Extracted from:

Xcode Treasures

Master the Tools to Design, Build, and Distribute Great Apps

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Xcode Treasures

Master the Tools to Design, Build, and Distribute Great Apps

Chris Adamson
edited by Tammy Coron

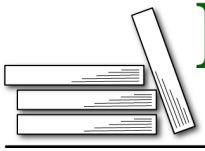
Xcode Treasures

Master the Tools to Design, Build, and Distribute Great Apps

Chris Adamson

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Development Editor: Tammy Coron
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-586-3
Book version: P1.0—October 2018

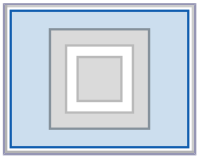
Embedded Scenes

One thing storyboards tend to reinforce is that a single view controller is responsible for one screen-full of content. After all, there is a one-to-one correspondence of scenes to view controllers, and the scenes themselves are shaped like iPhone or iPad screens, depending on what device you've set the "View As" control at the bottom of Interface Builder to show.

This leads to the problem of the "Massive View Controller". If there's a lot going on in one scene, the natural place for all the code to deal with that is in the view controller. And when that view controller is responsible for handling UI events, populating table or collection views, dealing with rotation or remote-control media events, etc., it quickly leads to the view controller's size and complexity getting out of hand. Nobody sets out to write a 2,000-line `UIViewController`, but some mornings you wake up and *there it is*.

This section is inspired by Dave DeLong's talk "A Better MVC" at Swift by Northwest 2017,[1] and also his follow-up blog post.

One strategy to avoid the massive view controller is to break the 1 view controller == 1 screen mindset. Storyboards give us a technique to break that habit: *container views*. With this approach, we use multiple view controllers in a scene, each one smaller and more focused than would be possible otherwise.



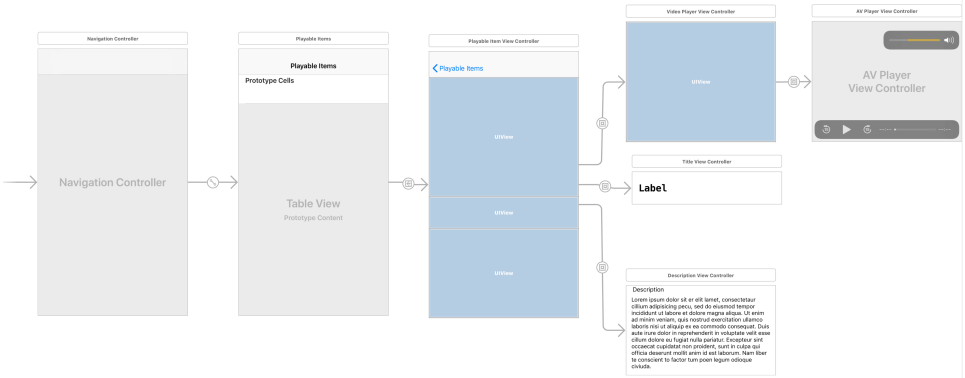
If you pick up the *container view* icon from the library and drop it in a scene, two interesting things will happen. It will place a view in the scene, which works like any other plain `UIView`, in that it can be laid out in the scene with Auto Layout constraints. But it also adds another whole scene to the storyboard, which is connected to the container view with a segue.

The idea here is that the second scene is where all the content comes from; this is where you can add subviews like labels, buttons, sliders and what have you. More importantly, since this is a completely separate scene, it has its own view controller. And this is the key to breaking up a massive view controller: if parts of that parent screen can take care of themselves, you can split out their functionality into completely separate view controllers.

There's an example of this in the download code. Take a look at the following storyboard and we'll walk through how it works. The app shows a table of `PlayableItem` instances, which is just a struct with a streaming video URL, a

1. https://davedelong.com/blog/2017/11/06/a-better-mvc-part-1-the-problems/

title string, and a description string. When the user selects one of the playable items, the app segues a detail screen, the PlayableItemViewController.



The PlayableItemViewController scene in the middle is where things get interesting. This has just three subviews, but all of them are container views. One goes to a VideoPlayerViewController scene, one to a TitleViewController, and one to a DescriptionViewController. These scenes are trivial; for example, the title screen manages a single label, and the description screen just has a text view, each of which is updated by setting a playableItem property. The video scene is a little weirder, so we'll get back to that one.

Because these scenes are so simple, they've almost no code. They're easy to expose to unit testing (which is covered in Chapter 8, *Automated Testing*, on page ?), and if there is a problem with, say, the description, it's easier to go straight to that source file, rather than searching through a massive view controller's source.

There is, however, one trick to using container views. Notice that the connections to the host scene are actually segues. These are *embed segues*, and they have an important use. Since the parent scene doesn't have a direct connection to the child scenes, it can't access them or their properties. This is a problem, because you need a way for the parent to pass the selected PlayableItem to the child scenes, which will use it to update themselves.

The solution is to use the segue. When the child scene loads and is embedded into the parent, the parent gets a one-time call to prepare(for:sender:), with the embed segue as its parameter. The segue's destination is the view controller being embedded. So the trick here is to save a reference to the view controller that's being embedded.

In the PlayableItemViewController, you set up properties for the three child view controllers:

storyboards-behavior/EmbeddedVCDemo/EmbeddedVCDemo/PlayerSceneVCs/PlayableItemViewController.swift

```
private var videoPlayerVC: VideoPlayerViewController?
private var titleVC: TitleViewController?
private var descriptionVC: DescriptionViewController?
```

Then, you implement prepare(for:sender:) to catch each of the embed segues and save off its destination to the correct property. In the example, there are identifier strings on each of the segues to make this step work nicely with a switch statement (and it's a good habit to always name your segues anyway):

storyboards-behavior/EmbeddedVCDemo/EmbeddedVCDemo/PlayerSceneVCs/PlayableItemViewController.swift

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }

    switch identifier {
    case "embedTitle":
        if let titleVC =
            segue.destination as? TitleViewController {
            self.titleVC = titleVC
            titleVC.playableItem = playableItem
        }
    case "embedDescription":
        if let descriptionVC =
            segue.destination as? DescriptionViewController {
            self.descriptionVC = descriptionVC
            descriptionVC.playableItem = playableItem
        }
    case "embedVideoPlayer":
        if let videoPlayerVC =
            segue.destination as? VideoPlayerViewController {
            self.videoPlayerVC = videoPlayerVC
            videoPlayerVC.playableItem = playableItem
        }
    default:
        break
    }
}
```

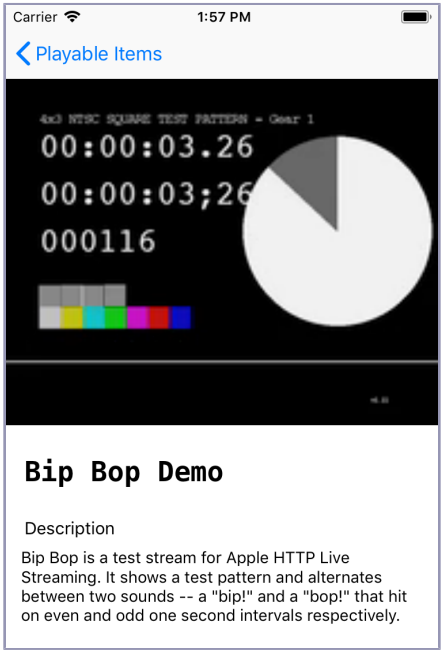
Now that you've got references to the child view controllers, any time the parent's playableItem is set, it can send that struct to those child view controllers:

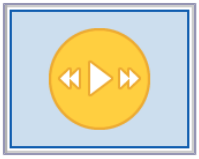
storyboards-behavior/EmbeddedVCDemo/EmbeddedVCDemo/PlayerSceneVCs/PlayableItemViewController.swift

```
var playableItem: PlayableItem? {
    didSet {
        videoPlayerVC?.playableItem = playableItem
        titleVC?.playableItem = playableItem
        descriptionVC?.playableItem = playableItem
    }
```

```
}
```

And that's the key! With that, the child view controllers can update themselves from whichever fields of the playableItem are relevant to them. The running app is shown in the following figure:





The idea of container views and embedded view controllers also helps to explain how the "AVKit Player View Controller" icon works. As a view controller, it can't be dropped directly into a scene, and dropping it on a storyboard makes it its own scene. For beginners, this is confusing: can the player only be its own full-screen scene? Nope, the way you want to use it is as a child view controller. This is shown at the top right of the storyboard figure shown earlier. The VideoPlayerViewController child scene has a child view of its own. To do this, the example project deletes the default scene that came with the container view icon, and replaces it with an embed segue to the AVKit Player scene.