Andreas Dormann

Ionic 9

Create awesome and AI-able apps for any platform

D&D Verlag

Ionic 9

Create awesome and AI-able apps for any platform

Andreas Dormann

D&D Verlag Bonn, Germany
www.dunddverlag.de
ISBN (eBook): 978-3-945102-62-6
ISBN (Paperback): 978-3-945102-63-3
ISBN (Hardcover): 978-3-945102-64-0

Book & Cover Design:
multimedia & more

1st Edition: October 2026

For Anna and our children

Table of Contents

Preface

How did I come to write this book? . . . 1
Why there was no “Ionic 8” book . . . 1
Why there is an “Ionic 9” book after all . . . 1
What this book covers . . . 2
I would like to thank... . . . 3

1 Introduction

1.1 The app you will build . . . 5
1.2 The idea behind Ionic . . . 6
1.3 The way through this book . . . 8
1.4 What to install . . . 9
1.5 The code for this book . . . 11

2 Angular Essentials

2.1 What is Angular? . . . 13
2.2 Standalone Components . . . 15
2.3 Signals . . . 19
2.4 Dependency Injection with inject() . . . 22
2.5 Lifecycle Hooks . . . 24
2.6 Data Binding . . . 29
2.7 Pipes . . . 30
2.8 Built-in control flow . . . 32
2.9 Routing and Loading . . . 36
2.10 Observables . . . 40
2.11 Promises and async/await . . . 43
2.12 Signals, Observables and Promises — which one when . . . 46
Summary . . . 49

3 The first app

3.1 Creating an app . . . 51
3.2 Running it . . . 54
3.3 The first change, without a reload . . . 56
3.4 What the CLI generated . . . 57
3.5 What a fresh Ionic app is really like . . . 62
3.6 BoB Tours — the app for this book . . . 67
Summary . . . 75

4 Navigation

4.1 Have a plan . . . 77
4.2 Generate the pages . . . 79
4.3 Routes, and the outlet they land in . . . 81
4.4 The shell around the pages . . . 83
4.5 Forward, back, and the stack . . . 85
4.6 The page you left is still there . . . 93
Summary . . . 96

5 Services & Data

5.1 A server to talk to . . . 97
5.2 From a signal to a request . . . 99
5.3 Loading, error, data . . . 103
5.4 The wire is not your model . . . 105
5.5 Where data sleeps . . . 106
Summary . . . 107

6 UI Components

6.1 Introduction . . . 109
6.2 Accordion . . . 111
6.3 Action Sheet . . . 114
6.4 Alert . . . 117
6.5 Badge . . . 120
6.6 Breadcrumbs . . . 122

6.7 Buttons . . . 125
6.8 Card . . . 129
6.9 Checkbox . . . 133
6.10 Chip . . . 137
6.11 Date & Time Pickers . . . 140
6.12 Floating Action Button . . . 144
6.13 Grid . . . 147
6.14 Icons . . . 150
6.15 Inputs . . . 153
6.16 List . . . 158
6.17 Menu . . . 162
6.18 Modal . . . 165
6.19 Popover . . . 170
6.20 Progress Indicators . . . 173
6.21 Radio . . . 178
6.22 Range . . . 181
6.23 Refresher . . . 187
6.24 Searchbar . . . 191
6.25 Segment . . . 196
6.26 Select . . . 200
6.27 Toggle . . . 206
6.28 Toolbar . . . 209
6.29 Home . . . 214
Summary . . . 219

7 Form validation

7.1 A form around the request . . . 221
7.2 Fields get rules . . . 224
7.3 A component that does not fit by itself . . . 228
7.4 Fields that depend on each other . . . 230
7.5 A half-written request must not vanish . . . 232
7.6 Send . . . 235

7.7 Success is a moment . . . 238
7.8 The job is done . . . 240
7.9 The right container for each control . . . 242
Summary . . . 248

8 Theming, Styling, Customizing

8.1 The app in the shop window . . . 251
8.2 One color, and what it changes . . . 252
8.3 Two looks, one setting . . . 255
8.4 Where a style lives . . . 258
8.5 Ionic's utility classes . . . 261
8.6 A color of your own . . . 263
8.7 Inside a component . . . 267
8.8 Type . . . 271
8.9 The logo . . . 275
8.10 Motion . . . 279
8.11 Dark, and who decides . . . 283
8.12 The large screen . . . 290
Summary . . . 294

9 Accessibility

9.1 Where the framework stops . . . 297
9.2 Seeing what a screen reader sees . . . 299
9.3 Structure: headings and landmarks . . . 304
9.4 Keyboard and pointer . . . 307
9.5 Text size and zoom . . . 312
9.6 Contrast, and who asks for it . . . 315
9.7 Motion, for those who asked for less . . . 319
9.8 Messages that arrive on their own . . . 321
9.9 Checking it stays that way . . . 324
Summary . . . 326

10 Native functionality with Capacitor

10.1 The bridge that was there all along . . . 329
10.2 An iOS project . . . 332
10.3 Live reload on the device . . . 336
10.4 An Android project . . . 339
10.5 The safe area and the system bars . . . 343
10.6 Remembering things . . . 348
10.7 Share, and the FAB unfolds . . . 354
10.8 Where the tour begins . . . 358
10.9 Without a connection . . . 363
Summary . . . 367

11 Debugging & Testing

11.1 Where to look first . . . 369
11.2 The browser's DevTools on BoB Tours . . . 370
11.3 Angular DevTools . . . 375
11.4 The app in its shell . . . 379
11.5 The first test — asking the form . . . 382
11.6 Ionic and Signal Forms together . . . 386
11.7 The submit boundary . . . 388
11.8 The HTTP boundary . . . 391
11.9 Now break something . . . 394
11.10 Checking it stays that way, automatically . . . 396
Summary . . . 400

12 Build, Deploy & Publish

12.1 What is left to do . . . 403
12.2 The build, now that the app is whole . . . 406
12.3 Where the API lives . . . 407
12.4 Which number the store reads . . . 412
12.5 Icon and launch screen . . . 414
12.6 What Apple wants declared . . . 418

12.7 One app, two languages . . . 421
12.8 A signed Android artifact . . . 430
12.9 The iOS archive . . . 434
12.10 The same app, as a PWA . . . 436
12.11 A build pipeline with GitHub Actions . . . 440
12.12 Live updates after Appflow . . . 443
Summary . . . 446

B1 Ionic with or without frameworks

B1.1 What changes if you move . . . 449
B1.2 An app with no framework at all . . . 451
B1.3 Who pays for the work . . . 456
B1.4 What you can plan on . . . 457
B1.5 What your team brings along . . . 461
B1.6 Beyond the app . . . 463
B1.7 Two portraits . . . 465
B1.8 The decision, and when it goes the other way . . . 466
Summary . . . 470

B2 Ionic and AI

B2.1 Two questions, kept apart . . . 471
B2.2 The key stays on the server . . . 473
B2.3 The form assistant . . . 482
B2.4 What it costs, and what the stores want to know . . . 493
B2.5 A rule, the phone, or the cloud . . . 496
B2.6 What an assistant builds, and what it needs to know . . . 499
B2.7 Two assistants, two roles . . . 508
B2.8 When nobody is watching . . . 514
B2.9 When it pays off . . . 525
No summary . . . 529

Preface

How did I come to write this book?

"Ionic 9" is my sixth book on the Ionic framework, the toolkit for building apps for Android, iOS, the web and the desktop from a single code base.

It all began with a search. Back in the days of Ionic 3 I went looking for a good book on the subject and found a handful of outdated tutorials and a lot of silence. So I did what stubborn people do: I wrote the book I had been looking for myself. Then I wrote the next one. And the one after that.

Why there was no "Ionic 8" book

Ionic 8 came and went without a book from me. That was not an accident.

I thought nobody would read it. Around the time Ionic 8 appeared, AI assistants started writing code that actually ran, and soon the whole thing had a name: vibe coding. Describe what you want, accept what comes back, ship it. I watched this and came to a sobering conclusion: in a world like that, who would ever again sit down with five hundred pages on how to build apps properly?

I did not feel like it. I had written five Ionic books. The sixth one would have been the same journey through the same framework, with little more than a new version number. My enthusiasm for that was about as big as a squirrel's for a nut-free winter.

I wanted to write something completely different. A novel. No components, no build errors, no store review. Just a story. So I gave it the time Ionic 8 did not get.

Why there is an "Ionic 9" book after all

Because I was wrong, or at least not entirely right. The first wave of enthusiasm for vibe coding is giving way to a more sober view, and I am not the only

one who sees it that way. An assistant produces working code in seconds. Whether you understand that code, can maintain it, make it accessible and get it through a store review is a different question, and it still takes your own judgment to answer it. Good programming still has to be learned. AI makes you faster at what you already know; it does not replace knowing it. That is exactly why this book has a bonus chapter on developing *with* AI, not just a chapter on putting AI into your app.

Because Ionic 9 made me want to write again. An Angular app on Ionic 9 runs without Zone.js by default and is built from standalone components, and together with Angular's signals and Signal Forms the code has become clearer than in any of my books before. Add the European Accessibility Act, which has turned accessibility from a nice extra into a legal requirement for many apps and services offered in the EU, and a build and publishing workflow that had to be rebuilt without Ionic's own cloud services. That is not the same book with a new version number. That is a new book, and I wanted to write it.

Because the novel is finished. It now lies on the desks of literary agents and publishers, waiting to be discovered. Waiting is not my strong suit, so I spent the time with Ionic 9.

What this book covers

The book follows one app, "BoB Tours", from the idea behind Ionic to its release in Google Play and Apple's App Store. Every listing is written against Ionic 9 and Angular 22 and uses standalone components, the new control flow, signals and `inject()`. Chapter 1 maps out that journey; these are the subjects I care about most:

- forms built on Signal Forms,
- a chapter of its own on accessibility,
- Capacitor as the foundation for everything native,
- tests written with Vitest,
- a build pipeline that runs on GitHub Actions,
- and two bonus chapters: one that helps you choose a framework for Ionic, or none at all, and one on AI, both in your app and while you develop it.

I would like to thank...

... Ben Sperry and Max Lynch, who set out in 2012 to build a framework for hybrid apps and could hardly have guessed that it would one day fill six of my books.

... the team at GFU Cyrus AG in Cologne, Germany, which has hosted my Ionic seminars for more than ten years.

... my friend Harald for his delight in talking shop about software development (which, as an IT entrepreneur, he ought to know a thing or two about), and his wife Soledad, whose city tours in and around Bonn inspired the BoB Tours app.

... the universities, colleges and schools around the world that use my books in their courses.

... Claude and ChatGPT for their excellent assistance with the code and the copyediting of this book. Yes, really. They were fast, tireless and, just occasionally, very confidently wrong, which is the best argument for this book I could have wished for.

... the readers of my books for their feedback, which kept me going, and which eventually brought me back.

... my loved ones, who first endured Ionic, then the novel, and then Ionic again. Their patience deserves a standing ovation. Raise a glass to them!

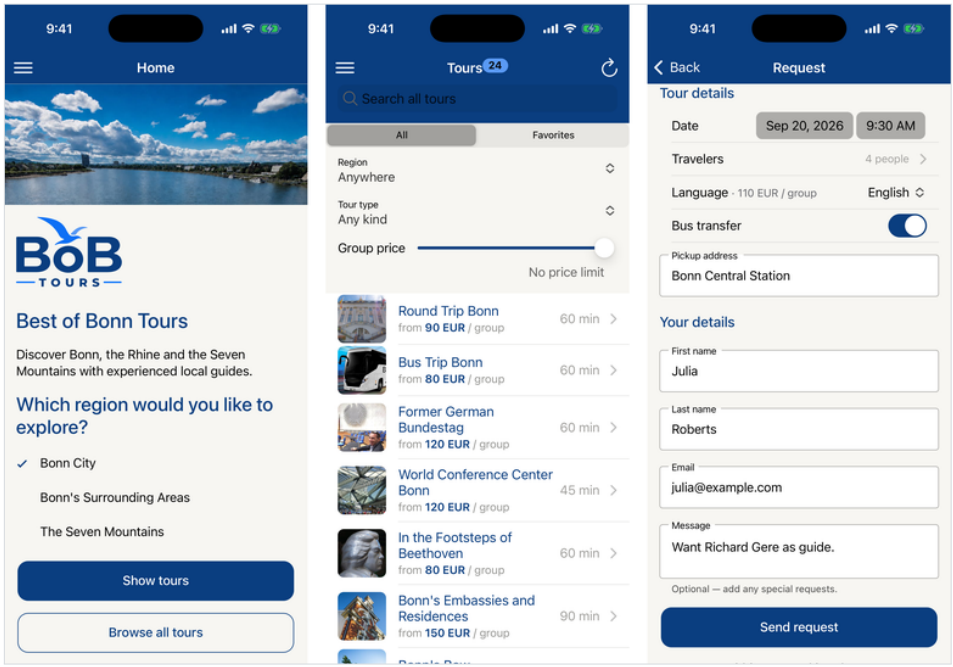
Bonn, October 2026
Andreas Dormann

1 Introduction

1.1 The app you will build

Before the first command, a fair question: what will you be holding when you are through? The answer is an app, and it is quicker to show than to describe.

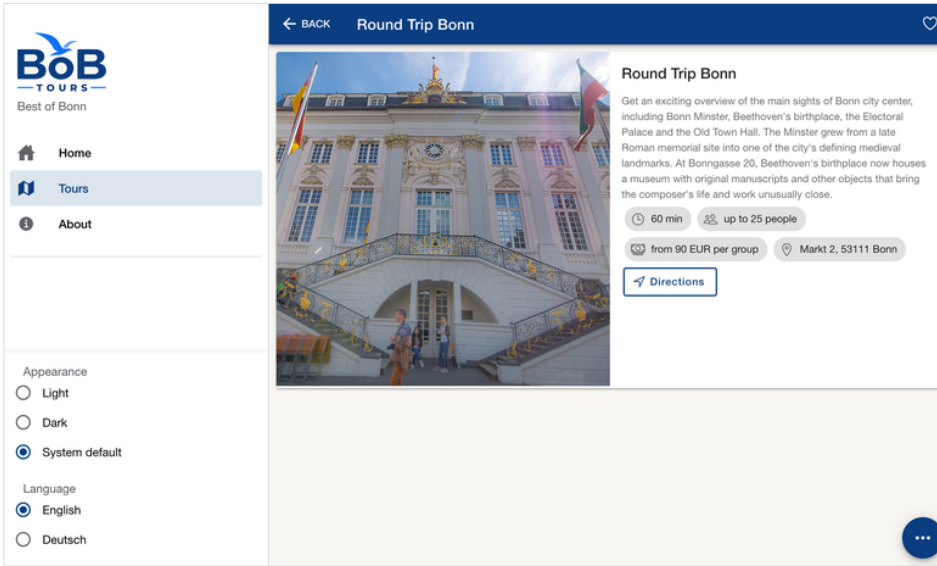
BoB Tours is the app of Best of Bonn Tours, a company that takes visitors through the city of Bonn, along the Rhine and into the hills beyond it, on foot, by bike, by bus and by boat. The app says it plainly on its own About page: *BoB Tours is fictional. The beauty of Bonn is not.* Forgive me, dear reader: on this point I am, of course, perfectly objective. I know my city, and I love it. ;-)



Someone planning a day out opens the app and is asked which region they would like to explore. The catalog shows what is on offer there, and it can be searched, narrowed to a kind of tour and to what a group is willing to pay, or switched to the tours already remembered with a heart. Each tour has a page of its own, with a photograph, what it costs and where it starts. And when

one looks right, a form sends a request: which day, how many people, the language the guide should speak, and whether a bus should collect the group.

The pictures above show that path on a phone. Widen the window and the same app arranges itself for the room it has: the menu that waited behind a button now stands beside the page, and the tour sets its photograph and its details next to each other.



You will build all of it, starting from the first command. It runs in a browser, in the iOS Simulator and in the Android Emulator, and by the end of chapter 12 it has become something that can be installed: a signed Android package, and a web app that can be added to a home screen. An iOS build for distribution needs a team enrolled in Apple's developer program, and putting the app into a store is where this book stops. Chapter 12 says why.

The phone and the wide window above are not two apps. They are the same code, unchanged, in two of the places it runs.

1.2 The idea behind Ionic

The same code in more than one place sounds like a trick. It has a plain shape: Ionic provides the controls, Angular arranges the app around them, and what comes out can be delivered in more than one way.

Start with the layer that makes Ionic Ionic. The toolbar, the list and the fields of the request form are not drawings of controls. They are elements, the way `<p>` and `<button>` are, except that Ionic defines them: a piece of JavaScript registers `<ion-button>` with the browser as a custom element, and from then on the browser creates it wherever the tag appears, like any element it was born knowing. Custom elements are one of the standards behind what are called web components. They belong to the web platform rather than to any framework, so the controls are not tied to the one a project happens to use. They also bring two looks along, one modeled on iOS and one on Material Design, Google's design language. The phones in the previous section and the wide window beside them show the same components in those two looks; which one appears, and how you decide that yourself, belongs to chapter 8, after chapter 6 has gone through the components one by one.

Components are parts, and an app is more than its parts: pages, the data behind them, the way from one page to the next. Arranging that is the work of a framework, and in this book the framework is Angular. It is not the only choice: Ionic officially supports React and Vue as well, and its components run with no framework at all. The first bonus chapter weighs what a different choice would change. Ionic provides Angular bindings, so its components fit into Angular templates, take bound data and report what the user did through events. Chapter 2 covers what the book needs from Angular.

Angular and Ionic are written in TypeScript, and so is the code behind every page in this book. TypeScript is JavaScript with types added: you state that a price is a number and that a tour has a title, and the compiler checks every use against that before anything runs. The types last only until the build. What reaches the browser is plain JavaScript.

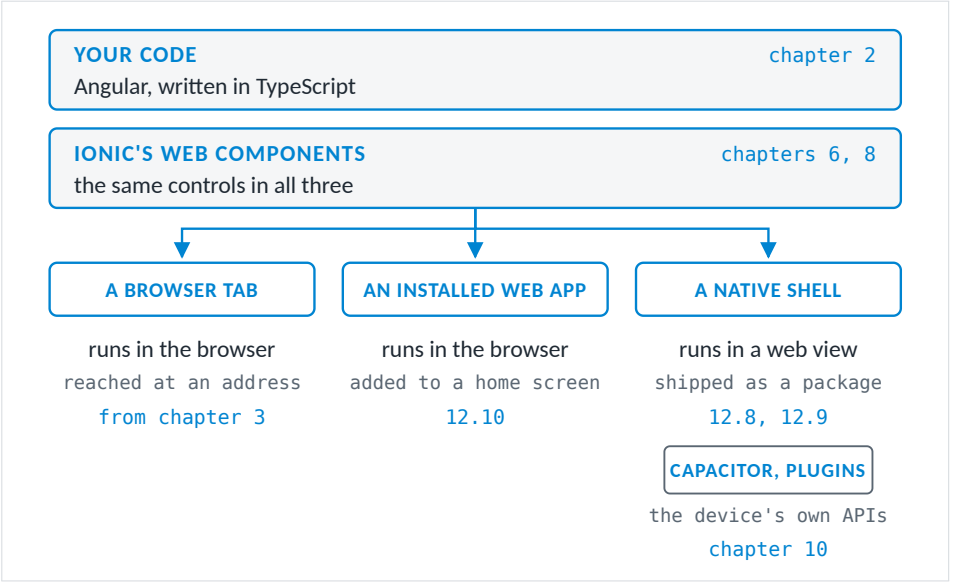
In every form the app takes, its interface and your code run in a web engine. What changes is where that engine lives and what stands around it.

The simplest place is a browser tab. The app is served at an address and opened like any web page, and that is how you will see it from chapter 3 on, every time you change something.

The second is a web app that the browser installs. Given a few extra files, a site can be added to a home screen, where it gets an icon and opens in a window of its own, without an address bar. The browser is still the one running it. Section 12.10 does this to BoB Tours.

The third is a native shell. Capacitor creates an Xcode project and an Android

Studio project whose screen is a web view, the browser engine the operating system lends to apps, and loads your build into it. Plugins carry calls from your code to the device's own interfaces and back, for things like the share sheet or settings that survive a restart. Chapter 10 builds the shell.



The last two are easy to confuse, and they differ in the way they usually reach people. An installed web app comes from a web server: anyone who opens its address can install it, it can use what the browser offers, and a change is deployed to the server and reaches installed copies when the browser picks up the new version. An app in a native shell bundles the web code into a package, which usually travels through an app store; its plugins can reach what the web does not offer, or offers differently, and a change to the bundled code usually means a new package. Neither is a lesser version of the other, and BoB Tours becomes both.

1.3 The way through this book

The table of contents tells you what is in this book. It does not tell you where to begin, and that depends less on the book than on what you bring to it.

What you need is HTML, CSS and JavaScript: reading a page's markup, knowing why one style rule wins over another, following a function that works through an array. What you do not need is Angular, TypeScript or any experi-

ence building apps. Those are what the book is for.

Angular arrives in chapter 2, and that chapter is written to be returned to rather than read straight through. If you already know Angular, start with chapter 3. Wherever a later chapter relies on something from chapter 2, it says so and names the section, so the foundation is waiting at the point where you first stand on it.

From there on, the book builds one app and never starts over. Chapter 3 creates BoB Tours, and every chapter up to chapter 12 picks it up where the previous one put it down. Chapters 3 to 5 lay the frame: a project, pages to move between, data from a server. Chapters 6 to 9 are about the surface, what a user touches, fills in and looks at, and whether it still works for someone who perceives or operates it differently from you. Chapter 10 takes the app out of the browser and onto a device, and chapters 11 and 12 give it what a finished project needs before somebody else can install it.

Chapter 6 reads differently from the rest. Most of its sections are named after a single component and stand in alphabetical order, so it also works as a catalog: when a modal is the question, open it at *Modal*, the way nobody reads a dictionary from *aardvark* onward. It is still part of the build. Each of those sections leaves the app a step further along and expects what the sections before it left behind.

The two bonus chapters come after the build and relate to it in different ways. The first stands beside it: it helps with a decision and leaves the app as it is. The second continues it, and its first half calls a paid service; section B2.1 says what that takes before you need it.

You can also join later than chapter 3. The book provides the project as it stands along the way, from chapter 6 on after every section, and section 1.5 shows how to get it. What the code cannot bring along is what the chapters before it explained. A chapter you enter in the middle leans on them, and its references back tell you where to look.

1.4 What to install

Chapter 3 begins with a command, and a few programs have to be on your machine before it can work. Everything else, including the tools for iOS and Android, arrives with the chapter that needs it.

Node.js comes first, because everything else runs on it. Install the current

LTS release, the one with long-term support; this book was written against Node 26 and also checked on Node 24. The download page offers a version manager alongside the installer, and that is the one to choose. A version manager keeps Node, and the tools you install globally with npm, in your own user folder, so installing them does not need administrator rights.

- nodejs.org/en/download

Node brings npm along, and npm installs the **Ionic CLI**, with the first command in the book:

```
npm install -g @ionic/cli
```

The `-g` installs it globally, so `ionic` works in every folder rather than in a single project.

Git follows, and it is not optional. `ionic start` creates a repository for your new project and records a first commit in it, and chapter 3 counts on both. If Git is missing, the project arrives without a repository; if Git does not know who you are, the repository arrives without its first commit. Either way the command ends with a cheerful success message. So tell Git your name and address once, right after installing it:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

- git-scm.com/downloads

The commands in this book are written for a Unix shell, the terminal on macOS and Linux. On Windows, use Git Bash, which the Git installer puts on your machine anyway.

Chrome is the browser the book opens whenever it looks under the hood of the app: its developer tools, and the audits that run inside them. Other current browsers show the app too; the instructions for looking under the hood are written for Chrome.

- google.com/chrome

Last, an **editor**. The book does not depend on any particular one. It names Visual Studio Code in a single place, where the editor can start the browser for the debugger. If you use it, one extension is worth adding on purpose: the Angular Language Service, from the Angular team, which checks your templates while you type them, the way the compiler does later. The editor will also ask, the first time you open your project, whether to install an extension

the project recommends. Accept or decline as you like; nothing in this book needs it.

- code.visualstudio.com
- marketplace.visualstudio.com/items?itemName=Angular.ng-template

Beyond this book, two more extensions earn their place in almost any Ionic project. ESLint shows in the editor what the project's lint rules object to, and a new Ionic project comes with those rules already set up. Prettier formats the code the way a project has agreed on, once the project has a Prettier configuration to agree on. For anything that writes code for you, prefer extensions from the people who make the tool. Snippet collections and extension packs age faster than the frameworks they were made for, and what they write is code from the day they were last updated.

- marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint
- marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode

Xcode for iOS and Android Studio for Android are the kind of download you start before lunch, and neither is needed before chapter 10, which says what each of them wants.

1.5 The code for this book

The code of BoB Tours lives in a public repository, and it holds more than the finished app. It keeps the app as it stands at each point of the way, so you can compare your own code with it or pick it up and carry on from there:

- ionic.andreas-dormann.de/ionic-9-book-code

The book itself is not in there, only the app and what it needs.

I recommend cloning the repository once, into a folder of its own next to the `bob-tours` you create yourself in chapter 3. That way the two never get in each other's way:

```
git clone https://github.com/dormanna/ionic-9-angular-book
cd ionic-9-angular-book
git checkout 6.8
npm install
```

Git answers the checkout with a few lines about a *detached HEAD*. That is Git's way of saying that you are now looking at a stored state rather than working

on a branch, which is exactly what you asked for. The tour photographs are downloaded with the clone; switching to another state does not fetch them again. What a state does not contain is `node_modules`, and the packages the app depends on change along the way, so run `npm install` again after every switch.

If you would rather not use Git for this, the repository page offers every state as a ZIP file under its tags. Unpack it and run `npm install` in the folder.

The tags follow the book's section numbers: `6.8` marks the app at the end of section 6.8, so to begin section 6.9 with the book's code, check out `6.8`. Chapters 3 to 5 were built in one go and have fewer tags: `3.1` is the project exactly as `ionic start` leaves it, and `3.6`, `4.6` and `5.5` mark the ends of those three chapters. From chapter 6 on, and in both bonus chapters, every section has a tag; where a section changes no code, its tag marks the same state as the one before.

A state shows the app at that point in the book, not at the end of it. Where a section tries something out that the next one takes away again, its state still has it, and the next one's does not. That is not a lost commit; the section in question says so.

Some files come with the repository because nobody should have to type them. The server in `api/`, which chapter 5 starts, arrives with `5.5`. The photographs it serves arrive with section 6.8. The complete German translation from section 12.7 is in `12.7`, where the section itself marks only a few texts. And the page from section B1.2, built without any framework, lies in `no-framework/` from `B1.2` on.

The code is yours to use under the MIT license. The photographs, the logo and the icons are not part of that: the photographs are mine, taken in a city I have already admitted I am not objective about. The two framework logos in section B1.7 are not mine either: React's is a trademark of Meta Platforms, Inc., and Vue's was designed by Evan You and is used under the CC BY-NC-SA 4.0 license. If you find a mistake in the book or in the code, report it as an issue in the repository; confirmed ones are collected in its errata file.

Chapter 2 is in the book

In the full book, this is where chapter 2, *Angular Essentials*, begins, on page 13. It is the ground BoB Tours stands on: the Angular the rest of the book builds with.

Standalone components and what they import. Signals for the state of a page. Dependency injection with `inject()`. `@if` and `@for` in place of the directives older tutorials still show. Routes whose pages stay out of the first download. And the question that trips up nearly everyone who comes to Angular with some experience from elsewhere: when a signal, when an Observable, when a Promise. The chapter answers it with one table and three questions, asked in a fixed order.

It is written to be returned to rather than read straight through. So you can carry on without it: chapter 3 starts on the next page, and when it relies on one of these ideas, it points back to the section that explains it. If Angular is new to you, those pointers are where the full book would catch you.

3 The first app

3.1 Creating an app

Every project begins as a directory full of files that somebody had to put there. The Ionic CLI puts them there for you, and the whole ceremony is one command:

```
ionic start bob-tours sidemenu --type=angular-standalone \
  --package-id=de.dormann.bobtours
```

Every part of it is doing something. `bob-tours` becomes the folder and the project name. `sidemenu` is the template — a starting point with a menu that slides in from the left, which is what our app wants. `--type` decides which flavor of Angular project you get, which sounds like a detail and is not; it has its own subheading further down. And `--package-id` is the one most people leave out and later wish they had not: the app identifier the native platforms and the app stores use to tell your app apart from every other app in the world, by convention a domain you control, reversed. Once an app has been published under an identifier, that identifier *is* the app — a later change produces a new app rather than an update, with a new listing and none of the reviews. Left unsaid, the CLI writes `io.ionic.starter`, a placeholder with a talent for surviving into places it should not.

The backslash at the end of the first line only carries the command on to the next, so that it fits on the page. The terminal on macOS and Linux understands it, and so does Git Bash on Windows, the shell section 1.4 recommends there.

Then the CLI does a great deal of work and, in our run, asks a single question:

```
✔ Preparing directory ./bob-tours in 475.83µs
✔ Downloading and extracting sidemenu starter in 154.88ms
> ionic integrations enable capacitor --quiet --
  bob-tours de.dormann.bobtours
> npm i --save -E @capacitor/core@latest
added 600 packages, and audited 601 packages in 30s
> npm i -D -E @capacitor/cli@latest
added 70 packages, and audited 671 packages in 4s
> capacitor init bob-tours de.dormann.bobtours --web-dir www
✔ Creating capacitor.config.ts in ./bob-tours in 697.38µs
[OK] Integration capacitor added!
```

```
> npm i
> git init
? Create free Ionic account? (y/N) n
> git add -A
> git commit -m "Initial commit" --no-gpg-sign
[main (root-commit) 16889e3] Initial commit
 32 files changed, 12679 insertions(+)
```

Answer `n`. The account is free and it is genuinely optional; nothing in this book needs it, and you can create one later if you ever want to.

More happened there than you asked for, and the rest is worth knowing about.

Capacitor was installed and initialized. Capacitor is the bridge to the native platforms — camera, filesystem, share sheet, the App Store. You will not touch it until chapter 10, but it is wired in from the first minute, and that is deliberate on Ionic's part. An Ionic app is a web app that can be packaged as a native app without being rewritten as one.

`capacitor.config.ts` was written with the app ID you passed in. The identifier went straight through: `--package-id` on your command line, `capacitor init` two lines later, the file after that. What the CLI did choose for you is the app *name* — `bob-tours`, the folder name — and that one still wants correcting. Section 3.4 opens the file and looks at both.

A git repository was created and everything committed. If you already have a repository in mind, that first commit is easy enough to undo. If you do not, the CLI has quietly done you a favor. This one is yours; the book's own, from section 1.5, belongs in a folder of its own beside it.

npm said a lot of things in passing, including warnings about deprecated packages, a list of packages that run install scripts, and a summary of security advisories. None of it means you did something wrong. Section 3.5 takes that output apart properly, because it is more interesting than it looks.

Finally, the CLI tells you where to go next:

```
Your Ionic app is ready! Follow these next steps:
- Go to your new project: cd ./bob-tours
- Run ionic serve within the app directory to see your
  app in the browser
- Run ionic capacitor add to add a native iOS or
  Android project using Capacitor
```

Standalone is the default

Now to `--type`, and why it is written out at all.

Ionic still ships two kinds of Angular project. The module-based one has `app.module.ts`, a routing module beside every page, the bookkeeping chapter 2 described before standalone components made it unnecessary. `--type=angular-standalone` leaves all of it out. Ask for plain `--type=angular` and the CLI puts the question to you instead — *Standalone Components or NgModules?* — with standalone listed first and described, in the CLI's own words, as the default way to build with Angular. This book is standalone throughout, which is also where Angular and Ionic have arrived: Ionic 9 deprecates `IonicModule`, and the module-based type stays for codebases that were built that way. Chapter 2 names the schematic that brings those forward.

So why type the long flag at all? Because a command that states its intention keeps working when defaults change. The word `standalone` in your terminal history is a record of the architecture you asked for. What it does not do is freeze the template or anything inside it — a reader two years from now types the same command and gets a newer starter. Freezing is what `package-lock.json` and `npm ci` are for, and chapter 2 has already introduced them.

The templates on offer

`sidemenu` is one of several — `blank`, `list` and `tabs` are the others you are likely to want, and `ionic start --list` prints the current set. We take `sidemenu` because BoB Tours needs somewhere to put a growing list of destinations, and a menu that slides in from the left is the least intrusive place for it. A starter is only a shell, though, and not a commitment: the pages underneath stay independent of the navigation around them, which is what chapter 4 takes apart.

What the CLI is responsible for

Worth knowing before you wonder whom to ask when something is wrong: the CLI does less than you might think. It fetches a starter, wires up Capacitor and hands the project to Angular's own tooling, which then does the building, serving and updating. The starter templates live in a repository of their own and move on their own schedule. So the version of the CLI on your machine

says almost nothing about how current the app on your disk is — those are three different clocks, and the one that matters now is the dependency tree recorded in the project you just generated.

Which is also the thing the CLI cannot do for you: guarantee that what it fetched is current. Whatever the template held on the day it was saved is what you have, and it starts aging the moment it lands. Section 3.5 measures how old this one is rather than assuming.

More in the documentation

- ionicframework.com/docs/cli/commands/start
- capacitorjs.com/docs/config

3.2 Running it

Change into the project and start the development server:

```
cd bob-tours
ionic serve
```

A browser window opens by itself, and the terminal reports what happened:

```
> ng run app:serve --host=localhost --port=8100
[ng] ✔ Building...
[ng] Initial chunk files     | Names         | Raw size
[ng] styles.css              | styles        | 57.36 kB
[ng] main.js                 | main          | 13.67 kB
[ng]                         | Initial total | 71.03 kB
[ng] Lazy chunk files        | Names         | Raw size
[ng] folder.page-GRNFVGZQ.js | folder-page   |  5.52 kB
[ng] Application bundle generation complete. [0.918 seconds]
[INFO] Development server running!
       Local: http://localhost:8100
       Use Ctrl+C to quit this process
```

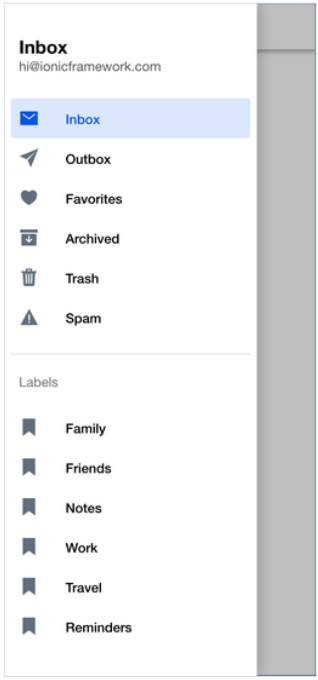
Between the building and the summary there is also a wall of text about browsers, which I have left out above and which you should not ignore: Angular warns that the project is configured for browsers outside the range it supports, and names every one of them. Nothing is broken, and section 3.5 comes back to it with the other things a fresh project says about itself on its first day.

The first line gives the game away, and the `[ng]` prefixes on most of the others

confirm it: `ionic serve` runs the Angular CLI. It picks the port, hands over, and prefixes whatever Angular says so you can tell the two apart. Most of what you would hand to Angular's own `ng serve` you can hand to `ionic serve` as well: put the options after a `--` separator and they are passed straight through.

Which raises a fair question: why use it at all? Neither answer is dramatic. It picks port **8100**, Ionic's conventional development-server port, rather than Angular's 4200. And in chapter 10 the same server gets a second job: `ionic capacitor run` with live reload points the app on your phone at it, so that a saved file reloads on the device.

For now, `npm start` will show you the app too: it is `ng serve` underneath, on Angular's 4200, and it leaves opening the browser to you. Use `ionic serve`.



That is the sidemenu starter as it comes: a menu with Inbox, Outbox, Favorites and a set of labels, and a page that tells you its own name. Nothing in it belongs to BoB Tours yet. Section 3.6 changes that.

At a narrow window the menu hides behind the button in the top left. Widen the window to 992 pixels or more and it stays open beside the content instead — one component, `ion-split-pane`, doing what a tablet layout needs without being asked twice. Chapter 4 takes that apart.

One option is worth remembering now: `ionic serve --external` prints the addresses the server is reachable on, so you can open the app on a phone on the same Wi-Fi — a quick way to see how a layout behaves on a real screen. The rest wait in `ionic serve --help`.

More in the documentation

- ionicframework.com/docs/cli/commands/serve
- ionicframework.com/docs/cli/livereload

3.3 The first change, without a reload

Leave the server running. That is the whole point of it.

Open `src/app/app.component.html` and look at the top of the menu. Line 6 holds the heading that currently says `Inbox`. Overwrite it with the name of the app — throughout this book, **bold in a listing is the part you type**:

```
<ion-list-header>BoB Tours</ion-list-header>
```

Save the file and watch the terminal:

```
[ng] ✔ Changes detected. Rebuilding...
[ng] Initial chunk files | Names | Raw size
[ng] main.js             | main  | 13.68 kB
[ng] Application bundle generation complete. [0.193 seconds]
[ng] Component update sent to client(s).
```

A fifth of a second, and the heading in the browser has changed. The browser did not reload — nothing flickered, and the page you were on is the page you are still on. Angular patched the changed template into the running application and left the rest of it standing.

The mechanism has a name: **hot module replacement**, HMR. It is worth keeping separate from *live reload*, which means the older and blunter thing — a change happens, the page reloads. Both are useful and they are not the same, and the difference shows up exactly when it costs you. When you are three screens deep in a booking flow, a reload starts the application over at the address you were on, and whatever it held only in memory — the form you were filling in — is gone; a template or stylesheet edit that HMR can handle leaves all of it standing. It does not stretch to everything. Angular hot-replaces templates and styles, not arbitrary TypeScript, and you are about to see a change that falls outside it.

The lights, and how to turn them on

Now a change of a different kind. Open `src/global.scss` and go to the very bottom:

```
/* @import "@ionic/angular/css/palettes/dark.always.css"; */
/* @import "@ionic/angular/css/palettes/dark.class.css"; */
@import '@ionic/angular/css/palettes/dark.system.css';
```

Ionic ships dark mode as a palette you import, and the three import lines are the three ways to apply it — always, on a class you set yourself, or when the

operating system asks. Two of them are commented out. The starter picks `dark.system`, which is why the app follows whatever your machine is set to.

Switch your system to dark and the app goes dark under you, without being asked and without a reload — it is a media query, and media queries do not wait.

For this book I am turning it off, and for a thoroughly unglamorous reason: dark screenshots eat printer's ink by the bucket, and your copy would arrive with pages that stick together. Comment the line out:

```
// @import '@ionic/angular/css/palettes/dark.system.css';
```

The terminal answers with a **stylesheet** update this time rather than a component one, and again without reloading the page. A third message waits for you further on: `Page reload sent to client(s)` is the server admitting it cannot patch what you changed. Touch something the whole application is built on — `main.ts`, for instance — and there is no single component to swap out, so the page reloads. Not a failure; just the moment the application starts again and whatever it held only in memory goes with it.

Your app is now light regardless of what the operating system wants, which is the right setting for following along with a printed book and not yet a decision about the app. Chapter 8 puts dark mode back, properly, along with the question of who should get to decide — the system, or the user — and the six high-contrast palettes that sit in the same folder belong to chapter 9.

More in the documentation

- ionicframework.com/docs/theming/dark-mode
- angular.dev/tools/cli/serve

3.4 What the CLI generated

The starter is a directory of files, and you will spend real time in a handful of them. This section is the map: what each part is for, which ones you will open daily, and which ones you can leave alone for the next five hundred pages.

Where the application starts

`src/main.ts` is the first code that runs. In an application built from modules this file was three lines long and the interesting work happened elsewhere. Here it is the interesting work:

```
import { bootstrapApplication }
        from '@angular/platform-browser';
import { RouteReuseStrategy, provideRouter,
         withComponentInputBinding, withPreloading,
         PreloadAllModules } from '@angular/router';
import { IonicRouteStrategy, provideIonicAngular }
        from '@ionic/angular';

import { routes } from './app/app.routes';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, {
  providers: [
    { provide: RouteReuseStrategy,
      useClass: IonicRouteStrategy },
    provideIonicAngular(),
    provideRouter(routes,
      withPreloading(PreloadAllModules),
      withComponentInputBinding()),
  ],
});
```

`bootstrapApplication()` starts the app at a single component and takes a list of providers — everything the application needs to have available, stated in one place. Three of them are worth reading now; the fourth, `withComponentInputBinding()`, waits until section 3.6, where it suddenly matters.

`provideIonicAngular()` is the one that makes this an Ionic application rather than an Angular one. It runs Ionic's own initialization — the global configuration every `ion-` component reads, with the platform detection and the default mode, iOS or Material Design — and it registers the Angular-side services: the overlay controllers, and Ionic's own binder that hands route parameters to a page's inputs inside `ion-router-outlet` — switched on by `withComponentInputBinding()`, carried out by Ionic.

It is worth taking out for a moment, just to see what it does. Comment the line out and what you get is not an error page — it is an app with the Ionic taken out of it. The split pane collapses, the toolbar and the menu never appear, and all that survives is an unstyled button in the top left corner and a line of

text in the middle of an empty page. The browser console has an opinion too:

```
ERROR RuntimeError: NG0950: Input "folder" is required but
no value is available yet.
```

Every `ion-` component in that page is still imported and still known to Angular. What is gone is the application-level scaffolding they lean on — the configuration with the platform and mode detection, and Ionic's half of the input binding: the router still offers the value, nothing accepts it, and that is what the console is complaining about. Put the line back before going on.

`provideRouter()` hands over the route table you will meet in a moment. `withPreloading(PreloadAllModules)` tells Angular to fetch the lazy parts of the app quietly in the background once the first screen is up, so a later navigation feels instant. The name is a leftover from the module era: the preloader also picks up routes written with `loadComponent`, which is every lazy route in this book. Chapter 4 goes into when that is a good idea and when it is not.

`IonicRouteStrategy` is the odd one out and worth knowing about. Angular's default behavior is to destroy a page when you navigate away from it. Ionic's routing layer does not: `ion-router-outlet` keeps the page you came from in the DOM, hidden but alive, so that state and scroll position are still there when you go back — which is what a mobile app is expected to do. `IonicRouteStrategy` is the piece that lets that model work with Angular's own router: it tells Angular not to reuse the current page when only a route parameter changes, so `folder/inbox` to `folder/outbox` is a new page on the stack rather than the same component with a new value. The arrangement has consequences for component lifecycle, and chapter 4 deals with them where they first bite.

Where it goes

`src/app/app.routes.ts` is the route table:

```
import { Routes } from '@angular/router';

export const routes: Routes = [
  {
    path: '',
    redirectTo: 'folder/inbox',
    pathMatch: 'full',
  },
  {
    path: 'folder/:id',
```

```
    loadComponent: () =>
      import('./folder/folder.page').then((m) => m.FolderPage),
  },
];
```

The empty path sends visitors to `folder/inbox`, and `folder/:id` matches anything after `folder/` and hands the value over as a parameter.

The part to notice is `loadComponent`. The `import()` inside it is not executed when the app starts — it is executed when somebody navigates to that route. The page becomes its own bundle, fetched on demand. You saw that bundle in the terminal output earlier, listed separately under **Lazy chunk files**.

The root component and the one page

`src/app/app.component.ts` is the application shell: the split pane, the menu, and the router outlet that everything else appears in. It is also where the menu entries live, as a plain array in the class — which is why changing the heading in section 3.3 was a one-line edit.

`src/app/folder/` holds the only page the starter ships, in four files: the class, its template, its styles and its test. It reads the `:id` from the route and shows it as a title. Everything BoB Tours will become grows out of this pattern, and the shape of it outlives the rebuild later in this chapter: one component, several addresses.

Styling

Both global stylesheets are registered in `angular.json` under `styles`, and they have separate jobs. `src/theme/variables.scss` is where you overrule Ionic's global CSS custom properties, starting with the app's primary color, and `src/global.scss` holds styles that apply everywhere — you have already been in it once, at the bottom, where the palettes are. Chapter 8 makes the distinction matter. Anything else you write goes in a component's own `.scss` file, where it stays scoped to that component.

The configuration files

File	What it decides
`angular.json`	How the app is built and served
`package.json`	Which dependencies, at which versions

File	What it decides
`package-lock.json`	The exact tree that was installed
`ionic.config.json`	Project name and type, for the Ionic CLI
`capacitor.config.ts`	App ID, app name, where the web build lands
`tsconfig*.json`	TypeScript settings, strict mode among them

One of these has already been half settled on the command line. Open `capacitor.config.ts` and give `appName` a capital letter and a space:

```
const config: CapacitorConfig = {
  appId: 'de.dormann.bobtours',
  appName: 'BoB Tours',
  webDir: 'www'
};
```

The app ID is already right, because `--package-id` put it there. `appName` came from the CLI, which used the folder name, and it is the label under the icon on the home screen — not permanent, but worth reading properly. And `webDir: 'www'` tells Capacitor where Ionic puts the web build — `www/`, not the `dist/` a plain Angular project would use.

The ones that can wait

The rest is infrastructure, and most of it can wait: the two environment files that `angular.json` swaps at build time, the test setup, linting and editor settings. Each gets its turn on the day it first matters. The browser list is the exception — section 3.5 shows why its default is worth a look today.

The tree also shows where Angular has got to. There is no `polyfills.ts` and nothing configuring Zone.js: fresh Ionic 9 projects on Angular 21 and later are zoneless by default, so the zoneless model chapter 2 described is not an option you switch on here, it is simply what you have. And `src/test-setup.ts` alongside `vitest` in `package.json` means the tests run on Vitest; chapter 11 makes proper use of it.

That is the whole tree: the files above are where you will work, and the rest quietly does its job.

More in the documentation

- ionicframework.com/docs/angular/your-first-app

3.5 What a fresh Ionic app is really like

Questions nobody asks about a project on its first day, and all of them have answers you can measure in about a minute. How big is it? How current is it? And how much of what `npm` is muttering about should worry you?

Do this now, while the app is still essentially the starter and before any real application code goes in. The numbers are more interesting while nothing in them is yours.

Every figure below is a dated measurement, taken on 21 September 2026 with Ionic CLI 7.2.1 and the starter as the previous section generated it. Yours will read differently. The procedure is what the section teaches, and it answers whenever you run it.

How big is it?

Stop the development server and build the app properly:

```
ionic build
```

That is a production build. You do not need a `--prod` flag: in this project's `angular.json` the build target sets `defaultConfiguration` to `production`, so that is what you get by asking for nothing. Ionic's CLI still accepts the flag and maps it to that same configuration, which is why you will see it in older scripts.

The last line of the summary is the one that matters:

```
                    | Initial total | 626.89 kB | 145.71 kB
```

Two numbers, and they are not two ways of saying the same thing. **626.89 kB** is the raw size of the chunks Angular counts as the initial load — the code that arrives as the app starts, not the weight of the whole `www/` directory, which also holds everything that loads later. **145.71 kB** is Angular's *estimate* of what those same chunks will weigh travelling down the wire, once a web server has compressed the JavaScript on its way out. What your users really transfer depends on that server and on what their browser already has cached, so treat it as an estimate and not a promise.

Even so, it is the more useful of the two when somebody asks how heavy your app is. Quoting the first is like giving your weight in winter clothing.

Where does it come from? Ionic ships its components as web components, and

a web component brings its own behavior with it — gestures, animations, the platform detection that makes a button look native on both iOS and Android. Your own code is a few lines so far, so the figure is framework and starter, which is worth knowing before you go hunting for a leak: most of it is the thing you came for. It is not a fixed price either. Which components you use and how they are loaded move the number — this is the measured starting point of this starter, not the floor under every Ionic app.

Why nothing warned you

Angular can fail a build for being too large. It did not fail this one, and the reason is worth knowing, because it is not "the app is small".

Open `angular.json` and look for `budgets` in the production configuration. The warning threshold for the initial bundle is **two megabytes** — the build fails at five — and at 627 kB the app is nowhere near either, so nothing is said. A workspace generated by Angular's CLI starts at **500 kB** for the warning and one megabyte for the error. The Ionic starter raised the warning fourfold on your behalf, and the error fivefold.

It is a striking choice next to `tsconfig.json`, where the same starter switches on Angular's full strict set, `strictTemplates` included. Strict about types, relaxed about size. Not a scandal — a budget is a policy rather than a law of physics — but this policy was written by somebody who has never seen your app.

My suggestion: set it deliberately, somewhere just above where you are now, and treat the warning as a conversation rather than an alarm. A budget that never fires tells you nothing. One that fires every day gets ignored, which is worse.

And when it does fire, the answer is rarely to hunt for a leak. It is to stop things arriving before they are needed — which is exactly what lazy routes and `@defer` are for. You have already seen the mechanism in chapter 2; the `folder` page in this app is already loaded that way, and you saw it listed under **Lazy chunk files** in the build output.

How current is it?

Now the more uncomfortable question. Your own version numbers and audit counts will differ from the ones printed here by the time you read this — that

is the nature of both. The procedure is the part worth keeping.

At the end of the installation, `npm` said this:

```
5 vulnerabilities (4 moderate, 1 critical)
```

A critical finding in a project that is four minutes old and has not done anything yet. That word sends people into a spiral, so let us take it apart.

Start by asking what is actually behind:

```
npm outdated
```

```
Package         Current  Wanted  Latest
angular-eslint   22.0.0  22.0.0  22.5.0
jsdom            26.1.0  26.1.0  30.1.0
typescript        6.0.3   6.0.3   7.0.2
vitest           4.0.18  4.0.18   5.0.1
```

Wanted is the newest version the range in `package.json` allows, and for Vitest, the test runner, the starter wrote `~4.0.18`: patches of 4.0 and nothing beyond, of which 4.0.18 is the last. **Latest** is the newest version published, whatever the template saved. `npm audit` names the critical finding: it is in Vitest, fixed in 4.1.11.

The column does not stop at your version line, though. For Vitest it offers 5, a new major version, and the build tooling this starter installs accepts Vitest 4 and nothing else; for TypeScript it offers 7, and Angular 22 accepts 6.0. **Latest** reports what exists, not what fits. The move that fixes the finding stays inside Vitest 4:

```
npm install -D vitest@4
```

This is not a jump to a new major version. It installs the newest Vitest 4, 4.1.11, and writes `^4.1.11` into `package.json`, so later releases of Vitest 4 arrive with a plain `npm update`. Run `npm audit` again:

```
3 moderate severity vulnerabilities
```

Five to three, the critical one gone, without leaving the version line this book is built on.

Angular itself is not on the list, and that is worth a look rather than a shrug. Its packages move with `ng update`, which also runs the migrations a new version brings. Run it without arguments and it lists what it could do. On the day of this measurement that was only `angular-eslint`, the linter; the framework arrived current. That lasts until the next Angular release, and

then Angular heads the list. This follows it without leaving version 22:

```
ng update @angular/core@22 @angular/cli@22
```

The starter has moved on since the day this book's code was generated; the next project you create may be the one that is behind.

That is the lesson of the section, and it is worth separating from the numbers: **a generated project is a snapshot, not a subscription.** It is the template as it stood on the day it was saved, and it has been aging quietly ever since — the template may well have moved on without it. Whether yours is two patches behind or ten, the first thing to do with a generated project is find out.

There is one more thing the build has been telling you since the very first `ionic serve`:

```
[ng] One or more browsers which are configured in the project's
Browserslist configuration fall outside Angular's browser
support for this version.
```

`.browserslistrc` in the project root names the oldest browsers the app is compiled for — Chrome 107, Safari 16.1 and their neighbors — and Angular 22 has since raised its own floor above every one of them. Nothing is broken; Angular is telling you that it will not promise anything about browsers that old.

The fix is a decision rather than a command, and it has an order. Start from the browsers Angular 22 supports — that is the outer edge of what the framework stands behind — then narrow it to the ones your users actually run. Narrowing is yours to do; widening is not, because naming an older browser in `.browserslistrc` does not oblige Angular to support it. If your users genuinely need something below that edge, you are holding a framework-version decision rather than a Browserslist entry. Within the edge it is the same shape of question as the bundle budget: somebody chose a default, and it was not somebody who knows your users.

Rebuild after the update and the figure has not moved. Vitest runs your tests; nothing of it travels to a browser. An Angular update may add weight or take some off, and a build-tool release can move the figure by rearranging what counts as an initial chunk at all. Either way, the size is not what you updated for. Updating is not a diet. It is maintenance.

How healthy is it?

Three findings remain, all of them moderate, and the Vitest update did not touch them — so it is worth learning to read them rather than to fear them.

Ask where they live:

```
npm ls uuid
```

```
`-- @capacitor/cli@8.5.2
  `-- xcode@3.0.1
    `-- uuid@7.0.3
```

Three levels down, and none of them is a package you asked for. `@capacitor/cli` is the only one you will find in `package.json`, put there by `ionic start` as a development dependency — the tool that builds native projects on your machine. It brings a library for reading Xcode project files, which in turn brings an old version of a package for generating identifiers.

That chain is the whole point. **This dependency is build tooling, not app runtime.** Nothing in that list reaches the runtime bundle your users download; it belongs to the tooling that produces the app — your machine, your build server, whatever environment packages the native projects. Which does not make it irrelevant — a compromised build tool is a serious problem, for you and for anyone who can reach your machine — but it is a different problem with a different urgency, and one you can explain to whoever arrives holding a security report.

`npm ls <package>` is the tool for that conversation. It answers the first question in it: how did this get in here, and who asked for it? Put the answer next to `package.json`, which lists `@capacitor/cli` under `devDependencies`, and runtime or tooling is settled. How urgent it is depends on what the flaw lets someone do, and where.

The same question sorts out the other list npm printed after every install, the one about *install scripts not yet covered by allowScripts*. A few packages run code of their own while they are being installed, and npm now names each one that nobody has approved. npm 11.19.1, measured here, still runs them; the list is there so that somebody looks. `npm install-scripts ls` prints it again, and `npm ls` shows where every entry comes from: `esbuild`, `lmdb` and the rest all arrive with `@angular/build`. Build tooling again, not app runtime.

npm's own output ends with an offer to fix everything at once, and the offer carries a `--force`. That word is doing a lot of work: it lifts npm's obligation

to stay inside the versions your project declares compatible. Here it would install `@capacitor/cli` 8.4.3, older than the one you have, and npm itself calls that a breaking change. The discipline is the one this section has been practising: update within your version line, check what is left, and be able to say why it is still there. That is a better answer than a clean report bought with a broken build.

More in the documentation

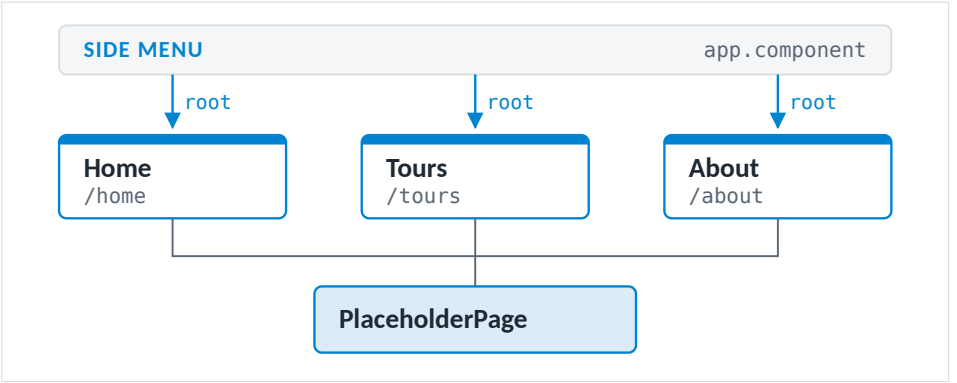
- angular.dev/reference/configs/workspace-config
- docs.npmjs.com/cli/v11/commands/npm-audit
- angular.dev/reference/versions

3.6 BoB Tours — the app for this book

Everything so far has been somebody else's app. The menu still offers Inbox and Spam, the one page it can show reports the name of a mail folder, and the shell is decorated with labels called Family and Work. It was a starting point, and it has started us.

BoB Tours is a tour app, and the initials stand for **Best of Bonn** — the city this book was written in, and one with rather more to look at than its size suggests. The app will end up showing tours worth taking, letting people book one, and remembering which ones they liked. Not today, though. Today it gets a menu with three entries and one page standing behind all three, because that is the skeleton every following chapter hangs something on.

Worth a minute before you build it. The shape of an app is a great deal easier to change on paper than in a project, and this one is small enough to draw:



Home greets, and will later show what is on. Tours is the list, and from chapter 5 it arrives from a real source rather than from you. About holds the imprint — the dullest page in the app, and reliably the first one on which you notice that a change has landed.

Nothing sits underneath them yet, which is why all three arrows run into the same box. Chapter 4 draws this picture again with a floor added: a tour gets a page of its own, and from there a booking request. Today the three names lead to the same place.

One page, for now

Delete `src/app/folder/` — the whole directory. In its place, one page under `pages/placeholder/`, and one is enough: Home, Tours and About are three names on a menu today, not three different screens. What each of them becomes is a question chapter 4 answers, with the plan in front of it. Until then a single page stands in for all three, and says so.

Create `src/app/pages/placeholder/placeholder.page.ts` and write:

```
import { Component, input } from '@angular/core';
import { IonHeader, IonToolbar, IonButtons, IonMenuButton,
         IonTitle, IonContent }
        from '@ionic/angular';

@Component({
  selector: 'app-placeholder',
  templateUrl: './placeholder.page.html',
  imports: [IonHeader, IonToolbar, IonButtons, IonMenuButton,
            IonTitle, IonContent],
})
export class PlaceholderPage {
```

```
  readonly title = input.required<string>();
}
```

A component with one line of logic in it, and it still has to say which Ionic components its template uses. That is the standalone bargain: nothing is available implicitly, so nothing is a mystery. If the template mentions `ion-title`, the `imports` array mentions `IonTitle`.

Write the template beside it, `placeholder.page.html`:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title>{{ title() }}</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <p>Nothing here yet.</p>
</ion-content>
```

`ion-menu-button` is the three-line button that opens the menu on a narrow screen. It knows to hide itself when the split pane has the menu open already, which is why you will not see it on a wide window. It belongs in the toolbar of every top-level page: that is what keeps the menu reachable from wherever the user happens to be standing.

One line in that class does the work the three separate pages would otherwise have done. `input.required()` declares a value the component cannot do without, and the template reads it as `title()` because an input is a signal. Where the value comes from is the next section.

Where they live

Replace `src/app/app.routes.ts` with a route each. All three want the same component, so the loader is worth a name of its own:

```
import { Routes } from '@angular/router';

const placeholder = () =>
  import('./pages/placeholder/placeholder.page')
    .then((m) => m.PlaceholderPage);

export const routes: Routes = [
```

```
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home',  data: { title: 'Home' },
    loadComponent: placeholder },
  { path: 'tours', data: { title: 'Tours' },
    loadComponent: placeholder },
  { path: 'about', data: { title: 'About' },
    loadComponent: placeholder },
];
```

Three paths, three titles, one component. **A route and a component are not the same thing**, and this is the shortest demonstration of it the book will get: the router decides which address exists, the component decides what an address looks like, and nothing says the two have to be counted together. Chapter 4 leans on this from its second page, where one page component serves every tour in the app.

The `data` on each route is arbitrary information travelling with it, and getting it into the component takes a switch in the router. You do not have to add that switch — it has been sitting in `src/main.ts` since the first minute of this chapter, and section 3.4 walked past it without stopping:

```
provideRouter(routes, withPreloading(PreloadAllModules),
              withComponentInputBinding()),
```

This is the moment it earns its place in the file. It is the switch chapter 2 described: with it on, anything attached to a route — a path parameter, a query parameter, static data like this — arrives at a component input of the same name. The route says `title: 'Tours'`, the class declares `title`, and nothing else has to be arranged.

Build the app now and the effect shows up in the output as a single entry under **Lazy chunk files**:

```
chunk-ddWihDvV.js | placeholder-page | 798 bytes | 798 bytes
```

One chunk, not three, because all three routes point at the same lazy entry point: the build has one component to emit, not three copies of it. The first navigation fetches it; choosing Tours or About afterwards downloads nothing at all — measured in the browser's network panel, where the second and third menu taps produce no request. How a build divides code into chunks is otherwise the bundler's business, and not something to read a rule into.

Which leaves `PreloadAllModules` with little to do for the moment. It is still switched on in `main.ts`, and there is still only one lazy chunk to fetch — the one the first navigation has already taken. Chapter 4 gives it something more

interesting, and asks whether it should have it.

The menu

Now the shell. `src/app/app.component.ts` loses the mail folders and gains this:

```
interface MenuEntry {
  title: string;
  url: string;
  icon: string;
}
```

and, inside the class, add the menu itself:

```
readonly pages: MenuEntry[] = [
  { title: 'Home', url: '/home', icon: 'home' },
  { title: 'Tours', url: '/tours', icon: 'map' },
  { title: 'About', url: '/about', icon: 'information' },
];
```

The icons need registering before they can be drawn, in the same file:

```
import { addIcons } from 'ionicons';
import { homeOutline, homeSharp, mapOutline, mapSharp,
         informationOutline, informationSharp }
        from 'ionicons/icons';

constructor() {
  addIcons({
    homeOutline, homeSharp,
    mapOutline, mapSharp,
    informationOutline, informationSharp,
  });
}
```

The template still walks the starter's array under the starter's name. Change the loop in the same file, and the name it binds along with it:

```
@for (page of pages; track page.url) {
  <ion-menu-toggle auto-hide="false">
    <ion-item
      routerDirection="root"
      [routerLink]="[page.url]"
      routerLinkActive="selected"
      lines="none"
      detail="false">
      <ion-icon
        aria-hidden="true"
```

```
        slot="start"
        [ios]="page.icon + '-outline'"
        [md]="page.icon + '-sharp'"></ion-icon>
      <ion-label>{{ page.title }}</ion-label>
    </ion-item>
  </ion-menu-toggle>
}
```

`track page.url` tells Angular how to recognize an entry it has already drawn. `routerLinkActive="selected"` puts a class on whichever item matches the current route, which is what highlights the page you are on. `routerDirection` takes one of three values — `forward`, `back` or `root` — and decides which way the page transition animates; `root` says this is a jump to a top-level destination rather than a step deeper or a step back, which is exactly what a menu does. And `ion-menu-toggle` closes the menu after a tap on a narrow screen, so that choosing a page and dismissing the menu are one gesture rather than two. Its `auto-hide="false"` keeps the entries visible even while the menu is inactive, which is what the wide layout needs — there the menu is not covering anything, so there is nothing to hide.

The `[ios]` and `[md]` bindings on the icon are a pair, and the pairing is the point. Ionicons draws each symbol in three styles — filled, outline and sharp — and the entry asks for the outline variant in iOS mode and the sharp one in Material Design mode.

Which explains the shape of the `icon` value in `MenuEntry`: it is a stem, not a name. `'home'` becomes `home-outline` or `home-sharp`, depending on the mode Ionic is running in — normally `ios` on iOS and iPadOS, `md` on other platforms, though the mode is a setting rather than a hardware fact.

`addIcons()` fills a registry that is global to the running app rather than local to this component, so registering the menu icons in the shell makes those names available everywhere from the moment the shell is created. The rule is only this: an icon referenced by name has to be in the registry before Ionic tries to draw it. Each icon goes in under both spellings, `homeOutline` and `home-outline`, which is why the import is written one way and the template asks for it the other.

The practical discipline follows: keep two lists in step — the names the template asks for, and the imports handed to `addIcons()`. When you add an entry to the menu, you add its icon in the same edit.

The test that came with the app

One file is easy to forget, and it will tell you so. The starter shipped `src/app/app.component.spec.ts`, and it asserts the menu it was written for — twelve entries, the first of them linking to `/folder/Inbox`. Run the tests now:

```
npm test
```

```
❯ src/app/app.component.spec.ts (3 tests | 2 failed)
   × should have menu labels
   × should have urls

FAIL  AppComponent > should have menu labels
AssertionError: expected 3 to deeply equal 12
FAIL  AppComponent > should have urls
AssertionError: expected 3 to deeply equal 12
```

The app is fine; the spec is stale. The tests are doing their job: you changed what the menu contains, and they noticed. Each failure is only the first assertion its test got to — there are more behind them. So go through the file rather than through the error message. The expected twelve labels, twelve items and six links all become three; `Inbox` and `Outbox` become `Home` and `Tours`; `/folder/Inbox` becomes `/home`, and the assertion below it has no `/folder/Outbox` left to check. Then all three pass, and they describe the app you actually built.

That is the whole lesson for now: **a test describes the app, so it changes when the app does.** Chapter 11 takes the subject seriously, with the runner the starter already provides. Until then, keeping this one file honest costs a minute and means `npm test` always tells you the truth.

The last of the mail app

What the mail app leaves behind is easy to miss.

`src/app/app.component.scss` styles the menu through element IDs — `#inbox-list` and `#labels-list`. The labels list is gone, so those rules style nothing; delete them. The other is worth renaming rather than keeping: `id="menu-list"`, in the template and the stylesheet in the same edit, because the rule that gives the heading its weight selects on that ID and the two files have to agree for the menu to look like the screenshots here. Twelve lines lighter, and the file stops describing an app you are not building.

And `src/index.html` still carries the starter's `Ionic App`, which is the name

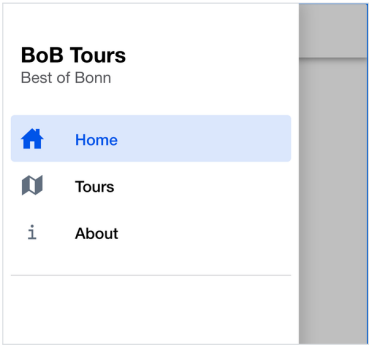
in the browser tab. Overwrite it the same way:

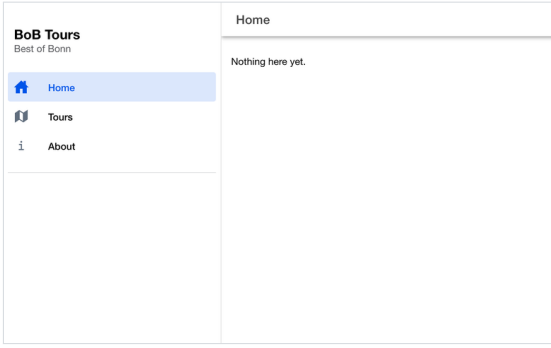
```
<title>BoB Tours</title>
```

When we make the app installable in chapter 12, its web app manifest gets the same name from a separate field of its own.

Where we are

Run it. A menu with three entries, each reaching a page that knows its own name, the current entry highlighted, and a shell that behaves like a phone app on a phone and a tablet app on a tablet. There is nothing in it yet, and that is exactly right — the next chapter gives BoB Tours something to navigate between, which is the point at which an app stops being a shell.





More in the documentation

- ionicframework.com/docs/api/menu
- ionicframework.com/docs/api/split-pane
- ionic.io/ionicons/usage

Summary

In this chapter, an application came into existence:

`ionic start` with the standalone project type, `ionic serve` and the development server, hot module replacement, the generated project file by file, `bootstrapApplication()` and `provideIonicAngular()`, lazy routes with `loadComponent`, the app's bundle size and its dependency health — and finally BoB Tours, with a menu of its own and one page standing behind all of it.

A generated project is a snapshot, not a subscription to future updates. It is the template as it stood on the day it was saved, and it has been aging quietly ever since. Ours ran its tests on a release with a known flaw, compiled for browsers Angular no longer promises anything about, and carried bundle budgets several times more generous than Angular's own. None of that is a scandal, and all of it is worth knowing on the first day, because everything you build sits on top of it. So measure before you build: `ionic build` for the size, `npm outdated` for the distance to current, `npm audit` for the state of the tree, and `npm ls` when you need to know who asked for a package.

Standalone makes a component's template dependencies explicit. A standalone component imports the components, directives and pipes its template uses, while icons referenced by name have to reach the registry before Ionic can draw them. That is more typing than the module-based arrangement, and it is the reason you can open any component in this project and see what its template depends on.

The app so far has three addresses, one page and no reason to move between them. That is what the next chapter is for.

Where BoB Tours goes from here

Chapter 3 left BoB Tours with three addresses, one page and no reason to move between them. The full book picks it up exactly there and never starts over. By the end of chapter 12, it is an app somebody else can install.

- **Navigation**, chapter 4: pages to move between, the stack behind the back button, and why the page you left is still there.
- **Services & Data**, chapter 5: a real server, the states between asking and knowing, and a model that does not simply mirror the server's response.
- **UI Components**, chapter 6: Ionic's components from Accordion to Toolbar, in the app and as a catalog.
- **Form validation**, chapter 7: a tour request with Signal Forms, fields that depend on each other, and a half-written request that does not vanish.
- **Theming, Styling, Customizing**, chapter 8: one color and everything it changes, dark mode and who decides, and the large screen.
- **Accessibility**, chapter 9: headings and landmarks a screen reader can move by, keyboard, text size, contrast, and checks that keep it that way.
- **Native functionality with Capacitor**, chapter 10: the app on an iPhone and an Android phone, live reload on the device, sharing, the way to a tour's starting point, and a life without a connection.
- **Debugging & Testing**, chapter 11: tests that notice when a well-meant tidy-up breaks the request.
- **Build, Deploy & Publish**, chapter 12: icons, signing, two languages, the iOS archive, a PWA and a pipeline on GitHub Actions.

Then two bonus chapters: **Ionic with or without frameworks** helps you choose between Angular, React, Vue and none at all, and **Ionic and AI** asks what a coding assistant builds for you, and when that actually pays off.

The eBook appears in October 2026, as an accessible PDF (PDF/UA-2); paperback and hardcover follow in November. The eBook is here:

► leanpub.com/ionic9