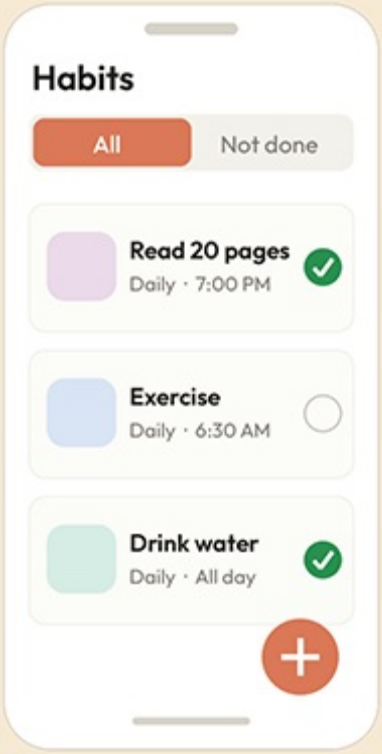
«packt»

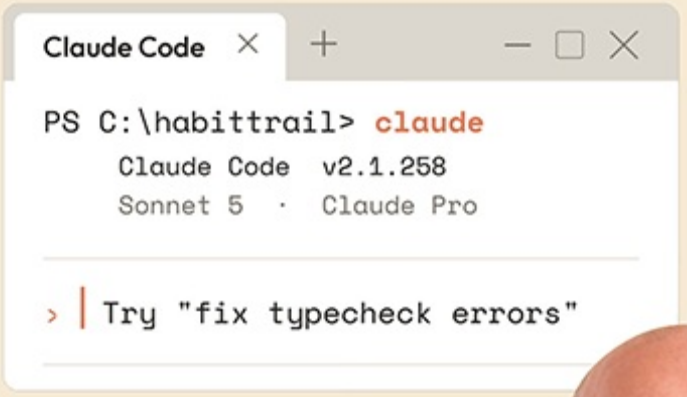
First Edition

Claude Code

Visual Bible

A step-by-step guide to creating and publishing websites and mobile apps with AI







Mark J. Price

Claude Code Visual Bible

A step-by-step guide to creating and publishing websites and mobile apps with AI

Mark J. Price



Claude Code Visual Bible

Copyright © 2026 Packt Publishing

All rights reserved . No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Ashwin Nair

Relationship Lead: Nitin Nainani

Project Manager : Ruvika Rao

Content Engineer : Hayden Edwards

Technical Editor: Arjun Varma

Copy Editor: Hayden Edwards

Production Designer: Ajay Patule

Growth Lead : Sayantani Saha

First published: October 2026

Production reference: 1091026

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80865-367-4

www.packtpub.com

Contributors

About the author

Mark J. Price helps beginners and everyday knowledge workers get real, practical value out of AI tools like ChatGPT, Claude, Copilot, Siri AI, and Gemini. Based in Brighton, England, he brings two decades of experience as a technical author, including ten editions of a bestselling C# and .NET series for Packt, to making AI approachable for readers with no technical background. A former maths teacher and Microsoft Certified Trainer, he comes from a family of teachers and now chats with readers in Packt's Discord.

> *Thank you to all my readers. Your support means I get to write these books and celebrate your successes. Special thanks to the readers who give me actionable feedback via my GitHub repository and email, and interact with me and the book communities on Discord. You help make my books even better with every edition.*

Get in touch

Feedback from our readers is always welcome.

General feedback : If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata : Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata` , click **Submit Errata** , and fill in the form.

Piracy : If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author : If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/` .

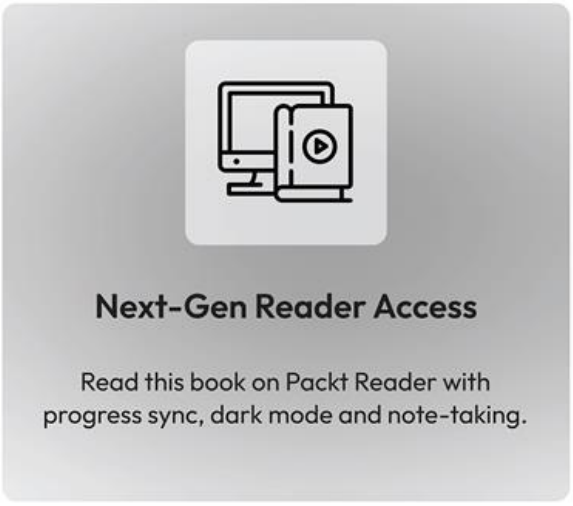
Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the *" How to Unlock "* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:







How to Unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require one

Introduction

Welcome to *Claude Code Visual Bible* !

In this introduction, you will learn:

- Why you picked up this volume
- Who this book is for
- A flesh-and-blood author writing for a respected publisher
- How this volume is structured
- How to read the three books
- Get the sample files
- A note about the figures in this book
- How this book handles a fast-changing Claude Code
- Key terms to help you follow along
- What prerequisites do you need?

Why you picked up this volume

By now, you've probably used Claude, ChatGPT, or another AI for something. Maybe you've asked it a few questions, had it rewrite an email, or turned to it when you got stuck on an idea. If you've already read my companion book, *Claude Visual Bible* , you've gone further: you've used Claude to research a topic, draft and edit documents, analyze a spreadsheet, or automate a procedure you used to do by hand using a scheduled task or reusable skill.

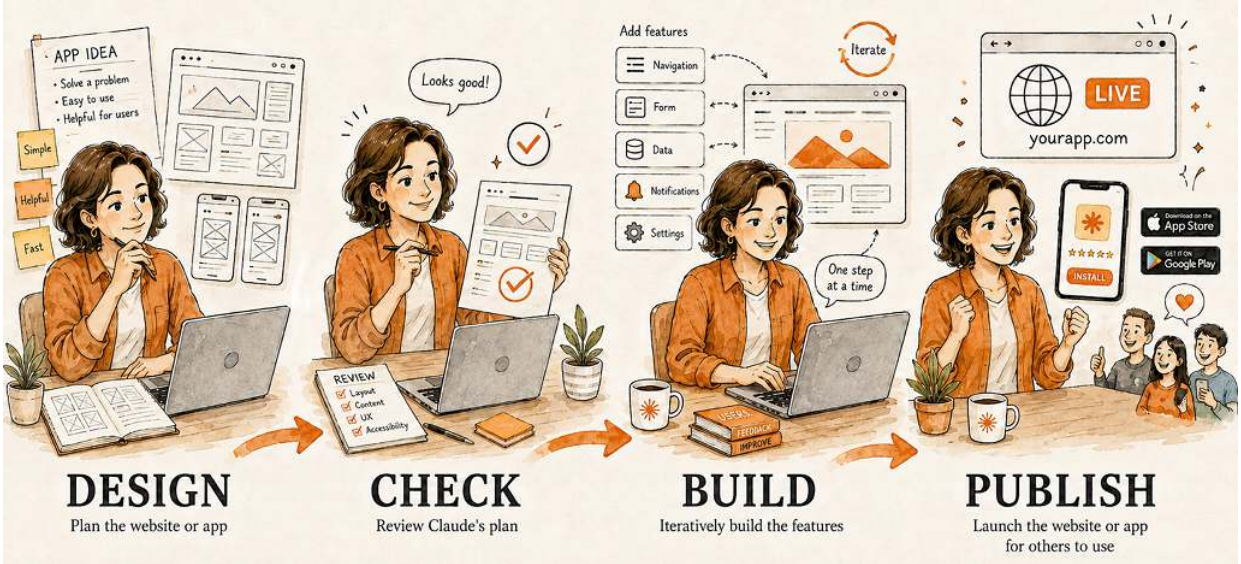
Either way, you've probably still only used Claude through conversation, typing a request and reading back an answer. You haven't yet used it to build something interactive that you can publish or ship: a website visitors can open, or a mobile app someone can install on their phone.

Closing the gap between using Claude Chat or Claude Cowork to improve your work and personal life, and using Claude Code to build websites and mobile apps, is why I wrote this volume. You do not need a technical background to close the gap. You need a clear path from your website and mobile app ideas to confident, professional building, without having to learn the nitty-gritty details of programming languages like JavaScript, Swift, or Kotlin. Claude Code can handle that code while you manage the high-level goals.

Most guides to Claude Code go too far into explaining all the technical underpinnings because they are written by people who love those details. Some try to cover little bits of everything, in one slim volume, so each subject gets miserly treatment. Others cover one big project end-to-end, but it is rarely what you want to learn how to build.

None of these approaches match how people successfully build websites and mobile apps with AI. Here's the path that works, one step at a time: design the website or app, produce a checked piece of work, iteratively build the features, then publish the website publicly or deploy the mobile app to an app store for others to download and enjoy.

Claude Code Visual Bible is different because it follows that better path, one step at a time:



Introduction Figure 1: Better path to building websites and mobile apps with AI

Each chapter has practical hands-on tasks that come with one or more prompts you can copy straight into Claude Code. You're not left to guess how to phrase a request or peer at tiny text in a poorly cropped screenshot. You get a working prompt as a starting point, then a clear explanation of which parts to tweak for your own website or mobile app, your own job, and your own goals.

Who this book is for

This book is for you if you have the curiosity to build your own mobile apps and websites, and the everyday computer skills most people already have, such as using a web browser, installing an app, and working with documents, spreadsheets, and images.

You will get the most from this book if you recognize yourself in one or more of these situations:

- You want to progress from casual Claude Chat and professional Claude Cowork use to building useful websites and mobile apps with Claude Code
- You want to build a habit of checking AI output instead of trusting it automatically
- You want to use source code control to provide an "undo" after Claude Code makes changes to your project files
- You are looking for a path from knowledge work with Claude to building a completed interactive app or website
- You are a novice developer who already knows a little HTML, CSS, or JavaScript, or has dabbled with mobile app development

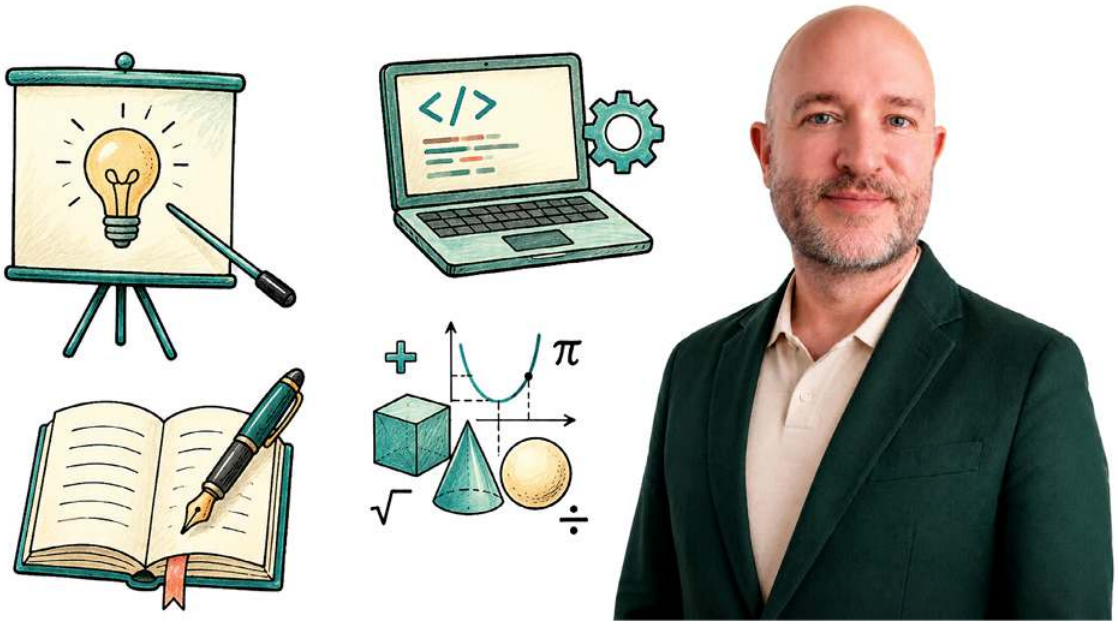
A flesh-and-blood author writing for a respected publisher

I am a living, breathing human based in Brighton, on England's south coast. Close enough to London for exciting cultural weekends, breezy enough to survive the increasingly sweltering British summers.

I've written more than 20 programming titles for my publisher Packt since 2016, including ten editions of my best-selling C# and .NET book. I update it every year in November and I plan to do frequent updates for *Claude Code Visual Bible* too since Claude Code is improving so rapidly.

Teaching is in my blood. My mother was a maths teacher, my sister a primary school teacher, my grandfather taught at the Royal Military Academy Sandhurst, and teaching and military careers run through my uncles and cousins too. I taught GCSE and A-Level maths in two London secondary schools, and I hold a Computer Science BSc (Hons) from the University of Bristol.

That mix, maths teacher and software developer, turns out to be exactly what a beginner's book about an AI coding assistant needs. Teaching maths to teenagers taught me how to break an intimidating idea into the smallest step that still makes sense, and how to spot the exact place a student gets stuck before they even ask. Software development taught me precise problem-solving: to test whether something actually works rather than assume it does, and to write instructions a computer, or a nervous beginner, can follow without guessing. Put those together and you get a guide that walks you from confusion to confidence one checked step at a time, the way a good maths teacher walks you from "I don't get it" to "oh, that's actually simple."



Introduction Figure 2: Me and my background

There's another reason my software developer background matters here, beyond teaching precision: programmers are the canary in the coal mine for what AI does to a profession. I've spent the last few years watching AI move from autocomplete to something that can draft, test, and ship working code. That's a transition I lived through in my own daily work, which is exactly why I can show you, concretely, what to watch for as the same pattern reaches and transforms your profession.

Writing books for a decade took my face-to-face classroom teaching instinct further: it's how I now reach far more people, in far more places, than any classroom could, and I love hearing from readers all over the world. It's my way of making a dent in the universe, as Steve Jobs mused he wanted to do.

My involvement didn't end when I delivered the final drafts to my publisher. I am active in Packt's Discord channels, where you can

chat with me and other readers about my books and share tips and tricks on how to get the most out of AI.

You can learn more about me and confirm my credentials at the following links:

- GitHub repositories: https://github.com/markjprice/
- LinkedIn: https://www.linkedin.com/in/markjamesprice/
- Amazon: https://www.amazon.com/stores/Mark-J.-Price/author/B071DW3QGN
- Packt: https://subscription.packtpub.com/search?query=mark+j.+price
- Discord channel: https://discord.com/channels/963435343232696621546416952715776030

How this volume is structured

This single volume contains three books. Each one covers a stage of that path:



Introduction Figure 3: Three books, one destination: building solutions

The following table explains what you will learn in each of the three books:

Book title	What you will learn
Book 1 – *Build Tools with Claude Code*	Set up a safe workspace in VS Code, turn an idea into a build brief, give Claude Code instructions it can act on, review its plans and changes, save checkpoints with Git, and debug one problem at a time. You finish by building a small command-line tool.
Book 2 – *Build and Publish Websites*	Structure a page with HTML, design it with CSS, make it responsive, add navigation and behavior with JavaScript, build a working contact form, check accessibility and speed, and publish the site so anyone can visit it.
Book 3 – *Build and Ship Mobile Apps*	Build an iOS and Android app with React Native and Expo: create screens and navigation, capture and store data on the device, add reminders and sharing, test it on a real phone, and prepare it for beta testing and app-store submission.

Introduction Table 1

Each of the three books in this volume contains a series of concise chapters and ends with a practical hands-on project, so you finish every book holding a completed piece of work, not just a lot of theory swimming around in your head.



Watch out : Unfortunately, none of Claude Code is available on the **Free** plan. Your plan must be **Pro** or higher, which covers everything you'll need: the Claude Code for VS Code extension, as well as the Claude desktop app and Claude Code CLI.

How to read the three books

If you are new to Claude Code, then read the three books in order. Each one builds on the skills from the one before it. You cannot create a cross-platform mobile app using Expo in *Book 3* until you can build a website in *Book 2*, and you cannot build a website until you can build a simple command-line tool and use products like Git to manage your source code, which is what *Book 1* teaches. Don't worry when you read unfamiliar words like Expo and Git; there is a glossary later in this *Introduction* to explain what those terms and others mean.

Inside each chapter, look for several types of recurring boxes:

- **Watch out** boxes warn you about a specific mistake or risk
- **Good practice** boxes give you a habit that improves your results
- **What to ask next** boxes give you one optional follow-up prompt you can send right after finishing the chapter's task

- **Real-world example** boxes give you an example of a real-world need solved by real people using AI

Two more boxes point you outside the book:

- **Learn more online** boxes link to an official resource, such as documentation from the AI provider, or extra material like bonus books. Interfaces and features change often. When a detail is likely to change after you buy this book, I give you a link to a verified source instead of guessing.
- **Sample files** boxes link to a file or folder in this book's online repository. Download these files when a chapter's activity needs them, if you don't want to use your own. You can open them, upload them into Claude Code, or copy from them, depending on what the chapter asks you to do.

You don't need to use every box. Read the ones that help with what you're doing right now.

Most chapters end with a **Try it now** section that gives you one task you can finish in five to fifteen minutes. The three end-of-book project chapters need an hour or more. The chapters also give you a few questions to check what Claude Code produced, so reviewing its work becomes a habit rather than an afterthought.

Get the sample files

Other Claude Code guides expect the reader to supply their own files to work with while learning skills. *Claude Code Visual Bible* provides sample files, so you don't have to scramble to think of or create your

own. These files are stored in a repository in my GitHub account so I can keep them updated.

Don't worry! Although GitHub is designed for professional software developers, anyone can use it. For example, you can download all the folders and files in one go using the following link:

`https://github.com/markjprice/claude-code-vb/archive/refs/heads/main.zip`

Once the ZIP file is downloaded, extract its contents to your local computer folder.

A note about the figures in this book

The word *Visual* in *Claude Code Visual Bible* is a promise. Every screenshot and diagram does one specific job: it helps you decide what to do, shows you what Claude Code is doing, walks you through an action, or helps you judge whether a result is good enough to use.

- **Screenshots** are cropped to the part of the screen that matters and enlarged so the interface text stays readable on a printed page. Where a screenshot shows several controls, I number them and explain each one in order.
- **Diagrams** are reserved for ideas that are hard to explain in a sentence, such as how a project holds context, or how a process moves from a trigger to a finished result.

I never place essential information only inside an image. When a figure matters to a task, the surrounding text explains it too, so the instructions still work even if a picture is hard to see.

One of my favorite benefits offered by my publisher is that for every purchase, Packt provides free PDF and online web versions. You can unlock your free benefits by turning to the *Free benefits with your book* section in the book's front matter. This will allow you to zoom into full-color screenshots if that helps.

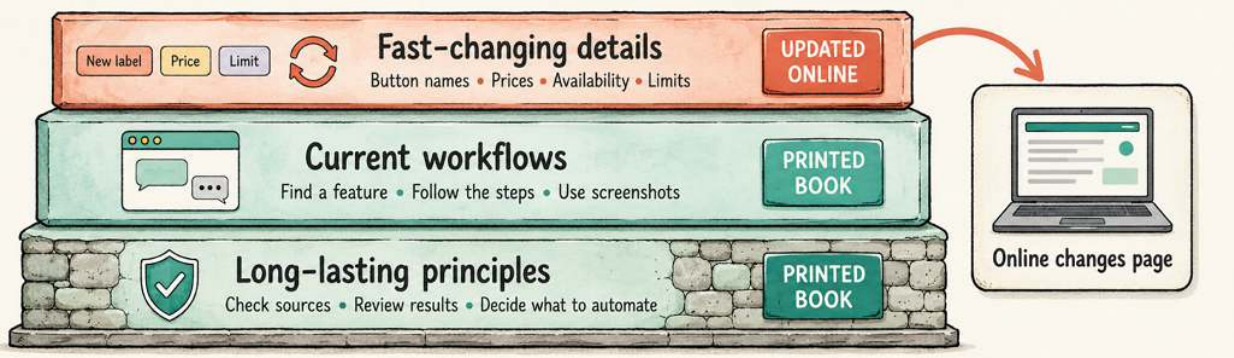
How this book handles a fast-changing Claude Code

Claude Code's menus, buttons, and features change often. Books built only around today's exact user interface would be out of date within months.

So, the books teach decisions and tasks that stay useful after the interface moves: how to judge whether code does what it's supposed to, how to decide whether a user interface is easy for a user to understand, how to tell when a procedure is stable enough to automate. These long-lasting principles will serve you well while Claude Code evolves rapidly over the coming months and years.

Screenshots illustrate these ideas using the interface as it exists today, and each set of instructions combines a stable landmark, such as a menu's purpose, with the exact label currently shown on screen. If a label changes, the landmark still points you to the right place.

Where a detail is likely to move, such as plan prices or a settings screen, I have placed it on a companion page you can check online instead of printing it here. I will keep that page updated after the book is printed, so the printed pages can stay focused on what to decide and why, not on details a future software update could make wrong.



Introduction Figure 4: A solid foundation, current screenshots, with fast-changing details updated in an online changes page in the book's GitHub repository

Visit https://github.com/markjprice/claude-code-vb/blob/main/changes.md for the current list of changes. I organize it by chapter number, so you can jump straight to the section that affects the page you are reading. This page shows:

- **Interface changes** : A new menu name, a moved button, or a renamed setting, matched to the chapter and step where the old version appears in this book.
- **Plan and pricing changes** : Current plan names, prices, and usage limits, since these move more often than the underlying features.
- **Feature availability** : Notes on when a feature described as a preview becomes generally available, or when a feature is removed.
- **Region and account differences** : Cases where a feature works differently, or not at all, depending on your country, device, or account type.
- **Corrections** : Any error identified in the book after printing.

Key terms to help you follow along

This glossary covers the terms used most often throughout the books:

Term	Definition
Agent	A program that can complete a multi-step task on its own, using tools such as reading a file, editing it, or running a command, instead of only replying with text. Claude Code works as an agent inside your local computer's project folder.
API	Short for application programming interface. A defined way for one piece of software to ask another piece of software for something, such as your app asking a weather service for today's forecast.
Build brief	A short document that describes the user and their problem, the smallest useful version to build, a few acceptance criteria, and a few non-goals.
Claude Chat	The Claude mode built for conversations: questions, discussion, and shorter, focused work.
Claude Code	The Claude mode built for software development: writing code, running tests, and working with Git repositories.
Claude Cowork	The Claude mode built for longer work: research, connected files and apps, and finished documents,

	spreadsheets, presentations, or reports.
CLI	Short for command-line interface. A way to use a program by typing commands instead of clicking buttons. The Claude Code CLI is the version of Claude Code you run this way, from a terminal or command line.
Commit	A saved checkpoint in Git. It records the current state of your files, along with a short message describing what changed.
Component	A reusable, self-contained piece of an interface, such as a button or a card, that you can use again in different parts of a website or app.
Context	Information you give Claude, through your message, an uploaded file, or an earlier part of the conversation, that affects its response.
CSS	Short for Cascading Style Sheets. The language used to control how a webpage looks by setting colors, layout, fonts, and spacing. Files have the `.css` file extension.
Deploy / **Publish**	To make a finished website or app available for other people to actually use: publishing a website means putting it online at a public web address

	anyone can visit, while deploying a mobile app means submitting it to an app store (or sharing a test build through TestFlight or Google Play's internal testing) so others can install it on their own phone.
Diff	A side-by-side view of exactly what changed between two versions of a file, usually shown with removed lines in red and added lines in green.
Expo	A framework built on top of React Native that handles most of the setup a mobile app normally needs. *Book 3* uses Expo, together with Expo Router and Expo Go, to build, test, install, and run the mobile app you're building directly on your own iPhone or Android phone while you develop it, without needing to submit it to an app store first.
Extension	A small add-on you install inside VS Code to give it a new feature, such as the Claude Code extension, which lets you use Claude Code directly beside a code editor.
Flexbox / Grid	Two CSS tools for arranging elements on a page: Flexbox arranges items in a single row or column; Grid arranges items into rows and columns.
Git	A tool that tracks changes to your files over time, so you can save checkpoints of your work and return

	to an earlier version if something breaks.
GitHub	A website for storing and sharing project files and their version history using Git. It is used here to host this book's sample files and store extra materials like a list of corrections and bonus book content.
HTML	Short for HyperText Markup Language. The language used to structure the content of a webpage, such as headings, paragraphs, images, and links. Semantic HTML means using HTML elements that describe what a region of a page is for, such as `<header>`, `<nav>`, `<main>`, and `<footer>`, instead of generic containers for everything. Files have the `.html` or `.htm` file extension.
JavaScript	A programming language that adds interactivity and behavior to a website, such as responding to clicks or validating a form. Claude Code writes this code for you. Files have the `.js` file extension.
Markdown	A simple text format that lets you add headings, bold words, and lists to a plain text file with ordinary keyboard characters. Files have the `.md` file extension. Claude Code and other coding assistants use Markdown files extensively.
Package	A piece of code someone else has written that you

	add to your project so you don't have to write it yourself, such as a tool for handling dates or validating a form.
Permission	Claude Code's built-in check before it does something that could affect your files or system, such as running a command, so you can review it first.
Persistence	Saving data somewhere, such as the phone's own file storage, that survives the app closing, so it's there again next time it opens.
Project	The folder of files that make up what you're building: your code, assets, and configuration. Claude Code reads and edits files inside your current project.
Prompt	The message you send Claude: what you want it to do, and what a good answer should include.
React Native	A framework for building mobile apps for both iOS and Android from one shared codebase. *Book 3* uses React Native together with Expo to build and test the app.
Repository (aka repo)	A folder that Git is tracking. It holds your project's files together with their full history of changes.

Responsive design	A design that rearranges its layout to fit the visitor's screen width, from a narrow phone to a wide desktop, rather than simply shrinking.
Terminal (aka command line)	The window where you type and run commands, such as moving between folders or starting Claude Code.
TypeScript	JavaScript with an added layer that catches certain mistakes before your code even runs, such as treating a number like a word. Files have the `.ts` or `.tsx` file extension.
Validation	Checking that what someone entered in a form makes sense, such as a required field or a genuine-looking email address, before it's submitted.
VS Code	Short for Visual Studio Code. A free, popular code editor from Microsoft. This book uses it together with the Claude Code extension to edit code and configuration files.

Introduction Table 2

What prerequisites do you need?

You need very little to start *Book 1* :

- Claude Code works on Windows, macOS, and Linux computers. Building a website or a mobile app needs a full computer with a code editor and a terminal. You cannot use a phone or a tablet. This book primarily uses the Claude Code extension for VS Code, although you will also be introduced to the Claude desktop app and the Claude Code CLI that provide alternative ways of working.
- A Claude Pro subscription, or higher, is required for all Claude Code features.
- An iPhone or Android phone. You'll use the hardware device in *Book 3* to install and test the mobile app you build.
- Everyday comfort with your computer: installing an application, and finding and organizing files and folders. A willingness to read short pieces of code, try things out, and investigate the occasional error message. You do not need any programming background beyond that. Claude Code handles the code; you direct it and check its work.

A few more things will help before you turn the page:

- *Claude Code Visual Bible* covers Claude Code. It does not teach Chat or Cowork. If your goal is to do research, have conversations, create documents, analyze spreadsheets, define repeatable skills, or automate recurring tasks, look for the companion title, *Claude Visual Bible* .
- Treat every Claude response in this book as a draft to review, not a finished answer to trust automatically. Each chapter shows you what to check. A wrong answer stated with confidence can still cost you time, money, or credibility, so this habit is worth building early and often.

- Keep a notebook, document, or note app open as you read. Several chapters ask you to try a short task, and you will want somewhere to keep what you produce for the project at the end of each book.



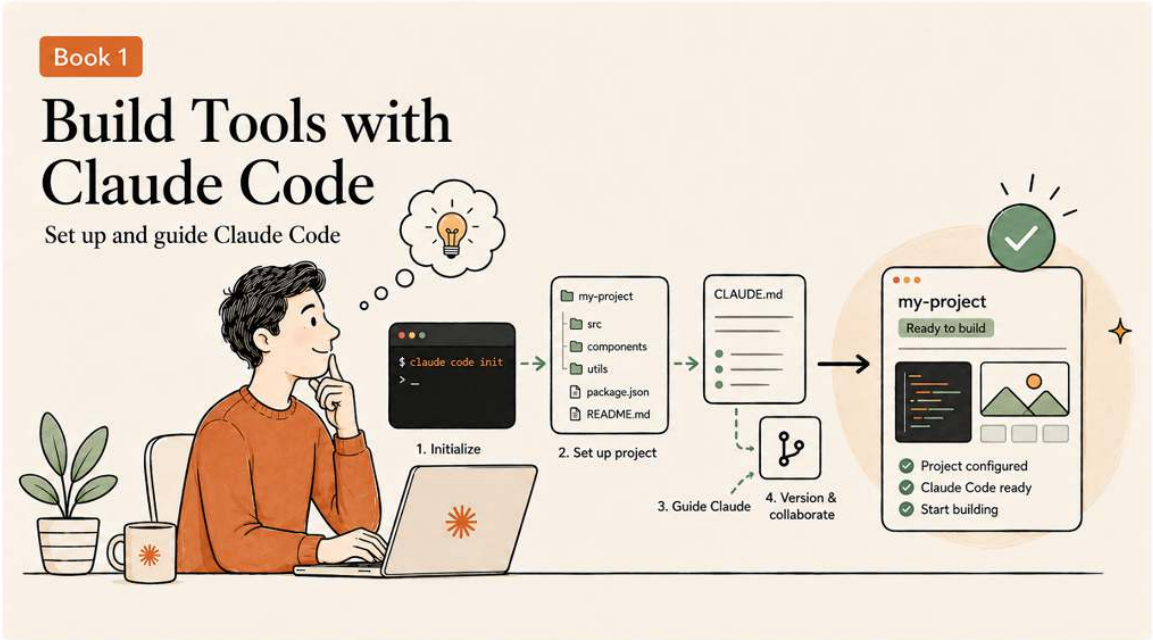
Introduction Figure 5: You supply the ideas, taste, and judgment; Claude supplies the coding knowledge and implementation skills

With that, you are ready to open *Book 1* and learn how to build tools with Claude Code.

Book 1

Build Tools with Claude Code

In *Book 1*, you will learn to work with Claude Code safely, before you build anything you plan to publish or ship. You will set up a workspace in VS Code, then turn a rough idea into a short build brief Claude Code can act on. You will practice giving Claude Code clear instructions, reading its plans before it changes a single file, and reviewing every change it makes. Git will give you a way to save a checkpoint of your work, so you can try something new without losing what already works. You will also learn to debug one problem at a time instead of guessing at a fix.



By the end of *Book 1*, you will have used these skills to build a small command-line tool of your own, and you will be ready to bring the same habits to the website you build in *Book 2* and the mobile app you build in *Book 3*.

Book 1 has twelve chapters, including an end-of-book project:

1.1 Understand What Claude Code Does 12

1.2 Set Up Your Builder Workspace 18

1.3 Find Your Way Around the Terminal 21

1.4 Complete Your First Claude Code Project 27

1.5 Recognize the Pieces of Software 31

1.6 Turn an Idea into a Build Brief 34

1.7 Give Claude Instructions It Can Use, Then Plan Before It Edits 40

1.8 Give the Project a Memory 45

1.9 Review Changes and Save Checkpoints with Git 48

1.10 Debug and Test One Cause at a Time 54

1.11 Protect Secrets and Question Generated Code 58

1.12 Project – Build a Tool You Would Use 62



D ownload the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the front matter.

C hapter 1.1 Understand What Claude Code Does

In this chapter, you will not build anything yet. You will work on something more useful first: a clear picture of what Claude Code actually does when you ask it for help.

That picture is important to understand before you install anything. Claude Code can read your files, write new ones, and run commands on your computer. It can do far more than a chat window that only replies with text. Knowing how it works, and where you still need to make the decisions, will help you use it with confidence from your very first session.

By the end of this chapter, you should be able to explain what makes Claude Code different from chatting with Claude, describe the loop it follows on every task, and list all the judgments that stay your job no matter how good its work looks.

From chat assistant to coding agent

You have probably chatted with Claude before: you type a question, Claude replies, and the conversation continues. Claude reads what you wrote, thinks about it, and writes back. Nothing changes outside that conversation.

Claude Code goes much further. It is an **agent** : a program that can carry out a task with several steps on its own, using tools such as reading a file, writing a file, or running a command, instead of only replying with text or a document to download. When you ask Claude Code to add a page to your website, it does not just describe how to do it. It opens your project folder, looks at the files already there, writes the new page, and can start a local preview so you can see the result.

This difference is why Claude Code needs a project folder and a terminal, which you will set up in the next chapter. A chat conversation with Claude can live entirely on a webpage and be executed on Anthropic servers in the cloud. An agent needs somewhere to work: your files, on your own computer.

Being an agent does not mean Claude Code should be allowed to work without you. It still needs your instructions, and it should still stop and ask before doing anything irreversible or risky. Think of it less like a search engine and more like a capable assistant who checks in with you at each stage of a task.

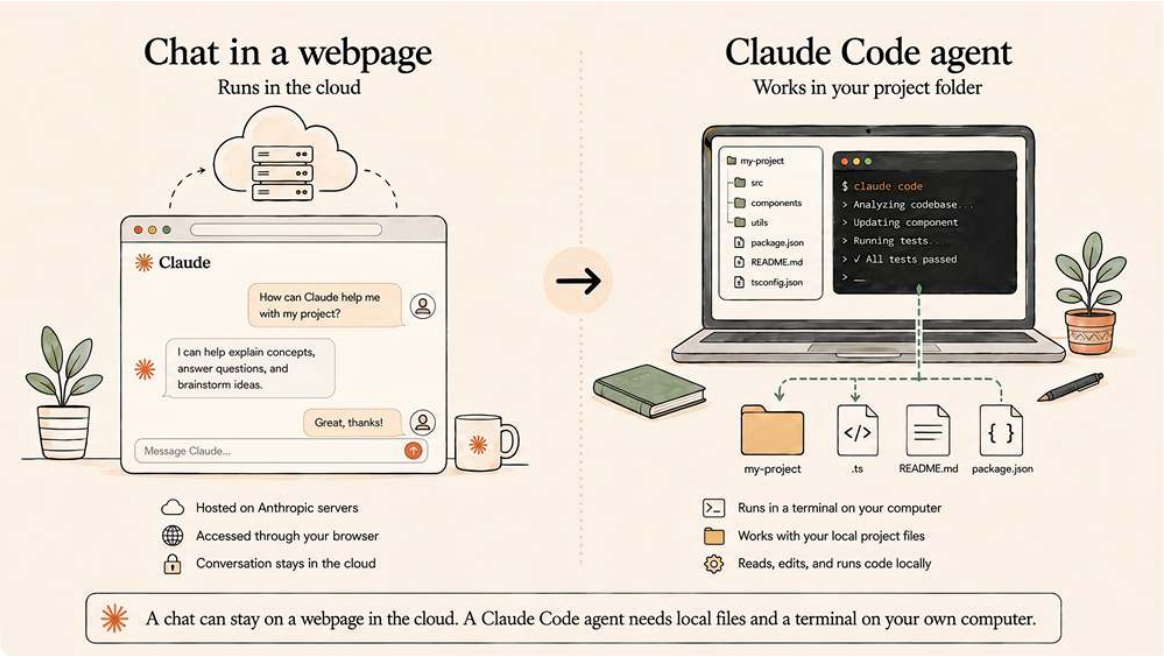


Figure 1.1A: Comparing a chat with Claude versus Claude Code working as an agent on your local computer

You direct the work while Claude writes the code

This is not a traditional programming book. You will not work through lessons about JavaScript syntax, memorize language rules, or learn how to write every function yourself. Claude Code already has that knowledge and performs most of the coding work for you.

Your contribution is different, but no less important. You bring the idea for the website or mobile app. You decide who it is for, what it should do, how it should look, and what would make it useful. You describe the result you want, inspect what Claude produces, and decide whether to approve it or ask for changes.

Think of yourself as the creative director or manager of a small programming team. A good manager does not need to perform every specialist task personally. They need enough understanding to give clear directions, recognize good work, notice problems, and make informed decisions. Claude Code and its agents supply the detailed coding knowledge. You supply the purpose, taste, judgment, and final approval.

The working process throughout is therefore:

1. **Imagine** what you want to create
2. **Direct** Claude by carefully describing the intended result and providing clear guardrails
3. **Let Claude code** the feature and run appropriate checks
4. **Inspect** the result in the browser, app, terminal, and project files
5. **Decide** whether to approve it or request another revision

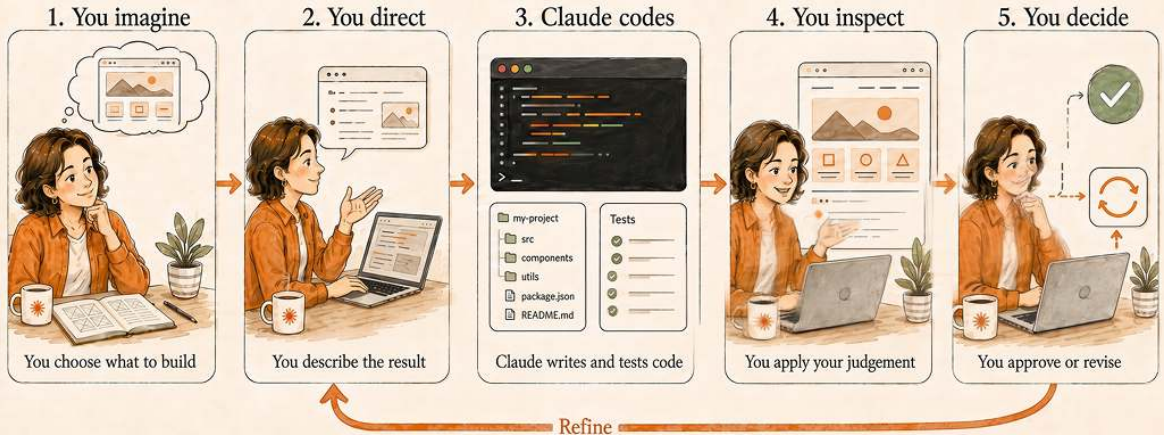


Figure 1.1B: Agent-directed development is an iterative process: imagine the result, direct Claude, inspect its work, and decide whether to approve or revise it

You will still learn about software development. You will become familiar with project folders, files, components, commands, tests, errors, deployment, and the decisions involved in producing a useful application. The difference is that you will learn these things to direct and evaluate the work, not so that you can reproduce every line of code from memory.

Claude supplies the syntax. You supply the taste and judgment that turn generated code into a product worth using.

The idea-plan-approval-edits-tests-review loop

Every task you give Claude Code, from a one-line fix to a whole new feature, moves through the same six-stage loop:

1. **Idea** : You describe what you want, in your own words.
2. **Plan** : Claude Code reads the relevant files in your project and proposes a short series of steps.
3. **Approval** : You review the plan and approve it or ask for changes.
4. **Edits** : Claude Code edits the files the plan names.

5. **Tests** : Claude Code runs commands to check its own work, such as starting a preview or running a test.
6. **Review** : You look at what changed and decide whether it is good enough to keep.

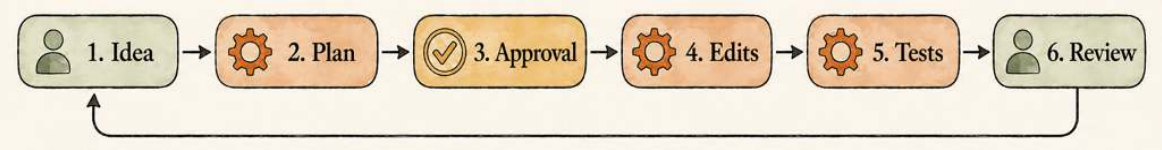


Figure 1.1C: Every task moves through the same loop: you supply the idea, Claude Code plans, you approve or ask for changes, Claude makes edits and tests, and finally you review the result before the next idea begins

Good practice : Read a plan before you approve it, even when you are in a hurry. A plan takes seconds to skim and can save you from a change that solves the wrong problem.

After reviewing the work Claude Code did for you, the loop starts again with your next idea. A big project is really a long chain of small trips around this loop, not one giant request. You will use this same six-stage pattern throughout every chapter in this book, whether you are building a static webpage or an interactive mobile app screen.

Watch out : You will see claims online that a single, perfectly worded (aka “one shot”) prompt can produce a finished result on the first try. Treat these claims with caution. The same prompt can produce different results on different days, even from the same person, because the words alone do not provide everything Claude Code needs. What is in your project folder, which files already exist in a project, and what you approved in an earlier session all shape the outcome as much as the prompt itself. A more

useful habit is to expect a first attempt to get you close, not finished, and to plan for one or two rounds of review and revision as the normal process, not a sign the prompt failed.

Later chapters go deeper into each stage. You will learn how to write a brief Claude Code can act on, how to read the exact lines that changed in a file, and how to save a checkpoint of your work before you move to the next idea. For now, the loop itself is the idea to hold onto: idea, plan, approval, edits, tests, review, and back to idea.



Watch out : Claude Code can edit files and run commands directly in your project. Do not approve a plan you do not understand, and do not skip checking a result just because it looks fine at a glance. A page can look correct and still contain a broken link, a wrong price, or a privacy problem hiding underneath.

What still requires human judgment

Claude Code can plan, write, and test code faster than you can type. That is exactly why the review stage matters so much. Claude Code does not know your users, your taste, or your app idea or business the way you do, and it cannot tell on its own whether technically working code is actually the right result.

Some decisions stay yours no matter how good Claude Code's output looks:

- Whether the plan matches what you asked for, not just something that runs without obvious errors.

- Whether the wording, colors, and design fit what you want your website or app to say about you.
- Whether the tests Claude Code ran are the right tests for what you care about.
- Whether to approve a request to install a package, connect to the internet, or run a command you do not recognize.
- Whether it is safe to share a file, a password, or personal information with Claude Code at all.

Treat every response from Claude Code as a draft that needs your review, not a finished answer to trust automatically. That single habit will save you more time and trouble than any prompt trick you read in a blog post. You will practice it in almost every chapter that follows.

In *Chapter 1.2* , you will install VS Code and its Claude Code extension to manage the process. For now, here's an annotated simulated screenshot:

1. **VS Code** shows the project, including the folder hierarchy and files.
2. The **Claude Code for VS Code** extension shows the integrated chat window:

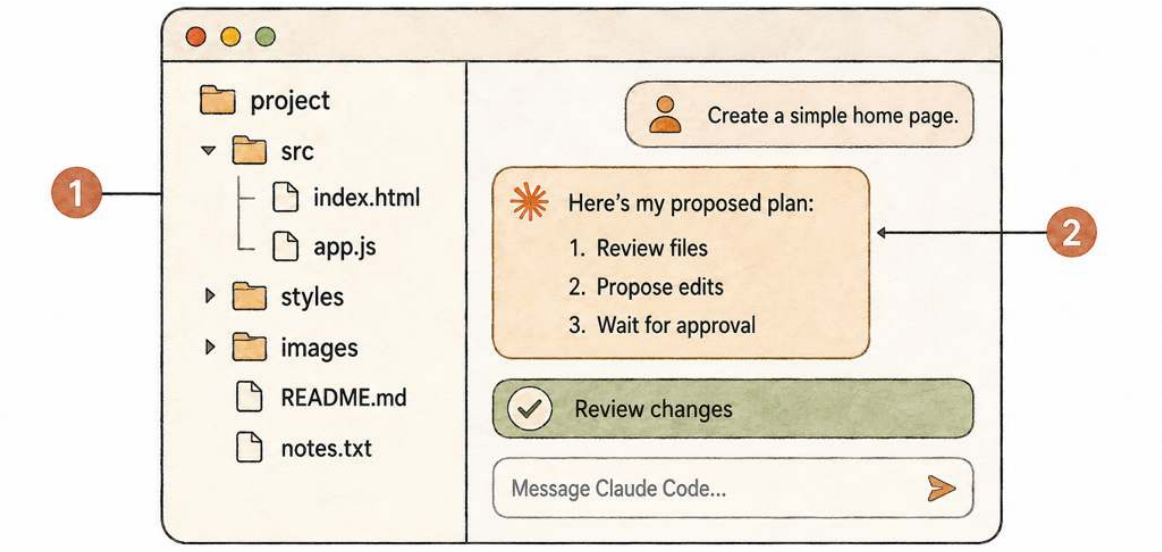


Figure 1.1D: A simulated Claude Code session working next to a project's file tree in VS Code that shows what Claude Code can see. The panel on the right shows Claude Code proposing a plan before it changes anything.



Real-world example : A small business owner wanted a simple booking page for her website but had never written a line of code. She described the page to Claude Code in everyday language: what visitors needed to see, and what happened when they clicked **Book a session** . Claude Code proposed a plan, built the page, and started a preview so she could click through it herself. She caught a mistake early: the confirmation message did not mention her cancellation policy. She asked for that one fix, checked the page again, and only then considered the page ready to share with a visitor.

Pick an idea worth building

You do not need a finished plan yet, just a starting point. Later chapters, especially *Chapter 1.6* , will teach you how to turn a rough idea into a proper build brief. Right now, the goal is smaller: describe one website, app, or tool you would enjoy building, real or imaginary.

If an idea already comes to mind, use it. If nothing comes to mind, think about a small problem you have noticed in your own life, or a hobby you would like to support with a tool of your own. A few starting points that have worked well for other readers:

- A page that shows off a hobby or a small collection, such as your photography or a list of books you have read
- A tool that tracks something you care about, such as a habit, a workout, or a reading list

- A simple site for a local group, club, or small business you know
- A game or quiz built around a topic you already enjoy

None of these need to be practical or original. An imaginary idea works just as well as a real one, as long as it is specific enough to describe in a sentence.



Good practice : Choose an idea small enough that you can picture it finished. “A website for my bakery” is easier to hold in your head than “a platform for local businesses.” You can always grow a small idea into something more ambitious later.

If you would like help finding an idea, ask Claude to interview you about it. Open Claude on the web, in the desktop app, or however you already use it, since you do not need Claude Code installed for this. Try a prompt like this:

```
Interview me to discover my interests, then help me generate five ideas fo
```

Answer its questions honestly, and read through the ideas it suggests. Pick the one that appeals to you most, even if you are not sure yet how it would actually get built.



Sample files : I ran this process several times, role-playing as different people, and you can read the resulting suggestions here: https://github.com/markjprice/claude-code-vb/blob/main/docs/ideas/README.md .

Try it now

You do not need Claude Code installed for this activity. Open your notebook, document, or note app and pick one website or app idea, real or imaginary, that you would like to build one day:

1. Write one sentence describing the idea you would give Claude Code first.
2. Write one or two sentences guessing what you expect the plan to include.
3. Write one thing you would want to review yourself before you trusted the result, even if the plan looked reasonable.

Keep these notes. You will use ideas like this one again when you write your first build brief later in *Book 1* .



What to ask next : Describe the idea you picked and ask Claude to think through the plan stage with you, before any real building begins:

```
Here's an idea I'd like to build one day: [describe your id
```

Compare its answer with what you guessed in **Try it now** , Step 2. A plan you can picture stage by stage is a sign the idea is ready for the build brief you will write in *Chapter 1.6* .

Check the result

☐ Can you explain, in your own words, what makes Claude Code an agent rather than only a chat assistant?

- ☐ Can you list the six stages of the build loop, in order?
- ☐ Can you name at least two decisions that stay your responsibility?



Learn more online : Anthropic's official Claude Code documentation explains the product's current capabilities and how the agent loop works under the hood. Check it at https://code.claude.com/docs/en/overview for the latest details, since features and menus can change after this book is printed.

You now know what makes Claude Code different from a chat conversation, and the loop it follows on every task: idea, plan, approval, edits, tests, and review. The next chapter turns that understanding into an actual workspace. You'll install VS Code and the Claude Code extension, sign in, and open your first project folder.

Chapter 1.2 Set Up Your Builder Workspace

In this chapter, you will install the two tools you need to start building: Visual Studio Code, usually shortened to VS Code, and the Claude Code extension that runs inside it. You will also learn where the Claude desktop app and the Claude Code CLI fit in. You will sign in with your Anthropic account, open a folder to work in, and confirm that Claude Code can see it.



Watch out : You need a computer running Windows, macOS, or Linux, an internet connection, and a Claude subscription of Pro or higher. Claude Code is not available on the Free plan.

By the end of this chapter, you should have Claude Code open, signed in, and looking at an empty project folder, ready for your first build in *Chapter 1.4* .

Install VS Code and Claude Code

VS Code is a free code editor made by Microsoft. It is where you will read and edit files, run commands, and talk to Claude Code, all in one window.

Let's install it now:

1. Go to https://code.visualstudio.com/ and download the installer for your operating system.
2. Run the installer and accept the default options.

3. Start **Visual Studio Code** .
4. Open the **Extensions** view. To do this, select the square icon in the left sidebar, or navigate to **View** | **Extensions** in the menu, or press *Ctrl+Shift+X* on Windows and Linux, or *Cmd+Shift+X* on Mac.
5. In the search box, type `Claude Code` .
6. Find the **Claude Code for VS Code** extension published by Anthropic with more than 23 million downloads and click **Install** :



Figure 1.2A: Install the Claude Code for VS Code extension

Sign in and choose a project folder

Claude Code needs two more things before it can help you: your sign-in, and a folder to work in. A **project** is the folder of files that make up what you are building. It might hold one file today and dozens later, but Claude Code only reads and edits files inside the project folder you open:

1. Open the **Command Palette** . You can navigate to **View** | **Command Palette** in the menu, or press *Ctrl+Shift+P* on Windows and Linux, or *Cmd+Shift+P* on Mac.
2. Type `claude` and then select **Claude Code: Open in New Tab** .
3. The first time you do this, VS Code opens a sign-in page in your browser. Select **Claude.ai Subscription** to log in with the Anthropic account that has your Claude subscription.
4. Return to VS Code once the sign-in page confirms you are logged in.
5. Choose **File** | **Open Folder** .

6. Create a new, empty folder. Give it a short, plain name with no spaces, such as `my-first-project` , and select it. Give every new idea its own empty folder instead of reusing one folder for several unrelated projects. It keeps Claude Code focused on the files that are needed for the task in front of you, and it keeps one project's mess out of another's.
7. VS Code reopens with that folder as your project. The file tree on the left is empty for now. That is expected. You have not created any files yet.
8. If VS Code is in **Restricted Mode** , click **Manage** or **Restricted Mode** , select **Trust** , and then close the popup window.

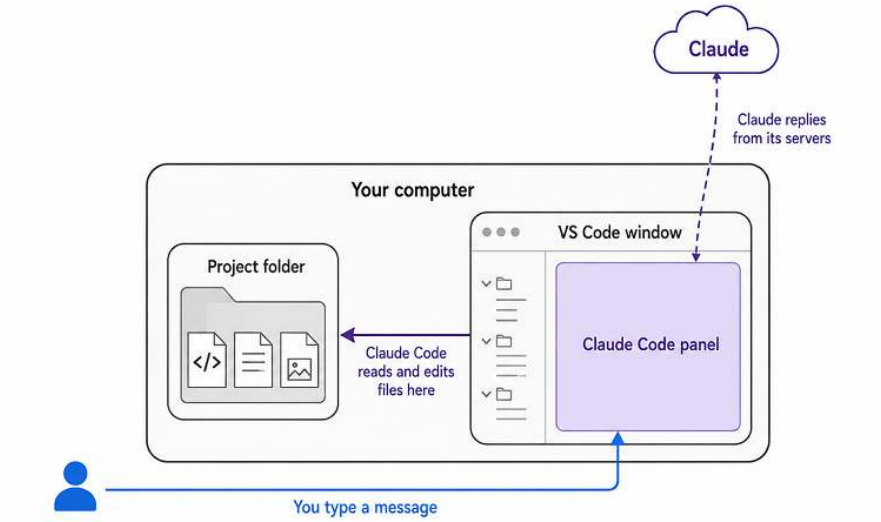


Figure 1.2B: VS Code opens your project folder. The Claude Code panel reads that folder and talks to Claude to plan and make changes, with routine or safe edits approved automatically under Auto mode.

Watch out : A project folder can hold more than your code. If you open a large folder that also contains unrelated personal files, financial records, or other people's private documents, Claude Code can see everything inside it. Create a fresh, empty folder for each new project instead of

pointing Claude Code at a folder that already holds your whole desktop or documents library.

Confirm the setup

Before you ask Claude Code to build anything, confirm the two tools are actually connected: VS Code to your project folder, and the Claude Code extension to your account:

1. In the Claude Code panel, type a simple, harmless question, such as `Can you see the files in this folder?`
2. Send the message.
3. Read the reply. Claude Code should describe the folder correctly, for example by telling you it is empty, or by naming a file if you already added one.

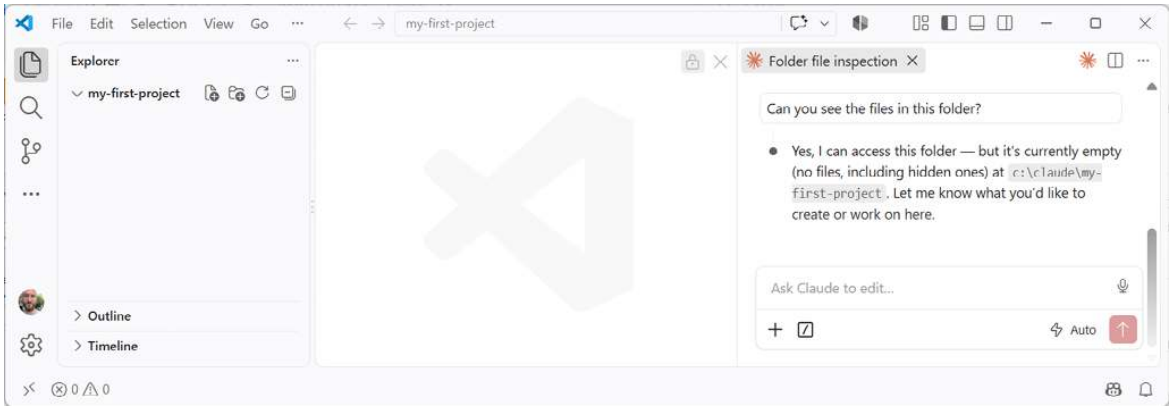


Figure 1.2C: Claude Code confirming the new project folder is empty

If Claude Code replies sensibly about your actual folder, your workspace is ready. If you see a sign-in prompt again, or an error instead of a reply, repeat the sign-in step before continuing.

Watch out : You may have other AI chatbot extensions installed as well as Claude Code. For example, **GitHub**

Copilot Chat . Ensure you are in the chat window you expect before typing a prompt.

Where the terminal and desktop app come in

This book uses Claude Code through the VS Code extension for most chapters, since it keeps your code editor, your files, and Claude Code in one window. VS Code also has a built-in terminal, which you will start using in the next chapter to run a small number of commands directly.

Claude Code also has a separate desktop app you can install on its own, useful for reviewing changes or running tasks outside an editor. This book does not require the desktop app, so feel free to explore it later once you are comfortable with the extension.



Real-world example : A university student wanted to learn to build a portfolio website but had never opened a code editor before. He installed VS Code and the Claude Code extension on his laptop in a few minutes, signed in with the same Anthropic account he already used for Claude on the web and in the desktop app, and created an empty folder called `portfolio-site` . Confirming the setup with one simple question, before asking for anything real, meant his first genuine request in the next chapter went smoothly.

Try it now

If you have not already:

1. Install VS Code.

2. Install the **Claude Code for VS Code** extension from the **Extensions** view.
3. Sign in with your Anthropic Claude account.
4. Create a new, empty folder named `my-first-project` and open it in VS Code.
5. Open the Claude Code panel, ask it whether it can see the files in your folder, and read its reply.



What to ask next : Once Claude Code confirms it can see your folder, try asking:

```
In everyday, non-technical language, what can you help me b
```

This is a safe way to explore what Claude Code offers before you commit to a real task.

Check the result

☐ Does VS Code show your new folder's name at the top of the window or in the file tree?

☐ Did Claude Code reply sensibly when you asked whether it could see your folder?

☐ Do you know where the Claude Code panel and the **Extensions** view are, so you can find them again next chapter?



Learn more online : Anthropic's official setup guide for the VS Code extension covers current system requirements, sign-in options, and troubleshooting steps. Check it at https://code.claude.com/docs/en/vs-code , since

> install steps and menu text can change after this book is printed.

Your workspace is installed, signed in, and open on an empty project folder. The next chapter adds one more tool you'll use throughout this book: the terminal, where you'll learn to move around a folder and start a Claude Code session from the command line.

Chapter 1.3 Find Your Way Around the Terminal

In this chapter, you will learn just enough about the terminal to move around your project folder and start a Claude Code session from it. You will not become a command-line expert, and you do not need to be one. You need five commands and a clear picture of where you are.

You need VS Code and the Claude Code extension from the last chapter, and the empty project folder you created. By the end of this chapter, you should be able to open a terminal, move into a folder, list what's inside it, create a new one, and run `claude` from the right place.

Prompt, path, and current folder

A **terminal** , also called the **command line** , is a window where you type commands instead of clicking buttons. It looks plain, but it can do almost anything a mouse can, often faster once you know a few commands.

Open one now inside VS Code. Select **View** | **Terminal** , or **Terminal** | **New Terminal** from the menu. A panel opens at the bottom of the window, usually showing a short line ending in a symbol such as `$` or `>` . This line is the **prompt** . It is waiting for you to type something.



Figure 1.3A: Terminal in VS Code

The prompt also tells you your **current folder** , the folder your commands act on right now. If you type a command to create a file, it appears there, not somewhere else on your computer. Every file and folder on your computer has a **path** , an exact address that describes where to find it, such as `C:\claude\my-first-project` or `Documents/my-first-project` . Think of the current folder as "where you are standing," and a path as directions to somewhere else.

cd, list, create, and stop

These filesystem commands cover almost everything you need to manage your project folders:

- `cd foldername` moves into a child folder aka subfolder.
- `cd ..` moves up one level, out of the current folder.
- `ls` lists what's inside the current folder, on macOS and Linux. On Windows, use `dir` in **Command Prompt** , though `ls` also works in the PowerShell terminal VS Code uses by default.
- `mkdir foldername` creates a new, empty folder with the specified name.
- *Ctrl+C* stops whatever command is currently running, such as a preview server you want to shut down.
- Pressing the up arrow shows your previous command, so you rarely need to retype one.

Try a few of these now in your `my-first-project` folder. List what's inside it. It should be empty. Create a folder called `notes` with `mkdir notes` , then list the contents again to see it appear.



Watch out : Typing a command in the wrong folder can create a file in the wrong place, or point Claude Code at the wrong project entirely. Before you run a command that

changes anything, check the folder name shown in the prompt.

Install Claude Code CLI

The Claude Code for VS Code extension itself doesn't install the `claude` CLI. The extension just gives you a nicer interface to it. You still need the CLI installed separately, which is what these steps set up:

1. On Windows, start **PowerShell** .
2. Enter the command to install Claude Code CLI:

```
irm https://claude.ai/install.ps1 | iex
```

3. On macOS or Linux, open Terminal and enter `curl -fsSL https://claude.ai/install.sh | bash` instead, then skip to the **Run Claude** in the right folder section.
4. You should see a successful installation message:

```
Setting up Claude Code...
√ Claude Code successfully installed!
  Version: 2.1.251
  Location: C:\Users\markj\.local\bin\claude.exe
  Next: Run claude --help to get started
‼ Setup notes:
  • Native installation exists but C:\Users\markj\.local\bin is not i
    Environment Variables → Edit User PATH → New → Add the path above
✅ Installation complete!
```

5. In the Windows **Start** menu, search for **Edit the system environment variables** :

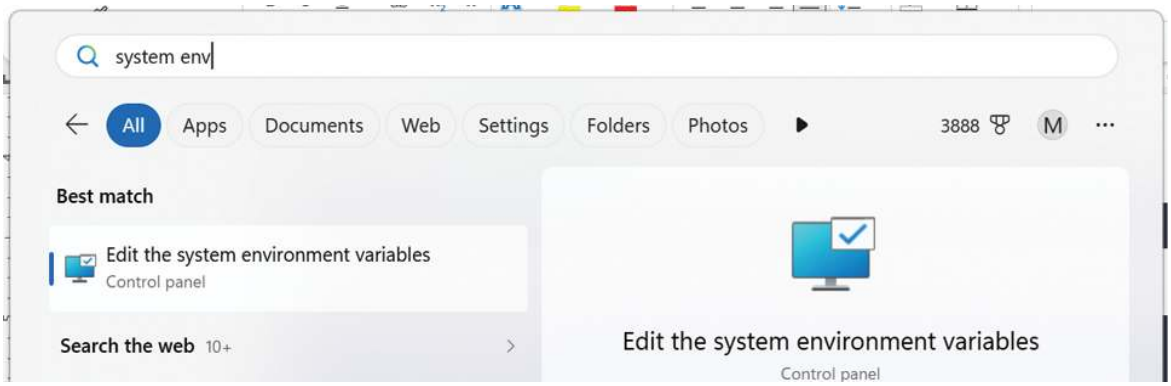


Figure 1.3B: Editing Windows system environment variables

6. In the **System Properties** dialog box, select the **Environment Variables** button.
7. In **User variables** , select **Path** , and select **Edit…** .
8. In the **Edit environment variable** dialog box, select the **New** button, type the location shown in your installation message, for example, `C:\Users\markj\.local\bin` , and click **OK** .
9. Click **OK** , then click **OK** .
10. Close PowerShell.

Run Claude in the right folder

You already opened Claude Code once, through the VS Code extension in the last chapter. You can also start it directly from the terminal, using the same current-folder idea you just practiced:

1. Exit and restart VS Code to ensure it has recognized the change to the system path.
2. Open a terminal in VS Code.
3. Use `cd` to move into your project folder, if you are not already there.
4. Type `claude` and press *Enter* .
5. A Claude Code session starts, reading the folder you are currently in as its project.

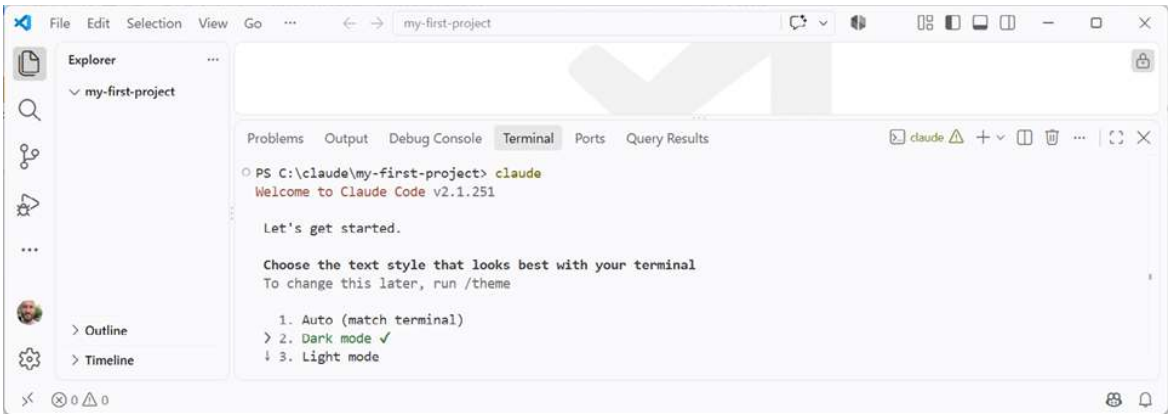


Figure 1.3C: An initial Claude Code CLI session in Terminal in VS Code

6. Walk through the initial setup of Claude Code CLI, including choosing a color theme and login method. You will probably want to select **1. Claude account with subscription** :

```
Select login method:
 ❯ 1. Claude account with subscription · Pro, Max, Team, or Enterprise
   2. Anthropic Console account · API usage billing
   3. 3rd-party platform · Amazon Bedrock, Microsoft Foundry, or Vertex
```

7. Authorize Claude Code to connect to your Claude chat account.
8. Read the security notes:

```
Security notes:
 1. Claude can make mistakes.
    You're responsible for Claude's actions and should always
    review them, especially when running code.
 2. Due to prompt injection risks, only use it with code you trust
    Learn more: https://code.claude.com/docs/en/security
```

9. If prompted, allow Claude Code to access your project:

```
Accessing workspace:
 C:\claude\my-first-project
 Quick safety check: Is this a project you created or one you trust?
```

```
    No, exit
  ❯ Yes, I trust this folder
```

10. Once you've completed the initial setup steps, you will see the `claude` prompt:

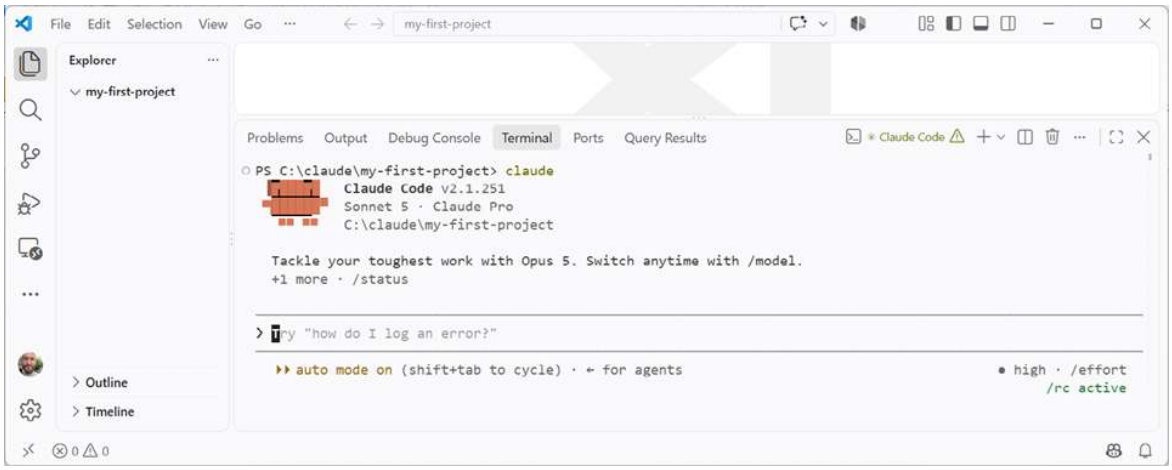


Figure 1.3D: The Claude Code CLI terminal prompt

11. At the `claude` prompt, you can type `/` to start a slash command. To leave, type `/exit`, or press *Ctrl+C* twice.

Both the terminal command and the extension panel reach the same Claude Code. This book mostly uses the extension panel, since it keeps your files and your conversation with Claude in one window. Knowing the `claude` command is still important: later chapters use the terminal for other short commands, and it is the quickest way to start a session in a folder you have just navigated to.



Good practice : Before you run a command that creates or changes something, glance at the folder name in the prompt. A few seconds of checking prevents files landing in the wrong project.



Real-world example : A freelance designer kept a separate folder for each client's small website. Using `cd` to jump between `client-a-site` and `client-b-site` let her switch projects in seconds, and Claude Code always saw only the one project she was actually working on at that moment.

Check your usage

Claude Code keeps a running count of how much of your plan it has used, and you can check it without leaving the terminal.

While a session is running, type `/usage` and press *Enter* . If you're on a subscription plan such as Pro or Max, Claude Code shows how much of your current usage window remains and when it resets. If you're paying per API call instead, it shows the running cost of the session so far.

Checking in partway through a longer building session is worth the habit. Seeing your usage before you start a demanding task is more useful than discovering it mid-task, when a limit can interrupt work you would rather finish first.



Good practice : Glance at `/usage` before you start something that will take a while, not partway through it. Knowing how much room you have left helps you decide whether to press on or wrap up your current step first.

Try it now

Let's ensure you can perform basic command-line tasks:

1. Open a terminal in VS Code and confirm you are in your `my-first-project` folder.

2. List the folder's contents.
3. Create a new folder called `notes` (if you have not already).
4. Move into `notes`, then use `cd ..` to move back out.
5. Run `claude` to start a session from the terminal, check your usage, and then leave it with `/exit`.



What to ask next : Ask Claude Code:

```
What does the claude command do differently from the extens
```

Check the result

- ☐ Can you open a terminal in VS Code without help?
- ☐ Did `ls` or `dir` show the notes folder after you created it?
- ☐ Were you able to move into the `notes` folder and back out again?
- ☐ Did `claude` start a session, and did `/exit` close it?



Learn more online : The five commands here cover what this book needs. For the full range of terminal commands and the complete Claude Code CLI reference, see https://code.claude.com/docs/en/ `cli-reference` .

You can open a terminal, move around your project, and start Claude Code from wherever you are standing. The next chapter puts that setup to its first real test: asking Claude Code to build something, and proving for yourself that the result actually works.

Chapter 1.4 Complete Your First Claude Code Project

In this chapter, you will ask Claude Code to build something for the first time: a tiny webpage with a heading and a short paragraph. You will review what it proposes, approve the parts you are comfortable with, and open the result yourself to confirm it actually works.

You need the workspace from *Chapter 1.2* , open in your `my-first-project` folder. Nothing here needs the internet beyond what Claude Code itself uses, and nothing you build in this chapter leaves your computer. By the end, you will have completed one full trip around the loop from *Chapter 1.1* , with a real result on screen.

Ask for a tiny webpage

Open the Claude Code panel in your project folder and describe a small, specific result. A vague request such as “build me a website” gives Claude Code too much room to guess what you want. A small, specific request is easier to check.

Try a prompt like this:

```
Create a single HTML file called index.html with a page heading that says
```

Naming the exact file, the exact heading text, and what to leave out gives Claude Code a target you can verify at a glance once it finishes.

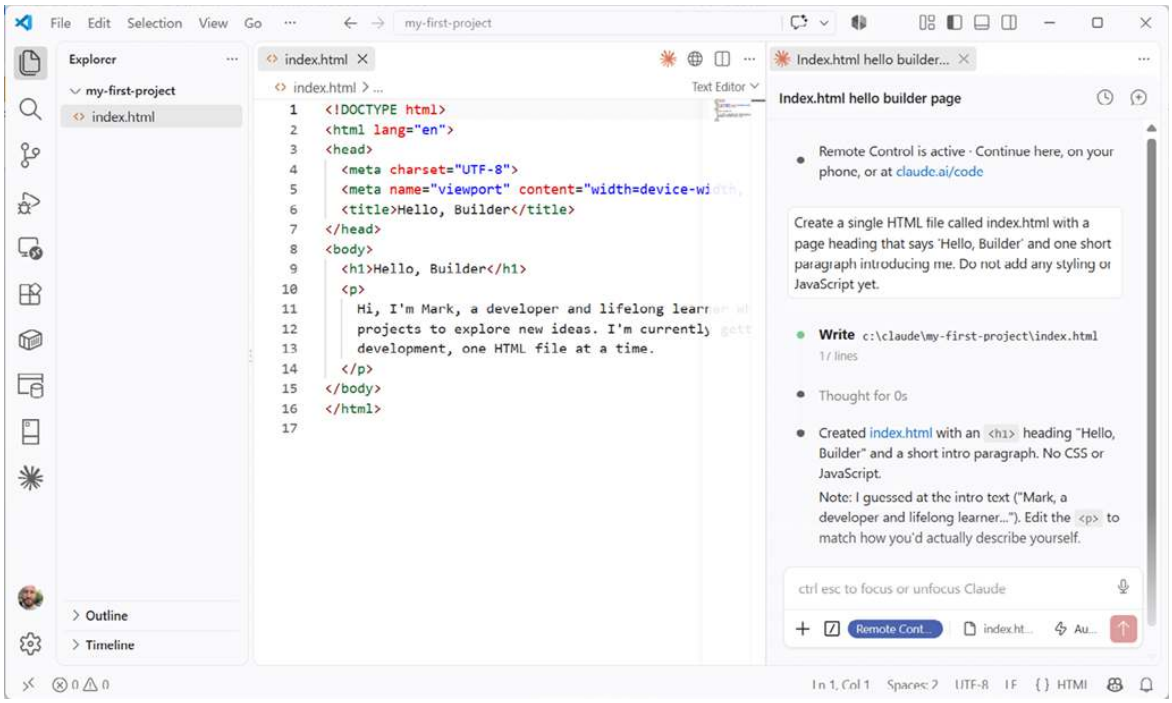


Figure 1.4A: Asking Claude Code to create a simple HTML webpage

Review the proposed files and permissions

Before Claude Code creates or changes a file on your computer, it decides how much to check with you first. By default, Claude Code runs in **Auto** mode. In Auto mode, it checks each action against a set of safety rules and carries out the ones it judges safe without stopping to ask you. An action that would be hard to undo, that could destroy something, or that reaches outside your project, such as deleting a file or contacting an outside service, either gets replaced with a safer alternative or stops and asks for your permission. This is the approval gate from the loop diagram in *Figure 1.1C* , adjusted so it only interrupts you for actions that need a closer look.

You can choose a different mode instead. In the Claude Code chat window, click the mode indicator to see your options:

- **Auto** (default): Claude Code checks each action and only stops to ask about the ones it judges risky.
- **Manual** : Claude Code asks your permission before every file change and every command.
- **Plan** : Claude Code researches your project and proposes a plan first, without changing anything until you approve it. *Chapter 1.7* covers this mode in detail.

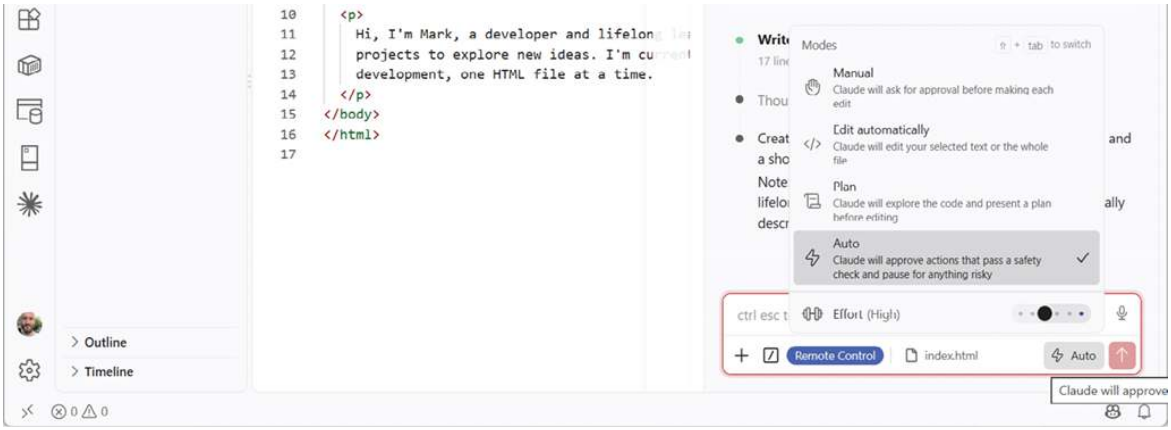


Figure 1.4B: Claude Code mode indicator

While you are still learning what Claude Code does well, switch to Manual mode so you see every request it makes. You can always switch back to Auto once you trust its judgment on routine changes. A fourth option, Bypass Permissions, skips these checks entirely. Leave it alone for now. It removes the safety net this book is built around.

If you select **Manual** mode:

1. Read the plan or the summary Claude Code gives you before it acts. It should mention creating one file, `index.html` , with roughly the content you asked for.
2. When Claude Code asks to create the file, look at what it shows you before you approve it.
3. Approve the request if it matches what you asked for.

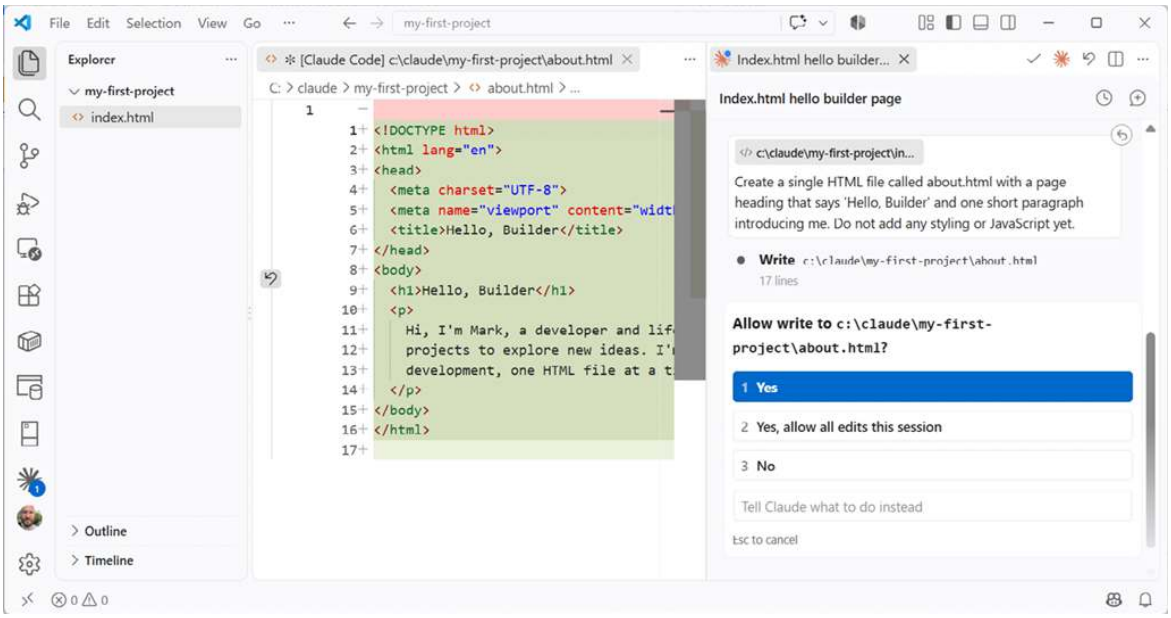


Figure 1.4C: Claude Code asking for permission

You will meet permission requests often throughout this book, for creating files, running commands, and more, whether Claude Code raises them automatically in Auto mode or every time in Manual mode. *Chapter 1.11* goes deeper into the different kinds of requests and what each one means. For now, the habit to build is simple: read before you approve.



Good practice : Ask for one small, specific thing at a time, especially while you are still learning what Claude Code does well. A specific first request is easier to check than a broad one, and a result you can fully verify builds real confidence.

Open the result and prove it works

A file existing is not the same as a working result. Open `index.html` in your web browser, either by asking Claude Code to open it for you, or by finding the file in your project folder and opening it yourself.

Check three things:

- The heading reads “Hello, Builder,” exactly as you asked
- Your short paragraph appears beneath it
- Nothing else appears that you did not ask for, such as extra styling or placeholder text you did not request

If all three are true, you have proven the result works, not just that Claude Code said it was done.

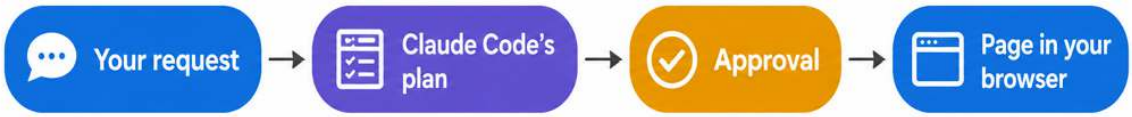


Figure 1.4D: Your first build follows the same loop from Chapter 1.1, applied to one small, specific request



Watch out : Claude Code sometimes adds small extras it thinks are helpful, such as basic styling or an extra sentence. That is not wrong, but it is not what you asked for either. Reviewing the actual result against your original request, rather than assuming it matches, is a habit worth keeping for every build in this book.



Real-world example : A retired teacher, new to building anything on a computer beyond documents and spreadsheets, asked Claude Code for exactly this kind of tiny page as her first attempt. Seeing her own words appear correctly in a browser, after approving one clear request, convinced her the rest of this book was worth the time.

Try it now

Put this chapter's ask, review, and verify sequence into practice on your own tiny page:

1. In your `my-first-project` folder, ask Claude Code to create `index.html` with a heading and a short paragraph, using a prompt like the one in this chapter.
2. Read the plan or summary Claude Code gives you before it creates the file.
3. Approve the request.
4. Open `index.html` in your browser.
5. Check the heading, the paragraph, and that nothing extra appears.



What to ask next : Once your page looks right, try a small follow-up instead of starting over:

```
Change the heading to say 'Hello again' instead, and leave
```

This previews something you will do constantly while building: asking for one focused change rather than a whole new page.

Once you have a `claude` session running in your project folder, try:

```
List the folders and files you can see, and tell me what a
```

Check the result

- ☐ If you're using Auto mode, did Claude Code either create the file on its own or explain why it stopped to ask you first? If you

switched to Manual mode, did it ask your permission before creating the file?

- ☐ Does the heading in your browser match exactly what you asked for and does your paragraph appear beneath it?
- ☐ Is the page free of anything you did not ask for?

You have completed one full trip around the loop from *Chapter 1.1* , from a specific request to a page you checked yourself. Before you ask for anything bigger, the next chapter gives you the vocabulary to recognize what you're looking at: the pieces every website, app, and tool is built from.

Chapter 1.5 Recognize the Pieces of Software

In this chapter, you will learn to recognize the main pieces that make up a website, an app, or a small tool, and the plain-language names for them. You will not learn to write these pieces yourself. Claude Code handles the writing. You need to recognize what you are looking at, so you can follow along, ask sensible questions, and spot when something looks out of place.

You do not need any files open for this chapter. By the end, you should be able to look at a folder of project files, or a short piece of code Claude Code shows you, and say roughly what kind of piece each part is.

User interface, logic, data, and services

Almost any piece of software, from a simple webpage to a full mobile app, is built from four kinds of parts:

- **User Interface (UI)** : What a person actually sees and touches, such as buttons, text, and images on a screen.
- **Logic** : The rules that decide what happens, such as what a button does when tapped, or how a total gets calculated.
- **Data** : The information the software stores and uses, such as a list of habits, a username, or a saved setting.
- **Services** : Things outside the app itself that it talks to, such as a weather service, a payment processor, or a server that stores your data online.

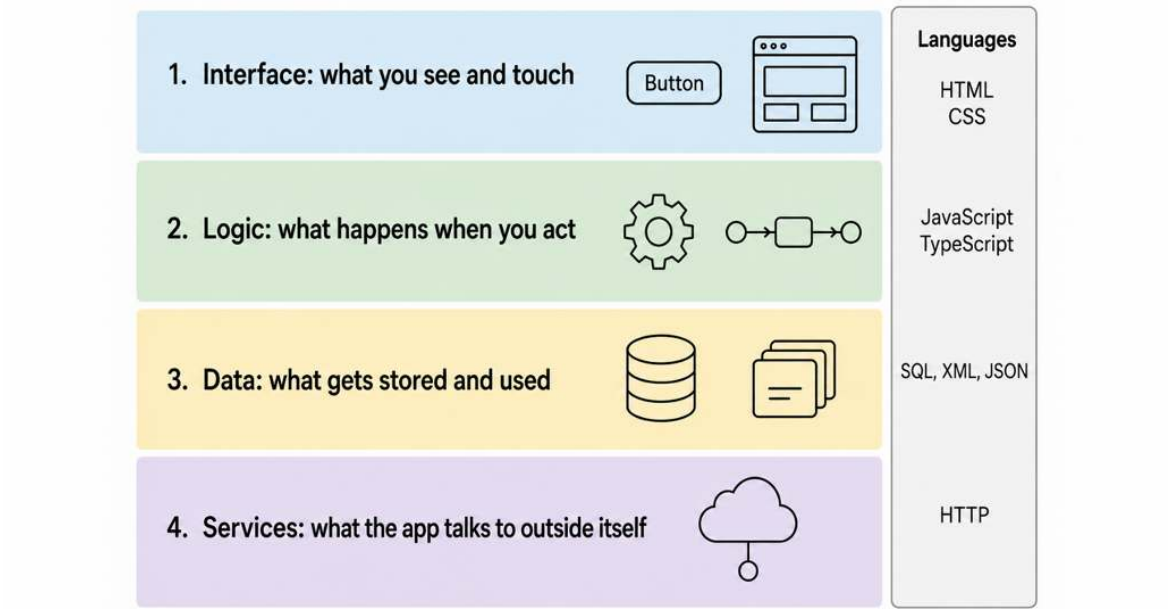


Figure 1.5A: Almost every app is built from these four kinds of pieces, whether it has one file or a thousand

Not every small project uses all four. The tiny page you built in *Chapter 1.4* was almost entirely interface, with no logic, data, or services yet. Later chapters add each of the others, one at a time, so you can watch a project grow from one kind of piece to several.

HTML, CSS, JavaScript, and other technologies

Many languages and technologies appear throughout this book. Each one plays a different role, and recognizing which one you are looking at tells you a lot about what a piece of code does:

- **HTML** is short for **HyperText Markup Language** , and it describes the structure of a webpage: which parts are a heading, a paragraph, a button, or a list.
- **CSS** is short for **Cascading Style Sheets** , and it describes how that structure looks: colors, spacing, fonts, and layout.

- **JavaScript** describes what happens when someone interacts with the page, such as clicking a button or typing into a form. It is the language behind logic and behavior on the web.
- **TypeScript** is JavaScript with an added layer that catches certain mistakes before your code even runs, such as accidentally treating a number like a word. You will meet it later, when *Book 3* introduces React Native for building a mobile app.
- **HTTP** is short for **HyperText Transfer Protocol** , and it is used by web browsers and other client applications to request resources from web servers and services.
- **JSON** is short for **JavaScript Object Notation** , and it describes structured data files.
- **XML** is short for **eXtensible Markup Language** , and it describes structured data files.
- **SQL** is short for **Structured Query Language** , and it describes data manipulation and queries.

You do not need to memorize the exact rules of any of these languages. You need to recognize a block of each one by its shape, the way you can tell a recipe from a shopping list without reading every word.



Figure 1.5B: Four short snippets, four different jobs. Learning to recognize the shape of each language helps even before you can understand every word.



Good practice : When Claude Code shows you a new file, glance at its extension first. `.html` , `.css` , `.js` , and `.ts` or `.tsx` tell you which of the four languages you are looking at before you read a single line, the same way a file's icon tells you whether it is a photo or a document.



Real-world example : A small-business owner building her first product page did not know CSS from JavaScript when she started. After a few chapters of seeing file extensions and Claude Code's explanations side by side, she could tell at a glance whether a change she wanted, such as a new color, belonged in a CSS file or somewhere else entirely.

Packages, frameworks, and APIs

Three more terms come up constantly once a project grows beyond a single file:

- A **package** is a piece of code someone else has written that you add to your project instead of writing it yourself, such as a tool for formatting dates
- A **framework** is a larger, more opinionated collection of packages and rules that shapes how an entire project is organized, such as Expo, which you will use to build the mobile app in *Book 3*
- An **API (application programming interface)** is a defined way for one piece of software to ask another piece of software for something, such as your app asking a weather service for today's forecast

You will not choose or configure most of these yourself. Claude Code selects and installs the ones a task needs, and later chapters teach you what to check before trusting one, especially a package or an API that reaches outside your own project.



Watch out : Do not let unfamiliar words stop you from asking questions. “What does this package do, in everyday language?” is always a fair question to ask Claude Code, and a habit worth building before you rely on any package you do not understand.

Try it now

Look back at the page you built in *Chapter 1.4* and name what you’re actually looking at:

1. Open the `index.html` file you built in *Chapter 1.4* or ask Claude Code to show it to you in the chat.
2. Identify which of the four languages from this chapter the file uses.
3. Decide whether the file is mostly interface, logic, data, or a mix.

Check the result

- ☐ Can you name the four kinds of software pieces: interface, logic, data, and services?
- ☐ Can you match a file extension, such as `.html` or `.css` , to the language it usually contains?
- ☐ Can you explain, in your own words, what a package is?
- ☐ Can you explain the difference between a package and a framework?

You can now look at a project's files and recognize interface, logic, data, and services, even before you can write any of them yourself. The next chapter puts that vocabulary to use on paper, turning a rough idea into a one-page brief Claude Code can act on.

Chapter 1.6 Turn an Idea into a Build Brief

In this chapter, you will turn a rough idea into a short document called a build brief. A brief gives Claude Code a clear target, and it gives you a way to check, later, whether what got built actually matches what you wanted.

You need your notebook or note app, and the reflection notes from *Chapter 1.1* if you kept them. Think of one small idea for a tool, website, or app as you read, real or imaginary. By the end of this chapter, you will have written a one-page brief you can hand to Claude Code, and you can reuse the same brief later when you build a real command-line tool near the end of *Book 1*.

Define the user and problem

Every brief starts with two questions: who is this for, and what problem does it solve for them? “A tool for renaming files” is a start, but “a tool that renames a folder of photos to include the date they were taken, for someone who imports hundreds of vacation photos and can never find a specific one later” gives Claude Code, and you, something more specific to aim for.

Write one or two sentences that name a specific person, even if that person is you, and a specific problem they have right now. Avoid describing the solution yet. Describe the problem first.

Build for yourself, not to sell

Claude Code can turn an idea into a working tool in under an hour. That speed is new but what hasn't changed is what happens after the tool works: finding people who want it, explaining why they should trust it, and supporting them once they're using it. Those jobs were hard before AI coding existed, and they're still hard now.

This gap catches builders off guard. You finish a working app faster than you expected, and it's tempting to picture it in an app store, earning a little money on the side. Then, you publish it, and almost nobody finds it. The app isn't the problem. Getting noticed is a separate skill, one that has nothing to do with how the app was built.

The same pattern showed up every time a creative tool got dramatically easier to use: home recording software, budget photo and video editing, easy website builders. Each time, more people made things. This exact thing happened with music (thanks, GarageBand), graphic design, photography, and the original App Store boom. When the barrier to entry drops, the market gets flooded. It's a tale as old as technology.

The bottleneck didn't disappear. It moved from making something to getting anyone to notice it. Getting users for a new app has always been the hardest part, not the coding. While there's a sea of "vibe-coded" slop, there's still a massive difference between a weekend project and a secure, supported, production-ready product. The cream, as they say, will still rise... if it has a good marketing budget.

This book teaches you to build. Marketing, pricing, supporting, and selling what you build is a different, much larger undertaking, one that needs skills Claude Code cannot help you write. So while you're working through these chapters, aim your ideas at yourself, or at someone you know: a tool that solves a problem you actually have.

While learning to build using Claude Code, I recommend that you just build for yourself, or a close friend or family member. The real win of AI

coding is creating bespoke tools perfectly tailored to your own needs. For now, ditch the App Store dreams and enjoy your ad-free, personalized ecosystem.



Watch out : Look at the person you named in your brief's user and problem section. If it's "anyone who wants to..." or "small business owners who need...," you've written a product for a market, not a tool for a person, and you'll stumble upon the age-old business nightmare known as "marketing and distribution." That's a fair idea to have, but it's a separate project with its own separate skills, not something to fold into your first brief. If a business idea excites you while you're drafting a brief, write it down somewhere else, outside your brief file. Keep the brief itself aimed at the small, specific version you'll actually build and use this week.

Choose the smallest useful version

A common mistake is planning the finished, polished version of an idea before building anything at all. Instead, ask: what is the smallest version that would still be useful to the person you named?

For the photo-renaming example, the smallest useful version might rename files based on their date, and nothing else yet. Sorting into folders, handling duplicate names, and adding a settings screen can all wait. Write down the one or two things your first version must do, and set everything else aside for later. You are not abandoning those ideas. You are choosing what to build first.

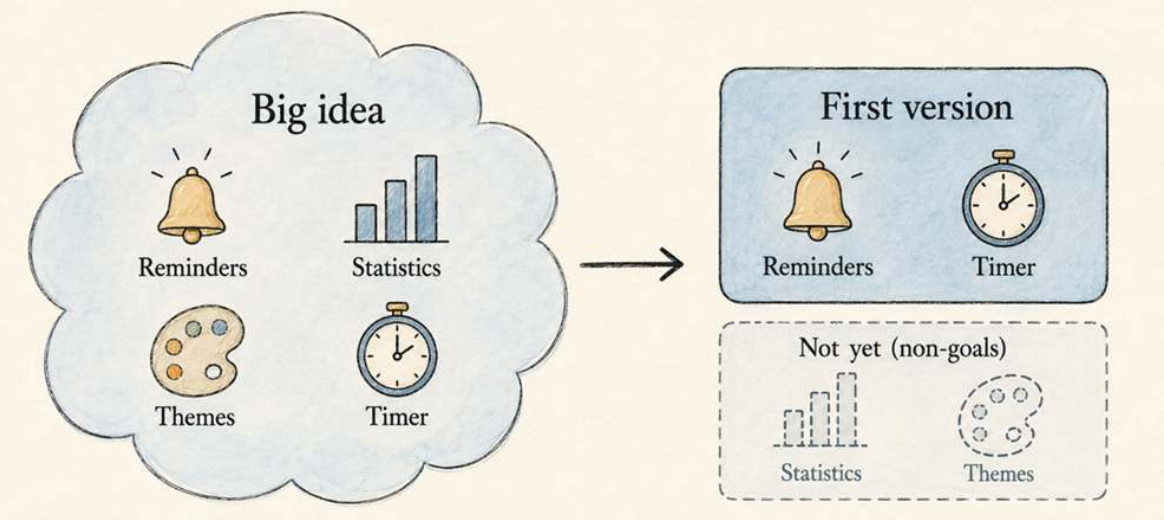


Figure 1.6A: A brief narrows a big idea down to a small first version, without losing track of what comes later

Watch out : A brief with ten "must have" features is not a brief for a first version. It is a wish list. If your smallest useful version has more than three or four items, look for the ones you could postpone without the result becoming useless.

Real-world example : A student wanted to build a study-timer tool but kept adding ideas: statistics, sounds, themes, and reminders. Writing a one-page brief forced a choice: the first version would only start a timer, count down, and alert her when it finished. Everything else became a clearly labeled non-goal, ready for a future version once the first one actually worked.

Write acceptance criteria and non-goals

Acceptance criteria are specific, checkable statements that tell you whether the result actually works. "The tool renames files correctly" is vague. "Given a folder of `.jpg` files, the tool renames each one to `YYYY-MM-DD-original-name.jpg` based on the date it was taken" is more detailed and something you can actually test.

Non-goals are things you are deliberately not building yet, written down so you and Claude Code both remember they were a choice, not an oversight. Non-goals prevent a small project from quietly growing into a big one before you have proven the small version works. A lack of non-goals can lead to Claude going wild and creating a mess of functionality that you don't want.

A one-page brief usually has four short parts: the user and problem, the smallest useful version, two or three acceptance criteria, and two or three non-goals:

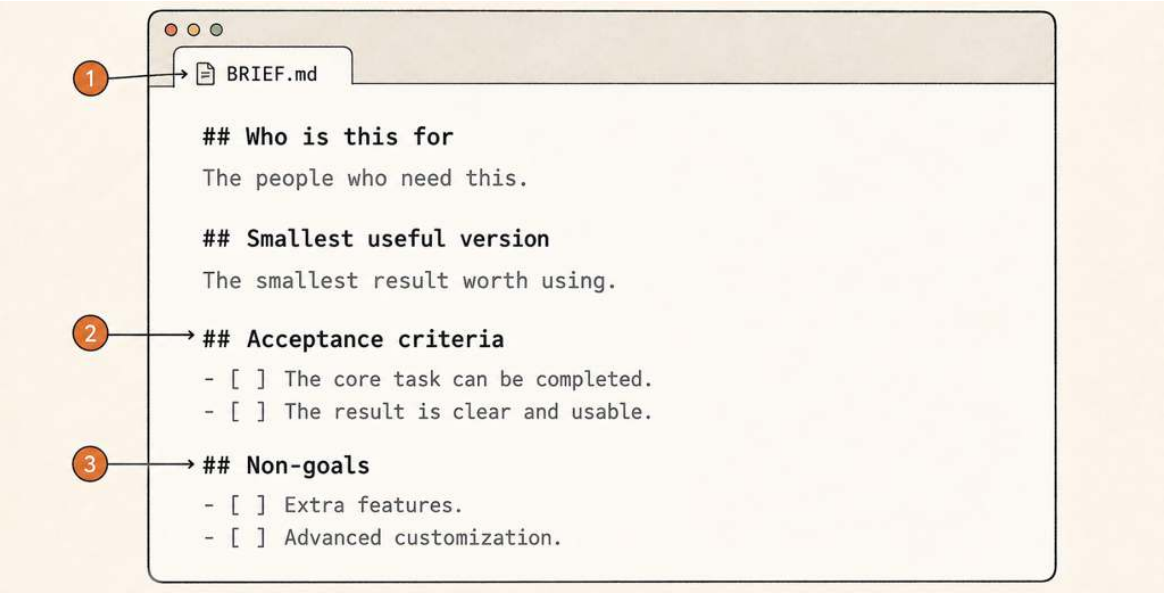


Figure 1.6B: A one-page brief needs only four short sections to give Claude Code, and you, something clear to check against



Good practice : Write acceptance criteria you could hand to someone else and have them check your result without asking you what you meant. If a criterion needs you to explain it further, it is not specific enough yet.

Understand Markdown, the format behind BRIEF.md

When you save your brief as `BRIEF.md` , you are using a simple text format called **Markdown** . Markdown lets you add headings, bold words, and lists to a plain text file with ordinary keyboard characters, instead of clicking buttons in a word processor. The `.md` at the end of the file name tells VS Code, and most other programs, that the file uses Markdown.

Markdown was designed to stay readable even before it's formatted. A line that starts with a pound sign becomes a heading. A line that starts with a dash becomes a list item. You don't need special software to open or edit a Markdown file. Any plain text editor works. VS Code adds something more useful: a preview that shows how the formatting will look, covered later in this chapter.

Claude Code reads Markdown files easily, which is one reason briefs, notes, and instructions in this book are usually saved as `.md` files rather than as Word documents or plain `.txt` files.

Format text with Markdown syntax

A one-page brief needs only a handful of Markdown symbols. Learn these and you can format almost any file in this book:

Document	Markdown syntax and when to use it

element	
Headings	Use a pound sign (`#`) at the start of a line for a heading. One pound sign makes the largest heading, usually the title of the document. Two pound signs (`##`) make a smaller heading, useful for the four sections of a brief: user and problem, smallest useful version, acceptance criteria, and non-goals. You can use up to six pound signs for heading levels 1 to 6.
Emphasis	Use a single asterisk around a word or phrase to make it italicized, written as `*italic text*` . Use two asterisks around a word or phrase to make it bold, written as `**bold text**` . Bold works well for a label you want to stand out, such as `**Acceptance criteria**` .

Bullets	Use a dash and a space or an asterisk and a space at the start of a line for a bullet list item, such as one of your acceptance criteria:
	`- Renames each .jpg file to include the date it was taken`
	`- Leaves the file extension unchanged`

Numbered steps	Use a number, a period, and a space for a numbered list when the order matters, such as steps someone should follow to test your tool.
	`1. First step.`
	`2. Second step.`

Vertical spacing	Leave a blank line between a heading and the text that follows it, and between one paragraph and the next. Markdown uses blank lines to tell where one block of text ends and another begins. Without that blank line, a list or a paragraph may not display the way you expect.

Code	Use a single backtick around a word or phrase to make it monospace, written as `` `monospace` ``. This is good for code fragments and filesystem names. For a block of code or terminal output, surround the block with triple-backticks with a language code to apply color syntax highlighting in preview:
	
	```` ```html
	<title>This is the title of this webpage.</title>
	``` ````

Tables	Lay out data by defining vertical-bar-separated column values and three-dashed rows: ``` Fruit|Price ---|--- Apples|US$2.99 Bananas|US$3.99

Table 1.6A: Styling common document elements using Markdown syntax

Here's a short brief written in Markdown, showing headings, bold text, and bullet lists together:

```
# Photo Renamer
## User and problem
Someone who imports **hundreds** of vacation photos and can never find
```

```
a specific one later.
## Smallest useful version
Rename each photo to include the date it was taken.
## Acceptance criteria
- Given a folder of `.jpg` files, renames each one to `YYYY-MM-DD-original
- Leaves files that already match this pattern *unchanged*
## Non-goals
- Sorting photos into folders
- Handling file types other than `.jpg`
```



Good practice : Keep a blank line above and below every heading and every list. This one habit prevents most Markdown formatting problems.

Create BRIEF.md in VS Code

You can create your brief directly in VS Code, the same editor you use to work with Claude Code:

1. In VS Code, open the **Explorer** panel on the left, showing the files in your project folder.
2. Right-click an empty area of the **Explorer** panel and choose **New File** .
3. Type `BRIEF.md` as the file name, including the `.md` ending, and press *Enter* .
4. Type your brief into the new file, using the headings, bold text, and bullet lists from the previous section.
5. Save the file with *Ctrl+S* on Windows or *Cmd+S* on a Mac.



Good practice : In VS Code, ensure that **File** | **Auto Save** is selected to automatically save changes to files.

You should now see `BRIEF.md` listed in the **Explorer** panel, with your text saved inside it.

Preview your Markdown file in VS Code

Before you hand your brief to Claude Code, it helps to see how the formatting will actually look, not just the raw text with its pound signs and asterisks:

1. Open `BRIEF.md` in VS Code if it isn't already open.
2. Click the preview icon in the top-right corner of the editor tab. It looks like a small page with a magnifying glass.
3. VS Code opens a second pane showing your brief with real headings, bold text, and bullet points, updated as you type in the file on the left.

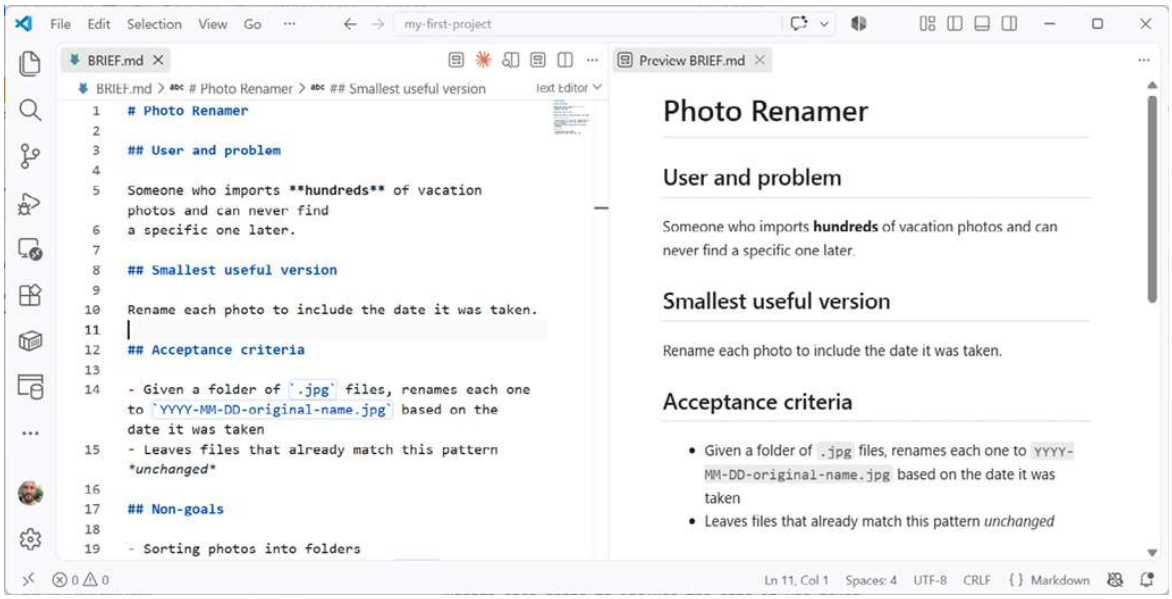


Figure 1.6C: VS Code showing a preview of BRIEF.md

If you don't see the preview icon, you can open the same view with *Ctrl+Shift+V* on Windows or *Cmd+Shift+V* on a Mac.



Watch out : A missing blank line above a list is the most common reason a Markdown file doesn't preview the way you expect. If a section looks wrong in the preview, check for a blank line between the heading and the list below it.

With Markdown syntax and a working preview, you can write a `BRIEF.md` that looks as clear to Claude Code as it does to you. The next section puts this into practice with your own idea.

Try it now

Turn one of your own ideas into a complete one-page brief:

1. Choose one small idea, real or imaginary, for a tool, website, or app.
2. Write one or two sentences naming the user and their problem.
3. Write one or two sentences describing the smallest useful version.
4. Write two or three acceptance criteria you could actually check.
5. Write two or three non-goals for this first version.
6. Save this as a file named `BRIEF.md` in a project folder, or keep it in your notes for now.



What to ask next : Once you have a draft brief, ask Claude Code to pressure-test it before you build anything:

```
Read this brief and tell me if any part is vague, missing,
```

Check the result

☐ Does your brief name a specific user and a specific problem, not just a feature?

☐ Could someone else check your acceptance criteria without asking what you meant?

☐ Have you written down at least two things you are deliberately not building yet?

☐ Would you feel comfortable handing this brief to Claude Code as a starting point?



Learn more online : Writing good acceptance criteria is a skill borrowed from professional software teams. If you want to go deeper than this chapter covers, search for "user story acceptance criteria" from a reputable software development source for further examples. For example, *What is acceptance criteria? Examples and best practices* (Atlassian): https://www.atlassian.com/work-management/project-management/acceptance-criteria .

With a brief written, you have a clear target and a way to check the result against it later. The next chapter sharpens how you actually ask for that result, turning a rough request into a prompt with the context, task, constraints, and checks Claude Code needs.

Chapter 1.7 Give Claude Instructions It Can Use, Then Plan Before It Edits

In this chapter, you will learn a simple structure for writing prompts, so your requests to Claude Code are easier to act on and easier to check. You will also practice asking Claude Code to inspect a project and propose a plan before it changes anything, splitting a larger task into small, safe steps you can review before a single file changes.

You do not need a project open for the first half of this chapter, though it helps to have one nearby to try these prompts for real. For the second half, you need a project folder open, ideally one with a few files in it already, such as the one from *Chapter 1.4* . By the end of this chapter, you should be able to spot the difference between a weak prompt and a strong one, and feel comfortable asking for a plan first on any task that touches more than one small file.

Supply context, task, constraints, and checks

A prompt that gets a useful result usually has four parts, even when they are only a sentence each:

- **Context** : What Claude Code needs to know before it starts, such as which file or folder you mean, or what already exists.
- **Task** : Exactly what you want done, in specific, concrete terms.
- **Constraints** : What to avoid or keep unchanged, such as “don’t add any new packages” or “keep the existing colors.”
- **Checks** : How you or Claude Code will know the result is right, such as “the page should still open without errors.”

You do not need to label these four parts every time you write a prompt. Once you know they exist, you will naturally include them, the same way a clear email covers who, what, and by when without formal headings.

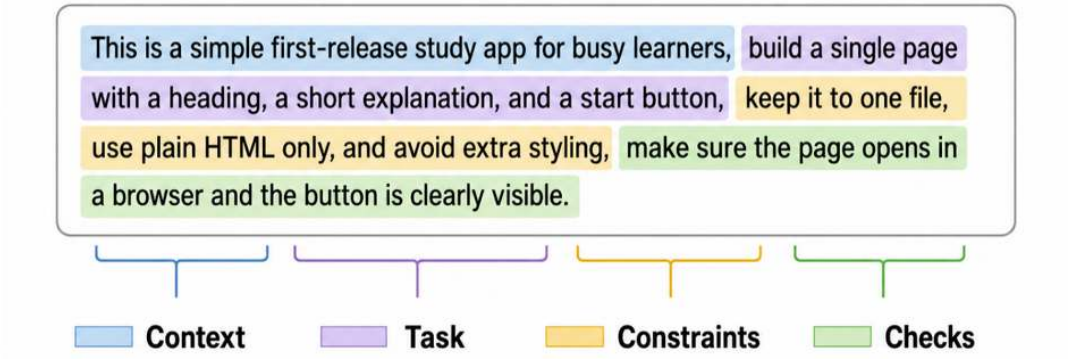


Figure 1.7A: A strong prompt specifies its context, task, constraints, and checks, even in a single sentence



Watch out : Packing several unrelated changes into one prompt, such as a new color scheme, a new feature, and a bug fix all at once, makes it hard to review the result and hard to undo just one part if something goes wrong. Keep requests focused on one outcome at a time.

Ask for explanations at your level

Claude Code adjusts how it explains things when you tell it what you already know. If a reply uses a term you do not recognize, say so instead of guessing or skipping past it.

Try phrases such as “explain that in plain language, I haven’t used this term before,” or “I understand HTML but not JavaScript yet, can you explain this part with that in mind.” Asking this way is not a sign you are behind. It is how you keep control of a project you are directing, rather than just approving whatever appears.

Replace vague prompts with testable requests

Compare these two requests:

- Weak: `Make the page look better.`
- Strong: `Increase the spacing between the heading and the paragraph, and make the heading a darker shade of blue. Leave everything else unchanged.`

The weak version leaves Claude Code guessing what "better" means, and leaves you unable to judge whether it succeeded. The strong version specifies an exact change, and gives you something specific to check once it's done: the spacing, the shade of blue, and that nothing else moved.

When a request feels hard to check afterward, that's usually a sign it needs to be more specific before you send it, not after you see the result.



Good practice : Before sending a prompt, ask yourself: "If Claude Code does exactly what I just asked, will I be able to tell whether it worked?" If the answer is no, add a detail or two until it is yes.



Real-world example : A hobbyist building a recipe-saving tool noticed his early prompts, like "make it nicer," produced results he then struggled to describe what was wrong with. Switching to specific requests, one visual change at a time, made every result something he could accept or reject in seconds, instead of staring at a page trying to name what still felt off.

Ask Claude to plan first

For a task bigger than the tiny build in *Chapter 1.4* , it helps to separate planning from doing. Ask Claude Code to read the relevant files and describe what it finds, before asking it to change anything:

```
Before you change anything, look at the files in this project and tell me
```

Many builders call this **Plan mode** : Claude Code researches and drafts a plan but waits for your approval before it starts editing. Look for a way to turn this on in whichever way you are running Claude Code, since the exact control can change between the terminal and the extension. The result is the same either way: you see the plan before a single file changes.

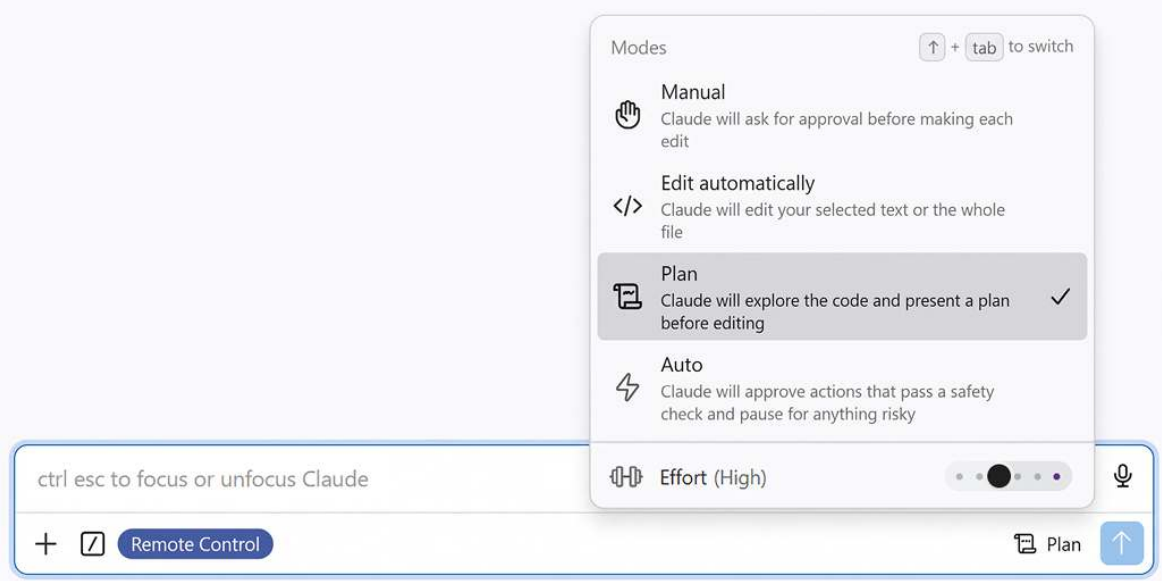


Figure 1.7B: Switching to Plan mode in Claude Code

Split work into safe increments

A good plan reads like a short numbered list, with each step small enough that you could reasonably review it on its own. “Add a contact form” is one step disguised as a whole project. “Add the form’s HTML fields, then style them, then add the validation, then connect the submit

button" is four steps, each one small enough to check before moving to the next.

If Claude Code proposes a plan with only one or two large steps, it is fair to ask for it to be broken down further. Smaller steps are easier to review, easier to test, and much easier to undo if one of them turns out wrong.

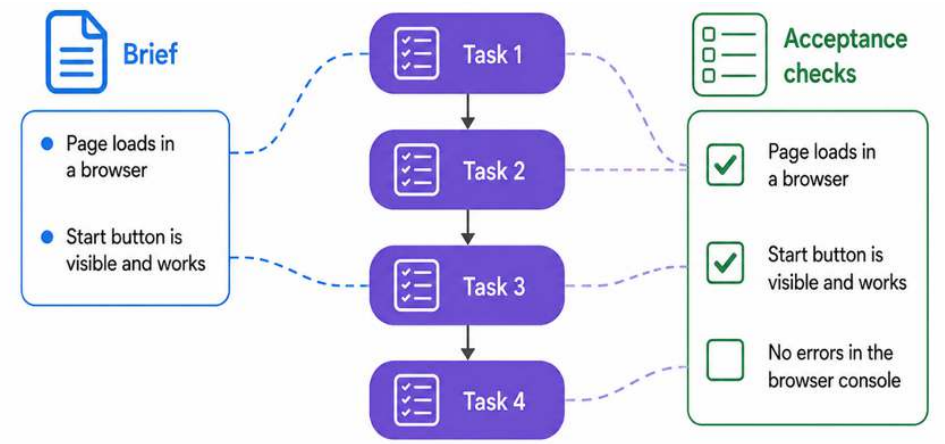


Figure 1.7C: A good plan connects back to your brief on one side, and to a checkable result on the other

Review and revise the plan

A plan is a draft, not a contract. Read each step and ask yourself whether it matches what you actually want, not just whether it sounds reasonable. If a step is missing, unclear, or solves the wrong problem, say so before approving anything:

```
Step 3 isn't quite right. I want the button to open a confirmation message
```

Revising a plan before any files change costs you a sentence or two. Revising after Claude Code has already written code across several files costs considerably more of your time.



Good practice : Read a proposed plan the way you would read a contractor's quote: check that every step is there, check that nothing surprising has been added, and ask about anything unclear before you say yes. Approving a plan you skimmed rather than read defeats the purpose of asking for one. If you find yourself approving plans without reading them, that is a sign to slow down, not a sign the habit isn't working.



Real-world example : A small-business owner asked Claude Code to add online booking to her site in one large request. The proposed plan, once she asked for it, showed the work split into six steps: a booking form, date validation, a confirmation screen, error handling, a summary email, and a test pass. Reviewing each step let her catch that step five would email her own address by default, something she wanted changed before any code was written.

Try it now

Practice both halves of this chapter: strengthen a vague prompt, then ask for a plan before anything changes:

1. Write one deliberately vague prompt about a change you would like to make to any project, real or imaginary.
2. Rewrite it using context, task, constraints, and checks.
3. Compare the two. Note what you added and why it matters.
4. If you have a project open, send the strong version and check the result against what you asked for.

5. Pick a small feature to add to an existing project, or imagine one for the project from *Chapter 1.4* .
6. Ask Claude Code to inspect the project and propose a plan, without editing anything yet.
7. Read every step of the plan.
8. Ask for at least one revision, even a small one, to practice the habit.
9. Only then approve the plan and let Claude Code proceed.



What to ask next : Practice turning a vague idea into a specific request by asking Claude Code to help:

```
Here's a rough idea: [describe it loosely]. Help me turn th
```

If a plan looks too large or too vague, ask directly:

```
Break this plan into smaller steps, and tell me what you wo
```

Check the result

☐ Does your strong prompt name a specific file, page, or part of the project?

☐ Does it say what to leave unchanged, not only what to change?

☐ Could you check the result afterward without guessing what "done" means?

☐ Did you ask for an explanation at your own level anywhere Claude Code used an unfamiliar term?

☐ Did you see a plan before any files changed?

- ☐ Was each step small enough that you could judge it on its own?
- ☐ Did you ask for at least one change to the plan before approving it?
- ☐ Can you explain, in your own words, what plan mode is for?

You can now write a prompt Claude Code can act on, ask for a plan before it touches a single file, and revise that plan until it matches what you actually want. The next chapter extends that control across every session this project will ever have, by giving it a memory of its own: a `CLAUDE.md` file that carries your instructions forward automatically.

Chapter 1.8 Give the Project a Memory

In this chapter, you will create a file called `CLAUDE.md` and use it to give Claude Code instructions that persist across sessions, instead of repeating yourself every time you open a new conversation.

You need a project folder open, ideally one you plan to keep working in across more than one sitting. By the end of this chapter, you should have a short `CLAUDE.md` file in your project, and understand what belongs in it and what does not.

Create CLAUDE.md

Each time you start a new Claude Code session, it begins without memory of your earlier conversations. A file named `CLAUDE.md`, placed at the top of your project folder, changes that. Claude Code reads this file automatically at the start of every session in that project, so anything written there does not need to be explained again.

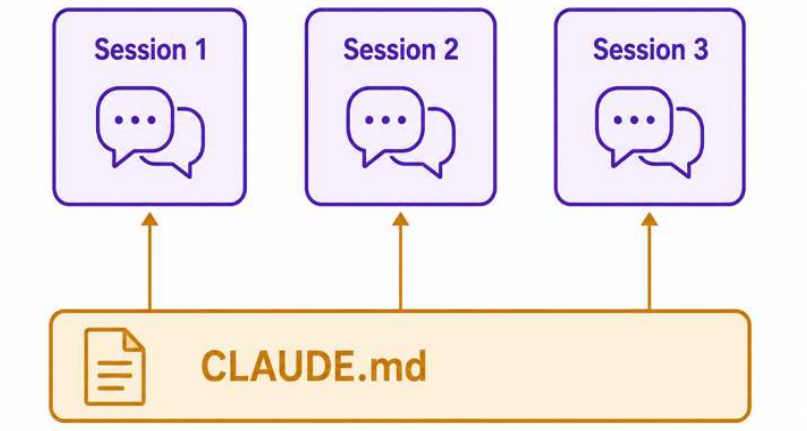


Figure 1.8A: Each session starts fresh, but CLAUDE.md gives every one of them the same starting instructions

The easiest way to create one is to ask Claude Code to do it for you:

```
Look at this project and create a starting CLAUDE.md file with what you fi
```

Review what it produces. A generated starting point is a draft, like everything else Claude Code makes, not a finished file.

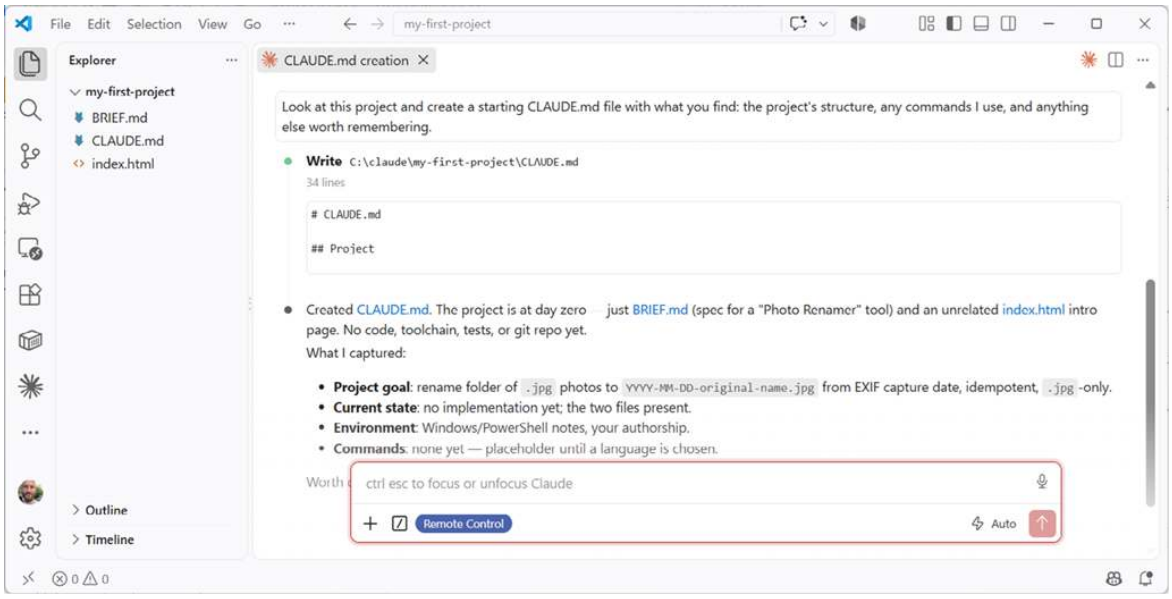


Figure 1.8B: Creating a CLAUDE.md file from an almost empty "day zero" project



Good practice : Update `CLAUDE.md` the moment you notice yourself repeating the same correction to Claude Code more than once. That repetition is the clearest sign something belongs in the project's memory instead of in every new conversation.

Record structure, commands, style, and done criteria

A useful `CLAUDE.md` stays short and specific. Good candidates include:

- The project's folder structure, in a sentence or two, so Claude Code doesn't need to rediscover it every session
- Commands you actually use, such as how to open a preview
- Style preferences that matter to you, such as "use plain HTML and CSS, no frameworks yet"
- What "done" looks like for this project, drawn from the acceptance criteria in your brief from *Chapter 1.6*

Keep the whole file under 200 lines at the very most, and usually well under a page for a small project. A short, specific file gets followed more reliably than a long one, the same way a short list of house rules gets remembered better than a long one.

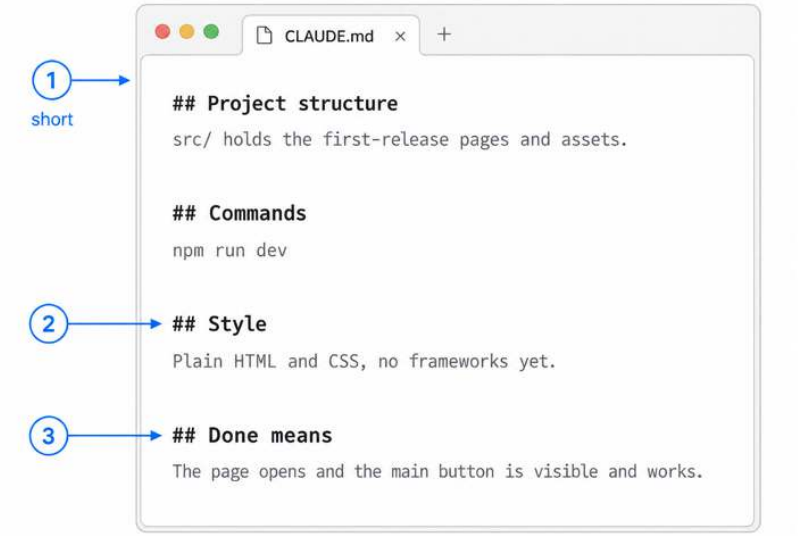


Figure 1.8C: A short CLAUDE.md, covering structure, commands, style, and what done looks like, is followed more reliably than a long one



Watch out : A `CLAUDE.md` file that has grown long and disorganized is easy to create by accident, one addition at a time. Skim it occasionally and remove anything outdated or no longer true. Outdated instructions can be followed just as readily as current ones.

Keep personal data and secrets out

`CLAUDE.md` is a project file, and later in this book you will save projects to Git and, for the website in *Book 2*, push them to GitHub. Treat `CLAUDE.md` the way you would treat any file you might eventually share: keep passwords, API keys, and personal information out of it entirely.

If you want private notes just for yourself, that should not travel with the shared project, the **Learn more online** link at the end of this chapter explains how to keep a personal file out of version control. For now, the rule is simple: if you would not want a stranger reading it, it does not belong in `CLAUDE.md`.



Real-world example : A hobbyist building a small collection of command-line tools kept forgetting to remind Claude Code, session after session, that he preferred short, single-purpose scripts over combined ones. Adding one line to `CLAUDE.md`, "keep each tool in its own file, doing one job," ended the repeated reminders for good.

Try it now

Give one of your own projects a `CLAUDE.md` file, then put it through a quick review:

1. Open a project folder, such as the one from *Chapter 1.4* or *Chapter 1.7* .
2. Ask Claude Code to create a starting `CLAUDE.md` for it.
3. Read the result and remove anything you do not want, such as guesses that are not quite right.
4. Add one instruction of your own, such as a style preference or a command you use often.
5. Confirm there is nothing personal or private in the file.



What to ask next : Once your `CLAUDE.md` exists, check that it actually says what you think it says:

```
Read the current CLAUDE.md file back to me, and tell me if
```

Check the result

- ☐ Does your project now have a `CLAUDE.md` file at its top level?
- ☐ Does it describe the project's structure in a sentence or two, not a full essay?
- ☐ Does it include at least one instruction you added yourself?
- ☐ Have you checked it for anything personal, private, or secret, and removed it if found?



Learn more online : `CLAUDE.md` supports more advanced options than this chapter covers, including personal files that stay off your team's shared copy, and organizing longer projects into several topic files. See https://code.claude.com/docs/en/memory for the current details.

Your project now remembers your instructions between sessions. The next chapter makes sure your files remember their own history too, using Git to save checkpoints you can always return to if something goes wrong.

Chapter 1.9 Review Changes and Save Checkpoints with Git

In this chapter, you will learn to read a diff, accept or reject a specific change, and use Git to save a checkpoint of your project you can return to later. This is the safety net that lets you try things confidently, because a mistake can be undone. You will learn the concepts in theory first, then you will initialize a Git repository and make a tracked change.

You need a project folder with at least one file in it, such as the one from *Chapter 1.4* . By the end of this chapter, your project will be tracked by Git, with at least one meaningful checkpoint saved.

Read a diff

A **diff** shows exactly what changed between two versions of a file, usually with removed lines marked in red and added lines marked in green. When Claude Code proposes an edit, VS Code can show you this comparison directly in the editor, right where the change happens, instead of asking you to compare two files by eye.

Reading a diff is faster than it sounds. You are not reading every unchanged line again. You are scanning for the red and green, and asking whether each change matches what you expected:

1. A one-line color change should show one red line and one green line, not twenty of each.
2. When committing a change, write a short, specific commit message describing what changed, such as “Add heading and paragraph to homepage,” not just a vague “updates.”

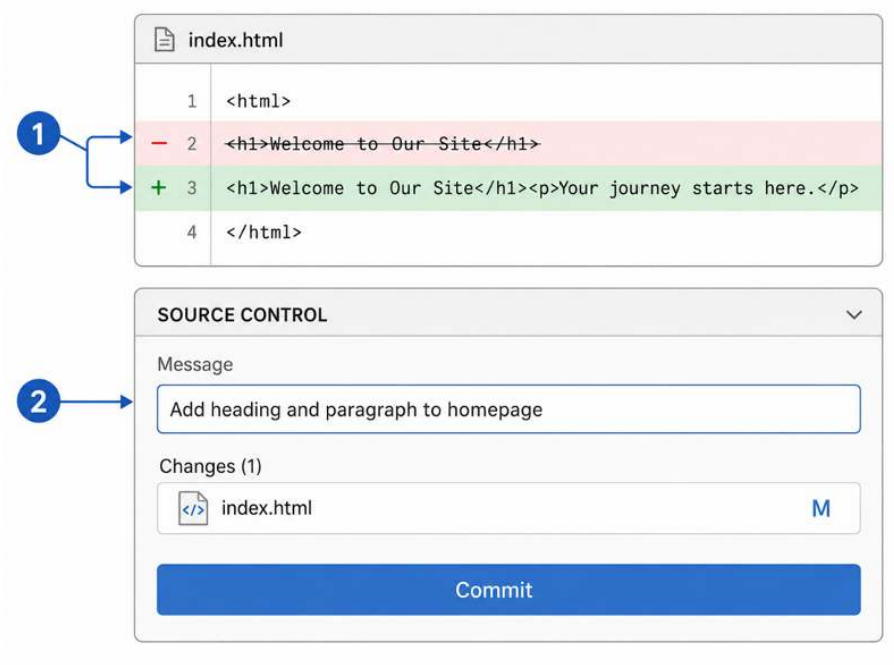


Figure 1.9A: Red and green show exactly what changed. Writing a specific commit message turns that change into a checkpoint you can trust later.

Accept or reject a focused change

Not every change in a diff needs the same answer. VS Code lets you accept or reject changes individually, not only as an all-or-nothing block. If four lines changed and only one looks wrong, you can reject that one line and keep the rest.

This is useful because Claude Code sometimes gets most of a change right and one detail wrong. Rejecting the whole thing and starting over wastes the parts that were already correct. Reviewing at this level of detail is part of what keeps you in control of a project, even as Claude Code writes most of the code.

Git commits and repositories

Git is a tool that tracks changes to your files over time, so you can save a checkpoint and return to it later if something breaks. A saved checkpoint

is called a **commit** , and a folder Git is tracking is called a **repository** , or **repo** for short.

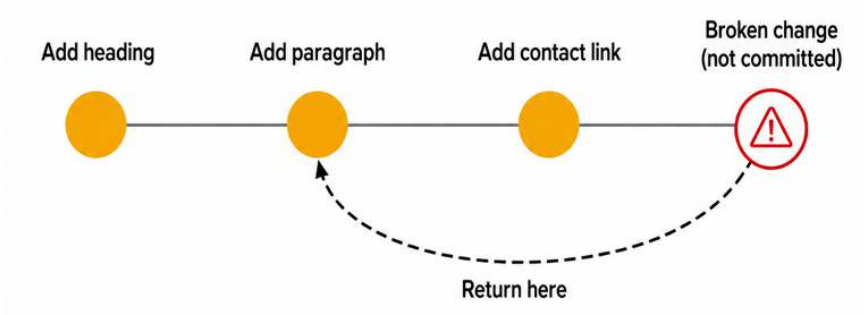


Figure 1.9B: Each commit is a safe checkpoint. A broken change you haven't committed can be set aside in favor of the last one that worked.



Good practice : Commit at natural stopping points: after a feature works, after a bug is fixed, or before you try something risky or experimental. Frequent, small, well-described commits are far more useful later than one giant commit at the end of a session.

You will now turn your project folder into a Git repo and save your first checkpoint. You can do this by asking Claude Code, by using VS Code's Source Control view, or by typing commands directly in the terminal from *Chapter 1.3* . All three do the same job. Pick whichever feels most natural, and switch between them any time.

Download and install Git for Windows

If you are using Windows, you might need to download and install Git:

1. Use your preferred browser to download Git for Windows: https://git-scm.com/install/windows
2. Run the installer. When asked, choose the following options:
    - Use VS Code as Git's default editor.

- Override the default branch name for new repositories. Enter `main` instead of `master` .
- Use Git from the command line and also from 3rd-party software.
- Use the default for all other options and then select **Install** .

3. Select **Finish** .
4. Restart VS Code and any terminal or command prompt windows.

Initialize Git in your project

Ask Claude Code:

```
Initialize Git in this project and confirm it worked.
```

Or, in VS Code:

1. Open the **Source Control** view from the icon on the left side of the window. It looks like three connected dots. You can press *Ctrl+Shift+G* on Windows.
2. Click the **Initialize Repository** button:



Figure 1.9C: Source Control in VS Code

Or, in the terminal, from your project folder:

```
git init
```

In VS Code, you should now see a **Source Control** icon with a small number badge, showing how many files have changes Git has noticed but not yet saved, as shown in *Figure 1.9D* .

Put your files in a good, working state

Before you stage anything, open `index.html` from *Chapter 1.4* in your browser and confirm the heading and paragraph still look right. A checkpoint saved from a broken file is not a checkpoint worth having.

Your project folder likely has more than one file in it by now. Alongside `index.html` , you should also have `BRIEF.md` from *Chapter 1.6* and `CLAUDE.md` from *Chapter 1.8* . Open `CLAUDE.md` and confirm, once more, that it contains nothing personal or private, since this commit is about to give it a permanent place in your project's history.

Stage the changes you want to save

Staging tells Git which changes belong in the next checkpoint. Since `BRIEF.md` , `CLAUDE.md` , and `index.html` were all created before Git started tracking this project, they'll all appear as new, unstaged files. This first checkpoint is a good place to capture all three together, since they represent where your project already stands, not three unrelated changes.

In VS Code, as shown in *Figure 1.9D* :

1. In the **Source Control** view, you should see `BRIEF.md` , `CLAUDE.md` , and `index.html` listed under **Changes** .
2. If you want to stage a single file, then hover over the file name and click the **+** icon that appears beside it. This stages the individual file.

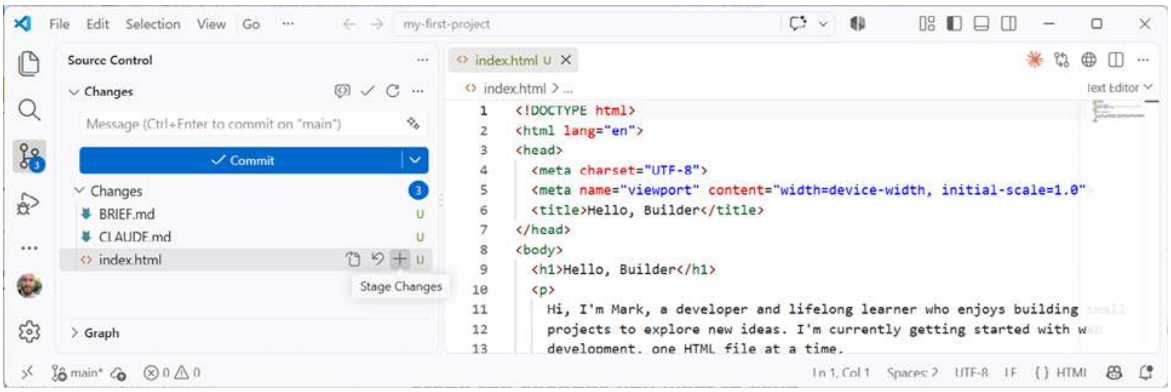


Figure 1.9D: Staging the index.html webpage

3. Hover over **Changes** itself, rather than a single file, and click the **+** icon that appears. Its tooltip should say **Stage All Changes** . This stages all three files at once.
4. The three files should now move from **Changes** to **Staged Changes** .

If you would prefer, you can ask Claude to stage the files for you:

```
Stage BRIEF.md, CLAUDE.md, and index.html in Git.
```

Alternatively, you can stage the files using the CLI in the terminal:

```
git add BRIEF.md CLAUDE.md index.html
```



Good practice : Stage one finished piece of work at a time. These three files aren't unrelated changes bundled together, they're the actual starting point of your project, so committing them together is the right call. Going forward, when you touch two unrelated things in the same session, stage and commit them separately, with their own commit messages.

Write a specific commit message and save the commit

In the **Source Control** view, type your message into the text box above the **Commit** button. For the homepage you built in *Chapter 1.4* , a good first commit message is:

```
Add project brief, CLAUDE.md, and homepage
```

Avoid a vague message like `updates` or `changes` . Months from now, a specific message tells you exactly what that checkpoint contains without opening the file to check.

In VS Code, click the **Commit** button above the message box (it may show a checkmark icon).

Or ask Claude Code:

```
Commit the staged changes with the message "Add project brief, CLAUDE.md,
```

Or, in the terminal:

```
git commit -m "Add project brief, CLAUDE.md, and homepage"
```

You should now see **Staged Changes** empty in the **Source Control** view, and your commit listed if you click the **Graph** or history icon at the top of the panel. All three files belong to that one checkpoint.

Practice with a small change

Practice the whole loop once more with a small, visible edit:

1. Open `index.html` and change the paragraph text, such as replacing your placeholder introduction with one real sentence about yourself.
2. Save the file, then open it in your browser to confirm the new text appears.
3. In the **Source Control** view, click `index.html` under **Changes** to review the diff. Confirm you only see the one sentence you meant to change.

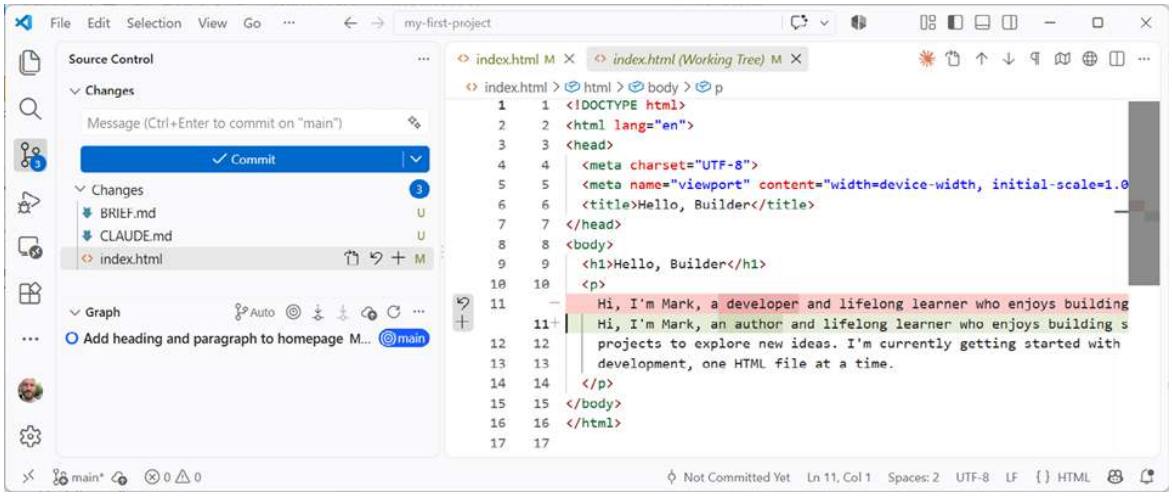


Figure 1.9E: Viewing the diff for index.html

4. Stage the file, then commit it with a message such as:

```
Update paragraph wording on homepage
```

You now have two commits: one that captures your project's starting point, brief, `CLAUDE.md`, and homepage together, and one that captures a single, specific edit to it. That's exactly how a commit history should grow: a broader checkpoint for what already existed, then small, focused, well-described steps from there on, each one something you could explain to someone else without opening a single file.



Watch out : A commit is only a safe return point for files it includes. If you forget to stage a file before committing, Git cannot protect changes to it. Check what a commit includes before you rely on it as your safety net.



Real-world example : A career changer building his first small tool made an experimental change that broke something he could not immediately figure out how to fix. Because he had committed a working checkpoint ten minutes earlier, he asked Claude Code to help him return to that commit, rather than trying to manually undo an unclear tangle of changes:

```
I broke something with my last change and can't figure out
```

Try it now

Turn your project into a tracked Git repository and save one real, reviewed commit:

1. Ask Claude Code to initialize Git in your project, if it is not already tracked.
2. Make a small, visible change to a file, such as the heading text from *Chapter 1.4* .
3. Review the diff before accepting it.
4. Stage the change in the **Source Control** view.
5. Write a specific commit message and save the commit.



What to ask next : Before committing, it helps to see the story your commits will tell later:

```
Show me the changes since my last commit, summarized in pla
```

Check the result

- ☐ Does your project now have Git tracking it?
- ☐ Did you review the diff, red and green, before accepting the change?
- ☐ Does your commit message describe what changed, specifically enough to understand months from now?
- ☐ Can you find your new commit listed in the Source Control view's history?

With Git tracking your work, a mistake stops being something to fear. The next chapter gives you a calm, repeatable way to handle the moment something actually breaks, working through one cause at a time instead of guessing.

Chapter 1.10 Debug and Test One Cause at a Time

In this chapter, you will practice a calm, repeatable way to handle something that breaks: reproduce it, collect what you know, and ask Claude Code for one hypothesis and one small fix at a time, rather than guessing wildly or asking for a full rewrite.

You need a project you can experiment with, such as the page from *Chapter 1.4* . If nothing is currently broken, you will introduce a small problem so you have something real to practice on. By the end of this chapter, debugging should feel like a routine event that you know how to handle, not an emergency.

Reproduce the problem

Before asking Claude Code to fix anything, make sure you can make the problem happen again, on purpose, whenever you want. “It sometimes doesn’t work” is much harder to fix than “clicking the button a second time does nothing, every time I try it.”

Write down the exact steps that trigger the problem. If you cannot reliably reproduce it yet, spend a few minutes narrowing down when it happens before asking for help. A problem you can reproduce on demand is a problem you can also confirm is actually fixed later.

Add a bug on purpose

Before you can practice reproducing, collecting, and fixing a problem, you need a real one to work with. Here’s how to add a small, safe bug to the

page from *Chapter 1.4* , so you have something genuine to debug instead of an imaginary example.

The page from *Chapter 1.4* only has a heading and a paragraph, nothing yet that can actually misbehave. Give it one small feature first: a link that jumps back to the top of the page:

1. Ask Claude Code:

```
Add an id of "top" to the heading in index.html, and add a new
```

2. Open `index.html` in your browser once Claude Code confirms the change.
3. Scroll down to the bottom of the webpage and then click **Back to top** .

You should now see the page jump to the heading when you click the link. This is your working baseline, the known-good state you'll return to once the bug is fixed.

Break it on purpose

Now, introduce a small, realistic mistake, such as a link that points at the wrong place:

1. Open `index.html` in VS Code.
2. Find the link you just added. It should look something like `<a href="#top">Back to top</a>` .
3. Change the `href` value so it no longer matches the heading's `id` , for example `href="#tpo"` with the letters swapped as a deliberate typo.
4. Save the file.

This is exactly the kind of bug Claude Code sometimes introduces on its own: one character wrong, everything else fine. It's also a bug with no

error message at all, which makes it a good first case to practice on.



Watch out : Not every bug shows an error message. A broken link like this one fails silently: nothing turns red, nothing appears in the browser's console. The only evidence is that the page doesn't do what you expected. Get comfortable noticing that kind of problem too, not just the ones that announce themselves.

Work the loop on your own bug

With a real, reproducible bug in front of you, walk through the same steps from this chapter.

- **Reproduce it** : Reload `index.html` in your browser, and click **Back to top** . Confirm that nothing happens every time you click it.
- **Collect what you know** : There's no error message here, so write down what actually happened ("Clicking the link does nothing") and what you expected instead ("The page should jump up to the heading").

Ask for a hypothesis before a fix:

```
Clicking the "Back to top" link in index.html does nothing. There's no err
```

Read the hypothesis Claude Code gives you. It should point at the mismatch between the link's `href` and the heading's `id` , the exact thing you changed a moment ago.

Ask for the smallest fix, then verify:

```
Please fix it with the smallest possible change.
```

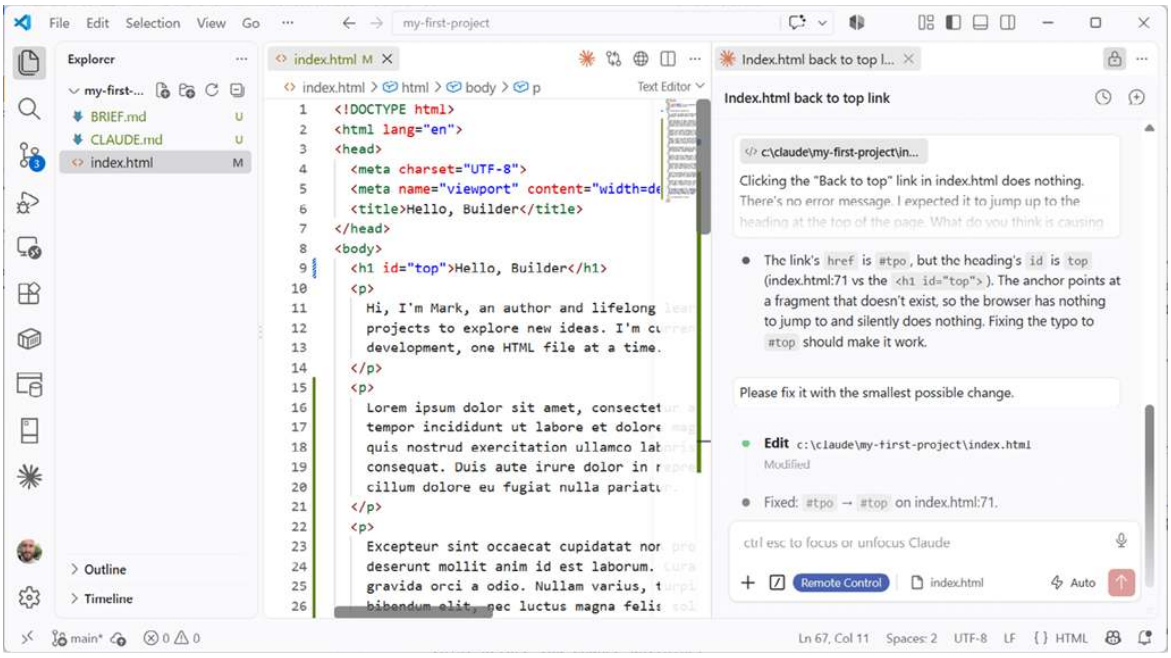


Figure 1.10A: Fixing a bug

Reload the page and click **Back to top** again. You should now see the page jump to the heading, just like it did before you introduced the bug.



Good practice : Once a bug is fixed, check it the same way you checked it while it was broken, not by reading the new code and assuming it's right. If clicking the link is how you noticed the problem, clicking the link is how you should confirm it's gone.

You now have a complete example of the reproduce, collect, hypothesize, fix, verify loop, applied to a bug you introduced and fixed yourself from start to finish. The **Try it now** exercise at the end of this chapter asks you to repeat the same loop on a bug of your own choosing, without a worked example to lean on.

Collect the error and expected behavior

To fix a more realistic bug, gather two things before you ask Claude Code for a fix: what actually happened, and what you expected instead.

"What actually happened" might be an error message from your browser's console, a test failure, or simply a wrong result on screen. Copy the exact wording of any error message rather than paraphrasing it. Small details in an error message often point directly at the cause.

"What you expected instead" is easy to forget to state, but it matters just as much. "The total should show $45, not $54" tells Claude Code exactly what success looks like, the same way an acceptance criterion does in your build brief.

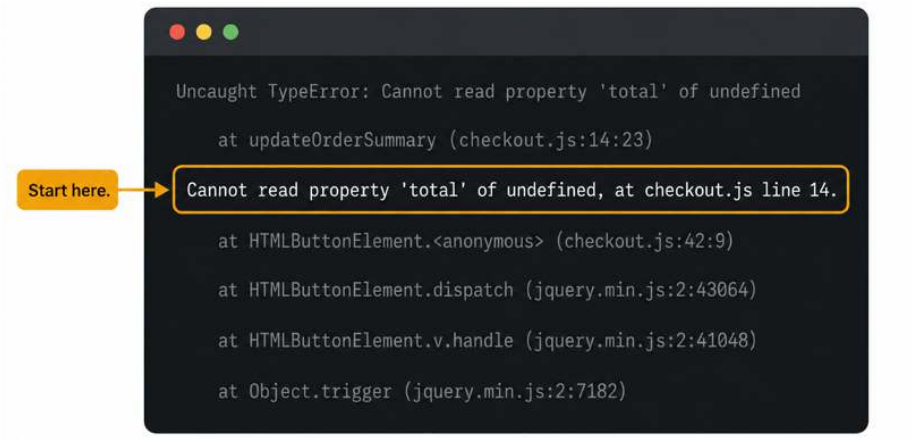


Figure 1.10B: Most of an error message is context. One line usually points at the actual problem, and it's worth learning to spot it.

Ask for a hypothesis, small fix, and test

Instead of asking for a fix outright, ask Claude Code to explain what it thinks is causing the problem first.

```
Here's what happens: [steps to reproduce]. Here's the error: [exact messag
```



Good practice : Ask for the smallest fix that addresses the stated cause, not a broader rewrite "while we're in here." A small, focused fix is easier to verify and far easier to undo if the hypothesis turns out to be wrong.

A stated hypothesis, such as "the total is being calculated before the discount is applied," gives you something to evaluate before any code changes. Once the hypothesis makes sense to you, ask for the smallest fix that addresses it, then test by reproducing the original problem again to confirm it's actually gone, not just changed.

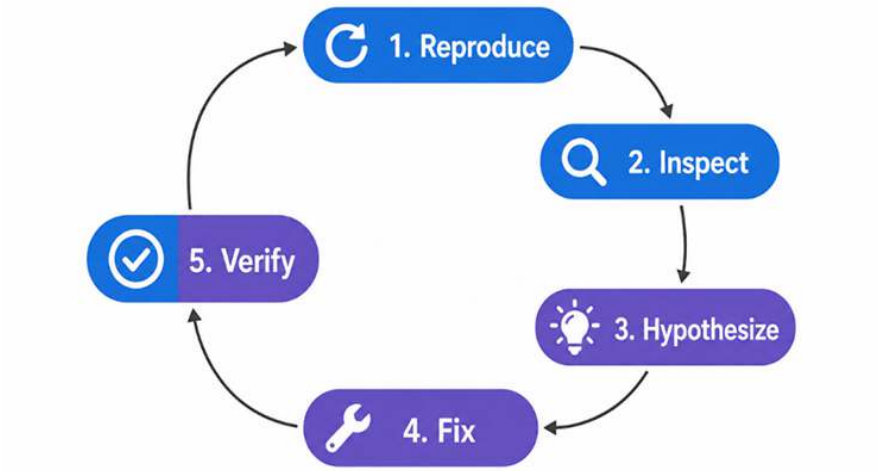


Figure 1.10C: Debugging follows its own small loop: reproduce the problem, understand it, propose a cause, fix it, then verify the original problem is actually gone



Watch out : A fix that makes the specific symptom disappear is not the same as a fix that addresses the actual cause. If you cannot explain, in your own words, why a fix worked, ask Claude Code to explain the connection between the cause and the change before you accept it.



Real-world example : A student's small tool crashed only when given an empty input, something she had never tested herself. Reproducing the crash on purpose, then sharing the exact error message with Claude Code, led to a hypothesis in one exchange: the tool wasn't checking for empty input before trying to use it. A two-line fix, tested against the same empty input again, resolved it.

Try it now

Run the reproduce, hypothesize, fix, verify loop on a bug of your own choosing, this time without a worked example to lean on:

1. Introduce a small, deliberate problem into a project, such as changing one word in a file so a link stops working, or renaming a file Claude Code expects to find.
2. Reproduce the problem and write down the exact steps.
3. Collect the exact error message, if there is one, and write down what you expected instead.
4. Ask Claude Code for a hypothesis before a fix.
5. Once you understand and accept the hypothesis, ask for the smallest fix, then reproduce the original problem again to confirm it's resolved.



What to ask next : Once a bug is fixed, it's worth checking whether the fix could break something else:

```
Now that this is fixed, are there other situations similar
```

Check the result

☐ Could you reproduce the problem reliably before asking for help?

☐ Did you collect the exact error message, not a paraphrase?

☐ Did Claude Code explain a hypothesis before changing anything?

☐ Did you confirm the fix by reproducing the original problem again, and seeing it resolved?

You now have a routine for handling the moment something breaks. The next chapter looks more closely at what Claude Code asks your permission for, and how to keep passwords, keys, and your own judgment protected as your projects grow.

Chapter 1.11 Protect Secrets and Question Generated Code

In this chapter, you will look more closely at Claude Code's permission requests, learn how to keep passwords and API keys out of your project and out of Git, and build the habit of questioning generated code instead of trusting it automatically.

You need a project folder, ideally one with Git already tracking it, from *Chapter 1.9* . By the end of this chapter, you should understand what a permission request is actually protecting you from, and know where secrets belong instead of inside your code.

Understand permissions

Since *Chapter 1.4* , you've seen Claude Code check its own actions before taking them. Auto mode, the default, runs every action through a safety check first. Reading files inside your project almost always passes that check right away, since reading cannot change anything. An action that would be hard to undo, such as deleting a file, or one that reaches outside your project, such as sending data to an outside service, is far more likely to make Claude Code stop and ask, or to find a safer way to do the same job instead.

If several actions in a row get stopped this way, Claude Code switches to asking about everything for the rest of that session, the same way Manual mode always works. This gives you a chance to see what's actually being proposed instead of hitting the same block over and over.

When Claude Code does ask permission to run a command, read what it's actually asking to do before approving it, the same way you would

read a plan before accepting it in *Chapter 1.7* . You can usually approve a specific, familiar command permanently for a project, so you are not asked again for that exact action, while anything unfamiliar still stops and asks.

If you would rather see every request yourself, switch to Manual mode from the mode indicator you used in *Chapter 1.4* . Manual mode asks before every file change and every command, with no judgment calls made on your behalf.

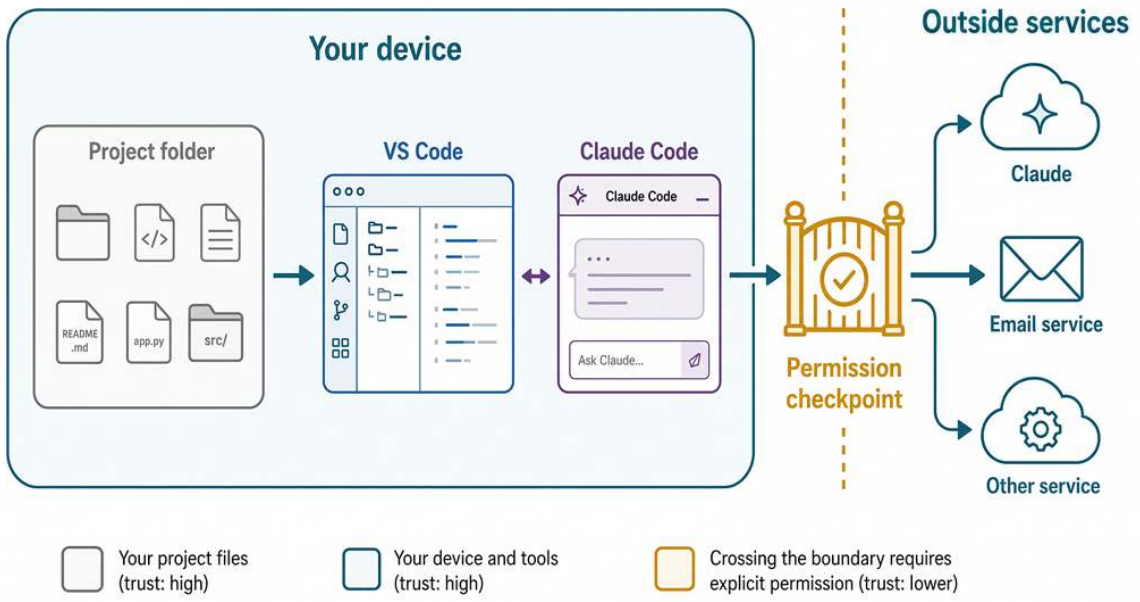


Figure 1.11A: Most of what happens stays inside your project and your device. The amber gate marks where something reaches outside, and where permission matters most.

Watch out : A permission request you don't understand is not something to approve out of impatience. If a proposed command looks unfamiliar, ask Claude Code to explain what it does in plain language before you decide.



Learn more online : Claude Code's permission system has more configuration options than this chapter covers, including how to set a default mode and ways to approve or block specific commands ahead of time. See https://code.claude.com/docs/en/permissions for the current details.

Keep keys out of code and Git

An API key is like a password that lets your project use an outside service, such as sending an email or fetching weather data. If a key ends up inside a file that gets committed to Git, and later pushed to GitHub in *Book 2* or *Book 3*, it can become visible to anyone, even if you later delete it, since Git remembers old versions.

The standard fix is a file usually named `.env`, kept outside version control, that holds keys and other sensitive values separately from your actual code. A file named `.gitignore` tells Git which files to leave out of every commit entirely, and `.env` almost always belongs on that list:

```
Set up a .env file for any keys this project needs, and make sure .gitigno
```

Most of the projects in this book do not need external services yet, so you may not create a real `.env` file for some time. Knowing the pattern now means you will recognize it immediately when *Book 2* or *Book 3* introduces a service that does need one.

If you ask Claude Code to set up a `.env` file in your current project, it will likely refuse, although it could create a `.gitignore` file pre-emptively ignoring `.env` and `.env.*` (but allowing a committed `.env.example` if you ever want one to document expected variables):

```
# Environment / secrets
.env
.env.*
!.env.example
```

Check packages, privacy, security, and unsupported claims

Before accepting a new package into your project, it is fair to ask a few questions: what does it actually do, is it still maintained, and does the project need it. Claude Code can answer these questions directly:

```
Before adding this package, explain what it does, whether it's actively ma
```

The same questioning habit applies to explanations, not only code. If Claude Code states something as fact, such as a claim about a service's pricing, a law, or a security guarantee, treat it as a claim to verify rather than a settled answer, especially where money, privacy, or safety are involved.

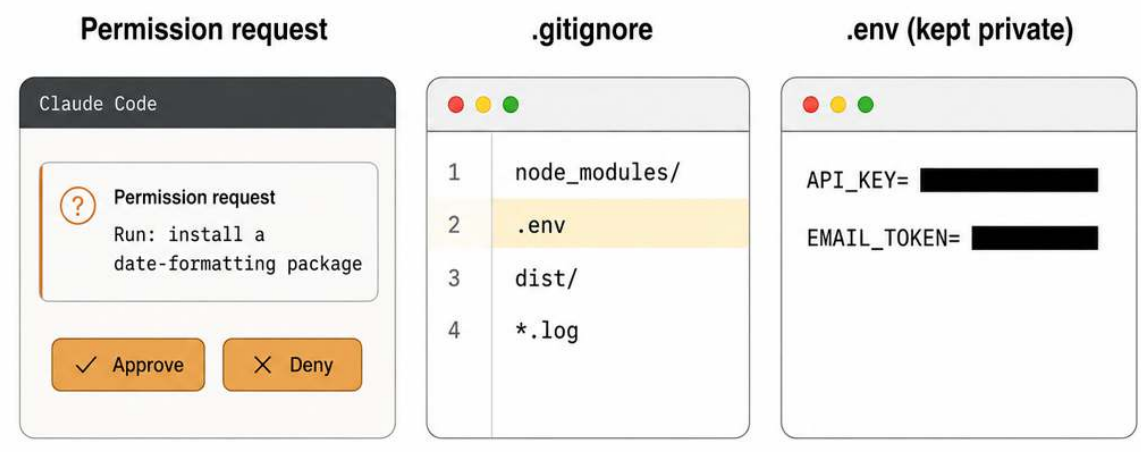


Figure 1.11B: A permission request, a .gitignore entry, and a redacted .env file: three small habits that keep secrets out of your project's shared history



Good practice : Before typing anything into a prompt, ask yourself whether you would be comfortable with that text appearing in a public place. Treat prompts the same way you would treat information you are about to paste into an email to a stranger.



Real-world example : A freelancer testing a new package for handling dates asked Claude Code whether it was actively maintained, out of habit built from this chapter. The answer revealed the package had not been updated in several years, and a well-maintained alternative did the same job. Ten seconds of asking avoided building on something fragile.

Try it now

Give one of your own projects a quick secrets-and-dependency check:

1. Open a project with Git tracking it.
2. Ask Claude Code whether it can see any files that look like they contain secrets, passwords, or personal data.
3. If your project uses, or will soon use, an API key, ask Claude Code to set up a `.env` file and confirm `.gitignore` excludes it.
4. Pick one package already in your project, if any, and ask Claude Code whether it is actively maintained.



What to ask next : Once your project is set up with secrets kept separate, it's worth a periodic check:

```
Scan this project for anything that looks like a passw
```

Check the result

- ☐ Can you explain, in your own words, why some actions ask permission and others do not?
- ☐ Do you know where an API key should live in a project, and where it should never live?
- ☐ Does your `.gitignore` file exclude anything that should stay private?
- ☐ Can you name one question worth asking before adding a new package to a project?

You have covered every habit this book set out to teach: writing a brief, giving clear instructions, planning before editing, keeping a project's memory, reviewing diffs, saving checkpoints, debugging calmly, and questioning what you approve. The final chapter puts all of it to work on one project, built from a blank folder to a finished tool.

Chapter 1.12 Project - Build a Tool You Would Use

This chapter is your first complete project. You will decide what a small command-line tool should do, build it with Claude Code using everything from this book so far, and manage it with Git from the first commit to the last. There is no new skill to learn here. There is a project to finish.

You need everything from this book up to this point: comfort with the terminal, a habit of writing a brief, a habit of asking for a plan, a `CLAUDE.md` file, and a habit of committing checkpoints as you go. Budget more time for this chapter than earlier ones, at least an hour, possibly spread across more than one sitting.

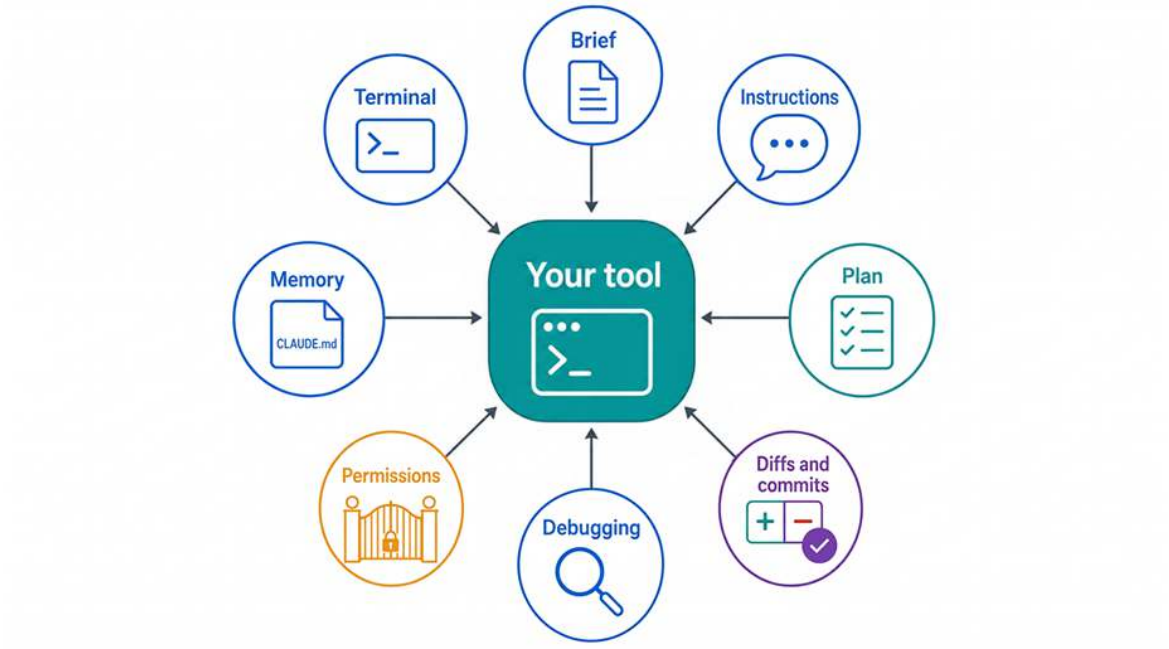


Figure 1.12A: Every skill from this book, from writing a brief to committing a checkpoint, comes together in one finished tool

Decide what the command-line tool will do

A command-line tool is a small program you run by typing a command, rather than clicking through a graphical app. It reads some input, does one job, and shows you a result. That makes it a good first project: small enough to finish, real enough to actually use afterward.

If you already have an idea, use it. If not, here are a few starting points that suit a first tool well:

- A word counter that reports how many words, sentences, and characters are in a text file you give it.
- A simple to-do list you can add to, list, and check off from the terminal.
- A unit converter for something you personally use often, such as recipe measurements or currency estimates.
- A journal logger that appends a dated entry to a file each time you run it.

Whichever idea you choose, write a brief for it now, using the structure from *Chapter 1.6* : the user and problem, the smallest useful version, two or three acceptance criteria, and two or three non-goals. If you wrote a brief back in *Chapter 1.6* for an idea like this, you can reuse it here.

You should end up with a `BRIEF.md` file, like one of the following:

- **Word Counter** : Reports words, sentences, and characters for a text file, with criteria that force it to handle a one-word file and an empty file, not just a normal one: https://github.com/markjprice/claude-code-vb/blob/main/files/book-1/word-counter/BRIEF.md

- **To-do List** : Add, list, and mark tasks done from the terminal, with due dates, syncing, and permanent deletion pushed out as non-goals: https://github.com/markjprice/claude-code-vb/blob/main/files/book-1/todo-list/BRIEF.md
- **Unit Converter** : One category you actually use (recipe measurements or a travel currency), with an explicit non-goal of covering every unit, so the brief doesn't quietly grow into a universal converter: https://github.com/markjprice/claude-code-vb/blob/main/files/book-1/unit-converter/BRIEF.md
- **Journal Logger** : Appends a dated entry to a single file, with criteria covering both the file-doesn't-exist-yet case and the same-day-twice case: https://github.com/markjprice/claude-code-vb/blob/main/files/book-1/journal-logger/BRIEF.md



Watch out : A project without any non-goals tends to grow without end. If you notice yourself adding features that weren't in your original brief, pause and ask whether they belong in this first version, or in a clearly separate next one.

Manage source control with Git

Treat Git as part of building the tool, not something you do afterward:

- Initialize Git early, before you have anything you'd be upset to lose
- Commit after each working increment, with a specific message describing what that checkpoint contains
- If an experiment goes badly, use an earlier commit as your way back, instead of trying to manually undo a tangle of changes
- Before you consider the tool finished, look back through your commit history. It should read like a short, sensible story of how the

tool came together

Build the tool with Claude Code

Work through the same loop you have practiced all through this book, one small increment at a time:

1. Create a new project folder and open it in VS Code.
2. Add your `BRIEF.md` to the project folder.
3. Ask Claude Code to create a starting `CLAUDE.md`, based on your brief.
4. Ask Claude Code to inspect the brief and propose a plan, broken into small steps, before it edits anything.

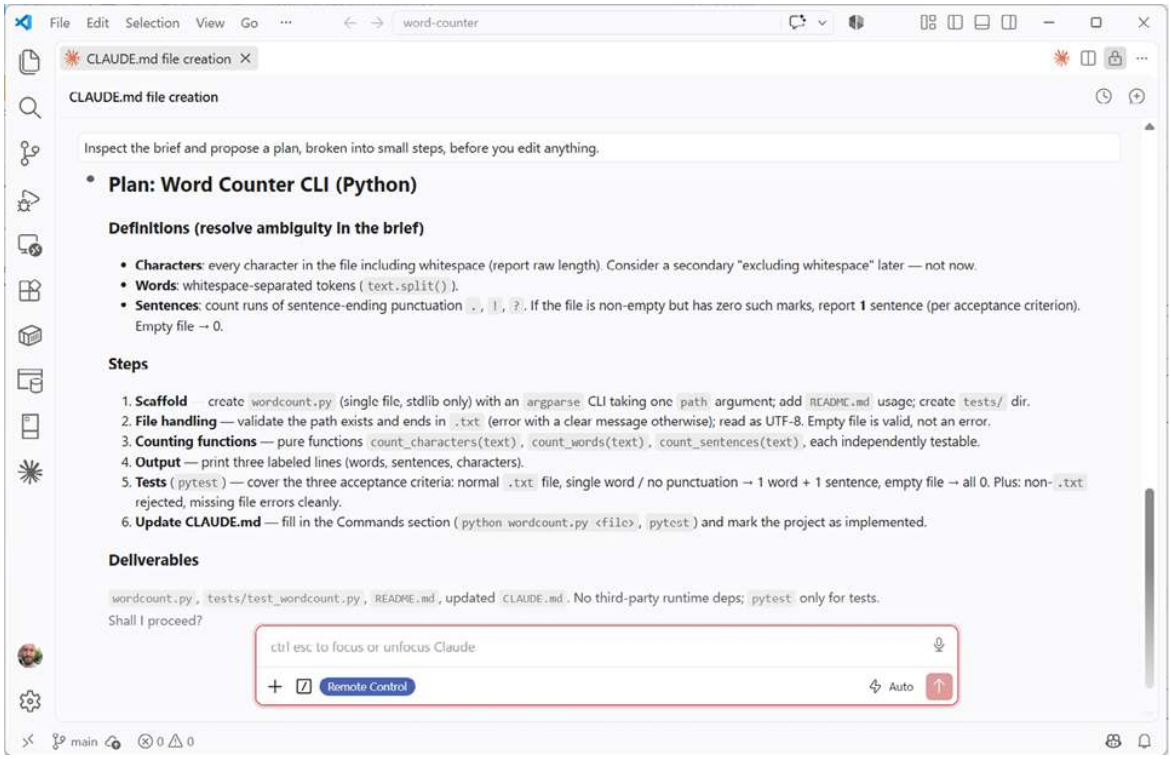


Figure 1.12B: Claude Code's plan for a word counter command-line tool

5. Review the plan. Revise anything that doesn't match what you actually want.

6. Approve the plan, and then review each permission request, if any, and each diff as the tool takes shape.

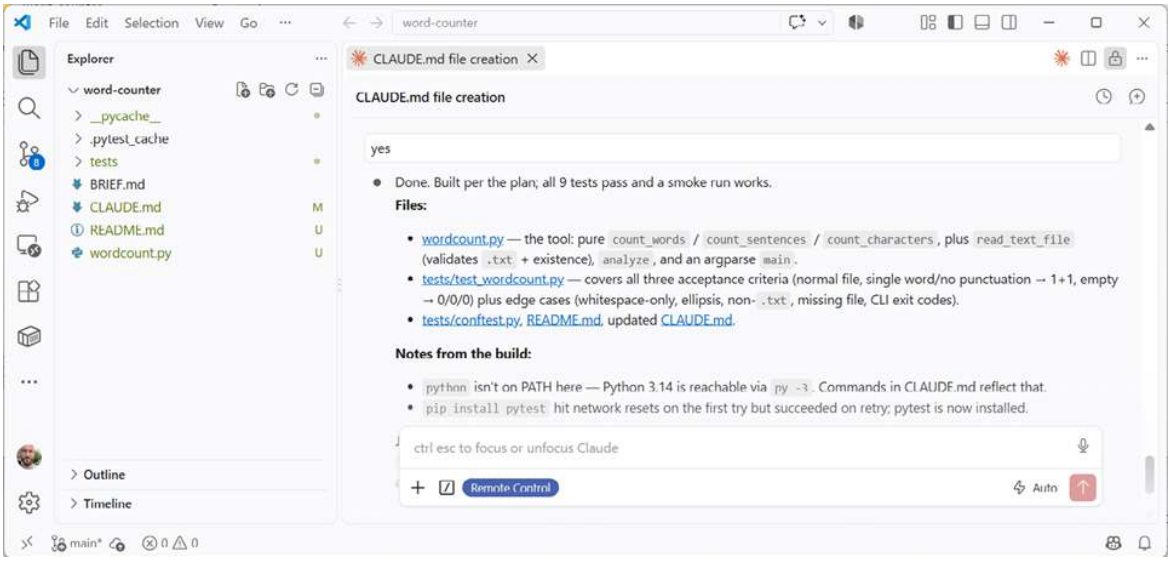


Figure 1.12C: Finished word counter project

7. Commit the initial project implementation.
8. Read the `README.md` file to learn how to run the tool.
9. Run the tool yourself after each meaningful step, checking it against your acceptance criteria, not just trusting that it looks right.



Figure 1.12D: Testing the word counter in VS Code Terminal with a sample text file

10. If something breaks, use the reproduce, hypothesize, fix, verify pattern from *Chapter 1.10* instead of asking for a full rewrite.

Resist the urge to ask for everything at once. The whole point of this chapter is practicing the loop on a real project, not skipping straight to a finished result.



Figure 1.12E: A small, working tool, built one reviewed step at a time, doing exactly the job its brief described



Good practice : Run your tool against its acceptance criteria from your brief, not just against whatever input you happen to type first. A word counter that works on one sentence should also be checked against an empty file and a very long one.



Real-world example : A reader with no programming background before starting this book built a journal-logging tool as her *Book 1* project: one command to add a dated entry, one to list recent entries. It took under an hour, using the same loop from every earlier chapter, and it was still something she used most mornings, months later.

Project solutions

Claude Code is likely to offer several languages to implement your project:

- **Python** : This is often the best choice because it is cross-platform (meaning it works on Windows, macOS, and Linux) and easy to read and test. Python must be installed.
- **Node.js** : This is a good second choice because it is also cross-platform.
- **PowerShell script** : Although this is cross-platform, it is mostly used on Windows.
- **Go, Bash, or others** : More complex setup so I do not recommend them for beginners.

Complete project solutions using Python for the four suggestions in this chapter can be found at the following links:

- **Word Counter** : https://github.com/markjprice/claude-code-vb/blob/main/files/book-1/word-counter/
- **To-do List** : https://github.com/markjprice/claude-code-vb/blob/main/files/book-1/todo-list/
- **Unit Converter** : https://github.com/markjprice/claude-code-vb/blob/main/files/book-1/unit-converter/
- **Journal Logger** : https://github.com/markjprice/claude-code-vb/blob/main/files/book-1/journal-logger/

Try it now

Put every habit from this book to work, from a blank folder to a finished command-line tool:

1. Write a brief for your command-line tool, using the structure from *Chapter 1.6* .

2. Set up the project: a new folder, Git initialized, and a `CLAUDE.md` based on your brief.
3. Ask for a plan, review it, and revise it if needed.
4. Build the tool in small, committed increments, checking each one against your acceptance criteria.
5. Fix anything that breaks using the reproduce, hypothesize, fix, verify pattern.
6. Review your finished commit history from start to finish.



What to ask next : Once your tool works, it's worth asking what a slightly more capable version would need:

```
Now that the first version works, what would be the smalles
```

Keep the answer as a note for later. You do not need to build it now.

Check the result

☐ Does your tool do what your brief's acceptance criteria describe, checked for real, not assumed?

☐ Does your project have a `CLAUDE.md` that still matches how the tool actually works?

☐ Does your commit history read as a sensible sequence of small, working steps?

☐ Could you explain, to someone else, every major decision behind how this tool works?

You built something from a blank folder to a working tool, checked against a brief you wrote yourself, backed by commits you can return to if anything goes wrong. Every habit from this book went into getting it there. *Book 1* is complete.

End of Book 1

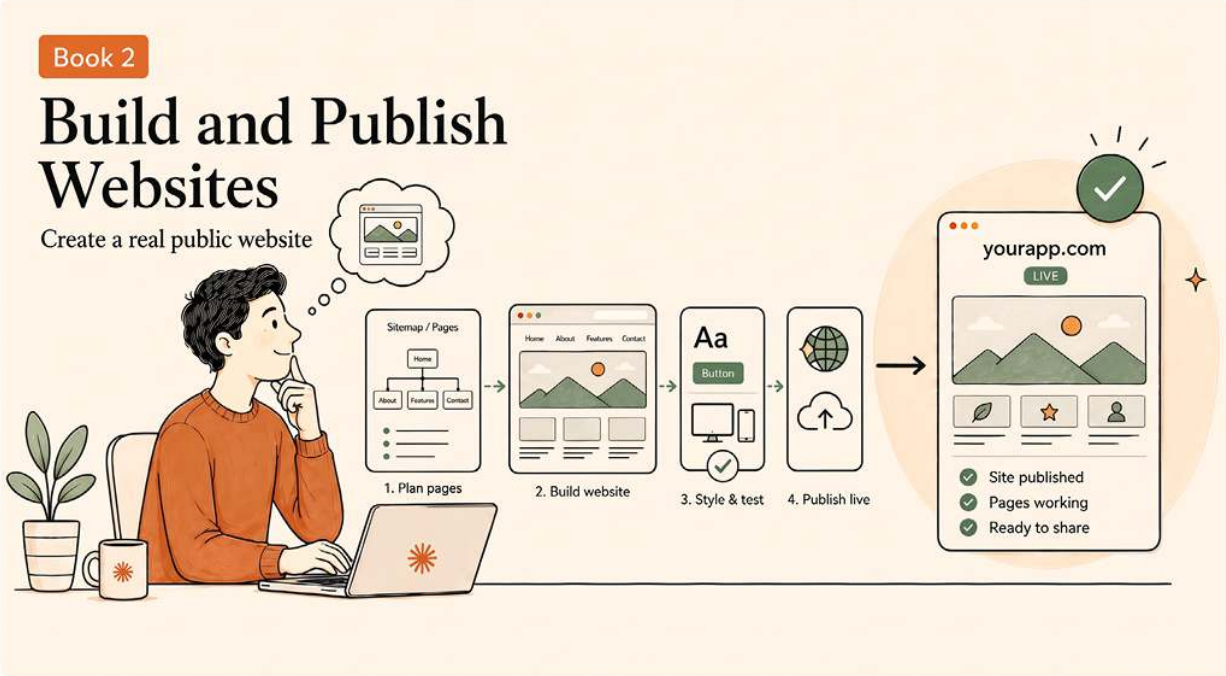
You started *Book 1* without knowing what Claude Code actually did. You finish it with a working method: write a brief, give clear instructions, ask for a plan before anything changes, keep a project's memory, review every diff, save checkpoints you trust, debug one cause at a time, and question what you approve. You used that method to build a working tool of your own, from a blank folder to a finished result.

Book 2 puts the same method to work on a bigger, more visible project: a public website. You'll build the product, support, and privacy pages for a habit-tracking app, the same app whose mobile version you'll build in *Book 3*. The languages change – HTML, CSS, and JavaScript instead of a single command-line file – but the loop stays the same: ask, plan, review, and check the result for yourself.

Book 2

Build and Publish Websites

Book 1 gave you a working method for building with Claude Code: write a brief, give clear instructions, review every plan and every change, keep a project's own memory, and save checkpoints you can trust. You used that method to build something only you needed to see: a small command-line tool, checked and finished on your own computer.



Book 2 puts the same method to work on something built for other people to see. You will build the public website for HabitTrail, a

habit-tracking app that we introduce here and that you will build in *Book 3*. The website will have a home page introducing what HabitTrail does, a support page answering common questions, and a privacy page explaining what data it collects and why. The mobile version of HabitTrail comes later, in *Book 3*. For now, your job is smaller in scope: give a visitor enough information, in a few well-built pages, to decide whether they trust HabitTrail.

This book uses plain HTML, CSS, and JavaScript, without a framework, so you can see how the web actually works before *Book 3* introduces React Native and TypeScript on top of it. You will plan the site around its visitors before writing any code, then build it up in the same small, reviewed increments from *Book 1*: structure first, then a visual system, then layout, then a design that holds up on a phone as well as a desktop screen. Navigation, images, and real interactivity follow, along with a contact form built to fail gracefully instead of losing what a visitor typed.

Chapters 2.10 and *2.11* step back from adding anything new. You will audit the whole site for accessibility, speed, and how it represents itself to search engines and shared links, then publish it to a public address anyone can visit, using the same commit habit you already know to keep it updated safely. By the end of *Book 2*, HabitTrail's website will be live, and the chapter list below shows everything you'll build to get there.

Book 2 has twelve chapters, including an end-of-book project:

2.1 Plan the Website and Its Visitors 70

2.2 Create the Website Project and Structure the Home Page with HTML 75

2.3 Create a Visual System with CSS 80

2.4 Lay Out Sections with Flexbox and Grid 85

2.5 Make the Design Responsive 89

2.6 Add Navigation and Multiple Pages 92

2.7 Add Images, Icons, and Polished Sections 94

2.8 Add Behavior with JavaScript 97

2.9 Build and Validate a Contact Form 100

2.10 Audit Accessibility, Speed, and Search Metadata 104

2.11 Publish the Website and Update It Safely 107

2.12 Project – Build and Publish a Website You Would Use 114

C hapter 2.1 Plan the Website and Its Visitors

In this chapter, you will plan the website you will build across the rest of this book: the public product, support, and privacy site for a habit-tracking app. This book calls that app **HabitTrail** . You will define who visits the site and what they need to do, sketch a site map, and set out what each page must contain before you write a line of HTML.

You need your notebook or note app, and the build-brief habit from *Book 1* . Nothing here requires Claude Code yet. By the end of this chapter, you will have a short brief you can hand to Claude Code in *Chapter 2.2* .

In *Book 3*, you will build the mobile app HabitTrail's website describes. For now, the app does not need to exist. You only need to know enough about it to describe it honestly to a visitor: a habit tracker that helps someone build a small daily habit and see their progress.

Define the visitor and action

A product website usually serves more than one kind of visitor, each wanting something different. HabitTrail's site needs to serve at least three:

1. Someone who has never heard of HabitTrail, deciding whether to try it.
2. An existing user who needs help with something, such as setting a reminder.
3. Anyone who wants to know what happens to their data before they trust the app with it.

For each visitor, name the one action that page should make easy: trying the app, finding an answer, or reading the privacy policy with confidence. A page that tries to do all three at once usually does none of them well.

Create a site map

A site map lists every page a site will have and how they connect. HabitTrail's site needs three pages: a home page introducing the app and its features, a support page for common questions, and a privacy page explaining what data the app collects and why.

Sketch this now, even roughly: three boxes, labeled **Home** , **Support** , and **Privacy** , with lines showing how a visitor gets from one to another. The home page links to both Support and Privacy. Support and Privacy link back to Home. That's the whole map for this project. A small site map is not a limitation. It's a choice that keeps every page focused.



Figure 2.1A: Three pages, each with one clear job. HabitTrail's whole site map fits in a single glance.

Write your site map in words, not just a sketch

A sketch on paper is enough to think clearly about HabitTrail's pages. But you'll also need a written version, because the prompt at the end of this chapter asks you to describe your site map to Claude Code in words:

```
Here's my site map and what each page needs to contain: [describe it].
```

A drawing on a napkin doesn't paste into a chat message. A few lines of text do.

Two formats work well for this:

- A short text outline
- A flowchart written in Mermaid, a simple text-based way to describe diagrams

Either one turns your sketch into something you can copy, paste, save, and hand to Claude Code without redrawing it.

Option 1: a text outline

A text outline lists each page and the pages it links to, using indentation or arrows. This is the fastest option, and it needs no special format or tool.

For HabitTrail, a text outline looks like this:

```
Home
  -> Support
  -> Privacy
Support
  -> Home
Privacy
  -> Home
```

Each line names a page, and each arrow shows a link a visitor can click. Reading this outline top to bottom, you can see that Home links out to both other pages, while Support and Privacy only link back to Home.

Option 2: a Mermaid diagram

Mermaid is a way of describing a diagram using plain text, which a compatible app then draws as boxes and arrows. If you'd rather see your site map as an actual picture, and you're already comfortable

typing a few lines of code, Mermaid gives you that picture without needing a drawing tool.

The same HabitTrail site map, written in Mermaid, looks like this:

```
flowchart TD
    Home --> Support
    Home --> Privacy
    Support --> Home
    Privacy --> Home
```

Each line describes one link: the page it starts from, an arrow, and the page it points to. Many note-taking apps and code editors, including VS Code with a free extension, can turn this text into a rendered diagram. Claude Code can also read Mermaid text directly, even without a picture, since it's just describing the same links as the text outline above.



Good practice : Choose whichever format you'll actually keep using. A text outline is quicker to write and just as clear for a small site. Save Mermaid for when you want a picture to check against your own sketch, or when your site map grows past a handful of pages.

Follow these steps to turn your paper sketch into a written site map:

1. List each page's name on its own line: Home, Support, and Privacy.
2. Under each page, write the pages it links to, using an arrow or an indent.

3. Choose a text outline or Mermaid, based on whether you want a picture as well as text.
4. Save this as a short note in your notebook or note app, the same place you're keeping the rest of your plan from this chapter.

You should now have a site map you can read at a glance, and a written version ready to paste into a message once you reach the prompt at the end of this chapter.



Watch out : A site map with a missing return link, such as a Privacy page with no way back to Home, traps a visitor on that page. Check every page in your outline has at least one link back toward Home before moving on.

Choose content and success criteria

For each page, write one or two sentences describing what it must contain, and one success criterion you could actually check:

- **Home** : Introduces HabitTrail, lists its main features, and gives a visitor a clear next step.
    - *Success* : A first-time visitor can explain, in one sentence, what HabitTrail does.
- **Support** : Answers the questions a new user is most likely to have.
    - *Success* : A visitor with a common question can find the answer without leaving the page.
- **Privacy** : Explains what data HabitTrail collects, why, and how a user could ask for it to be deleted.

- *Success* : A visitor can find this information without searching.

These criteria are your acceptance criteria for the finished website, the same idea from *Chapter 1.6* , applied to a bigger project with several pages instead of one file.

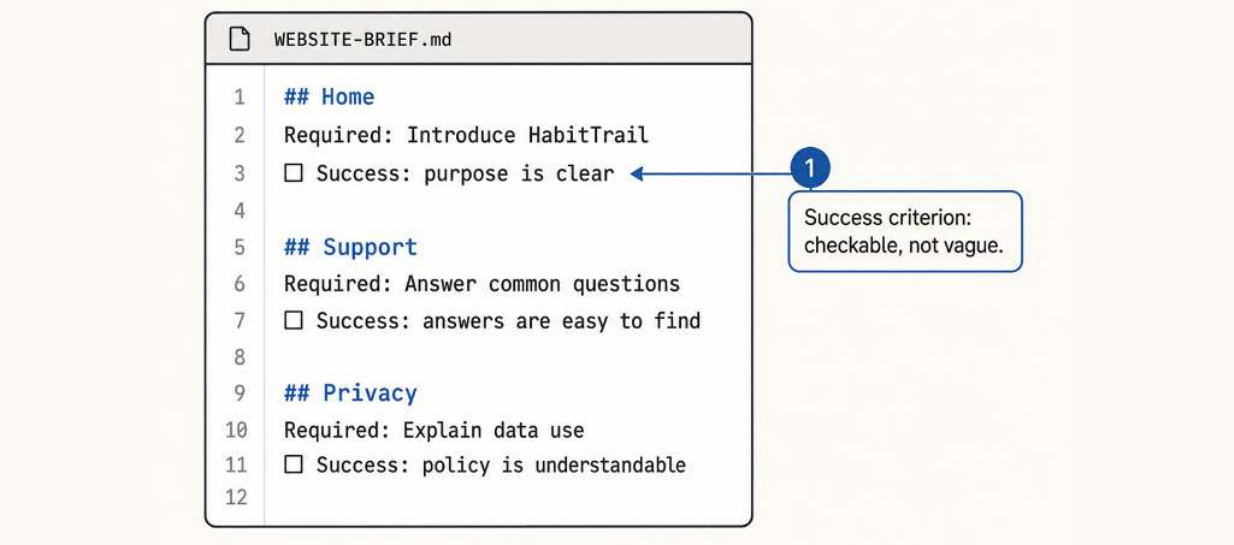


Figure 2.1B: A website brief extends the same pattern from Book 1: what each page needs, and how you'll know it worked



Good practice : Write your site map and page content before opening Claude Code, the same way you wrote a brief in *Book 1* before your first build. Planning on paper costs a few minutes and saves you from restructuring pages later.



Real-world example : A small nonprofit planning its first website started by listing every page they thought they needed: fourteen in total. Naming the one action

each page should support cut that list to five. The other nine ideas became sections on those five pages instead of pages of their own.

Try it now

Put HabitTrail's plan into writing, ready to hand to Claude Code in the next chapter:

1. List the three kinds of visitors HabitTrail's site needs to serve, and the one action each should be able to complete.
2. Sketch a site map with three pages: Home, Support, and Privacy, and the links between them.
3. Write one or two sentences of required content for each page.
4. Write one success criterion per page that you could actually check once the site exists.



What to ask next : Once your plan feels solid, ask Claude Code to look for gaps before you build anything:

```
Here's my site map and what each page needs to contain
```

Check the result

☐ Does your site map show all three pages and how they connect?

☐ Does each page have one clear action tied to a specific kind of visitor?

☐ Could you check each success criterion yourself, once the page exists?

☐ Do you have a one-sentence description of what HabitTrail does, ready to guide your home page?



Sample files : A sample solution to this activity can be found here: https://github.com/markjprice/claude-code-vb/blob/main/files/book-2/2-1-website-brief.md

You now have a plan for a three-page website, with a clear job for each page and a way to check whether it succeeded. The next chapter turns that plan into an actual project folder, with starter files Claude Code creates and a local preview you can open in your browser.

Chapter 2.2 Create the Website Project and Structure the Home Page with HTML

In this chapter, you will turn your plan from *Chapter 2.1* into a real project folder. You will ask Claude Code to set up a sensible starting structure, create starter files for each page, and open a local preview so you can watch the site change as you build it:

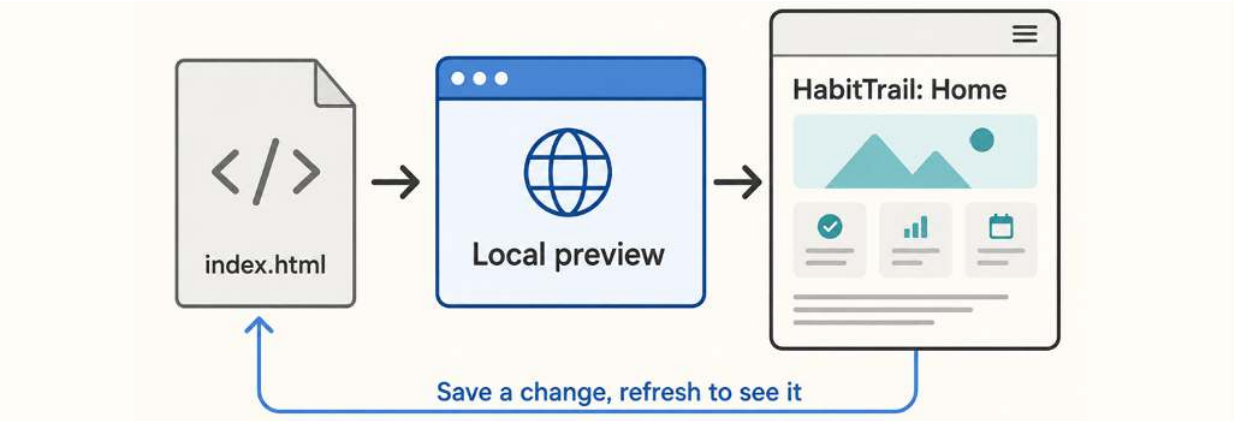


Figure 2.2A: A saved file becomes a page in your browser the moment your local preview refreshes

You need the site map and page content from *Chapter 2.1* .

You will give the home page real structure: meaningful regions, headings, paragraphs, links, and a clear call to action. You will also learn to inspect a page's document outline, so you can check that its structure makes sense, not just that it looks right.

By the end of this chapter, your home page will have real content, organized so both a sighted visitor and a screen reader can understand it.

Ask Claude for the folder structure with starter files

This website uses plain HTML, CSS, and JavaScript, with no build tools or frameworks. That keeps the project structure simple enough to fully understand.

Ask Claude Code to create one minimal HTML file per page, using your site map from *Chapter 2.1* : `index.html` for Home, `support.html` for Support, and `privacy.html` for Privacy. At this stage, each file needs only a page title and a single heading. You are confirming the structure works, not writing final content yet.

Follow these steps to set up your project folder and starter files:

1. Create a new project folder, such as `habittrail-website` , and open the folder in VS Code.
2. Describe your plan and ask for a structure:

```
I'm building a three-page website for a habit-tracking app
```

   A reasonable structure separates pages from the styles and scripts they share: an HTML file for each page at the top level, an `assets` or `images` folder for image and similar files, a `css` or `styles` folder for CSS, and a `js` or `scripts` folder for JavaScript you'll add later in this book:

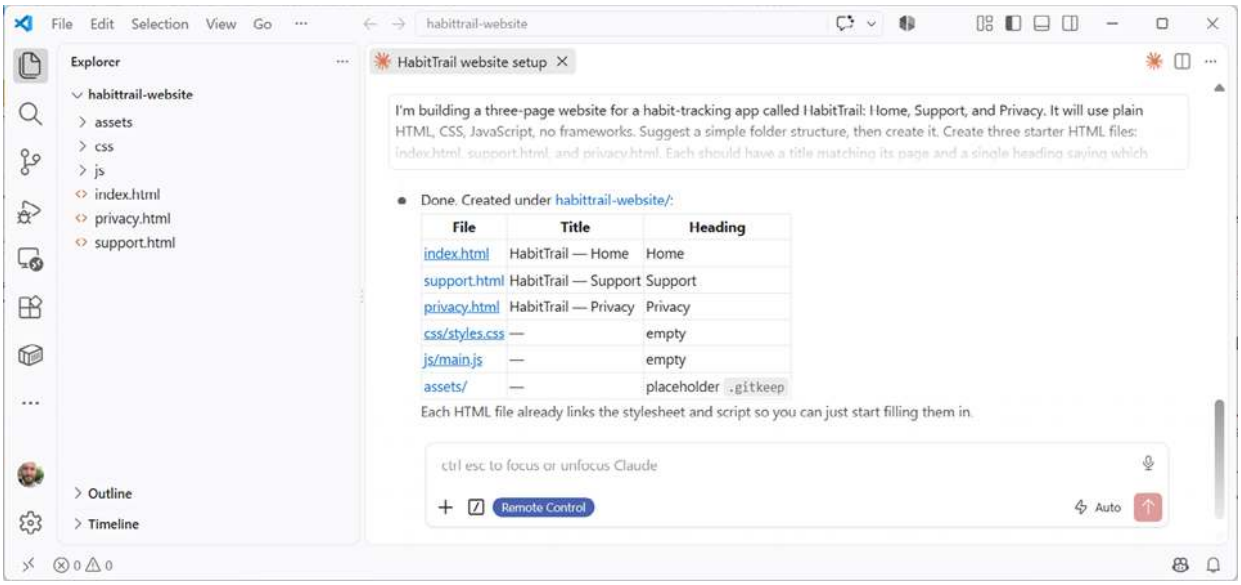


Figure 2.2B: Claude Code creating a website structure with starter files for HabitTrail

3. Expand the folder structure and select each file to view it.
4. Note that the HTML pages each have the same structure and reference the shared CSS and JavaScript files (although both are currently empty, so do not do anything yet):

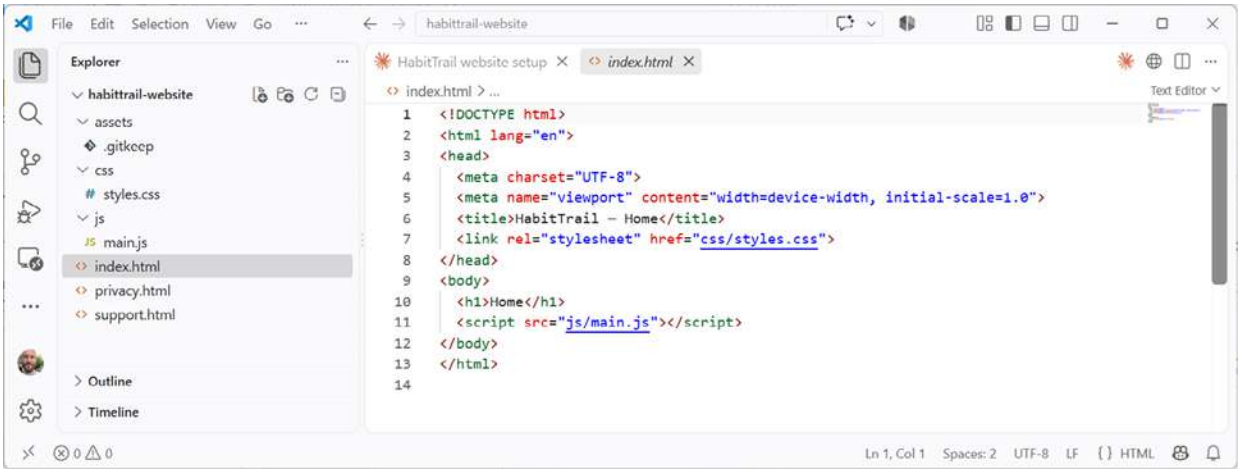


Figure 2.2C: The index.html home page with references to a main.js script file and a styles.css file

Run a local preview

A local preview shows you the site running in a browser, updating as you change files, without publishing anything publicly. Ask Claude Code to start one:

```
Start a local preview of this project and tell me the address to open
```

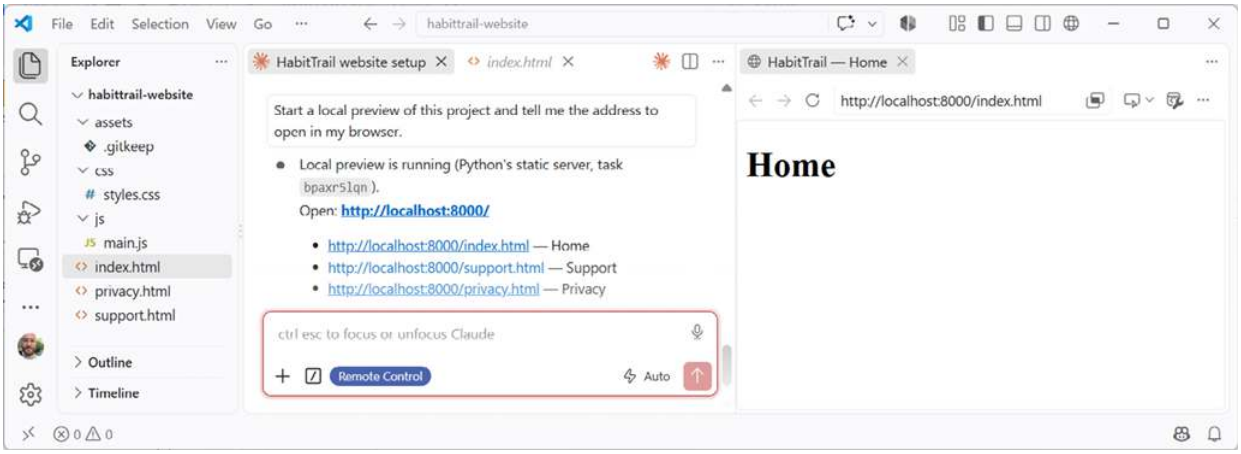


Figure 2.2D: A simple, separated structure: one file per page, shared styles and scripts of their own, and a local address only your computer can see



Watch out: A local preview address only works on your own computer while the preview is running. It is not a public website yet, and nobody else can open that address. Publishing for real is covered in *Chapter 2.11* .

Open the address you're given, usually something like `http://localhost:8000` or similar, and confirm you see your starter home page. Try changing the address to view `support.html` and `privacy.html` too, confirming all three starter pages load.



Good practice : Confirm all three starter pages load in your local preview before adding any real content. A structural mistake is far easier to fix now, with three nearly empty files, than after each page is full of content. Now would be a good time to initialize a Git repository in the project folder and make an initial website implementation commit.



Real-world example : A career changer building her first multi-page site initially let Claude Code create files with names that didn't match her site map, such as `about.html` instead of `support.html` . Asking for the exact structure up front, matching the plan from her own site map, avoided a confusing mismatch between what she'd planned and what actually existed.

Use meaningful page regions

HTML offers elements that describe what a region of a page is for, not just how it looks. A `<header>` holds introductory content, a `<nav>` holds navigation links, `<main>` holds the page's primary content, and a `<footer>` holds closing information such as copyright or contact details. Using these instead of generic containers for everything is called **semantic HTML** : the tags themselves describe meaning, not just appearance.

Ask Claude Code to structure the page this way:

```
Structure index.html using semantic HTML: a header with the HabitTrail
```

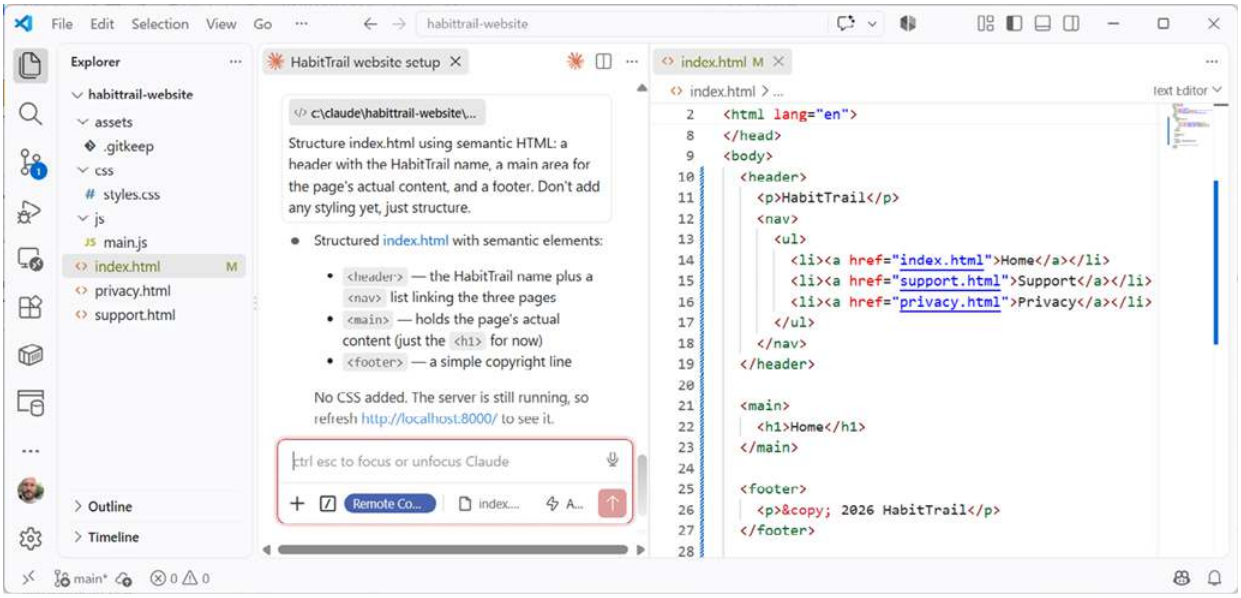


Figure 2.2E: Each semantic tag produces a matching region on the page

Write headings, paragraphs, links, and calls to action

Within `<main>`, build the home page's actual content, using your plan from *Chapter 2.1* :

- One clear heading introducing HabitTrail
- A short paragraph explaining what it does and who it's for
- A features section, with a heading and a short description for each main feature
- A call to action: a clear, specific link or button telling the visitor what to do next, such as "Try HabitTrail" rather than a vague "Click here"

Headings should nest in order, one main heading for the page and smaller headings beneath it for each section, rather than jumping levels or using a heading just because it looks bold enough.

Upload the website brief that you created in *Chapter 2.1* and then prompt Claude to create content:

```
Read the attached website brief and then build the home page's actual
```



Watch out : It's tempting to pick a heading level because of how large it looks, rather than what it means. A heading that looks too big for its place in the outline should be resized with CSS in the next chapter, not demoted to a lower heading level just to shrink it.

Inspect the document outline

A page's document outline is the structure its headings form, independent of how anything looks. You can check it using your browser's built-in accessibility or developer tools, or by asking Claude Code directly:

```
List this page's headings in order, showing their levels, so I can che
```

A sensible outline reads like a table of contents: one top-level heading for the page, and clearly nested subheadings beneath it for each section. If a heading is skipped, out of order, or used purely for visual

size rather than structure, that's worth fixing now, before you add any CSS.



Good practice : Check a page's structure before you style it. Confirming the outline makes sense on a plain, unstyled page is far easier than untangling structure and appearance problems at the same time later.



Real-world example : A small-business owner building her product page initially used a for every section, since it worked visually. Asking Claude Code to review the structure against semantic HTML revealed that a screen reader user would have no way to jump between sections, something no amount of visual polish would have fixed.

Try it now

Turn HabitTrail's plan into a real project folder and a properly structured home page:

1. Create a new project folder and open it in VS Code.
2. Describe your three-page plan and ask Claude Code to suggest and create a folder structure.
3. Ask for three minimal starter HTML files, one per page.
4. Start a local preview and open it in your browser.
5. Confirm all three starter pages load correctly.

6. Once you are happy, initialize a Git repository in the project folder and then commit all the files as an initial website implementation.
7. Structure `index.html` with semantic `<header>`, `<main>`, and `<footer>` regions.
8. Add a heading and short paragraph introducing HabitTrail inside `<main>`.
9. Add a features section with a heading and short description for two or three features.
10. Add one clear call to action.
11. Check the document outline and fix any heading that's out of order.



What to ask next : Once your website feels solid, ask for a second opinion focused on structure and meaning, not appearance:

```
Explain this folder structure to me: what's each
```

Check the result

☐ Does your project have a sensible structure separating pages, styles, and scripts?

☐ Do `index.html`, `support.html`, and `privacy.html` all exist?

☐ Does your local preview open in a browser and show your starter home page?

- ☐ Can you load the other two starter pages by changing the address?
- ☐ Does the page use `<header>`, `<main>`, and `<footer>` instead of generic containers for everything?
- ☐ Does the document outline read sensibly from top to bottom, with no skipped or misused heading levels?
- ☐ Is your call to action specific, rather than a vague “click here”?
- ☐ Does the page still look plain and unstyled, since styling comes next?

You now have an organized, working project with three starter pages loading in a local preview. You built the home page’s structure with meaningful HTML, so a screen reader or search engine, not just a sighted visitor, can understand what each part is for. The next chapter gives your home page a visual system: colors, type, spacing, and the CSS that connects them to the HTML you just wrote.

Chapter 2.3 Create a Visual System with CSS

In this chapter, you will give HabitTrail's website a consistent look, using CSS to define colors, type, spacing, and rounded corners once, then reuse them everywhere. You will also check that your color choices stay readable for every visitor.

You need the structured home page from *Chapter 2.2* . By the end of this chapter, your plain page will have a defined, consistent visual style, built from a small set of reusable choices rather than one-off decisions on every element.

Choose color, type, spacing, and radius tokens

A design token is a named, reusable value, such as a specific shade of blue or a specific spacing amount, defined once and used everywhere it applies. Instead of picking a slightly different blue on five different elements, you define "primary blue" once and reuse it.

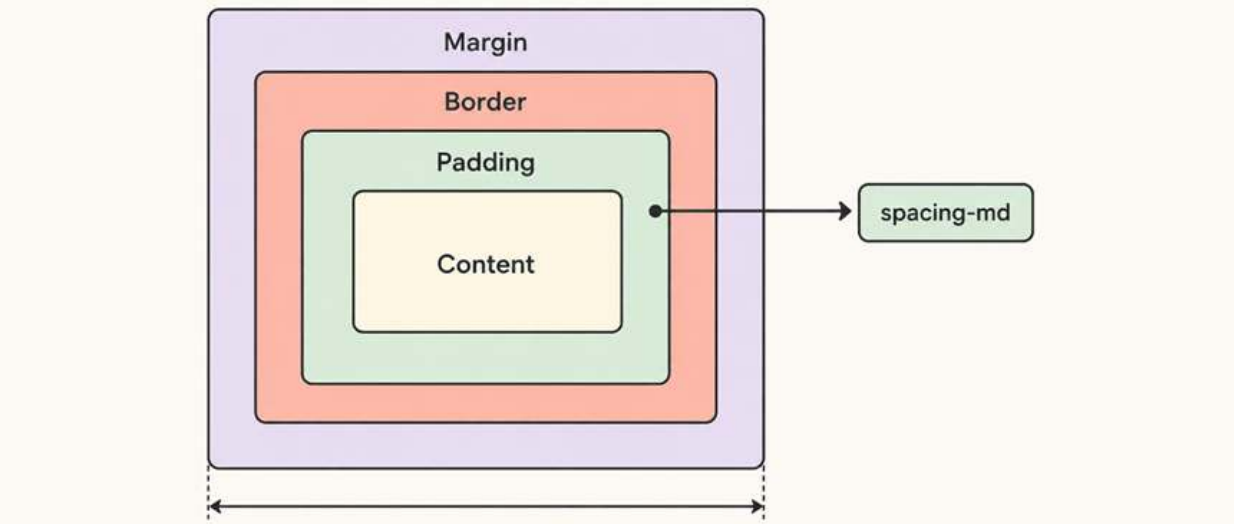


Figure 2.3A: Every element on a page is a box. Spacing tokens control the padding and margin layers around its content

Ask Claude Code to propose a small starting set:

```
Suggest a simple visual design system for HabitTrail's website: a prim
```

CSS lets you store these as custom properties, a way of naming a value once at the top of your stylesheet so every other rule can refer to it by name instead of repeating the raw value.

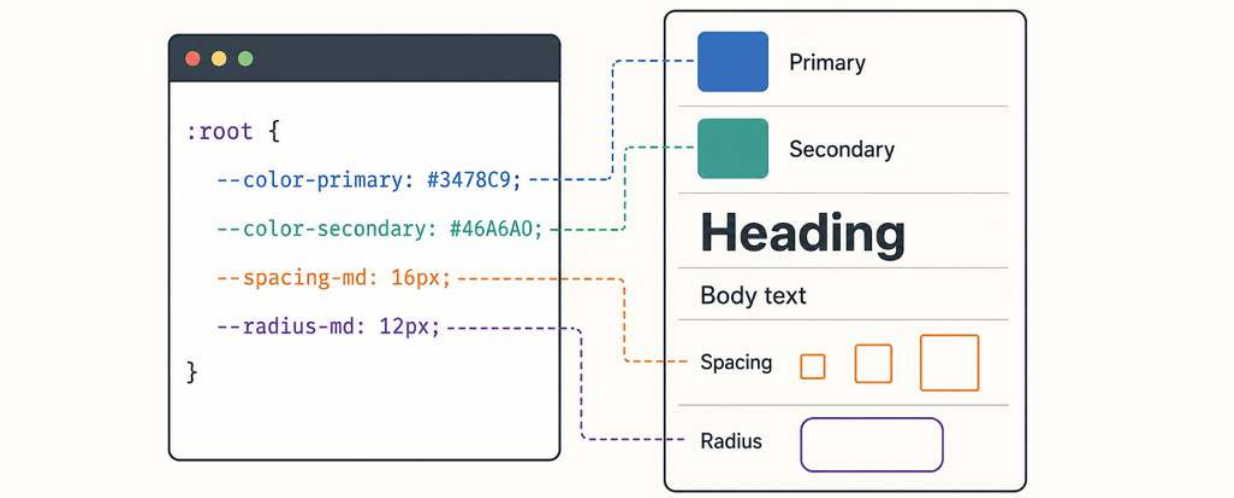


Figure 2.3B: Define a small set of named values once, then reuse them everywhere they apply



Watch out : A color that looks fine on your own screen may not be fine on someone else's, due to lighting, screen quality, or vision differences. Checking contrast against a guideline, not just your own eyes, protects visitors you'll never see using the site.

Connect CSS to HTML

A stylesheet only affects a page once the page knows to use it. Confirm your HTML file links to your CSS file, usually through a `<link>` element in the page's `<head>`, and that your CSS rules target the actual elements and classes in your HTML.

Ask Claude Code to apply your new design tokens to all three webpages:

```
Apply this design system to all three webpages: use the primary color
```

Refresh your local preview after each change. Seeing your design tokens take visible effect, one at a time, makes it easy to catch a token that isn't quite right before it spreads across the whole site.

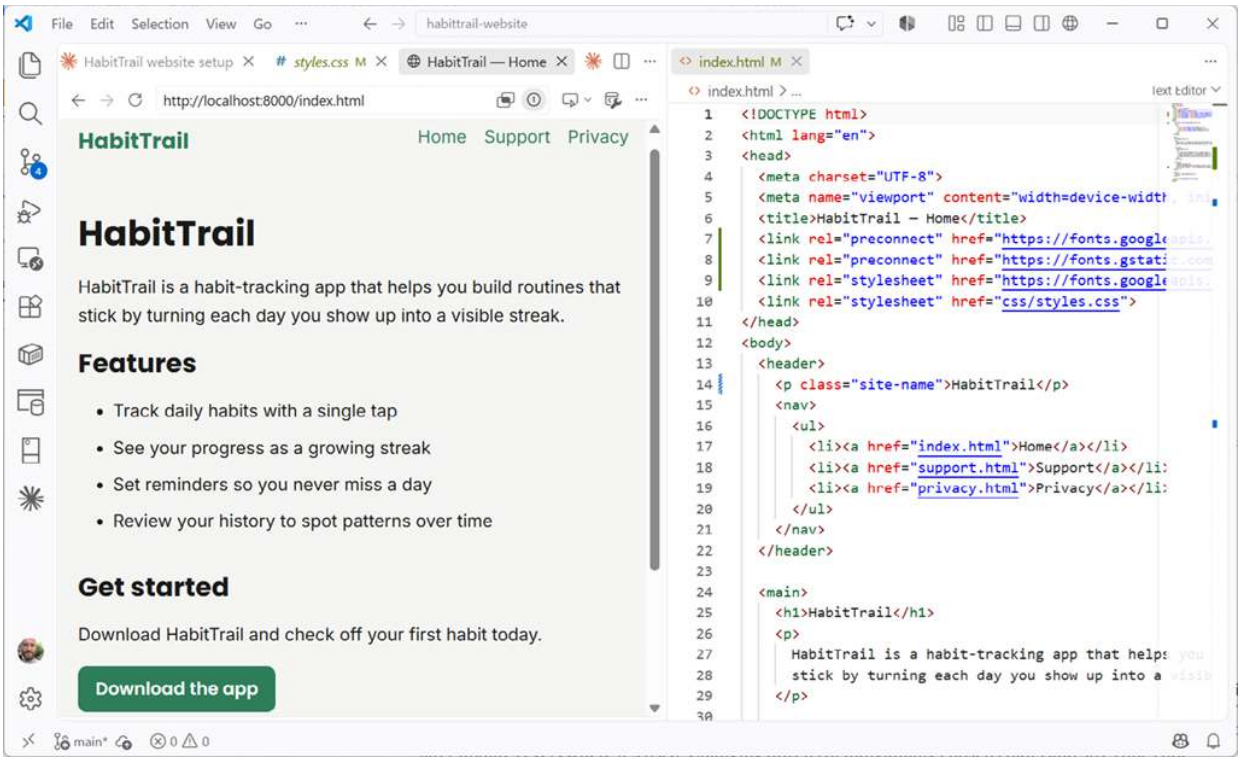


Figure 2.3C: CSS applied to the index.html home page with Google Fonts

Choose free fonts from Google Fonts

Google Fonts is a free library of web fonts, hosted by Google, that any site can load with a couple of lines of code. It's the easiest place to pick real typefaces for HabitTrail's site, instead of settling for whatever font a visitor's device happens to have installed.

Left to its own judgment, Claude Code tends to reach for a small handful of fonts on almost every project: Inter, Roboto, and the visitor's system font are common defaults. None of these are bad fonts. Inter and Roboto are both free, well-made, and easy to read. But using them by default, on every site, is how a lot of AI-built websites end up looking alike. Naming a font pairing yourself, even a

simple one, gives HabitTrail's site more of an identity than accepting whatever Claude Code reaches for first.

For example, Claude Code might choose Poppins for headings and Inter for body text, with sensible fallbacks if those fonts are not found on the local device:

```
/* Fonts */
--font-heading: "Poppins", system-ui, -apple-system, "Segoe UI", sar
--font-body: "Inter", system-ui, -apple-system, "Segoe UI", sans-ser
```



Watch out : Not every font Claude Code suggests is a free Google Font. Some AI-recommended type pairings include fonts from other type foundries, with their own licenses and sometimes a cost for commercial use. Before you load a suggested font, check that it's actually listed at https://fonts.google.com . If it isn't, treat it as a starting idea and look for a similar free option in the Google Fonts library instead.

A font pairing usually works best as two fonts: one for headings, chosen for character, and one for body text, chosen for readability at small sizes over a paragraph. *Table 2.3A* shows four pairings, all free on Google Fonts, that suit a straightforward site like HabitTrail's:

Style	**Heading font**	**Body font**
Clean and neutral	IBM Plex Sans	Source Sans 3

Warm and editorial	Fraunces	Crimson Pro
Distinctive	Bricolage Grotesque	Source Sans 3
Technical	Space Grotesk	IBM Plex Sans

Table 2.3A: Google Fonts pairings for a small site

You don't need to try every option. Pick the row that sounds closest to how you want HabitTrail to feel, or browse https://fonts.google.com yourself and choose two fonts that pair well: usually one with more character for headings, and one plainer, highly readable font for body text.

Ask Claude Code to do the wiring for you once the fonts are chosen:

```
Load IBM Plex Sans for headings and Source Sans 3 for body text from G
```

Refresh your local preview after this change. You should see your heading and body text switch to the new fonts immediately, without any other part of the page changing.



Good practice : Name a specific font pairing when you ask Claude Code to set up type, rather than asking for "nice fonts" and accepting whatever comes back. Claude Code will pick something reasonable either way, but naming your own choice, even from the table above, keeps the decision yours.

Keep contrast readable

Contrast is the difference in brightness between text and its background. Low contrast, such as light gray text on a white background, can be unreadable for many visitors, not just a matter of taste.

Ask Claude Code to check this directly:

```
Check the contrast between my text colors and their backgrounds agains
```

Fix any flagged combination before moving on. A visual system only helps visitors if they can actually read it.

When Claude Code or a browser tool checks HabitTrail's site for accessibility, it's usually measuring the result against the Web Content Accessibility Guidelines (WCAG), the standard most accessibility reports and laws refer to. WCAG groups its rules into three levels: A, AA, and AAA.

Level A covers the most basic requirements, the kind of gap that blocks some visitors outright, such as an image with no `alt` text. Level AA adds stricter, widely expected requirements, such as text with a contrast ratio of at least 4.5 to 1 against its background, and it's the level most laws and most real sites aim for. Level AAA goes further, requiring a contrast ratio of at least 7 to 1 and covering rules suited to specific audiences rather than every site.

A report that shows HabitTrail passing AA but failing AAA usually isn't a defect to fix. It means the site meets the standard most visitors

and most legal requirements expect, without also meeting a stricter standard built for a narrower set of needs:

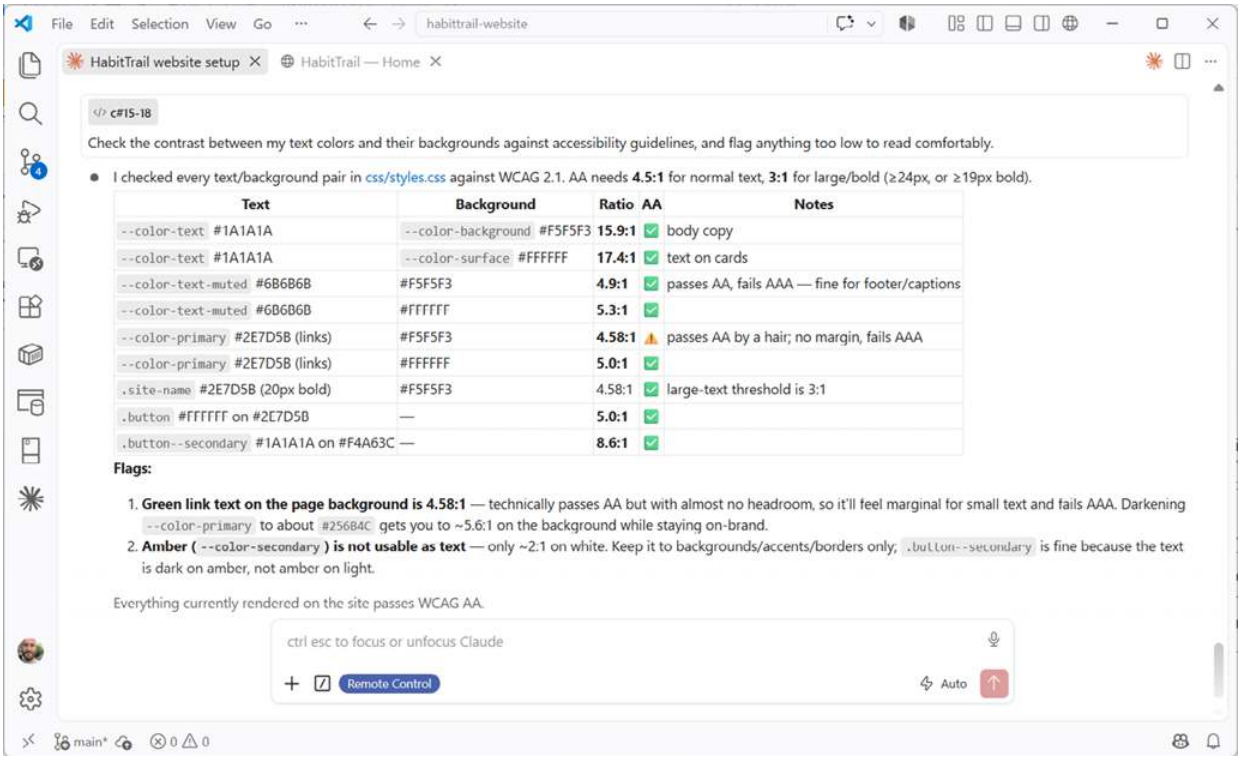


Figure 2.3D: A visual accessibility report from Claude Code about the colors used by the website



Good practice : Define your design tokens before writing more than a handful of CSS rules. Changing one token later updates everywhere it's used. Changing dozens of individually chosen colors and sizes does not.



Real-world example : A designer new to CSS kept nudging colors slightly differently across different buttons on her site, "just a little," until nothing quite matched. Switching to a small set of named tokens,

defined once, made every button consistent automatically, and made a later color change take seconds instead of an hour of hunting.

Try it now

Give HabitTrail's home page a real visual identity, built from a small set of reusable choices:

1. Ask Claude Code to propose a small design system: colors, type, spacing, and a corner radius.
2. Review the suggestions and adjust anything that doesn't feel like HabitTrail.
3. Apply the tokens to your home page's heading, call to action, and section backgrounds.
4. Check contrast between your text and background colors, and fix anything flagged.
5. Once you are happy, commit the design system changes to Git.



What to ask next : Once your design tokens are applied, it's worth confirming they're actually being reused, not duplicated:

```
Check my CSS for colors or spacing values that should
```

Check the result

- ☐ Does your CSS define a small set of named colors, type, spacing, and radius values?
- ☐ Does your home page visibly use these tokens, rather than one-off values?
- ☐ Has every text and background combination passed a contrast check?
- ☐ Would changing one token update every place it's used, without editing each one by hand?

Your home page now has a real visual identity, built from a small, consistent set of choices instead of scattered decisions. The next chapter arranges those styled pieces into an actual layout, using Flexbox and Grid to build a hero section and a row of feature cards.

Chapter 2.4 Lay Out Sections with Flexbox and Grid

In this chapter, you will arrange your styled content into an actual layout: a hero section introducing HabitTrail, and a row of feature cards beneath it. You will learn just enough about how CSS positions elements to direct Claude Code confidently, without needing to master layout code yourself.

You need the styled home page from *Chapter 2.3* . By the end of this chapter, your home page will have a real layout, arranged with tools built for the job rather than manual positioning.

Understand boxes and spacing

Every element on a page is a box, as you saw in *Figure 2.3A* 's box model diagram. Layout is the job of deciding how boxes relate to each other: side by side, stacked, evenly spaced, or aligned to match. Two CSS tools handle almost all of this for a site like HabitTrail's: Flexbox, best for arranging items in a single row or column, and Grid, best for arranging items into rows and columns at once, like a table of cards.

You do not need to memorize their rules. You need to recognize which situation calls for which tool, and ask for it by name.

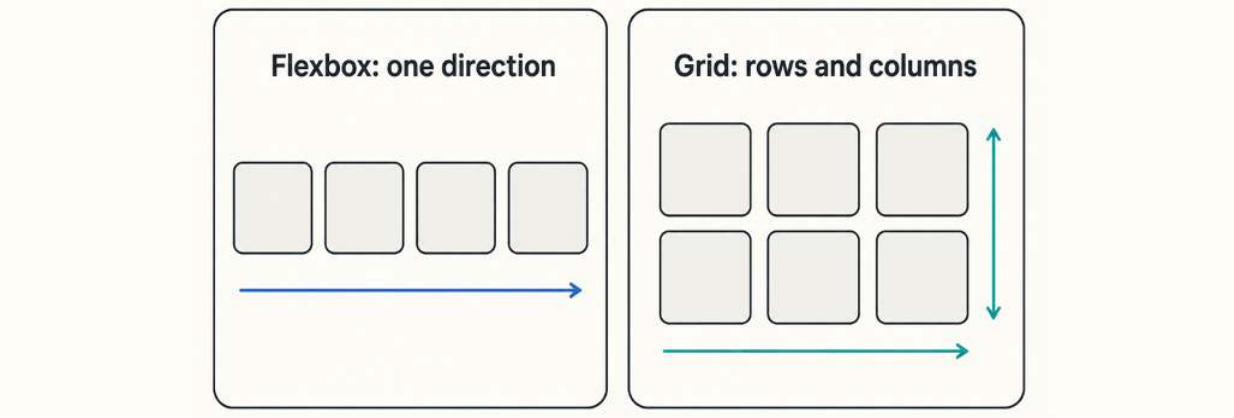


Figure 2.4A: Flexbox arranges items in one direction. Grid arranges them in two at once. HabitTrail's hero uses Flexbox, its feature row uses Grid.



Good practice : Ask Claude Code which layout tool, Flexbox or Grid, fits a section before it writes any layout code. Naming the tool yourself, once you understand the difference, keeps you following the logic instead of just approving whatever appears.

Build hero and feature layouts

A hero section is the large, prominent area near the top of a page, usually holding the main heading, a short description, and a call to action. Ask Claude Code to lay this out clearly:

```
Lay out the hero section on the home page using Flexbox: heading and c
```

For the features section, a grid works well:

```
Lay out the features section as a Grid: three columns on wide screens,
```

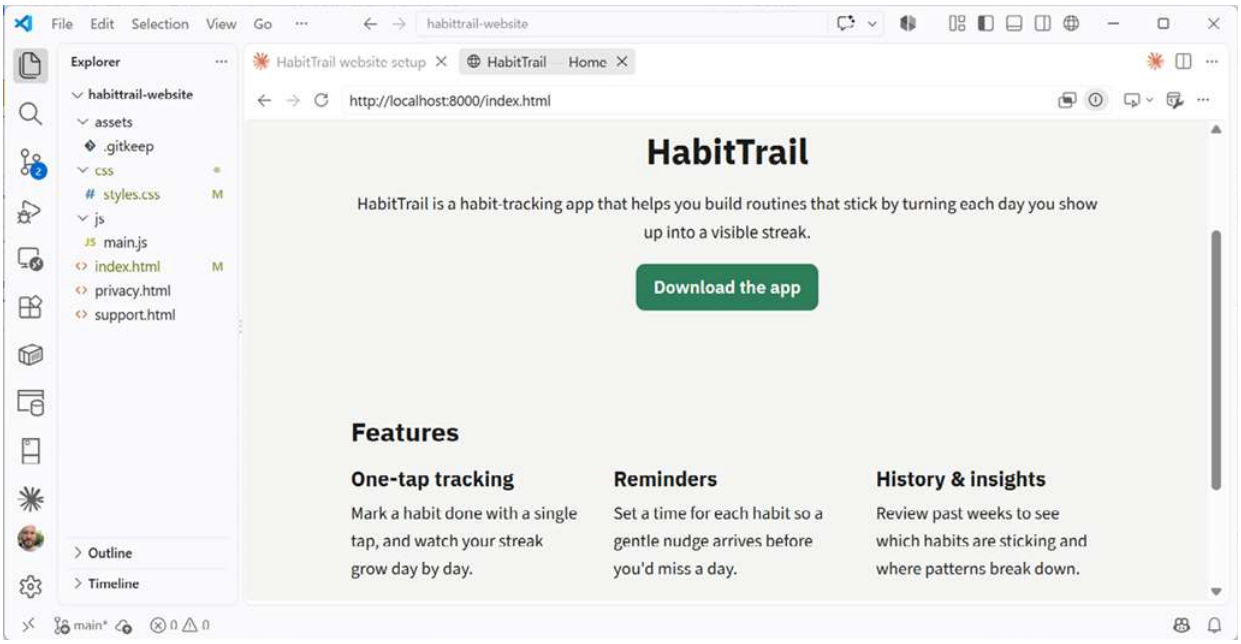


Figure 2.4B: Home page hero section using Flexbox and features section using Grid

Refresh your local preview after each change. You should see boxes that previously stacked awkwardly or ran together now aligning cleanly.

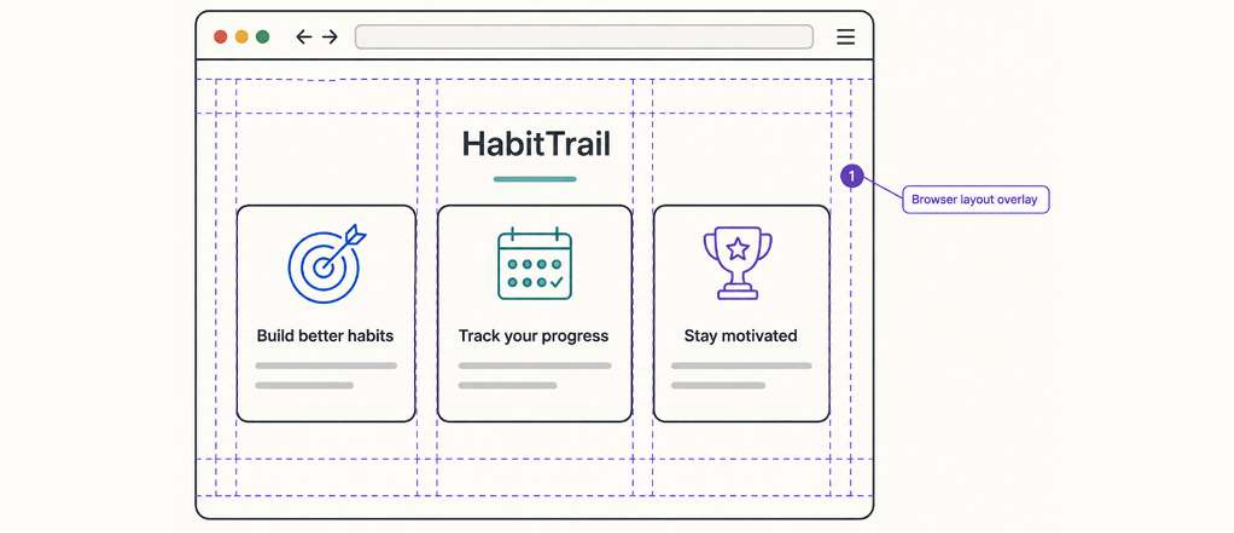


Figure 2.4C: Your browser can show you the invisible grid holding a layout together, which is a fast way to confirm a layout is doing what you expect

Show layout in your browser's Developer Tools

Every browser includes Developer Tools, a set of panels built into the browser itself for inspecting a page: its HTML, its CSS, and how each element is actually being laid out. You don't need to install anything. It's already there, and it's the fastest way to confirm Flexbox and Grid are doing what you asked, instead of guessing from how the page looks.

Open your local preview of HabitTrail's home page, then open **Developer Tools** using whichever method matches your browser:

- In Chrome or Edge, press *F12* , or right-click anywhere on the page and choose **Inspect** .
- In Firefox, press *F12* , or right-click the page and choose **Inspect** .
- On a Mac, *Cmd+Option+I* opens Developer Tools in Chrome, Edge, or Firefox.

A panel opens, usually docked to the side or bottom of the browser window, showing your page's HTML on one side and its CSS on the other.

To see how a specific element, such as your feature card grid, is actually being laid out:

1. Click the small arrow-and-square icon near the top-left of the **Developer Tools** panel, sometimes called the element picker.
2. Click the element on the page you want to inspect, such as one feature card.
3. In the CSS panel, look for a `display: grid` or `display: flex line` . This confirms which layout tool is controlling that element.

4. In Chrome or Edge, a small grid or flex badge appears next to the element in the HTML panel. Click it to overlay colored lines directly on the page, showing the actual columns, rows, or spacing Flexbox or Grid is applying.

Figure 2.4C , shown earlier in this chapter, shows this overlay in action: colored lines drawn directly over your feature cards, showing the grid holding them in place. It's the difference between assuming a layout is correct because it looks right, and confirming it with the same grid the browser itself is using.



Good practice : Open Developer Tools before asking Claude Code to fix a layout problem you can see but can't quite describe. Pointing at the exact element and checking its display value often tells you which layout tool is actually involved, so your request to Claude Code can name it directly, rather than describing the problem vaguely.

Align cards without fixed positions

It's tempting to nudge an element into place with an exact pixel position, but a fixed position breaks as soon as content or screen size changes. Flexbox and Grid instead describe relationships, such as "evenly spaced" or "three equal columns," that hold up automatically as content changes:

```
Check the feature cards for any fixed widths or positions, and replace
```

This habit matters more once you reach the next chapter, where the same layout needs to work at very different screen widths.



Watch out : A layout that looks right with exactly three feature cards can break in unexpected ways with two or four. Test your layout with a different number of items before trusting it's flexible enough.



Real-world example : A hobbyist building a portfolio site had manually positioned every project card with exact pixel coordinates, then spent an evening nudging each one every time he added a new project. Switching the section to Grid meant a new card simply took its place in the layout automatically, with no manual repositioning at all.

Try it now

Arrange HabitTrail's styled content into a real layout, then prove it holds up when the content changes:

1. Ask Claude Code to lay out the hero section with Flexbox: heading, description, and call to action, stacked and centered.
2. Ask Claude Code to lay out the features section with Grid: an even row of feature cards.
3. Check for any fixed widths or positions and replace them with Flexbox or Grid rules.

4. Temporarily add a fourth feature card and confirm the layout still looks right.



What to ask next : Once your layout looks right, it's worth testing its resilience directly:

```
Add a fourth feature card temporarily and show me how
```

Check the result

- ☐ Does the hero section's content stack cleanly and stay centered?
- ☐ Do the feature cards align evenly in a row, without manual positioning?
- ☐ Does the layout still work correctly with a different number of feature cards?
- ☐ Can you explain, in your own words, when you'd reach for Flexbox versus Grid?

Your home page now has a real layout, built with tools designed to hold up as content changes. The next chapter puts that flexibility to its real test, making sure this same layout works well on a small phone screen, not just the wide window you've been previewing in.

Chapter 2.5 Make the Design Responsive

In this chapter, you will make HabitTrail's home page work well on a small phone screen, a tablet, and a wide desktop monitor, all from the same HTML and CSS. You will start from the smallest screen, add breakpoints where the layout needs to change, and test all three sizes yourself.

You need the laid-out home page from *Chapter 2.4* . By the end of this chapter, your page should read comfortably and look intentional at any common screen width, not just the one you've been previewing in.

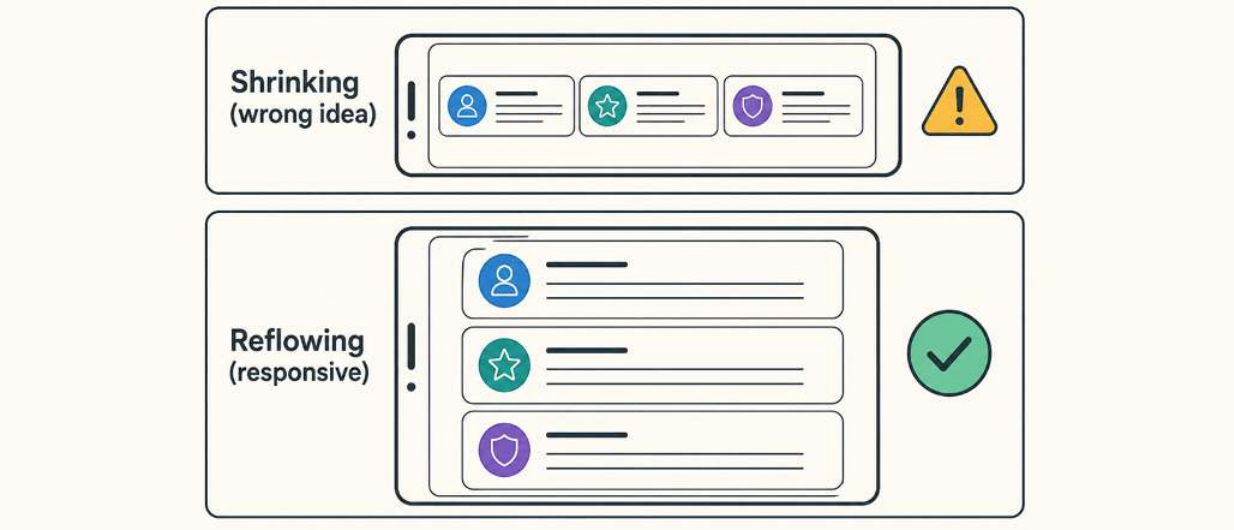


Figure 2.5A: Responsive design rearranges content to fit the screen. It doesn't just shrink everything until it's too small to use.

Watch out : Responsive design means content rearranges to fit the screen. It does not mean the same



layout simply shrinks until everything is too small to read or tap comfortably. If text or buttons are shrinking rather than rearranging, that's a sign a breakpoint is needed, not a smaller font.

Start with the small screen

A responsive design adapts to the visitor's screen instead of assuming everyone has a wide monitor. The most reliable way to build one is to design for the smallest, most constrained screen first, then add complexity as space allows, rather than starting wide and hoping things shrink gracefully.

Ask Claude Code to check your current layout against this starting point:

```
Assume this page is being viewed on a phone screen about 375 pixels wi
```

Ask Claude to fix what's flagged before worrying about wider screens at all. A page that works well on the smallest screen and gets more spacious from there tends to age better than one built the other way around.

Add useful breakpoints

A breakpoint is a screen width where your layout deliberately changes to suit the space available, defined in CSS using a media query. HabitTrail's three-column feature grid, for example, likely needs to become a single column on a narrow phone screen and

perhaps two columns on a tablet, before reaching three columns on a wide desktop:

```
Add breakpoints so the features grid shows one column on narrow screer
```

Add breakpoints only where the design actually needs to change, not at every possible screen size. A handful of well-chosen breakpoints is easier to maintain than dozens of narrow adjustments.

Test narrow, medium, and wide layouts

Use your browser's responsive preview mode to check the page at several widths, not just by resizing the window and glancing at it. Confirm text stays readable, buttons stay easy to tap, and nothing overlaps or runs off the edge of the screen at any width you test:

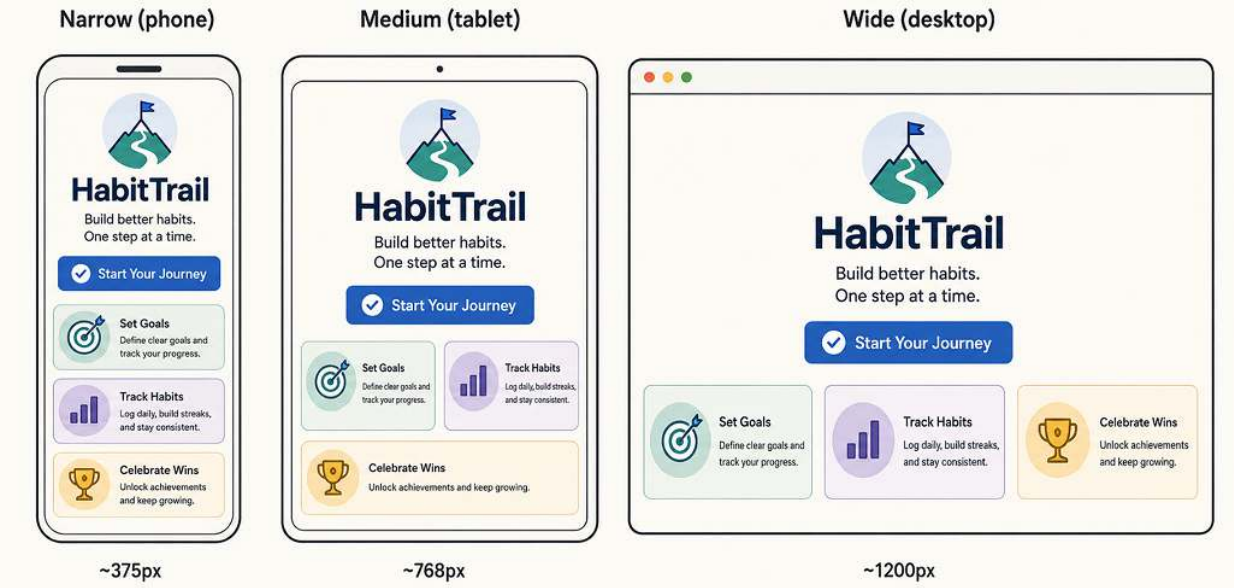


Figure 2.5B: One layout, three widths. The content rearranges at each breakpoint instead of simply scaling.



Good practice : Test your responsive layout at a narrow width, not just a slightly smaller version of your desktop window. Many real visitors browse on phones far narrower than most laptop screens appear when simply resized.



Real-world example : A student's first attempt at a responsive layout looked fine on his laptop but left visitors on phones squinting at feature cards squeezed into a tiny three-column row. Adding one breakpoint, so cards stacked into a single column below a certain width, fixed the phone experience without changing anything about how the wide desktop version looked.

Try it now

Test HabitTrail's layout at three different screen widths, starting from the narrowest:

1. Check your hero and features sections against a narrow phone width and fix anything cramped or cut off.
2. Add breakpoints so the features grid adapts: one column narrow, two medium, three wide.
3. Use your browser's responsive preview mode to test narrow, medium, and wide layouts.
4. Confirm text stays readable and buttons stay tappable at every width you test.



What to ask next : Once your layout responds well at the widths you tested, ask for a check beyond your own device:

```
List a few common phone, tablet, and desktop screen wi
```

Check the result

- ☐ Does the layout work well on a narrow phone width, not just a shrunk desktop window?
- ☐ Does the features grid rearrange at your breakpoints, rather than just shrinking?
- ☐ Have you tested at least three distinct widths: narrow, medium, and wide?
- ☐ Can you explain, in your own words, the difference between content reflowing and a page simply shrinking?

Your home page now adapts to the visitor's screen instead of assuming everyone has the same one. The next chapter builds the site's navigation and its two remaining pages, Support and Privacy, so a visitor can actually move around the site you've planned.

Chapter 2.6 Add Navigation and Multiple Pages

In this chapter, you will build the Support and Privacy pages from your site map in *Chapter 2.1* , and connect all three pages with navigation a visitor, and a keyboard user, can actually use.

You need the responsive home page from *Chapter 2.5* , and the starter `support.html` and `privacy.html` files from *Chapter 2.2* . By the end of this chapter, HabitTrail's site will be a real, connected three-page site:

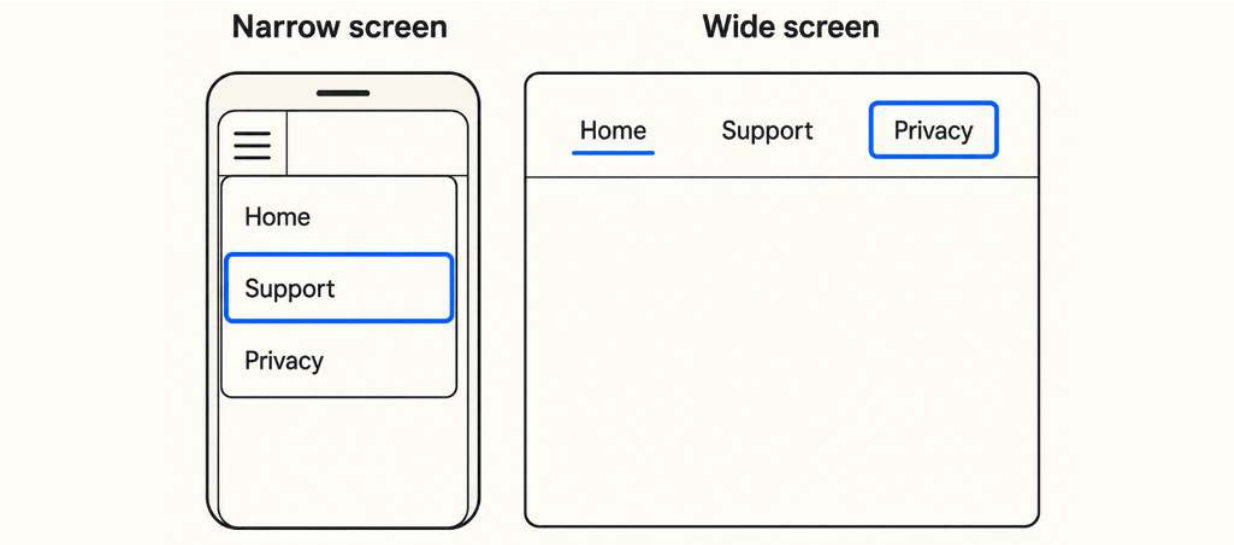


Figure 2.6A: Navigation should look different at different widths, but the ability to reach every link with a keyboard, shown here by the focus outline, should never disappear

Build accessible navigation

Navigation is how a visitor moves between pages, usually a set of links near the top of every page. Accessible navigation means it

works for everyone, including someone using a keyboard instead of a mouse, or a screen reader instead of sight.

```
Add a navigation menu inside the header, with links to Home, Support,
```

Test this yourself: click somewhere on the page, then press *Tab* repeatedly to move the current selection around the webpage. Press *Enter* to select a link. You should be able to reach and activate every navigation link this way, and see clearly which one currently has focus.



Good practice : Test keyboard navigation on every new page you add, not just the home page. A menu that works with the keyboard on one page but not another usually means the structure differs somewhere between them.

Create support and privacy pages

Using your plan from *Chapter 2.1* , fill in the Support and Privacy pages with real structure and content, the same way you built the home page in *Chapter 2.2* .

For Support, a short list of common questions and answers usually works well. For Privacy, explain plainly what data HabitTrail collects, why, and how someone could ask for it to be removed. Avoid vague reassurance. State specifically what happens, since this page exists precisely so a cautious visitor can check:

```
Structure support.html with a heading, an introduction, and three or f
```



Watch out : A privacy page written in vague, reassuring language, without actually stating what data is collected, fails the one job it has. If you can't state specifically what HabitTrail would collect and why, that's worth deciding before writing the page, not glossing over in the writing.

Keep labels and active states consistent

Every page should show the same navigation, with the same labels, in the same order. An active state shows a visitor which page they're currently on, usually through a visual difference such as an underline or a different color on that one navigation link.

Ask Claude Code to make the navigation consistent:

```
Make sure the navigation appears identically on all three pages, and a
```

Consistency here matters more than it might seem. Navigation that changes shape or wording between pages forces a visitor to relearn it every time, even on a small three-page site.

Real-world example : A freelancer's first attempt at a privacy page copied generic boilerplate language he



found online, without actually describing his own app. Rewriting it specifically for what his app actually did, in plain language, took less time than the boilerplate had, and left visitors with an honest answer instead of vague reassurance.

Try it now

Connect HabitTrail's three pages with navigation a keyboard user can actually use:

1. Add semantic, keyboard-accessible navigation to the header, linking Home, Support, and Privacy.
2. Build out the Support page with a short set of real questions and answers.
3. Build out the Privacy page with plain, specific language about data collection and deletion.
4. Add a consistent active state showing which page a visitor is currently on.
5. Test keyboard navigation on all three pages.



What to ask next : Once all three pages exist and connect, it's worth checking the whole site as a visitor would experience it:

```
Walk through this site as a new visitor would: land on
```

Check the result

- ☐ Can you reach and activate every navigation link using only the *Tab* key, on every page?
- ☐ Do the Support and Privacy pages contain real, specific content, not placeholder text?
- ☐ Does the navigation look and behave identically across all three pages?
- ☐ Does the active state clearly show which page a visitor is currently viewing?

HabitTrail's site is now a real, connected three-page site, with navigation that works for a keyboard user as well as a mouse user. The next chapter adds images, icons, and a few more polished sections, so the site looks as finished as it now behaves.

Chapter 2.7 Add Images, Icons, and Polished Sections

In this chapter, you will add images and icons to HabitTrail's site the right way: sized responsibly, described for visitors who can't see them, and legally yours to use. You will also add a few sections that make a site feel finished: testimonials, an FAQ, and a download panel.

You need the connected three-page site from *Chapter 2.6* . By the end of this chapter, HabitTrail's home page should feel like a finished product page, not just correctly structured text.

Choose lawful assets

An image you didn't create isn't automatically yours to use. Stick to images and icons you have clear rights to: ones you made yourself, ones explicitly licensed for free use, such as many icon sets and stock photo sites offer, or ones you've paid a license for. Keep a note of where each asset came from and what its license allows, in case you need to check later:

```
Suggest a source for free, clearly licensed icons and illustrations su
```

Now, let's ask Claude to add images to the website:

```
Select some images from your recommended sites and use them on the Hab
```

Create responsive images and helpful alt text

An image should scale sensibly across the screen widths from *Chapter 2.5* , rather than staying a fixed size that overflows a narrow screen or looks tiny on a wide one. Every meaningful image also needs `alt` text: a short written description that a screen reader announces instead of the image, and that displays if the image fails to load.

Ask Claude Code to add responsive sizing and helpful alt text to images:

```
Add responsive sizing to every image on the home page, and write alt t
```

A decorative image, one that adds nothing a visitor needs to know, can have empty `alt` text instead, so a screen reader skips it rather than describing something irrelevant:

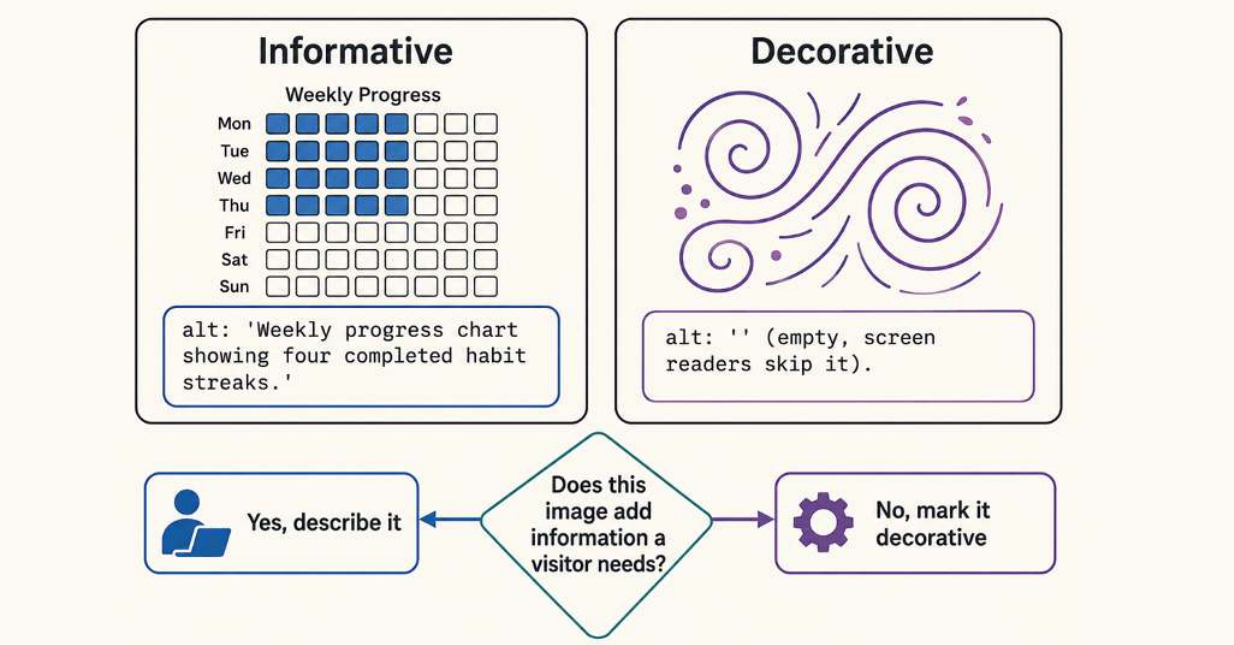


Figure 2.7A: Ask what an image actually adds. Describe it if the answer matters. Mark it decorative if it doesn't.



Watch out : `alt` text that just repeats an image's filename, or says "image," tells a screen reader user nothing useful. Describe what the image actually shows and why it's there, the way you'd describe it to someone on the phone who couldn't see your screen.

Add testimonials, FAQ, and download panels

These sections help a visitor trust and act on what they're seeing. Since HabitTrail is a book example rather than a real app, write clearly fictional, placeholder testimonials, labeled as examples in your own notes, rather than anything that could be mistaken for a real review.

Ask Claude Code to add testimonials, FAQs, and download links for the app on Android and iOS:

```
Add a testimonials section with two or three short, clearly fictional
```

Structure each new section the same way you structured earlier ones: meaningful headings, readable text, and a layout that responds well across screen widths.

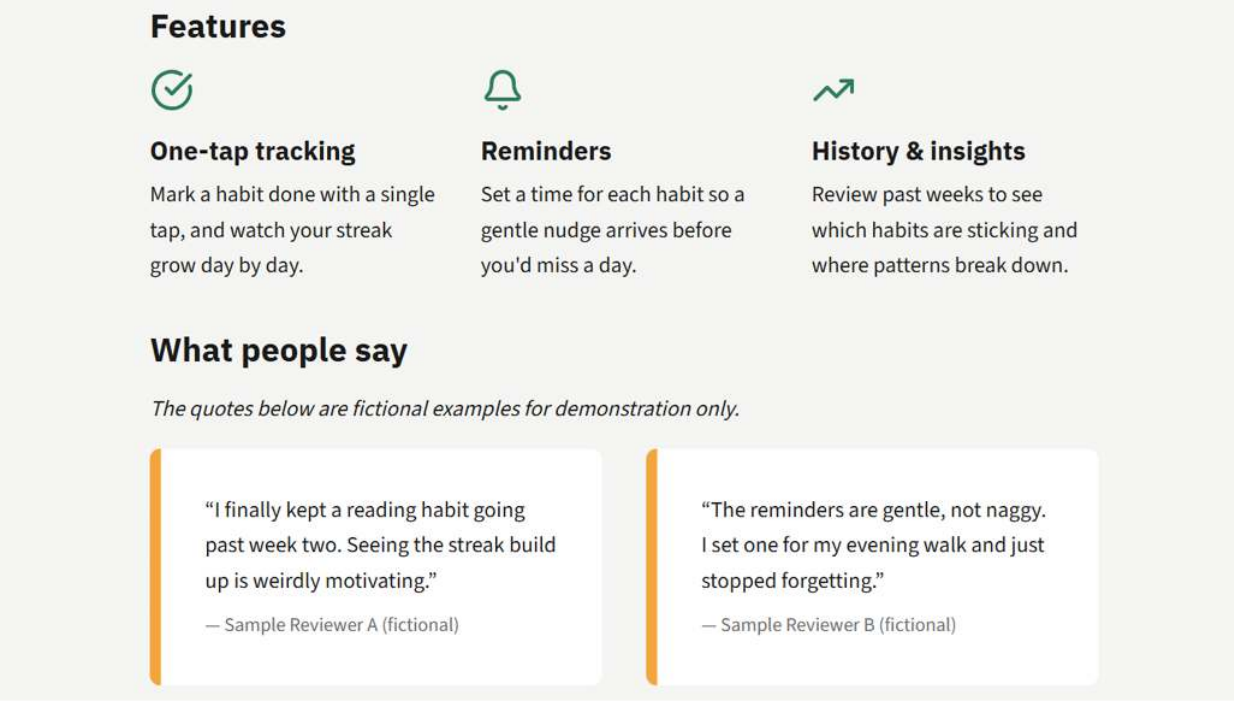


Figure 2.7B: Icons for features and a testimonials section

Good practice : Note an image's source and license the moment you add it, not later when you can no longer remember where it came from. A simple list in your project notes, one line per image, is enough.

Real-world example : A small-business owner had used a striking photo she found through a search engine for her homepage, without checking its license. A friend pointed out the risk before she published, and she replaced it with a similar image from a site that explicitly offered free commercial use, avoiding a problem she hadn't realized she was creating.

Try it now

Add real images and a few finishing sections to HabitTrail's home page, the right way:

1. Choose or confirm a source of lawfully usable icons and images for the site.
2. Add responsive images to the home page, with real alt text for meaningful images.
3. Mark any purely decorative image so screen readers skip it.
4. Add a testimonials section with clearly fictional example quotes, an FAQ, and a download panel.



What to ask next : Once your images and new sections are in place, check accessibility together rather than assuming it's fine:

```
Go through every image on this page and confirm each o
```

Check the result

☐ Do you know the source and license of every image on the site?

☐ Does every meaningful image have `alt` text describing what it actually shows?

☐ Are decorative images marked so screen readers skip them?

- ☐ Do the new sections look and respond consistently with the rest of the page across screen widths?

HabitTrail's home page now looks and feels like a finished product page: styled, laid out, responsive, and dressed with real images and content sections. The next chapter adds the one thing a static page still lacks: behavior that responds when a visitor interacts with it.

Chapter 2.8 Add Behavior with JavaScript

In this chapter, you will make HabitTrail's site respond to visitors instead of just displaying content. You will learn the basic shape of how JavaScript works: selecting something on the page, responding to an event, and updating what the visitor sees:

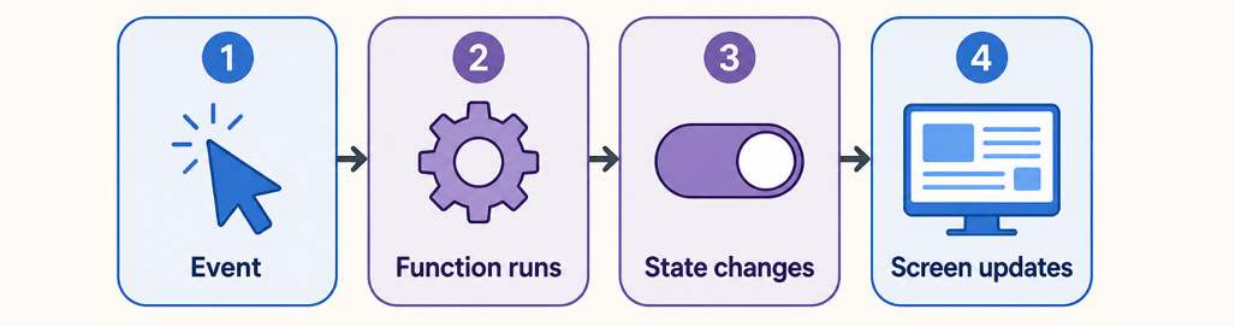


Figure 2.8A: The same four-step pattern powers the FAQ and the theme toggle: something happens, code responds, a value changes, the screen updates

Then, you will use it to build an expandable FAQ and a saved theme preference.

You need the polished, image-complete site from *Chapter 2.7* . By the end of this chapter, at least three parts of HabitTrail's site will react to something a visitor does.

Select an element

Before JavaScript can do anything to part of a page, it needs to identify which part. Selecting an element means telling JavaScript

exactly which piece of HTML to work with, usually by referring to its `id` or `class`, the same names your CSS already uses to style it.

Ask Claude Code to show you how the download buttons can be selected in code:

```
Show me how the download buttons for iOS and Android are currently ide
```

Recognizing this connection, that CSS and JavaScript both refer back to the same HTML elements, makes the rest of this chapter easier to follow, even without writing the code yourself.

Respond to an event

An event is something that happens on a page: a click, a key press, typing into a field. JavaScript can listen for a specific event on a specific element, and run a small block of code, a function, whenever it happens.

Ask Claude Code to explain how event handling works:

```
Explain, in plain language, what happens step by step when a visitor c
```

You don't need to write this code yourself. You need to be able to read Claude Code's explanation and confirm it matches what you actually want to happen.

Build a menu, FAQ control, and theme preference

With the pattern in mind, ask Claude Code to build three small interactive features:

```
Add a menu button that opens and closes the navigation on narrow scree
```

Test each one directly in your local preview: click the menu button and confirm it opens and closes, click an FAQ question and confirm its answer appears, and toggle the theme and reload the page to confirm your choice was remembered.

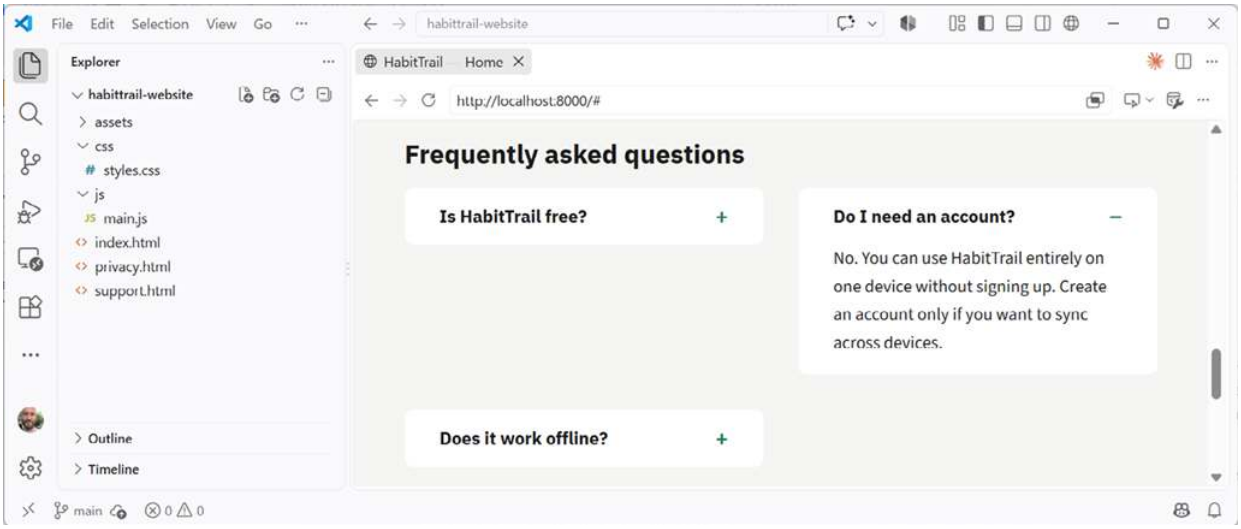


Figure 2.8B: A small function change is enough to take a page from static text to something a visitor can actually interact with

Good practice : Test each interactive feature the moment it's added, rather than building all three before checking any of them. A menu that doesn't close, tested immediately, is a quick fix. Discovered after two more features were added on top of it, it's a harder problem to isolate.



Real-world example : A hobbyist added a theme toggle to his site and was pleased with how it looked, until he refreshed the page and watched his chosen dark theme reset to light every time. Asking specifically for the choice to be remembered between visits, rather than assuming "add a toggle" implied it, fixed a gap he hadn't realized he'd left in his own request.

Try it now

Make three parts of HabitTrail's site respond to what a visitor actually does:

1. Ask Claude Code to explain what happens, step by step, when the download menu button is clicked.
2. Add working open and close behavior to the mobile menu.
3. Add expand and collapse behavior to each FAQ question.
4. Add a theme toggle that remembers the visitor's choice between visits.
5. Test all three features with both a mouse and a keyboard.



What to ask next : Once your interactive features work with a mouse, it's worth confirming they work more broadly:

```
Confirm that the menu button and every FAQ question ca
```



Watch out : An interactive feature that only works with a mouse click excludes keyboard and screen reader users. Ask Claude Code to confirm that anything clickable, such as the menu button and FAQ questions, also works when reached and activated with a keyboard.

Check the result

- ☐ Does the mobile menu open and close correctly when its button is clicked?
- ☐ Does each FAQ question expand and collapse its answer correctly?
- ☐ Does the theme toggle remember your choice after reloading the page?
- ☐ Can you reach and activate all three features using only the keyboard?

HabitTrail's site now responds to what a visitor actually does, not just what they read. The next chapter adds one more interactive piece, a contact form, along with the validation and feedback that make a form trustworthy to use.

Chapter 2.9 Build and Validate a Contact Form

In this chapter, you will add a contact form to HabitTrail's Support page, so a visitor can reach out directly. You will label its inputs properly, validate what a visitor enters, and show clear errors and success messages without ever losing what they've already typed:

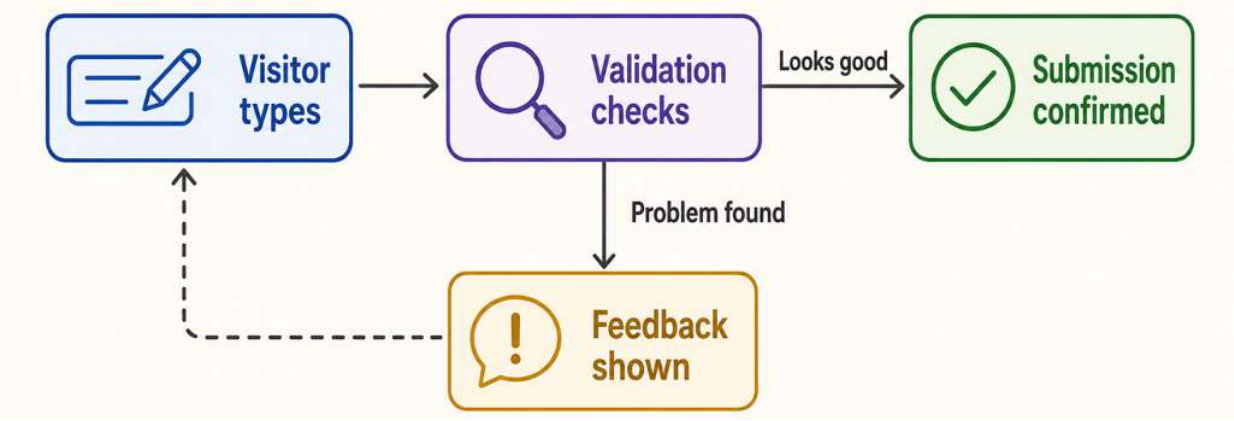


Figure 2.9A: A good form checks before it submits, and sends the visitor back to fix a problem without losing what they already typed

You need the interactive site from *Chapter 2.8* . By the end of this chapter, a visitor should be able to fill out the form, see helpful feedback if something's wrong, and get clear confirmation once it's submitted successfully.

Label inputs

Every field in a form needs a label: text that clearly identifies what belongs in that field, connected to it in the HTML so a screen reader

announces the label when the field receives focus, not just placeholder text that disappears the moment someone starts typing.

Ask Claude Code to add a contact form:

```
Add a contact form to support.html with fields for name, email, and me
```

Placeholder text can still help as an example of what to type, such as showing a sample email format, but it should never be the only way a field is identified.

By default, Claude Code may rely on the built-in validation of standard elements. For example, setting `required` and `type="email"`:

```
<input type="email" name="email" required/>
```

Validate useful constraints

Validation checks that what a visitor entered makes sense before the form submits, catching problems early rather than after submission fails somewhere else. Useful constraints for this form include: the name and message fields aren't left empty, and the email field looks like a genuine email address.

Ask Claude Code to add validation to the contact form:

```
Add validation to the contact form: name and message can't be empty, a
```

Keep validation focused on constraints that are important. A form that rejects entirely reasonable input over an overly strict rule

frustrates visitors more than it protects anyone.

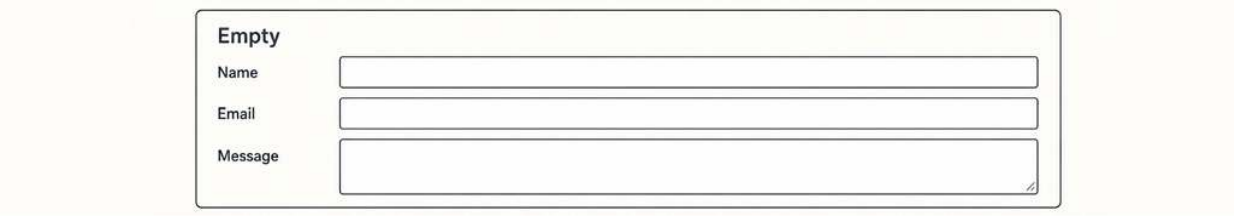


Figure 2.9B: Three states: empty

Show errors and success without losing data

A form that clears everything a visitor typed the moment one field has a problem is one of the more frustrating experiences on the web. Confirm that an error only points out what's wrong, without discarding the rest of what was entered:

```
When validation fails, keep everything the visitor already typed in pl
```



Watch out : A validation message that just says "invalid input" doesn't tell a visitor what to fix. "Enter an email address including an @ symbol, like name@example.com" gives them something they can actually act on.

Test this yourself: fill out the form with one deliberate mistake, such as an invalid email, and confirm your other answers are still there after the error appears.

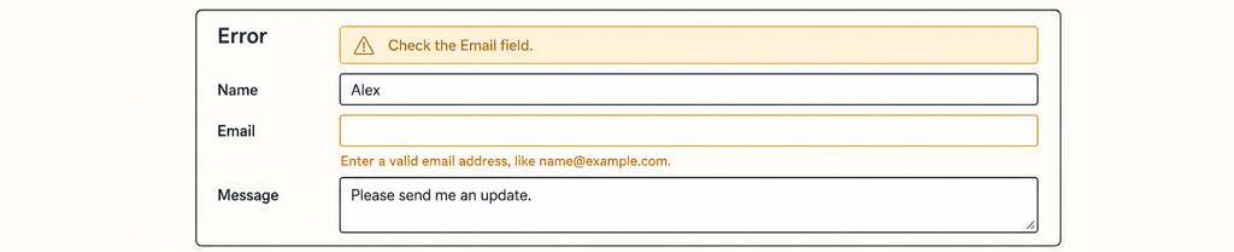
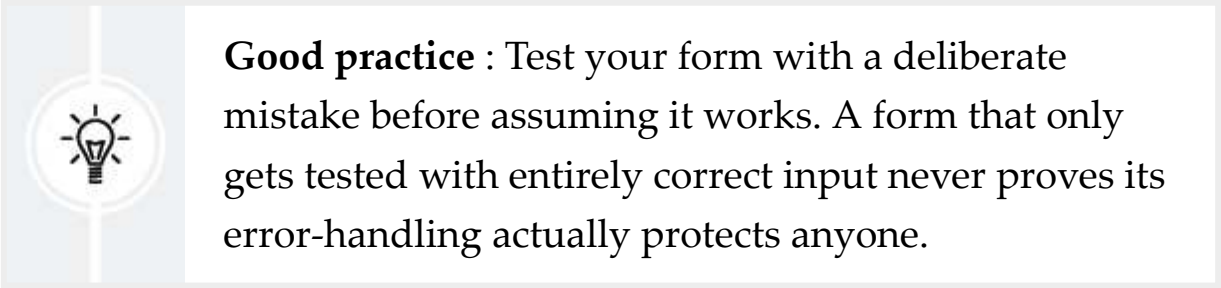


Figure 2.9C: Three states: a specific and recoverable error



Good practice : Test your form with a deliberate mistake before assuming it works. A form that only gets tested with entirely correct input never proves its error-handling actually protects anyone.

Then, test it again, this time with valid data input:

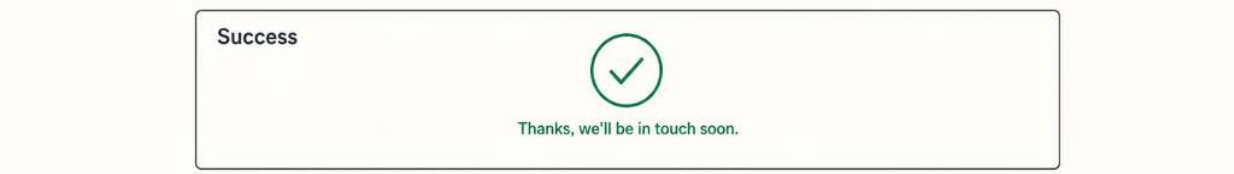
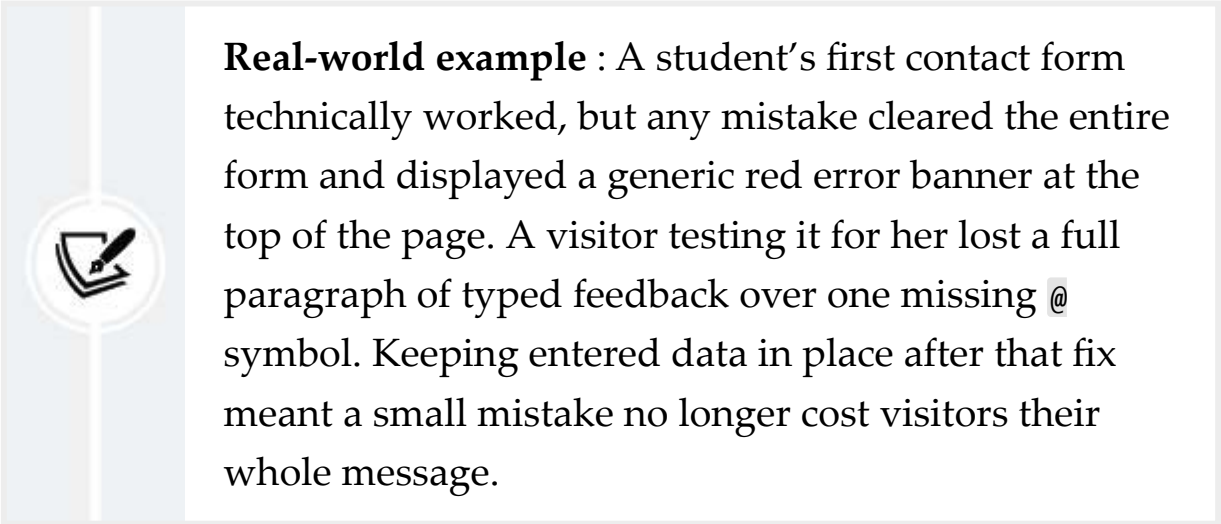


Figure 2.9D: Three states: a clear confirmation once it works



Real-world example : A student's first contact form technically worked, but any mistake cleared the entire form and displayed a generic red error banner at the top of the page. A visitor testing it for her lost a full paragraph of typed feedback over one missing `@` symbol. Keeping entered data in place after that fix meant a small mistake no longer cost visitors their whole message.

Use a mailto link instead of a full contact form

The contact form you just built in this chapter sends what a visitor types to a server, using an HTTP method called `POST`. That's the right choice when you want every submission collected in one place, such as Netlify's form dashboard (a hosting service you will meet in *Chapter 2.11*), and when you want to validate what a visitor enters before it goes anywhere. It also means more to build: labeled fields, validation rules, and error and success messages that don't lose what the visitor already typed.



Watch out : Clicking a `mailto` link doesn't send anything by itself. It only opens the visitor's email app with a draft ready to go, and they still have to press send there. On a phone or computer with no email app set up, such as one used only through a browser, the link may fail to do anything useful at all. A `mailto` link suits a small site with one point of contact and no need to store submissions. A site that depends on hearing from every visitor who tries to reach it is better served by the validated form from earlier in this chapter.

Not every site needs that much. A `mailto` link is a lighter alternative: instead of submitting anything to a server, it opens the visitor's own email application with a message already addressed to you.

```
<a href="mailto:hello@example.com?subject=Website%20inquiry&body=Hi,%2
  Email us
```

```
</a>
```

The address after `mailto:` is where the message will go. The `subject` and `body` parts are optional. They fill in a starting subject line and a first line of text, which the visitor can still edit before sending. There's no server involved at all, which is why a `mailto` link works even on GitHub Pages, without needing Netlify or a form service like Formspree.



Good practice : Tell visitors what a `mailto` link will do before they click it, such as labeling it "Email us" rather than "Contact us" or "Submit." A visitor who expects a form and gets their email app open instead can be confused about what just happened.



Real-world example : A `mailto` link is what one reader from *Chapter 2.12* 's project chose for his dog-walking round's website, alongside a phone number in the footer, once he decided his three-client business didn't need the form validation this chapter builds.

Try it now

Add a contact form to HabitTrail's Support page that fails gracefully instead of losing what a visitor typed:

1. Add a contact form to the Support page with properly labeled name, email, and message fields.

2. Add validation: no empty required fields, and a genuine-looking email address.
3. Confirm error messages are specific and point to the exact field with a problem.
4. Confirm previously entered data survives a validation error.
5. Confirm a successful submission shows a clear confirmation message.



What to ask next : Once your form handles mistakes gracefully, it's worth confirming where a real submission actually goes:

```
Where does this form's data go when it's submitted? Ex
```

Check the result

- ☐ Does every field have a properly connected label, not just a placeholder?
- ☐ Does the form catch an empty required field and an invalid email address?
- ☐ Does a validation error keep the visitor's other answers in place?
- ☐ Does a successful submission show a clear confirmation message?

HabitTrail's site can now receive a message from a visitor, with a form that fails gracefully instead of losing their work. The next

chapter steps back from adding anything new, and audits everything you've built so far for accessibility, speed, and how it appears to search engines.

Chapter 2.10 Audit Accessibility, Speed, and Search Metadata

In this chapter, you will step back from adding anything new and audit HabitTrail's whole site instead: checking it with a keyboard, checking how fast it loads, and adding the information search engines and social media need to represent it correctly when someone shares a link.

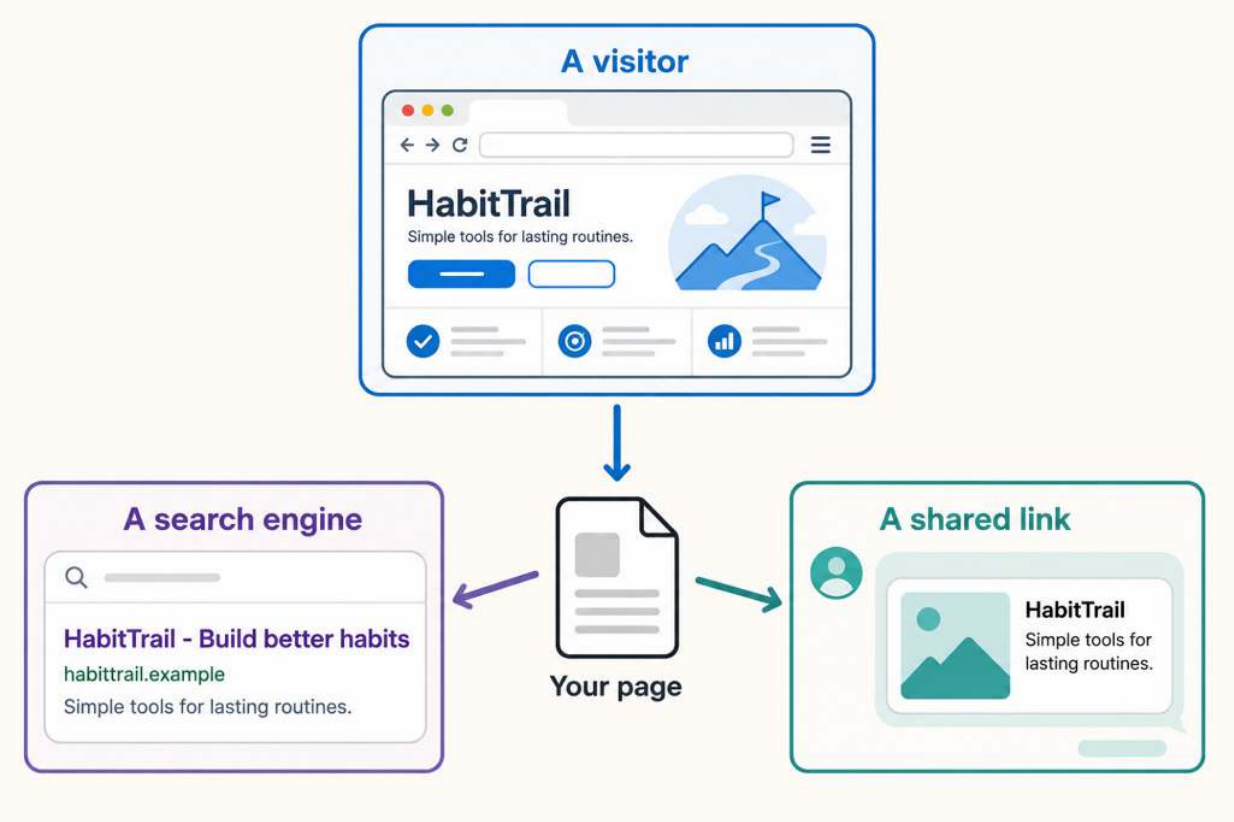


Figure 2.10A: The same page needs to make sense to a visitor, a search engine, and a link preview, all from the same title, description, and structure

You need the complete, interactive site from *Chapter 2.9* . By the end of this chapter, you will have a short list of fixes applied, and a page

that's ready to represent itself well beyond your own browser.



Good practice : Run this audit on the whole site in one sitting, rather than checking accessibility on one visit and speed on another weeks later. Issues in one area, such as an oversized image, sometimes affect another, such as how long a screen reader takes to reach real content.

Use keyboard and focus checks

You've tested keyboard access on individual features throughout this book. Now, do it across the whole site, start to finish:

```
I'm going to tab through this entire site using only the keyboard, sta
```

Then, do it yourself: load the home page, and press *Tab* repeatedly, all the way through the navigation, the content, the form, and into the other pages. Every interactive element should be reachable, in a sensible order, with a visible indicator showing where you are.

Optimize images and loading

A page that takes too long to appear loses visitors before they see anything you've built. Ask Claude Code to check for common, fixable causes of a slow page:

```
Check this site for anything likely to slow down loading, especially c
```

Common fixes include reducing an image's file size without visibly harming its quality, and making sure images below the visible screen area don't need to load before the visitor can see and use the top of the page.

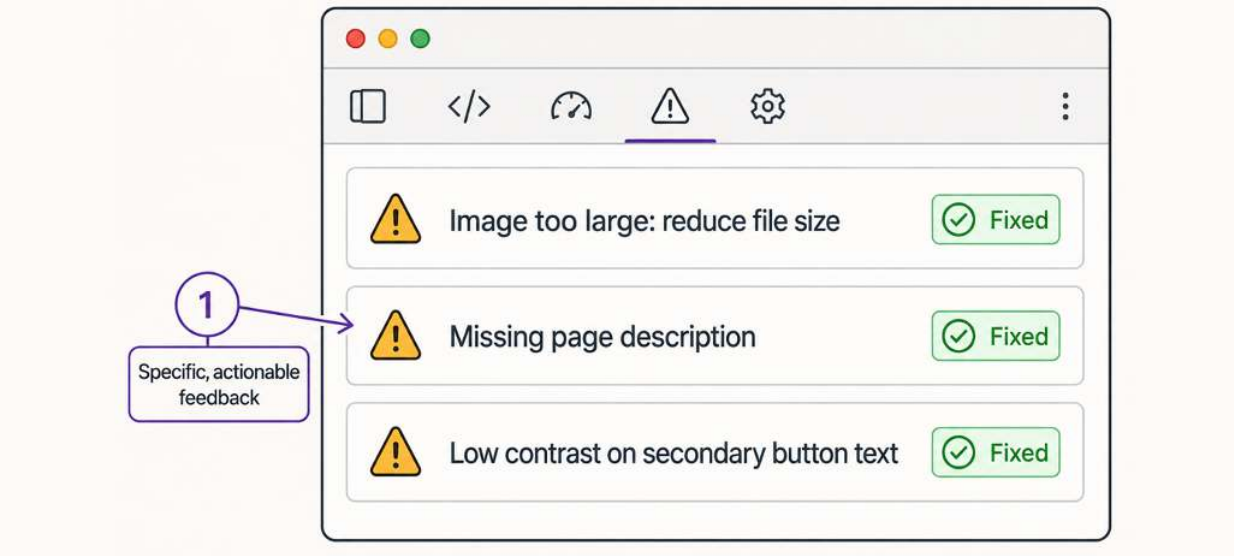


Figure 2.10B: A useful audit names specific, fixable problems. This chapter's work turns each flagged item into a fixed one.

Add title, description, and social card

Search engines and messaging apps rely on specific information in a page's `<head>` to represent it well: a title, shown in a browser tab and search results; a description, a short summary often shown beneath a search result; and social card information, controlling how a shared link appears when pasted into a chat or social post:

```
Add a distinct, accurate title and description to each of the three pa
```

Confirm each page's title and description describe that specific page, rather than reusing the same generic text everywhere.

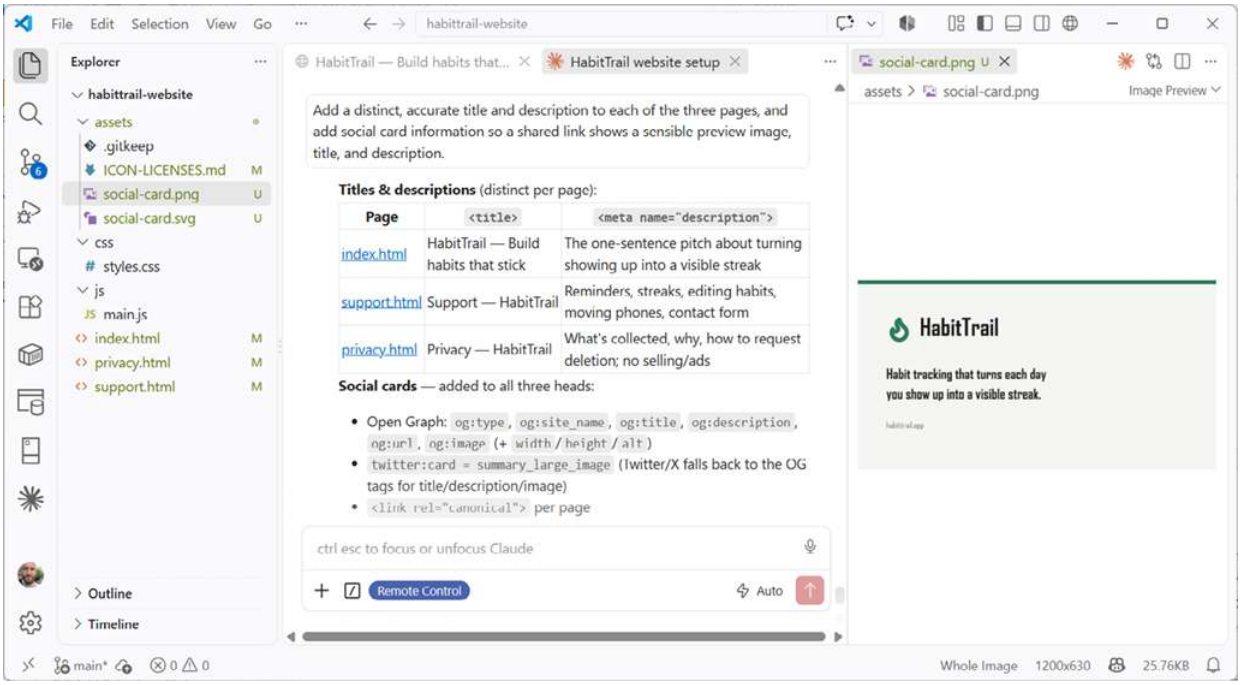


Figure 2.10C: Open Graph elements and a social card added to the website

Watch out : A fast, accessible site with an inaccurate or missing page title still fails visitors who find it through search or a shared link, before they ever load the page itself. All three parts of this audit matter, not just the one that's easiest to check.

Real-world example : A small-business owner discovered, only after her friend shared her site's link in a group chat, that it displayed with a broken preview image and a generic title reading only "Untitled." Adding proper social card information

fixed how the site appeared everywhere it got shared afterward, not just in that one chat.

Try it now

Audit HabitTrail's whole site the way a stranger would experience it: by keyboard, by loading speed, and by what a shared link shows:

1. Tab through the entire site by hand, confirming every interactive element is reachable with a visible focus indicator.
2. Ask Claude Code to check for and fix anything slowing the page down, especially images.
3. Add an accurate, distinct title and description to each of the three pages.
4. Add social card information and confirm it produces a sensible preview when a link is shared.



What to ask next : Once your fixes are applied, it's worth a final combined check:

```
Give me a summary of this site's current accessibility
```

Check the result

☐ Can you reach every interactive element on every page using only the keyboard?

☐ Has anything slowing the page down, especially oversized images, been fixed?

- ☐ Does each page have its own accurate title and description?
- ☐ Does sharing a link to the site produce a sensible preview image, title, and description?

HabitTrail's site now holds up under a real audit: usable by keyboard, reasonably fast, and ready to represent itself accurately wherever a link to it appears. The next chapter of this book takes it from your local preview to a public address anyone can visit.

Chapter 2.11 Publish the Website and Update It Safely

In this chapter, you will publish HabitTrail's website to a real, public address anyone can visit. You will push your project to GitHub, then connect it to Netlify or GitHub Pages for hosting, and confirm everything works correctly once it's live, not just in your local preview.

You need the completed, audited site from *Chapter 2.10* , and Git tracking your project. You also need a free GitHub account and a free Netlify account; create both now if you don't have them:

- Sign up for a GitHub account: https://github.com/signup/
- Sign up for a Netlify account: https://app.netlify.com/signup/

By the end of this chapter, HabitTrail's site will be live at a public web address:

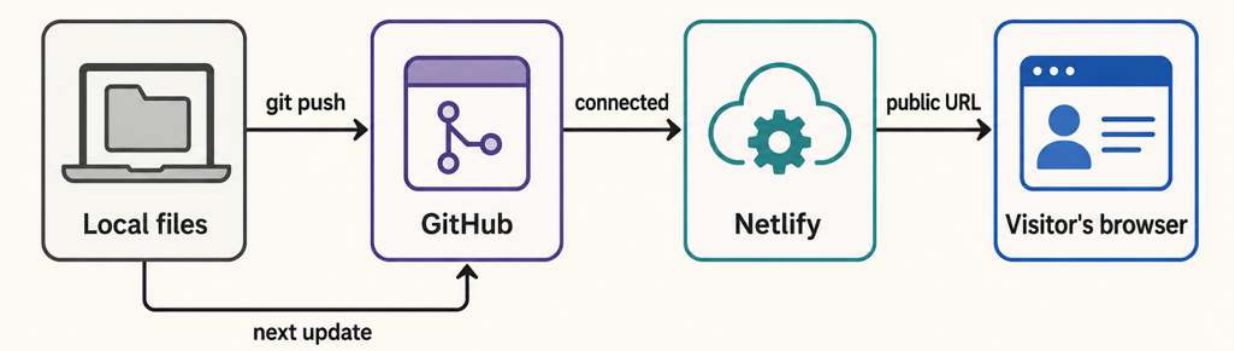


Figure 2.11A: The same commit habit from Book 1 now goes one step further: GitHub holds your history, and Netlify turns it into a public website

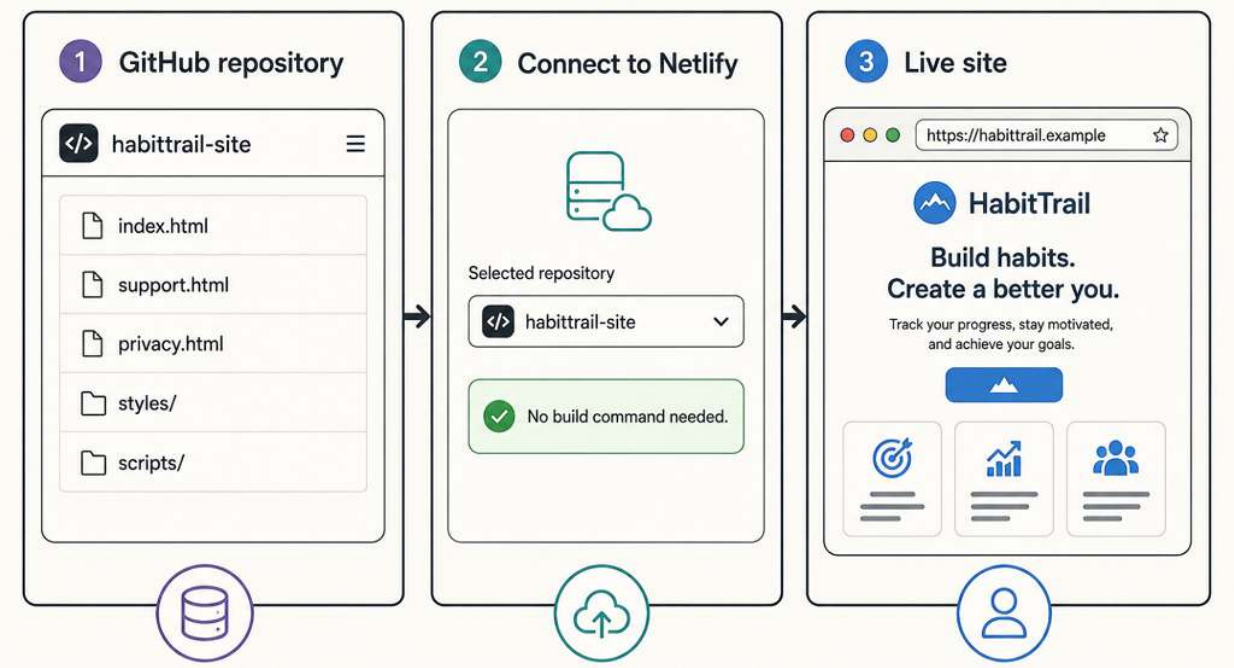


Figure 2.11B: A repository, a hosting connection, and a public address: three steps between your local project and a website anyone can visit

Run final checks

Before publishing, take one more pass through the site as a visitor would experience it, not as its builder. Load each page, click through the navigation, submit the contact form, and toggle the theme, one more time:

```
Before I publish this site, do a final review: check all three pages ]
```

Fix anything you find now. Publishing doesn't make a mistake harder to fix, but catching it beforehand means no visitor ever sees it.

Choose a hosting service: GitHub Pages or Netlify

Both GitHub Pages and Netlify are free services that turn a GitHub repository into a live website, and either one can host a plain HTML, CSS, and JavaScript site like HabitTrail's. They're not identical, though, and the difference that matters most for this project comes down to one thing: HabitTrail's Support page has a working contact form.

GitHub Pages only serves files. It has no way to run code on a server, so it can't do anything with a form submission on its own. A visitor could fill out the form and click submit, but there'd be nowhere for that data to go, unless you connected a separate service built for exactly this, such as Formspree, and wired your form up to it.

Netlify can run that same form without any extra service. Add one attribute to the form element, and Netlify collects every submission for you, viewable from its own dashboard. That's the reason this book uses Netlify: it's the option that makes the form you already built in *Chapter 2.9* actually work, without adding a second account and a second set of instructions this late in the book.

Table 2.11A summarizes the difference for a small site like HabitTrail's:

	GitHub Pages	**Netlify**
Setup for a static site	Flip on Pages in your repository's	Connect your repository through Netlify's

	settings	dashboard
Handles a contact form	No, needs a separate service	Yes, built in
Custom domain and HTTPS	Free, manual DNS setup	Free, guided setup
Cost for a small site	Free	Free

Table 2.11A: GitHub Pages compared with Netlify for hosting a small website

If HabitTrail's site had no form at all, GitHub Pages would be the simpler choice, since it needs nothing beyond a GitHub account you already have. The form is what tips the decision toward Netlify here.



Good practice : If you ever build a static site with no form and no need for one, GitHub Pages is worth choosing for its simplicity. Pick the hosting service that matches what your specific site actually needs, rather than defaulting to whichever one you used last time.

Push to GitHub and deploy to Netlify

GitHub hosts a copy of your Git repository online, and Netlify is a hosting service that can automatically publish a website from that

repository. Together, they turn your local project into a live, public site.

1. Create a new, empty repository on GitHub (https://github.com/new):
    - **Repository name** : `habittrail`
    - **Add README** : Off

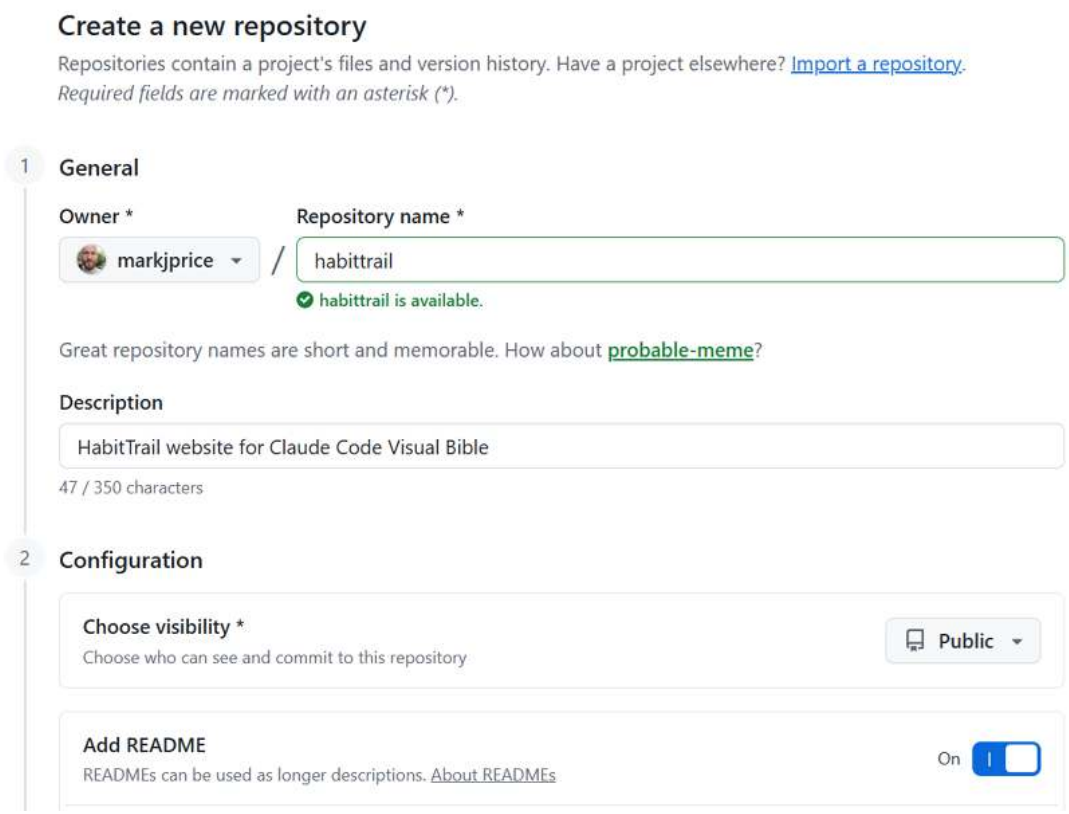


Figure 2.11C: Setting options for a new GitHub repository

2. Select **Create repository** .
3. Ask Claude Code to connect your local project to it and push your commits:

```
Connect this project to the GitHub repository I just create
```



Watch out : Replace `<your-account>` with your GitHub account name.

4. Sign in to Netlify and connect it to that same GitHub repository, following Netlify's prompts to select your repository and confirm the project needs no special build step, since this site is plain HTML, CSS, and JavaScript.

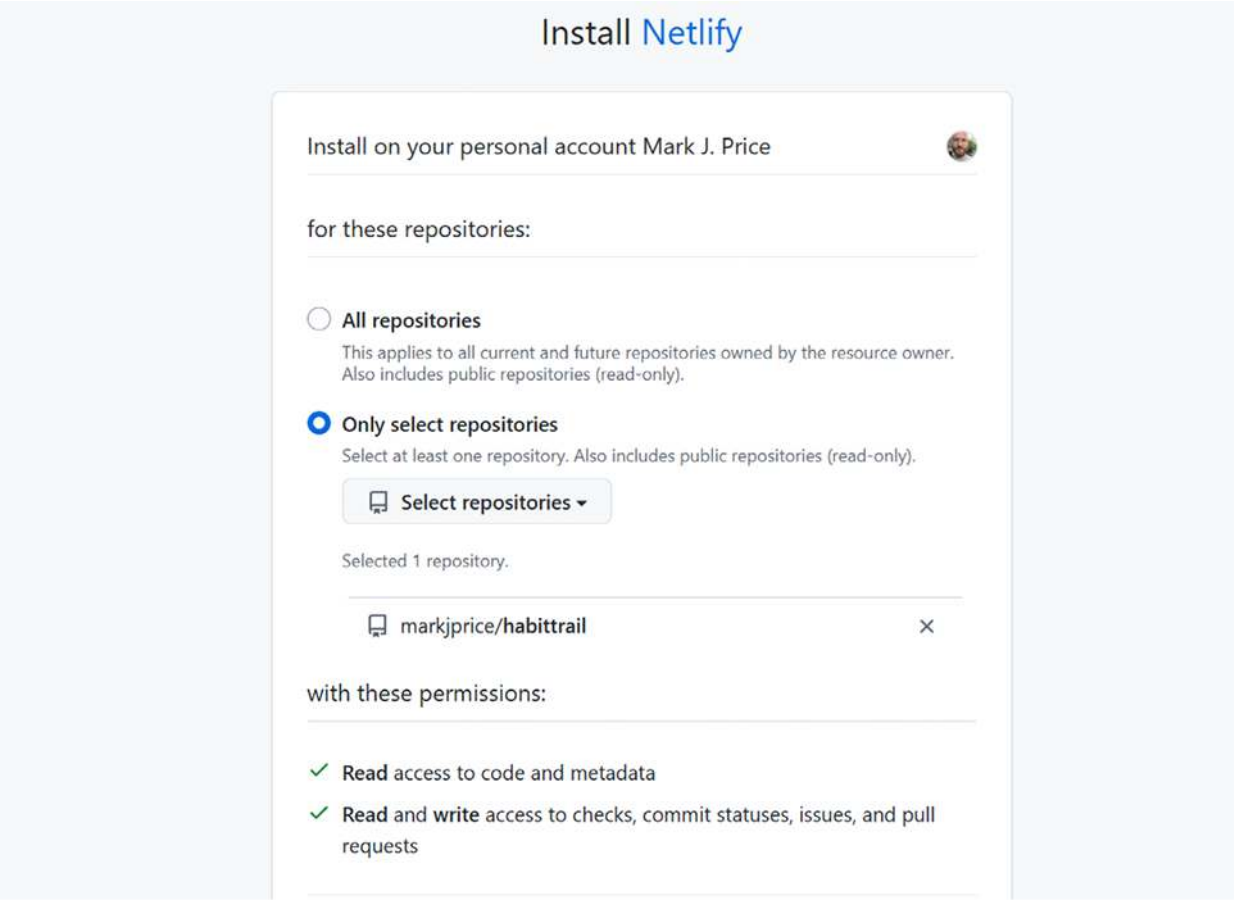


Figure 2.11D: Installing Netlify for the HabitTrail GitHub repository

5. Once connected, Netlify builds and publishes your site, giving you a public web address.

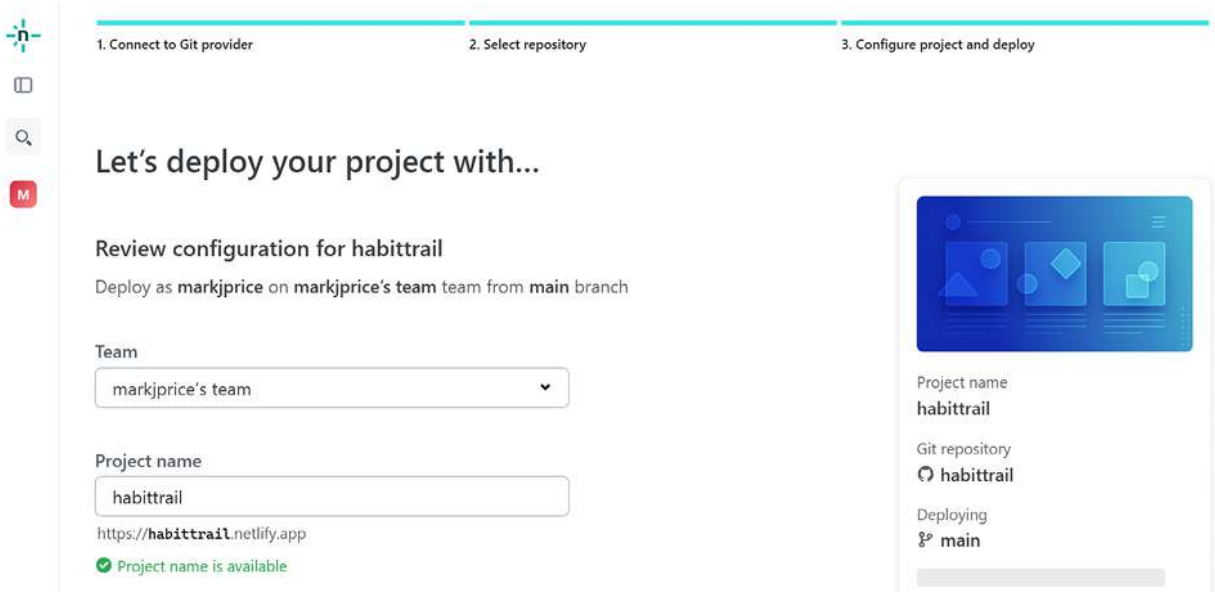


Figure 2.11E: Deploying the website to Netlify

6. Start your preferred browser and navigate to the public URL. For example, my version is available at the following link: `https://habittrail.netlify.app`
7. By default, for protection, your site will only be visible to project members like yourself:

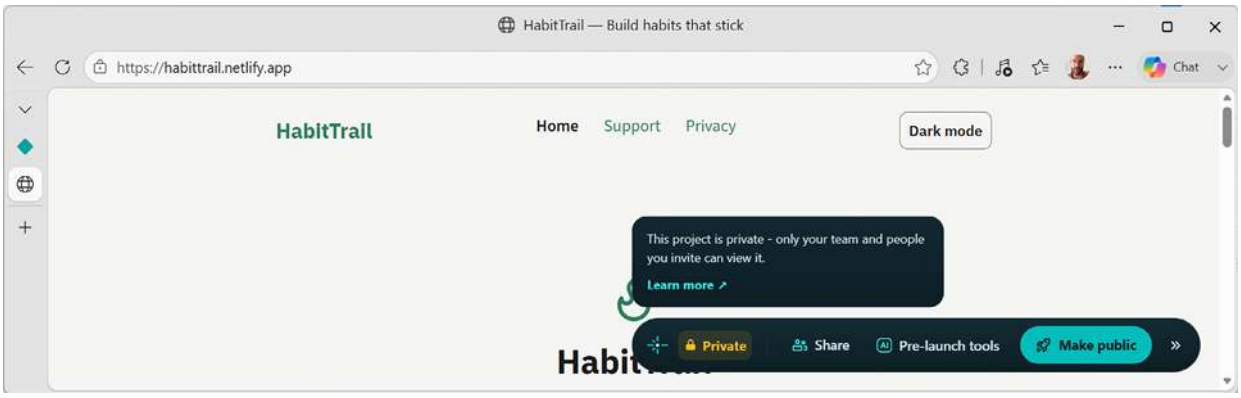


Figure 2.11F: Your website is private to your Netlify project members by default

8. Select **Make public** .

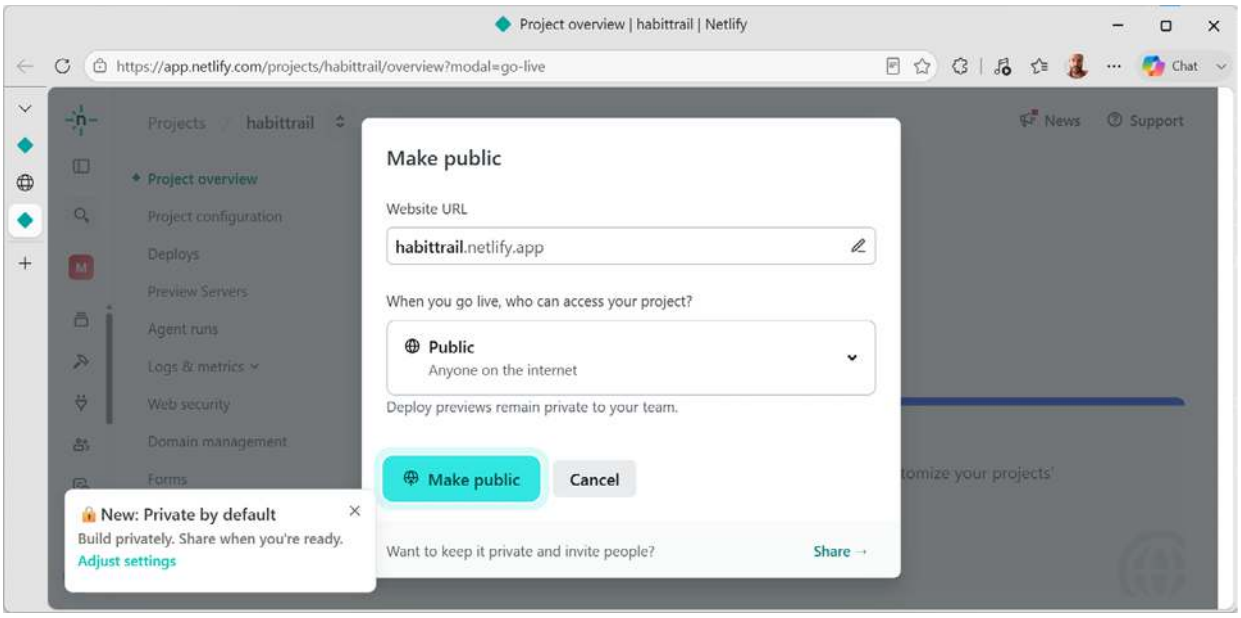
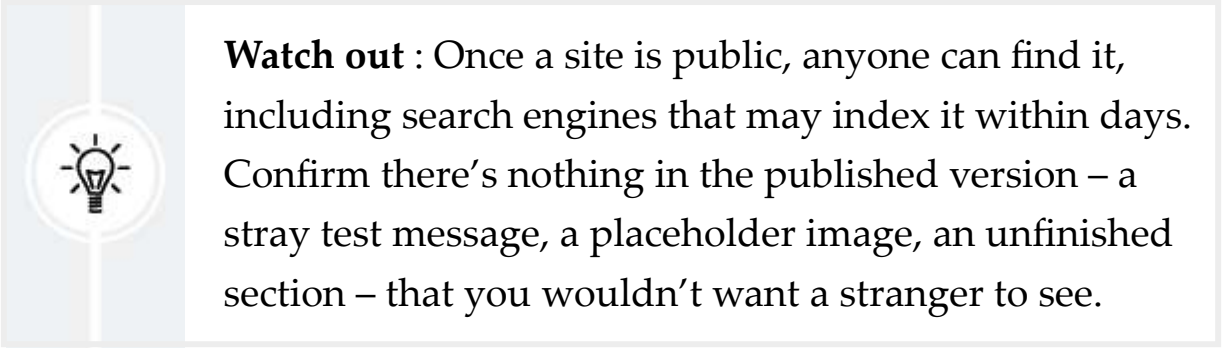


Figure 2.11G: Making the website public

9. Anyone on the public internet will now be able to visit your website.



Watch out : Once a site is public, anyone can find it, including search engines that may index it within days. Confirm there's nothing in the published version – a stray test message, a placeholder image, an unfinished section – that you wouldn't want a stranger to see.

Verify HTTPS, links, support, and privacy URLs

Once your site is live, check it the way a real visitor would: open the public address in a browser you weren't using while building it, ideally on your phone as well as your computer.

Confirm the address begins with `https://` , showing the connection is encrypted, not just `http://` . Click through every navigation link, and specifically confirm your Support and Privacy pages load correctly at their public addresses, since these are the pages a cautious visitor is most likely to check before trusting HabitTrail at all:

```
Here is my site's public address: [paste it]. Walk through what a firs
```

To update the site later, push a new commit to GitHub. Netlify rebuilds and republishes automatically, so keeping the site current is as simple as continuing the same commit habit from *Book 1* .

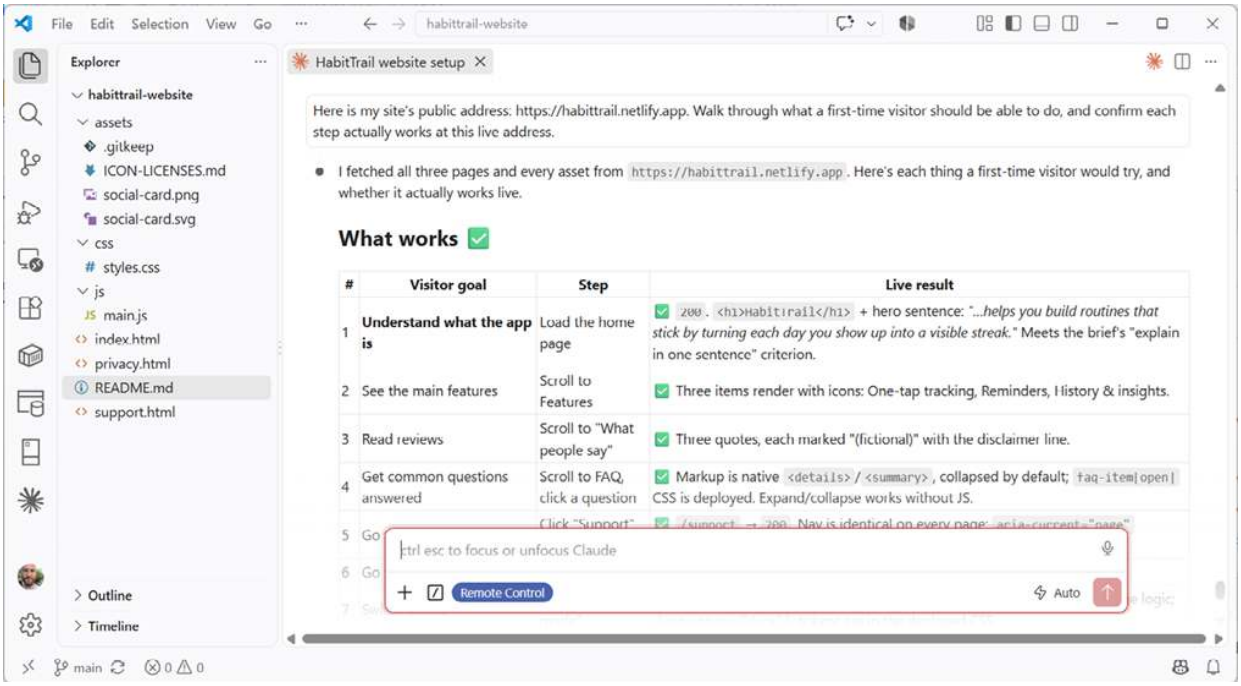


Figure 2.11H: Asking Claude to walk through the public website



Good practice : Test your live site from a device and network you weren't using while building it, such as your phone on mobile data. A problem invisible on

your own development computer sometimes only appears somewhere else.



Real-world example : A freelancer published her first site late one evening and moved on without checking it from her phone. A friend messaged her the next morning to say the contact form was broken on mobile, something that had worked perfectly on her laptop. Checking from a different device before considering a launch finished would have caught it sooner.

Try it now

Take HabitTrail from your local preview to a public address anyone can visit:

1. Do a final review of the whole site as a visitor would experience it, and fix anything you find.
2. Create a GitHub repository and push your project's commits to it.
3. Connect that repository to Netlify and let it build and publish your site.
4. Open the public address on a different device and verify HTTPS, navigation, and both the Support and Privacy pages.



What to ask next : Once your site is live and verified, it's worth planning how you'll keep it current:

```
What's the simplest way for me to update this site
```

Check the result

- ☐ Does your site's public address begin with `https://` ?
- ☐ Do all three pages load correctly at the public address, on a device you weren't using to build it?
- ☐ Does the contact form still validate correctly on the live site?
- ☐ Do you know how you'll publish an update the next time you change something?



Learn more online : GitHub and Netlify both offer more advanced options than this chapter covers, including custom domain names and preview deployments for changes you haven't published yet. Search each service's own documentation for current, detailed steps, since account flows and settings can change after this book is printed.

HabitTrail's website is live, public, and yours to keep updating with the same commit-and-push habit you already know.

Chapter 2.12 Project – Build and Publish a Website You Would Use

This chapter is your final project for *Book 2*. You will decide what a small, real website should say, build it with Claude Code using everything from this book, and publish it to a public address anyone can visit. There is no new skill to learn here. There is a website to finish.

You need everything from this book up to this point: a habit of planning a site map and success criteria before opening Claude Code, a habit of asking for a plan and reviewing it, a structured home page built with semantic HTML, a small visual system in CSS, a layout that holds up on a phone as well as a desktop screen, real navigation between pages, and a habit of committing checkpoints as you go. Budget more time for this chapter than earlier ones, at least ninety minutes, possibly spread across more than one sitting, since this time your website is going all the way to a public web address instead of stopping at a local preview:

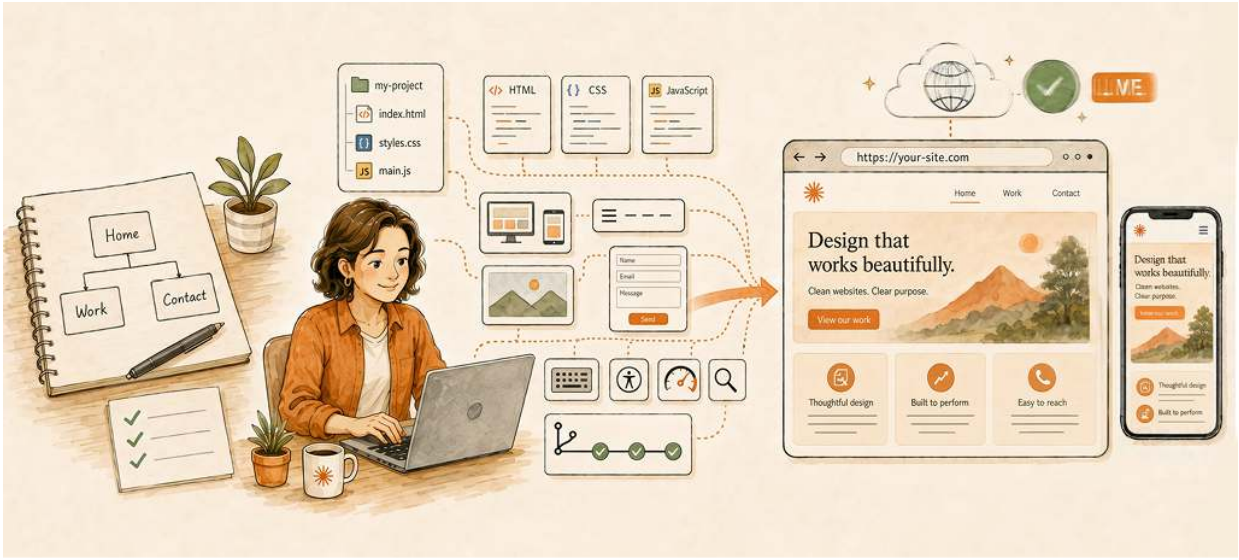


Figure 2.12A: Every skill from this book, from a site map on paper to a published address, comes together in one finished website

Decide what your website will be

Throughout this book, you built HabitTrail's website: a home page, a support page, and a privacy page for an app that doesn't exist outside these pages. That kept the book focused on one example while you learned HTML, CSS, JavaScript, and the habits around them. This chapter asks for something different: a small website for an idea of your own, built and published for real.

A good first website doesn't need a business behind it, or even a real product. It needs a specific person, a specific reason someone would visit, and two or three pages that do that job well. If you already have an idea, use it. If not, here are five starting points that suit a first published website well, each based on a kind of person many readers will recognize, including possibly themselves.

A recent graduate or course-completer

Someone who just finished a degree, a bootcamp, or a certificate and wants a place to show finished work to someone who might hire them. A home page introducing who they are and what they just completed, a work or projects page with two or three finished pieces, and a contact page or a clear way to reach them, covers this well.

A creative with work to show

A photographer, illustrator, musician, or writer who has real photos, images, or writing sitting on a hard drive or a phone, with nowhere public to point people. A home page, a gallery or portfolio page holding that actual media, and a short about page explaining who made it and how to get in touch, is enough for a first version.

A small business owner

Someone running a small, local business, such as a café, a tutoring service, a dog-walking round, or a craft stall, who wants visitors to find their hours, what they offer, and how to reach them, without depending entirely on a social media page they don't fully control. A home page, a services or menu page, and a contact page with real hours and a real way to get in touch covers the essentials.

A professional looking for a new position

Someone between jobs, or ready for a change, who wants one link to send instead of a résumé attachment: a home page summarizing their experience, a page listing specific skills and past work, and a way to download a résumé or get in touch.

Someone running a local group or event

Someone organizing a hobby club, a community meetup, a book club, or a one-off event, who needs a single place to explain what it is, when it meets, and how to join, instead of repeating the same explanation in every message they send. A home page explaining the group, a schedule or details page, and a page explaining how to join or attend covers this well.

None of these need to be practical or entirely real. If nothing on this list fits, adapt one, or combine ideas: a creative professional's portfolio can double as their job-search site, for example. Pick whichever one you can picture finished, since that's the version you're actually about to build.



Good practice : Choose an idea backed by content you actually have or can quickly gather, such as real photos, a real short biography, or a real list of skills. A website built around content you have to invent as you go is harder to judge than one built around something real, even if the business or project behind it is imaginary.

Plan the site the way you planned HabitTrail's

Reuse the planning habit from *Chapter 2.1* , applied to your own idea this time:

- Name the one or two kinds of visitor your site needs to serve, and the one action each should be able to complete

- Sketch a site map, including which pages you need and how they link to each other (two to four pages is enough for a first published site)
- Write one or two sentences of required content for each page
- Write one success criterion per page that you could actually check once the site exists



Watch out : A portfolio site with ten planned pages, or a small business site trying to cover every service you might ever offer, is not a first version. It's a wish list. If your site map has more than four pages, look for the ones you could publish later instead, once the first version is live and you've learned something from having it out in the world.



Real-world example : A reader finishing a UX design course wanted a portfolio covering four different projects, a case study for each, a blog, and a contact form with file uploads for potential clients. Writing a site map first cut that down to a home page and a single projects page holding three finished pieces, with a `mailto` link standing in for a contact form. The bigger version became a plan for later, once the small one was live and she had feedback to act on.

Manage source control with Git

Treat Git as part of building the site, not something you do afterward, the same habit from *Chapter 1.9* and every chapter since:

- Initialize Git early, before you have anything you'd be upset to lose
- Commit after each working increment, with a specific message describing what that checkpoint contains
- If an experiment goes badly, such as a layout change that breaks on mobile, use an earlier commit as your way back instead of trying to manually undo a tangle of changes

Before you consider the site finished, look back through your commit history. Like your project in *Chapter 1.12* , it should read like a short, sensible story of how the site came together.

Build the site with Claude Code

Work through the same loop you've practiced all through this book, one small increment at a time:

- Create a new project folder and open it in VS Code.
- Describe your site map and page content to Claude Code, and ask it to suggest and create a sensible folder structure with starter HTML files, the same way you did in *Chapter 2.2* . Start a local preview, confirm every starter page loads, then initialize Git and commit the starting structure. Structure your home page with semantic HTML: meaningful regions, a clear heading, and a specific call to action.
- Apply a small visual system: colors, type, and spacing, checked for contrast, the way you did in *Chapter 2.3* .

- Lay out your sections with Flexbox and Grid, and confirm the layout still works if you add or remove one item, the way you did in *Chapter 2.4* .
- Make the design responsive, starting from a narrow phone width, the way you did in *Chapter 2.5* .
- Build out your remaining pages and connect them with accessible, keyboard-reachable navigation, the way you did in *Chapter 2.6* .
- Add real images or media if your idea needs them, with responsive sizing and accessible `alt` text, the way you did in *Chapter 2.7* .
- If your idea benefits from a small piece of behavior, such as an expandable section or a theme toggle, add it the way you did in *Chapter 2.8* . Not every site needs this. Skip it if your site works better as simple, static pages.
- If your site needs a way for a visitor to reach you, build and validate a contact form the way you did in *Chapter 2.9* , including what happens to a message once it's submitted.
- Run through the same audit from *Chapter 2.10* : check keyboard access across every page, fix anything slowing the site down, and add an accurate title, description, and social card for each page.
- Commit your work after each of these increments, not only once at the end.



Good practice : Test your site against the success criteria you wrote for each page, not only against whatever you happen to click first. A portfolio page that looks right on your own laptop should also be

checked on a narrow phone width and with the keyboard alone.



Real-world example : A reader building a site for his dog-walking round skipped the contact form entirely and used a single mailto link and a phone number in the footer instead, once he realized Chapter 2.9's form validation was more building than a one-person, three-client business actually needed. Recognizing which chapters his idea needed, and which it could skip, was as useful a decision as any of the building itself.

Choose where to publish it

Before you publish, decide between GitHub Pages and Netlify using the same question from *Chapter 2.11* : does your site need to handle a form submission on its own?

If your site has a working contact form, use Netlify, the same way HabitTrail's site does.

If your site has no form, or only a `mailto` link or an external booking link, GitHub Pages is the simpler choice, since it needs nothing beyond the GitHub account you already have.



Watch out : Once your site is public, anyone can find it, including search engines that may index it within days. Check every page for placeholder text, a testimonial

you meant to replace, or a photo you didn't have the rights to use, before that happens.

Try it now

Put the whole chapter to work on your own idea, from a blank folder to a public address:

1. Pick one of the five ideas above, or your own, and write a short site map with two to four pages, plus a success criterion for each.
2. Set up the project: a new folder, Git initialized, and starter files matching your site map.
3. Build the site in small, committed increments: structure, visual system, layout, responsiveness, navigation, and any images or interactivity your idea needs.
4. Audit the finished site for keyboard access, loading speed, and accurate page titles and descriptions.
5. Choose GitHub Pages or Netlify based on whether your site needs to handle a form, and publish it.
6. Open the public address on a device you weren't using to build it, and confirm every page and every link works there.



What to ask next : Once your site is live, it's worth asking what you'd change if you kept going:

```
Now that this site is live, what would be the smallest
```

> Keep the answer as a note for later. You do not need to build it now.

Check the result

- ☐ Does your published site do what your site map and success criteria describe, checked for real, not assumed?
- ☐ Does every page load correctly at your site's public address, on a device you weren't using to build it?
- ☐ Could you send this address to the actual kind of person it's for, the way you described them at the start of this chapter, without feeling like you need to explain what's missing?
- ☐ Does your commit history read as a sensible sequence of small, working steps, from an empty folder to a published site?

You planned a real idea, built it in small reviewed steps, and published it to an address anyone can visit. *Book 2* is complete.

End of Book 2

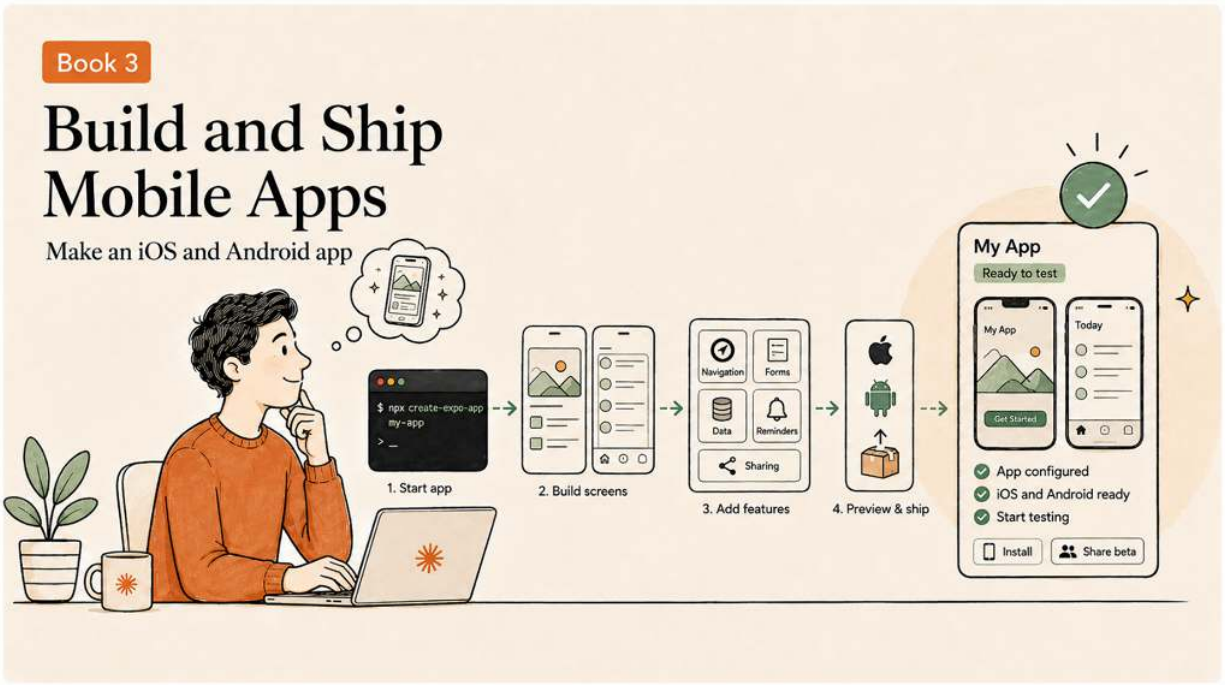
You started *Book 2* with a plan on paper for HabitTrail's three pages, and you finish it having planned, built, and published a second website of your own choosing, from a site map to a live public address. The languages and the loop are now yours: HTML for structure, CSS for a visual system and layout, JavaScript for behavior, and the same idea-plan-approval-edits-tests-review habit from *Book 1* applied to something the rest of the world can actually open.

Book 3 returns to HabitTrail, but changes what you're building and how. Instead of a website anyone can open in a browser, you'll build the habit-tracking app itself, an iOS and Android app made with React Native and Expo, installed on a real phone rather than visited at a URL. The habits carry forward unchanged: plan first, build in small increments, review every change, commit checkpoints you trust, and check the result yourself before calling anything finished.

Book 3

Build and Ship Mobile Apps

Book 3 continues with the HabitTrail product. *Book 2* put your working method to use on something anyone could visit: a public website, live at its own address, that a visitor could open in a browser. *Book 3* keeps the same product, HabitTrail, but instead of a webpage someone visits, you'll build a mobile app that they install.



HabitTrail becomes a real iOS and Android app in this book, built with React Native, Expo, Expo Router, and TypeScript.

Where *Book 2* used plain HTML, CSS, and JavaScript so you could see the web's foundations, *Book 3* adds a framework and a typed language on top, because a mobile app asks more of its tools: native screens, on-device storage, and behavior that has to hold up on a phone in someone's hand, not just in a browser window. The app stays offline-first throughout. Everything it does happens on the user's own device, with no account and no server, which keeps the project small enough to finish while still teaching real, useful mobile skills.

You'll start by defining one specific person and the habit they're trying to build, then create a real Expo project and get it running on your own phone within your first sitting. From there, the same small, reviewed increments from the last two books carry you through building screens from reusable components, moving between them with a router, making the app respond to a tap, and building a form that won't let through a habit with no name. A full set of create, read, update, and delete operations follows, along with the two things that make a habit tracker worth opening again: data that survives closing the app, and a reminder that arrives at the right time.

The final chapters turn your attention outward. You'll test HabitTrail on a real device and with a screen reader, then prepare its name, icon, and store listing, and finally build, beta test, and submit it, so someone who has never seen HabitTrail before can install it and try it for themselves. By the end of this book, you'll have carried one idea through all three books in this volume: planned, built, published, and shipped.

Book 3 has fourteen chapters, including an end-of-book project:

3.1 Define a Mobile App’s Features 122

3.2 Create and Run an Expo App 126

3.3 Understand the Mobile Project 131

3.4 Build Screens from Components and Styles 134

3.5 Move Between Screens with Expo Router 139

3.6 Make the App Respond with Props and State 142

3.7 Capture and Validate Form Input 146

3.8 Create, Read, Update, and Delete Habits 148

3.9 Keep Data Between Sessions 151

3.10 Add Reminders and Sharing 154

3.11 Test on Real Devices and Repair Problems 157

3.12 Prepare the App for Release 160

3.13 Build, Beta Test, and Submit 162

3.14 Project – Build and Ship a Mobile App You Would Use 165

C hapter 3.1 Define a Mobile App’s Features

In this chapter, you will define the mobile app you will build across the rest of this book: HabitTrail, an offline-first habit tracker for iOS and Android, matching the website you built in *Book 2*. You will choose a specific user and the habit they’re trying to build, scope the

smallest useful version of the app, and map its screens and the data each one needs.

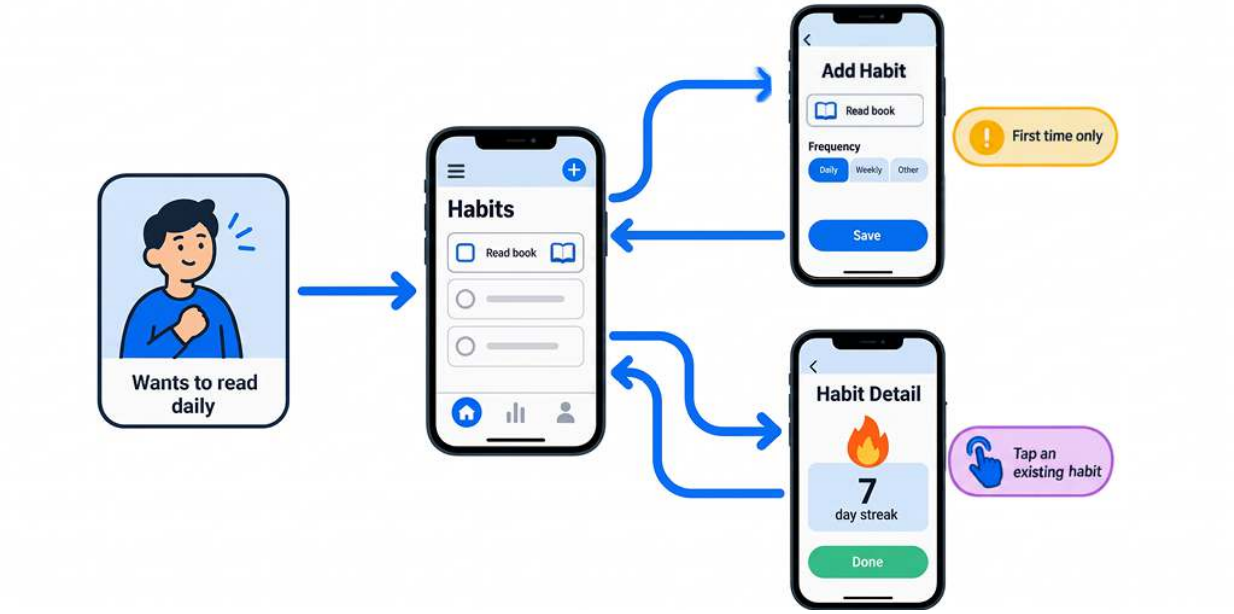


Figure 3.1A: One person, one repeated habit, three screens: add it once, then check in and mark it done

You need your notebook or note app. Nothing here requires Expo or a phone yet. By the end of this chapter, you will have a project brief you can hand to Claude Code, and a three-screen map that guides the rest of this book.

HabitTrail stays offline-first throughout this book: everything it does happens on the user's own phone, with no account, no server, and no internet connection required. That keeps the project realistic to finish while reading this book, while still teaching real, useful mobile app skills.

Choose the user and repeated task

A habit tracker only matters if it helps with something a person actually repeats. Pick one specific, believable example to design

around, such as someone trying to read for ten minutes every day, or someone trying to drink more water. Naming a specific habit, rather than "habits in general," makes every screen easier to design, since you can ask concretely what that person needs to see and do.

Write one or two sentences naming this person and their repeated task, the same way you named a user and problem in your *Book 1* brief.

Scope the habit tracker

The smallest useful version of HabitTrail lets someone add a habit, mark it done for today, and see how they're doing. That's enough to be genuinely useful, and small enough to actually finish.

Write down what this first version will and won't do. Non-goals are just as important as what the app will do since they provide guardrails for Claude Code and limit its scope: no user accounts, no syncing between devices, no social features, no reminders more complex than a single daily time. Each of these is a reasonable idea for a later version, and each is explicitly out of scope for the app you're building in this book.



Figure 3.1B: The same brief pattern from Book 1, applied to a mobile app: what it does, what it doesn't yet, and the screens that will be implemented



Watch out : It's tempting to plan reminders and social sharing all at once. Reminders and sharing appear later in this book, in *Chapter 3.10* . Planning them now as non-goals keeps this chapter's brief honest about what version you're actually building first.

Map screens and stored data

HabitTrail needs three screens:

- **Habits** : A list of the user's habits, showing which ones are done today.
- **Add Habit** : A form for creating a new habit.
- **Habit Detail** : A single habit's history and streak, reached by tapping it from the list, with options to edit or delete it.

For data, each habit needs at minimum a name, the date it was created, and a record of which days it was marked complete. Sketch this as a short list, not code. You'll turn it into an actual data shape once you're inside a real project in later chapters.



Good practice : Design around one specific, believable habit, not an abstract idea of "any habit." A concrete example makes it far easier to notice when a screen is missing something the user would actually need.



Real-world example : A reader planning her own version of this app wanted to track a daily stretching routine. Naming that specific habit, rather than designing for "fitness tracking" broadly, made it obvious her Add Habit screen needed a simple daily reminder time and nothing more elaborate, since that was genuinely all her own habit needed.

Try it now

Turn your idea for HabitTrail into a written brief and a three-screen map, ready to hand to Claude Code:

1. Choose a specific person and the repeated habit they're trying to build.
2. Write the smallest useful version of HabitTrail: what it will do, and what it deliberately won't yet.
3. Sketch the three screens: Habits, Add Habit, and Habit Detail.

4. List the data each habit needs to store.



What to ask next : Once your brief and screen map feel solid, ask Claude Code to check them before you build anything:

```
Here's my plan for a habit-tracking app: [describe you
```

Check the result

- ☐ Have you named a specific user and a specific habit, not habits in general?
- ☐ Does your non-goals list include at least three things you're deliberately not building yet?
- ☐ Does your screen map show all three screens and how a user moves between them?
- ☐ Have you listed the minimum data one habit needs to store?



Sample files : A complete example mobile app project brief is available in the GitHub repository: https://github.com/markjprice/claude-code-vb/blob/main/files/book-3/3-1-mobile-app-brief.md

You now have a clear brief and a three-screen map for an app small enough to finish and real enough to actually use. The next chapter

turns that plan into an actual project, using Expo to create a real app you can run on your own phone within minutes.

Chapter 3.2 Create and Run an Expo App

In this chapter, you will create a real Expo project and run it on your own phone. Expo is a framework built on top of React Native that handles much of the setup a mobile app normally needs, so you can go from an empty folder to an app running on your phone in one sitting.



Requirements : You need a computer with Node.js installed, and a phone with the free Expo Go app installed from its app store. Instructions to install Node.js are available online: https://github.com/markjprice/claude-code-vb/blob/main/docs/install-nodejs.md

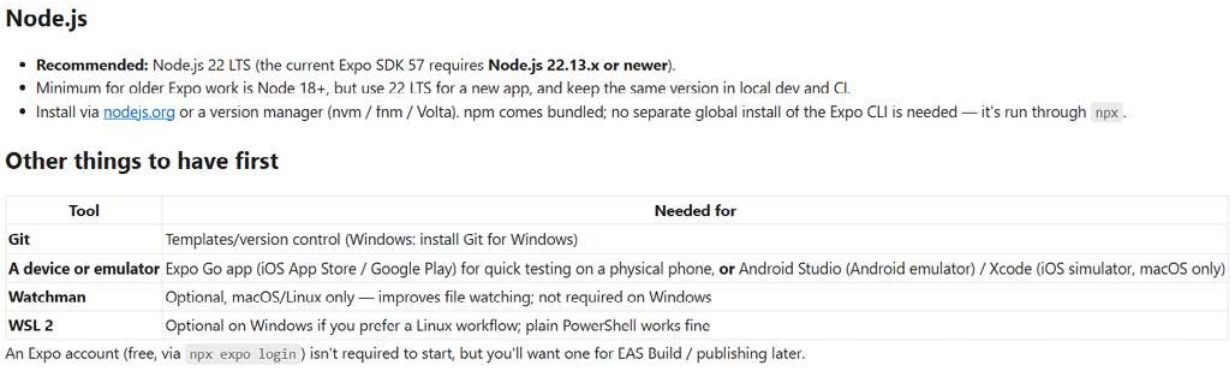
By the end of this chapter, a starter app will be running live on your phone, updating as you make changes on your computer.

Install the current prerequisites

Expo's exact requirements can change between versions, so confirm the current ones before starting, rather than relying on what an older guide says:

```
Check the current requirements for creating a new Expo app: what versi
```

Follow whatever Claude Code finds. If anything is missing, ask it to help you install it:



Node.js

- **Recommended:** Node.js 22 LTS (the current Expo SDK 57 requires **Node.js 22.13.x or newer**).
- Minimum for older Expo work is Node 18+, but use 22 LTS for a new app, and keep the same version in local dev and CI.
- Install via nodejs.org or a version manager (nvm / fnm / Volta). npm comes bundled; no separate global install of the Expo CLI is needed — it's run through `npx`.

Other things to have first

Tool	Needed for
Git	Templates/version control (Windows: install Git for Windows)
A device or emulator	Expo Go app (iOS App Store / Google Play) for quick testing on a physical phone, **or** Android Studio (Android emulator) / Xcode (iOS simulator, macOS only)
Watchman	Optional, macOS/Linux only — improves file watching; not required on Windows
WSL 2	Optional on Windows if you prefer a Linux workflow; plain PowerShell works fine

An Expo account (free, via `npx expo login`) isn't required to start, but you'll want one for EAS Build / publishing later.

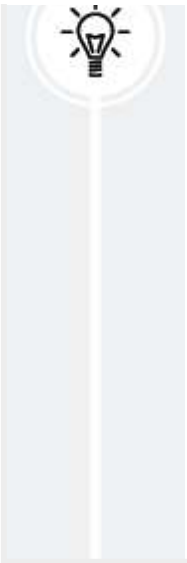
Figure 3.2A: Claude Code checking requirements for building Expo mobile apps

Ask Claude to create a TypeScript Expo project

Create a new folder for this project, separate from your *Book 1* and *Book 2* projects, for example named `habittrail-app`, open the folder in VS Code, and then ask Claude Code to set it up:

```
Create a new Expo app in this folder called habittrail-app, using Type
```

Watch out : At the time of writing in September 2026, on the iOS App Store, Expo Go 54.0.2, released September 23, 2025, is the latest version available. On Android Play Store, Expo Go 54.0.8, updated May 12, 2026, is slightly ahead of iOS but it is the same SDK major version. They both only work with Expo projects created with Expo SDK 54, despite the latest SDK



version being 57. A version of Expo Go for later SDKs is still waiting for Apple App Store approval, with no timeline available, so Expo Go for SDK 54 continues to be the version on both the App Store and Play Store in the meantime. The workaround for developers who need newer SDK support is `eas go`, which builds you a personal copy of Expo Go delivered via TestFlight but this requires a paid Apple Developer account. In this book, we will limit ourselves to SDK 54.

A new Expo project created this way starts with TypeScript and file-based routing already configured, along with a basic navigation structure you'll build on in later chapters. You don't need to configure either from scratch.

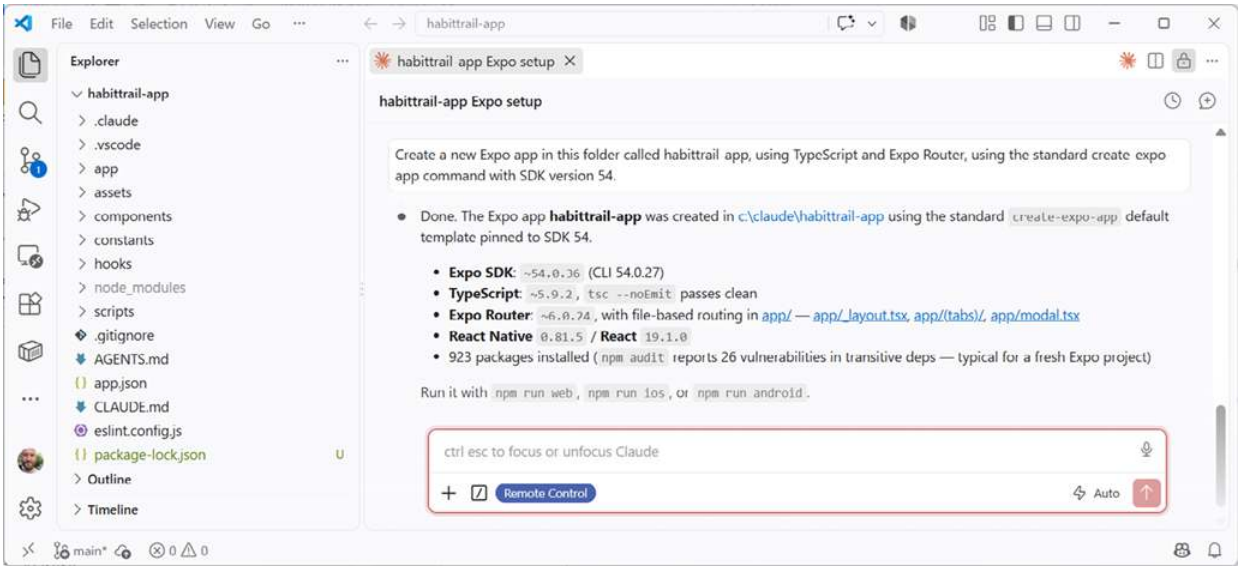


Figure 3.2B: A newly scaffolded Expo mobile app project using SDK 54

Run it on a phone with Expo Go

Expo Go is a free app that runs your in-progress project directly on your phone, without needing a full app store build. Ask Claude Code to start your project's development server:

```
Start the development server for this project.
```

Select the **Metro Bundler** link to open the app in the integrated browser:

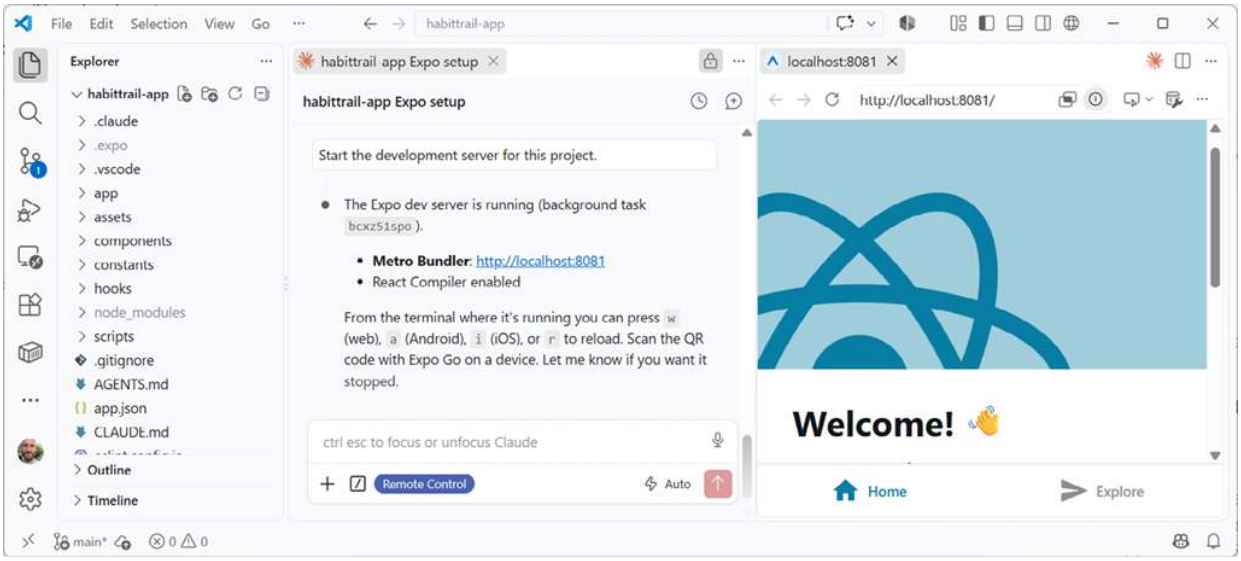


Figure 3.2C: Running the web version of the mobile app in the integrated browser

Claude Code might run the server as a background process, so its QR code never appears in your VS Code terminal. You can run it yourself and see the QR code:

```
npx expo start
```

A QR code should appear in your terminal:



Figure 3.2D: Your computer runs the project. Expo Go on your phone shows it live, updating the moment you save a change.

On Android, open Expo Go and scan it from within the app. On iOS, scan it using your phone's regular camera app, which will prompt you to open it in Expo Go. Your starter app should appear on your phone within moments.



Watch out : Your phone and computer usually need to be on the same Wi-Fi network for this to work. If the QR code doesn't connect, that's the first thing worth checking.

Navigate around the mobile app:

1. Initially a note about the developer menu will pop up above the app. Select **Continue** to see the developer menu.

2. The developer menu shows which Expo SDK was used to build the app, and provides tools like a performance monitor that you can show or hide. You can get back to the developer menu by shaking your phone or long-pressing with three fingers.
3. You will then be presented with the **Home** page, with three steps.
4. Selecting the **Explore** button will open the **Explore** page, with features you tap to expand and collapse more details.

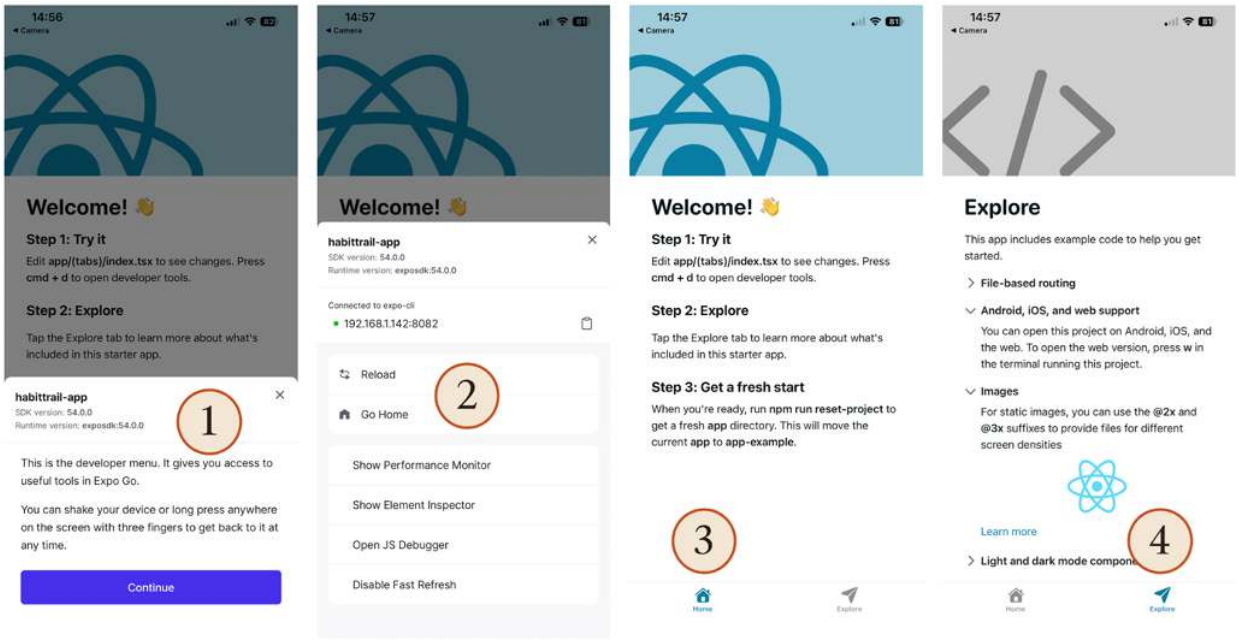


Figure 3.2E: The example project template app running on an iPhone using Expo Go

Making live changes to the mobile app

Try changing a small piece of visible text in the starter project, save the file, and watch it update on your phone (press *r* in the terminal if it does not):

- `app/(tabs)/index.tsx` : this is the **Home** page in the mobile app

- `app/(tabs)/explore.tsx` : this is the **Explore** page in the mobile app

This live-updating connection between your computer and your phone is one you'll rely on throughout this book.



Good practice : Confirm your starter app runs successfully on your phone before making any changes to it. A working starting point makes it much easier to tell, later, whether a problem came from something you changed.



Real-world example : A student building her first mobile app was surprised how quickly she saw something real on her own phone, compared to how long she'd expected mobile development to take to produce a visible result. Seeing a change she made on her laptop appear on her phone within a second or two, without reinstalling anything, made the rest of this book feel achievable rather than intimidating.

Stopping an Expo app

To stop an Expo project once you're done testing on your phone:

- **Stop the dev server on your computer.** In the terminal where you ran `npx expo start` , press *Ctrl+C* (the same on Windows, Mac, and Linux). That kills the Metro bundler process. You should see the terminal return to a normal prompt and the message, `Stopped server` .

- **On your phone.** Just close the Expo Go app normally (swipe it away from your app switcher) because there's nothing to "stop" on the phone side beyond closing the app, since it was just connecting to your computer's dev server over your local Wi-Fi network.

Occasionally the Metro bundler port (usually 8081) can stay held by a lingering process, especially on Windows. If you try to start again and get a "port already in use" error, you can find and kill it in PowerShell:

```
netstat -ano | findstr :8081
taskkill /PID <PID_number> /F
```

Try it now

Create a real Expo project and get it running live on your own phone:

1. Confirm you have the current prerequisites installed, following Claude Code's check.
2. Create a new Expo app using TypeScript and Expo Router.
3. Start the development server and scan the QR code with your phone.
4. Confirm the starter app appears and runs on your phone.
5. Change a small piece of visible text, save, and watch it update live on your phone.



What to ask next : Once your app is running, it's worth understanding what actually happened:

```
In plain language, explain what just happened wher
```

Check the result

- ☐ Does the starter app run successfully on your own phone?
- ☐ Did a small text change you made appear on your phone without restarting anything?
- ☐ Can you explain, in your own words, what Expo Go is doing?
- ☐ Are your phone and computer connected to the same Wi-Fi network?



Learn more online : Expo's requirements, exact commands, and version numbers can change. See https://docs.expo.dev for current setup instructions and troubleshooting.

You now have a real Expo project running live on your own phone hardware. The next chapter takes you on a tour of that project's files, so you understand what you're looking at before you start changing it in earnest.

Chapter 3.3 Understand the Mobile Project

In this chapter, you will take a guided tour of the Expo project Claude Code created, so you recognize what you're looking at before you start changing it. You will follow one import from a screen to a shared piece it depends on, and ask Claude Code for a plain-language map of the whole project.

You need the running Expo project from *Chapter 3.2* . By the end of this chapter, you should be able to look at the project's file tree and explain, roughly, what each part is for.

Tour routes, components, assets, configuration, and packages

An Expo Router project organizes itself around a few recognizable kinds of files and folders:

- **Routes** : Files inside an app folder, where each file's location determines a screen and its address within the app, similar in spirit to how each HTML file in *Book 2* became a page.

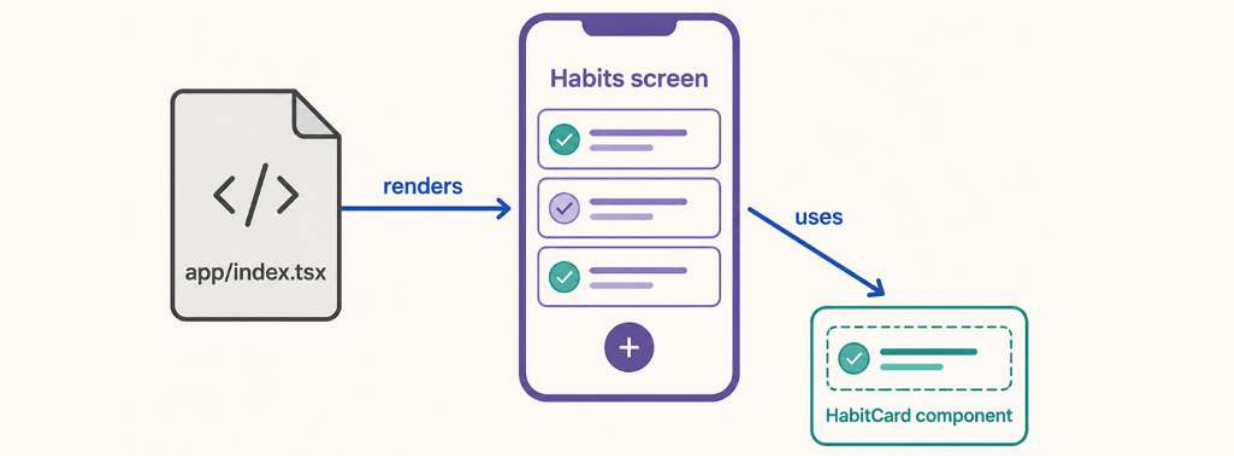


Figure 3.3A: A route file becomes a screen. A screen is often built from smaller, reusable components.

- **Components** : Reusable pieces of interface, often kept in their own folder, that screens are built from.
- **Assets** : Images, icons, and fonts the app uses, usually kept in an assets folder.
- **Configuration** : Files such as `app.json` , which describe the app's name, icon, and other settings you'll return to in *Chapter 3.12* .
- **Packages** : The same idea from *Chapter 1.5* , listed in `package.json` , providing pieces of functionality you didn't write yourself.

Ask Claude Code to point out one real example of each kind of file and folder in your actual project.

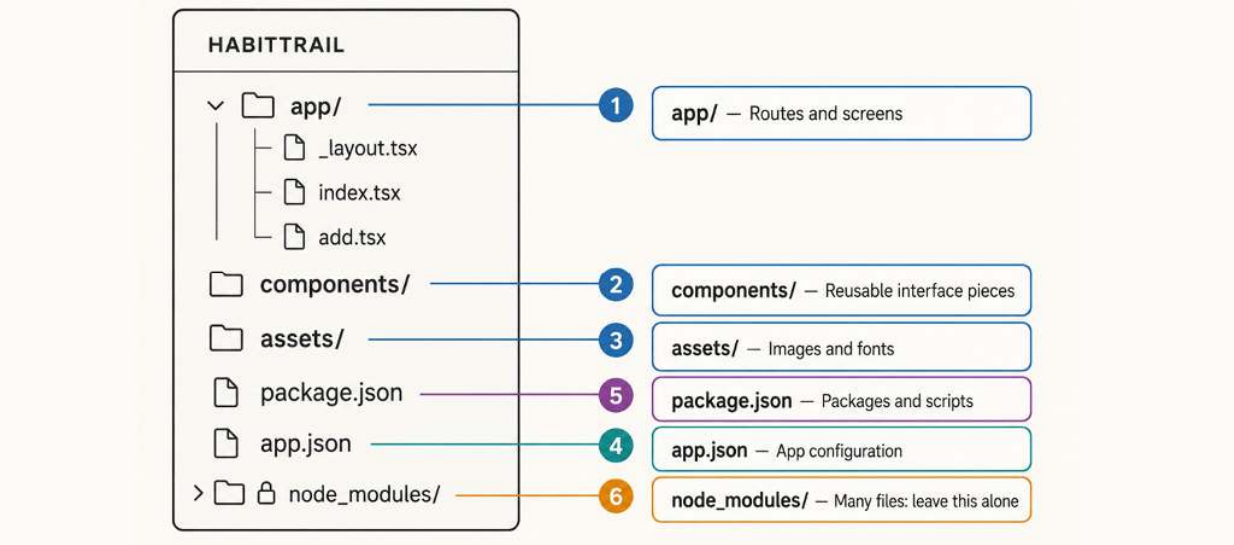


Figure 3.3B: Routes, components, assets, and configuration are yours to work in. node_modules is machine-managed. You'll never need to open it.



Watch out : An Expo project includes a large `node_modules` folder, holding every package's actual code. You will never need to open or edit anything inside it directly. Recognizing it on sight, and knowing to leave it alone, saves confusion later.

Follow an import

An import is a line of code that brings something defined in one file into another, such as a screen using a shared button component. Following one import by hand builds a concrete sense of how the pieces connect:

```
Show me one import statement in the starter project's home screen, and
```

Trace this one connection until it makes sense, rather than trying to understand every file in the project at once.

Ask Claude for a project map

Once you've looked at a few pieces individually, ask for a wider view:

```
Give me a plain-language map of this whole project: what the main folc
```

Keep this explanation somewhere you can find it again. Referring back to a map like this is often faster than re-deriving your understanding of the project's shape every time you return to it.

Folder	What it's for
`app/`	Your screens. This is the whole navigation structure. Every file here becomes a route/URL automatically. `_layout.tsx` files define the shell around a group of screens (stack, tabs). Parens like `(tabs)` are a grouping that doesn't show up in the URL.
`components/`	Reusable UI pieces that are not screens, like buttons, text wrappers, a collapsible panel, and an animated wave. Screens assemble these. `components/ui/` holds the lowest-level primitives, some with platform-specific versions (`icon-symbol.ios.tsx` vs `icon-symbol.tsx`).

`hooks/`	Reusable behavior (not visuals) like `use-color-scheme`, `use-theme-color`. Functions starting with `use-` that components call.
`constants/`	Fixed values shared everywhere. Here it's just `theme.ts` (the light/dark color palettes).
`assets/`	Static files shipped with the app. Currently just images (icons, splash, logos). Referenced from code with `require(...)` and from `app.json` by path.
`scripts/`	Dev tooling. `reset-project.js` moves the starter `app/` aside so you can start clean (npm `run reset-project`).
Root config files	`app.json` (Expo app config: name, icons, plugins), `package.json` (dependencies/scripts), `tsconfig.json` (TypeScript + the `@/` project-root alias), `eslint.config.js`, `CLAUDE.md`, `AGENTS.md`, `README.md` (docs).
Hidden / generated	`node_modules/` (installed packages), `.expo/` (dev-server cache), `.git/`, `expo-env.d.ts` (auto-generated types). Don't edit these.

Table 3.3A: Plain-language map of habittrail-app



Good practice : Ask what a file or folder is for the first time you notice it, rather than skipping past it and hoping it becomes clear later. A project's structure gets

easier to navigate the more of its pieces you can already name.



Real-world example : A reader new to mobile development felt overwhelmed the first time he opened a real Expo project's file tree, expecting to need to understand every file before touching anything. Following a single import, from a screen into the button component it used, gave him one concrete thread he could actually follow start to finish, which made the rest of the project feel far less intimidating.

Try it now

Tour the project Claude Code created and connect its structure back to your app plan:

1. Ask Claude Code to point out one real example of a route, a component, an asset, a configuration file, and a package in your project.
2. Follow one import in the starter home screen back to the file it points to.
3. Ask Claude Code for a plain-language map of the whole project.
4. Ask where your three planned screens would likely go, based on the current structure.



What to ask next : Once the structure feels familiar, connect it back to your plan from *Chapter 3.1* :



```
Based on this project's current structure, where would
```

Check the result

- ☐ Can you point to a real route, component, asset, configuration file, and package in your project?
- ☐ Did you follow one import from a screen to the file it depends on?
- ☐ Do you have a plain-language map of the project you can refer back to?
- ☐ Do you know roughly where your three planned screens will go?

You can now recognize the main pieces of a real Expo project and trace at least one connection between them. The next chapter starts building for real: turning native interface components into the reusable cards and buttons your three screens will be made of.

Chapter 3.4 Build Screens from Components and Styles

In this chapter, you will build the first real pieces of HabitTrail's interface: a reusable card for showing a habit, and a button styled for touch. You will learn about the native interface components React Native provides, and how to style them so they work comfortably on a real phone screen, not just look right in theory.

You need the Expo project from *Chapter 3.2* , with the structure you toured in *Chapter 3.3* . By the end of this chapter, you'll have a HabitCard component you can reuse across every screen that needs to show a habit.

Use native interface components

Where *Book 2* 's website used HTML elements such as `<div>` and `<p>` , a React Native app uses native interface components: View for a general container, Text for any text, Pressable for anything a user taps, and others for more specific needs. These render using each phone's own real interface pieces, rather than simulating them:

```
Explain, in plain language, the difference between View, Text, and Pre
```

Recognizing these three alone covers most of what HabitTrail's screens will need:

I want to	Use

Arrange or visually group things, add spacing or a background	`View`
Show any text or numbers	`Text`
React when the user taps	`Pressable`

Table 3.4A: Quick decision guide for React Native components

For HabitTrail, the screen background and each habit card are Views, the habit name and streak count are Texts, and tapping a card to toggle today's completion is a Pressable.

Create reusable cards and buttons

You will now build a custom `HabitCard` component to use in your mobile app.

A `HabitCard` will show one habit's name and whether it's done today, and get reused on both the Habits list and, in a different form, the Habit Detail screen. Building it as its own reusable component now saves rebuilding the same layout repeatedly later.

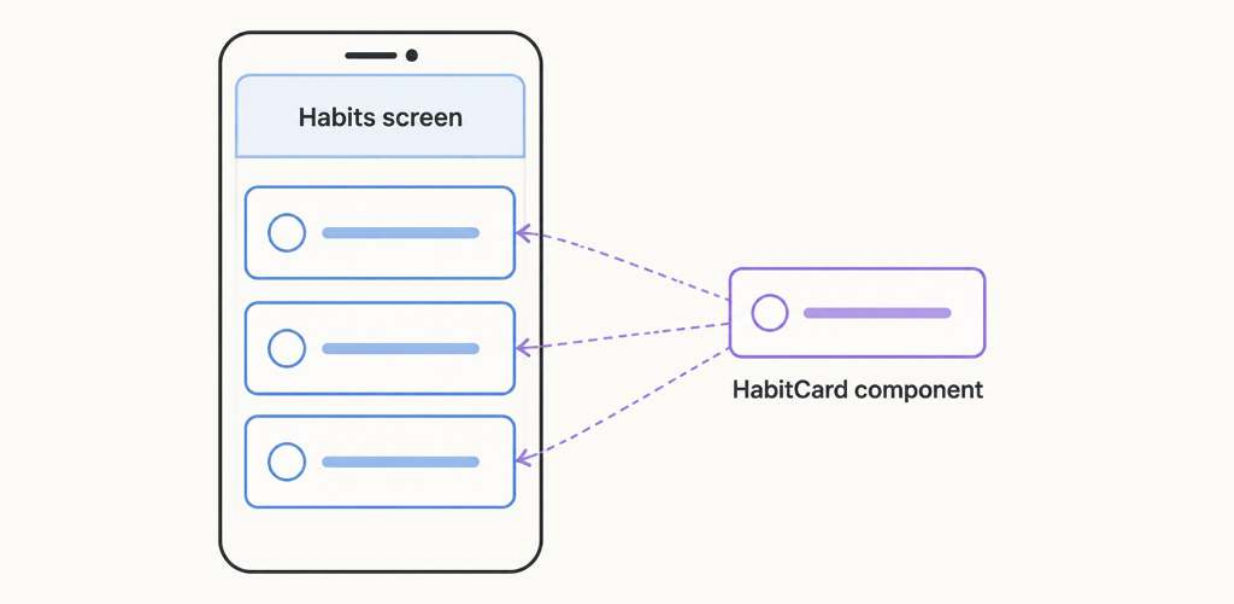


Figure 3.4A: One HabitCard component, reused three times, builds the whole list. Change the component once, and every card updates.

Ask Claude to build a `HabitCard`:

```
Create a HabitCard component that shows a habit's name and a checkbox-
```

Claude should respond with detailed output about its work, and a summary:

```
Created components/habit-card.tsx.
```

Note that `habit-card.tsx` is created in the `components/` folder:

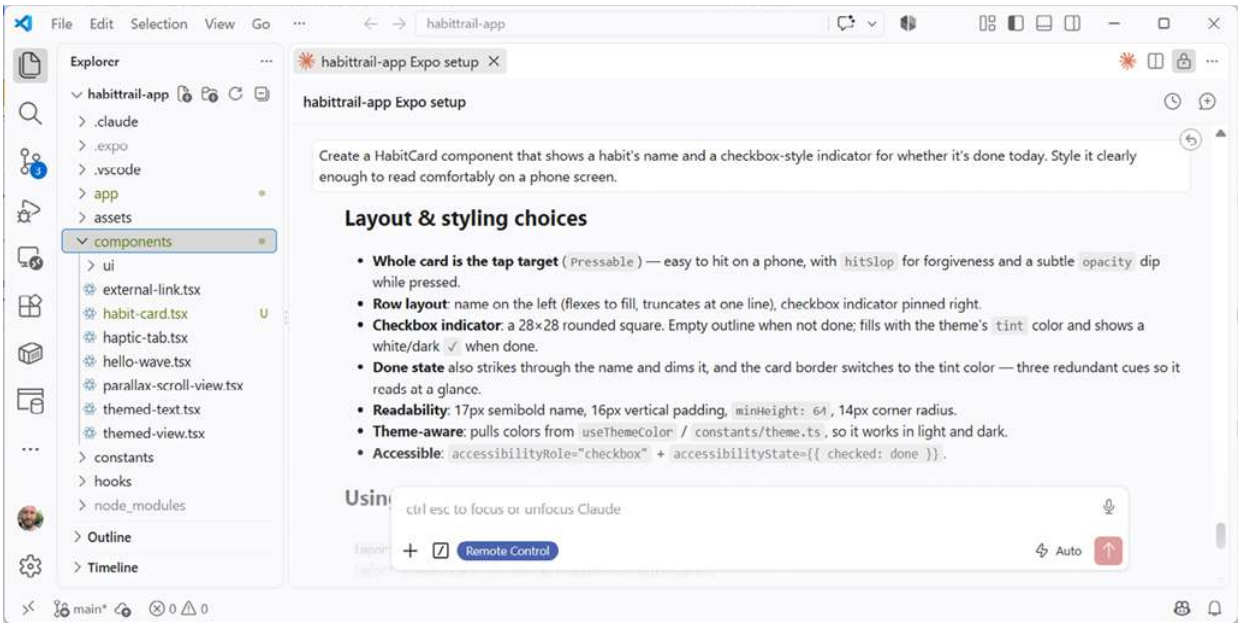


Figure 3.4B: HabitCard component layout and styling choices Claude made



Good practice : Build one component fully, test it on your phone, and confirm it looks and feels right before building the next one. A `HabitCard` that's slightly too small to tap comfortably is a quick fix now, and a repeated annoyance across every screen if it goes unnoticed.

Ask for a matching button style at the same time, since HabitTrail needs a consistent style for buttons like "Add Habit" and "Mark Done" throughout the app:

```
Now create a matching button style I can reuse for buttons like "Add H
```

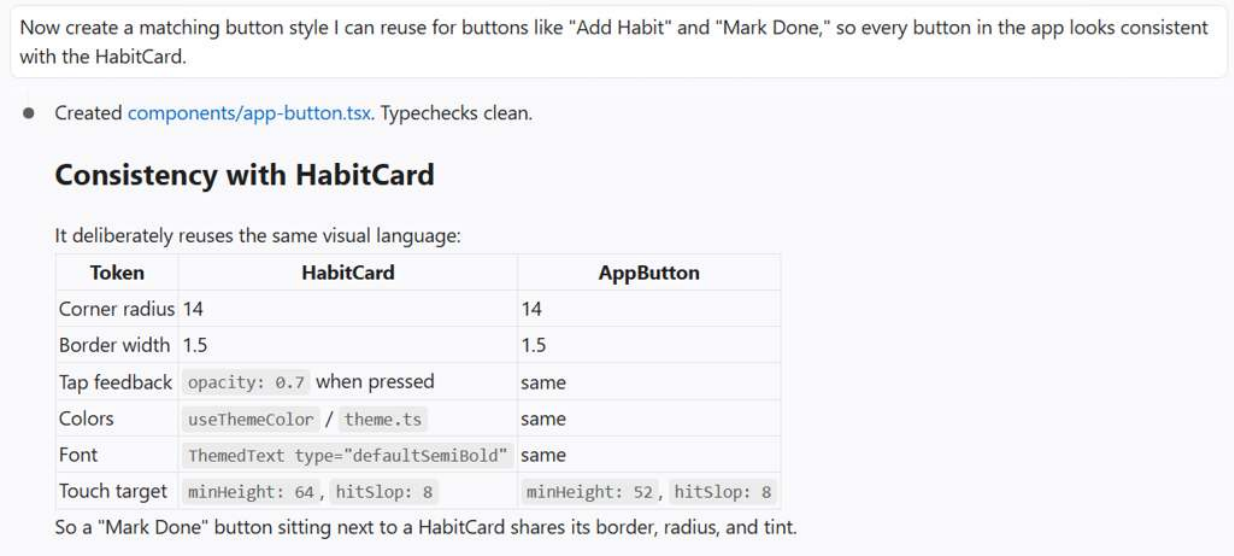


Figure 3.4C: AppButton component consistency with HabitCard

Style for touch and safe areas

A phone screen isn't just smaller than a laptop screen. It's touched with a finger, not clicked with a precise cursor, and parts of it, such as the very top and bottom, can be covered by a notch, a camera, or a home-gesture bar depending on the device:

```
Check that every tappable element, especially the HabitCard and buttor
```

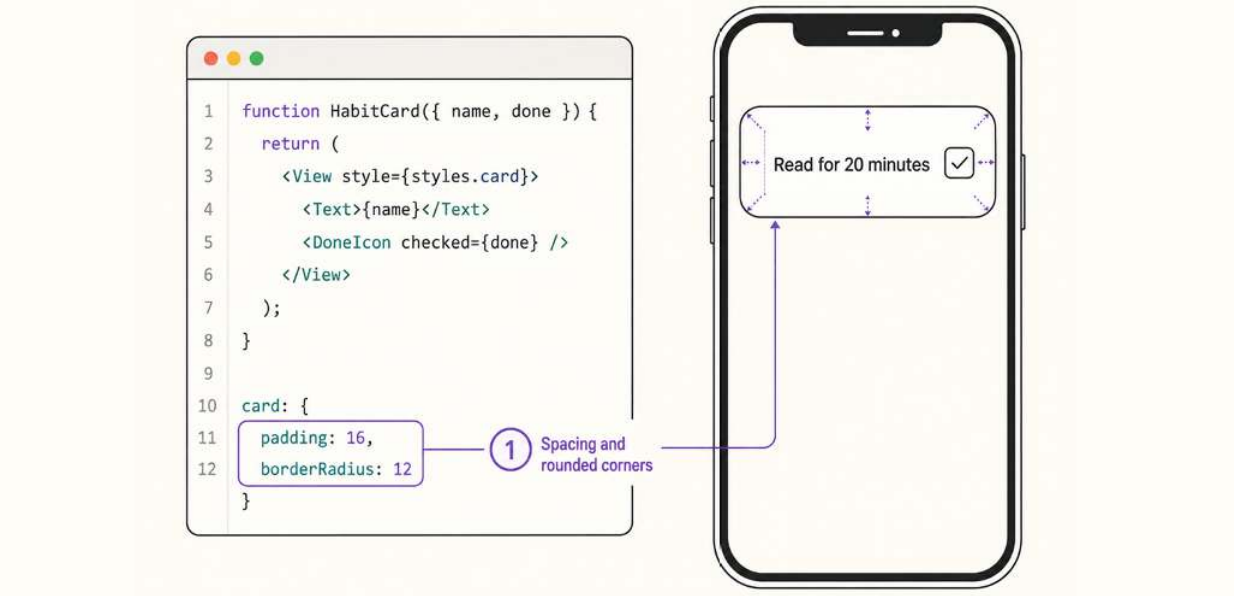


Figure 3.4D: The same styling ideas from Book 2 – padding, spacing, rounded corners – apply here too, now producing something you tap instead of click

Since your app's real screens don't exist yet, ask Claude Code to temporarily show the HabitCard and button with example data on your starter screen, just so you have something to test on your phone:

```
Temporarily add an example HabitCard and button, with made-up sample d
```

You'll replace this temporary preview with real screens in the next chapter, once Habits, Add Habit, and Habit Detail exist as actual routes.

Test this on your actual phone from *Chapter 3.2* , not just by looking at the code. What counts as comfortable to tap is something you can genuinely feel, not just estimate. Ask Claude Code to start your project's development server:

```
Start the development server for this project.
```

Or start it manually so you can see and scan the QR code:

```
npx expo start
```

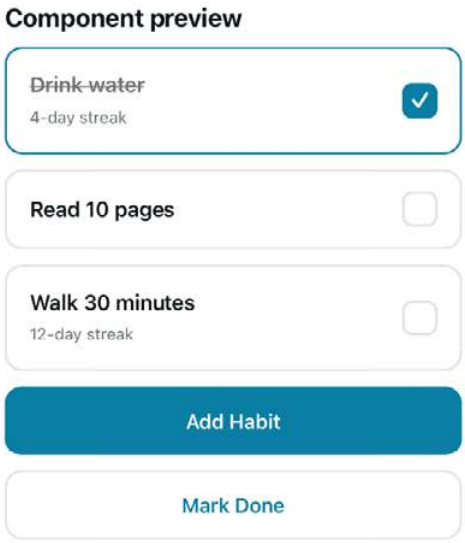


Figure 3.4E: Three demo HabitCards and two buttons on the Home screen in the mobile app on iPhone



Watch out : A layout that looks fine in a screenshot can still place something important under a phone's notch or gesture bar. Always check safe areas on an actual device, since this is easy to miss when only looking at code.



Real-world example : A reader's first version of his `HabitCard` used a small checkbox that looked reasonable on his computer screen but was awkward to tap accurately on his phone. Asking Claude Code to check tap target sizes against standard touch guidelines, then

testing on his own phone, led to a slightly larger, easier target that felt obviously better the moment he tried it.

Try it now

Build HabitTrail's first reusable pieces, a HabitCard and a matching button, and test both on your phone:

1. Ask Claude Code to explain View, Text, and Pressable in plain language.
2. Build a `HabitCard` component showing a habit's name and a done-today indicator.
3. Build a matching, consistently styled button.
4. Check tap target sizes and safe areas, and test both on your actual phone.



What to ask next : Once your `HabitCard` and button styles feel right, it's worth confirming they'll hold up as content varies:

```
Show me how the HabitCard looks with a very short
```

Check the result

☐ Can you explain, in your own words, what View, Text, and Pressable are each for?

☐ Does your HabitCard look clear and readable on your actual phone screen?

- ☐ Do all tappable elements feel comfortable to tap with a finger, not just technically clickable?
- ☐ Does the layout avoid sitting under your phone's notch or gesture bar?

You now have a reusable `HabitCard` and button, styled and tested on a real phone. The next chapter puts them to use across real screens, using Expo Router to move between the Habits list, Add Habit, and Habit Detail screens from your plan.

Chapter 3.5 Move Between Screens with Expo Router

In this chapter, you will build all three of HabitTrail's screens as real, connected routes, add both tab and stack navigation, and let a user tap a habit on the list to open its own detail screen safely.

You need the `HabitCard` and button from *Chapter 3.4* . By the end of this chapter, you should be able to move between all three screens on your own phone.

Add the app's BRIEF.md to the project

In *Chapter 3.1* , you wrote a project brief. Now save it, or the sample file below, as `BRIEF.md` in the `habittrail-app` project to guide Claude Code.



Sample files : A complete example mobile app project brief is available in the GitHub repository: https://github.com/markjprice/claude-code-vb/blob/main/files/book-3/3-1-mobile-app-brief.md

I recommend renaming the file to `BRIEF.md` when you add it to the `habittrail-app` project.

Create routes

Expo Router turns files inside the app folder into screens, using each file's location to determine its address within the app, the same file-based idea you saw briefly in *Chapter 3.3* :

```
Create three routes matching my app plan in BRIEF.md: a Habits screen
```

Confirm each new route's file exists and loads without errors in the terminal. You won't be able to reach these screens on your phone yet, since navigation isn't wired up until the next section, but that's expected, not a sign something's broken.

Add tab and stack navigation

Tab navigation shows a small set of top-level destinations, usually at the bottom of the screen, that a user can switch between directly. Stack navigation pushes a new screen on top of the current one, such as opening a specific habit's detail, with a way to go back to where you came from:

```
Set up tab navigation with Habits and Add Habit as the two main tabs.
```

Test this on your phone: switch between the two tabs, then tap a habit from the list and confirm you can navigate back to the list afterward.

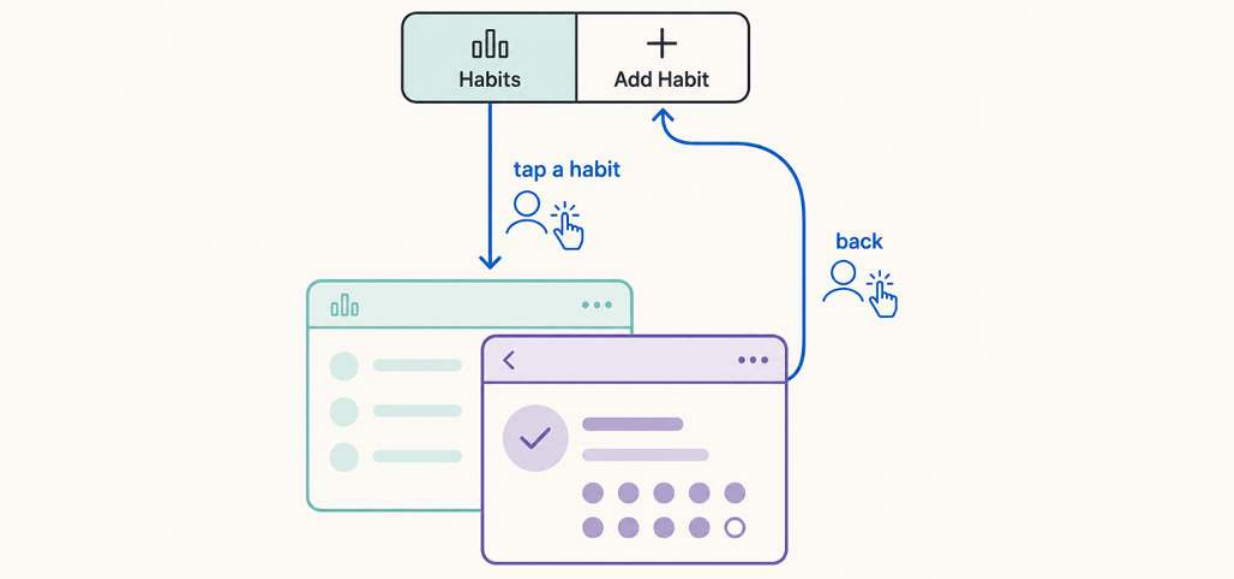
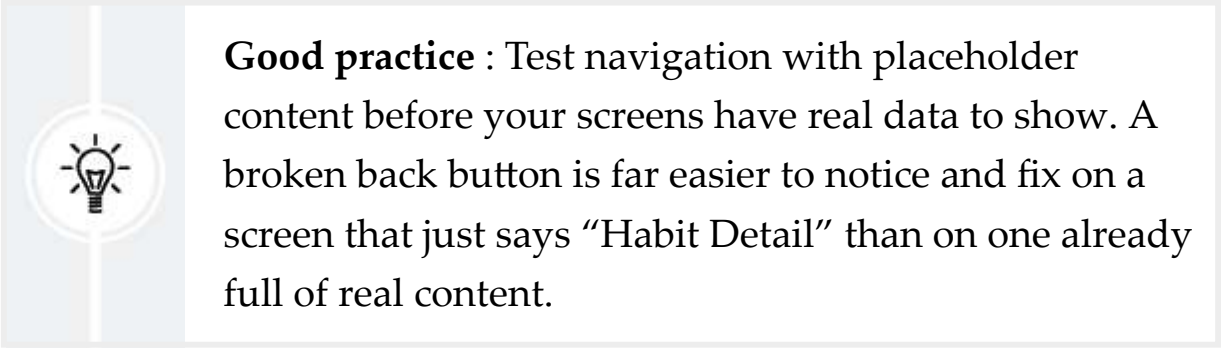


Figure 3.5A: Two tabs for top-level destinations, and one screen that stacks on top when you tap into a specific habit



Good practice : Test navigation with placeholder content before your screens have real data to show. A broken back button is far easier to notice and fix on a screen that just says “Habit Detail” than on one already full of real content.

Pass a safe identifier to a detail screen

When a user taps a habit on the list, the Habit Detail screen needs to know which habit to show. This is usually done by passing that habit’s identifier as part of the route, then using it to look up the right habit’s data:

```
When a habit is tapped on the Habits screen, navigate to its detail sc
```

That last part is important: a detail screen that assumes the identifier it received is always valid can crash or show confusing results if something unexpected happens. Handling a missing or invalid identifier gracefully is a small habit worth building now, before it causes a real problem later.

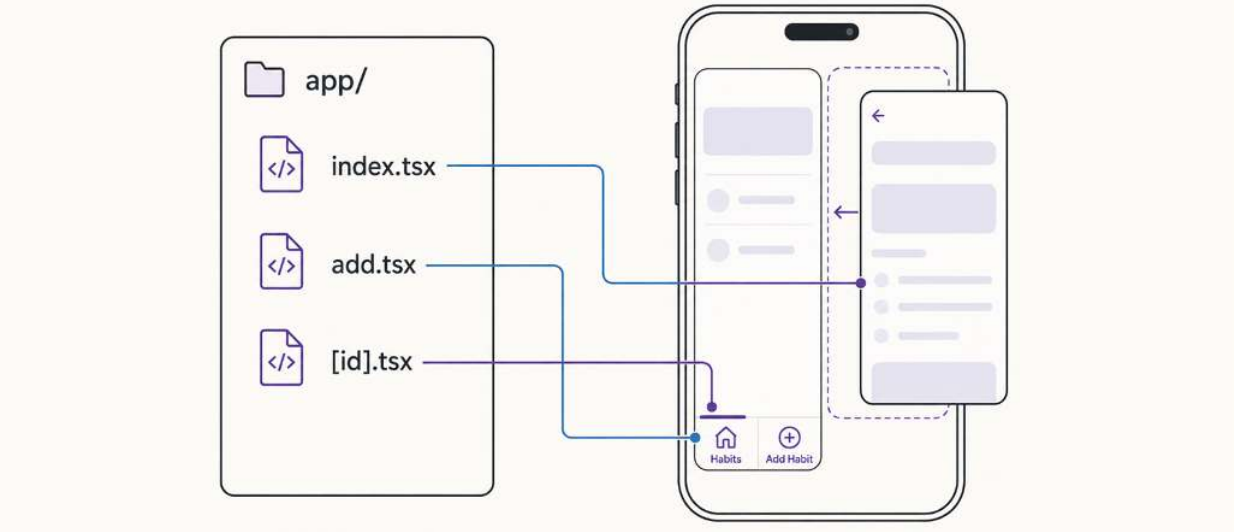


Figure 3.5B: Each route file becomes part of the navigation a user actually taps through



Watch out : A detail screen that trusts its identifier without checking it can fail badly if that identifier is ever wrong, missing, or unexpected. Confirm Claude Code has added a check, and see for yourself what the screen does if you deliberately give it a nonsense identifier.



Real-world example : A reader testing his own detail screen tried navigating to it directly with a made-up identifier that didn't match any real habit, out of curiosity. Without a check in place, the app showed a

blank, confusing screen instead of a clear message. Adding a simple check and a friendly fallback message turned a confusing dead end into a graceful result.

Try it now

Turn your three planned screens into real, connected routes a user can navigate between:

1. Create the three routes matching your plan: Habits, Add Habit, and Habit Detail.
2. Set up tab navigation between **Habits** and **Add Habit** .
3. Set up stack navigation so tapping a habit opens its detail screen, with a working back button.
4. Test navigating to a habit's detail screen and back, and deliberately test an invalid identifier.



What to ask next : Once navigation works, it's worth confirming the back button behaves exactly as expected:

```
Confirm that tapping back from the Habit Detail sc
```

Check the result

☐ Do all three screens exist and load, even with placeholder content?

☐ Can you switch between the Habits and Add Habit tabs?

- ☐ Does tapping a habit open its detail screen, with a working back button?
- ☐ Does the app handle an invalid or missing habit identifier gracefully, instead of crashing or showing something confusing?

HabitTrail's three screens are now real and connected, with navigation that behaves safely even when something unexpected happens. The next chapter makes those screens actually respond to what a user does, using props and state to update what's on screen after a tap.

Chapter 3.6 Make the App Respond with Props and State

In this chapter, you will make HabitTrail's screens actually respond when a user taps something, using two ideas at the core of how React Native apps work: props and state. You will also learn to calculate information from what's already stored, instead of tracking it separately and risking it going out of sync.

You need the connected screens from *Chapter 3.5* . By the end of this chapter, tapping a habit's done indicator should visibly update what's on screen.

Pass data into a component

A **prop** is a piece of data passed into a component from whatever's using it, similar to how you'd hand someone a specific document rather than expecting them to already have it. Your HabitCard from *Chapter 3.4* needs props for the habit's name and whether it's done today, so the same component can show any habit, not just one hardcoded example:

```
Confirm HabitCard receives its habit name and done status as props, ra
```

This is what makes one component reusable across every habit in the list, as you saw in *Chapter 3.4* 's diagram.

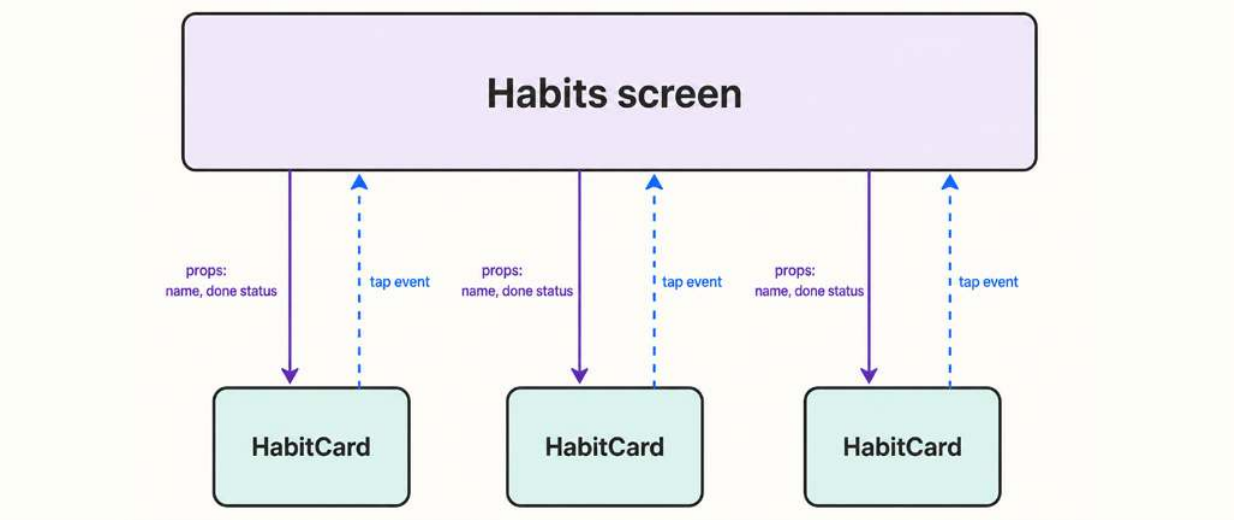


Figure 3.6A: Data flows down into a component as props. A user's action flows back up as an event, so the screen above can respond.

Update state after a tap

State is data a component keeps track of that can change over time, such as whether a specific habit is currently marked done. When a user taps a HabitCard's done indicator, that tap needs to update the relevant habit's state, which then causes the screen to redraw with the new value shown:

```
When a user taps a HabitCard's done indicator, update that habit's dor
```

Test this directly: tap a habit's indicator and confirm it visibly changes, without needing to reload or navigate away and back.

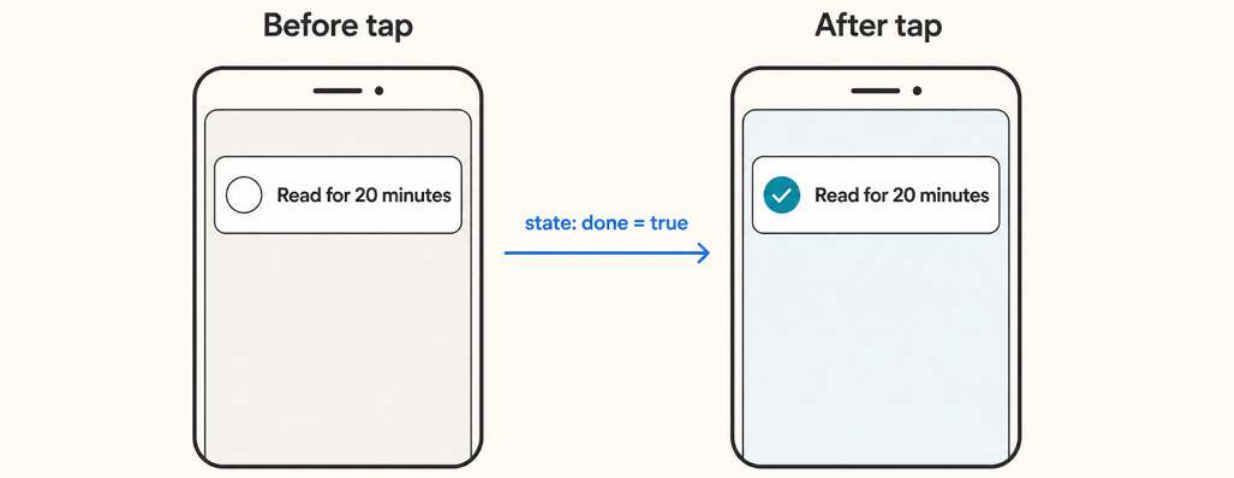


Figure 3.6B: One tap, one state change, one visible update. The whole interaction happens in that single step.

Derive progress instead of duplicating data

It's tempting to store a separate "streak count" value that you update every time a habit is marked done. This risks the stored number quietly drifting out of sync with the actual data over time, especially once you add editing and deleting in *Chapter 3.8* .

Instead, derive the streak by calculating it fresh from the actual record of completed days, every time it's needed, rather than storing it as a separate number that could become wrong.

```
Instead of storing a separate streak count, calculate a habit's currer
```

This is a small decision now that prevents a much harder bug to track down later: a displayed number that no longer matches reality.



Good practice : Ask, for any number your app displays, whether it could instead be calculated from data you're already storing. A derived value can't drift out of sync, since it's recalculated fresh every time.



Real-world example : A reader's early version of HabitTrail stored a streak count separately, updating it each time a habit was marked done. After adding the ability to edit a habit's completed days directly, he noticed the displayed streak sometimes disagreed with the actual record underneath. Switching to calculating the streak fresh from the stored days, every time, made the number correct by construction instead of something he had to remember to keep updated.

Share state across screens

So far, every piece of state you've built, a habit's done-today status, a derived streak, has lived inside a single screen. But HabitTrail's three screens all need to see the same underlying list of habits. A habit created on the **Add Habit** screen has to appear on the **Habits** list. A habit marked done on the **Habits** screen has to show correctly if you tap into its **Habit Detail** screen. None of that works if each screen keeps its own private copy of the list.

React Context is a way to make one piece of data, such as your list of habits, available to any screen or component that needs it, without manually passing it down through every layer in between. A

Provider is the component that actually holds that data and makes it available. Think of it as a shared shelf that any screen in the app can reach, rather than each screen keeping its own separate copy in a drawer:

```
Set up a shared way for the Habits, Add Habit, and Habit Detail screer
```

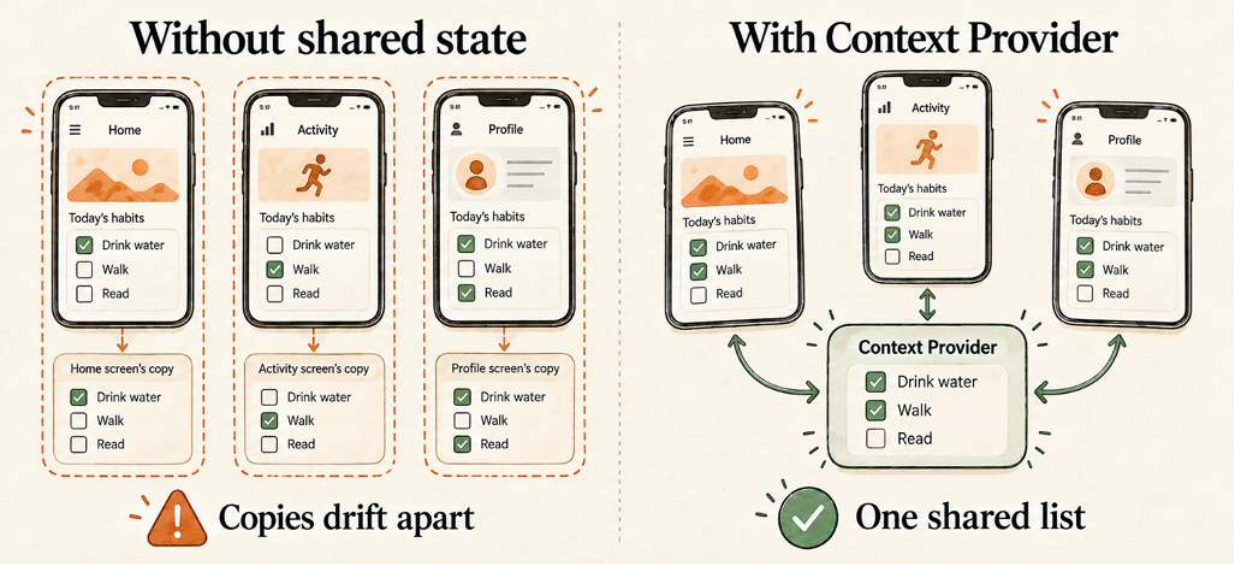


Figure 3.6C: Without shared state, each screen holds its own copy of the habit list, and they drift out of sync. With a Context Provider, every screen reads from and writes to the same shared list.

Ask Claude Code to walk you through what it built before moving on, the same way you'd ask about any new term:

```
In plain language, explain what you just created: what a Provider is,
```

Confirm this makes sense before continuing. You'll rely on this shared list for the rest of the book, starting with the Add Habit form in the next chapter.



Good practice : When Claude Code introduces a new architectural piece like a Provider, treat it the same way you'd treat any unfamiliar term: ask for a plain-language explanation before moving on, not just an assurance that it works. Understanding why your habits need a shared home now makes every later chapter that touches this data easier to follow.

Try it now

Wire up props, shared state, and a derived streak so HabitTrail's screens actually respond to a tap:

1. Confirm HabitCard receives its data through props, not hardcoded values.
2. Add state that tracks whether each habit is done today.
3. Set up shared state (React Context) so the Habits, Add Habit, and Habit Detail screens all read and update the same list of habits.
4. Connect a tap on the done indicator to updating that state, and confirm the screen updates visibly.
5. Replace any separately stored streak count with a value calculated fresh from stored data.



What to ask next : Once tapping a habit updates the screen correctly, check the calculation logic directly:

```
Walk me through exactly how a habit's current streak g
```

Check the result

- ☐ Does tapping a habit's done indicator visibly update the screen immediately?
- ☐ Can you explain, in your own words, the difference between a prop and a piece of state?
- ☐ Is a habit's streak calculated from its actual stored data, rather than tracked as a separate number?
- ☐ Can you explain, in your own words, what a Provider is and why HabitTrail needs one?
- ☐ Does `HabitCard` work correctly for more than one different habit, using the same component?

HabitTrail's screens now respond to what a user actually does, and calculate what they show from real, current data instead of numbers that could quietly go stale. The next chapter builds the Add Habit form, so users can create their own habits instead of only interacting with placeholder ones.

Chapter 3.7 Capture and Validate Form Input

In this chapter, you will build the Add Habit form: a real screen where a user types a habit's name, chooses a few simple options, and sets an optional daily reminder time. You will manage its input and prevent it from creating an incomplete or invalid habit.

You need the Add Habit route from *Chapter 3.5* , and the shared habit state from *Chapter 3.6* . By the end of this chapter, a user should be able to create a real habit that appears correctly on the Habits list.

Build the add-habit form

Using your plan from *Chapter 3.1* , the Add Habit form needs, at minimum, a text field for the habit's name, and a way to optionally set a daily reminder time, which you'll connect to real notifications in *Chapter 3.10* :

```
Build the Add Habit screen with a text input for the habit name, and a
```

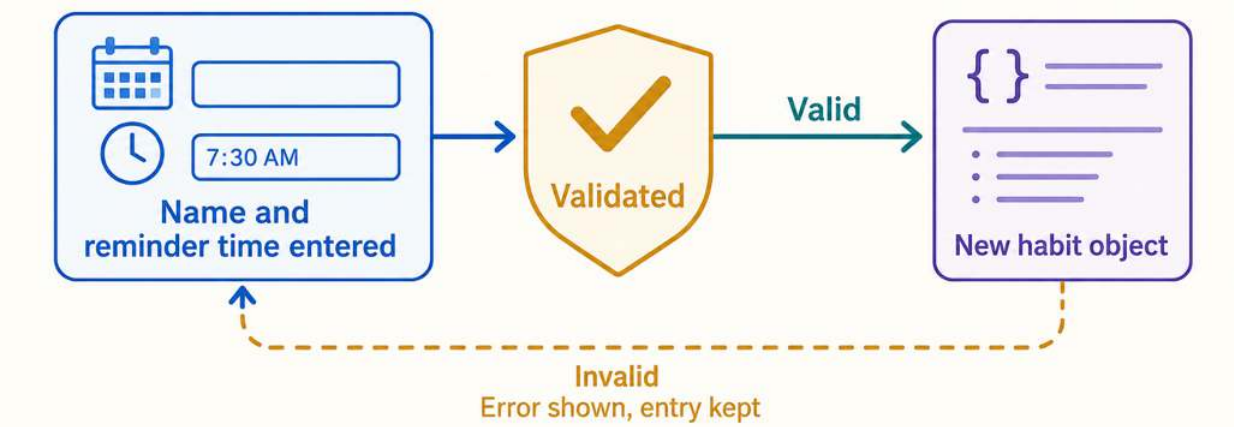


Figure 3.7A: A habit only gets created once its entry passes validation. An invalid entry sends the user back to fix it, not back to an empty form.

Manage text, choices, and reminders

A text input needs to reflect what the user is currently typing, updating as they type rather than only once they finish. A time picker, similarly, needs to reflect the currently chosen time, and offer a clear way to leave the reminder unset if a user doesn't want one for this habit.

Ask Claude Code to update the name as the user types and leave the reminder unset:

```
Confirm the name field updates as I type, and the reminder time picker
```

Test this directly: type a habit name and confirm it appears in the field as you type, then try setting a reminder time and clearing it again.

Prevent incomplete or invalid records

Before a new habit gets added to the list, it needs to pass a few basic checks, the same validation idea from *Book 2*'s contact form, now applied to a mobile form:

```
Prevent the form from creating a habit if the name field is empty. Shc
```

Test this by trying to submit the form with an empty name, and confirming you get a clear, specific message rather than a silent failure or a confusing error.



Figure 3.7B: The same form pattern from Book 2: empty

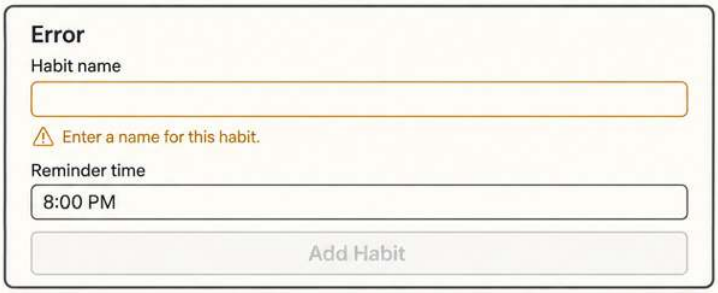


Figure 3.7C: The same form pattern from Book 2: a specific and recoverable error

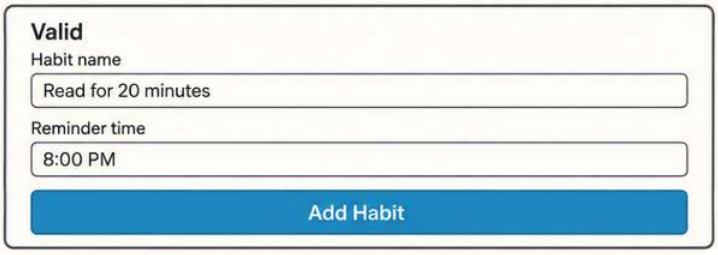


Figure 3.7D: The same form pattern from Book 2: a clearly valid, ready-to-submit state



Good practice : Test your form by trying to break it on purpose: submit it empty, submit it with only a reminder time set and no name, and submit it correctly. Each of these should behave sensibly, not just the case where everything is filled in correctly. A form that silently does nothing when submission fails is more frustrating than one that shows a clear error. If tapping a disabled or failing submit button gives no feedback at all, a user has no way to know what to fix.



Real-world example : A reader's first version of the Add Habit form let an empty name through, creating a habit that showed up on the list as a blank card with nothing to tap or identify it. Adding a simple check, requiring a name before submission, and a clear inline message when it's missing, prevented a confusing, hard-to-remove blank entry from ever being created.

Try it now

Build the Add Habit form and confirm it rejects an incomplete entry instead of silently accepting one:

1. Build the Add Habit form with a name field and an optional daily reminder time.
2. Confirm the name field updates as you type, and the time picker works and can be cleared.

3. Add validation preventing submission with an empty name, showing a clear message.
4. Test submitting empty, partially filled, and correctly filled forms.



What to ask next : Once your form validates correctly, it's worth confirming what a completed submission actually produces:

```
Walk me through exactly what happens when I submit a v
```

Check the result

- ☐ Does the name field update correctly as you type?
- ☐ Can you set and clear an optional reminder time?
- ☐ Does submitting with an empty name show a clear, specific message instead of failing silently?
- ☐ Does a correctly filled form create a real habit you can see reflected elsewhere in the app?

You now have a working Add Habit form that prevents incomplete records instead of silently accepting them. The next chapter connects everything so far into a complete set of habit operations: creating, reading, updating, and deleting a habit from the list.

Chapter 3.8 Create, Read, Update, and Delete Habits

In this chapter, you will complete the full set of operations HabitTrail needs for managing habits: rendering the list, marking habits done and filtering them, and editing or deleting one from its detail screen. Together, these four operations are often called CRUD: create, read, update, and delete.



Figure 3.8A: Create, read, update, and delete: the four operations any real list of data needs. Delete gets an extra step, since it can't be easily undone.

You need the working Add Habit form from *Chapter 3.7* and the connected screens from *Chapter 3.5* . By the end of this chapter, a user will be able to fully manage their list of habits, not just add to it.

Render a list

You've already built the Habits screen to show HabitCards, but confirm it now handles a real, changing list correctly: showing every habit that currently exists, in a sensible order, and updating immediately when a new one is added from *Chapter 3.7* 's form:

```
Confirm the Habits screen renders every current habit correctly, in th
```

Complete and filter habits

Marking a habit done for today is the interaction you already built in *Chapter 3.6* . Now, add a simple way to filter the list, such as showing only habits not yet done today, so a user can quickly see what's left:

```
Add a simple filter to the Habits screen: a way to show only habits nc
```

Edit and confirm deletion

On the Habit Detail screen, let a user change a habit's name or reminder time, reusing the same validation approach from *Chapter 3.7* . Deleting a habit is different: it's destructive, meaning it can't be easily undone, so it needs a clear confirmation step before it happens:

```
On the Habit Detail screen, let the user edit the habit's name and rem
```

Test deletion carefully: confirm the confirmation step actually appears, that canceling it leaves the habit untouched, and that confirming it removes the habit and returns to an updated list.

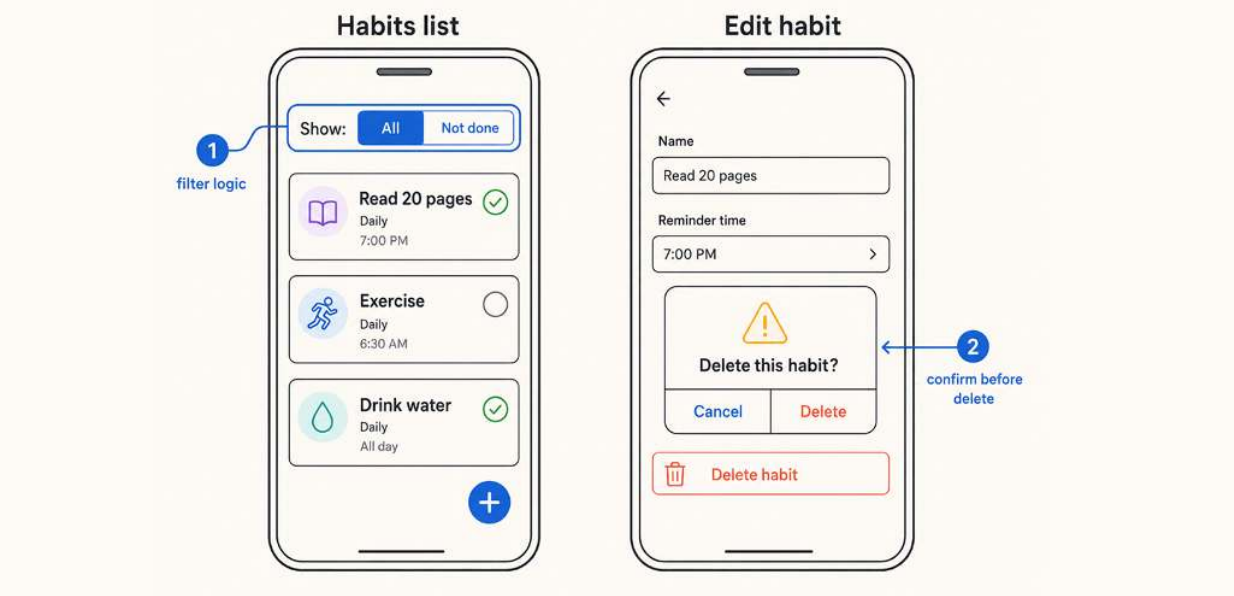


Figure 3.8B: A filterable list and a careful edit screen: the same four operations, now visible on real screens



Good practice : Test the delete confirmation by canceling it at least once before confirming it. A confirmation step only protects a user if canceling it genuinely leaves everything unchanged. A destructive action without a confirmation step is one accidental tap away from lost data. Deletion is exactly the kind of action worth deliberately slowing down, even in an app this small.



Real-world example : A reader testing his own app tapped a habit's delete option without thinking, out of habit from browsing his phone quickly, and was relieved to see a confirmation dialog rather than an immediately deleted habit he'd tracked for two weeks. That one small pause, which he'd almost skipped

adding, saved real data he would have been genuinely annoyed to lose.

Try it now

Complete HabitTrail's full set of habit operations, including a delete step a user has to confirm:

1. Confirm the Habits list renders correctly and updates when a habit is added.
2. Add a filter showing only habits not yet done today.
3. Add editing on the Habit Detail screen, using the same validation as the Add Habit form.
4. Add a delete option with a confirmation step, and test canceling it before confirming it.



What to ask next : Once all four operations work, it's worth checking they hold up together, not just individually:

```
Walk through a full cycle: add a habit, mark it done, 
```

Check the result

☐ Does the Habits list update immediately when a habit is added?

☐ Does the filter correctly show only incomplete habits when selected?

- ☐ Can you edit a habit's name and reminder time from its detail screen?
- ☐ Does deletion require confirmation, and does canceling it leave the habit untouched?

HabitTrail now supports the full set of operations a real app needs: creating, viewing, updating, and safely deleting a habit. The next chapter makes sure none of this disappears the moment the app closes, by saving it to the phone itself.

Chapter 3.9 Keep Data Between Sessions

In this chapter, you will make HabitTrail remember its habits after the app closes, using the phone's own storage. You will also handle the moments a real app has to plan for: the very first launch, the brief loading period, and data that somehow gets damaged.

You need the fully working CRUD screens from *Chapter 3.8* . By the end of this chapter, closing and reopening HabitTrail should show exactly the habits you left it with.

Understand local persistence

Right now, HabitTrail's habits exist only in the app's memory while it's running. Close or reload the app, and everything resets. Persistence means saving data somewhere that survives the app closing, so it's there again the next time it opens.

For an offline-first app like HabitTrail, that somewhere is the phone itself, not a server. Nothing about your habits needs to leave your device for this app to work.

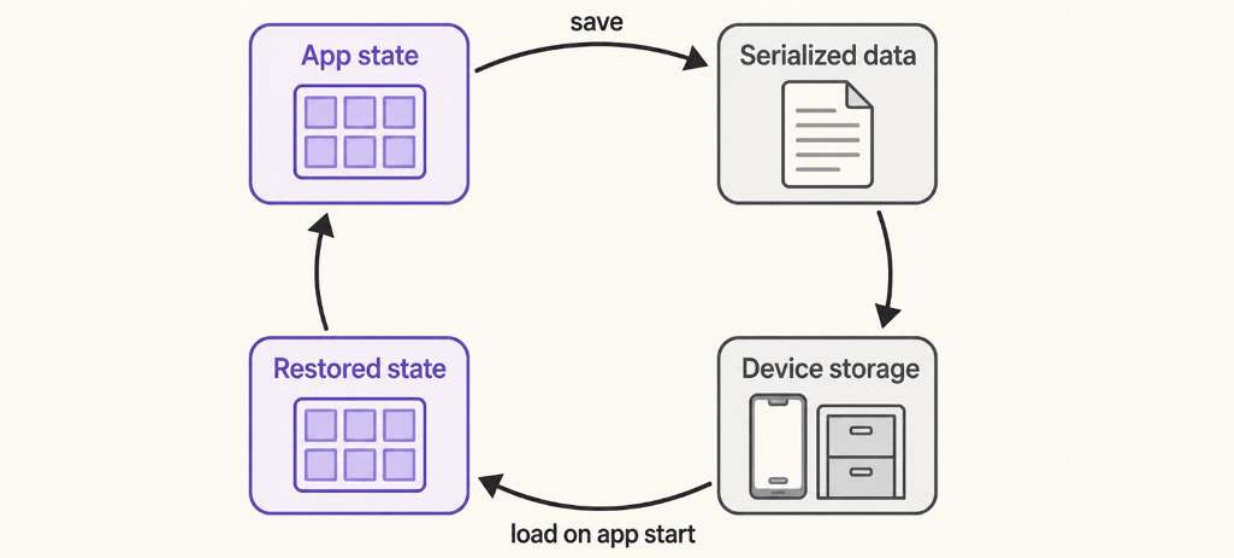


Figure 3.9A: Habits move from the app's working memory, to storage on the phone, and back again every time HabitTrail opens

Save and load with AsyncStorage

AsyncStorage is the standard way an Expo app saves simple data directly on the phone. Ask Claude Code to connect it to HabitTrail's habits:

```
Save the current list of habits to AsyncStorage whenever it changes, a
```

Test this directly: add a habit, force-close the app completely, then reopen it and confirm the habit is still there.

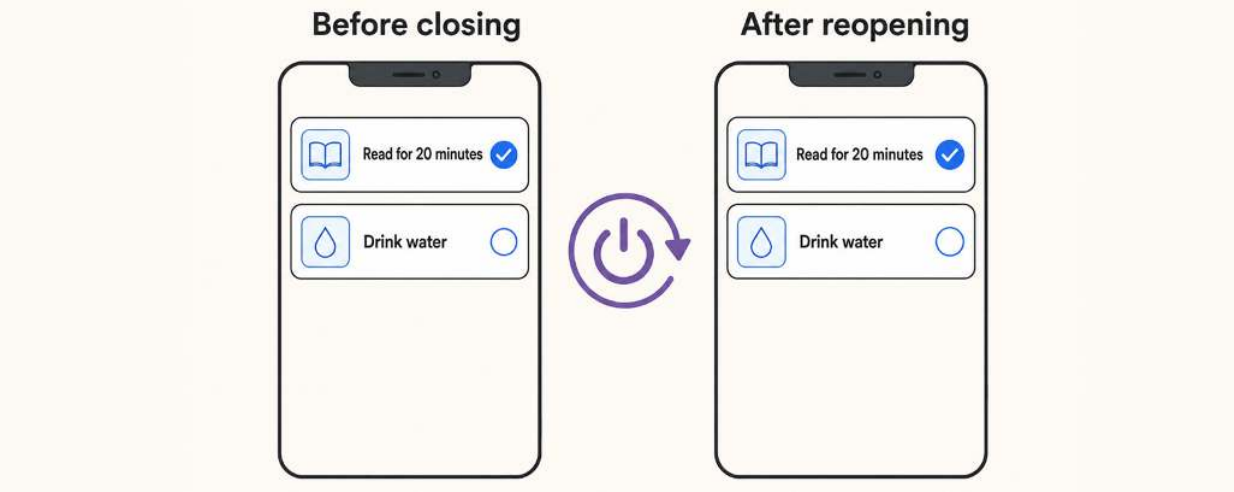


Figure 3.9B: Close the app completely, reopen it, and your habits should look exactly as you left them

AsyncStorage stores data unencrypted on the device. That's a reasonable fit for habit names and completion dates, information that isn't sensitive. It would not be appropriate for a password or anything genuinely private, a distinction worth remembering if you ever extend this app or build another one that handles more sensitive information.



Watch out : AsyncStorage's data is unencrypted. It's a fine choice for HabitTrail's habit names and completion dates, but never store a password, a private key, or anything else genuinely sensitive this way, in this app or any other.

Handle empty, loading, and corrupted data

A real app needs a sensible plan for at least three situations beyond the normal case of loading existing habits successfully:

- **Empty** : A brand-new user with no habits yet. The Habits screen should explain this clearly, not just show a blank space.
- **Loading** : A brief moment while stored data is being read. The screen shouldn't flash confusingly between an empty state and the real data.
- **Corrupted** : Stored data that, for whatever reason, can't be read correctly. The app should recover gracefully, rather than crashing.

Here's an example prompt:

```
Handle three situations on the Habits screen: no habits yet, a brief l
```



Good practice : Test persistence by actually force-closing the app, not just navigating away and back within it. Some problems with saving and loading only show up once the app has genuinely restarted.



Real-world example : A reader testing his app's launch was confused when the Habits screen briefly showed "No habits yet" before his actual habits appeared a moment later, since the loading state and the genuinely empty state looked identical. Adding a distinct, brief loading indicator resolved the confusion, so an empty state only ever meant a user genuinely had no habits yet.

Try it now

Connect AsyncStorage so HabitTrail's habits survive a full close and reopen, then test its edge cases:

1. Connect AsyncStorage so habits save automatically and load on app start.
2. Force-close the app after adding a habit, then reopen it and confirm the habit is still there.
3. Add a clear message for a brand-new user with no habits yet.
4. Add a brief loading state, distinct from the empty state.
5. Ask Claude Code to simulate corrupted stored data and confirm the app recovers gracefully.

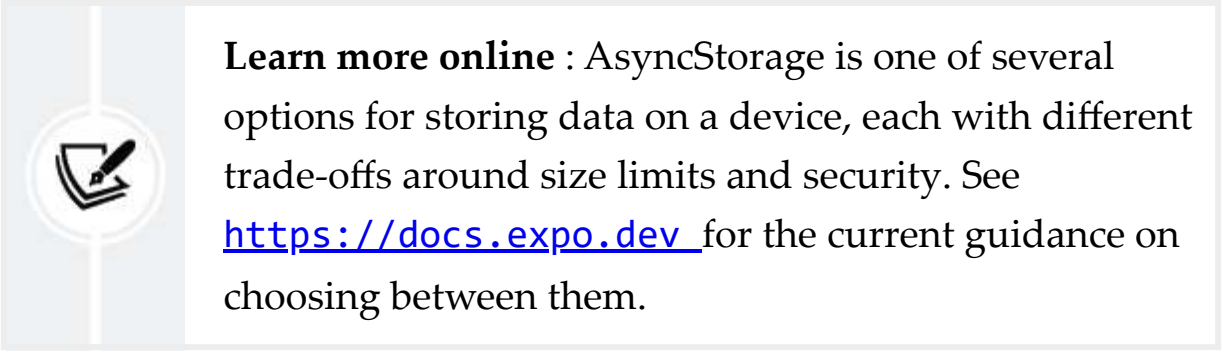


What to ask next : Once persistence works reliably, it's worth testing its edges:

```
Simulate corrupted data in AsyncStorage and confirm th
```

Check the result

- ☐ Do habits survive a full force-close and reopen of the app?
- ☐ Does a new user see a clear message, not just a blank screen?
- ☐ Is the loading state visually distinct from the empty state?
- ☐ Does the app recover gracefully from corrupted stored data, rather than crashing?



Learn more online : AsyncStorage is one of several options for storing data on a device, each with different trade-offs around size limits and security. See https://docs.expo.dev for the current guidance on choosing between them.

HabitTrail now remembers its habits the way a real app should, and handles the edge cases around that reliably rather than optimistically. The next chapter adds the two features that make a habit tracker feel genuinely useful day to day: reminders and sharing progress.

Chapter 3.10 Add Reminders and Sharing

In this chapter, you will add two features that make HabitTrail feel like a real, useful app: a daily reminder for each habit, and a way to share progress with someone else. Reminders need the phone's permission, requested at the right moment. Sharing uses the phone's own share sheet.

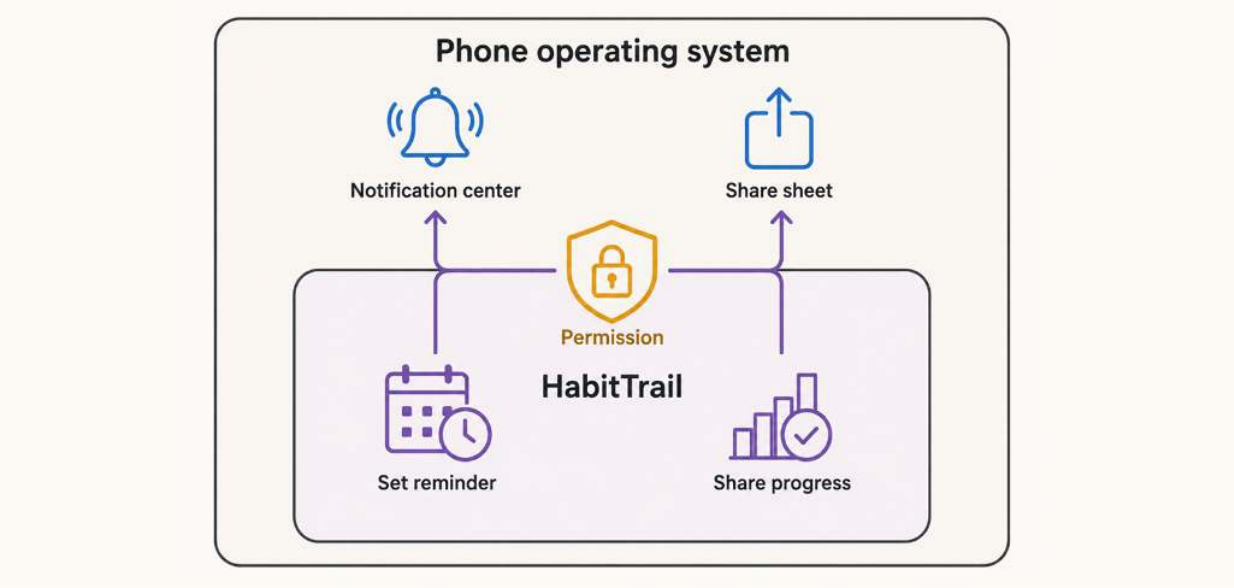


Figure 3.10A: Reminders need the phone's permission, requested at the right moment. Sharing uses the phone's own share sheet.

You need the persisted, working app from *Chapter 3.9* . By the end of this chapter, HabitTrail can remind a user about a habit and let them share how they're doing.

Request notification permission at the right moment

Every notification a habit tracker sends requires the user's permission first. Asking for that permission the instant the app opens, before a user has any reason to want reminders yet, tends to get refused. Asking at the moment it's actually relevant, when a user sets their first reminder time in *Chapter 3.7* 's form, gives the request context:

```
Request notification permission only when a user sets a reminder time
```

This mirrors the permission habits from *Book 1* : a request that's clearly connected to what the user just asked for is easier to trust than one that arrives with no explanation. A user might set their first reminder from either screen, so this check needs to live somewhere both share, not be duplicated separately on each form.



Good practice : Request a permission only at the moment it's needed, with a clear reason attached. A user is far more likely to grant a request they understand than one that arrives unexplained.

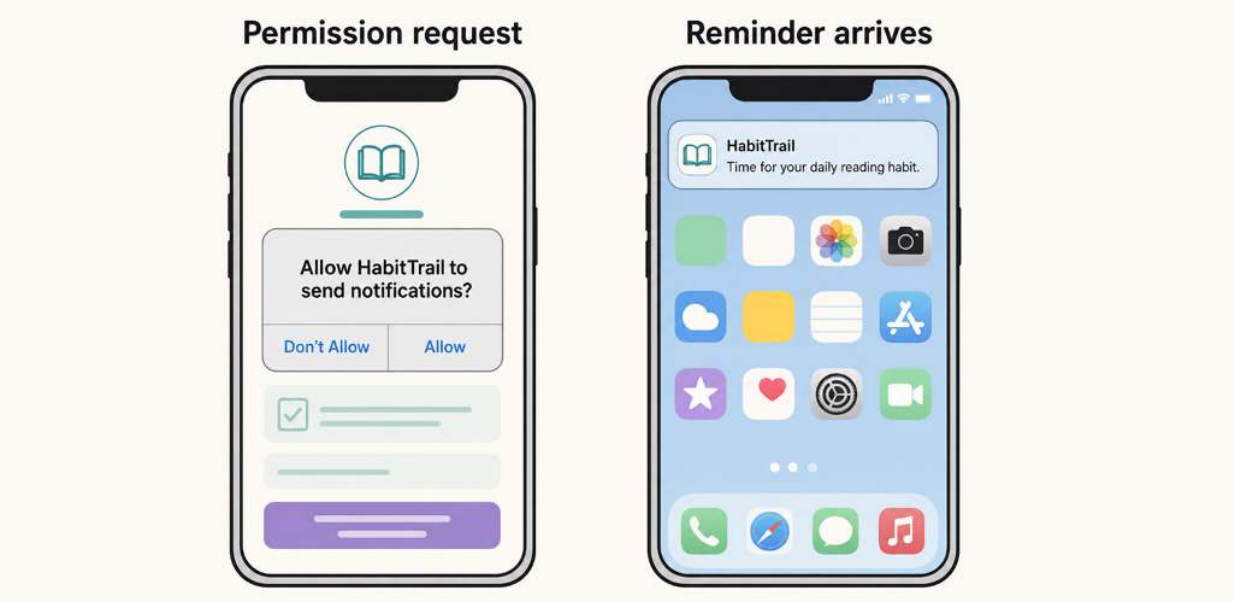


Figure 3.10B: A permission dialog that looks like the phone asking, because it is: HabitTrail only gets to send this reminder once you say yes

Schedule and cancel local reminders

Once permission is granted, HabitTrail can schedule a local notification: a reminder generated and delivered entirely by the phone itself, with no server involved, fitting this app's offline-first design.

Ask Claude Code to implement this feature:

```
When a user sets a reminder time for a habit, schedule a daily local r
```

Test this by setting a reminder a few minutes in the future and confirming it actually arrives, then confirming it stops if you clear that habit's reminder.

Any habit you created while testing *Chapters 3.7* through *3.9* may already have a reminder time saved, even though nothing was

scheduled for it yet, since that logic didn't exist until now. Reconcile those on the app's next launch, rather than only handling reminders set from this point forward:

```
Habits created earlier may already have a reminder time stored, from t
```

Test this by opening the app with a pre-existing habit that already has a reminder time set from earlier testing, and confirming a notification gets scheduled for it without you re-entering the reminder.

Share a progress message

Let a user share a short, encouraging message about their progress, using the phone's own built-in share feature, the same one apps use to open a share sheet for messaging, email, or social apps:

```
Add a share option on the Habit Detail screen that shares a short mess
```



Watch out : A scheduled reminder that doesn't get canceled when its habit is deleted will keep arriving for a habit that no longer exists. Confirm deletion and reminder cancellation are actually connected, not two separate, disconnected features.



Real-world example : A reader testing his app noticed a reminder still arrived for a habit he'd deleted two days earlier, since his first version only removed the

habit from the list, not its scheduled notification. Connecting deletion to notification cancellation fixed a small but genuinely confusing bug before it reached anyone else.

Try it now

Add a daily reminder that requests permission only when it's relevant, and a share option:

1. Confirm notification permission is only requested when a user sets their first reminder.
2. Confirm this works whether the reminder is set from the Add Habit form or from editing a habit on the Habit Detail screen.
3. Schedule a daily local reminder and test it a few minutes in the future.
4. Confirm that a habit with a reminder time already set from earlier testing gets a notification scheduled for it the next time the app opens.
5. Confirm clearing a reminder or deleting its habit cancels the corresponding notification.
6. Add a share option using the phone's standard share feature.



What to ask next : Once reminders and sharing work, it's worth confirming the permission flow feels right, not just functional:

```
Walk me through exactly what a user sees and is asked,
```

Check the result

- ☐ Is notification permission requested only when relevant, with a clear reason?
- ☐ Did a test reminder actually arrive at the scheduled time?
- ☐ Does requesting permission work correctly whether the reminder is set from the Add Habit form or the Habit Detail screen?
- ☐ Does a habit with a pre-existing reminder time, set before this chapter, get its notification scheduled automatically on the next app launch?
- ☐ Does deleting a habit or clearing its reminder cancel the corresponding notification?
- ☐ Does the share option produce a clear, correct progress message using the phone's standard share feature?

HabitTrail can now remind a user about a habit and let them share their progress, both using permissions requested at a moment that makes sense. The next chapter steps back from adding features and tests the whole app thoroughly, on a real device, the way an actual user would encounter it.

Chapter 3.11 Test on Real Devices and Repair Problems

In this chapter, you will test HabitTrail thoroughly on a real device, going beyond the quick checks you've done after each feature. You will inspect logs to understand a failure, and check the app's accessibility the way a screen reader user would actually experience it.

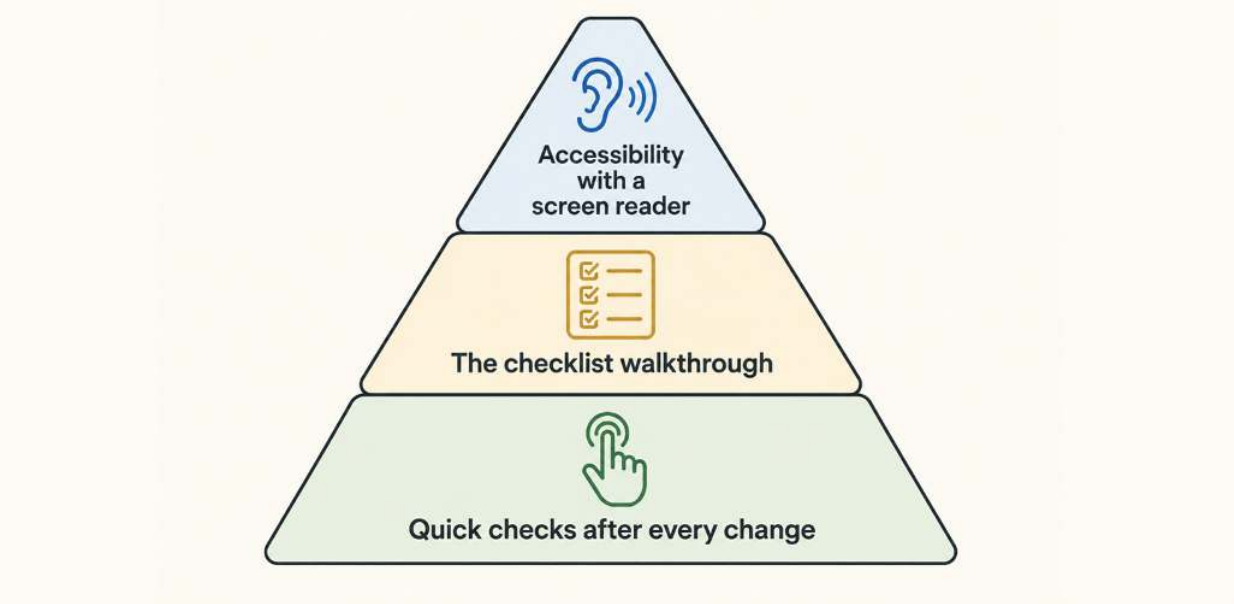


Figure 3.11A: Most testing in this book has already happened in small, quick checks. This chapter adds two more deliberate layers: a full walkthrough and a real accessibility check.

You need the complete, feature-full app from *Chapter 3.10* . By the end of this chapter, you will have a short list of issues, found deliberately rather than by accident, and fixed using the debugging habit from *Book 1* .

Test behavior, layout, offline use, and edge cases

Go through HabitTrail as a genuinely new user would, not as someone who already knows exactly what to expect:

1. Add several habits, including one with a very long name, and confirm the layout still looks right.
2. Mark habits done and undone repeatedly, confirming the state and any derived streak stay correct.
3. Turn on airplane mode and confirm the app, being offline-first, still works normally.
4. Try deleting your last remaining habit and confirm the empty state from *Chapter 3.9* appears correctly.

Here's an example prompt:

```
Walk through this checklist with me: a very long habit name, repeated
```

Inspect logs and reproduce failures

If something breaks during testing, use the same reproduce-inspect-hypothesize-fix-verify loop from *Chapter 1.10* , now applied to a mobile app. Expo's development server prints logs you can read directly, and most of the same debugging habits transfer directly from the web projects earlier in this volume:

```
Here's what happened: [steps to reproduce]. Here's what the logs show:
```

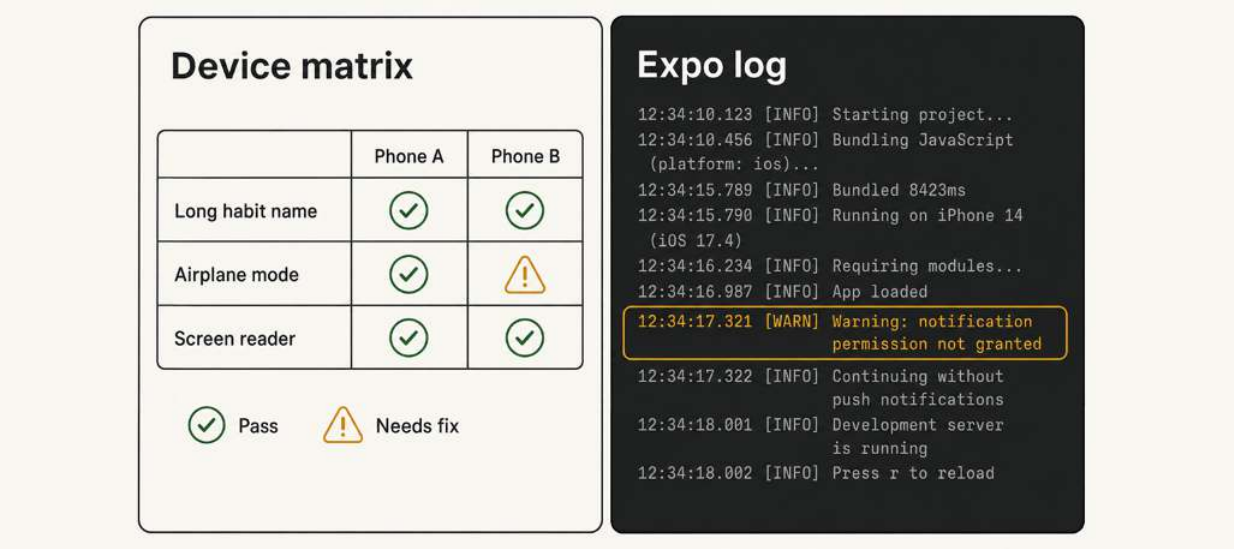


Figure 3.11B: A simple record of what you've tested, and the same habit from Book 1: most of a log is context, one line usually points at the real problem



Watch out : A bug that only appears offline, or only with a very long habit name, or only after marking and unmarking a habit several times, is easy to miss if you only ever test the easy, expected path. Deliberately testing edge cases is what catches these before a real user does.

Check touch targets, contrast, labels, and screen readers

Beyond functional testing, check HabitTrail the way a user with different needs would experience it. Turn on your phone's screen reader, VoiceOver on iOS or TalkBack on Android, and try navigating through the Habits list and Add Habit form using only its cues:

```
Check every tappable element in the app for adequate touch target size
```

If a button announces only as “button,” with no indication of what it does, that’s a specific, fixable problem worth catching now, before real users depend on it.



Good practice : Test with your phone’s screen reader turned on at least once, even briefly. Reading about accessibility and actually experiencing a screen through it are different, and the second one catches problems the first one can miss.



Real-world example : A reader testing his app in airplane mode discovered the Add Habit form’s submit button silently did nothing without an internet connection, even though the app was supposed to be entirely offline-first. The button had accidentally been wired to check for a connection it never actually needed. Testing offline use directly, rather than assuming an offline-first app was automatically offline-safe everywhere, caught a real problem before it reached anyone else.

Try it now

Work through HabitTrail’s behavior and accessibility checklist on a real device, and fix what you find:

1. Work through the behavior and edge-case checklist: long names, repeated marking, airplane mode, deleting the last habit.
2. Fix anything found, using the reproduce, inspect, hypothesize, fix, verify pattern from *Chapter 1.10* .
3. Turn on your phone's screen reader and navigate the Habits list and Add Habit form using only its cues.
4. Fix any touch target, contrast, or missing label problems you find.



What to ask next : Once your testing checklist is clear, it's worth asking what you might have missed:

```
Based on everything HabitTrail does, what's one realis
```

Check the result

- ☐ Does the app behave correctly with a very long habit name and repeated marking and unmarking?
- ☐ Does everything still work correctly in airplane mode?
- ☐ Can you navigate the app's main screens using only your phone's screen reader?
- ☐ Have you fixed every specific problem this chapter's testing found?

HabitTrail has now been tested deliberately, not just casually, and holds up under real conditions and real accessibility needs. The next

chapter turns your attention outward, preparing the app's identity, icon, and store presence for the world beyond your own phone.

Chapter 3.12 Prepare the App for Release

In this chapter, you will prepare HabitTrail's identity and store presence: its name, icon, and the assets and information both app stores expect to see before they'll list an app at all.

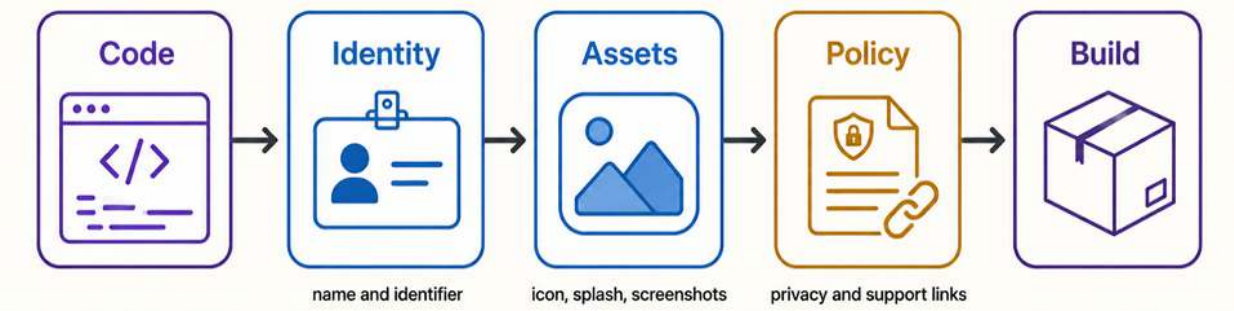


Figure 3.12A: A working app is only part of what a store submission needs. Identity, assets, and policy all have to line up before the build in the next chapter.

You need the tested app from *Chapter 3.11* . By the end of this chapter, HabitTrail will be configured and ready for the actual build and submission process in *Chapter 3.13* .

Choose app name and identifiers

An app needs a display name, shown to users, and a unique identifier, a behind-the-scenes technical name that identifies it to both app stores, usually written in reverse-domain form such as `com.yourname.habittrail` . This identifier can't easily change once you've submitted the app, so choose it deliberately now.

Ask Claude Code to help you choose and update the app configuration:

```
Help me choose a suitable bundle identifier for HabitTrail, and update
```



Good practice : Choose your app's identifier carefully before your first real submission. Changing it later effectively means starting over with the app stores as a new listing, not simply renaming the existing one.

Create icon, splash, screenshots, and listing copy

Both app stores need a set of visual and written assets before they'll accept a submission: an app icon at several sizes, a splash screen shown briefly while the app loads, screenshots of the app in use, and listing copy describing what it does.

Ask Claude Code to help you complete this task:

```
List everything HabitTrail's app.json currently needs for its icon and
```

Take your own screenshots directly from your phone, showing the real app you built, rather than anything staged or invented.



Watch out : Store policies, required assets, and exact size specifications change over time, and differ between Apple and Google. Confirm current requirements before finalizing your assets, rather than trusting a

fixed list that may be out of date by the time you read it.

Finalize privacy, support, permissions, and version settings

Both app stores require a privacy policy and a support contact before listing an app. HabitTrail already has both: the Privacy and Support pages you built and published in *Book 2*.

Ask Claude Code to add links to privacy and support pages:

```
Add links to HabitTrail's published Privacy and Support pages into the
```

Set an initial version number too, such as `1.0.0`, which you'll increase each time you submit an update.

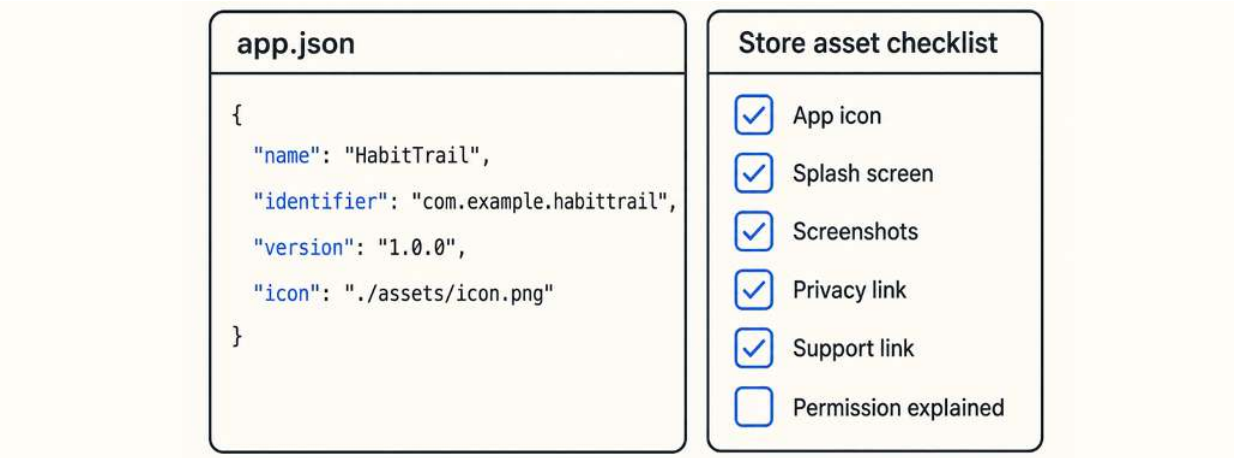


Figure 3.12B: A short, concrete checklist turns 'ready for the store' from a feeling into something you can actually confirm



Real-world example : A reader preparing his app for release realized, only while writing his listing copy, that his privacy page from *Book 2* hadn't been updated to mention the notification permission he'd since added. Reviewing his actual app against his actual privacy page caught a real mismatch before a store reviewer, or a cautious user, could have caught it instead.

Try it now

Set HabitTrail's final identity and assets, and confirm your listing matches what the app actually does:

1. Choose and set HabitTrail's final app name and bundle identifier.
2. Create or generate an app icon and splash screen, and take real screenshots from your phone.
3. Draft honest listing copy based on your *Chapter 3.1* brief.
4. Link your *Book 2* Privacy and Support pages, list your one permission with a plain-language reason, and set an initial version number.



What to ask next : Once your assets and settings are in place, it's worth a final consistency check:

```
Compare what HabitTrail actually does, including its o
```

Check the result

- ☐ Does HabitTrail have a final name and a deliberately chosen identifier?
- ☐ Do you have an icon, splash screen, and real screenshots ready?
- ☐ Does your listing copy accurately describe what the app actually does?
- ☐ Does your privacy information accurately reflect every permission the app actually requests?



Learn more online : Icon sizes, screenshot requirements, and listing requirements are set by Apple and Google and change periodically. Search each store's current developer documentation for exact, up-to-date specifications before finalizing your assets.

HabitTrail is now configured with the identity, assets, and information both app stores expect. The next chapter of this book turns that preparation into real, signed builds, gets them in front of a few test users, and submits HabitTrail for review.

Chapter 3.13 Build, Beta Test, and Submit

This is the end of HabitTrail's path from an idea to something someone else could actually install. You will create real, signed builds of the app, get them in front of a few test users, gather their feedback, submit HabitTrail for app store review, and once it passes review, it will move into production where it will be available for anyone to install:

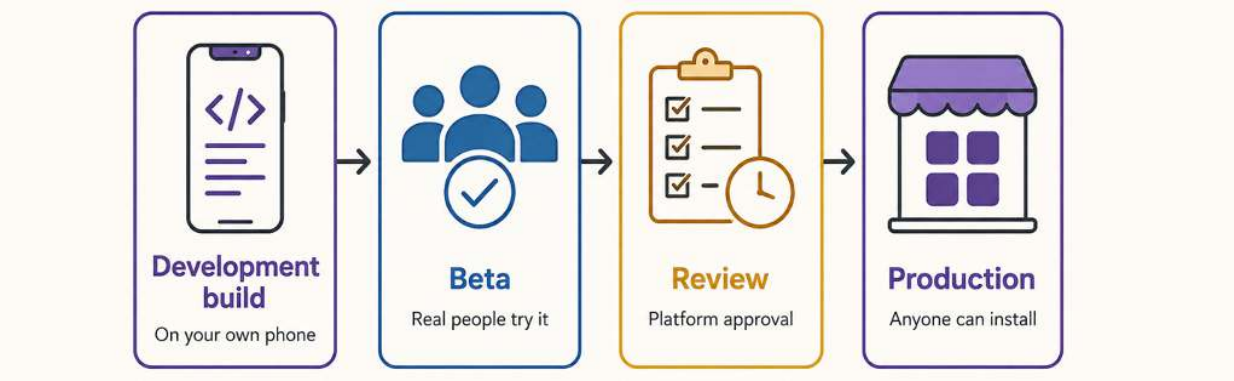


Figure 3.13A: From a build on your own phone, to a small group of real testers, to each platform's own review, to something anyone can install

You need the fully configured app from *Chapter 3.12* , an Expo account, and developer accounts with Apple and Google, which signed builds and both testing services require. Store requirements, fees, and review steps change over time, so this chapter teaches the workflow and points you to a maintained source for current specifics, rather than printing details likely to go out of date.

Create signed Android and iOS builds

Until now, HabitTrail has run through Expo Go, which is convenient for development but isn't how a finished app reaches real users. A signed build is a genuine, installable version of your app, cryptographically tied to your developer account, the form both app stores require.

Ask Claude Code to help you set up signed builds of the app:

```
Set up EAS Build for this project, and create signed builds for both A
```

This step depends on current Expo tooling and both platforms' current requirements. Follow Claude Code's guidance closely, and expect it to involve creating or confirming developer accounts with Apple and Google if you haven't already.

Distribute through internal testing and TestFlight

Before releasing HabitTrail publicly, both platforms offer a way to share it with a small group of real testers first: Google Play's internal testing track and Apple's TestFlight. Neither requires a full public review to use.

Ask Claude Code to help you distribute the app:

```
Help me distribute this build through Google Play's internal testing t
```

Invite a handful of people you trust, ideally including at least one person who hasn't seen the app before, and ask them to genuinely try

it: add a habit, set a reminder, mark it done over a few days if they're willing.

Gather feedback and submit for review

Ask your testers specific questions, the same way you'd want a specific bug report rather than a vague "it's fine." What was confusing? Did anything not work as expected? Would they actually keep using it?

Address anything real that comes back before moving on. Once you're satisfied, submit HabitTrail for the platforms' own review process.

Ask Claude Code to help you submit the app for review:

```
Once I've addressed my testers' feedback, help me submit HabitTrail fc
```

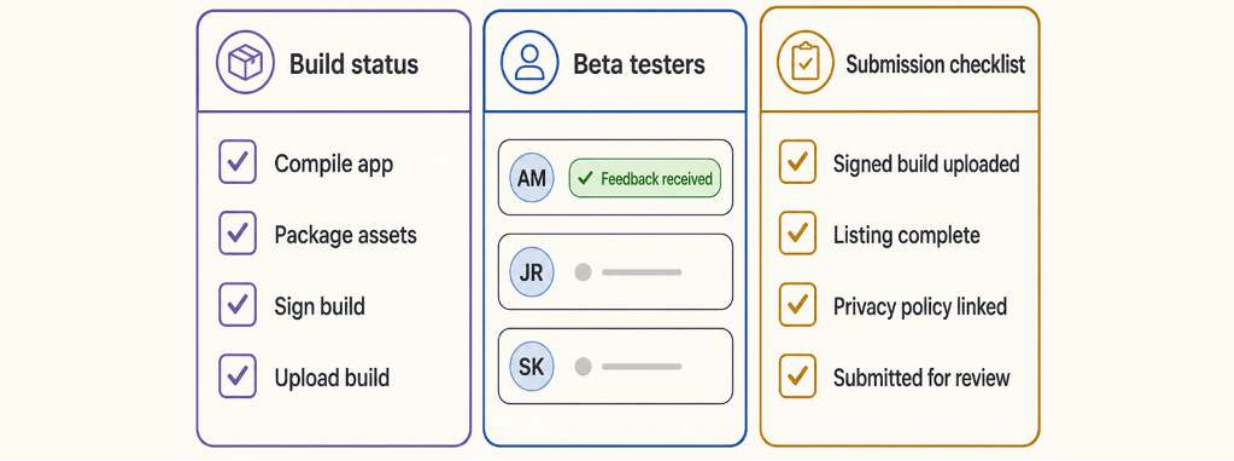


Figure 3.13B: A build, a small group of real testers, and a completed checklist: the last concrete steps between your project and someone else's phone



Good practice : Ask testers specific questions rather than only “any thoughts?” Specific questions, such as “did the reminder arrive at the right time?,” produce feedback you can actually act on.



Watch out : App store review requirements, timelines, and forms change and differ between platforms, and can differ again by the time you read this. Confirm current requirements directly with each platform’s own documentation before submitting, rather than assuming a fixed process.



Real-world example : A reader’s first beta tester, someone who hadn’t seen HabitTrail before, immediately got confused by an icon whose meaning was obvious to the reader after weeks of building the app, but meant nothing to someone seeing it for the first time. That one piece of feedback, from someone genuinely outside the project, caught something no amount of testing it himself would have revealed.

Try it now

Build, beta test with real users, and submit HabitTrail for app store review:

1. Set up EAS Build and create signed builds for Android and iOS.

2. Distribute a build through Google Play's internal testing track and Apple's TestFlight.
3. Invite a small group of testers, including at least one person new to the app, and ask specific questions.
4. Address meaningful feedback, then submit HabitTrail for review on both platforms.



What to ask next : Once your app is submitted, it's worth understanding what comes after:

```
What typically happens between submitting for revi
```

Check the result

☐ Do you have signed builds for both Android and iOS?

☐ Has at least one person outside yourself genuinely tried the app?

☐ Have you addressed the specific, meaningful feedback you received?

☐ Has HabitTrail been submitted for review on at least one platform?



Learn more online : Signed builds, beta distribution, and store submission depend on Expo's current tooling and both platforms' current policies, fees, and forms, all of which change over time. See

https://docs.expo.dev for Expo's current build and submission guidance, and each platform's own developer documentation for current store requirements.

You built HabitTrail from a one-sentence idea about a single repeated habit, through screens, state, storage, reminders, and testing, to a real, signed app that someone else has actually tried. That's every habit from this entire volume, used together on a project real enough to put in front of a stranger. In the final chapter, you will build and ship your own project, something you would really use yourself.

Chapter 3.14 Project – Build and Ship a Mobile App You Would Use

This chapter is your final project for *Book 3*. You will decide what a small, real mobile app should do, build it with Claude Code using everything from this book, and test it thoroughly enough to hand it to someone else.

You need everything from this book up to this point: a good idea for a working Expo project you're comfortable creating from scratch, plans for screens built from reusable components, navigation between them with Expo Router, a screen that responds to state, a form that rejects an incomplete entry, a full set of create, read, update, and delete operations, data that survives closing the app, and a habit of testing on a real device before you trust anything works. Budget considerably more time for this chapter than for any single chapter earlier in this book, several sessions spread across more than one sitting, especially if you decide to carry your app all the way to a signed build.

Decide what the mobile app will do

If you already have an idea for a small, offline-first app, use it. If not, here are a few starting points that suit a first mobile app well:

- **A reading log:** Someone working through a stack of books wants a simple way to log what they're currently reading, mark a book finished, and see how many they've completed this year. Add a

book, mark it done, see the count: the same shape as HabitTrail, applied to books instead of habits.

- **A workout counter:** Someone building a habit around one repeated exercise routine wants to log a session: which exercise, how many sets and reps, and when. A short daily reminder helps the habit stick.
- **A packing list:** Someone who travels often for work or family visits wants a reusable checklist they can check off before each trip and reset afterward, rather than starting from a blank list every time.
- **A recipe box:** Someone who cooks the same handful of meals often wants a short list of recipes with ingredients and steps, kept on their phone instead of scattered across bookmarked pages and screenshots.
- **A gratitude or mood journal:** Someone wants to log one short entry a day, at a reminder time that suits them, and look back over recent entries.

None of these need to be practical or entirely real. If nothing on this list fits, adapt one, or combine ideas. Pick whichever one you can picture finished, since that's the version you're actually about to build.

This is a good point to remind you that this book has an online discussion board hosted by my publisher Packt. There you can interact with me and other readers, so it is a great place to share ideas for projects and encourage each other. Join us at the following link: https://discord.gg/6axrHZzWe .



Good practice : Choose an idea backed by something you actually track or would use yourself, such as books you're currently reading or a workout you actually do. An idea built around data you have to invent as you go is harder to judge finished than one built around something real, even if you're the only person who'll ever open it.

Plan the app the way you planned HabitTrail

Reuse the planning habit from *Chapter 3.1* , applied to your own idea this time:

1. Write one or two sentences naming the person and their repeated task, the same way you named HabitTrail's user in *Chapter 3.1* .
2. Write down what a first version will and won't do. The smallest useful version of most of the ideas above lets someone add one thing, mark it done or complete, and see how they're doing. That's enough to be genuinely useful, and small enough to actually finish.
3. Map your screens and the data each one needs, in the same three-screen shape as HabitTrail if your idea fits it: a list, a way to add to it, and a detail view for one item.

Watch out : A reading log tracking pages, ratings, genres, a wishlist, and reading streaks all at once is not



a first version. It's a wish list. If your screen map has more than three or four screens, look for the ones you could add later, once the first version works on your own phone.



Real-world example : A reader who traveled most weeks for work built a two-screen packing list app instead of another habit tracker: one screen to manage a reusable master list of items, and one to check items off before a trip and reset them afterward. It reused every screen and storage pattern from this book, took a single weekend, and was still the first thing he opened while packing, months later.

Build the app with Claude Code

Work through the same loop you've practiced all through this book, one small increment at a time:

1. Create a new Expo project with TypeScript, the same way you did in *Chapter 3.2* , and confirm it runs on your own phone with Expo Go before you add anything to it.
2. Initialize Git and commit the starting project, then commit again after each step below.
3. Build your screens from reusable components and styles, the way you did in *Chapter 3.4* .
4. Connect your screens with Expo Router, the way you did in *Chapter 3.5* .

5. Make your app respond to a tap with props and state, the way you did in *Chapter 3.6* .
6. Build and validate your main form, the way you did in *Chapter 3.7* , so an incomplete or invalid entry can't get through.
7. Add the full set of create, read, update, and delete operations your idea needs, the way you did in *Chapter 3.8* , including a confirmation step before anything gets deleted.
8. Save your data to the phone with AsyncStorage, the way you did in *Chapter 3.9* , and confirm it survives a full force-close and reopen.
9. If your idea benefits from a reminder or a way to share progress, add one or both the way you did in *Chapter 3.10* . Not every idea needs both. Skip whichever doesn't fit yours.

Before you consider the app finished, look back through your commit history. Like your projects in *Chapter 1.12* and *Chapter 2.12* , it should read like a short, sensible story of how the app came together.



Good practice : Test each increment against your own brief as you build it, not only once everything exists. A form that validates correctly is easier to confirm the moment you build it than after three more screens have gone on top of it.

Test it thoroughly

Run through the same testing pass from *Chapter 3.11* , applied to your own app:

- Test behavior, layout, offline use, and any edge case your idea has, such as a very long entry or an empty list.
- Turn on airplane mode and confirm the app still works, since it should be entirely offline-first.
- Turn on your phone's screen reader and navigate your main screens using only its cues.
- Fix anything you find using the reproduce, hypothesize, fix, verify pattern from *Chapter 1.10* .



Watch out : A bug that only shows up with a very long entry, or only after several edits and deletions, is easy to miss if you only test the easy, expected path through your own app. Testing edge cases on purpose is what catches these before a real user does.

Decide how far to ship it

A finished app, tested thoroughly on your own phone and matching your brief, is already a genuine achievement on its own. Taking it further, all the way to a signed build a stranger can install, or all the way to public store review, is optional, and depends on whether you already have, or want, developer accounts with Apple and Google.

Choose a target by asking yourself: are you building this to practice the whole workflow yourself, or do you want a real listing other people can find and install?

If you only want to practice the workflow, follow *Chapter 3.13* as far as a signed build, and share it with at least one person outside

yourself through Google Play's internal testing track or Apple's TestFlight. That is enough to complete this project.

If you want a real public listing, follow every step of *Chapter 3.13* through to submission for both platforms.



Watch out : Apple and Google both charge a developer account fee, and their review times and requirements change over time. Check each platform's current developer site before deciding whether a public submission is worth it for this particular project, rather than assuming last year's figures still apply.



Learn more online : Signed builds, beta distribution, and store submission depend on Expo's current tooling and both platforms' current policies, fees, and forms. See `https://docs.expo.dev` for Expo's current build and submission guidance, and each platform's own developer documentation for current store requirements.

Try it now

Put the whole book to work on your own idea, from a blank folder to something a real tester opens on their own phone:

1. Pick one of the ideas above, or your own, and write a short brief: the user, the smallest useful version, and a three or four-screen map.

2. Set up the project: a new Expo project, Git initialized, and starter screens matching your map.
3. Build the app in small, committed increments: components and styles, navigation, state, a validated form, and full create, read, update, and delete operations.
4. Persist your data with AsyncStorage, and add a reminder or sharing feature if your idea benefits from one.
5. Test the finished app thoroughly on your own phone, including offline use and a pass with your screen reader.
6. Decide how far to ship it, and follow *Chapter 3.13* that far: a signed build shared with at least one tester, or a full public submission.



What to ask next : Once your app works and you've tested it yourself, it's worth finding out how it lands with someone who has never seen it:

```
Here's what my app does: [describe it in one or two sen
```

Check the result

☐ Does your app do what your brief describes, checked for real on your own phone, not assumed?

☐ Does your data survive a full force-close and reopen of the app?

☐ Have you tested the app offline and with your phone's screen reader at least once?

- ☐ Has at least one person outside yourself actually opened and tried the app?
- ☐ Does your commit history read as a sensible sequence of small, working steps, from an empty folder to a finished app?

You picked a real idea, mapped it into screens before you touched Expo, and built it the same way you built HabitTrail: in small, reviewed increments, checked against your own brief rather than assumed to work. Whether you stopped at a signed build tested by one honest tester, or carried it all the way to public review, you carried one idea from a blank folder to something a real person opened on their own phone. *Book 3* is complete.

End of Book 3

You started *Book 3* with a single sentence: one user, one repeated habit. You finish it with a working iOS and Android app, tested on a physical device, installed by testers who had never seen it before, and either submitted or ready to submit for public review. Every skill from every book in this volume went into getting there: a brief that scoped the work, instructions Claude Code could act on, a plan reviewed before every change, a project's own memory, checkpoints you trusted, careful debugging, and a habit of questioning what you approve.

HabitTrail, as you built it, is complete on its own terms and limited on purpose: it runs entirely on one device, with no account, no server, and no ongoing cost. Products people actually use often grow past that point, and the *Epilogue* that points you to what comes next: a second AI coding assistant, bonus books online, and where to learn Claude Code's more advanced capabilities, which were left out of this volume so you could stay focused on finishing something.

Epilogue

In this epilogue, you will learn:

- Where this book ends and your work continues
- Run the loop again, on your own idea
- What stays true as Claude Code changes
- Why you should spend your next US$20 on a second AI
- Adding a third AI coding assistant
- Bonus books available online
- What to do if you want to “go pro”
- A last word

Where this book ends and your work continues

You started this volume with an empty project folder and an unfamiliar terminal. You ended it with a published website and a tested mobile app ready for app-store submission. That progress did not come from one clever trick. It came from three stages, each building on the one before it.

Book 1 taught you to work safely with Claude Code: turn an idea into a brief, review its plans before it edits anything, and save your progress with Git checkpoints.

Book 2 taught you to structure, style, and publish a responsive website using HTML, CSS, and JavaScript.

Book 3 showed you how to turn that same design into a working iOS and Android app with React Native and Expo, complete with stored data, reminders, and sharing.

Run the loop again, on your own idea

Everything in this book followed the same four-stage loop from the *Introduction*: design, check, build, publish. That loop doesn't just belong to the habit-tracking app. It's yours to apply to your own projects.

The next time you have an idea, whether it's a website for a hobby, a small tool for your own work, or another app, you already know how

to start:

- Write a one-page brief, the same way you did in *Book 1*, including who it's for, the smallest useful version, and how you'll know it works
- Give Claude Code a `CLAUDE.md` with your project's structure, commands, and style
- Ask Claude Code to plan before it edits and review the plan before you say yes
- Build in small, checked steps, with Git checkpoints you can return to
- Publish it, even a rough version, so it's real enough to test

You don't need a new book to do this again. You need your next idea:



Epilogue Figure 1: Your next idea feeds the design-check-build-publish loop

What stays true as Claude Code changes

Claude Code will keep changing. Menu names, default modes, and exact controls may look different by the time you read this. Some of the screenshots in these books are probably already a step behind the current interface.

What should not change is the way you approached each task: define what you want, review the plan, verify what Claude Code gives you, and decide for yourself whether the result is good enough to use. That judgment is the skill that lasts. It will transfer to whatever Claude looks like next year, and to any AI tool you use after that.

Why you should spend your next US$20 on a second AI

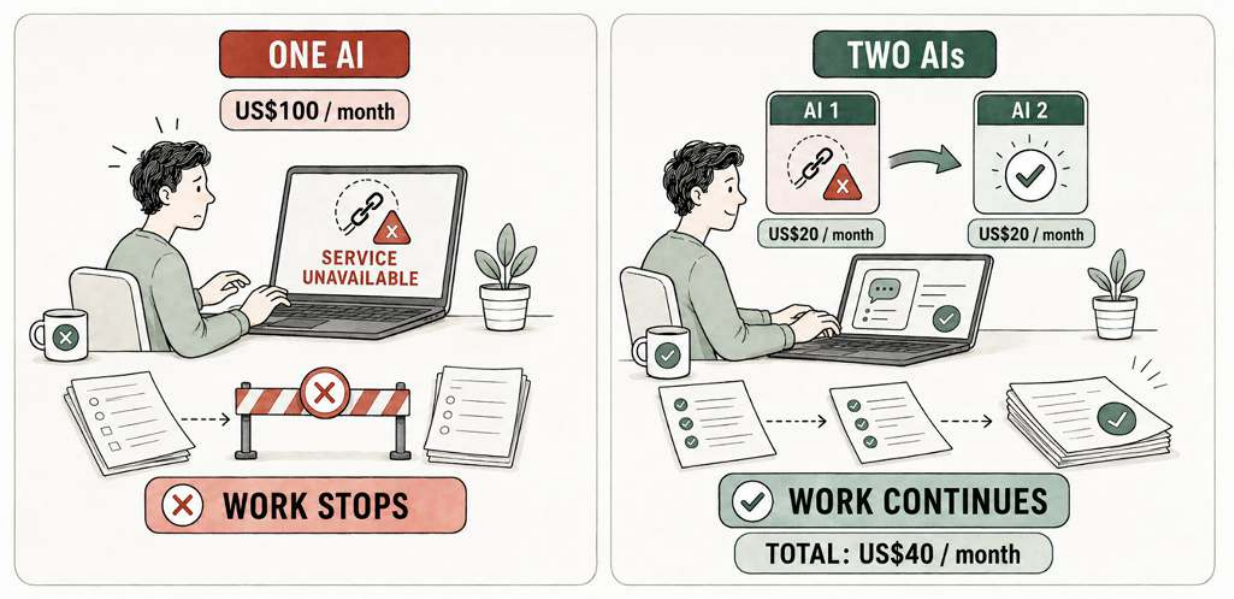
When Claude Code starts running into its limits, the obvious next step is to pay Anthropic more: upgrading from Pro to Max, from around US$20 a month to US$100 or US$200, depending on the tier available in your country at the time. Before you make that jump, consider a different one. Keep your Pro plan at US$20 and spend another US$20 on a second AI coding tool instead.

Think of Claude Code and OpenAI Codex as the Coke and Pepsi of AI coding assistants. Both are genuinely capable general-purpose tools, and picking a primary between them is closer to personal taste than a clear technical verdict. Whichever one you did not choose as your primary is usually the strongest possible secondary: not a

downgrade, but a second frontier-grade model with its own way of coding, reasoning, and handling a task.

A second full coding tool earns its US$20 in a way a bigger single-vendor plan often doesn't. This volume has asked you, in almost every chapter, to check a result rather than trust it on the first try. A second AI extends that same habit: run a demanding request past both, and you get a genuine second opinion instead of the same model reasoning with itself twice. It also protects you from a single company's outage, a single model's blind spot, or losing access to work stored only with one vendor.

That last risk is the easiest to underestimate until it happens to you. Major AI coding assistant providers can suspend or terminate accounts, sometimes for a genuine policy violation and sometimes because an automated fraud or abuse system flags something ordinary as suspicious, and the reason given is rarely detailed enough to argue with.



Epilogue Figure 2: Using two AI coding tools instead of one larger plan provides redundancy as well as a second perspective

Anthropic publishes account-enforcement statistics every six months. Its January–June 2026 report lists 11.4 million banned accounts, about 398,000 appeals, and about 42,000 appeal overturns. Those figures change substantially between reporting periods, so check Anthropic's current Transparency Hub rather than relying on the numbers printed here: https://www.anthropic.com/transparency/system-trust-reporting_.

OpenAI describes its appeals process for banned Codex and ChatGPT accounts but doesn't publish how often it bans accounts or how often appeals succeed.

Losing access can still be disruptive even when the suspension is later overturned. If losing this work would set you back, keep important outputs and instructions somewhere you control rather than leaving your only copy inside one AI account.

This is the brutal version of the case made earlier in this section for a second AI subscription: not just a second opinion, but a working account you can still open the day your primary one goes dark. Export or back up anything you would genuinely miss, such as your project's code, its `BRIEF.md` and `CLAUDE.md` file, the same way you would back up a hard drive, rather than trusting your only copy to an account you do not ultimately control. Remember that a project pushed to GitHub is already backed up outside the AI account.

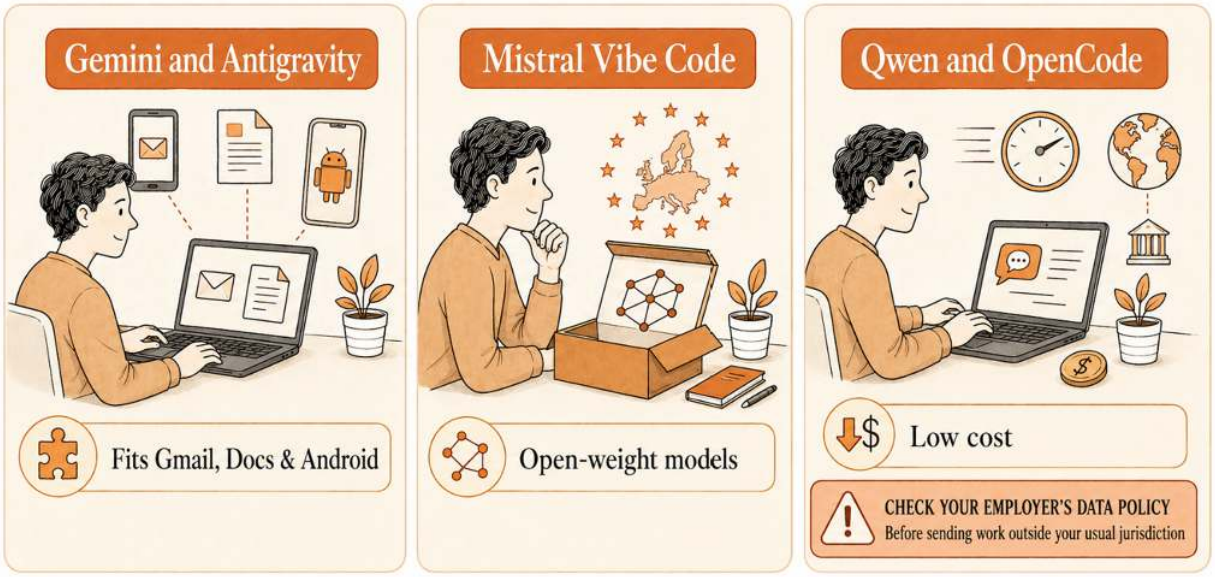
With Claude Code as your primary AI coding assistant, if you choose OpenAI Codex as your second one and you want to learn Codex

properly rather than just hope that all its features work in the same way that Claude Code does, I am planning to publish an *OpenAI Codex Visual Bible* that walks through it with the same three-stage structure as this volume.

Adding a third AI coding assistant

A third AI is even more optional and worth adding only once your first two are earning their keep.

Google's Gemini in combination with Google Antigravity is the natural choice if you already live inside Gmail, Docs, or Android. Mistral Vibe Code is worth a look if open-weight models or EU-based data handling matter to your work. And one of the fast-improving Chinese providers, such as DeepSeek, Qwen Code, or Kimi, can be worth a try for their low cost. A growing set of open-source agents like OpenCode can run any of them through one interface, though check your employer's data policy before you send anything work-related through a platform hosted outside your usual jurisdiction.



Epilogue Figure 3: Choosing a third AI coding assistant provider

Bonus books available online

I have also written two bonus books, which you can find in the online GitHub repository:

- *Beyond Your First AI Coding Assistant* : This explains when a different AI coding service is the better tool for a specific job, or a backup when your primary one is unavailable. It covers alternatives like Google Antigravity, Mistral Vibe Code, or some Chinese providers like Qwen Code. https://github.com/markjprice/claude-code-vb/blob/main/docs/bonus/beyond/README.md
- *Local AI with Open Models* : This covers running an AI model entirely on your own laptop or desktop, with nothing leaving your device, using free and open-weight models like Google Gemma 4, Alibaba Qwen3.8, Meta Muse Glimmer, and NVIDIA Nemotron 3.5 Lightning. Check the online page for current open-

weight model names and recommendations. They change often, which is why I've kept this material online instead of printing it. https://github.com/markjprice/claude-code-vb/blob/main/docs/bonus/local-ai/README.md

What to do if you want to "go pro"

In *Claude Code Visual Bible* , you built websites and mobile apps for your own use to solve problems you had. You may have wondered what it would take to become a professional software developer or engineer, instead of only asking Claude Code to build apps for you without understanding the underlying code and frameworks. That is the boundary between this volume and the one I recommend if you want to "go pro".

Agentic Coding with Claude Code picks up exactly where *Claude Code Visual Bible* left off. It teaches you context engineering, persistent memory, and how to bring MCP, plugins, skills, hooks, and multi-agent workflows into a professional development workflow. This book pays off most if you're planning a career in software or want to go deeper into the code Claude Code writes for you. If that describes you, *Agentic Coding with Claude Code* is the next book worth picking up.

A last word

Three books and 38 chapters is a lot of material to work through, especially if you diligently completed all the **Try it now** sections, and you likely skipped some sections that did not apply to you yet. Keep this volume where you can reach it. The chapter you skimmed on Git

checkpoints or the accessibility audit may be exactly the one you need in six months.

This book was written to close the gap between using Claude Code and getting genuine value from it. What you do with that next is up to you. Every chapter in this book asked you to check the result before you trusted it. That's the skill that outlasts any single tool. Everything else, Claude Code included, will keep changing.

Go build something.

Share your thoughts

Now you've finished *Claude Code Visual Bible* , we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



https://packt.link/r/180865367X

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.