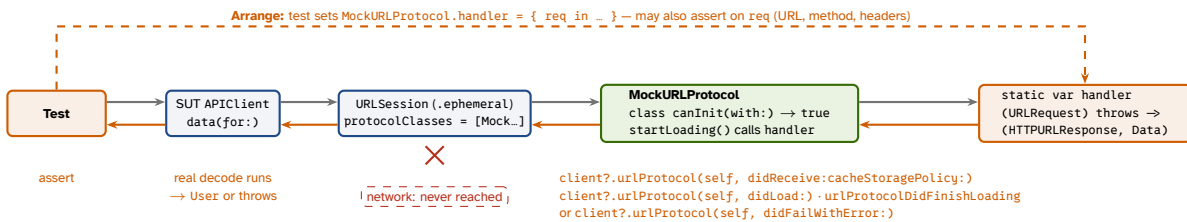


In one line: Wait for the event, never for time. async code: make the test async throws and await. Callbacks: XCTestExpectation + wait(for:timeout:). Network: run the *real* URLSession with a MockURLProtocol on an .ephemeral config, and cover **all three** failure paths. Time: inject a **clock**/scheduler and advance it — no sleep.



How it works

- **Callback API** — let exp = expectation(description:); exp.fulfill() in the handler; wait(for: [exp], timeout: 1). Knobs: expectedFulfillmentCount = N; isInverted = true (fails if fulfilled — “must not happen”, always waits the full timeout); wait(..., enforceOrder: true); over-fulfilling fails (assertForOverFulfill is on for expectation(description:)).
- **async API** — func test_x() async throws and await; XCTest awaits the method. Inside an async test, wait(for:) is noasync (Xcode 14.3+) — use await fulfillment(of: [exp], timeout: 1). Swift Testing: @Test func x() async throws + #expect(throws:); await confirmation { c in ... } replaces expectations.
- **Bridge** a callback you cannot change: await withCheckedContinuation { c in ... } — resume *exactly once* (twice = crash, never = hang).
- **Timeout** is a failure ceiling, not a duration: a passing test returns the moment the expectation is met. 1-2 s for stubbed work.

Example — MockURLProtocol

```
final class MockURLProtocol: URLProtocol {
    static var handler: ((URLRequest) throws -> (HTTPURLResponse, Data))? // set per test
    override class func canInit(with r: URLRequest) -> Bool { true } // intercept all
    override class func canonicalRequest(for r: URLRequest) -> URLRequest { r }
    override func startLoading() {
        do { let (resp, data) = try Self.handler!(request)
            client?.urlProtocol(self, didReceive: resp, cacheStoragePolicy: .notAllowed)
            client?.urlProtocol(self, didLoad: data)
            client?.urlProtocolDidFinishLoading(self)
        } catch {
            client?.urlProtocol(self, didFailWithError: error) } }
    override func stopLoading() {} }

let config = URLSessionConfiguration.ephemeral // no disk cache
config.protocolClasses = [MockURLProtocol.self] // not global
let sut = APIClient(session: URLSession(configuration: config))
```

Remember

Await, don't sleep. Stub bytes, not objects. Three ways to fail: wire · status · decode.

The three failure paths — test all of them

Path	handler does	what happens
Transport	throw URLError(.notConnectedToInternet)	data(for:) throws URLError → map to .offline
HTTP status	returns 404 / 500	URLSession does not throw; you check 200 .. <300 → .status(500)
Decoding	200 + malformed JSON	JSONDecoder throws DecodingError (keyNotFound, typeMismatch, valueNotFound, dataCorrupted)

Time, streams, cancellation

- **Clock**: depend on any Clock<Duration> — ContinuousClock() in prod, a test clock (TestClock / ImmediateClock, Point-Free *swift-clocks*) in tests: await clock.advance(by: .seconds(5)) — 0 ms real time. Plain Date: inject () → Date.
- **Combine**: inject the scheduler (ImmediateScheduler, or a TestScheduler you advance); sink into an array and *keep* the AnyCancellable; or publisher.values.
- **AsyncStream**: for await *bounded* — break after N or stream.prefix(3); an unfinished stream hangs the test.
- **Cancellation**: start a Task, cancel() it, try await task.value must throw CancellationError — but a cancelled URLSession call throws URLError(.cancelled). Also assert the side effect did *not* happen.

Interview traps

- ! Thread.sleep/Task.sleep “to let it settle” — slow *and* flaky. Await the signal or advance a test clock.
- ! Task { await vm.load() } then assert at once = race; await directly or await task.value. @MainActor VM → @MainActor test.
- ! .default config can serve a *cached* response; URLProtocol.registerClass is *process-wide*. Use .ephemeral + protocolClasses; nil the handler in tearDown.
- ! static var handler is shared state: tests using it must not run in parallel; Swift 6 mode needs nonisolated(unsafe).
- ! request.httpBody is often nil in the protocol — the body arrives as httpBodyStream.

Likely questions

1. Expectation vs async test? — callbacks/delegates vs anything awaitable.
2. Why URLProtocol over a protocol-wrapped session? — real request + decode run.
3. Does a 500 make data(for:) throw? — no; check statusCode yourself.
4. Test a 300 ms debounce fast? — inject a clock, advance (by:).