TYPESCRIPT · WEB APPS PREVIEW

React from Zero

Build Your First Real Web Apps with TypeScript, One Small Step at a Time



Gérald Barré

MICROSOFT MVP · MEZIANTOU.NET

Copyright © 2026 Gérald Barré

All rights reserved. No part of this publication may be reproduced, distributed, transmitted, stored in a retrieval system, or otherwise used in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the copyright holder, except as permitted by applicable copyright law.

Published by Gérald Barré
Montréal, Québec, Canada

First edition, 2026

Legal deposit, Bibliothèque et Archives nationales du Québec, 2026
Legal deposit, Library and Archives Canada, 2026

The information in this book is provided on an "as is" basis, without warranties of any kind, express or implied. Although the author has taken reasonable care in preparing this work, the author makes no representations or warranties regarding the accuracy, completeness, or suitability of its contents. To the fullest extent permitted by applicable law, the author shall not be liable for any loss or damage arising directly or indirectly from the use of, or reliance on, the information contained in this book.

Product names, company names, logos, brands, and other trademarks mentioned in this book are the property of their respective owners. Their use does not imply affiliation with, sponsorship by, or endorsement by the respective trademark owners.

Unless otherwise indicated, source-code examples included in this book are provided for educational purposes and may be used and adapted in your own projects. The source code is provided without warranty of any kind.

Contents

A free preview — the introduction and 4 chapters. The full edition has 25 chapters and 6 appendices.

Introduction in this preview
- What you will build
- Who this book is for
- Why TypeScript from the first page
- Written the way professionals write it
- How the code in this book works
- Versions
- How to read this book
- Using the samples
- Asking for help

Part I — Getting Ready

Chapter 1 — What Is React, and Why Should You Care? in this preview
- The problem: keeping the page in sync
- The big idea: describe the page, let React update it
- Components: the building blocks
- Why this book uses TypeScript
- What React is not
- A tour of the finished app
- The tools you'll use
- Exercises

Chapter 2 — Setting Up Your Workshop
- Node.js and npm
- A terminal survival kit
- A code editor: VS Code
- The browser and its developer tools
- Creating the project
- A tour of the project
- Hot reload
- The companion archive
- Making the project yours
- The book's strict setup
- Why did that happen? Two errors you'll meet
- Saving your work with Git
- Exercises

Chapter 3 — Just Enough Modern JavaScript
- Running the examples
- `const`, `let`, and template literals
- Arrow functions
- Objects and arrays
- `map`, `filter`, and `find`
- Immutability: copy, don't change
- Conditions in expressions: `? :`, `&&`, `??`, and `?.`
- Modules: `import` and `export`
- Promises and `async`/`await`
- Exercises

Chapter 4 — Just Enough TypeScript in this preview
- What a type is
- Inference: TypeScript already knows
- Writing types yourself
- Describing objects with `type`
- Union and literal types
- `null` and `undefined`
- Why `books[0]` might be `undefined`
- `import type` for types
- Union types instead of `enum`
- Reading a type error
- Gotcha: why the book never uses `any`
- Exercises

Part II — Thinking in Components

Chapter 5 — JSX: HTML's Clever Cousin
- What JSX is
- The rules of JSX
- Curly braces: TypeScript inside JSX
- Semantic HTML in JSX
- The accessibility lint rules
- Exercises

Chapter 6 — Your First Components in this preview
- What a component is

Why the capital letter matters
Components that use other components
Splitting a page into components
Components must be pure
Never define a component inside another component
The component tree in React Developer Tools
Shelfie: your first components
Exercises

Chapter 7 — Props: Passing Data Down
Props are arguments for components
Typing props
Passing different kinds of values
Children: components that wrap content
Props are read-only
Optional props: missing, or `undefined`?
Shelfie: every card gets its own book
Exercises

Chapter 8 — Rendering Lists and Conditional Content
From an array to a list
Keys: how React tells items apart
Showing things conditionally
The first book might not exist
Shelfie: the book list comes from data
Exercises

Chapter 9 — Styling Your Components
Plain CSS files, and why they reach everywhere
CSS Modules: class names that belong to one component
Conditional class names
Inline styles
Images and other static files
Good CSS habits
Other ways to style: Tailwind CSS and component libraries
Shelfie: from plain to polished
Exercises

Part III — Making Things Interactive

Chapter 10 — Events: Responding to the User
What an event is
Your first event handler
Inline handlers and passing arguments
The event object
Forms and `preventDefault`
Passing handlers down as props
Only interactive elements get click handlers
A button's name
Shelfie: a "Remove" button
Exercises

Chapter 11 — State: Giving Components a Memory in this preview
Why a normal variable isn't enough
`useState`: a value that React remembers
What happens when state changes
Each component has its own state
State is a snapshot
The Rules of Hooks
Why did that happen?
Toggle buttons and `aria-pressed`
Shelfie: a reading status on each card
Exercises

Chapter 12 — Updating Objects and Arrays in State
Changing state in place does nothing
Adding, removing, and updating items
Updating a nested object
Picking what to store in state
Resetting state with a `key`
`readonly`: a safety net
Shelfie: the list becomes state
Exercises

Chapter 13 — Forms
Controlled inputs
One state object, or one state per field?
Checking what the user typed
Submitting a form with an action
`useActionState`: results and pending state
Uncontrolled inputs
Accessible forms
Two layers of checking
File inputs
Shelfie: an "Add a book" form
Exercises

Chapter 14 — Sharing State Between Components
Two components that need the same data
Lifting state up
Controlled components and a single source of truth
Passing typed callbacks down
Derived values: compute them while rendering
Where state lives in Shelfie
Shelfie: search and filter the shelf
When props travel far
Exercises

Part IV — Talking to the Outside World

Chapter 15 — Effects: Stepping Outside React
What a side effect is
Your first effect: `useEffect`
Cleaning up
Why does my effect run twice?
List everything the effect uses
You might not need an effect
Shelfie: remembering the shelf
Exercises

Chapter 16 — Fetching Data from an API
APIs, JSON, and `fetch`
The book's mock API
Typing the response
Loading, error, and success
Fetching in an effect, carefully
The better way: TanStack Query
Sending a file to the server
Error boundaries
Where things are heading: `use` and Suspense
Shelfie: searching a catalog
Exercises

Chapter 17 — Refs: Reaching the DOM
What a ref is
Reaching a DOM element
A ref is not state
Remembering a value without rendering
Passing a ref to your own component
Dialogs with the `<dialog>` element
Shelfie: confirm, focus, and a shortcut
Exercises

Part V — Growing Your App

Chapter 18 — Custom Hooks: Reusing Logic
The same code in two components
Writing a custom hook
The rules, again
Logic is shared, state isn't
A hook for any type: generics
The React Compiler
Shelfie: hooks and the compiler
Exercises

Chapter 19 — Context and Reducers: Managing Bigger State
When props travel too far
Context: a value that a whole part of the tree can read
`useReducer`: every change in one place
Context and a reducer together
Shelfie: a shelf store and a theme picker
When to reach for a library
Exercises

Chapter 20 — Multiple Pages with React Router
One HTML file, many pages
Installing React Router
Your first router
A layout with `<Outlet>`
URL parameters
Search parameters
"Not found" pages
Navigating from code
Good page changes
Loading pages only when they're needed
An error boundary for each route
Shelfie: four pages
Exercises

Chapter 21 — Testing Your Components
Why test?
Your first test
Reading a failing test

Testing a component
Test what the user sees
If a test can't find it, neither can a screen reader
Shelfie: tests for the components
End-to-end tests with Playwright
Exercises

Chapter 22 — Speaking Your Reader's Language
Translation is the easy part
The browser already knows
Words are data
Shelfie learns French
When a catalog is not enough
Exercises

Part VI — Shipping and Next Steps

Chapter 23 — Building and Deploying Your App
From your code to files a browser can read
Development and production
Environment variables
The published demo and its mock API
Deploying to a static host
Publishing with GitHub Actions
Before you share the address: Lighthouse
Frameworks: the next step
Checking your work
Exercises

Chapter 24 — Debugging and Getting Unstuck
A method for any bug
Where errors show up
React Developer Tools in depth
Reading long TypeScript errors
Reading lint errors
The most common errors
Checking accessibility yourself
Reading the official documentation
Asking for help

Chapter 25 — Where to Go Next
What you can do now
A map of the ecosystem
Next steps in TypeScript
Five projects to build next
Going deeper on accessibility
Going deeper on performance
Staying current without chasing everything
The one idea, again

Appendix A — JavaScript and TypeScript Quick Reference
Modern JavaScript
TypeScript

Appendix B — Hooks Cheat Sheet
React's hooks
The book's custom hooks
React Router's hooks
TanStack Query's hooks

Appendix C — Old Code Alert
Class components
Lifecycle methods
Error boundaries: the class that stays
Refs: string refs, `createRef`, and `forwardRef`
Typing components: `React.FC`, `PropTypes`, and `defaultProps`
`import React from "react"` everywhere
Context: `.Provider` and `useContext`
Starting the app: `ReactDOM.render` and Create React App
Routing: `<Switch>` and `component=`
Hand-made memoization
Forms: `onSubmit` and `preventDefault`
Fetching data in an effect
Redux with `connect`
The whole list

Appendix D — Exercise Solutions

Appendix E — Code Review Checklist
Before you ask for a review
TypeScript
Components
State and effects
Lists, forms, and data
HTML and accessibility
Languages and formats
CSS

Tests
Security and publishing
Appendix F — Glossary

Introduction

You want to build things for the web that people can click on, type into, and come back to. Maybe you have written a few HTML pages, styled them with CSS, and added a bit of JavaScript to make a button do something. It worked, until the page grew. Then every new feature meant finding the right element, changing its text, remembering to change the other element that showed the same information, and hoping you didn't forget one.

React is a tool for exactly that moment. Instead of changing the page step by step, you describe what the page should look like for the data you have, and React keeps the page in sync when the data changes. That one idea is why React became the most widely used way to build web interfaces, and it's what this book teaches, from the very beginning.

What you will build

In this book, you build one app from start to finish: **Shelfie**, a personal reading list. It lets you keep track of the books you want to read, the ones you are reading, and the ones you've finished.

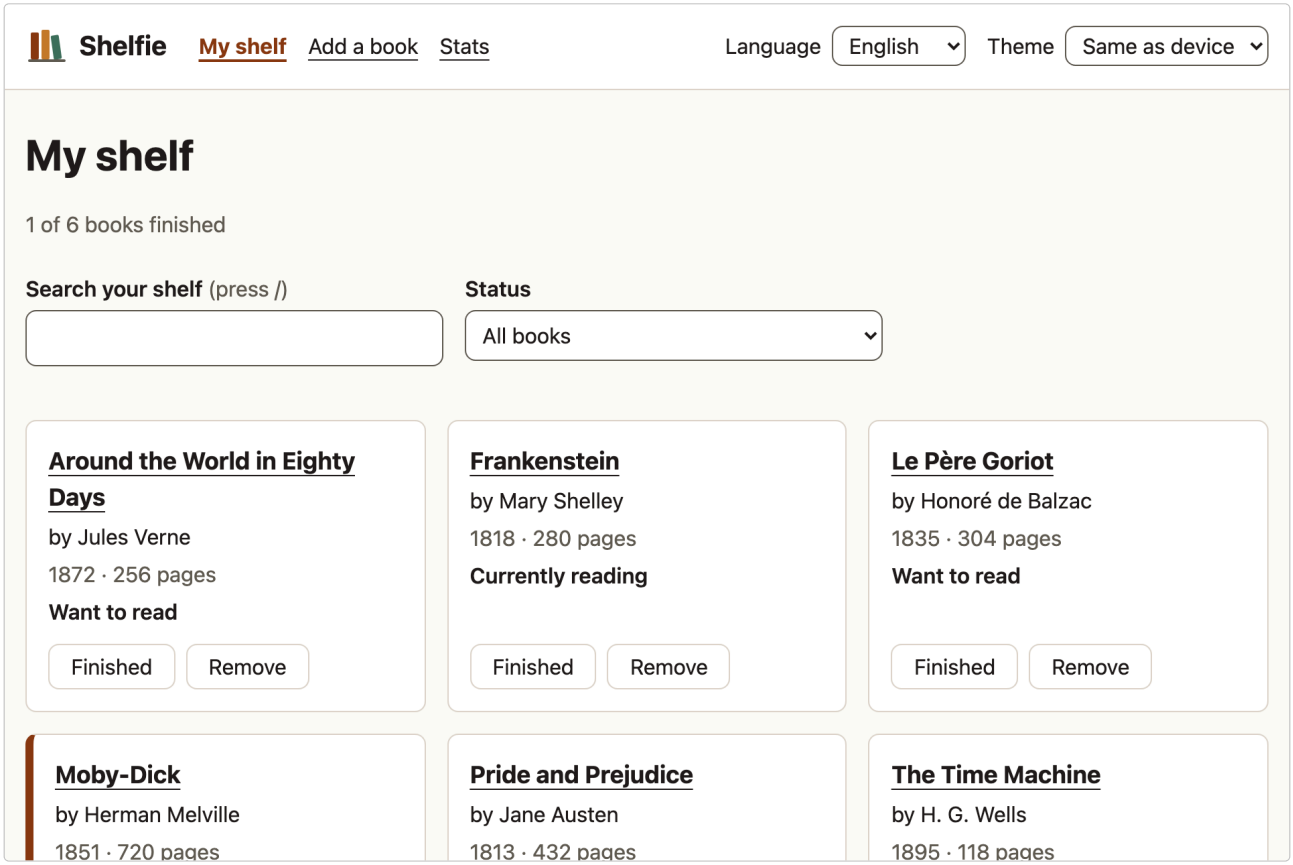


Figure 1: Shelfie, the app you build in this book, as it looks at the end

By the last chapter, Shelfie:

- shows your books as a list of cards that you can search and filter;
- lets you add a book with a form that checks what you typed and explains what's wrong, in a way that screen reader users can follow too;
- searches a book catalog on a server, and shows what happens while it waits and when the server fails;
- remembers your shelf and your preferences when you close the browser;
- has several pages, with links you can bookmark and share;
- works with a mouse, a keyboard, or a screen reader, in light and dark mode, on a phone and on a desktop;
- has automated tests that prove all of this still works after a change;
- speaks English or French, with dates and numbers to match;
- is published on the internet.

You don't build it all at once. Each chapter teaches one idea with small examples. Most chapters then add that idea to Shelfie, and at the end of each of them, you have a working app that is a little better than before.

Who this book is for

This book is for you if:

- you know some HTML: you've written a page with headings, paragraphs, links, lists, and a form;
- you know some CSS: you can change a color or a margin with a selector;
- you've written a little JavaScript: a variable, a function, an `if`;
- you can open a terminal and type a command, even if you don't feel at home there yet.

You don't need to know React, TypeScript, Node.js, npm, Git, or any build tool. The book teaches each of them when you first need it, and only as much as you need.

If you have used React before by copying code from tutorials, this book is for you too. It explains *why* that code works, which is what makes the next piece of code easier to write.

Why TypeScript from the first page

Every example in this book is written in **TypeScript**, a version of JavaScript that lets you say what kind of data each variable holds. That might sound like more to learn at a moment when you're already learning a lot. It is a little more to learn, but it pays for itself almost at once: your editor starts to underline mistakes while you type them, before you even open the browser. A misspelled property, a missing piece of data, a function called with the wrong thing — TypeScript points at each of these and tells you what's wrong.

Chapter 4 teaches the TypeScript you need to get started, and later chapters add a little more when the code needs it. You never have to learn TypeScript "properly" before you can use React.

Written the way professionals write it

A book for beginners is a book people copy from, so every example in it is written the way it should be written in a real project, today:

- **TypeScript in strict mode**, with extra checks turned on, and no shortcuts that switch the checks off.
- **Modern React**: function components and hooks, the form features of React 19, and no patterns that the React team now advises against.
- **The right HTML element for the job**: real buttons, real links, real lists, labeled form fields, and headings in order. This is what makes an app usable with a keyboard or a screen reader, and it's much easier to learn from the start than to fix later.
- **Tools that check all of this for you**: a linter with strict rules for TypeScript, React, and accessibility, a code formatter, and automated tests.

When the book shows code that is wrong, it's on purpose: to show you the error you'll get, and how to fix it. Those examples are always clearly marked, and the correct version always follows.

How the code in this book works

Every complete listing in this book is a real file, read in when the book is built, not typed into the text by hand. What you see on the page is exactly what is in the companion archive.

The same goes for everything the book says you'll *see*. The screenshots are taken automatically from the real app running in a real browser. The error messages are copied from real runs of the TypeScript compiler, the linter, or the browser's console. Every time the book is built, all the examples are checked, built, and tested, and every page in the screenshots is checked for accessibility problems. If a new version of a tool changes an error message, the book changes with it.

In captured console output, `[error]`, `[warning]`, and `[uncaught]` stand for the red and yellow lines the browser shows.

Versions

The book is written for **React 19.3**, **TypeScript 6.0**, **Vite 8**, and **React Router 8**, running on **Node.js 24**. The exact version of every package is pinned in the samples, so what you install is what was checked.

React releases new versions regularly, and the samples are kept up to date. The ideas in this book don't change between minor versions, so the text rarely mentions a version number. When a feature is new, a *Version note* says so.

> **Old code alert** You'll find lots of React tutorials online that were written years ago. If one tells you to run `create-react-app`, or shows components written as `class`es, it's out of date. Appendix C shows the old patterns you might meet and what replaced them.

How to read this book

The book has six parts, and each one opens with a page telling you what you already know, what the next chapters teach, and what to watch out for.

Part	Chapters	What it covers
I — Getting Ready	1–4	What React is, setting up your tools, and the JavaScript and TypeScript you need.
II — Thinking in Components	5–9	JSX, components, props, lists, and styling. Shelfie starts to take shape.
III — Making Things Interactive	10–14	Events, state, updating lists, forms, and sharing state between components.
IV — Talking to the Outside World	15–17	Effects, saving data, loading data from a server, and working with the page directly.
V — Growing Your App	18–22	Custom hooks, context, reducers, several pages, automated tests, and other languages.
VI — Shipping and Next Steps	23–25	Publishing your app, getting unstuck, and where to go next.

Read the chapters in order: each one builds on the ones before it. Chapters 24 and 25, and the appendices, are reference material you'll come back to.

Most chapters follow the same shape. They start with what you'll build, explain the idea in plain words, show it in a small example, and then add it to Shelfie. They end with a **Carry forward** box that sums up what the next chapters expect you to know, and with exercises.

After the exercises comes an **Expected observations** box. It tells you, for each exercise, what you should see and what the exercise was meant to teach. Try each exercise first, and guess the result before you look: that guess is where most of the learning happens. If your answer is different, that's worth understanding, not hiding.

Six appendices sit at the end of the book. Appendix A is a JavaScript and TypeScript quick reference, and Appendix B gives each hook a card with its signature and an example. Appendix C shows the old code you'll meet in tutorials, next to what this book writes instead. When an exercise's answer is code, Appendix D shows it. Appendix E gathers the book's rules as checklists to review your own code, and Appendix F, the glossary, collects every term the book defines.

Some boxes in the text ask you to stop for a moment:

> **Try it** A small change to make yourself, to see what happens.

> **Why did that happen?** A real error or surprise, with what it means and how to fix it.

Gotcha A mistake that is easy to make and hard to spot.

Jargon A new word, explained.

Type tip A TypeScript detail worth knowing, which you can skip on a first read.

Old code alert Something you'll see in older code and tutorials, and what this book writes instead.

Version note A detail that depends on the version of a tool or library the book uses.

Under the hood What React is really doing. Interesting, and safe to skip.

Don't do this Code that is wrong on purpose. The right version always follows.

Using the samples

The companion archive — `react-from-zero-samples.zip` — contains every example in this book. Download it from `https://goto.meziantou.net/react-from-zero-samples.zip`. Chapter 2 walks you through installing the tools and the archive, and running your first project. After that, you'll mostly work in your own Shelfie project, and use the archive when you want to compare.

In the archive	What it is
`samples/shelfie/chNN/`	Shelfie as it is at the end of chapter NN: the chapter's *checkpoint*. Only chapters that change Shelfie have one. If your version stops working and you can't find why, compare it with the checkpoint for the chapter you're reading, or start again from the previous chapter's checkpoint.
`samples/shelfie/final/`	The finished app.
`samples/chNN-.../`	The small examples of chapter NN, and the solutions to its exercises. `samples/ch27-old-code/` holds the old code of Appendix C.
`README.md`	How to install and run everything in the archive.

Every path printed in this book is written relative to the folder that holds the archive's `README.md`. When a chapter describes a Shelfie file instead of printing all of it, copy that file from the chapter's checkpoint folder, then read the explanation.

Asking for help

You will get stuck sometimes. Everyone does, and it doesn't mean you're doing it wrong. Chapter 24 is about that: how to read an error message, how to find where a problem comes from, and how to ask a good question. You don't have to wait until Chapter 24 to use it: skim it early, and come back to it whenever something doesn't make sense.

Let's start with the question every React book has to answer first: what is React, and why would you want it?

Chapter 1 — What Is React, and Why Should You Care?

React is a JavaScript library for building user interfaces: the part of an application that people see and interact with. It was created at Facebook, released as open source in 2013, and is now used by millions of websites and apps, from small personal projects to some of the largest sites on the internet.

That's the dictionary definition, and it doesn't tell you much. This chapter explains what React is *for*: the problem it solves, the one big idea behind it, and why that idea makes web apps easier to build. You won't install anything yet — Chapter 2 does that — so sit back and read. Every example in this chapter is in the companion archive, and you'll be able to run all of them once your tools are set up.

The problem: keeping the page in sync

Let's start with a tiny app written without React: a counter of the books you've read. It has a heading with the count, a message under it, and two buttons.

Here is its HTML page:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0"
    />
    <title>Reading counter</title>
  </head>
  <body>
    <main>
      <h1>Books read: <span id="count">0</span></h1>
      <p id="message">You haven't finished a book yet.</p>
      <button id="add" type="button">I finished a book</button>
      <button id="reset" type="button">Start over</button>
    </main>
    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

And here is the code that makes it work. It's written in TypeScript, which you haven't learned yet, but it reads almost like plain JavaScript: find the elements on the page, keep a `count` variable, and change the page when a button is clicked.

```
import "./style.css";

// Find the elements this script needs to change.
const countElement = document.querySelector("#count");
const messageElement = document.querySelector("#message");
const addButton = document.querySelector("#add");
const resetButton = document.querySelector("#reset");
if (
  countElement === null ||
  messageElement === null ||
  addButton === null ||
  resetButton === null
```

```
) {
  throw new Error("The page is missing an element.");
}

let count = 0;

addButton.addEventListener("click", () => {
  count = count + 1;
  countElement.textContent = String(count);
  messageElement.textContent =
    count === 1
      ? "You finished your first book!"
      : `You finished ${count} books.`;
});

resetButton.addEventListener("click", () => {
  count = 0;
  countElement.textContent = "0";
  // Forgot something? The message still shows the old count.
});
```

Click “I finished a book” twice, and the page does what you expect:



Figure 2: After two clicks, the heading and the message agree

Now click “Start over”:



Figure 3: After “Start over”, the heading says 0 and the message still says 2

The heading says 0, but the message still says you finished 2 books. The code that handles “Start over” resets the count and updates the heading, but it forgets the message. The comment in the listing gives the bug away, but in a real app, nothing would.

This bug is small, and it’s easy to fix: add one more line to the “Start over” handler. But look at what caused it. The same piece of information — the number of books — is shown in two places, and every piece of code that changes the number has to remember to update *both* places. Add a third place (a progress bar, a page title, a badge in a menu), and every handler has to update three places. Add a third button, and it has to know about all three places too.

In a real app, with dozens of pieces of information shown in hundreds of places, this is how bugs are born. Code like this is called **imperative**: it gives the browser a list of step-by-step instructions — find this element, change its text, find that element, change its text — and it’s your job to make sure the list is complete every single time.

> **Jargon** The **DOM** (Document Object Model) is the browser's live version of your page: a tree of objects, one for each element, that code can read and change. `document.querySelector` finds an element in the DOM, and setting `textContent` changes what that element shows. When people say "update the DOM", they mean "change the page".

The big idea: describe the page, let React update it

Here is the same counter written with React:

```
import { useState } from "react";

function describe(count: number) {
  if (count === 0) {
    return "You haven't finished a book yet.";
  }
  if (count === 1) {
    return "You finished your first book!";
  }
  return `You finished ${count} books.`;
}

export function App() {
  const [count, setCount] = useState(0);

  return (
    <main>
      <h1>Books read: {count}</h1>
      <p>{describe(count)}</p>
      <button type="button" onClick={() => setCount(count + 1)}>
        I finished a book
      </button>
      <button type="button" onClick={() => setCount(0)}>
        Start over
      </button>
    </main>
  );
}
```

Don't worry about the details: `useState` is explained in Chapter 11, and the HTML-like syntax inside the function, called JSX, is the subject of Chapter 5. For now, notice what the code *doesn't* do:

- It never looks for an element on the page.
- It never changes the text of an element.
- The "Start over" button only says "the count is now 0". It doesn't know that there's a heading and a message that depend on the count.

Instead, the `App` function *describes* what the page should look like for a given count: a heading with the count, a message that `describe` computes from the count, and two buttons. When a button changes the count, React calls `App` again, gets the new description, compares it with the previous one, and changes only what it found different on the page.

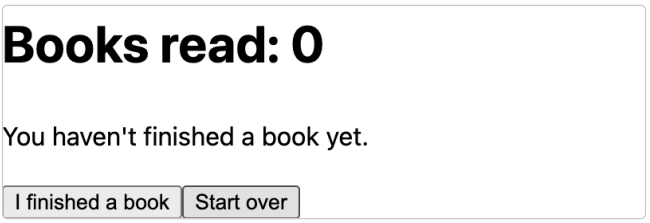


Figure 4: The React counter after the same three clicks: everything agrees

The bug from the previous version can't happen here. The message is computed from the count every time, so there's no way for it to be out of date. Adding a progress bar would mean adding one line to the description, and the buttons would never need to know about it.

This is the one idea that the rest of the book builds on:

> *You describe what the page should look like for the current data. When the data changes, React updates the page to match.*

Code written this way is called **declarative**: it declares the result you want, not the steps to get there. The difference is like the difference between giving a friend directions ("take the second left, then walk 200 meters, then turn right at the bakery") and giving them an address. With directions, one wrong step and they're lost. With an address, finding the way is someone else's job. In React, that someone is React.

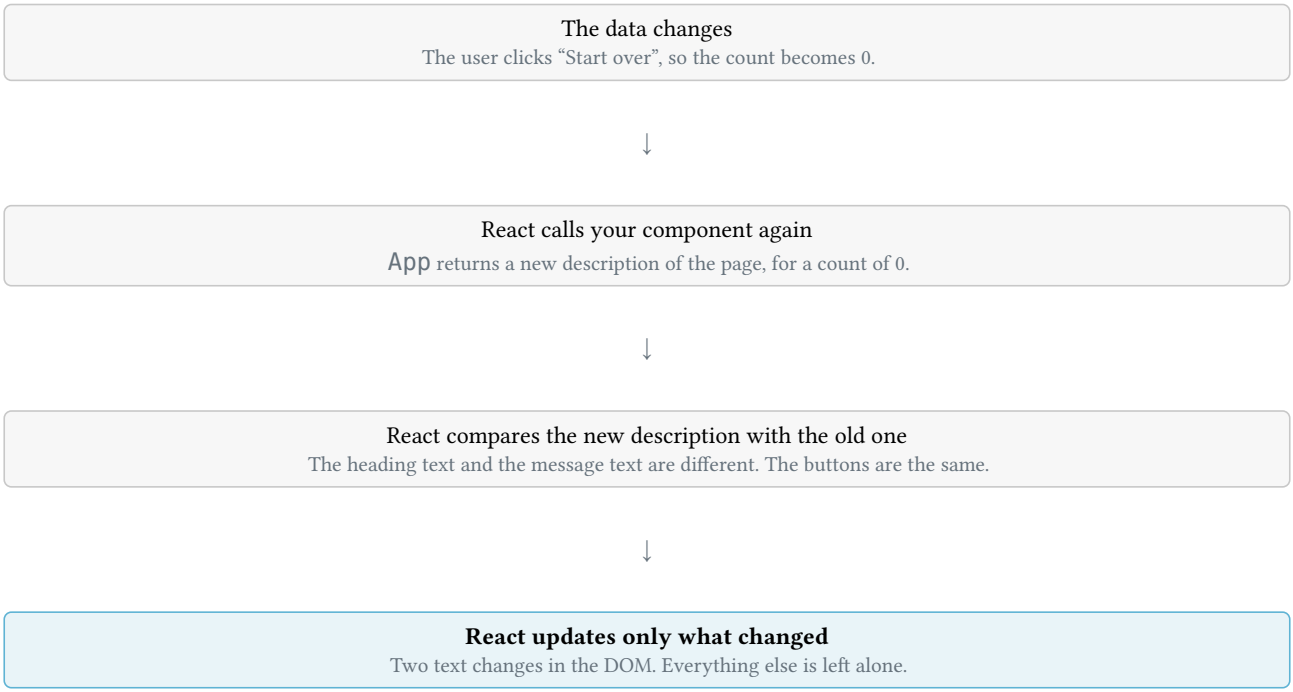


Figure 5: What happens when data changes in a React app

> **Under the hood** Comparing two descriptions is fast because they are plain JavaScript objects, not real DOM elements. React only touches the real page for the differences it finds, and touching the page is the slow part. You'll sometimes read that React uses a "virtual DOM": that's the name people give to this tree of plain objects, and the comparison is called *reconciliation*. You never need to think about it to write a React app.

Components: the building blocks

The React counter is a single function called `App`. A function like this, which returns a description of part of the page, is called a **component**.

Components are to React what LEGO bricks are to a LEGO model. Each one is small and does one job. You combine small components into bigger ones, and those into a whole page. A page built this way is a tree of components: a frame at the top, which contains a header, a main area, and a footer, which contain smaller components, and so on down to buttons and labels.

Here is how Shelfie, the app you'll build in this book, is split into components:

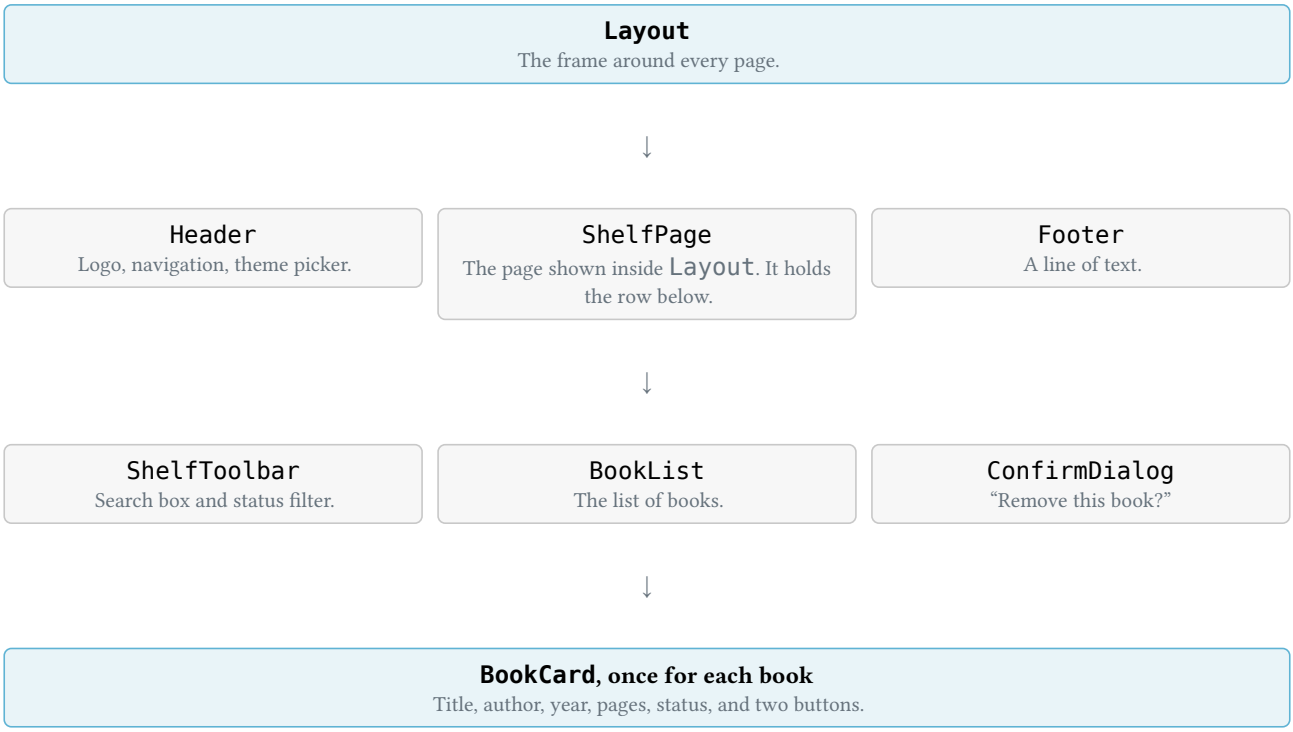


Figure 6: Shelfie's shelf page as a tree of components

Components have two big advantages:

- **You write something once and use it many times.** `BookCard` is written once, and React shows one for every book on your shelf, each with its own title and author. If you change `BookCard`, every card changes.
- **You can think about one piece at a time.** When the search box misbehaves, you open `ShelfToolbar` and ignore the rest of the app. A large app stays manageable because every piece is small.

Chapters 5 to 9 are all about building a page out of components.

Why this book uses TypeScript

Every example in this book is written in TypeScript. TypeScript is JavaScript with one addition: you can say what kind of data a variable holds, and a tool called the *type checker* checks that your code uses the data correctly, before the code ever runs.

Here is a small program with a typo. It describes a book, then prints the book's title:

```
const book = {
  title: "Frankenstein",
  author: "Mary Shelley",
};

console.log(`Now reading: ${book.tittle}`);
```

The last line says `tittle` instead of `title`. If you run this program as plain JavaScript, it doesn't fail. It prints this:

```
Now reading: undefined
```

JavaScript doesn't complain when you ask for a property that doesn't exist: it quietly gives you `undefined`, and the word "undefined" ends up in the output. In a real app, it ends up on the page, and you might not notice until a user sends you a screenshot.

The TypeScript type checker notices right away. This is what it says about the same file:

```
console/typo.ts(6,34): error TS2551: Property 'tittle' does not exist on type
    '{ title: string; author: string; }'. Did you mean 'title'?
```

It points at line 6, column 34, tells you that `tittle` doesn't exist, and even suggests the fix. Your editor shows the same message as a red squiggle under the typo, while you type. That's the whole point of TypeScript: it moves a whole family of bugs from "a user finds it" to "your editor finds it".

TypeScript's messages can look intimidating at first. This book shows you the most common ones on purpose, one at a time, and explains how to read them. Chapter 4 teaches the TypeScript you need to get started.

What React is not

A few misconceptions are worth clearing up early:

- **React is not a language.** React code is TypeScript (or JavaScript). The HTML-like parts are a syntax extension called JSX that a tool turns into normal function calls before the browser sees them.
- **React is not a framework that does everything.** React takes care of components and of keeping the page in sync with your data. Other libraries handle other jobs: React Router for several pages (Chapter 20), TanStack Query for loading data from a server (Chapter 16), and so on. Frameworks like Next.js bundle React with many of these pieces; Chapter 23 explains when you might want one.
- **React is not only for big companies.** It's as useful for a small personal app like Shelfie. The ideas you learn here are the same at every scale.
- **React doesn't replace HTML and CSS.** Everything React shows on the page is HTML elements, styled with CSS. What you already know about both keeps being useful, and this book makes a point of using HTML well.

A tour of the finished app

Here is Shelfie as it will be at the end of the book. The front page, which you saw in the introduction, shows your shelf: every book as a card, with a search box and a filter.

The "Add a book" page lets you search a catalog of books on a server. Type a few letters, and Shelfie finds the matching books and tells you which ones are already on your shelf:

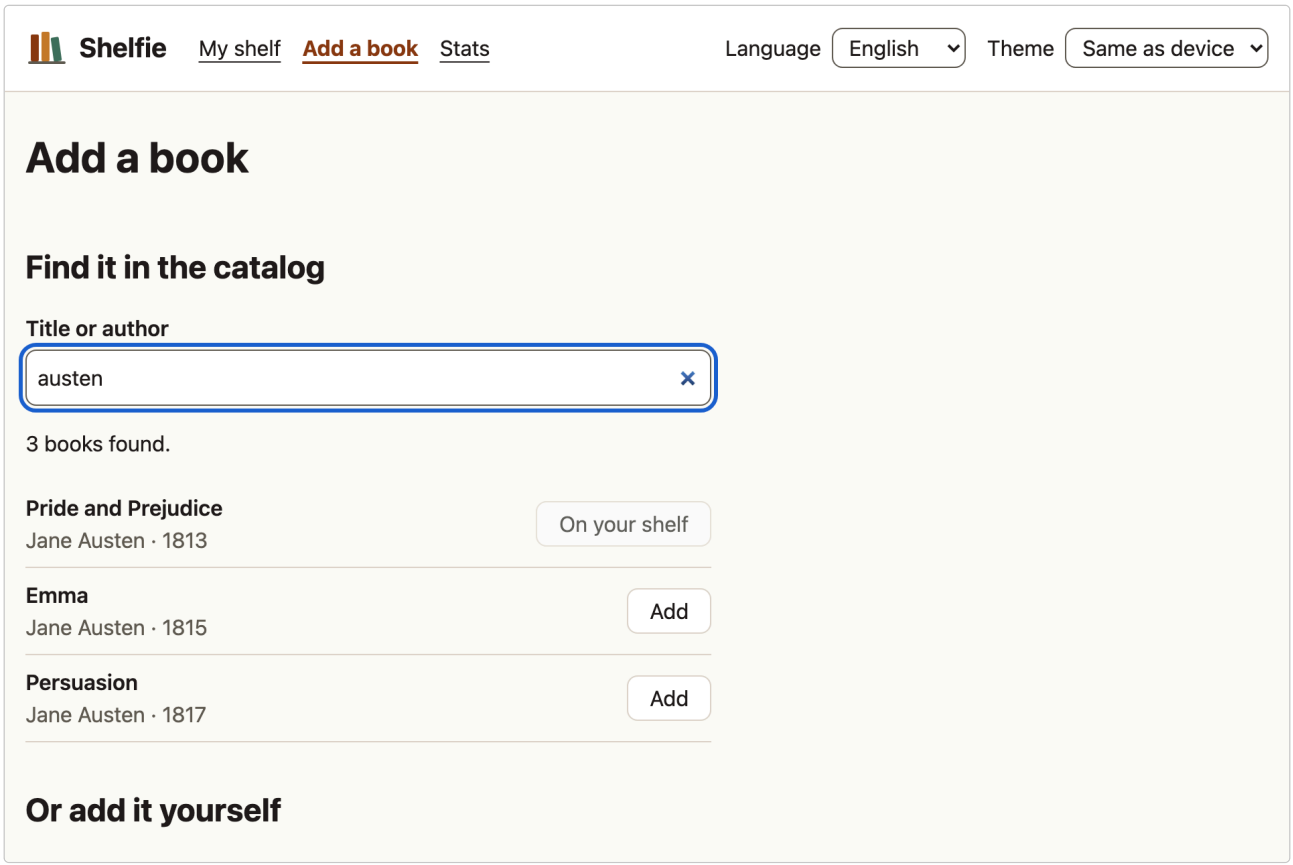


Figure 7: Searching the catalog for “austen” on the “Add a book” page

Each book has its own page, with its own address, so you can bookmark it. And if your computer is set to dark mode, so is Shelfie, unless you choose otherwise in the theme picker:

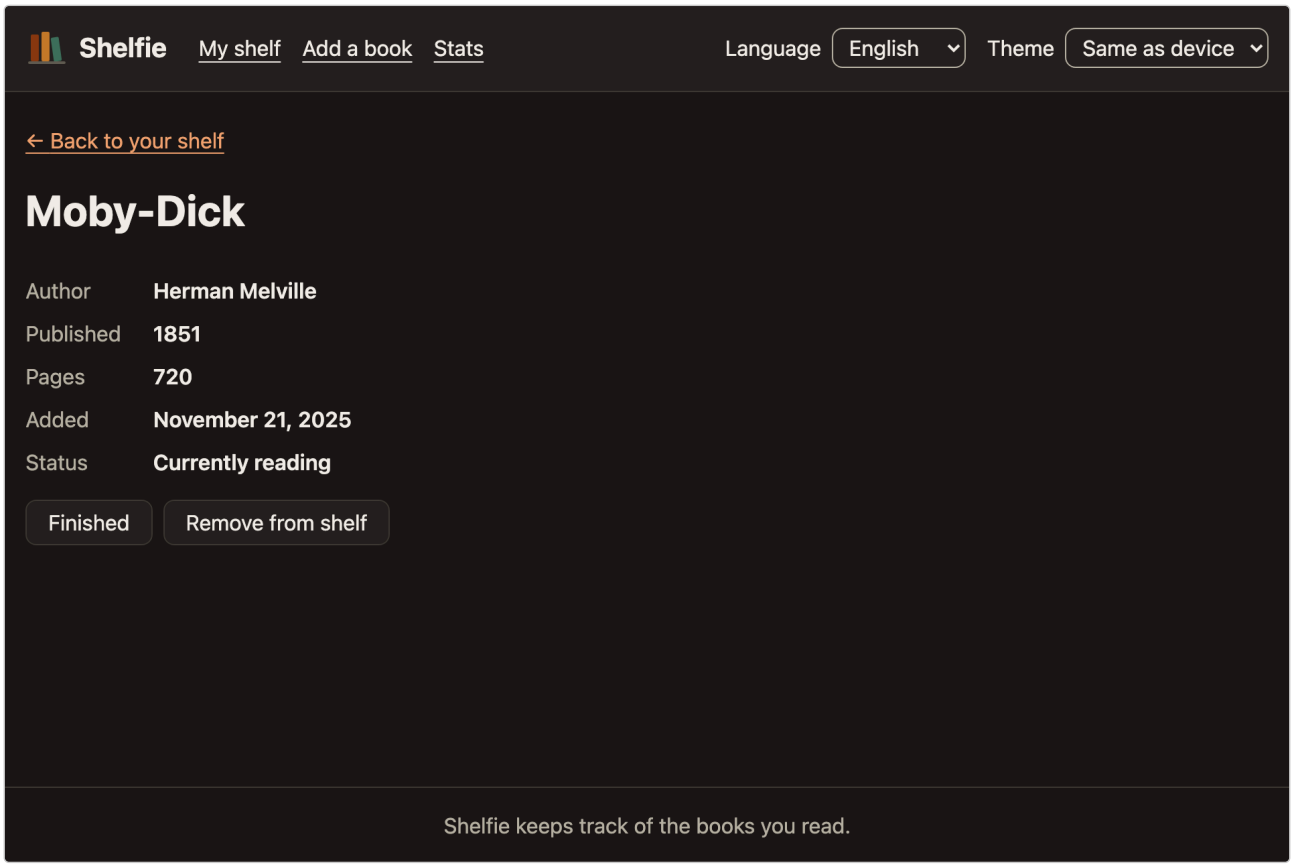


Figure 8: A book's page, in dark mode

Shelfie works on a phone too. The layout adapts to the width of the screen:

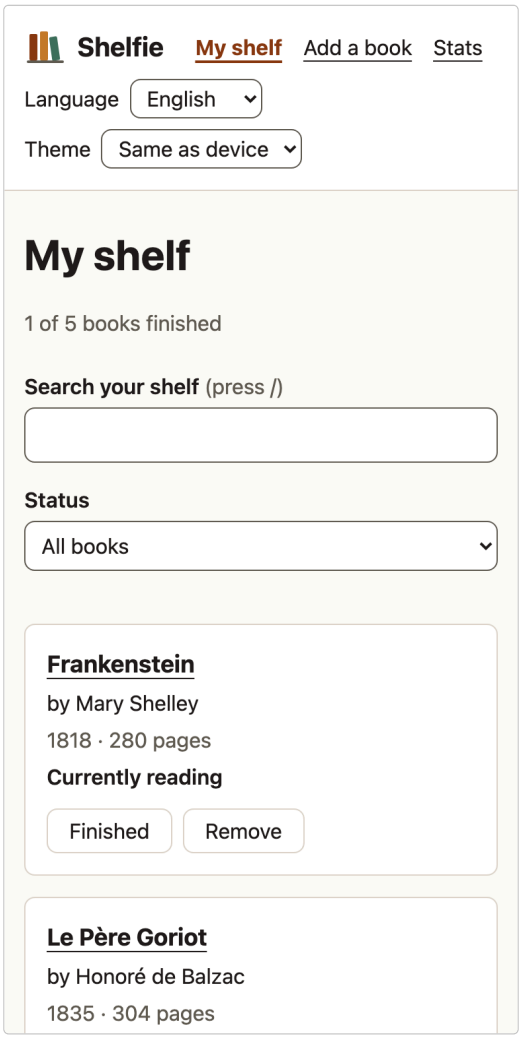


Figure 9: The shelf on a phone-sized screen

Some of Shelfie's best features don't show in a screenshot. You can use all of it with only a keyboard. A screen reader announces what each button does, which fields have errors, and when a search is done. Your shelf is saved in the browser, so it's still there tomorrow. And automated tests check all of this every time the code changes.

You'll build every piece of it, one chapter at a time.

The tools you'll use

Building a React app involves a handful of tools. You'll install them in the next chapter, but here is what each one does, so the names aren't a mystery:

Tool	What it does
Node.js	Runs JavaScript and TypeScript outside the browser. The other tools are written for it.
npm	Comes with Node.js. Downloads the libraries your app uses, like React, and runs your project's commands.
Vite	Turns your TypeScript and JSX into files a browser understands, and runs a local web server that updates the page as soon as you save a file.
TypeScript	Checks your code for mistakes before it runs.
ESLint and Prettier	ESLint points out code that is likely to be a bug or that breaks good practices. Prettier formats your code consistently, so you never have to think about spaces and line breaks.
VS Code	A free code editor that shows TypeScript and ESLint errors as you type.
React Developer Tools	A browser extension that shows the components on a page and their data.

Carry forward

- React keeps the page in sync with your data. You describe what the page should look like for the current data, and React updates the page when the data changes.
- Code that describes the result is *declarative*. Code that lists the steps to change the page is *imperative*, and it's where out-of-sync bugs come from.
- A *component* is a function that describes a piece of the page. An app is a tree of components.
- TypeScript checks your code before it runs and catches mistakes like misspelled property names.

Exercises

You don't have any tools installed yet, so these exercises are about thinking, not running code. Come back to them after Chapter 2 if you want to try them for real.

1. Someone asks you to add a feature to the plain TypeScript counter: a line that says how many books are left to reach a goal of 10. List every place in `main.ts` that would have to change. Then do the same for the React version.
2. Fix the bug in the plain counter: write down the line that is missing from the "Start over" handler, and where it goes.
3. Open a website you use often — an online shop, a news site, a streaming service. Draw boxes around the parts of one page that look like components. Which ones appear more than once?
4. Go back to the typo example, and suppose you write `book.Title` instead of `book.title`. Predict what plain JavaScript prints, and what kind of message TypeScript gives.

Expected observations

Predict first, then check. A different answer is worth understanding, not correcting away.

1. In the plain version, the new line needs an element in `index.html`, one more `querySelector` (and one more check that it was found), and one more line in **both** click handlers — and in every handler anyone adds later. In the React version, it's one new function or expression and one new line in the JSX, in one place. The buttons don't change at all.
   Takeaway: *In imperative code, every piece of code that changes the data must know every place the data is shown. In React, only the description knows, and it's in one place.*
2. The handler needs to update the message too, for example `messageElement.textContent = "You haven't finished a book yet.";` after the line that resets `countElement`. Notice that this text now exists in two places — the HTML file and the handler — and has to be kept the same by hand.
   Takeaway: *Fixing an out-of-sync bug in imperative code often means duplicating something, which sets up the next out-of-sync bug.*
3. **Open.** A good answer finds a few large boxes (a header, a main area, a footer), smaller boxes inside them, and at least one box that repeats with different content: a product card, an article teaser, a video thumbnail. The repeated boxes are the best candidates for a component, like `BookCard` in Shelfie.
   Takeaway: *Seeing a page as a tree of repeated boxes is the first skill of a React developer, and you can practice it on any website.*
4. JavaScript prints `Now reading: undefined`, as with `tittle`: property names are case-sensitive, so `Title` and `title` are different properties, and `Title` doesn't exist. TypeScript reports the same kind of error, "Property 'Title' does not exist", and suggests `title`.
   Takeaway: *To TypeScript, a wrong capital letter is as much a mistake as a wrong letter. That's the kind of slip it's good at catching.*

Chapter 4 — Just Enough TypeScript

Every program you've seen so far ends in `.ts` or `.tsx`, and Chapter 3 already slipped a few types into its programs, like the `: number` after a parameter. This chapter explains them. It covers the TypeScript you need for the next ten chapters, and nothing more: later chapters add the rest when the code needs it.

There's no new page to look at in this chapter. You'll work in small programs that run in the terminal, as in Chapter 3, and what you'll build is a skill: reading what TypeScript tells you. You'll meet the errors beginners see most, each one on purpose, and learn what each one means. You'll also write the `Book` type that Shelfie uses from Chapter 7 on.

The programs are in `samples/ch04-typescript/basics`. Open that folder in VS Code (*File > Open Folder*), and open its built-in terminal. Run a program with `node`, as in Chapter 3, and check the whole folder with the type checker:

```
node console/inference.ts
npm run typecheck
```

The programs are in the `console` folder, and the files they import are in `shared`. `typecheck` should print only the name of the command it runs. The broken examples are in `samples/ch04-typescript/_nobuild`, one folder per mistake, so that they don't stop the check from passing.

What a type is

A variable can hold a book title, a number of pages, a list of authors, or a whole book. Each of those is a different *kind* of value, and you can do different things with each kind: you can divide a number, but not a title.

> **Jargon** A **type** describes a kind of value: what it can hold and what you can do with it. `string` is the type of text, `number` the type of numbers (whole or not), and `boolean` the type of `true` and `false`.

JavaScript only looks at types while the code runs, and it's very forgiving about them. TypeScript adds a step that happens before the code runs.

> **Jargon** The **type checker** is the part of TypeScript that reads your code without running it, works out the type of every value, and reports every place where a value is used in a way its type doesn't allow. `npm run typecheck` runs it on the whole project, and VS Code runs it while you type.

Picture a spell checker for code: it underlines the mistake while you write, and it doesn't change what you wrote. Here's a function that adds two numbers of pages. The `: number` after each parameter says what kind of value the function expects. The call passes one of them as text, in quotes:

Don't do this

```
function totalPages(first: number, second: number): number {
  return first + second;
}

console.log(totalPages(280, "432"));
```

```
console/wrong-argument.ts(5,29): error TS2345: Argument of type 'string' is not
    assignable to parameter of type 'number'.
```

Line 5, column 29 is `"432"`: the function wants a `number`, and got a `string`. Now run the program anyway, with `node`:

```
280432
```

No error at all. JavaScript "added" a number and a text by gluing them together, and a real app would claim you've read 280,432 pages.

This is the most important fact about TypeScript: **types are only checked before the code runs, never while it runs.** `node` and Vite's development server (as Chapter 2 showed) don't check types: they remove them and run the JavaScript that's left. So the type checker only helps if you look at what it says: the red squiggles in the editor, and `npm run typecheck`.

> **Under the hood** `node` replaces each type with blank space and runs the rest, and Vite removes the types as quickly. Node.js can only do this because the book avoids the few TypeScript features that aren't simple types, like `enum` (`erasableSyntaxOnly` keeps it that way).

Inference: TypeScript already knows

You don't have to write a type on every variable. Most of the time, TypeScript works it out from the value.

> **Jargon** **Type inference** is TypeScript working out a type by itself, from the value you give a variable or from what a function returns. The type it works out is the **inferred type**.

Here's a program with no types written anywhere:

```
const title = "Frankenstein";
let pagesRead = 0;
pagesRead = pagesRead + 120;
const finished = pagesRead >= 280;
const authors = ["Mary Shelley", "Jane Austen"];

console.log(title, pagesRead, finished);
console.log(authors.join(" and "));
```

```
Frankenstein 120 false
Mary Shelley and Jane Austen
```

TypeScript still knows the type of every variable. Point your mouse at a variable name in VS Code, and a small box, called a *hover*, shows its type:

- `pagesRead` is `number`: it started as `0`. Putting text into it later would be an error, because a variable's type is decided when it's declared.
- `finished` is `boolean`, because `>=` always gives `true` or `false`.
- `authors` is `string[]`. The square brackets mean "an array of".
- `title` is `"Frankenstein"`. That's not a mistake: a `const` can never change, so TypeScript uses the most exact type there is, this one text and nothing else. You'll use types like this later in this chapter.

> **Try it** Hover over `console.log`, and over `join` in `authors.join`: the hover shows what each function expects and returns. Hovering is the quickest way to answer "what is this?", and you'll use it in every chapter of this book.

Writing types yourself

When TypeScript can't work a type out, or when you want to state what you expect, you write the type yourself.

> **Jargon** A **type annotation** is a type you write in the code, after a colon: `pages: number`. It tells TypeScript what a variable, a parameter, or a function's result must be.

You'll write annotations most on function parameters, on the value a function returns, and on empty arrays:

```
function pagesLeft(pages: number, pagesRead: number): number {
  return pages - pagesRead;
}

function progress(title: string, finished: boolean): string {
  return finished ? `${title}: done` : `${title}: in progress`;
}

const wishlist: string[] = [];
wishlist.push("Moby-Dick");

console.log(pagesLeft(280, 120));
console.log(progress("Frankenstein", false));
console.log(wishlist);
```

```
160
Frankenstein: in progress
[ 'Moby-Dick' ]
```

- `pages: number` and `pagesRead: number` say what each parameter must be. Every call is checked against them, as you saw with `"432"`.
- `): number {` is the **return type**, after the parentheses: `pagesLeft` returns a `number`, and `progress` a `string`. It's a promise that TypeScript checks: if the body broke it, the error would be inside the function, not everywhere the function is used. TypeScript can infer return types too (hover over `progress`). This chapter writes them so you can see them; components, from Chapter 5 on, don't.
- `const wishlist: string[] = []`: an empty array doesn't say what will go in it, so the annotation does. Now `wishlist.push(42)` would be an error.

A parameter of a function you declare yourself always needs a type. It has no value until someone calls the function, so there is nothing to infer from. (Callbacks you pass to `map` or `filter` are the exception: TypeScript knows what they'll receive.) Here's the `totalPages` from the start of the chapter without its annotations:

Don't do this

```
function totalPages(first, second) {
  return first + second;
}

console.log(totalPages(280, 432));
```

```
total.ts(1,21): error TS7006: Parameter 'first' implicitly has an 'any' type.
total.ts(1,28): error TS7006: Parameter 'second' implicitly has an 'any' type.
```

`any` is TypeScript's word for "any type at all, and don't check anything". Giving a parameter that type *silently* would switch the checks off, so the book's strict settings make it an error. The fix is to write the type of each parameter. The end of the chapter shows why the book never uses `any`.

Describing objects with `type`

A book is more than one value: it has a title, an author, a year, and more. You describe the shape of an object with a **type alias**.

> **Jargon** A **type alias** gives a name to a type, with the `type` keyword: `type Book = { ... };`. After that, you can write `Book` anywhere a type is expected. An **object type** lists the properties an object has, and the type of each one.

Here's the type of every book in Shelfie, in `shared/types.ts`:

```
export type Book = {
  id: string;
  title: string;
  author: string;
  year: number;
  pages: number;
};
```

Each line inside the braces is a property: its name, a colon, and its type. A `Book` has five properties, and all of them are required. The `id` is a short text that identifies the book, like `"frankenstein"`; Chapter 8 needs it. Shelfie has no data yet, so you'll add this type to it in Chapter 7, as `src/types.ts`, with exactly this content. The `export` lets other files import `Book`, as with functions.

> **Old code alert** Many tutorials write `interface Book { ... }` instead. For the objects in this book, both do the same job, so the book uses `type` everywhere, and its lint rules enforce that.

Once a type has a name, you can use it in an annotation. `sampleBooks.ts` holds three books, and says so (the file continues with two more books):

```
import type { Book } from "./types.ts";

export const sampleBooks: Book[] = [
  {
    id: "pride-and-prejudice",
    title: "Pride and Prejudice",
    author: "Jane Austen",
    year: 1813,
    pages: 432,
  },
```

`Book[]` means "an array of `Book`s", and TypeScript checks every object in the array against `Book`. The `import type` on the first line imports something that is only a type; it has its own section later. Here's `console/books.ts`, a program that uses the books:

```
import { sampleBooks } from "../shared/sampleBooks.ts";
import type { Book } from "../shared/types.ts";

function describeBook(book: Book): string {
  return `${book.title}, by ${book.author} (${book.year})`;
}

const longBooks = sampleBooks.filter(
  (book) => book.pages > 400,
```

```
);

console.log(sampleBooks.map(describeBook).join("\n"));
console.log(`Long books: ${longBooks.length}`);
```

```
Pride and Prejudice, by Jane Austen (1813)
Frankenstein, by Mary Shelley (1818)
Moby-Dick, by Herman Melville (1851)
Long books: 2
```

Apart from the import, `book: Book` and the `: string` return type are the only types in the program. Inside the arrow function passed to `filter`, `book` is inferred as a `Book`, and `longBooks` as a `Book[]`, because `filter` returns the same kind of array. Hover over them to check. That's the pattern for the whole book: annotate the data where it comes into your program and the parameters of your functions, and let inference do the rest.

The object type is a checklist. An object with a missing property is an error, and so is a misspelled one:

Don't do this

```
const emma: Book = {
  id: "emma",
  titel: "Emma",
  author: "Jane Austen",
  year: 1815,
  pages: 474,
};
```

```
emma.ts(5,3): error TS2561: Object literal may only specify known properties, but
    'titel' does not exist in type 'Book'. Did you mean to write 'title'?
```

A `Book` has no `titel` property, and TypeScript guesses what you meant. It doesn't also report that `title` is missing: once it finds a property it doesn't know, it stops checking the object. Fix the spelling, and both problems go away.

Optional properties and `readonly`

A review of a book always has some text, but maybe no star rating. And the book a review is about should never change. Two small words describe that:

```
type Review = {
  readonly bookId: string;
  text: string;
  stars?: number;
};

const quickNote: Review = {
  bookId: "moby-dick",
  text: "Long, but worth it.",
};

const fullReview: Review = {
  bookId: "frankenstein",
  text: "Much sadder than the films.",
  stars: 5,
```

```
};

quickNote.text = "Long, and worth every page.";

console.log(quickNote);
console.log(fullReview);
```

```
{ bookId: 'moby-dick', text: 'Long, and worth every page.' }
{
  bookId: 'frankenstein',
  text: 'Much sadder than the films.',
  stars: 5
}
```

- The question mark in `stars?: number` makes the property **optional**: an object can have it or not. `quickNote` has no `stars`, and that's fine.
- `readonly` makes `bookId` **read-only**: it gets its value when the object is created, and can't be changed afterward. `text` can change.

Try to change a `bookId`, and the type checker stops you. The file in `_nobuild/readonly` has the same `Review` type and one review, then does this:

Don't do this

```
review.bookId = "frankenstein";
```

```
reviews.ts(12,8): error TS2540: Cannot assign to 'bookId' because it is a read-only
    property.
```

Like every type, `readonly` only exists for the type checker, but it turns a rule you'd have to remember into one the editor checks. Chapter 12 makes every property of Shelfie's `Book` read-only, because React needs you to create new objects instead of changing existing ones (the immutability of Chapter 3).

Type tip Reading an optional property gives a type that includes `undefined`: `quickNote.stars` is a `number | undefined`. The next two sections explain that `|`, and how to deal with `undefined`.

Union and literal types

A book on your shelf is either being read or finished. TypeScript can say exactly that:

```
type ReadingStatus = "reading" | "finished";

function statusLabel(status: ReadingStatus): string {
  if (status === "reading") {
    return "Reading now";
  }
  return "Finished";
}

console.log(statusLabel("reading"));
console.log(statusLabel("finished"));
```

```
Reading now
```

```
Finished
```

> **Jargon** A **literal type** is a type with exactly one value: `"reading"` is the type of the text `"reading"` and nothing else. A **union type** combines types with a vertical bar, `|`, and means "one of these": `"reading" | "finished"` accepts those two texts and nothing else.

So `ReadingStatus` has exactly two possible values. Unlike a plain `string`, it catches typos, both in the values you pass and in the values you compare with:

Don't do this

```ts
type ReadingStatus = "reading" | "finished";

function statusLabel(status: ReadingStatus): string {
  if (status === "finshed") {
    return "Finished";
  }
  return "Reading now";
}

console.log(statusLabel("done"));
```

```
status.ts(4,7): error TS2367: This comparison appears to be unintentional because the
    types 'ReadingStatus' and '"finshed"' have no overlap.
status.ts(10,25): error TS2345: Argument of type '"done"' is not assignable to
    parameter of type 'ReadingStatus'.
```

The first error is the comparison: `"finshed"` can never equal a `ReadingStatus` ("no overlap" means no value is both), so the comparison "appears to be unintentional". The second is the call: `"done"` isn't one of the two values. VS Code also completes the values: type `statusLabel("` and it lists `"reading"` and `"finished"`.

Narrowing with `if`

A union can also mix different kinds of types. Here, your progress in a book is either a number of pages read, or the text `"finished"`:

```ts
type Progress = number | "finished";

function progressLabel(
  progress: Progress,
  pages: number,
): string {
  if (progress === "finished") {
    return "Finished";
  }
  const percent = Math.round((progress / pages) * 100);
  return `${percent}% read`;
}

console.log(progressLabel(120, 280));
console.log(progressLabel("finished", 280));
```

```
43% read
Finished
```

Hover over `progress` on the `if` line: its type is `Progress`. Hover over it in the division on line 10: `number`. If `progress` were `"finished"`, the function would have returned already, so only the number is left.

> **Jargon** **Narrowing** is TypeScript working out a more exact type for a value after a check, such as an `if`. Inside the `if`, `progress` is `"finished"`. After it, `progress` is `number`.

Narrowing is what makes unions useful. Without the `if`, TypeScript won't let you divide:

Don't do this

```
function progressLabel(
  progress: Progress,
  pages: number,
): string {
  const percent = Math.round((progress / pages) * 100);
  return `${percent}% read`;
}
```

```
progress.ts(7,31): error TS2362: The left-hand side of an arithmetic operation must
    be of type 'any', 'number', 'bigint' or an enum type.
```

The "left-hand side" is `progress`, and `Progress` isn't in the list of types you can divide, because it might be `"finished"`. (The book uses neither `any` nor `enum`; the end of the chapter explains why.) Dividing `"finished"` by 280 is a real bug, and the `if` is the code that handles it.

`null` and `undefined`

JavaScript has two values that mean "nothing here":

- `undefined` is what you get when something has no value: a variable you never assigned, a property that isn't there, or an array item that doesn't exist. `find` also returns `undefined` when nothing matches.
- `null` is a value you use on purpose to say "nothing", for example for "the book I'm reading now" when you're not reading any.

The book's `tsconfig.base.json` sets `strict: true`, so `null` and `undefined` are **not** part of any type unless you say so. A `Book` is always a book. A value that might be missing has a union type: `Book | undefined`, or `Book | null`.

```
import { sampleBooks } from "../shared/sampleBooks.ts";
import type { Book } from "../shared/types.ts";

function findBook(id: string): Book | undefined {
  return sampleBooks.find((book) => book.id === id);
}

function currentTitle(current: Book | null): string {
  if (current === null) {
    return "Nothing right now";
  }
  return current.title;
}

const found = findBook("emma");
console.log(found?.title);
console.log(found?.title ?? "No such book");
```

```
console.log(findBook("moby-dick")?.title ?? "No such book");
console.log(currentTitle(null));
console.log(currentTitle(findBook("frankenstein") ?? null));
```

```
undefined
No such book
Moby-Dick
Nothing right now
Frankenstein
```

- `findBook` returns `Book | undefined`, because `find` might not find anything (hover over `find` to see it).
- In `currentTitle`, after `if (current === null)` returns, TypeScript narrows `current` to `Book`, as in the previous section.
- `found?.title` uses optional chaining, from Chapter 3. Its type is `string | undefined`, and it prints `undefined`: there's no book `"emma"`.
- `?? "No such book"` replaces `null` or `undefined` with a value you choose, so the type becomes `string`.
- `findBook("frankenstein") ?? null` turns `undefined` into `null`, because `currentTitle` accepts `null`, not `undefined`. To TypeScript, they differ.

When you forget that a value might be missing, TypeScript reminds you:

Don't do this

```
import type { Book } from "./types.ts";

const books: Book[] = [];

const found = books.find((book) => book.id === "emma");
console.log(found.title);

function currentTitle(current: Book | null): string {
  return current.title;
}

console.log(currentTitle(null));
```

Why did that happen?

```
books.ts(6,13): error TS18048: 'found' is possibly 'undefined'.
books.ts(9,10): error TS18047: 'current' is possibly 'null'.
```

Both errors say "possibly": the type allows a missing value. `found` is a `Book | undefined`, and `current` is a `Book | null`. Reading `.title` on `undefined` or `null` crashes the program, so TypeScript won't let you until you've handled the missing case, with one of the three fixes you've seen: an `if` that returns early, `?.` when "nothing" is an acceptable result, or `??` to fall back to a value.

Gotcha Some code "fixes" these errors with an exclamation mark: `found!.title`. As Chapter 2 showed, `!` only silences the question; if the value is missing, the program still crashes. The book's lint rules forbid it.

Why `books[0]` might be `undefined`

An array can be shorter than you think. It can even be empty, and then `books[0]` doesn't exist:

Don't do this

```ts
import type { Book } from "../types.ts";

function describeFirst(books: Book[]): string {
  return `First on your shelf: ${books[0].title}`;
}

console.log(describeFirst([]));
```

Run it with an empty array, and the program crashes, one of the most common crashes in JavaScript:

```
samples/ch04-typescript/_nobuild/unchecked-index/console/first-book-throws.ts:4
  return `First on your shelf: ${books[0].title}`;
                                          ^

TypeError: Cannot read properties of undefined (reading 'title')
```

The type checker saw it coming. "Object" means the value before `.title`, `books[0]`, which has no name:

```
console/first-book-throws.ts(4,34): error TS2532: Object is possibly 'undefined'.
```

This check comes from `noUncheckedIndexedAccess`, one of the settings Chapter 2 added. With it, reading an array by position gives a `Book | undefined`, and you handle the missing case like any other (`console/first-book.ts`):

```ts
function describeFirst(books: Book[]): string {
  const first = books[0];
  if (first === undefined) {
    return "Your shelf is empty.";
  }
  return `First on your shelf: ${first.title}`;
}
```

Hover over `first` where it's declared: `Book | undefined`. On the last line: `Book`, thanks to the `if`. You'll rarely read arrays by position in React code: Chapter 8 goes through whole arrays with `map`, and to get one item, `find` with a check for `undefined` is usually clearer.

`import type` for types

`books.ts` imports a value, `sampleBooks`, which exists when the program runs, and a type, `Book`, which doesn't. `node` and Vite remove the types before running the code, and `import type` tells them "this whole line is about types, remove it too".

> **Jargon** A **type-only import** is an import that brings in only types: `import type { Book } from "./types.ts";`. It's removed entirely before the code runs.

Here's a plain `import` for a type:

Don't do this

```ts
import { Book } from "../types.ts";

const emma: Book = {
```

```
  id: "emma",
  title: "Emma",
  author: "Jane Austen",
  year: 1815,
  pages: 474,
};

console.log(emma.title);
```

Why did that happen?

```
console/import-throws.ts(1,10): error TS1484: 'Book' is a type and must be
    imported using a type-only import when 'verbatimModuleSyntax' is enabled.
```

`Book` is a type, so it must be imported with `import type`. The setting named in the message, `verbatimModuleSyntax`, is in the book's `tsconfig.base.json`: imports stay as written when the types are removed, so you must mark the type-only ones yourself. Run the program anyway, and you see why:

```
samples/ch04-typescript/_nobuild/import-type/console/import-throws.ts:1
import { Book } from "../types.ts";
         ^^^^
SyntaxError: The requested module '../types.ts' does not provide an export named
    'Book'
```

`node` removed the annotation on `emma`, but kept the `import` line. Once its type is removed, `types.ts` is empty, so it has no export named `Book`, and the program stops before its first line runs.

The fix is the word `type`. The book's lint rules check it too.

Union types instead of `enum`

Many TypeScript tutorials describe a fixed set of values with an `enum`:

Don't do this

```
enum ReadingStatus {
  Reading,
  Finished,
}

function statusLabel(status: ReadingStatus): string {
  if (status === ReadingStatus.Reading) {
    return "Reading now";
  }
  return "Finished";
}

console.log(statusLabel(ReadingStatus.Finished));
```

```
console/enum-throws.ts(1,6): error TS1294: This syntax is not allowed when
    'erasableSyntaxOnly' is enabled.
```

`erasableSyntaxOnly`, another of the book's settings, only allows TypeScript features that can be erased: removed completely with the types. An `enum` can't: it creates a real object when the program runs, so `node` refuses it:

```
SyntaxError [ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX]: TypeScript enum is not supported in
    strip-only mode
```

A union of literal types, like `ReadingStatus` earlier, does the same job with types only. You write `"reading"` instead of `ReadingStatus.Reading`, and a printed value is a word instead of a number (the enum's values are `0` and `1`). Same checks, same completion, one fewer concept.

> **Old code alert** Replace an `enum` from a tutorial with a union of strings: `enum Status { Reading, Finished }` becomes `type Status = "reading" | "finished";`, and `Status.Reading` becomes `"reading"`. Exercise 3 practices that.

Reading a type error

All the type errors you've seen have the same shape. Look again at the `readonly` one, and read it from left to right:

1. **The file**: `reviews.ts`.
2. **The position**: line 12, column 8 (written `(12,8)` in the book, `:12:8` in your terminal). In VS Code, that's a red squiggle, and hovering over it shows the message. The *Problems* panel (*View* > *Problems*) lists the errors in the files you have open, and Error Lens, from Chapter 2, prints each one at the end of its line.
3. **The code**: `TS2540`. You don't need to remember codes, but searching the web for one finds explanations.
4. **The message**: what's wrong, in one sentence, usually naming the values and types involved, in quotes.

Some messages are longer. Here, a book's year was written as text, and the book is passed to a `describeBook(book: Book)` function:

Don't do this

```ts
const emma = {
  id: "emma",
  title: "Emma",
  author: "Jane Austen",
  year: "1815",
  pages: 474,
};

console.log(describeBook(emma));
```

Why did that happen?

```
emma.ts(15,26): error TS2345: Argument of type '{ id: string; title: string;
    author: string; year: string; pages: number; }' is not assignable to parameter
    of type 'Book'.
  Types of property 'year' are incompatible.
    Type 'string' is not assignable to type 'number'.
```

The first line is the summary: the object you passed isn't a `Book`. Each indented line explains the one above it, one step deeper, and the last line is the most precise: a `string` was found where a `number` was expected. The fix is to remove the quotes around `1815`. So, read the first line to know *where* the problem shows, and the last line to know *what* it is.

The error is on the call, line 15, not on the object with the mistake. `emma` has no annotation, so TypeScript saw nothing wrong until it was used as a `Book`. With `const emma: Book = {`, the error moves to the `year` line. That's why the book annotates data where it's created.

The three errors you'll see most

Most of the errors you'll meet in the next chapters are one of these three. The second one uses a word you'll read in type errors all the time:

> **Jargon** **Assignable** means "fits". "Type `A` is not assignable to type `B`" means that a value of type `A` was put where a `B` is required: a variable of type `B`, a parameter of type `B`, a prop of type `B`. The value is always on the left, and what it was supposed to fit on the right. Read it as "this `A` doesn't fit in that `B`".

The message says	It usually means	Try this first
Property `'x'` does not exist on type `'T'` (`TS2339`, `TS2551`)	A typo in a property name, or the value isn't what you think it is.	Check the spelling (TypeScript often suggests it), then hover over the value to see its real type.
Type `'A'` is not assignable to type `'B'` (`TS2322`, `TS2345`)	You gave a value of the wrong kind: text instead of a number, or an object that doesn't match its type.	Read the last line of the message. Fix the value, not the type, unless the type is wrong.
`'x'` is possibly `'undefined'` or `'null'` (`TS18048`, `TS18047`, `TS2532`)	The value might be missing, and the code reads it as if it's always there.	Handle the missing case: an `if` that returns early, `?.`, or `??`.

Table 1: The three most common type errors

> **Gotcha** When there are several errors, fix the first one first. One mistake often causes others further down, and they disappear with it. Then run `npm run typecheck` again.

Gotcha: why the book never uses `any`

`any` switches the type checker off: a value of type `any` can be used in any way at all. Older code and tutorials use it all over, usually to make a red squiggle go away. Here's what it costs:

Don't do this

```
function describeBook(book: any): string {
  return `${book.title}, ${book.pagse} pages`;
}

const emma = { title: "Emma", pages: 474 };
console.log(describeBook(emma));
```

There's a typo on line 2, `pagse`. `book` is `any`, so the type checker says nothing, and the program prints:

```
Emma, undefined pages
```

The typo went all the way to the output, as in plain JavaScript: the cost of TypeScript, and none of the help. The book's lint rules catch it, though. If this function were in your Shelfie project, `npm run lint` (Chapter 2) would report:

```
any-bug.ts:1:29  error  Unexpected any. Specify a different type.  @typescript-
    eslint/no-explicit-any
any-bug.ts:2:13  error  Invalid type "any" of template literal expression.
    @typescript-eslint/restrict-template-expressions
any-bug.ts:2:18  error  Unsafe member access .title on an `any` value.  @typescript-
    eslint/no-unsafe-member-access
any-bug.ts:2:28  error  Invalid type "any" of template literal expression.
    @typescript-eslint/restrict-template-expressions
any-bug.ts:2:33  error  Unsafe member access .pagse on an `any` value.  @typescript-
    eslint/no-unsafe-member-access
```

The first error forbids writing `any`. The other four point at each place the `any` value is used, because a bug can hide there: two say an `any` can't go in a template literal, and two flag reading a property of an `any`. The last one, at `2:33`, is the typo `.pagse`. The fix is to describe what the function really receives. In `_nobuild/any-replaced`, `book` has a small type with a `title` and `pages`, and the typo is caught again:

```
summary.ts(7,33): error TS2551: Property 'pagse' does not exist on type 'Summary'.
    Did you mean 'pages'?
```

If you really don't know the type of a value yet, as with data from a server, TypeScript has a safe cousin of `any`, `unknown`, that makes you check the value before you use it. Chapters 15 and 16 show how. For everything else, write the type. The same goes for `value as Book`, a *type assertion*: it makes TypeScript believe you without checking, and the lint rules forbid it.

Carry forward

- Types are checked before the code runs, never while it runs. `node` and Vite remove the types and run what's left, so watch the squiggles and run `npm run typecheck`.
- Let inference work, and hover to see what TypeScript knows. Write annotations on parameters, on empty arrays, and on data where it's created (`const books: Book[]`).
- Describe objects with `type`, using `?` for optional properties and `readonly` for properties that never change. Shelfie's `Book` has `id`, `title`, `author`, `year`, and `pages`.
- A value that might be missing has a union type (`Book | undefined`, `Book | null`), and so does `books[0]`. Handle the missing case with `if`, `?.`, or `??`. Unions of literal types replace `enum`.
- Import types with `import type`. Read a type error from the position and the first line, then the last line. Never use `any`.

Exercises

1. Here's a program with three mistakes. Without running anything, find them, and for each one, predict the line and which of the three common errors TypeScript reports. Then copy the program into `console/predict.ts` in the `basics` project, change its first line to `import type { Book } from "../shared/types.ts";`, and run `npm run typecheck` to check. (Delete the file when you're done.)

```ts
import type { Book } from "./types.ts";

const books: Book[] = [
  {
    id: "emma",
    title: "Emma",
    author: "Jane Austen",
```

```
    year: 1815,
    pages: 474,
  },
];

function describeBook(book: Book): string {
  return `${book.title}, ${book.pageCount} pages`;
}

const firstBook = books[0];
console.log(firstBook.author);
console.log(describeBook("Emma"));
```

2. In a new file in the `basics/console` folder, write a type named `ShelfEntry` for a book on your shelf: a `book` that never changes (a `Book`), a `status` that is `"want-to-read"`, `"reading"`, or `"finished"`, and an optional number of `stars`. Then write a function `describeEntry(entry: ShelfEntry): string` that returns "Emma: on the wish list", "Emma: reading now", "Emma: finished, not rated", or "Emma: finished, 4 stars". Call it on four entries and print the results.

3. This program uses an `enum`. Copy it into `basics/console/format.ts` and predict what `npm run typecheck` says. Then rewrite it with a union of literal types, so that it type-checks and runs with `node`.

```
enum Format {
  Paperback,
  Hardcover,
  Ebook,
}

function formatLabel(format: Format): string {
  if (format === Format.Ebook) {
    return "E-book";
  }
  return "Printed book";
}

console.log(formatLabel(Format.Paperback));
console.log(formatLabel(Format.Ebook));
```

4. Write a function (in a new file in `basics/console`) `describeLast(books: Book[]): string` that returns "Last on your shelf: " followed by the title of the *last* book, or "Your shelf is empty." for an empty array. (The last item of an array is at position `books.length - 1`.) Before you handle the empty case, predict what the type checker says and what hovering over your variable shows.

5. In `shared/sampleBooks.ts`, remove the `: Book[]` annotation and the `import type` line. Run `npm run typecheck`, and hover over `sampleBooks` and over `longBooks` in `books.ts`. Then delete the `pages` line of Moby-Dick and run `npm run typecheck` again. Where do the errors appear? Undo all the changes when you're done.

Expected observations

Predict first, then check. A different answer is worth understanding, not correcting away.

1. Run: `npm run typecheck`
   Three errors, one of each kind:

```
predict.ts(14,33): error TS2339: Property 'pageCount' does not exist on type
    'Book'.
predict.ts(18,13): error TS18048: 'firstBook' is possibly 'undefined'.
predict.ts(19,26): error TS2345: Argument of type 'string' is not assignable to
    parameter of type 'Book'.
```

Line 14: a `Book` has `pages`, not `pageCount` (property doesn't exist). Line 18: `firstBook` comes from `books[0]`, so it might be `undefined` (possibly undefined). Line 19: `describeBook` wants a whole `Book`, and the call passes only its title (not assignable).

Takeaway: The first line of each message is enough to tell the three common errors apart.

2. Run: `node solutions/shelf-entry.ts`
   The four lines, one per entry. The solution is `samples/ch04-typescript/basics/solutions/shelf-entry.ts`. Its `describeEntry` checks `"want-to-read"` and `"reading"` with `if` and returns early, so the rest of the function only runs for finished books. Then it checks `entry.stars === undefined` to choose between "not rated" and the number of stars. `stars?: number` makes the property optional, and `readonly book: Book` makes the book read-only.
   Takeaway: An object type with a literal union, an optional property, and `readonly` describes the data exactly, and narrowing handles each case.
3. Run: `npm run typecheck`, then `node` on your rewritten file
   The type checker reports

   ```
   format.ts(1,6): error TS1294: This syntax is not allowed when
       'erasableSyntaxOnly' is enabled.
   ```

   at the `enum` keyword's name, `Format`. The rewrite, `samples/ch04-typescript/basics/solutions/format.ts`, declares `type Format = "paperback" | "hardcover" | "ebook";`, compares with `"ebook"`, and calls `formatLabel("paperback")`. It type-checks and prints "Printed book", then "E-book".
   Takeaway: Every `enum` of this kind becomes a union of strings, with the same checks and nothing left when the types are removed.
4. Run: `npm run typecheck`, then `node solutions/last-book.ts`
   Hovering over the variable that holds `books[books.length - 1]` shows `Book | undefined`. Without the check, the type checker reports

   ```
   last-book.ts(5,33): error TS18048: 'last' is possibly 'undefined'.
   ```

   The solution, `samples/ch04-typescript/basics/solutions/last-book.ts`, returns early when `last === undefined`, and prints "Last on your shelf: Moby-Dick", then "Your shelf is empty.".
   Takeaway: Any position in an array can be missing, not only `[0]`.
5. Run: `npm run typecheck`, twice
   The first time, there's no error: the objects still have the shape of a `Book`. (If you forgot to remove the import, you get `TS6133`: `'Book'` is declared but its value is never read.) Hovering shows the whole shape instead of the name `Book`: `const sampleBooks: { id: string; title: string; ... }[]`, and the same for `longBooks`. The second time, `sampleBooks.ts` still has no error. The errors appear in every file that uses the books as `Book`s: `books.ts`, `first-book.ts`, `missing.ts`, and `solutions/last-book.ts`. In `books.ts`:

   ```
   books.ts(9,13): error TS18048: 'book.pages' is possibly 'undefined'.
   ```

   on the `filter` line, then a very long "not assignable" error on the `map(describeBook)` call (line 12), whose last line says that `pages` is optional in one type but required in `Book`. Without the annotation, TypeScript inferred a type that allows a book without `pages` (`pages?: never` means "no `pages` here"), and only complained when that book was used as a `Book`.
   Takeaway: Annotate data where it's created: the error then shows up next to the mistake, with a short message, instead of far away with a long one.

Chapter 6 — Your First Components

Until now, everything you've written lived in one function, `App`. That's fine for a page with a heading and a paragraph. It stops being fine the moment the page has a header, a list of books, a card for each book, and a footer: one function holding all of that is long, hard to read, and full of repeated markup.

React's answer is the idea this whole book is built on: **components**. In this chapter you'll write your first ones, learn the few rules they have to follow, and turn the Chapter 2 starter project into the first version of Shelfie. Here's where you'll end up:

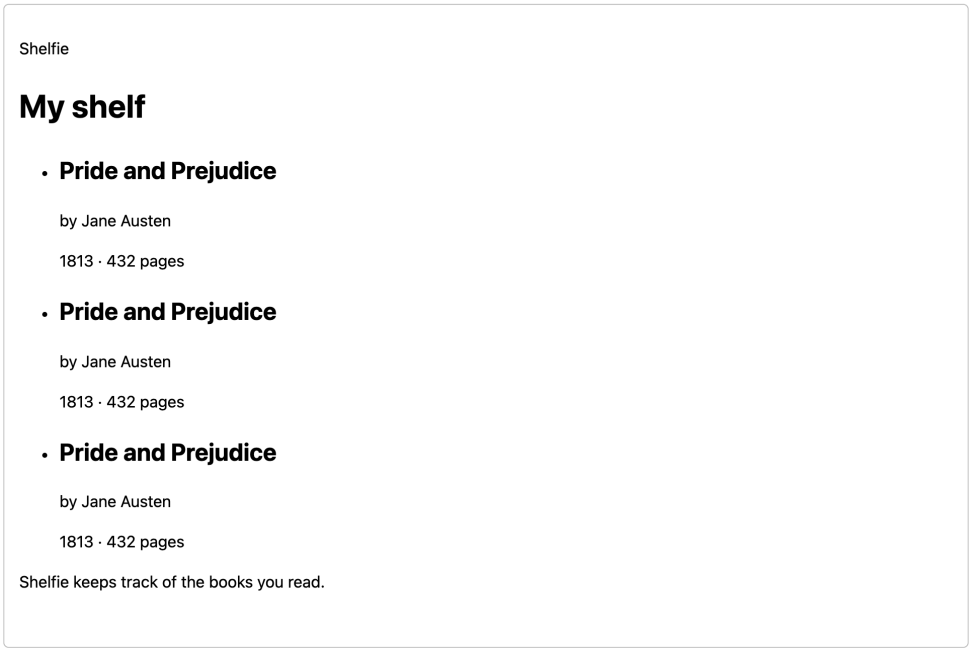


Figure 10: Shelfie at the end of this chapter: a header, a heading, a list of books, and a footer

It's plain, and it shows the same book three times. Both are on purpose. There is no styling yet (that's Chapter 9), and giving each card its own book needs *props*, the subject of Chapter 7. This chapter is about the shape of the app: which pieces it's made of, and where each piece lives.

What a component is

Chapter 1 compared components to LEGO bricks: small pieces that you snap together to build something bigger. Here's the more exact version.

> **Jargon** A **component** is a function that returns JSX. It describes one piece of the page. You use it in JSX like an HTML tag, and React calls the function to find out what to show.

That's the whole definition. There is no special syntax to learn, no class to extend, and nothing to register. Here's a complete component:

```
export function Greeting() {
  return <p>Hello! Welcome to your first component.</p>;
}
```

Read it line by line:

- `export` makes the function available to other files. You met it in Chapter 3, and you'll see more about it later in this chapter.
- `function Greeting()` is an ordinary function declaration. Its name starts with a capital letter, which matters a lot (the next section explains why).

- `return <p>…</p>;` returns JSX: the markup this component stands for.

To show a component on the page, you write its name as a tag inside the JSX of another component. The demo project for this chapter, `samples/ch06-components/demos`, has an `App` component that shows each demo in its own `<section>`, with an `<h2>` heading:

```
<section>
  <h2 id="greeting">One component, used three times</h2>
  <Greeting />
  <Greeting />
  <Greeting />
</section>
```

`<Greeting />` is a self-closing tag, like `<img />` in Chapter 5. Since `Greeting` has nothing to put between an opening and a closing tag, the short form is the one to use. The section uses the component three times, and the browser shows three paragraphs:

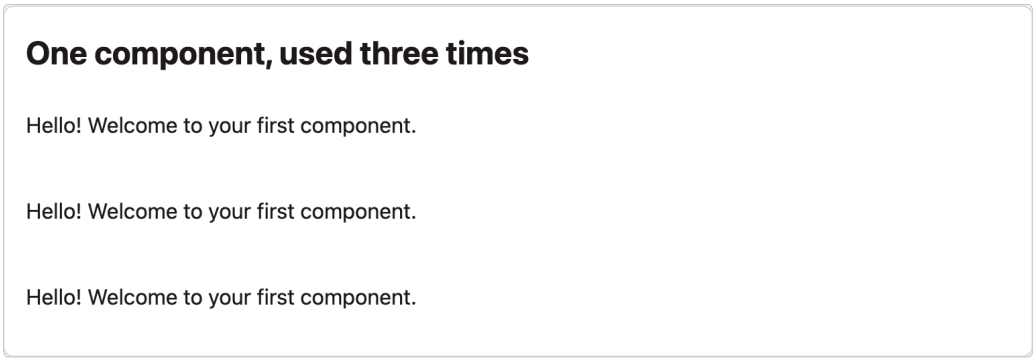


Figure 11: One component, written once, used three times

This is the first thing components give you: you write a piece of the page once and use it as often as you like. If you change the text in `Greeting.tsx`, all three paragraphs change.

> **Jargon** When React calls your component function to find out what it returns, it **renders** the component. "The page renders three `Greetings`" means React called `Greeting` and put what it returned on the page, three times.

You'll also meet components written as arrow functions, such as `const Greeting = () => <p>Hello!</p>;`. They work the same way. This book uses `function` declarations for components, because the name is easier to spot at the start of the line.

Forgetting the `return`

A component must *return* its JSX. The mistake everyone makes at least once is to write the JSX but not return it:

Don't do this

```
function Greeting() {
  <p>Hello! Welcome to your first component.</p>;
}
```

The editor underlines `<Greeting />` where it's used, and `npm run typecheck` prints this:

Why did that happen?

```
App.tsx(9,8): error TS2786: 'Greeting' cannot be used as a JSX component.
  Its type '() => void' is not a valid JSX element type.
    Type '() => void' is not assignable to type '(props: any) => ReactNode |
        Promise<ReactNode>'.
      Type 'void' is not assignable to type 'ReactNode | Promise<ReactNode>'.
```

The first line says what's wrong: `Greeting` can't be used as a component. The lines below it are TypeScript explaining why, one step at a time, and the most useful one in a long error like this is usually the last. Here it says that `void` isn't a `ReactNode`. `void` is TypeScript's word for "returns nothing", and `ReactNode` means "anything React can show". So: the function returns nothing, and React needs something to show. Chapter 7 comes back to `ReactNode`. The `any` in `props: any` is the type Chapter 4 told you to avoid; here it only means that React accepts any props.

Without the `return` keyword, the line `<p>Hello! …</p>;` creates the paragraph and then throws it away. It's the same trap as the arrow function with curly braces in Chapter 3.

The fix is the one word that was missing, as in the `Greeting` you saw at the start of this section:

```tsx
export function Greeting() {
  return <p>Hello! Welcome to your first component.</p>;
}
```

Why the capital letter matters

In Chapter 5, every tag you wrote was an HTML element: `<p>`, `<main>`, `<ul>`. Now tags can also be components. JSX needs a way to tell the two apart, and the rule is short:

- A tag that starts with a **lowercase** letter, like `<p>`, is an HTML element. React creates an element with that name.
- A tag that starts with a **capital** letter, like `<Greeting />`, is a component. React looks for a variable with that name and calls it.

So the capital letter isn't a style preference. It's how React knows what you mean. Here's what happens with a lowercase component:

Don't do this

```tsx
function greeting() {
  return <p>Hello! Welcome to your first component.</p>;
}

export function App() {
  return (
    <main>
      <h1>Component demos</h1>
      <section>
        <h2 id="lowercase">A lowercase component</h2>
        <greeting />
      </section>
    </main>
  );
}
```

Why did that happen?

```
src/App.tsx(1,10): error TS6133: 'greeting' is declared but its value is never
    read.
src/App.tsx(11,9): error TS2339: Property 'greeting' does not exist on type
    'JSX.IntrinsicElements'.
```

Two errors, and together they tell the whole story. Each line starts with the file, then the line and column in parentheses: `(11,9)` is line 11, column 9.

The second error is on `<greeting />`. `JSX.IntrinsicElements` is TypeScript's list of the HTML elements that JSX knows about (`p`, `main`, `ul`, and so on). Because the tag is lowercase, TypeScript looks for `greeting` in that list, and it isn't there.

The first error is the clue: the function `greeting` is "declared but its value is never read". The tag `<greeting />` didn't use your function at all. To JSX, it's the name of an HTML element that happens to be spelled like your function.

Chapter 2 showed that the page still runs while the editor shows a type error, because Vite doesn't check types. So what does the browser do with this code? The section shows its heading and nothing else:



Figure 12: The greeting is missing: React never called the `greeting` function

The browser console explains why:

```
[error] The tag <greeting> is unrecognized in this browser. If you meant to render a
    React component, start its name with an uppercase letter.
```

React did what the lowercase letter asked for: it created an HTML element named `greeting`. That element is empty, and the browser has no idea what a `greeting` element is, so nothing appears. React even guesses what you meant and tells you to start the name with an uppercase letter.

Jargon **PascalCase** means writing every word with a capital first letter and no spaces: `Greeting`, `BookCard`, `WelcomeCard`. Components are always named in PascalCase. Ordinary variables and functions use **camelCase**, which starts with a lowercase letter: `bookTitle`, `countPages`.

The fix is to rename the function `Greeting` and use `<Greeting />`, like the demo in the previous section.

Components that use other components

A component's JSX can use other components, not only HTML elements. That's how you build a page out of small pieces. Here's a card that uses `Greeting`:

```tsx
import { Greeting } from "./Greeting.tsx";

export function WelcomeCard() {
  return (
    <article>
      <h3>Welcome</h3>
      <Greeting />
      <p>This card is a component that uses another one.</p>
    </article>
```

```
  );
}
```

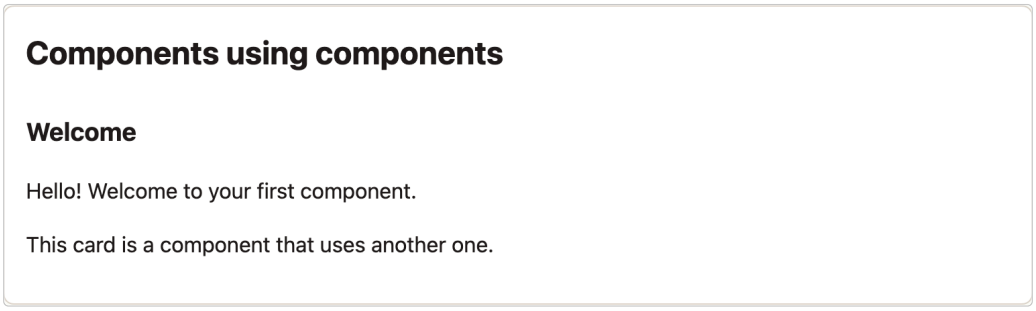


Figure 13: `WelcomeCard` shows its own heading and paragraph, with `Greeting` in between

`WelcomeCard` doesn't know or care what `Greeting` returns. It puts `<Greeting />` where the greeting should go, and React fills that spot with whatever `Greeting` renders.

> **Jargon** When `WelcomeCard` uses `Greeting` in its JSX, `WelcomeCard` is the **parent** component and `Greeting` is its **child**. Following every parent to its children, starting from `App`, gives you the **component tree**: the family tree of your app.

One component per file

Each component in the demo project lives in its own file, and the file has the same name as the component: `Greeting` is in `Greeting.tsx`, `WelcomeCard` is in `WelcomeCard.tsx`. That's the convention this book follows everywhere, and most React projects do the same:

- When you're looking for `BookCard`, you know which file to open.
- Each file stays short enough to read in one go.
- The extension is `.tsx` because the file contains JSX. A file with no JSX, such as a list of types, uses `.ts`.

A component that's in its own file needs to be exported, and the file that uses it needs to import it. These are the modules from Chapter 3, with nothing new:

- `export function Greeting()` is a **named export**: the function is exported under its own name.
- `import { Greeting } from "./Greeting.tsx";` imports it by that name. The curly braces must contain the exact name that the other file exports.
- `"./Greeting.tsx"` is the path to the file, relative to the current file. `./` means "the same folder". The book's TypeScript settings let you write the `.tsx` extension, so the path matches the real file name.

As Chapter 3 explained, the book uses named exports everywhere, because the name you import is checked against the name that was exported, so the editor catches a typo in either one.

> **Old code alert** Many tutorials write `export default function App()` and import it with `import App from "./App"`. That's a *default export*. It works, but the importing file can call the component anything it likes, so the same component can have different names in different files. Older tutorials also start every component file with `import React from "react";`. React no longer needs that line for JSX, and in this book's projects TypeScript even reports it as unused.

Forgetting `export` is the most common mistake with modules. Here's a `Greeting.tsx` that doesn't export its component, and an `App.tsx` that tries to import it:

Don't do this

```
function Greeting() {
  return <p>Hello! Welcome to your first component.</p>;
}
```

```
import { Greeting } from "./Greeting.tsx";

export function App() {
  return (
    <main>
      <h1>Component demos</h1>
      <Greeting />
    </main>
  );
}
```

Why did that happen?

```
App.tsx(1,10): error TS2459: Module '"./Greeting.tsx"' declares 'Greeting'
    locally, but it is not exported.
Greeting.tsx(1,10): error TS6133: 'Greeting' is declared but its value is never
    read.
```

TypeScript found the file and found a function named `Greeting` in it ("declares 'Greeting' locally"), but the function isn't exported, so no other file can use it. The second error is the same problem seen from inside `Greeting.tsx`: nothing in that file uses the function, and nothing outside can.

The fix is to add `export` in front of `function`, which gives you the `Greeting.tsx` from the start of the chapter.

Splitting a page into components

How do you decide which components to write? A good way to start is with a picture of the page and a pencil. Draw a box around each piece that has one clear job, and give the box a name. Each named box becomes a component. Boxes inside boxes become children.

Here's the Shelfie page from the start of the chapter, with the boxes drawn in:

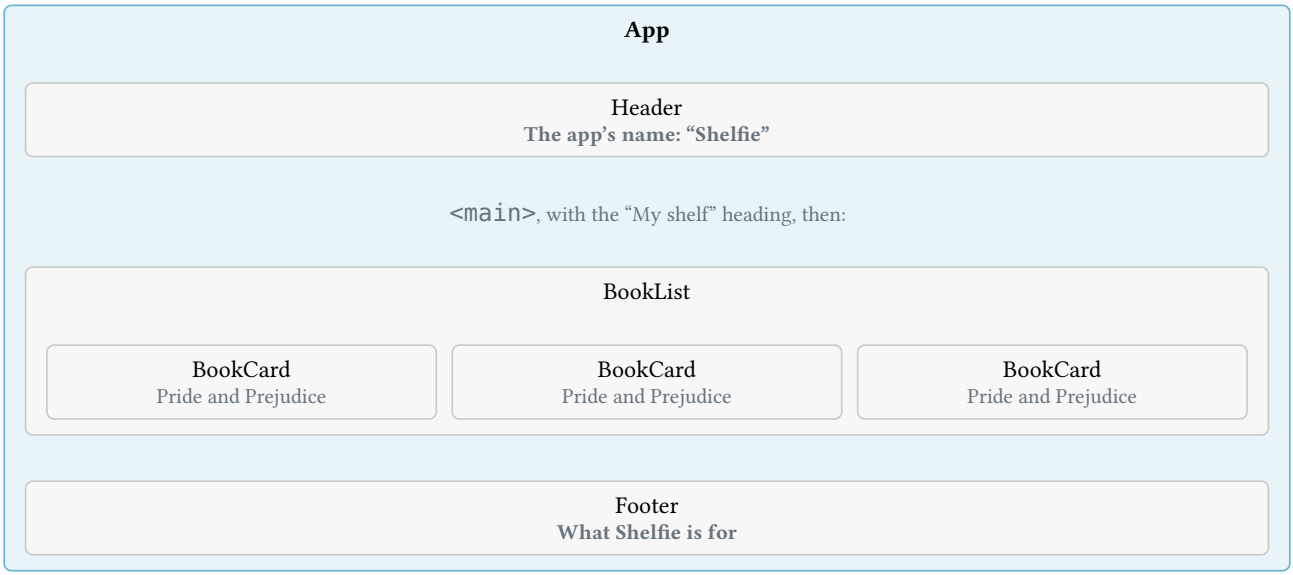


Figure 14: The Shelfie page with a box around each component. The `main` element and its heading belong to `App` itself.

Each box earned its place for a reason:

- **`BookCard`** shows one book. It's the piece that repeats, and repetition is the clearest sign that you need a component: write it once, use it many times.
- **`BookList`** groups the cards. It's the one place that knows the books form a list.
- **`Header`** and **`Footer`** each have one job, at the top and the bottom of the page. Neither repeats, but giving them names keeps `App` short and makes each one easy to find and change.
- **`App`** is the whole page. It arranges the other pieces and adds what belongs to the page itself: the `<main>` element and the page's heading.

Not every element deserves a box. The "My shelf" heading is a single line, with no job beyond being the heading, so it stays in `App`. A useful test: if you can't give a box a clear name, it probably isn't a component yet. You can always split a component later when it grows.

The same boxes, drawn as a tree, give you Shelfie's component tree:

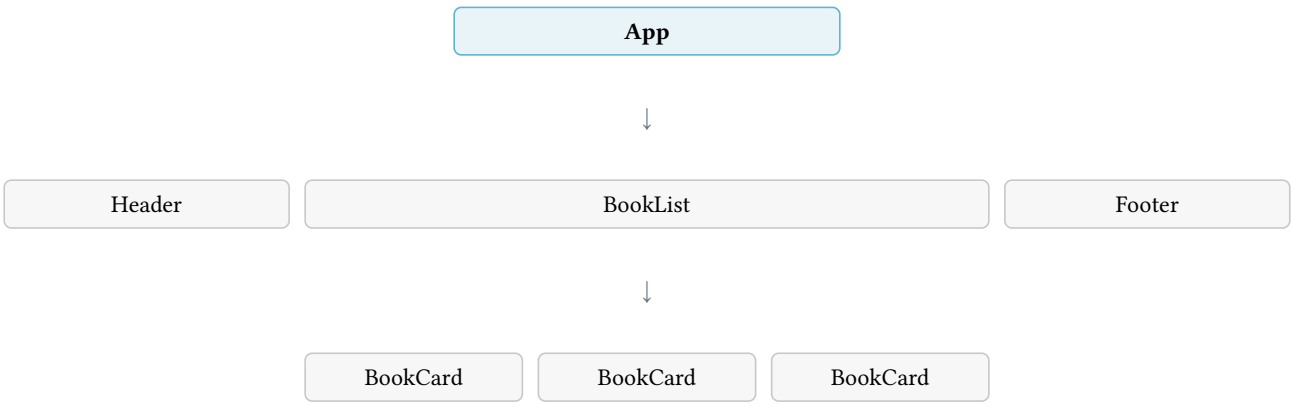


Figure 15: Shelfie's component tree. `App` is the parent of `Header`, `BookList`, and `Footer`, and `BookList` is the parent of the three `BookCard`s.

You'll build this tree at the end of the chapter. First, there are two rules that every component has to follow.

Components must be pure

In math, a formula like “area = side × side” always gives the same answer for the same side, and working it out doesn’t change anything else in the world. React expects the same from your components.

> **Jargon** A **pure** function gives the same result every time you call it with the same inputs, and changes nothing outside itself while it runs. Changing something outside (a variable declared elsewhere, the page, a file) is called a **side effect**.

React needs components to be pure because **you** don’t decide when a component renders. React does. It may render a component once, many times, or render it and throw the result away. If rendering has side effects, every one of those renders changes something, and the page ends up depending on how many times React happened to call your function.

To help you find impure components early, `<StrictMode>` (which Chapter 2 left on in `main.tsx`) makes React render every component **twice** while you’re developing. A pure component returns the same JSX both times, so you never notice. An impure one gives itself away. Strict Mode only does this during development: the version of your app that you publish renders each component once.

Here’s an impure component. It wants to number the tickets, so it keeps a counter in a variable declared outside the function, and adds one to it every time it renders:

Don't do this

```tsx
let ticketCount = 0;

export function Ticket() {
  ticketCount = ticketCount + 1;
  return <li>Ticket number {ticketCount}</li>;
}
```

```tsx
import { Ticket } from "./Ticket.tsx";

export function TicketList() {
  return (
    <ul>
      <Ticket />
      <Ticket />
      <Ticket />
    </ul>
  );
}
```

Three tickets, so you’d expect 1, 2, and 3. Here’s what the browser shows:

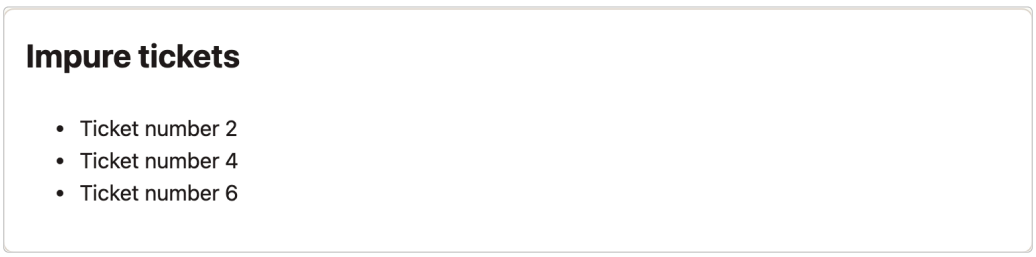


Figure 16: Three tickets, numbered 2, 4, and 6

> **Why did that happen?** Strict Mode rendered each `Ticket` twice. The first `Ticket` added one to the counter (1), then added one again when it was rendered a second time (2), and the page shows the 2. The second `Ticket` counted 3, then 4, and the third counted 5, then 6.
>
> Strict Mode didn't create this bug; it made it visible. Without Strict Mode, you'd see 1, 2, and 3 and think the code works. It would break the next time React rendered the list again, because `ticketCount` would keep counting from where it stopped. Exercise 3 shows that happening.

You don't have to wait for the browser to find this bug. The linter (`npm run lint`, and the squiggles in VS Code) reports it:

```
Ticket.tsx:4:3  error  Error: Cannot reassign variables declared outside of the
    component/hook

Variable `ticketCount` is declared outside of the component/hook. Reassigning this
    value during render is a form of side effect, which can cause unpredictable
    behavior depending on when the component happens to re-render. If this variable
    is used in rendering, use useState instead. Otherwise, consider updating it in an
    effect. (https://react.dev/reference/rules/components-and-hooks-must-be-pure#
    side-effects-must-run-outside-of-render).
```

The first line gives the file, line 4, column 3, and a summary: the component changes a variable declared outside it. The paragraph after it explains that this is a side effect, the term defined a few paragraphs earlier. It also mentions `useState`, React's way to give a component a memory. You'll meet it in Chapter 11.

So how do you number the tickets without a counter? Here, you can let the browser count. An ordered list, `<ol>`, numbers its items by itself, and the `Ticket` component doesn't need to know its number at all:

```tsx
export function Ticket() {
  return <li>Admit one</li>;
}
```

```tsx
import { Ticket } from "./Ticket.tsx";

export function TicketList() {
  return (
    <ol>
      <Ticket />
      <Ticket />
      <Ticket />
    </ol>
  );
}
```

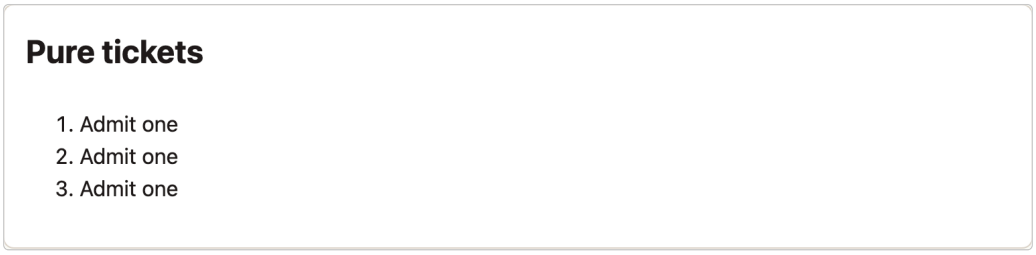


Figure 17: The pure version: the `<ol>` numbers the tickets 1, 2, and 3

`Ticket` now returns the same JSX every time and touches nothing else, so rendering it twice, or a hundred times, changes nothing. When a component really does need to show a value that differs from one use to the next, the parent passes that value in. That's what props are for, in Chapter 7.

Pure doesn't mean a component can't use variables. A component can create its own variables and change them while it runs, because they're created fresh on every render. What it must not change is anything that existed *before* it started rendering.

Never define a component inside another component

A component is a function, and JavaScript lets you declare a function inside another function. So it's tempting to tuck a small component inside the one that uses it:

> **Gotcha** Never declare a component inside another component. Declare every component at the top level of a file.

Here's what that mistake looks like:

Don't do this

```
export function ReadingNow() {
  function BookTitle() {
    return <h3>Pride and Prejudice</h3>;
  }

  return (
    <article>
      <BookTitle />
      <p>by Jane Austen</p>
    </article>
  );
}
```

This code type-checks, and the page looks right. The problem is invisible. Every time `ReadingNow` renders, the line `function BookTitle()` runs again and creates a **brand-new** function. To React, a new function is a new component, even if its code is the same. So on every render, React throws away everything `BookTitle` put on the page and builds it again from scratch. That's slower, and it also erases anything the component remembered. Once components have state (Chapter 11), that means text typed into a field vanishes, for no visible reason.

> **Why did that happen?** The linter catches this mistake, and reports it twice:
>
> ```
> ReadingNow.tsx:2:3  error  Do not nest component definitions inside other
>     components or props. Move it to the top level.  react-x/no-nested-component-
>     definitions
> ReadingNow.tsx:8:8  error  Error: Cannot create components during render
>
> Components created during render will reset their state each time they are
>     created. Declare components outside of render.
> ```
>
> The first message comes from one set of React lint rules and points at the inner function (line 2). The second comes from the React team's own rules and points at the place where it's used (line 8). Its explanation is the one you read above: the component's state is reset every time it's created.

The fix is to move the inner component to the top level. Since the book keeps one component per file, `BookTitle` gets its own file, and `ReadingNow` imports it:

```
export function BookTitle() {
  return <h3>Pride and Prejudice</h3>;
}
```

```
import { BookTitle } from "./BookTitle.tsx";

export function ReadingNow() {
  return (
    <article>
      <BookTitle />
      <p>by Jane Austen</p>
    </article>
  );
}
```

Now `BookTitle` is created once, when the file is loaded, and every render of `ReadingNow` uses the same component.

The component tree in React Developer Tools

In Chapter 2 you installed React Developer Tools in your browser. It adds a **Components** tab to the browser's developer tools (open them with F12, or Cmd+Option+I on macOS). The Elements tab shows the HTML on the page; the Components tab shows the component tree that produced it.

When you finish Shelfie at the end of this chapter, open that tab. You'll see the tree from Figure 15: `App` at the top, with `Header`, `BookList`, and `Footer` under it, and three `BookCard` entries under `BookList`. Click the small arrows to fold and unfold a component's children.

A few things to notice:

- HTML elements like `<main>` and `<ul>` aren't in the list. By default, the Components tab shows only your components, which makes a big app much easier to navigate. The HTML is still in the Elements tab.
- When you hover over a component in the tree, the browser highlights the part of the page it rendered. It's the quickest way to answer "which component drew this?" in an app you didn't write.
- When you select a component, the panel on the right shows details about it. There isn't much to see yet. It becomes useful once components receive props (Chapter 7) and have state (Chapter 11).

Firefox has the same extension, with the same Components tab. Chapter 24 shows more ways to use these tools when something goes wrong.

Shelfie: your first components

Time to turn the Chapter 2 starter into Shelfie. Here's everything you'll change:

Files you change in this chapter

changed	`index.html`
deleted	`src/App.tsx`
new	`src/components/App.tsx`
new	`src/components/BookCard.tsx`
new	`src/components/BookList.tsx`
new	`src/components/Footer.tsx`
new	`src/components/Header.tsx`
changed	`src/index.css`
changed	`src/main.tsx`

The steps below go from the smallest components up to `App`, so that every file you import already exists when you import it, and the page keeps working after every step. Keep `npm run dev` running while you work, and watch the page after each step.

The page title

Start with `index.html`. The browser tab still says "Hello, React", so give it the app's name:

```
<title>Shelfie</title>
```

A `components` folder

Create a folder named `components` inside `src`. All of Shelfie's components will live there, one file each. `main.tsx` and `index.css` stay directly in `src`: they set up the app, and they aren't components.

`BookCard`

The smallest piece is the card for one book. Create `src/components/BookCard.tsx`:

```
export function BookCard() {
  return (
    <article>
      <h2>Pride and Prejudice</h2>
      <p>by Jane Austen</p>
      <p>1813 · 432 pages</p>
    </article>
  );
}
```

The title, the author, and the details are typed straight into the JSX. That's called **hard-coded** content, and it's why all three cards will show the same book. Chapter 7 replaces it with data.

Two choices in this file are about HTML, not React:

- **`<article>`** is the element for a self-contained piece of content that makes sense on its own, like a blog post, a comment, or here, a book. Screen readers can announce it and let users jump from one article to the next.
- **`<h2>`** gives each card a heading. Chapter 5 explained that headings form the outline of the page, and that screen reader users often move around by heading. The page's `<h1>` is "My shelf", so each book is one level below it. An `<article>` should always have a heading, because the heading is what names it.

`BookList`

Next, the list. Create `src/components/BookList.tsx`:

```
import { BookCard } from "./BookCard.tsx";

export function BookList() {
  return (
    <ul>
      <li>
        <BookCard />
      </li>
      <li>
        <BookCard />
      </li>
      <li>
        <BookCard />
      </li>
    </ul>
  );
}
```

It imports `BookCard` from the file next to it (`./BookCard.tsx`) and uses it three times. Each card sits inside an `<li>`, and the items are inside a `<ul>`. Shelfie's books are a list of things, so they use the list elements. A screen

reader announces "list, 3 items" before reading them, which tells the user how much is coming. Nothing on the page is visible yet, because no component uses `BookList`. That comes with `App`.

Yes, this file repeats itself, three times. Keep it that way for now. Chapter 7 gives each card its own book, and Chapter 8 builds the list from an array instead of copying `<li>`s.

`Header` and `Footer`

Create `src/components/Header.tsx`:

```
export function Header() {
  return (
    <header>
      <p>Shelfie</p>
    </header>
  );
}
```

And `src/components/Footer.tsx`:

```
export function Footer() {
  return (
    <footer>
      <p>Shelfie keeps track of the books you read.</p>
    </footer>
  );
}
```

`<header>` and `<footer>` are the landmark elements from Chapter 5. At the top level of the page, they mark the banner area and the information area, and screen reader users can jump straight to either one.

The app's name in the header is a paragraph, not a heading. A page has one `<h1>`, and it should describe what's on *this* page: "My shelf". Once Shelfie has more pages (Chapter 20), each page gets its own `<h1>`, while the header stays the same on all of them.

`App`, in its new home

Now put the pieces together. Create `src/components/App.tsx` with the new `App`:

```
import { BookList } from "./BookList.tsx";
import { Footer } from "./Footer.tsx";
import { Header } from "./Header.tsx";

export function App() {
  return (
    <>
      <Header />
      <main>
        <h1>My shelf</h1>
        <BookList />
      </main>
      <Footer />
    </>
  );
}
```

`App` returns a Fragment (`<>…</>`, from Chapter 5), because the header, the main area, and the footer are siblings: none of them should be inside another, and a wrapper `<div>` would add nothing. Inside `<main>` go the page's

only `<h1>` and the book list. The result is the layout every page of Shelfie will keep: a `<header>`, a `<main>`, and a `<footer>`.

The imports are sorted by file name. The order doesn't change anything, but a sorted list is easier to scan.

Pointing `main.tsx` at the new `App`

The page hasn't changed yet, because `main.tsx` still imports the old `App` from `src/App.tsx`. Change that line in `src/main.tsx`:

```
import { App } from "./components/App.tsx";
```

The path now goes into the `components` folder, and the rest of `main.tsx` doesn't change. When you save, the page shows Shelfie.

Now delete the old `src/App.tsx`. It was the "Hello, React!" page from Chapter 2, and nothing imports it anymore. (If you'd rather move the old file into `components` and rewrite it, VS Code may offer to update the import in `main.tsx` for you. Either way, check that the line looks like the one above.)

The stylesheet

Last, `src/index.css`. The starter centered the `<main>` element and gave it padding. Now that the page also has a header and a footer outside `<main>`, only the middle of the page would be centered. Replace the file with this:

```
:root {
  font-family: system-ui, sans-serif;
  line-height: 1.5;
  color-scheme: light dark;
}

body {
  margin: 0;
  padding: 1rem;
}
```

The `main` rule is gone, and the padding moved to `body`, so the header, the main area, and the footer line up along the same left edge. That's all the styling Shelfie gets for now. Chapter 9 makes it look like a real app.

Checking your work

The page should now look like the screenshot at the start of the chapter (Figure 10). Then run the two checks from Chapter 2. Both should finish without printing any error:

```
npm run typecheck
npm run lint
```

If you're lost, `samples/shelfie/ch06`, this chapter's *checkpoint*, contains Shelfie as it should be now. You can compare your files with it, or copy it and continue from there.

Here's a summary of the HTML elements Shelfie uses so far, and why:

Element	Why it's there
`<header>`	The banner at the top of the page, with the app's name. Screen readers list it as a landmark.
`<main>`	The main content of the page. There's exactly one, and screen reader users can jump straight to it.
`<h1>`	The page's title, "My shelf". One per page.
`<ul>`, `<li>`	The books are a list of things, and the browser tells screen reader users how many items it has.
`<article>`	One self-contained book, named by its heading.
`<h2>`	The book's title, one level below the page's `<h1>`.
`<footer>`	The information at the bottom of the page. Also a landmark.

Table 2: The elements in Shelfie's first version

Carry forward

- A component is a function that returns JSX, with a name in PascalCase. You use it as a tag, like `<BookCard />`, and a lowercase tag is always an HTML element.
- Each component lives in its own file with the same name, is exported with a named export, and is imported with `import { Name } from "./Name.tsx"`.
- Split a page by drawing boxes: a piece that repeats or has one clear job becomes a component. The parent-child relationships form the component tree, which the Components tab of React Developer Tools shows you.
- Components are pure: same JSX every time, and nothing outside changes while they render. Strict Mode renders twice in development to expose the ones that aren't, and the linter catches many of them before that.
- Declare every component at the top level of a file, never inside another component.

Exercises

1. Add a component named `ShelfSummary` to Shelfie. It shows one paragraph: "You have 3 books on your shelf." Put it in its own file, and show it in `App` between the `<h1>` and the book list. Then remove the `export` keyword and predict what `npm run typecheck` says before you run it. (Remove the component when you're done: the Chapter 7 checkpoint doesn't have it.)

2. In Shelfie, rename the `BookCard` function to `bookCard`, and change the import and the three tags in `BookList.tsx` to match (keep the file name `BookCard.tsx`). Before you save, predict what `npm run typecheck` reports, what the page shows, and what the browser console says. Then change it back.

3. Copy the impure `Ticket.tsx` and `TicketList.tsx` from `samples/ch06-components/_nobuild/impure-tickets/src/` into Shelfie's `src/components` folder, and show `<TicketList />` in `App`. Note the numbers. Then remove `<StrictMode>` and its closing tag from `main.tsx` (and its import, which is now unused), and reload the page. Next, with the page still open, change `<ul>` to `<ol>` in `TicketList.tsx` and save. What numbers do you see after each step? Undo all the changes when you're done.

4. In Shelfie, move `BookCard` inside `BookList`: cut the `BookCard` function from `BookCard.tsx`, paste it at the top of the `BookList` function's body (without `export`), remove the import, and delete `BookCard.tsx`. Does the page change? Run `npm run typecheck` and `npm run lint`. Then undo the change.

5. On paper, draw the component tree of the demo project's `App` from this chapter, including the components that `WelcomeCard`, `TicketList`, and `ReadingNow` use. Then open the demo project, or your own copy of Shelfie, in the browser, and compare your drawing with the Components tab of React Developer Tools.

Expected observations

Predict first, then check. A different answer is worth understanding, not correcting away.

1. Run: `npm run dev`, then `npm run typecheck` without the `export`
   With the export, the paragraph appears under "My shelf", above the first card, and both checks pass. Without it, `typecheck` fails with the two errors from "One component per file": `TS2459` in `App.tsx` ("declares 'ShelfSummary' locally, but it is not exported") and `TS6133` in `ShelfSummary.tsx` ("declared but its value is never read"). The solution is `samples/ch06-components/demos/src/solutions/ShelfSummary.tsx`, plus `import`

`{ ShelfSummary } from "./ShelfSummary.tsx";` and `<ShelfSummary />` in `App`.
Takeaway: A new component is always the same three steps: a file with the same name, an `export`, and an `import` where it's used.

2. Run: `npm run typecheck`, then look at the page and the console
TypeScript reports `TS6133` on the import (`'bookCard'` is declared but never read) and `TS2339` three times, once per tag: `'bookCard'` doesn't exist on `JSX.IntrinsicElements`. The page still loads, with the header, the heading, and the footer, but the list shows three empty bullet points where the books were. The console shows two kinds of error: "`<bookCard />` is using incorrect casing. Use PascalCase for React components, or lowercase for HTML elements.", and "The tag `<bookCard>` is unrecognized in this browser."
Takeaway: One lowercase letter turns a component into an unknown HTML element. TypeScript catches it before the browser does, and React's warning tells you the fix.
3. With Strict Mode, the tickets are numbered 2, 4, and 6. Without it, a reload shows 1, 2, and 3. After saving `TicketList.tsx`, Vite updates the page without a reload, React renders the list again, and the tickets become 4, 5, and 6: the counter kept going from where it stopped. (The list items also show numbers now, because the list is an `<ol>`.)
Takeaway: Removing Strict Mode hides an impure component; it doesn't fix it. The component breaks the next time React renders it, whatever the reason.
4. Run: `npm run typecheck`, then `npm run lint`
The page looks the same, and `typecheck` passes. `lint` fails with four errors in `BookList.tsx`: one `react-x/no-nested-component-definitions` error on the inner function, and one `react-hooks/static-components` error ("Cannot create components during render") for each of the three `<BookCard />` tags.
Takeaway: Some bugs don't show on the page or in the type checker. That's why the book runs the linter too, and why its rules are on from the start.
5. The demo `App` has six children, in order: three `Greeting`s, then `WelcomeCard` (with one `Greeting` inside), `TicketList` (with three `Ticket`s), and `ReadingNow` (with one `BookTitle`). Shelfie's tree is the one in Figure 15. The Components tab shows the same trees, with no HTML elements in them: the `<section>`s and headings of the demo `App` aren't there.
Takeaway: The component tree follows the JSX: each component's children are the components it uses in its `return`.

Chapter 11 — State: Giving Components a Memory

In Chapter 10, Shelfie got its first buttons. Clicking "Remove" runs your code, but the page doesn't change. That's because, so far, a component can't *remember* anything: every time React calls it, it starts from its props and nothing else.

This chapter gives components a memory, called *state*. It's what turns a page that shows data into an app that responds to what you do. You'll learn how to create state, what React does when it changes, the rules that come with it, and the two errors every beginner meets. At the end, each book in Shelfie gets a "Finished" button that remembers whether you've read it:

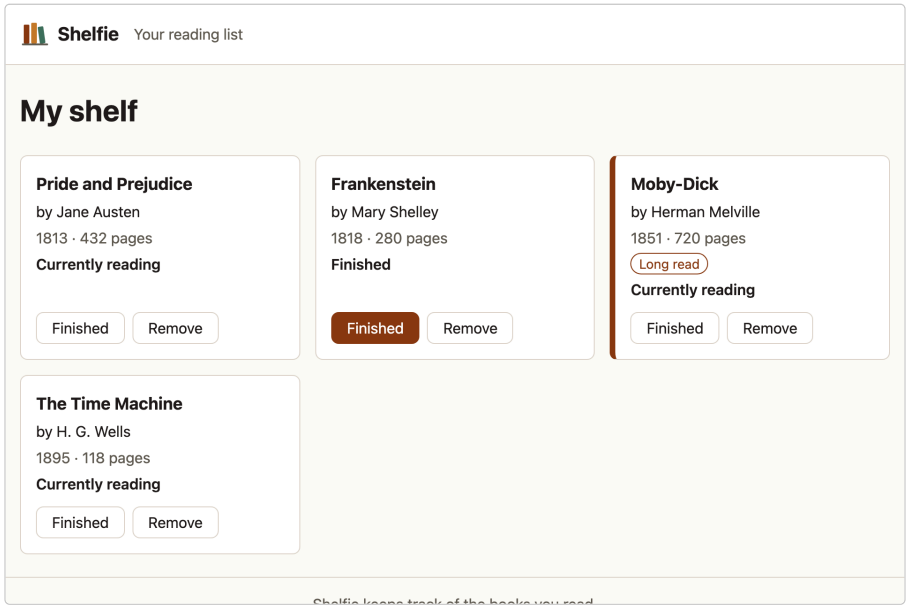


Figure 18: Shelfie after a click on *Frankenstein*'s "Finished" button: its status says "Finished", and the button is filled in

Why a normal variable isn't enough

A counter is the smallest program that needs a memory: a button, and a number that goes up by one on each click. With what you know from Chapters 3 and 10, you might keep the number in a `let` variable. This first attempt doesn't work, so it's in its own small app, apart from the chapter's demo project (`samples/ch11-state/demos`):

Don't do this

```
export function BrokenCounter() {
  let count = 0;

  function handleClick() {
    count = count + 1;
    console.log(`count is now ${count}`);
  }

  return (
    <p>
      <button type="button" onClick={handleClick}>
        Add one
      </button>{" "}
      Count: {count}
    </p>
  );
```

```
}
```

The handler adds one to `count`, then logs the new value. After three clicks, the console shows that the variable changed, but the page doesn't:

```
count is now 1
count is now 2
count is now 3
```



Figure 19: Three clicks later, the page still says "Count: 0"

Why did that happen? First, React doesn't know that anything happened. React puts JSX on the page when it calls your component, and changing a variable doesn't call the component again.

Second, even if React did call `BrokenCounter` again, it wouldn't help. Its first line is `let count = 0;`, so every call creates a brand-new `count`, starting at zero. A variable declared in a component lives for one call and no longer.

The linter sees the problem before you click anything. `npm run lint` reports it where the variable changes (line 5), and again where the button receives the function that changes it:

```
BrokenCounter.tsx:5:5  error  Error: Cannot reassign variable after render completes

Reassigning `count` after render has completed can cause inconsistent behavior on
    subsequent renders. Consider using state instead.
```

```
BrokenCounter.tsx:11:38  error  Error: Cannot modify local variables after render
    completes

This argument is a function which may reassign or mutate `count` after render, which
    can cause inconsistent behavior on subsequent renders. Consider using state
    instead.
```

"After render completes" means "after React has called your component and put the result on the page". The click comes long after that. So a component needs two things a plain variable can't give it: a value that survives from one call to the next, and a way to tell React "this changed, please update the page". React gives you both in one function.

`useState`: a value that React remembers

Jargon **State** is data that a component remembers between renders and that can change while the app runs, usually because the user did something. When state changes, React updates the page.

Here's the counter again, with state. Now each click adds one to the number on the page:

```tsx
import { useState } from "react";

export function Counter() {
```

```
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <p>
      <button type="button" onClick={handleClick}>
        Add one
      </button>{" "}
      Count: {count}
    </p>
  );
}
```

Read the new lines one by one:

- `import { useState } from "react";` imports the function from React.
- `useState(0)` asks React for a piece of state, and `0` is its **initial value**: the value it has the first time the component renders.
- `useState` returns an array of two items: the current value, and a function that changes it. `const [count, setCount] = ...` is the array destructuring from Chapter 3. Because it takes items by position, each component picks its own names.
- `setCount(count + 1)` asks React to change the state. It's the "please update the page" message that the plain variable couldn't send.

> **Jargon** The function that `useState` returns is the **set function**, or **setter**. By convention, it's the value's name with `set` in front: `count` and `setCount`, `isOpen` and `setIsOpen`. A **hook** is a function provided by React whose name starts with `use`, such as `useState`. Hooks let a component use React features. You'll write your own in Chapter 18.

The initial value is only used once. When you click, React stores `1`, and on the next render, `useState(0)` returns `1`: React ignores the argument once the state exists. That's the difference from `let count = 0;`, which starts over every time.

The type comes from the initial value

Hover over `count` in VS Code, and you'll see `const count: number`. TypeScript inferred it from `0`, as it does for `const total = 0;` (Chapter 4). `useState(false)` gives you a `boolean`, and `useState("")` a `string`. That's usually enough, except when the value is one of a few fixed choices, or when it can be missing.

A fixed set of values: `ReadingStatus`

Chapter 4 described a book's reading status with a union of literal types: `"reading" | "finished"`. Here's a component that keeps that status in state, and lets TypeScript infer the type:

Don't do this

```
import { useState } from "react";

type ReadingStatus = "reading" | "finished";

function statusLabel(status: ReadingStatus) {
  return status === "reading"
    ? "Currently reading"
    : "Finished";
```

```
}

export function StatusPicker() {
  const [status, setStatus] = useState("reading");

  return (
    <>
      <p>{statusLabel(status)}</p>
      <button type="button" onClick={() => setStatus("done")}>
        Finished
      </button>
    </>
  );
}
```

`npm run typecheck` rejects line 16, `<p>{statusLabel(status)}</p>`:

```
StatusPicker.tsx(16,23): error TS2345: Argument of type 'string' is not assignable to
    parameter of type 'ReadingStatus'.
```

Why did that happen? From the initial value `"reading"`, TypeScript infers the wider type, `string`, as it does for `let status = "reading";`. A `string` might be `"hello"`, so it can't be passed where only `"reading"` or `"finished"` is allowed. For the same reason, TypeScript says nothing about `setStatus("done")`, which sets a value that isn't a status.

The fix is to tell `useState` which type you want, between angle brackets. The demo's `StatusPicker.tsx` imports `ReadingStatus` from its `types.ts`, and declares its state like this:

```
const [status, setStatus] =
  useState<ReadingStatus>("reading");
```

It reads "a piece of state of type `ReadingStatus`, starting at `"reading"`". You saw angle brackets in `Promise<...>` (Chapter 3): the type between them fills in a blank in another type. The rest of the demo is two buttons, each with an inline arrow function, `onClick={() => setStatus("finished")}`, as in Chapter 10.

Jargon A **type argument** is a type that you pass to a function between angle brackets, like `ReadingStatus` in `useState<ReadingStatus>("reading")`. Chapter 18 shows how to write functions that accept one.

Now `setStatus` only accepts the two values. A button that calls `setStatus("done")` becomes an error:

```
StatusPicker.tsx(12,54): error TS2345: Argument of type '"done"' is not assignable to
    parameter of type 'SetStateAction<ReadingStatus>'.
```

`SetStateAction<ReadingStatus>` is what `setStatus` accepts: "a `ReadingStatus`, or a function that computes one" (the function part is coming later in this chapter). `"done"` is neither. And VS Code now completes the two valid values when you type `setStatus("`.

A value that can be missing: `Book | null`

Sometimes a piece of state has nothing in it at the start. This component shows the details of a book once you pick one. Before the first click, no book is selected, which Chapter 4 wrote as `null`:

```tsx
import { useState } from "react";
import { books } from "./books.ts";
import type { Book } from "./types.ts";

export function BookPicker() {
  const [selected, setSelected] = useState<Book | null>(null);

  return (
    <>
      <ul>
        {books.map((book) => (
          <li key={book.id}>
            <button
              type="button"
              onClick={() => setSelected(book)}
            >
              {book.title}
            </button>
          </li>
        ))}
      </ul>
      {selected === null ? (
        <p>Pick a book to see its details.</p>
      ) : (
        <p>
          {selected.title}, by {selected.author}, published in{" "}
          {selected.year}.
        </p>
      )}
    </>
  );
}
```

`useState(null)` alone would create state whose type is `null`, and *only* `null` (Exercise 3 shows the errors). `useState<Book | null>(null)` says what the state can really hold. TypeScript then makes you handle the missing case: after `selected === null` is ruled out, it narrows `selected` to `Book` (Chapter 4), so `selected.title` is allowed. Before any click, the page says "Pick a book to see its details."; after a click, it shows the book:

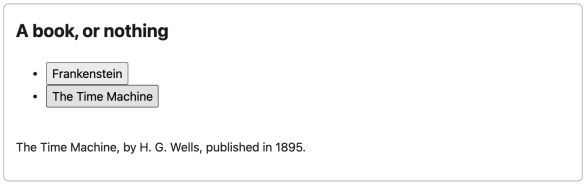


Figure 20: The state now holds a `Book`

Chapter 12 comes back to this demo, and asks what's best to keep in state: the whole book, or only its ID.

> **Type tip** You need a type argument when the initial value doesn't describe every value the state can have: a union of literals, `null` at the start, or an empty array (see "Why did that happen?" below). When the initial value says it all, as `0` or `false` do, let inference do its job.

What happens when state changes

Chapter 6 defined *rendering*: React calls your component to find out what to show. When you call a setter, React goes through three steps. Most surprises with state come from this sequence:

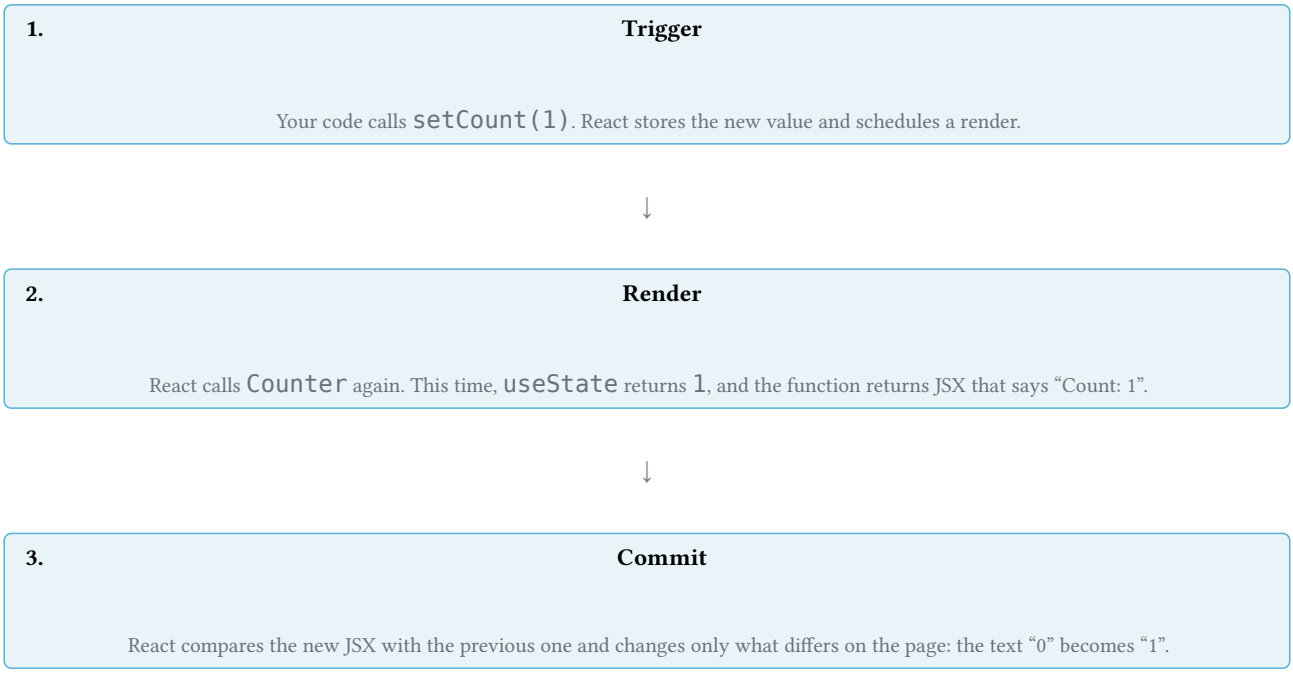


Figure 21: What React does after a setter is called. The same steps run for the first render, when the app starts.

> **Jargon** A **re-render** is any render after the first one. A state change **triggers** a re-render. When React puts the result on the page, it **commits** it.

You can watch it happen. The demo's `LoggedCounter` is `Counter` with two `console.log` calls, one in the body and one in the handler:

```
const [count, setCount] = useState(0);
console.log(`LoggedCounter renders with count = ${count}`);

function handleClick() {
  console.log("Click!");
  setCount(count + 1);
}
```

Here's the console when the page loads, then after two clicks:

```
LoggedCounter renders with count = 0
LoggedCounter renders with count = 0
Click!
LoggedCounter renders with count = 1
LoggedCounter renders with count = 1
Click!
LoggedCounter renders with count = 2
LoggedCounter renders with count = 2
```

Each click is followed by a render with the new count. Every render is logged twice because of Strict Mode (Chapter 6), which calls each component twice during development. A few things follow from the three steps:

- **The whole component function runs again**, from the first line to the `return`. Variables declared in it are computed again from the new state. That's why `isLongRead` in Shelfie's `BookCard` is always up to date without being state.
- **React only touches what changed.** In the counter, only the number's text changes. The button stays the same element, so it keeps the keyboard focus.
- **Rendering must stay pure** (Chapter 6), because a state change can make React render at any time. The `console.log` here is there to teach; don't leave logs in real components.

> **Try it** In the demo, open the console and click "Add one" in the first section, "A counter with state". Nothing is logged: a state change re-renders the component that owns the state (and the components inside it), not the whole app.

Each component has its own state

The demo's third section uses `<Counter />` twice. Here it is after two clicks on the *first* button:

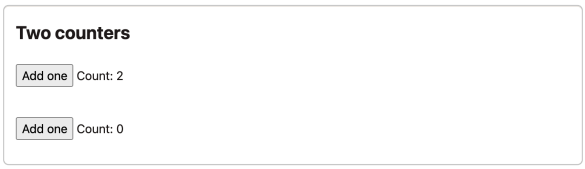


Figure 22: The first counter says 2, the second still says 0

> **Jargon** An **instance** of a component is one use of it on the page. `<Counter />` written twice creates two instances. They share the code, and React keeps a separate state for each, like two cooks following the same recipe, each with their own bowl.

That's what makes components reusable, and each Shelfie card will rely on it to remember its own reading status.

State is a snapshot

Here's a puzzle. This button's handler calls `setCount(count + 1)` three times, in a component that's otherwise the same as `Counter`. What does one click do?

Don't do this

```
function handleClick() {
  setCount(count + 1);
  setCount(count + 1);
  setCount(count + 1);
  console.log(`After three updates, count is ${count}`);
}
```

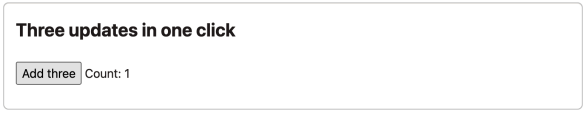


Figure 23: One click on "Add three" shows 1, not 3

The console, for that click:

```
After three updates, count is 0
```

Why did that happen? A setter doesn't change `count`. It asks React to use a new value *in the next render*. Inside the handler, `count` is a `const` created by the render the button belongs to, and it keeps that value: the log after the three calls shows 0. So all three lines run `setCount(0 + 1)`, the state ends up at 1, and React renders once, with that value.

Jargon State is a **snapshot**: during one render, and in every event handler created by that render, a state variable has one fixed value, like a photo of a scoreboard. A setter changes the value for the *next* render, not for the code that's running.

Updater functions

When the new value depends on the previous one, give the setter a *function* instead of a value. The demo's `PlusThree.tsx` does that:

```
function handleClick() {
  setCount((c) => c + 1);
  setCount((c) => c + 1);
  setCount((c) => c + 1);
}
```

Jargon An **updater function** is a function that you pass to a setter. It receives the latest state and returns the new state: `(c) => c + 1`.

React puts the updaters in a queue, and during the next render, it runs them in order, passing each one the result of the one before. So one click adds three:

Updater	It receives	It returns
First `(c) => c + 1`	0	1
Second `(c) => c + 1`	1	2
Third `(c) => c + 1`	2	3

The parameter's name is up to you: often the first letter of the state (`c`), or `prevCount`. When a click sets the state once, as in `Counter`, `setCount(count + 1)` works too, and you'll see it in a lot of React code. From here on, this book uses an updater whenever the new value is computed from the old one, as Shelfie's toggle does: the code keeps working if someone adds a second update later. Exercise 2 mixes both forms.

The Rules of Hooks

`useState` comes with rules that apply to every hook, including the ones you'll write in Chapter 18:

1. **Call hooks at the top level of your component.** Not inside an `if`, a loop, a nested function, or after an early `return`.
2. **Call hooks only from components or from your own hooks.** Not from an ordinary function, and not from an event handler.
3. **Give your own hooks a name that starts with `use`**, so that React's tools know to check these rules inside them.

Why? React never sees the names `count` and `setCount`. It remembers each component's state as a list: the first `useState` call gets the first item, the second call the second item, and so on. If a hook is skipped in one render and not in the next, every call after it gets the wrong item. This component breaks the first rule, by creating state only when the rating can be edited:

Don't do this

```
export function StarRating({ canEdit }: StarRatingProps) {
  if (canEdit) {
    const [stars, setStars] = useState(3);
    return (
      <button
        type="button"
        onClick={() => setStars((current) => current + 1)}
      >
        {stars} stars
      </button>
    );
  }

  return <p>Read only</p>;
```

It type-checks, but the linter's `react-hooks/rules-of-hooks` rule, which the book's configuration turns on, catches it:

```
StarRating.tsx:9:31  error  React Hook "useState" is called conditionally. React
    Hooks must be called in the exact same order in every component render. Did you
    accidentally call a React Hook after an early return?  react-hooks/rules-of-hooks
```

Line 9 is the `useState` call. "Called conditionally" means "only when a condition is true", and the message mentions the other way to make the same mistake: an early `return` before a hook. The fix is to call the hook first, whether or not the value will be used:

```
export function StarRating({ canEdit }: StarRatingProps) {
  const [stars, setStars] = useState(3);

  if (canEdit) {
    return (
      <button
        type="button"
        onClick={() => setStars((current) => current + 1)}
      >
        {stars} stars
      </button>
    );
  }

  return <p>Read only</p>;
```

Gotcha Never try to silence `rules-of-hooks` (the book's configuration doesn't allow it anyway). A hook in the wrong place causes bugs that are very hard to track down, such as one piece of state suddenly showing another one's value.

Why did that happen?

Two errors are so common with state that every React developer has met them.

"Too many re-renders"

Chapter 10 warned you: `onClick` needs a function, not a call. Here's what happens when the call is a setter:

Don't do this

```
const [count, setCount] = useState(0);

return (
  <p>
    <button type="button" onClick={setCount(count + 1)}>
      Add one
    </button>{" "}
```

TypeScript catches it first, with the error from Chapter 10: a setter returns nothing (`void`), so `onClick` receives nothing useful.

```
src/Counter.tsx(8,29): error TS2322: Type 'void' is not assignable to type
    'MouseEventHandler<HTMLButtonElement> | undefined'.
```

But Vite doesn't check types. If you open the page anyway, the counter never appears, and the console explains why:

```
[uncaught] Too many re-renders. React limits the number of renders to prevent an
    infinite loop.
[warning] An error occurred in the <Counter> component.

Consider adding an error boundary to your tree to customize error handling behavior.
Visit https://react.dev/link/error-boundaries to learn more about error boundaries.
```

Why did that happen? The braces in `onClick={setCount(count + 1)}` run their content *while the component renders*. So rendering `Counter` calls `setCount(1)`, which triggers a render, which calls `setCount` again, and so on, forever. React gives up after 25 renders in a row, rather than freeze your browser. The line that starts with `[uncaught]` (the browser shows it in red, as "Uncaught Error: …") means that nothing handled the error, so React shows nothing at all. The *error boundaries* in the warning are a way to show a friendly message instead; Shelfie gets them later in the book.

The fix is to pass a function, which React only calls on a click: a handler, as in `Counter.tsx`, or an arrow function, `onClick={() => setCount(count + 1)}`. As a rule, don't call a setter directly in a component's body. Setters belong in event handlers (and, from Chapter 15, in a few other places that run after rendering).

`never[]`: a list that starts empty

This component shows a list of suggested books after a click. The list is empty at the start, so the initial value is `[]`:

Don't do this

```
const suggestedTitles = ["Emma", "Dracula", "Middlemarch"];

export function Suggestions() {
  const [titles, setTitles] = useState([]);

  return (
    <>
      <button
        type="button"
        onClick={() => setTitles(suggestedTitles)}
```

```
      >
        Suggest books
      </button>
      <ul>
        {titles.map((title) => (
          <li key={title}>{title}</li>
        ))}
      </ul>
```

```
Suggestions.tsx(12,34): error TS2345: Argument of type 'string[]' is not assignable
    to parameter of type 'SetStateAction<never[]>'.
  Type 'string[]' is not assignable to type 'never[]'.
    Type 'string' is not assignable to type 'never'.
```

Why did that happen? `[]` contains nothing, so TypeScript can't learn from it what the list will hold. It gives the state the type `never[]` (hover over `titles` to see it). **`never`** is TypeScript's type with no values at all. You'll rarely write it, but in the type of state, it almost always means "the initial value was empty".

Read the error from the bottom up, as Chapter 4 showed: a `string` isn't a `never`, so a `string[]` isn't a `never[]`, and `setTitles` can't accept `suggestedTitles`.

The fix, in the demo's `Suggestions.tsx`, is a type argument that says what the list will contain:

```
const [titles, setTitles] = useState<string[]>([]);
```

It reads "a list of strings, empty at the start". The demo also shows "No suggestions yet." while the list is empty. The button replaces the whole list; adding or removing one item is the subject of Chapter 12.

Toggle buttons and `aria-pressed`

Shelfie's new button is a *toggle*: a button that's either on or off, like a light switch. Here's the smallest one, with a `boolean` state:

```
export function FavoriteButton() {
  const [isFavorite, setIsFavorite] = useState(false);

  return (
    <p>
      <button
        type="button"
        className={styles.button}
        aria-pressed={isFavorite}
        onClick={() => setIsFavorite((current) => !current)}
      >
        Favorite
      </button>{" "}
      {isFavorite ? "In your favorites" : "Not a favorite"}
    </p>
```

`(current) => !current` switches the value: `!` turns `true` into `false`, and `false` into `true`. The new part is `aria-pressed`.

> **Jargon** **`aria-pressed`** is an ARIA (Accessible Rich Internet Applications) attribute: one of the `aria-` attributes, made for screen readers. It turns a `<button>` into a **toggle button**. `aria-pressed="true"` means that it's switched on, and `"false"` that it's off. Screen readers announce the button as a toggle, and say whether it's pressed.

A sighted user sees the toggle's state from its style; `aria-pressed` gives the same information to everyone else. React writes the boolean as the text `"true"` or `"false"`. The style reads the same attribute, in `FavoriteButton.module.css`:

```
.button[aria-pressed="true"] {
  border-color: var(--color-accent);
  background-color: var(--color-accent);
  color: var(--color-surface);
}
```

`.button[aria-pressed="true"]` is a CSS attribute selector: it matches an element with the `button` class *and* `aria-pressed` equal to `"true"`. The file's other rule, `.button`, is the usual button style. Because the style and screen readers read the same attribute, they can never disagree.



Figure 24: The toggle, switched on: the button is filled with the accent color

Notice that the label stays "Favorite". Don't flip a toggle's label as well: "Not a favorite, pressed" contradicts itself for a screen reader user. Even without `aria-pressed`, a button that says "Finished" could mean "this book is finished" or "click to mark it as finished". A fixed label names the setting, the pressed style says whether it's on, and the text next to the button describes the state. Shelfie does the same.

> **Try it** In Chrome's developer tools, select the button in the Elements tab, then open the Accessibility pane. Click the button, and watch the pressed state change while the name stays "Favorite".

Shelfie: a reading status on each card

Time to give Shelfie's cards a memory. Here's everything you'll change:

Files you change in this chapter

changed	`src/components/BookCard.module.css`
changed	`src/components/BookCard.tsx`
changed	`src/components/Button.module.css`
changed	`src/types.ts`

The page keeps working after every step. First, open `src/types.ts` and add the union type from Chapter 4 above `Book`, exported, because `BookCard.tsx` will import it:

```
export type ReadingStatus = "reading" | "finished";
```

State in `BookCard`

Open `src/components/BookCard.tsx`. Import `useState` from `"react"`, and add `ReadingStatus` next to `Book` in the type import. Then add the state at the top of the component, and a new variable below `isLongRead`:

```
export function BookCard({ book, onRemove }: BookCardProps) {
```

```
  const [status, setStatus] =
    useState<ReadingStatus>("reading");

  const isLongRead = book.pages >= longReadPages;
  const isFinished = status === "finished";
```

The state comes first, as the Rules of Hooks want, and every card starts as "reading". `isFinished` is computed from the state on every render, like `isLongRead` from the props. It isn't state itself: storing it too would only give the two a chance to disagree.

Next, the click handler, next to `handleRemoveClick`. The new status is computed from the old one, so it uses an updater function:

```
function handleFinishedClick() {
  setStatus((current) =>
    current === "finished" ? "reading" : "finished",
  );
}
```

In the JSX, add a status line after the "Long read" badge, and the toggle at the start of the `actions` `<div>`, before "Remove":

```
<p className={styles.status}>
  {isFinished ? "Finished" : "Currently reading"}
</p>
<div className={styles.actions}>
  <button
    type="button"
    className={buttonStyles.button}
    aria-pressed={isFinished}
    onClick={handleFinishedClick}
  >
    Finished{" "}
    <span className="visually-hidden">{book.title}</span>
  </button>
```

The status line turns the state into words. The button follows the toggle pattern: `aria-pressed={isFinished}`, and a fixed label, "Finished", followed by the book's title in a visually hidden `<span>`, like "Remove" (Chapter 10). That gives each card's toggle a unique name, such as "Finished Frankenstein".

Save, and click a "Finished" button. The status line changes, but the button looks the same whether it's pressed or not. Two small CSS changes fix that:

- At the end of `src/components/Button.module.css`, add the `.button[aria-pressed="true"]` rule from the demo, unchanged. The accent background and surface-colored text have enough contrast in both themes.
- At the end of `src/components/BookCard.module.css`, add a `.status` rule with `font-weight: 600` ("semibold") to make the status line stand out. `vite-css-modules` adds `status` to the type of `styles`, so `styles.status` is checked like the other classes.

Checking your work

Click "Finished" on *Frankenstein*: the page should look like the screenshot at the start of the chapter (Figure 18). Click it again, and it goes back to "Currently reading". The toggle also works with the keyboard: Tab to it, then press Space or Enter. Then run `npm run typecheck` and `npm run lint`: neither should print an error. If you're lost, `samples/shelfie/ch11` contains Shelfie as it should be now.

In the Components tab of React Developer Tools, select a `BookCard`. As Chapter 6 promised, the right-hand panel now has a *hooks* section with the card's state, which changes from `"reading"` to `"finished"` when you click the card's toggle.

This version has limits. Every book starts as "Currently reading", the status is forgotten when you reload the page (state lives in the browser's memory), and "Remove" still only logs to the console, because the list of books isn't state yet. Chapter 12 moves the status into the book data and turns the list into state in `App`, so "Remove" finally removes. Chapter 15 makes Shelfie remember your books across reloads.

Carry forward

- A plain variable in a component starts over on every render, and changing it doesn't update the page. State does both: `useState` returns the current value and a setter, and calling the setter makes React render again.
- TypeScript infers the type of state from the initial value. Add a type argument when that value doesn't tell the whole story: `useState<ReadingStatus>("reading")`, `useState<Book | null>(null)`, `useState<string[]>([])`.
- A state change triggers a render, React calls the component again, then commits only what changed to the page. Each instance of a component has its own state.
- State is a snapshot: a setter changes the value for the next render, not for the code that's running. Use an updater function, `(c) => c + 1`, to update the same state several times in a row.
- Call hooks at the top level of components and custom hooks, never conditionally. Pass handlers to `onClick`, never a setter call. A toggle button keeps its label and exposes its state with `aria-pressed`.

Exercises

1. In the demo project, write a `CounterWithReset` component: a copy of `Counter` with a second button, "Reset", that sets the count back to 0. Show it twice in `App`. Click the first one's "Add one" a few times, then the second one's "Reset". What happens to the first count?
2. In the demo's `PlusThree.tsx`, replace the three lines of `handleClick` with `setCount(count + 1);` followed by `setCount((c) => c + 1);`. Predict what one click shows, starting from 0, then check. Then try `setCount((c) => c + 1);` followed by `setCount(count + 10);`. Put the original lines back when you're done.
3. In the demo's `BookPicker.tsx`, add a "Clear" button, shown only when a book is selected, that sets the state back to `null`. Then remove the type argument, so that the line reads `useState(null)`, and predict what `npm run typecheck` reports before you run it. Put the type argument back.
4. In the fixed `StarRating.tsx`, move the `if` above the `useState` line and turn it around, so that the component returns early: `if (!canEdit) { return <p>Read only</p>; }`, then the `useState` line, then the `return` of the button. The logic is the same. Predict what `npm run lint` says, check, and undo the change.
5. In Shelfie, click "Finished" on *Frankenstein*, then reload the page. What does the card say? Then, without the mouse, use Tab to reach *Frankenstein*'s "Finished" button, and press Space, then Enter. What happens after each key?

Expected observations

Predict first, then check. A different answer is worth understanding, not correcting away.

1. Run: `npm run dev`, then click the buttons
   The first counter keeps its count. "Reset" only changes the counter it belongs to, because each instance has its own state. The solution is `samples/ch11-state/demos/src/solutions/CounterWithReset.tsx`: a `handleResetClick` function that calls `setCount(0)`, and a second button that uses it.
   Takeaway: A setter only changes the state of the instance whose render created it.
2. The first version shows 2: `setCount(count + 1)` queues "replace with 1", and the updater then adds one to that. The second version shows 10: the updater queues "add one", but `setCount(count + 10)` then queues "replace with 10" (0 + 10, from the snapshot), and a replacement ignores what came before it.
   Takeaway: A value replaces the state; an updater computes from the latest one. React applies them in the order you called them.

3. Run: `npm run typecheck`
   With the "Clear" button, a click on a book shows its details and the button, and "Clear" brings back "Pick a book to see its details." The solution is `samples/ch11-state/demos/src/solutions/BookPickerWithClear.tsx`: the second branch of the ternary is a Fragment with the details and a `<button>` whose `onClick` calls `setSelected(null)`. Without the type argument, `typecheck` fails with several errors: `TS2345` on `setSelected(book)` ("`Argument of type 'Book' is not assignable to parameter of type 'SetStateAction<null>'`"), `TS2339` three times ("Property 'title' does not exist on type 'never'", then `author` and `year`), and `TS6133`, because the `Book` import is no longer used.
   Takeaway: *`useState(null)` creates state that can only ever be `null`. After `selected === null` is ruled out, nothing is left, which is what `never` means.*
4. Run: `npm run lint`
   The same `react-hooks/rules-of-hooks` error as in the chapter, now on the `useState` line after the `if`: "React Hook "useState" is called conditionally. … Did you accidentally call a React Hook after an early return?" When `canEdit` is `false`, the component returns before reaching the hook, so the hook isn't called on every render. The fixed version, `samples/ch11-state/demos/src/demos/StarRating.tsx`, calls the hook before any `return`.
   Takeaway: *An early `return` before a hook is a condition too. Put hooks first, then decide what to return.*
5. After the reload, *Frankenstein* says "Currently reading" again, and its button isn't filled in: state lives in memory, and a reload starts from the initial value. With the keyboard, the focus outline shows on the button. Space switches it on (the status says "Finished"), and Enter switches it off again. Remove still only logs a message to the console.
   Takeaway: *A `<button>` handles the keyboard for free, and state lasts only as long as the page is open.*

End of preview

The full edition continues with the other 21 chapters and 6 appendices.

meziantou.net

React from Zero

Build Your First Real Web Apps with TypeScript

You know a little HTML, a little CSS, and some JavaScript, and you want to build real, interactive web apps. This book takes you from your first component to a finished, tested, deployed application — one small, working step at a time. Most chapters add a feature to the same app, explain why it works, and show the mistakes beginners actually make, with the real error messages and how to fix them. The code is written the way professionals write it today: TypeScript in strict mode, modern React, accessible HTML, and tests.

What you'll learn

- Just enough JavaScript and TypeScript
- Forms that everyone can use
- Several pages with React Router
- Testing with Vitest and Playwright
- Components, props, and state
- Fetching data from an API
- Context, reducers, and custom hooks
- Building and deploying your app

About the Author

Gérald Barré — known online as **meziantou** — is a software developer with more than ten years of industry experience and a recognized **Microsoft MVP**.

He builds web applications and the tooling around them, and has spent a large part of his career helping developers move from "it works on my machine" to code that is typed, tested, accessible, and shipped with confidence. He maintains a number of open-source projects, including **Meziantou.Analyzer** and **Meziantou.Framework**.

Gérald writes regularly at **meziantou.net**, where he has published hundreds of articles about web development, .NET, and developer tooling.

Gérald Barré
Microsoft MVP · meziantou.net

MVP Microsoft Most Valuable Professional