4 Better code maintenance with developer tooling

This chapter covers

- Writing error-free code with linting
- Increasing productivity with formatting
- Making components more robust with property constraints
- Debugging React applications using developer tools

Code quality degrades over time—unfortunately, a hard fact of our chosen profession. As our web applications grow more complicated, maintaining code quality becomes harder and harder. This is especially the case for multideveloper projects, in which different people invariably do things in different ways, but it is also a problem for single-person projects. If you are working on the same codebase over time, you will pick up new ways of doing things, and these ways will most likely differ from the ones you followed earlier. You might open a file you haven't touched in months and suddenly wonder, "What's going on here?"

Although good comments, documentation, and code structure are all relevant solutions to the above problems, several external tools at your disposal will improve your code quality immensely without requiring you to change your ways or even do more work.

This chapter is a bit special, in that it does not cover anything that's directly related to React or required when writing React. If you're playing with React on your own in smaller projects, you probably don't need much of the contents of this chapter; you can safely skip it until you feel more confident in your coding. If you're joining a small development team, you will need most of this material, so you might as well get ahead of the curve by learning it now. If you're looking to become a professional React developer, this content is practically mandatory.

We will use some of the tools presented here in future chapters, but I'll make sure to refer to the relevant material and give a brief recap where applicable in case you decide to skip this chapter for now. Here in chapter 4, I will introduce four concepts for better code maintenance and the tools that can help you obtain these new improvements:

- *Linting*—Linting is the process of automatically checking your source code for minor programmatic and stylistic errors. We will carry out this task with the `eslint` tool.
- *Formatting*—Formatting is the process of automatically adhering to predefined stylistic coding standards. We will carry out this task with the `prettier` tool.
- *Property constraints*—Property constraints are ways of making components more robust and easier to use by specifying types and ranges for valid properties. We will perform this task by using the built-in feature `.propTypes` combined with the external package `prop-types`.
- *Debugging*—Debugging with React Developer Tools (an external browser plugin) allows you to inspect and debug your React components more effectively, helping you identify and fix problems in your application's UI and state. We will explore the powerful features of React Developer Tools to streamline your debugging process.

Note There's a slight overlap in the responsibility and performance of each of these guiding concerns; I will get more into that topic later.

The Venn diagram in figure 4.1 shows the reach and responsibilities of the four key concerns, which we can think of as guideposts to keep us in line as we write our code.

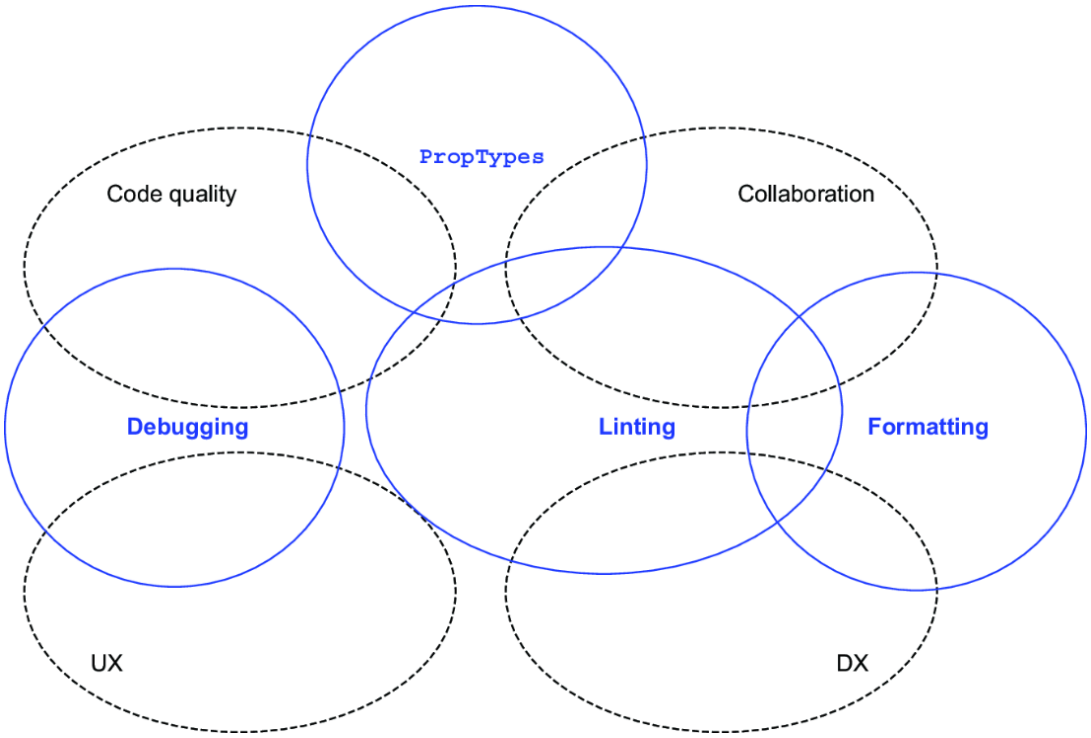

Figure 4.1 Effective software development requires paying attention to code quality, collaboration, user experience (UX), and developer experience (DX), represented here as ovals with dotted lines. The overlapping ovals with solid lines show how our four guideposts affect the outcome of each of these four areas. Notice how the responsibilities overlap.

With these four guideposts configured in your project, you will find errors and problems long before committing your code to your teammates. Did you misspell a variable? Your linter can point that problem out to you. Are you unsure of the best practice for adding newlines inside a long and convoluted piece of code? Your formatter will solve that problem for you automatically. Are you passing the wrong type of property to a component? Property types will highlight this situation. Are you unable to figure out why your component renders all the time? Developer tools will tell you.

A lot of teams are using at least one of these tools, if not all four, and these tools are often configured by default in many setups in common web development frameworks. In fact, some of these tools are so standard that most editors have built-in support for them and almost expect them to be used because their use has become so omnipresent in modern web development.

In this chapter, we will go through each of the four concepts, discussing the reason for using each tool and what problem it solves, as well as some

more technical details about how to set it up, with links to packages and modules that are relevant to your editor.
One thing I can promise you: After you've been using any (or all) of these tools in a React project, you may not want to go back. Things tend to get a whole lot easier with tooling!

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch04**.

4.1 Reducing errors with linting

If you have some lint on your clothes, you can use a lint roller to get rid of them. That tool allows you clean your clothes effectively with minimal effort. In the same way, if you have some minor stylistic or bad-practice problems in your source code, you use a linter tool to get rid of them effectively with minimal effort. Figure 4.2 highlights the responsibilities of linting, with only the relevant parts in focus.

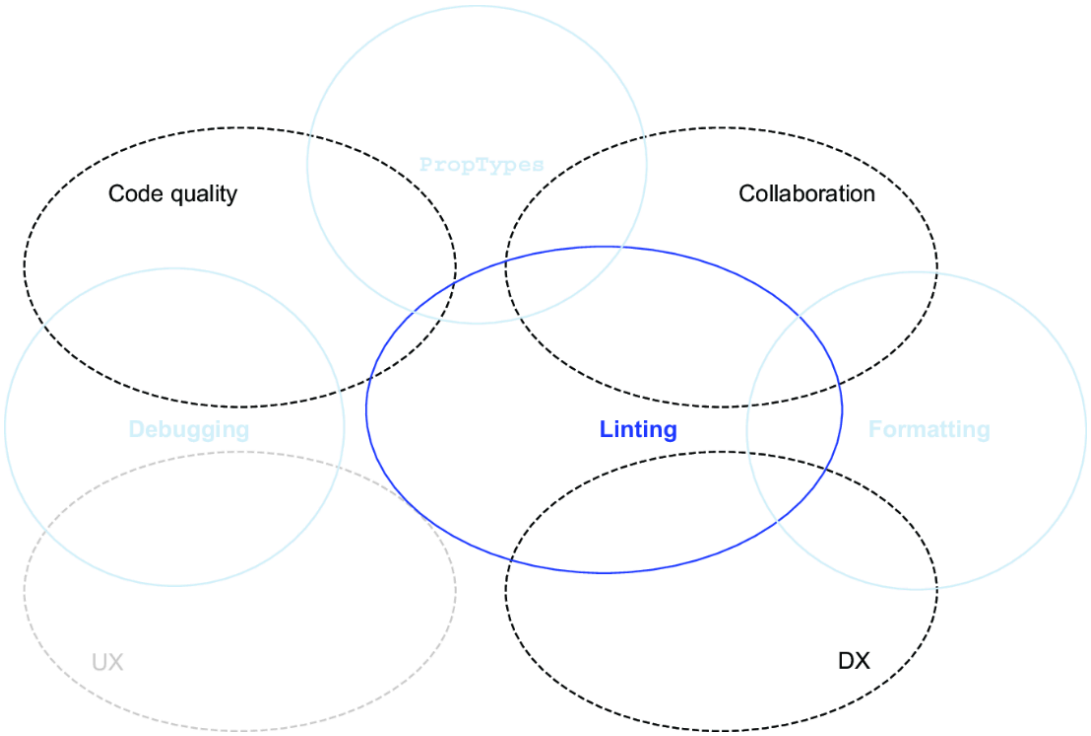

Figure 4.2 Linting affects several aspects of software development, including code quality, collaboration, and DX.

Here's an inexhaustive list of the types of problems that linting can help you solve automatically:

- Potential coding errors due to disambiguation (language weirdness), such as disallowing duplicate keys in an object literal (which doesn't

work as expected anyway)
- Best practices such as requiring multiple returns in the same function to return the same type of variable (which makes the result consistent)
- Stylistic rules, such as requiring newlines at the end of a file

Code linting is becoming more and more popular, to the point where it's almost standard in most projects these days. The primary linter tool used today is ESLint, so that's the one I'll cover in this section.

4.1.1 Problems solved by ESLint

Suppose that you work on a large team, and to make things smoother, you create a coding style guideline that you want all your developers to follow. ESLint is one of the first tools for JavaScript that allows you to codify such a style in enforceable rules, which will be applied automatically to all the code for every developer, regardless of their setup.

You can not only set these rules to inform developers when (and how) they're breaking the style guide but also have ESLint fix a huge percentage of these violations automatically. You can set your system up so that it tries to fix the code as you commit it to the central repository.

4.1.2 ESLint configurations

ESLint consists of more than 300 rules, and you can specify how you want each rule to be enforced in your project. You can also specify the severity of violating a rule. Sometimes, not doing something according to your ruleset is a straight-up error. At other times, you may find that a warning is sufficient, but you might allow the violation if the developer in question finds it to be proper in the given situation.

Suppose that you want your code to be formatted with semicolons at the end of every statement line or that you want your code to never have semicolons. Those rules are two sides of the same coin. You specify a configuration for the rule named `semi`. If you want to enforce semicolons where applicable and enforce them as an error otherwise, you'd configure the rule as

```
semi: ["error", "always"]
```

Conversely, if you want to enforce the rule that nobody can put semicolons anywhere, you'd configure the rule as

```
semi: ["error", "never"]
```

Then you can go over every rule in the ESLint ruleset and specify your preference as you've agreed on with the team. That approach does sound like a lot of work, though, so you could opt for an existing group of configurations that someone else made and recommends.

ESLint itself has a recommended ruleset that includes configurations for most of the rules. You can extend this ruleset and vary it as your organization sees fit. You can find other predefined rulesets; the best known is probably the great set from Airbnb derived from its extensive style guide, which the company gladly posts online for anyone to copy or be inspired by:

- *ESLint rules* (**https://eslint.org/docs/latest/rules**)—Includes a list of configurations included in the ESLint recommended ruleset
- *Airbnb style guide* (**https://github.com/airbnb/javascript**)—Contains information about how to use it in ESLint as well

4.1.3 How to get started using ESLint

To get started using ESLint, follow these three steps:

1. Initialize ESLint in your project.
2. Modify the default configuration file, potentially extending a predefined ruleset and/or custom rule configurations.
3. Enforce the rules in your editor and your build.

STEP 1: INITIALIZE ESLINT CONFIGURATION

To carry out this step, you use `npm`:

```
npm init @eslint/config
```

When it's installed and configured, you're ready to start using the tool.

STEP 2: MODIFY THE CONFIGURATION FILE

As you run the initializer in step 1, a configuration file is added to the root of your project in your desired format. This file is named `<root>/.eslintrc.{js,yml,json}` (with the extension following your choice of format). Note the initial dot in the filename. In this file, you can extend an existing ruleset by adding an `extends` property (shown here in JSON):

```
{
  "extends": "eslint:recommended",
}
```

Also, you can include rules by listing them in a `rules` block:

```
{
  "rules": {
    "semi": ["error", "always"],
    "quotes": ["error", "double"]
  }
}
```

The configuration doesn't take any more work, and you don't even need to do the second part if you're happy with a predefined ruleset.

STEP 3: ENFORCE RULES IN YOUR EDITOR AND BUILD

All the popular editors have a package for ESLint to enforce automatically as you write your code. If your code editor has some form of package manager for extensions, it most likely has an ESLint package. Editors with ESLint support include the classic editors (such as VIM and Emacs) as well as modern ones (such as Visual Studio Code and Brackets).

ESLint is also easy to include in your existing build setup regardless of which setup you use. ESLint works with esbuild (the default build tool used in Vite), webpack, rollup, grunt, gulp, and many other tools. You can see all the integrations for both your editor and your build setup at **https://mng.bz/ng65**.

4.2 Increasing productivity with formatters

Tabs or spaces? Come on—I know you have an opinion, so out with it! But do you wanna know a secret? I don't—not anymore. Since the introduction of Prettier, a commonly used code autoformatter, I don't care one bit. Figure 4.3 highlights the responsibilities of formatting, with only the relevant parts in focus.

Although tabs versus spaces is probably the most hotly contested problem, it is not alone. Other purely formatting-related choices that you can make in your code include

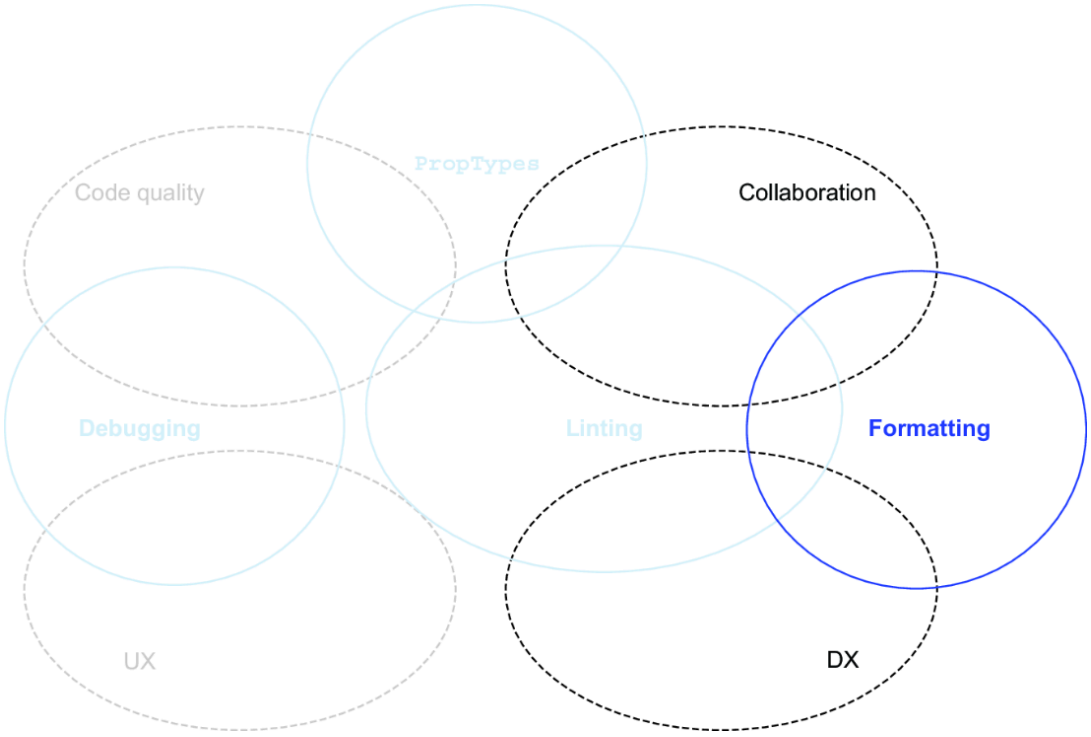

Figure 4.3 Formatting affects two aspects of software development: collaboration and DX.

- *Single or double quotes*
  - Either `var welcome = "Hello";`
  - Or `var welcome = 'Hello';`
- *Newline or space after function definition*
  - Either `function Component() {`
  - Or `function Component()` `{`
- *Space inside brackets in object literals*
  - Either `{property: 1}`
  - Or `{ property: 1 }`

- *Space inside parentheses around function arguments*
  - Either `useState(0)`
  - Or `useState( 0 )`

Sure, none of these matters are relevant to the performance of the code—only to DX. As I've stated again and again, however, DX matters—a lot. It is the difference between joining a project and being productive on day one versus spending days or even weeks getting up to speed and learning the customs of a new project (not that formatting alone takes weeks to master, of course).

Figure 4.4 shows how Prettier can format a messy piece of code into something much more consistent. More important, formatting choices need to be uniform within a project. Everything else leads to pure anarchy. If one file uses tabs and another uses spaces, you're in for a long day at work!

Figure 4.4 A snippet of code before and after Prettier takes a crack at it. This example uses the default settings in Prettier. The "after" code still is not elegant code, as it could be structured better, but at least the formatting is consistent.

4.2.1 Problems solved by Prettier

First, know that Prettier is an *opinionated* ruleset. The makers of Prettier have an opinion about how code should be formatted, and they aren't shy about it. How code should be formatted is constantly up for discussion, of course, but at its base, Prettier is an opinion, so your team doesn't need to have one.

I've worked on development teams that discussed coding standards and style guidelines *ad nauseam*. With Prettier, you eliminate this discussion by referring to the result of *someone else's* discussion. Prettier is that result. A lot (and I mean *a lot*) of developers all over the world, working on all sorts of projects and teams, have weighed in. Prettier is the result and the consensus choice.

The number of problems that Prettier solves is impossible to list, but one of its best features is that you never have to worry about formatting your files again. Often, you have stylistic design choices to make as a developer. Take this code snippet, for example:

```
const someCars = cars.filter((car) => car.make === "Fiat");
```

This snippet could have been formatted in many ways. All of the following variants do the same thing:

```
const someCars = cars
  .filter((car) => car.make === "Fiat");    #1
const someCars = cars.filter(car => car.make === "Fiat");    #2
const someCars = cars.filter((car) => car.make === 'Fiat');   #3
const someCars = cars.filter(
  (car) => car.make === "Fiat"    #4
);
const someCars = cars.filter( ( car ) => car.make === "Fiat" );    #5
```

#1 You could add newlines before method invocations.
#2 You could add or remove parentheses around single-argument arrow functions.
#3 You could use single quotes rather than double quotes.
#4 You could add some line breaks around function arguments
#5 You could add extra spaces around function arguments and parameters.

All these choices are purely stylistic. Some people prefer one style; others prefer another. Style makes no difference after the code is compiled. With Prettier, however, you don't have to make a single one of those choices. You invoke the formatter in your editor, and *voilà*—the correct style is applied automatically. With the default settings in Prettier, you'd get the following result (identical to the original statement):

```
const someCars = cars.filter((car) => car.make === "Fiat");
```

If you try to format this snippet in any other way and run Prettier, your code immediately reverts to this example.

You may be thinking, “But what if the line is longer? I don’t want all my code to run on in long lines!” Well, you’re in luck. Prettier is smart. It analyzes the code and figures out the best way to format any piece of code so that it looks great (under the given ruleset) and doesn’t simply follow specific rules as a dumb machine. So although Prettier would format the preceding code as stated, suppose that the code was a little bit more complex, like this:

```
const someCars = originalListOfCars.filter((car) => car.make === "Fiat"
  && !car.isPickup);
```

If you apply Prettier with the default rules, that code would suddenly become

```
const someCars = originalListOfCars.filter(
  (car) => car.make === "Fiat" && !car.isPickup
);
```

Prettier decided that this line is too long and that breaking it up with the method argument on a newline is the best formatting for the code in question. What if we make this example a bit more complex?

```
const someCars = originalListOfCars.filter(
  (car) => car.make === "Fiat" && !car.isPickup
  ↪&& !car.isHatchback && car.cylinders >= 6
);
```

Prettier once again decides that this line is too long and breaks up the code like this:

```
const someCars = originalListOfCars.filter(
  (car) =>
    car.make === "Fiat" &&
    !car.isPickup &&
    !car.isHatchback &&
    car.cylinders >= 6
);
```

Prettier does not have simple rules like those in ESLint, which say that any array with more than *X* elements has to be in multiple lines and so on. Instead, Prettier adapts dynamically to the code in question and uses whatever formatting looks best (with *best* being an opinion, of course).

4.2.2 Nonstandard rules with Prettier configuration

Although uniformity is good, there are a lot of different people on this planet. Getting the whole world to agree on tabs versus spaces or single versus double quotes is an uphill battle. The good folks behind Prettier initially had the best intentions of unifying the world behind a single standard, but they've since relented; now they allow an ever-growing number of rules to be customized in each setup. So now it is possible to specify whether your particular project wants to use tabs or spaces, single or double quotes, and so on. Not all the internal rules are customizable, but the most hot-button ones are.

You can apply the configuration options when invoking Prettier on the command line or—better yet—by writing them in a configuration object. Prettier loads in the configuration options by using the `cosmiconfig` library, so you can load it in several ways. One of the most common ways is to format the object as JSON and store it in a file named `.prettierrc` or `.prettierrc.json` located at the root of your project. (Note the leading `"."` in the filename, which is common for configuration files like this one.)

TIP See more about how to store the configuration object at **https://prettier.io/docs/en/configuration.html**.

Suppose that you want to format your files with Prettier, but you want your project to use tabs rather than the default spaces, and you want to

exclude semicolons at the end of statements. (Yes, some people want that format. Go figure!) You can set those options on the command line like so:

```
$ prettier --use-tabs --no-semi src/
```

Alternatively, you can create a file located at `<root>/.prettierrc.json` with the following contents:

```
{
  useTabs: true,
  semicolon: false
}
```

Prettier will format your code according to your custom rules but keep everything else in the Prettier ruleset. You've now opined on top of the already opinionated library.

TIP To customize your setup, you have to know what the options are, of course, and you can see the full list on the Prettier documentation website right here: **https://prettier.io/docs/en/options.html**.

Prettier allows a single case of rule customization, which you can do without configuring it specifically or even globally for your project. This rule is about whether the keys of an object go on separate lines or the entire object is defined on a single line (if it fits). Take this piece of code:

```
const car = { make: "Fiat", model: "500", isPickup: false };
```

Prettier will format the code this way if that's how you typed it. But if you added optional newlines before each property, Prettier would allow that option:

```
const car = {
  make: "Fiat",
  model: "500",
  isPickup: false,
};
```

Prettier is even clever enough to know when you want to change from one style to the other. Let's say you have the entire object defined in a single line. If you add a newline before the first property and invoke Prettier, the tool will guess that you want each property to be on a separate line:

```
const car = {
  make: "Fiat", model: "500", isPickup: false };    #1
```

#1 The newline at the start of the second line instructs Prettier that you want the properties to be on separate lines.

Conversely, if you have the object on multiple lines and want it to be more compact, you can remove the newline before the first property, and Prettier will inline the whole thing:

```
const car = {make: "Fiat",     #1
  model: "500",
  isPickup: false,
};
```

#1 The lack of a newline before the first property instructs Prettier that you want the object to be on a single line.

Note that this option works only if the object fits on a single line. If it doesn't, Prettier will always format it on multiple lines.

This feature is readily available to ensure consistency between objects. If you have an array with multiple objects, some of which are formatted on multiple lines because the contents are longer than others, you can ensure that all objects are formatted on multiple lines to make the code look more consistent. Compare the code snippet

```
const cars = [
  { make: "Fiat", model: "500" },
  {
    make: "BMW",
    model: "Individual M760i xDrive Model V12 Excellence THE NEXT 100 YEARS",
  },
];
```

with this code snippet, which forces the first object to be formatted on multiple lines as well:

```
const cars = [
  {
    make: "Fiat",
    model: "500",
  },
  {
    make: "BMW",
    model: "Individual M760i xDrive Model V12 Excellence THE NEXT 100 YEARS",
  },
];
```

This latter version does take up more space but looks better with both objects being styled identically. You can play with the examples in this section in the `prettier` example.

EXAMPLE: PRETTIER

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

```
$ npm run dev -w ch04/prettier
```

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/ch04-prettier**.

4.2.3 How to start using Prettier

Getting started with Prettier normally requires up to four steps, the second and fourth of which are optional but often used:

1. Add Prettier to your project as a package.
2. Add a configuration file if you want to deviate from the standard configuration.
3. Enforce formatting in your editor.
4. Enforce formatting on code commits.

STEP 1: ADD PRETTIER AS A PACKAGE

You use `npm` for this step:

```
npm install --save-dev prettier
```

After it's installed, you're ready to start using the tool.

STEP 2: OPTIONALLY ADD A CONFIGURATION FILE

If you want to deviate from the standard ruleset, feel free to add a configuration file. By convention, this file is saved to `<root>/.prettierrc` in the root of your project. Refer to section 4.2.2 for the available configuration options.

STEP 3: ENFORCE FORMATTING IN YOUR EDITOR

Prettier works in most text editors that allow you to install custom packages. So whether you use Visual Studio Code, Atom, Sublime Text 3, or some other editor, search your package manager for Prettier to find the proper package. You can also check out this page, which has direct links to the relevant packages: **https://prettier.io/docs/en/editors.html**.

STEP 4: OPTIONALLY ENFORCE FORMATTING ON CODE COMMITS

Sometimes, files are changed outside your editor, or a code merge results in files being combined in a way that violates your formatting ruleset. To make sure that you *never, ever* commit any code that violates your formatting ruleset, a common approach is to make sure to run your formatter on any changed files before a commit is approved. To do this, you use what is known as a *precommit hook* (which is a version control hook and not at all related to React hooks), which runs a bit of code before any commit is affected.

Note that this step is optional; not every project needs it. You have many ways to achieve this task, depending on your setup and which other tools you have in your stack, but one of the easiest ways is to use a tool named `lint-staged`.

If you have already run the command for installing `lint-staged` in your project, it automatically picks up the fact that you also have Prettier set up, and it will start validating your changed files against your Prettier ruleset. If you haven't installed it, the command to install `lint-staged` is as follows:

```
npm install --save-dev lint-staged
```

TIP To see a list of other options for using Prettier in a precommit hook, see this page: **https://prettier.io/docs/en/precommit.html**.

4.2.4 Alternative formatters

Prettier has become the industry standard, and although it was a bit controversial at first, it is mostly a nonissue now. Some alternatives do exist, though, and formatting guidelines can be solved by other tools:

- *ESLint*—ESLint has quite a few rules available related to formatting, and even common standard ESLint setups come with several of these rules enabled. If you use both ESLint and Prettier, make sure that they don't have competing rules!
  - *ESLint formatting rules*—**https://eslint.org/docs/user-guide/formatters**
- *StyleLint*—This tool also lints your project, but only for style-related problems, so it is basically a formatter. You can even run StyleLint with your Prettier ruleset, if that's what you like, using the aptly named package `stylelint-prettier`.
  - *StyleLint*—**https://stylelint.io**
  - *StyleLint using Prettier*—**https://github.com/prettier/stylelint-prettier**
- *EditorConfig*—EditorConfig is a basic tool for formatting rules that almost all editors support out of the box. It's not really a tool but a configuration file that you put in the root of your project; it dictates some simple rules for your files to obey, such as whether you're using tabs or spaces. The tool has limited language support, but you often see it used in projects anyway (in combination with other tools), as it gives you a good baseline set of formatting rules even without installing all the extra packages that other tools require.

- *EditorConfig*—https://editorconfig.org

4.3 Making components more robust with property constraints

As your application grows more complex, you may not know exactly how a given component works when you need to use it. Suppose that you know you have an `Input` component in your codebase, but you didn't create it and haven't used it before. How would you go about figuring out which properties to pass and which values to assign to them?

Checking the entire code inside a given component can be time consuming, so you might read the component definition. Let's say the `Input` component is defined as follows:

```
function Input({ name, label, value, change }) { ... }
```

The first three properties—`name`, `label`, and `value`—seem to be straightforward. They're most likely strings that you need to pass in. But what about that last one, `change`? Is that a change callback function or maybe a Boolean that indicates whether the value can be changed? Although you could have hoped for better naming that would have made the answer more obvious (such as calling it either `onChange` or `readonly`), you could also guess or use your intuition.

Another example is a date-display component that takes a `date` property. Is this date supposed to be a `Date` object, a string with the date already formatted, or even a timestamp with milliseconds since a certain date as a number? It's hard to tell directly from the component definition unless we explicitly name the property `dateString`, `dateObject`, or `timestampNumber`. But adding type information as part of the variable name is a terrible practice, mostly because it's not enforced anywhere, so it might not even be true.

Here's a related problem: what about required versus optional properties? When you read a component definition, you can't tell which properties are required to make the component work and which are optional to add functionality. Let's go back to our `Input` component, which might

take properties for specifying whether the input is read-only or has a maximum or minimum value. These properties would most likely be optional, but can you be sure?

Finally, what happens when your components are passed the wrong properties, either in terms of wrong types or missing required properties? Most of the time, the components fail silently, but sometimes, you get JavaScript errors because the component tries to invoke a method on a property that isn't set. Wouldn't it be nice if you could be warned if any component anywhere in your application received a wrong property?

Those examples are a lot of different problems. Would you believe that all of them have a single solution? They do, and it's built into React, though you need an additional library to make it work. The answer is . `prop-Types` , a configuration object appended to any component that tells React which property types it should expect your component to have. Figure 4.5 highlights the responsibilities of `PropTypes` , with only the relevant parts in focus. Let's dive in!

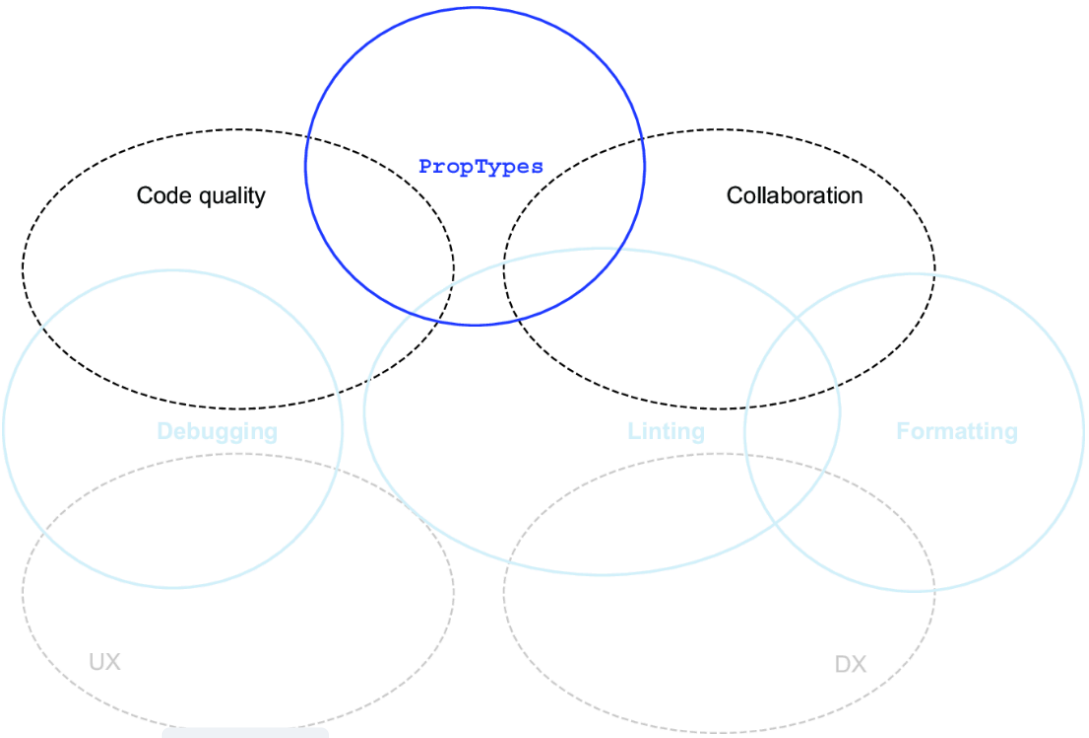

Figure 4.5 `PropTypes` affects two aspects of software development: collaboration and code quality.

NOTE `.propTypes` is no longer supported as of React 19 and will be ignored. If you do not need to use property types, feel free to skip this section and go straight to section 4.4. If you are working on a codebase that

uses React 19 or want to make sure you're prepared for anything, however, reading this section might be helpful.

4.3.1 How to apply property types

Going back to our fictional `Input` component from earlier in this chapter, let's see how we can apply property types to it correctly. The component takes four properties; the first three are strings, and the last one is a function. Following is the full component, including the property types:

```
import PropTypes from 'prop-types';  #1
function Input({ name, label, value, onChange }) {    #2
  return (
    <label>
      {label}
      <input name={name} value={value} onChange={onChange} />
    </label>
  );
}
Input.propTypes = {    #3
  name: PropTypes.string,     #4
  label: PropTypes.string,   #4
  value: PropTypes.string,  #4
  onChange: PropTypes.func,    #5
};
export default Input;
```

#1 First, we import the PropTypes library.

#2 The component takes four properties and is otherwise defined as usual.

#3 After the component definition, we add a property to the component function: propTypes, which is an object with a key named the same as each property that the component takes.

#4 Three of the keys in the propTypes object are set to the value PropTypes.string, indicating that the relevant property should be a string.

#5 The last key in the propTypes object is set to the value PropTypes.func, which indicates that the onChange property is expected to be a function. Note the abbreviated spelling of function.

As you can see, we add a property to the component after the definition. This value is set directly on the component definition function. Defining values on a function this way may look a bit weird, but it works well for this purpose.

The keys of the property-types object are the names of all the properties that the component takes, and the values are variables taken from the `PropTypes` library. This library includes a ton of values that you can use to define all sorts of properties with everything from simple types (such as strings and numbers) to complex and nested structures (such as lists of a certain type or objects with specific attributes).

GENERAL TYPES

The *PropTypes* library, of course, has support for all the simple types that we know and use every day: Booleans as `PropTypes.bool`, strings as `PropTypes.string`, and numbers as `PropTypes.number`. `PropTypes` also have support for symbols using `PropTypes.symbol`. These symbols are a special type of JavaScript variable that we don't use much in React. Then, of course, you can use functions (as `PropTypes.func`), arrays (as `PropTypes.array`), and objects (as `PropTypes.object`).

Note the special spelling of a few of these functions. They're called `PropTypes.bool` and `PropTypes.func`, not `.boolean` and `.function`.

SPECIFIC TYPES

But what if you have a component that takes an object of a specific type? Suppose that you have a component that displays a user with their name and age. The component expects a user object with a `name` property (which should be a string) and an `age` property (which should be a number). That object is called a *shape* in `PropTypes` lingo, and you define it like this:

```
function UserDisplay({ user }) {    #1
...
}
UserDisplay.propTypes = {
  user: PropTypes.shape({   #2
    name: PropTypes.string,    #3
    age: PropTypes.number,    #3
  }),
};
```

#1 This component takes a single property, but we need it to be structured in a specific way.
#2 We define the property as a shape, which is further defined by another property-types object.
#3 The shape consists of two properties, a string and a number, respectively.

This example can be nested, so you could have a shape with a property that was another shape. You can also specify that an array has to be of a certain type. So if you have a component that takes an array of numbers, you would use the value `PropTypes.arrayOf(PropTypes.number)`. You could combine this array notation with the previous shape to create a component that shows a list of users with full property validation performed by React:

```
function Users({ userList }) {
...
}
Users.propTypes = {
  userList: PropTypes.arrayOf(
    PropTypes.shape({
      name: PropTypes.string,
      age: PropTypes.age,
    }),
  ),
};
```

As you can see, that code quickly becomes quite complex to maintain. (What if users are changed so they have both a first name and a last name?) You can store your types in a shared file somewhere, so your components could look like this:

```
// types.js
export const UserType = PropTypes.shape({     #1
  name: PropTypes.string,
  age: PropTypes.age,
});
// UserDisplay.js
import { UserType } from './types';     #2
function UserDisplay({ user }) {
...
}
UserDisplay.propTypes = {
  user: UserType,    #3
};
// Users.js
import { UserType } from './types';    #2
function Users({ userList }) {
...
}
Users.propTypes = {
  userList: PropTypes.arrayOf(UserType),    #3
};
```

#1 We can define the type as a regular variable in a central file.

#2 Then we import the type as any other variable . . .

#3 . . . and use the type where appropriate.

This example only scratches the surface of what the `PropTypes` library can do. You can even create custom validators if you have some special components. Check the documentation for the `PropTypes` library at **https://mng.bz/v8Mp**.

REQUIRED PROPERTIES

One final aspect of the `PropTypes` library that I want to introduce here is the notion of required versus optional properties, which often go hand in hand with default property values. Suppose that we have a component that can display data for a car. Some cars have sunroofs, but most cars do not, so we can set the default value for a sunroof to `false`; then we need to set the property only for cars that have a sunroof. All cars have a make and model, however, so those properties do not have default values and are required for all instances of the car component.

We can specify this relationship by appending the suffix `.isRequired` to the required property types in the `propTypes` object and leave it out for the optional ones. We can do that like so:

```
function Car({ make, model, hasSunroof = false }) {     #1
...
}
Car.propTypes = {
  make: PropTypes.string.isRequired,     #2
  model: PropTypes.string.isRequired,    #2
  hasSunroof: PropTypes.bool,    #3
};
```

#1 The car component takes three properties; the last one has a default value.
#2 The two required property types have the suffix .isRequired.
#3 The last optional property type does not have this suffix.

Any type of property can be made required, even nested or complex property types. Note that there's no requirement that optional properties have defaults and required properties don't. Those details are up to you to track.

4.3.2 Drawbacks of using property types

You may have a few problems with using property types; the main one is the fact that validation happens only at run time. You have no error messages as you type the code—only when you run the code. This situation delays valuable feedback, which, in turn, slows your development efficiency.

Another problem with property types is that you have type safety only for components, not all the other things in your codebase. Those other things can be hooks, functions, external libraries, and much more. Therefore, the solution is only partial.

Compare both of these drawbacks with TypeScript, which we get to in chapter 5. TypeScript offers in-editor type safety for your entire application, which is why it is a more widely adopted solution today than merely using property types.

Finally, and most important, property types are not supported in React 19. Using property types does not cause any errors, but the `.propTypes` property will be ignored, and no validation will happen at run time. If you are using property types today and considering upgrading to React 19, you might as well switch to TypeScript, if you haven't already, to get at least the same level of type safety.

4.3.3 Default property values

A concept related to `.propTypes` was once fairly common, and you may still see it in live codebases today. This concept concerns default values for properties. This specific method is mostly irrelevant today, but it may be good to know about it in case you come across it in your work or are working on a codebase with class-based components. If neither situation applies to you, feel free to skip this section and go straight to section 4.4.

You may be thinking, "We already use default values for properties; we can specify them directly in the component definition. So what the heck are you going on about?" First, you don't need to get so aggressive! Second, yes, that is true today, but it wasn't back in the days of class-based components.

For a class-based component, there was (and still is) no elegant way to set a default value for a property, so the React team created one for you: the `.defaultProps` object, which you can apply to a component the same way that you assign `.propTypes` to a component. Look at this component, which has a link with a default target:

```
function MenuItem({ label, href, target="_self" }) {    #1
  return (
    <li>
      <a href={href} title={label} target={target}>
        {label}
      </a>
    </li>
  );
}
MenuItem.propTypes = {    #2
  label: PropTypes.string.isRequired,
  href: PropTypes.string.isRequired,
  target: PropTypes.string,    #3
};
```

#1 The property target has a default value: the string "_self".

#2 Note how we added propTypes.

#3 We don't specify the target property as a required value because we have a default value in case it is not specified.

But if we were to create this same component as a class-based component, we'd have a different way to add this default property value:

```
class MenuItem extends React.Component {
  render() {
    return (
      <li>
        <a
          href={this.props.href}
          title={this.props.label}
          target={this.props.target}     #1
        >
          {this.props.label}
        </a>
      </li>
    );
  }
}
MenuItem.propTypes = {
  label: PropTypes.string.isRequired,
  href: PropTypes.string.isRequired,
  target: PropTypes.string,     #2
};
MenuItem.defaultProps = {    #3
  target: "_self",
};
```

#1 Remember that properties in class-based components are never defined anywhere, such as when we destructure them in the functional component definition. We use them when necessary directly from the this.props object.
#2 We still specify the target property as not required because we want a default value for it.
#3 The new thing here is adding an object named defaultProps to the component variable. That object contains default values for properties where relevant. Note that we specify only the properties that have default values; required properties never do.

Now these two variants of the `MenuItem` component are identical. You can specify the `target` property or leave it out and have the component use a default value.

You probably already know how you can add default property values to a functional component by using the destructuring syntax, and you've seen how you can add default property values to any component by using the `.defaultProps` object. You may wonder what would happen if you did

both things. This question is why we can't have nice things; you want to burn the whole thing down!
Well, don't worry; nothing bad happens. There's no need to do both things, which would only make your component difficult to use. If you did, the component would still work. But the defaults specified in the functional component definition will be ignored if the same property names are specified in the `.defaultProps` object.

Consider a concrete example. We want to create a greeting component that says "hi" to the user by name. If the user hasn't provided a name, we want to display a default name, `"Stranger"`. Due to a slight mixup with multiple developers working on this component over time, a default property value was specified in two ways, both as a destructuring default and as part of a `.defaultProps` object:

```
function Greeting({ name="Stranger" }) {    #1
  return <h1>Hello {name}</h1>;
}
Greeting.propTypes = {
  name: PropTypes.string,
};
Greeting.defaultProps = {
  name: "Anonymous",     #2
};
```

#1 We set a default value for the name property to "Stranger" in the component definition.
#2 We also set the default value for the name property to "Anonymous" by using defaultProps.

What happens then? As I just mentioned, the value listed in `.defaultProps` takes precedence. In this instance, if you use this component and don't specify a value for the name property, the greeting would become `<h1>Hello Anonymous</h1>`.

But don't do this. Using `.defaultProps` is technically possible for functional components but not recommended. It might even become a warning in future versions of React. For now, this practice is valid but not recommended. If you have a codebase with only functional components, there's no need to use `.defaultProps`, so stick with destructured default values.

As of React 19, in fact, `.defaultProps` is no longer supported for functional components and will be ignored. You can still use the `.defaultProps` object for class components, as no reasonable alternative is available.

4.3.4 How to get started using property types

To begin using property types, you need to do only a limited amount of setup: install the `prop-types` packages. Use `npm` with the following command:

```
npm install --save prop-types
```

Using default property values with the `.defaultProps` object requires no setup. This functionality is entirely built into React. If you want to make sure that you remember to use property types throughout your project, you can apply two useful ESLint rules:

- `react/prop-types` throws an error by default if a component uses properties that are not defined in the `.propTypes` object or if the `.propTypes` object is missing.
- `react/no-unused-prop-types` throws an error by default if you have a `.propTypes` object that defines properties, which aren't used in your component.

4.4 Debugging applications with React Developer Tools

Despite your using all of the preceding tools, and maybe even more, bugs will creep into your code—a sad fact of software development. Over time, bugs happen. They're unavoidable and not a direct reflection of your skills as a developer, only an unwanted consequence of lines of code plus time.

Your ability to find the source of a bug and fix it efficiently, however, *is* a reflection of your experience and ability as a developer. Luckily, React has some good tools that allow you to quickly identify root causes of com-

mon (and uncommon) bugs. Figure 4.6 highlights the responsibilities of debugging, with only the relevant parts in focus.

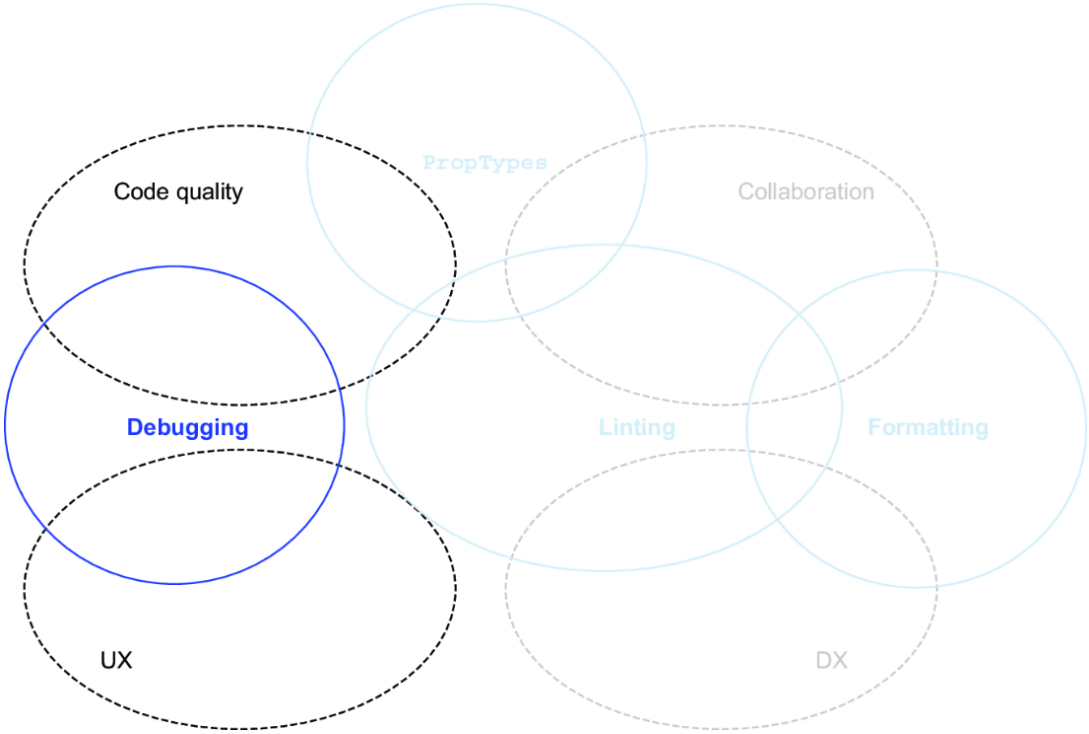

Figure 4.6 Debugging affects two aspects of software development: code quality and UX.

Our two main weapons in React are the developer tools built into our browser and the React Developer Tools. Although I won't cover your browser's developer tools (please check your browser documentation for information), I will go over React Developer Tools, a browser extension available directly for the Google Chrome, Mozilla Firefox, and Microsoft Edge browsers and indirectly for all other browsers.

4.4.1 Problems solved by debugging

Most of the problems we've dealt with so far in this chapter are related to static code analysis. Linters and formatters operate on source code only and have no idea about what happens when the code is running.

Debugging is the opposite situation. You operate a debugger in the live browser environment where your application code is running. A debugger can inspect the values of variables as they change over time, pause script execution, monitor when certain functions are running or lines of code are executed, and even hotswap new code into the running application.

The React Developer Tools plugin consists of two separate tools: the components inspector and the profiler. These tools work differently and are used for different purposes, but combined, they give you all the insights you require. I'll focus more on the components inspector because it's more useful and you'll reach for it more often. The profiler is a more specialized tool for a smaller range of problems. Using these tools, you can do the following:

- See why certain application states are reached by tracking down the exact values of variables over time.
- Identify the properties with which a certain component is instantiated by using the components inspector.
- Use the profiler to find which components are rendering too frequently or too slowly.
- Visualize which components are rendering and track down why by using the rendering highlighter in the components inspector.
- Inspect the full component tree by using the components inspector.

These tools will equip you to solve a lot of problems in your React application. To unsuspecting team members, you might even seem like a debugging wizard wielding these tools, which are immensely powerful when you fully understand them.

4.4.2 How to get started using React Developer Tools

React Developer Tools is a browser plugin for the most common browsers. But the tool also works as an application running separately from your browser as long as you make sure to connect your React application to an extra web server. You don't want to take this approach if you can avoid it, though, so go for the plugin if it exists for your browser.

INSTALLATION IN CHROME, FIREFOX, AND EDGE

For Chrome, Firefox, and Edge, React Developer Tools can be installed as a plugin directly from the browser's plugin store:

- *Chrome* —**https://mng.bz/4J7R**
- *Firefox* —**https://mng.bz/QZ56**
- *Edge* —**https://mng.bz/X1X9**

After you installed the plugin, you're basically done. Open any local React application running in development mode (the default when you're running it locally), and you can start using the plugin.

INSTALLATION IN OTHER BROWSERS

For other browsers, including Apple's Safari, you have to do three things to get the plugin running:

1. Install React Developer Tools via npm as a global package.
2. Launch React Developer Tools as a separate application via a command in your terminal.
3. Manually connect your web page with your development version of a React application to the React Developer Tools application.

First, you need to install the application:

```
$ npm install -g react-devtools
```

Notice the `-g` flag, which installs the package as global rather than local for a single specific application. Next, you need to launch the tool with this simple command:

```
$ react-devtools
```

This command launches an application that looks like an instance of developer tools inside a browser but is an independent application. This application won't be connected to anything until you complete step 3 (connect your HTML page with a local React application to this tool).

The standalone tool will run a web server on port 8097, and we simply need to load a script from there; that script takes care of everything else. So add the following HTML snippet to the `<head>` section of your application (which for most applications in this book requires updating the file in `<root>/index.html`):

```
<script src="http://localhost:8097"></script>
```

Include this snippet as the first thing inside `<head>`, and you should be good to go. When you reload your React application in your browser, you should see the React Developer Tools application connect to your React application and allow you to debug it.

Note that this approach has some drawbacks, including the fact that you need to have an extra application running. Also, and slightly more troublesome, you can have only one instance of React Developer Tools running, so you can debug only one React application at a time.

4.4.3 Using the components inspector in React Developer Tools

The first tool in the React Developer Tools plugin is the easiest to understand, use, and benefit from: the components inspector. This tool is incredibly powerful, and I won't cover everything it does, but I'll go over the most important bits. It allows you to

- Inspect the entire component tree (with or without HTML nodes)
- Easily and quickly see which nodes are memoized
- Inspect and manipulate component props
- Inspect and manipulate some stateful hooks and their values

INSPECTING THE COMPONENT TREE

Let's see how we can use the components inspector tool to inspect the component tree in various ways. For this example, let's reuse a simple application from chapter 2 with a React context, memoization, and some hooks so we can see how they show up in the inspector. In particular, let's look at listing 4.1. This code is the first iteration of the dark mode application (using native React context only), with a slight change to introduce an extra-simple component, `Title`. Figure 4.7 shows the slightly modified application running.

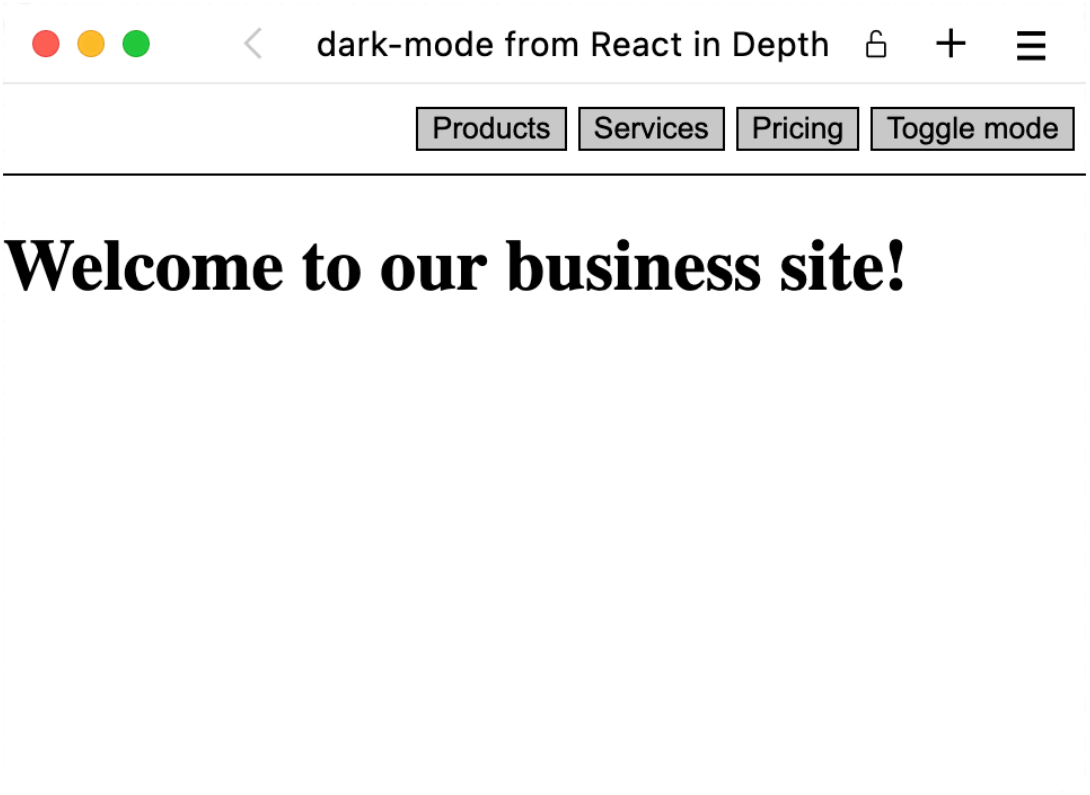

Figure 4.7 The application looks the way it did before, with a large "Welcome" headline and a dark mode toggle button in the top-right corner (which also still works as expected).

Listing 4.1 Dark mode application with React context

```
import { useContext, useState, createContext, memo } from "react";
const DarkModeContext = createContext({});
function Button({ children, ...rest }) {
  const { isDarkMode } = useContext(DarkModeContext);
  const style = {
    backgroundColor: isDarkMode ? "#333" : "#CCC",
    border: "1px solid",
    color: "inherit",
  };
  return (
    <button style={style} {...rest}>
      {children}
    </button>
  );
}
function ToggleButton() {
  const { toggleDarkMode } = useContext(DarkModeContext);
  return <Button onClick={toggleDarkMode}>Toggle mode</Button>;
}
const Header = memo(function Header() {
  const style = {
    padding: "10px 5px",
    borderBottom: "1px solid",
    marginBottom: "10px",
    display: "flex",
    gap: "5px",
    justifyContent: "flex-end",
  };
  return (
    <header style={style}>
      <Button>Products</Button>
      <Button>Services</Button>
      <Button>Pricing</Button>
      <ToggleButton />
    </header>
  );
});
const Title = memo(function Title({ isPrimary = false, children }) {     #1
  const Heading = isPrimary ? "h1" : "h2";
  return <Heading>{children}</Heading>;
});
const Main = memo(function Main() {
```

```
    const { isDarkMode } = useContext(DarkModeContext);
    const style = {
      color: isDarkMode ? "white" : "black",
      backgroundColor: isDarkMode ? "black" : "white",
      margin: "-8px",
      minHeight: "100vh",
      boxSizing: "border-box",
    };
    return (
      <main style={style}>
        <Header />
        <Title isPrimary>Welcome to our business site!</Title>    #2
      </main>
    );
  });
  function App() {
    const [isDarkMode, setDarkMode] = useState(false);
    const toggleDarkMode = () => setDarkMode((v) => !v);
    const contextValue = { isDarkMode, toggleDarkMode };
    return (
      <DarkModeContext.Provider value={contextValue}>
        <Main />
      </DarkModeContext.Provider>
    );
  }
  export default App;
```

#1 The new thing in this application compared with the same example in chapter 2 is this simple component. It's used only once but takes a Boolean property for the purpose of the example.
#2 Uses the Title component and sets the Boolean flag to true to render it as a primary headline

EXAMPLE: DARK-MODE

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

```
$ npm run dev -w ch04/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/ch04-dark-mode**.

Next, take a look at figure 4.8 to see React Developer Tools open in the components inspector.

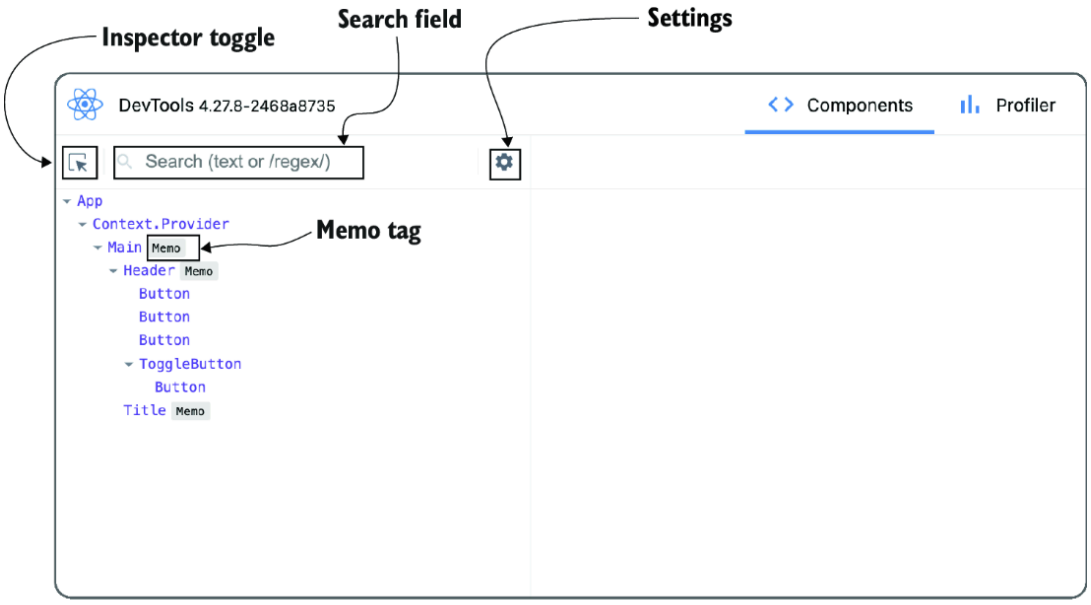

Figure 4.8 The initial view in the components inspector as you connect to the dark mode application. You can see the entire component tree on the left and an empty inspector on the right. Notice that several extra tools are included: the inspector toggle, a search field, a settings gear, and a small tag highlighting memoized components.

The components inspector includes a few extra tools (annotated in figure 4.8) that you will find useful, including the inspector toggle, which allows you to select a component in your application visually like inspecting HTML nodes in the regular document object model (DOM) tree browser, and the search field at the top, which allows you to filter the tree to view the matching subset (including searching by regular expressions). These tools are straightforward to use, so I recommend that you play around with them to see how they aid you.

The settings gear, annotated in figure 4.7, allows you to toggle additional modes and options in the components inspector. You can show all regular HTML nodes in the components tree, which are hidden by default. Figure 4.9 shows how to toggle that option; figure 4.10 shows the resulting component tree.

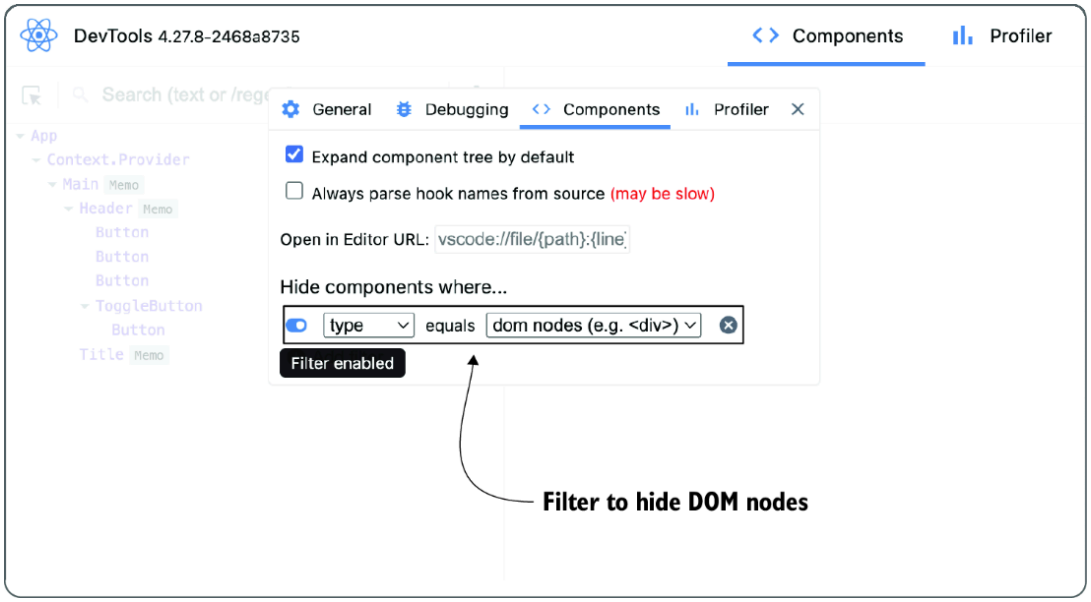

Figure 4.9 On the components tab of the components inspector's settings dialog box, you can disable the filter that hides all HTML nodes by default. You can use this same dialog box to add filters that hide other types of components, such as all memoized components, all class components, all contexts, and other types of components.

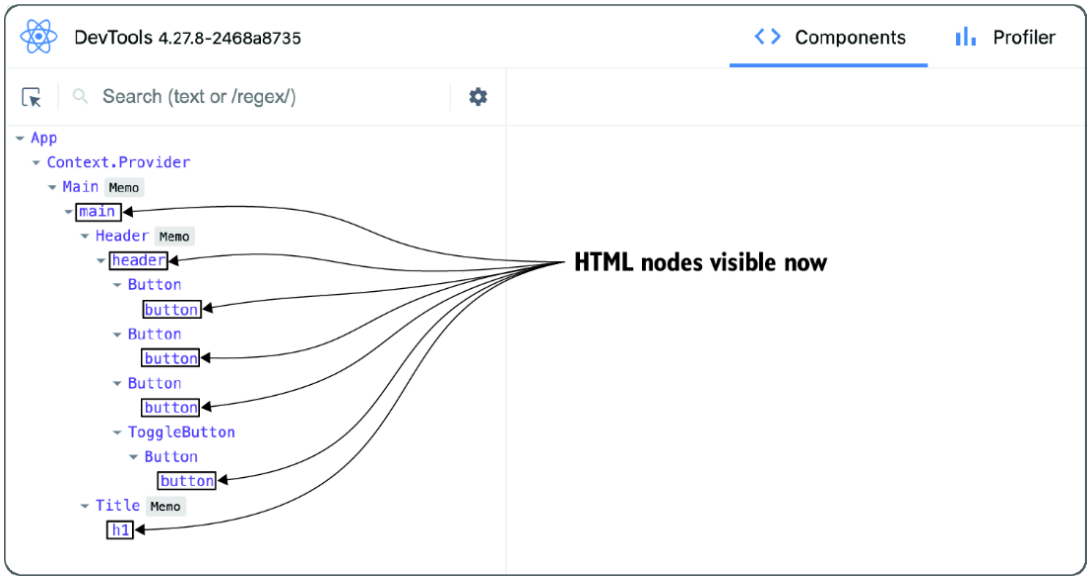

Figure 4.10 With the default HTML node filter disabled, the component tree is expanded to include all the HTML nodes.

INSPECTING COMPONENTS

When you click a component in the component tree on the left side of the inspector, you see all the properties of said component, as well as information about the hooks used in it. Figure 4.11 shows a simple component without hooks: the `Title` component.

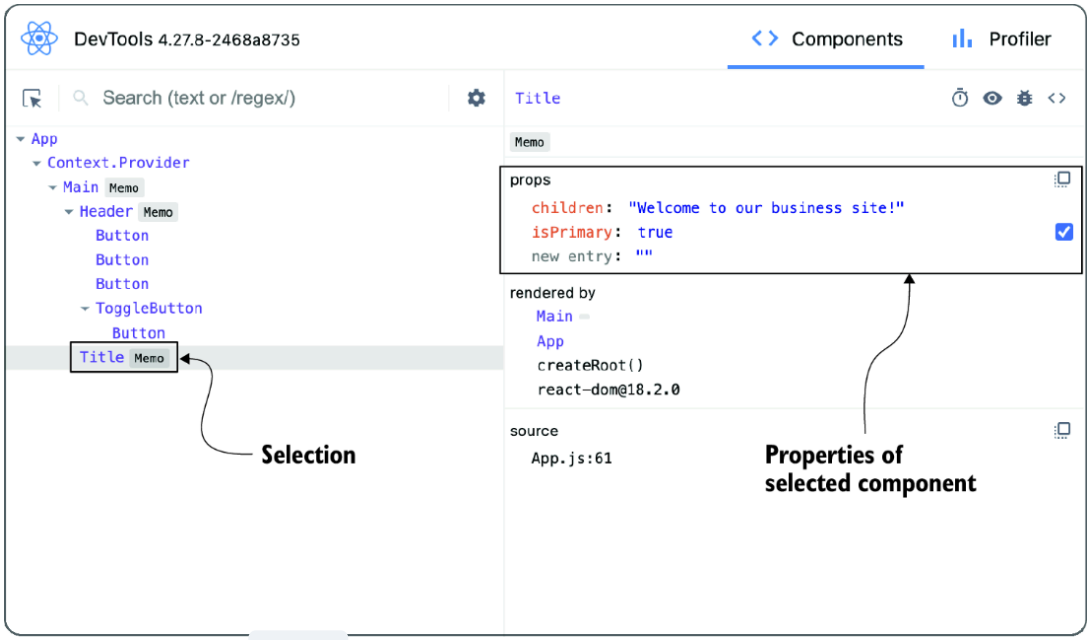

Figure 4.11 The `Title` component is selected in the component tree, and all the properties are displayed on the right side.

Besides viewing the properties, we can edit them directly and inline right in the components inspector. We can input strings and numbers directly in the input field when it's focused, and we can use the check box to toggle Booleans. Objects are a bit more tricky to manipulate directly but can be manipulated to some extent. We can't use this method to edit functions, however, because of scope and other concerns. Let's update the `Title` text and disable the primary headline Boolean flag, as shown in figure 4.12.

Figure 4.12 We can update strings directly and toggle Booleans.

Manipulating components this way causes the relevant components to re-render instantly and update the view. If the parent component ever re-renders, of course, our changes would be overridden. We could trigger that activity in this case by toggling dark mode. Changing the values directly in the inspector is a temporary local change, not a change of the application. The resulting application looks like figure 4.13 until the parent component re-renders and resets the properties to their "real" values.

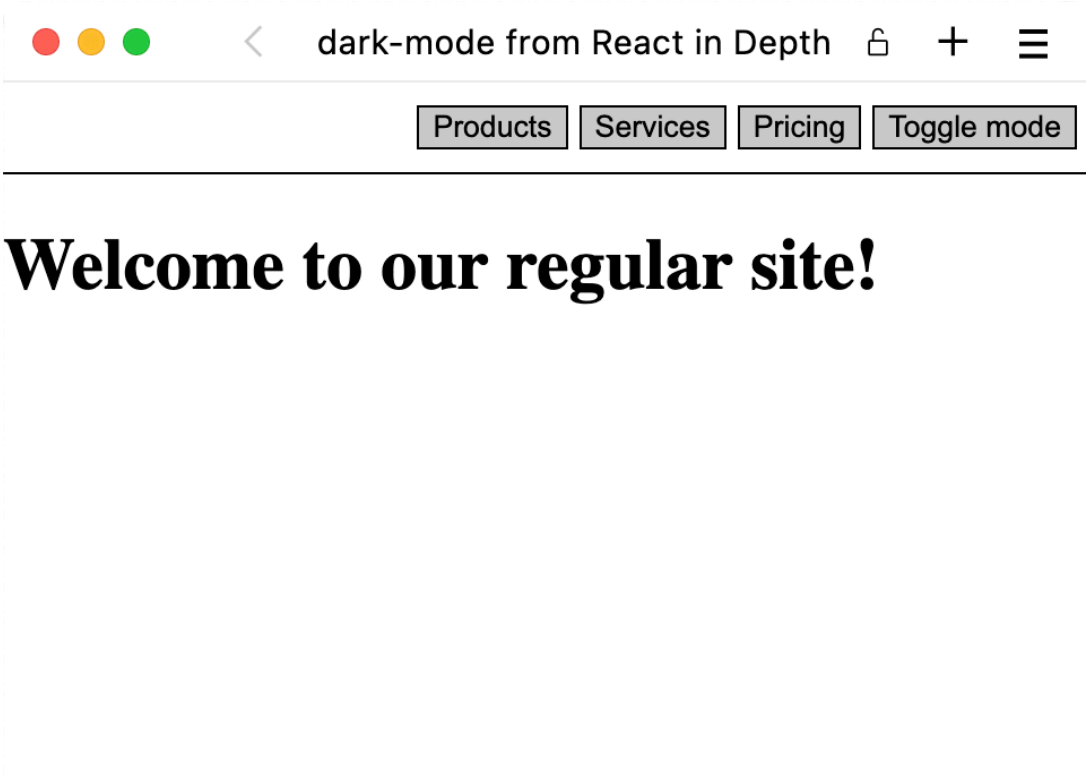

Figure 4.13 Manipulating component properties in the components inspector causes the specific components to re-render immediately.

INSPECTING STATEFUL COMPONENTS

The root `App` component in the dark mode application is stateful, using the `useState` hook. Let's see how we can interact directly with that state without going through the regular setter function. Using the components inspector, we can inspect a stateful component and see the state directly, as well as manipulate it. If we manipulate the state, the component will re-render with the new state as though it were updated correctly (figure 4.14).

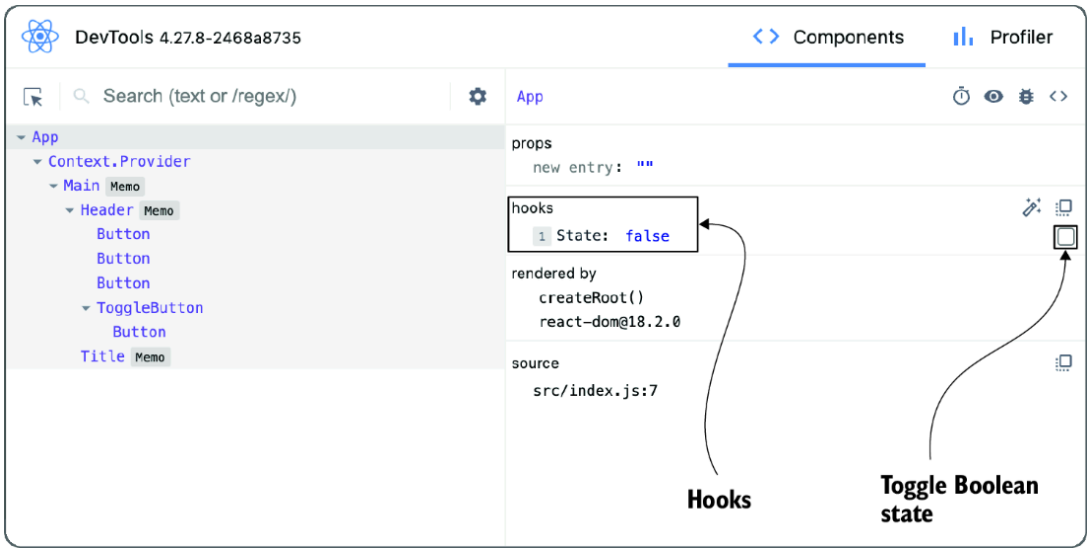

Figure 4.14 When you're inspecting a stateful component with a `useState` or `useReducer` hook, you can interact with the state directly through the components inspector. You can manipulate the value, just as you can manipulate properties, as shown in figure 4.12.

If you consume a context, your component also becomes stateful, and you can use the components inspector to inspect the context state. But you cannot update the con-text state directly by using the inspector from a context consumer; you have to go through the context provider and manipulate the value property from there instead (figure 4.15).

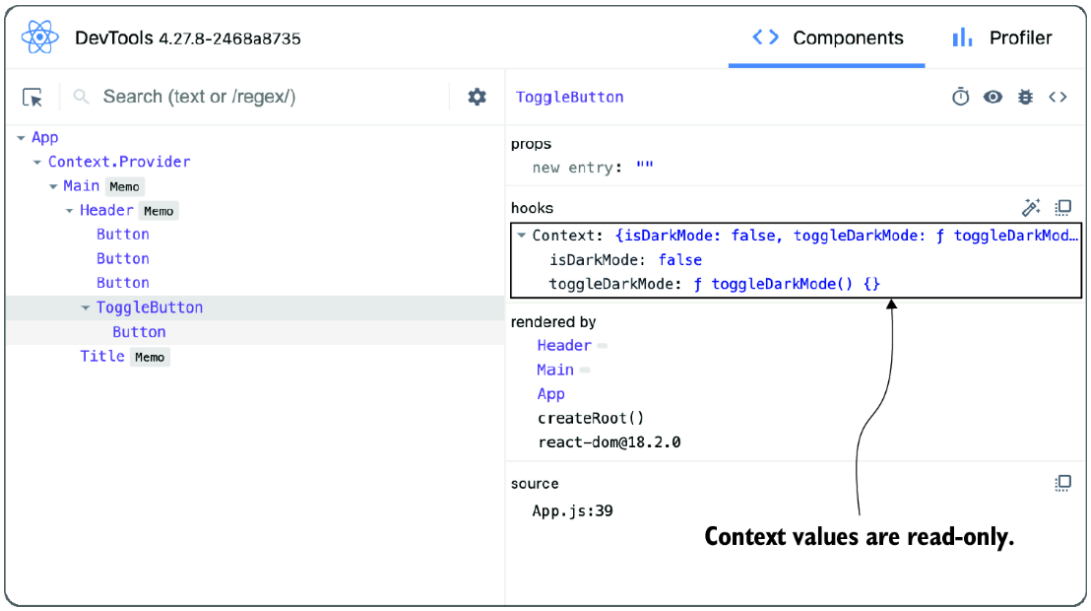

Figure 4.15 You can see but not modify the value of a context when you're inspecting a context consumer.

If instead you highlight the nearest context provider inside the components inspector's tree view, you can manipulate the value property as shown in figures 4.11 and 4.12.

4.4.4 Using the profiler in React Developer Tools

The profiler tool is the lesser-used tool in the React Developer Tools utility belt. This tool is a lot more complex and is used for specific purposes. It's comparable to the built-in performance as provided by your browser's regular developer tools.

Suppose that you have an application that is acting slow. When you click a specific button, the application freezes for about half a second before the click is registered and the resulting update happens. Why? That situation is where the profiler comes in.

In the profiler tool, you can start a recording before you click the button, keep the recording going while the button action takes place, and stop the recording afterward. Then you can look through the recording to see what happens: which functions are invoked, which components re-render and why, if tracking re-rendering is enabled, which effects run when, which events are invoked on what components, and many other details. The interface is complex, with a lot of information that can be hard to understand; the only good way to learn it is through experience.

Try to debug the dark mode application from listing 4.1. Start a recording in the profiler, toggle the dark mode button, and then stop the recording. The result should look something like figure 4.16.

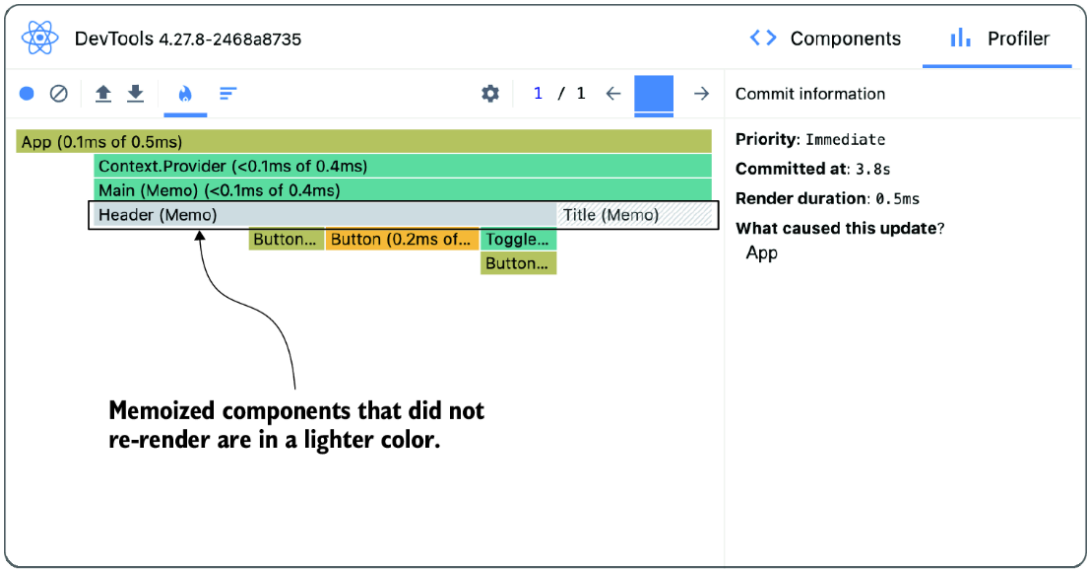

Figure 4.16 The result of a React profiler recording session, displaying which components re-render, when, and for how long, as well as how the renders cascade through the component tree. The lightly shaded components (`Header` and `Title`) are so colored because they didn't render. (They're memoized and had no changes.) All the other components displayed in the view did re-render for some reason.

If you enable the setting to track why components re-rendered in the profiler settings menu (shown in figure 4.17), inspecting the button inside the toggle button allows you to see stats for that component (figure 4.18).

The profiler recording session, shown in figure 4.18, displays a particular component (in this case, `Button`).

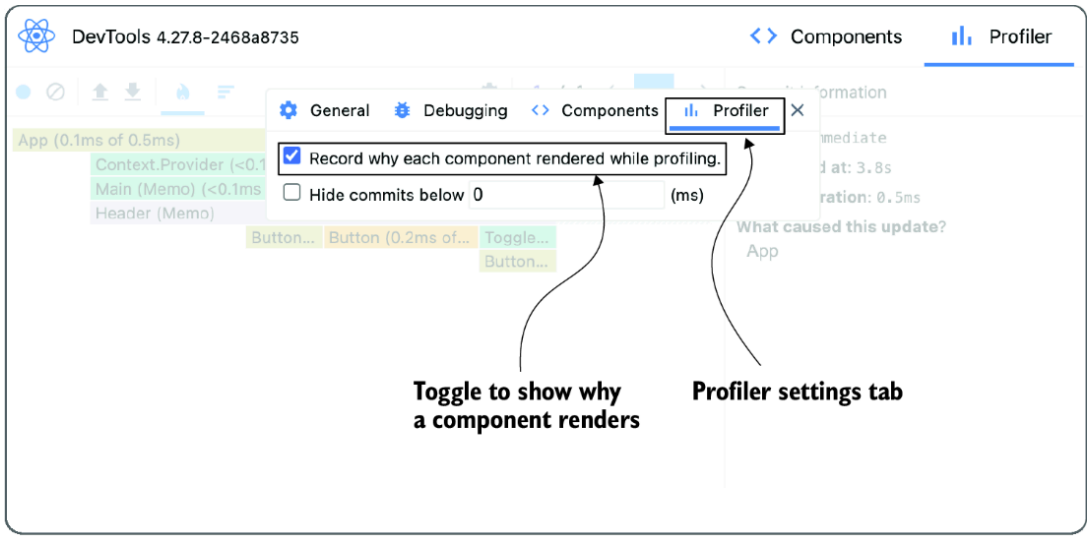

Figure 4.17 You can enable the setting to track why components re-render in the settings for the profiler tool.

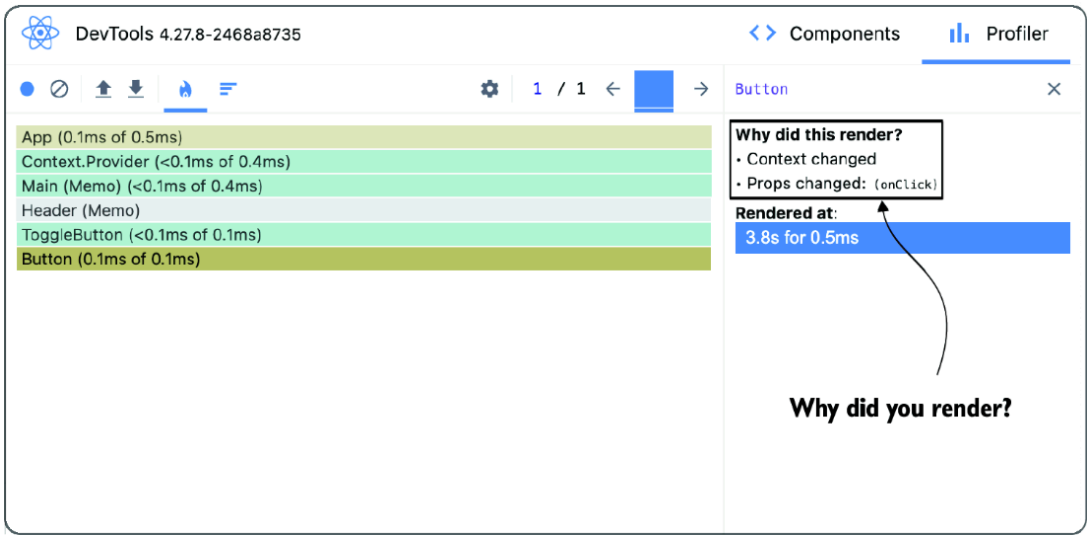

Figure 4.18 When we inspect this component, we see that it is rendered for two reasons: it consumes a context that has updated, and its properties have been updated. Either event would be sufficient to cause a re-render, but in this case, both happened. We can also see that the component re-rendered for a whopping 0.5 milliseconds, which seems to be the smallest unit of time that's trackable in the profiler.

As mentioned, the profiler is a complicated debugging tool that requires a bit of experience to fully understand, so play around with it if you want to master it. Using this tool in small applications that you understand can be a good way to go about learning.

4.4.5 Alternatives and other tools

Depending on your stack and the other technologies in use, a few other libraries may be relevant for you. Here's a brief list of libraries and tools that might apply to your application:

- If you're using Redux or any of the derived libraries (such as Redux Toolkit or RTK-Query), Redux DevTools is a must-have for debugging. You can find it at **https://github.com/reduxjs/redux-devtools**.
- If you're using React Native, the React Native Debugger is a great bridge application that allows you to use tools such as React Debugging Tools even while running React Native applications. You can get it at **https://github.com/jhen0409/react-native-debugger**.
- If you're using Electron to build React apps as desktop applications, Reactotron is the library for you. It works as a bridge between your Electron-wrapped application and the regular developer tools, like the

React Native Debugger. Get it at **https://github.com/infinitered/reactotron**.

- If you need to re-create and debug errors experienced by other people, consider using Replay.io. This tool is a freemium service, but at the free level, you're able to debug only locally. If you subscribe at higher pricing tiers, you can re-create and debug error sessions experienced by other users. Even at the free tier, it's an awesome debugging tool inspired by Redux DevTools, but it works regardless of your state management library. Read more at **https://www.replay.io**.

If you're using other complex technologies in your stack, they may also have some dedicated debugging tools, so remember to search for them. Good luck debugging!

Summary

- Although React and JavaScript, in general, are great on their own, both can become significantly more pleasant to work with if you introduce some developer tooling to assist with managing complexity.
- Linting is great for enforcing style guides and reducing errors. ESLint is among the most popular linters available.
- Formatters reduce discussion and time spent on stylistic debates and fully automate the process of formatting the code as agreed on. Prettier is the tool most commonly used for formatting.
- React has a built-in system for validating property types passed to all the components in the codebase, but it is an older system that has significant drawbacks compared with TypeScript.
- The React Developer Tools plugin includes a powerful components inspector tool that allows users to inspect and manipulate component props, memoized nodes, and stateful hooks. The inspector provides features such as searching and filtering components, toggling modes and options, and highlighting memoized components.
- The profiler tool, although more complex, helps with understanding application performance by recording and analyzing function invocations, component re-renders, effects, events, and more. You need experience to use it and fully understand its capabilities.
- These four utilities have overlapping responsibilities, but you might see several or all of them applied to projects in the wild. You'll even see quite a few later in this book.