

In one line: Put logic in a **ViewModel with protocol-injected dependencies** and unit-test its **state transitions** (fast, in-process); test a Coordinator's **routing decision**, not UIKit; keep **XCUITest** (separate process, accessibility tree) for a few critical journeys; pin **snapshots** to one device/OS/locale/appearance.

How it works

- **ViewModel:** a SwiftUI View is an opaque value description — you can't read what it shows, and `@State` only lives inside a rendered hierarchy. So: thin View, logic in an `@Observable/ObservableObject` VM. Test = **construct with doubles** → **call an input** → **assert state**. A VM that builds `URLSession.shared` itself is untestable — DI is the enabler.
- **@MainActor VM** → mark the test `@MainActor` and await the call, or you read stale state. **Sequence** `[.idle, .loading, .loaded]`: record via `$state.sink` (Combine) or with `ObservationTracking` (`@Observable` has no \$ publisher — assert properties directly).
- Model state as an **enum** (`idle/loading/loaded/error`); put formatting in the VM (`priceText == "$10.00"`); inject `@Environment` values (`locale`, `scenePhase`) as plain params.
- **Coordinator:** inject a navigation abstraction (Router protocol / a path: `[Route]`) and assert the **route produced** for an event — a spy records `show(detail(id: 7))`; no real push.
- **XCUITest:** let `app = XCUIApplication()`; set `launchArguments / launchEnvironment` before `app.launch()`; query `app.buttons["login_button"]`; `continueAfterFailure = false`; sync with `waitForExistence(timeout:)`; `exists != isHittable`. Alerts: `addUIInterruptionMonitor`. On failure attach `XCTAttachment(screenshot:)`.
- **Page Object:** one type per screen owns the selectors + actions (`LoginPage.login(email:password:)`); tests read as journeys; one place to fix.
- **Snapshot** (Point-Free swift-snapshot-testing): `assertSnapshot(of: view, as: .image(on: .iPhone13))` — first run **records** a reference in `__Snapshots__`, later runs **assert** (compare). Strategies: `.image`, `.json`, `.dump`, `.recursiveDescription`. Tolerance: `.image(precision: 0.98, perceptualPrecision: 0.98)`.

Example — ViewModel state transition

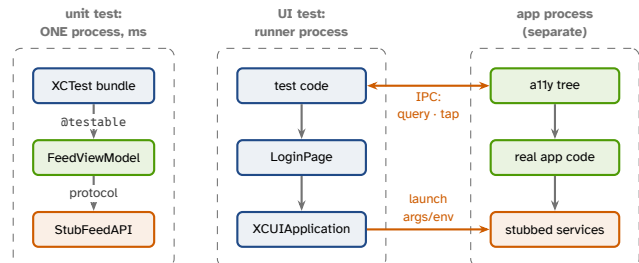
```
protocol FeedAPI { func items() async throws -> [Item] }
struct StubFeedAPI: FeedAPI {
    let result: Result<[Item], Error>
    func items() async throws -> [Item] { try result.get() } }

@MainActor func test_load_failure_showsError() async {
    let sut = FeedViewModel(api: StubFeedAPI(result: .failure(
        NSError(domain: "notConnectedToInternet")))
    var seen: [FeedState] = []
    let c = sut.$state.sink { seen.append($0) } // record sequence
    await sut.load()
    XCTAssertEqual(seen, [.idle, .loading, .error("Offline")])
    _ = c
}
```

UI test skeleton — stub + Page Object

```
override func setUpWithError() throws {
    continueAfterFailure = false
    app.launchArguments += ["-uitesting", "-disableAnimations"]
    app.launchEnvironment["API_BASE"] = "http://localhost:9999"
    app.launch() // args BEFORE launch
    func test_login_showsWelcome() {
        LoginPage(app).login(email: "a@b.co", password: "pw")
        let welcome = app.staticTexts["welcome"] // an identifier
        XCTAssertTrue(welcome.waitForExistence(timeout: 5))
    }
    // app side, at launch:
    if CommandLine.arguments.contains("-uitesting") { useStubs() }
```

Picture — where each kind of test runs



Coordinator test — spy the navigation

```
final class SpyRouter: Router { // Router = protocol
    var shown: [Route] = []
    func show(_ r: Route) { shown.append(r) }
    func test_selectingItem_routesToDetail() {
        let router = SpyRouter()
        let sut = FeedCoordinator(router: router)
        sut didSelect(item(id: 7)) // the event
        XCTAssertEqual(router.shown, [.detail(id: 7)]) // the DECISION
    }
}
```

Interview traps

- ! **accessibilityIdentifier** = test hook, never localized, never read aloud; **accessibilityLabel** = VoiceOver text, localized. Query by label → passes locally, fails on a de CI or a copy edit.
- ! `sleep(3)` before a tap: flaky when short, slow when long, waits for time not the event → `waitForExistence`.
- ! Pointing UI tests at the **real backend**: nondeterministic, slow, may mutate prod. Stub via `launchEnvironment["API_BASE"]` / a `-uitesting` flag; disable animations.
- ! Record mode committed (`record: .all`): it **always fails** and rewrites goldens. And “just re-record” green-lights real regressions — review the diff.
- ! Snapshot passes locally, fails on CI: recorded on a different simulator/OS/scale. **Pin** device, OS, locale, `userInterfaceStyle`, content size; inject fixed dates/IDs.
- ! “80% View coverage via ViewInspector” = logic lives in Views + brittle structure tests.

Remember

VM: inject → drive → assert state. UI test: **ID, wait, stub, Page Object, few**. Snapshot: **pin, record once, review diffs**.

Likely questions

1. Why not unit-test a SwiftUI View? — opaque body; `@State` is runtime-owned.
2. Test a Coordinator? — fake nav/router; assert the route chosen, not the push.
3. Stub the backend in XCUITest? — launch args/env read at app start → stub services.
4. Can a UI test read VM state? — no; separate process, a11y tree only.
5. Record vs assert? — record writes the reference (and fails); assert compares.