# SwiftUI Snapshot Testing

Episode #86 • Dec 23, 2019 • Free Episode

In this week's free holiday episode we show what it looks like to snapshot test a SwiftUI application in our architecture and compare this style of integration testing against XCTest's UI testing tools.






## Introduction

00:05

Today we have a surprise holiday edition of Point-Free! We wanted to end the year on a free episode for everyone to enjoy, and we wanted to cover a fun, lighthearted topic. So today's episode will be a little different than our usual ones. It'll be a little shorter and we won't be digging super deep into abstract topics.

00:20

We want to add screenshot testing to the application we have been building. We've already seen that the Composable Architecture we've been covering for the past 18 weeks gives us the ability to write incredibly deep tests with very little work, but we want to take things to the next level.

00:26

We will show that the snapshot testing library we open sourced over a year ago can be combined with the Composable Architecture and SwiftUI to create some amazing, end-to-end testing of our application.

00:38

Let's get started!

## Adding the SnapshotTesting module

00:41

We will start by adding the snapshot testing library to our PrimeTime application. We can do this by adding a new Swift package dependency to our project. We can go to **File**, **Swift Packages**, **Add Package Dependency…**.

01:02

We can then type in the URL of our library:

> https://github.com/pointfreeco/swift-snapshot-testing

01:07

And then we can choose which version we want to use:

01:18

And finally we can choose which target we want to add this dependency to. We will be writing our tests in the `Counter` module, so let's add it there.

01:32

And just like that a dependency has been added to our project. A lot easier than it used to be! Just to show where these dependencies live, we can open up our project settings and check out the new "Swift Packages" tab.

01:42

With this done we can already go into our counter tests file and import `SnapshotTesting`.

```
import SnapshotTesting
```

## Snapshot testing a SwiftUI view

01:56

So that was surprisingly easy. It's awesome that we now have properly Swift Package Manager support for Xcode.

02:02

Next, let's try writing our first test. We could add some snapshot testing to one of our existing tests, but just for clarity let's create a new test to play with:

```
func testSnapshots() {
}
```

02:13

Recall that we make snapshot assertions by using the `assertSnapshot` helper from the library. It requires a value that we want to match and a snapshot strategy that determines how a snapshot is created and compared with any existing artifact on disk:

```
assertSnapshot(matching: Value, as: Snapshotting<Value, Format>)
```

02:40

The thing we wanna snapshot is our counter view, which can be constructed like so:

```
let view = CounterView(store: Store<CounterViewState, CounterViewAc
```

02:48

But to do that we need to provide a store:

```
let store = Store(
  initialValue: CounterViewState(),
  reducer: counterViewReducer
)
```

03:00

And to do that we need to import the Composable Architecture:

```
import ComposableArchitecture
```

03:26

And now we can create our view:

```
let view = CounterView(store: store)

assertSnapshot(matching: view, as: Snapshotting<Value, Format>)
```

03:29

Then the question is, which snapshot strategy do we use? We want to ultimately snapshot this view into an image, so we could try:

```
assertSnapshot(matching: view, as: .image)
```

03:41

> 🔴 Generic parameter 'Format' could not be inferred 🔴 Type 'Snapshotting<CounterView, Any>' has no member 'image'

However, we don't currently have an image snapshot strategy defined for SwiftUI views. We have strategies for `CALayer`s, `UIView`s, `UIViewController`s and more, but nothing for SwiftUI views.

03:54

We could try to create one from scratch, but before even try to do that we could also just wrap this view in a hosting controller, which is a `UIViewController` subclass, and then try out our snapshot strategies on it.

04:06

Let's create a hosting controller:

```
let vc = UIHostingController(rootView: view)
```

04:10

> 🔴 Use of unresolved identifier 'UIHostingController'

Well, we also need to import `SwiftUI` to get access to that:

```
import SwiftUI
```

04:21

And then we can try snapshotting this controller:

```
assertSnapshot(matching: vc, as: .image)
```

04:24

If we run tests this fails as expected…

04:30

> ❌ failed - No reference was found on disk. Automatically recorded snapshot:
> …

…because there was no reference file on disk and so we had to record a new snapshot.

04:40

We can use this handy shortcut that the test failure message gives us to open the snapshot that was just recorded to disk.

04:44

Well, that's not right. It seems that by default the hosting controller doesn't adjust its size based on the content of the view. We could manually set the controller's size to something hard coded, but a slightly faster way would be just to take the screen's bounds:

```
vc.view.frame = UIScreen.main.bounds
```

05:12

Now let's put our test into record mode so that we can record a fresh snapshot:

```
record=true
```

05:20

And when we run we get a failure, as expected since we are in record mode:

05:22

> ❌ failed - Record mode is on. Turn record mode off and re-run "testSnapshots" to test against the newly-recorded snapshot.

But if we now open our snapshot artifact we will see it has a good size.

05:30

And that's all it takes to get snapshot testing working with SwiftUI!

## Snapshot testing our architecture

05:36

But let's take this to the next level by flexing some of the muscles we gained from introducing the Composable Architecture. What if I wanted to simulate the user tapping on the increment button? As we have seen previously, everything inside the the body of a SwiftUI is basically hidden from us. We have no ability to go in and simulate tapping one of the buttons on the inside.

06:06

However, we control the store, which is the thing really powering our view, and we can easily send it user actions along the way.

06:20

Let's write another snapshot assertion, but before we do, let's send the store an action representing a user tapping the increment button.

```
store.send(.counter(.incrTapped))
assertSnapshot(matching: vc, as: .image)
```

06:36

And amazingly we get a screenshot of what the UI looks like after that button was tapped, in particular the count has now gone up to 1.

07:00

Let's try it again, just for fun:

```
store.send(.counter(.incrTapped))
assertSnapshot(matching: vc, as: .image)
```

07:05

And now we get a screenshot that counts up to 2.

07:21

So we are seeing that we can basically play a script of user actions in the UI, and capture an image of what the UI looks like at each step of the way. That is already seeming pretty powerful.

07:45

But let's keep going, because there are more things the user can do in this screen. For example, they can tap the "nth prime" button:

```
store.send(.counter(.nthPrimeButtonTapped))
assertSnapshot(matching: vc, as: .image)
```

08:11

Looking at the screenshot recorded we see that the "nth prime" button has been disabled, and this is exactly what we expect since that button is disabled while the API request is in flight.

08:35

Now, the API request that is fired off when this button is tapped is fully controlled by our environment (in particular in the line where we set `Current = .mock`), so we don't have to worry about it accidentally reaching out into the real world. We do however have to wait a small amount of time to get the response from our mocked API client, and this is because we used `receive(on:)` from Combine when using the effect:

```
return [
  Current.nthPrime(state.count)
    .map(CounterAction.nthPrimeResponse)
    .receive(on: DispatchQueue.main)
    .eraseToEffect()
]
```

09:12

This causes a thread hop, which means even if our mocked `nthPrime` effect returns immediately the overall effect will need a tick of the runloop to emit. Once we wait that small amount of time we would hope that we get an alert on the screen showing us what the nth prime is. So let's do that with an `XCTestExpectation` to wait for a fraction of a second before snapshotting again:

```
let expectation = self.expectation(description: "wait")
DispatchQueue.main.asyncAfter(deadline: .now() + 0.01) {
  expectation.fulfill()
}
self.wait(for: [expectation], timeout: 0.01)
assertSnapshot(matching: vc, as: .image)
```

10:08

If we inspect this new image we will see that the "nth prime" button has gone back to enabled, which is what we expect since the API request completed. But unfortunately no alert is being shown. Maybe it's just a matter of the alert doing an animation and that taking some time to show. Let's try making the timeout interval a little longer:

```
let expectation = self.expectation(description: "wait")
DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
  expectation.fulfill()
}
self.wait(for: [expectation], timeout: 0.5)
assertSnapshot(matching: vc, as: .image)
```

10:50

Unfortunately we get the same results.

## Snapshot testing alerts

10:57

Turns out, alerts don't render in the view controller that presents them. They are instead rendered at the window level, and we are not getting access to that right now because we are only snapshotting the view controller.

11:22

Let's try tackling this problem by exploring it a bit further.

11:29

Because alerts are rendered at the window level, we might be tempted to create a window and slap a view controller into it:

```
let window = UIWindow()
window.rootViewController = vc
assertSnapshot(matching: window, as: .image)
```

11:44

Turns out we can only hope to snapshot alerts if we actually run our tests in a host application. Now you might remember a few episodes ago when we first started exploring testability we explicitly disabled the host application because it slows down the test feedback cycle. Running a test in a host application requires launching the application in the simulator, and removing that step can save a few seconds or more off of test run time.

12:11

So, we are going to turn it back on for the counter module. In your own code you may want to split out tests that do this kind of window-level snapshot testing into another test target that runs with a host application, and leave all of your reducer unit tests in a target that has that option turned off. That way your reducer tests can be nice and speedy, and your snapshot tests can get full access to the simulator environment.

12:38

And rather than doing all of this messy window management stuff directly inline, let's create a new snapshot strategy that can encapsulate this work. I'm not going to create it from scratch but instead just gonna paste in the final answer:

```
extension Snapshotting where Value: UIViewController, Format == UII
  static var windowedImage: Snapshotting {
    return Snapshotting<UIImage, UIImage>.image.asyncPullback { vc :
      Async<UIImage> { callback in
        UIView.setAnimationsEnabled(false)
        let window = UIApplication.shared.windows.first!
        window.rootViewController = vc
        DispatchQueue.main.async {
          let image = UIGraphicsImageRenderer(bounds: window.bounds
            window.drawHierarchy(in: window.bounds, afterScreenUpda
          }
          callback(image)
          UIView.setAnimationsEnabled(true)
        }
      }
    }
  }
}
```

12:58

There's a lot going on in here, but some of it is to look out for future problems we might have. In a nutshell this is doing the following:

13:02

- First we are leveraging the image snapshot strategy on `UIImage` to power this by pulling it back to work on `UIViewController`. This means that if the closure provided can somehow convert a `UIViewController` into a `UIImage`, then the base image strategy will take care of the rest.

13:18

- To do this we get access to the first window in our application, and we slap the controller into the window's root.

13:28

- Then we turn the controller into an image by invoking window's `drawHierarchy` method, which is necessary to capture the alert views sitting inside the window.

13:40

- And finally we send that image back to the snapshotting strategy by invoking the callback.

13:44

- We also have a few other things happening in here to help prevent future problems:

13:46

- We turn off animations when we start the snapshot and turn it back on when we finish. This means we don't have to wait for certain animations to finish before snapshotting, like modals and the like. Unfortunately alerts seem to not fully respect this setting, and so when snapshotting alerts will still need to wait a bit of time in the test.

14:06

- We also need to wait for a single tick of the runloop to pass before we can snapshot, because even if animations are turned off, views are not put into their final state until a tick has happened.

14:18

Now let's change all of our snapshots to use this strategy:

```
assertSnapshot(matching: vc, as: .windowedImage)

store.send(.counter(.incrTapped))
assertSnapshot(matching: vc, as: .windowedImage)

store.send(.counter(.incrTapped))
assertSnapshot(matching: vc, as: .windowedImage)

store.send(.counter(.decrTapped))
assertSnapshot(matching: vc, as: .windowedImage)

store.send(.counter(.nthPrimeButtonTapped))
assertSnapshot(matching: vc, as: .windowedImage)

let expectation = self.expectation(description: "wait")
DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
  expectation.fulfill()
}
self.wait(for: [expectation], timeout: 0.5)
assertSnapshot(matching: vc, as: .windowedImage)
```

14:29

Now when we run our tests, lo and behold we get a snapshot of the alert!

14:46

And even better, we can also send the action of the user dismissing the alert to show that it should go away:

```
store.send(.counter(.alertDismissButtonTapped))
expectation = self.expectation(description: "wait")
DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
  expectation.fulfill()
}
self.wait(for: [expectation], timeout: 0.5)
assertSnapshot(matching: vc, as: .windowedImage)
```

15:13

And if we inspect the snapshot its back to a screen with no alert.

15:23

This is so powerful. We are able to play a simple user script by sending actions to the store and then take screenshots of what the application looks like at each step of the way. We are even automatically running effects in the store and being able to see how the UI reacts to those effects. For example, when we sent the `nthPrimeButtonTapped` action, that causes an actual effect to be fired off, one that is supposed to reach out to the world but we have controlled with the environment, and when that effect is done it feeds the result back into the store, which triggers the alert to show. This is true end-to-end testing!

## Snapshot testing modals

16:03

But it gets better. Much better.

16:05

Remember that embedded inside this counter view is another piece of functionality: the prime modal view. All of the logic and UI for that feature lives in a completely different module, and we have unit tested it isolation. We also wrote a few integration tests on the reducer that proves that when the prime modal functionality is embedded in the counter reducer they play nicely together.

16:30

All of this is possible, but with one small caveat. Let's take a look…

16:35

The caveat is that currently the presentation state of the prime modal is currently held in a `@State` field on our counter view:

```
public struct CounterView: View {
  // …
```

```
  @State var isPrimeModalShown = false
```

16:41

In fact, it is the last `@State` field we have anywhere in our app. Previously we moved a few fields like this to our `AppState` so that we could control them from our reducers, which allowed us to write tests on that functionality, which otherwise would have been impossible.

17:01

We need to do one more refactor in order to lift this state up into `AppState`, and then we will be able to accomplish everything we just described. It's very straightforward and mechanical to do this, but in the spirit of the holidays let's keep things lighthearted and just skip straight to the final code.

17:11

We're not going to worry about all the details of how the refactor works, our viewers can check out the episode code sample to look deeper into it. But the most important change in the refactor is the additional of two new actions to the `CounterAction`:

```
public enum CounterAction: Equatable {
  // …
  case isPrimeButtonTapped
  case primeModalDismissed
}
```

17:28

These actions are sent to the store when the user taps on the "is this prime?" button and when the prime modal is dismissed. That right there is enough for us to write even better tests. After our last assertion we can further tap on the "is this prime?" button to see what happens:

```
store.send(.counter(.isPrimeButtonTapped))
assertSnapshot(matching: vc, as: .windowedImage)
```

18:20

If we run this and pull up the latest snapshot we will see the prime modal presented over the counter view.

18:59

Let's go ahead and play out more of the user script. Let's also save the prime to their favorites:

```
store.send(.primeModal(.saveFavoritePrimeTapped))
assertSnapshot(matching: vc, as: .windowedImage)
```

19:15

If we pull up the snapshot we will see exactly what we expect.

19:33

Finally, let's dismiss the modal.

```
store.send(.counter(.primeModalDismissed))
assertSnapshot(matching: vc, as: .windowedImage)
```

20:04

I think this is absolutely amazing. We are getting end-to-end test coverage of a flow in the app that works with multiple features, multiple screens, side effects, and it's all being driven off the store.

## What's the point?

20:32

So, this is definitely very cool, but at the end of each episode on Point-Free we like to ask "what's the point?" This is our opportunity to check ourselves and make sure that we are actually making something practical and useful. And in this case, it just so happens that Apple gives us a technology that is supposed to allow us to write precisely these kinds of tests, and it's called XCUITest. Why would we invest time and energy into this style of testing when there's a solution provided by Apple?

21:01

While it is true that XCUITest aims to solve a similar problem to what we have done here, it isn't quite the same. We believe XCUITest wants to solve a much bigger problem than what we have done, which is to simulate what happens when someone actually taps on things and interacts with the app. This is a much more difficult problem to solve, and unfortunately that means XCUITest is a lot harder to work with and a lot more flaky. To see why, let's try to recreate the test we just wrote in XCUITest.

21:44

We start by creating a new target in our project, and we use the **UI Test Bundle** template:

22:10

This creates a test for us with some things stubbed out, but we can clean it up to just be the minimal work needed to get an XCUITest going:

```
import XCTest

class PrimeTimeUITests: XCTestCase {
  override func setUp() {
    continueAfterFailure = false
  }

  func testExample() {
    let app = XCUIApplication()
    app.launch()
  }
}
```

22:51

And then the way we write our first test is to hit the little record button, which will launch a simulator, and then click around to do whatever we want. In particular, let's repeat what our other test did, which was to increment twice, ask for the "nth prime", then ask if the current count is prime, save it as a favorite, and then dismiss the modal.

24:10

After we do this we see that Xcode has inserted a bunch of code for us. Unfortunately, it seems to have inserted too much, because it repeated this line…

```
let app = XCUIApplication()
```

…which is a compiler error since that variable was already defined above. Let's just delete it for now. Now all that's left is this:

```
app.tables.buttons["Counter demo"].tap()

let button = app.buttons["+"]
button.tap()
button.tap()
app.buttons["What is the 2nd prime?"].tap()
app.alerts["The 2nd prime is 3"].scrollViews.otherElements.buttons['
app.buttons["Is this prime?"].tap()
app.buttons["Save to favorite primes"].tap()
app.children(matching: .window).element(boundBy: 0).children(matchi
```

24:29

It's mostly readable, but it's a little intense. We can certainly tell that it has some API in order to search the view hierarchy for certain elements, but it's a stringy API. We are searching by the title of the button, which is fragile. If we ever change the title then this test will break. Alternatively we can set an accessibility identifier on the button, and then use that instead of the title. However, it's still just a string, and so we will have no static guarantees that things are spelled properly.

25:20

There are also some strange implementation details leaking out. For example, in order to find the "OK" button inside the alert, it needed to traverse into the alert (which is found by its string title), and then we needed to know to traverse into a scroll view (for some reason), and then finally search it for the "OK" button so that we could tap it. This means changes to the view hierarchy could break this test in the future.

26:00

And finally there is this super strange line of code that is responsible for simulating the swipe down gesture I did to dismiss the modal. I really have no idea what any of this code is doing, and would never be able to reproduce it myself.

26:18

But, regardless of those weird things, at least we now have the test. Though, what exactly is it testing? It is only testing that these various UI elements exist. We need to write additional code to make more nuanced asserts, like what value is held in the counter label, which requires more strange view traversals:

```
XCTAssert(app.staticTexts["1"].exists)
button.tap()
XCTAssert(app.staticTexts["1"].exists)
```

27:01

But all of that aside, at least we have our test. So let's run it to make sure it works.

```
app.alerts["The 2nd prime is 3"].scrollViews.otherElements.buttons['
```

27:17

> Failed to get matching snapshot: No matches found for Descendants matching type Alert from input {{ Application, pid: 54184, label: 'PrimeTime' }}

Well, already a test failure. It seems that the alert isn't found when we try to make this assertion. This is because that alert only shows when we get a response from the API request, and currently the test app is making a live API request instead of using our mock. Unfortunately we can't even mock our the environment like we would for a normal test:

```
@testable import Counter

class PrimeTimeUITests: XCTestCase {
  override func setUp() {
    continueAfterFailure = false
    Current = .mock
  }
```

28:02

And this is because the test application is launched in a completely separate process from the process that runs this test code. So we aren't affecting the environment that is used in that process.

28:23

Perhaps the simplest way to fix this test is to allow the test to wait until the alert because visible before executing the rest of the test:

```
let alert = app.alerts["The 2nd prime is 3"]
XCTAssert(alert.waitForExistence(timeout: 5))
alert.scrollViews.otherElements.buttons["OK"].tap()
```

29:23

And now the tests pass. But this is problematic for two reasons:

29:32

- First, it isn't great to be hitting live API endpoints in tests. That is a recipe for flaky, unreliable tests, where tests can sporadically fail based on the network availability, API availability, and a variety of other factors. The nice thing about controlling our dependencies was that we could create a reliable, reproducible environment for our application's code to run in.

29:53

- Second, this test now takes a very long time to run. In fact, it takes about 10 seconds. If we go back to our screen test suite we will see that it runs in about 2 seconds (do that), and the majority of that time is spent just computing screenshots.

30:10

So, let's try to see what it would take to mock out our dependencies when running these tests. We can't do it directly from this test code since the app runs in a different process, so what we can do is set an environment variable on the launched application and use that to mock out dependencies in the application code:

```
let app = XCUIApplication()
app.launchEnvironment["UI_TESTS"] = "1"
app.launch()
```

```
@testable import Counter

class AppDelegate: UIResponder, UIApplicationDelegate {
  func application(_ application: UIApplication, didFinishLaunching

    if ProcessInfo.processInfo.environment["UI_TESTS"] == "1" {
      Counter.Current.nthPrime = { _ in .sync { 3 } }
    }

    return true
  }
```

Unfortunately, we would have to make `Current` public to make it accessible, but let's do a `@testable` import to get things building.

31:08

Now when we run tests they still pass, but they ran a lil faster, although still about 8 seconds. One thing we can do to speed this up is disable animations like we did for our snapshot testing. But we can't do this in the test file, we have to add more test code to our app delegate:

```
if ProcessInfo.processInfo.environment["UI_TESTS"] == "1" {
  UIView.setAnimationsEnabled(false)
  Counter.Current.nthPrime = { _ in .sync { 3 } }
}
```

31:43

Now tests run a little faster, about 5-6 seconds. Still 3 times slower than our screenshot tests, and we aren't even trying to take screenshots of our UI.

32:03

However, even this isn't ideal. We no longer have a direct way to mock dependencies in nuanced ways directly in our tests. In our previous test we could simply copy and paste the script we had, swap out the `nthPrime` dependency for one that fails and we would immediately get test coverage on the unhappy path. To do the same for a XCUITest we would need to set an additional environment variable that we could interpret in the app delegate to change how we mock out dependencies.

## Conclusion

32:53

So, what we have seen here is that using XCUITest is slower, flakier and more difficult to use for this particular test. If we are simply trying to exercise a bit of user interaction in a screen and a little bit of integration between two features, we think that testing with just the store directly, whether it be as a unit test on its state or as screenshot tests, is going to give us more bang for the buck. We can very succinctly describe a script of user actions, assert exactly what happens after each action (including how side effects behave), and can take a screenshot of the UI after each step.

33:22

XCUITest still has a use in your arsenal of development tools. First of all, it works on any app, regardless of architecture or how well structured it is. It's amazing you are able to take any messy, legacy application and get some kind of test coverage on it. In fact, it can be super useful for refactoring a screen that is difficult to test, in which you get some basic XCUITest coverage on a screen, and then start refactoring it into a more testable shape, all the while verifying that your XCUITest doesn't fail. And further, XCUITest is always going to be useful for the times you want true end-to-end testing of your application, allowing you to simulate app start and user interactions.

33:59

OK, that's it for this episode. We hope everyone enjoyed this little holiday surprise episode. We hope everyone has wonder rest of 2019, and we'll see you in the new year!

## References

### Testing and Declarative UI's

Nataliya Patsovska • Thursday Oct 24, 2019

Nataliya gives a great talk on some of the ways to test SwiftUI views, and leverages Xcode previews to make snapshot testing even more powerful.

> With SwiftUI and Combine, Apple is changing its approach for how we define data flows and UI. As we move from playing with sample code to writing production apps, it's time to start thinking about testing. Nataliya will show how to apply several learnings from her experience with declarative UIs to this new reality.

https://www.youtube.com/watch?v=tk0HzScvW2M

### pointfreeco/swift-snapshot-testing

Brandon Williams & Stephen Celis • Monday Dec 3, 2018

A delightful snapshot testing library that we designed over the course of many Point-Free episodes. It allows you to snapshot any type into any format, comes with many snapshot strategies out of the box, and allows you to define your own custom, domain-specific snapshot strategies for your types.

https://github.com/pointfreeco/swift-snapshot-testing

### Snapshot Testing in Swift

Stephen Celis • Friday Sep 1, 2017

Stephen gave an overview of snapshot testing, its benefits, and how one may snapshot Swift data types, walking through a minimal implementation.

https://www.stephencelis.com/2017/09/snapshot-testing-in-swift

### Elm: A delightful language for reliable webapps

Elm is both a pure functional language and framework for creating web applications in a declarative fashion. It was instrumental in pushing functional programming ideas into the mainstream, and demonstrating how an application could be represented by a simple pure function from state and actions to state.

https://elm-lang.org

### Redux: A predictable state container for JavaScript apps.

The idea of modeling an application's architecture on simple reducer functions was popularized by Redux, a state management library for React, which in turn took a lot of inspiration from Elm.

https://redux.js.org

### Composable Reducers

Brandon Williams • Tuesday Oct 10, 2017

A talk that Brandon gave at the 2017 Functional Swift conference in Berlin. The talk contains a brief account of many of the ideas covered in our series of episodes on "Composable State Management".

https://www.youtube.com/watch?v=QOligosUNGU

## Downloads

**Sample Code**

0086-swiftui-snapshot-testing