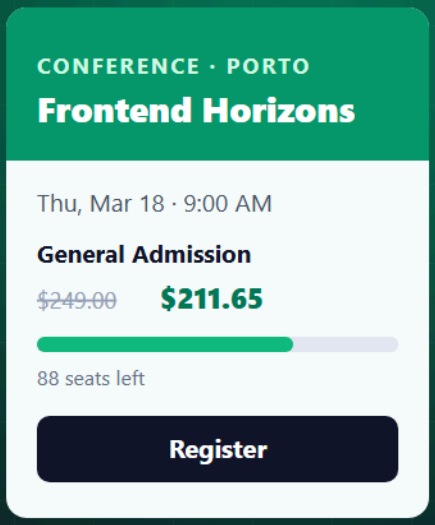
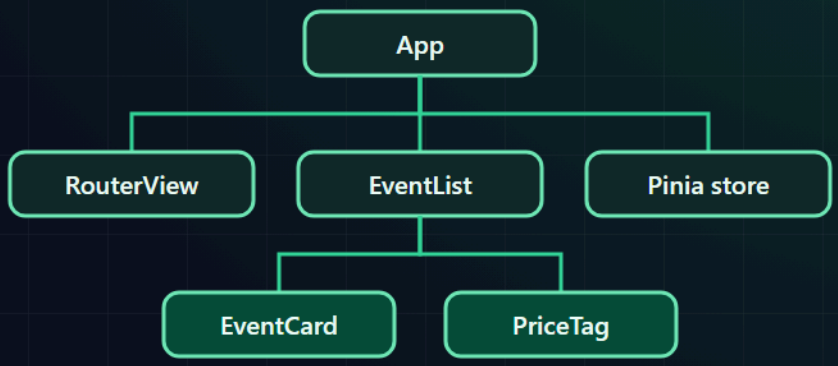
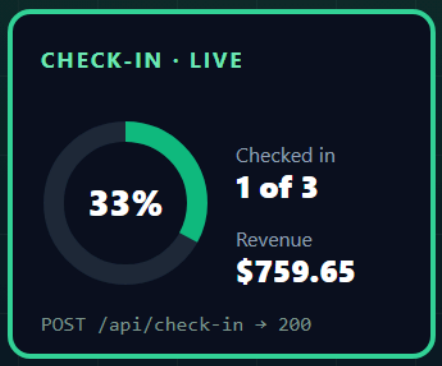








A COMPLETE, PRACTICAL GUIDE

Vue.js 3 and Nuxt from Zero to Production

Build modern web apps with the Composition API — from first component to deploy

SFCs · Reactivity · Composables · TypeScript · Vue Router · Pinia
Vitest · Playwright · Nuxt · SSR · Nitro · Security · CI/CD

VUE 3 · NUXT · TYPESCRIPT · TESTING · PRODUCTION

THENEXBOOKS

Vue.js 3 and Nuxt from Zero to Production

Build Modern Web Apps with the Composition API
From your first component to a tested, secured, server-rendered application — built step by step

First Edition

THENEXBOOKS

Vue.js 3 and Nuxt from Zero to Production
Build Modern Web Apps with the Composition API

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Vue, Nuxt, Vite, and the other tools in this book are actively developed open-source projects; APIs, defaults, command-line prompts, and folder layouts can change between versions. Always consult the official documentation for the versions you use.

Vue.js, Nuxt, Vite, Vitest, Pinia, Node.js, TypeScript, Visual Studio Code, Playwright, GitHub, Docker, and all other product and project names mentioned in this book are trademarks or registered trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the Vue.js project, the Nuxt project, or any company or foundation behind them.

Gatherly, the event-management platform used as the running example in this book, is fictional, as are its events, venues, speakers, companies, attendees, registrations, ticket codes, and all sample data. Email addresses and domains use reserved example domains.

Preface

Modern web applications are expected to do a lot. They must load quickly on a phone, react instantly to every click, keep complex data consistent across many screens, work with a keyboard and a screen reader, show up properly in search results, and stay secure while real people use them. Building all of that by hand with plain JavaScript and the DOM quickly turns into a tangle of event listeners and manual updates that nobody wants to maintain.

Vue was designed to make this kind of work approachable. You describe what the screen should look like for a given state, and Vue keeps the page in sync as the state changes. Its Composition API lets you organize logic by feature and reuse it across components, its single-file components keep markup, logic, and styles together, and its ecosystem — Vue Router, Pinia, Vite, Vitest — covers everything a real application needs. Nuxt builds on Vue to add file-based routing, server rendering, server API routes, and a production-ready deployment story, so the same skills take you from a small interactive widget to a full-stack application.

This book teaches Vue 3 and Nuxt from the very beginning and takes you all the way to production. We do not tour the APIs one by one. We build one real application step by step, introducing each feature at the moment it solves a problem we actually have, and explaining *why* it works the way it does, so you can make good decisions in your own projects.

Who This Book Is For

- **Developers who know basic JavaScript, HTML, and CSS** and want to build modern, maintainable web applications with a component framework.
- **Front-end developers coming from jQuery or plain JavaScript** who want reactivity, components, routing, and state management without the complexity.
- **Developers who already know another framework** and need to become productive in a Vue or Nuxt codebase at work.
- **Full-stack and back-end developers** who want to build the whole application — pages, API routes, authentication, and deployment — in one TypeScript project with Nuxt.

You do not need any previous experience with Vue, Nuxt, or TypeScript. We introduce TypeScript gently and use only what a Vue application needs.

What You Will Learn

- How Vue thinks: templates, directives, events, and the reactivity system — `ref`, `reactive`, `computed`, and watchers — and how Vue tracks and applies changes.
- How to design components with single-file components and `<script setup>`: props, emits, `v-model`, slots, provide/inject, and built-in components such as Transition, Teleport, and KeepAlive.
- How to reuse logic with composables and write type-safe Vue code with TypeScript.

- How to build a complete single-page application: Vue Router with nested routes and guards, Pinia stores, validated forms, data fetching with loading and error states, and Vite tooling.
- How to test with confidence: unit and component tests with Vitest and Vue Test Utils, store and composable tests, and end-to-end tests with Playwright.
- How to make applications accessible and fast, and how to measure performance instead of guessing.
- How to build full-stack applications with Nuxt: file-based routing, layouts, `useFetch` and `useAsyncData`, server routes with Nitro, and SSR, static, and hybrid rendering.
- How to ship to production: authentication and sessions, runtime configuration, SEO and meta tags, security, Docker, deployment, and CI/CD.

How This Book Is Organized

Part I — Vue Foundations (Chapters 1–5) sets up your toolchain and teaches template syntax, reactivity, lists, forms, watchers, lifecycle hooks, and template refs. **Part II — Components and Composition** (Chapters 6–10) covers single-file components, props and emits, slots, provide/inject, composables, TypeScript with Vue, styling, and the built-in components. **Part III — Building the App** (Chapters 11–15) adds Vue Router, Pinia, forms and validation, data fetching against a mock API, and Vite in depth. **Part IV — Quality** (Chapters 16–20) covers unit, component, store, and end-to-end testing, accessibility, and performance. **Part V — Nuxt** (Chapters 21–24) rebuilds the application with Nuxt: project structure, data fetching, server routes, and rendering modes. **Part VI — Production** (Chapters 25–28) covers authentication, runtime configuration, SEO, security, deployment, CI/CD, and a complete capstone. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter includes exercises under **Try It Yourself** and ends with a **Key Takeaways** box for quick review.

The Running Example: Gatherly

Throughout the book we build **Gatherly**, a fictional event-management platform. Organizers publish events — a two-day conference called Frontend Horizons 2027 in Porto, an online workshop, and an evening meetup in Toronto — each with sessions, speakers, and ticket types. Attendees browse events, read the agenda, pick a ticket, apply a promo code, and register. On the day of the event, staff check attendees in at the door with a live dashboard that shows who has arrived.

Gatherly has real business rules: early-bird prices that expire at a precise moment, a promo code that does not apply to student tickets, limited capacity per ticket type, ticket codes, and a check-in window. All money is stored as whole cents, so every price in the book can be checked by hand. In Parts I–IV we build Gatherly as a single-page application with Vue, Vue Router, Pinia, and a small mock API you can run with Node.js. In Parts V and VI we rebuild it as a full-stack Nuxt application with server routes, server rendering, authentication, and a production deployment.

Versions and Conventions

The examples target **recent Vue 3.x versions** with the Composition API and `<script setup>`, together with recent versions of Vue Router, Pinia, Vite, Vitest, Vue Test Utils, and Playwright. The Nuxt chapters target **recent versions of Nuxt** and use the `app/` directory layout that recent versions generate; where a default changed between major versions, a Note says so. Install a current long-term-support (LTS) release of Node.js and check the minimum version your Vite and Nuxt releases require. Command-line prompts, generated files, and default options can change between releases; if something on your screen differs slightly from the book, the official documentation for your version is the final word.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause bugs, data loss, or security problems.

Code appears in `monospace`. A caption above each listing shows the file it belongs to, such as `src/stores/checkIn.ts` or `server/api/check-in.post.ts`; "Terminal" marks commands you type. Long lines are wrapped to fit the page. Code follows the style the official project generators use: two-space indentation, single quotes, and no semicolons. Dates in the sample data are stored in UTC and displayed in each event's own time zone, and prices are shown in US dollars.

Contents

Preface

PART I — VUE FOUNDATIONS

1 Why Vue: The Progressive Framework, Your Toolchain, and Your First App

2 Template Syntax: Interpolation, Bindings, Directives, and Events

3 Reactivity Fundamentals: ref, reactive, computed, and How Vue Tracks Changes

4 Lists, Conditionals, and Forms with v-model

5 Watchers, Lifecycle Hooks, and Template Refs

PART II — COMPONENTS AND COMPOSITION

6 Components: Single-File Components, Props, and Emits

7 Slots, Provide/Inject, and Flexible Component Design

8 Composables: Reusable Logic with the Composition API

9 TypeScript with Vue: Typed Props, Emits, Refs, and Composables

10 Built-In Components and Styling: Transitions, Teleport, KeepAlive, and Scoped CSS

PART III — BUILDING THE APP

11 Vue Router: Routes, Params, Nested Views, and Navigation Guards

12 State Management with Pinia

13 Forms and Validation: Registration and Checkout

14 Fetching Data: APIs, Loading States, Errors, and the Mock Server

15 Vite in Depth: Environment Variables, Builds, Linting, and Project Structure

PART IV — QUALITY

16 Unit and Component Testing with Vitest and Vue Test Utils

17 Testing Stores, Composables, Routing, and Async Code

18 End-to-End Testing with Playwright

19 Accessible Vue Applications

20 Performance: Rendering, Lazy Loading, and Bundle Size

PART V — NUXT

21 Nuxt Fundamentals: Project Structure, File-Based Routing, Layouts, and Auto-Imports

22 Data Fetching in Nuxt: useFetch, useAsyncData, and $fetch

23 Server Routes with Nitro: Building the Gatherly API

24 Rendering Modes: SSR, SSG, Hybrid Rendering, and Route Rules

PART VI — PRODUCTION

25 Authentication, Sessions, Runtime Config, and Route Middleware

26 SEO, Meta Tags, and Security

27 Deployment and CI/CD

28 Capstone: The Gatherly Check-In Dashboard, End to End

APPENDICES

A Vue and Nuxt Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Vue Foundations

Setting up a modern toolchain and learning how Vue thinks: templates and directives, events, the reactivity system with ref, reactive, and computed, lists and forms with v-model, watchers, lifecycle hooks, and template refs — while the first Gatherly screens come to life.

1 · Why Vue and Your First App

2 · Template Syntax

3 · Reactivity Fundamentals

4 · Lists, Conditionals, and Forms

5 · Watchers, Lifecycle Hooks, and Template Refs

CHAPTER 1 · PART I — VUE FOUNDATIONS

Why Vue: The Progressive Framework, Your Toolchain, and Your First App

Before we write a single component, we want you to understand what Vue actually does for you, why it is called a "progressive" framework, and what the tools around it are for. By the end of this chapter you will have a real project called `gatherly-web` *running on your machine and showing the first piece of Gatherly, the event-management platform we build throughout the book.*

What Vue Is, in Plain Words

Vue is a JavaScript framework for building user interfaces. That sentence is true of many tools, so let us be more specific about the problem it solves. When you build an interactive page with plain JavaScript, you spend a lot of your time keeping two things in sync: the *data* your program holds (a list of events, the number of seats left, whether a user has saved a session to their agenda) and the *DOM* the user sees. Every time the data changes, you have to find the right elements and update them by hand. Every time the user types or clicks, you have to read values back out of the DOM and update the data. The bigger the page, the more of these manual updates you write, and the easier it becomes to forget one.

Vue removes most of that bookkeeping. You describe what the page should look like *for a given state*, using an HTML-based template, and Vue keeps the DOM matching that description as the state changes. You change a variable; Vue works out which parts of the page depend on it and updates exactly those parts. This idea has two names that you will hear constantly: **declarative rendering** (you declare the result, not the steps) and **reactivity** (the page reacts to data changes automatically).

Here is the smallest useful illustration. The template says "show the title and the number of seats left." The script holds the data. Nothing in the code says "find the paragraph and change its text."

A tiny Vue component (illustration)

```
<script setup lang="ts">
import { ref } from 'vue'

const title = ref('Frontend Horizons 2027')
const seatsLeft = ref(91)
</script>

<template>
  <h2>{{ title }}</h2>
```

```
  <p>{{ seatsLeft }} seats left</p>
  <button type="button" @click="seatsLeft--">Sell one seat</button>
</template>
```

When you click the button, `seatsLeft` goes down by one and the paragraph updates. You did not touch the DOM. You changed the data, and Vue handled the rest. Do not worry about the details of `ref`, the double curly braces, or `@click` yet; Chapters 2 and 3 take each of them apart. For now, notice the shape: a script block with state, a template that describes the output, and a framework in between.

Why "progressive"?

The Vue team describes Vue as a **progressive framework**. The word means you can adopt as much or as little of it as your project needs. At one end of the scale, you can drop a single script tag into an existing server-rendered page and make one widget interactive, with no build step at all. In the middle, you can build a full single-page application (SPA) with a build tool, a router, a state store, and tests. At the far end, you can use a meta-framework such as Nuxt to render pages on the server, generate static sites, and write your API in the same project.

Components: the unit of everything

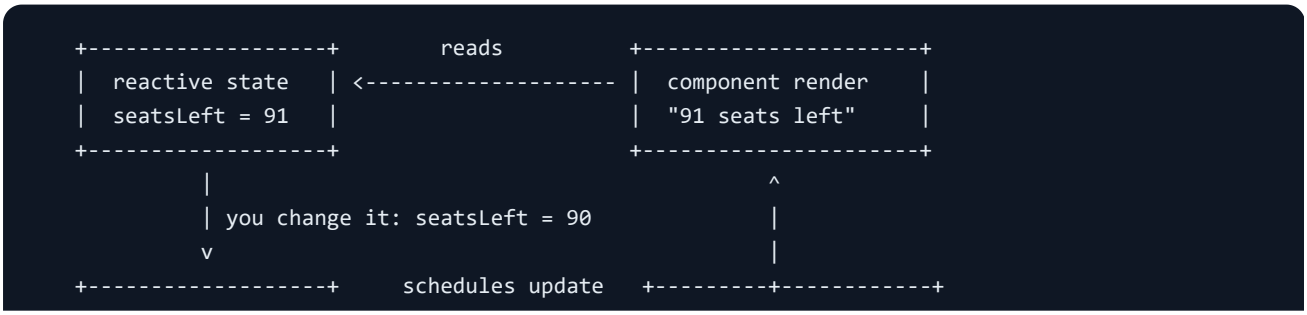
The second big idea is **components**. A component is a self-contained piece of UI with its own template, logic, and styles. An event card is a component. A price tag is a component. The whole registration form is a component that uses smaller components such as text inputs and a ticket picker. You compose components into a tree, and the tree is your application.

In Vue, components usually live in **Single-File Components** (SFCs): files ending in `.vue` that hold three blocks side by side, the script, the template, and the styles. The illustration above is a complete SFC except for the optional style block. Keeping the three together means everything about one piece of UI lives in one file, which makes it easy to find, change, and delete. A build tool (Vite, in our case) compiles `.vue` files into plain JavaScript that browsers understand.

Reactivity in one picture

The third idea is how Vue knows *what* to update. When a component renders, Vue records which pieces of reactive state the render read. Later, when one of those pieces changes, Vue knows which components depend on it and re-renders only those. Here is the loop as a sketch:

Diagram: the reactivity loop



```
    |  Vue's tracker    | ---------------------> |  re-render + patch   |
    |  "who read this?" |                        |  only changed nodes  |
    +-------------------+                        +----------------------+
```

Chapter 3 explains how this tracking works under the hood (it is based on JavaScript Proxies) and why updates are batched. You do not need those details to start, but it helps to know there is no magic: Vue simply remembers who read what.

Composition API and Options API

Vue offers two styles for writing component logic. Both are fully supported, and you will meet both in real projects, so it is worth seeing them side by side once.

The **Options API** is the older style. You describe a component as an object with named sections: `data` for state, `methods` for functions, `computed` for derived values, and so on. Vue wires the sections together and exposes everything on `this`.

Counter written with the Options API

```
<script lang="ts">
export default {
  data() {
    return { interested: 0 }
  },
  methods: {
    markInterested() {
      this.interested++
    },
  },
}
</script>

<template>
  <button type="button" @click="markInterested">
    Interested ({{ interested }})
  </button>
</template>
```

The **Composition API** is the newer style. Instead of filling in named sections, you write ordinary JavaScript (or TypeScript) inside `<script setup>`: you create reactive state with functions such as `ref`, declare normal functions, and anything you declare is available in the template.

The same counter with the Composition API

```
<script setup lang="ts">
import { ref } from 'vue'

const interested = ref(0)

function markInterested() {
  interested.value++
```

```
}
</script>

<template>
  <button type="button" @click="markInterested">
    Interested ({{ interested }})
  </button>
</template>
```

For a counter, the two look about the same. The difference shows up as components grow. With the Options API, the code for one feature (say, a countdown timer) is split across `data`, `computed`, `methods`, and lifecycle hooks, mixed in with the code for every other feature. With the Composition API, the code for one feature can sit together, and you can move it into a reusable function, called a **composable**, with almost no changes. Chapter 8 builds several of those for Gatherly. The Composition API also works naturally with TypeScript, because it is just functions and variables, which TypeScript understands without special help.

This book uses the Composition API with `<script setup lang="ts">` everywhere. The Options API is not wrong or deprecated, and plenty of healthy codebases use it. Chapter 6 includes a short "reading older code" sidebar so that you can work comfortably in an Options API project when you meet one.

> **NOTE**
>
> The `lang="ts"` attribute tells the build tool that the script block is TypeScript. In the first chapters we use only very light TypeScript, such as a type in angle brackets after `ref`, and we explain each annotation in a sentence when it appears. Chapter 9 teaches the TypeScript you need for Vue properly. If you know plain JavaScript, you can read every example in Part I.

Where Vue sits among frameworks

You have probably heard of other UI frameworks, such as React, Angular, Svelte, and Solid. They all solve the same core problem of keeping a UI in sync with state, and they make different trade-offs: some describe UI with JSX in JavaScript, some with templates; some track dependencies at run time, some lean more on a compiler; some ship a large set of built-in features, some keep the core small and rely on the ecosystem. Vue uses HTML-based templates compiled to efficient render functions, fine-grained reactivity, and an official set of companion libraries (Vue Router, Pinia, Vite, Vue DevTools, and Nuxt as the meta-framework). None of these choices is universally better; they are choices. If your team already knows another framework well, that knowledge transfers, because the concepts of components, props, state, and routing are shared across all of them.

Meet Gatherly

Learning a framework through isolated snippets leaves gaps: you know what each feature does, but not how they fit together. So every chapter in this book builds a real piece of one application. That application is **Gatherly**, a fictional event-management platform at `https://gatherly.example.com`.

Gatherly has three kinds of users. **Organizers** publish events such as conferences, workshops, and meetups, each with sessions, speakers, and ticket types. **Attendees** browse events, read the agenda, and register for a ticket. **Staff** stand at the door on the day and check attendees in by scanning their ticket code, while a live dashboard shows how many people have arrived. Our sample users are Priya Shah (an organizer), Diego Morales (door staff), and Maya Chen (an attendee); you will meet them properly when we add login in Chapter 12.

The sample data has four events. Three are published and visible to the public; one is still a draft:

Event	Format	City / venue	Status
Frontend Horizons 2027	in-person	Porto / Riverside Congress Hall	published
Component Craft Workshop	online	Online / Gatherly Live	published
Design Systems Meetup	in-person	Toronto / Maple Street Hub	published
Accessibility Summit 2027	in-person	Berlin / Spreeview Forum	draft

Frontend Horizons 2027 (id `ev-101`) is our flagship example. It is a two-day conference in Porto starting on Thursday, March 18, 2027, with three ticket types (General Admission, Workshop + Conference, and Student), five sessions, and four speakers. Many of the numbers you will see repeatedly come from it: 91 seats left in total, a sold-out Workshop + Conference ticket, a Student ticket with only 3 seats left, and the worked pricing example where Maya buys an early-bird General Admission ticket with the promo code `COMMUNITY15` and pays $211.65. We build up to all of that step by step.

The plan of the book

Here is the route we take, so you always know where a chapter fits:

- **Part I — Vue Foundations (Chapters 1–5).** Templates, reactivity, lists and forms, watchers and lifecycle. Everything lives in one `App.vue` for now, on purpose, so that you can focus on the language of Vue.
- **Part II — Components and Composition (Chapters 6–10).** We split the page into components, learn slots and dependency injection, extract composables, add TypeScript properly, and style everything.
- **Part III — Building the App (Chapters 11–15).** Routing, Pinia stores, the registration form with validation, a small mock API that runs on plain Node.js, and Vite configuration in depth.
- **Part IV — Quality (Chapters 16–20).** Unit, component, and end-to-end tests, accessibility, and performance.
- **Part V — Nuxt (Chapters 21–24).** We rebuild Gatherly with Nuxt: file-based routing, server-side data fetching, API routes, and rendering modes.
- **Part VI — Production (Chapters 25–28).** Authentication, SEO, security, deployment, CI/CD, and a capstone that finishes the live check-in dashboard.

Setting Up Your Toolchain

A Vue project needs surprisingly little: Node.js to run the tools, an editor that understands `.vue` files, and a browser extension for inspecting your app. Let us install each and say why it matters.

Node.js

You do not need Node.js to *run* a Vue app in the browser, but you need it for everything around the app: the package manager (npm), the Vite development server, the build, the test runners, and later the Nuxt server. Install a **current Node.js LTS release (20 or newer)** from the official Node.js website or through a version manager such as nvm (macOS and Linux) or nvm-windows. LTS stands for long-term support; these releases are the stable choice for tooling. Vite and Nuxt each document the minimum Node version they require, and those minimums move over time, so check the docs for the versions you install if something refuses to start.

Open a terminal and confirm that both commands work:

Terminal

```
node --version
npm --version
```

You should see two version numbers. If the first one starts with a number lower than 20, upgrade before you go on.

An editor that understands Vue

Any editor can open a `.vue` file, but you want one that understands the template, the script, and the styles together: syntax highlighting inside the template, autocompletion for component names and props, type errors in the template, and "go to definition" across files. We use Visual Studio Code with the **Vue (Official)** extension, which was formerly known as Volar. Install VS Code, open the Extensions view, search for "Vue (Official)", and install it. Other editors (WebStorm, Neovim, and others) have Vue support too; the concepts in this book do not depend on your editor.

Vue DevTools

Vue DevTools lets you inspect a running Vue app: the component tree, each component's state and props, Pinia stores, router information, and a timeline of events. It is available as a browser extension for Chromium-based browsers and Firefox, and as a Vite plugin (`vite-plugin-vue-devtools`) that adds the panel directly to your page during development. Recent versions of create-vue can add the Vite plugin for you, so you may not need the extension at all; install the browser extension too if you prefer to have it in the browser's own developer tools. You will use DevTools in almost every chapter to see what Vue sees.

Creating gatherly-web

Now we create the project. The official scaffolding tool is **create-vue**, and you run it through npm. Open a terminal in the folder where you keep your projects and run:

Terminal

```
npm create vue@latest gatherly-web
```

The `@latest` part asks npm for the newest version of the tool, so you always start from current defaults. The tool asks a series of questions. The exact wording and order depend on the create-vue version (recent versions present some of them as a checklist of features), but you will be asked about the following options. Here is what to choose and why:

Option	Choose	Why
TypeScript	Yes	Type checking for props, events, and data; taught gently from Chapter 9.
JSX support	No	We write templates, not JSX.
Vue Router	Yes	Multiple pages: event list, event detail, registration (Chapter 11).
Pinia	Yes	Shared state for events, checkout, auth, and check-in (Chapter 12).
Vitest	Yes	Unit and component tests (Chapters 16 and 17).
End-to-end testing	Playwright	Browser tests of real user flows (Chapter 18).
ESLint	Yes	Catches mistakes in templates and scripts (Chapter 15).
Prettier	Yes	Consistent formatting, so reviews focus on logic.

If the tool offers other extras, accept the Vue DevTools Vite plugin and leave anything labeled experimental unchecked. Choosing Vue Router and Pinia now does not mean we use them now: the scaffold adds the packages and some starter code, and we leave that code alone until the chapters that teach it.

When the questions are done, the tool prints the next steps. Follow them:

Terminal

```
cd gatherly-web
npm install
npm run dev
```

`npm install` downloads the dependencies listed in `package.json` into a `node_modules` folder. `npm run dev` starts the Vite development server. You should see something like this:

Output

```
  VITE vX.Y.Z  ready in 600 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
```

Vite's development server uses port **5173** by default (if that port is busy, it picks the next free one and tells you). Open `http://localhost:5173` in your browser and you should see the create-vue welcome page. Leave the server running; it watches your files and updates the browser the moment you save, usually without a full page reload. This is called hot module replacement (HMR), and it is one of the reasons the Vue development loop feels quick.

> **WARNING**
>
> If `npm create vue@latest` fails with an error about an unsupported engine or a syntax error inside the tool, your Node.js version is almost certainly too old. Upgrade to a current LTS release and run the command again in a fresh terminal. Do not try to work around it by installing older versions of the tools, because the rest of the book assumes recent versions.

A Tour of the Scaffold

Open the `gatherly-web` folder in VS Code. There are more files than you need to understand on day one, so let us focus on the four that explain how a Vue app starts, and then glance at the rest.

index.html: the page the browser loads

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <link rel="icon" href="/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Vite App</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="/src/main.ts"></script>
  </body>
</html>
```

Unlike many older build setups, Vite treats `index.html` as the entry point of the app, and it sits at the project root, not in `public/`. It contains an empty `<div id="app">`, which is where Vue will render everything, and a module script that loads `src/main.ts`. Your generated file may differ slightly (for

example, the `lang` attribute may be empty). Change the title to `Gatherly` and set `lang="en"` now; both matter for accessibility and for how the page shows up in browser tabs.

src/main.ts: creating and mounting the app

src/main.ts

```
import './assets/main.css'

import { createApp } from 'vue'
import { createPinia } from 'pinia'

import App from './App.vue'
import router from './router'

const app = createApp(App)

app.use(createPinia())
app.use(router)

app.mount('#app')
```

This short file is the whole startup sequence, and you can read it top to bottom:

1. Import the global stylesheet. Vite lets you import CSS from JavaScript; it injects the styles into the page.
2. `createApp(App)` creates an application instance whose root component is `App.vue`.
3. `app.use(...)` installs plugins. Pinia (state management) and the router are both plugins. Installing them makes their features available to every component in the app.
4. `app.mount('#app')` finds the element with id `app` in `index.html`, renders the root component, and puts the result inside it.

The generated file may be laid out a little differently, but these calls are the core. You will see the same chain written in one line in many docs and examples: `createApp(App).use(router).use(pinia).mount('#app')`. It does the same thing. The order matters in one way: plugins must be installed with `use` *before* `mount`, because mounting renders the components, and the components expect the plugins to be there already.

src/App.vue: the root component

`App.vue` is the root of the component tree. In the generated project it shows a header, some links, and a `<RouterView />` where routed pages appear. We replace it entirely in the next section. Every `.vue` file in this book follows the same block order: `<script setup lang="ts">` first, then `<template>`, then an optional `<style scoped>`. Putting the script first means you read the data before the markup that uses it.

vite.config.ts: the build tool's settings

vite.config.ts

```
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'
import vueDevTools from 'vite-plugin-vue-devtools'

// https://vite.dev/config/
export default defineConfig({
  plugins: [
    vue(),
    vueDevTools(),
  ],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url)),
    },
  },
})
```

Vite is the build tool and development server. On its own it understands JavaScript, TypeScript, and CSS; the `@vitejs/plugin-vue` plugin teaches it to compile `.vue` files. The DevTools plugin is present only if you accepted it during setup. The `alias` section lets you write `import X from '@/components/X.vue'` instead of long relative paths such as `'../../components/X.vue'`; `@` always means the `src` folder. Chapter 15 returns to this file to add a proxy for our API, environment variables, and build options.

Your First Gatherly Screen

Time to make the project ours. We will replace the welcome page with a small Gatherly screen that shows the title of our flagship event and a button that counts how many people have said they are interested. It is tiny, but it uses every part of an SFC and shows reactivity working.

First, replace the contents of the global stylesheet. The generated `main.css` contains layout rules for the welcome page that would fight with our markup. We start almost from nothing; Chapter 10 builds the full emerald and slate Gatherly theme with design tokens.

src/assets/main.css

```
*,
*::before,
*::after {
  box-sizing: border-box;
}

body {
  margin: 0;
  font-family: system-ui, -apple-system, 'Segoe UI', Roboto, sans-serif;
```

```
  line-height: 1.5;
  color: #0f172a;
  background: #f8fafc;
}
```

The scaffold may also import a `base.css` file from `main.css`; removing that import line is fine, because the rules above replace it. Now open `src/App.vue`, delete everything in it, and type the following. Typing it rather than pasting helps the syntax stick.

src/App.vue

```
<script setup lang="ts">
import { ref } from 'vue'

const eventTitle = 'Frontend Horizons 2027'
const venue = 'Riverside Congress Hall, Porto'
const interested = ref(0)

function markInterested() {
  interested.value++
}
</script>

<template>
  <main class="hello">
    <p class="eyebrow">Gatherly</p>
    <h1>{{ eventTitle }}</h1>
    <p>{{ venue }}</p>
    <button type="button" @click="markInterested">
      Interested ({{ interested }})
    </button>
    <p v-if="interested > 0">
      {{ interested }} {{ interested === 1 ? 'person is' : 'people are' }} interested.
    </p>
  </main>
</template>

<style scoped>
.hello {
  max-width: 40rem;
  margin: 4rem auto;
  padding: 0 1rem;
}

.eyebrow {
  margin: 0;
  color: #047857;
  font-weight: 700;
  text-transform: uppercase;
  letter-spacing: 0.08em;
}

button {
  padding: 0.5rem 1rem;
  border: 0;
```