OSINSKI.DEV

Snapshot testing self-sizing table view cells

September 3, 2019

Recently I was tasked with creating documentation for one of our app's features - screen which can be fed a json with specific schema and then renders it to display rich content. Schema contains several types of sections which view can display: headers, content, single image, image gallery etc. Simplified version looks as follows:

```
{
    "title": "Screen title",
    "sections": [
        {
            "type": "header",
            "text": "Header title"
        },
        {
            "type": "paragraph",
            "text": "Content of paragraph"
        },
        {
            "type": "image",
            "image_url": "http://example.com/image.jpg"
        },
        {
            "type": "paragraph",
            "text": "Content of other paragraph"
        },
        {
            "type": "gallery",
            "image_urls": [
                "http://example.com/gallery_image1.jpg",
                "http://example.com/gallery_image2.jpg",
                "http://example.com/gallery_image3.jpg"
            ]
        }
        ...
    ]
}
```

Each section is represented in the UI by its own `UITableViewCell` subclass. Sections can appear in any kind of order and view will display them. It gives us pretty nice and flexible way of composing content in the app.

Up to this point, json of this screen had been embedded into the app's bundle and we, developers, were responsible for creating and updating it based on content team needs. It was fine for the first phase of the project, but we started to feel the need to be able to update this screen without updating the app (i.e. fetching it from backend). We also wanted to move the responsibility of updating it to content team. To make it as easy as possible for them, the need for thorough documentation arose.

I decided it would be best to document each section type with screenshot of corresponding cell. Easy, right? Just run the simulator and press `Cmd+S`. Sure, but I wanted more. It felt like great opportunity to introduce snapshot tests to the project - they will both assure that screen looks as expected and can be used as screenshots in documentation! Truly killing two birds with one stone (no animal was harmed during writing this post)!

Setting up

Most popular iOS Snapshoting library is iOSSnapshotTestCase (formerly known as FBSnapshotTestCase). It's pretty easy to set up. After adding it to your project and setting up image paths (See project's README) you can start writing your tests:

```
import FBSnapshotTestCase
@testable import MyApp

class MyAppViewSnapshotTests: FBSnapshotTestCase {
    func testSomeViewControllerSnapshot() {
        // recordMode = true // Needed when running particular test for the first time to create re

        let vc = SomeViewController

        FBSnapshotVerifyView(vc.view)
    }
}
```

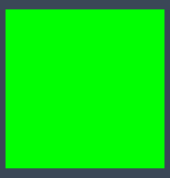
This will correctly layout `SampleViewController` (according to selected simulator's screen size) and take snapshot of it.

Snapshot tests limitations

Unfortunately, only view controllers' views are automatically sized. Trying to test other views will result in error, as snapshot will have size of 0x0. To fix that, we need to set view's frame by hand:

```
func testSomeView() {
    let view = UIView()
    view.backgroundColor = .green
    view.frame.size = CGSize(width: 100, height: 100)

    FBSnapshotVerifyView(view)
}
```

which results with:



This view is plain `UIView` which has no intrinsic size, so its natural size is zero. What about view, which has intrinsic size, like label? It should size itself correctly, shouldn't it?

```
func testLabel() {
    let label = UILabel()
    label.backgroundColor = .white
    label.numberOfLines = 0
    label.text = "ABC\nDEF\nGHI"

    FBSnapshotVerifyView(label)
}
```

Our assumption is wrong, this test crashes!

```
CellSnapshotTesting[15593:482640] *** Assertion failure in +[UIImage fb_imageForLayer:], /Users/seb
<unknown>:0: error: -[CellSnapshotTestingTests.LabelSnapshotTests testLabel] : failed: caught "NSIn
```

There is an easy fix for that, we just need to call `label.sizeToFit()` to set size of our label:

```
func testLabel() {
    let label = UILabel()
    label.backgroundColor = .white
    label.numberOfLines = 0
    label.text = "ABC\nDEF\nGHI"
    label.sizeToFit()

    FBSnapshotVerifyView(label)
}
```

Created snapshot looks as follows:

Great, but our label's text has explicit newlines, what if we wanted to test how it behaves when there are no explicit newlines but width is constrained?

```
func testLabelWithoutNewlines() {
    let label = UILabel()
    label.backgroundColor = .white
    label.numberOfLines = 0
    label.text = "ABC DEF GHI"
    label.translatesAutoresizingMaskIntoConstraints = false
    label.widthAnchor.constraint(equalToConstant: 80).isActive = true

    FBSnapshotVerifyView(label)
}
```

This test crashes with the same exception as our first label test approach. Label has implicit `_UITemporaryLayoutWidth` constraint which breaks constraint we added.

After some trial and error I finally found out solution for this - our tested views need a superview!

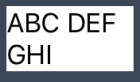
I decided to create generic container view which will help testing views for different width conditions (e.g. different screen widths):

```
class SnapshotContainer<View: UIView>: UIView {
    let view: View

    init(_ view: View, width: CGFloat) {
        self.view = view

        super.init(frame: .zero)

        translatesAutoresizingMaskIntoConstraints = false
        view.translatesAutoresizingMaskIntoConstraints = false

        addSubview(view)

        NSLayoutConstraint.activate([
            view.topAnchor.constraint(equalTo: topAnchor),
            view.bottomAnchor.constraint(equalTo: bottomAnchor),
            view.leadingAnchor.constraint(equalTo: leadingAnchor),
            view.trailingAnchor.constraint(equalTo: trailingAnchor),
            view.widthAnchor.constraint(equalToConstant: width)
        ])
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

Let's now use it in test:

```
func testLabelInContainer() {
    let label = UILabel()
    label.backgroundColor = .white
    label.numberOfLines = 0
    label.text = "ABC DEF GHI"

    let container = SnapshotContainer<UILabel>(label, width: 80)

    FBSnapshotVerifyView(container)
}
```

And now our snapshot looks as expected:



Testing table view cells

Great! We finally found out the way to test self-sizing views! We can now get to the topic of this post and start testing table view cells. Let's start by defining our test subject - message cell with avatar view and multiline label in it. Cell has all constraints set up correctly so it can be sized automatically.

```
class MessageTableViewCell: UITableViewCell {
    let avatar = UIView()
    let label = UILabel()

    override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
```

```
    override init(style: UITableViewCell.CellStyle, reuseIdentifier: String?) {
        super.init(style: style, reuseIdentifier: reuseIdentifier)

        setup()
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    private func setup() {
        label.numberOfLines = 0

        contentView.addSubview(avatar)
        contentView.addSubview(label)

        avatar.translatesAutoresizingMaskIntoConstraints = false
        label.translatesAutoresizingMaskIntoConstraints = false

        NSLayoutConstraint.activate([
            avatar.topAnchor.constraint(equalTo: contentView.layoutMarginsGuide.topAnchor),
            avatar.bottomAnchor.constraint(lessThanOrEqualTo: contentView.layoutMarginsGuide.bottom
            avatar.leadingAnchor.constraint(equalTo: contentView.layoutMarginsGuide.leadingAnchor),
            avatar.widthAnchor.constraint(equalTo: contentView.widthAnchor, multiplier: 0.15),
            avatar.heightAnchor.constraint(greaterThanOrEqualTo: avatar.widthAnchor),
            label.topAnchor.constraint(equalTo: contentView.layoutMarginsGuide.topAnchor),
            label.bottomAnchor.constraint(lessThanOrEqualTo: contentView.layoutMarginsGuide.bottomA
            label.leadingAnchor.constraint(equalTo: avatar.trailingAnchor, constant: 20.0),
            label.trailingAnchor.constraint(equalTo: contentView.layoutMarginsGuide.trailingAnchor)
        ])
    }
}
```

As we did before, we'll put the cell in `SnapshotContainer` and set container width to 375 (width of iPhone 6/7/8/X/Xs screen). That should do the trick, shouldn't it? Let's see.

```
func testMessageTableViewCell() {
    let cell = MessageTableViewCell(style: .default, reuseIdentifier: nil)
    cell.label.text = "Message"
    cell.avatar.backgroundColor = .green

    let container = SnapshotContainer(cell, width: 375)

    FBSnapshotVerifyView(container)
}
```

Surprisingly, this test crashes with the same error as our first try of testing label:

```
CellSnapshotTesting[26726:1070571] *** Assertion failure in +[UIImage fb_imageForLayer:], /Users/se
<unknown>:0: error: -[CellSnapshotTestingTests.MessageTableViewCellSnapshotTests testCellNotWorking
```

Our cell doesn't resize correctly due to clashing constraints, which you can see at WTFAutoLayout.

I tried many approaches to correctly layout cells for snapshot testing:

- adding more constraints
- trying to remove implicit constraints
- adding `contentView` to container view

After few hours I decided to start from beginning and asked myself a question - **what's the natural environment for table view cell?** `UITableView` of course! It knows how to handle sizing cells, it understands its hidden behaviors.

So now my goal was to create specialized container which will use `UITableView` to layout cell and adjust to its size. After some tinkering I came up with this:

```
final class TableViewCellSnapshotContainer<Cell: UITableViewCell>: UIView, UITableViewDataSource, U
    typealias CellConfigurator = (_ cell: Cell) -> ()
    typealias HeightResolver = ((_ width: CGFloat) -> (CGFloat))

    private let tableView = SnapshotTableView()
    private let configureCell: (Cell) -> ()
    private let heightForWidth: ((CGFloat) -> CGFloat)?

    /// Initializes container view for cell testing.
    ///
    /// - Parameters:
    ///   - width: Width of cell
    ///   - configureCell: closure which is passed to `tableView:cellForRowAt:` method to configure
    ///   - cell: Instance of `Cell` dequeued in `tableView:cellForRowAt:`
    ///   - heightForWidth: closure which is passed to `tableView:heightForRowAt:` method to determ
    init(width: CGFloat, configureCell: @escaping CellConfigurator, heightForWidth: HeightResolver?
        self.configureCell = configureCell
        self.heightForWidth = heightForWidth

        super.init(frame: .zero)

        // 1
        tableView.separatorStyle = .none
        tableView.contentInset = .zero
        tableView.tableFooterView = UIView()
        tableView.dataSource = self
        tableView.delegate = self
        tableView.register(Cell.self, forCellReuseIdentifier: "Cell")

        translatesAutoresizingMaskIntoConstraints = false
        addSubview(tableView)
        tableView.translatesAutoresizingMaskIntoConstraints = false

        NSLayoutConstraint.activate([
            tableView.topAnchor.constraint(equalTo: topAnchor), // 2
            tableView.bottomAnchor.constraint(equalTo: bottomAnchor),
            tableView.leadingAnchor.constraint(equalTo: leadingAnchor),
            tableView.trailingAnchor.constraint(equalTo: trailingAnchor),
            widthAnchor.constraint(equalToConstant: width),
            heightAnchor.constraint(greaterThanOrEqualToConstant: 1.0) // 3
        ])
    }

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
        let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath) as! Cell

        configureCell(cell)
```

```
        return cell
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return 1
    }

    func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
        return heightForWidth?(frame.width) ?? UITableView.automaticDimension // 4
    }
}

/// `UITableView` subclass for snapshot testing. Automatically resizes to its content size.
final class SnapshotTableView: UITableView {
    override var contentSize: CGSize {
        didSet {
            // 5
            invalidateIntrinsicContentSize()
        }
    }

    override var intrinsicContentSize: CGSize {
        // 6
        layoutIfNeeded()

        // 7
        return CGSize(width: UIView.noIntrinsicMetric, height: contentSize.height)
    }
}
```

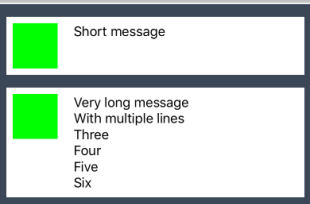
Let's go through the code step by step:

1. We need to configure `tableView` to not show anything else besides our cell content. To do that, we disable separators and table footer view.
2. Container needs to encompass whole `tableView`, so we pin it to all sides of `tableView`.
3. Container has to have initial non-zero height, otherwise it won't resize correctly after `tableView` resizes to cell size.
4. Our container allows to pass closure which calculates height of cell based on its width (for cells which do not use automatic sizing). If such closure is not passed, then we fallback to `UITableView.automaticDimension` height which makes `tableView` autolayout the cell.
5. Each time `contentSize` changes, we trigger intrinsic content size invalidation.
6. Each time `intrinsicContentSize` is accessed, we trigger layout. Otherwise `tableView` won't resize correctly.
7. We use `contentSize.height` as intrinsic height of `tableView`.

This container is much more complicated than `SnapshotContainer`, but so is the nature of `UITableViewCell`'s and `UITableView`'s 😅.

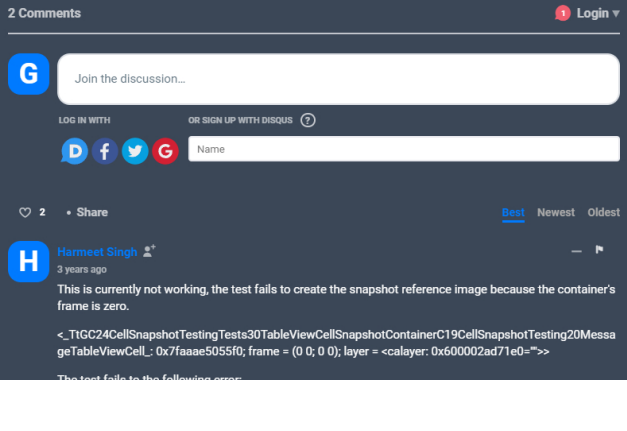
So, does it work? Yes it does! Here we have two tests checking how cell looks for short and long multiline text for iPhone 6/7/8/X/Xs width.

```
func testCellWithShortText() {
    let container = TableViewCellSnapshotContainer<MessageTableViewCell>(width: 375, configureCell:
        cell.label.text = "Short message"
        cell.avatar.backgroundColor = .green
    })

    FBSnapshotVerifyView(container)
}

func testCellWithMultilineText() {
    let container = TableViewCellSnapshotContainer<MessageTableViewCell>(width: 375, configureCell:
        cell.label.text = "Very long message\nWith multiple lines\nThree\nFour\nFive\nSix"
        cell.avatar.backgroundColor = .green
    })

    FBSnapshotVerifyView(container)
}
```



Conclusion

Voila! We managed to extend functionality of `iOSSnapshotTestCase` to test single table view cells. It needed some work, as both `iOSSnapshotTestCase` and `UITableViewCell`'s have their quirks, but I think it's worth it. In my project, it's now not only really easy to quickly test cells but also to update screenshots in documentation when designs change - just run the tests and copy new snapshots 😍. If you want to play with above code on your own, I created sample project which you can find here. See you next time!



The test fails to the following error:

failed - Snapshot comparison failed: Optional(Error Domain=FBSnapshotTestControllerErrorDomain Code=1 "Unable to load reference image." UserInfo={NSLocalizedFailureReason=Reference image not found. You need to run the test in record mode, NSLocalizedDescription=Unable to load reference image., FBReferenceImageFilePathKey=/TableViewCellSnapshotTesting-master/CellSnapshotTestingTests/ReferenceImages_64/CellSnapshotTestingTests.MessageTableViewCellSnapshotTests/testCellWithShortText@3x.png})

Would be nice if you could get this working! Appreciate the effort and post :)

0 0 Reply • Share ›

Sebastian Osiński Mod Harmeet Singh

3 years ago

Hi! Thanks for raising this issue. I just tested the sample project and all tests pass. The only thing is that tests must be run on iPhone 8 simulator or other 2x scale device. When tests are run on 3x device (e.g. iPhone 11 Pro) then you get error you provided. When you read into it, you can see that it tries to find snapshots with *@3x.png name, but sample project provides only *@2x.png snapshots.

I assume that container's frame is zero because FBSnapshotTestCase doesn't even start layouting it if there is no snapshot to compare with.

0 0 Reply • Share ›