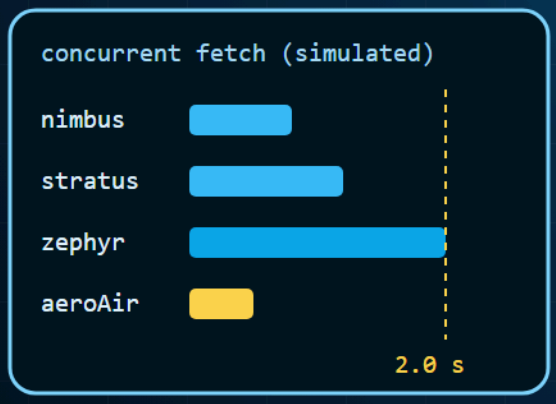
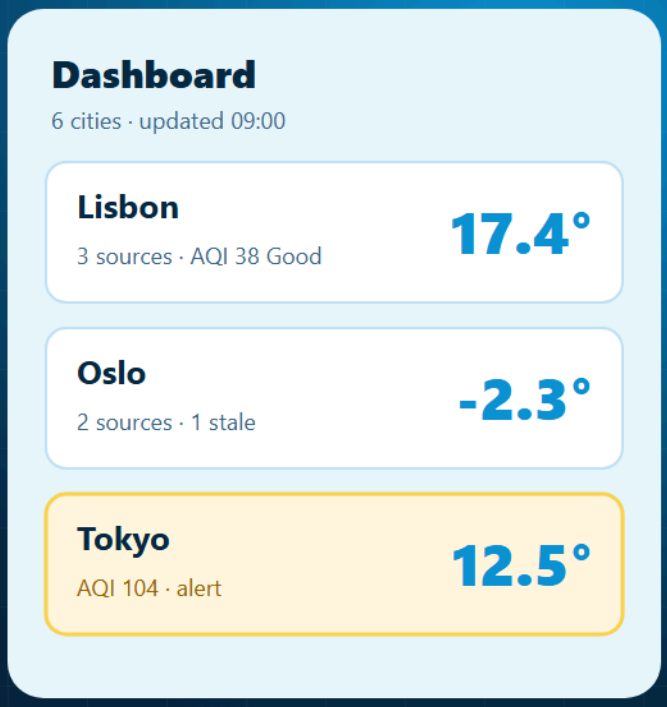







```
actor ForecastCache {
  func snapshot(for city: City) async throws -> CitySnapshot
}
async let air = aeroAir.airQuality(for: city) // Sendable
```

A PRACTICAL DEEP DIVE

Swift Concurrency for iOS Developers

async/await, Tasks, Actors, Sendable, and Swift 6 Data-Race Safety in Real Apps

Structured concurrency · Task groups · AsyncStream
@MainActor · Isolation · Swift 6 migration · Testing

FROM GCD AND CALLBACKS TO RACE-FREE CODE

THENEXBOOKS

Swift Concurrency for iOS Developers

async/await, Tasks, Actors, Sendable, and Swift 6 Data-Race Safety in Real Apps
From GCD and completion handlers to structured, race-free code — built step by step in one real app

First Edition

THENEXBOOKS

Swift Concurrency for iOS Developers
async/await, Tasks, Actors, Sendable, and Swift 6 Data-Race Safety in Real 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. Swift, Xcode, the iOS SDK, and their compiler diagnostics change from release to release; settings, menu labels, error messages, and API availability can differ in the version you use. Compiler messages quoted in this book are paraphrased and shortened, and timings shown are simulated. Always consult Apple's documentation and the Swift Evolution proposals for the versions you target.

Apple, iPhone, iPad, iOS, Xcode, Swift, SwiftUI, Instruments, and related names are trademarks of Apple Inc., registered in the U.S. and other countries and regions. 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. All illustrations and interface sketches in this book are original.

Skyvale, the weather dashboard app used as the running example in this book, is fictional, as are its weather and air-quality providers (Nimbus Weather, Stratus Open Data, Zephyr Live, and AeroAir Quality), their APIs, and all readings and sample data. City names are real places, but every reading is invented. Web addresses use the reserved `.example` domain and do not exist.

Preface

Most iOS codebases carry two eras of concurrency at once. There is the code written with dispatch queues, completion handlers, delegates, operation queues, and Combine pipelines — code that works, mostly, until a callback fires twice, a cache is read on one queue while another writes it, or a view model updates the UI from a background thread. And there is the newer code written with `async` and `await`, which reads beautifully until the compiler starts asking questions about actors, isolation, and `Sendable` that nobody on the team can quite answer.

Swift's concurrency model is not just nicer syntax. It is a different way of thinking: work forms a tree of tasks with clear lifetimes, cancellation flows down that tree automatically, mutable state lives in isolation domains, and — with the Swift 6 language mode — the compiler proves that your code is free of data races before it ever runs. Once that mental model clicks, the diagnostics stop feeling like obstacles and start feeling like a colleague who reviews every line for thread safety.

This book teaches that model from the ground up for developers who already build iOS apps. We do not re-teach Swift or SwiftUI. Instead we take one realistic app, start from the callback-based code many teams still have, and rebuild its concurrency layer step by step — explaining at every stage *which* isolation domain runs the code, *where* the suspension points are, and *why* the compiler accepts or rejects it.

Who This Book Is For

- **iOS developers moving from GCD and completion handlers** who want to write async/await code with confidence instead of by trial and error.
- **Teams migrating to Swift 6** who face hundreds of concurrency warnings and need a strategy, not just individual fixes.
- **Combine and delegate users** who need to bridge existing APIs into async sequences and streams cleanly.
- **Developers who already use async/await** but are unsure about actors, reentrancy, `@MainActor`, `Sendable`, `sending`, and the newer approachable-concurrency settings.

You should be comfortable with Swift, closures, protocols and generics, and building simple SwiftUI or UIKit screens in Xcode. No prior experience with Swift concurrency is required.

What You Will Learn

- How async/await works: suspension points, async functions and properties, error handling, and what can change across an `await`.
- How to bridge legacy code with continuations, turn delegates into `AsyncThrowingStream`s, and interoperate with Combine.

- How to create, prioritize, cancel, and time out tasks, and how structured concurrency with `async let` and task groups replaces dispatch groups.
- How to model values over time with `AsyncSequence` and `AsyncStream`, and build retry, backoff, debounce, and polling patterns.
- How actors protect mutable state, why reentrancy matters, and how to design actor APIs that stay correct.
- How `@MainActor`, global actors, isolation domains, `Sendable`, region-based isolation, and `sending` fit together.
- How to turn on the Swift 6 language mode, read its diagnostics, use the approachable-concurrency settings of recent Swift versions, and migrate a legacy codebase incrementally.
- How to apply all of it in SwiftUI, Core Data, SwiftData, URLSession, and background refresh — and how to test, debug, and tune concurrent code.

How This Book Is Organized

Part I — Foundations of Swift Concurrency (Chapters 1–5) covers the move from GCD to async/await, bridging legacy APIs, tasks, cancellation, and timeouts. **Part II — Structured Concurrency in Practice** (Chapters 6–10) covers `async let`, task groups, async sequences and streams, async patterns, and URLSession's async APIs. **Part III — Actors and Isolation** (Chapters 11–15) covers actors, reentrancy, the main actor and global actors, isolation in depth, and `Sendable`. **Part IV — Swift 6 Data-Race Safety** (Chapters 16–19) covers region-based isolation, the Swift 6 language mode, approachable concurrency, and migrating a legacy codebase. **Part V — Concurrency in Real Apps** (Chapters 20–23) applies everything to SwiftUI, persistence, background work, and Skyvale's aggregation engine. **Part VI — Quality and Production** (Chapters 24–28) covers testing, debugging, performance, architecture, and a final 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: Skyvale

Throughout the book we build the concurrency core of **Skyvale**, a fictional weather dashboard for iPhone. For each city the user saves — Lisbon, Oslo, Seattle, Sydney, Tokyo, and Nairobi in the sample data — Skyvale asks three weather providers (Nimbus Weather, Stratus Open Data, and the streaming Zephyr Live) and one air-quality provider (AeroAir Quality) for current conditions, discards stale readings, and combines the rest into a single snapshot. Along the way it must cope with slow sources, a source that never answers, rate limits, retries, caching, live updates, air-quality alerts, offline data, and background refresh — exactly the problems that expose weak concurrency code. Skyvale also contains a legacy module written with dispatch queues and completion handlers, which we bridge

early and migrate to Swift 6 later. Because the providers are fictional, the app runs in a **Demo data** mode with fixed sample readings and simulated network latency, so every number in the book can be checked by hand.

Versions and Conventions

The code targets the **Swift 6 language mode** with **Xcode 26-era tooling** and a deployment target of **iOS 18**. Features that require iOS 26 are marked with availability checks and labeled "iOS 26+". Features introduced in the most recent Swift releases, such as the approachable-concurrency settings, are described as being available in "recent Swift versions"; proposals still under review are labeled as such and never presented as finished features. Unless a chapter says otherwise, the Skyvale target uses explicit `@MainActor` annotations rather than main-actor-by-default isolation; Chapter 18 explains both styles.

Swift cannot promise the order in which concurrent work finishes, so console output is shown as "something like" what you will see, and timings come from Skyvale's simulated latencies rather than real measurements. Compiler diagnostics are paraphrased; the exact wording varies between Swift versions.

> **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 data races, hangs, or crashes.

Code appears in `monospace` with a caption naming its file, such as `Skyvale/Services/ForecastService.swift`. Long lines are wrapped to fit the page. Type names are consistent throughout the book and summarized in Appendix B.

Contents

Preface

PART I — FOUNDATIONS OF SWIFT CONCURRENCY

1 Why Structured Concurrency: From GCD and Completion Handlers to async/await

2 async/await in Depth: Suspension Points, Async Functions, and Errors

3 Bridging Legacy Code: Continuations, Delegates, and Combine

4 Tasks: Creation, Priorities, Detached Tasks, and Task-Local Values

5 Cancellation: Cooperative Checks, Cancellation Handlers, and Timeouts

PART II — STRUCTURED CONCURRENCY IN PRACTICE

6 async let: Running a Fixed Number of Operations Concurrently

7 Task Groups: Dynamic Fan-Out, Partial Failure, and Limiting Concurrency

8 AsyncSequence and AsyncStream: Modeling Values over Time

9 Async Patterns: Retry, Backoff, Debounce, Polling, and Async Algorithms

10 Networking with URLSession's Async APIs: Requests, Streaming Bytes, and Downloads

PART III — ACTORS AND ISOLATION

11 Actors: Protecting Mutable State Without Locks

12 Actor Reentrancy and Designing Actor APIs

13 The Main Actor and Global Actors: Keeping UI Code Correct

14 Isolation in Depth: nonisolated, Isolated Parameters, #isolation, and Executors

15 Sendable: What Can Safely Cross Isolation Boundaries

PART IV — SWIFT 6 DATA-RACE SAFETY

16 Region-Based Isolation and the sending Keyword

17 The Swift 6 Language Mode: Strict Checking and Reading Diagnostics

18 Approachable Concurrency: Default Main-Actor Isolation, nonisolated(nonsending), and @concurrent

19 Migrating a Legacy Codebase to Swift 6, Step by Step

PART V — CONCURRENCY IN REAL APPS

20 SwiftUI and Concurrency: .task, @Observable Models, and Refresh

21 Core Data and SwiftData with Concurrency

22 Background Work: App Refresh, Scene Phases, and Long-Running Transfers

23 Caching, Deduplication, and Live Updates: The Skyvale Aggregation Engine

PART VI — QUALITY AND PRODUCTION

24 Testing Async Code with Swift Testing and XCTest

25 Debugging Concurrency: Instruments, Thread Sanitizer, and Logging

26 Performance: Contention, Main-Thread Hops, and the Cooperative Thread Pool

27 Concurrency Architecture: Dependencies, Error Strategy, and Team Guidelines

28 Case Study: Shipping Skyvale under Swift 6

APPENDICES

A Swift Concurrency Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations of Swift Concurrency

Why callbacks and dispatch queues stopped scaling, how async/await really works at its suspension points, how to bridge completion handlers, delegates, and Combine into async code, and how tasks, priorities, task-local values, cancellation, and timeouts fit together.

1 · Why Structured Concurrency

2 · async/await in Depth

3 · Bridging Legacy Code

4 · Tasks

5 · Cancellation and Timeouts

CHAPTER 1 · PART I — FOUNDATIONS OF SWIFT CONCURRENCY

Why Structured Concurrency: From GCD and Completion Handlers to async/await

Before we write a single `await`, we look honestly at the code most of us still maintain: completion handlers, dispatch queues, and shared caches guarded by good intentions. In this chapter you will see exactly where that style breaks down, what structured concurrency promises instead, and how Skyvale, the weather dashboard we build throughout the book, is set up so that every later chapter has a concrete, measurable starting point.

The Code We Already Have

Most iOS developers learned concurrency one tool at a time. First came `DispatchQueue.main.async` to get back to the main thread after a network call. Then a private serial queue to protect a dictionary. Then `DispatchGroup` to wait for three requests at once, `OperationQueue` for dependencies, a `DispatchSemaphore` to limit how many downloads run together, and finally Combine to glue streams of values to the UI. Each tool solves a real problem, and each one works when it is used carefully. The trouble is that nothing checks whether it *was* used carefully. The rules live in comments, in code review, and in the heads of the people who wrote the code.

Skyvale, our running example, has exactly this kind of history. Before the rewrite you will follow in this book, the app talked to its first weather provider through a class called `LegacyWeatherClient`. It still exists in the project's `Legacy` folder, because real migrations never happen in one afternoon. Here is the heart of it, as it was written years ago in the Swift 5 language mode.

Skyvale/Legacy/LegacyWeatherClient.swift (original version, Swift 5 mode)

```
import Foundation

final class LegacyWeatherClient {
    static let shared = LegacyWeatherClient()
    private let queue = DispatchQueue(label: "com.example.skyvale.legacy")
    private var cache: [String: WeatherReading] = [:]

    func fetchConditions(for city: City,
                         completion: @escaping (Result<WeatherReading, Error>) -> Void) {
        queue.async {
            if let cached = self.cache[city.id] {
                DispatchQueue.main.async { completion(.success(cached)) }
                return
            }
            let request = self.makeRequest(for: city)
            URLSession.shared.dataTask(with: request) { data, response, error in
```

```
                if let error {
                    DispatchQueue.main.async { completion(.failure(error)) }
                    return
                }
                guard let data else { return }          // BUG: completion never called
                do {
                    let reading = try self.decodeReading(data, for: city)
                    self.queue.async { self.cache[city.id] = reading }
                    DispatchQueue.main.async { completion(.success(reading)) }
                } catch {
                    DispatchQueue.main.async { completion(.failure(error)) }
                }
            }.resume()
        }
    }

    func cachedReading(for cityID: String) -> WeatherReading? {
        cache[cityID]                                   // BUG: read outside `queue`
    }

    private func makeRequest(for city: City) -> URLRequest { ... }
    private func decodeReading(_ data: Data,
                               for city: City) throws -> WeatherReading { ... }
}
```

Read it slowly, the way a reviewer would. There is nothing exotic here. The class uses a private serial queue to protect its cache, hops back to the main queue before calling the completion handler, and maps errors into a `Result`. It is the code you would find in thousands of apps. It also contains two bugs that no compiler of its time would report, and a design that makes several other bugs easy to write in the code that calls it. Let's name them.

Six Ways Callback Code Goes Wrong

1. The pyramid of doom

One request with a completion handler is pleasant. The trouble starts when one result feeds the next. Skyvale's dashboard needs current conditions from three weather providers and an air-quality reading for every city, then combines them into a single snapshot. Written with callbacks, even the sequential version for one city drifts to the right:

Playground-style snippet (illustrative legacy code)

```
func loadSnapshot(for city: City,
                  completion: @escaping (Result<CitySnapshot, Error>) -> Void) {
    fetchNimbus(for: city) { nimbusResult in
        switch nimbusResult {
        case .failure(let error): completion(.failure(error))
        case .success(let nimbus):
            fetchStratus(for: city) { stratusResult in
                switch stratusResult {
                case .failure(let error): completion(.failure(error))
```

```
                case .success(let stratus):
                    fetchZephyr(for: city) { zephyrResult in
                        // ... and one more level for air quality,
                        // then aggregation, then the UI update
                    }
                }
            }
        }
    }
}
```

The indentation is the visible problem; the invisible one is control flow. Error handling is repeated at every level, a `return` only leaves the innermost closure, and loops are almost impossible: "fetch these six cities one after another" becomes a recursive helper function or a hand-built state machine. Every developer who has written this code has, at some point, called `completion` twice by accident because two branches both fell through to a final call.

2. Forgotten completion calls

Look again at `guard let data else { return }` inside `fetchConditions`. If the session ever reports neither data nor an error, the function simply ends. The caller's completion handler is never called, so the spinner on screen spins forever and nothing is logged. The compiler cannot help, because from its point of view `completion` is just a closure value that you may or may not choose to call. In Chapter 3 we fix this exact line before wrapping the method with a continuation, because left as it is, it would turn the stuck spinner into a task that hangs forever.

3. Wrong-thread UI updates

`URLSession`'s completion handlers run on the session's delegate queue, which is not the main queue. The legacy client remembers to hop to the main queue before calling `completion`, but that is a convention, not a guarantee. Callers cannot tell from the signature which thread they will be called on, so defensive developers wrap their handlers in another `DispatchQueue.main.async`, and less defensive ones update a label from a background thread. The Main Thread Checker catches some of these at runtime, in debug builds, when the path happens to execute.

4. Cancellation by flags

When the user leaves the city detail screen, its requests should stop. The callback world has no shared idea of cancellation. `URLSessionTask` has `cancel()`, `Operation` has `isCancelled`, `DispatchWorkItem` has its own `cancel()`, and our own classes usually grow a `private var isCancelled = false` that someone remembers to check in some of the callbacks. None of these connect to each other. Cancelling a "refresh" means finding every piece of in-flight work it started and stopping each one by hand.

5. Data races on shared caches

`cachedReading(for:)` reads `cache` directly, from whichever thread calls it, while `fetchConditions` writes the same dictionary on `queue`. Two threads touching a Swift dictionary at the same time, with at least one of them writing, is a data race: undefined behavior that may corrupt memory, crash in an unrelated place, or appear to work for months. The fix in GCD terms is easy (`queue.sync { cache[cityID] }`), but nothing told us it was needed. Thread Sanitizer can find this race, but only if a test or a manual run happens to execute both paths at the same moment.

6. Thread explosion

Dispatch queues hide threads, but threads still exist. When work submitted to a concurrent queue blocks, waiting on a semaphore, a lock, synchronous I/O, or `DispatchGroup.wait()`, GCD may bring up additional threads so that other work can make progress. An app that fans out many blocking tasks can therefore end up with far more threads than the device has CPU cores, each with its own stack and its own context-switching cost. Skyvale's old `WeatherManager` did exactly this with a semaphore-based limiter; Chapter 19 migrates it, and Chapter 26 explains why the Swift runtime takes a different approach.

> **NOTE**
>
> None of these problems mean GCD is broken. Dispatch queues remain an excellent low-level tool, and Swift's concurrency runtime on Apple platforms is itself built on top of Dispatch. The issue is that GCD gives you mechanisms without a model: it cannot tell you who owns a piece of work, when it ends, or which data it may touch. That is what the rest of this book adds.

The Same Work, Written Structurally

Here is the one-city load again, written with the async APIs that Skyvale's sources expose. Do not worry about every keyword yet; Chapter 2 takes this apart line by line.

Playground-style snippet

```
func loadSnapshot(for city: City,
                  sources: [any WeatherSource],
                  air: any AirQualitySource,
                  aggregator: AggregationService) async throws -> CitySnapshot {
    var readings: [WeatherReading] = []
    for source in sources {
        readings.append(try await source.currentConditions(for: city))
    }
    let quality = try await air.airQuality(for: city)
    return try aggregator.aggregate(city: city, weather: readings, air: quality,
                                    failedSources: [], now: Date())
}
```

Several things changed at once. The code reads top to bottom. A `for` loop works again. Errors propagate with `try`, so there is one error path instead of four. The function either returns a value or throws; it cannot silently forget to do either, because the compiler enforces that every path returns or throws, exactly as it does for synchronous code. And there is no queue in sight: where this code runs is decided by its isolation, which we will make explicit in later chapters.

This version is still *sequential*. It waits for Nimbus, then Stratus, then Zephyr, then AeroAir. Making it concurrent is the subject of Chapters 6 and 7, and doing so will not require going back to callbacks. That is the real promise: the readable shape of the code survives when you add concurrency.

What async/await does not fix by itself

It is worth being precise about the limits of the syntax, because teams that migrate "mechanically" are often disappointed. Rewriting a completion handler as an `async` function does not make anything run in parallel; the loop above is exactly as slow as the callback chain. It does not decide where code runs; that is the job of isolation, which you will learn to read in Chapters 13 and 14. It does not protect shared mutable state either. If the async version of `LegacyWeatherClient` still kept a plain dictionary that several tasks read and write, the race would still be there; the difference is that, in the Swift 6 language mode, the compiler would now refuse to build it until you protected the state, usually by moving it into an actor.

What async/await does fix is the *shape* of asynchronous code. Once the shape is ordinary, the compiler can reason about it: it can check that every path returns or throws, track which values cross from one task to another, and know where each suspension point is. Those guarantees are the foundation the rest of the model builds on, and they are why we start the book with syntax before moving on to tasks and actors.

One more detail in the snippet deserves a comment: it passes `now: Date()` to the aggregator. In Demo data mode that would mark every fixture reading as stale, because the sample data is pinned to a fixed reference time; real Skyvale code injects the current time instead of reading the clock directly, which is also what makes it testable. You will see that pattern from Chapter 2 onward.

What Structured Concurrency Promises

"Structured" is borrowed from structured programming, the idea that control flow should have a clear entry and exit, like a block or a function call, rather than arbitrary jumps. Structured concurrency applies the same idea to concurrent work: when a function starts concurrent child work, that work must finish before the function returns. The result is a tree of tasks whose shape mirrors the shape of your code.

Task tree for one dashboard refresh (from Chapter 7)

```
refresh (DashboardModel, main actor)
 +-- snapshots(for: 6 cities)           task group
      +-- Lisbon                        child task
      |    +-- aeroAir (async let)   nimbus stratus zephyr (group)
```

```
        +-- Oslo
        |    +-- aeroAir (async let)   nimbus stratus zephyr (group)
        +-- ... Seattle, Sydney, Tokyo, Nairobi
```

From that tree come three concrete guarantees:

- **Lifetimes are scoped.** A child task cannot outlive the scope that created it. When `snapshots(for:)` returns, every city's work has finished, failed, or been cancelled. There are no stray callbacks arriving after the screen is gone.
- **Cancellation propagates automatically.** Cancelling the refresh cancels every child below it, all the way down to the network requests. Nobody has to keep a list of in-flight operations.
- **Errors propagate upward.** A child's error flows to its parent with `try`, where the parent decides whether to fail, retry, or record a partial result.

Alongside the task tree, the Swift 6 language mode adds a fourth guarantee that GCD never could: **compile-time data-race checking**. The compiler knows which isolation domain each piece of code runs in and which values cross between domains, and it refuses to compile code like `cachedReading(for:)` above, where mutable state is reachable from more than one domain without protection. Chapters 15 to 17 cover how it does this and how to read its diagnostics.

> **TIP**
>
> When you meet an unfamiliar piece of concurrent code, ask three questions before reading the details: *Who owns this state? Which isolation domain runs this code? Where are the suspension points?* Every chapter in this book returns to these questions, and they are a fast way to review a pull request.

The Three Pillars

Swift's concurrency model is easiest to learn as three cooperating features, each of which replaces a family of GCD habits.

Pillar	What it gives you	Replaces	Chapters
async/await	Asynchronous code with ordinary control flow	Completion handlers, nested callbacks	2, 3, 10
Structured tasks	Child tasks, cancellation, timeouts	`DispatchGroup`, flags, work items	4 to 9
Actors and Sendable	Isolated mutable state, checked at compile time	Serial queues, locks, conventions	11 to 19

The pillars depend on each other. `await` marks the places where a task may suspend, which is exactly where an actor may let other work in. Tasks carry cancellation and priority across those suspension points. `Sendable` describes which values may travel between tasks and actors safely. You can start

using async/await without understanding actors, but you cannot reason about correctness without all three, so Part I covers the first two pillars thoroughly before Part III turns to actors.

Meet Skyvale

Skyvale is a weather dashboard for iPhone. For each of the user's saved cities it collects current conditions from three weather providers and one air-quality provider, combines them into a single snapshot, caches it, streams live updates, raises alerts when air quality is poor or when the providers disagree, saves the last good data for offline launch, and refreshes in the background. It is a small app with a surprisingly complete set of concurrency problems, which is why we chose it.

Four fictional providers

All four providers are fictional and live on reserved `.example` domains, which means they do not exist and never will. We show live network clients for realism, but you run Skyvale in **Demo data** mode, where fixture sources answer from built-in sample data with simulated latencies.

SourceID	Display name	Kind	Endpoint host
`.nimbus`	Nimbus Weather	weather + hourly	`api.nimbus.example`
`.stratus`	Stratus Open Data	weather	`data.stratus.example`
`.zephyr`	Zephyr Live	weather, NDJSON stream	`stream.zephyr.example`
`.aeroAir`	AeroAir Quality	air quality	`api.aeroair.example`

Nimbus and AeroAir require an API key sent in an `X-Api-Key` header. The keys are placeholders (`YOUR_NIMBUS_KEY` and `YOUR_AEROAIR_KEY`) read from `Config.xcconfig` through the Info.plist keys `NimbusAPIKey` and `AeroAirAPIKey`. Demo data mode never reads them.

The models you will see everywhere

Skyvale's models are small value types, and every one of them is `Sendable`. That choice is deliberate and pays off in every chapter: values can be handed from one task or actor to another without any locking, because each recipient gets its own copy. Here are the three you will meet first.

Skyvale/Models/City.swift

```
struct City: Identifiable, Hashable, Codable, Sendable {
    let id: String            // "SKV-LIS"
    let name: String
    let countryCode: String   // ISO 3166-1 alpha-2, "PT"
    let latitude: Double
    let longitude: Double
    let timeZoneID: String    // "Europe/Lisbon"
}
```

Skyvale/Models/SourceID.swift

```
enum SourceID: String, Codable, CaseIterable, Sendable {
    case nimbus, stratus, zephyr, aeroAir      // allCases order = display/priority order

    var displayName: String {
        switch self {
        case .nimbus: "Nimbus Weather"
        case .stratus: "Stratus Open Data"
        case .zephyr: "Zephyr Live"
        case .aeroAir: "AeroAir Quality"
        }
    }

    static let weatherSources: [SourceID] = [.nimbus, .stratus, .zephyr]
}
```

Skyvale/Models/WeatherReading.swift

```
enum Condition: String, Codable, Sendable {
    case clear, partlyCloudy, cloudy, rain, thunderstorm, snow, fog
}

struct WeatherReading: Hashable, Codable, Sendable {
    let cityID: City.ID
    let source: SourceID
    let observedAt: Date
    let temperatureC: Double
    let humidityPercent: Int
    let windKph: Double
    let condition: Condition
}
```

The combined result for a city is a `CitySnapshot`: mean temperature, humidity, and wind across the usable readings, the condition from the first usable source, the spread between the warmest and coolest reading, the air-quality index and its category, and two lists recording which sources were used and which were missing. Chapter 6 builds the `AggregationService` that computes it.

Six cities and a reference time

The sample data is pinned to a reference time of 2026-03-14 09:00 UTC, so that results are reproducible. The six cities, in this order, are also the default dashboard list:

ID	Name	Country	Time zone
SKV-LIS	Lisbon	PT	Europe/Lisbon
SKV-OSL	Oslo	NO	Europe/Oslo
SKV-SEA	Seattle	US	America/Los_Angeles
SKV-SYD	Sydney	AU	Australia/Sydney

SKV-TYO	Tokyo	JP	Asia/Tokyo
SKV-NBO	Nairobi	KE	Africa/Nairobi

Two cities are deliberately awkward. Oslo's Stratus reading was observed 40 minutes before the reference time, so it is stale and must be excluded. Tokyo's Zephyr source never answers at all, so it must be timed out. A third quirk is good news: Tokyo's air-quality index is 104, which is "unhealthy for sensitive groups" and raises Skyvale's only air-quality alert in the sample data. You will meet these three cases again and again, because they are exactly the cases that separate a demo from an app: stale data, a hanging dependency, and an event the user must hear about.

Demo data mode

Skyvale's dependencies are gathered in a single `AppEnvironment` value with two factory methods: `AppEnvironment.live()`, which wires the network sources, and `AppEnvironment.demo(simulateLatency:)`, which wires fixture sources. The default build uses demo data. With latency simulation on, the fixture sources wait before answering, to give the app realistic timing:

Source	Simulated latency	Notes
Nimbus	800 ms	Also provides hourly forecasts
Stratus	1,200 ms	Oslo's reading is stale
Zephyr	2,000 ms	Tokyo never answers
AeroAir	500 ms	Air quality for all six cities

These numbers are simulated delays built into the fixtures, not measurements of any real service. They are chosen to make the effect of each concurrency technique visible, and every timing in this book is computed from them. In tests we pass `simulateLatency: false` so that the fixtures answer immediately.

Setting Up the Project

Skyvale is a single iOS app target named `Skyvale` in `Skyvale.xcodeproj`, with a test target `SkyvaleTests` and the bundle identifier `com.example.skyvale`. The source tree follows the layout below; you will add to each folder as the book progresses.

Project layout (abbreviated)

```
Skyvale/
  App/            SkyvaleApp.swift, AppEnvironment.swift
  Models/         City, SourceID, WeatherReading, AirQualityReading, CitySnapshot, ...
  Networking/     HTTPClient.swift, APIConfig.swift
  Sources/        WeatherSource.swift, NimbusSource.swift, ..., FixtureSources.swift
  Concurrency/    Sleeper.swift, Timeout.swift, Retry.swift, AsyncBroadcaster.swift
```

```
  Services/       AggregationService.swift, ForecastService.swift, ForecastCache.swift
  Persistence/    SwiftData models, SnapshotStore.swift, legacy Core Data controller
  Features/       Dashboard/, CityDetail/, Search/, Settings/
  Background/     RefreshScheduler.swift
  Support/        Logging.swift
  Legacy/         LegacyWeatherClient.swift, RadarTileLoader.swift, WeatherManager.swift
SkyvaleTests/     AggregationServiceTests.swift, ForecastServiceTests.swift, ...
```

The build settings this book assumes

Concurrency behavior depends on a handful of build settings, and since recent versions of Xcode some of them have new defaults. Every example in this book assumes the following values for the `Skyvale` target:

Setting	Skyvale value
Swift Language Version	Swift 6
iOS Deployment Target	iOS 18.0
Default Actor Isolation	nonisolated (explicit `@MainActor`)
Approachable Concurrency	Off in Chapters 1 to 17; discussed in Chapter 18
Strict Concurrency Checking	Complete (implied by Swift 6)

We use Xcode 26-era tooling, which ships Swift 6.2. The Swift 6 language mode turns data-race-safety problems into compile errors; the iOS 18 deployment target gives us the `Synchronization` module, Observation, and SwiftData without availability checks. Features that need iOS 26 are always wrapped in `if #available(iOS 26, *)` and labeled "iOS 26+".

The two Xcode 26 settings in the middle of the table deserve attention now, because they change how code you copy from this book behaves. **Default Actor Isolation** decides whether declarations without an explicit annotation are isolated to the main actor or are nonisolated. **Approachable Concurrency** turns on a group of upcoming language features, including one that changes where nonisolated async functions run. New projects created with Xcode 26 may enable main-actor default isolation and Approachable Concurrency, so check your target's build settings before you start. To make examples readable regardless of your defaults, Skyvale keeps both off and writes `@MainActor` explicitly wherever it matters. Chapter 18 shows what changes when you switch them on, and Chapter 28 explains the final choice for the shipped app.

1. Select the project in the navigator, then the `Skyvale` target, then the **Build Settings** tab with **All** and **Combined** selected.
2. Type "concurrency" into the search field. Set **Approachable Concurrency** to **No** and confirm **Strict Concurrency Checking** shows Complete.
3. Search for "isolation" and set **Default Actor Isolation** to **nonisolated**.
4. Search for "Swift Language Version" and set it to **Swift 6**. Repeat for `SkyvaleTests`.

Exact labels and their positions can differ slightly between Xcode versions; if a label is not where you expect it, the search field is the reliable route.

> **WARNING**
>
> Do not copy concurrency code from any source, including this book, into a project without knowing its Default Actor Isolation setting. The same unannotated class is main-actor isolated in one project and nonisolated in another, and the same `nonisolated` async function can run on a different executor depending on Approachable Concurrency. The code may compile in both projects and behave differently.

About the legacy folder

The `Legacy` folder holds the old code we bridge and migrate: `LegacyWeatherClient` (completion handlers and a private queue), `RadarTileLoader` (a delegate-based loader for nine radar tiles per city), `LegacySettingsStore` (Combine and `@Published`), and `WeatherManager`, a singleton "god object" that fans out with `DispatchGroup`, limits requests with a semaphore, broadcasts with `NotificationCenter`, and refreshes with a `Timer`. Under the Swift 6 language mode, some of that code no longer compiles as written. Chapter 3 shows the small, documented changes that let it build while we bridge it, and Chapter 19 migrates it properly.

The Baseline: 29 Seconds

Every optimization needs a baseline. If Skyvale fetched everything one request at a time, as the callback-chain version effectively did, a full dashboard refresh would cost the sum of all latencies. Each city needs four requests:

Ideal sequential timing (simulated latency)

```
One normal city:  Nimbus 0.8 + Stratus 1.2 + Zephyr 2.0 + AeroAir 0.5 = 4.5 s
Tokyo:            Nimbus 0.8 + Stratus 1.2 + Zephyr timeout 4.0 + AeroAir 0.5 = 6.5 s

Six cities:       5 x 4.5 s + 6.5 s = 29.0 s for 24 requests
```

Oslo's stale Stratus reading still costs its 1.2 seconds, because we only discover that it is stale after it arrives. Tokyo's Zephyr request costs the full four-second per-attempt timeout that Skyvale applies to every source. These are ideal numbers that ignore overhead; a real run takes something like this, slightly more. Twenty-nine seconds is, of course, unusable. Here is where the book takes it:

Strategy (24 requests)	Ideal wall time	Chapter
Fully sequential	29.0 s	1, 2
`async let` per city, cities one after another	14.0 s	6
Task group over cities, `async let` inside	4.0 s (five cities at 2.0 s)	7