# iOS 9 SDK Development

## Creating iPhone and iPad Apps with Swift

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

# iOS 9 SDK Development

## Creating iPhone and iPad Apps with Swift

Chris Adamson
with Janie Clayton

*edited by Rebecca Gulick*

# iOS 9 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson with Janie Clayton

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Rebecca Gulick (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)

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

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

Printed in the United States of America.
ISBN-13: 978-1-68050-132-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—August 2016

CHAPTER 5

# Connecting the UI to Code

You've learned how to build user interfaces with storyboards and Interface Builder, and, before that, you used playgrounds to learn the ins and outs of the Swift programming language. But from where you stand right now, these two things have nothing to do with each other: you can't write code in a storyboard, and you can't drag and customize buttons and labels in a playground.

Obviously, there has to be some way to bring your two skill sets together, so you can bring a user interface to life and have your code do more than just produce log messages.

This chapter will let you close the loop by bringing these two worlds together: you'll connect user interface to code, so buttons can react to taps and your code can update what's on the screen.

It's all about connections.

## Making Connections

So, how do we get the Send Tweet button tap to do something? After all, we've been creating the user interface in the Main.storyboard file, but it doesn't look like there's any place in this editor to start writing code.

In iOS, we use Interface Builder *connections* to tie the user interface to our code. Using Xcode, we can create two kinds of connections:

- An *outlet* connects a variable or property in code to an object in a storyboard. This lets us read and write the object's properties, like reading the value of a slider or setting the initial contents of a text field.
- An *action* connects an event generated by a storyboard object to a method in our code. This lets us respond to a button being tapped or a slider's value changing.

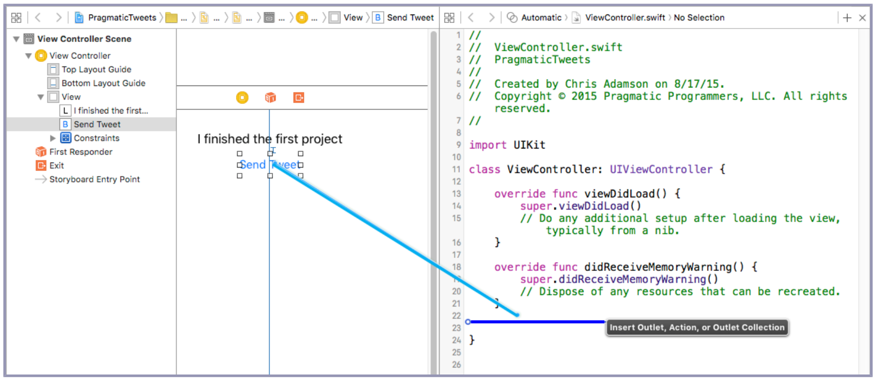
What we need here is an action connecting the button tap in the UI to a method in our code, which we'll write in a little bit. To create either kind of connection, we need to declare an IBOutlet or IBAction in our code, and then create the connection with Interface Builder. Fortunately, IB makes this pretty easy by giving us a way to combine the steps.

With the storyboard showing in the Editor area, go to the toolbar and click the Assistant Editor button (it looks like two linked circles). This brings up a side-by-side view with the storyboard on the left and a source file on the right. If there's not enough horizontal room on the screen to see things clearly, use the toolbar to hide the Utility area.

The pane on the right has a jump bar at the top to show which file is in the pane. After a pair of forward/back buttons, there's a button that determines how the file for this pane is selected: Manual, Automatic, Top Level Objects, and so forth. Set this to Automatic and the contents of the file ViewController.swift should appear in the right pane. We'll have more to say about why ViewController.swift is the file we need in the next few chapters, but for now, let's take the name at face value: this is the class that controls the view.

Xcode's template prepopulates ViewController.swift with trivial implementations of two methods: viewDidLoad() and didReceiveMemoryWarning(). We'll be adding a new method to this class.

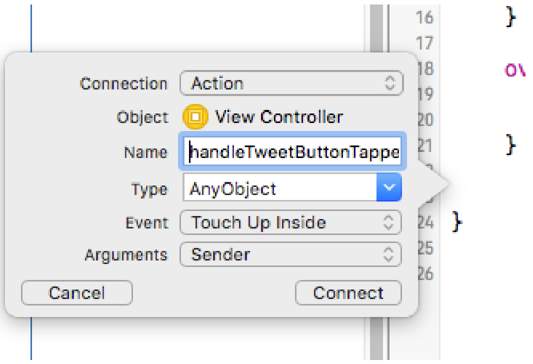
Creating the action is pretty easy. Control-click on the button in Interface Builder, and Control-drag a line over into the source code, anywhere between the set of curly braces that begin with class ViewController : UIViewController and end at the bottom of the file, and not within the curly braces of an existing method. Don't worry; a blue drop indicator and the tooltip "Insert Outlet, Action, or Outlet Collection" will appear only when we mouse over a valid drop zone. A good place to target is the line right before the final curly brace:



```
//
//  ViewController.swift
//  PragmaticTweets
//
//  Created by Chris Adamson on 8/17/15.
//  Copyright © 2015 Pragmatic Programmers, LLC. All rights
    reserved.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view,
            typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }


}
```

When we release the mouse in the source file, a popover asks us for the details needed to finish the method declaration. On the first line, change the Connection from Outlet to Action. This is important—for a button tap, we want a connection that goes from UI to code, and that's what an action is.

We need to give the method a name, so type handleTweetButtonTapped in the Name field. Next, the Type field determines what kind of object will be passed to the method as an argument identifying the source of the action. The default, AnyObject, represents any kind of object and works well enough, but we can save ourselves some typing later by switching it to UIButton so we know that the object calling us is a button. For the Event and Arguments fields we can take the default values. Click the Connect button to create the connection.



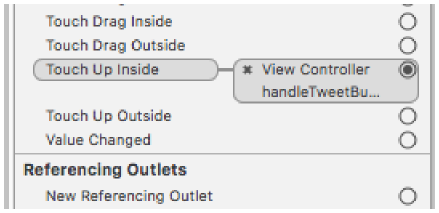
Connection: Action
Object: View Controller
Name: handleTweetButtonTappe
Type: AnyObject
Event: Touch Up Inside
Arguments: Sender

We're done with the Assistant Editor. Click the Standard Editor button in the toolbar to return to one-pane mode. Select ViewController.swift in the Navigator area amd you'll see that Xcode has stubbed out a method signature for us:

connecting/PragmaticTweets-5-1/PragmaticTweets/ViewController.swift

```
@IBAction func handleTweetButtonTapped(sender: UIButton) {
}
```

Xcode has also made a change to the storyboard, but it's not as easy to see. Switch to Main.storyboard and bring the Utility area back, if it's hidden. Click on the button to select it. Then, in the Utility toolbar, click the little circle with the arrow (or press ⌥⌘6) to bring up the *Connections Inspector*. This pane shows all the connections for an object in Interface Builder: all the outlets from code to the object, and all actions sent by the object into the code. In this case, one connection appears in the Sent Events section, from Touch Up Inside to View Controller handleTweetButtonTapped. This connection, shown in the figure that follows, is editable here. If we wanted to disconnect it, we could click the little "x" button, and then reconnect to a different IBAction method by dragging from the circle on the right to the View Controller icon in the scene.



Honestly, we don't break and remake connections very often, but if a connection ever gets inadvertently broken (for example, by renaming the method in the source file), looking in the Connections Inspector is a good approach for diagnosing and fixing the problem.

## Coding the Action

Now that we've added a button to our view and wired it up, we can run the app again. The app now has the Send Tweet button, and we can even tap it, but it doesn't do anything. In fact, we don't even know if we've made our connections correctly. One thing we can do as a sanity check is to log a message to make sure our code is really running. Once that's verified, we can move on to implementing our tweet functionality.

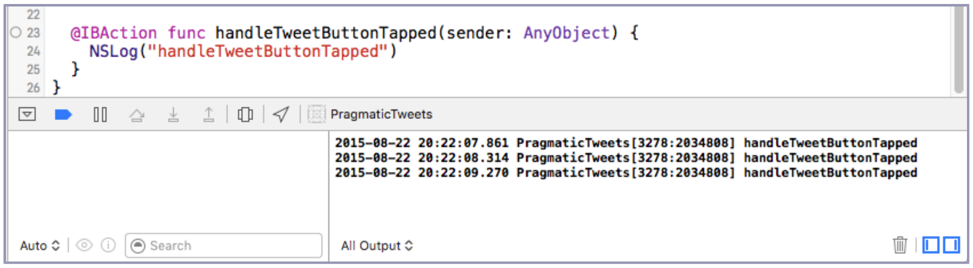
### Logging

Back in Chapter 2, we learned about the NSLog() function for logging timestamped messages to the Xcode console. We can use that in our action to just log a message every time the button is tapped, and thereby verify that the connections are working. Select ViewController.swift in the File Navigator (⌘1) to edit its source code and rewrite handleTweetButtonTapped() like this:

connecting/PragmaticTweets-5-1/PragmaticTweets/ViewController.swift

```
@IBAction func handleTweetButtonTapped(sender: UIButton) {
  NSLog("handleTweetButtonTapped")
}
```

Run the app again, and tap the button. Back in Xcode, the Debug area automatically appears at the bottom of the window once a log or error message is generated, as seen in the following figure. Every time the button is tapped, another line is written to the log and shown in the Debug area. If the Debug area slides in but looks empty, check the two rightmost buttons at the bottom of the Debug area, next to the trashcan icon; the left one enables a variables view (populated only when the app is stopped on a breakpoint), and the right (which we want to be visible) is the console view where log messages appear. Another way to force the console view to appear is to press ⇧⌘C.



```
@IBAction func handleTweetButtonTapped(sender: AnyObject) {
  NSLog("handleTweetButtonTapped")
}
}
```

```
2015-08-22 20:22:07.861 PragmaticTweets[3278:2034808] handleTweetButtonTapped
2015-08-22 20:22:08.314 PragmaticTweets[3278:2034808] handleTweetButtonTapped
2015-08-22 20:22:09.270 PragmaticTweets[3278:2034808] handleTweetButtonTapped
```

So now we have a button that is connected to our code, enough to log a message that indicates the button tap is being handled. The next step is to add some tweeting!

## Showing a Tweet Composer

To send a tweet, we need something in the iOS SDK to at least let us get out to the network. As it turns out, iOS is far more generous than that. Bring up the documentation viewer with the menu item Window > Documentation and API Reference (⇧⌘0). In the search field, type social framework. Locate the result for Social Framework Reference and choose that.

The Social framework lets apps connect to social networks like Twitter and Facebook easily. There are just three classes listed, one of which is SLComposeViewController. Click that, and read its documentation:

> The SLComposeViewController class presents a view to the user to compose a post for supported social networking services.

Hey, that sounds perfect! When the user taps Send Tweet, we'll just show the SLComposeViewController, and let it do all the work of composing and sending a tweet.

In ViewController.swift, rewrite the handleTweetButtonTapped() method as follows:

connecting/PragmaticTweets-5-2/PragmaticTweets/ViewController.swift

```
@IBAction func handleTweetButtonTapped(sender: UIButton) {
  if SLComposeViewController.isAvailableForServiceType(SLServiceTypeTwitter){
    let tweetVC = SLComposeViewController(forServiceType:
      SLServiceTypeTwitter)
    tweetVC.setInitialText(
    "I just finished the first project in iOS 9 SDK Development. #pragsios9")
    self.presentViewController(tweetVC, animated: true, completion: nil)
  } else {
    NSLog("Can't send tweet")
  }
}
```

## Getting in Trouble on Purpose

You will probably see some little error icons appear in the left gutter while typing this code. Sometimes these go away, as Xcode figures out that an incomplete line that wouldn't be valid code is in fact legitimate once it's completed. In this case, however, we're going to get in trouble on purpose, as will be explained and resolved shortly.

To start with, on line 2 we ask the SLComposeViewController class if it's even possible to send tweets: it might not be if a given social network isn't set up to post.

If we can send tweets, then we initialize a new SLComposeViewController on line 3, and we assign it to the variable tweetVC.

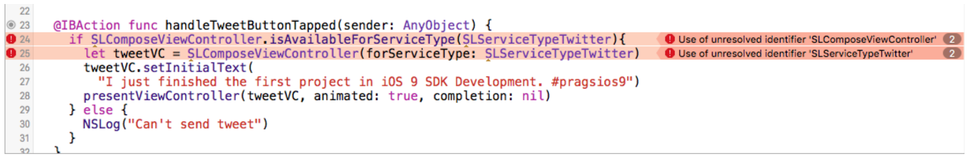
On lines 5–6, we set the initial text of the tweet to "I just finished the first project in iOS 9 SDK Development. #pragsios9" by calling the setInitialText() method on tweetVC.

This is all we need to do to prepare the tweet, so on line 7, we show the tweet composer by telling self (our own ViewController) to presentViewController() with the newly created and configured tweetVC, setting the animated parameter to true, which makes the tweet view "fly in." The third parameter, completion, specifies code to execute once the view comes up; we don't need that, so we send nil.

Finally, if isAvailableForServiceType() returned false, the else block on lines 8–10 logs a debugging message that we can't send tweets. As our skills improve, we'll want to actually show the user a message in failure cases like this.

And that's it. We did all the work in IB to create the button and have it call this method when tapped, so we should be able to just build and tweet at this point, right? Let's try running the app. Click the Run button and see what happens.

Disaster—the project doesn't build anymore! Instead, we get a bunch of error messages in red displayed alongside our code, as seen in the following figure. Worse, depending on the width of the window, the errors are likely truncated. What are we supposed to do?



```
@IBAction func handleTweetButtonTapped(sender: AnyObject) {
  if SLComposeViewController.isAvailableForServiceType(SLServiceTypeTwitter){
    let tweetVC = SLComposeViewController(forServiceType: SLServiceTypeTwitter)
    tweetVC.setInitialText(
      "I just finished the first project in iOS 9 SDK Development. #pragsios9")
    presentViewController(tweetVC, animated: true, completion: nil)
  } else {
    NSLog("Can't send tweet")
  }
}
```

```
Use of unresolved identifier 'SLComposeViewController'  2
Use of unresolved identifier 'SLServiceTypeTwitter'  2
```