Testing Swift Concurrency & SwiftUI

Rachel Brindle

@younata@hachyderm.io

Agenda

- Test Theory
- Testing Swift Concurrency
- Testing SwiftUI

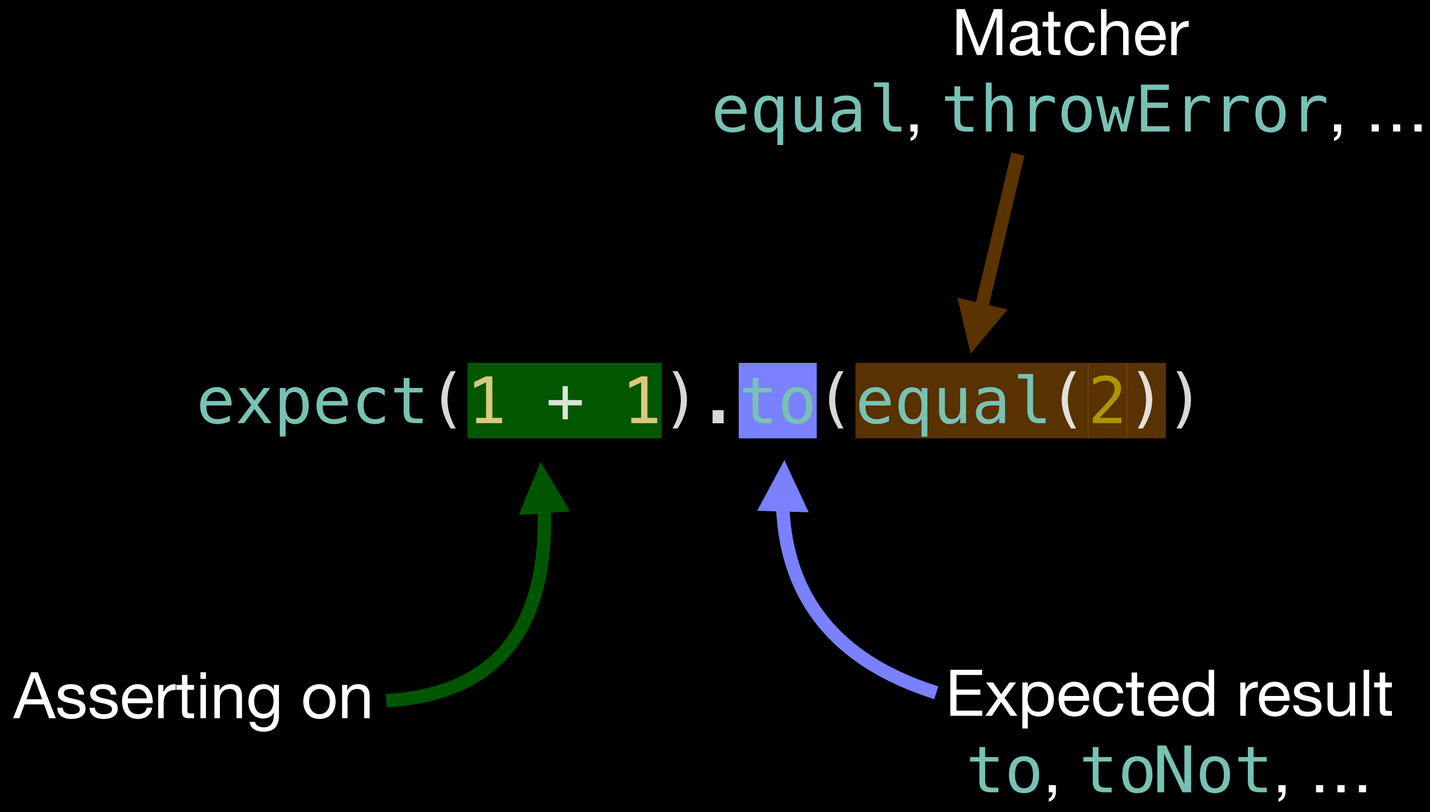
Who is Rachel Brindle

- Principal Software Engineer @ Autodesk
- 14 years iOS Experience
- 10 years Test Driven Development Experience
- Maintainer, Quick & Nimble

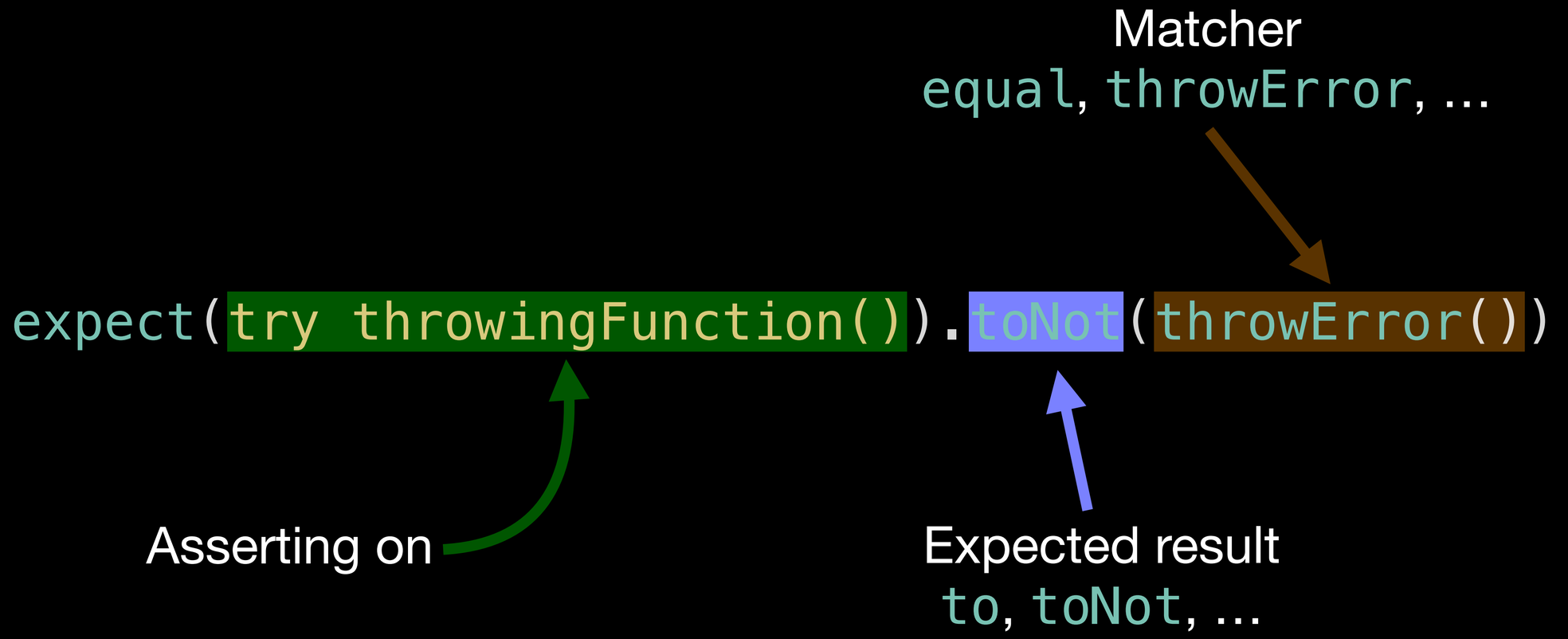
Quick - Testing Framework

```
beforeEach {}

it("shows the button") {}

describe("when the button is tapped") {
    beforeEach {}

    it("calls the action callback") {}

    it("shows the progress view") {}

    describe("when the action callback finishes") {
        beforeEach {}

        it("shows the button again”) {}
    }
}
```

Nimble - Assertion Framework



Nimble - Assertion Framework



Assumed Knowledge

- Basic familiarity with XCTest
  - Defining Test Cases and Test Methods - XCTest - Apple
- Familiar with Swift Concurrency
  - Meet async/await in Swift - WWDC21
  - Swift concurrency: Behind the scenes - WWDC21
- Familiar with SwiftUI
  - SwiftUI by Example - Hacking with Swift

Test Theory

- Properties of a Good Test
- Dependency Injection
- Test Doubles
- Testing Asynchronous Behavior

Properties of a test

- Be Short
- Be Simple
- Test one thing
- Check one behavior
- Run as quickly as possible

```
func testSimpleAdder() {
    // Arrange
    let subject = Adder()

    // Act
    let result = subject.add(2, 3)

    // Assert
    expect(result).to(equal(5))
}
```

Properties of a test

- Run as quickly as possible
  - Fast test suites provide more value
  - Like build time reduction, test runtime reduction pays off dramatically

Dependency Injection

- Providing dependencies to an object

```
struct NoInjectedDeps {
    let a = DependencyA()
    let b = DependencyB.shared
}
```

```
struct HasInjectedDeps {
    init(
        a: DependencyA,
        b: DependencyB
    ) {
        // ...
    }
}
```

Dependency Injection

- Improves your app design
  - Decouples your object graph
  - Encourages you to make your objects smaller
- Allows you to inject test doubles

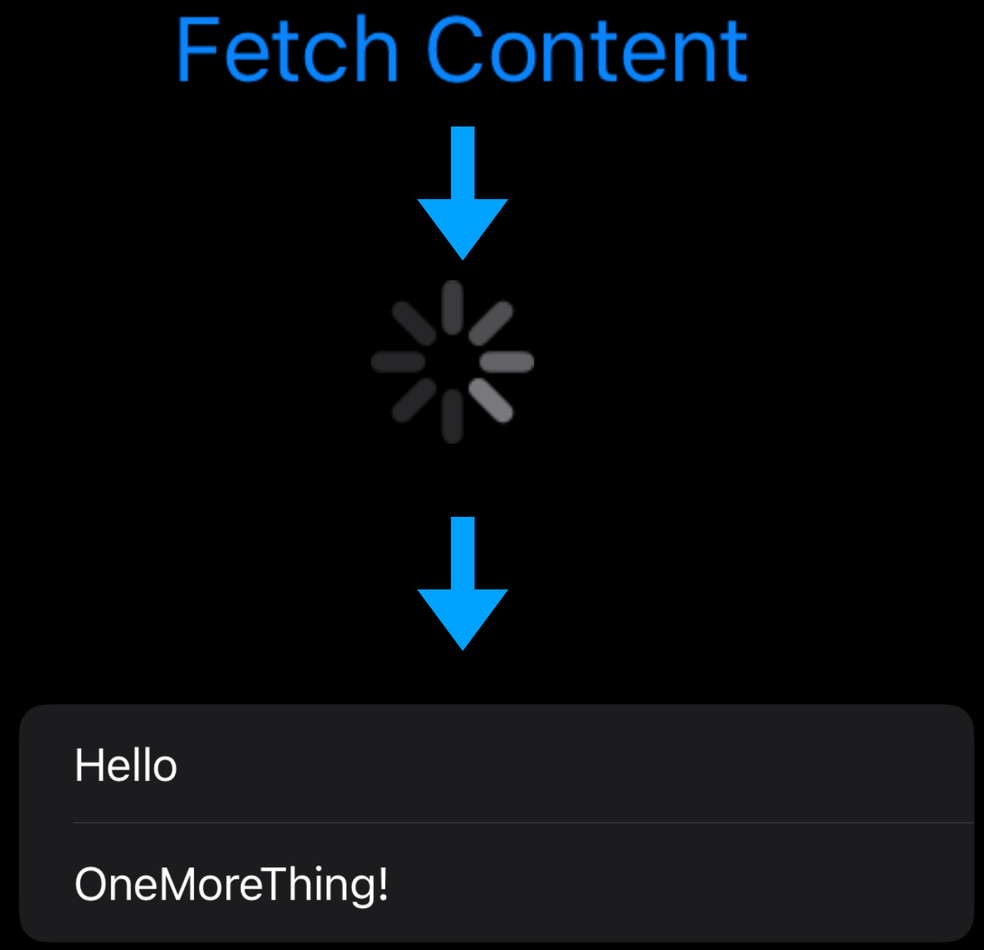
```
struct HasInjectedDeps {
    init(
        a: DependencyA,
        b: DependencyB
    ) {
        // ...
    }
}
```

Test Doubles

- Stand-ins for production code for the purpose of aiding tests
- Mock: Test double that **asserts** on arguments **at call time**
- Spy: Test double that **records** arguments **for later examination**
- Fake: Test double to replace an object or protocol implementation

Testing Asynchronous Behavior

- Asynchronous Code has 2 States
  - In Progress
  - Finished



Testing Swift Concurrency

```
struct OMTJsonService: OMTService {
    let client: HTTPClient

    func omtDemo() async throws -> [String] {
        let url = URL(
            string: "https://demos.rachelbrindle.com/omt2024.json"
        )!
        let (data, _) = try await client.data(
            for: URLRequest(url: url)
        )
        return try JSONDecoder().decode(
            [String].self,
            from: data
        )
    }
}
```

Testing Swift Concurrency

```
struct OMTJsonService: OMTService {
    let client: HTTPClient

    func getItems() async throws -> [String] {
        let url = URL(
            string: "https://demos.rachelbrindle.com/omt2024.json"
        )!
        let (data, _) = try await client.data(
            for: URLRequest(url: url)
        )
        return try JSONDecoder().decode(
            [String].self,
            from: data
        )
    }
}
```

Call network client with the correct URLRequest

If network call is unsuccessful, throw error

Data is returned as a list of strings

If unable to decode data, throw error

Testing Swift Concurrency

```
let (data, _) = try await client.data(
    for: URLRequest(url: url)
)
```

- Don’t make network calls in test
  - Slow
  - Unreliable
  - Can’t test the failure conditions
- Inject a test double instead

Swift Concurrency Test Doubles

- Swift Concurrency **requires** async calls to eventually be resolved
  - Otherwise you have deadlocks and crashes
- Test doubles need to take extra precaution to make sure to resolve async calls
- Naive approach: treat Swift Concurrency methods like synchronous methods and immediately return a value.
  - But then you can’t check in-progress states

Swift Fakes - Test Doubles for Swift

- github.com/quick/swift-fakes
- Offers `Pendable` and `Spy`
- `Pendable`: Stand-in for asynchronous value
- `Spy`: Stand in for implementation of any function

`Pendable<Value>`

- Lets you resolve async functions on demand
  - Allows resolving before and after calls
- Requires you to provide a fallback value on init
- Autoresolves with fallback after (default) 2 seconds
- Used as the return value of a Spy

Spy<Arguments, Returning>

- Typesafe & Threadsafe way to record arguments & return a value
- Composable with `Result` and `Pendable`

```
let processSpy = Spy<
    (first: String, second: String),
    Int
>(1)
func process(
    first: String,
    second: String
) -> Int {
    processSpy((first, second))
}
```

Async Test Subjects

- No In-Progress State?
  - Use `await`
  - Requires you to pre-resolve `Pendable`s

```
func testHandlesCorrectData() async throws {
    // Arrange
    let httpClient = FakeHTTPClient()
    // pre-resolve httpClient
    let subject = Service(client: httpClient)

    // Act
    let value = try await subject.omtDemo()

    // Assert
    expect(value).to(equal(["hello", "omt"]))
}
```

Async Test Subjects

- Checking In Progress State: use `async let` or `Task`
  - `async let` values must be awaited before asserting
  - `Task`s can be passed in to assertion, checked using `value` or `result`
  - Use `Task` when checking the value, otherwise `async let`

Checking Background Behavior

- XCTest provides expectation(…) for observing callbacks
- Nimble has polling expectations
  - `toEventually`, `toEventuallyNot`, `toNever`, `toAlways`
  - `expect(myObject.property).toEventually(equal(...))`

Demo

```swift
protocol HTTPClient {
    func data(for: URLRequest) async throws -> (Data, URLResponse)
}

extension URLSession: HTTPClient {}

protocol OMTService {
    func omtDemo() async throws -> [String]
}

struct OMTJsonService: OMTService {
    let client: HTTPClient

    func omtDemo() async throws -> [String] {
        let url = URL(
            string: "https://demos.rachelbrindle.com/omt2024.json"
        )!
        let (data, _) = try await client.data(
            for: URLRequest(url: url)
        )
        return try JSONDecoder().decode(
            [String].self,
            from: data
        )
    }
}
```

Testing SwiftUI

```
@MainActor
struct Refresher: View {
    let action: () async -> Void

    @State var isRefreshing = false

    var body: some View {
        if isRefreshing {
            ProgressView()
        } else {
            Button("Refresh") {
                isRefreshing = true
                Task {
                    await action()
                    isRefreshing = false
                }
            }
        }
    }
}
```

Testing SwiftUI

```
@MainActor
struct Refresher: View {
    let action: () async -> Void

    @State var isRefreshing = false

    var body: some View {
        if isRefreshing {
            ProgressView()
        } else {
            Button("Refresh") {
                isRefreshing = true
                Task {
                    await action()
                    isRefreshing = false
                }
            }
        }
    }
}
```

Button starts in the right state

Immediately shows progress view when button pressed

Calls action callback when button pressed

Shows Button again when action callback finishes

How to test Views

- SwiftUI Previews
  - Not automated
- UI Tests
  - Meant for app-wide tests, not individual views
  - Slow

How to test Views

- Fast, targeted, & automated
- Inspect & Manipulate the View
- ViewInspector - github.com/nalexn/ViewInspector

```
func testTappingButtonCallsCallback() throws {
    // Arrange
    let actionSpy = PendableSpy<Void, Void>()
    let subject = Refresher {
        await actionSpy().call()
    }
    // Act
    try subject.inspect().find(button: "Refresh").tap()
    // Assert
    expect(actionSpy).toEventually(beCalled())
}
```

Demo

```swift
import SwiftUI

@MainActor
struct Refresher: View {
    let action: () async -> Void

    @State var isRefreshing = false

    var body: some View {
        if isRefreshing {
            ProgressView()
        } else {
            Button("Refresh") {
                isRefreshing = true
                Task {
                    await action()
                    isRefreshing = false
                }
            }
        }
    }
}

```

Snapshot Testing

- Image comparison to a known-good image
- Great for catching visual regressions
- ios-snapshot-test-case - github.com/uber/ios-snapshot-test-case
- Nimble-Snapshots - github.com/ashfurrow/Nimble-Snapshots
- SwiftUI Views need to be in a window in order to render

Demo

```swift
final class RefresherController {
    init(action: @escaping () async -> Void) {
    }

    func refresh() {
        isRefreshing = true
        Task {
            await action()
            isRefreshing = false
        }
    }
}

@MainActor
struct Refresher: View {
    let controller: RefresherController

    var body: some View {
        if controller.isRefreshing {
            ProgressView()
        } else {
            Button("Refresh") {
                controller.refresh()
            }
        }
    }
}
```

Recap

- Theory
  - Properties of a great test
  - Dependency Injection
  - Test Doubles
  - Testing Asynchronous Behavior
- Testing Swift Concurrency
  - Swift Fakes - Spy & Pendable
    - github.com/quick/swift-fakes
  - Invoking async functions in test
  - Using Nimble’s polling expectations

Recap

- Unit Testing SwiftUI Views
  - ViewInspector - github.com/nalexn/ViewInspector
  - Snapshot Testing
    - github.com/uber/ios-snapshot-test-case
    - github.com/ashfurrow/Nimble-Snapshots

Thank You!

Rachel Brindle

@younata@hachyderm.io

github.com/quick/Quick

github.com/quick/Nimble

github.com/quick/swift-fakes