Assert pattern of test writing.
- If your component has external or complex internal dependencies, it can be a great help to mock them. You can mock dependencies in several ways that depending on where the dependencies are located.
- You can mock browser APIs, files, folders, and whole libraries. You can also mock a context, which is going to be a useful skill going forward, as you will most likely encounter a lot of contexts in your career.