The Pragmatic Programmers

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
Janie Clayton-Hasz

edited by Rebecca Gulick

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/adios2/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

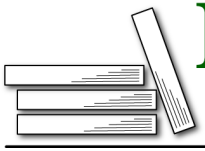
Andy & Dave

iOS 8 SDK Development

Creating iPhone and iPad Apps with Swift

Chris Adamson
Janie Clayton-Hasz

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

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

Printed in the United States of America.
ISBN-13: 978-1-941222-64-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—September 17, 2014

Contents

Introduction

Sometimes it starts with an idea: *"I'm going to write an app to organize my music,"*, *"I'm going to write a kick-ass game," "I'm going to start a website, and I'd better have a native app too."* Other times, it's more of an inkling: *"I bet I could figure out iOS apps," "I wonder how different that is," "If I have to write another freaking login webpage, I'm going to scream."* And sometimes, like Everest, it's worth conquering just because it's there

Whatever your reason for wanting to learn iOS development, it's a good one. The platform is well established as the leader of the smartphone and tablet revolution, and its thoughtful and comprehensive tools and frameworks empower us to make great applications for iPhone, iPad, and iPod touch users.

It's also an exciting time to get into iOS development, or even to come back if you dipped a toe in the water a while back but didn't take the plunge. With Apple's announcements at WWDC 2014, development for the platform just got a *lot* more interesting, thanks to a much greater level of interaction between apps, between iOS and other devices, and even between developers thanks to new testing and collaboration tools. To top it off, Apple introduced an entirely new programming language, Swift, which it intends as a safer, simpler, and faster alternative to Objective-C, which has served iOS and Mac OS X well throughout their entire lifetimes.

About This Edition

This is our third time around with an introductory iOS book for Pragmatic Programmers. *iPhone SDK Development* came out in 2009 and covered iPhone OS 3, then was replaced by a 100% rewrite, *iOS SDK Development* in 2012, covering iOS 6. The book you're reading started out as an update to that, but with all the changes in iOS 7 and 8 (including a change of programming language), it is overwhelmingly new material. If you do catch a bogus copy-and-paste from old material — a reference to a `.h` or `.m` file would be a dead giveaway — we hope you'll let us know on the errata page.

In preparing this edition, we looked for a balance of topics appropriate to our intended audience: developers already comfortable with object-oriented programming, who are new to the iOS platform. We always look for a mix of the new and the fundamental, the shiny and the solid. We're also interested in spending time on things we've learned from our own experience, both good and bad. You'll see we spend a lot of time on "soft" topics like refactoring, debugging, project organization, and testing, and less time on just touring our favorite frameworks (sorry, AV Foundation). We think it's important to think of iOS development not just as a grab-bag of APIs, but a process for turning time and enthusiasm into apps that work well, that can be maintained, and that delight and enlighten.

New hotness sometimes means clearing out the old and busted. In the previous edition, we bet pretty heavily on document-based apps and the ability to store them in iCloud. Two years later, we're not seeing much interest in document-driven apps on the App Store, and as for iCloud, as a certain Dark Lord of the Sith said, "you have failed me for the last time." In their place, we hope you'll enjoy some of the neat new features of iOS 8, including the ability to expose your app's functionality to other apps with extensions, and a variety of tools for making apps look and work well on both iPhone and iPad. On this latter point, our previous edition's sample projects were iPhone-only, but this time, we are all-in on Universal apps that support the iPhone and iPad form-factors, as well as the extra screen space afforded by the iPhone 6 models.

You may also notice a major change on the cover from the previous edition. The aforementioned books were written by Chris Adamson and Bill Dudney, the second coming after Bill did a two-year stint at Apple as a developer evangelist. In 2013, Bill returned to Apple to work on UIKit, so he's unable to contribute to this title, because of corporate policy and because he's really busy bringing us the very software this book is all about. So for this edition, please welcome new co-author Janie Clayton-Hasz. Janie is an iOS developer and self-described "human vertex shader" from Madison, Wisconsin, who's previously served as a technical reviewer for other Pragmatic Programmer iOS titles, and is a speaker at iOS conferences like CocoaConf and 360iDev. She also has uncanny nerd compatibility with Chris, the first person at CocoaConf Chicago to correctly identify the etymology of her youngest pug's name.[1] The first in-person editorial meeting for this book took place at Anime Central in Chicago, probably marking the first time a Prags book has kicked

1. Janie's dog is named "Delia Derbyshire", after the unjustly uncredited co-composer of the *Doctor Who* theme music.

off in cosplay… although considering *Build iOS Games with Sprite Kit* authors Jonathan Penn and Josh Smith, we can't be 100% sure of that.

So Here's the Plan

Our goal for this book is to get readers fully grounded in the essentials of iOS development, comfortable with the tools, the techniques, and the conventions they'll use to make iOS apps whether by themselves or as part of a team. We don't try to cover everything in the iOS SDK in part because doing so made for a 600-page book in 2009, and would be well into the thousands today. Instead, this book should serve as a prerequisite to all the other topic-specific iOS books published by Pragmatic Programmers, as well as those from other publishers.

In this edition, we have chosen to develop a single sample app through the course of the entire book. The advantages of this approach include less time spent starting projects and wiring up user interfaces, and a greater sense of how to manage an app as it evolves and adds features. This gives us a more reality-based approach to app development than we could get with a series of trivial projects, and more naturally leads to the payoff in the last chapter of submitting an app to the App Store and managing its ongoing development lifecycle.

Speaking of payoff, here's the course we're going to take through the book:

- Chapter 12, *Working with Photos*, on page 217,
- Chapter 13, *Launching, Backgrounding, and Exensions*, on page 219,
- Chapter 14, *Debugging Apps*, on page 221,
- Chapter 15, *Publishing to the App Store*, on page 223.

Expectations and Technical Requirements

The technical requirements for iOS development, are pretty simple: the latest version of Xcode, and a Mac OS X computer that can run it. As of our publication in late 2014, that means Xcode 6, and a Mac running either OS X 10.9 (“Mavericks”) or 10.10 (“Yosemite”).

All code in this book uses the Swift programming language, which is new in Xcode 6, making this book one of the first to offer a thorough guide to using Swift with the iOS frameworks. Since there are no experts on Swift today other than the people who created it, we expect it will be as new to you as it was to us over the Summer while we wrote the book (confounded by the fact that we had to reset when each beta of Xcode 6 changed the language). Fortunately, it’s a neat language that cleans out a lot of cruft from C and Objective-C, and we think you’ll be able to pick it up quickly, provided you’re a proficient programmer in at least one object-oriented language. That can be one of the many curly-brace descendants of C (C++, C#, or Java), or an OO scripting language like Ruby or Python.

Online Resources

This book isn’t just about static words on a page or screen. It comes with a web page, http://www.pragprog.com/titles/adios2, where you can learn more and access useful resources:

- Download the complete source code for all the code examples in the book as ready-to-build Xcode projects.
- Participate in a discussion forum with other readers, fellow developers, and the authors.
- Help improve the book by reporting errata, such as content suggestions and typos.

If you’re reading the ebook, you can also access the source file for any code listing by clicking on the gray-green rectangle before the listing.

Staged Projects

As we build our sample projects in this book, we will often write simple code, only to rewrite it with more ambitious code later as our knowledge increases.

All the different versions would be hard to put in one source file. So in the downloadable book code, we often have multiple copies of each project, each representing a different stage of its development. The different stages use numbered folders, like PragmaticTweets-1-1, PragmaticTweets-2-1, and so on, with the first number representing the chapter number and the second being a revision within that chapter. These folder names also appear in the captions for each code example in the text. You can either code along for the entire book from scratch, or copy over one of these "stages" and pick up from there.

And So It Begins

So, new platform, new programming language… new opportunities for new ideas and new apps. We can't wait to see what you come up with. Let's get started.

CHAPTER 1

Playing Around with Xcode

If we're going to create iOS apps, we need the right tools for the job. In this chapter, we're going to get properly equipped for the journey, and play around a little with the tools that Apple provides us.

To develop iOS 8 apps, we use Xcode 6. While "Xcode" generally refers to the *integrated development environment* (IDE), in which we develop code and user interfaces and run a build process to generate the actual apps, it can also mean the entire collection of material we'll need to build iOS applications. When we download Xcode, we get not only the Xcode app itself but also the software development kits (SDKs) for iOS and Mac OS X, which contain documentation, frameworks, helper applications, sample code, and more. These all live inside the Xcode application itself, so we don't have a bunch of fiddly little files to manage.

Xcode is available for free via the Mac App Store, so download it now if you haven't already done so. Xcode typically supports the current version of OS X and one back, so for Xcode 6, you'll need to be on Mac OS X 10.9.3 ("Mavericks") or newer (currently Mac OS X 10.10 "Yosemite"). The Mac App Store will put Xcode in the /Applications directory. It's a good idea to drag it from there to the Dock so it's always handy.

Before we commit to building a full-blown app, let's play around with Xcode a little.

Xcode Playgrounds

When we first launch Xcode, it may need to do some one-time-only setup work, such as asking permission to install components like the "Mobile Device framework." When Xcode finally comes up, we get a greeting window (like the one below) that shows a few quick-start buttons on the left, along with a list

of most recent items we've opened on the right. We'll be starting a project here shortly, but to warm up, we're going to play in a *playground*. This is a new feature in Xcode 6 that allows us to interact with Swift code, to try out ideas by just writing code and seeing the results immediately, without an entire build-and-deploy cycle.

Figure 1—Xcode greeting window

The greeting window offers a button to "Get started with a Playground". This will bring up a sheet asking if we want to create an iOS or OS X playground, as seen below. Give it a name (or accept the default `MyPlayground`), make sure the type is "iOS", and click the "Next" button. In the file dialog that appears, choose a name and location for the `.playground` file that will be created; in this example, we kept the default name `MyPlayground.playground`.

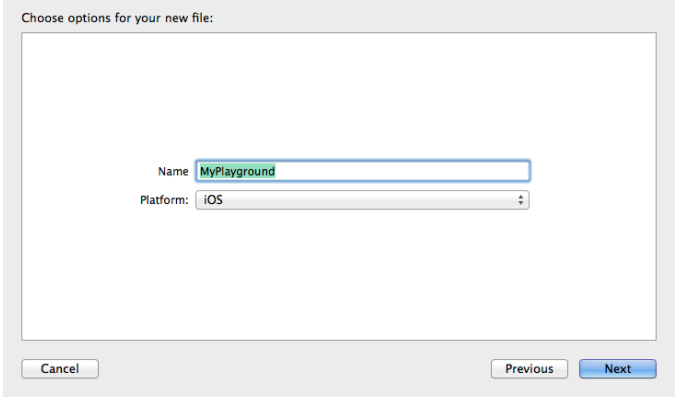


Figure 2—Creating a Playground file

The playground opens as a window with two panes: some source code on the left, and a strip on the right that shows the result of evaluating each line of code. The initial code looks like this:

```
// Playground - noun: a place where people can play

import UIKit

var str = "Hello, playground"
```

This is our first look at Swift, and it looks a lot like other C-based programming languages like Java or C#. The first line is obviously a comment, and the second says we want to use classes from "UIKit", the iOS user interface framework. The third line assigns the text "Hello, playground" to a variable called str. The "Hello, playground" in the right pane lines up with the variable assignment, because it represents the result of evaluating that line of code.

The way the playground works is that we can type code into the left pane, and see any results in the right. So that's what we'll do now: we'll write some code and see what happens.

We're putting off our formal introduction to Swift and frameworks like UIKit until the next few chapters , so take our code on faith for the moment. We're going to start by asking UIKit to start creating some UI objects. On the next blank line, write the following code:

```
var myLabel = UILabel (frame: CGRectMake(0.0, 0.0, 200.0, 100.0))
myLabel.text = str
```

This creates a UILabel, a non-interactive view that shows some text. We have to give it a "frame" that represents an x,y origin point and a width-by-height size, which we do with the embedded CGRectMake() function. Then on the second line, we set the label's text to the str variable, which was set earlier to "Hello, playground".

So what? The evaluation pane just says UILabel. But mouse over that and notice that two small buttons appear: an "eye", and a circle. The eye icon only appears when the Playground knows that the object its evaluated has a graphic representation, so click that. The result is a popover like the one shown here.

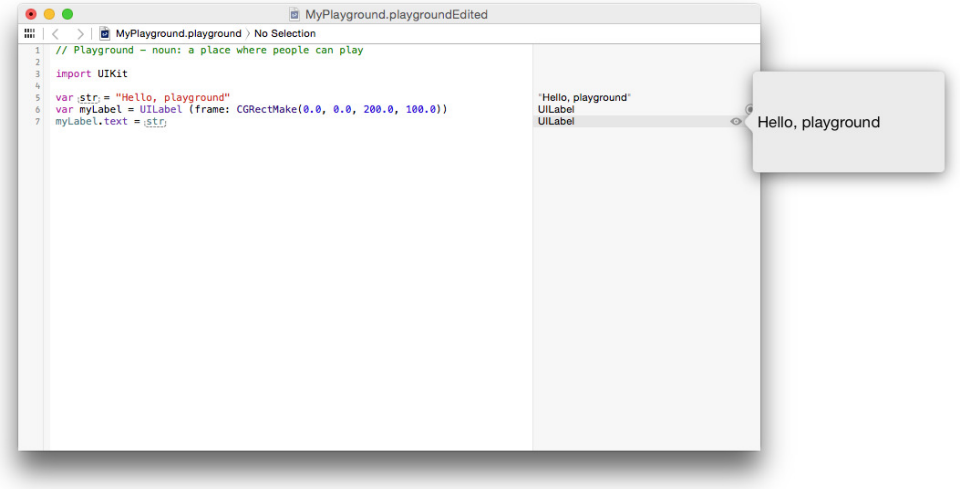


Figure 3—Creating a UILabel in a playground

OK, fun, we can create UI components in the Playground. We can customize it further. Add two more lines of code to set the background and foreground colors:

```
myLabel.backgroundColor = UIColor.redColor()
myLabel.textColor = UIColor.whiteColor()
```

Again, click the "eye" icon on the last line and see that the label does now indeed have white text on a red background. Now, also notice that each line of the evaluation pane has an "eye" icon. What this pane is showing is the state immediately after that line of code executes. So we can click the "eye" icons one after another to see popovers with state of the label changing as we create it, set its text, and reset its colors.

We can also click the round button on each line to split the window in two, with a new right pane showing the clicked line of code and its result. This lets us walk through the history of our changes and compare them side-by-side, rather than in ephemeral popovers. Here's what that looks like if we the buttons for each `myLabel`, from top to bottom:

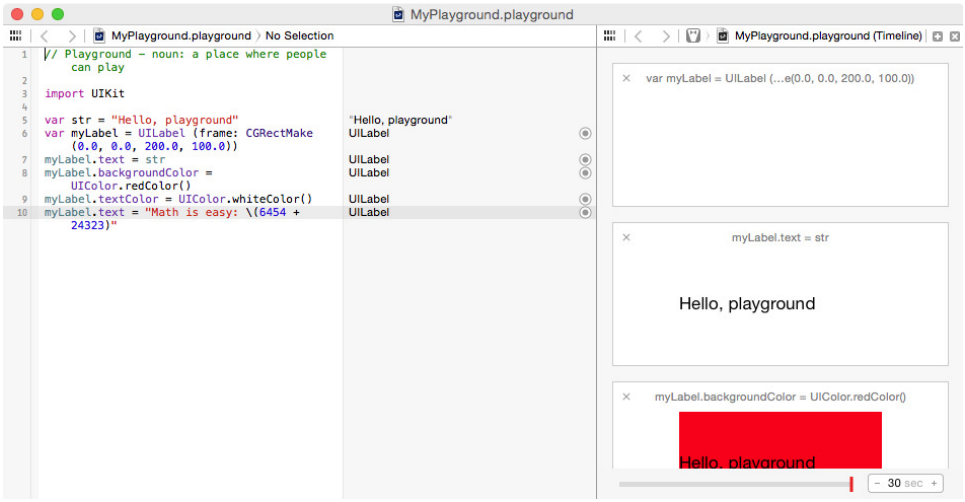


Figure 4—Value history of a UILabel in a playground

For a kicker, let's have our computer actually do some computing. Swift lets us build strings from smaller parts, and uses the sequence `\()` to say "evaluate what's inside the parentheses, and insert that into the string". So add one last line like the following, performing whatever mathematical expression suits you:

```
myLabel.text = "Math is easy: \(6454 + 24323)"
```

Click the "eye" icon next to this and we'll see our red-and-white label with the text `Math is easy: 30777`, or whatever the answer to our math problem ended up being.

Playgrounds are a fun way to test out ideas in code. When we find new classes in the documentation or online, we can always jump into a Playground and throw a few lines of code at it to see how stuff works.

But Playgrounds are limited, too. What we're doing looks nothing like a real app, it takes five lines of code just to create a label (with some manual coordinates to set the sizing), it doesn't have anything we can touch or interact with, and didn't we say there were special tools to build our GUIs without code? There's no icon, there's no screen… it seems like we're a long way from having a real app!

Our next step is going to be to figure out how to move past playing around in the playground, and start building real apps.

My First Computer (Chris)

Messing around in the Xcode Playground reminds me of how computers used to dump you into an interactive programming environment by default. My first computer was a Texas Instruments TI-99 4/A, bought by my father because the saleswoman at K-Mart was really persuasive, and maybe because it cost $400 when the Apple II's we had at school were over $1,000 (*Plus ça change, plus c'est la même chose*).

When you turned it on, the menu gave you two choices by default: `TI BASIC`, or whatever cartridge (if any) was in the slot. With no cartridge, all it could do was BASIC. Back in those days, messing around with some flavor of BASIC was what every home computer did. For a while, the idea was that anyone could program — and would actually want to — and pretty much any student in our school at least knew enough to do `10 PRINT "SARA IS GREAT" 20 GOTO 10`.

In the TV documentary *Triumph of the Nerds*, Steve Jobs once recalled that for every one person that wanted to hack on hardware in that era, there were another thousand that wanted to hack on software. And beyond Jobs' observation, it seems that for every one person that wanted to hack on software, another thousand just wanted to run the stuff, without necessarily knowing how it works. Inevitably, computers got away writing programs as being the primary user experience, and coding eventually became a specialist skill and no longer accessible to the layperson, despite the occasional programming renaissance like Hypercard.

Having a Playground is like going back to those Summer nights of the 1980's, with mosquitos banging off the window screen (attracted by the glow of the TV that served as a monitor), a Styx cassette playing on the tape player that was used to load and save programs, and a blinking cursor inviting me to write some code… just to see what happens.

Digging Into the Docs

Throughout the book, we're going to build out a single application, a Twitter client that gradually gains more and more features: sending Tweets, getting the user's timeline, getting details about a Tweet or a user, and so on. One reason this makes sense as a sample project is because the process of getting data from the internet, presenting it to the user, handling their input, repeat… this is the heart and soul of many iOS apps, and reconciling webservice APIs with the iOS frameworks is an essential skill for iOS developers to master.

The ability for apps to send tweets from apps has been in iOS for a while — most of us have used apps that offer to tweet on our behalf — but how would we know how to use it in our app? This calls for some documentation.

Xcode's documentation viewer is available via the menu item "Help → Documentation and API Reference", also accessible via the keyboard shortcut ⇧⌘0.

However, when we try to browse the documentation, we may be challenged to sign in with an Apple ID. If you haven't joined the developer program at http://developer.apple.com/, the only way to see the docs is to download a local copy. This is a good thing for all of us to do anyway, so we'll have a local copy to refer to when we don't have Internet access. In Xcode's preferences, click the Downloads icon and then the Documentation tab. This pane, shown below, lists the available doc sets and versions of the iOS Simulator and whether they're installed or need to be downloaded. If the current version of iOS—8.0 as of this writing—doesn't say "Installed," then just click the Install button to download it to your Mac.

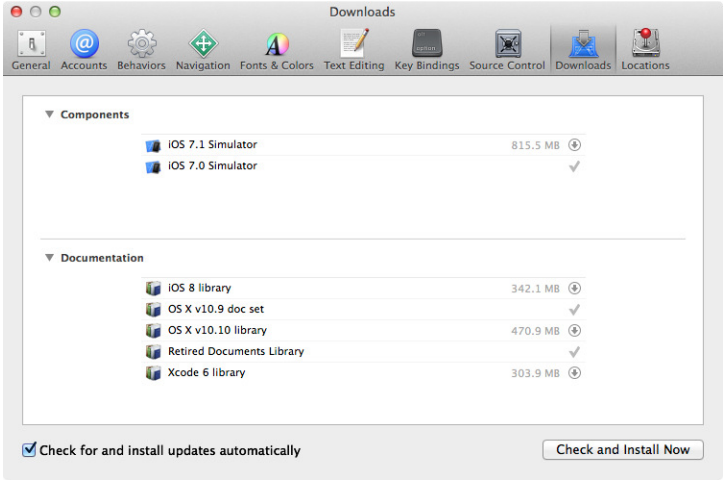


Figure 5—Downloading documentation with Xcode preferences

Let's go back to the documentation viewer, seen in Figure 6, *Viewing documentation in Xcode Organizer*, on page 8. This window is organized around a search bar at the top, with a content pane below it. There are also optional Bookmarks and Table of Contents panes on the right and left that can be shown and hidden with toolbar buttons, similar to the ones used for hiding and showing panes in the project workspace window. A magnifying glass icon on the search bar lets you choose whether to search iOS or OS X documentation (or both, or just let Xcode choose automatically). When we type a search term, the best results are shown as a popup list under the search bar, and we can choose one of these results, or just pick "Show all results" to present all matches as clickable links in the content pane.

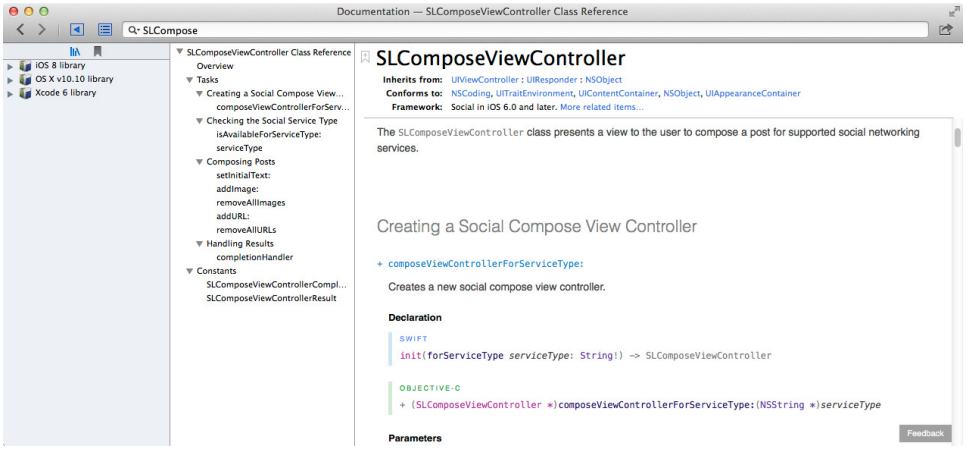


Figure 6—Viewing documentation in Xcode Organizer

What are we looking for exactly? Since we want to send a tweet, we can enter `twitter` as the search term, and we'll find a whole "Twitter Framework Reference". Tempting, but take it from us that that's old news — it was a Twitter-only framework for iOS 5 that has been supplanted by a new framework that works with multiple social networks. Keep looking and one of the results will be the constant `SLServiceTypeTwitter`. Select this result and the documentation will follow the link to the middle of a class called `SLRequest`, with `SLServiceTypeTwitter` defined as follows:

> A string constant that identifies the social networking site Twitter

Scroll to the top of this file to see the essential traits of this `SLRequest` class: what class it inherits from (`NSObject`), what versions of iOS provide it (iOS 6 and up), and what framework it's a part of. Search again for `social framework`, and select the result for "Social Framework Reference". This document begins:

> The Social framework lets you integrate your application with supported social networking services. The framework provides a template for creating HTTP requests and provides a generalized interface for posting requests on behalf of the user.

"Posting requests on behalf of the user"? Perfect, this sounds like just what we need. Scroll down to the class listing and discover that there are just four classes, including the `SLRequest` in which we found the `SLServiceTypeTwitter` constant and an `SLComposeViewController`. Click the link to the latter and take a look at its documentation, which begins like this:

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

This looks particularly promising. We can use this SLComposeViewController to post to Twitter, and presumably our same code will work just as well for other social networking services.

It's highly tempting to figure out how to use this in our playground, but it turns out to be impractical to do so. The SLComposeViewController needs to be presented by another "view controller", and we don't have one of those in our playground or a reasonable way to get one. At this point, it's going to make more sense to start building our first project, which will be the task of the next chapter.

Wrap-Up

Now let's tally up what we've learned so far. We've downloaded and installed Xcode, and we've tried out its playground feature. In a playground, we can try out little bits of Swift code to see what they do, making them a good place for experimentation and discovery. We've also familiarized ourselves with looking up documentation on the iOS SDK, so we can find the classes and methods we'll need to build our apps.

We're now ready to start building our Twitter app, so let's move ahead and see how to create Xcode projects.

CHAPTER 2

Building Adaptive User Interfaces

We're going to kick things off with a little secret about iOS development, something it inherited from Mac development: *You're supposed to create the user interface first.* This is totally backwards for a lot of seasoned developers. A lot of us think through an application's requirements and immediately start thinking of our data models and strategies and… *nuh-uh.* Build the UI first. Build what the user is going to see, what they're going to interact with, and start to understand how they'll experience it. Then figure out how the heck you're going to do that.

That philosophy is reflected in the tools provided to us for iOS development. If we built the user interface by writing code, it would be natural to code the functionality and then put buttons and views on top of it. Instead, the iOS SDK provides distinct tools for building the user interface graphically and for coding its functionality. The tools let us see our interface first, and then make it work.

We're going to begin work on our Twitter app with a super-simple UI: a button that sends tweets. From there, we'll revise our appearance and the code behind it, to the point where our app will approach the functionality of the Twitter clients on the App Store today.

Our First Project

To begin work on a new app, we need to start a new project, which we can do with the menu sequence File→New→New Project… (⇧⌘N). There was also a button on the Xcode greeting window for starting a project, so that's another way to do it.

When we create a new project, a window opens and immediately slides out a sheet that asks us what kind of project we want to create. This project template

sheet, seen below, has a list on the left side of project categories divided into iOS and Mac OS X sections. Since we're building an iOS application, we'll select "iOS Application" and then look at the choices in the main part of the frame. We can click each to see a general description of what kind of app to start on. For our first example, we'll select Single View Application.

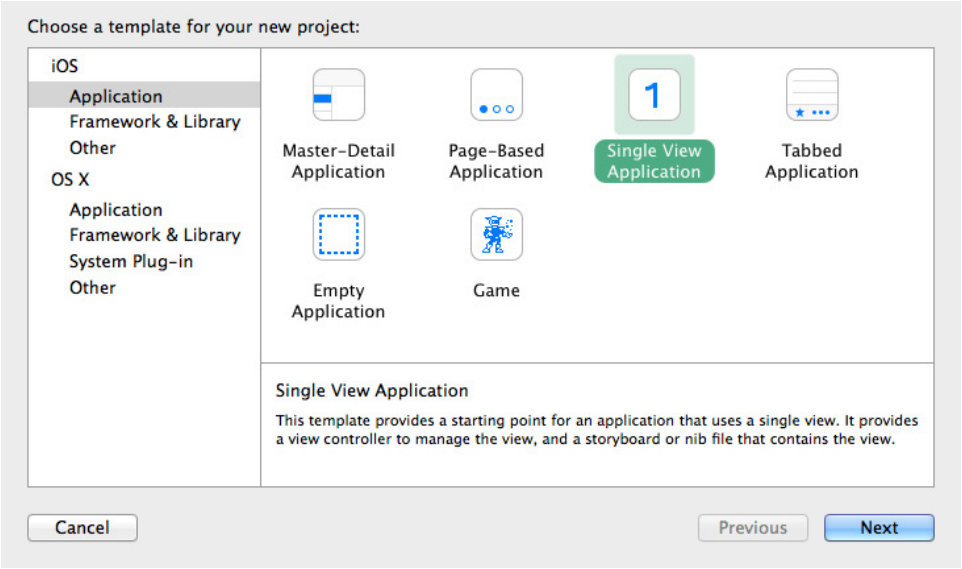


Figure 7—Xcode project templates

After clicking Next, the sheet then asks us for details specific to the project, as shown in the following figure. Some of these change based on the project type; in general, this is where we need to provide names and other identifiers to the app, indicate which device formats (iPhone and/or iPad) it's for, and so on. For our first app, here's how we should fill out the form:

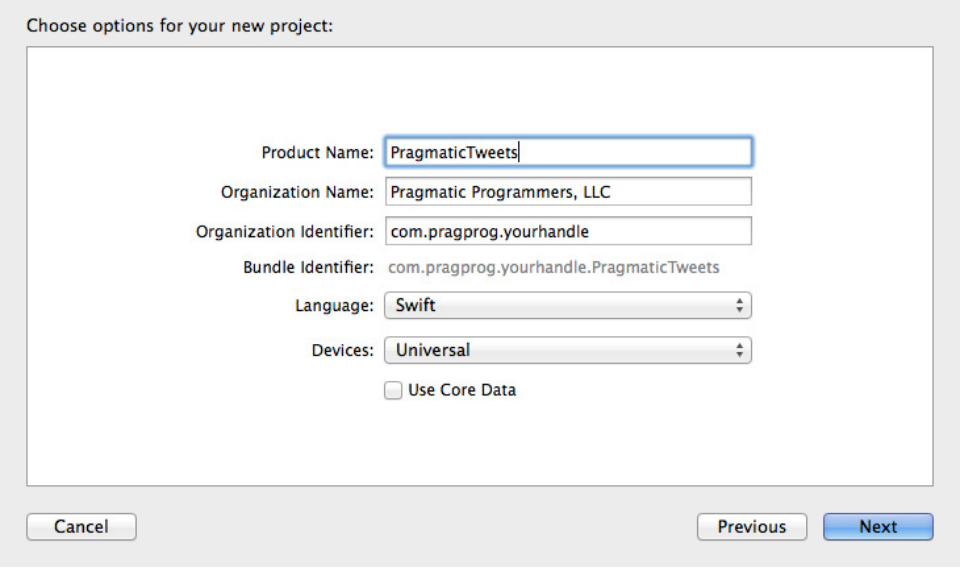


Figure 8—Xcode project creation options

- *Product Name*—A name for the product with no spaces or other punctuation. Our product will be called PragmaticTweets here.
- *Organization name*—This can be a company, organization, or personal name, which will be used for the copyright statement automatically put at the top of every source file.
- *Company Identifier*—This is a reverse-DNS style stub that will uniquely identify our app in the App Store, so if someone else creates a PragmaticTweets the two apps won't be mistaken for each other because they'll each have a unique *Bundle Identifier*, which is the auto-generated fourth line of the form. If you have your own domain, you can use it for the company identifier; otherwise, just invert your email address, such as in com.company.yourhandle.
- *Language*—There are two choices for this pop-up menu: *Swift* and *Objective-C*. Swift is a new language introduced by Apple in 2014 for iOS and OS X development, which is meant to clean up some of the cruft hat has accumulated around C and Objective-C over the decades. We'll have a lot more to say about Swift in the next chapter. For now, since we're starting a new project, with no need to support old code, we'll choose Swift here.

- *Devices*—This determines whether the template should set us up with an app that's meant to run on an iPhone (and iPod touch) or iPad or be a "universal" app with a different layout for each. Not all templates offer all three options. For iOS 8, Apple is pushing hard for developers to build Universal apps that run and look good on a variety of screen sizes, from iPhones to iPads and perhaps some new product sizes in between, so select Universal here.

After clicking Next, we choose a location on the filesystem for our project. There's also an option for creating a local Git source code repository for our files; we'll look at source control in *the (as yet) unwritten sect.publishing.source-control*I don't know how to generate a cross reference to sect.publishing.sourcecontrol, so we can leave it unchecked for now. Once we specify where the project will be saved, Xcode copies over some starter files for our project and reveals them in its main window.

The Xcode Project Workspace

Xcode 6 provides a single window for a project, called the *project workspace*. This window provides our view into nearly everything we'll do with a project: editing code and user interfaces; adjusting settings for how the project is built and run; employing debugging tools; and viewing logged output.

The project workspace is split into five areas; although, some of them can be hidden with menu commands and/or toolbar buttons. These areas are shown in an "exploded" view in the following figure:

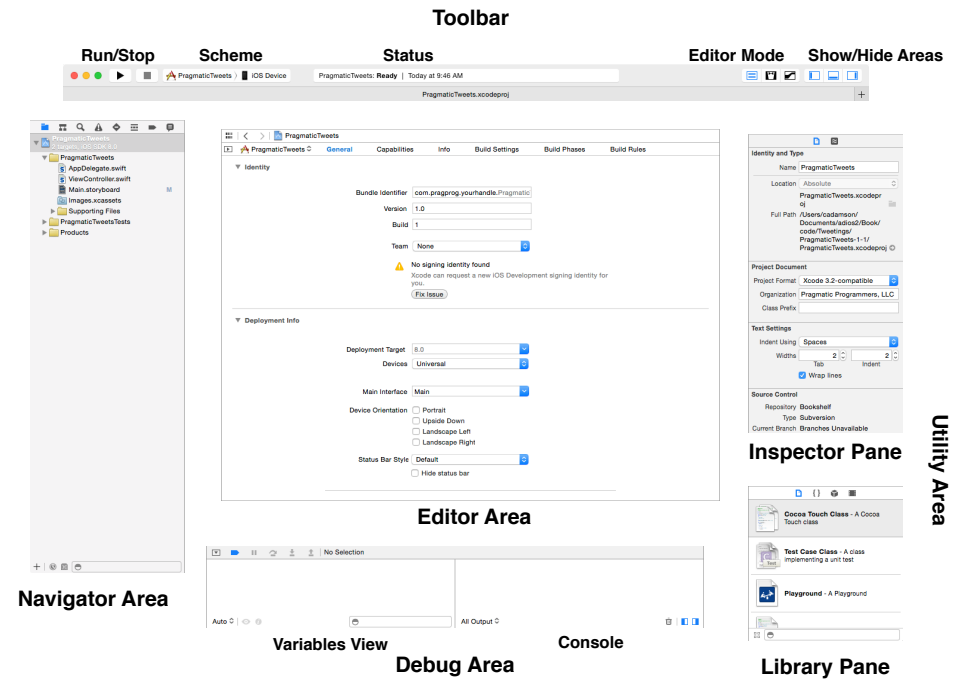


Figure 9—Parts of the Xcode project workspace

The project workspace is split up as follows:

Toolbar The toolbar at the top of the window offers the most basic controls for building projects and working with the rest of the workspace. The leftmost buttons, Run and Stop, start and stop build-and-run cycles. Next are two borderless buttons collectively known as the scheme selector, which chooses which "target" to run (currently "PragmaticTweets") and in what environment (the "iPhone Retina (4 inch)" simulator). Next comes an iTunes-like status display that shows the most recent build and/or run results, including a count of warnings and errors generated by a continual background analysis of the code. Further right, the Editor Mode buttons let us switch between three different kinds of editors, as explained below. Finally, three View buttons allow us to show or hide the Navigator, Debug, and Utility areas (see below).

Navigator Area The left pane (which may be hidden if the leftmost View button in the toolbar is unselected) offers high-level browsing of our project's contents. It has a mini-toolbar to switch between eight different navigators. The file navigator (⌘1) shows the project's source and resource files and

is therefore the most important and commonly used of the seven. Other navigators let us perform searches (⌘3), inspect build warnings and errors, inspect runtime threads and breakpoints, and more.

Editor Area The main part of the project workspace is the Editor area. This view cannot be hidden. Its contents are set by selecting a file in the Navigator area, and the form the editor takes depends on the file being edited. For example, when a source file is selected, we see a typical source code editor. But when a GUI file is selected, the Editor area becomes a visual GUI editor, and when an image file like a .gif or .jpeg is selected, the Editor area displays the image.

The Editor Mode buttons in the toolbar switch the editor pane between three modes: standard, which is the default editor for the type of file that's selected; assistant, which shows related files side-by-side; and version, which uses source control to show current and historical versions of the file side-by-side, a "blame" mode that shows the committer of each line of code, or a log of commit comments alongside the code. The Editor area also contains a *jump bar*, a bread-crumb-style strip at the top that shows the hierarchy of the thing being edited; for a source file, this might go "project, group, file, method." Each member of the jump bar is a pop-up menu that navigates to related or recent points of interest.

By default, a new project comes up with its top-level settings selected in the Navigator area, which means that the Editor area defaults to showing settings for things like the app version number, the targeted SDK version and device families, and so on. There may also be a scary-looking "No matching provisioning profiles found" warning, which just means we're not set up to run our app on a real device yet; we'll deal with that in *the (as yet) unwritten sect.publishing.signing.portal*I don't know how to generate a cross reference to sect.publishing.signing.portal.

Utility Area The right side of the project workspace is a utility area that provides detailed viewing and editing of specific selections in the Editor area. Depending on the file being edited, the toolbar atop this area can show different tools in its Inspector pane. Basic information about a selected file and quick help on the current selection are always available. For GUI files, there are inspectors to work with individual UI objects' class identities (⌥⌘3), their settable attributes (⌥⌘4), their size and layout (⌥⌘5), and their connections to source code (⌥⌘6). We'll be using all of these shortly. At the bottom of the Utility area, a library pane gives us click-and-drag access to common code snippets, UI objects, and more.

Debug Area The bottom of the window, below the Editor area and between the Navigator and Utility areas, is a view for debugging information when an app is running. Its tiny toolbar has a segmented button that lets us switch between the debugging-oriented *variables view* that lets us inspect memory when stopped on a breakpoint, a textual *console view* of logging output from the application, or a split view of both. We'll make use of the right-side console view in a little bit, while the left-side variables view will be our focus in Chapter 14, *Debugging Apps*, on page 221.

So that's how Xcode presents our initial project to us, but what can we do? Well, there's a big round Run button, and it's not like it's disabled. Let's try running the app. Make sure the scheme is some flavor of "iPhone" from the "iOS Simulator" section, and not "iPad" or the name of an actual device. Click the Run button. The status area will shade in with a progress bar that fills up as it builds all the files and bundles them into an app, and when it's done, it will launch the iOS Simulator. The Simulator is another OS X application, which looks and behaves more or less like a real iPhone or iPad. When our app runs in the Simulator, the main screen disappears and is replaced by a big gray box that fills the Simulator screen.

Building Our User Interface

That gray box in the Simulator is our app. It's not much, but then again, we haven't done anything yet. Let's start building our app for real. Press Stop in Xcode to stop the simulated app, and then take a look at the project in Xcode.

If the file navigator isn't already showing on the left side of the project workspace window, bring it up with ⌘1. The file navigator uses a tree-style hierarchy with a blue Xcode document at the top, representing the project itself as the root. Under this are files and folders. The folder icons are *groups* that collect related files, such as the views and logic classes for one part of the app; groups don't usually represent actual directories on the filesystem. We can expand all the groups to see the contents of the project, like this:

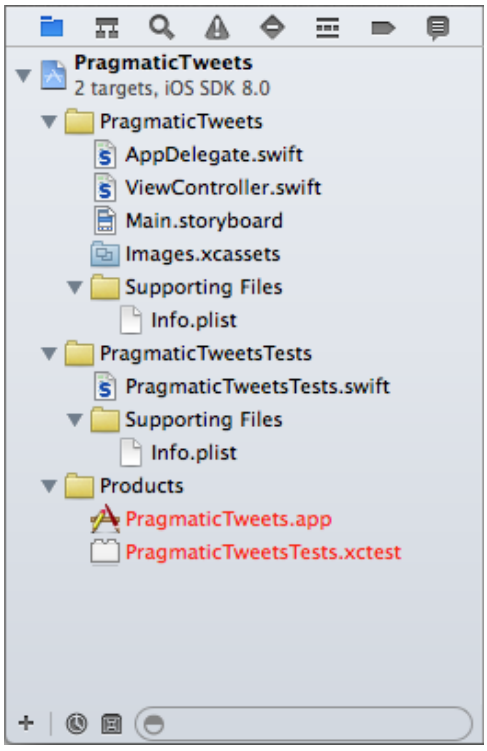


Figure 10—Default files in PragmaticTweets project

Different project templates will set us up with different files. For the view-based app, we get two source code files in the PragmaticTweets group, along with a Main.storyboard and a Images.xcassets. These are the files we'll be editing. The files in the Supporting Files group help build and run our app, but we won't need to edit them directly. The PragmaticTweetsTests group is where we will write unit tests to validate our code, something we'll do much later in Chapter 4, *Testing Apps*, on page 67. Finally, the Products group shows the files our build will create: in this case, PragmaticTweets.app for the app and PragmaticTweetsTests.xctest for the runnable unit tests. Files shown in red indicate they haven't been built yet; PragmaticTweets.app is red in the figure because although we've run it in the simulator, we haven't built it for the actual device yet.

We said at the outset that iOS development traditionally starts with the user interface. By focusing on what the user sees and how he or she interacts with it, we keep our focus on the user experience and not on the data models and logic behind the scenes. On iOS, we typically build our user interfaces visually

and store them in *storyboards*. The project has one such file, Main.storyboard, so let's click it.

Storyboards

When we click on Main.storyboard, the Editor area switches to a graphical view called *Interface Builder*, or IB for short. In iOS, IB works with user interface documents called *storyboards*. Just like in movie-making, where a storyboard is a process used to plan out a sequence of shots in a movie or TV show, the storyboard of an iOS app shows the progression through the different views the app will present. The initial storyboard is shown looks like the following figure.

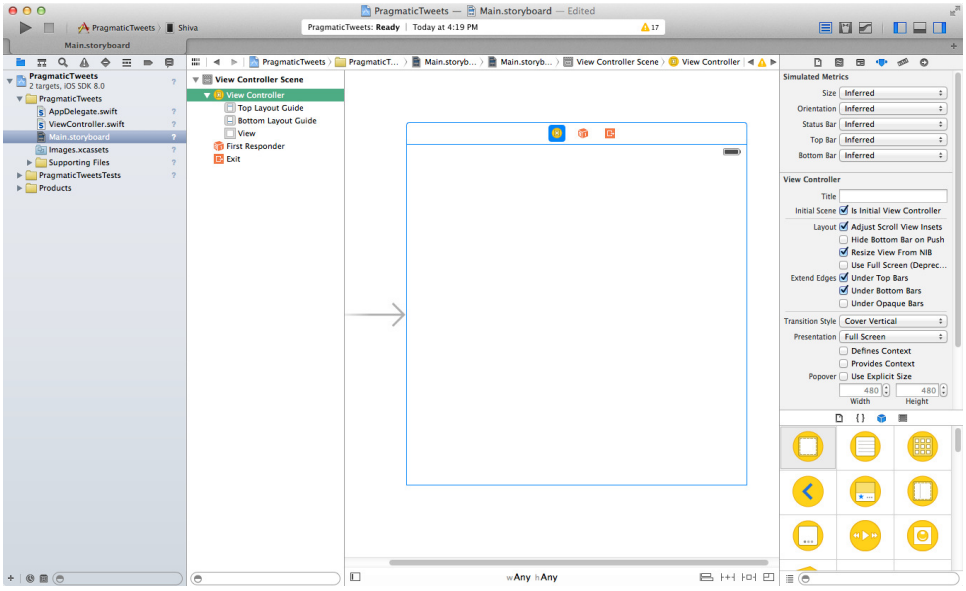


Figure 11—Interface Builder showing initial single-view storyboard

Our app uses a single view, so we follow the rightward arrow (which indicates where the app starts) into a square that represents the visible area of the screen. This is our app's one view; if we were building a navigation-style app, there would be one view rectangle for each screen of the navigation. Above the view we find a box with three icons. These are *proxy objects* that represent objects that will work with the view at runtime: a *view controller* that contains logic to respond to events and update the view, a *first responder* that represents the ability to handle events, and an *exit segue* used for when we back

out of views in navigation apps (something we'll visit in *the (as yet) unwritten chp.viewcontrollers*I don't know how to generate a cross reference to chp.viewcontrollers).

At the bottom left of the Editor area, IB shows a little view disclosure button. Click this to show and hide the scene list (already showing in this figure), which shows each "scene" of the storyboard and its contents as a tree structure. Currently, our one scene has the proxy objects discussed above, and inside the view controller, we find two layout objects and a "View". This view is the big square in the UI; as we add UI elements like buttons and labels, the scene's tree list will show them as children of this view.

But wait a minute!, you might say, *iPhones aren't square, and neither are iPads!* Quite right. What we're seeing in our startup view is Apple pushing developers to "think different" about device sizes. At the bottom of the IB pane, a label indicates our current layout as "w: Any h: Any". This is actually a button that allows us to try our user interface layouts in different sizes and orientations. Click the label to show the sizing popover, which looks like the one in this figure:

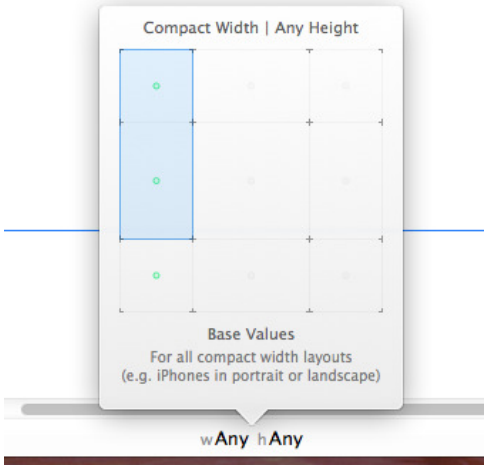


Figure 12—Interface Builder size preview popover

As we mouse over the grid of boxes in this popover, we can switch the height and width previews between "compact", "any", and "regular", and the popover titles will give us a hint of the class of sizes we're previewing, like "iPhone in landscape orientation". Click on the box to change the preview to see the main view change to this size and shape. Once we start laying out some contents for the view, this is how we will preview how they'll be laid out on

different device sizes, and when we rotate from portrait to landscape or vice versa.

Adding Buttons

So let's start adding some UI elements to our view. We'll begin by adding a button to send a tweet telling the world that our first app is running. To add components to our storyboard, use the toolbar to show the Utility area on the right (if it's not already showing), and find the Library pane at bottom right. There's a mini toolbar here which should default to showing user interface objects; if not, click the little cube (or press ^⌥⌘3). The bottom of the pane has button to toggle between list and icon views for the objects, and a search filter to find objects by name. Scroll down through this pane to find the "Button"; we can tap once on any of the objects to get its name, class, and description to appear in a popover. Drag the button from the object library into the iPhone-sized view in IB. This will create a plain button, as seen below.

Figure 13—An unmodified button in Interface Builder

It kind of leaves a lot to the imagination, huh? Without the edge and background decorations of earlier versions of iOS, it doesn't necessarily look like a button at all. It could easily be mistaken for a text label.

The new look of iOS, introduced back in iOS 7, has three stated themes: *deference, clarity, and depth.* The first of these, deference, means that the UI appearance focuses attention on our content rather than competing with a bunch of pseudo-realistic effects.

So maybe our problem is a lack of content. iOS expects us to tell the user what's going on in our app, and we're not holding up our end of the deal yet. Let's fix that. First, we'll say what the button does. Double click on the button to change its name to "Send Tweet". Now it says what it does, but it still doesn't exactly feel button-y.

Maybe we can fix that by contrasting the blue text of the button with a plain label. Back in the object library at lower right, find the "Label" object, and drag one above the button. Change its text to "I finished the first project". Drag both objects so that they're centered in the view; a dashed blue line will

appear when we're centered, and the drag will snap to this position. The view should now look like the following figure.

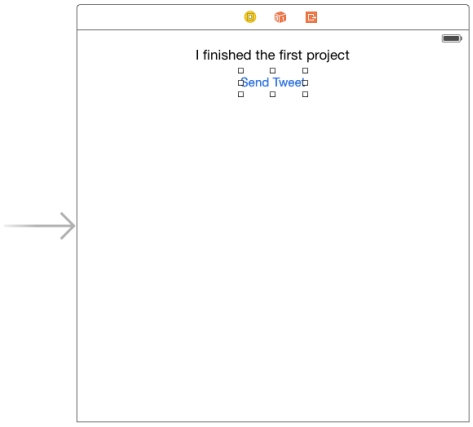


Figure 14—Initial app layout with label and button

Go ahead and click the Run button to run this app again in the iOS Simulator. We should just see the label and the button, right? Sure, but... there's a problem.

When we run the app in the simulator, we typically start in portrait orientation. And right now, that's going to be a problem, because our label and button are not centered in portrait; in fact, they're cut off on the right edge, as seen below. Rotate to landscape with "Rotate Left" and "Rotate Right" "Hardware" (⌘← and ⌘→, respectively), and it looks a little better, but it's still clearly not centered on tall models like the iPhone 5. What happened?

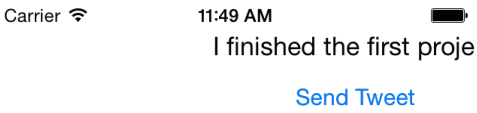


Figure 15—Initial app layout with mis-aligned label and button in portrait orientation

The problem is that we've been designing against a hypothetical square shape, and we never explicitly said these labels were supposed to be centered. What's happened instead is that they've kept a constant distance from the top and left sides of their parent view. In a way, it makes sense: iOS doesn't know

what matters to us: a constant distance from the top or bottom, or being centered, or some other relationship entirely.

Stop the simulator, go back to Xcode, and select the label. On the right side of the workspace, show the *Size Inspector*, by clicking the little ruler icon (or pressing ⌥⌘5). This inspector tells us about the size and location of elements in our UI. There's a section called "Contraints" which currently reads:

> The selected views have no constraints. At build time explicit left, top, width, and height constraints will be generated for the view.

Autolayout

In iOS, our UI elements are placed onscreen with an *autolayout* system that lets you determine where objects should go and how big they should be based on *constraints* that we set on them. This allows our interfaces to adapt to being rotated between portrait and landscape, and to handle the differing screen sizes of the 3.5-inch models (original iPhone through iPhone 4s), 4-inch models (iPhone 5, 5c, and 5s), the 4.7-inch iPhone 6, and 5.5-inch iPhone 6 Plus. Constraints allow us to express what matters to us — the size of components, their alignment with or distance from other components, etc. — and to let other factors vary as needed. In this example, we want our label and button to be horizontally centered, and we don't care what the resulting x and y coordinate values are.

Interface Builder puts a floating set of buttons at the bottom right of the pane to give us access to autolayout features. These buttons, shown in the following figure display a popover or pop-up menu when tapped.

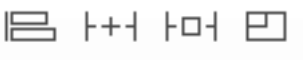


Figure 16—Interface Builder autolayout buttons

From left to right, these buttons are:

- *Align popover*: This lets us create constraints that align a views edges or horizontal or vertical center with another view, or horizontally or vertically centers it within its containing view (its *superview*).
- *Pin popover*: This lets us create constraints that specify a fixed value for spacing from one or more edges to another view (possibly the superview), and/or a fixed width or height.

- *Resolve menu*: The options here will adjust a view position or size so it matches its constraints, or do the opposite and create contraints based on its current position and size. We can also clear all constraints and start over with this menu.
- *Resizing Behavior menu*: This menu determines which views get their constraints adjusted when we drag handles to resize a view. By default, only its subviews get updated, but this menu lets us update sibling and ancestor views too.

Storyboard Zooming

Perhaps surprisingly, Xcode 6 removes zoom in/out buttons that used to be located alongside the autolayout buttons. With a trackpad, we can pinch zoom in and out to show more of the storyboard. Without a trackpad, there are zoom menu items available via a control-click in the Editor Area, or the menu item Editor→Canvas→Zoom.

So what we need to do is to just tell our label and button to be centered. Click the label and then click the Alignment button. This shows the popover seen below. Click the checkbox next to "Horizontal Center in Container". This will change the button at the bottom of the popover to say "Add 1 Constraint". Click this button to dismiss the popover (note that if we tap outside the popover instead of tapping the button, the popover will dismiss *without* creating the constraint).

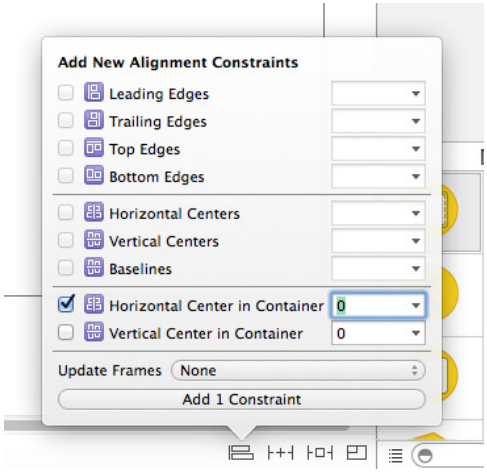


Figure 17—Creating a horizontal centering constraint

This causes an orange line to appear down the middle of the view when the label is selected. In Interface Builder, orange is a warning color, meaning *there aren't enough constraints*. The label is *under-constrained* because while we've provided a horizontal constraint, we haven't provided a vertical constraint, meaning autolayout can't know for sure how high or low on the screen to place the label.

Since we're happy with what the label looks like in Interface Builder, let's just tell it to keep this same distance from the top of the container view. We do that by *pinning* its distance from the top. With the label selected, click the Pin button to show the popover seen in the following figure.

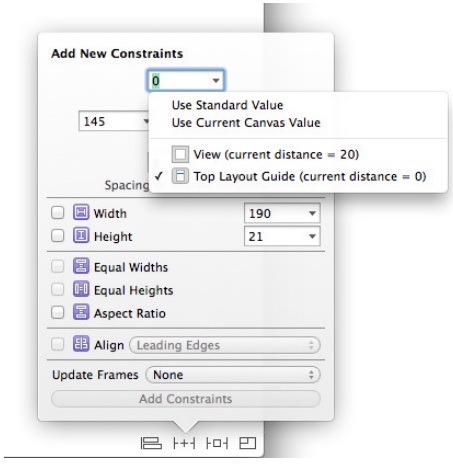


Figure 18—Creating a top spacing constraint

The pinning popover lets us lock down values such as width and height or distances from containers or other views, or force sides of multiple components to stay aligned. The top section is called "Spacing to nearest neighbor", and if we click the top popup menu in this part, we can select the value "Top Layout Guide", meaning we want to lock the distance between the top of our button and the area at the top of the screen reserved for the status bar (where the battery level, signal strength, clock, and other indicators appear), or any other menus at the top of the screen (like the navigation bar we'll introduce much later). When we select this menu item, the "brace" graphic under the menu becomes solid, and the button in the popover again says "Add 1 Constraint". Click this button to add the new constraint.

Warning: Size-specific constraints

When creating constraints, it's important that the sizing bar at the bottom of the pane is in "w:Any h:Any" mode. While this creates somewhat unrealistic square views, the danger is that constraints created with any other sizes set for width or height *will only apply for those sizes*. If we set width to "Compact" to preview the appearance of an iPhone in portrait and add a constraint like horizontal centering, that will only apply for compact width, so there will be *no* constraint for landscape, or on an iPad, since those aren't cases of compact width.

We learned this one the hard way, when one of our buttons went flying offscreen on the iPad. We'll talk about the underlying size concepts later, in *Size Classes*, on page 190.

Now when we select the label, IB shows the centering line as blue, and adds a blue brace from the top of the label up to the status bar area. Blue means that we have enough constraints to not be ambiguous to autolayout. Try running it again, and rotate the simulator. The label stays horizontally centered and maintains a constant distance from the top, while the button continues to be un-centered in landscape orientation.

We can repeat the same steps to fix the button. First, select the button, click the Alignment button, choose "Horizontal Center in Container", and do "Add 1 Constraint". This again gives us the orange line to tell us we're not quite done. Now click the pin button, and show the menu for the top spacing. This time we have a choice: we can pin either the distance to the Top Layout Guide as before, or the distance from the top of the button to the label. This is the power of constraints: we get to indicate what matters to us. Do we care about the button's distance to the top, or its relation to the label? In this case, the label helps explain what the button does, so it makes more sense to keep them together and pin the distance from the button to the label, rather than the button to the top. So, select "Label - I finished the …" and click "Add 1 Constraint". This gives us blue guides in IB, indicating that all is well, including a blue brace between the button and the label.

Run the app again and both our components are centered regardless of orientation. We can also change the device type between 3.5-inch and 4-inch iPhone models in the scheme selector to see the effect of the larger screen; it doesn't matter much now because our buttons' vertical positions are measured from the top of the screen, but it would be a big deal if we had anything pinned to the bottom, since we'd be losing a half inch of space in the middle as we go from an iPhone 5 to an older iPhone.

Also, think back to when we created the label in the Playground, explicitly setting its position and size in code. With autolayout, we get to describe size, shape, and position with constraints, whereas if we build our UI in code, we would be doing a bunch of math to set the position and size, using logic like "subtract the label's width from the superview's width and then indent half of that space to center up". For complex layouts on devices with different sizes and shapes, all of which can be rotated at any time, autolayout ends up being both easier and more dependable.

Connecting User Interface To Code

It's great that our UI handles resizing and orientation changes, but it still doesn't, y'know, *do anything* yet.

And that begs an interesting question: how do we get the 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 up to the toolbar and click the Assistant Editor button, which looks like a tuxedo. 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 for now.

The pane on the right side has a jump bar at the top to show which file is shown in that 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", etc. 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 file name at face value: this is the class that controls the view.

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

Creating the action turns out to be 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 or Action" 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, as seen in this figure:

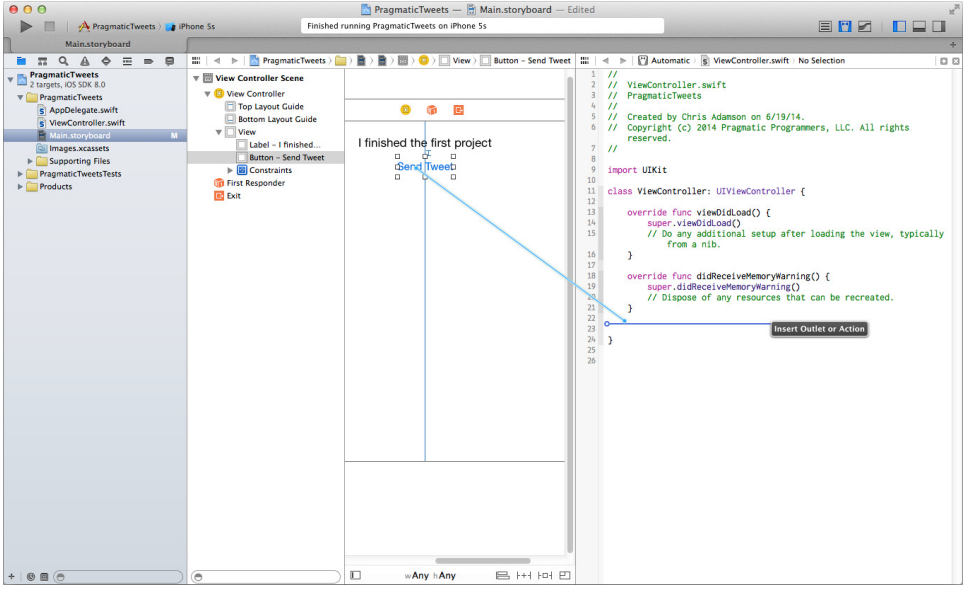


Figure 19—Connecting an IBAction with Assistant Editor

When we let up the mouse in the source file, a popover appears, asking us for the details needed to finish the method declaration, as seen in the figure below. On the first line, change the Connection from "Outlet" to "Action". Next, we need to provide a name for the method, 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 ”Argument” fields we can take the default values.

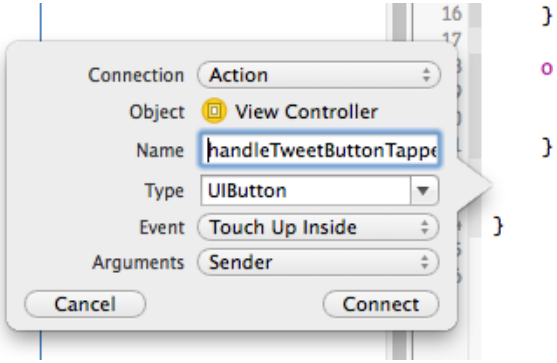


Figure 20—Configuring an IBAction

We’re done with the Assistant Editor, so click the Standard Editor button in the toolbar to return to one-pane mode. If we select ViewController.swift in the Navigator area, we can see that Xcode has stubbed out a method signature for us:

UserInterface/PragmaticTweets-2-1/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, there’s one connection shown in the “Sent Actions” section, from “Touch Up Inside” to “View Controller - handleTweetButtonTapped:”. This connection, shown in the next figure, 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.

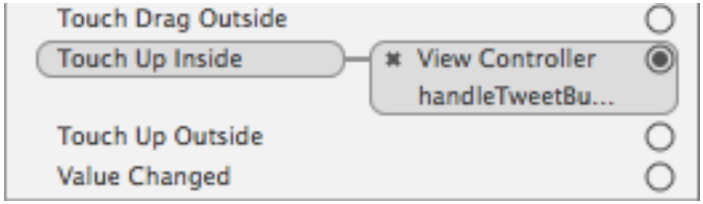


Figure 21—Touch Up Inside action as seen in Connections Inspector

Honestly, we don't break and re-make 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 App

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, then we can move on to implementing our tweet functionality.

Logging

On iOS, we can use the function println() to write a string out to the system's log file. We can implement 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:

UserInterface/PragmaticTweets-2-1/PragmaticTweets/ViewController.swift

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

Run the app again and tap the button. Back in Xcode, the Debug area automatically appears at the bottom of the project workspace 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 trash can 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.

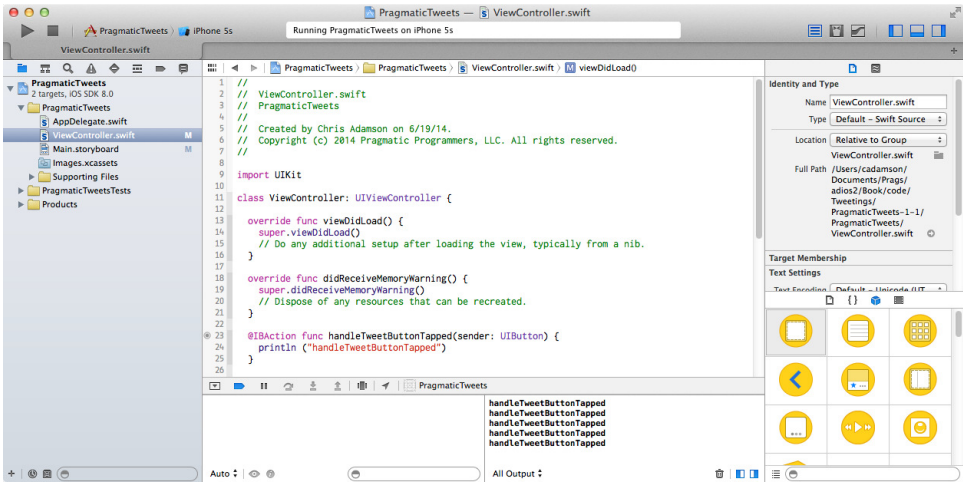


Figure 22—Logging to the Xcode Debug area

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!

Calling Up the SLComposeViewController

Back in *Digging Into the Docs, on page 6*, we discovered the SLComposeViewController class, which provides a fly-in view that lets the user compose and send a tweet. Armed with this class, we're ready to lay down some code. In PRPViewController.m, rewrite the handleTweetButtonTapped() method as follows:

UserInterface/PragmaticTweets-2-1/PragmaticTweets/ViewController.swift

```
Line 1 @IBAction func handleTweetButtonTapped(sender: UIButton) {
   2   if SLComposeViewController.isAvailableForServiceType(SLServiceTypeTwitter) {
   3     let tweetVC = SLComposeViewController (forServiceType: SLServiceTypeTwitter)
   4     tweetVC.setInitialText(
   5       "I just finished the first project in iOS 8 SDK Development. #pragsios8")
   6     self.presentViewController(tweetVC, animated: true, completion: nil)
   7   } else {
   8     println ("Can't send tweet")
   9   }
  10 }
```

We've replaced our one-line logging statement with several lines of Swift, which is what iOS uses for most of its high-level APIs, such as UIKit and the

Social framework. We'll have much more to say about the language in the next chapter, but for now, let's try to tease out how this code works. 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 4–5, we set the initial text of the tweet to "I just finished the first project in iOS 8 SDK Development. #pragsios8" by calling the setInitialText() method on tweetVC.

This is all we need to do to prepare the tweet, so on line 6, 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 canSendTweet() returned false, the else block on lines 7–9 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 below. Worse, depending on the width of the window, the errors are likely truncated. What are we supposed to do?

```
24  @IBAction func handleTweetButtonTapped(sender: UIButton) {
25    if SLComposeViewController.isAvailableForServiceType          Use of unresolved id...
          (SLServiceTypeTwitter) {                                  Use of unresolved identifi...
26      let tweetVC = SLComposeViewController (forServiceType: SLServiceTypeTwitter)
27      tweetVC.setInitialText(
28        "I just finished the first project in iOS SDK Development. #pragsios")
29      self.presentViewController(tweetVC, animated: true, completion: nil)
30    } else {
31      println ("Can't send tweet")
32    }
33  }
34
```

Figure 23—Build errors shown in source code editor

Broken Builds

Let's get a more detailed look at what's going on. Visit the log navigator using the rightmost button in the Navigator area toolbar, or just type ⌘8. This

replaces the list of files with a list of our builds and runs, with the most recent at the top. Click the top "Build", and the Content area shows a build log, as seen in the next figure. By default, the selected filter in this view is "All Issues", and aside from a possible warning about CODE_SIGN_ENTITLEMENTS (which you'll see so long as you aren't set up to build for actual iOS devices), most of the actual errors are Use of undeclared identifier 'SLComposeViewController' and Use of undeclared identifier 'SLServiceTypeTwitter', which in turn cause the later errors.

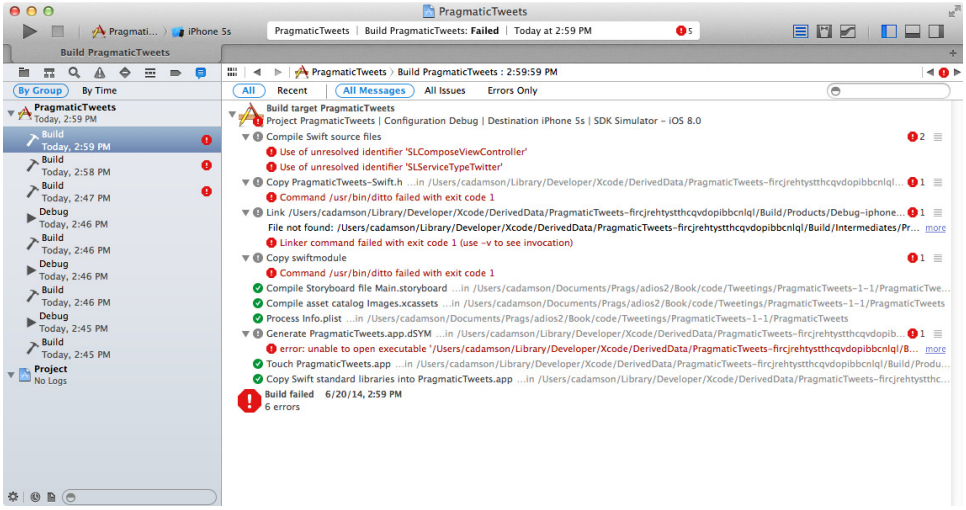


Figure 24—Build errors shown in log viewer

This error means that the compiler doesn't know we're using the Social framework, and therefore it doesn't recognize the SLComposeViewController. Xcode project templates only set us up to use the most common frameworks: Foundation, Core Graphics, UIKit, and XCTest. Anything else has to be added manually. So to tell the compiler about the Social framework, add the following line near the top of ViewController.swift, after the import UIKit line:

UserInterface/PragmaticTweets-2-1/PragmaticTweets/ViewController.swift

```
import Social
```

The import directive tells the compiler to pull in another framework. This tells the compiler and the linker about our dependency on the Social framework. Once we add the import Social declaration, the red error icons on the side of our code disappear. This is a good sign, so let's try running again.

Tweeting at Last

This time the build completes without errors, and the app will launch in the Simulator. Try clicking the button; it likely shows an error alert saying that no Twitter accounts have been configured, with buttons offering to take you to Settings or to cancel.

To fix this, we use the Simulator as we would a real iPhone: tap the Settings button in the alert, or use the Home button (menu item "Hardware → Home" or keyboard shortcut ⇧⌘H) to switch out of the app and launch the Settings app. In the Twitter settings, configure a Twitter account with a valid Twitter user name and password. If you don't have one, create a disposable one for free on http://twitter.com/, since we'll be using it throughout the book. With username and password entered, tap the Sign In button. Once the checkmarks appear to indicate the credentials have been accepted, use the Home button menu item again and switch back to PragmaticTweets. This time when you tap the button, the Tweet composer should come up, like the one below. Edit the text if desired and then click Send. Go visit your Twitter page on the Web with a browser—for style points, go ahead and use Safari in the Simulator—to see your brand-new tweet, posted for all the world to enjoy and admire.

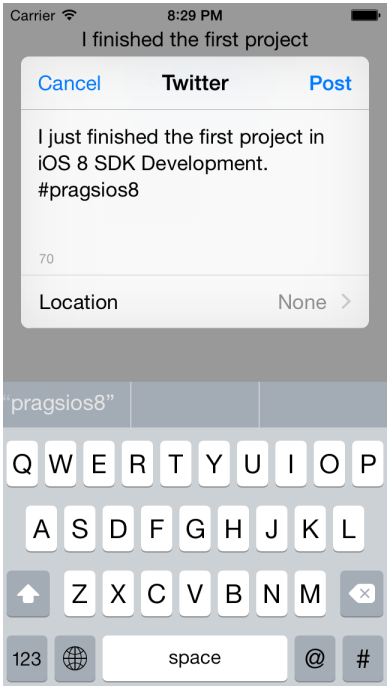


Figure 25—Sending a tweet with SLComposeViewController

Sweet, Tweet Success

In this chapter, we've gotten our first project built, launched, and sending data to the Internet. We downloaded and installed Xcode, created a view-based project, and started customizing it. We customized the user interface, setup autolayout constraints to deal with different device sizes and orientations, connected a `UIButton` in the storyboard to a method in our `ViewController` class, and implemented that method to show iOS' default tweet-composer UI. When it was all done, with just a little bit of code and UI tweaking, we ended up with an app that sends tweets. Not bad for one chapter's work.

Now that we've had our first experience with the tools that the SDK gives us, we're going to use the next chapter to learn more about the Swift programming language that iOS uses for its high-level frameworks, and in the process we'll make our app more interesting and more functional.

CHAPTER 3

Programming in Swift for iOS

In a single chapter, we've learned enough about the iOS SDK to write a simple app that can send tweets on our behalf to the Internet. Still, we did have to take it on faith that the code we used to create the SLComposeViewController would do what we needed it to. Now that we've gained some familiarity with our tools, it's time to do the same with the language and libraries.

In this chapter, we're going to look at the fundamentals of coding an iOS app: programming with Swift and calling the default frameworks, UIKit and Foundation. As we go, we'll make a series of enhancements to our Twitter app. We'll add the ability to see our twitter.com page, internationalize the app for other languages and locales, and see how the language and tools work together to make us more productive. Each time we need to change the UI, we'll edit the storyboard, and when we need new functionality, we'll write more Swift code.

Introducing Swift

So what is Swift and why are we using it? In the previous chapter, we had a choice of using Swift or Objective-C for our project, and we chose Swift. There's a story there, of course. Objective-C has been the primary language of iOS development since the iPhone's debut in 2007, and goes way back from there: through Mac OS X's "Cocoa" in the early 2000's, and the NeXTStep environment of the 1990's. It started as a means of adding object-oriented features to C, and while it was enhanced in many clever and compelling ways over the years, various C legacies held back efforts to make Objective-C faster, safer, and easier to program.

So, in June 2014, Apple surprised the developer community by announcing they had created a new language as a successor to Objective-C on Apple platforms, one that could build on Objective-C's strengths while eliminating

its weaknesses, and drawing on good ideas in other languages. According to creator, Chris Lattner, Swift took inspiration from "Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and far too many others to list."

Joe asks:

Why Didn't They Just Use Ruby?

With the transition from Objective-C to Swift, Apple is transitioning developers from one language under its control to another that is, if anything, even more proprietary. Why not use the opportunity to adopt a popular language like Ruby or Python?

One important reason is that compatibility with the existing frameworks — a mixture of Objective-C, C, and even some C++ — was required. Re-working thousands of classes and tens or hundreds of thousands of methods and functions would be a mammoth undertaking. It was easier to create a new language that could feel modern, even while adapting to old C conventions like enumerations and structures, and Objective-C's various idiosyncrasies.

In fact, Apple did try adopting Ruby and Python once before, in Mac OS X 10.5, which offered first class support for using these languages with Cocoa, the predecessor to iOS' Cocoa Touch. But it didn't take off (nor did support for Cocoa-Java, offered way back in OS X 10.0), so it was retired. You can't say they didn't try.

Swift is:

- *Compiled*, meaning that we perform an up-front conversion of source to bytecode when we build our project, rather than interpreting the source at runtime as with many scripting languages like JavaScript or Ruby.
- *Strongly-typed*, meaning objects are clearly identified as strings, floats, integers, and so on. Types are *static*, meaning they can't change, and these two traits help the compiler build safer and faster code. However, the compiler also uses *type inference*, so we don't have to explicitly indicate a type when the compiler can figure it out unambiguously for us.
- *Automatically reference-counted*, meaning that every object keeps track of how many other objects own references to it, and immediately frees the memory of any object that has zero incoming references (since this object can no longer be used, given that no other object knows about it anymore). This is different from *garbage collection*, popular in many modern languages (Java, C#, most scripting languages), in which a distinct system (the garbage collector) scans memory for objects it can free. Automatic Reference Counting (ARC) is implemented by the compiler and makes

objects themselves responsible for freeing their memory *immediately* when they are no longer needed.

- *Name-spaced*, like many modern languages but significantly unlike Objective-C. This makes it easier for our code to co-exist with others, because it's very unlikely our class and method names will get confused with those provided by Apple or other third parties. For example, the compiler and runtime won't mistake our ViewController class for some other ViewController, because our project settings indicate that our class is part of the com.pragprog.yourname namespace, and not some com.apple namespace. Since Objective-C couldn't do this, we used to have to put two- or three-letter prefixes on all our class names; this explains why all the Apple classes we'll see in this chapter start with NS or UI — for compatibility reasons, we're stuck with the old Objective-C names.

Swift Classes

We've already got some Swift code in our project, courtesy of the Xcode template we started with. Our project starts with two files, AppDelegate.swift and ViewController.swift. We did a little work in the ViewController in the last chapter, so let's start there. After the import that tells the compiler to use UIKit (the user interface framework we'll discuss later in this chapter), the class is defined like this:

```
class ViewController: UIViewController {
```

This says that we are creating a class called ViewController, as a subclass of the UIViewController exposed to us by the UIKit framework, meaning we inherit all the functionality of UIViewController, and all of its superclasses. The line ends an open curly-brace; any functions and properties between this and the matching closing curly-brace are part of the class.

Let's look at the corresponding line at the top of AppDelegate.swift:

```
class AppDelegate: UIResponder, UIApplicationDelegate {
```

What's going on here… does AppDelegate have two superclasses? No, only one, the UIResponder. The second term after the colon, UIApplicationDelegate, is a *protocol*, which is just a declared list of methods. The idea is that the UIAppDelegate protocol describes methods that can be called to tell us that the app is being backgrounded, is being launched via a URL from another app, etc., and our code can implement one or more of these methods to act on that call. We'll do some of those things much later, in Chapter 13, *Launching, Backgrounding, and Exensions*, on page 219.

Swift methods

In the previous chapter, we control-dragged in the storyboard to create an action connecting a tap on our button to a method in the file ViewController.swift. The method has the following signature:

```
@IBAction func handleTweetButtonTapped(sender : UIButton);
```

Let's break this down into parts:

- @IBAction is an *attribute* indicating that the GUI-builder tool, Interface Builder, can create or change connections to this method. If we wanted to drag a second kind of event to a method in this class, only methods with the @IBAction attribute could receive the drop. There are lots of other attributes, such as the override on some of the methods that Xcode already included in the file. We'll see other attributes that affect our methods' behaviors later.
- func indicates that we are declaring a *function*, a self-contained chunk of code with some distinct purpose. Inside a class (or an enumeration or structure), a function is a *method*. Methods operate on object instances by default; to create a *type method*, which is called on the class as a whole, we would use class func instead.
- handleTweetButtonTapped is the name of the method, followed by its parameters in parentheses. This method takes one parameter, sender, of type UIButton. A method that takes no parameters would have empty parentheses, while methods taking multiple parameters would separate each name-type pair with commas. For example, if we had asked Interface Builder to send us both the sender and the event for the button tap, the argument list would be (sender : UIButton, event : UIEvent).
- After the parameters, we indicate the return type, in the form -> *type*. Since this method does not return a value, that part of the declaration is absent (optionally, we could also write -> Void to indicate that there is no return value).

Let's look again at the entire method body to see how we called methods.

Programming/PragmaticTweets-3-1/PragmaticTweets/ViewController.swift

```
Line 1 @IBAction func handleTweetButtonTapped(sender : UIButton) {
   2   if SLComposeViewController.isAvailableForServiceType(SLServiceTypeTwitter) {
   3     let tweetVC = SLComposeViewController (forServiceType: SLServiceTypeTwitter)
   4     tweetVC.setInitialText(
   5       "I just finished the first project in iOS 8 SDK Development. #pragsios8")
   6     self.presentViewController(tweetVC, animated: true, completion: nil)
   7   } else {
```

```
8     println ("Can't send tweet")
9   }
10 }
```

Our first method call is on line 2, in which we ask whether we can even send tweets at all. This is a type method, called on the SLComposeViewController class rather than any specific instance. Indeed, we need to get a true response here to even bother creating an instance.

On line 3, we create an instance of SLComposeViewController by calling its *initializer*. For this particular class, initializing requires passing in a parameter, which is indicated by the label forServiceType:, and the value SLServiceTypeTwitter, which is a constant provided by the Social framework.

This initializer returns a new object of type SLComposeViewController. We assign it to a local variable called tweetVC. Declaring the local variable requires one of two keywords: let, if the value will not change, or var if will. We're not going to change the value of tweetVC within this short method, so using let allows the compiler to treat it as a constant, which can make for faster code.

Notice there's no semicolon at the end of line 3! Semicolons are only needed in Swift for multiple statements on the same line. It doesn't hurt to have them, but they're not needed, so out they go, much as our muscle-memory from C, Java, and other languages insists otherwise.

Lines 4-5 call setInitialText() on the tweetVC instance to pre-populate the Tweet composer UI with the given string. We've split this into two lines solely for the book's formatting; you can write it all on one line if you want.

Line 6 tells self, the object we're currently in, to call the presentViewController() method (inherited from the UIViewController superclass) to show the tweet composer. Look at how this takes three parameters, and the second and third are labelled. The first is effectively named by the method name, so we know it's a view controller. The second and third parameters are labelled as animated: and completion:, respectively. These labels are *mandatory*, and make the code more readable by making it clear what each parameter does: one indicates whether the appearance of the tweet composer is animated, and the second provides any completion code to be run when the composer is done. It may feel like extra typing, but it really isn't when we have auto-complete, and it's arguably preferable to C functions that have a half-dozen or more parameters that we can't easily tell the purpose of.

Finally, look at line 8 and notice what's missing: there's no object associated with the call to println()! Instead of being an instance method called on an object or a type method called on a class, this is just a function, which we can call

from *anywhere*. In Swift, the println() method logs text to the output console, similar to print() in PHP or System.out.println() in Java.

So, with just a few lines of Swift code, we handle the button tap, determine if we're configured to tweet, and either set up and present the UI for that, or log an error message if we can't. If this were C or older versions of Objective-C, we would have to worry about cleaning up the memory allocated to tweetVC at the end of the method, but in Swift with its Automatic Reference Counting, that's not our problem. Instead, we can go on to adding more functionality to our app!

Managing an Object's Properties

Now that we've gone back and figured out how our object allocation and method calls work in the last chapter's code, let's put this knowledge to work and add some new functionality to the app. Our original app lets us send a tweet, but there's no way to tell if we were successful. We'll gradually improve that throughout the next few chapters. For starters, let's use iOS's built-in web browser to bring up our Twitter page inside the app.

Adding a UIWebView

Select Main.storyboard to bring up the UI in Interface Builder. We're going to add a "reload" button at the top and a web view (a subview that renders web content) to fill up most of the bottom of our view. While we're at it, we can get rid of the "I finished the first project" label; having a second button named "Show My Tweets", an active verb, should provide enough context for users to know that these are both buttons.

Reworking GUIs in autolayout can be tricky, so let's go through the steps carefully. Select the label and press the backspace key or use the Cut or Delete menu items. Before we add our new button, select the "Send Tweet" button and look at its constraints. The centering constraint is now orange because the surviving button's layout is now under-constrained: it depended on the distance to the label above it to know where it should go vertically. We'll have to fix that.

Using the Object library (^⌥⌘3) at the bottom right, drag a new button above the existing one, and give it the title "Show My Tweets". Drag it until the center guide appears. Now drag it towards the top of the view until the top margin guide — another dashed blue line — appears. Click the autolayout "Align" button at the bottom of IB and use its popover to add a "Horizontal Center in Container" constraint. Then click the "Pin" button and add a con-

straint pinning the distance to the "Top Layout Guide" as 0, which should be the value that pops up automatically because we dragged up to the top margin.

That's enough to fully specify the new button's constraints, but we still have our old "Send Tweet" button. Drag it up or down until a horizontal dashed appears between it and the "Show My Tweets" button. Use the autolayout "Pin" button's popover to pin a distance from this button to "Button - Show My Tweets", at either the current distance or "Standard". This should turn the bottom button's constraints blue, indicating it is now adequately constrained.

Now we're ready for the web view that will show our tweets. Drag out a web view— as seen in the figure below, its icon in the Object library resembles the Safari app icon —and put it on the bottom portion of the view. Use its handles to drag the bottom and sides of the web view all the way to the bottom and sides of the parent view, and drag the top until a horizontal guide appears between it and the "Send Tweet" button. It may be easier to set the web view all the way at the bottom first, then fix the sides, and then drag up.

Figure 26—Web View icon in Object Library

We want this view to always fill the entire width of the screen, always stay at the bottom, and always respect the distance to the "Send Tweet" button, so we will need four constraints, all from the "Pin" button:

- 0 distance to the left and right sides of the parent "View"
- 0 distance to the "Bottom Layout Guide"
- "Standard" (or the current value, 8) distance to "Button - Send Tweet"

Click "Add 4 constraints" and the web view will be properly constrained for autolayout. It should look like the figure below:.

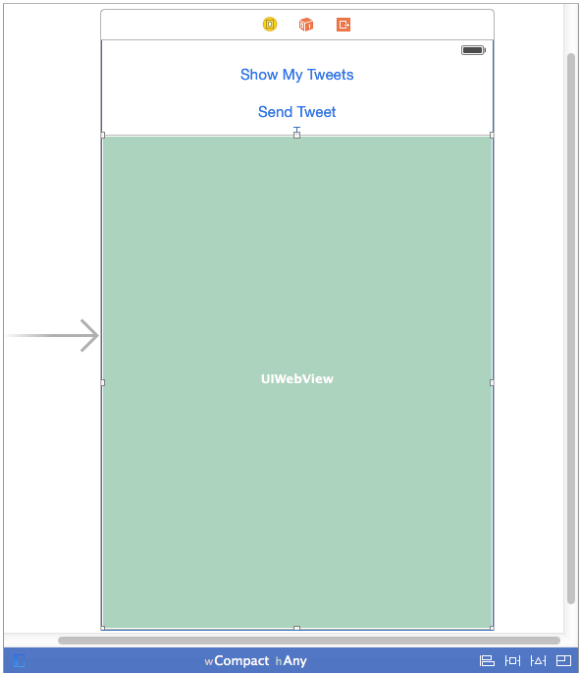


Figure 27—Testing portrait layout of UIWebView with compact width and any height

It's also possible to verify that our layout will work in landscape, too. From the blue sizing strip, select a rectangle that is one box tall and two boxes wide. The popover window describes this as "any width | compact height", and says it is for iPhones in landscape orientation. The layout should now look like the figure below. Notice that both buttons maintain their expected spacing from the top, the web view, and each other. We don't have a lot of vertical space to work with in landscape, but for now, the design is holding up.

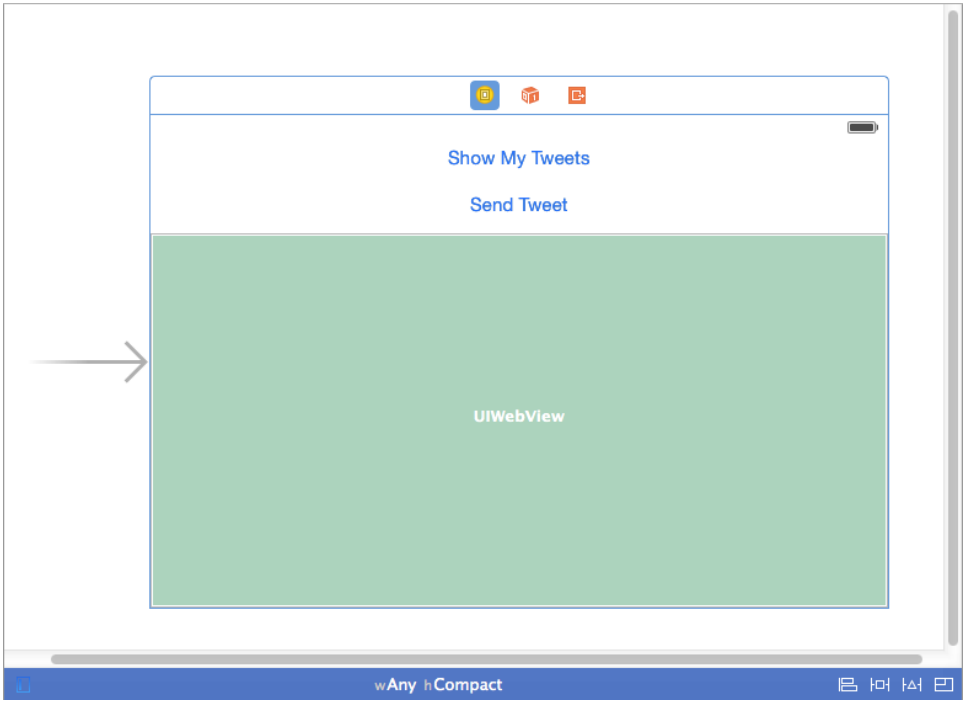


Figure 28—Testing landscape layout of UIWebView with any width and compact height

Connecting the UIWebView to code

Now let's get back to our original goal of showing tweets in the web view. For this to work, we need to write another event-handler method, one that handles a tap on "Show my tweets" That method will need to load the user's Twitter page in the web view. But wait: How do we make a call from this event-handler method into the web view we just created in the storyboard?

In the previous chapter, we talked about outlets, which are used to connect objects in our code to objects in the storyboard. In Swift, those objects are called *properties*. Preceding a property with the `@IBOutlet` modifier tells Interface Builder that a property can serve as an outlet.

Declaring Properties

So that begs the question of how to declare properties in Swift. But it turns out, we've already done that: in their simplest form, properties are just variables or constants within a class. So we can declare a property as simply as this:

```
class MyClass: NSObject {
```

```
  var myVariableInt: Int
  let myConstantString: String
  // methods or other class contents
}
```

In this example myVariableInt and myConstantString are properties of a hypothetical MyClass. These are different from the tweetVC variable we used earlier, since that existed only inside the scope of a method, while these are defined on the class as a whole.

Notice that in these declarations, we explicitly indicate the type of each property, by providing a colon and then the type of the property. We actually could have done that with tweetVC, but we didn't have to because of *type inference*: the ability of the compiler to figure out the type of the variable on the left side of an assignment based on the type returned by the right side. When we declare the property, we don't necessarily have a value to assign to it, so we need to declare the type we will eventually be using.

It's important to remember that in Swift, *properties aren't variables.* To understand this, let's make a distinction: the properties we declare here are called *stored properties*, which means that there is indeed a variable to store the value, but it's managed by Swift and we can't access it directly. Callers will get and set the property through getter and setter methods that Swift sets up for us, and this introduces another kind of property. A *computed property* is one that doesn't have a backing variable, but instead has get() and set() methods that we provide to compute the property's value on the fly. For now, though, all we need are the more conventional stored properties.

We're ready to create our first property, and to make things easy, we're going to let Xcode do it for us.

Creating IBOutlet properties in the storyboard

Select Main.storyboard and switch back to Assistant Editor (via the "tuxedo" button on the toolbar, ⌥⌘↩, or the "View" menu). To make room for the split view, we may want to hide the Utility Area on the right. This will show the storyboard on the left, and ViewController.swift on the right; if this isn't the case, check the ribbon above the right pane and make sure it's set to "Automatic", which picks the most appropriate counterpart file in the right pane given the selection on the left.

To create an outlet property, we do the same thing we did to create the action method for our button: control-drag from the storyboard into the code. Start a control-drag from the web view in the storyboard and drag over to the source code in the right pane, as shown in the following figure.

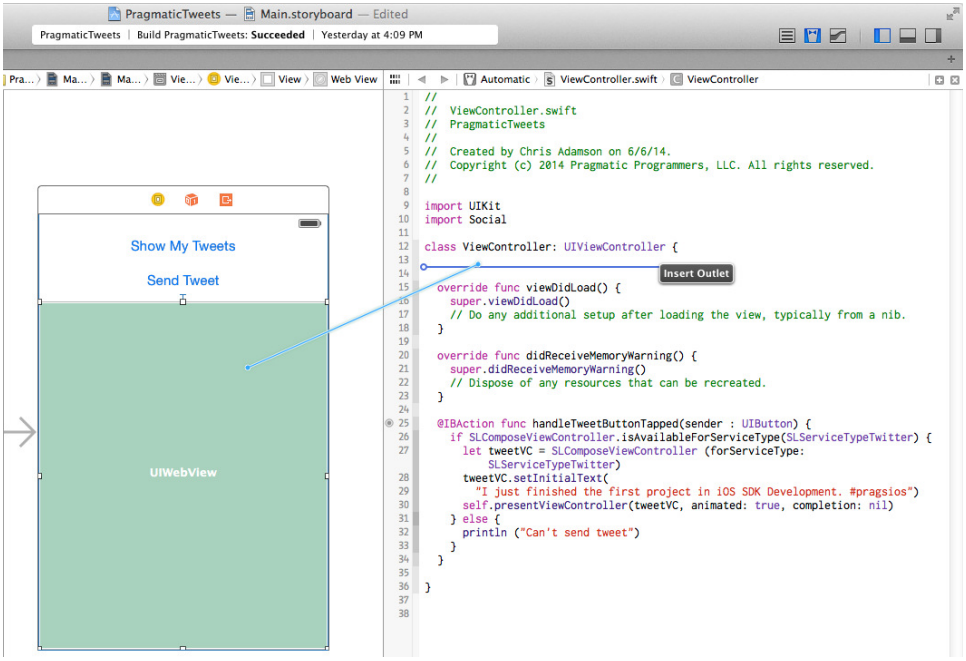


Figure 29—Control-dragging to create an IBOutlet property to a UIWebView

Drag over different parts of the source file without releasing the button; notice that the tip "Insert Outlet" only appears when our drop target is inside the curly brace that defines the class, and not within a method inside the class. Anywhere in here, but ideally in the whitespace just below the class declaration, release to end the drag. Xcode shows a popover to specify the outlet, much like it did when we created the action in the last chapter. Make sure the "Connection" says "Outlet", the "Storage" says "Weak", and give it the name twitterWebView. When we click "connect", the following declaration is inserted into the source at our drop point:

Programming/PragmaticTweets-3-1/PragmaticTweets/ViewController.swift

```
@IBOutlet var twitterWebView : UIWebView!
```

This line is a lot like the hypothetical instance variable mentioned earlier: it declares the attribute IBOutlet (which lets us connect to it with Interface Builder), the var keyword to indicate the property's value can change, the name twitterWebView, and class UIWebView. It also has an exclamation point character (!), which we'll explain in a little bit.

So now we have a property called twitterWebView. To access the property, we use dot notation, similar to calling a Swift method, just without arguments

in parentheses. That means we can read the property with a call of the form object.property and set it with object.property = value. Since twitterWebView is a property of ViewController, within the class we'll refer to it as self.twitterWebView. For properties that themselves have properties, we just chain dot-operators. For example, UIWebView has a canGoBack property, so our view controller class can test this with self.twitterWebView.canGoBack.

Calling into the UIWebView property

Now that we've synthesized the twitterWebView property, we're ready to use it in our code. We'll write an event handler for "Show my tweets" that loads the user's Twitter page into the web view. How do we do that? Well, if we look up the UIWebView in the documentation viewer, the docs tells us that UIWebView has a loadRequest() method that we can use, provided we use a string to create an NSURL (which we assume to be an object that represents a URL), and from that create an NSURLRequest.

But let's start with getting the button-tap event in the first place. Select Storyboard.main and again switch to the Assistant Editor. Make sure the right pane shows ViewController.swift. Control-drag from the "Show My Tweets" to anywhere inside the class' curly braces, so long as it's not within an existing method's curly braces. When the drag passes over a viable area of the source file, the drag point will show the popup tip "Insert Outlet or Action", which is what we want to do.

End the drag and fill in the popover, like we did in the previous chapter for "Send Tweet". Change the "Connection" to "Action", enter handleShowMyTweetsTapped for the method name, and change the type from AnyObject to UIButton. Leave the defaults for event ("Touch Up Inside") and "Arguments" ("Sender"). Click "Connect", and Xcode will stub out a method for us:

Programming/PragmaticTweets-3-1/PragmaticTweets/ViewController.swift

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

Switch back to Standard Editor mode and select ViewController.swift. The method that Xcode built for us with the drag says @IBAction which just means that Interface Builder, the storyboard editor, can work with it. It takes one parameter, sender, which is the UIButton that sent the event (that is to say, the button that was tapped). There's no return type stated, so the method doesn't return a value.

We sketched out a plan to implement this method above: we just have to work up a call to the UIWebView's loadRequest() method. Fill in the method like this:

Programming/PragmaticTweets-3-1/PragmaticTweets/ViewController.swift

```
Line 1 @IBAction func handleShowMyTweetsButtonTapped(sender : UIButton) {
   2   let url = NSURL (string:"http://www.twitter.com/pragprog")
   3   let urlRequest = NSURLRequest (URL: url)
   4   self.twitterWebView.loadRequest(urlRequest)
   5 }
```

On line 2, we create an NSURL, from its initializer that takes an argument called string:. We've used http://www.twitter.com/pragprog here, but feel free to put in your own Twitter user name. Next, line 3 takes this NSURL and makes a new NSURLRequest from it. That's the object we need on line 4 to tell the URL to load up that page, by using its loadRequest() method.

We wrote this example in a verbose way, so it's easier to follow. As with most languages, Swift lets us chain the results of one statement as parameters to another, so we could have written this in a more terse form, like this:

Programming/PragmaticTweets-3-1/PragmaticTweets/ViewController.swift

```
@IBAction func handleShowMyTweetsButtonTapped(sender : UIButton) {
   self.twitterWebView.loadRequest(NSURLRequest (URL:
      NSURL (string:
        "http://www.twitter.com/pragprog")))
  }
```

Either approach is fine, of course. Also, the line breaks in the terse version are only present to accomodate the book's formatting; this could all be written on one line in Xcode.

Now that we've written this simple method, we're ready to go, so click the Run button to launch the updated app in the Simulator, and then click the "Show my tweets" button. The event sent by the new button goes to the handleShowMyTweetsTapped() method, and its code makes a reference to the self.twitterWebView property to load up the Twitter page in the web view, as shown in the following figure:

Figure 30—Programmatically populating a UIWebView

Note that since the `UIWebView` is a real live web client, it acts just like Safari, so the first time we use it, we might get intercepted by an advertising page asking us to download the Twitter app for iOS, or worse yet a redirect; just look for a "close" button or link to dismiss it, as we would do in any other browser. Still, real live web browser component with just a little clicky-draggy and a couple lines of code…*not bad!*

The iOS Programming Stack

Now we're rolling: We can visually create automatically-resizing GUIs in the Storyboard, connect them to methods and properties in the view controller class that owns the view, and write code in Swift to do stuff. Life is good.

Except that we're still taking a lot on faith when it comes to actually calling stuff in our code. We can search the documentation for cool-looking methods

all day, but first we should make sure we understand where all these classes are coming from and how they're organized.

The iOS SDK divides its functionality into a set of frameworks. We saw this in the last chapter when we inspected the project's build phases and added Social.framework to the frameworks that were already part of the project. Conceptually, we can divide the SDK's frameworks into four layers:

Cocoa Touch Layer The top-level abstractions over applications and their UIs (UIKit) and integration with system-provided UI features like mapping (MapKit), and Notification Center.

Media Layer Graphics, sound, and video frameworks.

Core Services Frameworks for essential, non-UI functionality, like file-system access, in-app purchase (StoreKit), health-tracking device integration (HealthKit), and so on.

Core OS Low-level frameworks and libraries needed by the upper layers, including the BSD libraries that are the core of iOS and Mac OS X.

In this book, we will spend most of our time working with the frameworks that are included by default in the Xcode project templates: Foundation and UIKit.

Building Views with UIKit

The UIKit framework provides the building blocks of touch-based applications for iOS. That means it's responsible both for the concept of what an app *is* and how it interacts with the rest of the system, as well as for providing a suite of user-interface views. Every user interface control we add to the app comes from UIKit, as well as the systems for sending user interface events to our code, how we draw things, fonts, colors, gestures, etc.

UIKit's UIApplication class is the point of contact between our code and the rest of the system. By accessing its sharedApplication() method, we can open other apps by URL, receive remote events from Apple's Push Notification service, and set a number for our app icon's badge. But a lot of apps don't do any of these things, so we don't often use UIApplication directly. Instead, the Xcode template sets up a UIApplicationDelegate class for us to customize; this class gets callbacks when common events occur, like the app being started up or opened via a URL from another app, or when it's sent to the background by the user tapping the home button.

The *delegate pattern* is frequently used in the iOS SDK, often as an alternative to subclassing. The idea is that for certain responsibilities, usually the custom behaviors specific to an app, an object can delegate its behaviors to another object. In this case, the UIApplication class handles the activities that are common to all applications, but for cases where different apps will want to do different things, it makes callbacks to our AppDelegate. Delegates don't need to be their own classes like this: they are often classes with other purposes that just implement one or two methods (usually collected as a protocol) in order to serve as a delegate.

As for the app delegate itself, we'll be revisiting it in Chapter 13, *Launching, Backgrounding, and Exensions*, on page 219, when we look into what apps can do in iOS 8 even when they're not running.

Views

Many of the UIKit classes are views, which are the onscreen touch objects in our user interface. We've been using these in our Twitter example: our UI has a single view that fills the screen and has three subviews: two buttons and the web view. There are many other view classes available, like switches, tables, and sliders.

The top-level UIView defines the common functionality of all views. All views have visual properties, such as a backgroundColor, an alpha variable, and hidden and opaque flags. As we've already seen, a view can contain other views; these are accessible via a subviews property and can be added with convenience methods like insertSubview(). A child view can access whatever view it's a subview of via the superview property. Subviews are layered on top of one another by drawing them in the order of the subviews array, with the view at index 0 at the bottom, then index 1 on top of it, and so on. For visual styling needs, UIView also has a tintColor property that applies to all subviews, which makes it easier to apply custom theming to all the UI components on the screen.

Views also have frame and bounds properties that indicate their size and location. Each of these properties is a CGRect, a structure that defines an *x-y* origin (of type CGPoint, inherited from the *Core Graphics* framework) and a width-by-height size (of type CGSize, another structure). The difference is that the bounds values are in the view's own coordinate system, while the frame is in its superview's coordinate system. So a subview's origin is its top left corner, relative to its parent's top left corner at (0,0). Setting either property changes the other as needed, and these interact with two related visual properties, transform and center.

Swift Structures

Notice that the CGRect used for a view's frame is not a class. Classes typically have both state, expressed as properties, and functionality, expressed as methods. What if we only care about the state? For this, Swift provides *structures*, a very straightforward layer atop the C struct. Structures are useful in cases where we just want to pass a related set of values around.

We saw the CGRect back when we while messing around in the playground, manually setting the frame of our UILabel. Here's what that struct really looks like:

```
struct CGRect {
  var origin: CGPoint
  var size: CGSize
}
```

The CGPoint and CGSize are also structures, the first having an x and y, the second a width and height.

The other thing that's important to know about structures is that when we pass them to a function or method, Swift uses *pass-by-value* semantics, rather than *pass-by-reference* as it would with an object. This means the recipient is getting a copy of the values, and changing them inside the method won't change them anywhere else. It's more like receiving an Int as a parameter than receiving an object.

Along with views, UIKit provides the UIViewController class, which is meant as the place where we put the logic for our user interfaces. The view controller also has a number of life-cycle callbacks, telling it when its view is loaded from the storyboard and when the view will appear or disappear as a result of navigating to different parts of the app. We will look more at this relationship in *the (as yet) unwritten chp.viewcontrollers* **I don't know how to generate a cross reference to chp.viewcontrollers.**

Finally, UIKit provides classes for objects that are commonly needed by user interfaces, such as UIFont and UIImage. Taken together, the UIKit classes provide an extensive and extensible user interface toolkit.

Accessibility in UIKit

UIKit offers deep support for accessibility, the ability of a user interface to adapt to a user's needs, such as limitations in vision, hearing, and touch. Every UIView has accessibilityLabel and accessibilityHint attributes, along with accessibilityTraits that describe the view's behavior, that the system can use to better expose it to users who need help. For example, blind users can turn on the Voice Over feature to have the iOS speech synthesizer speak the names of UI elements, using the provided accessibility values if they have been set. These attributes can all be customized in the storyboard or in code.

Unfortunately, many developers don't customize their UIs for accessibility. The good news is, they often don't need to: the default behavior of iOS makes typical UIKit applications highly accessible. But it's good karma—and a legal requirement in some cases—to test the accessibility of our apps and customize these accessibility properties as necessary. And if we were to create our own views, we would have to implement these attributes on our own, so the system would know how to present our custom view to a disabled user.

Strings

The other major framework we import by default in iOS projects is *Foundation*, which provides fundamental data types for common concerns like dates and times, regular expressions, file I/O, and so on. We've already used two classes from Foundation: the NSURL and NSURLRequest that we used to populate the UIWebView.

Foundation also provides strings (as the NSString class) and collections (NSArray and NSDictionary), and in Objective-C, we would work with these as we would with any other class. However, in Swift, the language has taken more responsibility for strings and collections, and we can do a lot of common tasks without having to explicitly call methods on object instances. This makes Swift a lot easier to just get in and use.

Let's start with the String. We can create a string as we do with any other variable or constant, assigning a value with var or let. The value to assign on the right side of the equals is enclosed in straight quotes, and can include any Unicode characters.

```
let myConstantString = "iPhone"
var myVariableString = "iPad"
```

We can also build strings by using the concatenation operator, +:

```
var shoppingList = "I need to buy an " +
        myConstantString + " and an " + myVariableString
```

If the string was defined with the var keyword, it's mutable, so we can append it in place with the += operator:

```
shoppingList += ", and maybe an Apple TV"
```

It's also possible to build up strings by performing in-line evaluations of expressions in the form \(*expression*), like this:

```
var shoppingListCountString = "This list has \(1 + 1 + 1) items"
```

One rather surprising fact about Swift strings is that that they are *pass-by value*, rather than *pass-by reference*. This means that a method that takes a string as an argument gets the contents of the string, rather than a reference to a string object. It's a subtle difference but it means, among other things, that a function or method can count on the value of a string not being changed by code running at the same time and with a shared reference to the string object. In Objective-C, developers typically copied strings they received from callers just to prevent such problems. It's another way that Swift eliminates entire categories of subtle bugs.

Collections

As with strings, Swift moves support for the most common collections directly into the language, which makes them significantly easier to work with than the Objective-C approach of calling methods on objects.

Arrays

Swift provides direct language support for two essential collections: *arrays* and *dictionaries*. Arrays, as in most languages, are ordered lists of objects.

```
var musicGenres = ["Pop", "Rock", "Jazz", "Hip-hop", "Classical"]
```

Because of type inference, Swift knows this is an array of Strings. We could make that explicit by making the declaration `var musicGenres: String[]`.

We can access array members by a zero-based index, inside square braces. We can also get a sub-array by using the *range operator*, where `..<` includes the first index but not the second, and `...` includes both the first and last index.

```
let pop = musicGenres[0]
let popRock = musicGenres[0..<2]
let popRockJazz = musicGenres [0...2]
```

If the array was declared with the `var` keyword, then it is mutable and we can add to it with `+=`, or change values in-place.

```
musicGenres += "J-Pop"
musicGenres[1] = "Rock and Roll"
```

Arrays also have several self-descriptive methods for mutating their contents, such as `insert()`, `removeAtIndex()`, `removeFirst()` and `removeLast()`.

Special Types

What do you suppose happens if we try to add something that isn't a String to `musicGenres`? The following produces a build error:

```
musicGenres = 2.99
```

Since the original contents of musicGenres were all strings, Swift inferred that it was an array of Strings, effectively making the declaration var musicGenres: Array<String>, where the type to the right of the colon explicitly declares what musicGenres is: an Array of Strings. Since 2.99 is a Double, it can't be added. If we really needed to do something like this, we could instead use the special type Any in our declaration, like this:

```
var musicGenres2 : Array<Any> = ["Pop", "Rock", "Jazz", "Hip-hop", "Classical"]
musicGenres2 += 2.99
```

Any supports, well, pretty much anything: strings, numeric types, objects, etc. If we know we're dealing solely in objects — keeping in mind that String and collections are *not* objects in Swift — then we could use the somewhat more restrictive type AnyObject.

Dictionaries

Swift also provides *Dictionaries*, which map from one object to another. These are commonly used in "lookup"-style scenarios. As with arrays, we can create a dictionary by assigning some values into it (by putting *key : value* pairs in square braces like an array, each pair separated by commas), and let Swift figure out the types.

```
var planetaryMass = [
        "Mercury"       : 3.301E+23,
        "Venus"         : 4.867E+24,
        "Earth"         : 5.972E+24,
        "Mars"          : 6.417E+23,
        "Jupiter"       : 1.899E+27,
        "Saturn"        : 5.685E+26,
        "Uranus"        : 8.682E+25,
        "Neptune"       : 1.024E+26,
]
```

In this example, Swift will infer the declaration var planetaryMass :Dictionary <String, Double> (although we gave it help by using scientific notation for the large numeric values, without which it might have inferred the value type to be Int).

As with arrays, we use square braces to access members of the dictionary by name. The simplest use of this syntax is to add a member to the dictionary.

```
planetaryMass["Pluto"] = 1.471E+22
```

Of course, Pluto isn't a planet anymore, so this example is purely hypothetical. Anyways, we can also use square braces to look up a value by its key… or can we? Consider the following:

```
println ("Earth's mass is \(planetaryMass["Earth"]) kg")
```

This code won't even compile. But why not? The answer is a little tricky…

Optionals

To see what the problem is when we fetch a dictionary member by name, let's imagine if we added the following line:

```
var mass = planetaryMass["Gallifrey"]
```

Considering that "Gallifrey" is a fictional planet (and was, for a time, erased from history even within that fiction), it's not in our dictionary, so there's no valid answer here. So what value should be returned for a value that doesn't exist? Double is a numeric type, not an object, so we can't just have it be nil as a means of saying "no object" Can it be 0? No, 0 is a perfectly good value for a floating-point number. So what do we do here?

Swift introduces a new concept called *optionals*, the idea of which is to encapsulate both knowing whether or not there even *is* a value, and if so, what the value is. Dictionaries return optionals, which allows planetaryMass to just return nil when there is no value for a key.

We make a type into an optional by adding a ? character to the type. Then we expose the optional to an if statement; if the optional has no value, this will evaluate to false. So here's a safe way to print a value from the dictionary:

```
let mass : Double? = planetaryMass["Earth"]
if mass != nil {
        println ("Earth's mass is \(mass) kg")
} else {
        println ("No such planet")
}
```

The only problem with this is that, well, optionals can be a little burdensome. In this case, the println() output is:

```
Earth's mass is Optional(5.972e+24) kg
```

Ick. The println() puts that "Optional" stuff around the value. How do we get rid of it? Swift gives us an expedient way to work around cases like this: We can try to assign an optional to its non-optional base type inside an if statement, and the if evaluates to true or false based on whether the assignment works:

```
if let unwrappedMass : Double = planetaryMass["Earth"] {
        println ("Earth's mass is \(unwrappedMass) kg")
} else {
        println ("No such planet")
```

```
}
```

This prints Earth's mass is 5.972e+24 kg, without the Optional(...) stuff, because unwrappedMass is a real Double, and not a Double? (that is to say, a Double optional).

The as Operator

Swift also provides the as operator for casting between classes, and as? for an optional cast that may or may not succeed. So the following two if statements are equally valid ways to unwrap an optional:

```
if let unwrappedMass : Double = planetaryMass["Earth"]
if let unwrappedMass = planetaryMass["Earth"] as? Double
```

Fast-Unwrapping Optionals

Converting an optional to its base type is called *unwrapping*. We can do this carefully, as with the if let... construction. But that can be burdensome if we have to nest a bunch of if statements, just to get at the underlying types of some optional variables or properties.

For cases where we "just know" that the value isn't nil, we can accelerate the unwrapping with the ! operator. So in our earlier example of getting Earth's mass, we can unwrap the optional within the println(), without an if test, by using the ! operator.

```
let optionalMass : Double? = planetaryMass["Earth"]
if optionalMass != nil {
        println ("Earth's mass is \(optionalMass!) kg")
} else {
        println ("No such planet")
}
```

This unwraps optionalMass within the println(), so we don't get the Optional(...) junk in our output.

Doing a fast unwrap is great, but the problem with the ! operator is that whole part about assuming the optional isn't nil. If we were to foolishly write the else case like this:

```
println ("Failed to get planet's mass: \(optionalMass!)")
```

Then we would crash on the second line with unexpectedly found nil while unwrapping an Optional value. So much for what we "just know", huh?

We've actually seen the ! character much earlier in this chapter, back when we dragged over a connection to create the twitterWebView property. When we

apply the ! to a type, it becomes an *implicitly unwrapped optional*, meaning we can unwrap its value without the ! operator. In other words, we can just refer to twitterWebView and not have to write twitterWebView! or a bunch of if let unwrappedWebView : UIWebView = twitterWebView code every time we want to touch it. The unwrapping is implicit, hence the name.

Cool, right? But since it's still an optional, there is no guarantee that it even *has* a value. So it's inherently unsafe, and should only be used when we really know the variable or property will always have a value when we reference it. Since Xcode is responsible for connecting our storyboard elements to our code, it can confidently use the implicitly-unwrapped optional instead of the more burdensome optional type.

In fact, we'll see this a lot in the iOS code we're going to call. On http://devforums.apple.com, Apple's engineers have explained that implicitly unwrapped optionals are needed when bridging to old Objective-C code, where nil is always a possible value for Objective-C types, and they couldn't immediately audit every method in all the frameworks to guarantee that a given parameter can *never* be nil. Yet, on the other hand, if every parameter were a full-blown optional, we'd be writing lots of defensive "if not nil" code. So, for now, it's a tradeoff between safety and usability, and most of Apple's methods currently work with implicitly-unwrapped optionals. That said, Apple engineers are indeed auditing the iOS frameworks, and some implicitly-unwrapped optional parameters and return types are changing to either plain types or full-blown optionals with each new release of Xcode. We can plan on this being an ongoing process for a while.

Internationalization

We've talked a lot about the potential uses of strings, collections, and classes in the UIKit and Foundation frameworks, which in turn means we've spent a lot a mental time away from our code. Let's pull together some of their combined strengths to improve our app.

The Swift language, along with the Foundation and UIKit frameworks, do their part to support *internationalization*, the ability of code to adapt to local conventions in different parts of the world. These concerns include things like language, time and date formatting, and currency symbols and separators. If we properly adopt internationalization—often abbreviated as i18n for the first and last letters of the word and the 18 in between—then our Twitter-sender will be as useful to a French-speaking user as an English one. We'll wrap up this chapter by doing exactly that.

Let's identify what parts of our app are currently English-only. When the UI appears, there are two buttons in English, and when the user taps the first button, we fill in the tweet composer with an English message. Both of those need to change.

When we internationalize an app, we create a *localization* for each *locale* we want to support. The locale is a combination of language and geographic region (which lets us distinguish between, say, the variants of Portugese spoken in Brazil and Portugal), though we can omit the region and focus only on language issues for things that are consistent in the language. The localization is a collection of strings, currency formats, graphics, sounds, and other resources that are specific to one locale.

In Xcode, we declare supported localizations at the project level. Choose the PragmaticTweets project icon from the top of the file navigator. This gives us a view of the project as a whole, along with the various targets it builds, such as the app executable and its unit tests. We need to inspect the project itself, so find the small hide/show button in the strip atop the editor area, and use it to show the list of projects and targets on the left side of the editor area. Select the PragmaticTweets project (as opposed to the target of the same name), as in the following figure:

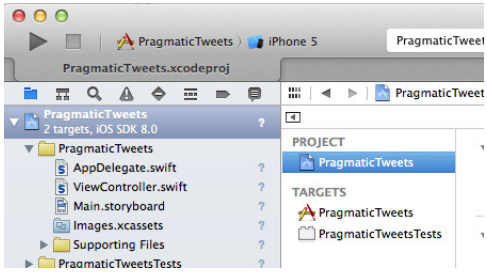


Figure 31—Selecting projects or targets in Editor Area

Click the Info tab at the top, and look down to the "Localizations" section to show the project's current localizations. This should show that for your default language, one file is currently localized. Press the plus (+) button at the bottom of this section to show a list of common locales and choose one. For our screenshots, we'll use French (locale fr)—choose any language you're familiar with. If you don't know another language, one useful technique is to use a made-up language like pig latin or Ubbi Dubbi. Either way, we'll be able to find text that needs internationalization by switching the runtime language and looking to see that all the onscreen text changes.

Once we've picked a language, a sheet slides out showing the files in the project that are internationalizable—currently just Main.storyboard—and asks for a "reference language" for each. The only choice for now is the "Base" language of the project, so just click "Finish." This adds French to the list of project localizations, as the figure shows:

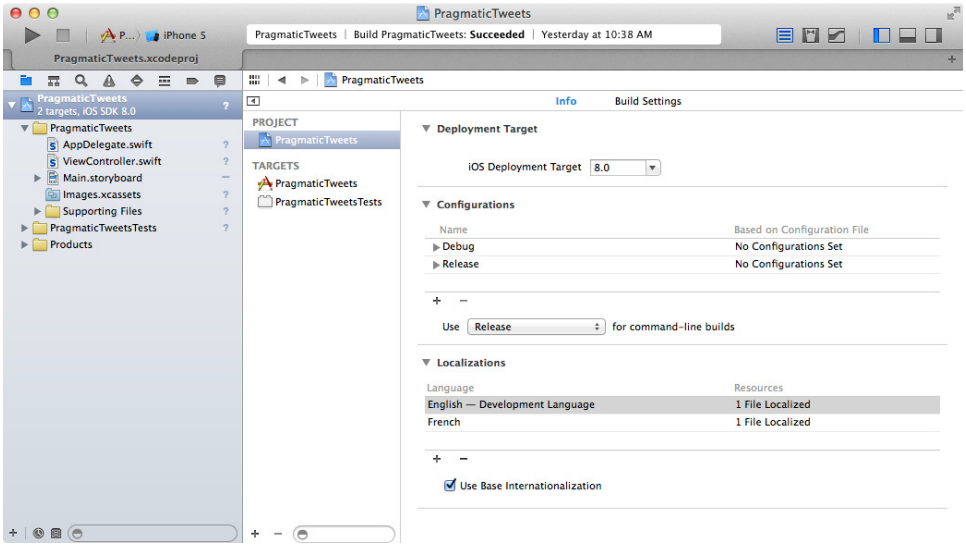


Figure 32—Multiple localizations for an Xcode project

Once a language has been added to the list of localizations, the project navigator shows any files we localized as containers with disclosure triangles, like in the following figure. In the case of Storyboard.main, expanding the disclosure triangle shows the storyboard as laid out in the language we started with (English) is shown as "(Base)", and each localization is a .strings file with a language name, like Main.strings (French). Select this file to show that its contents are just a list of strings to be localized:

```
/* Class = "IBUIButton"; normalTitle = "Send Tweet"; ObjectID = "H7b-6b-5eJ"; */
"H7b-6b-5eJ.normalTitle" = "Send Tweet";

/* Class = "IBUIButton"; normalTitle = "Show My Tweets"; ObjectID = "sbN-Jq-Jl8"; */
"sbN-Jq-Jl8.normalTitle" = "Show My Tweets";
```

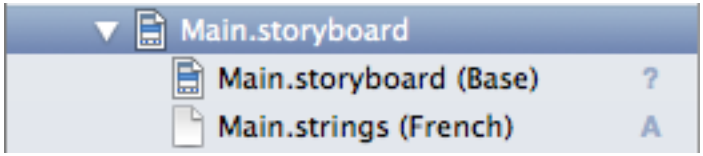


Figure 33—Multiple localizations for a storyboard file

Xcode copies over the English labels for everything textual in the storyboard, so localizing is just a matter of changing these strings. For French, change "Send Tweet" to "Envoyer Tweet" and "Show my tweets" to "Voir mes Tweets."

Does it work? We can actually check it in Xcode. Select `Storyboard.main` and switch to the Assistant Editor. On the right pane, change the jump bar from "Automatic" to "Preview". This shows how the layout will look on different devices, which can be selected via a "plus" button at the bottom left of the pane. On the bottom right, there's a language selection menu that defaults to our development language (English, in our case). Switch this to "French" and you'll see our UI layout as a French-speaking user would, as shown in the next figure.

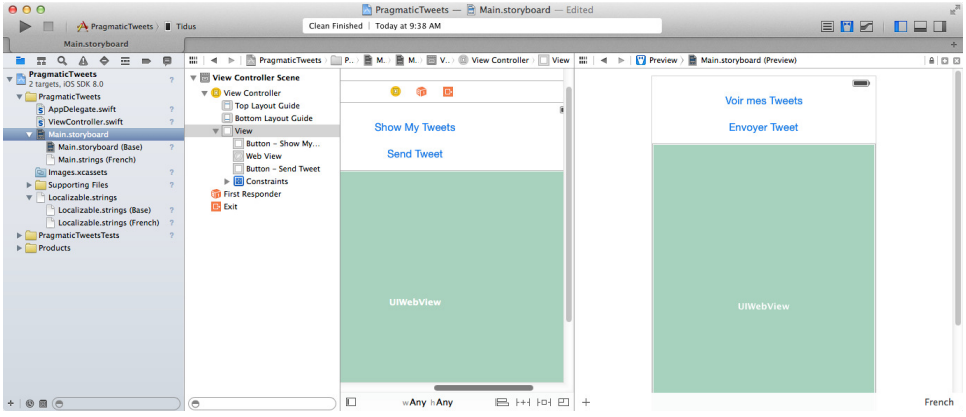


Figure 34—Previewing localized storyboard in Assistant Editor

That internationalizes our user interface…almost. When the user composes a tweet, we still set up the tweet composer with an English string. That's something we do in code, with an `NSString`. So we need a way to internationalize that too.

The solution comes to us from Foundation's `NSBundle` class, but that requires a little explaining. On iOS, everything in our app is stored in a *bundle*, a

directory structure with a known organizational scheme. The app's "main" bundle contains code, artwork, media assets like sound and video, metadata, and localizations, and we get to it by way of the class method mainBundle(). From there, we can use the method localizedStringForKey().

In ViewController.swift, replace the line that assigns tweetVC.message with the following:

Programming/PragmaticTweets-3-2/PragmaticTweets/ViewController.swift

```
let message = NSBundle.mainBundle().localizedStringForKey(
  "I just finished the first project in iOS 8 SDK Development. #pragsios8",
  value: "",
  table: nil)
tweetVC.setInitialText(message)
```

The way this works is pretty surprising the first time you see it. The key we ask for is the string we want, *in the development language.* In other words, we take the English string we had hard-coded before, and ask for a localized string corresponding to that. The variable parameter is what we get back if no match is found; since it's nil, we get back our original key. That's perfectly reasonable behavior: if there's no localized string, we default back to our original language string. Finally, providing nil for the table tells the bundle to look in a file called Localizable.strings, which is the default behavior.

So, having written this, all we need is a Localizable.strings file for French, and we'll be all set.

In the project file, click the Supporting Files group and use the menu command File→New→File… (⌘N). In the dialog, choose the Resource group and select Strings File. When prompted for a file name, we *must* use the name Localizable.strings for Foundation to find our localizations. In the list of Targets at the bottom of the file dialog, make sure "PragmaticTweets" is checked, but "PragmaticTweetsTests" is not.

This creates a Localizable.strings file for our default language, which we don't actually need localizations for. What we need to do now is to add a French localization to the strings file. Select Localizable.strings, show the Utility Area and bring up the File inspector (click the little document-shaped icon, use the View menu, or press ⌥⌘1), and in the Localization section, click the "Localized…" button. This slides in a sheet asking, "Do you want to localize this file?" and offering a pop-up menu of languages. Choose "Base" to create a base localization and click OK. This alone only gives us a baseline version of Localizable.strings, but the "Localized…" button is now replaced by check boxes for all the localizations defined in the project. Click French to add the French localization.

As with the storyboard, the Localizable.strings file will now have a disclosure triangle that exposes baseline and French versions of the file. Edit the French version as follows, with the English string in quotes, an equals sign, a French equivalent, and a semicolon. We've added a line break to suit the book's formatting; this isn't necessary for the file to work.

Programming/PragmaticTweets-3-2/PragmaticTweets/fr.lproj/Localizable.strings

```
"I just finished the first project in iOS 8 SDK Development. #pragsios8" =
"Je viens de terminer le premier projet en iOS 8 SDK Development. #pragsios8";
```

Let's try this out. Using the Simulator, visit the Settings application and switch languages by navigating through General→International→Language and then choosing "Français." Go back to Xcode and run the app in the Simulator. When the app comes up, the buttons will now be in French. Tap "J'ai fini le projet" and the tweet composer will appear, prepopulated with French text, like in the next figure.

Components provided by UIKit should be automatically localized as well—the Cancel and Send buttons become "Annuler" and "Envoyer," respectively (although readers have noted this is less consistent on the iOS Simulator than on actual devices).

From here on out, we won't necessarily call attention to providing translated text for all our storyboard elements, but we will continue to use localizedStringForKey(), because it's an easy enough habit to adopt. And it's straightforward to localize a project when we're ready to do so: localize the storyboard, and add any strings from the application code to Localizable.strings.

Wrap-Up

In the first chapter, we had to take a lot on faith about Swift and the iOS frameworks in order to get started with the tools. In this chapter, we've dug deeply into the language and libraries, revealing how they work and why. We started by learning about the Swift programming language and how it handles variables, classes, and methods. Then we looked at the two most important frameworks in the iOS SDK, UIKit and Foundation, along with Swift's built-in support for strings, arrays and dictionaries, all of which we'll use heavily from here on out. Finally, we brought the power of UIKit and Foundation to bear on a cross-cutting concern, internationalization, using UIKit's localized storyboard loading ability to bring up a locale-appropriate GUI and Foundation's localized string support to let us call up i18n'ed strings at runtime.

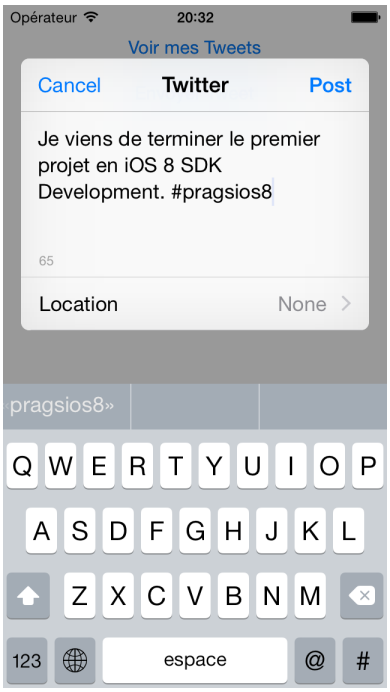


Figure 35—Localized programmatic text

It's great that our app works, but how can we be sure it will keep working? In the next chapter, we'll look at how Xcode supports automated testing, so we can programmatically put our code through its paces.

CHAPTER 4

Testing Apps

We have come a long way in a short time. We've got an app that can send tweets and show our Twitter web page. We now have a stable app that isn't going to crash on us, right?

Well, how do we know that? We have run the app a few times, but have we really pushed the limits of the app? Have we really tried everything that anyone could possibly do to our app? How to we prove that our app is not going to crash before we ship it off to Apple?

And as we start adding features, what proves that those changes work, or that they're not going to have weird side-effects that break the stuff that had been working?

The way we deal with this is to use *unit tests*.

Unit Tests

Unit tests are exactly what they sound like. Unit tests are small, self-contained segments of code that test very small, targeted units of functionality. Rather than check to see if the whole application works, we can break the functionality into pieces to pinpoint exactly where errors and bugs are occurring.

Unit tests are designed to either pass or fail. Is this feature working the way you want it to, yes or no?

The Parable of the Dinosaur

Here is an example of unit testing gone bad.

In Jurassic Park (the book, not the movie), Doctor Grant asks the scientists how they can be sure that the dinosaurs are not breeding.

The scientists assure Doctor Grant that every precaution has been taken. They engineered the dinosaurs to all be female. They had the island blanketed with motion detectors to count each and every dinosaur every five minutes. They created a computer algorithm to check the number and types of dinosaurs found by the motion sensors and the number only changed when a dinosaur died. There had been no escapes. They knew everything happening on the island and they were completely in control.

Doctor Grant asks them to change the parameters of the computer program to look for more dinosaurs than they were expecting to have. The scientist humor Doctor Grant and change the algorithm to search for more dinosaurs. Lo and behold! There are more dinosaurs. After running the program several more times with increasing numbers they eventually discovered there are over 50 extra dinosaurs on the island. Oops!

The program had been set up with the expectation that the number of dinosaurs could only go down, never up. Once the program reached the number of dinosaurs it was expecting to find, it stopped counting and the scientists never knew there was an issue. It anticipated the outcome of dinosaurs dying or escaping the island, but never the possibility that life could find a way.

Reasons We Unit Test

Bugs, like life, do find a way. The first thing to remember in computer programming is that the computer is stupid. The computer only does what you tell it to do. It can't infer what you meant. It is important to verify that you are giving the right directions to the computer and the best way to do that is to test your apps.

One major reason to unit test an application is to eliminate crashes. The single biggest reason that most app submissions are rejected by Apple is because they crash. Even if Apple doesn't catch your crash, users have a talent for finding the one combination of things that will cause your app to crash. These are the users who tend to leave one star reviews on the store, which is something we want to avoid if at all possible.

Unit tests also expose logic errors in our code. In the Jurassic Park example, the code being run had a logic error that prevented the scientists from discovering the problem until it was too late. We don't want that to happen to you.

Writing tests also helps you write your code. Have you ever started writing a piece of code only to figure out that one feature you spent days working on wasn't really going to work out in your project? By thinking critically about what specifically you want your application to do, you can avoid writing

overly complicated and unnecessary code. They can inform the design of our code: what part of the code has what responsibilities, and how we recover if something unexpected happens.

Designing Good Unit Tests

As we will soon discover, writing a unit test is not difficult. Writing a good unit test is another story altogether.

There are generally three types of unit tests:

- *Debugging:* These tests are built around bugs to ensure that when you change the code these bugs do not reappear. Sometimes when we are coding we make changes to the code that affect bugs that we have already resolved. Since we do not want to see that bug again we need to write tests to make sure that the bug has not reappeared when we change anything.
- *Assert Success:* We are testing to make sure you are getting a result you want.
- *Assert Failed:* We are testing to make sure you are not getting a result you don't want.

We might wonder why you would need a test to assert failure. Isn't the point of testing to make sure that features we created work properly?

Think back to the Jurassic Park example. The scientists created tests to make sure they were finding all of the dinosaurs they were looking for. They asserted success once the number of dinosaurs they were looking for was reached.

Sometimes it is as important to write a test that we expect to fail to make sure that we are not getting a result we don't want. Had the scientists also included a failure assertion test they would have discovered that they were getting results that made no sense: There are more dinosaurs in the park than there are supposed to be.

Creating Tests in Xcode

Testing functionality was introduced in Xcode 5. Apple based many of its built-in functions on accepted and open source frameworks and has been working very hard to make testing a vital and useful tool in your developer utility belt.

We are going to go over several aspects of testing in Xcode in this chapter. Since you have spent a great deal of time creating and developing your PragmaticTweets app, let's run it through some tests to see how it works.

Let's direct our attention to the File Navigator, shown in the following figure. There is a group entitled PragmaticTweetsTests. Xcode has conveniently created this group and sample template class, PragmaticTweetsTests.swift, for our first two tests.

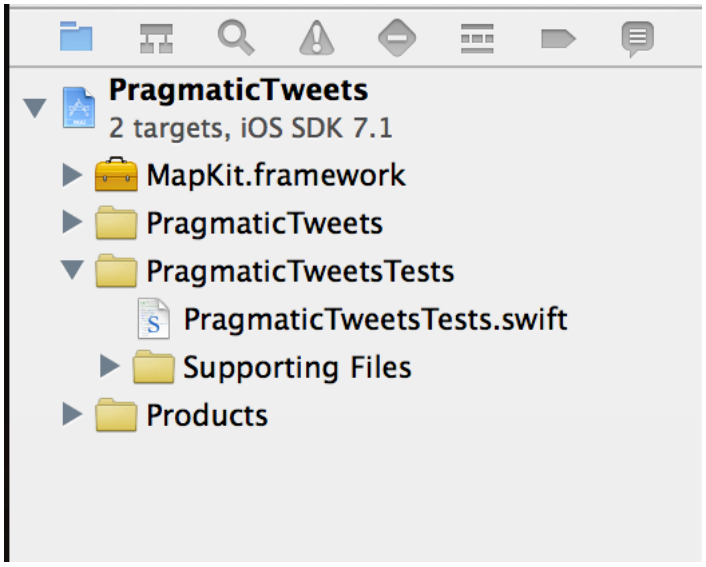


Figure 36—Test classes in Xcode File Navigator

Before we move on to actually looking at the included test file, let's look at the Test Navigator (⌘5), seen in the figure below. Rather than showing test files, this shows the tests themselves, and whether they passed or failed the last time they ran. This is another location in Xcode that makes it easy for you to get an overview of what tests you have and whether they are passing or not.

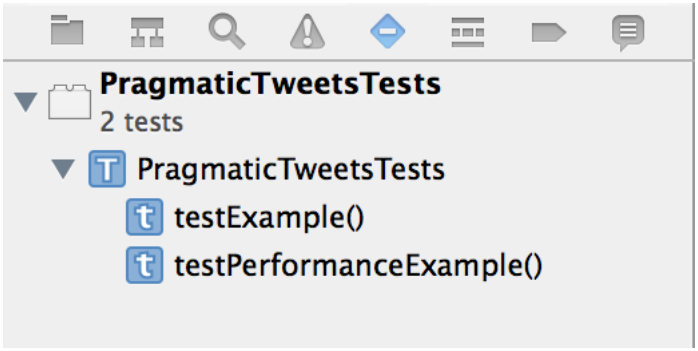


Figure 37—Xcode Test Navigator

Click on the PragmaticTweetsTests file in either the Project or the Test Navigator. There are four methods within this class: setUp(), tearDown(), testExample(), and testPerformanceExample(). Every test class that we create will have a setUp() and a tearDown() method method. setUp() is used to instantiate any boilerplate code you need to set up your tests and tearDown() is used to clear away any of the setup you needed to do for your tests. Whenever we find ourselves repeating code in multiple tests, it's a candidate for moving into setUp() and tearDown(). This is the principle of DRY: Don't Repeat Yourself.

Every test method we create will start with the word "test", just as the testExample() and testPerformanceExample() methods demonstrate. Additionally, every method in a test class will have no return type. We are doing this to make sure that our tests are found by the test compiler and that they get run. A test passes if it returns normally, and fails if it fails an assertion method before it returns.

For fun, let's just run the test included in the template. There are several ways to run your unit tests:

- Keyboard command: ⌘U
- Main Menu: Product→Test
- Clicking on the diamond icon next to either the test class or the specific test in Xcode.

The first two ways of running tests will run all of your tests, whereas the third way will allow us to run selected tests. This is useful if you have one test that is failing and we want to focus on that one without having to run all the others.

Run the test in the manner of your choice.

Let's take a closer look at testExample().

Testing/PragmaticTweets-4-1/PragmaticTweetsTests/PragmaticTweetsTests.swift

```
func testExample() {
  // This is an example of a functional test case.
  XCTAssert(true, "Pass")
}
```

Just for fun, go in and change the first parameter in the method to "false" from "true" to see what happens.

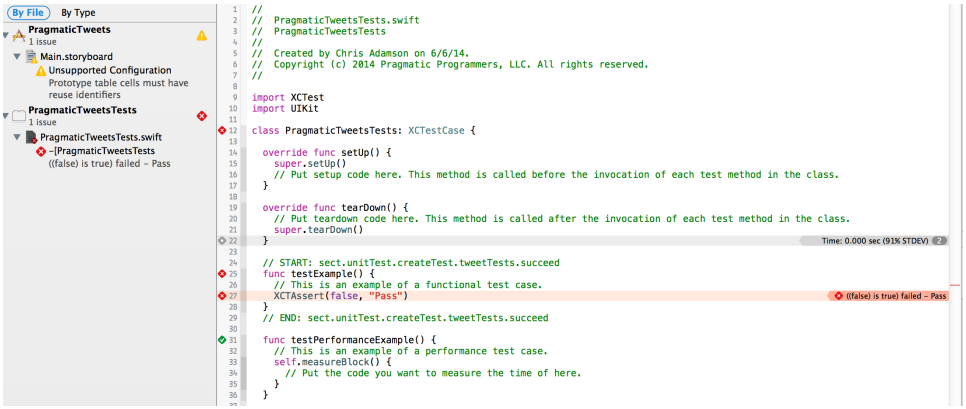


Figure 38—Failed Unit Test Run

Oh no! The test stopped working! What happened?

Well, we just changed the conditions of the test. XCTAssert must pass a true condition through the test to pass. Since we programmed the condition to false, the test fails. Option-clicking on XCTAssert doesn't give us nice documentation like most Swift methods, but we'll cover the most useful XCTest assertions later in *OCUnit and XCTest*, on page 73

At first blush this might seem like a useless exercise. Why would we want to write a test that always fails when you run it?

We run a test that is designed to fail so that we verify that the testing framework itself is working properly. If we simply created nothing but tests that are supposed to pass, we can't know for certain that the tests are passing because the code is correct. There could be an error and the tests would pass regardless. By prompting a failure, we now verify that when we write a test that passes that our code is, in fact, working correctly. As one wise person put it, "How do you know your smoke detector works if it never goes off?"

OCUnit and XCTest

At this point you may be wondering where we got XCTFail and XCTAssert from.

Prior to iOS 6, unit testing was done using an open source framework called OCUnit. As of iOS 8, OCUnit is deprecated, and as of Xcode 5, the preferred testing framework is XCTest. XCTest is built on top of OCUnit, so anyone already familiar with OCUnit should fine translating from OCUnit to XCTest is simple and straightforward.

As we saw earlier, the method in XCTest to assert a failure is XCTFail(). In OCUnit, the assertion call for a failing method is STFail(). It is a convention that any method you have in OCUnit can be converted to XCTest() by changing the "ST" at the beginning to "XCT". So, what was STFail() is now XCTFail().

There are about twenty different assertion methods in XCTest, but the ones we will be using most often are:

- XCTAssert()
- XCTAssertFalse()
- XCTAssertEqual()
- XCTAssertNotNil()
- XCTAssertThrowsSpecificNamed()

There is a complete list of every assertion in the "Testing with Xcode" programming guide in your Xcode documentation, if you want to see how deep the rabbit hole goes.

Test Driven Development

Now that we have a good handle on how to create a unit test, we are going to delve into the realm of Test Driven Development (TDD). TDD, in a nutshell, is figuring out the least number of objects you need to create in order to get your application to work the way you want it to. TDD utilizes the idea that you will write your tests first rather than after you have already completed your application.

If we write tests for the app now, we'll just be checking functionality we already know works. In TDD, we write the test first, fail for lack of any working functionality, then press ahead and actually create the functionality.

Why do we want to do all this extra work before we write a line of code? Let's jump in the Way Back machine and visit your elementary school English class. Remember back when you were learning how to write stories your teacher told you to write an outline. We write outlines for our stories so that we have an idea about how our story is going to go. We want to figure out the beginning, middle, and end so that we can write a tight and cohesive story

that follows a path and has an ending that makes sense. If you go into a story not really sure about what is going happen you wind up writing lots and lots of plot where nothing really happens.

Our time is valuable. It is in our best interest to figure out exactly which features are important and which ones are not before you spend a week trying to figure out and debug a feature that we figure out later doesn't really fit in with what we want our app to do.

So let's add a new feature, TDD style. Let's say we want to have the web view load itself when the app starts up, without having to tap "Show my tweets". We'll start by writing a test to make sure the web view got populated, initially failing because it's not being populated, then go back and add the feature. When the test passes, our feature is good to go.

We'll start by creating a new test class. Before you create this class, click on the PragmaticTweetsTests group. Use the menu item "File"→"New File..." to bring up a template of file types. Choose "iOS Source" from the left pane, and on the right, select the "Test Case Class" template. Name our new class WebViewTests. Make sure this class is a subclass of XCTest and that it is attached to the PragmaticTweetsTests target before creating the class as shown in the following figure.

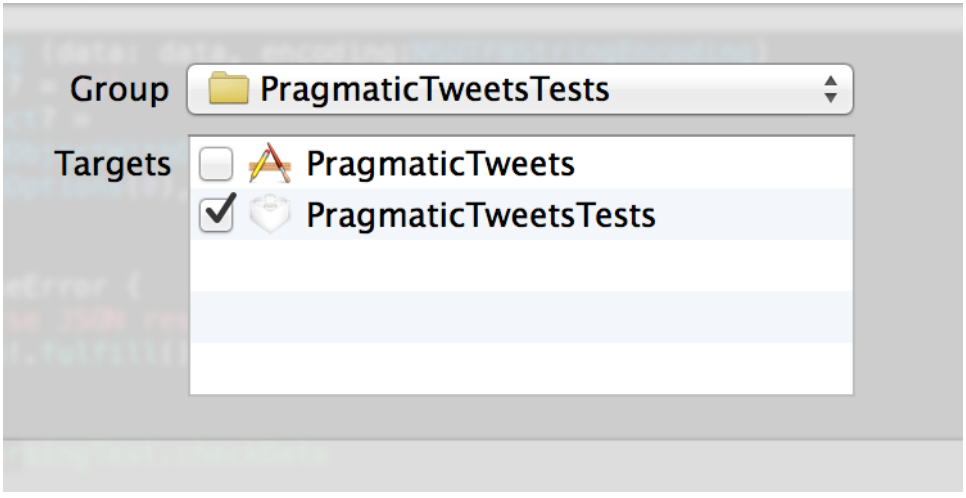


Figure 39—Choosing a target for your file

Xcode Targets

In Xcode, it is possible to create multiple applications based on the same code base. If we wanted to make, for example, a game where we had "full" and "lite" versions where the only difference is how many levels are included, we could create two targets that mostly differed by which level files were or weren't included.

Since the primary application does not know what to do with a testing class, we don't want to include it in the codebase for that application; it would just take up space on the end-user's device. Putting the test classes in their own target help us segregate them out.

Targets can also be used for other sophisticated build-tasks, like running arbitrary shell scripts prior to or after building our code. They can also be set as dependencies as one another. For example, the tests target is dependent on the main app target, so any time we run tests, any changes to the app's code will be built first.

Creating Tests

The WebViewTests class contains the same setUp(), tearDown(), testExample(), and testPerformanceExample() methods we saw in the other class. We can delete the latter two methods, since they're just meant as example and we will be writing our own.

Visibility Modifiers

What we need to do is to write a test method that can access the twitterWebView property of the ViewController class. This actually presents a little bit of a hassle that we haven't had to consider before. Swift considers all the classes in the PragmaticTweets to be one *module*, and classes in a module can see each other's properties and methods by default. However, PragmaticTweetsTests is a different target and thus a different module, so it cannot see the methods or properties of our app's classes. We'll have to fix that before we can test anything.

Start in WebViewTests.swift by adding an import statement, just like the default ones that pull in the UIKit and XCTest frameworks. In our case, we need to import the PragmaticTweets module:

Testing/PragmaticTweets-4-1/PragmaticTweetsTests/WebViewTests.swift

```
import PragmaticTweets
```

Now, we have to decide which parts of our app we want to expose to outside callers, like the test classes. Swift has three levels of visibility, set by special keywords:

Access modifier	Visibility
public	Visible everywhere
internal	Visible within the same module
private	Visible only within the class itself

So, for the sake of testing, we need to make the ViewController public, as well as its twitterWebView class. So, edit the top of the ViewController.swift file as follows:

Testing/PragmaticTweets-4-1/PragmaticTweets/ViewController.swift

```
public class ViewController: UIViewController {

  @IBOutlet public var twitterWebView : UIWebView!
```

As a side-effect, as soon as we declare the class public, Xcode will give us an error saying all the methods marked with the override keyword have to have the same accessibility as their class. So put a public modifier right or after (the order doesn't matter) the override on the viewDidLoad() and didReceiveMemoryWarning() methods.

Writing Unit Tests

Now we're ready to write our test. What we want to do here is to look at the contents of the twitterWebView. To keep things simple, we won't go scraping for any specific text — Twitter could always change their web page — and instead we'll just make sure it isn't blank.

The test is really an outsider, so it doesn't have direct access to the views on the screen or the logic behind them. However, we can ask the UIApplication object for the first view controller it's showing (luckily, we only have one in our app) and drill down from there. So let's write a testAutomaticWebLoad() class like this:

Testing/PragmaticTweets-4-1/PragmaticTweetsTests/WebViewTests.swift

```
Line 1 func testAutomaticWebLoad() {
   -   if let viewController =
   -     UIApplication.sharedApplication().windows[0].rootViewController
   -     as? ViewController {
     5     let webViewContents =
   -       viewController.twitterWebView.stringByEvaluatingJavaScriptFromString(
   -         "document.documentElement.textContent")
   -     XCTAssertNotNil(webViewContents, "web view contents are nil")
   -     XCTAssertNotEqual(webViewContents!, "", "web view contents are empty")
    10   } else {
   -     XCTFail("couldn't get root view controller")
   -   }
   - }
```

Lines 2-4 are how we get to the ViewController object. The shared UIApplication object has an array of UIWindows (one per screen, so usually just one unless we're doing AirPlay), and each window has a rootViewController. So we use an if let statement to try to get that object as our ViewController class.

If that works, then we want to inspect the contents of the twitterWebView. There's no method on UIWebView to just give us its contents, but there is the method stringByEvaluatingJavaScriptFromString(), which lets us run any JavaScript string on the contents of the UIWebView (seriously!). So on lines 5-7, we evaluate the DOM property document.documentElement.textContent to get the text of the web page.

We are now ready to test whether or not this got anything. On line 8, we use XCTAssertNotNil() to make sure the webViewContents is not nil. And then on line 9, we use XCTAssertNotEqual() make sure it's not an empty string. If we survive both of those test methods, the method executes normally and we pass the test.

Notice that we will also fail the test on line 11 if we can't get the rootViewController as our ViewController class. If we start getting reckless with the storyboard, and change how the app even works, this test will let us know.

We now have a test and no feature. So what do we do? This is test-driven development, so we run the test of course! Click the diamond to the left of testAutomaticWebLoad() to run just this test.

And we fail. We knew we'd fail, because we know the feature isn't there. The error message from the XCTAssertNotEqual() assertion appears next to that line in the source to show us where the test failed. Our pass/fail results also appear in the test navigator, and in the Report Navigator (⌘8), which has a nice summary of all tests run, the simulator or device we ran them on, and which tests failed and where. See the following figure:

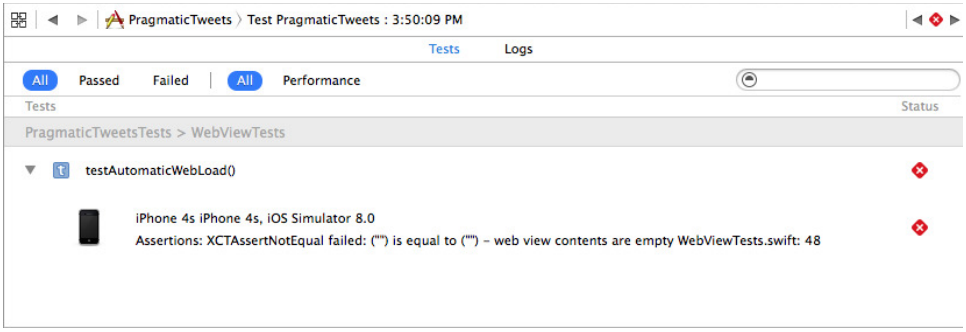


Figure 40—Failed test in Report Navigator

Finishing the Feature

We are following proper TDD practice: we built a test, we watched it fail. Now we can build the feature, and when the test stops failing, we know we have a working feature.

Go back to ViewController.swift. We want the web view to come up when the app does, so all the things we do in handleShowMyTweetsButtonTapped() should happen at in viewDidLoad() too. A good way to do that is to have them both call the same thing. Copy everything currently in handleShowMyTweetsButtonTapped() into a new method called reloadTweets(), and then make handleShowMyTweetsButtonTapped() just be a call to reloadTweets(), like this:

Testing/PragmaticTweets-4-1/PragmaticTweets/ViewController.swift

```
@IBAction func handleShowMyTweetsButtonTapped(sender : UIButton) {
  self.reloadTweets()
}

func reloadTweets() {
  let url = NSURL (string:"http://www.twitter.com/pragprog")
  let urlRequest = NSURLRequest (URL: url)
  self.twitterWebView.loadRequest(urlRequest)
}
```

Now, add the line self.reloadTweets() to the viewDidLoad() method. This will do our automatic web-page loading.

Run the app (not the test) with the "Run" button or ⌘R to make sure it works. The app comes up, the web page loads. We are good to go! Now run the test and we will have finished our first TDD development.

But wait, the test is still failing! What's wrong?

Testing Asynchronously

Take another look at running the app. After the app appears, it takes a second or two for the web page to load. But from the test's point of view, as soon as the app is up and running, it is ready to be tested. What's happening is that we are testing too soon. We need a way to wait before we run our test.

What we need is *asynchronous testing*, the ability to test things that happen at unpredictable times. If we wanted to test that 2 + 2 == 4, or that a string has a certain value, we could do that right away, because the value would be there right when we asked for it. But with the web view, we don't know when (or if) its contents will be set. Asynchronous testing lets us test these kinds of unpredictable events.

Prior to iOS 8, you could not run asynchronous unit tests using XCTest, so it was impossible to do testing on the network calls, background tasks, or anything else where the value to be tested was not immediately available.

iOS 8 introduces a new testing class called XCTestExpectation. XCTestExpectation creates expectation objects that describe events that you expect to happen at some point in the near future. We tell it how long it can wait, and then perform test assertions elsewhere — in parts of the code that run asynchronously — finally notifying the expectation when we're done. And if we fail to do so, that's considered a failure.

Joe asks:

What the heck is an "Expectation Object?"

There is a wonderful quote by the late John Pinette that goes: "Salad isn't food. Salad comes with the food. Salad is a promissory note that food will soon arrive."

Expectation objects are like salad. They are not the test, they are the promise to your program that something is going to happen a little later.

If you went to a restaurant and got a salad, then waited for an hour for food that never arrives, you would realize something is terribly wrong. You were set up to expect that another part of your meal was coming and if it never arrived, your meal would be a failure.

That, in a nutshell, is how asynchronous testing with expectation objects works.

Creating an Expectation

The first thing we will do is to create a XCTestExpectation object in the WebViewTests class:

Testing/PragmaticTweets-4-2/PragmaticTweetsTests/WebViewTests.swift

```
var loadedWebViewExpectation : XCTestExpectation?
```

We will need to create this expectation object when we start the test, and then when we know the web view has loaded, we tell it that we're done by calling its fulfill() method.

We will need a way to know when the web view has loaded. We will learn some general-purpose techniques for doing asynchronous tasks in *Do-It-Yourself Concurrency*, on page 128, but for now, UIWebView can help us out. It has a delegate object that gets notified when web pages load, fail to load, when the user submits a form, and other events. We can use that to know that the web page has loaded, and then pass or fail the test.

To be a delegate, we have to declare that our class implements the UIWebViewDelegate protocol, which declares the methods that the web view can send to the delegate.

Testing/PragmaticTweets-4-2/PragmaticTweetsTests/WebViewTests.swift
```
class WebViewTests: XCTestCase, UIWebViewDelegate {
```

We are going to rewrite the testAutomaticWebLoad() to do two things. The first is to become the web view's delegate. The second is to create our expectation object so that the tests know to wait a little while and don't just return a test fail. Here's how we do that.

Testing/PragmaticTweets-4-2/PragmaticTweetsTests/WebViewTests.swift
```
Line 1 func testAutomaticWebLoad() {
  -   if let viewController =
  -     UIApplication.sharedApplication().windows[0].rootViewController
  -       as? ViewController {
    5     viewController.twitterWebView.delegate = self
  -     self.loadedWebViewExpectation =
  -       expectationWithDescription("web view auto-load test")
  -     waitForExpectationsWithTimeout(5.0, handler: nil)
  -   } else {
   10     XCTFail("couldn't get root view controller")
  -   }
  - }
```

On line 5, our test class becomes the web view's delegate, so it can be notified of events from the twitterWebView.

Next, line 6-7 creates the loadedWebViewExpectation, and gives it the name "web view auto-load test". If we have many expectations, the name helps us figure out which one failed. We create as many expectations as we need — just one for now — and kick them off with a call to waitForExpectationsWithTimeout() on line 8. We have to call fulfill() on the expectation within 5 seconds, or we will get a timeout test failure.

Now that we have created our expectation object, we need to implement the UIWebViewDelegate protocol. If you look in the documentation, you will see this protocol has four methods that it can call: one to ask if it should start loading, one to report an error, and one each when the page starts and stops loading. We will implement two of these: if the web page load fails, our test fails, and if it succeeds, we use the JavaScript call to see if the web view has any contents. We will start with the easier failure case.

Testing/PragmaticTweets-4-2/PragmaticTweetsTests/WebViewTests.swift
```
func webView(webView: UIWebView, didFailLoadWithError error: NSError) {
  XCTFail("web view load failed")
  self.loadedWebViewExpectation!.fulfill();
```

```
}
```

If the web page doesn't load, we use the always-fail XCTFail() to tell the test suite we failed. Notice that we also have to fulfill() the expectation even though we've already failed; if we don't do this, we'll get a timeout that looks like a second failure.

Now we can look at the possible success case.

Testing/PragmaticTweets-4-2/PragmaticTweetsTests/WebViewTests.swift

```
func webViewDidFinishLoad(webView: UIWebView) {
  if let webViewContents =
    webView.stringByEvaluatingJavaScriptFromString(
      "document.documentElement.textContent") {
    if webViewContents != "" {
      self.loadedWebViewExpectation!.fulfill()
    }
  }
}
```

This is like our first version in how it gets at the twitterWebView and runs the JavaScript to get the web document's contents as a string. The difference is that all we do this time is fulfill() the expectation if we ever get contents that aren't an empty string. (It turns out we don't want to fail on an empty string because this will be called with an empty string when the app starts up… by not failing then, we give web view more tries to call us back with some contents.)

Whew! That was a lot of code. Congratulations, you are now capable of doing something that wasn't possible before. Now click the diamond next to testAutomaticWebLoad(). This time, we pass the test, as shown by the green icon in the Test Navigator in the following figure.

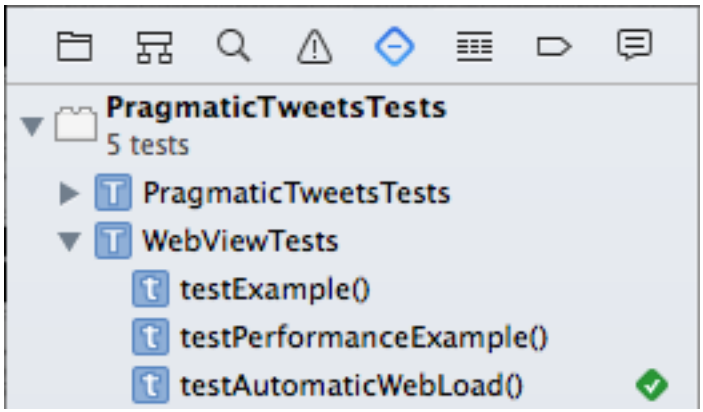


Figure 41—Passed test in Test Navigator

Testing Frameworks

Beyond Apple's built-in unit test framework, there are some further internal and external frameworks that exist to make your testing life easier. We are just going to touch on these briefly so that you are aware of what they are and what options there are available to you to make the best app that you can.

OCMock

At this point, you might be wondering how to test any functionality that requires a call to the network when we don't have internet, or a request for which we always want the exact same response. For this kind of testing, you would need to employ a mock object. A mock object is like a Patronus from Harry Potter. Like Harry Potter's Patronus, a mock object is an imaginary object we conjure into existence as a stand-in for a real being.

A mock object is a file, object, or piece of code that we create to fill in for the actual data we will be using in our application. We create it to stand in for something we would rather not muck around with in our production code.

Unit tests are supposed to be small and test on unit of functionality. If we want to test our Tweet parser without a mock object, that would require us to make an API call. Since the API call would require us to make a network call, there would be many things that would need to happen correctly for our test to pass. It is possible the server could be down or that there is an error in our networking code. Without having it be clear exactly which of those

possibilites is the root reason for a test failure, we would be unable to proceed with fixing the issue.

OCMock is a third-party framework created to solve the issues associated with testing networked apps or any apps that have dependencies on external parameters. OCMock is an Objective-C implementation of mock objects. OCMock provides the following functionality:

- Stub objects that return pre-determined values for specific method invocations
- Dynamic mocks that can be used to verify interaction patterns
- Partial mocks to overwrite selected methods of existing objects

OCMock has a rich repository of documentation and a great deal of user support. If you are interested in utilizing their functionality in your app, go to their website http://www.ocmock.org and read through their tutorials to get your mock objects up and running.

UI Automation

UI Automation is different than the types of unit tests we have been doing so far. UI Automation, like its name implies, simulates user interactions with your app. The parts of the app that we have been testing have been its internal logic. We did not set up any tests to see what happens when a user presses a button or interacts with the app in any way. UI Automation is the ideal way to test your app without having an army of monkeys pressing every single button and interaction you have programmed into your app. (If you do have an army of monkeys, they could probably be better used sitting at typewriters trying to recreate the works of Charles Dickens.)

UI Automation is best used to do the following:

- Access its UI element hierarchy
- Add timing flexibility by having timeout periods
- Log and verify the information returned by Instruments
- Handle alerts properly
- Handle changes in device orientation gracefully
- Handle multitasking

To use UI Automation you need to use Instruments, one of the tools that comes with Xcode (available though the “Product”→“Profile” menu item, ⌘I).

The fact that UI Automation is not integrated into Xcode's unit testing targets or XCTest illustrates the big pitfall of doing test-first development on iOS: many of the things we want to expose to unit tests are user interactions that are really difficult to turn into automated tests. We were really only able to do TDD in this chapter because it was kicked off by the application's start-up and not by a user-interaction. It's because of this that few iOS developers we know do true TDD, and those who do usually limit it to the models and logic of their apps, not the user interface.

For now, UI Automation is the best option we have for testing user interactions, but its features are beyond the scope of this chapter. If you are interested in utilizing UI Automation in your app, please check out the excellent book .

Wrap-Up

In this chapter we have gone on a nice tour of unit testing and gotten a taste for the fundamentals of Test Driven Development. We walked through the TDD process from idea to implementation.

We have explored Apple's built-in unit testing suite XCTest. We also touched on some of the most common tools in your toolbox for testing your application. You now have the tools to go forth into the world and test your apps so that you can be sure your users will not have to deal with a crash or erroneous behavior.

Now that we know how to write tests to ensure our features work as designed, we're going to start really reworking those features. In the next chapter, we'll start moving to the table view style of presentation that is so common on iOS.

CHAPTER 5

Presenting Data in Table Views

For organizing and presenting many of the kinds of data we see in iPhone and iPad apps, it's hard to beat a table view. Thanks to the intuitive flick-scrolling provided by iOS, it's comfortable and convenient to whip through lists of items to find just the thing we need, with each item visually presented in whatever way makes sense for the app. In many apps, the table view is the bedrock of the app's presentation and organization.

In this chapter, we're going start to turn our Twitter application into one that's based around a table view. However, it's going to take us a few chapters to completely move away from the web view. First, we'll put some fake data into a table view, and then in the following chapters we'll get real data from the Twitter API and load it into the table view.

Tables on iOS

Coming from the desktop, one might expect a `UITableView` to look something like a spreadsheet, with rows and columns presented in a two-dimensional grid. Instead, the table view is a vertically scrolling list of items, optionally split into sections.

The table view is essential for many of the apps that ship with the iPhone, as well as popular third-party apps. In Mail, tables are used for the list of accounts, the mailboxes within each account, and the contents of each mailbox. Reminders is little more than a table view with some editing features, as are the alarms in the Clock app. The Music app shows lists of artists or albums, and within them lists of songs. Even the Settings app is built around a table, albeit one of a different style than is used in most apps (more on that later).

And while our Twitter app currently displays a web view of all the tweets we've parsed, pretty much every Twitter app out there (including the official Twitter app, as well as *Twitterrific*, *Tweetbot*, *Echofon*, etc.) uses a table view to present tweets.

So our task now is to switch from the web view to a table view-based presentation of the tweets. We'll build this up slowly, as our understanding of tables and what they can do for us develops.

Table Classes

To add a table to an iOS app, we use an instance of `UITableView` . This is a `UIScrollView` subclass, itself a subclass of `UIView`, so it can either be a full-screen view unto itself or embedded as the child of another view. It cannot, however, have arbitrary subviews added to it, as it uses its subviews to present the individual cells within the table view.

The table has two properties that are crucial for it to actually do anything. The most important is the `dataSource`, which is an object that implements the `UITableViewDataSource` protocol. This protocol defines methods that tell the table how many sections it has (and optionally what their titles are) and how many rows are in a given section, and provides a cell view for a given section-row pair. The data source also has editing methods that allow for the addition, deletion, or reordering of table contents. There's also a `delegate`, an object implementing the `UITableViewDelegate` protocol, which provides method definitions for handling selection of rows and other user-interface events.

These roles are performed not by the table itself — whose only responsibility is presenting the data and tracking user gestures like scrolling and selection — but by some other object, often a view controller. Typically, there are two approaches to wiring up a table to its contents:

- Have a `UIViewController` implement the `UITableViewDataSource` and `UITableViewDelegate` protocols.
- Use a `UITableViewController`, a subclass of the `UIViewController` that is also defined as implementing the `UITableViewDataSource` and `UITableViewDelegate` protocols.

It's helpful to use the second approach when the *only* view presented by the controller is a table, as this gives us some nice additional functionality like built-in pull-to-refresh, or scrolling to the top when the status bar is tapped. But if the table is just a subview, and the main view has other subviews like buttons or a heads-up view, then we need to use the first approach instead.

Model View Controller

The careful proportioning of responsibilities between the view class and the controller comes from UIKit's use of the *model-view-controller* design pattern, or MVC. The idea of this design is to split out three distinct responsibilities of our UI:

- *Model* — the data to be presented, such as the array of tweets.
- *View* — the user-interface object, like a text view or a table.
- *Controller* — the logic that connects the model and the view, such as how to fill in the rows of the table, and what to do when a row is tapped.

This pattern explains why the class we've been doing most of our work in is a "view controller"; as a controller, it provides the logic that populates an on-screen view, and updates its state in reaction to user interface events. Notice that it is not necessary for each member of the design to be have its own class: the view is an object we created in the storyboard, and the model can be a simple object like an array. At this point in our app's evolution, only the controller currently requires a custom class.

Creating and Connecting Tables

We're going to need to make some major changes to our user interface to switch to a table-driven approach. In fact, we're going to blow away our original view entirely. We'll get all our functionality back eventually, and we'll be in a better position to build out deeper and more interesting features. Eventually, we'll we have an app that looks and feels like a real Twitter client.

We'll start by preparing our view controller to supply the table data. We can do this by either declaring that we implement UITableViewDataSource, or by becoming a subclass of UITableViewController. Since the table will be the only thing in this view, let's do the latter. In ViewController.swift, rewrite the declaration like this:

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

```
class ViewController: UITableViewController {
```

Adding a Table View to the Storyboard

Now switch to Main.storyboard and look through the Object Area at the bottom right for the "Table View Controller" object, shown in the following figure. Drag one into the storyboard, anywhere where it won't collide with the existing view controller. This adds a new "Table View Controller Scene" to the list of scenes in the storyboard.

Figure 42—Table View icon in Object Library

Select the View Controller from the previously existing scene and press ⌫ to delete the old scene. Notice that the table view automatically gets the incoming arrow from its left side, which indicates it is the app's initial view controller (this can also be inspected in the table view controller's attribute inspector). The view itself shows a status bar that says "Prototype Cells" above a Table View that has a single Table View Cell as a subview, as seen in the following figure.

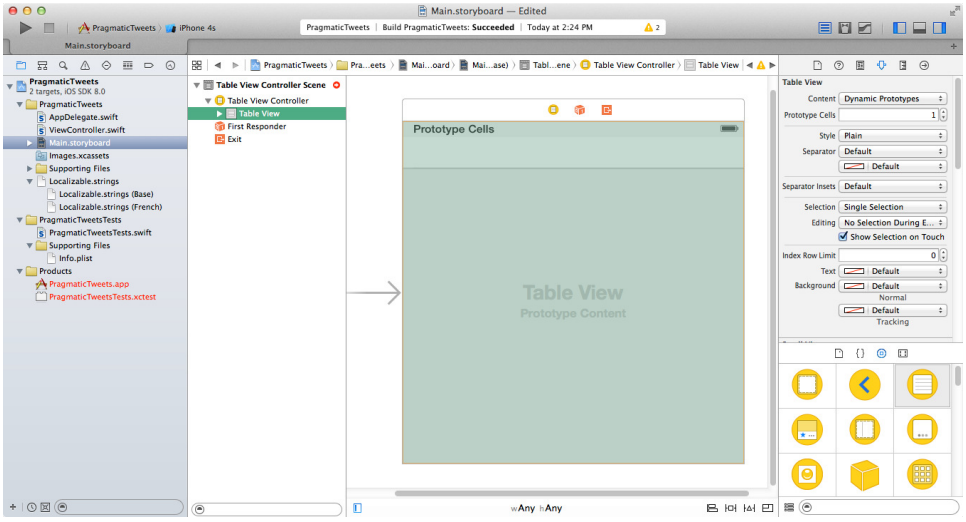


Figure 43—Empty Table View Controller Scene in Storyboard

We can run this app… but it shows an empty table! That's because the table is not yet connected to a data source that can provide it with cells or even a count of how many sections and rows there are. Let's get to work on that.

Providing a Temporary Table Data Source

As it is, the table in the storyboard doesn't know to use our class; it expects to create a generic UITableViewController for the table. We want it to use our View-

Controller instead. So, while still in Main.storyboard, choose the Table View Controller and visit its Identity Inspector in the right-side pane (⌥⌘3). In the "Custom Class" field, enter ViewController. This should auto-complete, since we declared that our ViewController class is a valid UITableViewController; although, we've done nothing to implement that behavior yet.

While here, control-click on the table view, or visit its Connections Inspector (⌥⌘6), and notice that table view's connections to the dataSource and delegate properties are already wired up, connected to the view controller.

As a warm-up, let's provide a trivial implementation of the data source methods, just to ensure the new storyboard and its connections are good to go. To do this, our data source needs to provide a minimum of three things: the number of sections; the number of rows in a given section; and a cell for a given section and row. In ViewController.swift, provide the following trivial implementations of the UITableViewDataSource methods numberOfSectionsInTableView(), tableView(numberOfRowsInSection:) and tableView(cellForRowAtIndexPath:), as well as the optional tableView(titleForHeaderInSection:), which will let us see the section breaks.

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

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
  return 5
}

override func tableView(_tableView: UITableView!,
  titleForHeaderInSection section: Int) -> String! {
    return "Section \(section)"
}

override func tableView(_tableView: UITableView!,
  numberOfRowsInSection section: Int) -> Int {
    return section + 1;
}

override func tableView (_tableView: UITableView!,
  cellForRowAtIndexPath indexPath: NSIndexPath!) -> UITableViewCell {
    let cell = UITableViewCell(style: UITableViewCellStyle.Default,
      reuseIdentifier: nil)
    cell.textLabel.text = "Row \(indexPath.row)"
    return cell
}
```

Also, while we're in the ViewController.swift file, let's delete the line that declares the twitterWebView that no longer exists, and all of the handleShowMyTweetsButtonTapped() method that populated it. We won't need those anymore.

Joe asks:

Why Are All The UITableViewDataSource Methods Named tableView()?

Notice that in our quick-and-dirty table code, three of our methods are called tableView(). The reason these methods don't get confused with one another is because they're differentiated by their named parameters: one takes titleForHeaderInSection, another takes cellForRowAtIndexPath, and so on.

By convention, all these methods take the table view in question as their first argument, so if we had multiple tables, a method would be able to figure out which table it's working with.

But as for why it has to be the *first* parameter, that's more of a legacy of Objective-C, where it was somewhat more natural to incorporate the name of your first parameter into the method name, and differentiate with the rest of the parameters. Swift came later, so we're stuck with the old naming schemes, at least for now.

In this book, when we encounter cases where the method name by itself isn't unique, we'll include the parameters for clarity. That way, we'll call out the difference between tableView(numberOfRowsInSection:) and tableView(cellForRowAtIndexPath:), but we won't feel the need to write viewWillAppear(animated:), when there's only one method that starts like that, so it can be written as just viewWillAppear().

In this implementation, we are telling the table that there are five sections, that each section has one more row than the section index (that is to say: There's one row in section 0, two rows in section 1, etc.), and that any time a new cell is needed, it should create a new UITableViewCell, get its textLabel property (a UILabel), and set the text property of the label to a string that shows the row number. When run, the table will look like this:

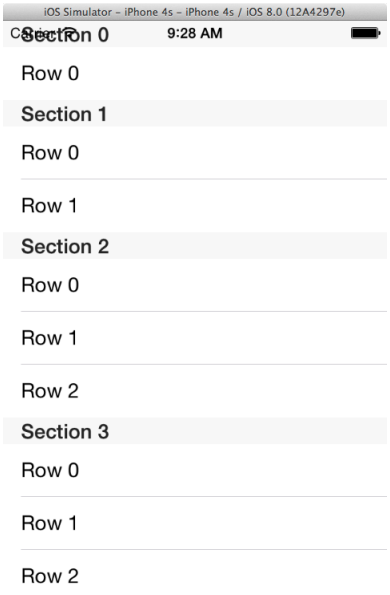


Figure 44—Trivial Implementation of UITableViewDataSource

Notice that tableView(cellForRowAtIndexPath:) passes in an NSIndexPath. This is a class originally intended for representing paths in tree structures, things like "the third child of the second child of the root node". In iOS, it is pressed into service representing table entries. NSIndexPath is extended to add the properties section and row (which are actually implemented as just the first and second entries in the path), and the combination of section and row can uniquely identify any cell in UITableView.

Now we have a table and a way to get data into it. What we need to do next is to provide a non-trivial implementation of the data source, one which actually shows some tweets.

Joe asks:

Why is the status bar overlapping the table?

One of the more controversial aspects of the iOS 7 visual design is that view controllers default into a full-screen mode. In fact, the property wantsFullScreenLayout was deprecated in iOS 7, and since then view controllers are assumed to *always* fill the screen with their views, even the space under the status bar.

It looks horrible at first, but the idea is that once we start scrolling and see content go under the status bar, the transparency of the status bar gives us a visual cue

about information that is about to come fully into view. In later chapters, we will add a navigation bar at the top and then it will look and feel a lot better.

Filling In The Table

Let's think about how we're going to go from this to a table of real tweets. Since the table can demand the contents of any row at any time, we will want to have a data structure representing all the tweets that we want to show in the table. This doesn't have to be anything fancy; in fact, an array will do just fine.

But an array of what? Well, one approach would be to just create a class including the parts of a tweet we care about — its text, the screen name of the person who sent it, etc. — and then have an array of those objects. We haven't had to create a new class of our own yet, so let's do that.

Creating a ParsedTweet Class

To create a new class in our project, we use File→New File… (⌘N), which causes a sheet to slide out showing templates for new files, shown below. From the "iOS" group on the left, choose "Source", and then from the icons on the right, choose the "Cocoa Touch class" template and press "Next".

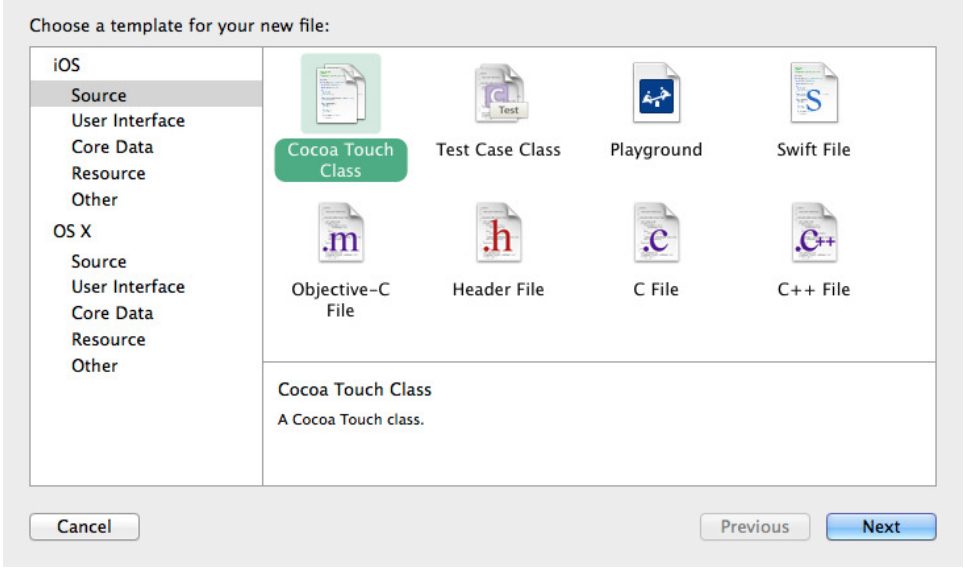


Figure 45—Xcode Templates for New File

This takes us to the second page of the sheet, in which we set the class name, parent class, and language. For class name, use ParsedTweet and for "Subclass of", use NSObject. Make sure the language is "Swift". Press "Next" to show a file dialog that indicates where to save the file, and which *targets* will build the class. The default will build it only for the "PragmaticTweets" app and not the "PragmaticTweetsTests" unit tests. This is what we want, so click "Create" to create a ParsedTweet.swift file in our project.

This class doesn't need to really *do* anything, it just needs to hold on to some values, so it'll be convenient to use in an array. For now, let's figure we'll want the tweet text, the user name, a created-at string, and a URL for the user's avatar. So we define public properties for those, in the ParsedTweet.swift.

Tables/PragmaticTweets-5-1/PragmaticTweets/ParsedTweet.swift

```
var tweetText : String?
var userName : String?
var createdAt: String?
var userAvatarURL : NSURL?
```

We'll make all of these optionals, since we are in no position to populate them when an instance of the class is instantiated. The alternative would be to assign a value like an empty string to one of these properties, but it's more expressive to use the absence of a value to say "this hasn't been set yet", even if it might be a bit more work later to defend against the optional and make sure the property isn't nil.

We can now create ParsedTweet instances in other classes by simply writing code like:

```
let myTweet = ParsedTweet()
myTweet.userName = "@pragprog"
myTweet.tweetText = "Check out our new iOS book!"
```

But it seems a little burdensome to have to create the object and then set all its properties one-by-one. Fortunately, Swift lets us define initializer methods that can set some or all of an object's properties at initialization time. We do this by writing an init() method, specifying which arguments we want to take. After the properties in ParsedTweet.swift, add the following initialization method:

Tables/PragmaticTweets-5-1/PragmaticTweets/ParsedTweet.swift

```
init (tweetText: String?,
      userName: String?,
      createdAt: String?,
      userAvatarURL : NSURL?) {
  super.init()
  self.tweetText = tweetText;
  self.userName = userName;
```

```
  self.createdAt = createdAt;
  self.userAvatarURL = userAvatarURL;
}
```

This initializer starts by calling its superclass' initializer, then takes a value for every property in the class and simply assigns the property. Because it handles every property, we call it a *designated initializer*. We could also create initializers that only set some of the properties; these are called *convenience initializers*, and are declared in the form convenience init (...). There's a key difference in implementation: convenience initializers should call the designated initializer of their own class (typically passing in nil for properties being left un-set), while designated initializers should call one of the *superclass'* initializers, as we do here with super.init().

Still, we might want to populate a ParsedTweet one property at a time, and by adding the initializer that takes four arguments, we lose the ability to call the no-argument version. Let's add that back in:

Tables/PragmaticTweets-5-1/PragmaticTweets/ParsedTweet.swift

```
override init () {
  super.init()
}
```

Notice that since there's a no-argument init() in the superclass (NSObject), we have to use the override modifier to tell Swift that we know we're overriding that initializer.

Building a Table Model of ParsedTweets

Until we're ready to get real tweets from the Twitter API, we'll have to make do with some *mock data*, predictable stand-in values that will let us figure out tables and put off dealing with network stuff. We can create an array of ParsedTweet objects, and just come up with our own values for the tweetText, userName, and createdAt strings.

Actually, let's start with the URL. Twitter's new user "egg" icon lives at a set of URLs like https://abs.twimg.com/sticky/default_profile_images/default_profile_0_200x200.png, where the 0 after profile_ can be any number between 0 and 6 inclusive, each showing a different color, and the 200x200 has several replacements for different sizes. For now, we'll just use this one image over and over. At the top of ViewController.swift, after the imports, add the following constant:

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

```
let defaultAvatarURL = NSURL(string:
  "https://abs.twimg.com/sticky/default_profile_images/" +
  "default_profile_6_200x200.png")
```

We had to split the URL string into two lines for the book's formatting; feel free to write it all on one line. We would if we could.

Now, we can create an array of ParsedTweet objects to serve as our data model. After the curly brace that begins ViewController's class declaration, declare an array of ParsedTweets, and then inside square braces, use the designated initializer to create as many tweet objects as you like (we'll just use three to save space). The beginning of the class should look like this:

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

```
class ViewController: UITableViewController {

  var parsedTweets : Array <ParsedTweet> = [
    ParsedTweet(tweetText:"iOS SDK Development now in print. " +
                          "Swift programming FTW!",
      userName:"@pragprog",
      createdAt:"2014-08-20 16:44:30 EDT",
      userAvatarURL: defaultAvatarURL),

    ParsedTweet(tweetText:"Math is cool",
      userName:"@redqueencoder",
      createdAt:"2014-08-16 16:44:30 EDT",
      userAvatarURL: defaultAvatarURL),

    ParsedTweet(tweetText:"Anime is cool",
      userName:"@invalidname",
      createdAt:"2014-07-31 16:44:30 EDT",
      userAvatarURL: defaultAvatarURL)
  ]
```

Now that we have an array that can serve as our data source, we can rewrite the UITableViewDataSource methods to use the ParsedTweets in this array to calculate the number of rows and the contents of each. Rewrite those methods as follows:

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

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
  return 1
}

override func tableView(_tableView: UITableView,
  numberOfRowsInSection section: Int) -> Int {
    return parsedTweets.count
}

override func tableView (_tableView: UITableView,
  cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = UITableViewCell(style: UITableViewCellStyle.Default,
      reuseIdentifier: nil)
```

```
    let parsedTweet = parsedTweets[indexPath.row]
    cell.textLabel!.text = parsedTweet.tweetText
    return cell
}
```

In this new version, we have a single section (and therefore can eliminate the entire tableView(titleForHeaderInSection:) method), and the number of rows in this section is just the size of the parsedTweets array. Then we use the indexPath's row property to figure out which ParsedTweet to fetch from our parsedTweets array, and put its tweetText into the cell's text label.

Now run the app and behold the tweets:

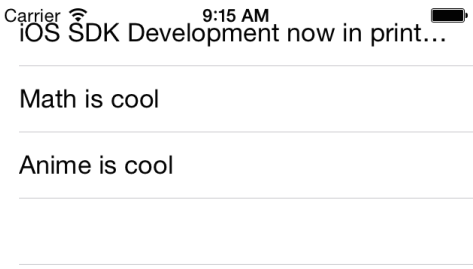


Figure 46—Parsed Tweets in a UITableView

Look at that… we've got our tweets in a table view! And they scroll, so if we coded 200 in our mock data array, we could just flick through them.

Of course, the one line of text isn't big enough for most tweets. So now that we've got our data where it needs to be, let's start improving the table's appearance.

Reloading table contents

One thing that might not be immediately evident, but might come back to bite us later: The only reason we can see any table contents is that the app's startup will do a one-time presentation of the first view's contents. Later on, we will be updating and changing the table's contents. So how do we refresh its contents?

We can add ParsedTweet objects to the parsedTweets array, or delete some of its contents, but the array doesn't have a way to tell the table that its contents have changed, so the table won't actually do anything if we just edit the array. As a controller, it's actually our job to keep the view and model in sync. UITableView offers methods to notify the table of distinct edits, like insertRowsAtIndexPaths() or removeRowsAtIndexPaths(). Sometimes, it's simpler to just

do a full-on reload of the table, with reloadData(). So let's write a method to do that for us:

Tables/PragmaticTweets-5-1/PragmaticTweets/ViewController.swift
```
func reloadTweets() {
  self.tableView.reloadData()
}
```

Later on, this view won't be the first thing we see in the app, so let's make sure we reload the table automatically any time it appears, which we can do by editing the viewDidLoad() method to call reloadTweets():

Tables/PragmaticTweets-5-1/PragmaticTweets/ViewController.swift
```
  override func viewDidLoad() {
    super.viewDidLoad()
    reloadTweets()
}
```

Customizing Table Appearance

While it's great to have the Twitter data in our table cells, only having access to a single, one-line text label makes it impossible to show the various fields of the ParsedTweet; at this point, we don't even know who sent which tweet! We need to change what these table cells look like, to provide more room to show our data.

Table Cell styles

When we create the UITableViewCell in tableView(cellForRowAtIndexPath:), our init method takes a style argument. As it turns out, this can allay our problems somewhat.

There are four cell styles defined in the UITableViewCellStyle enumeration, of which we've been using UITableViewCellStyleDefault. The cell class itself defines certain subviews — textLabel, detailTextLabel, imageView, and accessoryView — and the style determines if and where those subviews are laid out. The following figure shows a four-row table with the row number in the textLabel, and the name of the style in the detailTextLabel. Notice that for the UITableViewCellStyleDefault in row 0, the detailTextLabel is not shown at all.

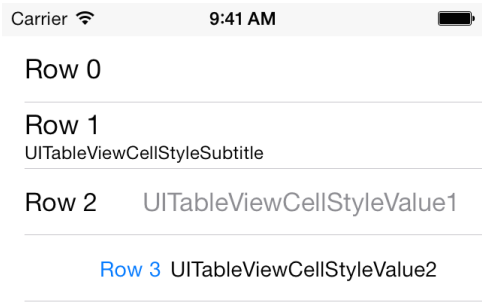


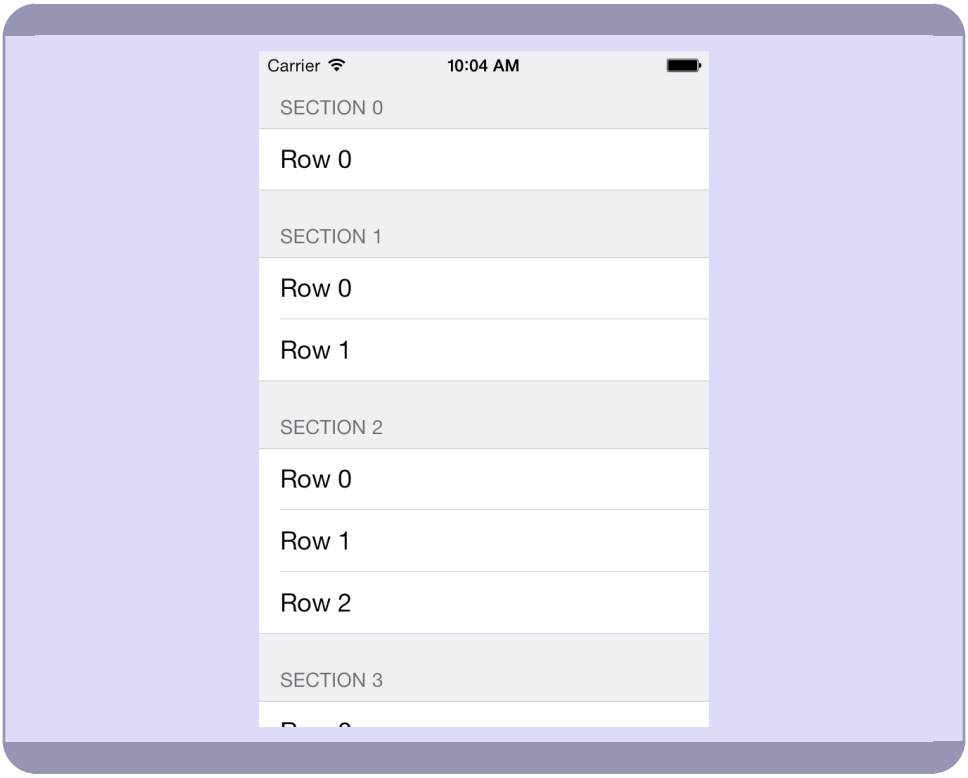
Figure 47—Cell Layouts for Different Values of UITableViewCellStyle

This figure doesn't show the cell's other possible subviews; if an imageView is set, it appears on the left side of the cell, and an accessoryView (usually a "show details" type button) appears on the right. Clearly, if all of those subviews are present, the cell is going to get pretty crowded, and that's something we'll have to deal with soon.

For now, we'll try out the two-line presentation of the UITableViewCellStyleSubtitle, and along the way we're going to fix a problem we've created for ourselves.

Grouped Tables

Another appearance option is to use *grouped tables*, which is just an attribute we can set on the table view in the storyboard. This sets the table's style to UITableViewStyleGrouped, which in turn makes the table look like the figure below. The major differences with a grouped table is that header and footer views do not "stick" to the top or bottom of the screen when scrolling, and on iOS 6 and earlier, the cells have rounded edges that make them look more like buttons. The grouped table is what the Settings app uses, and users will be familiar with its appearance from that.



Cell Re-use

Right now, we create a new UITableViewCell in every call to tableView(cellForRowAtIndexPath:). If we flick through a really long table, that might mean we create a cell that will only appear for an instant before it goes off the screen and is no longer needed. As it turns out, creating views is fairly expensive, so if we can avoid doing that frequently, it will make our app faster and more responsive.

The UITableView class is actually built to cache and re-use cells. It provides a method dequeueReusableCellWithIdentifier() that takes a string identifying a cell to be reused. The idea is that we can create a cell in the storyboard and identify it with a known string. In code, we'll ask for a cell by this name. If the table has already created a cell with this name *and* it isn't currently being shown — meaning it has scrolled off the top or bottom of the screen — the table will give us the old cell and allow us to re-use it. Otherwise, it will create a new cell from the *prototype* in the storyboard. This way, we will only create as many cells as we need to show on the screen at one time, and our scrolling performance will be much improved.

Go to Main.storyboard and select the table view. Notice that the table header says "Prototype Cells", and the table has a single "Table View Cell" as a child. The table's Attributes Inspector also has a field for "Prototype Cells", currently set to 1. We only need the one prototype cell, because new cells will be minted from this prototype — if we had different layouts for different rows (like advertising cells that were different from tweet cells), we could add more prototypes.

Select the cell, and visit its Attributes Inspector (⌥⌘4), and set its style to "Subtitle". Then, for the "Identifier", we need a string value that we can use to fetch this cell from our code. Let's use "UserAndTweetCell".

Now we need to fetch this cell in code and customize it. We do this in tableView(cellForRowAtIndexPath:), so go back to ViewController.swift, and rewrite the method as follows:

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

```
Line 1 override func tableView (_tableView: UITableView,
   2   cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
   3     let cell =
   4       tableView.dequeueReusableCellWithIdentifier("UserAndTweetCell")
   5         as UITableViewCell
   6     let parsedTweet = parsedTweets[indexPath.row]
   7     cell.textLabel!.text = parsedTweet.userName
   8     cell.detailTextLabel!.text = parsedTweet.tweetText
   9     return cell
  10 }
```

The changes we make here are to get the cell via dequeueReusableCellWithIdentifier:() (on lines 3-5), and then to use two fields from the parsedTweet: its userName can go in the textLabel, while the tweetText can go in the detailTextLabel. The result, shown below, is a lot more useful:

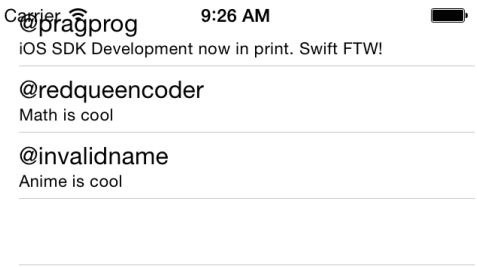


Figure 48—Showing User Names and Tweets with UITableViewCellStyleSubtitle

The result of dequeuing prototype cells instead of creating new cells every time doesn't have an immediate visual impact; although, there is a subtle performance improvement when scrolling a few hundred cells on the simulator, and this effect is much more pronounced when running on a genuine iOS device.

Still, while it's nice to have both the tweet and its author, nearly all the tweets are still being truncated. Clearly we need a multi-line label for those. How are we going to do that? It doesn't look like it's going to fit in the subtitle cell, and even if it does, we still would like to have a third label to show the tweet's timestamp.

Fortunately, we're not limited to the provided cell styles. We can create our own prototype cells with whatever views suit us — even tappable elements like buttons — and then use these in our table.

Custom Table Cells

To create a custom table cell, it usually makes sense to create a subclass of UITableViewCell, and give it public properties for the fields that we will need to update from tableView(cellForRowAtIndexPath:). So, in the File Navigator, select the "Pragmatic Tweets" group, go to File→New File… and choose the "Cocoa Touch class" template. In the next pane of the assistant, name the class ParsedTweetCell, and set the "Subclass of:" to UITableViewCell. We don't need to do anything with the code yet, but it will help us in the Storyboard to have this class already created.

Back in Main.storyboard, select the table view. We could edit the existing prototype cell, but just to prove that we can juggle multiple prototypes, let's create our custom cell as a second prototype. In the Attributes Inspector, tap the up button on the "Prototype Cells" field so the table has 2 prototypes. A second prototype is created, a copy of the first. Select the second and, in the Attributes Inspector (⌥⌘4), change its style to "Custom" and its identifier to "CustomTweetCell". Then, in the Identity Inspector (⌥⌘3), change its class to the ParsedTweetCell class that we just created.

Switch to the Size Inspector (⌥⌘5) and notice that the first field is "Row Height", currently shown with the placeholder text "Default" because the "Custom" checkbox is not checked. If we want to pack a bunch of fields in here, it's pretty clear that the default row height is not going to cut it for us, so click "Custom" and enter a height of 125. That should give us enough room.

Now we get to lay out a UI inside the cell pretty much the same way we built the app's original view with buttons and a web view in earlier chapters.

Within the cell, we can labels, image views, whatever we like… provided that we wrangle all the autolayout constraints to put them in their place (there had to be a catch, right?). We're going to add four subviews: labels for the user name, tweet text, and created-at string, plus an image view for the user's picture. Feel free to play around; for the sample code, we've used the following views:

- An image view with height and width constraints to lock its size at 75x75, plus top and leading constraints of 20 points each.
- A label for the user name, with font "System - Bold", leading constraint of 8 points from the image view, 20 point top and trailing space constraints.
- A label for the tweet text, "Lines" set to 3, with font "System" 14-point, leading and top constraints of 8 points, and trailing constraint of 20 points.
- A label for the created-at string, font "Caption 1", with center alignment, leading and trailing constraints of 20 points and bottom constraint of 8 points.

Once we've created the layout (shown in the following figure) we're going to connect this prototype cell to the custom class we created earlier. Since we used the Identity Inspector to assign the cell to our ParsedTweetCell, the Assistant Editor will let us connect these subviews to new IBOutlet properties in that class. With the cell selected in the storyboard, switch to Assistant Editor mode (it may help to hide the utility pane too). Ideally, it should bring up ParsedTweetCell.swift on the right, but sometimes it chooses ViewController.swift; in that case, use the jump bar at the top of the pane to load ParsedTweetCell.swift into the right side.

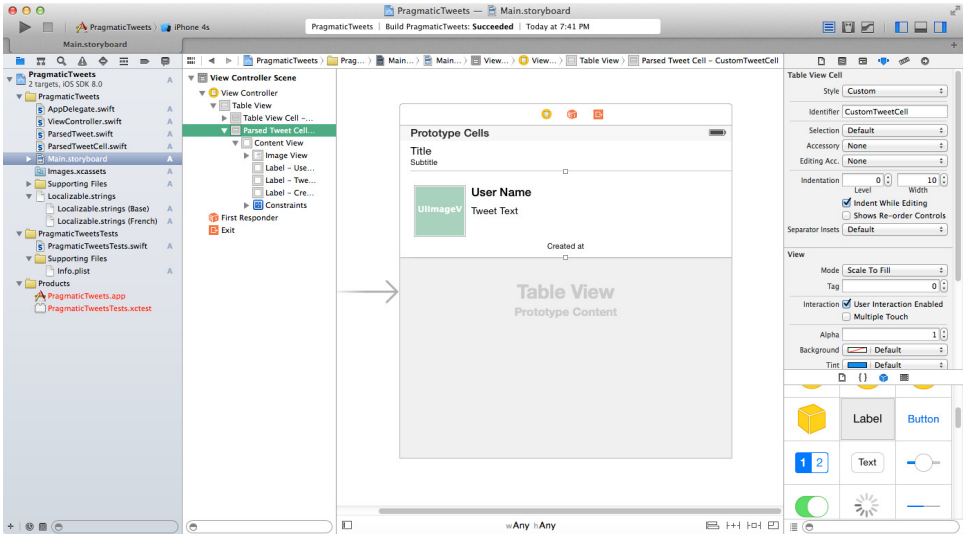


Figure 49—Creating a Custom Prototype Cell

Control-drag from each of the subviews over to the ParsedTweetCell.swift code, just under the class declaration and before the first method. After releasing the drag, give each property an appropriate name. With all the connections established, the properties should look like this:

Tables/PragmaticTweets-5-3/PragmaticTweets/ParsedTweetCell.swift

```
@IBOutlet var avatarImageView : UIImageView!
@IBOutlet var userNameLabel : UILabel!
@IBOutlet var tweetTextLabel : UILabel!
@IBOutlet var createdAtLabel : UILabel!
```

Problems When Connecting Custom Cell Subviews

When creating connections from subviews in the custom cell, be sure that the subview (the label, image view, etc.) is selected, and not the "Content View" that is a superview to all of them. If the popup that appears at the end of the drag wants to define the class of the outlet as UIView rather than UILabel or UIImageView or what have you, chances are the connection is being made to the Content View instead of thing we're really trying to connect. One way to be really sure is to start the drag from the subview's item in the scene's tree list, rather than from the storyboard view itself.

There's one more thing we need to do in the storyboard: The cells know they're 125 points tall, but the table doesn't, and will continue to assume the default row height of 44. Return to the Standard Editor mode, select the table, bring up its Size Inspector (⌥⌘5), and set the "Row Height" to 125. We could also

provide this height in code, which would in turn also make it possible for rows to be of different heights, but this easy approach is fine for now.

Now it's time to start populating these custom cells. Back in ViewController, we again need to update our method that dequeues and populates cells. Rewrite tableView(cellForRowAtIndexPath:) as follows:

Tables/PragmaticTweets-5-3/PragmaticTweets/ViewController.swift

```
Line 1 override func tableView (_tableView: UITableView,
  -   cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
  -     let cell =
  -       tableView.dequeueReusableCellWithIdentifier("CustomTweetCell")
  5         as ParsedTweetCell
  -     let parsedTweet = parsedTweets[indexPath.row]
  -     cell.userNameLabel.text = parsedTweet.userName
  -     cell.tweetTextLabel.text = parsedTweet.tweetText
  -     cell.createdAtLabel.text = parsedTweet.createdAt
 10     if parsedTweet.userAvatarURL != nil {
  -       cell.avatarImageView.image =
  -         UIImage (data: NSData (contentsOfURL: parsedTweet.userAvatarURL!))
  -     }
  -     return cell
 15 }
```

The first big change here is on lines 3-5, where we dequeue the cell with the identifier string CustomTweetCell and, since we know it's our custom cell, we can fetch it as the ParsedTweetCell class. Then, on lines 7-9, we set the values of the cell's properties that we connected.

Then we have the avatar image. The UIImageView has an image property we want to populate, of type UIImage. Unfortunately, UIImage can't be created directly from the contents of a URL. However, it will accept an NSData object containing image data, and we can initialize that with an NSURL, so we can chain those together. NSURL's contentsOfURL: requires a non-optional parameter, so we check the userAvatarURL against nil on line 10, and then load the image into the cell's image view on line 12.

Finally we return cell to the caller, just as before. The result looks like the following:

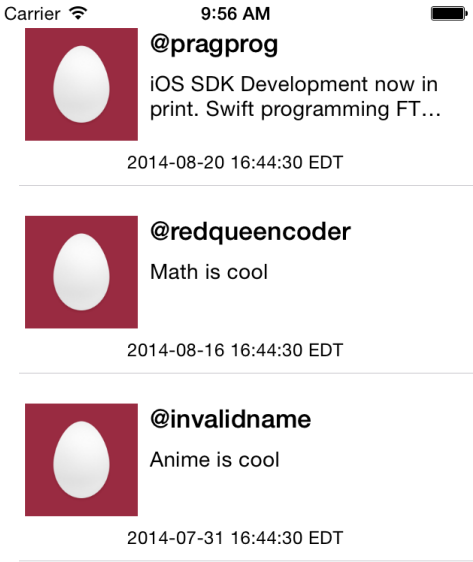


Figure 50—Displaying Custom Table Cells

This is looking a *lot* better. The avatar image is there, the author name is nicely set off from the tweet text and created-on date, and with a little tweaking — maybe a taller cell or a smaller font — we could completely eliminate instances of truncation.

There is one problem, though, and it's our images. If we have enough cells that we need to scroll — copy-and-paste a bunch of `ParsedTweet` initializers into the `parsedTweets` array initializer if you want to see it for yourself — the scrolling performance is pretty choppy. It's nowhere near as smooth as on a real iPhone or iPad, and this is running on the Simulator, where the power of a full-blown computer tends to run apps *faster* then they'll be on the device. So why is this happening?

The culprit is how we're loading our images. Our approach of loading the image data from the `NSURL` right when the `tableView(cellForRowAtIndexPath:)` needs it means that everything in our app just stops while we load that image from the network. We can't draw the cell, continue scrolling, or update anything else in the UI until the image data loads. Just imagine how well that's going

to go over with users who are only getting one bar of cellular signal. In the next two chapters, we'll fix this problem with a more sophisticated approach.

How I Gave Up Tables and Learned to Love Lists

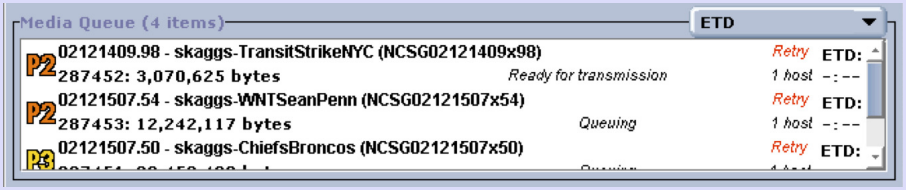
One of the things we haven't talked about is the fact that UITableView looks a lot more like a *list* than a *table*. Isn't a table supposed to be two-dimensional? With columns and headers, like a spreadsheet? That's what a table is in most other UI toolkits.

This takes me back to a job I had back in 2002 or so, building a Java UI for a Network Operations Center. The idea was it would feed out video clips, and the engineers there needed to see what was in the queue to go out, its priority, where it was going, and so on.

The first cut of the UI used a Java Swing JTable and, yeah, it did look like an Excel spreadsheet. I'd been playing with Mac OS X by this point — I was the only one in the office using it, and took a lot of crap for doing so — and realized that this was a lousy user interface. The wide table made it hard to trace a single item all the way across the window, and reading vertically down columns wasn't useful anyways. Also, JTable defaulted to making columns the same width, even when some of our fields were 4 characters while others were more than 40. It didn't need to be a table; it didn't work as a table.

So, I turned it into a list. I created custom cells that would group all the data together, using layout, fonts, and color to call out the stuff that mattered most, and how the items related to one another.

I liked it well enough that a decade later, with the company long gone, I still have screenshots. Here's one of the lists from the main status window:



The purple color scheme, programmer art, aliased text, and bland Windows NT fonts are hard on the eyes 10 years later, but at the time, this was a real breakthrough for us in how we thought about our in-house UI. You could easily see what was going out when, which items had priority, and why the queue was arranged the way it was. The cell layouts made the whole list more readable, and I've been a big fan of custom cells in lists and tables ever since.

Pull-To-Refresh

There's one other common table task we should attend to with our table: We haven't given the user a means of refreshing the tweets. We had this in the

last chapter with the “Show My Tweets” button, but now that our whole view is a table (for now, anyway), there’s no way to expose this functionality. It doesn’t matter now with our mock data, but it will matter a lot if we can’t get new tweets from the Internet. How are we going to fix that?

Many table-based iOS apps use a *pull-to-refresh* gesture, in which scrolling to the top of the table and then pulling down to scroll further is interpreted as a request for the app to refresh its data. There were a variety of third-party implementations of this gesture for a number of years, and in iOS 6, Apple provided a standard implementation in the form of the UIRefreshControl class.

The UITableViewController that our PRPViewController subclasses has a refreshControl property, meaning we inherit it, so all we need to do is to populate that with a refresh controller and default behavior will take care of the UI for us. In fact, the ease of using the refresh controller is one reason to choose to subclass UITableViewController, rather than to have a plain UIViewController that also happens to implement the UITableViewDataSource protocol.

Looking at the documentation, the UIRefreshControl is a subclass of UIControl and acts somewhat like a button or another generic control; when triggered, it sends an event called UIControlEventValueChanged. For us to do anything with it, we need to get a callback when that event occurs. We do that with the UIControl method addTarget(action:forControlEvents:). This method takes an object to call back to (such as our view controller), a method to call (which we’ll write), and the relevant events for this callback (UIControlEventValueChanged).

So let’s create and set up a suitable UIRefreshControl when our view controller first comes to life, in viewDidLoad:

Tables/PragmaticTweets-5-4/PragmaticTweets/ViewController.swift

```
override func viewDidLoad() {
  super.viewDidLoad()
  reloadTweets()
  var refresher = UIRefreshControl()
  refresher.addTarget(self,
    action: "handleRefresh:",
    forControlEvents: UIControlEvents.ValueChanged)
  self.refreshControl = refresher
}
```

The action, meaning the method that’s invoked by the callback, is passed as a *selector*, a string which uniquely identifies a method signature. In this case, we’re promising to write a method called handleRequest() that will take one parameter, as indicated by the single colon character. By convention, these action methods take a single argument to identify the sender, and have a return type of IBAction, so they can be used for connections in the storyboard

(so we could also connect it with the Interface Builder GUI rather than with code if we wanted to). So let's write this action method:

Tables/PragmaticTweets-5-4/PragmaticTweets/ViewController.swift

```
Line 1 @IBAction func handleRefresh (sender : AnyObject?) {
   2   self.parsedTweets.append(
   3     ParsedTweet (tweetText: "New row",
   4       userName: "@refresh",
   5       createdAt: NSDate().description,
   6       userAvatarURL: defaultAvatarURL))
   7   reloadTweets()
   8   refreshControl!.endRefreshing()
   9 }
```

This action method does three things:

- On lines 2-6, we append a new ParsedTweet to the parsedTweets array that serves as the table's data model. Notice that on line 5, we create a new NSDate object (which defaults to the current instant in time), and then use that class' description() method to turn it into a string.
- We call our existing reloadTweets() method on line 7.
- Finally, on line 8 we call endRefreshing() to tell the UIRefreshControl to hide the spinning wheel. UIRefreshControl also has a beginRefreshing() method to show the spinner, but this is called for us automatically as the UITableViewController processes the pull gesture.

When we run now, the spinner appears atop the table when we scroll to the top and pull again, as seen in the following figure. This starts a reload of the Twitter data and dismisses the spinner. If we're not following enough people for the reload to be obvious, we can temporarily comment out the reloadTweets() in viewDidLoad, so we'll have to pull to refresh when the app loads in order to see any tweets at all.

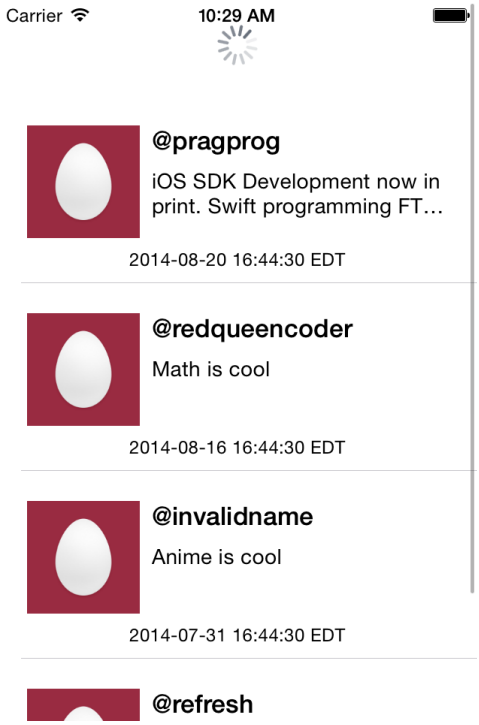


Figure 51—Pull-To-Refresh Activity Indicator

Wrap-Up

We've put our app through a radical make-over in this chapter, and in so doing we've turned it from a toy into something that's starting to resemble real-world Twitter clients. By switching to a table view, we've adopted what's arguably the most familiar and most useful iPhone user interface. We implemented the methods provided by UITableViewDataSource (which we inherited from UITableViewController) to structure our table data as sections, rows, and cells. We tried out the basic table cell styles, and then moved on to a custom cell approach that allows us to populate, lay out, and style the cell contents in whatever way best suits the contents.

Now that our user interface is ready, we're going to go out to the network to get real Twitter data to populate the table. This is going to introduce a bunch of new challenges with how (and *when*) our code runs, but along the way we'll actually solve the problem with how the image loading slows down our scrolling.

CHAPTER 6

Waiting for Things to Happen with Closures

It's very tempting to think of our code as a series of instructions, to be executed in order. But this falls down when any of these steps takes a long time, or worse yet, an *unknown* amount of time.

Imagine that instead of programming an iPhone, we're 50 years in the future, programming a household robot to do ordinary household tasks. Let's say we want to write a program to answer the phone (OK, and imagine there are still phones 50 years from now). We might write something like:

```
Pick up the phone.
Say "hello".
Wait for the other party to introduce themselves.
If they're a family member, let us know.
If they're a politician or an advertiser, hang up.
Otherwise, ask us what to do.
```

And so on. And this will be great, until we get a prank phone call that doesn't respond to our robot saying "hello". The script will wait on step 3. If the robot is really literal-minded, it will get hung up there forever, and won't attend to any of its other duties around the house.

We need the ability to express steps 4-6 as something to do once the waiting in step 3 is done. These tasks will be saved away for the future, to be performed when the other party finally responds (or maybe to be discarded when the call simply disconnects), while the robot can continue with other tasks in the meantime.

This is an example of *asynchronicity*, the occurrence of events in an unpredictable order or at unpredictable times. It's very important to us as developers, because it's a realistic model of how things happen in real life. We often don't know when things will happen or how long they will take. This is even true within the programming realm: we don't know when an event like a

button tap or a rotation gesture will occur, or how long it will take to get data from the network or write a document to the filesystem.

It's very expressive to be able to say "when event *foo* occurs, do *bar*", or "do *foo*, and if and when that finishes successfully, do *bar*, but don't wait around for it".

A lot of the iOS APIs are written with an expectation of asynchronous behavior. In this chapter, we'll learn how to write "closures" that function as completion handlers, meaning they do their thing only when some long-running or indeterminately-long task complete. To use the Twitter webservices, we'll have to use these in two different scenarios: asking the user to let our app use their Twitter account, and making the network call to Twitter, because one makes us wait for a user response and the other makes us wait for the network.

Setting Up Twitter API Calls

When we first set up our tweet-sending button, we found the documentation for the Social framework, and made use of the `SLComposeViewController`. To start using the rest of Twitter's features (or any other social network supported by iOS), we'll need to use another class in this framework. `SLRequest` lets us call the various social networks' web APIs by just providing a URL, whose contents vary by service and are documented at their various developer sites (such as http://dev.twitter.com). For Twitter, we have an additional detail to work through first: as of May, 2013, all Twitter requests need to be *authenticated*, meaning they need to come from a signed-in Twitter user.

Fortunately, iOS allows a user to sign in to her Twitter account from the Settings app, and the iOS SDK will allow us to use that authentication. The key to this is that the `SLRequest` includes a `account` property that represents an authenticated user. All we need to do is to set that property before we send off our request.

To use a social-networking account, we have to ask the `ACAccountStore` for access to it, by means of a `requestAccessToAccountsWithType()` method. And that raises an interesting question: what happens if the user says no? In fact, let's consider the worst case: that the user switches out of our app, goes to their privacy settings, and changes the permission setting for our app's access to Twitter *while our app is running*. We're basically going to have to plan on asking for permission to use Twitter every time we need to make a request. And as it turns out, that has some interestingly asynchronous behavior.

The first time we call requestAccessToAccountsWithType(), the user will be presented with an alert asking if she wants to grant our app access to her Twitter accounts, as shown below. We have no idea whether the answer will be "Don't Allow" or "OK", and certainly don't want to hold up the whole app waiting for an answer, so instead we'll make this call and move on with the rest of our app. If and when the user approves our use of their Twitter account, then we'll go ahead and call Twitter's webservice. Actually, the user will only ever see the alert once — after that, they can grant or deny access via the Settings app's "Privacy" settings — but our code won't behave any differently; the decision about whether or not to run our asynchronous code will just be made sooner.

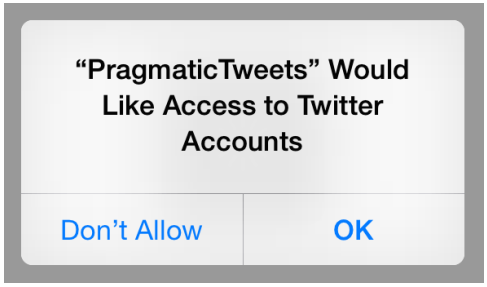


Figure 52—User alert when an app requests access to Twitter accounts via the ACAccountStore

Let's start by adding an import Accounts to the top of ViewController.swift, just like we did when we added the Social framework.

Now let's look at that requestAccessToAccountsWithType() method. It takes an ACAccountType, which has constants for Twitter, Facebook, and a few other services. The second argument is a dictionary of options whose use depends on the service we're using. And then the third parameter is of type ACAccountStoreRequestAccessCompletionHandler. That's new, so we click on its documentation and see this:

```
typealias ACAccountStoreRequestAccessCompletionHandler = (Bool, NSError!) -> Void
```

```
typealias ACAccountStoreRequestAccessCompletionHandler =
  (Bool, NSError!) -> Void
```

What...the...heck?

Encapsulating Code in Closures

Way back in *Swift methods*, on page 40, we mentioned that functions use the syntax -> to indicate the type of their return value. So, is that (Bool, NSError!) -> Void stuff a function? Actually yes… kind of! What this syntax expresses is a *closure*, a self-contained block of functionality. The syntax indicates what will be passed in (a Bool indicating if permission was granted, and an NSError optional in case the request for access totally failed) and what will be returned (Void, that is to say, there is no return value). So it's not that this parameter is a function; it's a closure… and in Swift, all functions (and thus all methods) are just a special case of closures!

So what's so great about closures? Well for one thing, it means we get to take some code *and treat it as an object.* In this case specifically, we get to write some code, give it to the ACAccountStore , and say "ask for permission to use the Twitter account, and run this when done". That's precisely what we want to do here: we want to write a closure that makes a Twitter request if and when the user OK's our use of their Twitter account.

Let's begin by rewriting reloadTweets to get the current Twitter account. We'll take it slowly and just concern ourselves for now with getting the Twitter account, not what to actually do with it yet.

Asynchronicity/PragmaticTweets-6-1/PragmaticTweets/ViewController.swift

```
Line 1 func reloadTweets() {
   -   let accountStore = ACAccountStore()
   -   let twitterAccountType = accountStore.accountTypeWithAccountTypeIdentifier(
   -     ACAccountTypeIdentifierTwitter)
     5 accountStore.requestAccessToAccountsWithType(twitterAccountType,
   -     options: nil,
   -     completion: {
   -       (Bool granted, NSError error) -> Void in
   -       if (!granted) {
    10         println ("account access not granted")
   -       } else {
   -         println ("account access granted")
   -       }
   -   })
    15 }
```

On line 2, we create an ACAccountStore object, which we'll need for the next few steps. The requestAccessToAccountsWithType() will require us to get the Twitter ACAccountType, which we do with a call to accountTypeWithAccountTypeIdentifier() on lines 3–4. Once we have that, we can ask for access to Twitter accounts, passing in the type (line 5), a set of options that can be nil for Twitter accounts (line 6), and a completion handler closure that will be called with the result

of our request, which receives a Bool to indicate if we were granted access, and an NSError that will describe any error associated with a failed request.

Our preliminary closure ranges from the opening curly brace on line 7 down to line 14, where it ends with a telltale sequence of characters — }) — which are the curly brace to end the closure, and the close parenthesis that ends the parameters to the requestAccessToAccountsWithType() call that began back on line 5. Inside this closure, our first decision to make is whether or not we can proceed; if the variable granted that's passed into the block is false, then we don't have access to Twitter accounts and should just give up. For now, on lines 9–10, we will just log a failure message in this case. Later on we should come back add a proper alert so the user knows what has happened.

On the other hand, if granted is true, we will be able to start talking to the Twitter API, which means a lot more work! For now, we'll just log a success message (on line 12).

Using the Twitter SLAccount

Now let's fill out the empty else case, replacing the simple success println(). We're going to work with the accountStore that the user has graciously given us permission to use. Because it's a local variable in scope at the time of the closure's creation, we can use it within the completion handler closure.

On iOS, the user may have set up several accounts of a given type; a fancier app would show them and let the user pick one, but for now, we'll just make sure there's at least one. We can use the ACAccountStore's accountsWithAccountType() to get all configured accounts of type twitterAccountType.

Asynchronicity/PragmaticTweets-6-1/PragmaticTweets/ViewController.swift

```
let twitterAccounts = accountStore.accountsWithAccountType(twitterAccountType)
if twitterAccounts.count == 0 {
  println ("no twitter accounts configured")
  return
} else {
```

Once again, we're just bailing out with an println() message to the console if there are no Twitter accounts configured. We can come back later and give the user a helpful UIAlertView dialog in this case.

Let's assume the array contains at least Twitter account. What do we do with it? A while back — before we had to work through getting access to the account — we noted there is a SLRequest class that accesses the webservice APIs of the social networks like Twitter. Looking at its documentation, it doesn't have a lot of methods, and one that we should focus on is performRequestWithHandler:(),

whose docs say it "performs an asynchronous request and calls the specified handler when done".

And look, it takes another closure! Well, actually it takes a SLRequestHandler, which if we click the link to its documentation, is defined as follows:

```
typealias SLRequestHandler = (NSData!, NSHTTPURLResponse!, NSError!) -> Void
```

So this is another closure, taking NSData, and NSHTTPURLResponse, and NSError as optional parameters and returning Void. We'll assume those parameters will give us everything we need to handle the response from the Twitter webservice. But since we'll probably have a bunch of work to do for that, and since we're already tabbed in pretty far in this else block, let's stub out a method to take those parameters and deal with the response when it comes in. Somewhere outside all the existing methods' curly braces — right before the class' closing curly brace would be a great place for it — let's stub out the following method, empty but for a simple println() that at least lets us know we got this far:

Asynchronicity/PragmaticTweets-6-1/PragmaticTweets/ViewController.swift

```
func handleTwitterData (data: NSData!,
  urlResponse: NSHTTPURLResponse!,
  error: NSError!) {
    if let validData = data {
      println ("handleTwitterData, \(validData.length) bytes")
    } else {
      println ("handleTwitterData received no data")
    }
}
```

Now we'll be able to have our performRequestWithHandler() closure just call this method, allowing us to put off for now just what's in the Twitter response and how we're going to deal with it.

Making a Twitter API Request

But what goes in our request? If we take a look at the Twitter REST API 1.1 at https://dev.twitter.com/docs/api/1.1, we'll find the call statuses/user_timeline, which is called via an HTTP GET, and which returns "a collection of the most recent Tweets posted by the user indicated by the screen_name or user_id parameters." Not shown on this page, but fundamental to Twitter API calls, is the fact that we can append .json to get JSON-formatted results, or .xml to get XML. Foundation's JSON parser is far easier to use than its XML parser, so our URL will be https://api.twitter.com/1.1/statuses/user_timeline.json. Note that as of January 2014, all Twitter API calls must use SSL, so our URLs will always start with https://.

So now we need to create an SLRequest.. The docs show us a single convenience initializer, which takes a service type, request method, URL, and a dictionary of parameters. We already know how to fill in these four parameters:

- serviceType: the constant SLServiceTypeTwitter
- requestMethod: the Twitter docs say we need an HTTP GET, so we use SLRequestMethodGET
- url: we already figured this out as https://api.twitter.com/1.1/statuses/user_timeline.json
- parameters: this is a dictionary of name/value pairs. The Twitter docs tell us we need to provide the screen_name or user_id parameters, so that's what will go in our dictionary.

Now we can fill in this inner-most else clause: we'll build up a call to performRequestWithHandler(), and have the handler block just call the handleTwitterData() method we stubbed out. Here's what goes inside the else:

Asynchronicity/PragmaticTweets-6-1/PragmaticTweets/ViewController.swift

```
Line 1 let twitterParams = [
   -   "count" : "100"
   - ]
   - let twitterAPIURL = NSURL.URLWithString(
   5   "https://api.twitter.com/1.1/statuses/home_timeline.json")
   - let request = SLRequest(forServiceType: SLServiceTypeTwitter,
   -   requestMethod:SLRequestMethod.GET,
   -   URL:twitterAPIURL,
   -   parameters:twitterParams)
  10 request.account = twitterAccounts[0] as ACAccount
   - request.performRequestWithHandler ( {
   -   (NSData data, NSHTTPURLResponse urlResponse, NSError error) -> Void in
   -   self.handleTwitterData(data, urlResponse: urlResponse, error: error)
   - })
```

There's a lot going on here! Let's take it slowly.

- Lines 1-3 set up a dictionary of parameters to provide to the request. The available parameters depend on the webservice's API. Twitter lets us send a count of how many tweets to return (the default is 20), so let's make things interesting and fetch 100.
- Lines 4-5 convert a String representation of the Twitter webservice URL into an NSURL, the type needed by the SLRequest initializer.
- Lines 6-9 creates the SLRequest with the URL and parameters we've set up, along with the constant for the Twitter service type, and the the SLRequestMethod enumeration value for GET requests.

- Line 10 gets the first object from the twitterAccounts array, and assigns it to the request's account property. Since the SLRequest wants an object of type ACAccount, and the array turns out to be of type AnyObject, we *cast* the value to the correct type with as ACAccount.
- Lines 11-14 finally performs our request. It takes a closure as its parameter, which is executed once the request finishes. We saw before that this closure is of type SLRequestHandler, which means it receives an NSData, NSHTTPURLResponse, and NSError as parameters. Inside the closure, we just pass these parameters to our handleTwitterData() method, to figure out later.

We've written a lot of code — and with closures two levels deep, this might be the hardest thing in the whole book so far — so let's run it to make sure it at least builds and starts up in the simulator. Once it does, tap the "Show My Tweets" button. Nothing interesting will happen in the simulator, but we should see a message like the following in the Debug Area at the bottom of the Xcode window:

```
handleTwitterData, 71706 bytes
```

This means the request is being sent to Twitter's webservice and being responded to. Now it's up to us to act on that response.

Parsing the Twitter Response

Inside our handleTwitterData() method, we receive the raw data from the Twitter API and can use it to update our UI.

We'll start by handing the raw data over to Foundation's NSJSONSerialization, which can easily produce either an NSArray or NSDictionary of the parsed data, an object that may itself be a deep structure of nested arrays and/or dictionaries. Let's do a quick sanity check by replacing the log statement with the following:

Asynchronicity/PragmaticTweets-6-1/PragmaticTweets/ViewController.swift

```
Line 1 func handleTwitterData (data: NSData!,
   -   urlResponse: NSHTTPURLResponse!,
   -   error: NSError!) {
   -     if let dataValue = data {
   5       var parseError : NSError? = nil
   -       let jsonObject : AnyObject? =
   -       NSJSONSerialization.JSONObjectWithData(dataValue,
   -         options: NSJSONReadingOptions(0),
   -         error: &parseError)
  10       println("JSON error: \(parseError)\nJSON response: \(jsonObject)")
   -     } else {
   -       println ("handleTwitterData received no data")
```

```
-     }
- }
```

We're doing a bunch of new things here, so let's take it slowly again:

- Lines 1-3 are our method declaration. As explained a while back in *Fast-Unwrapping Optionals*, on page 58, the bang characters (!), indicate the parameters are implicitly unwrapped optionals: They're still optionals, but we can access their values directly without converting to a non-optional type if we're *sure* they can't be nil. When Apple's frameworks call back to us like this, they usually send implicitly unwrapped optionals.
- On line 4, we create the non-optional local variable dataValue from the data parameter, which is an optional, since a JSON parse error could result in no data. Notice that we can do this unwrapping in an if statement to save a step, compared to casting to a non-optional type on one line and then testing against nil on the next. If data is nil inside this if statement, we bail out to the else case on line 12.
- The JSONObjectWithData()'s third parameter is listed as an NSErrorPointer. This is a pattern used by the older frameworks that could only return one value in C or Objective-C, even though they wanted to send back both data and maybe an error object , so we pass in a nil error object, and if an error occurs, the reference is changed to become an NSError object describing the error. However, it's a little unintuitive in Swift, at least for now: We don't pass an actual NSErrorPointer object. Instead, we create an NSError? optional on line 5, and the type will get converted for us when we pass it to JSONObjectWithData(). Note that since the point of this variable is to see if it changes from nil to having a value, we need to declare it with var to make it variable.
- Line 6 is the left side of an assignment that will be completed on the next few lines with our call to JSONObjectWithData(). That method returns AnyObject?, an optional, since it could give us an array, a dictionary, or nil, so that's the type we have to be prepared to receive.
- Lines 7-9 are the actual call to JSONObjectWithData(), passing in the data, behavior flags, and the error pointer. The behavior flags on line 8 are a Swift work-in-progress: the idea is to create a *bit-field* by mathematically OR'ing together behavior flags that each have one bit set, so multiple behaviors can be expressed with a single Int. This is a lot more natural in C and Objective-C than it currently is in Swift, but fortunately we don't need any of those behaviors, so we just pass 0 to the NSJSONReadingOptions struct.

- For the time being, we're done. On line 10, we'll use println() to print the parseError on one line, and the jsonResponse on a new line afterwards.

Run this, tap "Show My Tweets" and check the console at the bottom of the Xcode window. The result will be a deeply-nested structure, set off by tabs and curly-brace blocks, showing each tweet as a set of name-value pairs. The top of it will look something like this:

```
JSON error: nil
JSON response: (
        {
        contributors = "<null>";
        coordinates = "<null>";
        "created_at" = "Wed Jun 11 14:02:42 +0000 2014";
        entities =         {
            hashtags =             (
```

In the response, the first curly brace sets off all the data for one tweet, which has keys named contributors, "created_at", and so on. Notice that the value for entities is a curly brace, with its own child set of keys and values.

We can pick out interesting data like text, which is a string containing the tweet's text, and user, which is a dictionary of name-value pairs with items like screen_name and followers_count. Everything we could want to know about each tweet is in these entries, meaning we now have the data we need to populate the UI.

Wrap-Up

Because of the complexity of closures and asynchronous code, let's take a break here and assess what we've done.

We want to get at the raw Twitter data, so we ask for access to the user's accounts and wait for that to happen (since they might be blocked on an Allow / Don't Allow alert). If we are allowed to use the Twitter account, we send off a request, wait for that to come back, and then use NSJSONSerialization to turn the received NSData into an array of dictionaries, one entry per tweet. Both of the waiting parts are done with closures, telling the iOS frameworks what work we want them to do once they're able.

We've written two closures, both using the "completion handler" pattern that is common in iOS. In the next chapter, we're going to use closures again, this time to update the user interface. But instead of waiting for things to finish, these closures will allow us to do multiple things at once, which opens the door to a lot of cool possibilities.

CHAPTER 7

Doing Two Things at Once with Closures

We started the last chapter with the example of a hypothetical household robot, who would answer the phone for us. We dealt with the problem of prank callers that never respond to "Hello", by batching together all of our instructions for how to handle the greeting until after the caller responds. That leaves the robot free to do other tasks in the meantime. Now let's think about how that would work.

Some tasks will require the robot's limbs, some need its eyesight, and others its voice and hearing. If we're careful about how we divvy up tasks, the robot can do several things at once: We can prepare dinner while talking on the phone, and our robot should be able to as well.

So let's imagine we have lists of what each part of our robot's abilities can be working on: a list of manual tasks; a list of visual tasks; a list of spoken tasks. When it's time for the robot to continue dealing with the phone call, it can continue working on the manual tasks like cleaning or cooking, uninterrupted by the voice task of handling the call.

Our robot is a multitasking genius. And, if we're smart, our iPhone can be too.

In this chapter, we'll learn how Grand Central Dispatch offers the ability to break up work into distinct units — closures — and parcel that work out to whichever CPU core is most able to perform it at that point in time. We'll also see where the iOS frameworks force us to deal with concurrency, and successfully do so. With this skill in our toolbelt, we'll be able to keep our user interface fast and responsive as we get long-running tasks out of the way of the UI processing.

Grand Central Dispatch

Just like our hypothetical robot has lists of tasks to do with its hands, tasks to do with its voice, etc., iOS also has "lists" of tasks to be performed. They're called *queues*, as part of a work-dispatching system called *Grand Central Dispatch*, or GCD. The idea of GCD is that there are multiple queues of work, each with tasks to execute. The tasks are either C function calls, Objective-C blocks, or Swift closures. GCD can determine which tasks to execute based on the priority of the queue, whether the tasks are suitable for concurrent execution, how busy the CPU cores are, and other considerations.

Developers from other platforms will see an analogy to threading, and the queues are indeed performed by threads, but the difference in iOS is that the threads and their queues are managed by the system, which is in a unique position to best optimize the work. On other platforms, it's hard to reason about threads — if two threads are good, are four necessarily better? Maybe that's true on one CPU architecture, but not on another, and we can never know when we're coding. GCD takes responsibility for the problem and lets us off the hook: "Give me work to do" it says, "and I'll figure out how to best get it done."

GCD provides functions to create queues and to put work on them. The one we'll use the most is `dispatch_async()`, a function which takes a reference to a queue and a closure to execute on that queue. The `async` in the function name means that the call doesn't wait for the closure to finish executing; the related `dispatch_sync()` will actually wait until the closure finishes. Mostly, we'll want to use `dispatch_async()` so our app doesn't wait and instead can move on to other work.

Concurrency and UIKit

In fact, GCD is already splitting our work onto multiple queues. All our user interface events run on the *main queue*, the queue that launches the app and is responsible for listening for user interface events. When we get a button tap, the call into our code is made on the main queue. When a table asks our code for the number or rows or the cell at a given index path, it's on the main queue. In fact, UIKit has a rule: Calls to any method or property *must* be made on the main queue.

But when we perform certain other tasks, GCD will put that work on other queues. For example, since network calls are sometimes slow (and never predictable in how long they'll take), most of them are put onto other queues,

which allows the UIKit queue to get back to work processing user events and redrawing the views.

Building a New Table Model

So let's think of where we stand right now. By the end of *Parsing the Twitter Response*, on page 118, we had used the NSJSONSerialization to convert the raw JSON (as an NSData) into an array. If we inspect the output from the println(), we see that each member of the array is a dictionary that contains the text of the tweet, a created_at time (as a string), and a user value that is its own dictionary to give us things like the user's name, screen_name, a profile_image_url for their avatar, and so on.

Back in Chapter 5, *Presenting Data in Table Views*, on page 85, we built a ParsedTweet class to hold some of these values, and present them in our table view with with the custom cells. So our job now is pull values out of the array of tweets, put them into ParsedTweet objects, and use those to re-populate the parsedTweets array that serves as our table model.

So let's go back to our handleTwitterData() method, specifically the if block that ended with us just doing the println() to dump the tweet array to the debugging console. The first thing we'll do is bail out if there was an error.

Concurrency/PragmaticTweets-7-1/PragmaticTweets/ViewController.swift

```
if parseError != nil {
  return
}
```

Next, we want to start walking the JSON array of tweets, but we can't know for sure that it's really an array, since NSJSONSerialization could have produced a dictionary, if that's what the root object in the encoding is. To be safe, we'll use an if let to cast it to what we expect, and only enter the if block if it's safe to do so.

Concurrency/PragmaticTweets-7-1/PragmaticTweets/ViewController.swift

```
Line 1 if let jsonArray = jsonObject as? Array<Dictionary<String, AnyObject>> {
   2       self.parsedTweets.removeAll(keepCapacity: true)
```

It's not enough for something to be an array; Swift wants to know what's in the array. Put another way, it insists we say what it's an array *of*. We said above that the println() output shows us it's an array of dictionaries… which to Swift just begs the question "OK, buster, dictionaries of *what*?" We want to say it's a dictionary with strings for keys, but we really can't guarantee a consistent type for the values, since they can be strings, arrays, or dictionaries. So, our answer is "the array contains dictionaries with string keys and AnyObject

values." This is what we're doing on line 1, but the syntax merits an explanation.

Swift introduces the concept of *generics*, which allow us to specify the type of members of a collection. These are written with angle braces, like Array<String> to represent an array of Strings. In this case, our array of parsed tweets is described as Array<Dictionary<String, AnyObject>>, meaning "an array of dictionaries, each of which has string keys and AnyObject-type values".

Converting JSON Values to Swift Properties

Assuming this cast works, we enter the if block, and remove everything currently in the parsedTweets array, by use of the removeAll() provided for Swift arrays. Now we're ready to start walking the array.

Concurrency/PragmaticTweets-7-1/PragmaticTweets/ViewController.swift

```
Line 1 for tweetDict in jsonArray {
   2   let parsedTweet = ParsedTweet()
   3   parsedTweet.tweetText = tweetDict["text"]  as? NSString
   4   parsedTweet.createdAt = tweetDict["created_at"]  as? NSString
   5   let userDict = tweetDict["user"] as NSDictionary
   6   parsedTweet.userName = userDict["name"] as? NSString
   7   parsedTweet.userAvatarURL = NSURL (string:
   8     userDict ["profile_image_url"] as NSString!)
   9   self.parsedTweets.append(parsedTweet)
  10 }
```

Starting on line 1, we count over each tweetDict in the array, which we know is a dictionary because we only got into this if block by successfully casting as an array of dictionaries. The first thing we do in the loop, on line 2, is to create a new ParsedTweet. Instead of assigning its properties all at once with the designated initializer like we did before, we'll populate them one by one as we pull them out of the tweetDict.

Some of the values we want to put in our ParsedTweet are at the top level of the dictionary, so we assign tweetText and createdAt on lines 3 and 4, respectively. Unfortunately, the collections produced by NSJSONSerialization have values that can't be cast directly to Swift types. Presumably for legacy reasons, the contents of these collections are the formal classes that the Foundation framework offered in Objective-C: NSString, NSArray, NSDictionary, and so on. Fortunately, these can all be cast to their Swift equivalents for free, so even though we may need to ask for the tweet text as an NSString, we can simply assign it to our ParsedTweet's text property (defined as the optional type String?) with no additional effort. We'll chalk this up to iOS' growing pains as it changes languages, and hope it improves soon.

We face the same annoyance on getting the dictionary that describes the user who sent the tweet. On line 5, we have to get the value for the user key as an NSDictionary, but then we're free to use the resulting userDict local variable like a normal Swift Dictionary. We do so on line 6 to get the user's name, and on lines 7-8 to get their avatar URL as a string, and turn it into an NSURL.

Finally, with all the fields of the ParsedTweet populated, we add it to the parsedTweets array. This allows our table model to pick it up when it needs a cell for that row.

Refreshing the Table Model

In fact, after the for loop, all that's left to do is to tell the table to refresh its contents, and to close the if let jsonArray = jsonObject as? Array<Dictionary<String, AnyObject>> { block that gave us the tweets as an array:

Concurrency/PragmaticTweets-7-1/PragmaticTweets/ViewController.swift

```
  self.tableView.reloadData()
}
```

This leaves us at the else from before, the one that just contains println("handleTwitterData received no data"). So we should be good to go. Let's go ahead and run the app.

At first our table contains the three dummy values from before, then we see in the Xcode window as the app logs the raw JSON and its array-of-dictionaries representation. *And then it just sits there.* After about 10 seconds, the table finally updates with the tweets from our request, as seen in the following figure.

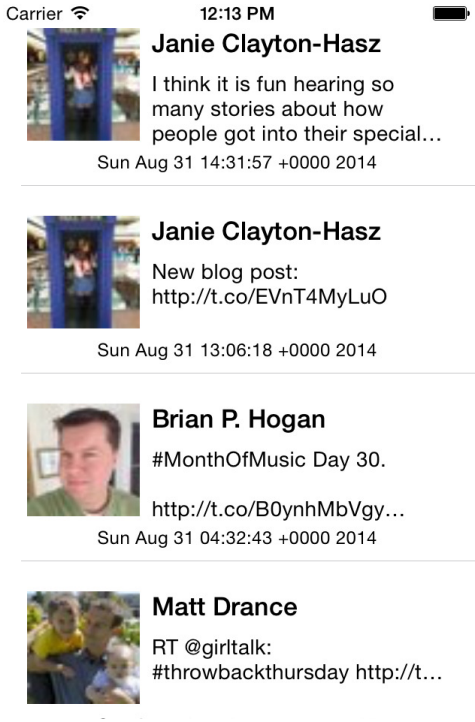


Figure 53—Table loaded from downloaded and parsed Twitter data

So, on one hand, it's a great success that our efforts of the last few chapters have finally paid off with a table of tweets pulled from the honest-to-goodness Twitter API. On the other hand, *why the heck is it so slow?*

Putting Work on the Main Queue

Early in this chapter, we were talking about queues and how they're used to keep the multiple cores of an iOS device busy. Maybe that's part of the problem.

Unfortunately, Swift won't tell us which queue is running our code, so a definite answer will have to wait until we play with breakpoints in Chapter 14, *Debugging Apps*, on page 221. But for now, there's an easy way to see how we've gotten ourselves in trouble. The main queue is run by the lower-level "main thread", and `NSThread` provides a class method `isMainThread()` to tell us if the current thread (and therefore queue) is main.

To try it out, plop the following line of code in any of the app's methods.

```
println (NSThread.isMainThread() ? "On main thread" : "Not on main thread");
```

Inside reloadTweets(), this will print On main thread. But in handleTwitterData() — or the completion handler closure of performRequestWithHandler() that calls it — and it will say Not on main thread. So, that's key to our problem.

Actually, we're kind of lucky. Back in Objective-C and iOS 7, this was a crashing bug. But crashing or suffering 10-second delays while updating will get us angry one-star reviews either way, so we need to fix it.

Our basic problem is that any calls to UIKit classes and their method must be made on the main queue, and handleTwitterData() is being called on some other queue. We're only doing one thing that touches UIKit — reloading the table — but that's enough to get us in trouble. We need a way to move at least that one line of code back to the main thread.

A Handy Concurrency Recipe

To do this, we need two things: a way to represent a chunk of code as an object, and a method that will take that code and put it back on the main queue. We already have the first of these: closures, which we used in the previous chapter. The other piece is the Grand Central Dispatch function dispatch_async(), which allows us to put work on a queue of our choosing, such as the main queue. So we have a recipe we can always fall back on:

```
dispatch_async(dispatch_get_main_queue(),
  {() -> void in
    // code to be performed on main thread
  })
```

dispatch_async() takes two parameters: a queue to perform the work on, and a closure with the work to be done. For the first, the point of this recipe is to use dispatch_get_main_queue(), so all we ever have to change is the contents of the closure.

So let's apply our recipe. Replace self.tableView.reloadData() with the following:

Concurrency/PragmaticTweets-7-1/PragmaticTweets/ViewController.swift

```
dispatch_async(dispatch_get_main_queue(),
  { () -> Void in
    self.tableView.reloadData()
})
```

Run the app again, and the table should reload as soon as the println() are done writing to the debug pane. In fact, take out the println()s that log the JSON and the array we make out of it, and this table should load in the simulator about a second after the app launches.

This is *much* better. Our app does its network stuff on one queue so that it doesn't block the GUI; we can unpack our data on that other queue and only touch the main queue for an update when we're good and ready. It's concurrency in action, and when we're smart about it, our apps can stay nimble and responsive, which makes our users happy.

Do-It-Yourself Concurrency

Actually, our app isn't as fast as we might like. Try scrolling the table. The scrolling is still choppy. This has been the case since way back in *Custom Table Cells*, on page 101, where we started fetching the avatar images from their URLs. So let's think about what's causing the problem and whether we can fix it.

When the table asks us for a cell — in `tableView(cellForRowAtIndexPath:)` — we can easily set all the labels with strings from the `ParsedTweet`, but what we have for the avatar image is an `NSURL`. So we stop and load the data for that URL, make a new `UIImage` from it, and assign that to our custom cell's `UIImageView`. This has to happen for each cell. Moreover, we can only work on one cell at a time. As a new cell comes into view, we have to wait to download the image data, and only when we have it can we continue on to the next cell. It makes swiping quickly through the table impossible.

So, we're blocking the UIKit queue on a slow network access. "Hey, wait a minute", we say, "isn't that exactly what concurrency is supposed to fix? And isn't it exactly why the Social framework does the Twitter API call on a different queue?" Exactly. And that means to fix our problem, we should do what Apple does: *get our network stuff off the main queue*.

They Don't Call It "Blocking" The Main Queue For Nothing

Lest anyone think the issue of keeping long-running tasks off the main queue is an academic problem… well, do we have a story for you.

Years ago, one of the authors of this book was working at a company with a product that worked with video. For a demo, we had to show that the application could copy this video to an analog video tape recorder (VTR). Our solution was to connect the output of the video card to the VTR, and to use an RS-232 cable to send "record" and "stop" commands to the VTR. It seemed easy: To copy the video, we start the VTR recording and play the video from the PC, and then stop the VTR when the video's done. Easy peasy.

Except that the guy who wrote this didn't know how threads work in Java, which is what the application was written in. And desktop Java works almost exactly like UIKit: there's a main thread with an endless loop that looks for events like keypresses

and mouse clicks, sends them to any code that handles the event, and repaints the window.

So when the user clicked the "Record" button, the code to play the video and start recording on the VTR was called… on the main thread. And that code effectively said "wait here until the video is done", which meant that the window didn't update and no further events were processed until the video was done playing.

Some of these videos were 15 minutes long. The application couldn't do any repainting or event-handling during this time, so if you covered up the window and then foregrounded it, it wouldn't repaint. On Windows, dragging the mouse over the window would leave a trail of un-erased mouse crud. Clicking a button did nothing. It was a disaster.

And this is pretty much where your author got to learn about threads, and had to completely rewrite this part of the program so that all of the video stuff happened on another thread, freeing up the main thread to immediately get back to work processing events and repainting, and then having the video thread put UI work back on the main thread only when ready.

And if you're still not convinced? Try plopping a `NSThread.sleepUntilDate(NSDate(timeIntervalSinceNow:900.0))` as the first line of one of the button handlers. This will block the main queue for 900 seconds, or 15 minutes, during which time the button won't return to its untapped state, rotation events will be ignored, and the user will basically be blocked out of the app. *That's* what we're trying to avoid!

Moving Work Off the Main Queue

When `tableView(cellForRowAtIndexPath:)` needs an avatar, it does a slow `NSURL` load, makes an image from it, and sets it on the `UIImageView`. Only the last of these steps needs to be on the main queue, and the first shouldn't be. So we need a recipe to move work *off* the main queue.

`dispatch_async()` comes to our rescue again. Recall that it takes two parameters: the queue to put work on, and a closure with the tasks we want performed. What we need now is a different value for that first parameter, one that isn't the main queue, but just some other queue. For this, there's the GCD function `dispatch_get_global_queue()`, which takes a constant that indicates the priority of the system-provided queue we want. We're not picky, so we can use `DISPATCH_QUEUE_PRIORITY_DEFAULT` to let GCD pick an ordinary background queue for us.

So now we have the pieces we need. In `tableView(cellForRowAtIndexPath:)`, find the `if parsedTweet.userAvatarURL != nil` block that sets the image, and replace it with the following version:

Concurrency/PragmaticTweets-7-2/PragmaticTweets/ViewController.swift

```
Line 1 dispatch_async(dispatch_get_global_queue(
   2   DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
   3   {() -> Void in
   4     let avatarImage = UIImage(data: NSData (
   5       contentsOfURL: parsedTweet.userAvatarURL!))
   6     dispatch_async(dispatch_get_main_queue(),
   7       {
   8         cell.avatarImageView.image = avatarImage
   9     })
  10 })
```

Lines 1-10 are one big dispatch_async() call. The difference here is that we want to get work *off* the main queue, so on lines 1-2, we use the GCD function dispatch_get_global_queue() with the constant DISPATCH_QUEUE_PRIORITY_DEFAULT to let GCD pick an ordinary background queue for us. That background queue gets the closure that runs from lines 3-10. This closure contains the "get a UIImage from an NSURL" logic from before, and then sets that image on the UIImageView. But since updating the image view has to happen on the main queue, we use a second dispatch_async() (lines 6- 9) to wrap the UIKit work with a closure and put it back on the main queue.

And it's great! Now our table scrolls nice and fast, not blocking on the image loading at all!

There's just one more problem. Look at the following figure. Every single one of the images is wrong: Chris is Janie, Janie is Chris, and *iOS Recipes* co-author Matt Drance is Janie, too.

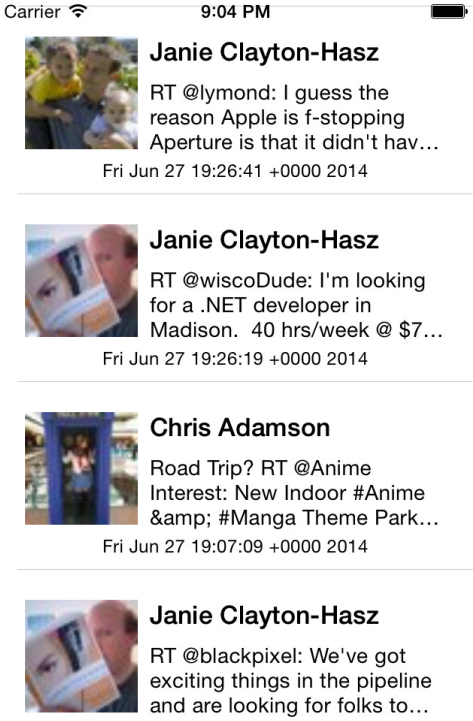


Figure 54—UIImageView in Custom Cell Showing Wrong Images

Race Conditions

What's happened? A *race condition*, actually. When a cell goes offscreen and is queued for reuse, it will eventually get dequeued and filled with new data. *But the closure doesn't know that.* In this case, there was some cell for one of Matt's tweets that went off screen, dequeued and repopulated with one of Janie's tweets (for the first row in the figure), but then the closure finished and filled in the image with Matt's picture. This doesn't happen often — we had to request 200 tweets, plus simulate poor network conditions to get the screenshot — but it *is* a bug, and if there's any way to make it happen in development, it's for sure going to hit someone in the real world.

The fix is to figure out when a closure has taken too long. How do we know that? Well, if the problem is that the cell has already filled in the contents from a different tweet, we can look to see if the `parsedTweet` that the closure started with has the same data that's displayed by the cell now.

Concurrency/PragmaticTweets-7-2/PragmaticTweets/ViewController.swift

```
Line 1 cell.avatarImageView.image = nil;
   - dispatch_async(dispatch_get_global_queue(
```

```
-  DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
-  {() -> Void in
5    let avatarImage = UIImage(data: NSData (
-      contentsOfURL: parsedTweet.userAvatarURL!))
-    dispatch_async(dispatch_get_main_queue(),
-      {
-        if cell.userNameLabel.text == parsedTweet.userName {
10         cell.avatarImageView.image = avatarImage
-        } else {
-          println ("oops, wrong cell, never mind")
-        }
-    })
15 })
```

The change is inside the closure that runs on the main queue (lines 7-14): It looks to see if the text already set on the name label matches the userName of the ParsedTweet that the closure captured at the moment the closure was created. If it does, then this image belongs with this cell. If not, then the cell the closure was downloading an image for has already been reused and no longer matches, so the closure can just bail.

The else block is optional of course, but it's interesting to play with our network conditions and see how often the log message pops up in good conditions versus bad (for a way to reproduce this, see *Can I slow down the simulator long enough to see the cells get the wrong image?*, on page 132). Suffice to say that if we hadn't fixed this, our Edge and 3G users would be *really unhappy.*

Joe asks:

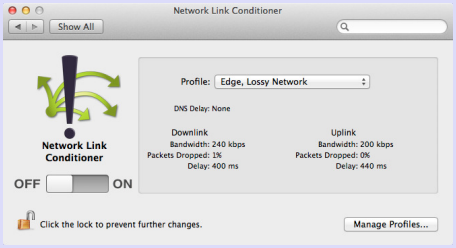
Can I slow down the simulator long enough to see the cells get the wrong image?

If your internet connection is really good, you may load the image data too fast to see the wrong-cells bug. This shows off one disadvantage of working with the simulator: Its performance is unrealistically good, particularly for networking tasks. A Mac Pro with gigabit ethernet is going to get a webservice response a lot more quickly than an iPhone with one bar of 3G coverage out in the woods somewhere.

Fortunately, a Mac can simulate lousy network conditions for this kind of testing. From the Xcode menu, select Open Developer Tool → More Developer Tools… to be taken to Apple's Xcode downloads page. After asking for a developer ID and password, the page shows optional downloads for Xcode. Look for the latest version of the "Hardware IO Tools For Xcode", download it, and double-click the Network Link Conditioner.prefPane to install it.

This adds a pane to the Mac's System Preferences called "Network Link Conditioner", which adjusts the performance of the Mac's current networking device (ethernet,

AirPort, etc.) to resemble real-world conditions an iOS device might face, from WiFi with good connectivity to the outdated Edge network experiencing packet loss.



Keep in mind, however, that the Network Link Conditioner degrades *all* network traffic on the Mac, not just the iOS Simulator application, so if we forget to turn it off when we're done testing, it will make everything we do seem like we're getting one bar in the middle of nowhere.

Now the race condition is fixed: If the image data comes in too late to use, we just don't use it. And we've once again been reminded of the promise and the hazards of working asynchronously. The following figure shows our snappy and accurate app:

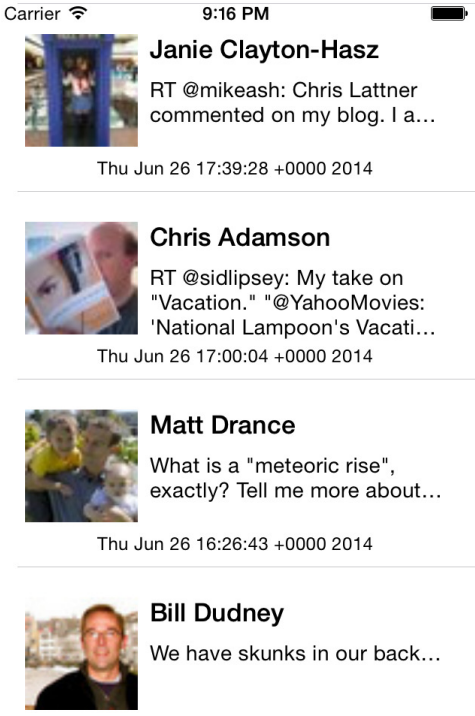


Figure 55—Images in Custom Cells With Race Condition Fixed

So we have a recipe for getting work onto and off of the main thread: Just call dispatch_async(), with the work to be done as a closure. For the queue, we use dispatch_get_main_queue() to put work on the main queue, or dispatch_get_global_queue() to get a system queue that can get our work off the main queue. Either way, we're exploiting concurrency, the ability of the system to do many things at once, and now we're smarter about how to let the main queue keep doing its event-dispatching and repainting thing, while we do ours.

Wrap-Up

In this chapter, we furthered our command of how to determine not just "what" to run, but "when" and "how". We built on last chapter's introduction of closures as an object wrapper for code, and used Grand Central Dispatch to put our closures onto the main thread when they need to access UIKit classes and methods, and get them off the main queue when they need to get out of main's way. Between the many built-in APIs that are designed for asynchronicity and concurrency, and our own ability to make things concurrent with GCD, we've got great tools to keep our app snappy.

We've now got a pretty full screen with this table of tweets, but we still want to do a lot more with our app. The only way to do that is going to be to start having several screenfuls of information and navigate between them.

CHAPTER 8

Growing Our Application

So far, our app interface has been restricted to a single view. We've swapped different functionality into and out of this view, but ultimately the small space of an iPhone screen limits what we can do in a single view. In this chapter, we're going to broaden our world by adding the ability to move among multiple view controllers, each of which will provide a unique interface and functionality for different concerns of our app. To do this, we'll learn about how iOS uses navigation controllers to help the user move between these different view controllers. This will open up our app to an almost unlimited potential for functionality; if we want to add a new screen, we just need to write a new view controller and have a way to get to it.

Working With Multiple View Controllers

With our switch to a table view as the main interface for our Twitter app, we're starting to resemble and work like the many other Twitter apps on iOS. However, our functionality is limited: All we can do is load and display the tweets. In fact, we've actually *lost* the "Send Tweet" functionality, because the full screen table view doesn't afford a good place for its button. So what are we going to do?

Usually, iOS tables *do something* when the user clicks on a table row. And most of the time, that something is to show the details of the thing that was clicked on, with a new interface appropriate to the detail view.

What's happening is that iOS is presenting a different view controller, one that is designed specifically for the task at hand. So for our app, that means we want to go from the view controller that shows all the tweets to one that shows just the specifics of one tweet. From here, we could go to another view controller: for example, we could click on the tweeter's profile image to go to a view controller that shows details about him or her. Each view controller

is built for one task — showing all the tweets, showing the details of one tweet, showing the details of the Twitter user — which allows us to divvy out functionality to different classes within our codebase.

To start adding new view controllers to our application, we'll want to make a few changes to our source code. We need the ability to arbitrarily grow our application by adding new classes and new storyboard scenes, and we need to start thinking about where we are going to put new code, and where we have opportunities for code reuse. All in all, it's a good time to tackle some much needed refactoring.

Refactoring in Xcode

Refactoring is the disciplined practice of making small changes to a code base that changes its internal structure without changing its perceived behavior. Xcode offers a handful of refactoring tools; the rest we'll have to do by ourselves.

Where do we start? Well, we've sketched out our idea above for using multiple view controllers, and once we've gone ahead and done that, the default name of our current `ViewController` is going to be a liability, since we might well ask, "*which* view controller?" Let's rename it to clear up any future confusion.

Renaming

What should we call it? Looking at its functionality, we could call this something like `TweetListViewController`. However, in navigation-based apps, we typically refer to the first view controller as the *root view controller*, so let's use `RootViewController` as our new name.

We might be tempted to just change the name of the file in the Finder or Xcode's File Navigator, but this would cause all kinds of breakage, since other files in the project would still be looking for `ViewController` files. And finding all those references, particularly in the storyboard, is a tedious and error-prone process. Instead, we'll have Xcode do the name change for us.

Switch to `ViewController.swift`, find the `class ViewController : UITableViewController` at the top of the file, and select the `ViewController` name. Bring up the "Refactor…" menu, either from the "Edit" menu or from the popup menu (via a control-click or right-click). The Refactor menu includes options to rename a selection, create a superclass, extract code into its own method, and a few others. What we want to do is to rename, so select "Rename…".

Oh no! An error message! Apparently, Xcode isn't yet able to refactor Swift code. We're willing to bet that it will be able to before long (hopefully before

Xcode 6 goes golden master), so keep in mind that the Refactor menu item is there. But for now, we'll have to do it by hand, just like we said we didn't want to. Ugh!

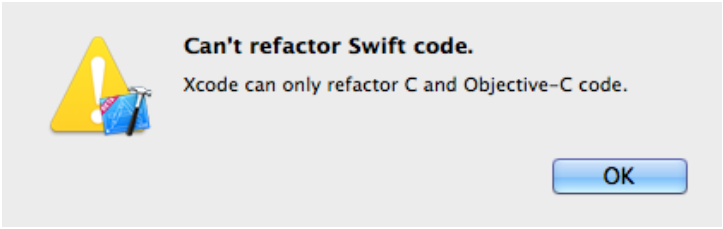


Figure 56—Error message when refactoring Swift code

With the class name ViewController still selected, change the name in the class declaration to RootViewController. For consistency, we want the file name to match its contents, so in the file navigator on the left (⌘1), select ViewController.swift, click again to edit the line, and change its name to RootViewController.swift

So far, that's two steps that a working Refactor menu item would have saved us, but there's one more. The storyboard still thinks that its one scene has a view controller of class ViewController. But that class no longer exists, so if we run now, our table is empty and the debug pane shows the error message Unknown class ViewController in Interface Builder file.

To fix this, go to the storyboard, select the "View Controller" in the scene list, and visit the Identity Inspector (⌥⌘3) in the utility pane on the right. The first field here is "Custom Class", which we used before when we were telling the storyboard to use our custom table cell class. Now we want to re-connect it with our custom view controller class, so change the value of the field to RootViewController. Save up, run, and the app works again.

The lack of a working "Refactor" menu item for Swift is a hassle, but at least we have a recipe: rename the class in its source file, rename the source file, and then use the identity inspector to rename any occurrences of it in the storyboard.

Snapshots

The first time we perform a refactoring — when it works, that is — we're asked immediately before applying the changes if we want to enable automatic *snapshots* of our project before making this change. Snapshots let us save the entire state of the project and all its contents, so if we make sweeping changes that don't end up working out, we can access the snapshots of previous states of the project (via the

Organizer window) and roll back to the a previous working state. It's similar to reverting back to earlier versions of files with Time Machine, but is controlled entirely within Xcode, and lets us label each snapshot so we know what it represents.

This is a simple enough means of keeping ourselves out of trouble, but it's limited: It only works for one user on one computer. For now, disable snapshots when asked, and we'll adopt a more comprehensive solution in Chapter 15, *Publishing to the App Store*, on page 223.

Organizing Xcode Projects with Groups

So we can rename our classes, and that's great. But if all we could do to keep our files straight was to use naming conventions, the contents of the File Navigator would still become hard to read, once we have dozens or even hundreds of classes.

One way to manage our code is to create groups. These are the folder icons in the file navigator, several of which were created for us by the Xcode template when we started the project. We can move files into groups to organize them and strategically show and hide them, to make our project easier to manage. Note that these aren't real folders on the filesystem, they're just an organizational tool within Xcode.

A common convention in iOS development is to create a group for a view controller and any custom classes used only by that view controller. For us, that would be the newly-renamed `RootViewController` and the `ParsedTweetCell`. Let's do that. Click on the "PragmaticTweets" folder, to indicate that's the group we want as the parent of our new group, and select File →New→Group. This adds a group folder with the name "New Group". Rename it to "Root VC" and then drag the `RootViewController.swift` and `ParsedTweetCell.swift` files into it. The folders in the File Navigator should now look like the following figure:

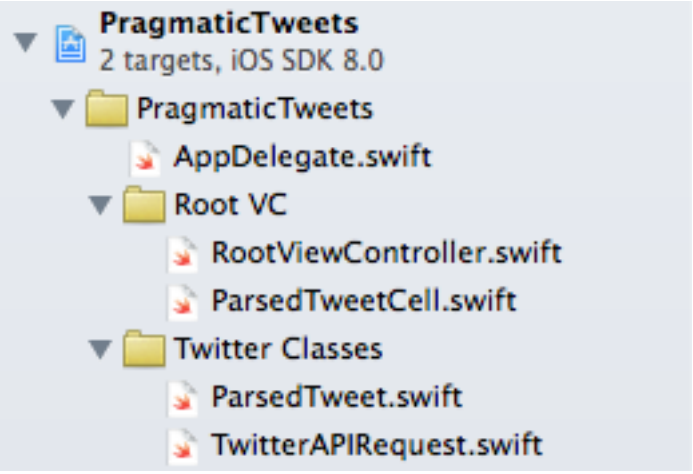


Figure 57—Source files in groups in File Navigator

The group's files are indented slightly and can be hidden entirely by turning the disclosure triangle on the side of the group. As we create new view controllers, we'll put them in their own groups, and we can expand just the groups we're interested in at a given time, so we don't see a bunch of files that we're not working on at the moment.

Extracting Method Code

When we're ready to write these other view controllers to show tweet details and user information, we're going to need to make new calls to the Twitter API. And considering all the work we did to get our first call working — talking to the ACAccountStore and getting an account and using it to make a request and so on… — we really don't want to repeat all that, right? But right now, that code is all in RootViewController. To make it more general purpose, we're going to need to *extract* it, and then generalize it.

To do this, let's think about how to make a more generic Twitter API caller. The current reloadTweets() method uses the ACAccountStore to construct an SLRequest, then calls its performRequestWthHandler(), which then calls back to our handleTwitterData() in the completion handler closure. The things that are specific to RootViewController are the URL and the parameters sent to the SLRequest (in this case, they specify the home_timeline.json call and its parameters), and the response handling, which is all in another method (handleTwitterData()). Put another way, everything in reloadTweets() other than the URL and the parameters is something we would do for *any* Twitter request, and is therefore reusable.

So what we can do to refactor is to move this code to a general-purpose version that can be called with any URL and parameters, and replace reloadTweets() with a one-line call to this new method, passing in the current URL and parameters. Everything we do in the response is already factored out into handleTwitterData(), so we don't need to change anything there.

The big difference is that the generic Twitter request method should be in a new class, so classes other than RootViewController can call it. Let's start by creating a whole group for the general-purpose Twitter classes. Do a File→New→Group to create another new group, and call it "Twitter Classes". Move the files for the existing ParsedTweet in here, as that's something we'll be sharing between multiple view controllers.

Select the new group and do File→New→File… to create a new class file. Choose the iOS →Cocoa Touch → Class template, and name the new class TwitterAPIRequest, a subclass of NSObject, with "Language" set to "Swift".

This is the class where we'll move all that code that gets the ACAccount and sends off the SLRequest, stuff we don't want to repeat in lots of places in our app. But of course, that's only one half of what we do with the Twitter API…

Building Our Own Delegate

Let's think ahead: most of our Twitter requests are going to be pretty much the same thing: get a Twitter ACAccount, send off a URL with some parameters. The only thing that's different is the URL and the parameters, and that's pretty easy. But the other thing our RootViewController does is all the stuff in handleTwitterData(), where it currently goes through the array of tweets to make a table model.

When we're ready to make a view controller to show the details of a tweet, or a user, or search results, or some other kind of response, that's going to be completely different code. So the response-handling code can't live in TwitterAPIRequest. In fact, for the app we have now, we want to leave that stuff in RootViewController, so it can populate the table. But we also want TwitterAPIRequest to be able to send its response to lots of different kinds of classes.

What we're going to do here is what so many of the iOS frameworks do: we're going to have a *delegate* that handles parsing the Twitter response data. Indeed, here we can see how the delegation pattern gets its name: our general-purpose Twitter request-maker doesn't know what to do with the response it gets for some arbitrary Twitter API, so it *delegates* that responsibility to another object.

To set up a class to work with a delegate, we typically create a protocol defining all the delegate methods. Select the "Twitter Classes" croup and do File→New→File… again. This time we don't want a full-blown "Cocoa Touch Class", so we have to choose the generic "Swift File", as seen in the following figure.

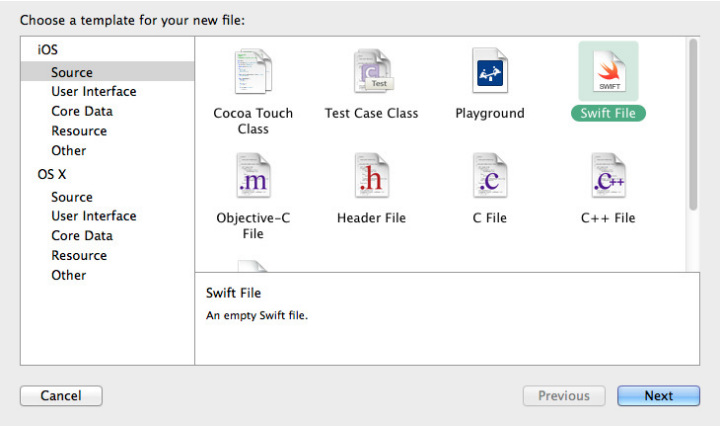


Figure 58—Creating a generic Swift file

Name the protocol TwitterAPIRequestDelegate and save it. This creates a .swift file that's empty, but for a copyright comment and a import Foundation statement.

Since protocols are just declarations of methods, with no method body, all we have to do is provide that declaration. Here's what we need to enter here:

Growing/PragmaticTweets-8-1/PragmaticTweets/TwitterAPIRequestDelegate.swift

```
Line 1 protocol TwitterAPIRequestDelegate {
   2     func handleTwitterData (data: NSData!,
   3      urlResponse: NSHTTPURLResponse!,
   4      error: NSError!,
   5      fromRequest: TwitterAPIRequest!);
   6 }
```

The protocol declaration starts on line 1 by simply providing a name for the protocol. This is followed by the method definition for handleTwitterData(), whose parameter list is exactly the same as the version we currently have in RootViewController, except that it adds a fourth parameter. This last parameter, fromRequest, identifies the object that's making the delegate callback. If some part of our yet-to-be written code needed to send off several TwitterAPIRequests at once, its implementation of handleTwitterData() could use this value to tell which was which. We saw this earlier in how the UITableViewDataSource and

UITableViewDelegate protocol methods typically include a parameter to say which table is calling back.

Genericizing the Twitter Code

Now that the delegate protocol is declared, we have the pieces in place for our genericized Twitter calls: callers will provide the TwitterAPIRequest with a URL and parameters for their request, and a delegate implementing TwitterAPIRequestDelegate to handle the response. The delegate is often the caller itself.

So, at the top of TwitterAPIRequest.swift, we'll start by importing the Accounts and Social frameworks, since we'll be using classes from both of them.

Growing/PragmaticTweets-8-1/PragmaticTweets/TwitterAPIRequest.swift
```
import Social
import Accounts
```

Now, inside the curly braces of class TwitterAPIRequest: NSObject, here's how we start declaring our generic Twitter-calling method:

Growing/PragmaticTweets-8-1/PragmaticTweets/TwitterAPIRequest.swift
```
func sendTwitterRequest (requestURL : NSURL!,
  params : Dictionary<String, String>,
  delegate : TwitterAPIRequestDelegate?) {
```

This method declaration takes the three parameters we discussed above: the Twitter URL and parameters for the request, and a delegate to handle the response. Notice the third argument is an optional of type TwitterAPIRequestDelegate, the delegate we just declared. That means any object provided for this argument has to implement the handleTwitterData() method that we just defined in the TwitterAPIRequestDelegate protocol.

Now we're ready to write the implementation. It's a lot of code, but it should look *very* familiar:

Growing/PragmaticTweets-8-1/PragmaticTweets/TwitterAPIRequest.swift
```
      let accountStore = ACAccountStore()
      let twitterAccountType = accountStore.accountTypeWithAccountTypeIdentifier(
        ACAccountTypeIdentifierTwitter)
      accountStore.requestAccessToAccountsWithType(twitterAccountType,
        options: nil,
        completion: {
          (Bool granted, NSError error) -> Void in
          if (!granted) {
            println ("account access not granted")
          } else {
            let twitterAccounts =
              accountStore.accountsWithAccountType(twitterAccountType)
            if twitterAccounts.count == 0 {
```

```
            println ("no twitter accounts configured")
            return
          } else {
            let request = SLRequest(forServiceType: SLServiceTypeTwitter,
              requestMethod:SLRequestMethod.GET,
              URL:requestURL,
              parameters:params)
            request.account = twitterAccounts[0] as ACAccount
            request.performRequestWithHandler ( {
              (NSData data,
              NSHTTPURLResponse urlResponse,
              NSError error) -> Void in
              if delegate != nil  {
                delegate!.handleTwitterData(data,
                  urlResponse: urlResponse,
                  error: error,
                  fromRequest: self)
              }
            })
          }
        }
    })
  }
}
```

That's a lot of code. We can write it fresh, or just copy and paste the contents from reloadTweets() in RootViewController, with the following changes:

- Remove the line that creates the twitterParams local variable.
- Remove the line that creates the twitterAPIURL local variable.
- In the SLRequest initializer, replace the twitterAPIURL and twitterParams local variables with the requestURL and params arguments.
- In the handleTwitterData() call, change the target from self to delegate, and add the fourth parameter, fromRequest: self.

Now we have a generic Twitter API request-maker that can be used by any and all view controllers that will want to make Twitter requests. To try it out, we'll finish our refactoring by having RootViewController use this class instead of its current code.

Back in RootViewController.swift declare that the class implements the delegate protocol, by appending TwitterAPIRequestDelegate to the class declaration:

Growing/PragmaticTweets-8-1/PragmaticTweets/RootViewController.swift

```
class RootViewController: UITableViewController, TwitterAPIRequestDelegate {
```

As soon as we do this, we start seeing an error icon in the left column, because we don't actually implement the TwitterAPIRequestDelegate protocol. We do have a handleTwitterData() method, but it takes three arguments, and our delegate expects four. So just update the method signature to include the fourth argument:

Growing/PragmaticTweets-8-1/PragmaticTweets/RootViewController.swift

```
func handleTwitterData (data: NSData!,
  urlResponse: NSHTTPURLResponse!,
  error: NSError!,
  fromRequest: TwitterAPIRequest!) {
```

Now we can rewrite a *much* simpler reloadTweets() to take just the arguments relevant to what this view controller needs, namely the user's home timeline:

Growing/PragmaticTweets-8-1/PragmaticTweets/RootViewController.swift

```
func reloadTweets() {
  let twitterParams : Dictionary = ["count":"100"]
  let twitterAPIURL = NSURL.URLWithString(
    "https://api.twitter.com/1.1/statuses/home_timeline.json")
  let request = TwitterAPIRequest()
  request.sendTwitterRequest(twitterAPIURL,
    params: twitterParams,
    delegate: self);
}
```

Run it and… nothing's changed! And that's exactly what we want! The point of refactoring is to change the code structure while maintaining the same apparent behavior, and that's just what we've done. Only now, it will be easier to grow the project, since much of the Twitter-specific code (and everything relating to the Accounts framework) is no longer in this view controller class, and what's left is directly related to the specifics of this VC's Twitter request, and the specifics of the handling the response. The latter is still in handleTwitterData(), unchanged but for the addition of a currently-unused parameter.

Now that we've completed this refactoring, any other view controllers we write that need to call the Twitter API can use code much like what's now in reloadTweets() in that they'll just need to provide a URL and a parameters dictionary, and parse the response in the delegate callback. The details will be specific to the Twitter API they're calling (tweet details, user info), but that's exactly what we'd want anyways. So let's start building out those new view controllers.

Using Another TwitterAPIRequest

To prove all this refactoring has been worth it, let's see how much easier it'll be to add Twitter calls to our existing classes or those we may yet create.

Just for kicks — and with the understanding we'll undo this silly code in a few minutes when we're done with it — take a look at AppDelegate.swift. This class has to do with how our app interacts with the rest of the application, something we'll talk about more in Chapter 13, *Launching, Backgrounding, and Exensions*, on page 219. For now, notice there is a method called applicationWillEnterForeground(). This is called when the app is brought back to life from the background. Let's do a Twitter call every time this happens.

To do so, we will be using a TwitterAPIRequest, and making our caller a TwitterAPIRequestDelegate to handle the response. So change the AppDelegate declaration as follows:

Growing/PragmaticTweets-8-1/PragmaticTweets/AppDelegate.swift
```
class AppDelegate: UIResponder, UIApplicationDelegate, TwitterAPIRequestDelegate {
```

This just adds TwitterAPIRequestDelegate to the list of protocols the class must implement, in addition to the UIApplicationDelegate that was already there.

Now we need a simple Twitter API to call. A simple one is users/suggestions, which will send us some suggested topics, based on who we follow. Thanks to our TwitterAPIRequest class, asking for this is really simple; just rewrite applicationWillEnterForeground() as follows:

Growing/PragmaticTweets-8-1/PragmaticTweets/AppDelegate.swift
```
func applicationWillEnterForeground(application: UIApplication) {
  let request = TwitterAPIRequest ()
  request.sendTwitterRequest(NSURL (
    string: "https://api.twitter.com/1.1/users/suggestions.json"),
    params: [ : ],
    delegate: self)
}
```

This is only a couple lines, mostly the stuff we care about: the URL to call and the parameters (this request doesn't need any, so we send an empty dictionary). Notice that we didn't have to import the Accounts or Social framework: all that drudgery is stashed away in TwitterAPIRequest, closures and all. Separation of concerns, FTW!

Now we need to provide a handleTwitterData() method, to implement the TwitterAPIRequestDelegate protocol. The AppDelegate doesn't have a UI of its own, so let's just log the JSON to the debug pane:

Growing/PragmaticTweets-8-1/PragmaticTweets/AppDelegate.swift
```
func handleTwitterData(data: NSData!,
  urlResponse: NSHTTPURLResponse!,
  error: NSError!,
  fromRequest: TwitterAPIRequest!) {
  let jsonObject : AnyObject? =
```

```
  NSJSONSerialization.JSONObjectWithData(data,
    options: NSJSONReadingOptions(0),
    error: nil)
  println("suggestions JSON: \(jsonObject)")
}
```

Now, run the app. Our table appears as usual. Our new code only runs when the app returns to active duty from the foreground, so use the simulator's Hardware→Home (⇧⌘H) to background the app, then bring it to the foreground by tapping its icon, or select it from the running apps list by double-tapping home (i.e., ⇧⌘H twice, quickly). The request will go out and populate our debug pane:

```
suggestions JSON: Optional(<__NSCFArray 0x7fd5ab92e2e0>(
{
    name = Music;
    size = 101;
    slug = music;
},
{
    name = Sports;
    size = 74;
    slug = sports;
},
```

So, with about 10 lines of new code, we can fire off and handle a new Twitter API request. And that's how we're going to build out the functionality of our application, creating new view controllers and letting them reuse this general-purpose Twitter class we've created for ourselves. Of course, this was a somewhat silly exercise — feel free to delete all these changes in AppDelegate.swift — but it does prove out our general-purpose TwitterAPIRequest pretty nicely.

Wrap Up

In this chapter, we learned techniques that are helpful as projects get bigger. We started by organizing our files into groups, which we can expand, put away, and nest within one another, so we can look at just the files we need at any one time.

Then we looked at Xcode's support for refactoring, which is currently unavailable for Swift but is likely to support it in the near future. Instead, we did our own refactoring to change class names. Then we took on a bigger project: taking the Twitter code in RootViewController and making it a general-purpose class that can be reused by other classes we'll be creating later. We looked at how iOS uses the delegate pattern to hand off responsibility for

special-purpose code: the TwitterAPIRequest knows to send requests, but what to do with the response is entirely up to delegates.

CHAPTER 9

Navigating Between View Controllers

Thanks to our refactoring work in the previous chapter, we're now ready to add lots of new view controllers and their corresponding views, each with its own ability to make, receive, and parse Twitter API requests. Thing is, where do they go? We can't just plop another view controller scene onto the storyboard, because the storyboard would have no way to get to it. What we need is a way to navigate between view controllers.

In this chapter, we're going to add view controllers that can show things like tweet and user details, and use three different ways of navigating between multiple view controllers. These are the common idioms we use for moving around an app on the iPhone and iPad, and they let us choose whether to use the entire screen, split things on different parts of the screen, or even have it both ways, depending on how much screen space we have to work with.

Navigation Controllers

The most common way to work with multiple view controllers is to use a *navigation controller*, which is a view controller that manages a stack of child view controllers. This `UINavigationController` will become the new point of entry to the storyboard and will have our current `RootViewController` as the first thing it shows. From there, we'll add more view controllers, and the navigation controller will keep track of which one we're looking at and how to go back to earlier ones in the stack.

We won't have to write a new class for this, as the `UINavigationController` is meant to be used as-is and is seldom subclassed.

Actually, we don't need to write code at all to use a navigation controller; we can do everything in the storyboard. In fact, the best thing about the story-

board is how it visualizes complex navigation schemes. Ours will be pretty simple, but it's nice to know we can grow.

Switch to the storyboard and then locate the Navigation Controller in the Object Library; it looks like a yellow circle with a blue "back" arrow in it, like this:

Figure 59—Navigation Controller icon in Object Library

Drag this into the storyboard. During the drag, it will appear as two views connected by an arrow. Drop it close to the existing view controller, but above it. Once you've dropped, the storyboard will have three scenes, and three view controllers with their attached views: our original "root view controller", a navigation controller, and a table view controller, as shown in the following figure. Note that we've zoomed out to get all three scenes on screen at once, and set the simulated width to "Compact" in the blue sizing bar at the bottom.

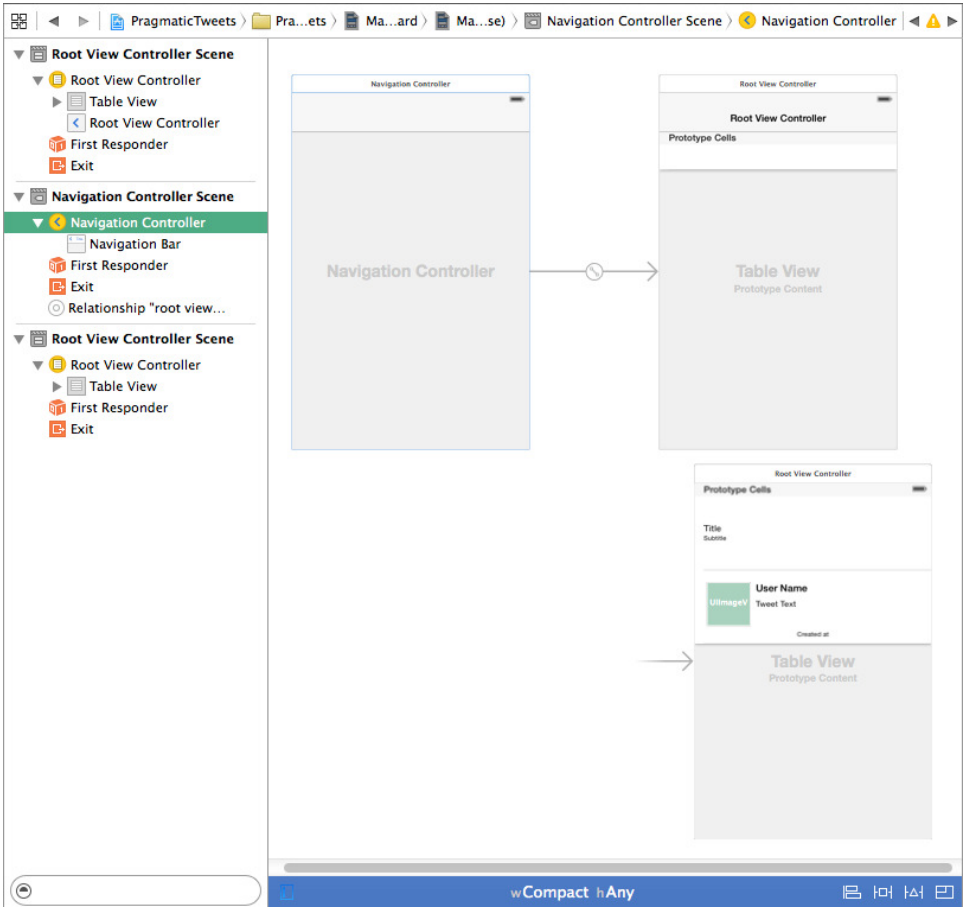


Figure 60—Adding a Navigation Controller to a Storyboard

Run the app now and… nothing's different! That's because there's no way in the storyboard to reach the navigation controller or its child view controller. We can change that by selecting the navigation controller — either in the scene list or from the yellow ball in the bar under its view — and bringing up its Attributes Inspector (⌥⌘4). Find the "Is Initial View Controller" checkbox and select it. In the storyboard graph, the arrow that went into our root view controller now goes into the navigation controller.

Run again and… now our tweets are gone, replaced by a table with the title "Root View Controller". What we're seeing is that the app now enters via the navigation controller, which in turn shows its first (root) child controller, which is the empty table view controller that Xcode gave us when we dragged

in the navigation controller. But we don't want this controller: all our custom table cell work is back in our old view controller.

What we need to do is to tell the navigation controller to use our old view controller as its root view controller. Notice that in the scene list, the last entry in the Navigation Controller scene is "Relationship 'Root View Controller'". That needs to change. Control click on the navigation controller, or bring up its Connections Inspector (⌥⌘6). Under "Triggered Segues", there's a connection called "Root View Controller". Starting from the connection's circle, begin a drag (which will stretch out a blue line) and drop on our old view controller, as shown below.

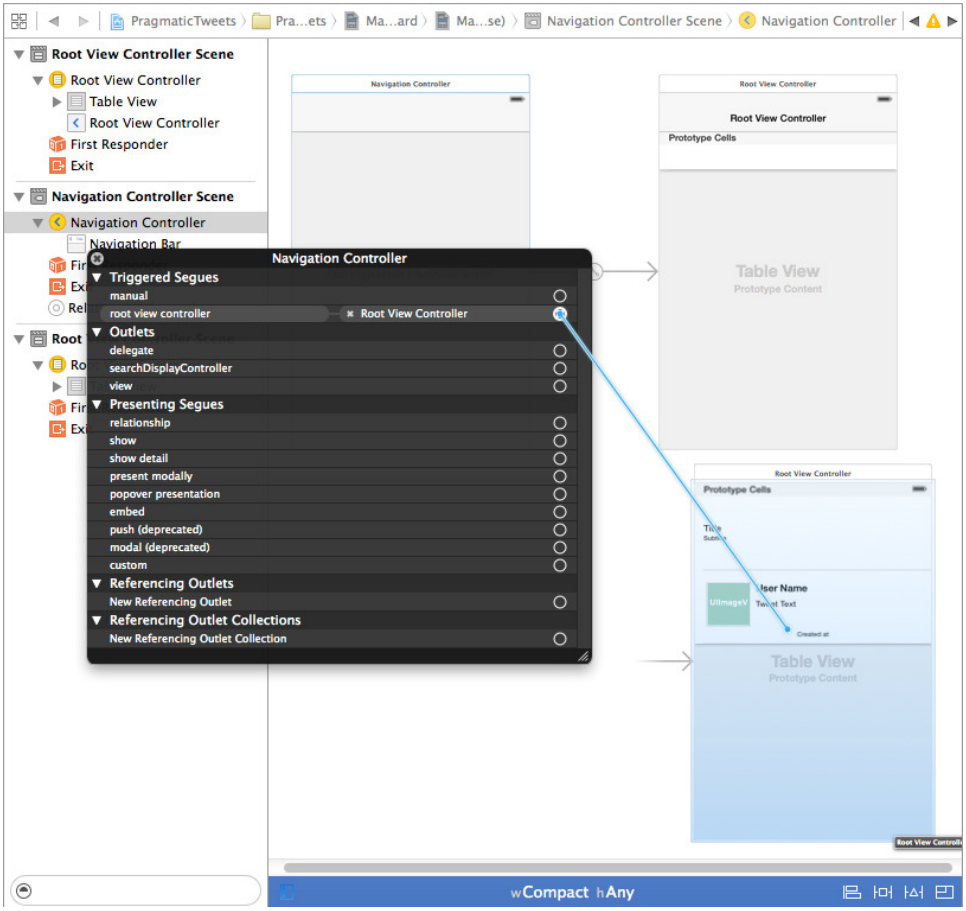


Figure 61—Reconnecting Navigation Controller's Root View Controller

Run again and our app is pretty much back to normal, showing all our tweets as before. The only change is that there's now a big blank space at the top of the screen. Still, progress!

The Navigation Bar

The space at the top of the screen is the *navigation bar*, which appears atop any view controller managed by a UINavigationController. The navigation bar has room for three UI elements, from left to right:

1. A left-side *bar button item*. For anything but the root view controller, this is typically a "Back" button, and a default back button will be provided if we don't set a different bar button item of our own.
2. A title, either as a string, or as a custom view.
3. A right-side bar button item.

We can easily customize all these things in the storyboard. First, as a bit of cleanup, we can get rid of that empty table controller that Xcode gave us, the one that was attached to the navigation controller. Select its view controller and press ⌫; the entire scene disappears. We can also grab the title bar atop our Root View Controller and move it around the storyboard to get it closer to the navigation controller; notice that as we do this, the arrow connecting the two (representing the navigation controller's "root view controller" relationship) stretches and bends as needed to keep the two connected.

Zoom back into a full size view of the Root View Controller and bring up its Attributes Inspector. Notice that the "Top Bar" says "Inferred". This is what it's always been set to; the storyboard figures out whether or not to show the navigation bar based on whether the view controller has a navigation controller as a parent, which it now does. Double-click in the center of the navigation bar and it'll turn into an editable text field. Type "Tweets" and press ↩ to finish editing. Notice that this changes the name of the scene to "Tweets Scene", and the view controller icon (the little yellow ball) to "Tweets". Run again and our tweets now have a nice title bar. Keep in mind that we haven't actually added a UILabel or UITextField. What we've actually done here is to tell the navigation bar what title to use for this view controller. To prove this point, notice in the scene list that what we're actually editing here is the properties of a "navigation item" within the Root View Controller scene, not a label or any other sort of view.

The two bar button items in the navigation bar also give us an opportunity to add functionality to our app. In fact, they give us a very nice way to bring

back our "new tweet" feature! In the Object Library, scroll down to the smaller "Item" icon; this is the "bar button item", shown in the figure below. The UIBarButton is very different from the UIButton we've used before; in fact, it's not even a subclass of UIView! It's an object that contains just enough state to be drawn in a bar and to be able to call a method when tapped.

Figure 62—Bar Button Item icon in Object Library

Drag the bar button item to the right side of the top bar in the Root View Controller scene. A well to accept the drop will appear, and after the drop, the bar button item will appear as "Item" in the bar. We can edit its text in place, but there's a better option. Select the bar button item and bring up the Attributes Inspector. The second attribute listed is "Identifier", with a default value of "Custom". Custom bar button items are those that have custom labels. However, there are about 20 other choices, representing common actions like "Search", "Refresh", and "Trash". From this list, choose "Add". This turns the bar button item into a plus (+) symbol, which is a reasonably intuitive way to tell the user that this is how they'll compose a tweet, and more practical in limited space than text like "Compose Tweet" would be. The figure below shows our finished navigation bar.

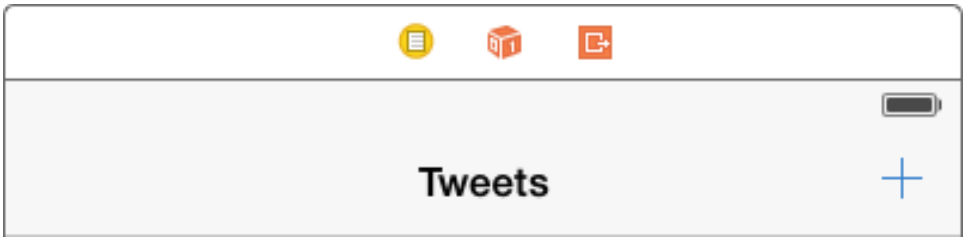


Figure 63—Customized Navigation Bar

Now that we've customized the button's appearance, we need to give it some functionality. Fortunately, our functionality already exists: it's the handleTweetButtonTapped() method we wrote way back in *Connecting User Interface To Code*, on page 27. So we just need to wire up a connection. Control-click on the add button to bring up the connections heads-up window, and notice that instead

of the UIButton's various events ("Touch Up Inside", etc.), there is just a "Triggered Segues action" and a "Sent Actions selector". What we want now is to just call a selector, so drag from "selector" over to the Root View Controller icon (the yellow ball that now says "Tweets"). When we drop, a popup will show us the selectors we can connect to.

Unfortunately, the one we want, handleTweetButtonTapped(), isn't in the list. The reason for this is in the code. handleTweetButtonTapped() currently takes a UIButton as an argument, and a bar button item isn't actually a button. So, in RootViewController.swift, edit that method definition to take AnyObject instead.

Navigation/PragmaticTweets-9-1/PragmaticTweets/RootViewController.swift

```
@IBAction func handleTweetButtonTapped(sender : AnyObject) {
```

Now, back in the storyboard, control-click the "+" bar button item, control drag from "selector" to the view controller icon, and after dropping, choose handleTweetButtonTapped(). Note that there's a faster way to do this: just control drag from the bar button item to the view controller, rather than bring up the popup, which figures out that you want to connect the selector, since the other kinds of connections don't make sense here.

At any rate, run the project again and tap the add button. The tweet compose view controller returns, although at this point, the default "I just finished the first project" text seems totally out of date. We've gotten a *lot* further since then! With the addition of the navigation bar, our app much more closely resembles the other major Twitter apps on iOS.

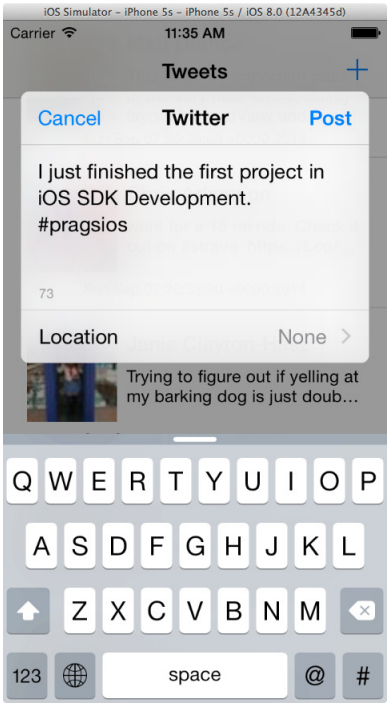


Figure 64—Tweet Compose View Controller Launched from Navigation Bar Button

Navigating Between View Controllers

Now that we have our root view controller managed by a navigation controller, we're reading to start navigating. Where shall we go? Since our root view shows a table of tweets, let's allow the user to select one of those tweets to inspect in detail. To do this, we'll add a new view controller scene to the storyboard, indicate how we navigate to it, and write a custom `UIViewController` subclass to provide the behavior for the new scene.

We can begin in the storyboard. From the Object Library, choose the generic View Controller icon, which looks like the rectangular `UIView` icon inside a yellow circle, as seen below:

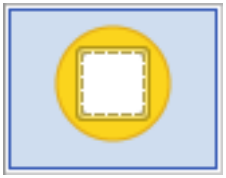


Figure 65—View Controller icon in Object Library

Drag the icon (which turns into a view with usual title bar underneath) into the storyboard, dropping it to the right of the existing Root View Controller. Once dropped, it appears as a completely empty view, and when not selected, the title bar above the view simply says "View Controller". This view controller has no visible contents, and cannot be reached from any other view controller, but we can change that easily enough.

From the Root View Controller scene on the left, select the "Parsed Tweet Cell", which is the one we customized with the styled labels and the icon. Do a control-drag from there to the new View Controller (either its entry in the scene list, or its icon, or its view out on the storyboard; any of these will work). This gesture indicates that you want to create a *segue* from the cell to the new view controller when the cell is tapped. Optionally, instead of control-dragging, we can bring up the connections popup with a control-click and drag the "Triggered Segues: selection" connection over to the new view controller.

Whichever gesture we use, upon ending the drag, a popup menu asks us to clarify what we want the connection to do, as seen below. For a selection segue, we have five main choices: show, show detail, present modally, popover presentation, or custom. Choose "show".

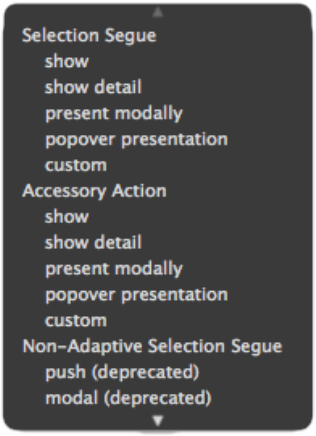


Figure 66—Choosing Selection Segue Type

Once we do this, two interesting things happen to our new view controller. First, it gets a simulated navigation bar, just like when we connected the root view controller to the navigation controller. That's because Interface Builder knows this view controller is managed by a navigation controller, so a navigation bar will be provided at runtime. However, we can't double-click in this one to set its title. The reason is that to customize the appearance of a non-root view controller, the scene needs to have a navigation item, which tells the navigation controller what's different about this scene, usually meaning a title and a right bar button item.

Drag a Navigation Item icon (shown below) from the Object Library to the new scene. You should now be able to double-click in the navigation bar and change its name to "Tweet".

Figure 67—Navigation Item icon in Object Library

The other thing that changed when we dragged the segue between the two scenes is that there's now an arrow connecting the Root View Controller to our new view controller. In the middle of the arrow is a circular icon that

represents the segue, which is the object managing the transition between the two view controllers. We'll have more to say about using segues a little later.

For now, let's run the app and see what we have. Once the tweets table gets populated, tap one of them. The tweets view will slide out to the left while the new view slides in from the right. While empty, we can easily get our bearings thanks to the navigation bar, which shows our "Tweet" title. The navigation controller also provides a back button on the left, which by default uses the title of the previous view controller: "Tweets". Not bad, getting navigation for free without having written any code for it!

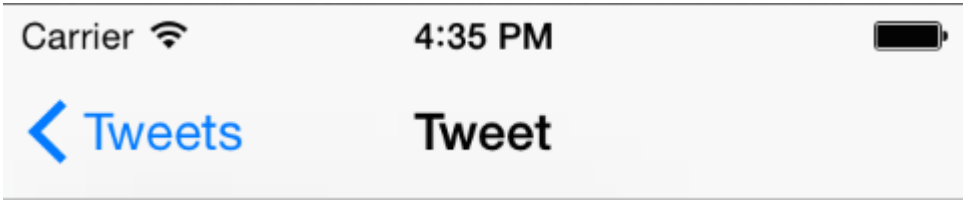


Figure 68—Default Navigation Bar for a Non-Root View Controller

Using the Storyboard Segue

When we tap a cell in the list of tweets, we navigate to the new view controller, which we'll customize to show the details fo the selected tweet. But hold up, how do we know which tweet was selected? And how will we communicate that to the other view controller?

This is where the segue can help us. Prior to performing a transition between view controllers, the current view controller gets a callback on the method `prepareForSegue()`, passing in details of the transition in a `UIStoryboardSegue` object. As inherited from `UIViewController`, this method does nothing, but we can override to to take some interesting action, based both on our current state and details of the segue.

The `UIStoryboardSegue` object provides properties for the `sourceViewController`, `destinationViewController`, and an `identifier`, which is a string that we can use to distinguish between different segues in the storyboard. It's a good habit to name any segue we intend to use in code, so click on the segue between the two view controllers and show the Attributes Inspector. The only attributes we can edit are the Identifier and the Style (which can be "Show", "Show Detail", "Present Modally", etc.). For the identifier, enter `showTweetDetailsSegue`.

Now visit RootViewController.swift. Write a new method to override prepareForSegue(), as follows

Navigation/PragmaticTweets-9-2/PragmaticTweets/RootViewController.swift
```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
  if segue.identifier == "showTweetDetailsSegue" {
    let row = self.tableView!.indexPathForSelectedRow()!.row
    let parsedTweet = parsedTweets [row] as ParsedTweet
    println ("tapped on: \(parsedTweet.tweetText!)")
  }
}
```

When called, this looks at the segue argument to see if it matches the identifier we put in the storyboard: showTweetDetailsSegue. If it does, it gets the selected row from the table, looks up the corresponding tweet, and logs its text to the console. Run the app and tap a row to verify this is working; if not, check the spelling of the segue identifier in the storyboard and the code to make sure they match *exactly*.

Sharing Data Between View Controllers

Now that we can get the selected tweet, we need a way to communicate between the view controllers. Actually, our tweet detail view controller may want more information than we have in the ParsedTweet, or things that the home_timeline API doesn't even provide, so we'll need a way to pass the tweet's unique identifier to the second view controller, and then let that view controller get whatever details it needs via a new Twitter API call.

So, add a tweetIdString to ParsedTweet.swift:

Navigation/PragmaticTweets-9-2/PragmaticTweets/ParsedTweet.swift
```
var tweetIdString : String?
```

In the Twitter API response, the tweet's unique ID string is identified with the key id_str, so that's what we need to get from the response dictionary, and set on the ParsedTweet. Put this assignment in RootViewController's handleTwitterData(), where we to the rest of our JSON unpacking:

Navigation/PragmaticTweets-9-2/PragmaticTweets/RootViewController.swift
```
parsedTweet.tweetIdString = tweetDict["id_str"] as? NSString
```

Now we're ready to send the tweet id to the second view controller, and let it get more detailed tweet information.

Sending Data to the Second View Controller

Now we need to put some code behind that second view controller, which we can do with a custom class. It's good practice to put each view controller class

and any helper classes in their own group, so in the File Navigator, create a new group called "Tweet Detail VC". Then select this group and do File→New →File… to create a new "Cocoa class" file. Call it TweetDetailViewController, and make sure it's a subclass of UIViewController and that the language is Swift.

Property Setters

This class will have one public property, a tweetIDString. When we set this property, we want the view controller to immediately take that id and fetch the details of the tweet from Twitter. To do that, we can use one of Swift's nifty features: *property setters*. The idea here is the property declaration can include some code that is called immediately anytime the value of the property is set. We do this by appending curly braces to the property declaration, and inside them, a didSet with the curly-braced code we want to run. Let's stub it out like this:

Navigation/PragmaticTweets-9-2/PragmaticTweets/TweetDetailViewController.swift

```
var tweetIdString : String? {
didSet {
  reloadTweetDetails()
}
}
```

This will cause the reloadTweetDetails() method to run anytime the tweetIdString is set. Of course, that method doesn't exist yet, so quickly stub out an empty implementation so we don't get build errors.

Navigation/PragmaticTweets-9-2/PragmaticTweets/TweetDetailViewController.swift

```
func reloadTweetDetails() {
}
```

Joe asks:

Is there also a didGet?

Yep! And if you remember all the way back in *Declaring Properties*, on page 45, we talked about a difference between "stored properties", which are backed by a variable, and "computed properties". didGet is how we'd implement a computed property: its value is implicit from other properties, which we'd compute in the didGet block.

Sending data via a segue

The storyboard doesn't know that the second view controller is supposed to use this class. Fix that by selecting the second view controller in the storyboard and bringing up the Identity Inspector (⌘⌥3). Under "Custom Class", change

the class from UIViewController to TweetDetailViewController (it should auto-complete as you type).

Now the pieces are all in place to deliver the tweetIdString from the first view controller to the second when the transition happens. Go back to RootViewController.swift, and rewrite the prepareForSegue:sender:() method as follows:

Navigation/PragmaticTweets-9-2/PragmaticTweets/RootViewController.swift

```
Line 1 override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
   2   if segue.identifier == "showTweetDetailsSegue" {
   3     if let tweetDetailVC = segue.destinationViewController
   4       as? TweetDetailViewController {
   5       let row = self.tableView!.indexPathForSelectedRow()!.row
   6       let parsedTweet = parsedTweets [row] as ParsedTweet
   7       tweetDetailVC.tweetIdString = parsedTweet.tweetIdString;
   8     }
   9   }
  10 }
```

The big change here is lines 3-4, where we ask the segue for its destinationViewController, and attempt to cast it to a TweetDetailViewController. If this works, when we can assign the tweetDetailVC's tweetIdString on line 7, which will kick off the setter method we wrote before.

Designing the Second View Controller

Now let's give the second view controller some UI elements to fill in. Add the following UI elements and their constraints:

- A button at upper left, with fixed width and height of 60 by 60, pinned 20 points in from the superview on both its top and leading edges. Use the Attribute Inspector to change its type to "custom", which will allow us to change its image.
- A label called "Real Name" top-aligned to the button, leading space 8 points from the button, stretched all the way across so its trailing edge is 20 points from the superview. Change the font to System Bold at 17 point size.
- A label called "User Name" just below it, leading space 8 points from the button, with a top edge 8 points down from the label above it (a fixed distance to the superview will also work), and trailing space of 20 to the superview. Change the font to make it smaller than the default, like System font at 15 point size.

- A label called “Tweet Text”, leading edge and trailing edges 20 points in from the superview, 8 points down from the button, and set to 0 lines so it can grow as needed to accommodate the tweet text.
- An `MKMapView` (its icon is shown below), top edge 8 points down from the “Tweet Text” label, leading and trailing, and bottom edges 20 points from the superview.

Figure 69—MKMapView Icon

When finished, the layout should look something like this:

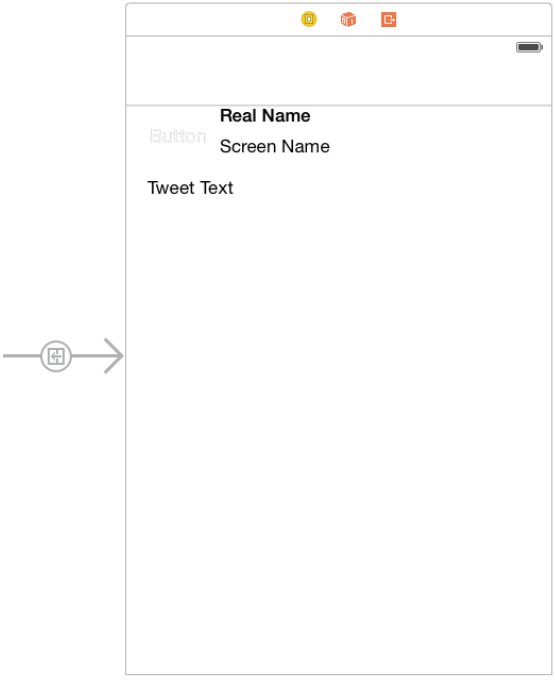


Figure 70—Layout of the TweetDetailViewController View

Switch to the Assistant Editor, and make sure that TweetDetailViewController.swift is visible in the right pane (use the jump-bar to bring up the right file if nec-

essary). Control-drag from each of these UI components in the view to create outlets in the class file, using the names userImageButton, userRealNameLabel, userScreenNameLabel, tweetTextLabel, and tweetLocationMapView. The resulting outlets should look like this:

Navigation/PragmaticTweets-9-2/PragmaticTweets/TweetDetailViewController.swift
```
@IBOutlet var userImageButton : UIButton!
      @IBOutlet var userRealNameLabel : UILabel!
      @IBOutlet var userScreenNameLabel : UILabel!
      @IBOutlet var tweetTextLabel : UILabel!
      @IBOutlet var tweetLocationMapView : MKMapView!
```

Add an import MapKit to the top of the file to make sure the project will still build, since the MKMapView is part of the Map Kit framework, which isn't imported by default.

Coding the Second View Controller

Now we need to add the code to make our second view controller get to work. We can wait until our view controller's viewWillAppear() method is called, so that if the tweetIdString has been set, it can immediately update the UI.

Navigation/PragmaticTweets-9-2/PragmaticTweets/TweetDetailViewController.swift
```
override func viewWillAppear(animated: Bool) {
  super.viewWillAppear(animated)
  reloadTweetDetails()
}
```

Notice that since we're overriding the viewWillAppear() inherited from UIViewController, we have to put an explicit override in the method declaration, and call the superclass' implementation as part of our own.

Now we're ready to use the Twitter classes we refactored in the previous chapter. We're going to use a TwitterAPIRequest to ask for the details, and parse the response. To be the delegate that handles the response, we have to declare that we implement the protocol, so update the class' declaration like this:

Navigation/PragmaticTweets-9-2/PragmaticTweets/TweetDetailViewController.swift
```
class TweetDetailViewController: UIViewController, TwitterAPIRequestDelegate {
```

The Twitter API provides the statuses/show.json call to get details about a single tweet, and takes a single parameter, id, with the unique ID of the tweet, so that's what we'll call in our reloadTweetDetails().

Navigation/PragmaticTweets-9-2/PragmaticTweets/TweetDetailViewController.swift
```
func reloadTweetDetails() {
  if tweetIdString == nil {
    return
  }
```

```
  let twitterRequest = TwitterAPIRequest()
  let twitterParams = ["id" : tweetIdString!]
  let twitterAPIURL = NSURL (string:
    "https://api.twitter.com/1.1/statuses/show.json")
  twitterRequest.sendTwitterRequest(twitterAPIURL,
    params: twitterParams,
    delegate: self)
}
```

This is similar to the code we refactored in RootViewController's reloadTweets() in the last chapter: we create a TwitterAPIRequest, set its URL and parameters, and fire off the request, declaring the current class to be the delegate, meaning its handleTwitterData() method will be called with the response.

A trivial implementation of the callback method could just convert the data parameter into a NSJSONSerialization object, and log it out to see what Twitter sends back to us. To save you that step: the response provides the tweet text, along with a user dictionary that contains a name, screen_name, and much, *much*, more. Let's pull out the easy stuff first.

Navigation/PragmaticTweets-9-2/PragmaticTweets/TweetDetailViewController.swift

```
func handleTwitterData (data: NSData!,
  urlResponse: NSHTTPURLResponse!,
  error: NSError!,
  fromRequest: TwitterAPIRequest!) {

    if let dataValue = data {
      let jsonString = NSString (data: data, encoding:NSUTF8StringEncoding)
      var parseError : NSError? = nil
      let jsonObject : AnyObject? =
      NSJSONSerialization.JSONObjectWithData(dataValue,
        options: NSJSONReadingOptions(0),
        error: &parseError)
      if let errorValue = parseError {
        return
      }
      if let tweetDict = jsonObject as? Dictionary<String, AnyObject> {
        dispatch_async(dispatch_get_main_queue(),
          {() -> Void in
            let userDict = tweetDict["user"] as NSDictionary
            self.userRealNameLabel.text = userDict["name"] as? NSString
            self.userScreenNameLabel.text = userDict["screen_name"] as? NSString
            self.tweetTextLabel.text = tweetDict["text"]  as? NSString
            let userImageURL = NSURL (string:
              userDict ["profile_image_url"] as NSString!)
            self.userImageButton.setTitle(nil, forState: UIControlState.Normal)
            self.userImageButton.setImage(
              UIImage(data: NSData(contentsOfURL: userImageURL)),
              forState: UIControlState.Normal)
```

```
            })
        }
      } else {
        println ("handleTwitterData received no data")
      }
}
```

The handling of the jsonResponse and using a closure to update the UI on the main queue is identical to what we did in the RootViewController; the only difference is which values we pull out of the tweetDict and then use to update our user interface. It's also different in that we are using a button for the user's icon, something we'll take advantage of in a little bit.

Once the response is received, the various labels and the image button are all updated, including the tweet text, which thanks to autolayout can grow or shrink to as many lines are needed to contain all the text. Try it now and verify that everything's OK.

Adding a Map

Since there's so much room on the tweet detail view, we added a map view to the storyboard, in case the tweet is geo-tagged. If this is the case, the response from twitter will contain a dictionary called geo, whose contents include information like the latitude and longitude of the tweet (inside an array called coordinates).

A full coverage of Map Kit is beyond the scope of the book, so for now, we will restrict ourselves to the MKMapView and the fact that we can add MKPointAnnotations to it, which show up as pins on the map. The annotation represents its latitude and longitude with the CLLocationCoordinate2D type (from the *Core Location* framework, which Map Kit imports for us), and the map also uses this type to represent the center of the map. So to quickly represent a tweet's location, we can pull out its latitude and longitude, construct a CLLocationCoordinate2D, center the map on this coordinate, and add an annotation with that coordinate to the map.

Add the following code to the handleTwitterData() method, right after the line that sets the userImageButton.

Navigation/PragmaticTweets-9-2/PragmaticTweets/TweetDetailViewController.swift

```
Line 1 if let geoDict = tweetDict ["geo"] as? NSDictionary {
   -   let coordinates = geoDict["coordinates"] as NSArray
   -   if coordinates.count == 2 {
   -     let latitude = (coordinates[0] as NSNumber).doubleValue
     5   let longitude = (coordinates[1] as NSNumber).doubleValue
   -     let tweetCoordinate =
   -       CLLocationCoordinate2D(latitude: latitude, longitude: longitude)
```

```
-     self.tweetLocationMapView.centerCoordinate = tweetCoordinate
-     let pointAnnotation = MKPointAnnotation()
10    pointAnnotation.coordinate = tweetCoordinate
-     self.tweetLocationMapView.removeAnnotations(
-       self.tweetLocationMapView.annotations)
-     self.tweetLocationMapView.addAnnotation(pointAnnotation)
-     self.tweetLocationMapView.setRegion(MKCoordinateRegion(
15      center: tweetCoordinate,
-       span: MKCoordinateSpanMake(1.0, 1.0)),
-       animated: true)
-     self.tweetLocationMapView.hidden = false
-   }
20 } else {
-   self.tweetLocationMapView.hidden = true
- }
```

Lines 1-22 check to see if there's geo-tag data at all; if not, the map view is hidden. On lines 4-7, we pull out the first two elements of the array and make them the latitude and longitude of a `CLLocationCoordinate2D`, which we then use to center the map on line 8. We create a new `MKPointAnnotation` and set its location on 10 which we add to the map on lines 11-13, while also clearing out any old annotations from the map. Finally, by declaring a "region", we can get the map view to zoom in on just the part of the world we're interested in; lines 14-17 use a coordinate span of 1 degree for both latitude and longitude (which, very roughly, will give us about a 100 km range).

There's a lot of details about MapKit that we can't get into here, so feel free to poke around the docs if you want a deeper understanding about how this part works. For now, run this version of the app and tap on a tweet to see it in the detail view. A tweet that has location tags will show the map, as seen below. We can drag around the map and pinch-zoom to zoom in on the pin; on the Simulator, use option-click with the mouse to perform pinch-zooming.

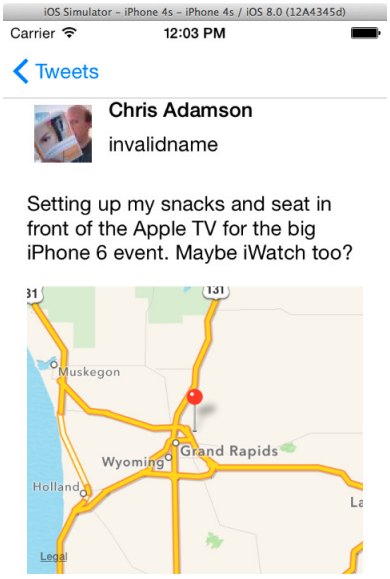


Figure 71—Custom view controller showing navigation controls and MKMapView

So, now we have a more interesting Twitter app, one that lets us pick a tweet and show it in detail, including whatever information we'd care to pull out of the response, such as presenting location information on a map. More importantly, we have a way forward for expanding the capabilities of the app: as we need new features or new ways to enter or present data, we can navigate to new view controllers, building them out in our storyboard and custom classes.

Modal Navigation

In fact, we'll close out the chapter with a different approach for presenting view controllers, and how it ties into one of the neatest things about navigating through storyboards.

In the tweet detail controller, we used a button rather than an image view to present the user's icon, and this is where we're going to use that: we'll allow the user to tap the icon to go to a third view controller, one which presents details about the user.

Modal Segues

Add another view controller to the storyboard, to the right of the tweet detail view controller (again, it may be necessary to zoom out to organize the views

nicely). Scroll so that both the tweet detail view controller and the new view controller are visible at the same time, and control-drag from the user image button to the new view controller; this can also be done by control-dragging through these entries in the scene tree on the left. When we end the drag and release, Interface Builder infers that we want to make a segue to the new view controller, triggered by a tap on the button, and shows a popup asking what kind of segue to create: This time, choose "present modally".

The storyboard will add an arrow connecting the second and third view controllers, with a circle-shaped segue icon in the middle. Notice that this icon is different from the icon for the push segue we used earlier. And there's another thing to notice: *the new view controller doesn't show a navigation bar*. That's because modal navigation is different. Showing another view controller modally doesn't require a navigation controller like a "show" presentation does, so we're always free to create this kind of transition in a storyboard. This is handy because we often need to do something modally, meaning we need to stop the user to either show them something or get some input from them before we continue.

On the other hand, no navigation bar means no back button, and we'll have to deal with that eventually.

While we're thinking about the segue, click its icon and bring up the Attributes Inspector, so we can give it an identifier string: `showUserDetailsSegue`.

Laying Out the User Detail View Controller

For now, let's build some utility into this third view controller. This will be where we show a user's details, so add the following elements, from top to bottom, so they look like the image below:

- An image view, top edge pinned to 0, width and height pinned to 100, horizontally aligned in container.
- A label called "User Name", center-aligned, System Bold 24.0 point, top edge pinned 8 points below the image view, left and right edges pinned 20 points from the superview.
- A label called "Screen Name", center-aligned, top edge pinned 8 points below the user name label, left and right edges pinned 20 points from the superview.
- A label called "Location", center-aligned, top edge pinned 8 points below the screen name label, left and right edges pinned 20 points from the superview.

- A label called “Description”, center-aligned, set to 0 lines so it can wrap as needed, top edge pinned 8 points below the location label, left and right edges pinned 20 points from the superview.
- A button titled “Done”, top edge pinned 8 points below the description label, horizontally aligned in the container.

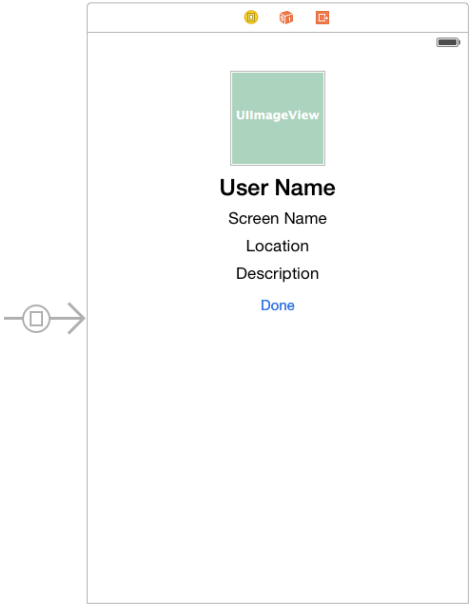


Figure 72—Layout of User Detail Scene

Now we want to tie this into code, so we’ll need a custom view controller, like we did before with the tweet details scene. In the File Navigator, create a group called “User Detail VC”. Within that group, create a new file, using the iOS “Cocoa Class” template, calling it UserDetailViewController and making it a subclass of UIViewController, written in Swift.

Back in the storyboard, select the view controller icon for this scene (either from the scene list on the left or the view’s lower title bar), and use the Identity Inspector (⌥⌘3) to set the class to UserDetailViewController. That will allow us to bring up the Assistant Editor (making sure that UserDetailViewController.swift is in the right pane) to control-drag outlets for the image view and all the labels. When done, the properties should look like this:

Navigation/PragmaticTweets-9-3/PragmaticTweets/UserDetailViewController.swift

```
@IBOutlet var userImageView : UIImageView!
@IBOutlet var userRealNameLabel : UILabel!
```

```
@IBOutlet var userScreenNameLabel : UILabel!
@IBOutlet var userLocationLabel : UILabel!
@IBOutlet var userDescriptionLabel : UILabel!
```

Coding the User Detail View Controller

Our code for this third view controller is going to be a lot like the second, we'll expose a public property, and let it use that to refresh itself from the Twitter API every time it needs data. Twitter lets us get the user details from just a screen name, so let's make that a public property in the UserDetailViewController.swift file:

Navigation/PragmaticTweets-9-3/PragmaticTweets/UserDetailViewController.swift
```
var screenName : String?
```

As before, we'll fire off a TwitterAPIRequest and expect the class to parse the result, which means it will be the delegate for the Twitter request. So we need to update the class declaration to indicate we implement that delegate protocol:

Navigation/PragmaticTweets-9-3/PragmaticTweets/UserDetailViewController.swift
```
class UserDetailViewController: UIViewController, TwitterAPIRequestDelegate {
```

Since we want to update the view whenever it appears, we'll make our Twitter call in viewWillAppear(). To get the user details, we'll use Twitter's users/show.json request, which takes just a screen_name parameter. So here's the method to write:

Navigation/PragmaticTweets-9-3/PragmaticTweets/UserDetailViewController.swift
```
override func viewWillAppear(animated: Bool)  {
  if screenName == nil {
    return;
  }
  let twitterRequest = TwitterAPIRequest()
  let twitterParams = ["screen_name" : screenName!]
  let twitterAPIURL = NSURL (string: "https://api.twitter.com/1.1/users/show.json")
  twitterRequest.sendTwitterRequest(twitterAPIURL,
    params: twitterParams,
    delegate: self)
}
```

When we get a response, we'll get the delegate callback as handleTwitterData(), which we can write as follows:

Navigation/PragmaticTweets-9-3/PragmaticTweets/UserDetailViewController.swift
```
func handleTwitterData (data: NSData!,
  urlResponse: NSHTTPURLResponse!,
  error: NSError!,
  fromRequest: TwitterAPIRequest!) {

    if let dataValue = data {
```

```
      let jsonString = NSString (data: data, encoding:NSUTF8StringEncoding)
      var parseError : NSError? = nil
      let jsonObject : AnyObject? =
      NSJSONSerialization.JSONObjectWithData(dataValue,
        options: NSJSONReadingOptions(0),
        error: &parseError)
      if let errorValue = parseError {
        return
      }
      if let tweetDict = jsonObject as? Dictionary<String, AnyObject> {
        dispatch_async(dispatch_get_main_queue(),
          {() -> Void in
            self.userRealNameLabel.text = tweetDict["name"] as? NSString
            self.userScreenNameLabel.text = tweetDict["screen_name"] as? NSString
            self.userLocationLabel.text = tweetDict["location"]  as? NSString
            self.userDescriptionLabel.text = tweetDict["description"] as? NSString
            let userImageURL = NSURL (string:
              tweetDict ["profile_image_url"] as NSString!)
            self.userImageView.image = UIImage(data:
              NSData(contentsOfURL: userImageURL))
        })
      }
    }
}
```

This is our third time unpacking a Twitter response, so it should be looking pretty familiar: use NSJSONSerialization to convert the data to an NSDictionary, then use known keys in the response to pull out interesting values and set them in the UI, using dispatch_async() to do all this on the main queue, as required by UIKit.

The last thing we need to do is to actually set the screenName property. That's something the second view controller will do as it begins the segue. Switch to TweetDetailViewController.swift and add an implementation of prepareForSegue:sender:().

Navigation/PragmaticTweets-9-3/PragmaticTweets/TweetDetailViewController.swift

```
override func prepareForSegue(segue: UIStoryboardSegue?, sender: AnyObject?) {
  if (segue!.identifier == "showUserDetailsSegue") {
    if let userDetailVC = segue!.destinationViewController
      as? UserDetailViewController {
      userDetailVC.screenName = self.userScreenNameLabel.text
    }
  }
}
```

Run this version of the app, choose a tweet to view in detail, and then click on the user icon button. This will perform a modal transition to the user detail view controller, showing the user in all his or her glory, like this:

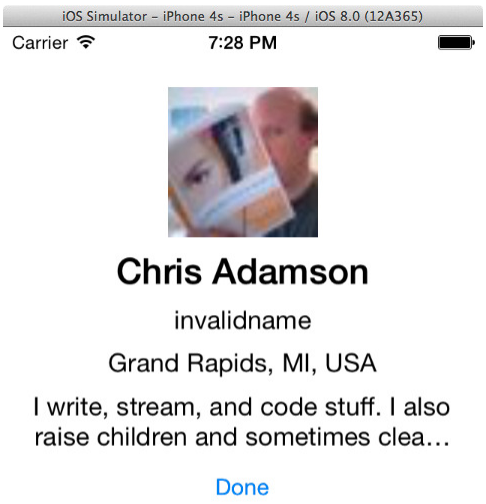


Figure 73—User Detail View

This looks great, but there's just one little problem: *we're trapped*. There's no back button, and the "Done" button does nothing. Now what do we do?

Exit Segues

There are a few ways we could implement the back button, but the most generally useful is the *exit segue*. With an exit segue, we can go backwards in a navigation, regardless of whether we came by way of push or modal segues.

What's tricky about exit segues is that they don't appear on the storyboard the same way push or modal segues do. Instead, their existence is implicit. We can only perform an exit segue if a previous view controller has exposed a method for us to come back to. These methods, commonly called *unwind methods* have to follow a certain signature — they take a `UIStoryboardSegue` parameter and have return type `IBAction` — but they don't have to actually have any code, they just have to exist.

We want our user detail view controller to unwind to the tweet detail view controller, so write the following method in TweetDetailViewController.swift:

Navigation/PragmaticTweets-9-4/PragmaticTweets/TweetDetailViewController.swift

```
@IBAction func unwindToTweetDetailVC (segue: UIStoryboardSegue?) {
}
```

An unwind method needs to have the @IBAction annotation (so we can make connections to it in the storyboard), take a UIStoryboardSegue as a parameter, and return nothing. This particular unwindToTweetDetailVC() implementation does nothing, but if we did need to collect data from the other view controller, this would be a great place to do so. In a sense, the unwind method is a counterpart to prepareForSegue:sender:() method, in that the prepare method can send data to a view controller that we're transitioning to, and the unwind can get data from it when it's done.

In the storyboard, click on the user detail view controller's "Done" button, and control-drag down to the orange box on the right of the title bar above the view, which shows the tooltip "Exit" as we hover over it, as shown below. When we complete the drop, a popover appears showing all the unwind methods we can connect to.

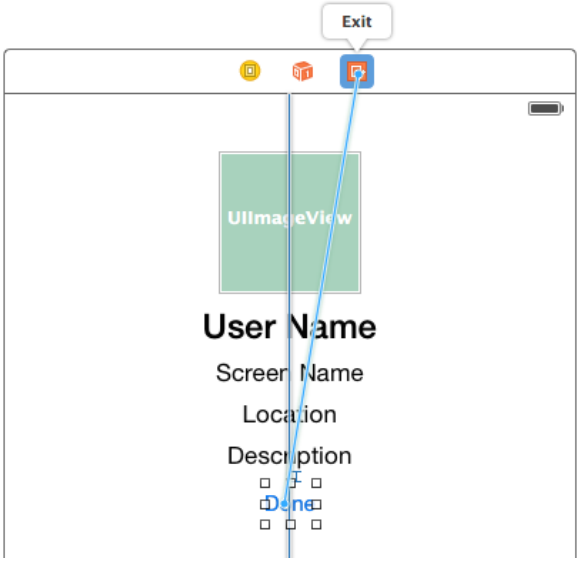


Figure 74—Connecting Button Tap to an Exit Segue

Now run the app, drill down to the user details, and tap "Done". The modal transition unwinds and we're back at the tweet details view controller. All

this with no code… well, no code that does anything, anyways. We could put an NSLog() in the unwindToTweetDetailViewController() to see that it's being called.

Perhaps more interestingly, we can unwind to *any* earlier view controller. For example, if we go to RootViewController, write an unwindToRootViewController() there, and connect to an exit segue that uses that method instead, our "Done button would take us all the way back to the root view controller, skipping over the tweet detail view controller entirely. This can be immensely helpful in complex storyboards where our navigation controllers get four or five view controllers deep, and we find the user may want a nice "start over" or "go home" button; exit segues make this really easy.

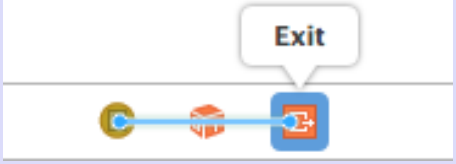
Joe asks:

Can I only perform segues on button or table-cell taps?

It's possible to perform segues programmatically, which can be useful if we have a long running action that should perform a segue when it's completed, like a login screen dismissing itself when a remote server sends us a response that the password has been accepted.

To programmatically go forward, a view controller can call the performSegueWithIdentifier() method. The identifier parameter it takes is the same string we've been using in prepareForSegue(), reminding us why we always want to put identifier strings on segues in the storyboard.

Programmatically performing exit segues is a little tricker, since they don't initially appear in the storyboard in a way that we can give them identifiers. But we can force the issue by control dragging from the view controller icon to the exit segue icon, as shown below. This adds an "Unwind segue from…" entry to the view controller's children scene list, which we can then select and edit its attributes to give it an identifier. And then we can call performSegueWithIdentifier() to perform the unwind programmatically.



Wrap Up

This has been a very long chapter, in which we've radically reworked our app into one that is far more capable and extensible than when we started. We've gone from being tied to one screen to having as many as we care to create.

To do this, we reworked the storyboard from a single-view design to a navigation metaphor, putting a navigation controller at the beginning of the storyboard and letting it manage the user's progress through our root view controller and a new tweet detail view controller, giving us forward/back navigation pretty much for free. We saw how to use storyboard segues to deliver information between view controllers, which allowed our root view controller to tell the tweet detail view controller just which tweet the user tapped on. Then we tried out a modal transition to show the user detail view controller, and how exit segues let us return to any previous view controller, and deliver data back to them in the unwind method.

This is how many popular apps work: navigating forward and backwards through view controllers, each specific to some part of the app's overall functionality. From here, we can add any new features we might think of.

In the next chapter, we're going to look at another way of managing multiple view controllers, one that's particularly well-suited to the iPad.

CHAPTER 10

Taking Advantage of Large Screens

We've got a pretty nifty Twitter app at this point, one that lets us scroll through tweets, navigate into a detailed view of a tweet, and then drill down to details about the account that sent it. It's pretty nice on an iPhone.

But, come to think of it, we haven't tried running it on an iPad. And we did make it a "Universal" app in the beginning. So let's see what that looks like. Use the scheme selector in Xcode's toolbar to change to a model of iPad — we often use iPad 2, because as a non-Retina device it fits on the Mac screen — and run the app. In landscape, it looks like this:

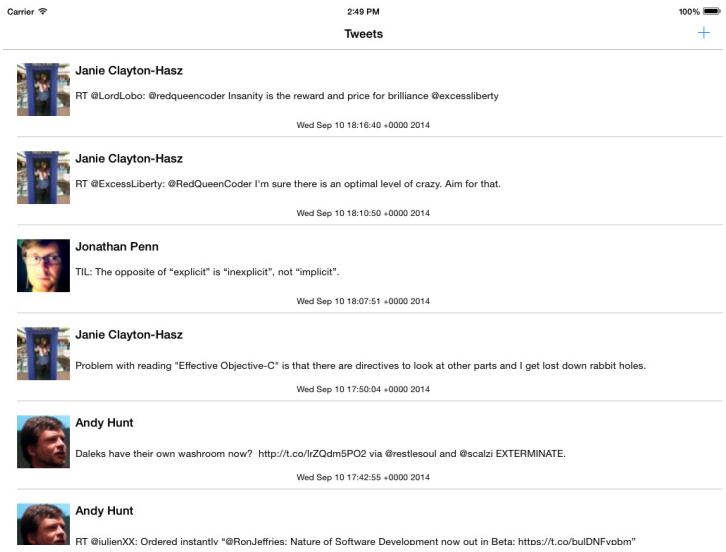


Figure 75—Table view on iPad in landscape orientation

It's… *OK*. Kind of. It's not like any of the views are in the wrong place or anything. And it works fine. It just doesn't take any advantage of all the extra room on the screen. In fact, it looks a lot like the Android screenshots that speakers at Apple events will show off to demonstrate how stretching a phone UI to a tablet screen doesn't work.

So, let's not do that. In this chapter, we're going to take advantage of some unique options that let us use the iPad screen to a better effect, while still working the way we want on iPhone. We'll adapt to larger screens — starting with the iPad and then the iPhone 6 Plus — so that our app can have the best of both worlds.

Split Views on iPad

When the iPad was first introduced, Apple changed the name of the operating system from "iPhone OS" to "iOS", and added some iPad-specific features to the SDK. The most distinctive is probably the *split view*. This is a UI metaphor that combines a narrow view on the left side of the screen with a wide view on the right. In portrait orientation, the left view usually can be shown or hidden, while on in landscape view the left view is always present.

Several built-in apps on iOS use the split view. Mail shows message senders and subjects in the left view, and the message content on the right. Settings has the master list of settings categories on the left, and the UI for the selected topic on the right. The split view lends itself well to this sort of a "master-detail" metaphor: The main list of items is in a table on the left, and selections in this table populate the contents of a detail view on the right.

Conveniently, this is also how our Twitter app works. We have a list of tweets, and when we tap one of them, we bring up details on it in a new view. For starters, let's adapt our app to use the split view like this.

Adding a Split View to the Storyboard

To adopt the split view, we need to go to the storyboard, and zoom out for a view of our current view controllers. Right now, they're in a left-to-right flow, starting with the navigation controller, and proceeding through the root view controller, the tweet detail view controller, and the user detail view controller. Go to the Object Library, find the "Split View Controller" icon (shown below), and drag it to the storyboard. The drag will put four scenes on to the storyboard, so do the drop someplace where there's lots of room to work with.

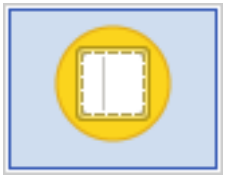


Figure 76—Split View Controller icon in Object Library

Post-drop, the default split view goes off in a couple different directions, as seen in the following figure. On the left, the split view controller scene has one connection that goes up and right to a navigation controller, and from there to a table view controller. The split view controller also has another connection that goes down and right, to a plain and empty view controller.

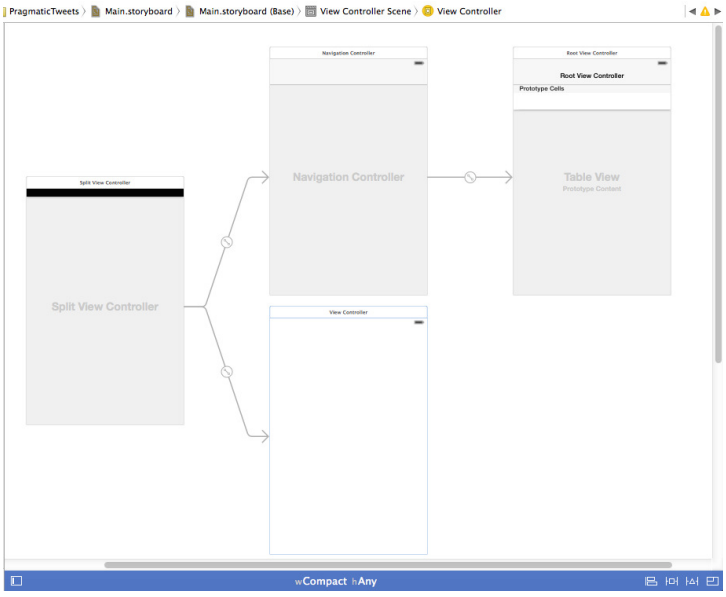


Figure 77—Default Split View Controller scenes in storyboard

With the default arrangement of scenes, the navigation controller exists largely to provide a navigation bar for the master table view, since the table will often want to have “add” or “edit” buttons. Meanwhile, the default detail view is empty, since its contents will totally depend on the content presented by the table, and which row is selected.

We already have suitable view controllers to play both of these roles, so rather than customizing the default scenes, we will delete them and replace them

with our own. Our RootViewController, the scene currently labeled “Tweets” will replace the default one, and our TweetDetailViewController will become the detail scene. Here’s how we’re going to do that:

- Start by deleting the split view’s default table view controller, the one at upper right that says “Root View Controller”. We do this by selecting the scene, or its view controller icon in the scene list, and pressing the delete key (⌫).
- Next, we won’t need our own navigation controller, so delete the navigation controller scene to the left of our “Tweets” scene (the one with the table view). Notice that this causes the navigation bar to disappear from all the later scenes in the navigation flow. It’s OK. We won’t need them, and we’ll get them back if we ever add these scenes to a navigation controller again.
- Since we’re not navigating anymore, select the segue — the circle in the arrow between the “Tweets” scene and the “Tweet Details” scene — and delete it as well.
- Now we can connect our table to the master portion of the split view. Drag the “Tweets” up to the right of the surviving navigation controller (the one that is connected to the split view controller). Control-click on the navigation controller and find the connection called “Root View Controller”. Drag from this connection over to the “Tweets” scene to make the connection.
- Now for the detail view. Delete the empty default detail scene. Drag the “Tweet Detail View Controller” into the space that was just vacated. Control click the split view controller to see its connections. The “Detail View Controller” connection is now empty; drag this to the Tweet Detail scene.
- Finally, select the split view controller, bring up its attribute inspector (⌥⌘3), and select the “Is Initial View Controller” checkbox. The navigation controller had been our initial VC, and without an entry point, our app wouldn’t know where to get started.

Wow! That’s a lot of clicky-draggy! Well, if nothing else, this should allay any fears about deleting and reconnecting storyboard scenes. And if things ever go truly bad, there’s always the “Undo” command. At any rate, the storyboard should now look like the following figure, with the upper branch of the split view going to a navigation controller and our “Tweets” table, and the lower branch going to the tweet detail scene, and then on to user detail.

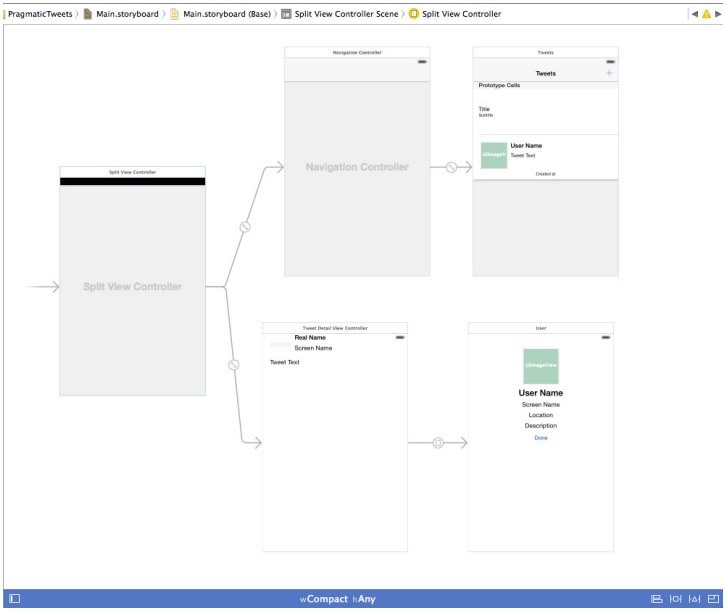


Figure 78—Split View Controller with RootViewController, TweetDetailViewController, and UserDetailViewController

Go ahead and run the app like this, with the scheme selector still set to some flavor of iPad. In portrait, all we'll see is the unpopulated detail view with its empty labels for the user name and tweet text. However, a left-to-right drag gesture will reveal the master view, the list of tweets, on the left. Rotating the simulator to landscape (⌘← or ⌘→) will cause the master list to always be visible, as seen in the following figure.

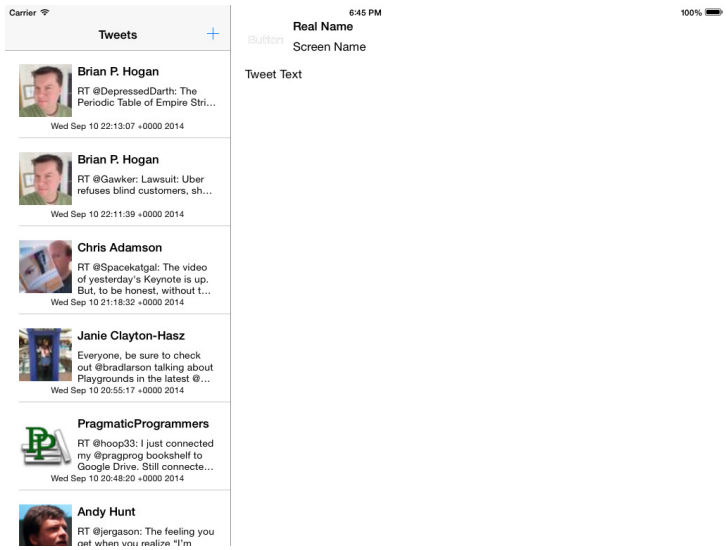


Figure 79—Split View showing tweets in master view controller

So far, so good! The master view appears when it needs to in landscape, can be brought up in portrait, and all the work we did to populate the list of tweets is still working as before.

There's just one thing: *tapping on the rows no longer does anything.* Previously, we had created a segue to connect the table to the tweet detail scene, which gave us the navigation between scenes. But we deleted that segue, and now there's no way to send data between the scenes. So what do we do now?

Connecting Scenes in a Split View Controller

When we built our navigation in the storyboard, creating the segue from the table to the detail scene took care of handling taps on the table for us, and telling us which destinationViewController was coming in, which is how we told the second view controller which tweet to show in detail. With that gone, we will have to handle things on our own.

The first thing to do is to go to RootViewController.swift, where we'll write an implementation of tableView(didSelectRowAtIndexPath:). The trick is going to be getting information to the TweetDetailViewController, which we don't have any kind of reference to: it's not a property, and we don't get told about it via a prepareForsegue() method anymore.

The only thing these two view controllers have in common anymore is that they're both connected to the same UISplitViewController. As it turns out, that's

exactly the key we need. The UIViewController class has an optional property, splitViewController, defined ad "the nearest ancestor in the view controller hierarchy that is a split view controller."

Now let's think about what we can do with that. The UISplitViewController has an array property, viewControllers, that represents the child view controllers it manages. So there should be two: the navigation controller that's in front of our RootViewController, and the TweetDetailViewController.

BigScreens/PragmaticTweets-10-1/PragmaticTweets/RootViewController.swift

```
Line 1 override func tableView(tableView: UITableView,
  -   didSelectRowAtIndexPath
  -   indexPath: NSIndexPath)  {
  -     let parsedTweet = parsedTweets[indexPath.row]
    5     if self.splitViewController != nil {
  -       if (self.splitViewController!.viewControllers.count > 1) {
  -         if let tweetDetailVC = self.splitViewController!.viewControllers[1]
  -           as? TweetDetailViewController {
  -             tweetDetailVC.tweetIdString = parsedTweet.tweetIdString
   10        }
  -       }
  -     }
  -   tableView.deselectRowAtIndexPath(indexPath, animated: false)
  - }
```

This short method starts on line 4 by getting the ParsedTweet from our model that corresponds to the clicked row, just like in the navigation segue case.

The next few lines are a defensive nest of if statements, by necessity. Line 5 tests to see if the splitViewController property is non-nil, since we would only want to perform this logic if we're a child of a split view. If so, line 6 checks to see that there are at least two child view controllers, since we will need to work with the second one. Lines 7-8 attempt to cast the second VC to a TweetDetailViewController, to make sure we're talking to the kind of view controller we think we are.

If all of this works out, then line 9 assigns the tweetIdString property. As a handy side-effect, this kicks off the reloadTweetDetails() called by the didSet() property setter that we wrote way back in *Property Setters*, on page 163. In fact, this is the reason we needed to write that setter: in the navigation case, we could always count on viewWillAppear() to call reloadTweetData(), but in the split view scenario, the detail view will appear at launch and just stay there, so we need to make sure that setting tweetIdString will update the display.

Finally, 13 de-selects the tapped row, so it doesn't stay highlighted.

Run again and our selecting a tweet populates the detail view as expected. With all the space afforded by the iPad, geo-tagged tweets that bring up the map view make particularly good use of the iPad screen space, as seen in the following figure. We could something similar for tweets that have images, and they'd look great on the big iPad screen too.

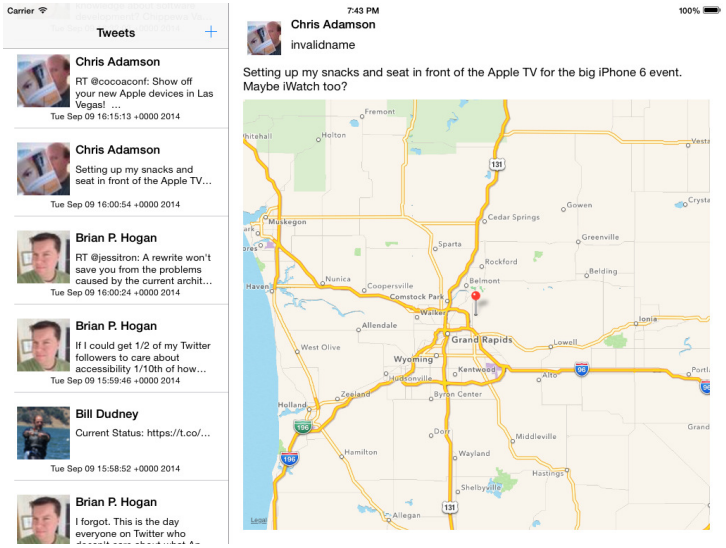


Figure 80—Selecting and displaying a geo-tagged tweet with split view

Split Views on iPhone

So it's great that we have our app making better use of the space on the iPad, but that begs the question of what's happening on the iPhone. Does it have a side-by-side split view too? How will that fit on a dinky iPhone 4s? We'd better check what's going on, so use the scheme selector to switch to one of the smaller iPhone models (the 4s, 5, or 5s) and run the app again.

What happens is pretty unexpected, as seen in the following figure.

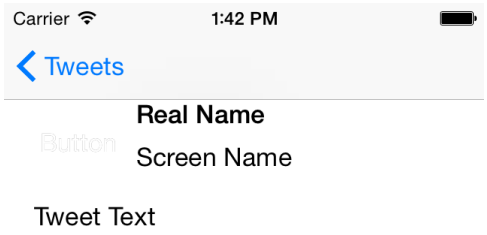


Figure 81—Unpopulated detail view on iPhone, with pseudo-navigation bar supplied by UISplitView

Somehow, our changes have caused us to start on a full-screen detail view instead of the master view with the list of tweets. Also, there's a navigation bar over the detail view, even though there wasn't one on top of the right pane of the split view on the iPad.

What's going on is that the split view controller realizes there's not enough space on the screen for both view controllers, so it has crunched down into a navigation-like metaphor for showing the two parts of the split on separate pages. This is a new feature for iOS 8, that lets us use a split view controller for both iPad and iPhone. In earlier versions of iOS, we had to have completely separate storyboards for iPhone and iPad. So this is a big win for building Universal apps — those that run on iPhone and iPad — if only it did the right thing out of the box.

Notice that the back button at the top left says "Tweets" (the title of the master view, which it gets from a navigation item), and we can tap it to go back to the list of tweets. However, if we tap one of the tweets, it doesn't

populate the detail view and take it to us. So we have two things to fix: we want to start on the master view controller (the list of tweets) instead of the detail, and we want tapping a table row to fill in the detail like it did in the old navigation app, and in the iPad version of the split view.

Handling Collapsing Split Views

The first step to dealing with starting on the wrong scene is knowing that we're even in this scenario and that we need to do something. Actually, we can get that by becoming the split view controller's delegate. The delegate gets told about changes like rotation, which cause it to rework how it presents its contents. It gets these callbacks at startup too, including one that says it's running in the compact space of an iPhone.

Start in RootViewController.swift by appending UISplitViewControllerDelegate to the comma-separated list of protocols in the @class declaration. This will allow our RootViewController to become the split view controller's delegate.

We want to become the delegate as soon as possible, so viewDidLoad() is a good place to do so. At the bottom of that method, add the following code:

BigScreens/PragmaticTweets-10-2/PragmaticTweets/RootViewController.swift

```
if self.splitViewController != nil {
  self.splitViewController!.delegate = self
}
```

All this does is to check if there's a splitViewController parent, just like we checked when we handled the table row tap. If there is, we become its delegate.

So now we can get the split view's delegate callbacks. We need to figure out which of its methods will tell us when we're running on the small space of an iPhone. It turns out the method we need is splitViewController(collapseSecondaryViewController:ontoPrimaryViewController:). This message is the split view telling the delegate that it doesn't have enough room for both view controllers, but gives the delegate a chance to adapt the second view controller into its user interface (perhaps by shrinking it or adding it to some other part of the UI), before the split view controller gives up and removes the second view controller.

Actually, removing the second view controller, the detail scene, is exactly what we want. If we just return true, the split view controller will give up on the detail view controller, leaving us with just the master view controller, which is the list of tweets. So implement the method like this:

BigScreens/PragmaticTweets-10-2/PragmaticTweets/RootViewController.swift

```
func splitViewController(_splitViewController: UISplitViewController!,
  collapseSecondaryViewController secondaryViewController: UIViewController!,
```

```
  ontoPrimaryViewController primaryViewController: UIViewController!) -> Bool {
    return true
}
```

Run the app on one of the iPhone models now, and we come up on the list of tweets.

Restoring discarded view controllers

Well, that's great, except that tapping on a row still doesn't do anything. And the reason for that is in our tap-handling logic in tableView(didSelectRowAtIndexPath:). When we implemented that before, we made sure there the split view controller had two child view controllers, so we could take the second one (the TweetDetailViewController) and populate it.

But we can't do that now, because *there is no second view controller.* We just told the split view controller that it was OK to discard the second view controller. So that's just great.

Maybe we'll just have to re-make that view controller ourselves! Fortunately, it's pretty easy to do so with storyboards. There's a UIStoryboard class that offers just three methods, two of which are for creating scenes from within the storyboard. The one we need is instantiateViewControllerWithIdentifier(), which takes a string and gives us back a UIViewController, with its view and all its subviews laid out exactly like we created them in the storyboard.

For this to work, we need to give the tweet detail scene a unique ID string. In the storyboard, select the "Tweet Detail View Controller", and bring up its Identity Inspector (⌥⌘3). In the field for "Storyboard ID", enter TweetDetailVC, as shown in the following figure.

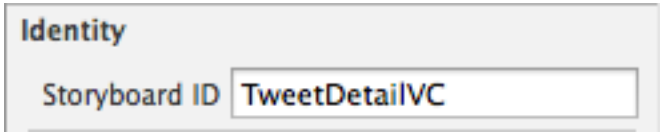


Figure 82—Setting Storyboard ID for TweetDetailViewController

Do a clean build ("Product"→"Build", or ⌘B), since changes to storyboards aren't always picked up by Xcode's build process, and we want to make sure this scene is findable by that string.

Now we can re-create this view controller when we need it. The place we're going to do so is in RootViewController's tableView(didSelectRowAtIndexPath:). We want to handle the case where the split view controller has only one child view

controller, so find the closing brace that matches `if (self.splitViewController!.viewControllers.count > 1) {`, and replace its closing brace with the following `else` block.

BigScreens/PragmaticTweets-10-2/PragmaticTweets/RootViewController.swift

```
Line 1 } else {
  2   if let detailVC =
  3     self.storyboard!.instantiateViewControllerWithIdentifier("TweetDetailVC")
  4       as? TweetDetailViewController {
  5     detailVC.tweetIdString = parsedTweet.tweetIdString
  6     self.splitViewController!.showDetailViewController(detailVC, sender: self)
  7   }
  8 }
```

On lines 2-4, we ask the storyboard to try to find the scene whose view controller has the ID `TweetDetailVC`, and cast it to a `TweetDetailViewController`. If that works, then we've got our detail view controller, and its whole view hierarchy, just as laid out in the storyboard. In turn, that means we can get it to load its contents like we always have, by setting its `tweetIdString` (on line 5). Then we just have to navigate to it. `UISplitViewController` gives us that ability with the `showDetailViewController()` method, on line 6.

And that's it! Run the app on a simulated iPhone, and it works just like the navigation version did from the previous chapter, perfectly well-suited to the small space of the iPhone. Back on the iPad, we get a side-by-side split that makes better use of all the screen real estate. Best of both worlds, and in iOS 8, we get it all with one storyboard and this little bit of tricky code.

Size Classes and the iPhone 6

So it's great that the split view gives us one behavior for iPad, and another for iPhone.

Except, well, that new iPhone 6 is *really* big. Maybe not iPad big, but it at least makes you wonder whether it's really appropriate to lump all iPhones together. After all, iOS 8 runs on everything from the 4s (with its 320x480, 3.5 inch screen) to the iPhone 6 Plus (414x736, 5.5 inch screen). Even within the iPhone range, there may be times we want to go to a side-by-side mode with our split view.

To do so, we need to understand how iOS 8 represents sizes.

Size Classes

In iOS 8, screens, view controllers, and views all have a collection of sizing information called a *trait collection*. These traits are collected by the `UITraitCollection` class, and include things like the points-to-pixels scaling factor (2.0 for

Retina devices, 3.0 for the iPhone 6 Plus, 1.0 for the old iPad 2), the device idiom (phone or pad), and a very general way to represent the available space in each dimension.

The available space is represented as a *size class*, and has two values: "compact" and "regular". Those should sound familiar, because they're the values of the sizing bar at the bottom of the storyboard pane, which we saw way back in *Storyboards*, on page 19. Let's think back to the grid that appeared in the popup, and the descriptions it provided: regular width and height were described as an iPad, while compact width and regular height represented an iPhone in portrait orientation.

Traits are inherited from the screen, to the one window that's always on the screen, through view controllers, and views, down to each individual view, like a button or table. Along the way, they can be changed. So an iPad screen will have regular width size class in either orientation, but the left side of a split view will have compact width, since the layout of the split view constrains how much space it can use.

The split view's decision about whether to use a side-by-side or a two-screen layout is based on the width size class it inherits. If it thinks it's in a compact space, it will use the navigation-style two-screen approach; if it inherits regular width, it will use a side-by-side presentation.

So, if we wanted to make the split view go side-by-side on an iPhone 6, we would just have to convince it that it has regular width to work with, not compact. Let's do that.

Container Controllers

We can't just tell the split view controller that it has regular width: it has to inherit this from a parent view or view controller. We can do that by creating a *container controller*, a view controller that contains other view controllers. This parent will own the split view controller, and be in a position to give it a trait collection that says it has regular width, if we decide we want to go side-by-side.

To get started, go to the storyboard, look through the Object Library (^⌥⌘3) and find the plain "View Controller" icon, a yellow ball with a dashed box inside it. Drag this icon to the left of the split view controller.

The way we get this view controller to "own" the split view controller is pretty weird. We need to use a *container view*, instead of the usual `UIView` that comes with a view controller. So, in the scene list, find the view that's a child of this

new view controller, and delete it. This will also delete the top and bottom layout guide objects from the scene. Now, in the object library, find the "Container View" icon, shown in the following figure as a gray box inside a white box. Drag it onto the new view controller icon or its box in the storyboard; it will become the sole child of the view controller.

Figure 83—Container View icon in Object Library

Now we need to associate this container view with the view controller we want it to contain. This is actually done with a segue. Control-drag from the container view to the split view controller. This will allow us to make a segue connection. At the end of the drag, choose "embed" as the segue type. We should always name our segues, so click the segue icon between the two scenes, bring up its Attribute Inspector (⌥⌘4), and give it the storyboard identifier `embedSplitViewSegue`.

Control click on the container view to show its connections. For "Triggered Segues", it will show `viewDidLoad()` via an "embed" segue to the "Split View Controller". This is the weird part: this segue isn't performed as part of a navigation like in the last chapter; it happens when the view loads, at which point the container view controller will get its one and only look at the child view controller that it's going to contain.

This scene is going to be the beginning of our app, so choose the view controller, go to the attributes inspector (⌥⌘4), and select the "Is initial view controller" box. This won't visually change anything, since control will flow immediately to the split view controller child. But it will ensure that this view controller loads first, which we need. The beginning of the storyboard should now look like the following figure.

Figure 84—Container View Controller connected via embed segue to Split View Controller in storyboard

Now we're at the point where we need to write some code, to grab the reference to the split view controller and be able to change its trait collection. In the File Navigator, select the "Pragmatic Tweets" group and use "New Group…" to create the group "Size Class Override". Within this group, do "New File…" to create a new "Cocoa Touch Class", with the name SizeClassOverridingViewController, a subclass of UIViewController written in Swift. Finally, choose the new view controller in the storyboard, bring up its Identity Inspector (⌥⌘3) and change the class to SizeClassOverrideViewController. Now the container VC will be our custom class.

What we need to do with this code is grab a reference to the split view controller, and send it our preferred size class traits. In SizeClassOverridingViewController.swift, start by creating a property to refer to the UISplitViewController.

BigScreens/PragmaticTweets-10-3/PragmaticTweets/SizeClassOverridingViewController.swift
```
var embeddedSplitVC : UISplitViewController?
```

As implied by the name of the embed segue, our one look at the split view controller happens when the view loads. This will make a call to prepareForSegue() — just like when we're navigating between scenes — so we override that method to grab the reference to the destinationViewController.

BigScreens/PragmaticTweets-10-3/PragmaticTweets/SizeClassOverridingViewController.swift
```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
  if segue.identifier == "embedSplitViewSegue" {
    self.embeddedSplitVC = segue.destinationViewController as? UISplitViewController
```

```
  }
}
```

Now we're in a position to start changing the split view's sense of how much room it has to work with!

Overriding Trait Collections

Now let's think about what we want to tell the split view controller, and when. It will use a side-by-side view when it thinks its width size class is "regular" and not "compact". We could probably swing the side-by-side view on an iPhone 6, but not a 4s. Let's look at our sizes.

Model	Dimensions (points)	Scale factor
iPhone 4s	320 x 480	2.0
iPhone 5 / 5s	320 x 568	2.0
iPhone 6	375 x 667	2.0
iPhone 6 Plus	414 x 736	3.0
iPad 2	768 x 1024	1.0
iPad Retina / iPad Air	768 x 1024	2.0

So, assuming that we don't want to try to go side-by-side on the 4s, and we definitely want to do so on the big iPhone 6 models, let's pick a value in the middle. We'll just say that any width bigger than 480 is enough for us to try side-by-side.

When the device is rotated, view controllers get a callback to the method viewWillTransitionToSize(). That's the perfect place to pull our trickery. We'll override that method, check the size we're transitioning to, and if it's wide enough for us, we'll override the split view controller's trait collection. Here's how we do that.

BigScreens/PragmaticTweets-10-3/PragmaticTweets/SizeClassOverridingViewController.swift

```
override func viewWillTransitionToSize(size: CGSize,
  withTransitionCoordinator coordinator: UIViewControllerTransitionCoordinator) {
  if size.width > 480.0 {
    let overrideTraits = UITraitCollection (
      horizontalSizeClass: UIUserInterfaceSizeClass.Regular)
    self.setOverrideTraitCollection(overrideTraits,
      forChildViewController: embeddedSplitVC!);
  } else {
    self.setOverrideTraitCollection(nil,
      forChildViewController: embeddedSplitVC!);
  }
}
```

We look at the incoming width on line 3. If it's greater than 480.0, lines 4-5 create a new UITraitCollection. The initializers for this class take either one of the four traits — horizontalSizeClass, verticalSizeClass, userInterfaceIdiom, or scaleFactor, — or an array of already-initialized trait collections to merge together. All we care about it setting the horizontalSizeClass to the enum value UIUserInterfaceSizeClass.Regular.

Then all we have to do is pass this to our embeddedSplitVC. A parent view controller can override a child's trait collection with setOverrideTraitCollection(), which is what we do on lines 6-7. This is only possible from a parent view controller — other VCs can't go changing each other's trait collections willy-nilly — which is why we had to go through the whole rigamarole of setting up our custom container controller.

Finally, if our width isn't big enough for the split view controller go to into side-by-side mode, we do setOverrideTraitCollection() with nil, on lines 9-10 which lets it inherit its traits as before.

With our sneaky override of the size class now complete, run the app again on different models in the simulator (keeping in mind you'll have to sign into Twitter on each one if you haven't already, as they store their system settings separately from each other). On a sufficiently large device, the spilt view controller will now go into side-by-side mode when rotated to landscape, as seen in the following figure.

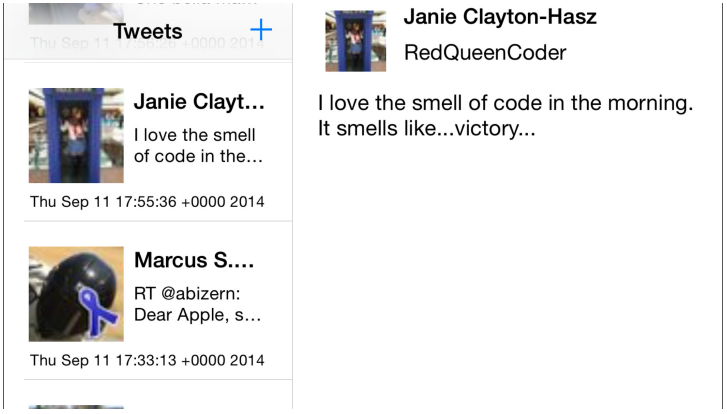


Figure 85—iPhone 6 Plus displaying UISplitView as side-by-side views in landscape orientation

Wrap Up

In this chapter, we stopped looking at our app through iPhone goggles. We thought about how it used the space available on an iPad, and switched to a split view to make better use of all the screen real estate. We discovered that this gives us a navigation-like experience for iPhone sizes when the side-by-side view doesn't really make sense, although we did have to do a little work to make it work like we wanted it to. And in between the extremes of the little iPhone 4s and the iPad, there are now several iPhone models at intermediate sizes, so we learned how to inspect our size classes and even change them.

Having said that, this isn't really where we started dealing with bigger screens. By using autolayout, we've been dealing with differing screen sizes all along. Instead of nailing our UI components to specific coordinates and sizes, we've used constraints like "center this button horizontally", "put this text view 8 points below this other one, wherever it is", and "let this label use whatever space is available inside 20-point insets from the superview's border". Thinking that way, and using these kinds of relative layout instructions work on screens of different sizes and shapes. Apple calls this an *adaptive* user interface, one that adapts not only to the physical factor of the device, but also to user preferences, like larger fonts for vision-impaired users.

So, we've done some work on how users see our app. Now it's time to enhance how they interact with it. In the next chapter, we'll look at how to create and handle touch gestures on our own, so our app won't just look great, it will literally feel great, too.

CHAPTER 11

Recognizing Gestures

Touch is the defining trait of user interfaces on the iPhone and iPad. It's what makes working with the data seem so direct: flicking a table to scroll it, pinching a photo to resize it, or drawing freehand with our fingers. iOS builds in sensible touch controls for all of its provided views, and we can build upon those further by creating our own.

Gesture Recognizers

The first versions of the iPhone SDK gave us only a low-level look at raw touch data. `UIView` provides the methods `touchesBegan()`, `touchesMoved()`, `touchesEnded()`, and `touchesCancelled()`. These methods delivered sets of `UITouch` events, and from the raw geometry and timing of these events, we could track events like swipes, using logic like "if the touch moved at least 50 points up, and not more than 20 points to either side, in less than 0.5 seconds, then treat it as an upwards swipe."

As one might expect, this was a huge pain in the butt to actually implement, and led to variations in user experience as different developers interpreted the touch data differently, based on what "felt right" to them.

Fortunately, the situation was cleaned up in later versions of iOS thanks to *gesture recognizers*. With these classes, iOS determines for us what counts as a swipe or a double-tap, and calls into our code only when it detects that a matching gesture has occurred.

The top-level `UIGestureRecognizer` class represents things like a gesture's location in a view, its current state (began, changed, ended, etc.), and a list of target objects to be notified as the recognizer's state changes. Subclasses provide the tracking of distinct gestures like taps, pinches, rotations, and swipes,

and these subclasses also contain properties representing traits specific to the gesture: how many taps, how much pinching, and so on.

Segue Gestures

One handy trick for our Twitter app would be to give the user a better view of a given tweeter's avatar. From the user detail screen we built in the last chapter, we could go to a new screen that shows the image in a larger view, and allow our user to pinch-zoom and move around the avatar in detail.

To do this, we'll need a new "user image detail" scene in the storyboard. Find the View Controller icon in the Object Library, and drag it into the storyboard, to the right of the User Detail View Controller scene that is currently the end of our storyboard. To this new scene, add:

- An image view, with width and height pinned to 280 points, vertically and horizontally aligned in its container.
- A button, with title "Done", pinned 20 points up from the bottom of the container, horizontally aligned in the container.

We'll put some logic into that scene later, but for now, we just need to create a way to get to it from the user detail scene. We could do that by replacing the detail scene's image view with a button and then adding a segue on the button tap. But to show how flexible gesture recognizers, we'll do functionally the same thing by giving the existing image view the ability to handle taps, thereby turning it into a de facto image button.

Scroll through the Object Library and find the gesture recognizer icons. They're displayed as blue circles against dark gray backgrounds, some with swooshes that represent movement. Find the tap gesture recognizer, which is represented as a single static circle, shown in the following figure.

Figure 86—Tap Gesture Recognizer Icon in Object Library

Drag this icon onto the `UIImageView` that's above the "User Name" label in the User Detail View Controller scene. This won't cause an immediate change to the image view, but the gesture recognizer will become a top-level member of

the of the scene, a sibling to the view controller and the various segues, in the list on the left. Select it from this list and view it with the Attributes Inspector (⌥⌘4). As shown below, the gesture recognizer allows you to configure the number of sequential taps (single-, double-, triple-, etc.) and the number of touches (how many fingers touching the screen) required to trigger the recognizer.

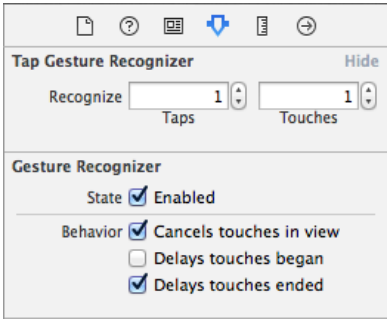


Figure 87—Tap Gesture Recognizer Properties

Connecting Gesture Recognizers

One thing we don't see here is how the gesture recognizer is related to the image view. For that, go to the connections inspector (⌥⌘7). There we see that the "Referencing Outlet Collections" have a property called gestureRecognizers that is connected to the "Image View" (if it just says "View", you probably dropped it on the full view and not the image view; delete the recognizer from the scene and try again).

So it's not that the recognizer refers to the view; instead, the view knows that the recognizer is one of its potentially many gestureRecognizers. Now let's address the question of the what the recognizer does when it's tapped. In the connections inspector, we see a few interesting properties: a triggered segue action, a delegate outlet, and a sent action selector. The delegate doesn't actually help us here: the UIGestureRecognizerDelegate is meant to let our code adjudicate when two gesture recognizers want to handle the same gesture. What's useful for us are the selector, which calls a method when the gesture begins, ends, or updates, and the segue action, which takes us to a new scene.

What we want is the segue, so draw a connection line by dragging from the circle next to action in the Connection Inspector to anywhere in the new image detail scene. It would also work to do a control-drag from the gesture recognizer in the user detail scene over to the image detail scene; Xcode will figure out that a connection between scenes can only be a segue (and not some

other kind of connection). At the end of the drag, a popup asks what kind of segue we want; coming from a modal scene, the only choice that will work is another "present modally" segue. Our gesture recognizer's connections should now look like the following figure:

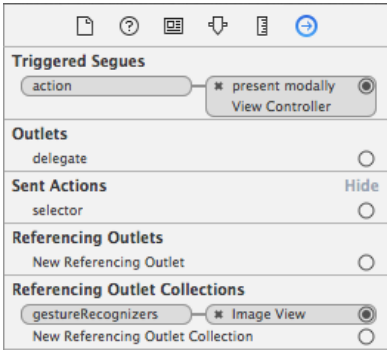


Figure 88—Tap Gesture Recognizer Connections

We can try running now, choosing a tweet, and drilling down to user details, but clicking the image won't perform this segue yet. To see why, select the image in the user detail scene, and bring up the Attributes Inspector (⌘⌥4). Notice that "User Interaction Enabled" is unchecked, since image views by default don't handle user input. But this means that it won't process touch events, which in turn means our gesture recognizer will never fire. Simple fix here: just check the "User Interaction Enabled" box.

Run the app now and we can drill all the way to our new scene, which at this point only shows a "Done" button, since we haven't populated the image view yet. Moreover, the "Done" button doesn't work, and we're trapped on this scene. Let's fix that before we move on. The fix is to use an unwind segue. Back in UserDetailViewController.swift, add an empty implementation for unwindToUserDetailVC()

Gestures/PragmaticTweets-11-1/PragmaticTweets/UserDetailViewController.swift

```
@IBAction func unwindToUserDetailVC (segue : UIStoryboardSegue) {
}
```

Now, we can go to the image detail scene in the storyboard, and control drag from the "Done" button to the green Exit Segue button. At the end of the drag, we have two methods we can unwind to: choose the unwindToUserDetailVC() method we just created. Run again, and we can go back from the image detail scene.

So what we've accomplished at this point is to bring tap handling to a UIImageView, a class that ordinarily supports no user interaction whatsoever. And

we did it without really writing any code — we just created the gesture recognizer in the storyboard, connected it to a new segue, and gave ourselves a no-op method to unwind to.

But we're just getting started. There's a lot more we can do to the default image view.

Populating the Image

Before we start gesturing around with the image view, it'll help to actually have an image we can see. So let's deal with that now.

In the File Navigator, create a new group called "User Image Detail VC", and within that, use "New File..." to create a new class UserImageDetailViewController, a subclass of UIViewController. At the top of this new UserImageDetailViewController.swift file, declare a property for the user image URL:

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserImageDetailViewController.swift
```
var userImageURL : NSURL?
```

We'll set that property every time we follow the segue to the new scene, so we have some work to do in the storyboard. First, select the image detail scene's view controller icon (either from the frame below the scene or in the scene's object list), go to the Identity Inspector (⌘⌥3) and change the class to UserImageDetailViewController. Since we will need to know when we're taking the segue to this scene, select the segue, bring up the Attributes Inspector (⌘⌥4), and set the identifier to showUserImageDetailSegue.

Now we're going to be able to set the image URL when we segue to the new scene. We do this back in UserDetailViewController.swift. Next, we need to save the URL of the image. Right now, the user detail scene just creates a UIImage to populate this class' image view, but there's a good reason it should save off the URL: it will let us get a higher-quality image. Currently, it uses the key profile_image_url to get an image URL from the Twitter response. The value is a URL string like https://pbs.twimg.com/profile_images/290486223/pp_for_twitter_normal.png. As it turns out, that _normal is used by Twitter to indicate an icon at a standardized 48x48 size. That's fine for the user detail VC, but it will be very blocky in the 280x280 image view in the next scene. Fortunately, if we just strip the _normal, we can get the image in the original size uploaded by the Twitter user, and that will look nicer in the next scene. So start by giving the UserDetailViewController this new property:

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserDetailViewController.swift
```
var userImageURL : NSURL?
```

Then, down in handleTwitterData(), inside the closure, change the last two lines so that they use this property to set the userImageView.image, rather than a local userImageURL variable.

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserDetailViewController.swift
```
self.userImageURL = NSURL (string:
  tweetDict ["profile_image_url"] as NSString!)
self.userImageView.image = UIImage(data:
  NSData(contentsOfURL: self.userImageURL!))
```

Now we're ready to send the good version of the user image URL to the UserImageDetailViewController, by writing a prepareForSegue:() method.

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserDetailViewController.swift
```
override func prepareForSegue(segue: UIStoryboardSegue?, sender: AnyObject?) {
  if segue!.identifier == "showUserImageDetailSegue" {
    if let imageDetailVC = segue!.destinationViewController
      as? UserImageDetailViewController {
      var urlString = self.userImageURL!.absoluteString;
      urlString = urlString!.stringByReplacingOccurrencesOfString ("_normal",
        withString: "")
      imageDetailVC.userImageURL = NSURL(string: urlString!)
    }
  }
}
```

We begin on line 2 by checking that we're doing the segue to the user image detail VC. If so, we can cast the destinationViewController on lines 3-4. We can then get an NSString version of the URL (line 5), and strip out the _normal substring (lines 6-7). Finally, we make a new NSURL for the full-size image and send it to the user image detail VC on line 8.

Once the segue is performed, the UserImageDetailViewController will have the URL for the full-size image. Now all we need to do is to populate the image view in that scene. Start by going to the storyboard, going to the last scene, and selecting the 280x280 image view. Bring up the Assistant Editor (the "tuxedo" toolbar button, or ⌥⌘↩), with UserImageDetailViewController.swift in the right pane, and control-drag from the image view in the storyboard to somewhere inside the @class (perhaps right after the userImageURL we created), to create an outlet that we'll name userImageView.

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserImageDetailViewController.swift
```
@IBOutlet weak var userImageView: UIImageView!
```

Now that we can see the image view in code, go back to the standard editor, visit UserImageDetailViewController.swift, and add a viewWillAppear() method:

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserImageDetailViewController.swift
```
override func viewWillAppear(animated: Bool)  {
```

```
  super.viewWillAppear(animated)
  if self.userImageURL != nil {
    self.userImageView.image = UIImage(
      data: NSData (contentsOfURL: self.userImageURL!))
  }
}
```

Run the app now, and we can navigate all the way to the image detail scene, which will show the higher-quality user image and not the 48x48 icon. In the figure below, we've drilled down for a look at Janie's Twitter avatar.

Figure 89—Original-Quality Image in User Image Detail Scene

That's a nice, normal looking image for now. But we're about to start letting our fingers have some fun with it.

Pinching and Panning

How can we play with images on iOS? The whole point of a touch interface is to provide the feeling of interacting directly with our data, so we should be thinking of moving the image around with a drag, zooming in and out of it with pinch gestures, etc.

Let's take a look at what gesture recognizers give us. Here's a table summarizing the concrete subclasses of UIGestureRecognizer, and the important properties and/or methods exposed by each:

Class	Important Properties and Methods
UILongPressGestureRecognizer	minimumPressDuration, allowableMovement
UIPanGestureRecognizer	translationInView:, velocityInView:
UIPinchGestureRecognizer	scale, velocity
UIRotationGestureRecognizer	rotation, velocity
UIScreenEdgePanGestureRecognizer	edges
UISwipeGestureRecognizer	direction
UITapGestureRecognizer	numberOfTapsRequired , numberOfTouchesRequired

As we look at the names of the gesture recognizers, we can start to get some ideas: UIPanGestureRecognizer handles dragging a finger around, so we can use that to move the image around. The UIPinchGestureRecognizer seems like it would be a natural for pinch-to-zoom functionality. So it looks easy enough to recognize the gestures we want. Question now is: what do we do with it? How is a scale or translationInView() going to help us change the appearance of the image view?

Affine Transformations

The properties and methods provided by the gesture recognizers work well with a trait common to all graphic objects in iOS: *affine transformations*. A transformation, speaking generally, changes how we draw something. More technically, transformations indicate how points in one coordinate system map to another. Affine transforms are special, because they maintain parallel lines between the two coordinate systems.

A specific example of how we already use affine transforms may be helpful here. Think of how when you print out a document, you can save paper by using the printer dialog to print two pages of the document on one sheet of physical paper, putting two portrait-oriented pages side-by-side on one landscape page. To do that, each page of your document goes through three transformations:

- The page is rotated 90°, so that it prints "sideways".
- The page is shrunk down (*scaled*) by about 50%.

- The page is moved (*translated*) so that odd pages are left-aligned against the edge of the portrait page, and even pages are left-aligned approximately along the center fold of the page.

Each of these can be represented as an affine transform. Moreover, all of them can go in a single transform, by simply applying each transform to the one that came before it.

For our purposes, every UIView has a transform of type CGAffineTransform. This is a struct, not a class, and consists of just six CGFloats: a, b, c, and d, tx and ty. These six values represent any combination of rotation, scaling, skewing, and translation (movement) operations. Technically, they represent six members of the matrix in the following equation.

$$[x'\ y'\ 1] = [x\ y\ 1] * \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ tx & ty & 1 \end{bmatrix}$$

Figure 90—Matrix Equation for CGAffineTransform

What this equation provides is the transformed values for any point, x' and y', given their original values (x and y) and the contents of the affine transform matrix. This works out to a pair of simple equations:

x' = ax * cy + tx
y' = bx * dy + ty

Notice that the tx and ty values stand alone as terms in the equations. These are the "transform" values. If a, b, c, and d are all 1.0, then tx and ty can be used directly along the x and y axes.

On the other hand, if we only work with a, b, c, and d, we can easily scale an object: if they're all 2.0, then every coordinate value will double, and this transform will represent doubling the size of of an object. Or we can use sines and cosines to represent rotation. Or we can use all the terms to combine scaling, rotation, and translation.

Transforms and Layers

The CALayer objects that provide the actual drawing of our views have a different way of representing transforms, the CATransform3D. As its name implies, this transform works in three dimensions, with a z-axis that comes out of the screen towards the viewer. Any time we want to do transforms that work with a sense of depth, like views that flip over or are viewable from the side, we need to work at the CALayer level.

Fortunately, we don't have to use the members of CGAffineTransform directly. In fact, we almost never do. Core Graphics provides a set of convenience functions to create affine transforms for rotation, scaling, and translation operations, either as absolute values, or as modifications of existing affine transforms. So if we ever found ourselves writing the printer driver that had to do side-by-side printing as described earlier, we could create one affine transform to to the rotation, use that to create a transform to do the scaling (of the rotated page), and then use that to make the translation (of the rotated and scaled page).

Transforming the Image View

Now that we see what affine transforms offer us, the properties and methods exposed by the gesture recognizers start to make more sense. The pinch gesture recognizer provides a scale that we could use to make a scale transform, and the pan recognizer offers translationInView() that will be perfect for making a translation.

To make use of these transforms, we have a few options. UIView has a transform property, so we can set that directly. The underlying CALayer that provides the view's appearance also has a transform property, although that one is of type CATransform3D and works in three dimensions. A more advanced option would be to write our own subclass of UIView or CALayer that draws its own contents; the Core Graphics library used for drawing allows us to set an affine transform on our drawing operations, and is the only way to use multiple transforms. To keep it simple for now, we'll just reset the UIImageView's transform property, which it inherits from UIView.

The Pan Transform

Let's start with a pan transform to move the image around. In the storyboard, go to the user image view detail scene, the one with the 280x280 image and the "Done" button. In the Object Library, find the Pan Gesture Recognizer, which looks like a blue circle leaving a streak below it (see figure below). Drag and drop the pan recognizer onto the image view.

Figure 91—Pan Gesture Recognizer Icon in Object Library

Now we need to give the recognizer a method it can call. Switch to Assistant Editor (⌥⌘↩), making sure UserImageDetailViewController.swift is in the right pane. Control drag from the gesture recognizer (either in the scene's object list or from the bar atop the scene), to any free space inside the @class, perhaps down by the closing curly-brace. At the end of the drop, a popup asks for what kind of connection to make — be sure to change from "Outlet" to "Action" and for a name for the action method. Let's call it handlePanGesture(). Also, before clicking "Connect", change the Type from the default id to UIPanGestureRecognizer.

This connection will call handlePanGesture:() when a pan gesture starts, updates or ends on the image view. At least it *would*, if image views processed touch events by default. Just as with the image in the previous view controller, we have to explicitly enable user interaction with this image view to make it respond to touch events. Switch back to the storyboard's standard editor, select the image view, bring up its attributes inspector, and select the "User Interaction Enabled" checkbox.

Switch back to the standard editor and bring up UserImageDetailViewController.swift, so we can write this method that we just connected. This is where we're going to ask the gesture recognizer how far it's moved, and use that to update the image view's affine transform.

For this to work, we need to understand what the gesture recognizer tells us. If we look up translationInView() in the documentation for UIPanGestureRecognizer, we find it returns "a point identifying the new location of a view in the coordinate system of its designated superview.". There's also an important note in the discussion of the method:

> The x and y values report the total translation over time. They are not delta values from the last time that the translation was reported. Apply the translation value to the state of the view when the gesture is first recognized—do not concatenate the value each time the handler is called.

What this is telling us is that as we get new callbacks as handlePanGesture:() is repeatedly called during the drag, the value reported back to us is relative to the image view's initial transform, not the last value we set it to. That means

we should plan on saving the image view's transform the first time we get called. Define that as a property up in the top of the @class:

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserImageDetailViewController.swift
```
var preGestureTransform : CGAffineTransform?
```

Now we can assign that property the first time we're called back by the gesture recognizer. When we're called back, we can ask the gesture recognizer for its state, which can be started, changed, ended, cancelled, or a few other administrative and error states. When the value is UIGestureRecognizerStateBegan, we'll save off the initial transform of the image view. Begin the handlePanGesture:() like this:

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserImageDetailViewController.swift
```
@IBAction func handlePanGesture(sender: UIPanGestureRecognizer) {
  if sender.state == UIGestureRecognizerState.Began {
    self.preGestureTransform = self.userImageView.transform
  }
```

When a pan gesture begins, this saves the image view's transform to our preGestureTransform property, since all subsequent event coordinates will be relative to this initial transform. Now we're ready to handle actually moving the view around:

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserImageDetailViewController.swift
```
Line 1 if sender.state == UIGestureRecognizerState.Began ||
   2   sender.state == UIGestureRecognizerState.Changed {
   3     let translation = sender.translationInView(self.userImageView)
   4     let translatedTransform = CGAffineTransformTranslate(
   5       self.preGestureTransform!, translation.x, translation.y)
   6     self.userImageView.transform = translatedTransform
   7 }
```

We get the translationInView() on line 3. This is a CGPoint whose x and y members represent how far we have moved along each axis from where the pan began. With that information, we can use the CGAffineTransformTranslate() function to create a new transform that represents that distance from the original preGestureTransform (lines 4-5). Then, on line 6, we just set that as the new transform property of the image view.

Does this work? Try it. Drill down to an user image detail and try dragging the picture around. You should find you have total freedom to put it wherever you like, even under the done button or partially off-screen, as seen in the following figure. Pretty cool, but we should clean up after ourselves before we go further.

Figure 92—Dragging UIImageView by Updating Its transform via a UIPanGestureRecognizer

The Identity Transform

So it's great that we can drag the image wherever we like… but that does mean we can drag it completely off the screen. Problem!

Let's give ourselves a "panic button": if the user double-taps the image, it'll go back to its default position.

In the storyboard, add a new tap gesture recognizer to the image view. Select the tap gesture recognizer icon from the scene's object list or the title bar under the scene, bring up the Attributes Inspector, and set the number of taps to 2. This means it will take a double-tap for the recognizer to fire.

Next, switch to Assistant Editor, and control drag from the tap gesture recognizer into UserImageDetailViewController.swift to create a new action method. When the popup appears at the end of the drag, call the method handleDoubleTapGesture(), and switch the type from id to UITapGestureRecognizer.

So how do we write this method? We want to go back to the image view's original transform, before any of our changes. By default, UIViews have an *identity transform*, which means no scaling, rotation, or translation. This is a CGAffineTransform where a, b, c, and d are all 1.0, and tx and ty are 0.0. Run that through the earlier formulas and we find that makes x' equal x and y' equal y. This "do nothing" is provided to us as the constant value CGAffineTransformIdentity.

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserImageDetailViewController.swift

```
    @IBAction func handleDoubleTapGesture(sender: UITapGestureRecognizer) {
self.userImageView.transform = CGAffineTransformIdentity
    }
```

Restoring the identity transform on a UIView is a one-line call. Run the app, drag the image around, and double-tap to send it back to where it started. Easy peasy!

The Scale Transform

The other common gesture we should add to our image viewer is a pinch-to-zoom feature. Again, this naturally links the scale property of the gesture recognizer — in this case a UIPinchGestureRecognizer — to the ability of affine transforms to perform scaling operations.

Back in the storyboard, go to the Object Library and locate the pinch gesture recognizer icon, whose icon is shown in the following figure. As before, drag it on to the image view to add it to the scene. Switch to Assistant Editor with UserImageDetailViewController.swift in the right pane, select the icon in the scene or the title bar, and control-drag to create a new action method. Name the action handlePinchGesture() and change the parameter type to UIPinchGestureRecognizer.

Figure 93—Pinch Gesture Recognizer icon in Object Library

What does the pinch gesture's scale give us? According to the docs, it's "the scale factor relative to the points of the two touches in screen coordinates". And, as was the case with the pan recognizer, this value is relative to the beginning of the gesture, not to the last time we were called. So, once again, we need to make use of the preGestureTransform to hold on to our initial value.

Gestures/PragmaticTweets-11-2/PragmaticTweets/UserImageDetailViewController.swift

```
Line 1 @IBAction func handlePinchGesture(sender: UIPinchGestureRecognizer) {
   -   if sender.state == UIGestureRecognizerState.Began {
   -     self.preGestureTransform = self.userImageView.transform
   -   }
   5   if sender.state == UIGestureRecognizerState.Began ||
   -     sender.state == UIGestureRecognizerState.Changed {
   -       let scaledTransform = CGAffineTransformScale(
   -         self.preGestureTransform!, sender.scale, sender.scale)
   -       self.userImageView.transform = scaledTransform
  10   }
   - }
```

As with the pan recognizer, we use the start state to save off the image view's initial transform, on lines 2-4. Then on lines 5-6, we deal with the scale value of a started or changed event. On lines 7-8, we use CGAffineTransformScale() to create a new CGAffineTransform by taking the original preGestureTransform and applying the scale value to both the x and y factors of the scaling transform. And then on line 9, we set this as the new value of the image view's transform.

Run the app and give it a whirl. To simulate a pinch gesture in the simulator, hold down the option key on the keyboard, which will show the pinch points as two circles that move with the mouse or trackpad. By adding the shift key, we can move the pinch points without registering as a pinch. In the following figure, we've panned to the right and pinch-zoomed in to pick out two *Neon Genesis Evangelion* cosplayers coming off the escalator behind Janie (yes, her Twitter avatar is from an anime convention, how did you guess?).

Figure 94—Scaling UIImageView by updating Its transform via a UIPinchGestureRecognizer

To better understand the math behind the transform, try changing the x- and y-scaling values sent to `CGAffineTransformScale()`. For example, if we set the last argument, `sy` to the constant value `1.0`, then the pinch will become a horizontal stretching operation, since the y value will always be the same after the transform (since it's being multiplied by 1). Another fun trick is to multiply the scaling value by `-1.0`, which causes the image to flip around the axis, making it an upside-down mirror image.

Subview Clipping

Thanks to the natural pairing of the gesture recognizers and the affine transforms, we've added the dragging and pinch-zooming functionality that will be familiar to our users from many other apps they use. However, views that are allowed to just sprawl all over the screen aren't something we usually see on iOS. It may be fun, but it feels wrong, and we hardly want to let the user make a mess of our user interface.

Let's rein in the madness a little bit. We'll put the image view into another view, and have it clipped off at that view's edges. That will put an end to sliding the image offscreen or under the "Done" button.

In the storyboard, select the image view and delete it (with the Backspace key or the Edit→Cut menu item). Notice that the three gesture recognizers survive this, because they are top-level objects in the scene, and not children of any view or view controller.

From the Object Library, find the plain View (the popover will show its class as UIView), and drag it to the middle of the scene's main view. Use the autolayout popovers to pin its width and height to 280 and to horizontally and vertically align it in the container. Then go to its Attributes Inspector and check the "Clip subviews" box. What this does is to constrain ("clip") drawing to the bounds of the view, so if the contained image view goes beyond those boundaries, it will just get cut off.

Next, drop an image view into this subview. It should allow itself to fill the parent subview; one way to make this work for sure is to drag the image view onto the subview's entry in the scene list, rather than onto the storyboard layout. Like its parent, create autolayout constraints pin its size to 280x280 and horizontally and vertically align it in its containers.

Now we need to fix our connections. Select the view controller from the scene members list and bring up the Connections Inspector (⌘⌥6). The userImageView is no longer connected, because we deleted the object it was connected to. Drag from that connection's circle to the new image view to make a new connection.

The gesture recognizers also have no incoming connections anymore, so they won't be called. To fix that, we're going to connect them to the 280x280 plain view, rather than directly to the image view. Select the subview, and repeatedly drag from its gestureRecognizers entry in the Connections Inspector to each of the gesture recognizers, ultimately creating three connections.

Try running the app and drill down to the image detail view. When we drag around, any part of the image that goes beyond the bounds of the 280x280 view is simply cut off, as shown in the figure below. We can also perform our gestures anywhere in the view, not just on the image view, and have them recognized. That means we can push the image entirely outside the bounds of the container view, but we can also bring it back with a double-tap anywhere in that view.

Figure 95—Clipping a Transformed Image to bounds of its Superview

Wrap Up

In this chapter, we took hold — literally — of the touch gestures that are the hallmark of iOS user interfaces. By giving the user the ability to manipulate an image by dragging it around with one finger and pinch-zooming it with two, we immediately create a sense of close contact with the image. Gesture recognizers make it easy to pick up the most common touch gestures, and have them call back to our code when gestures are detected. And because both the recognizers and the on-screen views are concerned with how much movement or scaling is indicated by a gesture, it works well to connect the two by means of affine transforms, which cleanly represent translation, rotation, scaling, and combinations thereof.

Armed with this knowledge, we can bring new touch handling features to scenes throughout our app. It will also be useful if we ever need to create our own custom views, since a view is basically a combination of appearance and interactivity, meaning that a custom view just needs to handle custom

drawing and custom event-handling. And we just learned how to do the second of those things.

CHAPTER 12

Working with Photos

CHAPTER 13

Launching, Backgrounding, and Exensions

CHAPTER 14

Debugging Apps

CHAPTER 15

Publishing to the App Store

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
http://pragprog.com/book/adios2
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *http://pragprog.com/book/adios2*

Contact Us

Online Orders:	*http://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://pragprog.com/write-for-us*
Or Call:	+1 800-699-7764