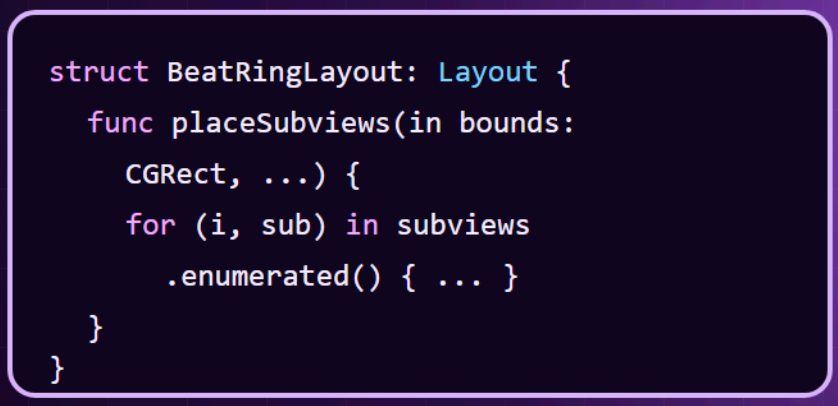
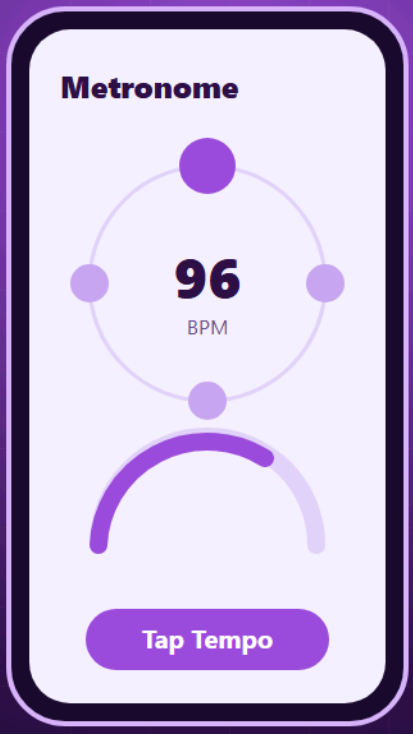
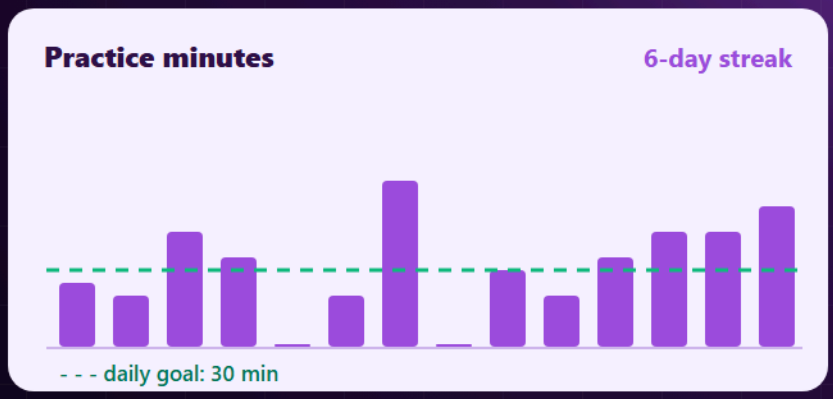










INTERMEDIATE TO ADVANCED

SwiftUI in Depth

Layout, State, Animation, Custom Components, and Performance for Production iOS Apps

Identity · Layout Protocol · Observation · Navigation
Gestures · Animation · Canvas · Charts · Accessibility

SWIFT 6 · iOS 18+ · iPHONE AND iPAD

THENEXBOOKS

SwiftUI in Depth

Layout, State, Animation, Custom Components, and Performance for Production iOS Apps

How SwiftUI really works, and how to build polished, fast, accessible apps with it

First Edition

THENEXBOOKS

SwiftUI in Depth
Layout, State, Animation, Custom Components, and Performance for Production iOS Apps

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. Apple platforms, SDKs, and developer tools change frequently; APIs, Xcode menus, Instruments templates, and default behaviors can differ between releases. Always consult Apple's current documentation for the versions you target.

Apple, iPhone, iPad, Mac, iOS, iPadOS, macOS, Xcode, Swift, SwiftUI, SwiftData, Instruments, VoiceOver, and SF Symbols are trademarks of Apple Inc., registered in the U.S. and other countries and regions. UIKit and other framework names are used to identify Apple technologies. 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 Apple Inc. Screen sketches in this book are original illustrations.

Tempora, the music-practice app used as the running example in this book, is fictional, as are its sample practice sessions and all other sample data. The musical works named in the sample data are used only as titles in a practice log; no scores or recordings are included.

Preface

Getting a first SwiftUI app on screen is quick. A few stacks, a list, some `@State`, a navigation stack, and something useful appears in an afternoon. The trouble starts later. A view resets its state for no obvious reason. A text field loses focus every time a timer ticks. An animation runs on the wrong element, or not at all. A layout that looked fine on one device clips on another, and the fix with `GeometryReader` breaks something else. A list stutters once it holds real data, and nobody can say which view is redrawing or why.

All of these problems come from the same place: SwiftUI is easy to write and harder to reason about. Its rules for identity, dependencies, layout, transactions, and the environment are consistent, but most tutorials never spell them out. Once you understand those rules, the surprises go away. You can predict when `body` runs, where state lives and how long it survives, what size a view will choose, and which part of a change will animate. Then you can build the custom layouts, containers, controls, and effects that make an app feel finished.

This book is about that second stage. We assume you have already built a basic SwiftUI app, and we go under the surface: how SwiftUI works, how layout is negotiated, how the Observation framework tracks changes, how to design navigation and data flow for a real app, how to animate and draw, how to build reusable components and styles, and how to make the result accessible, fast, and ready for iPad.

Who This Book Is For

- **iOS developers who know the basics of SwiftUI** and want to understand why it behaves the way it does, not just which modifier to add next.
- **UIKit developers moving to SwiftUI** who need the mental model, the interop story, and production patterns rather than another beginner tour.
- **Developers maintaining a growing SwiftUI codebase** who are fighting state bugs, slow lists, fragile layouts, or inconsistent components.
- **Designers who code and component-library builders** who want custom layouts, containers, styles, shapes, and animations that behave like built-in views.

You should be comfortable with Swift (structs, protocols, generics, closures, async/await) and with the everyday SwiftUI views and modifiers. We do not re-teach them.

What You Will Learn

- How SwiftUI turns your view values into a running interface: identity, lifetime, dependencies, and when `body` runs.
- How modifiers really compose, and how to pass information up the tree with preferences, anchors, and geometry changes.

- The layout process in detail: size proposals, stacks, alignment guides, grids, lazy containers, modern scroll APIs, the `Layout` protocol, and custom containers built on the subview APIs.
- State and data flow for real apps: the Observation framework, ownership, bindings and drafts, the environment as dependency injection, async work under Swift 6 concurrency, and a SwiftData persistence layer.
- Navigation and interaction: route-based navigation stacks, deep links and restoration, split views, sheets and inspectors, gesture composition, focus and forms, and custom styles for a reusable component library.
- Animation and graphics: transactions, springs, animatable data, transitions, matched geometry, phase and keyframe animators, shapes, paths, Canvas, Swift Charts, and TimelineView.
- Production quality: accessibility for custom controls, performance work and debugging view updates, UIKit interop, and adapting to iPad and other platforms.

How This Book Is Organized

Part I — How SwiftUI Works (Chapters 1–5) builds the mental model: identity, lifetime, the update cycle, modifiers, and preferences. **Part II — Layout** (Chapters 6–10) explains the layout process and goes all the way to custom layouts and custom containers. **Part III — State and Data Flow** (Chapters 11–15) covers Observation, ownership, the environment, async work and concurrency, and persistence. **Part IV — Navigation and Interaction** (Chapters 16–20) covers navigation architecture, presentations, gestures, focus and input, and custom styles. **Part V — Animation and Graphics** (Chapters 21–24) covers animation, transitions, drawing, and charts. **Part VI — Production Quality** (Chapters 25–28) covers accessibility, performance, UIKit interop, and iPad and multiplatform work, ending with a case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

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

The Running Example: Tempora

Throughout the book we build **Tempora**, a fictional music-practice tracker for iPhone and iPad. Musicians keep a library of pieces they are learning, each with a target tempo; they log practice sessions with duration, tempo reached, focus, and a rating; they practice with a live timer and a visual metronome with tap tempo; and they watch their progress through daily goals, streaks, charts, and a practice heatmap. Tempora is small enough to understand completely and rich enough to need everything in this book: custom layouts for tags and beat rings, a custom dial control, route-based navigation with deep links, drafts and validation, a persistence layer, animations tied to the beat, drawing with Canvas, charts, accessibility for custom controls, and an iPad layout. The sample data is fixed and small, so you can check every number in the book by hand.

Versions and Conventions

Code uses the **Swift 6 language mode** with complete concurrency checking and targets **iOS 18** as the minimum deployment version, built with a recent version of Xcode. Features that need newer systems, such as the Liquid Glass effects introduced with iOS 26, are wrapped in `if #available(iOS 26, *)` checks with a fallback, and the text says so. SwiftUI evolves every year; where a detail may differ in your version of Xcode or the SDK, a Note points it out. Output such as console logs is shown as "something like" what you will see, because exact formats vary between releases.

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

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

> **WARNING**
> A common mistake or something that can cause bugs, crashes, or poor performance.

Code appears in `monospace` with a caption naming the file, such as `Tempora/Features/Practice/PracticeView.swift`. Long modifier chains are wrapped one modifier per line. Screens are described in words and with simple text sketches.

Contents

Preface

PART I — HOW SWIFTUI WORKS

1 Beyond the Basics: The SwiftUI Mental Model and the Tempora Project

2 View Identity and Lifetime: Structural Identity, Explicit IDs, and State

3 The Update Cycle: Dependencies, Invalidation, and When body Runs

4 Modifiers in Depth: Order, ViewModifier, and Building Your Own

5 Preferences, Anchors, and Geometry: Communicating Up the View Tree

PART II — LAYOUT

6 The Layout Process: Proposals, Sizing, frame, and fixedSize

7 Stacks in Depth: Alignment Guides, Spacing, Priority, and ViewThatFits

8 Grids, Lazy Containers, and Modern Scroll Views

9 The Layout Protocol: Building Custom Layouts

10 Custom Containers: ViewBuilder, Subviews, and Container Values

PART III — STATE AND DATA FLOW

11 The Observation Framework in Depth

12 Ownership and Data Flow: @State, @Binding, @Bindable, and Drafts

13 The Environment: Custom Values, Dependency Injection, and App Settings

14 Async Work in Views: task, Cancellation, and Swift 6 Concurrency

15 A Persistence Layer with SwiftData: Repositories, Queries, and Previews

PART IV — NAVIGATION AND INTERACTION

16 NavigationStack Architecture: Routes, Paths, Deep Links, and Restoration

17 Split Views, Sheets, Popovers, and Inspectors

18 Gestures in Depth: Composition, GestureState, and Custom Controls

19 Focus, Forms, and Input: FocusState, Validation, and Search

20 Custom Styles and a Reusable Component Library

PART V — ANIMATION AND GRAPHICS

21 Animation in Depth: Transactions, Springs, and Animatable Data

22 Transitions and Choreography: matchedGeometryEffect, Phase and Keyframe Animators

23 Shapes, Paths, and Canvas: Drawing Custom Graphics

24 Data Visualization with Swift Charts and TimelineView

PART VI — PRODUCTION QUALITY

25 Accessibility in Depth: Custom Controls, Rotors, and Dynamic Type

26 Performance and Debugging View Updates

27 UIKit Interop: Representables, Hosting Controllers, and Coordinators

28 iPad, Multiplatform, and Shipping Tempora: A Case Study

APPENDICES

A SwiftUI Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

How SwiftUI Works

The mental model behind every SwiftUI screen: views as descriptions, identity and lifetime, the dependencies that decide when body runs, how modifiers really compose, and how information flows up the tree through preferences and geometry.

1 · Beyond the Basics

2 · View Identity and Lifetime

3 · The Update Cycle

4 · Modifiers in Depth

5 · Preferences, Anchors, and Geometry

CHAPTER 1 · PART I — HOW SWIFTUI WORKS

Beyond the Basics: The SwiftUI Mental Model and the Tempora Project

You can already build screens with SwiftUI. This chapter replaces "it usually works" with a precise model of what SwiftUI actually does with your code, and it sets up Tempora, the production-grade practice tracker we will grow, refactor, and polish for the rest of the book.

What This Book Assumes, and What It Adds

We assume you have shipped, or at least finished, a basic SwiftUI app. You know how to compose `VStack`, `List`, and `NavigationStack`, you have used `@State` and a simple `@Observable` model, and Xcode previews are part of your daily routine. We will not explain any of that again. When a basic API appears, we look at it from the other side: not "how do I use it" but "what does SwiftUI do with it, and what goes wrong when my mental model is off".

That other side is where intermediate apps get stuck. A timer resets when a modifier changes. A list scrolls sluggishly after a harmless-looking refactor. A sheet loses its half-typed form when the parent redraws. A custom layout fights its container. None of these are exotic bugs; they are the predictable consequences of four core mechanisms: **identity**, **dependencies**, **layout negotiation**, and **transactions**. Once you can reason about those mechanisms, most "SwiftUI is weird" moments turn into "of course it did that".

The book is organized around those mechanisms. Part I (this chapter through Chapter 5) builds the mental model: identity and lifetime, the update cycle, modifiers, and preferences. Part II is layout, all the way down to the `Layout` protocol and custom containers. Part III is state and data flow under Swift 6 concurrency. Part IV covers navigation architecture, gestures, forms, and a component library. Part V is animation and drawing, and Part VI is the production work that separates a demo from a product: accessibility, performance, UIKit interop, and iPad.

Every chapter applies its ideas to the same app, Tempora, so you see the trade-offs in a codebase with real constraints instead of in isolated snippets. When we take a shortcut early (a placeholder view, a simplified component), we say so and point to the chapter that replaces it.

SwiftUI Is a Description, Not the Views

The single most useful sentence about SwiftUI is this: **your view structs are not the views on screen**. A value of type `Text` or `PieceRowView` is a lightweight description of what you want. SwiftUI reads that description and maintains its own long-lived representation of the user interface, which it

uses to decide what to create, what to update, what to animate, and what to throw away.

Think of it as two layers:

- **The description layer** is your code. Every time SwiftUI asks for `body`, you build a fresh tree of small values. These values are created, compared, and discarded constantly. They have no lasting identity of their own.
- **The framework layer** is owned by SwiftUI. It holds the things that must survive between updates: the storage behind `@State`, the dependencies each view has recorded, layout results, running animations, and the platform objects that actually draw pixels and receive touches.

Apple has described the framework layer in conference sessions as an *attribute graph*: a dependency graph in which each node holds a value (a view's inputs, its body, its layout, its rendered output) and knows which other nodes it depends on. The graph is not public API and its details change between releases, so we will never write code against it. As a conceptual model, though, it explains almost everything: when a piece of state changes, SwiftUI marks the nodes that depend on it as outdated, re-evaluates only those, compares the new results with the old ones, and pushes the differences to the screen.

Conceptual model (not real API)

```
your code (values, rebuilt often)          SwiftUI (long-lived, owned by the framework)
---------------------------------          --------------------------------------------
TodayView()            -- body() -->       node: TodayView     [state, dependencies]
  VStack                                     node: VStack      [layout]
    Text("55 min")                             node: Text      [rendered text]
    StreakBadge(days: 6)                       node: StreakBadge [inputs: days = 6]
                                             |
state change --> mark dependents dirty --> re-run body --> diff --> update screen
```

Three consequences follow, and the next three chapters each explore one of them.

1. **SwiftUI must match old descriptions to new ones.** When `body` produces a new tree, how does SwiftUI know that this `Text` is "the same" one as before, so it can keep its state and animate the change instead of replacing it? The answer is *identity*, the subject of Chapter 2.
2. **SwiftUI decides when to ask for `body`.** You never call `body` yourself. SwiftUI calls it when a dependency changes, and the set of dependencies is determined by what your view actually reads. That is the update cycle, covered in Chapter 3.
3. **Modifiers build structure.** Each modifier wraps the view it is applied to in a new value, so the order of modifiers is the order of nesting. Chapter 4 makes that concrete.

Why the Values Are Cheap

A view struct usually contains a handful of stored properties: a few strings, numbers, maybe a closure. Creating one is about as expensive as creating any small struct, which is why SwiftUI can afford to rebuild descriptions freely. The expensive parts (text layout, image decoding, platform views,

compositing) live in the framework layer, and SwiftUI only redoes them when the description actually changed.

This has a practical implication that surprises many developers: **the initializer of a view is not a lifecycle event**. SwiftUI may create your view value many times while the thing on screen stays exactly the same. Code that runs in `init`, such as starting a timer, creating a formatter, or kicking off a network request, runs every time the parent rebuilds its description. Put long-lived work in state that SwiftUI owns, or in modifiers such as `.task` whose lifetime is tied to the on-screen identity. Chapter 2 makes that distinction precise, and Chapter 14 covers `.task` in depth.

body as a Function of State

The second half of the mental model is that `body` should behave like a function: given the same inputs and the same state, it returns the same description. It should not mutate state, perform side effects, or depend on anything SwiftUI cannot see (a global variable, the current time read directly, a singleton that is not observable). SwiftUI is free to call `body` more or less often than you expect, and correctness must not depend on how many times it runs.

That rule is why Tempora reads "now" through an environment value instead of calling `Date.now` deep inside views. A view whose output depends on the clock but does not tell SwiftUI about it will show stale data, because nothing invalidates it when the clock moves. Later we use `TimelineView` for views that must update on a schedule, and the `fixedNow` environment value to pin "now" in previews and tests.

> **NOTE**
> "Function of state" does not mean `body` must be trivial. It can compute derived values, filter arrays, and format text. It means the computation must be *pure* and reasonably cheap. Chapter 3 shows how to tell when a computation in `body` has become too expensive and where to move it.

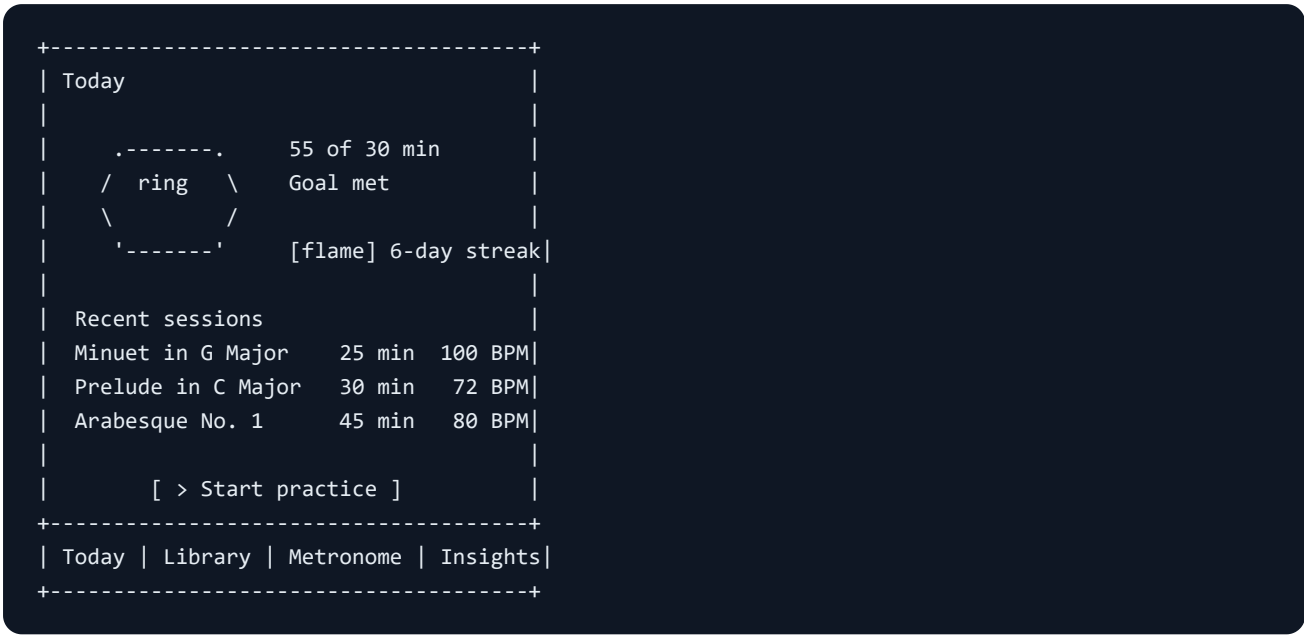
A Tour of Tempora

Tempora is a music-practice tracker for iPhone and iPad. A musician keeps a library of pieces they are learning, logs practice sessions with a duration, the tempo they reached, a focus area, and a rating, and watches their progress through goals, streaks, and charts. It is small enough to understand in an afternoon and rich enough to exercise every topic in this book: custom layouts for tags and beat indicators, a live timer, a gesture-driven tempo dial, animations, charts, a Canvas heatmap, persistence, deep links, and an iPad layout.

The app has four tabs. Here is what each one shows, sketched as wireframes.

Today

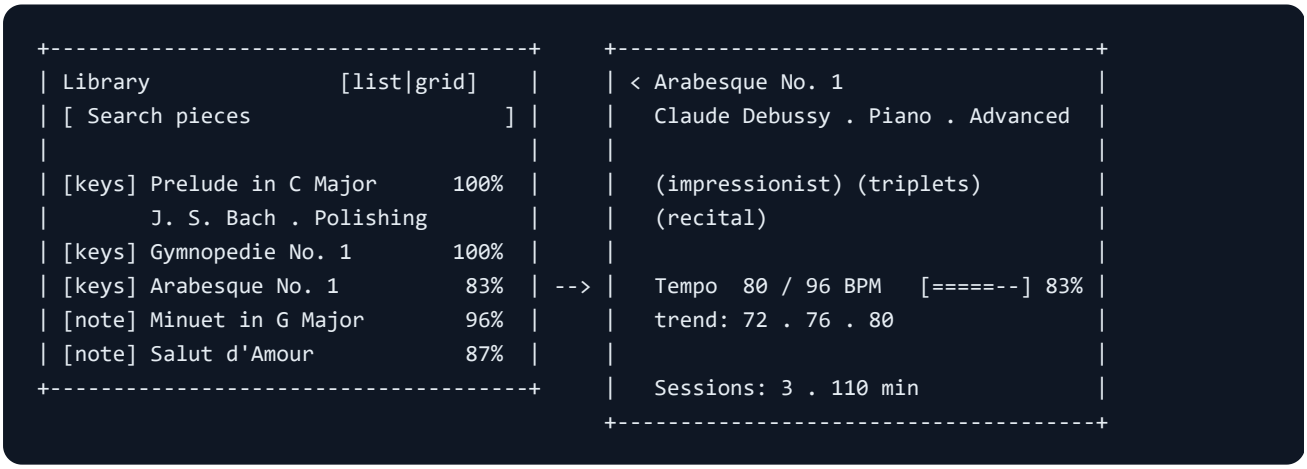
The Today tab answers "how am I doing right now?". A ring shows today's minutes against the daily goal (30 minutes by default), a badge shows the current streak of consecutive practice days, a list shows recent sessions, and a prominent button starts a practice session.



With the book's sample data and "now" pinned to Saturday, March 14, 2026, at 20:00, Today shows 55 minutes (two sessions: 30 and 25 minutes), a current streak of 6 days (March 9 through 14), and a current goal streak of 4 days.

Library and Piece Detail

The Library lists the five sample pieces with their instrument, status, and *tempo progress*: the latest tempo reached divided by the target tempo, capped at 100%. A toolbar button switches between a list and a grid, and a search field filters by title and composer. Tapping a piece pushes its detail screen, which shows its tags in a wrapping flow layout, a tempo trend chart, and its sessions.



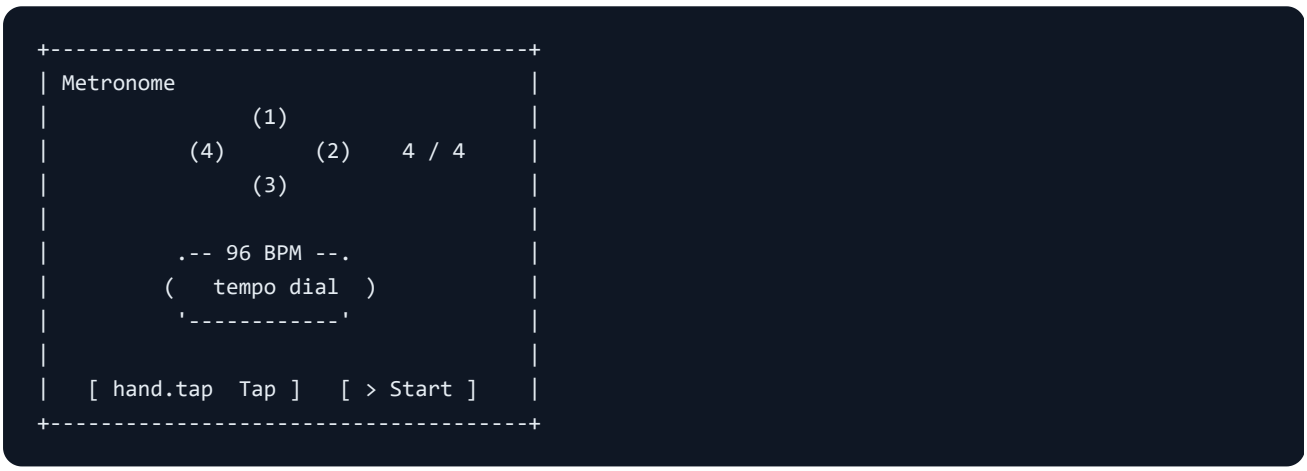
The wireframe drops the accent in "Gymnopédie" only because it is a plain-text sketch; the real data keeps it, and so does every listing in the book. Accented characters and apostrophes (as in "Salut d'Amour") are deliberate: they test search, sorting, and accessibility with realistic strings.

Practice

The practice screen is a live timer for one piece. It shows elapsed time, updated every second through a `TimelineView`, and a Finish button that turns the elapsed time into an editable session draft and opens the session editor. The editor is a form with duration, tempo, focus, rating, and notes, and it validates its input before saving.

Metronome

The metronome is visual and haptic: a ring of beat indicators lights up in time, with the first beat accented, and a circular tempo dial sets the BPM between 30 and 240. A tap-tempo button computes the BPM from your last few taps. Audio is deliberately out of scope, because scheduling sample-accurate sound is an AVFoundation topic, not a SwiftUI one.



Insights

Insights turns the session log into understanding: a bar chart of daily minutes with a goal line, a tempo trend per piece, a donut of minutes by focus area, and a calendar-style heatmap drawn with `Canvas`. With the sample data, the 16 sessions add up to 440 minutes over 12 practice days, the goal is met on 8 of those days, and technique accounts for 200 of the 440 minutes.

The Project Layout

Tempora is organized by layer and then by feature. The model is plain value types with no SwiftUI dependency, the store and repositories sit on top of it, and features contain views. A small design system holds reusable modifiers, styles, layouts, and containers.

```
Tempora/
  App/            TemporaApp, RootView, AppTab, Router
  Model/          Piece, PracticeSession, enums, SampleData, PracticeStats, SessionDraft
```

```
  Store/          PracticeStore, PracticeRepository, in-memory and SwiftData repositories
  Environment/    TemporaEnvironment (@Entry values, theme, settings keys)
  DesignSystem/   CardModifier, Styles/, Layouts/, Containers/, Shapes/, Effects/
  Features/       Today/, Library/, Practice/, Metronome/, Insights/, Settings/
  Interop/        UIKit bridges (PDF scores, session history)
TemporaTests/     Swift Testing suites for statistics and the metronome
```

Two rules keep the layers honest. First, files in `Model/` import only Foundation, which keeps them testable and lets them cross concurrency boundaries freely. Second, views never talk to a repository directly; they read and mutate a `PracticeStore` from the environment. That indirection pays off in Chapter 15, when the persistence layer switches from memory to SwiftData without touching a single view.

The Model Layer

Tempora's model is a set of value types. Value types are the natural fit for SwiftUI: they are cheap to copy, trivially `Equatable` and `Hashable`, and `Sendable` when their stored properties are, which matters a great deal under Swift 6. We start with the enums that classify pieces and sessions.

Tempora/Model/PracticeEnums.swift

```swift
import Foundation

enum Instrument: String, CaseIterable, Identifiable, Codable, Sendable {
    case piano, violin, guitar, cello
    var id: Self { self }
    var displayName: String { rawValue.capitalized }          // "Piano"
    var symbolName: String {                                   // SF Symbol
        switch self {
        case .piano: "pianokeys"
        case .violin: "music.note"
        case .guitar: "guitars"
        case .cello: "music.quarternote.3"
        }
    }
}

enum Difficulty: Int, CaseIterable, Identifiable, Codable, Sendable, Comparable {
    case beginner = 1, intermediate = 2, advanced = 3
    var id: Int { rawValue }
    var displayName: String { "\(self)".capitalized }          // "Beginner"
    static func < (lhs: Difficulty, rhs: Difficulty) -> Bool {
        lhs.rawValue < rhs.rawValue
    }
}

enum PieceStatus: String, CaseIterable, Identifiable, Codable, Sendable {
    case learning, polishing, performanceReady
    var id: Self { self }
    var displayName: String {
        switch self {
        case .learning: "Learning"
```

```
        case .polishing: "Polishing"
        case .performanceReady: "Performance Ready"
        }
    }
}

enum PracticeFocus: String, CaseIterable, Identifiable, Codable, Sendable {
    case technique, repertoire, scales, sightReading
    var id: Self { self }
    var displayName: String {
        switch self {
        case .technique: "Technique"
        case .repertoire: "Repertoire"
        case .scales: "Scales"
        case .sightReading: "Sight-Reading"
        }
    }
}
```

Each enum is `Identifiable` with itself (or its raw value) as the ID, which lets us pass `Instrument.allCases` straight to `ForEach` and `Picker`. The display names live on the model for now because the UI is English-only; in a localized app you would return a `LocalizedStringResource` instead. Note the use of `switch` as an expression, which keeps these computed properties compact.

Next come the two core records. A `Piece` is something you are learning; a `PracticeSession` is one block of practice on one piece.

Tempora/Model/Piece.swift

```
import Foundation

struct Piece: Identifiable, Hashable, Codable, Sendable {
    let id: UUID
    var title: String
    var composer: String
    var instrument: Instrument
    var targetTempo: Int            // BPM, 30...240
    var difficulty: Difficulty
    var status: PieceStatus
    var tags: [String]
    var scoreFileName: String?      // optional PDF in the app's Documents folder (Ch 27)
}
```

Tempora/Model/PracticeSession.swift

```
import Foundation

struct PracticeSession: Identifiable, Hashable, Codable, Sendable {
    let id: UUID
    var pieceID: Piece.ID
    var startedAt: Date
    var durationMinutes: Int        // 5...180
    var tempo: Int                  // BPM reached, 30...240
```

```
    var focus: PracticeFocus
    var rating: Int                // 1...5
    var notes: String
}
```

Notice that `id` is a `let`. Identity is the one property that must never change for the lifetime of a record, and Chapter 2 shows how much SwiftUI relies on that. Everything else is `var`, because editing a piece or session means producing a modified copy and handing it to the store. The session refers to its piece through `Piece.ID` rather than holding a `Piece`: a value type cannot share a reference, and copying the whole piece into every session would leave stale copies the moment the piece is renamed.

Sample Data You Can Reason About

Good sample data is a design tool. Tempora's sample data is fixed: five pieces, sixteen sessions in the first two weeks of March 2026, and a pinned reference "now" of Saturday, March 14, 2026, at 20:00 local time. Because every number is known in advance, every preview, test, and worked example in this book can state exactly what you should see. The IDs are deterministic too, so a deep link or a test can refer to "piece 3" by a stable UUID.

Tempora/Model/SampleData.swift

```
import Foundation

enum SampleData {
    /// ISO 8601 calendar: weeks start on Monday. Local time zone.
    static let calendar: Calendar = {
        var calendar = Calendar(identifier: .iso8601)
        calendar.timeZone = .current
        return calendar
    }()

    // The force unwraps below are safe: the format always produces a valid UUID string.
    static func pieceID(_ n: Int) -> UUID {
        UUID(uuidString: String(format: "B0000000-0000-4000-8000-%012d", n))!
    }

    static func sessionID(_ n: Int) -> UUID {
        UUID(uuidString: String(format: "A0000000-0000-4000-8000-%012d", n))!
    }

    /// A local date in 2026.
    static func date(_ month: Int, _ day: Int, _ hour: Int, _ minute: Int) -> Date {
        let parts = DateComponents(year: 2026, month: month, day: day,
                                   hour: hour, minute: minute)
        return calendar.date(from: parts) ?? Date(timeIntervalSince1970: 0)
    }

    /// Saturday, March 14, 2026, 20:00 local time. Previews and tests use this as "now".
    static let referenceNow = date(3, 14, 20, 0)

    static let pieces: [Piece] = [
        Piece(id: pieceID(1), title: "Prelude in C Major", composer: "J. S. Bach",
```

```
            instrument: .piano, targetTempo: 72, difficulty: .intermediate,
            status: .polishing, tags: ["baroque", "arpeggios"],
            scoreFileName: "bach-prelude-c.pdf"),
        Piece(id: pieceID(2), title: "Gymnopédie No. 1", composer: "Erik Satie",
            instrument: .piano, targetTempo: 66, difficulty: .beginner,
            status: .performanceReady, tags: ["calm", "pedaling"],
            scoreFileName: nil),
        Piece(id: pieceID(3), title: "Arabesque No. 1", composer: "Claude Debussy",
            instrument: .piano, targetTempo: 96, difficulty: .advanced,
            status: .learning, tags: ["impressionist", "triplets", "recital"],
            scoreFileName: "debussy-arabesque-1.pdf"),
        Piece(id: pieceID(4), title: "Minuet in G Major", composer: "Christian Petzold",
            instrument: .violin, targetTempo: 104, difficulty: .beginner,
            status: .polishing, tags: ["baroque", "bowing"],
            scoreFileName: nil),
        Piece(id: pieceID(5), title: "Salut d'Amour", composer: "Edward Elgar",
            instrument: .violin, targetTempo: 60, difficulty: .intermediate,
            status: .learning, tags: ["romantic", "vibrato", "recital"],
            scoreFileName: nil),
    ]

    // Columns: n, startedAt, piece n, minutes, tempo, focus, rating
    static let sessions: [PracticeSession] = [
        make(1, date(3, 1, 18, 30), 1, 25, 60, .technique, 3),
        make(2, date(3, 2, 7, 15), 2, 20, 58, .repertoire, 4),
        make(3, date(3, 3, 18, 45), 1, 30, 63, .technique, 3),
        make(4, date(3, 3, 19, 20), 4, 15, 88, .scales, 4),
        make(5, date(3, 4, 18, 30), 3, 35, 72, .sightReading, 2),
        make(6, date(3, 6, 8, 0), 2, 20, 62, .repertoire, 4),
        make(7, date(3, 7, 10, 0), 1, 40, 66, .technique, 4),
        make(8, date(3, 7, 16, 30), 5, 25, 48, .repertoire, 3),
        make(9, date(3, 9, 18, 30), 3, 30, 76, .technique, 3),
        make(10, date(3, 10, 7, 30), 4, 20, 94, .scales, 4),
        make(11, date(3, 11, 18, 40), 1, 35, 69, .repertoire, 4),
        make(12, date(3, 12, 18, 30), 5, 30, 52, .technique, 3),
        make(13, date(3, 12, 19, 15), 2, 15, 66, .repertoire, 5),
        make(14, date(3, 13, 18, 30), 3, 45, 80, .technique, 3,
             notes: "Left-hand triplets still uneven at 84."),
        make(15, date(3, 14, 9, 0), 1, 30, 72, .repertoire, 5,
             notes: "Clean run at target tempo!"),
        make(16, date(3, 14, 17, 0), 4, 25, 100, .sightReading, 4),
    ]

    private static func make(
        _ n: Int, _ startedAt: Date, _ piece: Int, _ minutes: Int, _ tempo: Int,
        _ focus: PracticeFocus, _ rating: Int, notes: String = ""
    ) -> PracticeSession {
        PracticeSession(id: sessionID(n), pieceID: pieceID(piece), startedAt: startedAt,
                        durationMinutes: minutes, tempo: tempo, focus: focus,
                        rating: rating, notes: notes)
    }
}
```

A few details are worth calling out. `SampleData` is an enum with no cases, so it cannot be instantiated by accident; it is a namespace. All stored properties are `static let` constants of `Sendable` types, which Swift 6 accepts as safe global state. The calendar is ISO 8601 so that weeks start on Monday, which the heatmap in Chapter 23 depends on, and it uses the device's time zone so that "18:30" means 18:30 wherever the reader runs the code. The sessions are listed in chronological order, matching the store's invariant that `sessions` is sorted by `startedAt`.

> **TIP**
> Pinning "now" is the cheapest reliability upgrade you can give a date-heavy app. With a fixed reference date, a preview of the Today screen always shows the same streak, a test of the streak logic always has the same expected value, and a screenshot never depends on the day you took it. We wire this up through the `fixedNow` environment value in Chapter 3 and explain the environment mechanics in Chapter 13.

The Repository and the Store

Data flows through two types. A **repository** is the persistence boundary: an `async` API that loads and saves records and could be backed by memory, SwiftData, a file, or a server. The **store** is the observable, main-actor object that views read. Keeping them separate means views never wait on I/O directly, and the store can apply changes optimistically while the repository saves in the background.

Tempora/Store/PracticeRepository.swift

```
import Foundation

protocol PracticeRepository: Sendable {
    func loadPieces() async throws -> [Piece]
    func loadSessions() async throws -> [PracticeSession]
    func save(_ piece: Piece) async throws
    func save(_ session: PracticeSession) async throws
    func deleteSession(id: PracticeSession.ID) async throws
}
```

The protocol inherits from `Sendable` because the main-actor store holds a repository and calls it from asynchronous code; a conforming type must therefore be safe to use from any isolation domain. An `actor` is the simplest way to satisfy that, and the in-memory version is all we need until Chapter 15.

Tempora/Store/InMemoryPracticeRepository.swift

```
import Foundation

actor InMemoryPracticeRepository: PracticeRepository {
    private var pieces: [Piece]
    private var sessions: [PracticeSession]

    init(pieces: [Piece] = [], sessions: [PracticeSession] = []) {
        self.pieces = pieces
```

```
        self.sessions = sessions
    }

    func loadPieces() async throws -> [Piece] { pieces }

    func loadSessions() async throws -> [PracticeSession] { sessions }

    func save(_ piece: Piece) async throws {
        if let index = pieces.firstIndex(where: { $0.id == piece.id }) {
            pieces[index] = piece
        } else {
            pieces.append(piece)
        }
    }

    func save(_ session: PracticeSession) async throws {
        sessions.removeAll { $0.id == session.id }
        sessions.append(session)
        sessions.sort { $0.startedAt < $1.startedAt }
    }

    func deleteSession(id: PracticeSession.ID) async throws {
        sessions.removeAll { $0.id == id }
    }
}
```

Actor methods are implicitly asynchronous when called from outside the actor, so they satisfy the protocol's `async throws` requirements even though their bodies neither await nor throw. Everything that crosses the actor boundary (`Piece`, `PracticeSession`, arrays of them, a `UUID`) is a `Sendable` value, which is exactly what the compiler checks in Swift 6 mode.

The store is the object views see. Here is its skeleton; Chapter 11 completes the mutation methods, and Chapter 14 adds `load()` and the loading and error states.

Tempora/Store/PracticeStore.swift (skeleton)

```
import Foundation
import Observation

@MainActor
@Observable
final class PracticeStore {
    private(set) var pieces: [Piece]
    private(set) var sessions: [PracticeSession]          // sorted by startedAt ascending
    private(set) var isLoading = false
    private(set) var lastError: String?
    @ObservationIgnored private let repository: any PracticeRepository

    init(repository: any PracticeRepository, pieces: [Piece] = [],
         sessions: [PracticeSession] = []) {
        self.repository = repository
        self.pieces = pieces
        self.sessions = sessions.sorted { $0.startedAt < $1.startedAt }
    }
```