IOS DEVELOPMENT

iOS Development — Architecture & Data

MVVM architecture, structured concurrency with async/await/actors/@MainActor, networking in depth with URLRequest/Codable/custom errors, real SwiftData persistence, dependency injection and testability with the real Swift Testing framework, UI testing and Xcode Instruments, device capabilities, and Combine vs. async/await reactive patterns — grounded throughout in real, current Apple APIs and versions (the real 2005 MVVM origin, Swift 5.5's actors, iOS 17's SwiftData and Observation framework, Xcode 16's Swift Testing framework, iOS 16's PhotosPicker) — closing with a capstone evolving Fundamentals' own TaskFlow into a real, persisted, tested, production-shaped app.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: APP ARCHITECTURE: MVVM & WHY IT FITS SWIFTUI

iOS Development — Architecture & Data

Chapter 1 · App Architecture: MVVM & Why It Fits SwiftUI

Fundamentals' own capstone, TaskFlow, worked — but its `TaskStore` quietly did two genuinely different jobs at once: holding raw task data, and deciding what a screen should display. This course opens by giving that split a real name, **MVVM**, and a deliberate structure.

What MVVM Actually Is

MVVM — **Model-View-ViewModel** — is a real, well-established UI architecture pattern originally developed at Microsoft by architects Ken Cooper and Ted Peters, and publicly announced by fellow Microsoft architect John Gossman on his blog in **2005**, as a variant of Martin Fowler's own earlier Presentation Model pattern.

Model

Plain data — `Task` from Fundamentals, unchanged. No UI knowledge at all.

ViewModel

Holds state and logic for one screen — fetching, validating, transforming Models into what the View needs.

View

A real SwiftUI `View` — displays what the ViewModel exposes, forwards user actions back to it.

AN HONEST, IMPORTANT DISTINCTION

MVVM is **not** an official Apple-mandated architecture — Apple's own documentation and sample code have never formally required one specific pattern for SwiftUI apps. MVVM is a real, widely adopted **community convention** that happens to map unusually cleanly onto SwiftUI's own tools, covered next — not a rule enforced anywhere by the compiler or the framework itself.

Why It Fits SwiftUI So Naturally

MVVM's own classic problem, in older UI frameworks, was wiring a View up to a ViewModel's changes by hand. SwiftUI's real Observation framework (Fundamentals Chapter 6) removes that problem almost entirely — an `@Observable` class **is** a real, natural ViewModel, and SwiftUI's

own view-diffing mechanism handles the "notify the View when something changes" job that MVVM traditionally needed a separate binding system for.

MVVM ROLE	REAL SWIFTUI TOOL
Model	A plain struct — `Identifiable` / `Codable` as needed, no framework dependency
ViewModel	An `@Observable` class (iOS 17/Swift 5.9) — real, automatic change tracking, no manual `@Published`
View	A real SwiftUI `View` struct, holding its ViewModel via `@State` or receiving it via `@Bindable`

Refactoring TaskFlow: A Real Before/After

Fundamentals' own `TaskStore` mixed a genuinely raw data array with screen-specific concerns — nothing was wrong with it for a 10-chapter capstone, but it doesn't scale cleanly once a real app grows past one screen's worth of logic.

BEFORE — TASKSTORE.SWIFT (FUNDAMENTALS, CHAPTER 10)

```
@Observable
final class TaskStore {
    var tasks: [Task] = [ /* ... */ ]

    func add(title: String, priority: Int) { /* ... */ }
    func toggleDone(for task: Task) { /* ... */ }
}
```

The real MVVM version keeps `Task` itself as a pure Model, and gives the task-list screen its own dedicated ViewModel — a real, meaningful rename, not just cosmetic, since it now explicitly owns **this screen's own** logic rather than being a single, all-purpose store:

AFTER — TASKLISTVIEWMODEL.SWIFT

```
@Observable
final class TaskListViewModel {
    private(set) var tasks: [Task] = [ /* ... */ ]
```

```
    var incompleteCount: Int {
        tasks.filter { !$0.isDone }.count
    }

    func add(title: String, priority: Int) {
        tasks.append(Task(title: title, priority: priority))
    }

    func toggleDone(for task: Task) {
        guard let index = tasks.firstIndex(where: { $0.id == task.id }) else { return }
        tasks[index].isDone.toggle()
    }
}
```

A REAL, GENUINE IMPROVEMENT, NOT JUST A RENAME

`incompleteCount` is a real, new computed property — exactly the kind of view-specific derived state MVVM says belongs on the ViewModel, not scattered as ad hoc logic inside a View's own `body`. And `private(set) var tasks` is a real, meaningful access restriction: any View can *read* the task list, but only the ViewModel's own methods can *change* it — genuinely enforced by the compiler, not just a convention.

The View itself barely changes — it still holds the object via `@State` / `@Bindable` exactly as Fundamentals Chapter 6 covered — only the type name changes, reflecting what it now genuinely represents:

```
struct TaskListView: View {
    @Bindable var viewModel: TaskListViewModel

    var body: some View {
        List(viewModel.tasks) { task in
            Text(task.title)
        }
        .navigationTitle("\(viewModel.incompleteCount) remaining")
    }
}
```

THE REAL RULE OF THUMB

If a View's own `body` needs an `if` / `switch` to decide *what data* to show (not just how to lay it out), that's a real, genuine sign the logic belongs on the ViewModel instead — `incompleteCount` above is exactly that kind of decision, moved out of the View.

Hands-On Exercises

EXERCISE 1

Add a real computed property `highPriorityTasks: [Task]` to `TaskListViewModel`, filtering `tasks` to only those with `priority == 3`, and use it in `TaskListView` to show a separate `Section("High Priority")` above the main task list.

View solution

EXERCISE 2

Explain, in your own words, what real problem `private(set) var tasks` prevents that a plain `var tasks` would allow, and why that matters once an app has more than one View sharing the same ViewModel.

View solution

EXERCISE 3

Explain, in your own words, why MVVM being a "community convention rather than an Apple mandate" is a genuinely important distinction to understand before adopting it in a real project, rather than treating it as a rule the compiler or SwiftUI itself enforces.

View solution

CHAPTER 1 QUICK REFERENCE

- MVVM: real 2005 Microsoft origin (Cooper, Peters, announced by Gossman), a variant of Fowler's Presentation Model — a community convention, not an Apple mandate
- Model = plain data, ViewModel = state/logic for one screen, View = a real SwiftUI `View`

- `@Observable` (iOS 17/Swift 5.9) makes a plain class a natural, real ViewModel — no manual `@Published` wiring needed
- `private(set) var` is a real, compiler-enforced way to let Views read but not directly mutate ViewModel state
- A rule of thumb: view-specific decision logic (filtering, computed summaries) belongs on the ViewModel, not inside a View's own `body`

CHAPTER 2: CONCURRENCY IN SWIFT: ASYNC/AWAIT, TASKS & ACTORS

iOS Development — Architecture & Data

Chapter 2 · Concurrency in Swift: async/await, Tasks & Actors

Fundamentals Chapter 9 covered `async` / `await` just enough to fetch one real thing. A ViewModel doing genuine work needs more: running several async operations at once, protecting shared mutable state from real data races, and guaranteeing UI-touching code actually runs on the main thread.

Unstructured Work: `Task {}`

Fundamentals' own `.task {}` view modifier is tied directly to a view's own lifecycle. A ViewModel method — not itself a view — needs a different real starting point: `Task { }`, which launches a genuine, independent unit of async work immediately, not bound to any one view's own appear/disappear cycle.

```
func refresh() {
    Task {
        do {
            tasks = try await fetchTasksFromServer()
        } catch {
            print("Refresh failed: \(error)")
        }
    }
}
```

A REAL, GENUINE TRADE-OFF

Unlike `.task {}`, a plain `Task { }` isn't automatically cancelled by anything — Fundamentals Chapter 9's own real safety guarantee doesn't apply here for free. A long-lived ViewModel method that starts background work this way is genuinely responsible for its own cancellation if that's ever needed (Course 3's own deployment chapters return to this).

Structured, Parallel Work: `async let`

`await`-ing two calls back to back runs them **sequentially** — the second doesn't start until the first finishes. `async let` is real, genuine structured concurrency: both operations start immediately, running in parallel.

```
func loadDashboard() async throws -> (tasks: [Task], quote: Quote) {
    async let tasks = fetchTasksFromServer()
    async let quote = fetchQuote()

    return try await (tasks, quote)   // waits for both, but they ran in parallel
}
```

REAL, AUTOMATIC STRUCTURE

Both child operations are genuinely tied to the surrounding function's own lifetime — if `loadDashboard()` is cancelled while both are still running, real, automatic cancellation propagates to both, with no manual bookkeeping needed. This is what "structured" means in structured concurrency: child work can never outlive its own parent scope.

Actors: Protecting Mutable State from Real Data Races

Introduced alongside `async`/`await` in **Swift 5.5**, an `actor` is a real reference type — like a class — that the compiler guarantees only ever executes one piece of its own code at a time, even when called from multiple concurrent tasks at once.

```
actor RequestCounter {
    private(set) var count = 0

    func increment() {
        count += 1
    }
}
```

THE REAL PROBLEM THIS SOLVES

A plain `class` with a shared, mutable `var count` — called concurrently from several real tasks at once — genuinely risks a data race: two increments reading the same starting value before either

writes back, silently losing an update. An `actor`'s own serialized access makes that specific class of bug structurally impossible, enforced by the compiler, not just careful coding.

Calling an actor's own method or property from **outside** the actor requires `await`, even though nothing about `increment()` itself looks asynchronous — the `await` here reflects real, potential waiting for the actor's own serialized turn, not network latency.

`@MainActor`: Guaranteeing the Main Thread

UIKit and SwiftUI both genuinely require UI updates to happen on the main thread — a real, hard rule, not a suggestion. `@MainActor` is Swift's own real, compiler-enforced way of guaranteeing that.

```
@Observable
@MainActor
final class TaskListViewModel {
    private(set) var tasks: [Task] = []

    func refresh() {
        Task {
            let fetched = try? await fetchTasksFromServer()
            tasks = fetched ?? []   // real, guaranteed-safe: this class is @MainActor
        }
    }
}
```

A REAL, GENUINE RISK WITHOUT IT

A ViewModel updating `@Observable` state — state SwiftUI reads to redraw the UI — from a background thread is real, genuinely undefined behavior: at best a visual glitch, at worst a crash. Marking a UI-facing ViewModel `@MainActor` makes the compiler itself enforce that every one of its own property writes happens on the main thread, closing off that entire class of bug at compile time rather than leaving it to be discovered at runtime.

Hands-On Exercises

EXERCISE 1

Write a function `loadProfile()` using `async let` to fetch a user's real name and avatar image URL in parallel (two separate, imagined async functions), then `await` both together and return a combined result.

View solution

EXERCISE 2

Define a real `actor` called `DownloadTracker` with a `private(set) var activeDownloads = 0` and two methods, `start()` and `finish()`, incrementing and decrementing it. Explain, in your own words, why calling `tracker.activeDownloads` from outside the actor requires `await`.

View solution

EXERCISE 3

Explain, in your own words, why marking a ViewModel `@MainActor` is specifically about *where code runs*, while marking a type an `actor` is specifically about *serializing access to shared state* — and why a typical SwiftUI ViewModel usually needs the former rather than the latter.

View solution

CHAPTER 2 QUICK REFERENCE

- `Task { }` — unstructured async work, not tied to a view's lifecycle, and not automatically cancelled the way `.task {}` is
- `async let` — real, structured, parallel concurrency; child work can't outlive its own parent scope
- `actor` (Swift 5.5, 2021) — a reference type with compiler-enforced serialized access, preventing real data races on shared mutable state
- Calling an actor's own members from outside it requires `await`, reflecting a real wait for serialized access, not necessarily network latency

- `@MainActor` — compiler-enforced guarantee that a type's own code runs on the main thread, essential for any ViewModel updating UI-facing state

CHAPTER 3: NETWORKING IN DEPTH: REST APIS, CODABLE & ERROR HANDLING

iOS Development — Architecture & Data

Chapter 3 · Networking in Depth: REST APIs, Codable & Error Handling

Fundamentals Chapter 9 covered a single, real GET request. Real apps need to send data too — creating, updating, deleting — and need real, descriptive errors when something goes wrong, not a generic failure. This chapter covers both, plus a small architectural pattern for keeping networking code out of ViewModels entirely.

`URLRequest`: Full Control Over an HTTP Request

`URLSession.shared.data(from:)` — Fundamentals' own tool — only issues a plain GET. `URLRequest` is the real, general-purpose struct needed for anything else: setting the HTTP method, real custom headers, and a real request body.

```
var request = URLRequest(url: url)
request.httpMethod = "POST"
request.setValue("application/json", forHTTPHeaderField: "Content-Type")
request.httpBody = try JSONEncoder().encode(newTask)

let (data, response) = try await URLSession.shared.data(for: request)
```

TOOL	REAL CAPABILITY
`URLSession.shared.data(from: url)`	GET only, no headers, no body — Fundamentals' own simple case
`URLSession.shared.data(for: request)`	Any real HTTP method, real custom headers, a real request body via `URLRequest`

Encoding a Request Body with `Encodable`

Fundamentals used `Decodable` to turn JSON into a struct. Sending data back needs the reverse — `Encodable`, or the combined `Codable` typealias covering both directions at once:

```swift
struct NewTaskRequest: Encodable {
    let title: String
    let priority: Int
}

let body = NewTaskRequest(title: "Buy milk", priority: 2)
request.httpBody = try JSONEncoder().encode(body)
```

Real JSON Key Mismatches: `CodingKeys` & `.convertFromSnakeCase`

Real-world APIs rarely use Swift's own camelCase convention — `snake_case` is genuinely common. Two real, complementary tools handle this:

```swift
// Option 1: automatic, applies to every property
let decoder = JSONDecoder()
decoder.keyDecodingStrategy = .convertFromSnakeCase

// Option 2: manual, per-property — needed for real exceptions
struct Task: Decodable {
    let title: String
    let dueDate: Date?

    enum CodingKeys: String, CodingKey {
        case title
        case dueDate = "due_date"
    }
}
```

A REAL, PRACTICAL RULE

`.convertFromSnakeCase` is genuinely convenient when an entire API consistently uses `snake_case` — one line, every property. `CodingKeys` is the real fallback for exceptions: a specific key that doesn't follow the pattern, or a Swift property name that needs to differ from its JSON key for a reason beyond casing alone.

Real, Descriptive Errors

Throwing a generic `URLError` for every possible failure tells a caller almost nothing useful. A real, custom error type conforming to `LocalizedError` can carry genuine, specific meaning:

```
enum APIError: Error, LocalizedError {
    case invalidURL
    case requestFailed(statusCode: Int)
    case decodingFailed

    var errorDescription: String? {
        switch self {
        case .invalidURL:
            return "The request URL was invalid."
        case .requestFailed(let statusCode):
            return "The server returned status \(statusCode)."
        case .decodingFailed:
            return "The server's response couldn't be understood."
        }
    }
}
```

A REAL, CONCRETE PAYOFF

A ViewModel catching `APIError.requestFailed(statusCode: 404)` can show a genuinely specific, useful message — "Task not found" — instead of a generic "Something went wrong," and can react differently to a `401` (real, likely an expired login) than a `500` (a real server problem, not the user's own fault).

A Centralized `APIClient`

Repeating `URLSession`/`JSONDecoder`/status-code-checking logic inside every ViewModel is real, genuine duplication. A small, shared `APIClient` — following Chapter 1's own separation-of-concerns discipline — centralizes it once:

```
struct APIClient {
    func fetch<T: Decodable>(_ type: T.Type, from url: URL) async throws -> T {
        let (data, response) = try await URLSession.shared.data(from: url)

        guard let httpResponse = response as? HTTPURLResponse else {
```

```
            throw APIError.requestFailed(statusCode: 0)
        }
        guard (200..<300).contains(httpResponse.statusCode) else {
            throw APIError.requestFailed(statusCode: httpResponse.statusCode)
        }

        do {
            return try JSONDecoder().decode(T.self, from: data)
        } catch {
            throw APIError.decodingFailed
        }
    }
}
```

WHY `200..<300`, NOT JUST `== 200`

A real, correctly-behaving server can succeed with several different real status codes in the 2xx range — `201 Created`, `204 No Content` — not just a plain `200`. Checking a real range rather than one exact number is the genuinely correct check.

`fetch`'s own generic `<T: Decodable>` means one real function serves every model type in the app — `apiClient.fetch(Quote.self, from: quoteURL)`, `apiClient.fetch([Task].self, from: tasksURL)` — with the status-code checking and error handling written exactly once.

Hands-On Exercises

EXERCISE 1

Write a real function `createTask(_ newTask: NewTaskRequest) async throws -> Task` that builds a `URLRequest` with method `"POST"`, a JSON-encoded body via `JSONEncoder`, and decodes the server's own response back into a `Task`.

View solution

EXERCISE 2

Add a real case `.unauthorized` to `APIError`, with a matching `errorDescription`, and update the chapter's own `fetch` function so a `401` status code throws `.unauthorized` specifically, rather than

the generic `.requestFailed(statusCode:)` case.

View solution

EXERCISE 3

Explain, in your own words, why centralizing status-code checking and decoding inside one `APIClient`, rather than repeating that logic inside every individual ViewModel's own networking code, is a genuine real-world maintenance win — connecting your answer to Chapter 1's own MVVM separation-of-concerns principle.

View solution

CHAPTER 3 QUICK REFERENCE

- `URLRequest` — real, full control over HTTP method, headers, and body, needed for anything beyond a plain GET
- `Encodable` / `JSONEncoder` — turns a real Swift struct into a JSON request body
- `.convertFromSnakeCase` (automatic) and `CodingKeys` (manual, per-property) — two real, complementary tools for JSON key mismatches
- A custom `Error` conforming to `LocalizedError` carries real, specific meaning a generic error can't
- A centralized, generic `APIClient` avoids repeating status-code/decoding logic inside every ViewModel

CHAPTER 4: LOCAL PERSISTENCE: SWIFTDATA VS. CORE DATA

iOS Development — Architecture & Data

Chapter 4 · Local Persistence: SwiftData vs. Core Data

TaskFlow's own tasks vanish every time the app relaunches — nothing was ever actually saved to disk. This chapter covers Apple's real, current answer for local persistence, **SwiftData**, honestly contrasted against the older framework it builds on, **Core Data**.

Core Data: Real, Established, Genuinely Complex

Core Data is Apple's long-established real object-graph and persistence framework — genuinely powerful, and genuinely more complex to set up than most of what this course has covered so far. A typical Core Data model requires a separate `.xcdatamodeld` editor file, `NSManagedObject` subclasses, and manual `NSPersistentContainer` setup before a single object can be saved.

> **A REAL, HONEST TRADE-OFF**
>
> Core Data isn't obsolete — it remains real, actively used in countless existing apps, and offers some advanced real capabilities (fine-grained migration control, CloudKit integration matured over many years) that SwiftData, being newer, is still catching up to in places. For a brand-new app's own basic persistence needs, though, the real complexity Core Data requires is rarely worth it anymore.

SwiftData: Apple's Real, Current Answer

Introduced at **WWDC 2023** alongside **iOS 17 and Swift 5.9** — the same real release that brought the Observation framework (Fundamentals Chapter 6) — SwiftData is Apple's modern, Swift-native persistence framework, built on Apple's own established Core Data persistence engine but exposed through a genuinely different, dramatically simpler real API.

`@Model`

A real macro turning a plain Swift class into a persisted

`@Query`

A real property wrapper fetching and automatically

`ModelContainer`

Manages the real schema and on-disk storage — set up

model — no separate schema editor file needed.

observing persisted data directly inside a SwiftUI view.

once, typically at the app's own root.

`ModelContext`

Tracks in-memory changes and saves them — SwiftData's real equivalent of Core Data's `NSManagedObjectContext`.

Turning `Task` into a Real, Persisted Model

```
@Model
final class Task {
    var title: String
    var isDone: Bool
    var priority: Int

    init(title: String, isDone: Bool = false, priority: Int = 1) {
        self.title = title
        self.isDone = isDone
        self.priority = priority
    }
}
```

A REAL, GENUINELY NOTABLE CHANGE

`Task` is now a real `class`, not the struct Fundamentals used — SwiftData's own `@Model` macro requires a reference type, since persisted objects need a real, stable identity across fetches and edits, the same underlying reason `NSManagedObject` has always been a class in Core Data too. This is a genuine, deliberate exception to Fundamentals Chapter 4's own "prefer structs" guidance — persistence is exactly the kind of case that guidance already flagged as needing shared, mutable identity.

Wiring Up the `ModelContainer`

```
@main
struct TaskFlowApp: App {
    var body: some Scene {
        WindowGroup {
            TaskListView()
        }
    }
    .modelContainer(for: Task.self)
}
```

The real `.modelContainer(for:)` modifier sets up SwiftData's own on-disk storage for `Task` once, at the app's own root — every view inside `WindowGroup` can then reach a shared `ModelContext` automatically, without passing it down by hand.

Fetching with `@Query`

```
struct TaskListView: View {
    @Query private var tasks: [Task]
    @Environment(\.modelContext) private var modelContext

    var body: some View {
        List(tasks) { task in
            Text(task.title)
        }
    }

    func addTask(title: String) {
        let newTask = Task(title: title)
        modelContext.insert(newTask)
    }
}
```

REAL, AUTOMATIC LIVE UPDATES

`@Query` behaves like a real, persisted cousin of Fundamentals Chapter 6's own `@State` — a view using it redraws automatically whenever the underlying data changes, with no manual refresh call

needed, whether that change came from this view's own `modelContext.insert(...)` or from anywhere else in the app.

	CORE DATA	SWIFTDATA
Real Introduction	2005 (macOS), long-established on iOS	iOS 17, Swift 5.9, WWDC 2023
Model Definition	A separate `.xcdatamodeld` editor file	A plain Swift class with `@Model`
Fetching	`NSFetchRequest`, real, verbose setup	`@Query`, a single property wrapper
Best Real Fit	Existing Core Data apps, advanced/legacy needs	New apps' own straightforward persistence

Hands-On Exercises

EXERCISE 1

Write a real `@Model` class `JournalEntry` with `var text: String` and `var date: Date`, plus a real initializer, and a view using `@Query private var entries: [JournalEntry]` to display all entries in a `List`.

View solution

EXERCISE 2

Write a real function `deleteTask(_ task: Task, context: ModelContext)` using SwiftData's own `context.delete(task)` method, and explain, in your own words, why this call doesn't need an explicit "save" step for the deletion to persist to disk.

View solution

EXERCISE 3

Explain, in your own words, why SwiftData's `@Model` requires a class rather than a struct, connecting your answer directly to Fundamentals Chapter 4's own value-type/reference-type distinction and its

real "when to reach for a class instead" guidance.

View solution

CHAPTER 4 QUICK REFERENCE

- Core Data — real, established, genuinely more complex; still the right real fit for existing apps and some advanced needs
- SwiftData (iOS 17/Swift 5.9, WWDC 2023) — built on Core Data's own engine, exposed through a dramatically simpler real API
- `@Model` turns a plain class into a persisted model — no separate schema file
- `@Query` fetches and automatically observes persisted data, like a persisted cousin of `@State`
- `ModelContainer` (schema/storage) and `ModelContext` (in-memory changes/saving) are SwiftData's real core pieces
- `@Model` classes are genuinely a deliberate exception to "prefer structs" — persistence needs real, stable, shared identity

CHAPTER 5: DEPENDENCY INJECTION & TESTABILITY

iOS Development — Architecture & Data

Chapter 5 · Dependency Injection & Testability

Chapter 3's own `APIClient` is a real improvement over scattered networking code — but if a ViewModel creates its own `APIClient` internally, there's still no real way to test that ViewModel without hitting an actual real network. This chapter covers the fix: dependency injection.

The Real Problem

```
@Observable
@MainActor
final class TaskListViewModel {
    private let apiClient = APIClient()   // created internally — a real, genuine testi

    func refresh() async {
        tasks = (try? await apiClient.fetch([Task].self, from: tasksURL)) ?? []
    }
}
```

WHY THIS IS A REAL, GENUINE PROBLEM

Every test of `refresh()` would genuinely hit the real network — slow, flaky, and dependent on a real server actually being reachable and returning predictable data. There's no real way to substitute a fake, predictable response without changing `TaskListViewModel`'s own source code.

The Fix: Depend on an Abstraction, Not a Concrete Type

Real dependency injection means a type receives its own dependencies from **outside**, rather than creating them itself — and depending on a real `protocol`, not the concrete `APIClient`, is what actually makes substituting a fake possible.

```swift
protocol APIClientProtocol {
    func fetch<T: Decodable>(_ type: T.Type, from url: URL) async throws -> T
}

struct APIClient: APIClientProtocol {
    func fetch<T: Decodable>(_ type: T.Type, from url: URL) async throws -> T {
        // the chapter 3 implementation, unchanged
    }
}
```

TASKLISTVIEWMODEL.SWIFT — REFACTORED WITH INITIALIZER INJECTION

```swift
@Observable
@MainActor
final class TaskListViewModel {
    private let apiClient: APIClientProtocol

    init(apiClient: APIClientProtocol = APIClient()) {
        self.apiClient = apiClient
    }

    func refresh() async {
        tasks = (try? await apiClient.fetch([Task].self, from: tasksURL)) ?? []
    }
}
```

A REAL, GENUINELY SMALL CHANGE WITH A LARGE PAYOFF

Real, ordinary app code needs no change at all — `TaskListViewModel()` still works exactly as before, thanks to the real default parameter value. But a test, or a SwiftUI Preview, can now pass in *anything* conforming to `APIClientProtocol` — including a fake that returns predictable data instantly, with no real network involved.

A Real Fake for Tests and Previews

```swift
struct FakeAPIClient: APIClientProtocol {
    func fetch<T: Decodable>(_ type: T.Type, from url: URL) async throws -> T {
```

```
        let sample = [Task(title: "Sample Task", priority: 2)]
        return sample as! T   // safe here: this fake is only ever used with [Task]
    }
}

// In a SwiftUI Preview:
#Preview {
    TaskListView(viewModel: TaskListViewModel(apiClient: FakeAPIClient()))
}
```

A REAL, PRACTICAL WIN BEYOND TESTING

This same fake makes SwiftUI Previews work reliably too — a Preview using the real `APIClient()` would either hang waiting on a real network call inside Xcode's own canvas, or show empty state. `FakeAPIClient` gives a Preview real, immediate, predictable sample data.

SwiftUI's Own Native DI: Custom `@Environment` Values

For app-wide shared services — reaching many unrelated screens, not just one ViewModel's own initializer — SwiftUI offers a second, real, framework-native form of dependency injection: a custom `EnvironmentKey`.

```
struct APIClientKey: EnvironmentKey {
    static let defaultValue: APIClientProtocol = APIClient()
}

extension EnvironmentValues {
    var apiClient: APIClientProtocol {
        get { self[APIClientKey.self] }
        set { self[APIClientKey.self] = newValue }
    }
}

// Reading it in a View:
@Environment(\.apiClient) private var apiClient
```

APPROACH	REAL BEST FIT
Initializer Injection	One ViewModel's own specific dependency — explicit, real, and directly testable per-instance
Custom `@Environment`	A real, genuinely shared service many unrelated Views need, without threading it through every intermediate initializer

Hands-On Exercises

EXERCISE 1

Define a real protocol `QuoteFetching` with one method, `func fetchQuote() async throws -> Quote`, make `Quote`'s own real fetch function conform via a struct, and refactor `QuoteHeaderView` (from Fundamentals' capstone) to accept a `QuoteFetching` value via its own initializer instead of calling `fetchQuote()` directly.

View solution

EXERCISE 2

Write a real `FakeQuoteFetcher: QuoteFetching` returning a fixed, predictable `Quote` instantly, with no real network call — then use it in a `#Preview` for `QuoteHeaderView`.

View solution

EXERCISE 3

Explain, in your own words, why depending on `APIClientProtocol` rather than the concrete `APIClient` struct is specifically what makes substituting `FakeAPIClient` possible, and why a default parameter value on the initializer means existing, real app code needs no changes at all.

View solution

CHAPTER 5 QUICK REFERENCE

- A type creating its own dependencies internally is a real, genuine testing dead end
- Depending on a real `protocol`, not a concrete type, is what makes substituting a fake possible
- Initializer injection with a default parameter value keeps ordinary app code unchanged while enabling real tests/Previews to substitute a fake
- A real fake conforming to the same protocol returns predictable data instantly — no real network involved
- A custom `EnvironmentKey` / `@Environment` value is SwiftUI's own native DI mechanism for services shared across many unrelated Views

CHAPTER 6: UNIT TESTING WITH SWIFT TESTING

iOS Development — Architecture & Data

Chapter 6 · Unit Testing with Swift Testing

Chapter 5's own `FakeAPIClient` was built specifically to make this chapter possible. Apple's real Swift Testing framework, introduced with **Xcode 16 (September 2024)**, is the tool that actually runs and verifies real tests against it.

Swift Testing vs. the Older XCTest

Swift Testing is a genuinely newer, Swift-native alternative to Apple's older XCTest framework — real, lighter-weight syntax built around Swift macros rather than XCTest's own class/method-naming conventions.

	XCTEST (OLDER)	SWIFT TESTING (REAL, CURRENT — XCODE 16+)
Marking a Test	A method starting with `test`	`@Test` on any real function
Asserting	`XCTAssertEqual(a, b)`	`#expect(a == b)`
Fatal Check	`XCTUnwrap(optional)`	`try #require(optional)`
Async Tests	Real, but more setup-heavy	Real, native `async` function support

`@Test` and `#expect`

```
import Testing

@Test func addingATaskIncreasesTheCount() {
    let viewModel = TaskListViewModel(apiClient: FakeAPIClient())

    viewModel.add(title: "Buy milk", priority: 1)

    #expect(viewModel.tasks.count == 1)
```

```
    #expect(viewModel.tasks.first?.title == "Buy milk")
}
```

> **THE REAL, DIRECT PAYOFF OF CHAPTER 5**
>
> `TaskListViewModel(apiClient: FakeAPIClient())` is only possible because Chapter 5 refactored the ViewModel to accept its dependency via a real, injected protocol. Without that refactor, this test would either need a real, live network connection, or simply couldn't be written at all.

A failed `#expect` is real but **non-fatal** — the test method keeps running, and every other `#expect` in the same test still executes, reporting its own real pass or fail independently.

Testing Real Async Code

```
@Test func refreshLoadsTasksFromTheFakeClient() async {
    let viewModel = TaskListViewModel(apiClient: FakeAPIClient())

    await viewModel.refresh()

    #expect(!viewModel.tasks.isEmpty)
}
```

> **REAL, NATIVE ASYNC SUPPORT**
>
> Marking the test function itself `async` is all that's needed — Swift Testing genuinely awaits it directly, with no extra expectation objects or completion handlers the way older async testing approaches often required.

`#require`: A Real, Fatal Check

`#expect` reports a failure and keeps going. `#require` is real and **fatal** — if the condition fails, the test stops immediately, and it doubles as a genuine, real way to safely unwrap an optional for the rest of the test:

```swift
@Test func theFirstTaskHasTheExpectedTitle() throws {
    let viewModel = TaskListViewModel(apiClient: FakeAPIClient())
    viewModel.add(title: "Buy milk", priority: 1)

    let firstTask = try #require(viewModel.tasks.first)   // real, fatal if tasks is emp

    #expect(firstTask.title == "Buy milk")   // safe: firstTask is a real, non-optional
}
```

A REAL, GENUINE REASON TO PREFER `#REQUIRE` HERE

Without `#require`, accessing `viewModel.tasks.first!` directly would force-unwrap (Fundamentals Chapter 2's own real risk) — a genuine crash, not a clean, reported test failure, if the array happened to be empty. `#require` turns that same risk into a real, clean, informative test failure instead.

Hands-On Exercises

EXERCISE 1

Write a real `@Test` function `togglingATaskFlipsItsDoneState()` that creates a `TaskListViewModel` with `FakeAPIClient()`, adds a task, calls `toggleDone(for:)` on it, and uses `#expect` to confirm `isDone` is now `true`.

View solution

EXERCISE 2

Write a real `@Test async` function that calls `refresh()` on a `TaskListViewModel`, then uses `try #require` to safely unwrap the first task before checking its title with `#expect`.

View solution

EXERCISE 3

Explain, in your own words, why `#expect` being non-fatal and `#require` being fatal are each the genuinely correct default for different real situations, using one concrete example of each from this

chapter.

View solution

CHAPTER 6 QUICK REFERENCE

- Swift Testing (Xcode 16, September 2024) — a real, current, Swift-native alternative to XCTest
- `@Test` marks any real function as a test — no naming convention required, unlike XCTest's `test`-prefixed methods
- `#expect(...)` — non-fatal; a failure is reported, but the test keeps running
- `try #require(...)` — fatal; stops the test immediately, and safely unwraps an optional for the rest of it
- Real, native `async` test function support — no extra expectation objects needed
- Chapter 5's own dependency injection is what makes real, network-free ViewModel testing possible at all

CHAPTER 7: UI TESTING & DEBUGGING WITH XCODE INSTRUMENTS

iOS Development — Architecture & Data

Chapter 7 · UI Testing & Debugging with Xcode Instruments

Chapter 6's own unit tests verify a ViewModel's own logic in isolation. This chapter covers two genuinely different real tools: **UI testing**, which drives the actual real app UI end to end, and **Instruments**, Apple's own real profiling suite for finding performance and memory problems a unit test can't catch.

UI Testing: A Genuinely Different Layer

Chapter 6's own tests never launch a real app — they test a ViewModel directly. UI testing (built on the real **XCUITest** framework) does the opposite: it launches the actual real app and drives it exactly as a user would, tapping buttons and reading real, rendered content.

```
func testAddingATask() {
    let app = XCUIApplication()
    app.launch()

    app.buttons["addTaskButton"].tap()
    app.textFields["newTaskTitleField"].typeText("Buy milk")
    app.buttons["saveButton"].tap()

    XCTAssertTrue(app.staticTexts["Buy milk"].exists)
}
```

Making Elements Findable: `.accessibilityIdentifier()`

`XCUIApplication` finds real UI elements by their **accessibility identifier** — a stable, real string that doesn't change with localization or a later text edit, unlike matching on a button's own visible label text.

```
Button("Add") { isShowingNewTask = true }
```

```
    .accessibilityIdentifier("addTaskButton")
```

A REAL, GENUINE BONUS

Real accessibility identifiers exist for accessibility itself first — VoiceOver and other real Apple accessibility tools use them too. Adding them for UI testing genuinely improves the app's own real accessibility as a direct side effect, not a separate task.

Debugging with the Xcode Debugger

Beyond automated tests, Xcode's own real, built-in debugger — breakpoints, stepping through code, and the console's real `po` (print object) command for inspecting a variable's own current value mid-run — remains the genuinely fastest tool for understanding a single, specific bug as it happens.

Instruments: Real Performance & Memory Profiling

Instruments is Apple's real, dedicated profiling application, launched from Xcode via **Product → Profile**, running the real app while measuring what it actually does.

Time Profiler

Real, sampled CPU usage — which functions are actually consuming the most processing time.

Allocations

Every real memory allocation and its full lifecycle — genuinely the right tool for retain cycles.

Leaks

Real, definite memory loss — allocated memory nothing can reach anymore at all.

A REAL, GENUINELY NON-OBVIOUS DISTINCTION

The **Leaks** instrument does **not** reliably catch a strong reference cycle (Fundamentals Chapter 4's own real ARC topic) — a retain cycle's own memory is technically still *reachable* through the cycle itself, so it never registers as a genuine "leak" the way memory with zero real references left does. The **Allocations** instrument is the genuinely correct real tool for retain cycles — watching for memory that keeps growing and never shrinks back down after a screen is dismissed is the real, practical signal something is being retained that should have been released.

A REAL, CONCRETE SYMPTOM TO WATCH FOR

Navigate into `TaskDetailView` and back out, repeatedly, while watching Allocations' own real memory graph. If the real memory count keeps climbing and never returns to its starting baseline, that's a genuine, practical sign a retain cycle is keeping old `TaskDetailView` instances alive that should have been deallocated.

Hands-On Exercises

EXERCISE 1

Add real `.accessibilityIdentifier()` modifiers to TaskFlow's own "Add" toolbar button and the "Save" button inside `NewTaskView`, then write a real XCUITest function that launches the app, taps "Add," types a title, taps "Save," and asserts the new task's own title appears somewhere in the app.

View solution

EXERCISE 2

Explain, in your own words, why a UI test finding a button by its real, visible label text ("Add") is genuinely more fragile than finding it by a real accessibility identifier ("addTaskButton"), giving one concrete, realistic scenario where the label-text approach would break.

View solution

EXERCISE 3

Explain, in your own words, why the Leaks instrument genuinely can't reliably detect a strong reference cycle, and why Allocations' own "memory that never shrinks back down" pattern is a real, practical signal of one instead.

View solution

CHAPTER 7 QUICK REFERENCE

- XCUITest launches and drives the real, actual app UI — a genuinely different layer from Chapter 6's own isolated ViewModel unit tests
- `.accessibilityIdentifier()` gives real UI elements a stable, findable identity that doubles as a genuine accessibility improvement
- The Xcode debugger (breakpoints, `po`) remains the fastest real tool for understanding one specific bug live
- Time Profiler measures real CPU usage; Allocations and Leaks measure real memory, but for genuinely different things
- Leaks catches definite memory loss only — Allocations, watching for memory that never shrinks back down, is the real, correct tool for finding strong reference cycles

CHAPTER 8: WORKING WITH DEVICE CAPABILITIES

iOS Development — Architecture & Data

Chapter 8 · Working with Device Capabilities

Every capability this chapter covers — photos, location, notifications — touches something genuinely private to the user. Apple's own real platform enforces that seriously: most of them require an explicit, real permission prompt, and a real, specific reason string declared in advance.

Real, Required Privacy Descriptions

Before requesting access to almost any sensitive real capability, Xcode's own `Info` tab (or a direct edit to `Info.plist`) needs a real, specific usage-description key — the actual message shown to the user in the real system permission prompt.

> **A REAL, HARD REQUIREMENT**
>
> Requesting a real, protected capability with no matching usage-description key doesn't produce a graceful error — it's a genuine, immediate app **crash** at the moment of the request. This is Apple's own real, deliberate enforcement, not an edge case to handle defensively.

Photos: `PhotosPicker` — A Real, Notable Exception

Introduced in **iOS 16**, SwiftUI's own real `PhotosPicker` is the current, idiomatic way to let a user choose a photo.

```
import PhotosUI

struct TaskPhotoPicker: View {
    @State private var selectedItem: PhotosPickerItem?
    @State private var selectedImage: Image?

    var body: some View {
        VStack {
            selectedImage?.resizable().scaledToFit()
```

```
            PhotosPicker(selection: $selectedItem, matching: .images) {
                Text("Attach a Photo")
            }
            .onChange(of: selectedItem) {
                Task {
                    if let data = try? await selectedItem?.loadTransferable(type: Data.
                       let uiImage = UIImage(data: data) {
                        selectedImage = Image(uiImage: uiImage)
                    }
                }
            }
        }
    }
}
```

A REAL, GENUINELY NOTABLE EXCEPTION

`PhotosPicker` needs **no** real `Info.plist` privacy key at all — it runs as a separate, real, sandboxed system process; the app itself never gains broad photo-library access, only the specific real image(s) the user actually picks. This is a deliberate, real design choice by Apple, not an oversight — a genuine, useful exception to this chapter's own opening rule.

Location: `CLLocationManager`

```
import CoreLocation

final class LocationProvider: NSObject, CLLocationManagerDelegate {
    private let manager = CLLocationManager()

    override init() {
        super.init()
        manager.delegate = self
        manager.requestWhenInUseAuthorization()
    }

    func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CL
        // real, delegate-based updates arrive here
```

```
    }
}
```

CAPABILITY	REAL REQUIRED INFO.PLIST KEY
Location (When in Use)	`NSLocationWhenInUseUsageDescription`
Photos (`PhotosPicker`)	*None required*
Notifications	*None required* — authorized via a real runtime request instead

A REAL, GENUINE NOTE ON ASYNC ALTERNATIVES

`CLLocationManagerDelegate`'s callback-based pattern remains the real, foundational way location updates are delivered. Newer, async-sequence-based alternatives exist on more recent real OS versions — worth investigating directly against Apple's own current documentation before adopting one, per this course's own recurring discipline of verifying fast-moving APIs rather than assuming.

Notifications: `UNUserNotificationCenter`

```
import UserNotifications

func requestNotificationPermission() async {
    let center = UNUserNotificationCenter.current()
    let granted = try? await center.requestAuthorization(options: [.alert, .sound, .bad
    print("Notifications authorized: \(granted ?? false)")
}

func scheduleReminder(for task: Task) {
    let content = UNMutableNotificationContent()
    content.title = "Task Reminder"
    content.body = task.title

    let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 3600, repeats: false)
    let request = UNNotificationRequest(identifier: task.id.uuidString, content: conten
```

```
    UNUserNotificationCenter.current().add(request)
}
```

A REAL, PRACTICAL ORDERING RULE

`requestAuthorization` should be called once, real and early — typically at a natural moment in the app's own flow, not immediately on first launch before the user has any context for why it's being asked. A denied real prompt can only be reversed by the user, manually, in Settings — there's no real way for the app to ask again.

Hands-On Exercises

EXERCISE 1

Write a real `func requestLocationAndNotify() async` function that requests notification authorization via `UNUserNotificationCenter`, and — only if `granted` is `true` — schedules a real local notification confirming permission was granted.

View solution

EXERCISE 2

Explain, in your own words, what would actually happen at runtime if `NSLocationWhenInUseUsageDescription` were missing from `Info.plist` and `requestWhenInUseAuthorization()` were called anyway.

View solution

EXERCISE 3

Explain, in your own words, why `PhotosPicker` needing no `Info.plist` key at all is a genuine, deliberate privacy design choice, rather than an inconsistency or an oversight compared to location and camera access.

View solution

CHAPTER 8 QUICK REFERENCE

- Most sensitive real capabilities need a real, specific `Info.plist` usage-description key — missing one causes a genuine crash, not a graceful error
- `PhotosPicker` (iOS 16) needs no privacy key at all — a real, deliberate exception, since it runs as a separate sandboxed system process
- `CLLocationManager`'s real `requestWhenInUseAuthorization()` + `CLLocationManagerDelegate` pattern requires `NSLocationWhenInUseUsageDescription`
- `UNUserNotificationCenter.requestAuthorization(options:)` — real, current async permission request; a denied prompt can only be reversed manually in Settings
- A real local notification is built from `UNMutableNotificationContent` + a trigger + `UNNotificationRequest`, added via `UNUserNotificationCenter.current().add(request)`

CHAPTER 9: COMBINE VS. ASYNC/AWAIT: REACTIVE PATTERNS IN SWIFT

iOS Development — Architecture & Data

Chapter 9 · Combine vs. async/await: Reactive Patterns in Swift

Every real, ongoing stream this course has touched so far — a single fetched quote, one refresh call — has been a genuine one-shot operation. Some real data genuinely arrives as a continuous stream of many values over time. This chapter covers both real tools for that: Combine, and async/await's own newer answer, `AsyncSequence`.

Combine: Real, Established, Genuinely Older

Introduced in **iOS 13, 2019**, Combine is Apple's own real, functional-reactive framework — a `Publisher` emits a real stream of values over time; a `Subscriber` receives them, often through real operators like `map`, `filter`, and the terminal `sink`.

```
import Combine

[1, 2, 3, 4, 5].publisher
    .filter { $0 > 2 }
    .map { $0 * 2 }
    .sink { print($0) }   // prints 6, 8, 10
```

A Real, Honest Positioning

> **APPLE'S OWN REAL, CURRENT GUIDANCE HAS SHIFTED**
>
> Since `async`/`await` and `AsyncSequence` arrived in **Swift 5.5, 2021**, Apple's own real emphasis for new code has genuinely moved toward them. Combine remains real, supported, and still genuinely useful for complex multi-stream reactive logic and for existing, already-Combine-based codebases — but it's no longer the automatic real default for new SwiftUI apps the way it once was.

`AsyncSequence`: The Real async/await Equivalent

Where a Combine `Publisher` emits values a `Subscriber` reacts to, an `AsyncSequence` is a real sequence a caller iterates directly with `for await` — conceptually the same real job, expressed in the same language this course has used throughout.

```
for await notification in NotificationCenter.default.notifications(named: .NSSystemClocl
    print("System clock changed")
}
```

A REAL, CONCRETE COMPARISON

`NotificationCenter.default.notifications(named:)` is a real, current, async/await-native alternative to what Combine's own `NotificationCenter.Publisher` used to be needed for — and it carries a genuine, verified structural advantage: observation **automatically stops** the moment the surrounding `for await` loop exits or its own `Task` is cancelled, with no manual subscription cleanup required at all.

Building a Real Custom Stream: `AsyncStream`

For a genuinely custom, multi-value stream — not one Apple already provides an `AsyncSequence` for — `AsyncStream` is the real, direct async/await equivalent of manually building a Combine `Publisher`:

```
func countdown(from start: Int) -> AsyncStream<Int> {
    AsyncStream { continuation in
        Task {
            for value in stride(from: start, through: 0, by: -1) {
                continuation.yield(value)
                try? await Task.sleep(for: .seconds(1))
            }
            continuation.finish()
        }
    }
}

for await value in countdown(from: 3) {
    print(value)   // 3, 2, 1, 0 — one per real second
}
```

	COMBINE	ASYNC/AWAIT
Real Introduction	iOS 13, 2019	Swift 5.5, iOS 15, 2021
Multi-Value Stream	`Publisher`	`AsyncSequence` / `AsyncStream`
Consuming	`.sink { }`, operator chains	`for await`, the same syntax as every other async call in this course
Apple's Real Current Emphasis	Existing codebases, complex multi-stream operators	New code, by real, current default

A REAL, PRACTICAL RULE FOR NEW CODE

Reaching for `AsyncSequence` / `AsyncStream` keeps a codebase speaking one real, consistent async language throughout — the exact same `async` / `await` / `Task` vocabulary this course has used since Chapter 2, rather than mixing in a second, genuinely different real paradigm just for streams of multiple values.

Hands-On Exercises

EXERCISE 1

Write a real function `timer(every interval: Duration, count: Int) -> AsyncStream<Int>` that yields `1` through `count`, one value every real `interval`, then finishes — and a real `for await` loop consuming it.

View solution

EXERCISE 2

Explain, in your own words, why `NotificationCenter.default.notifications(named:)`'s own real automatic-stop-on-cancellation behavior is a genuine, concrete advantage over a manually managed Combine subscription, which requires explicitly storing and later cancelling an `AnyCancellable`.

View solution

EXERCISE 3

Explain, in your own words, why choosing `AsyncSequence` / `AsyncStream` over Combine for new code in a project already using `async` / `await` throughout (like TaskFlow) is a genuine, practical consistency win, not just a matter of personal preference.

View solution

CHAPTER 9 QUICK REFERENCE

- Combine (iOS 13, 2019) — real, established functional-reactive framework: `Publisher`, `Subscriber`, operators like `map` / `filter` / `sink`
- Apple's own real, current guidance has genuinely shifted toward async/await for new code since Swift 5.5 (2021) — Combine remains real and supported, not deprecated
- `AsyncSequence` is the real async/await equivalent of a Combine `Publisher`, consumed with `for await`
- `NotificationCenter.default.notifications(named:)` — a real, current, verified example: automatically stops observing on scope exit or cancellation, no manual cleanup
- `AsyncStream` — the real, direct way to build a genuinely custom multi-value async stream

CHAPTER 10: CAPSTONE — A DATA-DRIVEN, TESTED SWIFTUI APP

iOS Development — Architecture & Data

Chapter 10 · Capstone: A Data-Driven, Tested SwiftUI App

Fundamentals' own TaskFlow worked, but everything lived in memory and nothing was tested. This capstone evolves it into a real, production-shaped app — persisted, networked, dependency-injected, and genuinely tested — wiring together every one of this course's own nine chapters.

CHAPTER	WHAT IT CONTRIBUTES
1 — MVVM	`TaskListViewModel` as a real, dedicated ViewModel — Model/View/ViewModel separation throughout
2 — Concurrency	`@MainActor` on the ViewModel, real `Task {}` for background sync
3 — Networking	`APIClient`, `URLRequest`, `APIError`, real status-code checking
4 — SwiftData	`Task` as a real `@Model`, `@Query` for live local display
5 — Dependency Injection	`APIClientProtocol`, real initializer injection, `FakeAPIClient`
6 — Unit Testing	Real `@Test` functions verifying `TaskListViewModel` with zero real network access
7 — UI Testing & Instruments	Real `.accessibilityIdentifier()` on every interactive element
8 — Device Capabilities	`PhotosPicker` for a task photo, a real local reminder notification
9 — Reactive Patterns	A real `AsyncStream`-based task-added event feed

File Structure

```
Task.swift                  — Ch4 (real @Model)
TaskDTO.swift               — Ch3, Ch4 (network shape, separate from the persisted Mo
APIClient.swift              — Ch3, Ch5 (protocol + real implementation)
FakeAPIClient.swift           — Ch5, Ch6
TaskEvents.swift             — Ch9 (AsyncStream)
```

```
TaskListViewModel.swift        — Ch1, Ch2, Ch3, Ch5
TaskListView.swift             — Ch4, Ch7
NewTaskView.swift              — Ch7, Ch8
TaskFlowApp.swift               — Ch4 (ModelContainer)
TaskListViewModelTests.swift    — Ch6
```

A Real, Deliberate Split: `Task` vs. `TaskDTO`

> **AN HONEST ARCHITECTURAL TENSION, RESOLVED**
>
> Chapter 3's own `Task` was `Decodable`; Chapter 4's own `Task` is a real `@Model` class. Mixing both roles onto one type genuinely works in simple cases, but real apps commonly keep them separate — a lightweight **DTO** (Data Transfer Object) matches the server's own real JSON shape, while the persisted `@Model` stays focused purely on local storage. This capstone makes that real, deliberate choice explicit.

TASK.SWIFT

```swift
@Model
final class Task {
    var title: String
    var isDone: Bool
    var priority: Int
    var photoData: Data?

    init(title: String, isDone: Bool = false, priority: Int = 1) {
        self.title = title
        self.isDone = isDone
        self.priority = priority
    }
}
```

TASKDTO.SWIFT

```swift
struct TaskDTO: Decodable {
    let title: String
    let priority: Int

    func toTask() -> Task {
```

```
            Task(title: title, priority: priority)
        }
    }
```

The ViewModel: MVVM, Concurrency, and Injected Networking

TASKLISTVIEWMODEL.SWIFT

```
@Observable
@MainActor
final class TaskListViewModel {
    private let apiClient: APIClientProtocol
    private let modelContext: ModelContext
    private(set) var isSyncing = false

    init(modelContext: ModelContext, apiClient: APIClientProtocol = APIClient()) {
        self.modelContext = modelContext
        self.apiClient = apiClient
    }

    func syncWithServer() async {
        isSyncing = true
        defer { isSyncing = false }

        guard let dtos = try? await apiClient.fetch([TaskDTO].self, from: tasksURL) else

        for dto in dtos {
            modelContext.insert(dto.toTask())
        }
    }
}
```

EVERY CHAPTER, IN ONE REAL TYPE

Chapter 1's own MVVM separation, Chapter 2's `@MainActor`, Chapter 3's `APIClientProtocol`, and Chapter 5's real initializer injection all land in this single, real ViewModel — the same real class shape this course has been building toward since Chapter 1's own `TaskStore` refactor.

The View: Real, Live Local Data

TASKLISTVIEW.SWIFT

```
struct TaskListView: View {
    @Query private var tasks: [Task]
    @Environment(\.modelContext) private var modelContext
    @State private var isShowingNewTask = false

    var body: some View {
        NavigationStack {
            List(tasks) { task in
                Text(task.title)
            }
            .navigationTitle("TaskFlow")
            .toolbar {
                Button("Add") { isShowingNewTask = true }
                    .accessibilityIdentifier("addTaskButton")
            }
            .sheet(isPresented: $isShowingNewTask) {
                NewTaskView()
            }
            .task {
                let viewModel = TaskListViewModel(modelContext: modelContext)
                await viewModel.syncWithServer()
            }
        }
    }
}
```

A REAL, DELIBERATE DIVISION OF LABOR

`@Query` keeps the View's own display genuinely live — any insert from anywhere, including the ViewModel's own `syncWithServer()`, updates `tasks` automatically. The ViewModel's real job is narrower and more focused than Course 1's own capstone: syncing with the network, not owning the displayed array directly — SwiftData's `@Query` already does that better.

Device Capabilities: Photo & Reminder

NEWTASKVIEW.SWIFT (EXCERPT)

```
func save() {
    let newTask = Task(title: title, priority: priority)
    newTask.photoData = selectedPhotoData
    modelContext.insert(newTask)

    TaskEvents.shared.taskAdded(newTask.title)   // Ch9

    let content = UNMutableNotificationContent()
    content.title = "Reminder"
    content.body = newTask.title
    let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 3600, repeats: false)
    let request = UNNotificationRequest(identifier: UUID().uuidString, content: content,
    UNUserNotificationCenter.current().add(request)

    dismiss()
}
```

A Real, Live Event Feed: `TaskEvents`

TASKEVENTS.SWIFT

```
final class TaskEvents {
    static let shared = TaskEvents()
    private var continuation: AsyncStream<String>.Continuation?

    lazy var stream: AsyncStream<String> = AsyncStream { continuation in
        self.continuation = continuation
    }

    func taskAdded(_ title: String) {
        continuation?.yield(title)
    }
}
```

Any part of the app can now observe every task-added event, in real time, using the exact same `for await` vocabulary Chapter 9 established — a real, live activity feed with zero Combine

involved.

Testing It All

TASKLISTVIEWMODELTESTS.SWIFT

```
import Testing
import SwiftData

@Test func syncingInsertsTasksFromTheFakeClient() async throws {
    let config = ModelConfiguration(isStoredInMemoryOnly: true)
    let container = try ModelContainer(for: Task.self, configurations: config)
    let context = ModelContext(container)

    let viewModel = TaskListViewModel(modelContext: context, apiClient: FakeAPIClient())
    await viewModel.syncWithServer()

    let fetchedTasks = try context.fetch(FetchDescriptor<Task>())
    #expect(!fetchedTasks.isEmpty)
}
```

A REAL, GENUINELY COMPLETE TEST — NO REAL NETWORK, NO REAL DISK

`ModelConfiguration(isStoredInMemoryOnly: true)` gives this test a real, working SwiftData stack that never touches disk, and `FakeAPIClient` means it never touches a real network either — the entire real sync flow, verified end to end, in milliseconds.

Hands-On Exercises

EXERCISE 1

Add a real `.accessibilityIdentifier("saveButton")` to `NewTaskView`'s own Save button, and write a real XCUITest confirming a newly-added task's own title appears in the list after the full add-task flow.

View solution

EXERCISE 2

Write a real `@Test` function using a fresh, in-memory `ModelContainer` that inserts one `Task` directly via `modelContext.insert(...)`, then confirms `context.fetch(FetchDescriptor<Task>())` returns exactly one task with the expected title — with no networking involved at all.

View solution

EXERCISE 3

Explain, in your own words, why splitting `Task` (a real `@Model`) and `TaskDTO` (a real `Decodable` struct) into two separate types is a genuinely better real design than making one type do both jobs, connecting your answer to Chapter 1's own MVVM separation-of-concerns principle.

View solution

Where to Go From Here

This capstone's own app still isn't ready to ship — no app icon, no signing, no real App Store submission, no CI pipeline running these very tests automatically. That's genuine **iOS Development — Production & Publishing** territory: provisioning, TestFlight, App Store Connect, and shipping this exact codebase to real users, the direct next course in this three-course arc.

WHAT THIS CAPSTONE DEMONSTRATES

- MVVM (Ch1), `@MainActor` concurrency (Ch2), and injected networking via `APIClientProtocol` (Ch3, Ch5) in one real ViewModel
- Real SwiftData persistence (`@Model`, `@Query`) deliberately kept separate from a network-facing `TaskDTO` (Ch4)
- Real, fast, isolated tests using `FakeAPIClient` and an in-memory `ModelConfiguration` — no real network, no real disk (Ch5, Ch6)
- Real accessibility identifiers ready for XCUITest (Ch7)
- `PhotosPicker` and a real local reminder notification (Ch8)

- A real, live `AsyncStream`-based event feed, using this course's own consistent async vocabulary throughout (Ch9)