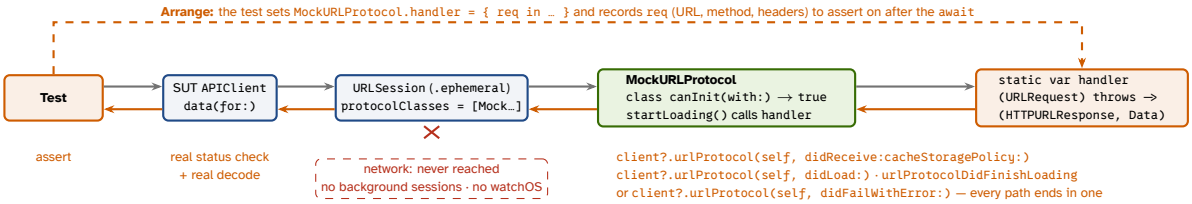
Async & network testing — expectations, confirmations, MockURLProtocol, clocks

testing · folio

In one line: Wait for the **event**, never for **time**. `async` code: an `async throws` test that `await`s. Callbacks: `XCTestExpectation` + `await fulfillment(of:)`, or Swift Testing's `confirmation` (which does *not* wait) or a continuation. Network: the *real* `URLSession` over a stub `URLProtocol`. Time: inject a `Clock` or scheduler and *advance* it.



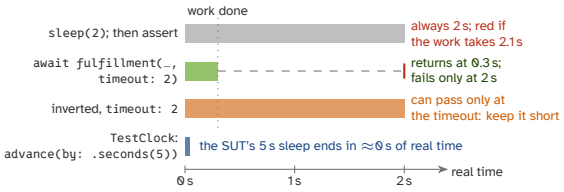
Pick the wait

need	XCTest	Swift Testing
async API	`func test() async throws` — arbitrary actor; sync tests run on the main actor	`@Test func f() async throws` — always an arbitrary task
callback, N times	`expectation(description:)`, `expectedFulfillmentCount = N`, `await fulfillment(of: [e], timeout: 2)`	`await confirmation(expectedCount: N) { c in … c() }` — all N *before the closure returns*
at least N	`assertForOverFulfill = false`	`expectedCount: N…` (6.1)
must not happen	`isInverted = true`: fails at once if fulfilled	`expectedCount: 0`
throws	`XCTAssertThrowsError` takes a *sync* autoclosure → `do { try await … } catch`	`await #expect(throws: E.self) { try await … }` returns the error (6.1)
hang guard	`executionTimeAllowance`: 600 s default, rounded *up* to whole minutes, only with test timeouts on	`.timeLimit(.minutes(n))`: minutes only; shortest wins; per argument case
parallel	a class runs its tests one by one	all at once, **one process**; `.serialized` orders one suite only

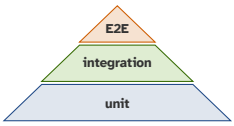
Expectations, precisely

- `wait(for:timeout:)` is **unavailable in async functions** since Xcode 14.3 (it can deadlock): `await fulfillment(of:timeout:)`. Left out, the timeout is `.infinity` — only `executionTimeAllowance` then ends a hang.
- Over-fulfilling fails only with `assertForOverFulfill`: `true` from the `XCTestCase` convenience methods, `false` from `XCTestExpectation(description:)`.
- Predicate expectations **poll** — slow; prefer callback, KVO or notification ones (inject a private `NotificationCenter()`, hand it to the expectation).
- Swift Testing has no wait: a confirmation neither blocks nor suspends. `XCTestExpectation` is unsupported in `@Test` (6.4's interop excludes it, though it made `XCTAssert` fail a `@Test`: before, a shared XCTest helper's failure left the test **green**). Bridge a callback:
  `try await withCheckedThrowingContinuation { c in api.fetch(42) { c.resume(with: $0) } }`
  Resume **exactly once**: a second resume is a `fatalError`; none leaves the test suspended for ever and logs "leaked its continuation".

What each wait costs



Network tests in layers



UI tests against a **mock server** you control; a few against the real one to pin the contract

real `URLSession` + stub `URLProtocol`: a couple of tests prove the pieces are wired together

request building and response parsing as pure functions: fast, clear failures — the bulk

On a device: Xcode's **Device Conditions** (Xcode 11) set a network-link profile (latency, loss, slow DNS) or a thermal state — the Network Link Conditioner, driven by Xcode.

Traps

! `static var handler` under Swift Testing: all suites run in parallel in one process, and `.serialized` does not order unrelated suites — two suites overwrite one stub. Key handlers per test (own host or header → table) or nest every network suite in one `.serialized` parent. Swift 6 mode: `nonisolated(unsafe)` or a lock.

! `URLProtocol.registerClass` is global (consulted in reverse order) — set the session's `protocolClasses`. `.ephemeral` still keeps caches, cookies in RAM.

The stub

```
final class MockURLProtocol: URLProtocol {
  nonisolated(unsafe) static var handler:
    ((URLRequest) throws -> (HTTPURLResponse, Data))?
  override class func canInit(with r: URLRequest) -> Bool { true }
  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
config.protocolClasses = [MockURLProtocol.self]
let sut = APIClient(session: URLSession(configuration: config))
MockURLProtocol.handler = { req in (HTTPURLResponse(url: req.url!,
  statusCode: 500, httpVersion: nil, headerFields: nil)!, Data()) }
await #expect(throws: APIError.status(500)) {
  try await sut.user(id: 42) }
```

Four ways a request fails — stub each

path	the handler	what the client sees
transport	`throw URLError(.notConnectedToInternet)` (−1009)	`data(for:)` throws that `URLError` → map to `.offline`
HTTP status	returns `404` / `500`	**no throw**: `(Data, URLResponse)` comes back; *you* cast and check `200 … 299`
decoding	`200` + malformed JSON	`DecodingError`: `keyNotFound` · `typeMismatch` · `valueNotFound` · `dataCorrupted`
cancelled	never finishes; the test cancels the `Task`	`URLError(.cancelled)` (−999), *not* `CancellationError`

Assert the **request** too. Its body arrives as `httpBodyStream`; `httpBody` is often `nil` inside the protocol.

Time, streams, Combine

- **Clock**: depend on `any Clock<Duration>` (iOS 16); `sleep(for:)` works on the existential since Swift 5.9. Prod `ContinuousClock()`; tests (swift-clocks) `TestClock`: `await clock.advance(by: .seconds(5))`, `await clock.run()` until nothing sleeps; `ImmediateClock` never suspends; `UnimplementedClock` fails if used. Dates: inject `() -> Date`. Timers / GCD: a scheduler protocol whose mock fires at once.
- **Combine**: Apple's `ImmediateScheduler` ignores dates but has its own time types, so it cannot replace `DispatchQueue` without generics; combine-schedulers' `AnySchedulerOf<DispatchQueue>` + `DispatchQueue.test` + `advance(by:)` can. Keep the `AnyCancellable` (deinit cancels), or `for await` over `publisher.values` (iOS 15; `for try await` if it can fail).
- **AsyncStream**: drive the SUT through `AsyncStream.makeStream(of:)` (5.9) — the test holds the continuation. Consume *bounded*: `s.prefix(3)` or `finish()`; an unfinished stream never ends the `for await`.

Ordering & cancellation

- `Task { await vm.load() }` then assert: the next line usually runs before the task body. Await `task.value`, call `await vm.load()` directly, or use `Task.immediate` (6.2: synchronous up to its first suspension). `withMainSerialExecutor` (swift-concurrency-extras) serialises a test's tasks — tests only, flips a global.
- `CancellationError` is promised by `Task.checkCancellation()` and `Task.sleep`, not by every API (URLSession: `URLError`). Assert the outcome *and* the side effect that must not happen. Swift Testing: a `CancellationError` in a cancelled test = cancelled, not failed; `Test.cancel()` (6.3).
- Cleanup (6.4): `defer { await … }` runs to completion; `withTaskCancellationShield` hides cancellation from it (no back-deployment).

Remember

Await the event, never sleep. Stub bytes, not objects — the real status check and decoder run. Four failures: wire · status · decode · cancel. Time is a dependency: inject it, advance it.

Related: XCTest & Swift Testing · Test doubles · Testable design · Testing fundamentals · Flaky tests, CI, coverage & TDD · Testing ViewModels, Coordinators, UI & snapshots · URLSession & networking · Time in Swift · AsyncSequence & AsyncStream · Combine in depth · Swift 6 · strict concurrency & data-race safety · Engineering for bad networks