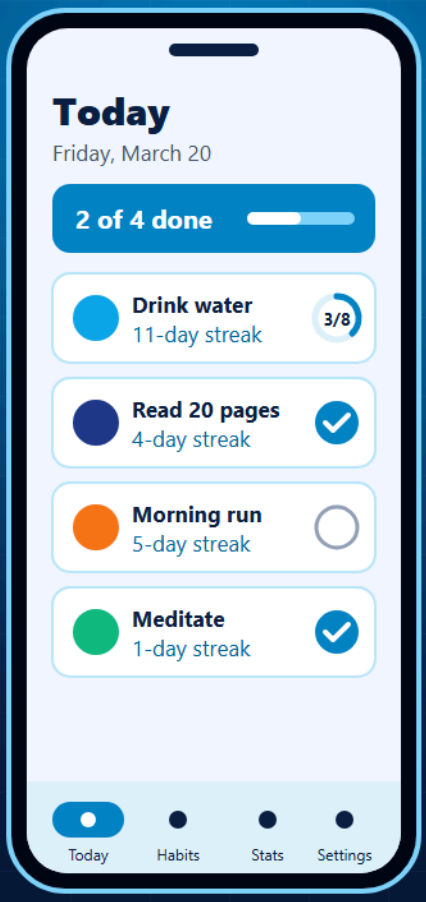
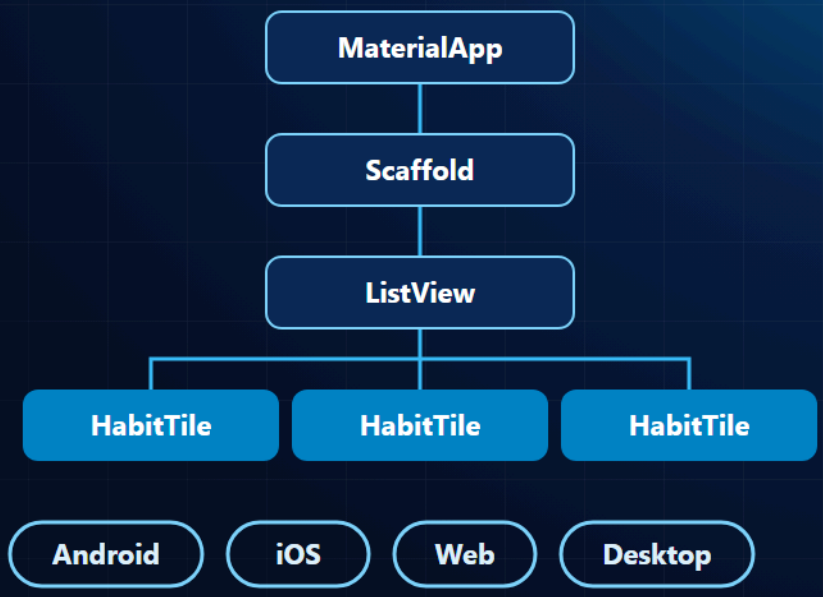
```dart
class HabitTile extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: Text(habit.name),
      trailing: StreakBadge(days: 11),
```





A COMPLETE, PRACTICAL GUIDE

Flutter from Zero to Production

Build cross-platform mobile apps with Dart — from your first widget to the app stores

Dart 3 · Widgets · Layout · Material 3 · go_router · Provider
REST & JSON · SQLite · Offline Sync · Testing · CI/CD · Publishing

DART · WIDGETS · STATE · DATA · TESTING · SHIPPING

THENEXBOOKS

Flutter
from Zero to Production

Build Cross-Platform Mobile Apps with Dart
From your first widget to a tested, offline-first app in Google Play and the App Store — built step by step

First Edition

THENEXBOOKS

Flutter from Zero to Production
Build Cross-Platform Mobile Apps with Dart

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. Flutter, Dart, their tools, third-party packages, and the policies of app stores change over time; commands, APIs, menu names, and requirements can differ between versions. Always consult the official documentation for the versions you use, and check each package's current documentation before relying on its API.

Flutter, Dart, Android, Google Play, and Firebase are trademarks of Google LLC. Apple, iOS, iPadOS, macOS, Xcode, Swift, TestFlight, and App Store are trademarks of Apple Inc., registered in the U.S. and other countries. Windows and Visual Studio are trademarks of the Microsoft group of companies. Linux is the registered trademark of Linus Torvalds. GitHub is a trademark of GitHub, Inc. Kotlin is a trademark of JetBrains s.r.o. Other product names are 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 Google LLC, Apple Inc., or any other company mentioned.

Dailyloop, the habit-tracking app used as the running example in this book, is fictional, as are Kitefield Labs, its team, its users, its REST API, and all sample data. Email addresses and domains use reserved example names.

Preface

Most teams that build a mobile app face the same uncomfortable question early on: do we build it twice? An Android version and an iOS version mean two codebases, two sets of bugs, two release schedules, and features that drift apart. A web version or a desktop version makes the problem worse. Flutter offers a different answer. You write your app once, in Dart, and Flutter compiles it for Android, iOS, the web, Windows, macOS, and Linux, drawing every pixel of the interface itself so that your app looks and behaves the way you designed it on every screen.

Learning Flutter well, however, takes more than learning a list of widgets. Real apps need a clear structure, state that stays consistent, data that survives restarts and bad network connections, reminders that fire at the right time, screens that work with a screen reader and in another language, tests that catch regressions, and a release process that ends with an app in the stores. Many tutorials stop at the counter app. This book goes all the way.

We teach Flutter from the very beginning and take you to a production-quality app. We do not just tour the API. We build one real app step by step, introducing each concept at the moment it solves a problem we actually have, and explaining *why* it works the way it does, so you can design and ship your own apps with confidence.

Who This Book Is For

- **Developers new to Flutter** — web, backend, Java, Kotlin, Swift, C#, or JavaScript developers who want to build mobile apps without maintaining two native codebases.
- **Native mobile developers** who know Android or iOS and want to understand how Flutter's model differs and where it fits.
- **Students and career changers** with basic programming knowledge who want a complete, practical path from first widget to published app.
- **Small teams and solo founders** who need one codebase for several platforms and want to get the architecture, testing, and release process right the first time.

You should know basic programming concepts such as variables, functions, and classes. You do not need any prior experience with Dart, Flutter, or mobile development.

What You Will Learn

- Dart from the ground up: types, functions, sound null safety, classes, mixins, enums, collections, records, patterns, sealed classes, futures, streams, and isolates.
- How Flutter works inside: widgets, elements, render objects, constraints, and the build cycle — so layout errors finally make sense.
- Building interfaces: stateless and stateful widgets, layout, scrolling, responsive screens, Material 3 theming, dark mode, and forms with validation.

- Navigation with the Navigator and the Router API, and declarative routing with go_router, including tabs, deep links, and redirects.
- State management: setState, InheritedWidget, ChangeNotifier, and a fair comparison of Provider, Riverpod, and Bloc — then one approach applied consistently.
- Data: JSON models, REST APIs with the http package, shared_preferences, SQLite with sqflite, async UI patterns, and offline-first sync.
- Polish and platform: local notifications and reminders, animations and custom painting, platform channels, accessibility, internationalization, and performance.
- Shipping: unit, widget, and integration tests, CI/CD pipelines, publishing to Google Play and the App Store, and taking the same app to the web and the desktop.

How This Book Is Organized

Part I — Dart and Flutter Foundations (Chapters 1–5) introduces Flutter, sets up your tools, and teaches the Dart language, including asynchronous programming. **Part II — Building UIs with Widgets** (Chapters 6–10) explains how Flutter renders, then covers stateful widgets, layout, Material 3 theming, and forms. **Part III — Navigation, State, and Architecture** (Chapters 11–15) covers navigation, go_router, state management, and the structure of a real app. **Part IV — Data, Networking, and Background Work** (Chapters 16–20) covers JSON, REST APIs, local storage, offline-first sync, and reminders. **Part V — Polish and Platform** (Chapters 21–24) covers animations, platform integration, accessibility, internationalization, and performance. **Part VI — Quality and Shipping** (Chapters 25–28) covers testing, CI/CD, store publishing, web and desktop, and a complete tour of the finished app. The appendices provide a cheat sheet, templates and reference material, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Dailyloop

Throughout the book we build **Dailyloop**, a habit-tracking app from the fictional studio Kitefield Labs. Dailyloop lets you define habits such as "Drink water" (eight glasses a day) or "Morning run" (Monday, Wednesday, and Friday), check them off each day, and watch your streaks grow. It reminds you at the time you choose, shows statistics and a weekly chart, works fully offline, and syncs with a simple REST backend when a connection is available. We start with a single static screen and finish with a tested, accessible, localized app that is ready for Google Play and the App Store, and that also runs on the web and the desktop.

The sample data is small enough to check by hand. Every streak, percentage, and chart value in the book follows from a few dozen check-ins and a handful of clearly stated rules, and the same numbers appear in the tests we write in Chapter 25.

Versions and Conventions

Examples target **recent stable versions of Flutter with Dart 3**, including sound null safety, records, patterns, and sealed classes. Flutter evolves quickly; where an API was recently renamed or behaves differently between versions, a Note says so. We use the core Flutter SDK wherever possible and a small set of popular packages — provider, go_router, http, shared_preferences, sqflite, flutter_local_notifications, and intl. Package APIs can change between major versions, so check each package's current documentation when you add it; the versions shown in `pubspec.yaml` listings are examples. Building for iOS and macOS requires a Mac with Xcode; everything else works on Windows, macOS, or Linux.

> **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 crashes, data loss, rejected releases, or security problems.

Code appears in `monospace`, with the file name shown above each listing, such as `lib/domain/streaks.dart`. Commands you type are labeled **Terminal**. Long lines are wrapped to fit the page. In examples, "today" is Friday, March 20, 2026, so that every result can be reproduced exactly.

Contents

Preface

PART I — DART AND FLUTTER FOUNDATIONS

1 Flutter Today: Cross-Platform Apps, the Toolchain, and What We Will Build

2 Setting Up Flutter and Running Your First App

3 Dart Essentials: Variables, Types, Functions, Control Flow, and Null Safety

4 Dart in Depth: Classes, Collections, Enums, Records, and Patterns

5 Asynchronous Dart: Futures, async/await, Streams, and Isolates

PART II — BUILDING UIS WITH WIDGETS

6 How Flutter Works: Widgets, Elements, Render Objects, and Constraints

7 Stateless and Stateful Widgets: Lifecycle, setState, and Keys

8 Layout: Rows, Columns, Stacks, Scrolling, and Responsive Screens

9 Material 3 Theming: Color Schemes, Typography, Dark Mode, and Components

10 Forms, Input, and Validation

PART III — NAVIGATION, STATE, AND ARCHITECTURE

11 Navigation Basics: Navigator, Routes, Dialogs, and the Router API

12 Declarative Routing with go_router

13 State Management Fundamentals: Lifting State, InheritedWidget, and ChangeNotifier

14 Choosing a State Management Approach: Provider, Riverpod, and Bloc Compared

15 App Architecture: Layers, Repositories, and the Dailyloop Project Structure

PART IV — DATA, NETWORKING, AND BACKGROUND WORK

16 JSON and Data Models: Parsing, Serialization, and Immutability

17 Networking: REST APIs with the http Package

18 Local Storage: shared_preferences, Files, and SQLite

19 Async UI and Offline-First Sync

20 Notifications and Reminders

PART V — POLISH AND PLATFORM

21 Animations: Implicit, Explicit, Hero, and Custom Painting

22 Platform Integration: Platform Channels, Plugins, and Permissions

23 Accessibility and Internationalization

24 Performance: Rebuilds, DevTools, Lists, Images, and Jank

PART VI — QUALITY AND SHIPPING

25 Testing Flutter Apps: Unit, Widget, and Integration Tests

26 CI/CD: Automated Checks, Builds, and Environments

27 Publishing to Google Play and the App Store

28 Beyond Mobile and Case Study: Web, Desktop, and the Complete Dailyloop App

APPENDICES

A Dart and Flutter Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Dart and Flutter Foundations

What Flutter is and what we will build, setting up the SDK, editors, emulators, and simulators, and learning the Dart language properly: types, functions, null safety, classes, collections, records, patterns, and asynchronous code with futures, streams, and isolates.

1 · Flutter Today

2 · Setting Up and Your First App

3 · Dart Essentials

4 · Dart in Depth

5 · Asynchronous Dart

CHAPTER 1 · PART I — DART AND FLUTTER FOUNDATIONS

Flutter Today: Cross-Platform Apps, the Toolchain, and What We Will Build

Before we install anything, we want you to understand what Flutter actually is, why it works the way it does, and where it fits among the other ways of building mobile apps. Then we will meet Dailyloop, the habit-tracking app you will build from an empty folder to a store-ready release over the course of this book.

Why Start with the Big Picture?

It is tempting to jump straight into code. Flutter makes that easy: you can create a project and see a running app within minutes. But developers who skip the big picture tend to hit the same walls a few weeks later. They do not understand why a widget rebuilds, why a layout overflows, why the app feels slow in debug mode but fast in release, or why a feature that works on Android needs extra setup on iOS. Every one of those surprises has a simple explanation once you know how Flutter is put together.

So this chapter is mostly ideas, with only a little code. We finish by introducing the fictional studio, the app, and the conventions that the rest of the book follows.

What Flutter Is

Flutter is Google's open-source UI toolkit for building apps from a single codebase. You write your app in **Dart**, a modern, statically typed language, and the same code can run on Android, iOS, the web, Windows, macOS, and Linux. That sentence sounds like a marketing line, so let us unpack the parts that matter to you as a developer.

One codebase. Your screens, business logic, networking, storage, and tests live in one Dart project. You do not maintain a Kotlin version of the Today screen and a Swift version of the same screen. When the product designer changes the layout of a habit row, you change it once. Platform-specific folders still exist (an `android/` folder, an `ios/` folder, and so on), but most of the time you only touch them for configuration such as app names, icons, permissions, and signing.

Dart. Dart was designed with client apps in mind. It compiles ahead of time to native machine code for release builds, which gives fast startup and predictable performance. During development it runs on a virtual machine with a just-in-time compiler, which makes hot reload possible. It has sound null safety, so the compiler can prove that a variable declared as non-nullable never holds `null`. Chapters 3, 4, and 5 teach the language from the ground up.

Flutter draws its own widgets. This is the single most important idea in this chapter. Many cross-platform tools work by asking the operating system to create its own buttons, text fields, and lists, and then controlling them from shared code. Flutter does not do that. It ships its own rendering engine and paints every pixel of your interface itself, the way a game engine paints a game. A Flutter button is not an Android `Button` or an iOS `UIButton`; it is a Flutter widget that Flutter lays out and draws onto a canvas that the platform provides.

That design choice explains most of Flutter's strengths and most of its trade-offs. Because Flutter owns the pixels, your app looks the same on every device, animations are under your full control, and a custom design is no harder to build than a standard one. On the other hand, Flutter has to re-implement the look and behavior of platform controls, and anything that is not about drawing, such as reading the battery level or scheduling a notification, must go through a bridge to platform code. We will come back to both sides.

The Rendering Engine: Impeller and Skia

The part of Flutter that turns drawing commands into pixels on the GPU is the rendering engine. For many years Flutter used **Skia**, the same 2D graphics library used in Chrome and Android. In recent versions Flutter introduced **Impeller**, a newer rendering engine designed to make animation smoother by preparing its shaders ahead of time instead of compiling them while the app runs. Impeller is the default on iOS and, in recent versions, on most Android devices. You rarely interact with the renderer directly, but you will hear both names in documentation and in performance discussions, and Chapter 24 explains why the switch matters for "jank", the visible stutter that happens when a frame takes too long.

> **NOTE**
>
> Flutter moves quickly, and details like the default renderer on a given platform can change between releases. Throughout this book we say "recent versions" when a fact depends on the release you install. When in doubt, the official Flutter documentation and the release notes for your installed version are the final word.

How Flutter Compares to Other Approaches

You can build a mobile app in several ways, and all of them ship real, successful products. We describe them neutrally here so you can recognize the trade-offs, not to rank them.

Native development means writing an Android app in Kotlin (or Java) with Android's UI toolkits, and a separate iOS app in Swift with SwiftUI or UIKit. You get direct, immediate access to every platform API and the platform's own look and feel. The cost is two codebases, two sets of skills, and two implementations of every feature. If you have built native apps before, much of your knowledge transfers to Flutter: lifecycle, permissions, signing, and store publishing all work the same way underneath.

Web-view approaches wrap a web application inside a native shell. The app's interface is HTML, CSS, and JavaScript rendered by the platform's embedded browser engine, and plugins expose device features. This is a natural path for teams with strong web skills and an existing web app. The interface behaves like a web page, which can be exactly what you want or something you must work around.

Bridge-based JavaScript frameworks let you write the app in JavaScript or TypeScript while the framework creates real native controls on each platform. You share logic and much of the UI code, and the result uses platform widgets. The framework has to coordinate between the JavaScript world and the native controls, and newer architectures in those frameworks have worked to make that coordination faster.

Flutter shares the UI code like the bridge approach, but instead of driving native controls it draws its own. Your Dart code is compiled to native machine code for release, and there is no browser in the middle on mobile.

Approach	Language	Who draws the UI	Shared code
Native	Kotlin, Swift	Platform UI toolkits	None between platforms
Web view	HTML, CSS, JS	Embedded browser engine	Most of the app
JS bridge	JavaScript, TypeScript	Platform controls	Most of the app
Flutter	Dart	Flutter's own engine	Most of the app

None of these choices is wrong. Teams pick Flutter when they want one codebase, a consistent custom design across platforms, strong tooling, and good performance without writing the UI twice. They pick native when they need the deepest platform integration on day one or already have two strong native teams. Keep that framing in mind; it will help you answer the question you will certainly be asked: "Why Flutter?"

The Platforms Flutter Targets

A Flutter project can contain support for six platforms. Each one needs its own toolchain on your computer, and some can only be built on specific operating systems.

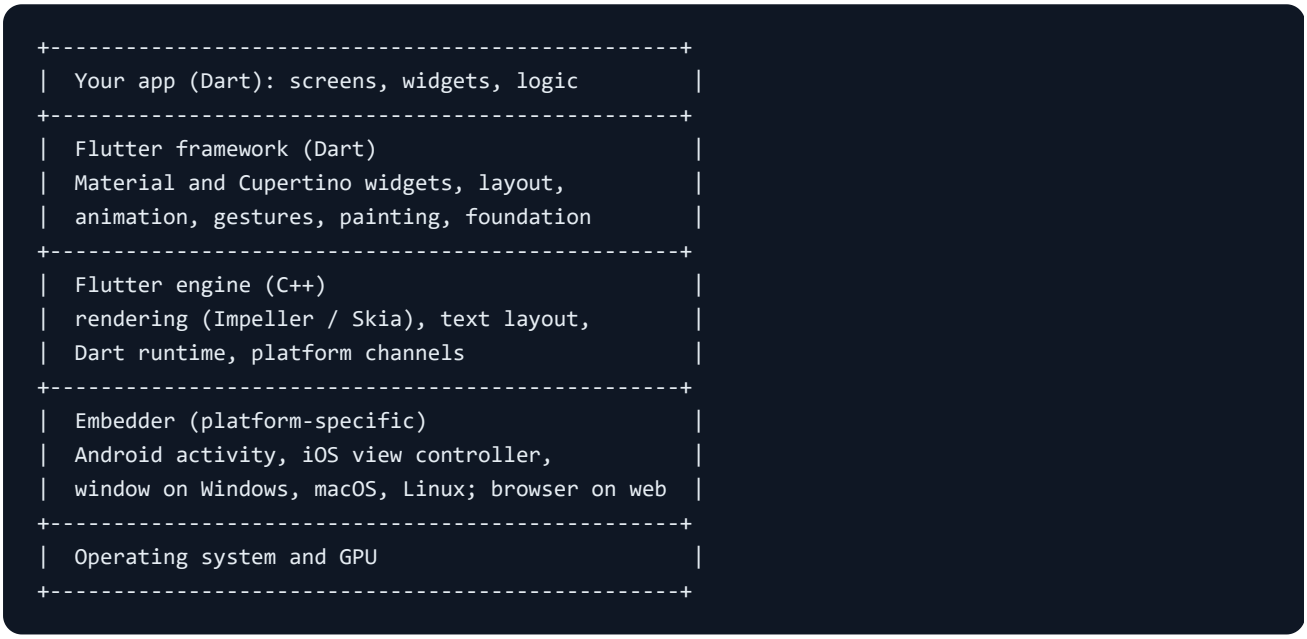
Platform	What you need to build	Notes
Android	Android SDK (via Android Studio)	Builds on Windows, macOS, and Linux
iOS	Xcode	Requires a Mac
Web	Chrome for development	Output is static files
Windows	Visual Studio with C++ tools	Builds on Windows
macOS	Xcode	Builds on a Mac
Linux	clang, CMake, ninja, GTK libraries	Builds on Linux

This book focuses on mobile, Android and iOS, because that is where most Flutter apps live and where the publishing process is most involved. We never require a Mac: everything except the iOS-specific steps works on Windows or Linux with an Android emulator or phone. Chapter 28 shows how the same Dailyloop code runs on the web and on desktop, and what has to change when it does.

> **TIP**
>
> If you do not own a Mac, do not let iOS hold you back. Build and test on Android for the whole book, read the iOS sections so you understand the process, and borrow or rent access to a Mac when you are ready to publish to the App Store. Your Dart code does not change.

Inside a Flutter App: Framework, Engine, and Embedder

A running Flutter app has three layers. Knowing their names makes the documentation much easier to read.



The **framework** is the part you work with every day. It is written in Dart, it is open source, and you can jump into its source from your editor to see exactly how a widget works. The **engine** is written in C++; it hosts the Dart runtime, lays out text, and rasterizes your frames. The **embedder** is the thin platform-specific shell that gives the engine a surface to draw on, forwards touch and keyboard input, and manages the app lifecycle. On Android the embedder lives inside a normal activity; on iOS it lives inside a normal view controller. That is why a Flutter app installs, launches, and is reviewed by the app stores like any other app.

When your Dart code needs something only the platform can do, such as reading the device's time zone, it sends a message over a **platform channel** to Kotlin or Swift code in the embedder layer, which answers asynchronously. Most of the time you use a **plugin**, a package that has already written that platform code for you. Chapter 22 builds Dailyloop's own small channel end to end.

Debug, Profile, and Release: JIT and AOT in Plain Words

Flutter builds your app in one of three modes, and the differences explain a lot of beginner confusion.

Mode	How Dart runs	Use it for
Debug	JIT on the Dart VM, with assertions	Everyday development and hot reload
Profile	AOT, with profiling hooks	Measuring performance on a real device
Release	AOT, fully optimized	What users install from the stores

JIT stands for just-in-time compilation: code is compiled while the program runs. That makes it possible to inject changed code into a running app, which is how hot reload works. It also means debug builds include extra checks, assertions, and debugging services, so they are noticeably slower and larger than what your users will run. **AOT** stands for ahead-of-time compilation: your Dart code is compiled to native machine code before the app is shipped. AOT builds start quickly and run at full speed, but you cannot hot reload them.

> **WARNING**
>
> Never judge your app's performance in debug mode. A list that stutters in debug may be perfectly smooth in profile or release mode. When something feels slow, measure it with `flutter run --profile` on a real device before you change any code. Chapter 24 covers this in depth.

Hot Reload and Why It Changes How You Work

Hot reload is the feature most developers mention first when they describe Flutter. While your app runs in debug mode, you edit a file and save it (or press `r` in the terminal), and within moments the running app shows your change. The crucial detail is that the app's **state is kept**. If you are three screens deep, with a form half filled in, you stay there; only the code that builds the interface is swapped.

This changes how you work. Instead of rebuilding, relaunching, and tapping back to the screen you care about, you adjust padding, colors, and text while looking at the result. You can try three variations of the habit row in a minute and keep the best one.

Hot reload has limits that come straight from how it works. It re-runs `build` methods with the new code, but it does not re-run code that already executed, such as your `main` function or the initializers of objects that already exist. When a change does not show up, use **hot restart** (`R` in the terminal), which restarts the Dart code and resets all state while keeping the native app running. If you changed native code, added a plugin, or edited platform configuration, stop the app and run it again from scratch. Chapter 2 shows all three in action.

A First Look at Flutter Code

You do not need to understand every symbol yet, but a small example makes the ideas concrete. This is a complete Flutter app that shows one line of text. We will not add this file to the project; it is only here to give you a feel for the code.

A tiny Flutter app (illustration)

```
import 'package:flutter/material.dart';

void main() {
  runApp(const GreetingApp());
}

class GreetingApp extends StatelessWidget {
  const GreetingApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: Scaffold(
        body: Center(
          child: Text('Small steps, every day.'),
        ),
      ),
    );
  }
}
```

Read it from the top. `main` is where every Dart program starts, and `runApp` hands Flutter the root of the interface. `GreetingApp` is a **widget**, a lightweight, immutable description of part of the interface. Its `build` method returns more widgets: a `MaterialApp` that sets up theming and navigation, a `Scaffold` that provides the basic page structure, a `Center` that centers its child, and a `Text`. The whole interface is a tree of small widgets nested inside each other, and that idea, "everything is a widget", is the heart of Chapter 6.

Notice what is not there: no XML layout file, no storyboard, and no separate styling language. Layout, style, and behavior are all ordinary Dart objects. The keyword `const` tells Dart that these widgets can be created once at compile time and reused, a small habit with real performance benefits that we will use throughout the book.

What a Flutter Project Contains

When you run `flutter create` in Chapter 2, you get a folder that looks roughly like this. We show only the most important entries.

```
dailyloop/
  lib/                  your Dart code; main.dart is the entry point
  test/                 unit and widget tests
```

```
android/              Android host project (Gradle)
ios/                  iOS host project (Xcode)
web/  windows/  macos/  linux/   other platform hosts
pubspec.yaml          name, version, dependencies, assets
pubspec.lock          exact resolved package versions
analysis_options.yaml lint rules for the analyzer
.gitignore            files Git should not track
```

Almost all of your work happens in `lib/` and `test/`. The `pubspec.yaml` file is the project's manifest: it names the package, sets the version, and lists dependencies on other packages from **pub.dev**, Dart's package repository. The platform folders are real native projects that you can open in Android Studio or Xcode. Flutter generates them for you and builds them for you; you edit them only for platform settings.

Honest Trade-offs

No technology is free. Here are the trade-offs we think you should know about before you invest your time, stated without exaggeration in either direction.

- **App size.** Because a Flutter app ships its own engine and framework code, even a simple app includes a baseline amount of code that a minimal native app would not. For most apps this is not a deciding factor, and Chapter 24 shows how to analyze and reduce size, but it is a real cost.
- **Platform-specific APIs need plugins.** Anything outside drawing and logic, such as notifications, sensors, biometrics, or files, goes through plugins. The plugin ecosystem is large, but when a new platform feature appears, you may need to wait for a plugin to support it or write a small platform channel yourself.
- **Platform look and feel is re-implemented.** Flutter provides Material widgets for Android-style design and Cupertino widgets for iOS-style design. They are carefully built, but they are Flutter's versions of those designs. Many teams embrace a single branded design instead, which is what Dailyloop does.
- **Web and desktop maturity varies.** Mobile is Flutter's most established target. Web and desktop are fully supported, but each has its own constraints, such as no direct file system access in the browser or different database options on desktop. Chapter 28 is honest about them.
- **Learning Dart.** Dart is approachable, especially if you know Java, Kotlin, Swift, C#, or TypeScript, but it is one more language. Chapters 3 to 5 exist so that this is a short step rather than an obstacle.

Weigh these against the benefits: one codebase, a fast development loop, a consistent design, a strong testing story, and excellent tooling. For a small studio building an app like Dailyloop, that balance is very attractive.

Meet Kitefield Labs and Dailyloop

Learning a framework through isolated snippets leaves gaps. You learn how a widget works but not where it belongs, how it talks to the rest of the app, or how to test it. So this book builds one realistic app from start to finish.

Kitefield Labs is a small fictional studio. **Nora Patel** handles product and design and writes the requirements. **Jonas Weber** is the backend developer who owns the REST API that the app syncs with. And **you** are the Flutter developer who turns Nora's requirements into a working app and connects it to Jonas's server. Our sample user is **Sam Rivera** (`sam@example.com`), whose habits fill the screens in every example.

The app is **Dailyloop**, a habit tracker. Nora's one-sentence pitch is "help people build small habits by making each day's check-in fast and each streak visible". Her feature list for version 1.0 looks like this:

- **Habits** with a name, optional notes, an icon, a color, the days of the week they are scheduled for, a daily target (for example, eight glasses of water), and an optional reminder time.
- **Today**: a list of the habits scheduled for today, with one tap to check in and a header showing how many are done.
- **Streaks**: the current and longest run of completed scheduled days for each habit, shown as a small flame badge.
- **Stats**: 7-day and 30-day completion rates and a weekly bar chart.
- **Reminders**: local notifications at each habit's reminder time on its scheduled days.
- **Offline first**: everything works without a network connection, and changes sync to the server when it is available.
- **Settings**: light, dark, or system theme, reminders on or off, and a "Sync now" button.
- **Polish**: animations, accessibility, and English and Spanish translations.

Sam's Habits

Every example in the book uses the same sample data, so the numbers you see in one chapter match the numbers in the next. "Today" is always **Friday, March 20, 2026**, and Sam started tracking on Monday, March 2.

ID	Habit	Scheduled	Daily target	Reminder
h1	Drink water	Every day	8	09:00
h2	Read 20 pages	Every day	1	21:00
h3	Morning run	Mon, Wed, Fri	1	06:30
h4	Meditate	Mon to Fri	1	None
h5	Learn Spanish	Every day	1	None (archived)

Learn Spanish is archived, so it appears only in the Habits list's "Archived" section and never on Today or Stats. On our sample Friday, four habits are scheduled. Sam has already read and meditated, has had three of eight glasses of water, and has not gone for the morning run yet. Here is a sketch of the Today screen we will build, drawn in plain text because this book has no screenshots:

```
+----------------------------------------+
| Today                    Fri, Mar 20   |
| 2 of 4 done                            |
+----------------------------------------+
| (water)  Drink water        3/8   *11  |
| (book)   Read 20 pages      [x]   *4   |
| (run)    Morning run        [ ]   *5   |
| (zen)    Meditate           [x]   *1   |
+----------------------------------------+
|  [Today]   Habits    Stats    Settings  |
+----------------------------------------+
   * = streak badge (flame icon + days)
```

Each row has a colored icon, the habit name, either a progress count such as "3/8" or a check box, and a streak badge. Drink water has an 11-day streak; today does not break it because the day is still in progress. Read 20 pages is on a 4-day streak, Morning run on 5, and Meditate on 1, because Sam skipped meditation on Thursday. If you tap the empty box next to Morning run, the header changes to "3 of 4 done" and the run streak grows to 6. Chapter 4 explains the streak rules precisely, and you will compute every one of those numbers yourself.

Along the bottom is a navigation bar with four destinations. On a tablet or desktop window that is 600 logical pixels wide or more, the same destinations move to a navigation rail on the left side. Chapter 12 builds both.

How Dailyloop Is Built

The finished app is organized in layers, and you will build them one at a time. Do not worry if some of these words are new; each gets its own chapter.

```
UI (screens, widgets)        lib/ui/
        |
State (controllers)          lib/state/      provider + ChangeNotifier
        |
Repository                   lib/data/repositories/
        |                \
Local store (SQLite)      REST API client (http)
lib/data/local/           lib/data/api/
        |
Domain rules (streaks, validation, dates)   lib/domain/, lib/core/
```

The domain rules sit at the bottom because everything else depends on them, and they depend on nothing from Flutter. That is why we write them first, in Chapters 3 and 4, using plain Dart. By the end of Chapter 4 you will have working date helpers, the `Habit` and `CheckIn` models, and the streak

calculations, all before drawing a single pixel.

The Plan for This Book

The book is divided into six parts. Each chapter builds on the previous one and adds real code to Dailyloop.

1. **Part I — Dart and Flutter Foundations** (Chapters 1–5): the toolchain, your first app, and the Dart language, including asynchronous code. You build the date helpers, models, and streak rules.
2. **Part II — Building UIs with Widgets** (Chapters 6–10): how Flutter works under the hood, stateful widgets, layout, Material 3 theming, and forms. You build the Today screen, the habit tiles, the theme, and the habit form.
3. **Part III — Navigation, State, and Architecture** (Chapters 11–15): navigation with `Navigator` and `go_router`, state management fundamentals, a neutral comparison of provider, Riverpod, and Bloc, and the app's final architecture.
4. **Part IV — Data, Networking, and Background Work** (Chapters 16–20): JSON, a REST client and a local mock server, SQLite storage, offline-first sync, and reminders.
5. **Part V — Polish and Platform** (Chapters 21–24): animations, platform channels, accessibility and translations, and performance.
6. **Part VI — Quality and Shipping** (Chapters 25–28): testing, CI/CD, publishing to Google Play and the App Store, and taking Dailyloop to the web and desktop.

We recommend reading straight through the first time; afterward, each chapter and the appendices work as a reference.

Conventions Used in This Book

A few conventions make the book easier to follow.

- **File captions.** Every code listing that belongs in the project starts with its path, such as `lib/domain/streaks.dart`. Paths are always relative to the project root and use forward slashes, even on Windows. When we show only part of a file, we say so and use `// ...` for the parts we left out.
- **Terminal listings.** Commands you type are captioned "Terminal". Output is shown as "you should see something like" because exact text varies between versions and machines.
- **Callouts.** A **Tip** is a shortcut or good habit. A **Note** adds background or a caveat. A **Warning** marks a mistake that can cost you time, data, or a store rejection.
- **Versions.** The book targets recent stable versions of Flutter with Dart 3. We say "recent versions" rather than quoting version numbers that would go out of date. When we use a third-party package, we remind you to check its current documentation, because package APIs can change between major versions.

- **Code style.** We follow the official Dart style: two-space indentation, trailing commas in widget trees, `const` wherever it is allowed, and the lint rules from `flutter_lints`. Code is formatted as `dart format` would format it.
- **Try It Yourself.** Every chapter ends with short exercises. They are optional, but doing them is the fastest way to make the material stick.

> **NOTE**
> AI coding assistants can be useful companions while you learn, for example to explain an error message or suggest a test case. When we mention one, we write "[your AI assistant]" rather than naming a product. Treat its suggestions like code from a colleague: read them, understand them, and run the analyzer and the tests before you trust them. Flutter APIs change, and an assistant may suggest deprecated names that this book deliberately avoids.

What You Need

You need a computer running Windows, macOS, or Linux with enough free disk space for the Flutter SDK, Android Studio, and an emulator image, which together take up several gigabytes. A physical Android phone or iPhone is helpful but not required; emulators and simulators are fine for nearly everything in this book. You need basic programming knowledge: variables, functions, classes, and loops in any mainstream language. You do not need prior Dart, Flutter, or mobile experience.

How to Get the Most from the Running Example

Because Dailyloop grows chapter by chapter, it helps to treat your project folder like a real product. Commit to Git at the end of every chapter with a message such as "Chapter 4: models and streak rules". If an experiment goes wrong, you can return to a known good state in seconds. When a later chapter replaces a temporary piece of code, such as the in-memory data that early screens use, the old version is still in your history if you want to compare.

Keep the sample numbers in mind, too. When the Today screen says "2 of 4 done" or a test expects a current streak of 5, those values come from the same small set of rules applied to the same sample data, and they agree across the domain code, the interface, and the tests.

Try It Yourself

1. Write down, in your own words, the difference between "Flutter draws its own widgets" and "a framework drives native controls". Name one advantage and one cost of Flutter's approach.
2. Look at the Today sketch above. Predict the header text and each streak badge if Sam also finishes the eighth glass of water today. (Check your prediction in Chapter 4.)
3. List the platforms you can build for on your own computer, based on the platform table. Which ones would need another machine?