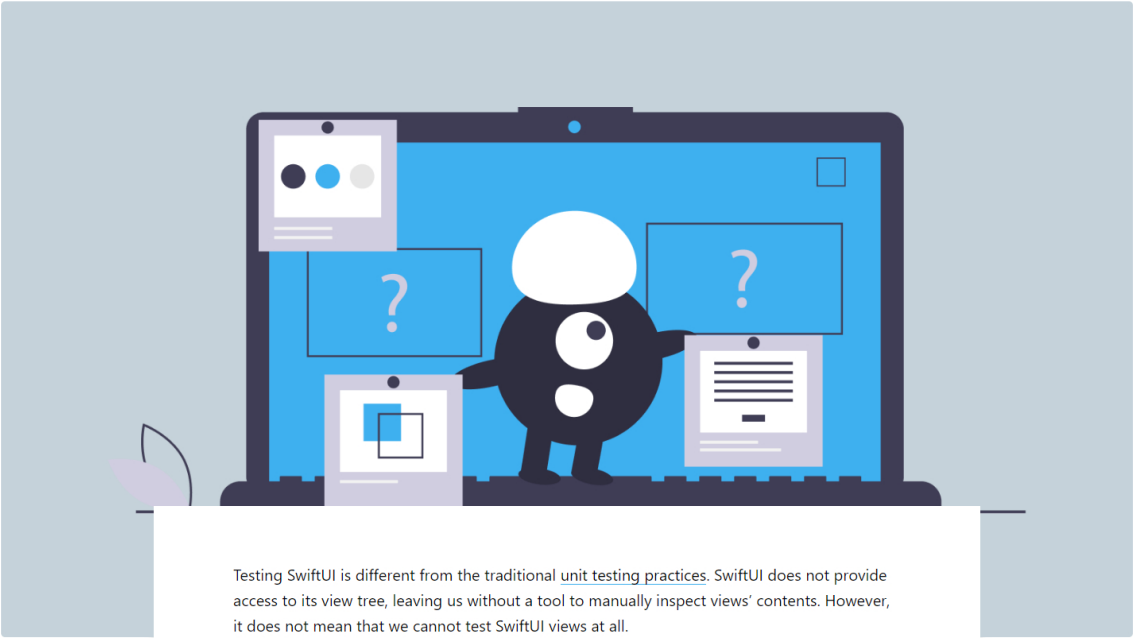
5 MAY 2020 / TESTING,SWIFTUI

Testing SwiftUI Views



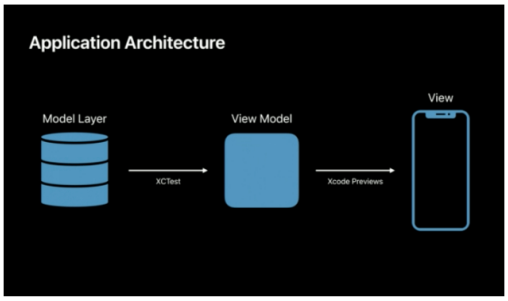
Testing SwiftUI is different from the traditional unit testing practices. SwiftUI does not provide access to its view tree, leaving us without a tool to manually inspect views' contents. However, it does not mean that we cannot test SwiftUI views at all.

In this article, we are going to explore the topic of testing SwiftUI views, covering the questions:

- What is Apple's suggested way of testing SwiftUI views?
- Why Apple's solution does not work?
- Why snapshot testing is the only viable way of testing SwiftUI views?
- What is snapshot testing?
- How can we benefit from it?
- How to snapshot test view appearance and presentation logic in SwiftUI?
- What are snapshot testing best practices?
- What are the downsides of snapshot testing?

Problem Statement

In WWDC2019 Mastering Xcode Previews, Apple shows us its opinion on testing SwiftUI views:



According to Apple, to test our SwiftUI views, we need to create Xcode previews for a variety of view states. Then manually inspect every preview to find visual regressions.

Although this may be sufficient for prototypes and proofs of concept, I recommend against following this approach. Here is why previews are not a replacement for an automated test suite:

- Manual inspection is error-prone, unreliable and tedious.
- It does not scale well. What if an app supports multiple languages, light and dark themes, dynamic content sizes?
- Xcode previews are in their early stage of development. They have lots of defects and instabilities.

The solution is *snapshot testing*.

Snapshot Testing

Snapshot testing is a software testing method that makes sure your UI does not have unexpected changes (i.e. *regressions*).

A *snapshot test* works differently than a regular unit test. In a unit test, we run some code to verify that a known fixed input produces a known fixed output. Conversely, in a snapshot test, we compare the output of a test to a previous test run's output, called the *baseline snapshot*.

A **snapshot** is a file of any serializable format that can be deterministically compared for equality. It can be a JSON blob, a Swift mirror, an image, you name it.

In snapshot tests, we render our SwiftUI views in a variety of states, take their snapshots, then compare them to the previously created baselines. A snapshot test passes if the current snapshot matches the baseline. Otherwise, a test fails, which means that (1) either the change is intended and we need to update the baseline snapshot, (2) or the change is unexpected and we need to fix our code.

Value of Snapshot Testing

Snapshot tests are not a replacement for unit tests. Unit tests are great for well defined and stable behavior. Conversely, snapshot tests are perfect for verifying behavior that changes frequently and is not clearly defined. iOS app user interface, and, especially, SwiftUI views, is a prominent example of this.

Let's discuss the benefits of snapshot testing your app's UI.

> 1. Verify what users see.

If you try to test a UI component, you will discover that it is almost impossible to verify that something looks good with a unit test. Such a test will also be very brittle since user interface changes very often and for many trivial reasons. Snapshot tests do not have such a limitation since they verify what users see.

> 2. Almost no efforts to update.

Updating failing tests is just a matter of taking new snapshots and storing them alongside the corresponding tests. Most libraries can do this automatically.

> 3. Trivial to write.

A typical test is just two-step: render a view, then compare the current view state with a baseline snapshot. A snapshot testing library usually takes care of capturing SwiftUI view as an image, image comparison, and diffs.

> 4. The only viable way of testing SwiftUI views.

Unit tests are not an option for testing SwiftUI views since SwiftUI doesn't provide access to the view tree. Therefore, snapshot testing is the only viable way of verifying SwiftUI views.

> 5. Obviously what has changed.

It is straightforward to spot what has changed when a snapshot test fails. Typically, you will be provided with three images: the baseline, current, and diff. It is clear what has changed and whether the change is expected.

> 6. Verify different appearances.

By leveraging the power of SwiftUI previews API, we can snapshot test our views on different devices, and even on different platforms. What is more, by overriding environment values, we can apply different locales, content size categories, light and dark modes, right-to-left, and left-to-right layout directions.

Testing View Appearance

The first thing we must do before we start writing tests is to pick a snapshot testing library. My choice is *SnapshotTesting* by *@pointfreeco*.

> This is not a tutorial on the SnapshotTesting library. Here you can learn about its features and basic usage, written by the authors.

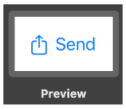
As an example, let's take the following SwiftUI component:

```
struct SendButton: View {
    let onAction: () -> Void = {}
```

```
    var body: some View {
        Button(
            action: onAction,
            label: {
                HStack {
                    Image(systemName: "square.and.arrow.up")
                    Text("common.button.send")
                }
        })
    }
}
```

The `SendButton` preview looks like:

```
struct SendButton_Preview: PreviewProvider {
    static var previews: some View {
        SendButton()
            .previewLayout(PreviewLayout.sizeThatFits)
            .padding()
    }
}
```



> Before you start, consider adding SwiftUI support to the SnapshotTesting library. Here is one way of doing it.

The first test verifies the default configuration of the button:

```
import XCTest
import SnapshotTesting
import SwiftUI
@testable import SnapshotTestingSwiftUI

// 1.
private let referenceSize = CGSize(width: 150, height: 50)

class SendButtonTests: XCTestCase {

    func testDefaultAppearance() {
        // 2.
        assertSnapshot(
            matching: SendButton().referenceFrame(), // 3.
            as: .image(size: referenceSize) // 4.
        )
    }
}

private extension SwiftUI.View {
    func referenceFrame() -> some View {
        self.frame(width: referenceSize.width, height: referenceSize.height)
    }
}
```

Here are the takeaways:

1. Make sure that the snapshot size equals to the size of the button.
2. Compare the current snapshot with the baseline.
3. Take `SendButton` snapshot.
4. Specify the kind of snapshot and size of the image taken.

> Here I discuss three techniques of managing SwiftUI previews at scale

If you run the test for the first time, it will fail. The library will create a reference snapshot alongside the failed test. For the second time, it will pass.

In the second test, let's verify right-to-left appearance:

```
func testRightToLeft() {
    let sut = SendButton()
        .referenceFrame()
        .environment(\.layoutDirection, .rightToLeft)

    assertSnapshot(matching: sut, as: .image(size: referenceSize))
}
```

The created snapshot looks like:

Another useful test is the localization check:

```
func testRuLocale() {
    let sut = SendButton()
        .referenceFrame()
        .environment(\.locale, Locale(identifier: "RU"))

    assertSnapshot(matching: sut, as: .image(size: referenceSize))
}
```

The created snapshot looks like:

Отправить

Testing Presentation Logic

Our next example is from the Landmarks app, which is a sample project from Apple SwiftUI tutorials:

```
struct Landmark {
    var name: String
    var imageName: String
    var isFavorite: Bool
}

struct LandmarkRow: View {
    var landmark: Landmark

    var body: some View {
        HStack {
            Image(landmark.imageName)
                .resizable()
                .frame(width: 50, height: 50)
            Text(landmark.name)
            Spacer()

            if landmark.isFavorite {
                Image(systemName: "star.fill")
                    .imageScale(.medium)
                    .foregroundColor(.yellow)
            }
        }
    }
}
```

Note that I've stripped some code from the Apple's tutorial to contain only what's relevant to our example.

First, verify that `LandmarkRow` correctly renders a `Landmark` model:

```
private let referenceSize = CGSize(width: 300, height: 70)

class LandmarkRowTests: XCTestCase {

    let landmark = Landmark(name: "Turtle Rock", imageName: "turtlerock", isFavorite: false)

    func testRenderLandmark() {
        assertSnapshot(
            matching: LandmarkRow(landmark: landmark).referenceFrame(),
            as: .image(size: referenceSize)
        )
    }
}

private extension SwiftUI.View {
    func referenceFrame() -> some View {
        self.frame(width: referenceSize.width, height: referenceSize.height)
    }
}
```

The snapshot looks like:

Turtle Rock

Second, verify that there is a star icon when landmark has been added to favorites:

```
class LandmarkRowTests: XCTestCase {
    ...
    let favoriteLandmark = Landmark(name: "Turtle Rock", imageName: "turtlerock", isFavorite: true)
    ...

    func testRenderFavorite() {
        assertSnapshot(
            matching: LandmarkRow(landmark: favoriteLandmark).referenceFrame(),
            as: .image(size: referenceSize)
        )
    }
}
```

Here is the baseline snapshot:

Turtle Rock ⭐

Best Practices

Treat snapshots as code

Consider snapshots as a part of a test. Commit them into the repository, and review them during your code review process [1].

Use descriptive snapshot names.

Snapshot name must describe the expected snapshot content. It must be straightforward to associate a snapshot with a test that relies on it. The SnapshotTesting library does a great job of this and stores snapshots under *'__Snapshots__/{TestCase}/{testMethod}'* [1].

Git

Consider using Git LFS as the project grows in size.

Disadvantages of Snapshot Testing on iOS

Snapshot testing is a great tool, it does, however, has its disadvantages:

- Tests coupling. A tiny change in a button may cause a cascade of failures in dozens of tests.
- When tests fail, it is very easy to update the snapshots without fixing the code and understanding the failure reason.
- Produce lots of false negatives since UI changes very often and for trivial reasons.
- Increases git repository size.

Source Code

You can find the complete source code here. It is published under the "Unlicense", which allows you to do whatever you want with it.

Further Reading

External resources that I can recommend:

- SnapshotTesting 1.0 and SwiftUI Snapshot Testing by @pointfreeco
- SwiftUI Snapshot Testing by @TrozWare
- Snapshot Testing with Jest
- Visual Regression Testing

My other articles on unit testing iOS apps with Swift:

- Real-World Unit Testing in Swift
- iOS Unit Testing Best Practices
- Unit Testing Asynchronous Code in Swift
- Async Unit Testing with Busy Assertion Pattern

Thanks for reading!

If you enjoyed this post, be sure to follow me on Twitter to keep up with the new content. There I write daily on iOS development, programming, and Swift.

Vadim Bulavin
Creator of Yet Another Swift Blog. Senior iOS Engineer at Pluto TV. Coding for fun since 2008, for food since 2012.

Follow

Unit Testing View Controllers and Views in Swift

See all 6 posts →

VADIM BULAVIN 7 MIN READ

compiler level? How to implement a custom function builder? These are the questions to answer in this article.

VADIM BULAVIN 5 MIN READ