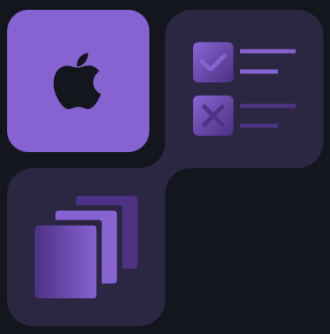


Snapshot Testing Tutorial for SwiftUI: Getting Started

Share

Aug 9 2021, **Swift 5, iOS 14, Xcode 12**

Learn how to test your SwiftUI iOS views in a simple and fast way using snapshot testing. By Vijay Subrahmanian.

Leave a rating/review
Download materials
Save for later

Contents

When you've meticulously crafted a wonderful UI for your app, making sure it stays intact despite changes in the codebase is important. Among the many paths you can take to achieve this, **snapshot testing** is an effective option because it's both lightning-fast and easy to create and maintain.

In this tutorial, you'll use snapshot testing to validate your UIs, ensuring that code changes haven't affected the UI you built. You'll be using a simple library app, **Ray's Library**, for creating these tests.

Specifically, you'll learn how to:

- Create a **baseline snapshot** of the UI in your app and use it to validate any changes.
- Update the baseline snapshot when you're redesigning the UI.
- Test the UI for different device types, orientations and traits.

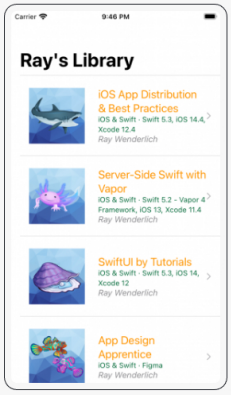
Excited to get started? :]

Getting Started

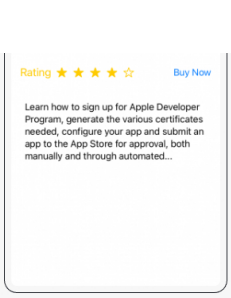
Use the **Download Materials** button at the top or bottom of this tutorial to download the starter project.

The starter project is a simple app called **Ray's Library**, which shows a list of — you guessed it! — books published by raywenderlich.com. The app was built using SwiftUI and incorporates the **SnapshotTesting framework**, using Swift Package Manager to test the UI.

Build and run in Xcode. Start by navigating through the app to familiarize yourself with it. The app has two screens, a landing screen and a book detail screen. The landing screen shows a list of books:



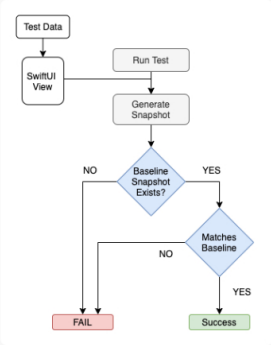
Scroll through the collection of books. Select any book from the list to view its detail screen, which includes a description and ratings:



Next, open **BookRowView.swift** and **BookDetailView.swift** to learn the UI layout. You'll write snapshot tests for these UIs in the coming sections. But first, you'll learn a little more about what snapshot testing is all about.

What Is Snapshot Testing?

Snapshot testing allows you to validate your UIs by comparing a snapshot of the UI at test time with a known valid snapshot of your UI, called a **baseline**. The test runner compares the current snapshot with the baseline snapshot. If there are any differences between the snapshots, the UI must have changed, so the test fails.



Snapshot Testing Strategies

The SnapshotTesting framework supports testing `UIView`s and `UIViewController`s using the image strategy or the recursive description strategy.

The **image strategy** takes actual snapshots of your UI as image files, then compares the images pixel by pixel. If the images are different, then the UI changed; your test will fail.

Similarly, the **recursive description strategy** compares string descriptions of the view hierarchy. If the descriptions have changed, your UI is different. Once again, your test will fail.

Finally, SnapshotTesting offers the **hierarchy strategy**, which generates the view controller hierarchy as plain text. Changes to the view controller hierarchy will cause your test to fail. Of course, this only works for `UIViewController` tests.

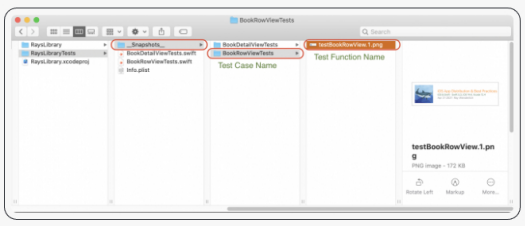
Using the Image Strategy

For this tutorial, you'll use the image strategy, where the framework compares the baseline image with the test image. The comparison works by checking for differences in the pixels of both images. Any difference in the pixels will cause your test to fail.

The image strategy supports many options while testing a `UIViewController`:

- **drawHierarchyInKeyWindow: Bool = false**: Renders the view on the simulator's **Key Window** and uses its appearance and visual effects.
- **on: ViewImageConfig**: Lets you choose a variety of devices as the `ViewImageConfig` option.
- **precision: Float = 1**: Indicates the percentage of pixels that must match for the test to pass. By default, it's **1**, which means 100% of the pixels must match.
- **size: CGSize = nil**: The view size for generating the snapshot of the test view.
- **traits: UITraitCollection = .init()**: Allows you to specify a plethora of traits to test with.

The first time you write and run a new test case, you'll find that the test will fail. That's because you don't have a baseline snapshot yet. During the first run, SnapshotTesting will save the baseline snapshot to your project directory.



Subsequent test runs will take a new snapshot and compare it to the baseline.

Now, you're ready to create your first snapshot test!

Testing the Row View

You'll start by adding a test case to validate `BookRowView`. Open **BookRowViewTests.swift**, and add the following line after the last import statement:

```
import SnapshotTesting
```

This imports the SnapshotTesting framework into your file.

Next, add these lines inside `testBookRowView()`:

```
let bookRowView = BookRowView(book: sampleBook)
let view: UIView = UIHostingController(rootView: bookRowView).view
```

Here you've created an instance of `BookRowView`, which is your SwiftUI view.

A SwiftUI view is a like a plan for a view, not a view itself. So you pass it to `UIHostingController` as its `rootView` and return a `UIView`. Now, you have a view object you can test.

Generating a Baseline Snapshot

Before you can test your view, you need to generate a baseline snapshot. Add the line below to the end of `testBookRowView()`:

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

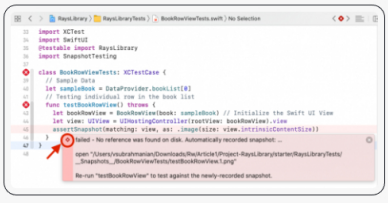
Here, you used `assertSnapshot(matching:as:)` to assert that your view is valid. The SnapshotTesting framework provides `assertSnapshot(matching:as:)`, which is similar to `XCTAssert`. However, instead of simply comparing two values for equality, `assertSnapshot(matching:as:)` compares two snapshots.

In this case, you passed your `view` as the `matching` parameter to tell SnapshotTesting which view you want to test. You've also directed SnapshotTesting to use the `.image` strategy, meaning it will actually save a graphical representation of your UI view at its intrinsic content size.

> **Note**: The image generated with an intrinsic content size might be larger than the iPhone's width and/or height, depending on its contents. Therefore, it's best to specify the appropriate size explicitly before testing a view.

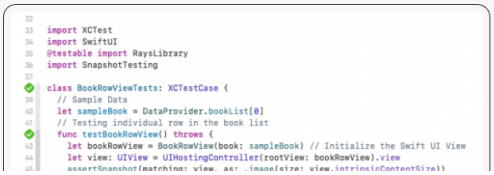
Build and run the test by selecting **Product** ▸ **Test** from the menu bar or by typing **Command-U**. This will run all the tests in the test target.

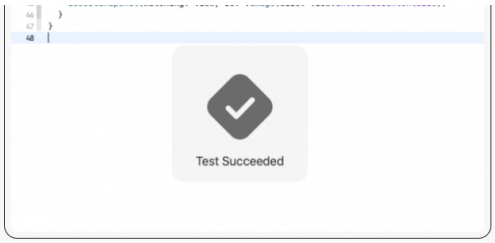
Your test failed. To read the failure message, expand it by clicking the **Cross** icon.



The test failed because there was no reference snapshot to compare to the test result. Remember, the first time you execute your test case, the SnapshotTesting framework will create the baseline for you.

Now that you've created the baseline snapshot, build and run again. Success!





Great! Your first snapshot test

Introducing a UI Change

From now on, when you run your test case, SnapshotTesting will take a new snapshot image of your view and compare it to the baseline snapshot. As long as your view code does not change, your test will pass.

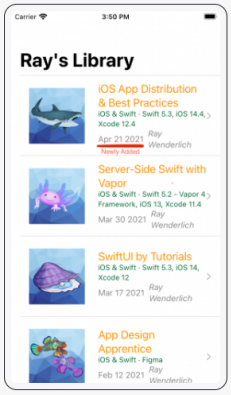
But this is software — things always change!

For example, assume the app's designer wants to add a release date to each book. To see what happens, open **BookRowView.swift**. Find `// Insert release Text` and add this code below the comment:

```
Text(book.release)
  .font(.subheadline)
  .foregroundColor(.secondary)
```

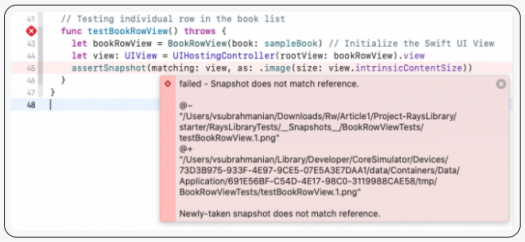
Here, you added a Swift UI `Text` element that shows the release date of the book.

Build and run the project using **Command-R**. You'll see the newly added release date information alongside the publisher name.



Now, see what the test has to say about this update.

Build and run the test using **Command-U**. The test fails with an error:

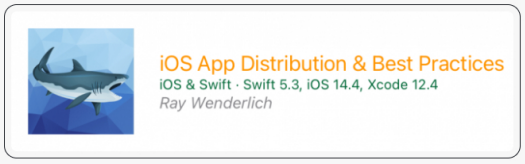


SnapshotTesting reports: **Newly-taken snapshot does not match reference**. This means that the baseline snapshot, without release dates, doesn't match the new snapshot, which includes the release dates.

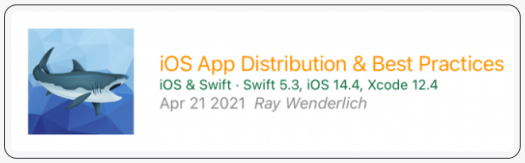
The error message also conveniently includes the paths of the baseline snapshot and the new snapshot. Copy the paths of each and open them in Finder to see the difference.

> **Note**: In Mac's Finder app, you can go to a specific path by pressing **Command-Shift-G** and then pasting in the path.

Here's the baseline snapshot:



And here's the new snapshot:



Of course, you expected this behavior. After all, you just changed the UI yourself! So to get your test case to pass, you'll have to update your baseline snapshot.

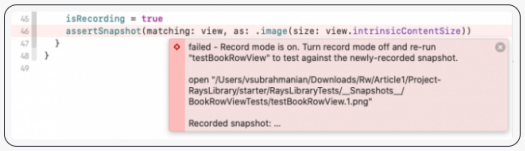
Updating the Baseline Snapshot

Open **BookRowViewTests.swift** and add the following line just above `assertSnapshot(matching:as:)`:

```
isRecording = true
```

When you set `isRecording` to `true`, it tells the SnapshotTesting framework that you want to create a new baseline snapshot.

Build and run the test.



Your test fails, but SnapshotTesting saved a new baseline snapshot, just like the first time you ran your test.

Now that you have a new baseline snapshot, remove the line `isRecording = true` again.

Build and run the test case again. Voila! The test passes.

You now know how to create a baseline snapshot, and you know how to update it when you make a UI change. But remember, if you see a failed test case when you didn't intentionally change your UI, be sure to fix the code and not the test case! :]

Testing the Detail View

Now that you've thoroughly tested your book row view, you'll test your book details screen. As you'll see, the process is similar. However, unlike the previous test, where the system under test was a `UIView`, you'll test a `UIViewController` here.

With a `UIViewController`, you can set your tests to use different iPhone versions, different screen orientations, different device types and even check how your app works in dark mode.

Setting up Your Testing Environment

With so many different testing options and variations, you might think that you'll have to create your detail view object over and over again. Fortunately, SnapshotTesting offers a better way to create the system under test.

Open **BookDetailViewTests.swift**. Add the following code below `// Setup - creating an instance of the BookDetailView`:

```
override func setUpWithError() throws {
  try super.setUpWithError()
  let bookDetailView = BookDetailView(book: sampleBook)
  viewController = UIHostingController(rootView: bookDetailView)
}
```

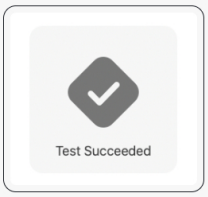
This code creates an instance of `BookDetailView` and adds it to `UIHostingController` as a root view, just as you did in `BookRowViewTests`. However, by putting this code in `setUpWithError()`, SnapshotTesting will automatically run this code before each test case. That way, you don't have to create your system under test in your test case functions.

Now find `// Tear down - Clear any instance variables`, and add the following below:

```
override func tearDownWithError() throws {
  try super.tearDownWithError()
  viewController = nil
}
```

Here, you clear the value of `viewController` after each test run. That way, each test case can start with a brand new `BookDetailView`.

Build and run the tests. Success! The tests pass, though you haven't made any assertions yet.



Testing for Specific iPhone Versions

Find `testBookDetailViewOniPhone()`, then add the following line to the function body:

```
assertSnapshot(
  matching: viewController,
  as: .image(on: .iPhoneX))
```

`image(on:)` accepts an argument of type `ViewImageConfig`, which is set to `.iPhoneX` here. This tells SnapshotTesting to render the `viewController` at an iPhone X screen size. The default value for orientation is portrait.

Build and run the individual test by clicking the **Diamond** icon in the gutter.

The test failed, but you've now recorded and saved the baseline snapshot.

Run the test again and it will succeed.

Note: You can choose a device to test the view against from a comprehensive list that the Snapshot Testing framework provides.

Testing for Device Orientation

Similarly, find `testBookDetailViewOniPhoneLandscape()` and add the following code to its body:

```
assertSnapshot(
  matching: viewController,
  as: .image(on: .iPhoneX(.landscape)))
```

This time, you test your UI on the iPhone X in landscape orientation by passing `.landscape` to the device object.

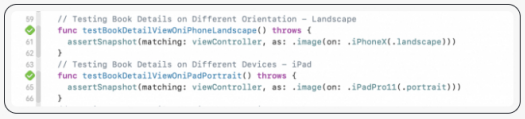
Next, find `testBookDetailViewOniPadPortrait()` and add this code to its body:

```
assertSnapshot(
  matching: viewController,
  as: .image(on: .iPadPro11(.portrait)))
```

This function tests the UI on an iPad Pro 11 in portrait orientation. Here, `ViewImageConfig` is set to `.iPadPro11` with a `.portrait` Orientation. Simple!

Build and run the tests. The test fails because there was no baseline.

Build and run — now, it succeeds.



Testing for Dark Mode

As previously mentioned, you can also test how your app behaves with Dark Mode.

Find `testBookDetailViewOniPhoneDarkMode()` and add the following code to its body:

```
let traitDarkMode = UITraitCollection(userInterfaceStyle: .dark)
assertSnapshot(
  matching: viewController,
  as: .image(on: .iPhoneX, traits: traitDarkMode))
```

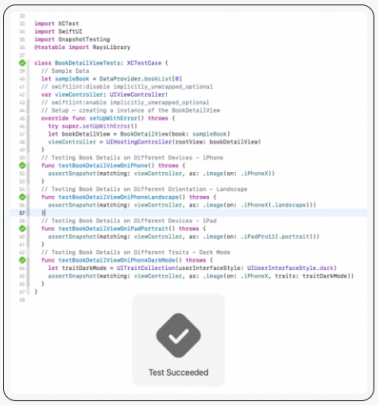
First, you create `UITraitCollection`, which sets the UI style to dark mode.

You then pass that trait collection to `image(on:traits:)`, which tells SnapshotTesting to apply dark mode when taking a snapshot.

> **Note**: You can combine a list of `UITraitCollection`s into a single trait collection, which opens up a way to validate a multitude of combinations!

Now, build and run your test. As you saw earlier, the tests fail when there's no baseline reference of the snapshot.

Again, build and run — all the tests pass!



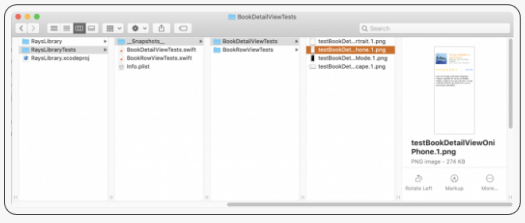
Next time you make a change in the code, you can be sure that the change looks fine on all devices just by running the test cases!

Finding the Baseline Snapshots

SnapshotTesting stores your baseline snapshots right inside your project directory. This allows you to add your baseline snapshots to your project's Git repository, so they'll be accurate across different branches, and so you can share them with your teammates.

To find the baseline snapshots, open Finder and navigate to your project directory. Open **RaysLibraryTests**, then open **_Snapshots_**. Inside, you'll see folders for each of the test files you created above. Inside those folders, you'll find the baseline snapshot images that SnapshotTesting created for each of your test cases.

Open **BookDetailViewTests** and view each of the saved baseline snapshots. They will have the right screen dimension, orientation and traits specified in their respective test cases.



> **Note**: As you add more tests to the project, the snapshots folder can grow in size. If you're using version control, like Git, use GIT LFS to save the snapshots in the repository.

You'll find this immensely useful when a snapshot test fails and you have to debug the reason

why.

Where to Go From Here?

Download the completed version of the project by clicking the **Download Materials** button at the top or bottom of this tutorial.

Continue learning by adding more tests to the project for various traits as well as by exploring the options and testing strategies available in the SnapshotTesting framework.

Here are a few resources to learn more:

- The iOS Unit Testing and UI Testing Tutorial is a great place to start your testing journey.
- This tutorial used SnapshotTesting library by Point-Free and the different testing strategies available in that framework.
- Take a look at this article that explains the SwiftUI View hierarchy: Exploring SwiftUI 3: View Hierarchies and State Updates.
- If you want to look at alternate strategies and frameworks to test SwiftUI, check out Who said we cannot unit test SwiftUI views? by Alexey Naumov.

I hope this tutorial has provided an overview of how to use snapshot testing and the benefits it has to offer. If you have any questions or comments, please join the forum discussion below!

Mark complete

Vijay Subrahmanian

Testing, iOS & Swift Tutorials

Leave a rating/review
Save for later
Comments
Download materials
Share

Contributors

Vijay Subrahmanian
Author

David Sherline
Tech Editor

Sandra Grauschopf
Editor

Luke Freeman
Illustrator

Morten Faarkrog
Final Pass Editor

Richard Critz
Team Lead

Libranner Santos
Topics Master

Over 300 content creators. Join our team.