Up to date for iOS 9, Xcode 7.3 & Swift 2.2

iOS 9 by Tutorials

Learning the new iOS 9 APIs with Swift 2.2

By the raywenderlich.com Tutorial Team

Jawwad **Ahmad**, Soheil **Azarpour**, Caroline **Begbie**, Evan **Dekhayser**, Aaron **Douglas**, James **Frost**, Vincent **Ngo**, Pietro **Rea**, Derek **Selander**, & Chris **Wagner**

iOS 9 by Tutorials

Updated for Swift 2.2

By the raywenderlich.com Tutorial Team

Jawwad Ahmad, Soheil Azarpour, Caroline Begbie,

Evan Dekhayser, Aaron Douglas, James Frost, Vincent Ngo,

Pietro Rea, Derek Selander, & Chris Wagner

Dedications

"To my parents, my wife, and two daughters,
for their support and encouragement."

— *Jawwad Ahmad*

"To my lovely, always supportive wife Elnaz, our son Kian and my parents."

— *Soheil Azarpour*

"To Ken who always encouraged me not to get a real job. Also to my weird and wonderful children Robin and Kayla - love you both!"

— *Caroline Begbie*

"To my parents, for funding and supporting my expensive hobby."

— *Evan Dekhayser*

"To my husband, Mike, and my parents - all whom have inspired me to do my best and keep plugging away throughout the years."

— *Aaron Douglas*

"To my wonderful, ever-patient wife Hannah, and our son Rupert,
who amazes me every day."

— *James Frost*

"To my parents and sister for always encouraging, and helping me along my journey. I ♥ you guys so much!"

— *Vincent Ngo*

"To my wife Emily and my father Aldo."

— *Pietro Rea*

"Thanks to Brittany & our doggie, Squid, for all the love and support...
as well as constantly licking me when I sleep."

— *Derek Selander*

"To my beautiful wife Sam, our ever curious son Hayden,
and most recent addition to the family Ryland."

— *Chris Wagner*

About the authors

Jawwad Ahmad is a freelance iOS Developer that dove into Swift head first and has not looked back. He enjoys mentoring and teaching and was the original founder of the of the NYC iOS Study Group. He's worked for companies as large as The New York Times, and as small as GateGuru, a 6 person startup.

Soheil Azarpour is an engineer, developer, author, creator, husband and father. He enjoys bicycling, boating and playing piano. He lives in Manchester, NH. Soheil creates iOS apps professionally and independently.

Caroline Begbie is an indie iOS developer and likes to relax with animation software, Arduino and electronics. In her previous life she taught the elderly how to use their computers, performed marionette shows in schools, and ran a software company in Silicon Valley.

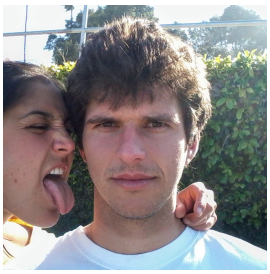
Evan Dekhayser is a high school student, as well as an iOS developer. He first learned Python in 2012, and has since built up his knowledge of Objective C and Swift. He enjoys playing and watching baseball, and is always looking for intriguing topics to learn and potentially write about.

Aaron Douglas was that kid taking apart the mechanical and electrical appliances at five years of age to see how they worked. He took an early interest in computer programming, figuring out how to get past security to play games on his dad's computer. He's still that feisty nerd, but at least now he gets paid to do it. Aaron works for Automattic (WordPress.com, Akismet, SimpleNote) as a Mobile Maker. Twitter: @astralbodies Blog: http://astralbodi.es

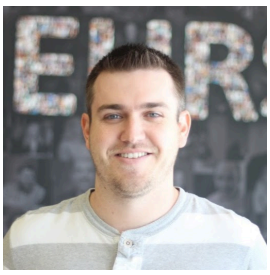
James Frost is a senior iOS developer at Mubaloo Ltd, and lives in Bristol, UK. He taught himself to code in the early 90s on his family's BBC B, and was instantly hooked. He loves learning new things and teaching others, and enjoys spending time with his wife and son, reading, playing games, and cooking. You can find him on his blog at: http://www.jamesfrost.co.uk or as @frosty on Twitter.

Vincent Ngo is a full time iOS software engineer at IBM. He graduated from Virginia Tech with a Computer Science degree. He has passion for developing, sharing, and learning about what's new with the iOS SDK. On the side he loves playing video games, strumming his guitar, hitting golf balls, and chilling with friends.

Pietro Rea is a software engineer at Quidsi, where he builds e-commerce iOS applications for Diapers.com, Wag.com, Soap.com and 7 other brands. Previously, he's worked on the Huffington Post's mobile team. You can find Pietro on Twitter as @pietrorea.



Derek Selander is an iOS developer who enjoys learning through debugging & disassembly to see how others have solved similar problems. In his free time, he enjoys surfing, playing classical guitar, and consuming bacon.



Chris Wagner leads iOS development at Infusionsoft and has been developing for iOS since the release of the SDK in 2009. His background consists of gaming, customer support, systems administration and web development. When he's not working he enjoys spending time with his wife and son. By the time this is published our second son will have arrived!

About the editors

James Frost was a technical editor for this book. He is a senior iOS developer at Mubaloo Ltd, and lives in Bristol, UK. He taught himself to code in the early 90s on his family's BBC B, and was instantly hooked. He loves learning new things and teaching others, and enjoys spending time with his wife and son, reading, playing games, and cooking. You can find him on his blog at: jamesfrost.co.uk or as @frosty on Twitter.

Jeff Rames was a technical editor for this book. He is a developer currently working at AirStrip where he builds enterprise iOS products in the healthcare space. He discovered his passion for mobile software shortly after the iPhone SDK was released, and made it his full time gig in 2011 after a decade in the industry. He spends his free time with his wife and daughters, except when he abandons them for trips to Cape Canaveral to watch rockets being launched into space.

Richard Turton was a technical editor for this book. He is an iOS developer for MartianCraft, prolific Stack Overflow participant and author of a development blog, Command Shift. When he's not in front of a computer he is usually building Lego horse powered spaceships (don't ask!) with his daughter.

Chris Belanger was an editor for this book. He spends his days developing real-time industrial control applications; he fills the rest of his time with writing, editing, travelling, composing music, enjoying the great outdoors and appreciating the finer things in life. He's excited to have worked on yet another book with the raywenderlich.com team and can't imagine life without this crazy, wonderful bunch.

Wendy Lincoln was an editor for this book. By day, she manages complex content development projects and by night she escapes into the world of iOS. Before all this, she produced a cooking show named Hot Kitchen, wrote a cookbook and taught cooking classes. A few years ago she realized her love for writing, editing and playing with computers; she's never looked back. Once in a while, her husband manages to tear her away from the computer for trips to the beach and random home improvement projects.

Sam Davies was the final pass editor for this book. Sam is a strange mashup of developer, writer and trainer. By day you'll find him recording videos for Razeware, writing tutorials, attending conferences and generally being a good guy. By night he's likely to be out entertaining people, armed with his trombone and killer dance moves.

He'd like it very much if you were to say "hi" to him on twitter at @iwantmyrealname.

About the artist

Julien Martin was the artist for the book. After years working in print, Julien discovered an unquenchable passion for icon and interface design while working with startups in New York City. Back home in France, he has since designed multiple successful and award-winning mobile apps for established businesses, entrepreneurs and passionate individuals who value the importance of great design. You can get in touch via http://julien.design.

Table of Contents: Overview

Table of Contents: Extended

Introduction

iOS 9 introduces a whole host of new features, many of them focused on improving the user experience. For example improvements to Siri, the News app and a re-engineered Notes app.

From a more developer-centric point of view, you'll see that there are loads of new APIs and technologies available for use in 3rd-party apps, many of which you get *for free*. For example, a huge new feature on iPad is Multitasking. If you've been using Adaptive Layout (as recommended in *iOS 8 by Tutorials*) you'll find that you have remarkably little work to do to become a multitasking-compliant app!

As part of the improvements to Siri, iOS 9 allows the indexing of the content *inside* your app through Core Spotlight. This not only allows your app to have influence *even when it's not running*, but through public indexing, you can have results from your app displayed on devices that *don't even have the app installed!* This is a great opportunity to extend the reach of your app, potentially gaining happy users and downloads.

As the iPhone lineup continues to advance, new hardware technologies are introduced, and this year is no different. iPhone 6s and iPhone 6s Plus represent the biggest shift in user interaction paradigm since the original iPhone, through 3D Touch. This measures the pressure with which the user is pressing the screen, and in turn, allows them to peek into view controllers. Users will expect this functionality within all apps, so it's really important to get up to speed fast.

The appearance of your app is of great importance to users, and over the past few iOS releases Apple has steadily been making improvements to Auto Layout. iOS 9 is no different, with stack views representing a huge simplification in layout implementation. With full support in Interface Builder, they allow you to achieve the complex designs you desire, without the explicit Auto Layout constraints that have caused the headaches of the past.

iOS is growing up fast — gone are the days when every 3rd-party developer knew everything there is to know about the OS. The sheer size of iOS can make new

releases seem daunting. That's why the Tutorial Team has been working really hard to extract the important parts of the new APIs, and to present this information in an easy-to-understand tutorial format. This means you can focus on what you want to be doing — building amazing apps!

Get ready for your own private tour through the amazing new features of iOS 9. By the time you're done, your iOS knowledge will be completely up-to-date and you'll be able to benefit from the amazing new opportunities in iOS 9.

Sit back, relax and prepare for some high quality tutorials!

What you need

To follow along with the tutorials in this book, you'll need the following:

- **A Mac running OS X Yosemite or later.** You'll need this to be able to install the latest version of Xcode.
- **Xcode 7.3 or later.** Xcode is the main development tool for iOS. You'll need Xcode 7.3 or later for all tasks in this book. You can download the latest version of Xcode for free on the Mac app store here: apple.co/1FLn51R
- **One or more devices (iPhone, iPad, or iPod Touch) running iOS 9 or later.** Most of the chapters in the book let you run your code on the iOS 9 Simulator that comes with Xcode. However, a few chapters later in the book require one or more physical iOS devices for testing.

Once you have these items in place, you'll be able to follow along with every chapter in this book.

Who this book is for

This book is for intermediate or advanced iOS developers who already know the basics of iOS and Swift development but want to upgrade their iOS 9 skills.

- **If you are a complete beginner to iOS development**, we recommend you read through *The iOS Apprentice, 4th Edition* first. Otherwise this book may be a bit too advanced for you.
- **If you are a beginner to Swift**, we recommend you read through either *The iOS Apprentice, 4th Edition* (if you are a complete beginner to programming), or *The Swift Apprentice* (if you already have some programming experience) first.

If you need one of these prerequisite books, you can find them on our store here:

- www.raywenderlich.com/store

As with raywenderlich.com, all the tutorials in this book are in Swift.

How to use this book

This book can be read from cover to cover, but we don't recommend using it this way unless you have a lot of time and are the type of person who just "needs to know everything". (It's okay; a lot of our tutorial team is like that, too!)

Instead, we suggest a pragmatic approach — pick and choose the chapters that interest you the most, or the chapters you need immediately for your current projects. Most chapters are self-contained, so you can go through the book in a non-sequential order.

Looking for some recommendations of important chapters to start with? Here's our suggested Core Reading List:

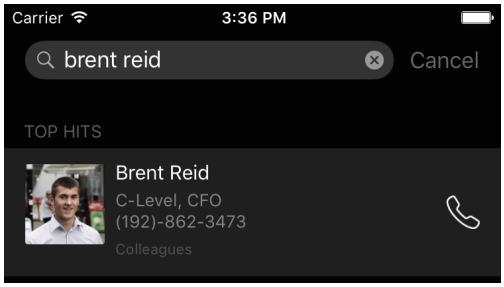
- Chapter 1, "Swift 2"
- Chapter 2, "Introducing App Search"
- Chapter 4, "App Thinning"
- Chapter 6, "3D Touch"
- Chapter 7, "UIStackView & Auto Layout changes"

That covers the "Big 5" topics of iOS 9; from there you can dig into other topics of particular interest to you.

Book overview

iOS 9 has a selection of new technologies and APIs that cover the entire OS. Here's what you'll learn about in this book:

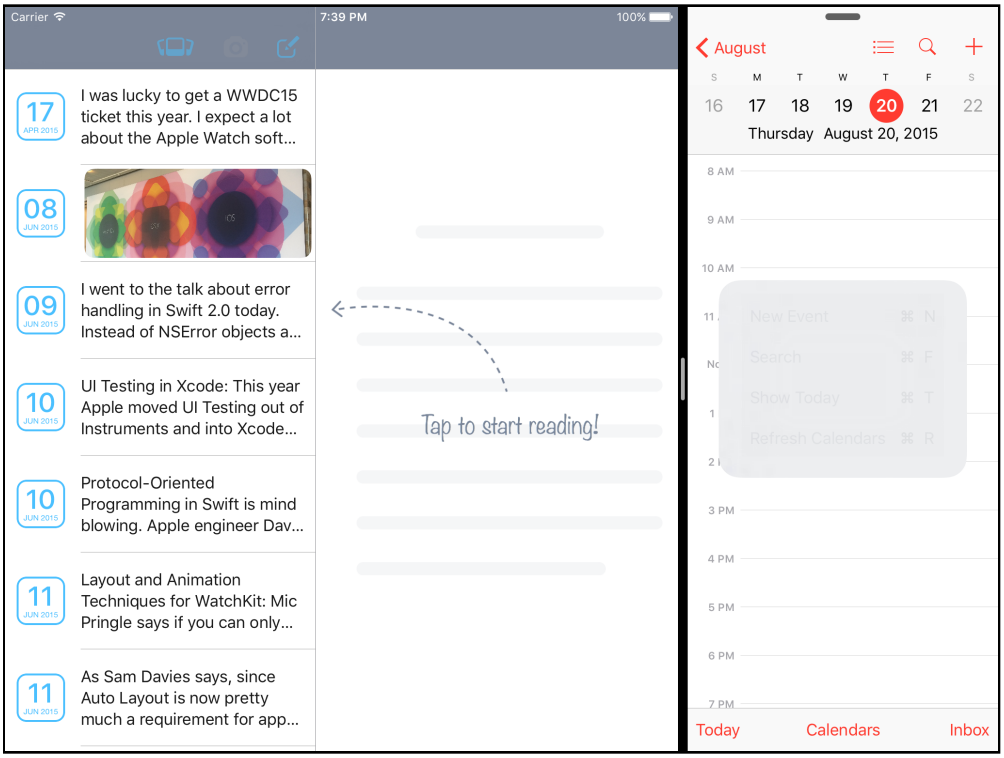
1. **Chapter 1, Swift 2:** Swift is only a year old, but a lot has changed since it was born. Discover the new error handling model, protocol extensions, new control flow features and much more! You'll be ready to tackle the rest of the book, and to upgrade your existing code to Swift 2!
2. **Chapter 2, Introducing App Search:** With iOS 9, users can search inside your apps using the system Spotlight utility. Learn how to adopt this new functionality, and make the content inside your apps more discoverable.



3. **Chapter 3, Your App on the Web:** Deep linking will allow you to direct users browsing your website directly to the correct point inside your iOS app. Find out how to integrate this functionality in your own site and app as you follow along with updating the RWDevCon website and app!

4. **Chapter 4, App Thinning:** App Thinning describes a collection of new App Store technologies that ensure that users downloading your app only download exactly what their specific device requires. Discover what you need to do to adopt App Thinning, together with how you can split your app's resources up into chunks that are only downloaded from the App Store when they are required.

5. **Chapter 5, Multitasking:** Completely new to iOS 9 is the ability to run two apps side-by-side on iPads. Get up to speed on what you need to do to ensure your apps are ready for multitasking.

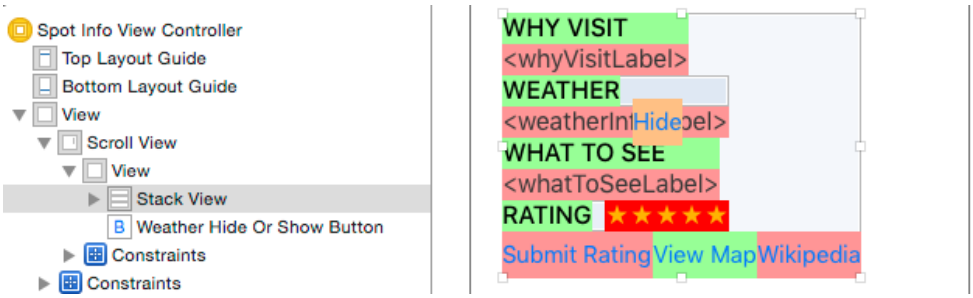


6. **Chapter 6, 3D Touch:** The iPhone 6s and iPhone 6s Plus introduced a new technology called 3D Touch. Discover how to detect the force of a touch in iOS

9, and the new interaction paradigms—such as pop and peek—that come along with it.

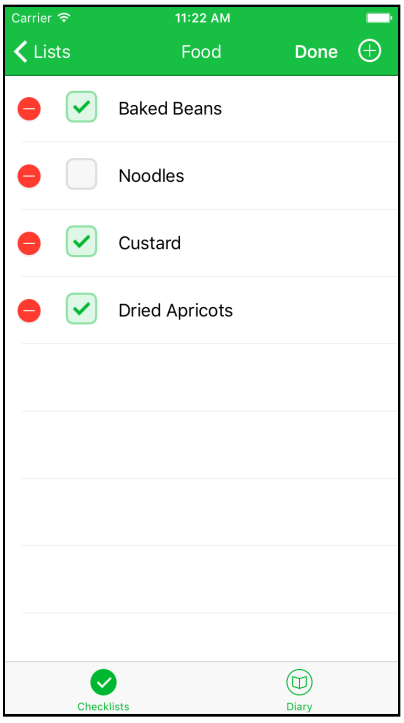
7. **Chapter 7, UIStackView & Auto Layout changes:** Stack views will change the way you build your interfaces. They remove a lot of the boilerplate Auto Layout and make it really easy to construct simple layouts. Learn the basics of stack views and many of the other new Auto Layout features.

8. **Chapter 8, Intermediate UIStackView:** Dive deeper into stack views — covering nesting, animation and working with them in code.



9. **Chapter 9, What's New in Storyboards?:** Ever faced the problem of unmanageably large storyboards? Discover how to refactor your huge storyboard into smaller segments with new storyboard references.

10. **Chapter 10, Custom Segues:** Creating custom view controller transitions has never been an easy task, but iOS 9 chips away at some of the complexity. Learn how to create your own custom segues to make custom transitions between view controllers far easier to understand.



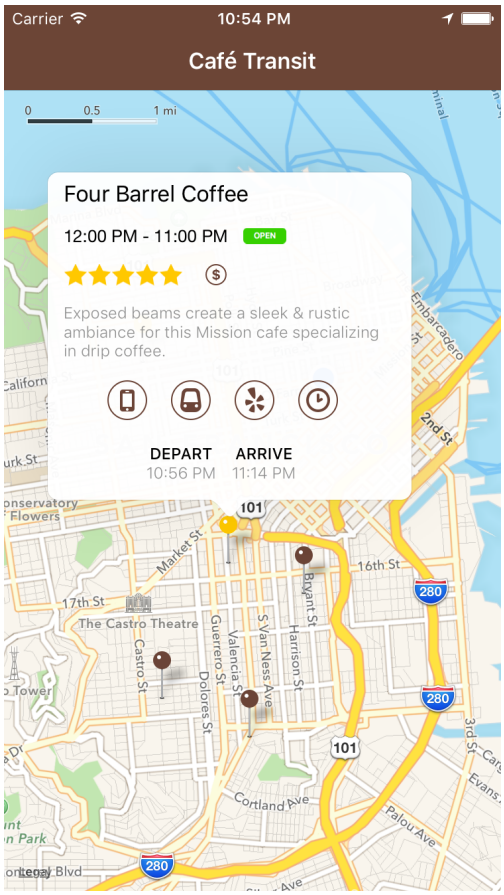
11. **Chapter 11, UIKit Dynamics:** iOS 9 includes loads of great improvements to UIKit Dynamics — making it really easy to model complex physics animations within your apps. Learn all about the new behaviors and debugging improvements as you build a real-world example.

12. **Chapter 12, Contacts:** Accessing the on-device contacts has been challenging in the past — with a C-level API. iOS 9 introduces two new frameworks to ease interaction with the contacts. Discover how to integrate with the device address book in a much simpler manner.

13. **Chapter 13, Testing:** Xcode 7 now includes a fully-featured solution for creating UI tests for your application. Discover how to add UI tests to your own apps, and review some of the other improvements to testing.

14. **Chapter 14, Location & Mapping:** It's finally possible to choose your favorite color for a map pin! Learn about this, transit directions, Core Location enhancements and more!

15. **Chapter 15, What's New in Xcode?:** There's loads of cool new stuff in iOS 9, but don't forget about your friendly IDE! Learn how the improved gauges make optimizing your app really easy, along with many other new features.



Book source code and forums

This book comes with the Swift source code for each chapter – it's shipped with the PDF. Some of the chapters have starter projects or other required resources, so you'll definitely want them close at hand as you go through the book.

We've also set up an official forum for the book at raywenderlich.com/forums. This is a great place to ask questions about the book, discuss making apps with iOS 9 in general, share challenge solutions, or to submit any errors you may find.

Book Updates

Great news: since you purchased the PDF version of this book, you'll receive free updates of the content in this book!

The best way to receive update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials published on raywenderlich.com that month, important news items such as book updates or new books, and a list of our favorite developer links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

License

By purchasing *iOS 9 by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *iOS 9 by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images, or designs that are included in *iOS 9 by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from the *iOS 9 by Tutorials* book, available at www.raywenderlich.com".
- The source code included in *iOS 9 by Tutorials* is for your own personal use only. You are NOT allowed to distribute or sell the source code in *iOS 9 by Tutorials* without prior authorization.
- This book is for your own personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, co-workers, or students; they must to purchase their own copy instead.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and non-infringement. In no event shall the authors or copyright holders be liable for any claim, damages or

other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the property of their respective owners.

Acknowledgments

We would like to thank many people for their assistance in making this possible:

- **Our families:** For bearing with us in this crazy time as we worked all hours of the night to get this book ready for publication!
- **Everyone at Apple:** For developing an amazing operating system and set of APIs, for constantly inspiring us to improve our apps and skills, and for making it possible for many developers to have their dream jobs!
- **And most importantly, the readers of raywenderlich.com — especially you!** Thank you so much for reading our site and purchasing this book. Your continued readership and support is what makes all of this possible!

Chapter 1: Swift 2

By Chris Wagner

The 2014 WWDC keynote was nearly over, and Craig Federighi appeared to be wrapping things up. But instead he shocked nearly everyone watching by announcing the Swift programming language, which he promised, perhaps optimistically, as "Objective-C without the baggage of C". Swift would have the benefits of terseness and safety, while still being extremely expressive.

Once the implications of a new programming language had sunk in, many developers set out to explore the ins and outs of Swift. The year following WWDC 2014 was an exciting time for developers on the Apple platform; each developer had a chance to reimagine and redefine they way they wrote software for iOS and OS X.

Whither Swift?

Why would Apple introduce a new language, since Objective-C has served them so well? It's likely because Swift gave Apple a fresh start; Swift takes the best features from many different languages and combines them into one. Apple has created a language that is modern, expressive, safe, and a lot of fun to develop in. Swift also interoperates seamlessly with existing Cocoa and Cocoa Touch frameworks as well as all of your existing Objective-C code. This is likely one of the primary reasons Swift has seen such success and widespread adoption in the developer community.

The Real "One more thing"

WWDC 2015's big announcement is the open-sourcing of Swift by the end of 2015! But what does this really mean?

- Swift source code will be released under an OSI-approved permissive license.
- Contributions from the community will be accepted — and encouraged.

- At launch, Apple intends to contribute ports of Swift for OS X, iOS, and Linux.
- Source code will include the Swift compiler and the standard library.

You might be thinking "This is amazing! I can write Android apps using Swift! Write once, debug everywhere!!" :] Well, back that excitement train up a bit. While open-sourcing Swift *is* great news, it's highly unlikely Apple has any intention of open-sourcing the Cocoa or Cocoa Touch frameworks you love so much. These frameworks make it "easy" to write Mac and iOS apps; consider how you'd write an Objective-C program without NS*Anything* or UI*Anything*. While Swift does offer a lot out of the box, you unfortunately won't have access to those frameworks on other platforms.

But don't let that get you down! The open source community does amazing things every day. Open-source Swift will attract smart, creative people who will make the language even better with Swift-only libraries and frameworks that work across platforms. Someday you could find yourself using Swift in embedded environments such as Arduino, or perhaps someday you'll write server-side web services in Swift. It's an exiting time for Swift developers!

What's New in Swift 2?

It's great to dream about the future of Swift, but this chapter highlights what Swift 2 offers you in the present day:

- New control flow using `guard`, `repeat`, `do`, and `defer`
- An entirely new error handling model
- Protocol extensions
- Pattern matching enhancements
- API availability checking
- Additional minor enhancements

This chapter is packed with information; you can read it end to end or use it as as a reference as you work with Swift 2.0.

What About Swift 2.2?

Since the first version of this chapter, there's been some big news: Apple has open sourced Swift!

On December 3, 2015 Apple launced swift.org: a website focused on the open source project and building a community around it. A few months later, Apple

released the first community driven version of Swift in Xcode 7.3: Swift 2.2. The process went extremely smoothly, and Apple has truly raised the standard with regards to open source projects and shocked the development community with their efforts. Take some the time to checkout the site, you may find yourself interested in contributing!

Let's take a look at what's new in Swift 2.2:

- **Compile-time Swift version checks**: Much like the OS Availability checks described in this chapter, this allows you to check the Swift version itself at compile-time and branch accordingly.
- **Compile-time checked selectors**: Gone are the days of stringly typed method names, there is a new `#selector` syntax that allows the compiler to validate the method you're referencing actually exists.
- **Allow more keywords as argument labels**: Now you have more freedom to use most of the language keywords as your method argument labels, e.g. `func su(do: () -> (), if: () -> Bool, for: Int) -> Result`
- **Tuple comparison**: The Swift standard library now defines comparison operators for tuples ranging from 1...6 elements. You can simply evaluate an expression like `("Foo", true, 1) == ("Bar", true, 1)` which returns `false`. The comparison operator is applied to each element respectively and the result of each comparison is &&'d, which means if any return `false` the entire comparison resolves to `false`.
- **C-style `for` loops deprecated**: The community agrees, Swift is looking toward the future. There's no need for the classic `for var i = 0; i < 5; i++ { }` syntax and therefore it has been deprecated. You will need to rely on the more Swifty variant using ranges, e.g. `for i in 0 ..< 5 { }` or fast enumeration e.g. `for cat in cats { cat.meow() }`.
- **++ and -- are deprecated**: Naturally, the C-style `for` loop deprecation lead to this as well. ++ and -- come with a lot of baggage with regards to postfix and prefix evaluation. It was decided that `foo += 1` is much easier to reason about and really isn't much more to write than `foo++`.

Beware that the deprecated items mentioned are slated to be completely removed for Swift 3.0, so get cracking at updating your code! This book has been completed updated to comply with the Swift 2.2 changes. There are other changes that you can read about on the Swift 2.2 announcement post. If you want to keep abreast with upcoming changes to Swift you may be interested in the various mailing lists Apple has provided.

The Logistics

Unlike most chapters in this book, you *won't* write or extend an app in this chapter.

Instead, you'll work in a multipage Xcode Playground with the Swift language features as the focus. The first part of the chapter will introduce you to some new features using somewhat contrived examples; the second half walks you thorough the solution of a specific String validation problem using Swift 2 features in a tutorial-led playground.

Open the provided **Chapter1_Swift2.playground** file in Xcode 7 and you'll be ready to dive right into the chapter!

> **Note:** When running Playgrounds in Xcode 7 GM you may often see error messages like the following in the Debug Area.
>
> `CGContextSaveGState: invalid context 0x0. If you want to see the backtrace, please set CG_CONTEXT_SHOW_BACKTRACE environmental variable.`
>
> According to Apple engineers in the developer forums (apple.co/1FbVE0l) it is safe to ignore these messages.

Control Flow

Control flow is a fundamental concept in any programming language. Not sure what "control flow" means? A basic example is `if/else`; any construct or keyword that causes the execution of your program to follow a different path can be considered "control flow". Swift 2 adds new control flow features and makes some minor changes to existing ones. The two new control flow featues covered in this section are found on your playground's **Control Flow** page.

repeat

The `do/while` control flow feature has been renamed to `repeat/while`. It operates the same way as before; the name has simply changed to reduce confusion with a new usage of `do` described later on in this chapter. This flow simply means "`repeat` this block of code `while` some condition remains `true`". Consider the following example:

```
var jamJarBeer = Beer()

repeat {
  jamJarBeer.sip()
} while (!jamJarBeer.isEmpty) // always finish your beer!
```

The above snippet repeats the line `jamJarBeer.sip()` until `jamJarBeer.isEmpty` is `true` — a common occurrence after-hours at RWDevCon 2015! :]

guard

Pre-condition checks are frequently required for proper state management, or to ensure that valid arguments were passed. The new `guard` control flow is the perfect

tool for doing these checks. Consider the `Beer().sip()` method above. What happens when you sip an empty beer? What does that even mean? (It probably means you've had too many *or* need a refill. :]) Perhaps it makes sense to verify that there is beer available for sipping, like so:

```
struct Beer {
  var percentRemaining = 100
  var isEmpty: Bool { return percentRemaining <= 0 }
  var owner: Patron?

  mutating func sip() {
    guard percentRemaining > 0 else {             // 1
      print("Your beer is empty, order another!")
      return
    }

    percentRemaining -= 10
    print("Mmmm \(percentRemaining)% left")
  }
}
```

The `//1` comment indicates the `guard` that verifies the amount of beer left is greater than 0 and if not, executes the code in the trailing `else` block, which instructs you to order another brew! You then return immediately so you don't end up in some weird state with a negative amount of beer. Beer debt is not a good thing.

The `guard` control is defined as `guard (condition) else { // code to execute if condition is false }`. You *could* use an `if` statement instead, but the logic is not quite as straightforward:

```
if beer.isEmpty {
  print("Your beer is empty, order another!")
  return
}
```

The snippet above flips the logic and checks that the beer is empty, instead of checking that the beer *isn't* empty. But isn't this just personal coding preference?

Not exactly; it comes down to *expressiveness*, which is a primary goal of Swift. Functionally, they're equivalent, but `guard` clearly states to anyone reading your code that you're performing a pre-condition check. Using a plain old `if` statement does not deliberately convey that information.

Now, before you write-off `guard` as *just* an enhancement in expressiveness, take a look at its true power: working with Optionals.

Consider the world of bartending: success is measured in the number of beers you sell. As a beer nears empty, the bartender tries to locate the owner so they can offer another. A harsh reality though, is that beer owners may abandon a half-drunk beer (known as a wounded soldier), hence the reason the `owner` property is Optional on `Beer`.

Here's a good model for a successful bartender:

```
struct Bartender {
  func offerAnotherToOwnerOfBeer(beer: Beer) {
    guard let owner = beer.owner else {
      print("Egads, another wounded soldier to attend to.")
      return
    }

    print("\(owner.name), would you care for another?") // 1
  }
}
```

Note that you access `owner` as a non-optional value in the last line above. Since you used `guard` in `guard let owner = beer.owner`, the unwrapped optional value becomes available in the same scope that `guard` was called. If the value of `beer.owner` were `nil` then the `else` block would execute.

This feature lets you perform your optional binding upfront in a very explicit manner. After writing the `guard` conditions, you can continue your implementation as you would in an `if let` block, but without the extra indention.

Error handling

The early days of Swift led to many questions on Stack Overflow about error handling, and in particular, *exceptions* as found in other popular languages. Apple instead opted to stick to the `NSError` approach Cocoa has always used and release Swift with the promise of advanced error handling in the next version.

Swift 2 has a first-class error handling model; you can declare that a method or function `throws` an error. This lets the caller/consumer know an error may occur. The compiler also requires that you either write the code to handle the error, or to explicitly ignore it.

All of the code for this section is included on the **Errors** page of the chapter's Xcode Playground.

Consider the following protocol:

```
protocol JSONParsable {
  static func parse(json: [String: AnyObject]) throws -> Self
}
```

This protocol defines a single static method that takes a JSON dictionary and returns an instance of `Self`, where `Self` is the type that conforms to this protocol. The method also declares that it can throw an error.

So, what exactly *is* a Swift error? Is it an `NSError`? No...and yes. :] A pure Swift error is represented as an `enum` that conforms to the protocol `ErrorType`. However, Apple Engineers conveniently made `NSError` conform to the `ErrorType` protocol,

which means this pattern works quite well between Swift and Objective-C. If you're interested to learn more about interoperability, the Swift and Objective-C Interoperability (apple.co/1He5uhh) session is a must-see!

You can create your own error type as below:

```
enum ParseError: ErrorType {
  case MissingAttribute(message: String)
}
```

Pretty easy, right? This error has a single case and includes an associative value of the type `String` as a message. When you throw this error type you can include extra information describing the issue. Since enums are used when creating `ErrorTypes` you can include any kind of associative values that you deem necessary for your use case.

Take a look at the following `struct` that implements `JSONParsable` and throws some errors:

```
struct Person: JSONParsable {
  let firstName: String
  let lastName: String

  static func parse(json: [String : AnyObject]) throws
    -> Person {

    guard let firstName = json["first_name"] as? String else {
      let message = "Expected first_name String"
      throw ParseError.MissingAttribute(message: message) // 1
    }

    guard let lastName = json["last_name"] as? String else {
      let message = "Expected last_name String"
      throw ParseError.MissingAttribute(message: message) // 2
    }

    return Person(firstName: firstName, lastName: lastName)
  }
}
```

The errors are thrown in the commented lines `//1` and `//2`. If either `guard` statement fails to validate, the method throws an error and returns immediately. The expressiveness of the `guard` statement makes it very clear what you're asserting at each stage.

When calling a method that `throws`, the compiler requires that you preface the call to that method with **`try`**. In order to capture any thrown errors, you must wrap your "trying" call in a `do {}` block followed by `catch {}` blocks. You can choose to catch specific types of errors and respond appropriately to each error type, or simply provide a "catch-all" if you don't know what errors you might receive.

> **Note**: At the time of this writing, Apple hasn't provided a way to infer the exact types of errors that can be thrown from a method or function. There is also no way for an API writer to declare what types of errors will be thrown. Therefore, it's good practice to include this information in your method's documentation.

Here's the do/try/catch block that checks for the error type you added above:

```
do {
  let person = try Person.parse(["foo": "bar"])
} catch ParseError.MissingAttribute(let message) {
  print(message)
} catch {
  print("Unexpected ErrorType")
}
```

When you can guarantee a throwing call will *never* fail, or that catching a thrown error doesn't provide any benefit, such as a critical failure point where the app can't continue to operate, it is possible to bypass the `do/catch` requirement: simply type an `!` after `try`.

Find and uncomment the following line in the playground:

```
let p1 = try! Person.parse(["foo": "bar"])
```

You'll notice a runtime error appears. Note, however, that the following works just fine without producing an error:

```
let p2 = try! Person.parse(["first_name": "Ray",
  "last_name": "Wenderlich"])
```

Now that you understand the basics of handling Swift 2 errors, you can focus on a specific problem to solve, rather than work with contrived examples.

The Project

In this section, you'll work at solving a String validation problem using some of the features discussed above along with some additional Swift 2 features. You're trying to write string validators that validate whether the input string conforms to any number of rules. You'll use this validator to create a password complexity checker.

String Validation Error

Switch to the next page in the chapter's playground, **String Validation**. Now that you're familiar with defining custom `ErrorTypes`, it's time to make use of robust error types for potential validation errors.

Take a look at the `ErrorType` defined at the top of the playground's "String Validation" page, right below `import UIKit`. This `ErrorType` has a number of cases with varying associative values that help describe the error.

After the spot where you define the cases, you'll see a computed variable `description`. This provides conformance to the `CustomStringConvertible` protocol and lets you display the error in a human-readable format.

Now that the error type is defined, it's time to start throwing them around! :] You'll first start with a protocol that defines a rule. You are going to be using protocol oriented programming patterns to make this solution robust and extendable.

Add the following protocol definition to the playground:

```
protocol StringValidationRule {
  func validate(string: String) throws -> Bool
  var errorType: StringValidationError { get }
}
```

This protocol requires two things. The first being a method that returns a `Bool` denoting the validity of a given string and also throws an error. The second is a property which describes the type of error that may be thrown by the `validate(string:)` method.

> **Note:** The `errorType` property is not a Swift requirement. It's here so that you can be clear about the types of error that might be returned.

To use multiple rules together, define a `StringValidator` protocol like so:

```
protocol StringValidator {
  var validationRules: [StringValidationRule] { get }
  func validate(string: String) -> (valid: Bool,
    errors: [StringValidationError])
}
```

This protocol requires an array of `StringValidationRules` as well as a function that validates a given string and returns a tuple. The first value of the tuple is a `Bool` that designates whether the string is valid; the second is an array of `StringValidationErrors`. In this case you're not using `throws`; instead, you're returning an array of error types since each rule can throw its own error. When it comes to string validation, it's best to let the user know of every rule they've broken so that they can resolve all of them in a single pass.

Think how you might implement a `StringValidator`'s `validate(string:)` method. You'd likely iterate over each item in `validationRules`, collect any errors, and determine the status based on whether any errors occurred. This logic will likely be the same for any `StringValidator` you need.

Surely you don't want to copy and paste that implementation into ALL of your

`StringValidators`! The good news is that Swift 2 introduces **Protocol Extensions** that let you define default implementations for all types that specify conformance to a protocol.

Next, you'll extend the `StringValidator` protocol to define a default implementation for `validate(string:)`. Add the following code to the playground:

```
extension StringValidator {                                  // 1
  func validate(string: String) -> (valid: Bool,
    errors: [StringValidationError]) {                       // 2

    var errors = [StringValidationError]()                   // 3
    for rule in validationRules {                            // 4
      do {                                                   // 5
        try rule.validate(string)                            // 6
      } catch let error as StringValidationError {           // 7
        errors.append(error)                                 // 8
      } catch let error {                                    // 9
        fatalError("Unexpected error type: \(error)")
      }
    }

    return (valid: errors.isEmpty, errors: errors)           // 10
  }
}
```

Here's what you do in each commented section above:

1. Create an extension for `StringValidator`.
2. Define the default implementation for `func validate(string: String) -> (valid: Bool, errors: [StringValidationError])`.
3. Create a mutable array to hold any errors that might be thrown.
4. Iterate over each of the the validator's rules.
5. Specify a `do` block since you'll catch any thrown errors.
6. Execute `validate(string:)` for each rule; note that you must precede the call with `try` as this method is throwable.
7. Catch any errors of the type `StringValidationError`.
8. Capture the error in your `errors` array.
9. If any error other than `StringValidationError` is thrown, crash with a message including which error occurred. Just as in a `switch` statement, your error handling must be exhaustive, or you'll get a compiler error.
10. Return the resultant tuple. If there are no errors validation passed, return the array of errors even if it's empty.

Now each and every `StringValidator` you implement will have this method by

default so you can avoid "copy and paste" coding. :]

Time to implement your very first `StringValidationRule`, starting with the first error type `.MustStartWith`. Add the following code to your playground:

```
struct StartsWithCharacterStringValidationRule
  : StringValidationRule {

  let characterSet: NSCharacterSet            // 1
  let description: String                     // 2
  var errorType: StringValidationError {      // 3
    return .MustStartWith(set: characterSet,
      description: description)
  }

  func validate(string: String) throws -> Bool {
    if string.startsWithCharacterFromSet(characterSet) {
      return true
    } else {
      throw errorType                         // 4
    }
  }
}
```

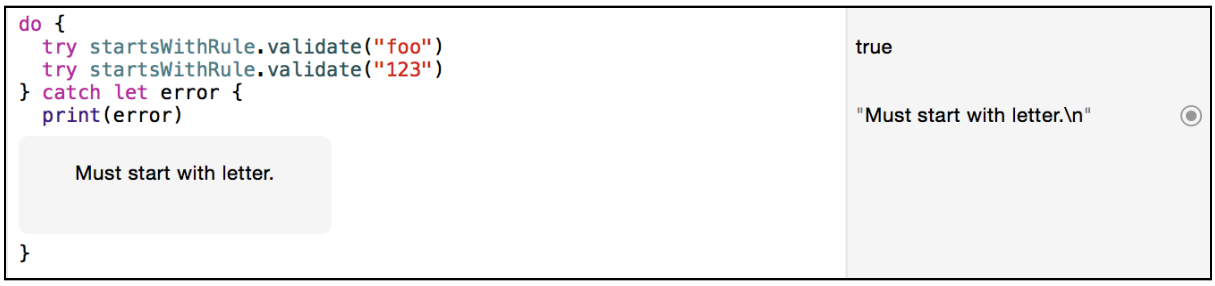
Breaking this method down, here's what you find:

1. This defines the allowed character set for the start of the string.
2. This is a description of the character set; if you used a set of numbers you might define this as "number".
3. This is the type of error this rule can throw.
4. Finally, if the validation fails you throw an error.

Time to take this new rule for a spin! Add the following code to the playground:

```
let letterSet = NSCharacterSet.letterCharacterSet()
let startsWithRule = StartsWithCharacterStringValidationRule(
  characterSet: letterSet,
  description: "letter")

do {
  try startsWithRule.validate("foo")
  try startsWithRule.validate("123")
} catch let error {
  print(error)
}
```

You should see the following output in your playground.



Note: You can display the result inline with your code by pressing the **Show Result** circle button to the right of the output in the playground's timeline.

Great work! You've written your first validation rule; now you can create one for "Must End With". Add the following to the playground:

```
struct EndsWithCharacterStringValidationRule
  : StringValidationRule {

  let characterSet: NSCharacterSet
  let description: String
  var errorType: StringValidationError {
    return .MustEndWith(set: characterSet,
      description: description)
  }

  func validate(string: String) throws -> Bool {
    if string.endsWithCharacterFromSet(characterSet) {
      return true
    } else {
      throw errorType
    }
  }
}
```

The above logic is quite similar to the first validator; it simply checks the ending character against your supplied character set.

Now that you have two different rules, you can create your first `StringValidator`. Create a validator that verifies a string both starts and ends with characters belonging to specific character sets by adding the following code:

```
struct StartsAndEndsWithStringValidator: StringValidator {
  let startsWithSet: NSCharacterSet             // 1
  let startsWithDescription: String
  let endsWithSet: NSCharacterSet               // 2
  let endsWithDescription: String

  var validationRules: [StringValidationRule] { // 3
    return [
      StartsWithCharacterStringValidationRule(
        characterSet: startsWithSet,
        description: startsWithDescription),
```

```
        EndsWithCharacterStringValidationRule(
          characterSet: endsWithSet ,
          description: endsWithDescription)
      ]
    }
}
```

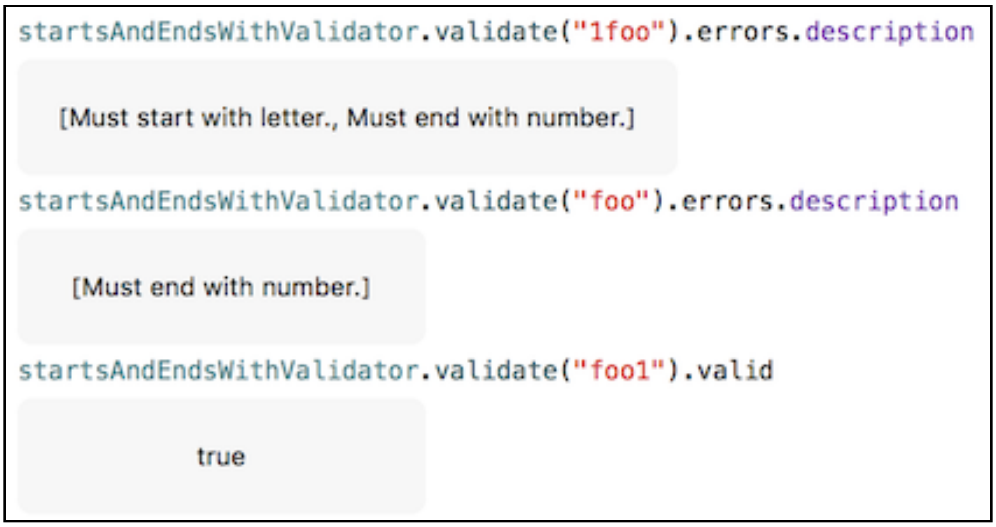
Since you wrote a protocol extension for `StringValidator` that provides a default implementation of `func validate(string: String) -> (valid: Bool, errors: [StringValidationError])` this implementation becomes quite straightforward:

1. This is the character set for the "starts with" rule.
2. This is the character set for the "ends with" rule.
3. Create an array with both rules for `validationRules` required by the `StringValidator` protocol.

Now give your new validator a try! Add the following to the playground:

```
let numberSet = NSCharacterSet.decimalDigitCharacterSet()

let startsAndEndsWithValidator =
  StartsAndEndsWithStringValidator(
    startsWithSet: letterSet,
    startsWithDescription: "letter",
    endsWithSet: numberSet,
    endsWithDescription: "number")

startsAndEndsWithValidator.validate("1foo").errors.description
startsAndEndsWithValidator.validate("foo").errors.description
startsAndEndsWithValidator.validate("foo1").valid
```

You should see the following result:



Password Requirement Validation

It's time to put your StringValidator pattern to work. You're the software engineer tasked with creating the sign-up form for your company's app. The design specifies that passwords must meet the following requirements:

- Must be at least 8 characters long
- Must contain at least 1 uppercase letter
- Must contain at least 1 lowercase letter
- Must contain at least 1 number
- Must contain at least 1 of the following "!@#$%^&*()_-+<>?/\[]{}"

If you hadn't worked through the protocol-oriented solution above, you might have looked at this list of requirements and groaned a little. But instead you can take the pattern you've built and quickly create a `StringValidator` that contains the rules for this password requirement.

Start by switching to the **Password Validation** page in the chapter's playground. For brevity, this playground page tucks away all of the previous work as a source file, which also contains two new rules you'll use. Click **Password Validation** > **Sources** > **StringValidation.swift** in the jump bar to view the code in that file.

LengthStringValidationRule

The first new rule is `LengthStringValidationRule` that has the following features:

- Validates that a string is a specified length, with the following two types:
- `Min(length: Int)`: must be at least `length` long
- `Max(length: Int)`: cannot exceed `length`

Both types can be combined in a `StringValidator` to ensure the String is between a specific range in length. Here's the rule implementation:

```
public struct LengthStringValidationRule
  : StringValidationRule {

  public enum Type {
    case Min(length: Int)
    case Max(length: Int)
  }
  public let type: Type
  public var errorType: StringValidationError { get }
  public init(type: Type)
  public func validate(string: String) throws -> Bool
}
```

ContainsCharacterStringValidationRule

The second rule is `ContainsCharacterStringValidationRule`, with the following requirements:

- Validates that a string contains specific character(s) with the following types:
- `MustContain`: the string must contain a character in the provided set.
- `CannotContain`: the string cannot contain a character in the provided set.
- `OnlyContain`: the string can only contain characters in the provided set.
- `ContainAtLeast(count: Int)`: the string must contain at least `count` characters in the provided set.

Here's the implementation:

```
public struct ContainsCharacterStringValidationRule
  : StringValidationRule {

  public enum Type {
    case MustContain
    case CannotContain
    case OnlyContain
    case ContainAtLeast(Int)
  }
  public let characterSet: NSCharacterSet
  public let description: String
  public let type: Type
  public var errorType: StringValidationError { get }
  public init(characterSet: NSCharacterSet,
    description: String,
    type: Type)
  public func validate(string: String) throws -> Bool
}
```

With these two new rules in your back pocket you can quickly implement the password requirement validator. Add the following to the **Password Validation** page in the playground:

```
struct PasswordRequirementStringValidator: StringValidator {

  var validationRules: [StringValidationRule] {
    let upper = NSCharacterSet.uppercaseLetterCharacterSet()
    let lower = NSCharacterSet.lowercaseLetterCharacterSet()
    let number = NSCharacterSet.decimalDigitCharacterSet()
    let special = NSCharacterSet(
      charactersInString: "!@#$%^&*()_-+<>?/\\[]}{")

    return [
      LengthStringValidationRule(type: .Min(length: 8)),
      ContainsCharacterStringValidationRule(
        characterSet:upper ,
        description: "upper case letter",
```

```
        type: .ContainAtLeast(1)),
      ContainsCharacterStringValidationRule(
        characterSet: lower,
        description: "lower case letter",
        type: .ContainAtLeast(1)),
      ContainsCharacterStringValidationRule(
        characterSet:number ,
        description: "number",
        type: .ContainAtLeast(1)),
      ContainsCharacterStringValidationRule(
        characterSet:special,
        description: "special character",
        type: .ContainAtLeast(1))
    ]
  }
}
```

You'll recognize this code as being very similar to the concrete string validator you created. It simply provides an implementation for the `validationRules` computed property, which returns an array of the four rules you need to satisfy your requirements, in a remarkably readable configuration.

Now, try it out! Add the following to the playground:

```
let passwordValidator = PasswordRequirementStringValidator()
passwordValidator.validate("abc1").errors
passwordValidator.validate("abc1!Fjk").errors
```

You should see the following result:

```
let passwordValidator = PasswordRequirementsStringValidator()
passwordValidator.validate("abc1").errors.description
```

[Must be at least 8 characters., Must contain at least 1 upper case letter., Must contain at least 1 special character.]

Great work - you've used protocol oriented programming with Swift 2 features to implement a solution to a real-world non-trivial problem.

Additional Things

The previous sections covered a number of new features in Swift 2, but wait - there's more! (Is it just me, or is this starting to sound like an infomercial? :])

The remainder of this chapter has you experimenting with some of the previously mentioned features and introduces you to some new features. The examples won't be as concrete as the string validation problem, but hopefully still interesting nonetheless!

Going further with Extensions

One other amazing thing about Extensions is that you can provide functionality with generic type parameters; this means that you can provide a method on arrays that contain a specific type. You can even do this with protocol extensions.

For example, say that you wanted to create a method that shuffles an array of names to determine the order of players in a game. Seems easy enough, right? You simply take an array of names, mix it up and return it. Done and done. But what if you later discover a need to shuffle an array of cards for the same game? Now you have to either reproduce that shuffle logic for an array of cards, or create some kind of generic method that can shuffle both cards and names. There's got to be a better way, right?

How about creating an extension on the `MutableCollectionType` protocol? Conformers of the protocol must have an Index since you need to use ordered collections to retain the sort order.

Add the following into the **Additional Things** page in the chapter's playground:

```
extension MutableCollectionType where Index == Int {
  mutating func shuffleInPlace() {
    let c = self.count
    for i in 0..<(c-1) {
      let j = Int(arc4random_uniform(UInt32(c - i))) + i
      guard i != j else { continue }
      swap(&self[i], &self[j])
    }
  }
}
```

Next, you need to create an array of people and invoke your new method. Add the following code:

```
var people = ["Chris", "Ray", "Sam", "Jake", "Charlie"]
people.shuffleInPlace()
```

You should see that the `people` array has been shuffled like so:

If your results aren't shuffled, verify that you typed the algorithm correctly or go buy a lottery ticket because it shuffled them into the same order that was received! You can reshuffle to see different results by pointing to **Editor/Execute Playground**.

> **Note:** Extending functionality to generic type parameters is only available to classes and protocols. You will need to create an intermediate protocol to achieve the same with structs.

Using defer

With the introduction of `guard` and `throws`, exiting scope early is now a "first-class" scenario in Swift 2. This means that you have to be careful to execute any necessary routines prior to an early exit from occurring. Thankfully Apple has provided `defer { ... }` to ensure that a block of code will always execute before the current scope is exited.

Review the following and notice the `defer` block defined at the beginning of the `dispenseFunds(amount:account:)` method.

```
struct ATM {
  var log = ""

  mutating func dispenseFunds(amount: Float,
    inout account: Account) throws {

    defer {
      log += "Card for \(account.name) has been returned " +
        "to customer.\n"
      ejectCard()
    }

    log += "====================\n"
    log += "Attempted to dispense \(amount) from " +
      "\(account.name)\n"

    guard account.locked == false else {
      log += "Account Locked\n"
      throw ATMError.AccountLocked
    }

    guard account.balance >= amount else {
      log += "Insufficient Funds\n"
      throw ATMError.InsufficientFunds
    }

    account.balance -= amount
    log += "Dispensed \(amount) from \(account.name)."
    log += " Remaining balance: \(account.balance)\n"
  }

  func ejectCard() {
```

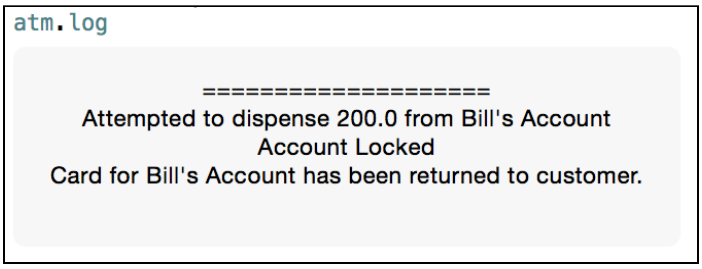
```
    // physically eject card
  }
}
```

In this example there are a multiple places that the method can exit early. One thing is for certain though, an ATM should always return the card to the user, regardless of what happens. The use of the `defer` block guarantees that no matter when the method exits, the user's card will be returned.

Find the line where `billsAccount` is declared and after it type the following to try dispensing funds:

```
do {
  try atm.dispenseFunds(200.00, account: &billsAccount)
} catch let error {
  print(error)
}
```

Attempting to dispense funds from a locked account throws an `ATMError.AccountLocked`, but add `atm.log` and read the output:



Bill's card was returned even though the method exited early.

Pattern Matching

Swift has had amazing pattern matching capabilities since the beginning, especially with cases in `switch` statements. Swift 2 continues to add to the language's ability in this area. Here are a few brief examples.

"for ... in" filtering combines a for-in loop and a `where` statement so that only iterations whose `where` statement is true will be executed. Use for ... in filtering to create an array of names that start with "C". Find the **Pattern Matching** section of the page and enter the following:

```
var namesThatStartWithC = [String]()

for cName in names where cName.hasPrefix("C") {
  namesThatStartWithC.append(cName)
}
```

You can also iterate over a collection of enums of the same type and filter out for a specific case. Given the array `authorStatuses` in the playground, write a for loop that matches on the `Late(Int)` case and calculates the total number of days that authors are behind.

```
var totalDaysLate = 0

for case let .Late(daysLate) in authorStatuses {
  totalDaysLate += daysLate
}
```

"if case" matching allows you to write more terse conditions rather than using `switch` statements that require a `default` case. The following iterates over the `authors` array and slaps each other who is late!

```
var slapLog = ""
for author in authors {
  if case .Late(let daysLate) =
    author.status where daysLate > 2 {

    slapLog += "Ray slaps \(author.name) around a bit " +
      "with a large trout.\n"
  }
}
```

Option Sets

Prior to Swift 2 in both Swift and Objective-C bitmasks were often used to describe a set of options flags. This should be familiar if you've ever done animations with UIKit.

You can now type a set of option flags like you would any other `Set<T>`, which was introduced in Swift 1.2.

```
animationOptions = [.Repeat, .CurveEaseIn]
```

Also, the fact that this is a Swift Set, you get all of the functionality that Sets have to offer. Creating your own option set is as simple as defining a structure that conforms to the `OptionSetType` protocol.

```
struct RectangleBorderOptions: OptionSetType {
  let rawValue: Int

  init(rawValue: Int) { self.rawValue = rawValue }

  static let Top = RectangleBorderOptions(rawValue: 0)
  static let Right = RectangleBorderOptions(rawValue: 1)
  static let Bottom = RectangleBorderOptions(rawValue: 2)
  static let Left = RectangleBorderOptions(rawValue: 3)
  static let All: RectangleBorderOptions =
    [Top, Right, Bottom, Left]
}
```

This section concludes the use of the chapter's playground, so you can put that aside.

OS Availability

As Apple continually introduces new versions of iOS (and OS X) they give developers new frameworks and APIs to utilize. The problem with this is that unless you drop support for previous versions of the OS you cannot safely use the new APIs without a lot of messy runtime checks which always leads to missed cases and basic human error, resulting in a crashing app.

In Swift 2 you can now let the compiler help you. There is not enough time to cover all of the use cases here but as a quick introduction consider the following example.

```
guard #available(iOS 9.0, *) else { return }
// do some iOS 9 or higher only thing
```

The code above "guards" on the iOS version running the application. If the iOS version is less than 9.0 the routine will exit immediately. This allows you to freely write against iOS 9 specific APIs beyond the `guard` statement without constantly checking for API availability using methods like `respondsToSelector(:)`.

The compiler will also you know if you've used a new API when your deployment target is set to some OS version where the API is not available.

Where to go from here?

While this chapter covered a lot of ground, you mostly just dipped your toes into each feature. There is a ton of power in the new features of Swift 2. And there are even more that were not covered here. It is highly recommended that you continue down the path of learning about Swift 2 features so that you can write better code and make better apps even faster. Never hesitate to crack open an Xcode Playground and start hacking away, prototyping ideas has never been easier. One pro-tip is to keep a playground in your Mac's Dock so that you can jump right in at a moment's notice.

You also should not miss the following WWDC 2015 sessions:

- What's New In Swift apple.co/1IBTu8q
- Protocol-Oriented Programming in Swift apple.co/1B8r2LE
- Building Better Apps with Value Types in Swift apple.co/1KMQesY
- Improving Your Existing Apps with Swift apple.co/1LiO462
- Swift and Objective-C Interoperability apple.co/1He5uhh
- Swift in Practice apple.co/1LPx2cq

All of these and more can be found at developer.apple.com/videos/wwdc/2015/.

And of course keep the official Swift Programming Language Book (apple.co/1n5tB6q) handy!

Chapter 2: Introducing App Search

By Chris Wagner

There's been a big hole in Spotlight on iOS for a long time. Although users can use it to find you app, they can't see *inside* it — to all the content that they really care about. Currently, when users want to get to content in an app, they have to flip through pages on their home screen to find it, open it and then search for what they're looking for — assuming you actually implemented a search feature in the first place!

An especially savvy user could launch your app by using Siri or searching for it in Spotlight, but neither of these tools help the user find what they want inside a non-Apple app. Meanwhile, Apple makes things like contacts, notes, messages, email and apps directly searchable within Spotlight. The user simply taps on the search result and goes straight to the content. No fair!

Sometimes it seems like Apple keeps all the fun features to itself, like using Spotlight. The good news is that after Apple developers finish playing around with a feature and feel it's ready for showtime, it often lets the masses play too, like it did with app extensions in iOS 8.

With iOS 9, Apple is passing a very exciting feature off to the rest of us; third party developers now have the ability to make *their* content searchable through Spotlight!

App search APIs

App search in iOS 9 comprises three main aspects. Each is broken into separate APIs that achieve distinct results, but they also work in concert with one another:

- NSUserActivity
- Core Spotlight
- Web markup

NSUserActivity

Being a clever little feature, this aspect of app search makes use of the same `NSUserActivity` API that enables Handoff in iOS 8.

Just in case you aren't aware, Handoff allows a user to start an activity on one device and continue it on another. For example, imagine you're reading an email on your iPhone as you sit down at your Mac. A special Mail icon will appear in your Mac's Dock, allowing you to launch Mail and resume reading the same email on your Mac. This is powered by `NSUserActivity`, which provides the OS with the information necessary to resume a task on another device. It's not voodoo — but is facilitated by a dance involving iCloud, Bluetooth and Wi-Fi.

In iOS 9, `NSUserActivity` has been augmented with some new properties to enable search. Theoretically speaking, if a task can be represented as an `NSUserActivity` to be handed off to a *different* device, it can be stored in a search index and later continued on the *same* device. This enables you to index activities, states and navigation points within your app, allowing the user to find them later via Spotlight.

For example, a travel app might index hotels the user has viewed, or a news app might index the topics the user browsed.

> **Note**: This chapter will not specifically cover Handoff, but you'll learn how to make content searchable once it's viewed.

Core Spotlight

The second, and perhaps most "conventional" aspect of app search is Core Spotlight, which is what the stock iOS apps like Mail and Notes use to index content. While it's nice to allow users to search for previously accessed content, you might take it a step further by making a large set of content searchable in one go.

You can think of Core Spotlight as a database for search information. It provides you with fine-grained control of what, when and how content is added to the search index. You can index all kinds of content, from files to videos to messages and beyond, as well as updating and removing search index entries.

Core Spotlight is the best way to provide full search capabilities of your app's private content. You'll learn how to use the new Core Spotlight APIs to index all of the content of an app.

Web markup

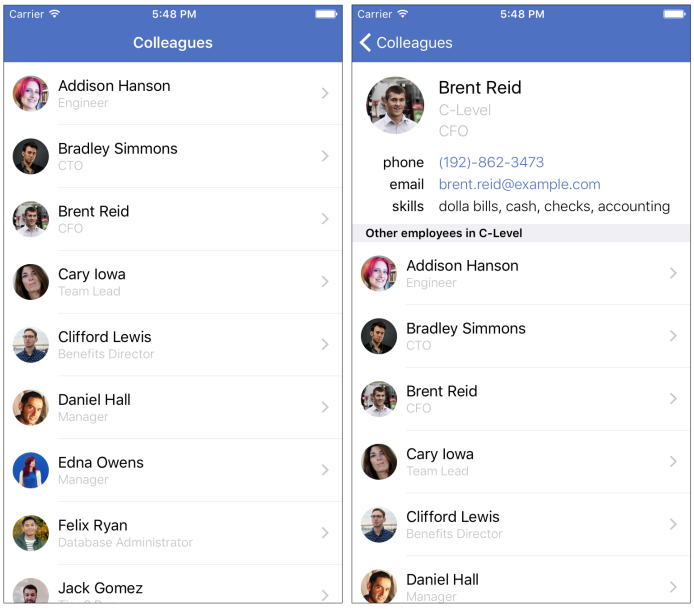
The third aspect of app search is web markup, which is tailored towards apps that mirror their public content from a web site. A good example is Amazon, where you can search the millions of products it sells, or even raywenderlich.com. Using open standards for marking up web content, you can show it in Spotlight and Safari

search results and even create deep links within your app.

This chapter will not cover web markup, but you can learn all about it in Chapter 3, "Your App On The Web".

Getting started

You'll work with a sample app named *Colleagues* that simulates a company address book. It provides an alternative to adding every coworker to your contacts, instead providing you with a directory of your colleagues. To keep things simple, it uses a local dataset, comprising of a folder of avatar images and a JSON file that contains employee information. In the real world, you would have a networking component that fetches contact data from a web-service. This is a tutorial, so JSON it is. Open the starter project, and before you do anything, build and run the app.



You'll immediately see a list of employees. It's a small startup, so there's only 25 staff. Select **Brent Reid** from the list to see all of his details. You'll also see a list of employees who are in the same department. And that is the extent of the app's features — it's very simple!

Search would make this app infinitely better. As it stands, you can't even search while you're *in* the app. You won't add in-app search, but instead add the ability to search from *outside* the app with Spotlight!

Sample project

Take a moment to familiarize yourself with the project's codebase. There are two targets: **Colleagues** which is the app itself, and **EmployeeKit** which is a framework to facilitate interactions with the employee database.

From the **EmployeeKit** group in Xcode, open **Employee.swift**. This is the model for an employee that has all the properties you might expect. Employee instances are initialized using a JSON model, which are stored under the **Database** group in a file named **employees.json**.

Moving on, open **EmployeeService.swift**. At the top of the file is an extension declaration, and there are two methods marked with `TODO`. You'll fill out these two method's implementation later. This service provides two documented public APIs:

- `indexAllEmployees()`: indexes all employee records via Core Spotlight
- `destroyEmployeeIndexing()`: destroys all indexing

There's more to the EmployeeKit target, but it's not related to app searches, so there's no need to go into it now. By all means though, feel free to poke around!

Open **AppDelegate.swift** in the **Colleagues** group. Notice there is only one method in here: `application(_:didFinishLaunchingWithOptions:)`. This implementation switches on `Setting.searchIndexingPreference`, which allows the user to change the behavior of search indexing.

Notice that the setting's value changes which method is called. If you recall, these are the service methods that had `TODO` comments to mark things you'll do later. You don't need to do anything other than just be aware of this setting. You can change the setting in the iOS system **Settings** app under **"Colleagues"**.

That concludes your tour. The rest of the code is view controller logic that you'll modify, but you don't need to know all of it to work with app search.

Searching previously viewed records

When implementing app search, `NSUserActivity` is the first thing to work with because:

1. It's dead simple. Creating an `NSUserActivity` instance is as easy as setting a few properties.
2. When you use `NSUserActivity` to flag user activities, iOS will rank that content so that search results prioritize frequently accessed content.
3. You're one step closer to providing Handoff support.

Time to prove how simple `NSUserActivity` can be to implement!

Implement NSUserActivity

With the **EmployeeKit group** selected, go to **File \ New \ File...**. Choose the **iOS \ Source \ Swift File** template and click **Next**. Name your new file **EmployeeSearch.swift** and verify that the target is set to **EmployeeKit**.

Within the new file, first import `CoreSpotlight`:

```
import CoreSpotlight
```

Next, still in **EmployeeSearch.swift**, add the following extension to the `Employee` struct:

```
extension Employee {
  public static let domainIdentifier =
    "com.raywenderlich.colleagues.employee"
}
```

This reverse-DNS formatted string identifies the type of `NSUserActivity` created for employees. Next, add the following computed property below the `domainIdentifier` declaration:

```
public var userActivityUserInfo: [NSObject: AnyObject] {
  return ["id": objectId]
}
```

This dictionary will serve as an attribute for your `NSUserActivity` to identify the activity. Now add another computed property named `userActivity`:

```
public var userActivity: NSUserActivity {
  let activity =
    NSUserActivity(activityType: Employee.domainIdentifier)
  activity.title = name
  activity.userInfo = userActivityUserInfo
  activity.keywords = [email, department]
  return activity
}
```

This property will come into play later to conveniently obtain an `NSUserActivity` instance for an employee. It creates new `NSUserActivity` and sets a few properties:

- `activityType`: The type of activity that this represents. You'll use this later to identify `NSUserActivity` instances that iOS provides to you. Apple suggests using reverse DNS formatted strings.
- `title`: The name of the activity — this will also appear as the primary name in a search result.
- `userInfo`: A dictionary of values for you to use however you wish. When the activity is passed to your app, such as when the user taps a search result in Spotlight, you'll receive this dictionary. You'll use it to store the unique employee ID, allowing you to display the correct record when the app starts.
- `keywords`: A set of localized keywords that help the user find the record when searching.

Next up, you are going to use this new `userActivity` property to make employee records searchable when the user views them. Since you added these definitions in

the EmployeeKit framework, you'll need to build the framework so that Xcode is aware they can be used from the Colleagues app.

Press **Command-B** to build the project.

Open **EmployeeViewController.swift** and add the following to the bottom of `viewDidLoad()`:

```
let activity = employee.userActivity

switch Setting.searchIndexingPreference {
case .Disabled:
  activity.eligibleForSearch = false
case .ViewedRecords:
  activity.eligibleForSearch = true
  activity.contentAttributeSet?.relatedUniqueIdentifier = nil
case .AllRecords:
  activity.eligibleForSearch = true
}

userActivity = activity
```

This retrieves `userActivity` — the property you just created in the `Employee` extension. Then it checks the app's search setting.

- If search is disabled, you mark the activity as ineligible for search.
- If the search setting is set to `ViewedRecords`, then you mark the activity as eligible for search, but also set `relatedUniqueIdentifier` to `nil`; if you don't have a corresponding Core Spotlight index item, then this must be `nil`. You'll give it a value when you perform a full index of the app's contents.
- If the search setting is `AllRecords`, you mark the activity as eligible for search.
- Finally, you set the view controller's `userActivity` property to your employee's activity.

Note: The `userActivity` property on the view controller is inherited from `UIResponder`. It's one of those things Apple added with iOS 8 to enable Handoff.

The last step is to override updateUserActivityState(). This ensures that when a search result is selected you'll have the information necessary.

Add the following method after `viewDidLoad()`:

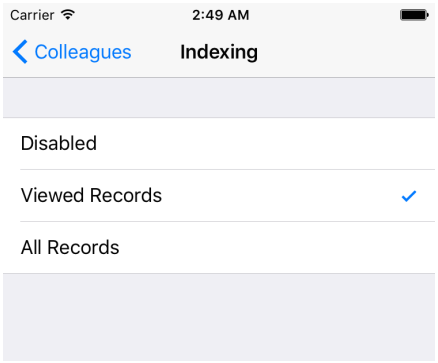
```
override func updateUserActivityState(activity: NSUserActivity){
  activity.addUserInfoEntriesFromDictionary(
    employee.userActivityUserInfo)
}
```

During the lifecycle of `UIResponder`, the system calls this method at various times

and you're responsible for keeping the activity up to date. In this case, you simply provide the `employee.userActivityUserInfo` dictionary that contains the employee's `objectId`.

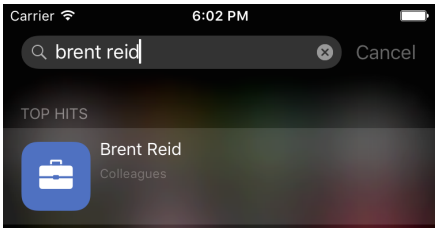
Great! Now when you pull up an employee, that bit of history will be tracked and become searchable, provided the setting is turned on.

In the simulator or on your device, open the **Settings** app and scroll down to **Colleagues**. Change the **Indexing** setting to **Viewed Records**.

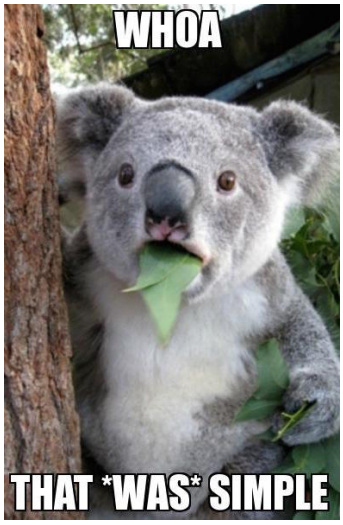


Now, build and run the app and select **Brent Reid**.

Okay, so it doesn't look like anything spectacular happened, but behind the scenes, Brent's activity is being added to the search index. Exit to the home screen (⇧⌘H) and bring up Spotlight by either swiping down from the middle of the screen or swiping all the way to the left of your home screen pages. Type **brent reid** into the search.



And there's Brent Reid! If you don't see him, you may need to scroll past other results. And if you tap on it, it should move up the list next time you perform the same search.



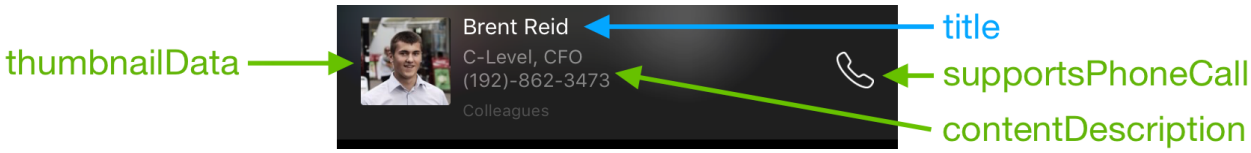
Now, of course this is *awesome*, but the result is a little...bland.

Surely you can do more than give a name? Time to crack open the **Core Spotlight** framework and discover how.

Adding more information to search results

`NSUserActivity` has a property named `contentAttributeSet`. It is of the type `CSSearchableItemAttributeSet`, which allows you to describe your content with as many attributes as necessary. Review the `CSSearchableItemAttributeSet` class reference to see the many ways to describe your content with these attributes.

Below is the desired result, complete with each component's property name called out:



You've already set `title` on `NSUserActivity`, and at the moment it's all you see. The other three, `thumbnailData`, `supportsPhoneCall` and `contentDescription` are all properties of `CSSearchableItemAttributeSet`.

Open **EmployeeSearch.swift**. At the top, import MobileCoreServices`:

```
import MobileCoreServices
```

`MobileCoreServices` is required for a special identifier that you'll use to create the `CSSearchableItemAttributeSet` instance. You've already imported `CoreSpotlight`, which is required for all of the APIs prefixed with `CS`.

Still in **EmployeeSearch.swift**, add a new computed property named `attributeSet` to the `Employee` extension:

```
public var attributeSet: CSSearchableItemAttributeSet {
  let attributeSet = CSSearchableItemAttributeSet(
    itemContentType: kUTTypeContact as String)
  attributeSet.title = name
  attributeSet.contentDescription =
    "\(department), \(title)\n\(phone)"
  attributeSet.thumbnailData = UIImageJPEGRepresentation(
    loadPicture(), 0.9)
  attributeSet.supportsPhoneCall = true

  attributeSet.phoneNumbers = [phone]
  attributeSet.emailAddresses = [email]
  attributeSet.keywords = skills

  return attributeSet
}
```

When initializing `CSSearchableItemAttributeSet`, an `itemContentType` parameter is required. You then pass in `kUTTypeContact` from the `MobileCoreServices` framework. (Read about these types on Apple's UTType Reference page (apple.co/1NilqiZ).

The attribute set contains the relevant search metadata for the current employee: `title` is the same as the title from `NSUserActivity`, `contentDescription` contains the employee's department, title and phone number, and `thumbnailData` is the result of `loadPicture()` converted to `NSData`.

To get the call button to appear, you must set `supportsPhoneCall` to `true` and provide a set of `phoneNumbers`. Finally, you add the employee's email addresses and set their various skills as keywords.

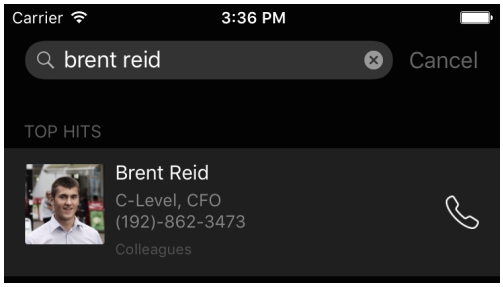
Now that these details are included, Core Spotlight will index each and pull the results during a search. This means that your users can now search for coworkers by name, department, title, phone number email and even skills!

Still in **EmployeeSearch.swift**, add the following line above the `return` in `userActivity`:

```
activity.contentAttributeSet = attributeSet
```

Here you tell the `contentAttributeSet` from `NSUserActivity` to use this information.

Build and run. Open **Brent Reid's** record so the index can do its thing. Now go to the home screen pull up Spotlight and search for "brent reid". If your previous search is still there, you'll need to clear it and search again.



Voila! Aren't you amazed with how little code it took to pull this off?

Great work! Now Spotlight can search for colleagues the user previously viewed. Unfortunately, there is one glaring omission...try opening the app from the search result. Nothing.

Opening search results

The ideal user experience is to launch the app directly to the relevant content without any fanfare. In fact — it's a requirement — Apple uses the speed at which your app launches and displays useful information as one of the metrics to rank search results.

In the previous section, you laid the groundwork for this by providing both an `activityType` and a `userInfo` object for your `NSUserActivity` instances.

Open **AppDelegate.swift** and add this empty implementation of `application(_:continueUserActivity:restorationHandler:)` below `application(_:didFinishLaunchingWithOptions:)`:

```
func application(application: UIApplication,
  continueUserActivity userActivity: NSUserActivity,
  restorationHandler: ([AnyObject]?) -> Void) -> Bool {

  return true
}
```

When a user selects a search result, this method is called — it's the same method that's called by Handoff to continue an activity from another device.

Add the following logic above `return true` in `application(_:continueUserActivity:restorationHandler:)`:

```
guard userActivity.activityType == Employee.domainIdentifier,
  let objectId = userActivity.userInfo?["id"] as? String else {
  return false
}
```

This `guard` statement verifies the `activityType` is what you defined as an activity for Employees, and then it attempts to extract the `id` from `userInfo`. If either of these fail, then the method returns `false`, letting the system know that the activity was not handled.

Next, below the `guard` statement, replace `return true` with the following:

```
if let nav = window?.rootViewController
    as? UINavigationController,
  listVC = nav.viewControllers.first
    as? EmployeeListViewController,
  employee = EmployeeService().employeeWithObjectId(objectId) {

  nav.popToRootViewControllerAnimated(false)

  let employeeViewController = listVC
    .storyboard?
    .instantiateViewControllerWithIdentifier("EmployeeView")
      as! EmployeeViewController

  employeeViewController.employee = employee
  nav.pushViewController(employeeViewController,
    animated: false)
  return true
}

return false
```

If the `id` is obtained, your objective is to display the `EmployeeViewController` for the matching Employee.

The code above may appear a bit confusing, but think about the app's design. There are two view controllers, one is the list of employees and the other shows employee details. The above code pops the application's navigation stack back to the list and then pushes to the specific employee's details view.

If for some reason the view cannot be presented, the method returns `false`.

Okay, time to build and run! Select **Cary Iowa** from the employees list, and then go to the home screen. Activate Spotlight and search for **Brent Reid**. When the search result appears, tap it. The app will open and you'll see it fade delightfully from Cary to Brent. Excellent work!

Indexing with Core Spotlight

Now that previously viewed records are indexed, you're close to indexing the entire database.

Why didn't you do this first? Well, `NSUserActivity` is generally an easy first step to make content searchable. Many apps don't have sets of content that can be indexed with Core Spotlight, but *all* apps have activities. Also, user activity affects what shows up in the list, so it's an easy way to put your app in the "spotlight".

Say that an employee shares a name with a famous actress. The user will likely see the actress's IMDB page as the first result the first time she searches. However, if she frequently visits that employee's record in Colleagues, it's likely that iOS will

rank that result higher than IMDB for her.

It's not a hard-set rule to implement `NSUserActivity` first; it's simply an easy point of entry. There's no reason why you couldn't implement Core Spotlight indexing first.

Start by opening **EmployeeSearch.swift** and add the following line to `attributeSet`, right above the return statement:

```
attributeSet.relatedUniqueIdentifier = objectId
```

This assignment creates a relationship between the `NSUserActivity` and what will soon be the Core Spotlight indexed object. If you don't relate the two records, there will be duplicate search results. You may remember you set this property to `nil` on your `NSUserActivity` when the app is in *Viewed Records* mode.

Next, you need to create the `CSSearchableItem`, which represents the object that Core Spotlight will index. Add the following computed property definition to **EmployeeSearch.swift**, below `attributeSet`:

```
var searchableItem: CSSearchableItem {
  let item = CSSearchableItem(uniqueIdentifier: objectId,
    domainIdentifier: Employee.domainIdentifier,
    attributeSet: attributeSet)
  return item
}
```

This is brief because you've already created the `CSSearchableItemAttributeSet` to hold most of the metadata. Notice that `uniqueIdentifier` is set to `objectId` to build the inverse relationship with your `NSUserActivity`.

Open **EmployeeService.swift** and import `CoreSpotlight` at the top of the file:

```
import CoreSpotlight
```

Now, within `indexAllEmployees()` replace the `TODO` comment with the following:

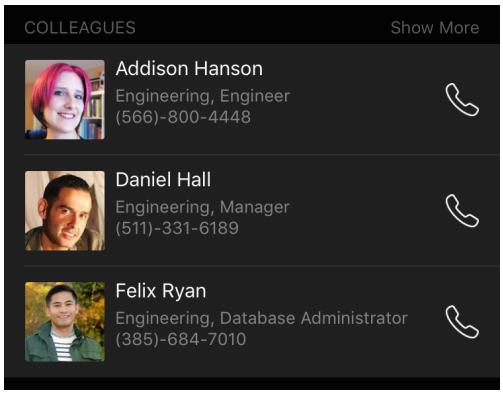
```
// 1
let employees = fetchEmployees()
// 2
let searchableItems = employees.map { $0.searchableItem }
CSSearchableIndex
  .defaultSearchableIndex()
// 3
  .indexSearchableItems(searchableItems) { error in
// 4
  if let error = error {
    print("Error indexing employees: \(error)")
  } else {
    print("Employees indexed.")
  }
}
```

Stepping through the logic...

1. All employees are fetched from the database as an array of `Employee`.
2. The employee array is mapped to `[CSSearchableItem]`.
3. Using Core Spotlight's default index, the array of `CSSearchableItems` is indexed.
4. Finally, log a message on success or failure.

And...that's it! Now when you launch the app with the option **All Records** set for the app's Indexing setting, all employee records become searchable.

Head over to the **Settings** app, and change the **Indexing** setting for Colleagues to **All Records**. Then build and run. In Spotlight, search for people in the list that you haven't looked at or search for an entire department, like engineering. You may need to scroll to see the results from Colleagues in the list.



> **Note**: You could see duplicate results because you were previously indexing `NSUserActivity` items without the `relatedUniqueIdentifier` set. You can delete the app to clear the index or continue to the next section to learn about removing indexed items.

Make the results do something

But what happens when you tap on a result? Not much! You need to handle results indexed with Core Spotlight a bit differently than you did for `NSUserActivity` results.

Open **AppDelegate.swift** and import `CoreSpotlight` at the top of the file:

```
import CoreSpotlight
```

Then replace `guard` statement in `application(_:continueUserActivity:restorationHandler:)` with the following:

```
let objectId: String
if userActivity.activityType == Employee.domainIdentifier,
  let activityObjectId = userActivity.userInfo?["id"]
  as? String {

  // 1
  objectId = activityObjectId
} else if userActivity.activityType ==
  CSSearchableItemActionType,
  let activityObjectId = userActivity
  .userInfo?[CSSearchableItemActivityIdentifier] as? String {

  // 2
  objectId = activityObjectId
} else {
  return false
}
```

The user activity supplied to this delegate method will now have one of two types, handled with the `if` statement above:

1. If the result was indexed by `NSUserActivity` then the `activityType` will be the one you defined in reverse-DNS notation. In this instance, the employee ID is obtained from the `userInfo` dictionary, as before.
2. A result that Core Spotlight indexed directly, will arrive as an `NSUserActivity` with an `activityType` of `CSSearchableItemActionType`. Furthermore, the unique identifier is stored in the `userInfo` dictionary under the key `CSSearchableItemActivityIdentifier`. This logic handles both cases, regardless of how the employees are indexed.

Build and run, then try to select an employee. The app should open to the chosen result, as you'd expect.

Deleting items from the search index

Back to the premise of your app. Imagine that an employee was fired for duct taping the boss to the wall after a particularly rough day. Obviously, you won't contact that person anymore, so you need to remove him and anybody else that leaves the company from the Colleagues search index.

For this sample app, you'll simply delete the entire index when the app's indexing setting is disabled.

Open **EmployeeService.swift** and find `destroyEmployeeIndexing()`. Replace the `TODO` with the following logic:

```
CSSearchableIndex
  .defaultSearchableIndex()
  .deleteAllSearchableItemsWithCompletionHandler { error in

  if let error = error {
    print("Error deleting searching employee items: \(error)")
  } else {
    print("Employees indexing deleted.")
  }
}
```

This single parameterless call destroys the entire indexed database for your app. Well played!

Now to test out the logic; perform the following test to see if index deletion works as intended:

1. Build and run to install the app.
2. Stop the process in Xcode.
3. In the simulator or on your device, go to **Settings \ Colleagues** and set **Indexing** to **All Records**.
4. Open the app again. This will start the indexing process.
5. Go back to the home screen and activate Spotlight.
6. Search for a known employee and verify the entry appears.
7. Go to **Settings \ Colleagues** and set **Indexing** to **Disabled**.
8. Quit the app.
9. Reopen the app. This will purge the search index.
10. Go to the home screen and activate Spotlight.
11. Search for a known employee and verify that *no* results appear.

So deleting the entire search index was pretty easy, huh? But what if you want to single out a specific item? Good news — these two APIs give you more fine-tuned control over what is deleted:

- `deleteSearchableItemsWithDomainIdentifiers(_:completionHandler:)` lets you delete entire "groups" of indexes based on their domain identifiers.
- `deleteSearchableItemsWithIdentifiers(_:completionHandler:)` allows you work with individual records using their unique identifiers.

This means that globally unique identifiers (within an app group) are *required* if

you're indexing multiple types of records.

> **Note**: If you can't guarantee uniqueness across types, like when replicating a database that uses auto-incrementing IDs, then a simple solution could be to prefix the record's ID with its type. For example, if you had a contact object with an ID of 123 and an order object with an ID of 123, you could set their unique identifiers to **contact.123** and **order.123** respectively.

It is also very important to keep indexes up-to-date with changes. In the case of Colleagues you need to handle promotions, department changes, new phone numbers or even name changes. To update indexed items, use the same method that you indexed them with in the first place: `indexSearchableItems(_:completionHandler:)`.

Great work! Once you have all of the above working, you can set the sample project aside. The next sections will discuss some advanced features of app search.

Private vs. public indexing

> **Note:** This section is inaccurate for iOS 9.0; Apple has not yet implemented the feature according to Technical Note TN2416 (bit.ly/1NC7u72). "Activities marked as `eligibleForPublicIndexing` are kept on the private on-device index in iOS 9.0, however, they may be eligible for crowd-sourcing to Apple's server-side index in a future release." Keep an eye on this technical note if the feature is of interest to you. The information described below is what Apple had outlined during the WWDC session and documentation.

Imagine if a user could search for something in Spotlight, and see a result from inside an app *that they don't even have installed*! How cool would that be? Well, with public indexing, this is possible. Content from your app would appear in front of more users — helping them out with contextually relevant information, and hopefully getting you some extra downloads.

By default, all indexed content is considered private. In fact, all content you index using Core Spotlight will always be private. You can, however, mark an `NSUserActivity` as being publicly indexable by setting the `eligibleForPublicIndexing` property to `true`.

There's actually a little more to it than that. In order for content to become public in Apple's cloud index, it must first be reported by an undefined number of unique users. This protects users' privacy as well as maintaining the quality of the public index.

As you might expect Apple has not quantified this threshold.

The other approach for making content publicly indexed is using *web markup*, which is covered in the next chapter.

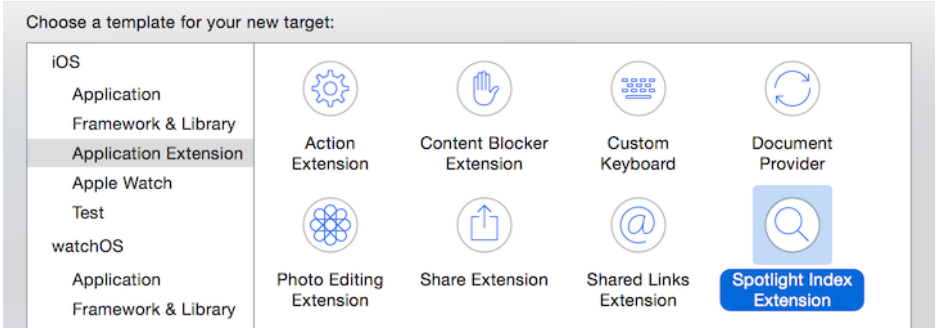
Advanced features

The Core Spotlight framework provides a couple of more advanced features you'll want to know about.

Core Spotlight App Extensions

A Core Spotlight app extension provides your app the opportunity to run maintenance operations on its index when the app is not running. In the event your app's index is lost or did not properly save the system may call your extension to perform its duty.

To add this extension, simply add a new **Spotlight Index Extension** target to your project.



Spotlight index extensions contain a subclass of `CSIndexExtensionRequestHandler`. This declares conformance to the `CSSearchableIndexDelegate` protocol, which has only two required methods:

- `searchableIndex(_: reindexAllSearchableItemsWithAcknowledgementHandler:)`
- `searchableIndex(_: reindexSearchableItemsWithIdentifiers: acknowledgementHandler:(`

These both request you to re-index some of the items of content in your app — the former specifying that all items should be reindexed, while the latter provides an array of IDs for the items that need updating.

Batch indexing

If you have a large amount of data to index with Core Spotlight, then rather than indexing each item individually, it's a lot more efficient to submit items in batches.

Core Spotlight provides APIs for this too — including providing the ability to continue indexing where you left off, should the process get interrupted.

You can't use the `defaultSearchableIndex` for batch indexing; you need to create your own `CSSearchableIndex` instance.

The conceptual approach to using the batch API is as follows:

1. Create a new `CSSearchableIndex`.
2. Designate that you'll begin batch indexing by calling `beginIndexBatch()` on the index.
3. Request the details of the last batch with `fetchLastClientStateWithCompletionHandler()`.
4. Prepare the next batch of `CSSearchableItem` objects to index.
5. Index the items using `indexSearchableItems(_:completionHandler:)` as before.
6. Use `endIndexBatchWithClientState()` to specify that this batch is finished. The client state you provide is what you'll be provided when you next reach step 3.
7. Repeat!

For further information on batching, check out the Apple documentation for `CSSearchableIndex`.

Where to go from here?

This chapter has covered iOS 9's simple yet powerful approach to indexing the content inside your app, either through Core Spotlight or user activities. The latter of these isn't limited to *content* though — you can also use it to index navigation points within an app.

Consider a CRM app that has multiple sections such as *Contacts*, *Orders* and *Tasks*. By creating user activities whenever a user lands on one of these screens, you'd make it possible for them to search for *Orders* and be directly to that section of your app. How powerful would this be if your app has many levels of navigation?

There are many unique ways to bubble up content to your users. Think outside the box and remember to educate your users about this powerful function.

To find out more about app search, be sure to watch *Session 709 - Introducing Search APIs* from WWDC 2015 (apple.co/1gvlfGi). The App Search Programming Guide (apple.co/1J0WBs1) is also an excellent reference for implementing search in your own apps.

Finally, if your app has a web counterpart, then jump straight in to the next chapter of this book, "Your App On The Web". You'll learn more about indexing the content that is available publicly, both on the web and within your app.

Chapter 3: Your App on the Web

By Pietro Rea

Native and web technologies have historically lived in two distinct camps within iOS: native apps are part of a closed ecosystem tightly controlled by Apple, whereas web technologies are based on open standards and frameworks. These two worlds don't cross paths often, and what happens on a mobile website often doesn't directly affect what happens in a native app.

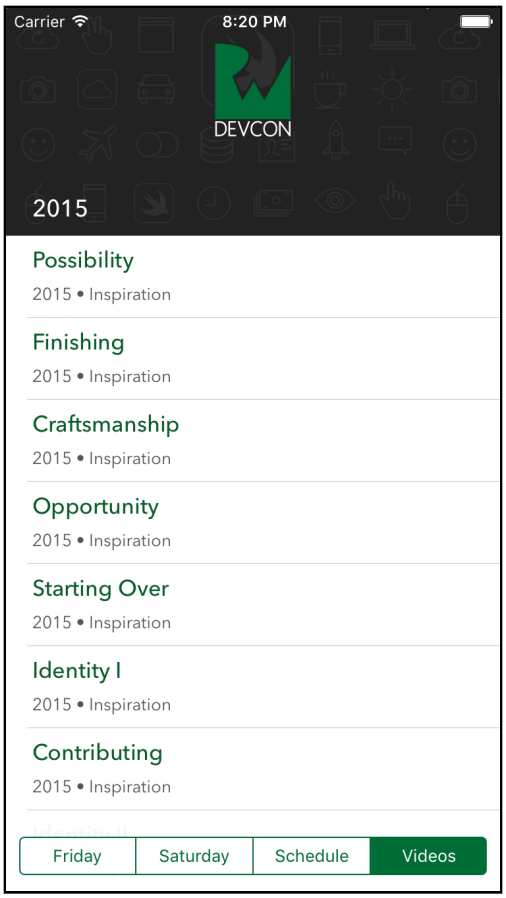
In the last few years, Apple has worked to bring the camps of web and native apps closer together. iOS 7 introduced JavaScriptCore, a framework to bridge native code with JavaScript. iOS 8 saw the release of features such as Continuity and shared web credentials.

iOS 9 pulls the web and native worlds just a little closer with **universal links** and **web markup**, which let you provide deep links directly into your app and surface web content in Spotlight and Safari search.

You probably have a bunch of ideas on how to use those features from that basic introduction alone, so jump straight into the next section to see how to blur the lines between web and native apps.

Getting started

In this chapter, you'll be modifying the code base for the **RWDevCon** app. For reference, **RWDevCon** is the conference organized by the folks behind raywenderlich.com. You'll also be making some tweaks to its accompanying website: rwdevcon.com.



In the starter files for this chapter, you'll find both the code for the iOS app and the code for the website. There's quite a lot there, but don't be put off – you'll only be editing one or two files and adding some extra functionality to the videos section. Feel free to take a look through the project to familiarize yourself with its contents.

> **Note:** Due to the infrastructure and security requirements for web markup and universal links, this chapter is unfortunately the only place in this book where you **won't** be able to verify your work as you follow along. There's no easy way to try out these features without having a real website accessible via HTTPS and an associated app in the App Store under an account where you're either the team agent or the team admin.
>
> The rest of this chapter includes a number of tutorial sections to give you some experience with universal links and web markup. You won't be able to run the sample app on a device, since you won't have the required provisioning profiles, but you can still get an understanding of how everything fits together.

Linking to your app

If you've ever linked into a native app, either from a website or from another app, then you're probably familiar with the predecessor to universal links: **deep links**. Before diving into universal links, read through the following refresher on deep links so you'll know exactly how they differ from the new technology you'll explore in this chapter.

Deep links

Before iOS 9, the primary way apps communicated with each other was to register a custom URL scheme using the `CFBundleURLTypes` key in **Info.plist**. For example, if you were developing a social network app for clowns you might have registered something like `clownapp://` or `clown://`. You might have seen some of Apple's own URL schemes in use, such as `tel://`, `sms://`, or `facetime://`.

Once you've registered your custom scheme, you can link into your app from other places in iOS by constructing a URL such as `clownapp://home/feed`. When the link is opened, iOS launches your app and passes in the entire URL via the app delegate method `application(_:handleOpenURL:)`. Your app then can interpret the URL in any manner and respond appropriately.

This system worked fairly well for a long time (since iOS 3.0, in fact!) but it has some major drawbacks:

- **Privacy:** In addition to `openURL(_:)`, `UIApplication` also has the method `canOpenURL(_:)`. The *intended* purpose of this innocent-looking method is to check if there's an app installed on the device that can handle a specific URL. Unfortunately, this method was exploited to gather a list of installed apps; if you know that a particular device can open a `clownapp://` URL then you also know that the device has the social clown app installed.
- **Collisions**: Facebook's custom URL scheme is `fb://`. There's nothing stopping another app from registering `fb://` as *their* URL scheme and capturing Facebook's deep links. When two apps register for the same custom URL scheme, it's indeterminate which app will win out and launch.
- **No fallback**: If iOS tries to open a link of a custom URL scheme that's not registered to *any* app, the action fails silently.

iOS 9 solves many of these problems and more with universal links; instead of registering for custom URL schemes, universal links use standard HTTP and HTTPS links. You can register to handle specific links for any web domains that you own.

Universal links

If you owned the domain `clownapp.com`, you could register `http://clownapp.com/clowns/*` as a universal app link. If the user installs your social clown app and taps

the link `http://clownapp.com/clowns/fizbo` within Safari or a web view, iOS takes them straight to Fizbo's profile within your app.

If they don't have the app installed, they are taken to the clown's information on your website, just as if they'd followed a standard HTTP link. You'll see the same behavior if you open the link with `openURL(_:)`.

Universal links have other advantages over deep links:

- **Unique:** There's no way other apps can register a handler for your domain.
- **Secure:** To tie your domain and app together, you have to upload a securely-signed file to your web server. There's also no way for other apps to tell whether your app is installed.
- **Simple:** A universal link is just a normal HTTP link which works for both your website and your app. So even if a user doesn't have your app installed, or isn't running iOS 9, the link will still bring them to Safari.

Enough theory for now — time to find out how to add universal links to your own apps.

Registering your app to handle universal links

To tie your website to your native app and prove that it's *your* website, there are two 'bonds' you have to create: you have to tell your native app about your domain, and you have to tell your domain about your native app. After that, you simply need to add some code to your app to handle incoming links.

First up: letting the RWDevCon app know about the `rwdevcon.com` domain.

Go to the starter files included with this chapter and open **RWDevCon.xcodeproj** from the **rwdevcon-app** directory. In the project navigator, select the **RWDevCon project**, then the main **RWDevCon target**. Switch to the **Capabilities** tab and add the following domains to the **Associated Domains** section:

- `applinks:rwdevcon.com`
- `applinks:www.rwdevcon.com`

Your Associated Domains section should look like the following when done:

This tells iOS which domains your app should respond to. Make sure to prefix your

domain names with `applinks:`.

Xcode may prompt you to select a development team when you attempt to change these settings. For the purposes of this tutorial, simply cancel out of this dialog when it appears. Unfortunately, you won't be able to run the demo app on a device since you're not a member of the Ray Wenderlich development team.

When you add the associated domains for your *own* app, make sure you first sign into Xcode with the appropriate Apple ID. Do this by going to **Xcode\Preferences\Accounts** and tapping on the **+** button. You'll then need to turn on the Associated Domains setting and add any domains you want to listen for.

> **Note**: Only a **team agent** or a **team administrator** of an Apple developer account can turn on the Associated Domains capability. If you're not assigned one of those roles, reach out to the right person on your team to make this change.

Registering your server to handle universal links

Next, you have to create the link from your website to your native app. To do this, you need to place a JSON file on your webserver that contains some information about your app. You won't be able to follow along for this section since you don't have access to the `rwdevcon.com` web server to upload a file, but here's what the contents of that file should look like:

```
{
  "applinks": {
    "apps": [],
    "details": [
      {
        "appID": "KFCNEC27GU.com.razeware.RWDevCon",
        "paths": [
          "/videos/*"
        ]
      }
    ]
  }
}
```

The file must be named **apple-app-site-association** and it *must not* have an extension — not even `.json`.

The file name might look familiar to you, and for good reason! Apple introduced the **apple-app-site-association** file in iOS 8 to implement shared web credentials between your website and your app as well as for Handoff tasks between web and native apps.

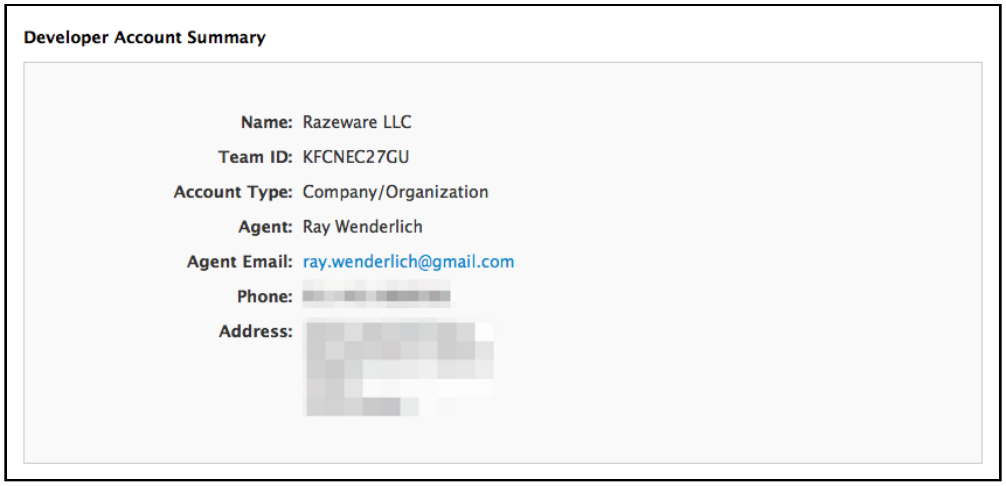
The `applinks` section of this file determines which of your apps can handle particular URL paths on your website. Somewhat confusingly, the `"apps"` property should *always* be an empty array.

The `details` section contains an array of dictionaries pairing an `appID` with a list of `paths`.

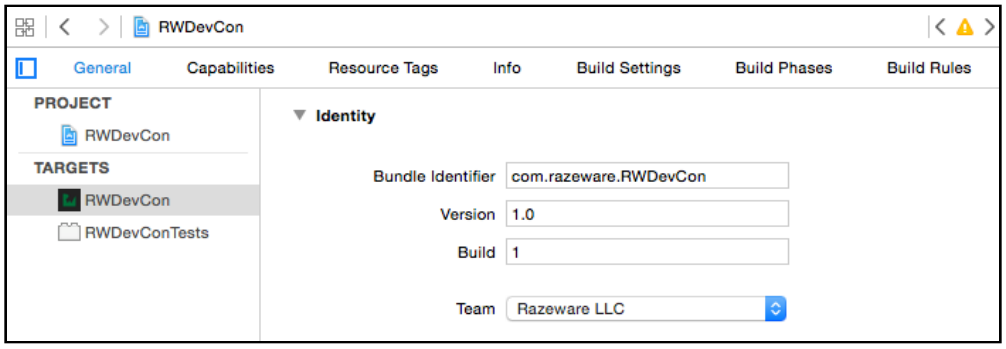
Your `appID` string consists of your **team ID** (`KFCNEC27GU` in this example) followed by your app's **bundle ID** (`com.razeware.RWDevCon` in this case).

The team ID is supplied by Apple and is unique to a specific development team. `KFCNEC27GU` is specific to the Ray Wenderlich development team; you'll have a different identifier for your own account.

If you don't know your team ID, the easiest way to find it is by logging into Apple's developer member center developer.apple.com/membercenter. Log in, click on **Your Account**, and then look for your team ID within the account summary:



If you don't know your app's bundle ID, go to Xcode and click on your project in the **project navigator**. Select the main target for your app and switch to the **General** tab. The identifier you're looking for is listed as **Bundle Identifier**:



A bundle ID typically uses reverse-DNS notation in the form `com.exampledomain.exampleappname`.

As the name of **apple-app-site-association** suggests, the `paths` array contains a list of "white-listed" URLs that your app should handle. If you're a little rusty on your URL components, the path in the following URL is **/videos/2015/**

inspiration/; it's the bit that follows the domain name:

```
https://www.rwdevcon.com/videos/2015/inspiration/?name=Inspiration
```

The `paths` array can support some basic pattern matching, such as the `*` wildcard, which matches any number of characters. It also supports `?`, which matches any single character. You can combine both wildcards in a single path, such as `/videos/*/year/201?/videoName`, or you can simply use a single `*` to specify your entire website.

That's all there is to path management; you can add as many paths as you like for your app to handle, or even add multiple apps to handle different paths.

Once you have **apple-app-site-association** ready, you have to upload it to the root of your web server. In the case of RWDevCon, since you specified both `rwdevcon.com` and `www.rwdevcon.com` in your associated domains, the file must be accessible at the following locations:

```
https://rwdevcon.com/apple-app-site-association
https://www.rwdevcon.com/apple-app-site-association
```

It must be hosted without any redirects, and be accessible **over HTTPS**.

Since you don't have access to the web servers that host `www.rwdevcon.com`, you can't do this step yourself. Luckily, Ray has already uploaded the file to the root of the web server for you – thanks Ray! You can verify it's there by requesting the file through your favorite web browser.

Before moving on to the next section, there are two caveats to consider when managing your site association file:

1. If your app must target iOS 8 because it contains Continuity features such as Handoff or shared web credentials, you'll have to sign **apple-app-site-association** using `openssl`. You can read more about this process in Apple's Handoff Programming Guide apple.co/1yG4jR9.
2. Before you upload **apple-app-site-association** to your web server, run your JSON through an online validator such as JSONLint jsonlint.com. Universal links won't work if there's even the slightest syntax error in your JSON file!

Handling universal links in your app

When your app receives an incoming universal link, you should respond by taking the user straight to the targeted content. The final steps in implementing universal links are to parse the incoming URLs, determine what content to show, and navigate the user to the content.

Head back to the **RWDevCon project** in Xcode and add the following class method to **Session.swift** at the bottom of the class:

```
class func sessionByWebPath(path: String,
  context: NSManagedObjectContext) -> Session? {

  let fetch = NSFetchRequest(entityName: "Session")
  fetch.predicate =
    NSPredicate(format: "webPath = %@", path)

  do {
    let results = try context.executeFetchRequest(fetch)
    return results.first as? Session
  } catch let fetchError as NSError {
    print("fetch error: \(fetchError.localizedDescription)")
  }

  return nil
}
```

In the RWDevCon app, the Core Data class `Session` represents a particular presentation and contains a `webPath` property that holds the path of the corresponding video page on `rwdevcon.com`. The method above takes a URL's path, such as `/videos/talk-ray-wenderlich-teamwork.html`, and returns either the corresponding session object or `nil` if it can't find one.

Next, open **AppDelegate.swift** and add the following helper method to `AppDelegate`:

```
func presentVideoViewController(URL: NSURL) {
  let storyboard = UIStoryboard(name: "Main", bundle: nil)
  let navID = "NavPlayerViewController"

  let navVideoPlayerVC =
    storyboard.instantiateViewControllerWithIdentifier(navID)
    as! UINavigationController

  navVideoPlayerVC.modalPresentationStyle = .FormSheet

  if let videoPlayerVC = navVideoPlayerVC.topViewController
    as? AVPlayerViewController {

    videoPlayerVC.player = AVPlayer(URL: URL)

    let rootViewController = window?.rootViewController
    rootViewController?.presentViewController(
      navVideoPlayerVC, animated: true, completion: nil)
  }
}
```

This method takes in a video URL and presents an `AVPlayerViewController` embedded in a `UINavigationController`. The video player and container navigation controller have already been set up and are loaded from the main storyboard. If you want to see how they're configured you can open **Main.storyboard** to have a

look.

Finally, still in **AppDelegate.swift**, implement the following `UIApplicationDelegate` method below the method you just added:

```
func application(application: UIApplication,
  continueUserActivity
  userActivity: NSUserActivity,
  restorationHandler: ([AnyObject]?) -> Void) -> Bool {

  //1
  if userActivity.activityType ==
    NSUserActivityTypeBrowsingWeb {

    let universalURL = userActivity.webpageURL!

    //2
    if let components = NSURLComponents(URL: universalURL,
      resolvingAgainstBaseURL: true),
      let path = components.path {

      if let session = Session.sessionByWebPath(path,
        context: coreDataStack.context) {
        //3
        let videoURL = NSURL(string: session.videoUrl)!
        presentVideoViewController(videoURL)
        return true
      } else {
        //4
        let app = UIApplication.sharedApplication()
        let url =
          NSURL(string: "http://www.rwdevcon.com")!
        app.openURL(url)
      }
    }
  }
  return false
}
```

The system calls this delegate method when there's an incoming universal HTTP link. Here's a breakdown of what each section does:

1. The system invokes this method for several types of `NSUserActivity`; `NSUserActivityTypeBrowsingWeb` is the type that corresponds to universal HTTP links. When you see a user activity of this type, you're guaranteed that the `NSUserActivity` instance will have its `webPageURL` property of type `NSURL?` set to something you can inspect. Therefore you can unwrap the optional.

2. You use an instance of `NSURLComponents` to extract the URL's path; you can then use `sessionByWebPath(_:context:)` along with the URL to map to the correct `Session` object.

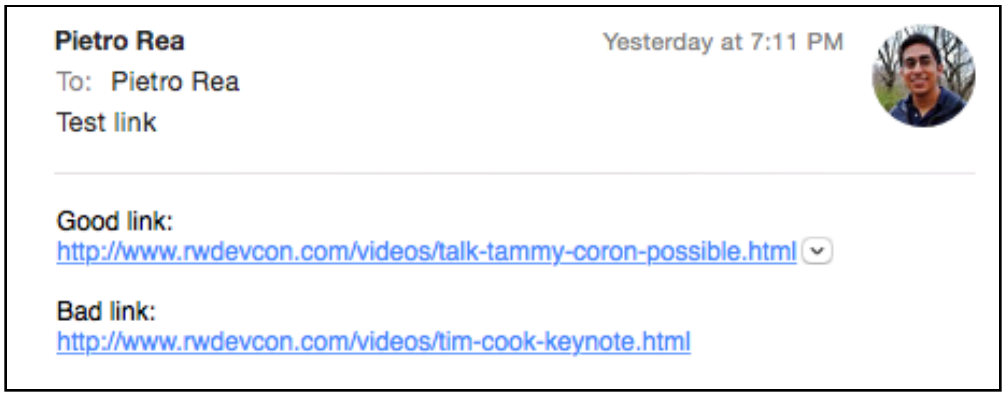
3. `sessionByWebPath(_:context:)` returns an optional `Session`. If there's a value behind the optional, you use the session's `videoURL` property to present the video player using `presentVideoViewController(_:)`.

4. If there's no value behind the optional, which can happen if you're handed a universal link the app can't understand, you simply launch the RWDevCon home page in Safari and return `false` to tell the system you couldn't handle the activity.

> **Note:** `application(_:continueUserActivity:restorationHandler:)` may look familiar to you; Apple introduced this `UIApplicationDelegate` method in iOS 8 to allow developers to implement Handoff. It also makes an appearance in Chapter 2, "Introducing App Search", which deals with the new search APIs in iOS 9. This method is a jack of all trades!

Although you won't be able to validate the code you just wrote, it's still useful to see how the app *would* handle an incoming link if the app were live in the App Store.

Let's pretend for a moment that you have the RWDevCon app installed on your device and you received an e-mail containing the following two links:



Tapping on the first link would open the app and start streaming Tammy Coron's 2015 inspiration talk titled "Possibility":

Looks great! Now if you were to return to your email client and tap the second link, the app would open but you'd be immediately bounce back to Safari, as shown below:

This is how the RWDevCon app would handle incoming universal links and how it would gracefully fall back to Safari.

Besides tapping a link, you can also load the URL directly in Safari, a `WKWebView`, a `UIWebView`, or use `openURL(_:)` on an instance of `UIApplication` to trigger your app to handle a universal link.

Did you notice the banner at the top of the previous screenshot? That's a **Smart App Banner**; you'll learn more about those in the second half of the chapter.

Working with web markup

Now that you know how to implement and handle universal links in iOS 9, it's time to move to the second topic of this chapter: web markup. As it turns out, web markup is part of a much bigger topic that's covered about in Chapter 2, "Introducing App Search".

Search includes three different APIs: `NSUserActivity`, `CoreSpotlight` and web markup. Chapter 2 covered `NSUserActivity` and `CoreSpotlight`; it's well worth a read through that chapter if you haven't done so already.

Search results that appear in Spotlight and in Safari can now include content from native apps in iOS 9, and you can use web markup to get your app's content to surface in those search results. If you have a website that mirrors your app's content, you can mark up its web pages with standards-based markup, Smart App Banners, and universal links your native app understands.

Apple's web crawler, lovingly named "Applebot", will then crawl your website and index your mobile links. When iOS users search for relevant keywords, Apple can surface your content *even if users don't have your app installed*. In other words, optimizing your markup on your site helps you get new downloads organically.

Making your website discoverable

Applebot crawls the web far and wide, but there's no guarantee when, or even if, it will land on your website. Fortunately, there are a few things you can do to make your site more discoverable and easier to crawl.

1. In iTunes Connect, point your app's **Support URL** as well as its **Marketing URL** to the domain that contains your web markup. These support URLs are Applebot's entry points to crawl your content:

2. Make sure the pages that contain your web markup are accessible from your support marketing URLs. If there aren't any direct paths from these URLs to your web markup, you should create them so Applebot can find the web markup.

3. Check that your site's **robots.txt** file is set up so that Applebot can do its job. **Robots.txt**, also known as the *robots exclusion protocol*, is a web standard for communicating with web crawlers and other web robots; it specifies which parts of the site the web crawler should not scan or process.

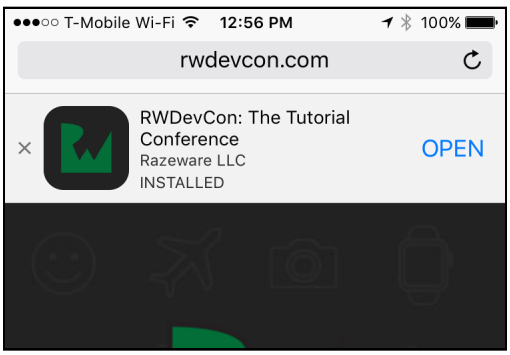
> **Note:** Not all web crawlers follow these directives, but Applebot does! You can learn more about the robots exclusion standard on Wikipedia bit.ly/1MNna6A.

Embedding universal links using Smart App Banners

Once Applebot can find and crawl your website, the next step is to add something worth crawling! Apple recommends the use of Smart App Banners to add mobile links to your site.

Smart App Banners have been around since iOS 6; they've typically been seen as advertising banners that promote apps on a website. Visitors who didn't have your app installed would get a link to the app in the App Store; for those who did have the app installed, the banner could provide an easy way to link to a page deep within your app.

Here's the Smart App Banner from the last section up close:



This particular Smart App Banner promotes the RWDevCon iOS app on the RWDevCon website. Since Safari's detected that the visitor has the app installed, the banner says **OPEN**; otherwise, the Smart App Banner would say **VIEW** and take you to the App Store page for the RWDevCon app. That's why they call them "smart" banners! :]

iOS 9 brings new uses to Smart App Banners by making them an integral part of search. In addition to their day job as marketing tools, Smart App Banners can also help surface universal links for Applebot to crawl and index.

> **Note:** The remainder of this chapter is all about editing the HTML of your website to improve its discoverability and presence in search results in iOS 9. It should be easy enough to follow, even if you're not familiar with HTML. If somebody else takes care of your website for you, be sure to show them this chapter so they can make the required changes! :]

Go to the starter files that came with this chapter and locate the source code for `www.rwdevcon.com` in the **rwdevcon-site** folder. Open **talk-ray-wenderlich-teamwork.html** in the **videos** folder and add the following `meta` tag inside the `head` tag, above the page title:

```
<meta name="apple-itunes-app" content="app-id=958625272, app-
argument=http://www.rwdevcon.com/videos/talk-ray-wenderlich-
teamwork.html">
```

The `name` attribute of this `meta` tag is very important, and must always be **apple-itunes-app**. This identifies the type of the meta tag as a Smart App Banner meta tag, which in turn tells Safari to display your Smart App Banner.

The `content` attribute contains two important parameters:

- **app-id**: This parameter corresponds to your app's Apple ID. Yes, apps have Apple IDs too! But this is different from the sort of Apple ID you use to log into iCloud. Your app's Apple ID is simply a unique number; all apps on the App Store have them. The easiest way to find your app's ID is to log into iTunes Connect, click **My Apps** and then navigate to the app in question. The Apple ID for RWDevCon is `958625272`; the ID would be different for your own app.
- **app-argument:** This contains the URL Safari will pass back to the app if it's installed. Prior to iOS 9, the value of this parameter was a custom URL scheme deep link, but Apple now strongly recommends you switch to HTTP/HTTPS universal links.

> **Note:** This was a quick overview of Smart App Banners. To learn more about their full capabilities, read Ray's Smart App Banners tutorial bit.ly/1iYlyea as well as the Safari Web Content Guide apple.co/1KYeI4I.

Adding Smart App Banners to your website is helpful for many reasons, including better odds of being indexed by Applebot. However, it's worth noting that Smart App Banners only work in Safari; if a visitor comes to your website through another browser such as Chrome, they won't see the banner.

Apple understands that not everyone wants to use Smart App Banners, which is why Applebot also supports two other methods of surfacing mobile links: Twitter Cards and App Links.

This is what the RWDevCon example would look like using Twitter Cards:

```
<meta name="twitter:app:name:iphone" content="RWDevCon">
<meta name="twitter:app:id:iphone" content="958625272">
<meta name="twitter:app:url:iphone" content="http://www.rwdevcon.com/
videos/talk-ray-wenderlich-teamwork.html">
```

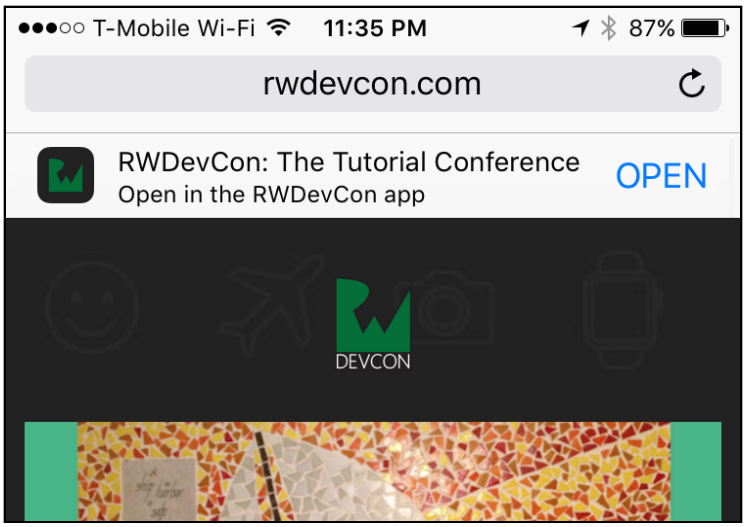
And with Facebook's App Links:

```
<meta property="al:ios:app_name" content="RWDevCon">
<meta property="al:ios:app_store_id" content="958625272">
<meta property="al:ios:url" content="http://www.rwdevcon.com/videos/talk-
ray-wenderlich-teamwork.html">
```

> **Note**: To learn more, read through Twitter's documentation page on Twitter Cards dev.twitter.com/cards/mobile as well as Facebook's App Links documentation applinks.org.

Since you don't have the privileges to deploy code to `rwdevcon.com` (sorry, Ray's kind of picky about things like that), you won't be able to see your changes in action.

However, let's pretend for a moment that everything is correctly deployed on the site and on the app. In that case, if you were to use mobile Safari to load http://www.rwdevcon.com/videos/talk-jake-gundersen-opportunity.html (that's the video for Jake Gundersen's 2015 talk), the top of the web page would look like this:



If you don't see the Smart App Banner, swipe down on the page until it comes into view. Notice anything different from the Smart App Banner you saw earlier? This one is thinner, and has changed to say **Open in the RWDevCon app**. This special banner only shows up for URLs that match at least one of the paths specified in **apple-app-site-association**.

You can verify this behavior by navigating to the site root rwdevcon.com. You'll see the regular-sized banner, not the thin banner you saw on the video page. Even though the homepage *also* has the appropriate meta tag, the URL in its `app-argument` parameter doesn't match the **/videos/** path you specified in **apple-app-site-association**.

> **Note:** Smart App Banners don't work on the iOS simulator, so you must use a device to view and interact with the banners.

Tap the thin banner; Safari opens the RWDevCon app and plays the correct video via your implementation of `application(_:continueUserActivity:restorationHandler:)` in the previous section.

Semantic markup using Open Graph

You've learned how to add Smart App Banners to a web page to make it easier for Applebot to index universal links. However, just because Applebot can find and crawl a website doesn't mean its content will show up in Spotlight! The content also has to be relevant and engaging if it has any chance of competing with other search results.

Apple doesn't reveal much about the relevance algorithm that determines the ranking for Spotlight search results, but it has said that *engagement* with content will be taken into consideration. If users tap or otherwise significantly engage with your search results, your site result will rank relatively higher than other site.

To this end, Apple recommends adding markup for structured data to allow Spotlight to provide richer search results.

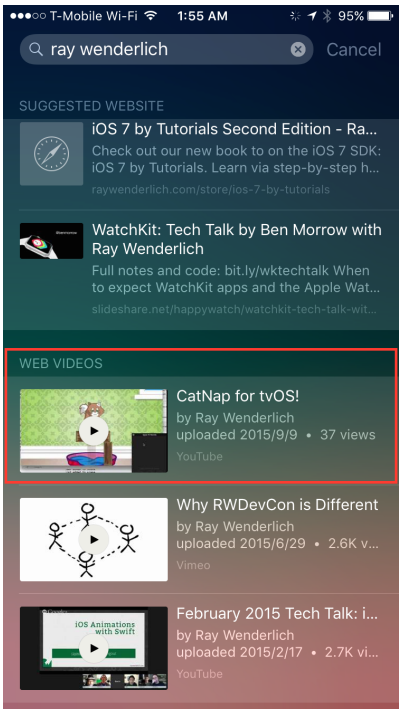
Open **/videos/talk-ray-wenderlich-teamwork.html** and add the following code below the `meta` tag you added earlier:

```
<meta property="og:image" content="http://www.rwdevcon.com/assets/images/
videos/talk-ray-wenderlich-teamwork.jpg" />
<meta property="og:image:secure_url" content="https://www.rwdevcon.com/
assets/images/videos/talk-ray-wenderlich-teamwork.jpg" />

<meta property="og:image:type" content="image/jpeg" />
<meta property="og:image:width" content="640" />
<meta property="og:image:height" content="340" />

<meta property="og:video" content="http://www.rwdevcon.com/videos/Ray-
Wenderlich-Teamwork.mp4" />
<meta property="og:video:secure_url" content="https://www.rwdevcon.com/
videos/Ray-Wenderlich-Teamwork.mp4" />

<meta property="og:video:type" content="video/mp4" />
<meta property="og:video:width" content="1280" />
<meta property="og:video:height" content="720" />

<meta property="og:description" content="Learn how teamwork lets you
dream bigger, through the story of an indie iPhone developer who almost
missed out on the greatest opportunity of his life." />
```

This adds rich web markup to the web page via the `video`, `image` and `description` meta tags, which explicitly points to information contained on the page, helping Applebot find the information it's looking for.

In the `property` fields above, "og" stands for **Open Graph**. To learn more about Open Graph, check out the Open Graph documentation at ogp.me; it's one of several standards Apple supports for structured markup. Other standards include schema.org, RDFA rdfa.info and JSON LD json-ld.org.

The goal of adding rich markup to your web pages is to adorn Spotlight's search results with more information. For example, as this book goes to press, a quick search for "ray wenderlich" comes up with these results:



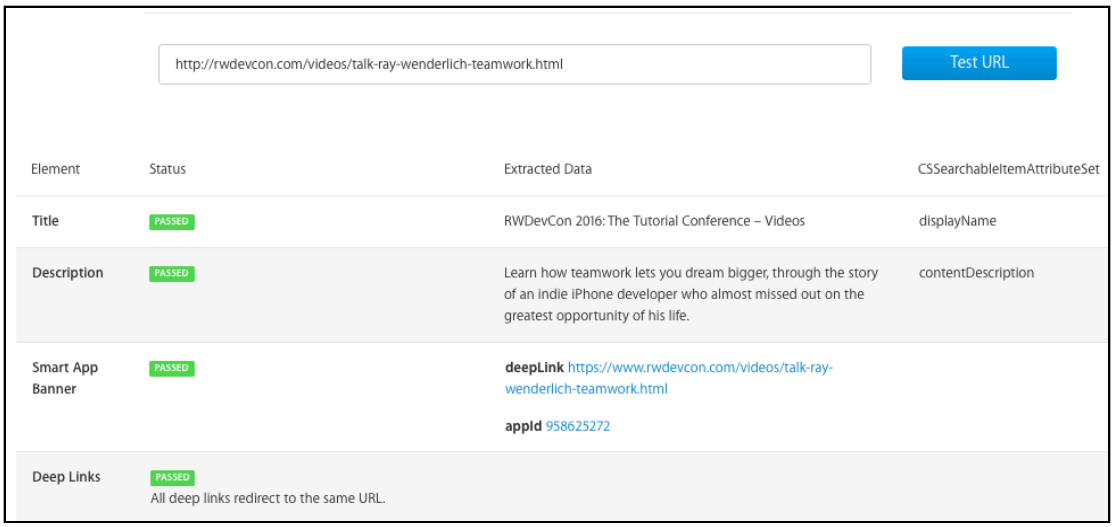
Notice the "CatNap for tvOS" video that Ray recently uploaded to YouTube, marked in red. In addition to the title of the web page, the search result also contains a video thumbnail as well as a description. YouTube was able to achieve this through rich semantic markup.

Validating your markup

Since there's no "compiler" for the web, how are you supposed to know if your web markup is correct? Apple's created a web-based App Search API Validation Tool apple.co/1F8tTGt for just that purpose.

To see the validation tool in action, try it out with the URL of the video page for Ray's 2015 inspiration talk. Simply visit the Validation Tool's web page, enter the video page URL, and click **Test URL**. The tool will provide you with a "report card" of the good parts of your markup, along with things that are missing or need to be improved.

For example, as of this writing, the validation tool returns this set of suggestions for `http://rwdevcon.com/videos/talk-ray-wenderlich-teamwork.html`:



Apple's validation tool checks for the `meta` tags you added earlier, as well as for the page's `title` tag, Smart App Banner and universal link.

Where to go from here?

iOS 9 brings the web and app ecosystems closer than ever before. Apple strongly suggests that you start using universal links as soon as you can to make linking from the web a seamless experience. Furthermore, if you have a website that mirrors your app's content, web markup can help you provide rich search results in Spotlight and Safari.

This chapter covered a lot of ground, but believe it or not you've only dipped your toes into each topic. There are other ways to add rich semantic markup to your sites that you haven't seen yet, such as supported schemas including `InteractionCount`, `Organization` and `SearchAction`. As time goes on, Apple will support more schemas and more ways to markup your web pages to make search results come alive.

Although you weren't able to test your changes as you worked through this chapter, I still hope you found it useful to walk through the steps required to add web markup and universal links to your own apps.

You should definitely check out the following WWDC sessions if you want to find out more about your app and the web:

- Seamless Linking To Your App apple.co/1Way2xz
- Introducing Search APIs apple.co/1LFjZZD
- Your App, Your Website, and Safari apple.co/1OGLhE3

Apple also provides the following excellent programming guides for universal linking

and web markup:

- App Search Programming Guide apple.co/1ip7lGE
- iOS Search API Best Practices and FAQs apple.co/1Mj4yJe

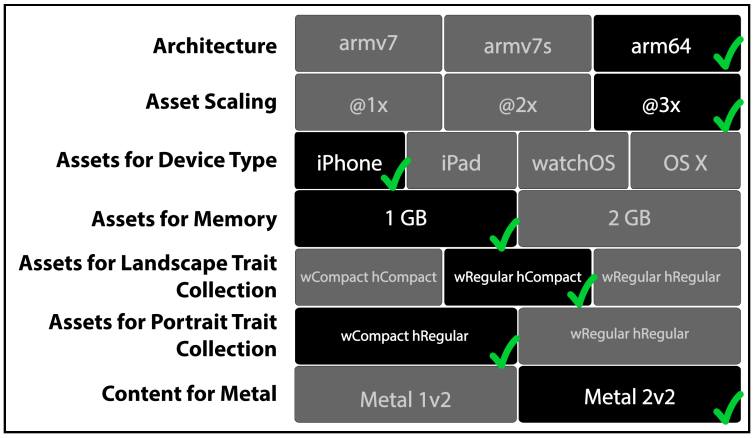
Chapter 4: App Thinning

By Derek Selander

Kicking yourself that you didn't drop the extra dinero to multiply your iOS device's disk storage size by a factor of two? Feeling constrained by how massively huge your brilliant app concepts would be? Well, don't! Apple now takes a more frugal approach to how apps are stored on a device.

With the introduction of iOS 8 and the demanding displays of the iPhone 6 and 6 Plus, Apple began pushing developers to adopt a more universal approach to building across devices. Adaptive Layout, Trait Collections, Universal split view controllers and (a more respectable) Auto Layout led to a seamless experience for the iOS developer to build universal applications for both iPhone & iPad.

However, it also meant that a so-called universal app requires a substantial chunk of device-specific content, and it sure has a huge impact to an app's bundle size. For an example, look at the chart below to see all the 1s and 0s that are stored locally on the device, but are never used unless the app runs on an iPhone 6+.



Fortunately, with the introduction of iOS 9, Apple introduced several solutions to address this problem:

- **App Slicing**: When you submit your iOS 9 binary to the App Store, Apple

compiles resources and executable architecture into variants that are specific to each device. In turn, devices only download the variant specific to their traits — meaning they only get content they will use. Traits include graphics capabilities, memory level, architecture, size classes, screen scaling and more.

- **On Demand Resources**: Application resources are downloaded as needed and can be removed if the device needs room for other resources.
- **Bitcode**: An intermediate representation of your compiled app can be sent when submitting to the App Store. This allows Apple to optimize your executables by compiling with the latest optimizations for a given target, including types that didn't exist when you submitted your app.

Packaged together, these techniques are known as **App Thinning**.

Getting started

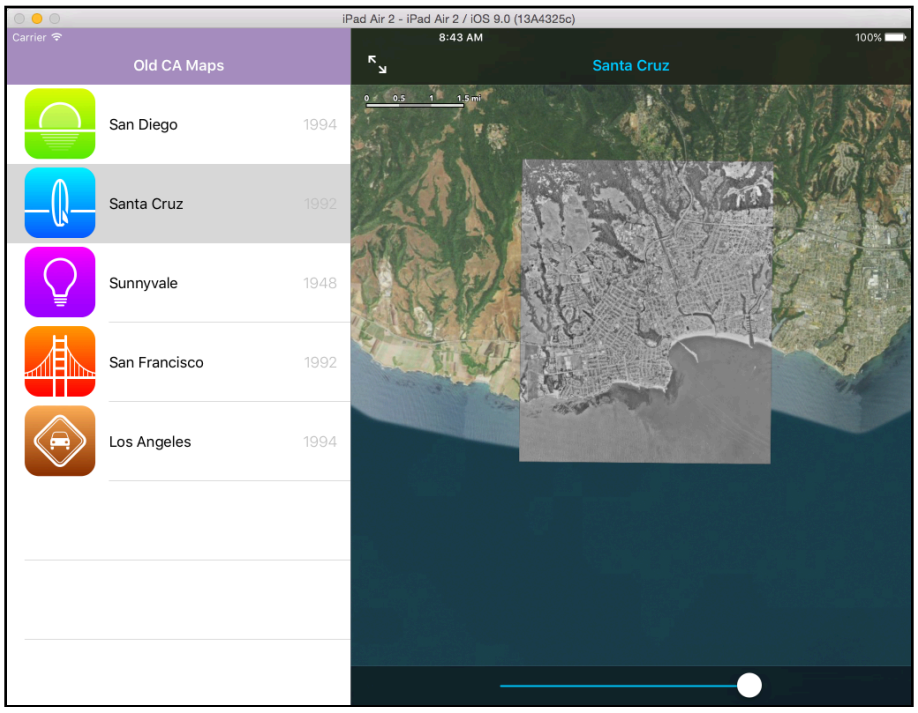
Open the **Old CA Maps** starter project. This application displays historical aerial overlays of different parts of California on a map.

What you have here is a close-to-final project that's about to be fired off to the App Store. But don't do it just yet, because the resources for this seemingly simple app make it a storage hog. It takes up over 200 megabytes!

Before sending it off, you'll use App Thinning techniques to hack-and-slash the end product to a more manageable size. But before you do that, take a tour around the app. It's pretty sweet.

With the Xcode project open, select the **iPad Air 2 Simulator** as the **scheme destination**, and then build and run the application.

Play around with the app for a bit. Tap on **Santa Cruz** and other overlays and see how the historical maps overlay with present-day maps.

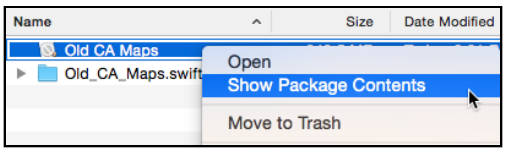


Note: These overlays are created from image tiles that are found in `NSBundle(s)` and passed into a `MKTileOverlayRenderer` for drawing. Curious about what's going on under the hood? Unfortunately, if this chapter explored all of it, it would be impossibly long. Think of this stuff as a black box – all you care about is how to make this app as small as possible for the end user. :]

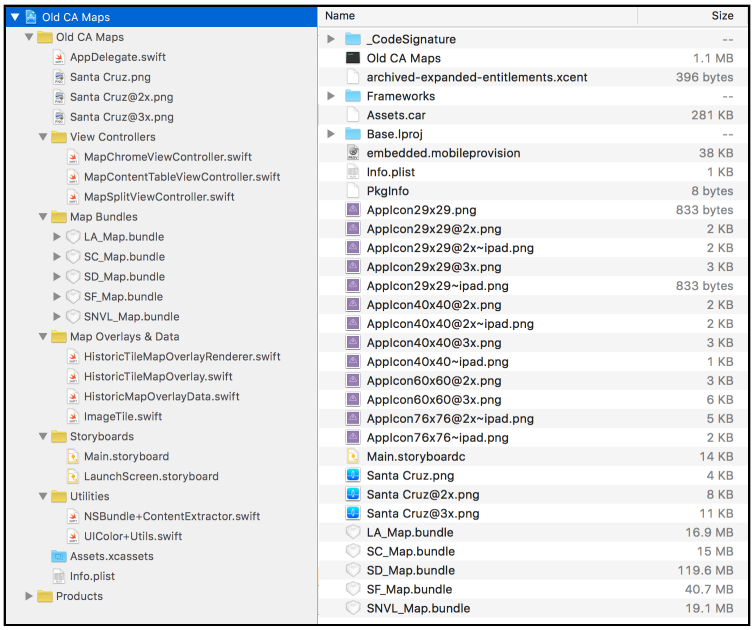
The anatomy of an app

When compiling your code into any iOS application, it's good to understand what Xcode does behind the scenes. So before you start thinning it out, take a few minutes to understand what happens.

This project contains a run script that launches a finder window with the location of the build directory where you'll see an app file — otherwise known as the application bundle. Build and run, and in Finder, right-click **Old CA Maps** and select **Show Package Contents** to view the compiled bundle.



Understanding the content that goes in to your completed application will be useful when working with App-Thinning. Below is a side-by-side comparison of the Old CA Maps Xcode project's directory (on the left) and a release build of Old CA Map's application bundle's contents (on the right). Your output might vary slightly depending on your device type, build configuration and Xcode version.



There are a few important items to note:

- The assets catalog in Xcode named **Assets.xcassets** will become a binary version named **Assets.car** in the application bundle. This file's job is to hold resources specific to different scales, size classes and devices.
- Check out the sizes of each of the bundles. Notice the **SD_Map.bundle** is nearly 120 MB!
- The item called **Old CA Maps** with the terminal icon is the executable for your application. This is the actual program that runs on an iOS device.
- Notice there are three **Santa Cruz PNGs** in the project – but not in a bundle or asset catalog – that did not copy into the **Assets.car** file. Instead, they copied to a top-level directory that won't get sliced! Guess what? You're going to fix that soon...

Measuring your work

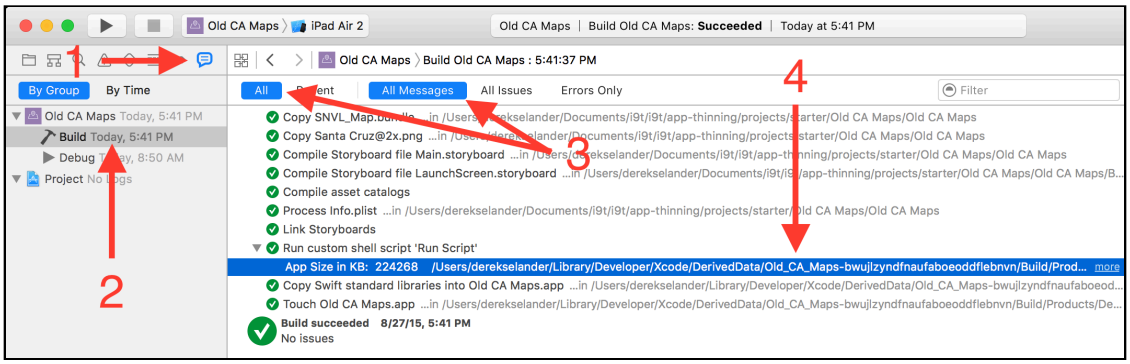
It's always helpful to quantitatively measure your progress when making changes, and working with App Thinning is no exception. Specifically, you'll want to know how big the application bundle is before and after.

Fortunately, there's already a build script in the app package that produces the size of the bundle in kilobytes.

To view the size of a build using this script, first build the project, then:

1. Navigate to the **Report Navigator**.
2. Select the **Build** you wish to inspect.

3. Make sure the **All** and **All Messages** are selected
4. Find the **Run custom shell script** output.



You'll occasionally come back to this output to see your progress as you whittle the app down to size.

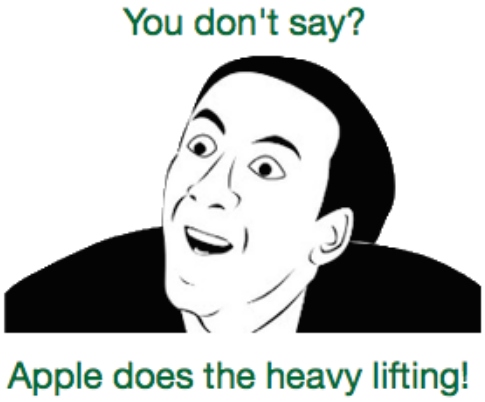
Although it won't show the exact size of the IPA you'll submit to the App Store, it'll give you a good idea of your success.

Slicing up app slicing

App slicing can be broken out into two parts: executable slicing and resource slicing.

Executable slicing simply means Apple delivers a single executable of the appropriate architecture for a given device. You don't need to do much to help the App Store make this happen.

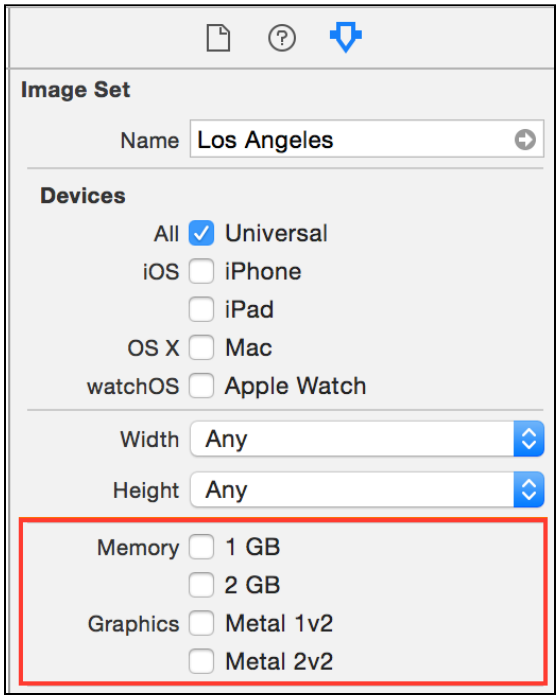
By default, release builds include all architectures configured in your build settings. When you submit such a build to the App Store, it automatically creates the variants needed on your behalf. All *you* have to do is compile for iOS 9.



Being smart with resources

Resource slicing takes a tiny bit more work than doing absolutely nothing...but not by much. :]

All you have to do is make sure your resources are compiled into **Asset Catalogs** and organized according to traits. You probably already organize your image assets according to scale factors. With Xcode 7, you can also tag assets by **Memory** and **Graphics** requirements in the Attributes Inspector.



- The **Memory** setting lets you target different assets to devices with different amounts of RAM.
- The **Graphics** setting allows you to target either first or second generation Metal-capable GPUs.

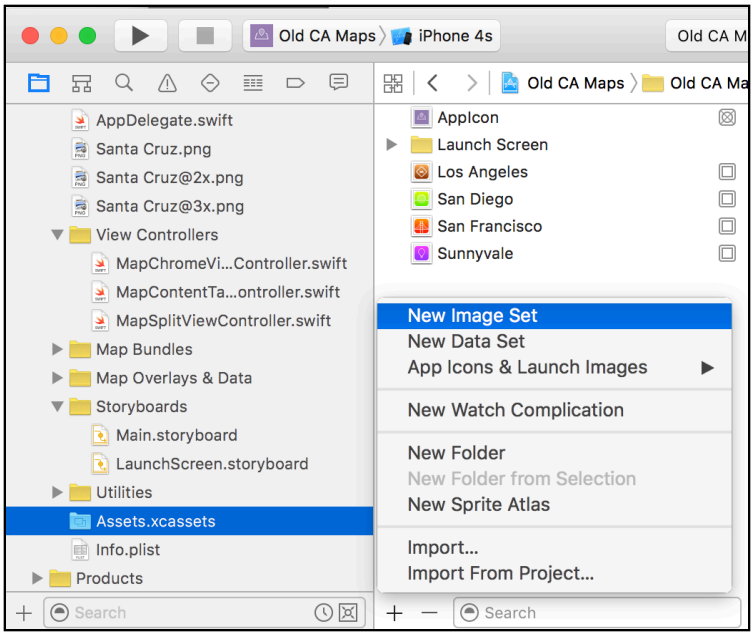
When you make use of these new settings, along with scale factors and device types, the App Store can slice your app into targeted bundles for specific devices.

As you noticed earlier, the Santa Cruz assets are not correctly compiled into the Assets.xcassets catalog within Xcode, resulting in the images being copied over to the main bundle. This means they won't be sliced, so they'll land on devices where they won't be used and tragically, consume storage for no real reason.

Your first fix

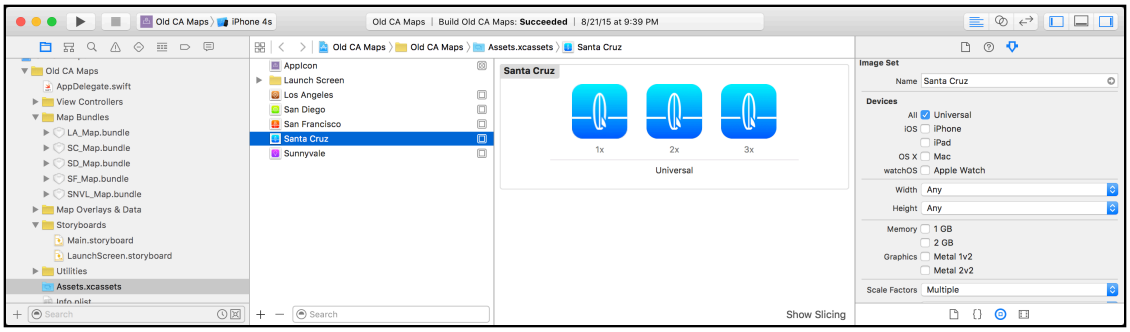
The fix for this is quite simple: Just stick the Santa Cruz PNGs into the asset catalog.

Click **Assets.xcassets**, then click the **+** and select **New Image Set**. Drag the Santa Cruz images from the project navigator into their respective @1x, @2x and @3x slots, and make sure the **Image Set** name is **Santa Cruz**.



Once the images copy to the assets catalog, delete the Santa Cruz PNGs from the resources folder.

After that, make sure the Santa Cruz assets in the catalog looks like:

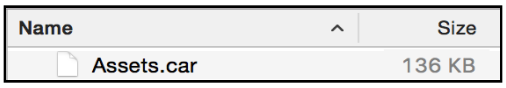


Build and run the application, again selecting the **iPad Air 2 Simulator**. Take a look at the size of **Assets.car** by looking at the package contents in the build directory as you did earlier.

This is using the @2x image for Santa Cruz, and it ends up at 107 KB. You may see a slight difference based on the compiler version you use.

> **Note**: Reviewing a debug build is a great way to see how App Thinning works, and even before App Thinning existed, Xcode was tailoring debug builds to the targeted device. So, App Thinning essentially builds on what Xcode already did, but now, the end user enjoys the benefits.

Now build and run with the **iPhone 6 Plus** simulator and take a look at the size of **Assets.car**:



As you can see, it's up to 144 KB. It makes sense that this build is larger given the higher resolution of @3x images used by the iPhone 6 Plus. While it may not directly reflect the size of the bundle on the store, this gives you a relative idea of how thinning works to size your bundle according to the needs of the target device.

> **Note**: Although PNGs are a good way to provide resources, you should also consider using vector-based PDFs. Xcode breaks down the PDF and resizes the image as needed, essentially future-proofing your app for whatever screen scales Apple comes up with. All the other thumbnail images in Old CA Maps use vector-based PDFs.

Lazily (down)loading content

Now that you've migrated all the images into asset catalogs, essentially removing unused images, it's time to take a more aggressive approach at limiting content by using **On-Demand Resources**, or simply, **ODR**.

ODR allows you to store resources on Apple's servers, and then your app can pull them down as needed.

`NSBundleResourceRequest` is responsible for dealing with ODR. By using this primary class, you can control the content that downloads with the use of **Tags**, which are string names you attach to resources that properly identify a group of content to download.

Through tags, Apple abstracts the remote resource storage location and retrieval URL for your ODR content.

So, what can you include when using ODR? They can be images, data, OpenGL shaders, SpriteKit Particles, Watchkit Complications and more. The main thing to understand is that *ODR can't be executable code*.

Fortunately for this particular application, NSBundles fall into the data file category. This means you can apply ODR to the bundles without changing any of the file

infrastructure within Old CA Maps.

Wire things up to use tags

Time to finally whip out your coding skills.

Navigate to **MapChromeViewController.swift** and hunt down the **downloadAndDisplayMapOverlay()** function. It's here that you'll replace the synchronous loading of a local bundle with an asynchronous load for a remote bundle obtained through a `NSBundleResourceRequest`.

Replace the contents of `downloadAndDisplayMapOverlay()` with the following:

```
// 1
guard let bundleTitle =
  mapOverlayData?.bundleTitle else {
    return
}

// 2
let bundleResource =
NSBundleResourceRequest(tags: [bundleTitle])

// 3
bundleResource.beginAccessingResourcesWithCompletionHandler {
  [weak self] error in

  // 4
  NSOperationQueue.mainQueue().addOperationWithBlock({

    // 5
    if error == nil {
      self?.displayOverlayFromBundle(bundleResource.bundle)
    }
  })
}
```

What's going on in there?

1. Here you're grabbing the `bundleTitle` associated with your `mapOverlayData`, which was already set with an appropriate title in the included **HistoricMapOverlayData.swift**. You're using the `guard` statement, a new feature in Swift 2.0, that lets you maintain the "Golden Path" of code by returning immediately if the unwrap fails.
2. You're instantiating an `NSBundleResourceRequest` with the `bundleTitle` tag associated with your bundle. You'll add tags to all the content bundles shortly.
3. `beginAccessingResourcesWithCompletionHandler(_:)` calls the completion block when your app finishes downloading on-demand content or upon error.
4. The completion handler is not called on the main thread, so you'll need to supply a block running on the main queue to handle any updates to the UI.

5. `NSBundleResourceRequest` has a read-only variable named **bundle**. Replacing `NSBundle.mainBundle()` with this variable makes the code more extensible if you decided to move the file structure of your resources around.

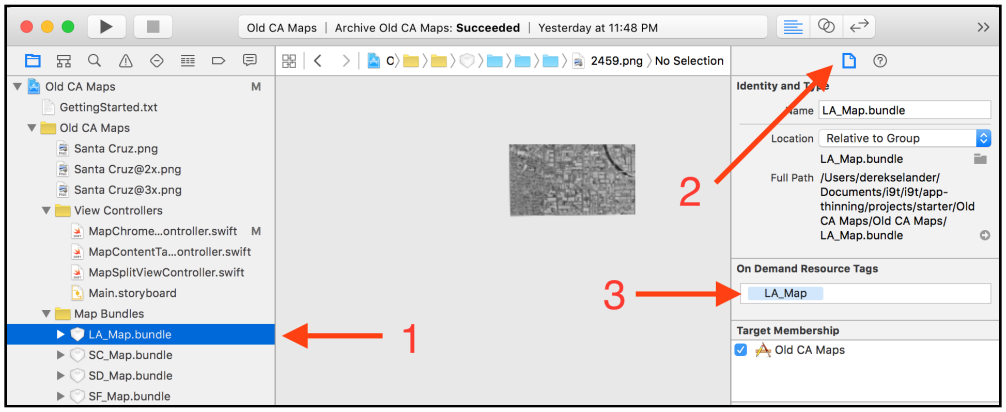
How about those tags?

Build, run, and click on one of the cities.

Whoops! Xcode should fail to load an overlay, and it'll spit out an error in the console. This is because you've told the `NSBundleResourceRequest` to look for tags that don't exist. Time to fix that.

Navigate to the Project Navigator tab and expand the **Map Bundles** group. Select **LA_Map.bundle**, and open Xcode's **File Inspector** tab on the right. Find the **On Demand Resource Tags** section.

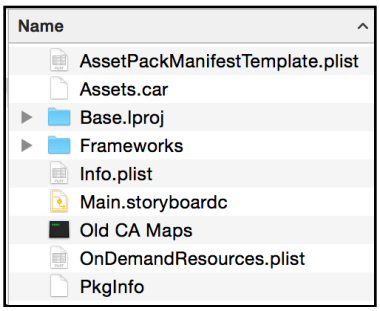
Give **LA_Map.bundle** the tag name **LA_Map**. Now go through the four remaining bundles and give each a tag name identical to the bundle name without the file extension. These will match the names used for the `bundleTitle` that were set in **HistoricMapOverlayData.swift**.



> **Note**: Make sure you spell the tag name with *exactly the same* spelling and case as the bundle file name. If you mistype it, you'll encounter issues.

Build your application for **iPad Air 2**, but don't run it yet, using **Command-B**. Take note of the application bundle size in the report navigator.

Originally, the app was over 200 MB. Now, Old CA Maps is around 10MB. Xcode has achieved this by removing the bundle resources from the main application bundle, which can be confirmed by reviewing its contents:



Now run your application. Select **Los Angeles** as the overlay and observe what happens. The app now downloads content on demand, then displays the overlay and adjusts the map when completed.

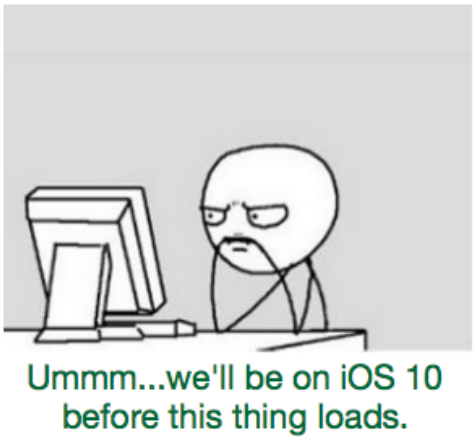
> **Note**: When your app is live in the App Store, it'll download these resources from there. However, to achieve the same effect while developing, Xcode makes a local network request from your device (or simulator) to your computer to download the ODR. This means that if you're testing your application and you turn your computer off, ODR will fail to work. It also means transfer time is significantly less when compared to what a user would see for assets housed on the store.

Make it download faster

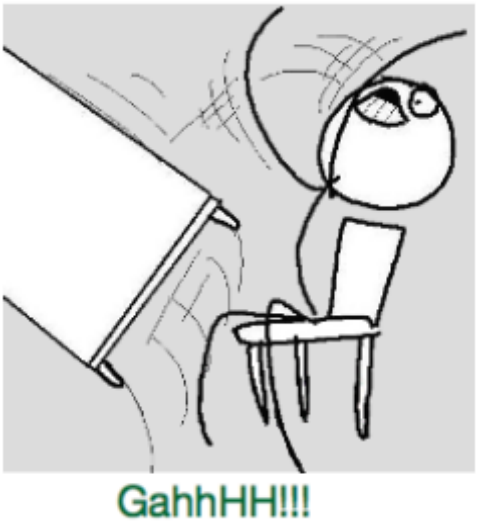
You tested loading Los Angeles, but as you may recall, the Los Angeles bundle asset is small in comparison to the San Diego bundle.

Try clicking on the **San Diego** overlay and see how long it takes to display the content.

> **Note**: If you choose a city that you've already viewed after building and running, you're likely going to notice it loads immediately, because ODR caches the assets until purge conditions are met. You'll learn more about this later.



That took a little bit too long to display, right? Can you imagine how long it'll take for users that pull assets that are hosted on the store?



To avoid a deluge of rotten tomatoes and bad reviews, you'll need to show the user that something is happening while the app downloads content.

Fortunately, `MapChromeViewController` already has an `IBOutlet property` called `loadingProgressView` which is a `UIProgressView`. You'll feed that view progress data to present the user while also displaying the network activity indicator.

Navigate back to **downloadAndDisplayMapOverlay()** and replace the content with the following:

```
guard let bundleTitle =
  mapOverlayData?.bundleTitle else {
    return
}

let bundleResource
  = NSBundleResourceRequest(tags: [bundleTitle])
```

```
// 1
bundleResource.loadingPriority
  = NSBundleResourceRequestLoadingPriorityUrgent

// 2
loadingProgressView.observedProgress
  = bundleResource.progress

// 3
loadingProgressView.hidden = false
UIApplication.sharedApplication()
  .networkActivityIndicatorVisible = true


bundleResource.beginAccessingResourcesWithCompletionHandler {
  [weak self] error in
  NSOperationQueue.mainQueue().addOperationWithBlock({
    // 4
    self?.loadingProgressView.hidden = true
    UIApplication.sharedApplication()
      .networkActivityIndicatorVisible = false

    if error == nil {
      self?.displayOverlayFromBundle(bundleResource.bundle)
    }
  })
}
```

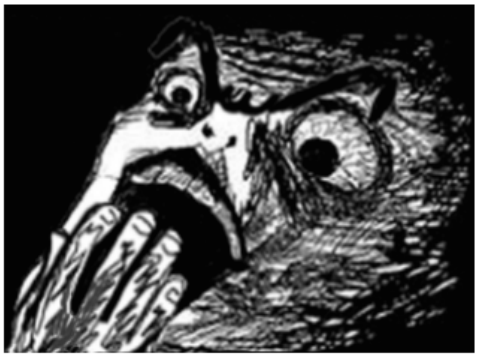
1. This tells the system that the user is "patiently" waiting for this download and the system should be diverting all resources to complete it ASAP.
2. The `loadingProgressView` hooks into the `NSProgress` of the `NSBundleResourceRquest`. It will begin updating automatically once `beginAccessingResourcesWithCompletionHandler(_:)` is kicked off.
3. Display the `loadingProgressView` and also the network activity indicator to inform the user that a network request is in progress.
4. Once the download completes, hide the `loadingProgressView` and network activity indicator. This code could result in unexpected results due to a potential race condition if there are concurrent downloads. However, for this simple implementation, this approach is sufficient.

Build and run the application. Try all the bundles again and you'll notice a progress indicator just below the navbar while a download is in progress.

It's better because at least there's a visual queue that something is happening, but the 120MB San Diego download still takes an eternity. Time to try something a bit more drastic.

The many flavors of tagging

Displaying the progress makes for a better experience, but nobody wants to wait for a download. Keep in mind that you're testing on a controlled device with Simulator and locally hosted resources. Imagine a real-world user moving in and out Wi-Fi or cellular coverage.



The San Diego asset is big and also likely to be the first thing the user selects since it's the first item in the table. It makes sense to include the San Diego asset along with the application itself so it feels snappy on initial use.

At the same time, you need the flexibility to remove this huge overlay asset if the user runs low on disk space.

Initial install tags

The answer to this is **Initial Install Tags**. They work the same as the tags you've used so far, but they download with the app and count towards the size of the IPA.

Open up the **Old CA Maps Project**, click on the Old CA Maps in the **Target** section and then select the **Resource Tags** tab.

Before you make any changes, note that tags come in three type categories that define how ODR handles them.

- **Initial Install Tags:** These install with your application. So why bother to manage them with ODR? Because you can remove it when it's no longer needed. These tags are perfect for resources you'll only need to use at first.
- **Prefetched Tag Order:** These tags download once the application finishes downloading and in the order in which they are arranged.
- **Download Only On Demand:** These resources are the ones you've worked with so far, and they download when you code them to do so.

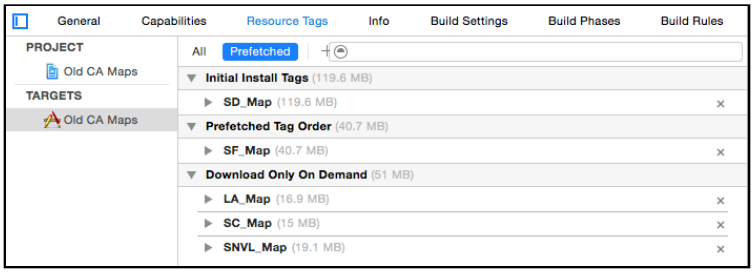
Next, move the San Diego bundle with the **SD_Map** tag from the **Download Only on Demand** section to the **Initial Install Tags** section.

To do this, select the tag and drag it into the Initial Install Tag section.

In addition to having San Diego load with the application, it would make sense to kick off San Francisco sooner than later since it is about 40MB.

Drag **SF_Map** over to the **Prefetched Tag Order section** to trigger downloading as soon as the app is installed.

Once you're done, your tag setup should look like this:



Unfortunately, testing these changes is a bit trickier. You will have to submit your app to **TestFlight Beta Testing** in order to see these changes propegated in your app.

Purging content

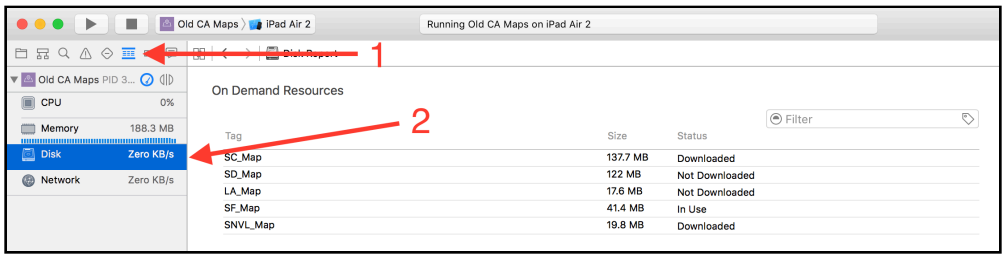
Since the OS can purge ODR content, provided that it's not in active use, it's important to be a good disk storage citizen. You can help guide the system to resources that you don't need anymore, which can decrease your app's footprint.

Change your build scheme to **iPhone 6 Plus**, then build and run the application. Tap on a city, then press the **back arrow** to jump back to the previous city selection screen.

For these particular bundles, it's likely that you no longer need them as soon as you exit the view of a `MapChromeViewController`. But how do you know what the system does with your ODR content behind the scenes?

Fortunately, Apple has anticipated this problem and Xcode 7 ships with a super useful debugging view to aid in understanding the status of your ODR content.

With the app running, open the **Debug navigator** tab (1), then click on the **Disk** cell (2). Xcode will reveal a disk report that includes valuable information regarding the current status of each tagged ODR.



As you can see, after clicking an image then clicking back, this view indicates that the ODR resource is still **In Use** while other resources are either **Not Downloaded** or **Downloaded**.

Although UIKit and Foundation provide their own logic about when the device can reclaim these bundles, it's ideal to tell the system when your app is done with them.

Set a resource to be purged

You'll indicate that the `NSBundleResourceRequest` is available for the system to reclaim as soon as you leave the `MapChromeViewController`.

Open up **MapChromeViewController.swift** and add the following property to the beginning of the class:

```
var overlayBundleResource: NSBundleResourceRequest?
```

Now, in the **downloadAndDisplayMapOverlay()** function, add the following line underneath the `NSBundleResourceRequest` instantiation:

```
overlayBundleResource = bundleResource
```

Finally, add a new method override to `MapChromeViewController` to tell the system that you're done with the resource request when the view disappears:

```
override func viewDidDisappear(animated: Bool) {
  super.viewDidDisappear(animated)
  overlayBundleResource?.endAccessingResources()
}
```

Rebuild and run the system and keep an eye on the **Disk Report** screen while exploring different cities. The report should now indicate that your ODR content is no longer in active use as soon as you leave the map view screen. Sweet.

Where to go from here?

Congratulations, you've learned the in and outs of App Thinning! Remember that the same cellular limits apply for ODR resources, so there are limits on the resource size you can download.

Make sure to thoroughly test your ODR tags in a real-life setting using **TestFlight** before shipping your app off to the App Store.

Also be sure to check out these WWDC videos:

- Introducing On Demand Resources apple.co/1HMTaju
- App Thinning in Xcode apple.co/1Kn8HIA

Now that you've tapped into App Thinning, you should be able to make those big, beautiful apps while being mindful of the user's storage space.

While you might see amazing benefits, like a gajillion more downloads or parades to celebrate your brilliance, you're more likely to see reasonable benefits like fewer uninstalls, more usage and more positive reviews. That should be enough incentive to get you excited about App Thinning.

Chapter 5: Multitasking

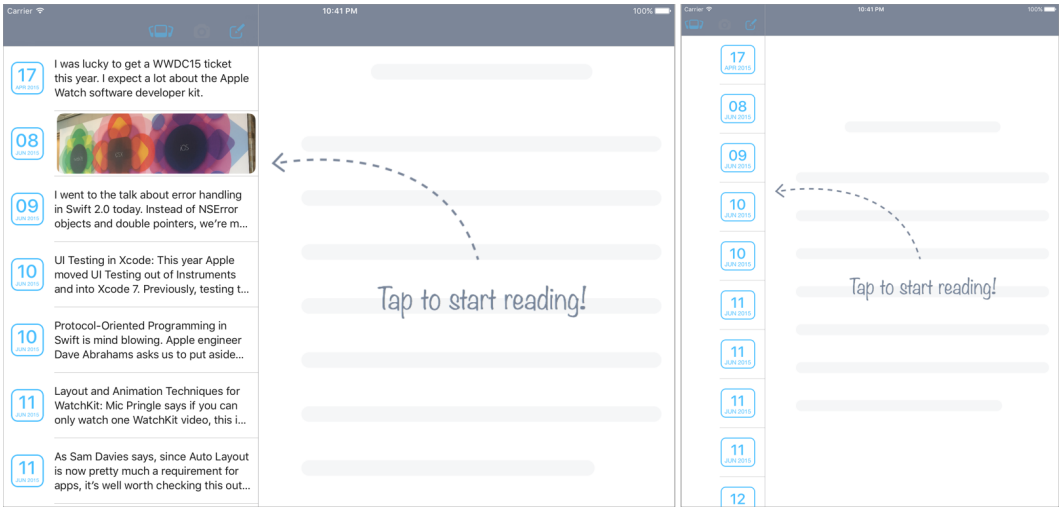
By Soheil Azarpour

iOS 9 introduces a phenomenal feature for the iPad — **multitasking**. For the first time, iPad users can run two apps on the screen at the same time. Maybe you want to read a proposal in your email app while you research the topic in Safari. Or you'd like to keep an eye on Twitter while you enjoy your favorite sports show. For a device that you can hold in one hand, this is a crazy amount of productivity power. It's undoubtedly going to change the way users interact with their iPads.

In this chapter, you'll learn how to update an existing app so that it plays nicely in a multi-tasking iPad environment.

Getting started

The starter project you'll use for this chapter is named **Travelog**. Open the project file in Xcode and build and run the application on the **iPad Air 2** simulator. You'll see the following:



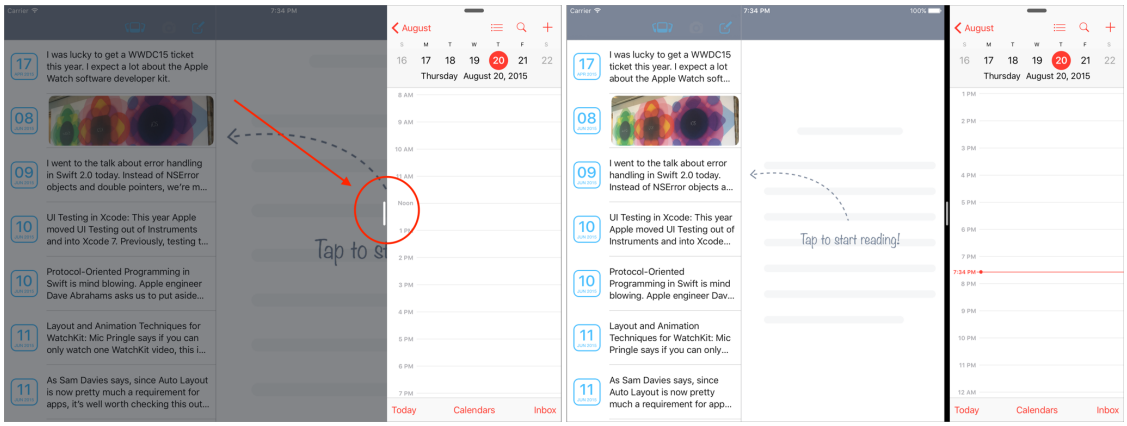
Travelog is a journaling app. The app uses `UISplitViewController` to display entries on the left side. Tap any entry to display it in the right-hand view; rotate the device and you'll find both master and detail views of the Split View Controller are visible in both orientations.

It's time to see how the app behaves in a multitasking environment. Swipe from the right edge of the screen to expose the list of multitasking-ready apps on your iPad. This can be tricky in the simulator; try starting with your mouse pointer just inside the simulator window to simulate a swipe in from the edge.

> **Note**: If the locale of the iPad is set to a region with right-to-left language, swipe from the left edge of the screen to activate multitasking.

Tap on any app to launch it. A small version of the app opens in the previous position of the list. At this point you're in **Slide Over** multitasking mode. Note that Travelog is dimmed out but otherwise unaffected. The app running in Slide Over mode sits on top of Travelog, and a short handle bar sits at top of the Slide Over. Swipe down on the handle to expose the list of multitasking apps and launch a different app in the Slide Over.

You'll notice a handle at the edge of the Slide Over view. Tap it, and you'll see the following:



W00t! The screen just divided in two! Isn't that neat?! This is **Split View** multitasking mode. Travelog is now available for use and resized itself to fit the new, narrower portion of the window.

> **Note**: If an app isn't multitasking ready, it won't appear in the list. Even more reason to get your app ready for multitasking as soon as possible! :]

The **primary app** is the original running app, while the **secondary app** is the newly opened app. If you drag the divider further out, the screen will split 50:50 between the apps. Drag it all the way to the other side and you're back to single

app mode. The primary app is backgrounded at this point.

The final type of multitasking, **Picture in Picture**, or **PIP**, works much like the picture-in-picture function on televisions. You can shrink the PIP window of a FaceTime call to one corner of the iPad and continue using other apps while you chat. PIP is only really applicable to apps that play video; therefore it won't be covered in this chapter.

> **Note:** At the time of writing, Split View is **only** available on the iPad Air 2. Picture in Picture and Slide Over is available on iPad Air, iPad Air 2, iPad Mini 2, and iPad Mini 3.

Preparing your app for multitasking

Here's the good news: if you paid attention at WWDC 2014 and built a universal app with size classes, adaptive layout and a launch storyboard or XIB, you're done! Rebuild your app with the iOS 9 SDK, go grab yourself a beverage and I'll see you in the next chapter!

What's that? You live in the real world and don't *quite* have all the above implemented in your app? Okay then; this chapter is here to walk you through what it takes to make your app multitasking-ready.

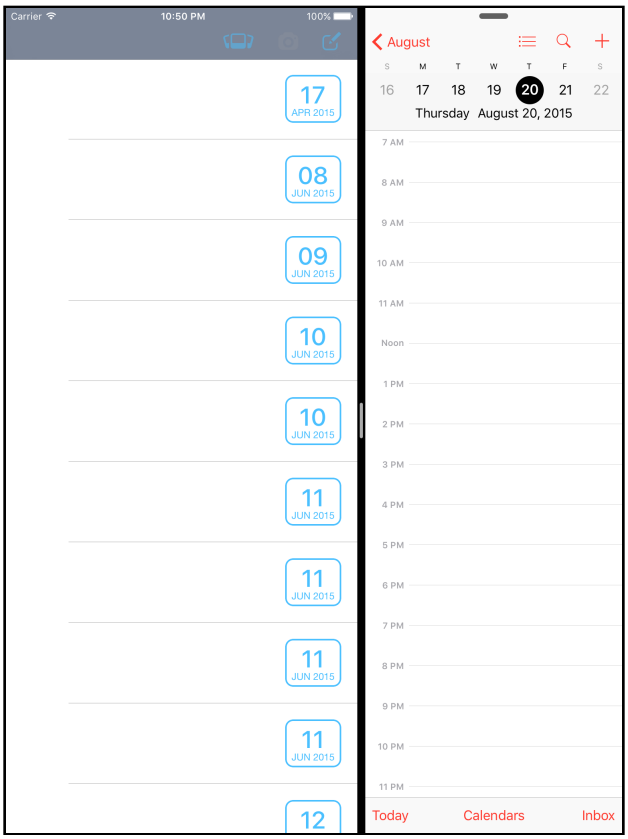
Any new project created in Xcode 7 is automatically multitasking-ready. An existing app you convert to Xcode 7 automatically becomes multitasking-ready if your app:

- Is a universal app
- Is compiled with SDK 9.x
- Supports all orientations
- Uses a launch storyboard

Since all the required criteria are in place, Travelog automatically becomes multitasking ready. That's great news, but just because it's multitasking ready doesn't mean that everything will work as expected. The remainder of this chapter will help you work through common pitfalls encountered when converting existing apps to multitasking apps.

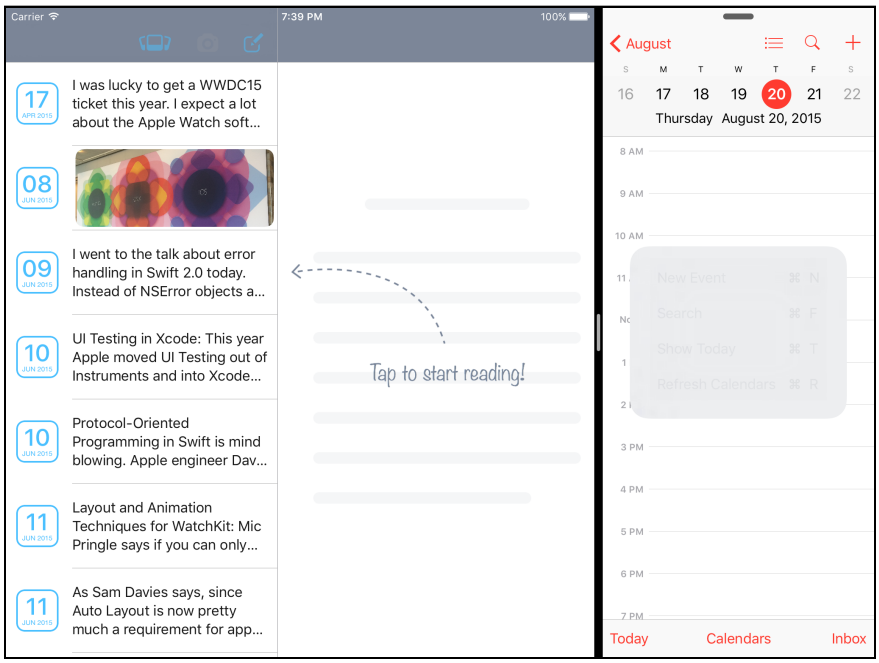
Orientation and size changes

Run Travelog in Split View mode and rotate the iPad to portrait orientation; you'll see the app layout as shown below:



While this layout is functional, it can certainly stand to be improved. There's whitespace wasted on the left hand side and all the labels are squashed over to the right hand side.

Rotate the device to landscape orientation; you'll see the following:



Again, it's functional, but the master view column is too narrow and the text inside the table view cells doesn't really provide any value.

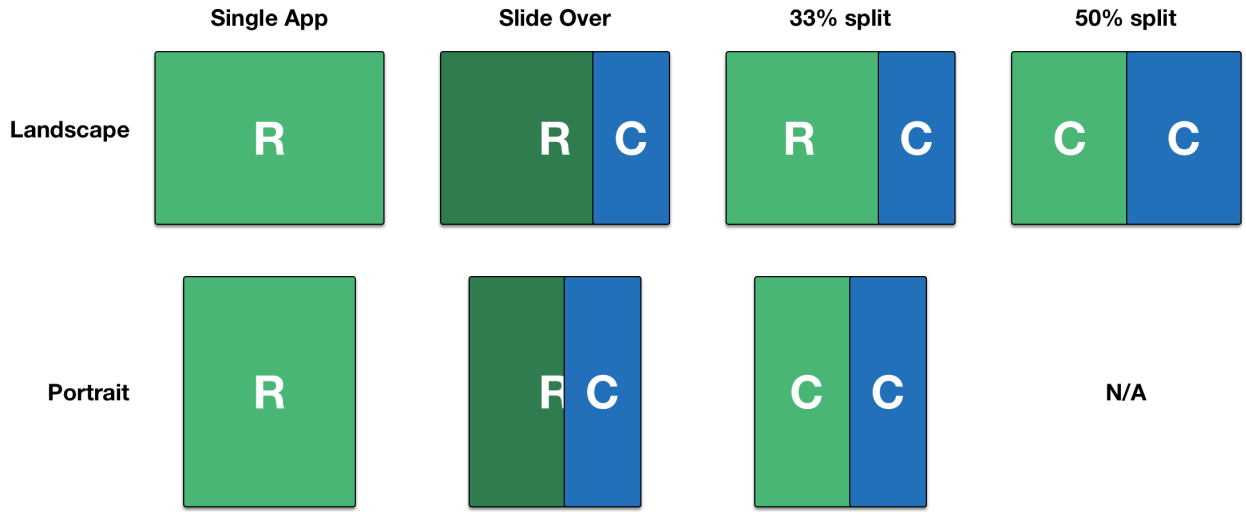
The app already performs some layout updates on orientation change; that seems like the best place to start.

Open **SplitViewController.swift**; this is a subclass of `UISplitViewController` and overrides `viewDidLayoutSubviews()` so it can update the maximum width of primary column via helper method `updateMaximumPrimaryColumnWidth()`. The implementation of `updateMaximumPrimaryColumnWidth()` checks the status bar orientation to determine what the maximum width should be. This approach won't work any longer, since the app can still have a narrow window in split view mode when it's in landscape orientation.

UIKit provides a number of anchor points where you can hook in and update your layout:

1. **willTransitionToTraitCollection(_:, withTransitionCoordinator:)**
2. **viewWillTransitionToSize(_:, withTransitionCoordinator:)**
3. **traitCollectionDidChange(_:):**

The diagram below shows how the horizontal size classes of your app change during multitasking events (**R** means **Regular** and **C** means **Compact**):



Not all multitasking or orientation changes trigger a size class change, so you can't simply rely on size class changes to provide the best user experience.

It looks like `viewWillTransitionToSize(_:, withTransitionCoordinator:)` is a good candidate for an update. Remove `viewDidLayoutSubviews()` and `updateMaximumPrimaryColumnWidth()` from **SplitViewController.swift** and add the following:

```
func updateMaximumPrimaryColumnWidthBasedOnSize(size: CGSize) {
  if size.width < UIScreen.mainScreen().bounds.width
    || size.width < size.height {

    maximumPrimaryColumnWidth = 170.0
  } else {
    maximumPrimaryColumnWidth =
      UISplitViewControllerAutomaticDimension
  }
}
```

This helper method updates the split view's maximum primary column width; it returns the smaller version when the split view is narrower than the screen, such as in a multitasking situation, or when the split view itself has a portrait orientation.

You'll need to call this helper method when the view is first loaded, so add the following:

```
override func viewDidLoad() {
  super.viewDidLoad()
  updateMaximumPrimaryColumnWidthBasedOnSize(view.bounds.size)
}
```

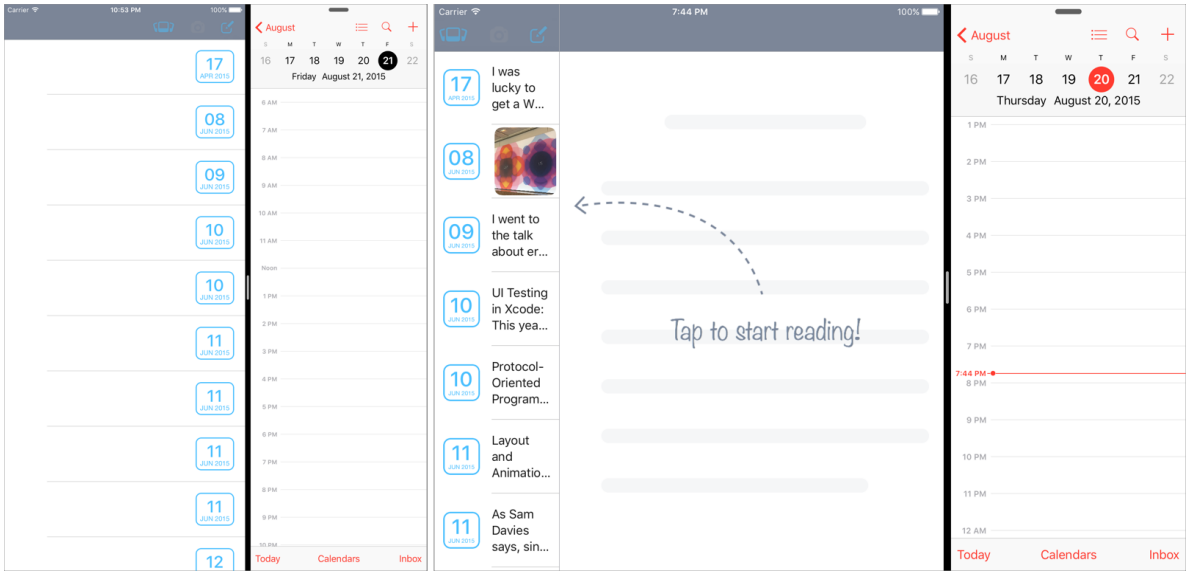
This ensures that the split view starts in the right configuration.

Add one final method:

```
override func viewWillTransitionToSize(size: CGSize,
  withTransitionCoordinator coordinator:
  UIViewControllerTransitionCoordinator) {

  super.viewWillTransitionToSize(size,
    withTransitionCoordinator: coordinator)
  updateMaximumPrimaryColumnWidthBasedOnSize(size)
}
```

This method updates the primary column when the size changes.

Build and run your app; first verify for all orientations that the app still looks and behaves as it did before multitasking. Then bring in another app in Split View and try some different orientations:



Hmm — it's certainly not fixed. It even looks more broken now: with multitasking enabled in landscape orientation, the master column view has been jacked up! It looks like the table view cell doesn't adapt to size changes appropriately.

Open **LogCell.swift** and find the implementation of `layoutSubviews()`; you'll see the code checks for `UIScreen.mainScreen().bounds.width` to determine whether it should use the compact view or regular view.

`UIScreen` always represents the *entire* screen, regardless of the multitasking state. However, you can't rely on screen sizes alone anymore. Update the implementation of `layoutSubviews()` as follows:

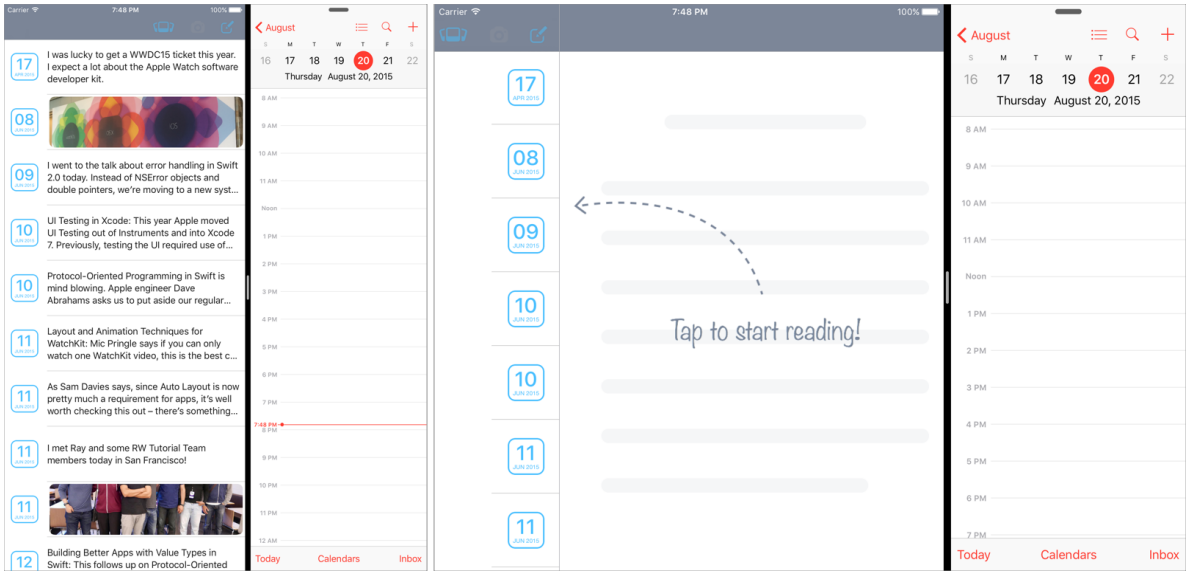
```
override func layoutSubviews() {
  super.layoutSubviews()
  let isTooNarrow = bounds.width <= LogCell.widthThreshold
  // some code ...
}
```

Also update `widthThreshold`, declared at the beginning of `LogCell`, as follows:

```
static let widthThreshold: CGFloat = 180.0
```

The updated code checks the width of the cell itself instead of the width of the screen. This decouples the view's behavior from that of its superview. Adaptivity is now self-contained! :]

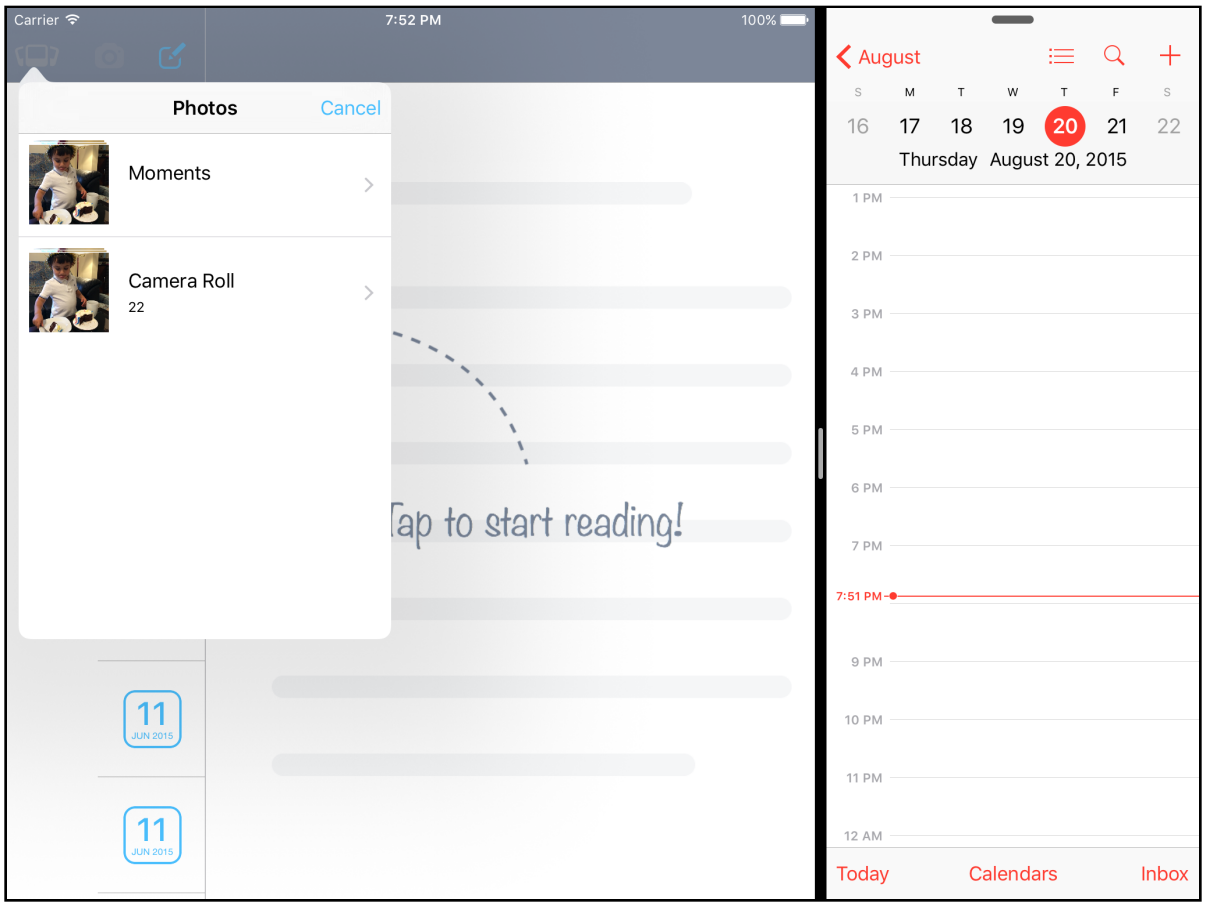
Build and run; again, verify the app still looks and behaves as it did before multitasking. This time around, Split View mode should play nicely in all orientations:



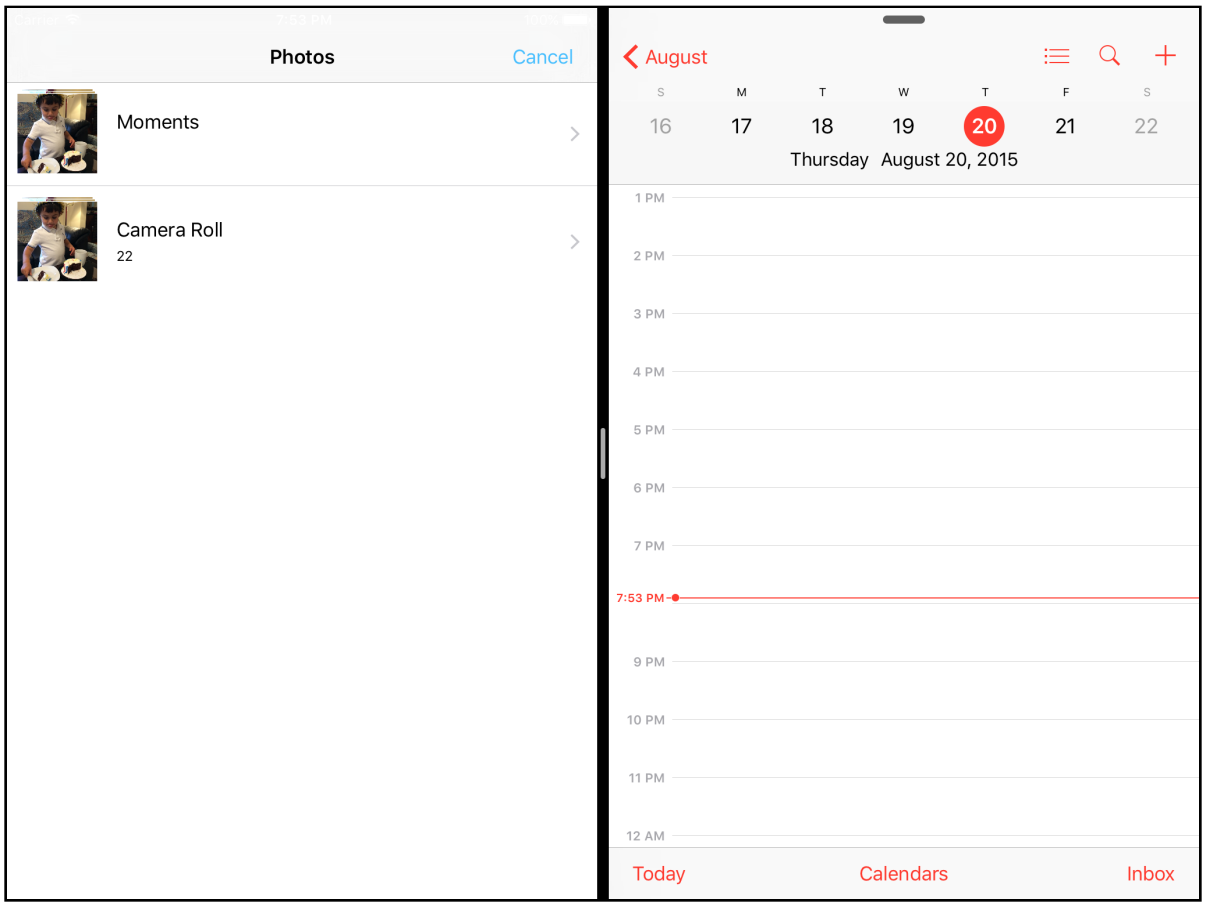
Note: Unlike `UIScreen`, `UIWindow.bounds` always corresponds to the actual size of your app and its origin is always `(0, 0)`. In iOS 9 you can create a new instance of `UIWindow` *without* passing a frame via `let window = UIWindow()`. The system will automatically give it a frame that matches your application's frame.

Adaptive presentation

Continue your evaluation of the app: this time with the device in landscape orientation and the Split View at 33%, tap the **Photo Library** bar button. You'll see the following popover:



With the popover still visible, drag the divider further to the left so the screen is evenly divided between the two apps:



The popover automatically turned into a modal view without any action on your part; dragging the divider to 50% changes the horizontal size class of the app from regular to compact. That's neat, but it's not quite the functionality you're looking for.

Instead, you only want to present the Photo Library in a modal fashion when the app is in Slide Over mode, or when it's the secondary (smaller) app in the 33% Split View mode. When your app is full screen or has 50% width, you'd prefer to present the Photo Library in a popover.

iOS 8 introduced `UIPopoverPresentationController` to manage the display of the content in a popover; you use it along with the `UIModalPresentationPopover` presentation style to present popovers. However, you can intercept the presentation and customize it with `UIPopoverPresentationControllerDelegate` callbacks.

Open **LogsViewController.swift** and add the following class extension to the end of the file:

```
extension LogsViewController:
  UIPopoverPresentationControllerDelegate {

  func adaptivePresentationStyleForPresentationController(
    controller: UIPresentationController,
```

```
      traitCollection: UITraitCollection)
      -> UIModalPresentationStyle {

      //1
      guard traitCollection.userInterfaceIdiom == .Pad else {
        return .FullScreen
      }

      if splitViewController?.view.bounds.width > 320 {
        return .None
      } else {
        return .FullScreen
      }
    }
  }
```

Here's a breakdown of the code:

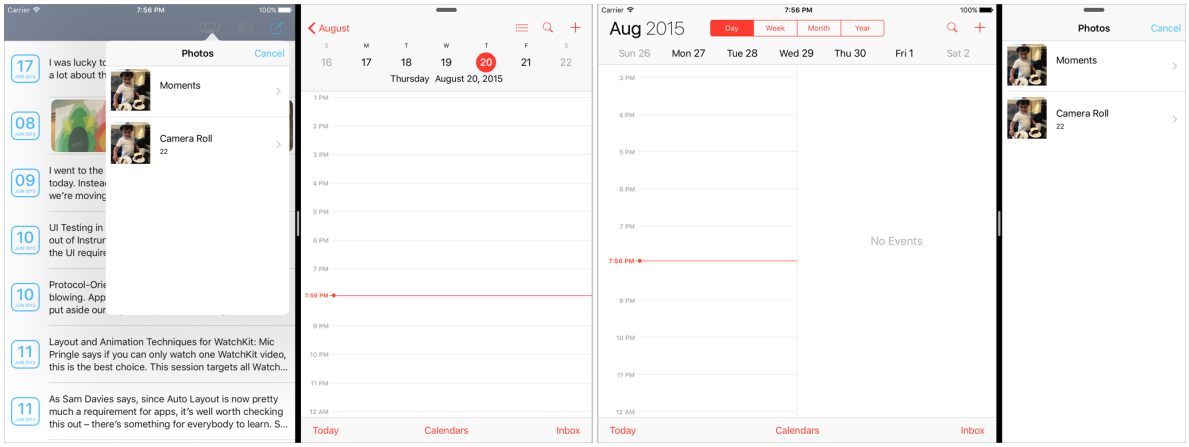
1. Check that the app is running on an iPad; the photo picker should always be presented modally on the iPhone.
2. Check that the split view controller is larger than 320 points — the size of the Slide Over / 33% view. If so, return `.None` to retain the popover, otherwise return `.FullScreen` for a modal presentation instead.

Now you can make `LogsViewController` a delegate of the popover presentation controller.

Find the implementation of `presentImagePickerControllerWithSourceType(_:)`. Read through the implementation and you'll see that when the source type is `.PhotoLibrary`, `UIImagePickerController` presents as a popover. Update the implementation by adding `presenter?.delegate = self` as shown below:

```
  func presentImagePickerControllerWithSourceType(sourceType:
    UIImagePickerControllerSourceType) {
    // some code...
    if sourceType ==
      UIImagePickerControllerSourceType.PhotoLibrary {
      // some code...
      presenter?.delegate = self
    }
    // some code...
  }
```

Build and run your app; verify that the popover transitions to a modal fullscreen view only when your app is in the Slide Over mode or when the Split View pane is sufficiently narrow.



The path to adaptivity

If you're not already using Auto Layout, Size Classes or other excellent responsive layout tools in UIKit, you should definitely consider upgrading your code to do so. UIKit has some new functionality to further assist you with multitasking, including `UIStackView`, `UIView.readableContentGuide` and `UITableView.cellLayoutMarginsFollowReadableWidth`. Want more information on this functionality? Chapter 7, "UIStackView & Auto Layout Changes" has you covered.

Other considerations

Beyond what's been covered in this chapter, there are a few other things to look out for when multitasking. You should already have incorporated most of these suggestions into your existing apps, but the sections below highlight a few extra considerations to be made in the new paradigm of multitasking apps.

Keyboard

Dealing with keyboard presentation has always been an "interesting" topic in iOS. :] You've probably had to adjust the layout of your view so critical elements weren't covered when the keyboard appeared, or perhaps you had to shuffle things around to give the keyboard enough room. In the multitasking world, you need to anticipate that the keyboard can appear at any time — and over any view controller.

Apps running next to yours may present the keyboard, which means you'll have to adjust the layout of your app in a way that a user can still effectively work with it — or you'll risk getting one-star reviews in the App Store! Judicious use of scroll views and/or table view controllers that automatically adjust for the keyboard will help you out here.

Designs

Above and beyond coding considerations, you'll need to change your approach to

app visual design a little differently:

- **Be flexible:** Step away from a pixel-perfect design for various platforms and orientations. You need to think about different sizes and how you can have a flexible app that responds appropriately to size changes.
- **Use Auto Layout:** Remove hardcoded sizes or custom code that resizes elements. It's time to consider Auto Layout and make your code more flexible and future-proof.
- **Use size classes:** One single layout won't always fit all displays. Use size classes to build a base layout and then customize each specific size class based on individual needs. But don’t treat each size class as a completely separate design; as you saw in this chapter, your app should easily transition from one size class to another, and you don't want to surprise your user with a dramatic change as they drag the divider.

Resources

You've worked hard to be a good memory citizen over the years, and that won't change with multitasking. You can potentially have up to three apps running at full speed, all at the same time: the primary app, the secondary app and Picture in Picture. Instruments is an invaluable tool for monitoring memory usage in your app and can help you whittle your memory usage back to the bare minimum.

Where to go from here?

This chapter only touched on the basics of multitasking — it's up to developers like you to help chart the course for accepted multitasking design patterns of the future. To help you along the journey to multitasking, here are some resources you can bookmark for future reference:

- Adopting Multitasking Enhancements on iPad– apple.co/1MdssbK
- Getting Started with Multitasking on iPad in iOS 9 (Session 205) – apple.co/1ItxCtH
- Multitasking Essentials for Media-Based Apps on iPad in iOS 9 (session 211) – apple.co/1hm8v5s
- Optimizing Your App for Multitasking on iPad in iOS 9 (Session 212) – apple.co/1T8CCcp

Chapter 6: 3D Touch

By James Frost

With the release of the iPhone 6s and 6s Plus, Apple surprised everybody with the addition of a feature that could very well redefine the way users interact with their devices: 3D Touch.

An extension of the Force Touch technology, 3D Touch builds on the same theme as Apple Watch and the sleek new MacBook trackpads. It's such an elegant, simple interface that you almost have to wonder what took so long.

How it works: When you're using an iPhone 6s or 6s Plus, you simply press "deeper" into the display to trigger a range of extra functionality. It's more than clever coding though. At the hardware level, advanced sensors detect the *microscopic* changes in distance between the iPhone's cover glass and its backlight.

3D Touch enables new ways to quickly preview content, smooth access to multitasking, and it can even be used to turn your iPhone's keyboard into a trackpad!

You're probably used to Apple dangling sweet new features in front of you without letting you play, almost as if to tease you. 3D Touch is different though—third-party developers get to utilize it at launch-time through a set of three APIs:

- `UITouch` now has a `force` property that tells you how hard the user is pressing.
- `UIViewController` has been extended with a set of APIs that allow you to present a preview of a new view controller — a **peek** — when the user presses on a specified view, and then **pop** it open to display the full monty after a deeper press.
- `UIApplicationShortcutItem` is a new class you can use to add quick actions to your application's home screen icon.

You'll implement each of these APIs in turn to add an extra dimension to a sample app. Strap in, it's going to be a multi-dimensional ride!

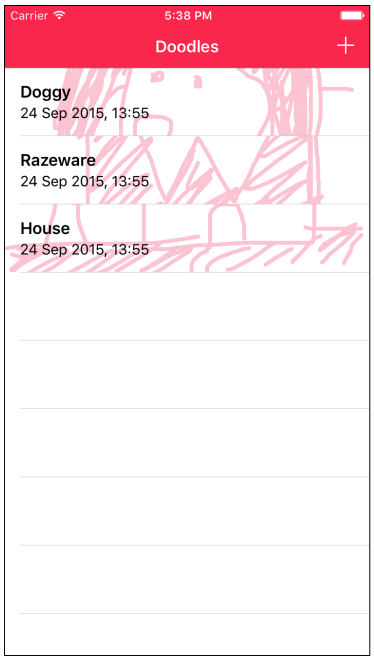
Note: When iOS 9 was first released, there was no way of testing 3D Touch in the simulator, and you had to use an actual iPhone 6s or 6s Plus. However, with Xcode 7.3 you can now use 3D Touch in the simulator if you're using a trackpad that has Force Touch capability.

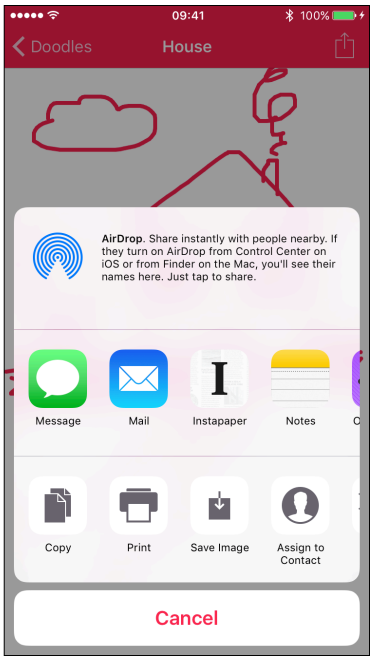
We'd still recommend that you use a device for testing where possible, as the drawing functionality you'll be adding to the app works much better on a real device. Unfortunately, unless you have access to an iPhone 6s, 6s Plus, or a trackpad with Force Touch, you won't be able to experience the full functionality of the demo app for yourself. But who can resist the temptation of a new device? You now have the perfect excuse to go get a shiny new iPhone! Don't worry, we'll wait right here. :]

Getting started

In this chapter, you'll add 3D Touch to a cool little sketching app called Doodles that lets you draw, save and share simple sketches.

Find the starter files for this chapter and open up **Doodles.xcodeproj**. Build and run to see it in action.





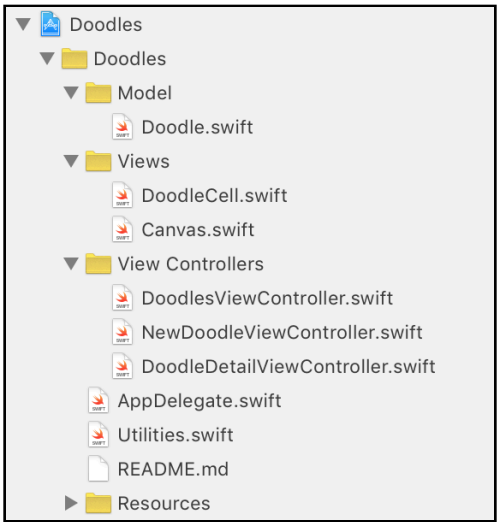
The first screen shows a list of saved doodles. The app comes with a few sample doodles, showcasing some incredible artistic skills – yeah, yeah, don't give up my day job, I know.

Tap on any row to see the full sketch in all its glory. When you're viewing a doodle, you can tap the share button to bring up the share sheet.

Head back to the list view, and tap the **+** button in the top right to create masterpieces of your own. Simply sketch in the blank area and tap **Save** when you're done.

Because this app is just a demo and not intended to be your new sketchpad, the sample app won't save your doodles between launches. So go ahead, draw *anything* you want.

The code behind the app is pretty straightforward. Take a look at the project in Xcode to familiarize yourself.



Here are the highlights:

- **Doodle.swift** contains the model for a doodle, including the sample content.
- **Canvas.swift** contains the custom `UIView` that allows you to draw your doodle on the screen.
- **DoodlesViewController.swift** contains the view controller that displays a list of all of your doodles.
- **DoodleDetailViewController.swift** contains the view controller that displays a single doodle.
- **NewDoodleViewController.swift** contains the view controller in which you create a new doodle.

UITouch force

It'd be great if you could add some more artistic flair to your drawings, and 3D Touch opens up a range of possibilities to satisfy your inner Van Gogh.

`UITouch` has a new `force` property. It's a `CGFloat` ranging from 0 to the value of

`UITouch`'s other new property `maximumPossibleForce`. A value of 1.0 represents the force of an average touch, and `maximumPossibleForce` has a value that ensures that the `force` property has a wide dynamic range.

You'll use the force — of the user's touch, that is — to make your sketches pressure sensitive. The harder you press, the thicker the line will be!

In Xcode, open up **Canvas.swift**. Find `addLineFromPoint(_:toPoint:)`. This method adds a line to the drawing based on the points passed into it.

Change the method definition to:

```
private func addLineFromPoint(from: CGPoint,
  toPoint: CGPoint, withForce force: CGFloat = 1) {
```

Here you're adding a `force` parameter with a default value of 1. In the body of this method, find the following line:

```
CGContextSetLineWidth(cxt, strokeWidth)
```

And replace it with:

```
CGContextSetLineWidth(cxt, force * strokeWidth)
```

Now the `force` parameter is multiplied by `strokeWidth` to make the line thicker or thinner depending on the force. At the moment a default of 1 is all that's being passed, so you need to change that.

Find `touchesMoved(_:withEvent:)`, and replace the call to `addLineFromPoint(_:toPoint:)` with the following code:

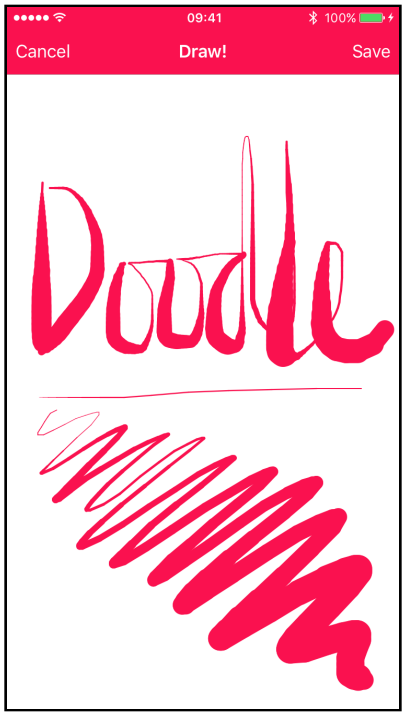
```
if traitCollection.forceTouchCapability == .Available {
  addLineFromPoint(touch.previousLocationInView(self),
    toPoint: touch.locationInView(self), withForce: touch.force)
} else {
  addLineFromPoint(touch.previousLocationInView(self),
    toPoint: touch.locationInView(self))
}
```

Here you're checking to see if Force Touch is available (the internal name for 3D Touch, and the name used for the related technology on Apple Watch), and if so, passing the touch's force into the method you just modified. If force touch is unavailable, the value of the `force` property is `0`, rather than the `1` you might expect.

Build and run the app, either on a device that supports 3D Touch or in the simulator if you have a Force Touch trackpad. If you're using the simulator, you'll need to select **Use Trackpad Force for 3D touch** from the **Hardware** menu to enable the 3D Touch functionality.

Tap the **+** button in the top right, and start sketching! You should see the width of

the line change as you vary the pressure you apply to the screen.



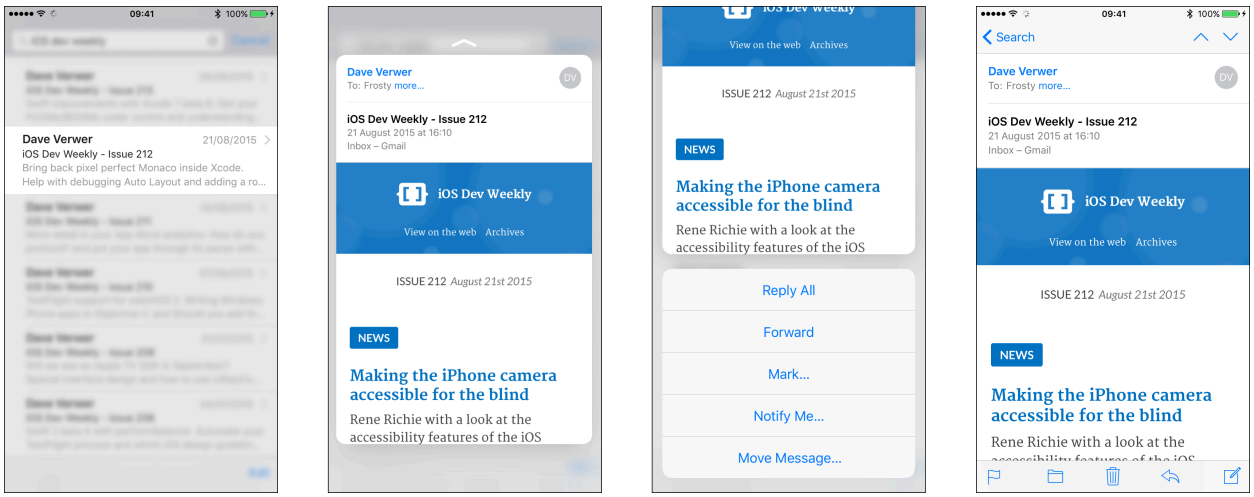
Peeking and popping

Next up, it's time to take a *peek* at the new `UIViewController` preview APIs.

On a 3D Touch enabled device, your view controllers can now respond to different touch pressures. From the users' perspective, a couple of things happen as they press more deeply:

1. First, if the view they're pressing has a preview available, it'll stay in focus while the rest of the view controller's views begin to blur.
2. Then, as the user presses deeper, a preview of the selected content pops up in the center of the screen. This is called a '*peek*'. If the user lifts their finger now, the peek will be dismissed. Alternatively, if the user swipes upwards, the preview can present a number of preview actions – typically actions like delete or share. Finally, if the user presses even more deeply, then...
3. **Pop!** The preview will open and the user will be navigated to the full content.

Here's an example of each of these stages in the built-in Mail app:



You'll use the peek and pop APIs to let users preview doodles from the list. By the time you're finished, the user will be able to lightly press a drawing in the doodles list to see a preview, and then press deeper to pop it open.

Open up **DoodlesViewController.swift** and add the following to the end of `viewDidLoad()`:

```
if traitCollection.forceTouchCapability == .Available {
  registerForPreviewingWithDelegate(self, sourceView: view)
}
```

Just as you did when using `force` from `UITouch`, you first check whether 3D Touch is actually available for use on this device. Then you call the new `registerForPreviewingWithDelegate(_:sourceView:)` method on the `UIViewController` subclass. `sourceView` is the view that will respond to 3D Touch events, so you set it to the view controller's entire view.

But there's one problem — this view controller doesn't implement `UIViewControllerPreviewingDelegate`. So, you need to add an extension to the end of **DoodlesViewController.swift** to make `DoodlesViewController` conform with the protocol:

```
extension DoodlesViewController:
  UIViewControllerPreviewingDelegate {

  func previewingContext(
    previewingContext: UIViewControllerPreviewing,
    viewControllerForLocation location: CGPoint)
    -> UIViewController? {

    // peek!
    return nil
  }

  func previewingContext(
    previewingContext: UIViewControllerPreviewing,
    commitViewController viewControllerToCommit:
```

```
    UIViewController) {

    // pop!
  }
}
```

A `UIViewControllerPreviewingDelegate` must implement both of these methods.

The first, `previewingContext(_:viewControllerForLocation:)` is called when the user initiates a *peek* action, and it gives the delegate an opportunity to return a view controller that contains a preview of relevant content.

This method has two parameters:

- `location` is the location within the view where the 3D Touch is occurring. You might want to use this to determine which part of a view is being touched.
- `previewingContext` is an instance of `UIViewControllerPreviewing`, which has properties for `sourceView` and `sourceRect`. The `sourceView` is the same one you passed into `registerForPreviewingWithDelegate(_:sourceView:)` earlier, while `sourceRect` defines the area within the source view that stays focus during the initial phase of the peek.

The second method, `previewingContext(_:commitViewController:)`, is called when the user presses even deeper and initiates a *pop* action. Here the delegate should present the full content of the popped item.

The `viewControllerToCommit` parameter that's passed in is the preview view controller that was previously returned from `previewingContext(_: viewControllerForLocation:)`. Usually, you'd simply present this same view controller modally or by pushing it onto a navigation controller.

To add peek functionality to Doodles, replace the stub implementation for `previewingContext(_: viewControllerForLocation:)` with the following:

```
func previewingContext(
  previewingContext: UIViewControllerPreviewing,
  viewControllerForLocation location: CGPoint)
  -> UIViewController? {

  // 1
  guard let indexPath =
    tableView.indexPathForRowAtPoint(location),
    cell = tableView
      .cellForRowAtIndexPath(indexPath) as? DoodleCell
    else { return nil }

  // 2
  let identifier = "DoodleDetailViewController"
  guard let detailVC = storyboard?
    .instantiateViewControllerWithIdentifier(identifier)
    as? DoodleDetailViewController else { return nil }
```

```
    detailVC.doodle = cell.doodle

    // 3
    previewingContext.sourceRect = cell.frame

    // 4
    return detailVC
  }
```

Let's take this line by line:

1. You want to show a preview for a specific table row when the user presses on it. So, first you use `location` to check whether there's a row at that position. If there is, then you get the associated cell from the table view.
2. You then instantiate a `DoodleDetailViewController` and set it to display the doodle for the selected cell.
3. You set `sourceRect` to the frame of the selected cell, so that the cell will stay in focus and the rest of the table view will blur.
4. Finally, you return the `DoodleDetailViewController` that represents the peeked content.

> **Note:** The preview view controller will display at a default size, but if you want to display it at a different size in your own app, simply override `preferredContentSize` for the preview view controller.

That's all it takes to implement a peek! When the user lightly presses on a row in **DoodlesViewController**, the content except for the pressed row will blur out, indicating that a peek is available. If the user presses a little deeper, an instance of **DoodleDetailViewController** will present as a preview and show the selected doodle.

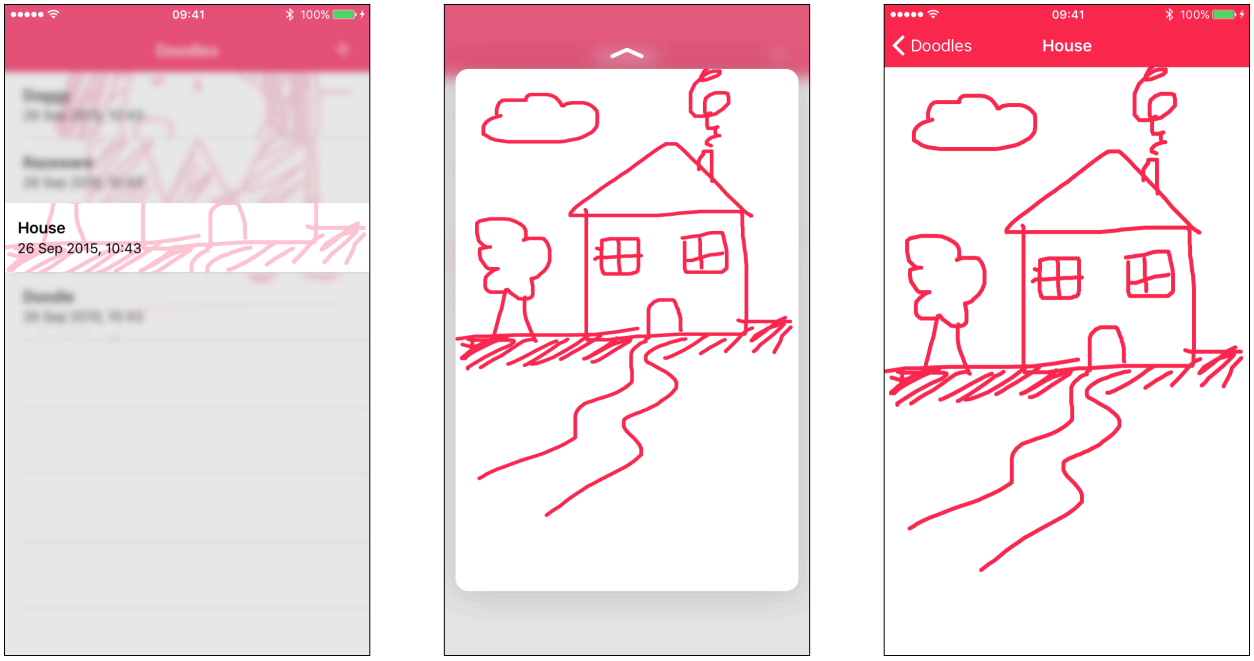
When the user presses even deeper, you'll want that preview to pop into the full display. So, still in **DoodlesViewController.swift**, replace the stub implementation of `previewingContext(_:commitViewController:)` with the following:

```
func previewingContext(
  previewingContext: UIViewControllerPreviewing,
  commitViewController
  viewControllerToCommit: UIViewController) {

  showViewController(viewControllerToCommit, sender: self)
}
```

Wow, simple huh? This method is called when the system wants to pop open, or *commit*, a preview. It hands you the `previewingContext` that you had access to in `previewingContext(_: viewControllerForLocation:)`, as well as the view controller that you returned from it.

In this situation, you simply want to push the **DoodleDetailViewController** onto the navigation stack, so you can just call `showViewController(_:sender:)` to display it.

Build and run the app. Press lightly on a row in the list of doodles, and then slowly press more deeply.



Peek.... pop!

> **Note:** In addition to being able to implement peek and pop in your own view controllers, `UIWebView` and `WKWebView` have some automatic peek and pop behaviour built in that you can take advantage of in your own apps. All you need to do is set their new `allowsLinkPreview` property to `true`, and you'll be able to peek into links.
>
> Even better: iOS 9's new `SFSafariViewController` supports peek and pop by default. No configuration required!

Preview actions

You've just implemented some awesome functionality with barely any coding, but the fun doesn't stop here! A view controller can also present some useful quick actions while in the peek state.

Open up **DoodleDetailViewController.swift** and add a property at the top of the class to store a reference to `DoodlesViewController`:

```
weak var doodlesViewController: DoodlesViewController?
```

Then add the following method to the bottom of the class, below `presentActivityViewController()`:

```
override func previewActionItems() -> [UIPreviewActionItem] {
  // 1
  let shareAction = UIPreviewAction(title: "Share",
    style: .Default) {
    (previewAction, viewController) in
    if let doodlesVC = self.doodlesViewController,
      activityViewController = self.activityViewController {

      doodlesVC.presentViewController(activityViewController,
        animated: true, completion: nil)
    }
  }

  // 2
  let deleteAction = UIPreviewAction(title: "Delete",
    style: .Destructive) {
    (previewAction, viewController) in
    guard let doodle = self.doodle else { return }
    Doodle.deleteDoodle(doodle)

    if let doodlesViewController = self.doodlesViewController {
      doodlesViewController.tableView.reloadData()
    }
  }

  return [shareAction, deleteAction]
}
```

This method is called when a view controller is peeked, and it gives the controller an opportunity to present some quick actions. Here, you're creating two types of `UIPreviewAction`:

1. A **share** action, which will present a `UIActivityViewController` that allows the user to share a doodle using the standard iOS share sheet. There's no way to present another view controller directly from the peeked view controller itself, because it's dismissed as soon as you select an action. Instead, you use the `doodlesViewController` property you created earlier and tell *it* to present the share sheet instead.

2. A **delete** action, which simply deletes the peeked doodle and tells `doodlesViewController` to reload its data.

`UIPreviewAction` is very much like the `UIAlertActions` you use with `UIAlertController`; it's initialized with a `title`, a `style` (like `.Default`, `.Destructive` or `.Selected`), and a `handler` closure. You can also group `UIPreviewActions` together using `UIPreviewActionGroup`.

A group is displayed the same way as a regular action, but it can contain multiple actions. When the user taps on a group, a submenu is opened revealing its child actions.

Finally, open **DoodlesViewController.swift** and find the `previewingContext(_:viewControllerForLocation:)` you defined earlier. Locate this line:

```
detailVC.doodle = cell.doodle
```

And add the following line below it:

```
detailVC.doodlesViewController = self
```

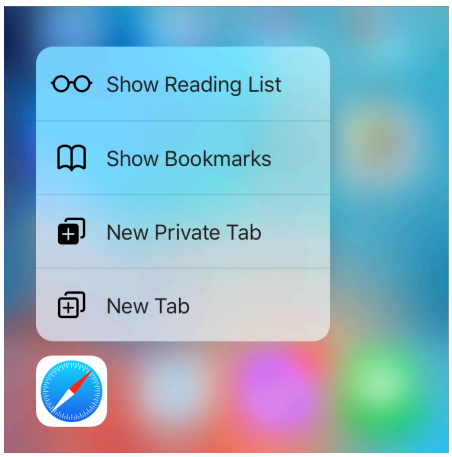
This hooks up the `doodlesViewController` property you just created.

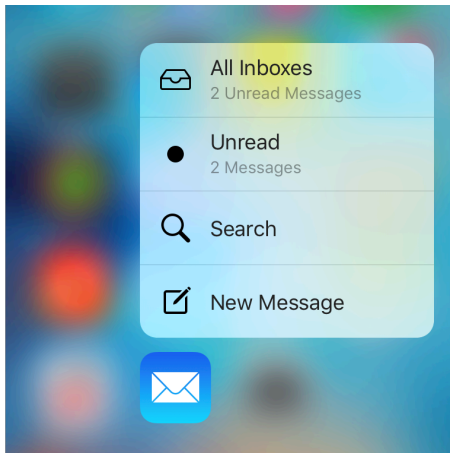
Build and run the app. Initiate a peek on one of the doodles, and then slide it upwards to reveal your quick actions. Tap on **Share** to display the share sheet, then cancel out of it. Peek at a doodle once more, slide it upward, and tap **Delete**. Whoosh, the doodle is gone!

Home screen quick actions

The final API that 3D Touch introduces is for adding home screen quick actions. These allow you to add some shortcuts to your app's home screen icon.

When the user presses deeply on your app's icon, a menu will pop up showing any shortcuts that you've defined. Here's an example of Safari and Mail's shortcut menus in iOS 9:





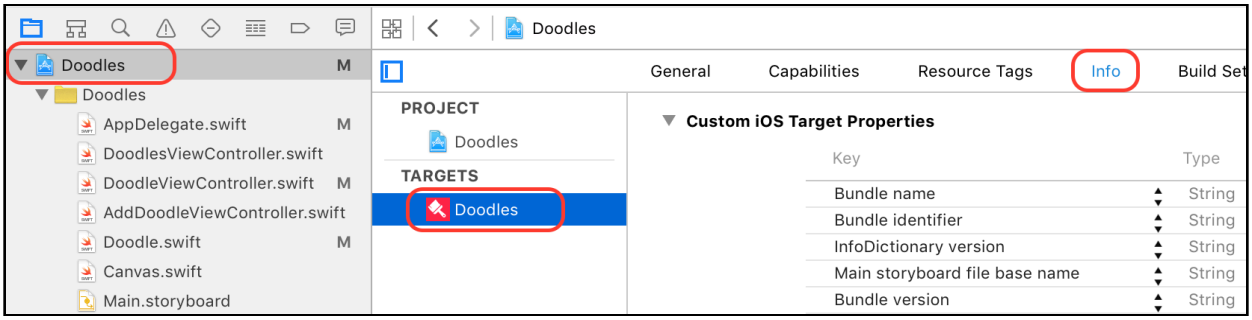
Home screen quick actions provide a great way for users to jump straight into a specific activity within your app, and they're super-easy to implement. Each app can display up to four shortcut items, and there are two types of shortcuts you can add:

- **Static shortcuts:** Static shortcuts are declared in your app's **Info.plist** file, and are available for use as soon as your app has been installed.
- **Dynamic shortcuts:** Dynamic shortcuts are configured at runtime, and can be added, removed and changed whenever you like. Because they're updated at runtime, users won't see your dynamic shortcuts until your app has run for the first time.

iOS will display your static shortcuts first, followed by your dynamic shortcuts, provided you have fewer than four static ones.

Adding a static shortcut

First, you'll add a static shortcut to take users straight to creating a new doodle. In Xcode, click the **Doodles Project** in the **project navigator**, then select the **Doodles target** and click the **Info** tab:



Add a new **Array** entry to the table, with a key of **UIApplicationShortcutItems**. Add a **Dictionary** to the array, and then add three **Strings** to the dictionary with the following keys and values:

- **UIApplicationShortcutItemTitle**: New Doodle
- **UIApplicationShortcutItemType**: com.razeware.Doodles.new
- **UIApplicationShortcutItemIconType**: UIApplicationShortcutIconTypeAdd

When you've finished, the entry should look like this:

▼ UIApplicationShortcutItems ▲▼	Array	(1 item)
▼ Item 0	Dictionary	(3 items)
UIApplicationShortcutItemIconType	String	UIApplicationShortcutIconTypeAdd
UIApplicationShortcutItemTitle	String	New Doodle
UIApplicationShortcutItemType	String	com.razeware.Doodles.new

Each dictionary within the `UIApplicationShortcutItems` array contains the definition for a shortcut item. The one you've just defined has three properties. `UIApplicationShortcutItemTitle`, as you'd expect, declares the title of the item. This one will say "New Doodle". `UIApplicationShortcutItemType` is a unique identifier (typically in reverse-DNS notation) that you'll use to identity the shortcut item in code. Finally, `UIApplicationShortcutItemIconType` declares the built-in icon to use for this entry. You're using the `UIApplicationShortcutIconTypeAdd` icon, which displays a **+** image.

There are a couple of other keys available that you haven't used here:

- **UIApplicationShortcutItemSubtitle** defines a subtitle to display below the main title.
- **UIApplicationShortcutItemIconFile** is used to provide a custom icon image.
- **UIApplicationShortcutItemUserInfo** allows you to provide a custom dictionary containing whatever data you may need.

See the `UIApplicationShortcutItems` documentation (apple.co/1KX35t4) for full details, including a list of built-in icon types.

Now that you've declared your shortcut item, what happens when somebody taps it? iOS 9 introduces a new `UIApplicationDelegate` method to do just this.

Open **AppDelegate.swift**, and add the following method below `application(_:didFinishLaunchingWithOptions:)`:

```
func application(application: UIApplication,
  performActionForShortcutItem
  shortcutItem: UIApplicationShortcutItem,
  completionHandler: (Bool) -> Void) {

  handleShortcutItem(shortcutItem)
  completionHandler(true)
}
```

This method gets called when a user selects one of your application shortcuts. Here

you're calling `handleShortcutItem(_:)` (which you'll implement in a moment) and then calling `completionHandler`, passing it `true` because you've handled the shortcut item. If for some reason you don't or can't handle a particular shortcut item, you should pass `false` to `completionHandler`.

Now, add this method below the previous one:

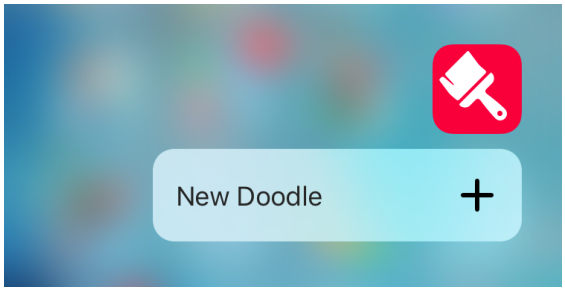
```
func handleShortcutItem(
  shortcutItem: UIApplicationShortcutItem) {

  switch shortcutItem.type {
  case "com.razeware.Doodles.new":
    presentNewDoodleViewController()
  default: break
  }
}
```

This code switches on the `type` property of `shortcutItem` that you defined earlier in **Info.plist**. If the type matches the New Doodle shortcut type, then you call `presentNewDoodleViewController()`. Add this method now below `handleShortcutItem(_:)`:

```
func presentNewDoodleViewController() {
  let identifier = "NewDoodleNavigationController"
  let doodleViewController = UIStoryboard.mainStoryboard
    .instantiateViewControllerWithIdentifier(identifier)

  window?.rootViewController?
    .presentViewController(doodleViewController, animated: true,
    completion: nil)
}
```

First you instantiate a view controller from the main storyboard with the identifier `NewDoodleNavigationController` — it's a navigation controller that has a `NewDoodleViewController` as its root view controller. Then you present it from the `rootViewController` on the `window`. This will let the user create a new doodle.

Time to test it out! Build and run, and then press firmly on the Doodles app icon on your device's home screen. You'll see your "New Doodle" application shortcut appear. Tap it, and you should land right in a new doodle screen.



Note: When an app *launches* after the user taps a shortcut, as opposed to being opened from the background, the `launchOptions` dictionary of `application(_:didFinishLaunchingWithOptions:)` will contain the `UIApplicationShortcutItem` in question as the value for the key `UIApplicationLaunchOptionsShortcutItemKey`. There are a few caveats to bear in mind when handling your shortcut items in this method, so be sure to check out the full documentation for more information: (apple.co/1P04D7q).

Adding a dynamic shortcut

You've just seen how to add a static shortcut to your **Info.plist**. Now you'll add a *dynamic* shortcut at runtime, which will allow the user to share their latest doodle right from the home screen.

Open **Doodle.swift** and add this static method to the end of the struct:

```
static func configureDynamicShortcuts() {
  if let mostRecentDoodle = Doodle.sortedDoodles.first {
    let shortcutType = "com.razeware.Doodles.share"
    let shortcutItem = UIApplicationShortcutItem(
      type: shortcutType,
      localizedTitle: "Share Latest Doodle",
      localizedSubtitle: mostRecentDoodle.name,
      icon: UIApplicationShortcutIcon(type: .Share),
      userInfo: nil)
    UIApplication.sharedApplication().shortcutItems =
      [ shortcutItem ]
  } else {
    UIApplication.sharedApplication().shortcutItems = []
  }
}
```

Just like you created a `UIApplicationShortcutItem` definition in Info.plist, here you're creating a `UIApplicationShortcutItem` in code.

The above properties and settings should look quite familiar: you set a `type` to a unique reverse-DNS string, a title and a subtitle, and a built-in icon. You've set the subtitle to the most recent doodle's name, so it'll display right in the shortcut menu.

To use programmatically-created shortcut items, you simply add them to the `UIApplication`'s `shortcutItems` property. You can update this collection at any time to ensure that your app presents useful shortcut items, depending on its current state.

Still in **Doodle.swift**, add the following line to the end of `addDoodle(_:)` ***and*** `deleteDoodle(_:)`:

```
Doodle.configureDynamicShortcuts()
```

Whenever a doodle is added or removed, this will update your dynamic shortcut

item, so that the subtitle always reflects the most recent doodle.

Open **AppDelegate.swift**, and add the same line to `application(_:didFinishLaunchingWithOptions:)`, just above `return true`:

```
Doodle.configureDynamicShortcuts()
```

Next, replace `handleShortcutItem(_:)` with the following implementation:

```
func handleShortcutItem(
  shortcutItem: UIApplicationShortcutItem) {

  switch shortcutItem.type {
  case "com.razeware.Doodles.new":
    presentNewDoodleViewController()
  case "com.razeware.Doodles.share":
    shareMostRecentDoodle()
  default: break
  }
}
```

This adds an extra case to handle the new shortcut item type, and calls `shareMostRecentDoodle()`. Add an implementation for that method now, below `presentNewDoodleViewController()`:

```
func shareMostRecentDoodle() {
  guard let mostRecentDoodle = Doodle.sortedDoodles.first,
    navigationController = window?.rootViewController as?
    UINavigationController
    else { return }
  let identifier = "DoodleDetailViewController"
  let doodleViewController = UIStoryboard.mainStoryboard
    .instantiateViewControllerWithIdentifier(identifier) as!
    DoodleDetailViewController

  doodleViewController.doodle = mostRecentDoodle
  doodleViewController.shareDoodle = true
  navigationController
    .pushViewController(doodleViewController, animated: true)
}
```

This method instantiates a new `DoodleDetailViewController` to display the most recent doodle, sets its `doodle` property and tells it to display a share sheet. Finally, it pushes it onto the navigation stack.

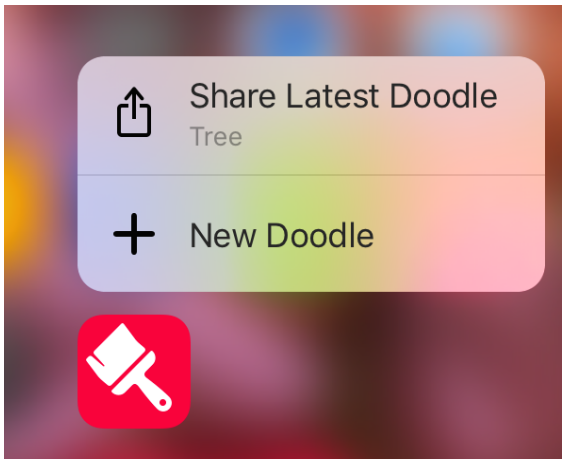
Build and run the app, and then return to the home screen. Press deeply on the app's icon to bring up the quick actions menu again.

This time, you should see two items: *New Doodle*, and *Share Latest Doodle*. Tap **Share Latest Doodle**, and you should be taken into the app. The most recent doodle, in this case House, will display and then the system share sheet will appear.

Cancel out of the share sheet, tap **Doodles** in the top left to return to the list of doodles, and then tap **+** in the top right to create a new doodle. Draw a picture of a

tree and tap **Save**. Name your new drawing *Tree*.

Press the home button to return to the home screen, and once again press deeply on the app icon to show the quick actions menu. You should see that the **Share Latest Doodle** now has the subtitle *Tree*, because it's the most recent doodle. Tap it, and you'll be able to share your latest masterpiece with the world.



Where to go from here?

Congratulations! You've reached the end of this chapter, and you've added some really cool features to the Doodles app. 3D Touch is an incredible new interaction method. With the powerful new `force` property in `UITouch`, UI candy in the form of peeks and pops, *and* home screen quick actions, you have a multitude of ways to put it to use in your own apps.

May the Force (Touch) be with you!

One of the most important things to note is that 3D Touch isn't available to all users, and it isn't as discoverable as icons and labels.

Hence, you can't have parts of your app that are only accessible via 3D touch, and shortcuts from the app icon aren't a substitute for sensible in-app navigation!

If you want to read more about 3D Touch, you should definitely check out Apple's "Adopting 3D Touch on iPhone" (apple.co/1JxalGK) documentation. It's a great resource for the API and it contains a top-level overview, links to detailed documentation, and sample code.

Chapter 7: UIStackView & Auto Layout changes

By Jawwad Ahmad

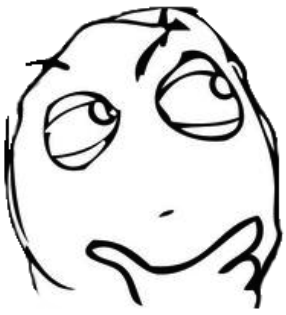
We've all been there. That annoying moment when you needed to add or remove a view at runtime and wished that other views knew how to reposition themselves automatically.

Perhaps you took the route of adding outlets to constraints in your storyboard so you could activate or deactivate certain ones. Maybe you used a third party library, or maybe you decided it's easiest to DIY it with code.

Perhaps you're one of the lucky ones, and your view hierarchy didn't have to change at runtime. But there were new requirements, and now you had to squeeze this one obnoxious view into your storyboard.

I bet you've found yourself clearing all constraints and re-adding them from scratch because it was easier than breaking out your virtual scalpel and performing painstaking *constraints-surgery*.

With the introduction of `UIStackView`, the above tasks become trivial. No more will you find yourself lying awake at night wondering how to wrangle your views!



That would leave more time for...sleep?

Stack views provide a way to horizontally or vertically position a series of views. By

configuring a few simple properties such as alignment, distribution, and spacing, you can define how the contained views adjust themselves to the available space.

In this chapter, you'll learn about stack views and about some of the other Auto Layout upgrades introduced this year, such as layout anchors and layout guides.

> **Note**: This chapter assumes basic familiarity with Auto Layout. If you're in new territory, you can do a primer on the subject by working through an Auto Layout tutorial or two on raywenderlich.com. For an in-depth look, see the "Auto Layout" chapters in the 3rd edition of *iOS 6 by Tutorials*.

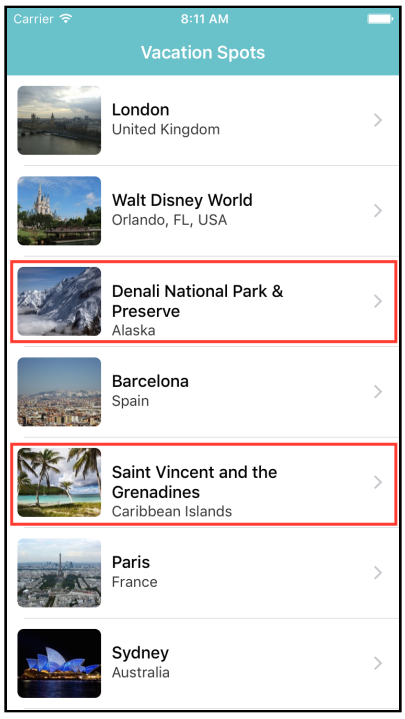
Getting started

In this chapter, you'll start working on an app called **Vacation Spots**, which will also be your guinea pig for chapter 8. It's a simple app that shows you a list of places to get away from it all. Hey, I bet you're ready for a vacation after working with constraints, right?

Don't pack the bags just yet, because there are a few issues you'll fix by using stack views, and in a much simpler way than if you were using Auto Layout alone.

Towards the end of this chapter, you'll also fix another issue with the use of layout guides and layout anchors.

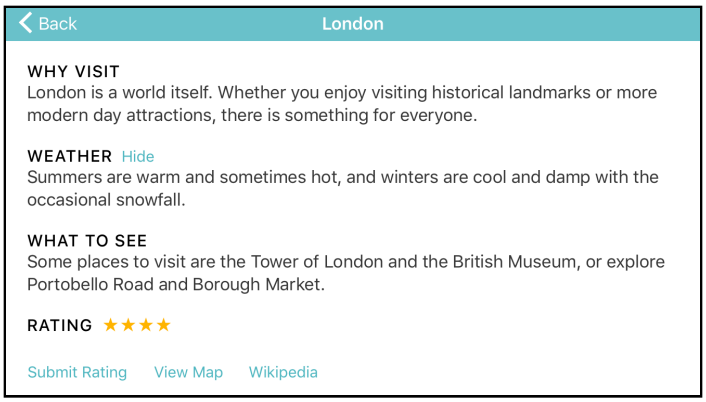
Open **VacationSpots-Starter**, and run it on the **iPhone 6 Simulator**. The first thing you'll notice is the name and location label in a few cells are off center.



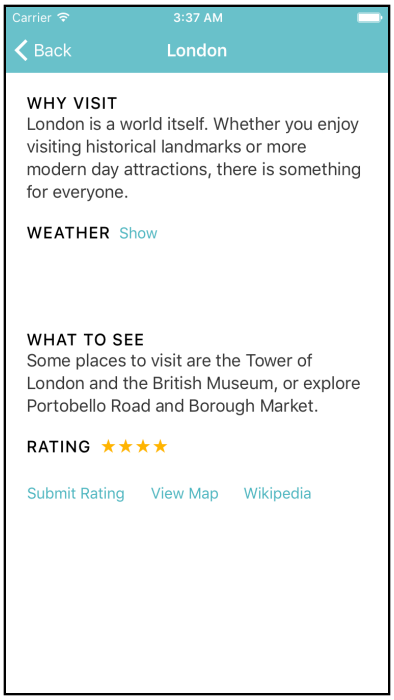
Both labels should be centered vertically (as a group) so there is an equal amount of space above the name label, and below the location label – you'll fix this towards the end of the chapter with a layout guide. For now, go to the info view for London by tapping on the **London** cell.

At first glance, the view may seem okay, but first impressions can be misleading.

1. Focus on the row of buttons at the bottom of the view. They are currently positioned with a fixed amount of space between themselves, so they don't adapt to the screen width. To see the problem in full glory, temporarily rotate the simulator to landscape orientation by pressing **Command-left**.



2. Tap on the **Hide** button next to **WEATHER**. It successfully hides the text, but it doesn't reposition the section below it, leaving a block of blank space.



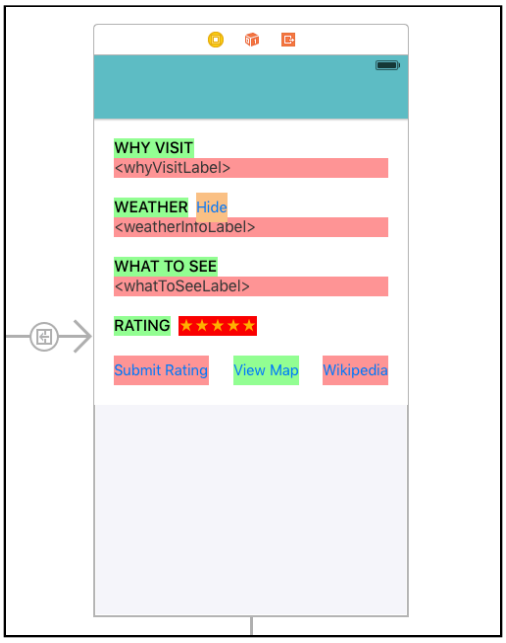
3. The ordering of the sections can be improved. It would be more logical if the **WHAT TO SEE** section was positioned right after **WHY VISIT**, instead of

having **WEATHER** in between them.

4. The bottom row of buttons is a bit too close to the bottom edge of the view in landscape mode. It would be better if you could decrease the spacing between the different sections – but only in landscape mode, i.e. when the vertical size class is compact.

You now know the what the problems are, and why they exist, so it's time to start planning your next vacation, as you enter the wonderful world of `UIStackView`.

Open **Main.storyboard** and take a look at the **Spot Info View Controller** scene. And boom! Have some color with your stack view.



These labels and buttons have various background colors set on them; they're simply visual aids to help show how changing various properties of a stack view will affect the frames of its embedded views.

Don't get too attached to the pretty colors. They're set only for the purpose of working with the storyboard and will vanish at runtime.

You don't need to do this now, but if at any point you'd actually like to see the background colors while running the app, you can temporarily comment out the following lines in `viewDidLoad()` inside `SpotInfoViewController`.

```
// Clear background colors from labels and buttons
for view in backgroundColoredViews {
  view.backgroundColor = UIColor.clearColor()
}
```

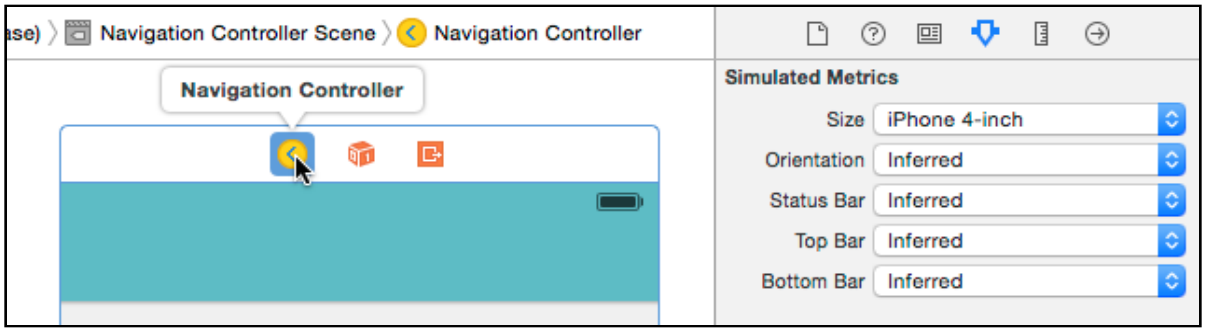
Also, any outlet-connected labels have placeholder text that's set to the name of the outlet variable to which they are connected. This makes it a bit easier to tell

which labels will have their text updated at runtime. For example, the label with text **<whyVisitLabel>** is connected to:

```
@IBOutlet weak var whyVisitLabel: UILabel!
```

Another thing to note is that the scenes in the storyboard are not the default 600 x 600 squares that you get when using size classes.

Size classes are still enabled, but the size of the initial Navigation Controller has been set to **iPhone 4-inch** under the **Simulated Metrics** section in the **Attributes inspector**. This just makes it a bit easier to work with the storyboard; the simulated metrics property has no effect at runtime — the canvas will resize for different devices.



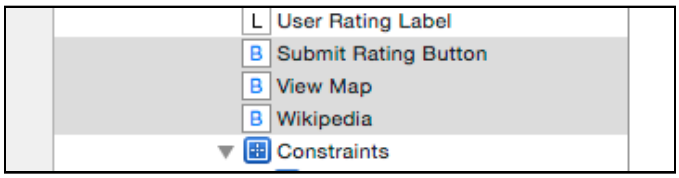
Your first stack view

First on your list of fixes is using a stack view to maintain equal spacing between the buttons on the bottom row. A stack view can distribute its views along its axis in various ways, one of which is with an equal amount of spacing between each view.

Fortunately, embedding existing views into a new stack view is not rocket science. First, select all of the buttons at the bottom of the **Spot Info View Controller** scene by **clicking** on one, then **Command-click** on the other two:

If the outline view isn't already open, go ahead and open it by using the **Show Document Outline** button at the bottom left of the storyboard canvas:

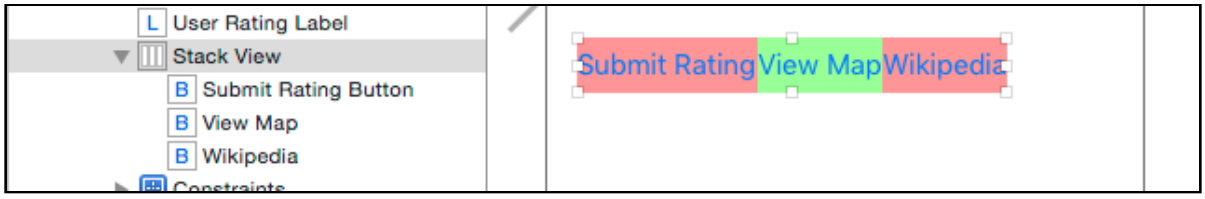
Verify that all 3 buttons are selected by checking them in the outline view:



In case they aren't all selected, you can also **Command-click** on each button in the outline view to select them.

Once selected, click on the new **Stack** button in the Auto Layout toolbar at the bottom right of the storyboard canvas:

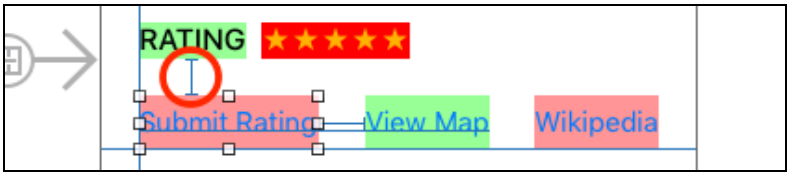
The buttons will become embedded in a new stack view:



The buttons are now flush with each other – you'll fix that shortly.

While the stack view takes care of positioning the buttons, you still need to add Auto Layout constraints to position the stack view itself.

When you embed a view in a stack view, any constraints to other views are removed. For example, prior to embedding the buttons in a stack view, the top of the **Submit Rating** button had a vertical spacing constraint connecting it to the bottom of the **Rating:** label:

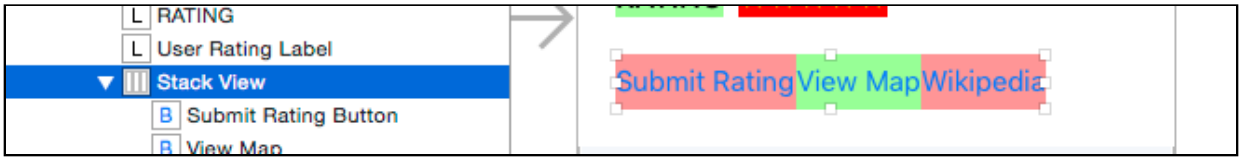


Click on the **Submit Rating** button to see that it no longer has any constraints attached to it:

Another way to verify that the constraints are gone is by looking at the **Size inspector** (⌥⌘5):

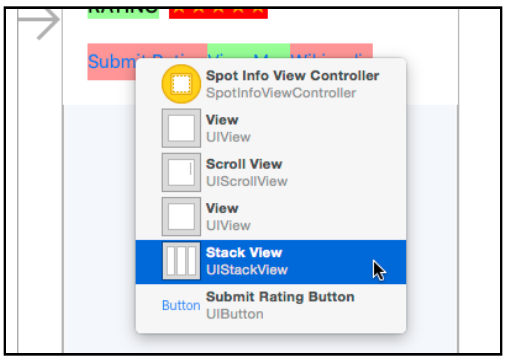
In order to add constraints to position the stack view itself, you'll first need to select it. Selecting a stack view in the storyboard can get tricky if its views completely fill the stack view.

One simple way is to select the stack view in the outline view:

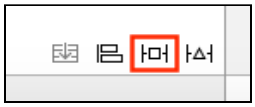


Another trick is to hold **Shift** and **Right-click** on any of the views in the stack view, or **Control-Shift-click** if you're using a trackpad. You'll get a context menu that shows the view hierarchy at the location you clicked, and you simply select the stack view by clicking on it in the menu.

For now, select the stack view using the **Shift-Right-click** method:



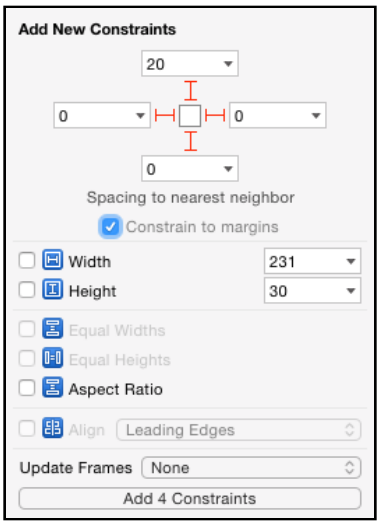
Now, click the **Pin** button on the Auto Layout toolbar to add constraints to it:



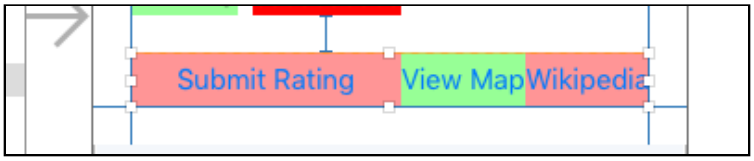
First add a check to **Constrain to margins**. Then add the following constraints to the edges of your stack view:

```
Top: 20, Leading: 0, Trailing: 0, Bottom: 0
```

Double-check the numbers for the top, leading, trailing, and bottom constraints and make sure that the **I-beams** are selected. Then click on **Add 4 Constraints**:



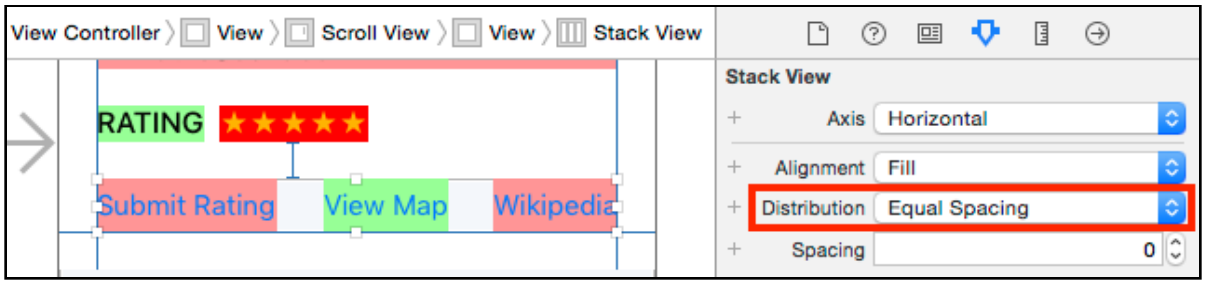
Now the stack view is the correct size, but it has stretched the first button to fill in any extra space:



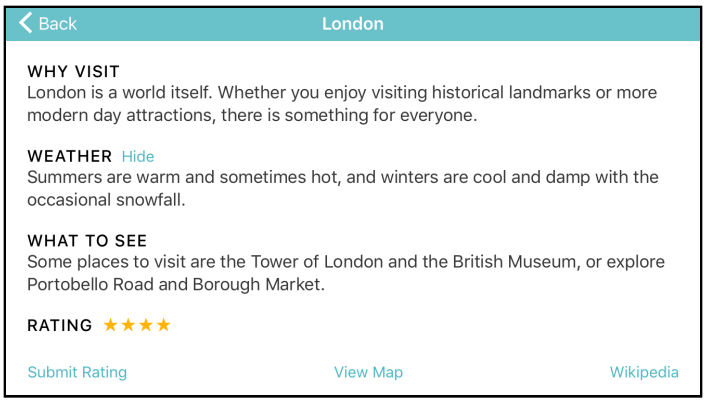
The property that determines how a stack view lays out its views along its axis is its `distribution` property. Currently, it's set to `Fill`, which means the contained views will completely fill the stack view along its axis. To accomplish this, the stack view will only expand one of its views to fill that extra space; specifically, it expands the view with the lowest horizontal content hugging priority, or if all of the priorities are equal, it expands the first view.

However, you're not looking for the buttons to fill the stack view completely – you want them to be equally spaced.

Make sure the stack view is still selected, and go to the **Attributes inspector**. Change the **Distribution** from **Fill** to **Equal Spacing**:



Now build and run, tap on any cell, and rotate the simulator (⌘→). You'll see that the bottom buttons now space themselves equally!

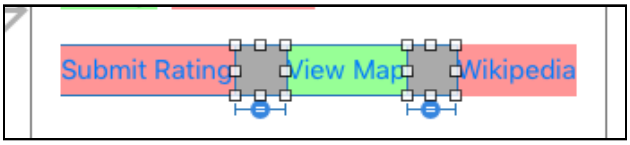


Consider the alternatives

Now that you've had your first taste of the ease of working with stack views, think about how you'd solve this problem without using them:

- You would have added dummy spacer views, one between each button pair of buttons.
- You'd select all of the spacer views and give them equal width constraints.
- Then you'd pin each spacer view to the buttons on either side of it.
- You'd also need to add constraints for the heights and vertical positions of the spacer views in the superview, or alternatively, you could pin the top and bottom edges to the adjacent buttons.

It would have looked something like the following. For visibility in the screenshot, the spacer views have been given a light gray background:

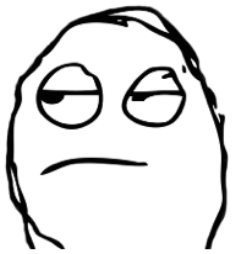


But I'm an Auto Layout master

Perhaps you're a seasoned Auto Layout veteran, and adding constraints like these is a piece of cake. And you might not feel like you've gained much by using a stack view.

You'd still have optimized your layout by not having to include unnecessary spacer views, but even if you ignore this benefit, think about the long term.

What happens when you need to add a new button? Oh, right, you could just add a new button because it's not too difficult for an expert like you to re-do all the constraints. But doesn't dragging and dropping the additional button into place, and having the stack view take care of the positioning sound better?



fffft! Will stack views do my laundry too?

There's more. What if you needed to conditionally hide and show one of the buttons and reposition all of the remaining ones at runtime? If you stuck to the old ways, you'd have to manually remove and re-add constraints in code as well as remove and add back the adjacent spacer view.

Thanks, but me and Auto Layout have a deal

And what if the requirement specified that more than one button could be removed and re-added at any time? At this point, you might as well do everything in code.

Code...it...all?

Stack views are just better

In order to hide a view within a stack view, all you have to do is set the contained view's `hidden` property to `true` and the stack view handles the rest. This is how you'll fix the spacing under the **WEATHER** label when the user hides the text below it.

But that's something for the next chapter, where you'll dive deeper into stack views. For now, you'll take a quick detour to learn about some of the other new Auto Layout updates in iOS 9.

Stack views are by far the biggest feature introduced to Auto Layout with iOS 9, but there are also other features that improve how you do things. Two of the most interesting are **layout anchors** and **layout guides**, represented by the new `NSLayoutAnchor` and `UILayoutGuide` classes.

Layout anchors

Layout anchors provide a simplified way to create constraints.

Imagine you have two labels, `bottomLabel` and `topLabel`. You'd like to position `bottomLabel` right below `topLabel` with 8 points of spacing between them. Prior to iOS 9, you'd use the following code to create the constraint:

```
let constraint = NSLayoutConstraint(
  item: topLabel,
  attribute: .Bottom,
  relatedBy: .Equal,
  toItem: bottomLabel,
  attribute: .Top,
  multiplier: 1,
  constant: 8
)
```

This creates a constraint in which the topLabel's `.Bottom` is `.Equal` to the bottomLabel's `.Top`, plus 8.

Even the explanation contained far fewer characters than the code itself, and you're not even using Objective-C! Surely there's a more concise way to express this?

Layout anchors allow you to do exactly that:

```
let constraint = topLabel.bottomAnchor.constraintEqualToAnchor(
  bottomLabel.topAnchor, constant: 8)
```

This achieves the same result. The view now has a layout anchor object representing the `.Bottom` attribute, and that anchor object can create constraints relating to other layout anchors.

That's all there is to it! `UIView` now has a `bottomAnchor` property, as well as other anchors that correspond to other `NSLayoutAttributes`. For example, for `.Width` it has `widthAnchor`, for `.CenterX` it has `centerXAnchor` etc.

They're all subclasses of `NSLayoutAnchor`, which has a number of methods for creating constraints relating to other anchors. In addition to the above method, there is also a convenience version for when the constant is 0:

```
let constraint = topLabel.bottomAnchor.constraintEqualToAnchor(
  bottomLabel.topAnchor)
```

That's a much more expressive and concise way to create a constraint!

Note: A view doesn't have an anchor property for every possible layout attribute. Any of the attributes relating to the margins, such as `.TopMargin`, or `.LeadingMargin` aren't available directly. `UIView` has a new property called `layoutMarginsGuide`, which is a `UILayoutGuide` (you'll learn about these in the next section). The layout guide has all the same anchor properties as the view, but now they relate to the view's margins, for example `.TopMargin` would be represented by `layoutMarginsGuide.topAnchor`.

Non-equal constraints

The method above creates an `EqualTo` constraint, but what if you wanted to create a `LessThanOrEqualTo` or a `GreaterThanOrEqualTo` constraint?

Using the old method, you'd just pass in `.GreaterThanOrEqual` or `.LessThanOrEqual` for the `relatedBy:` parameter, which takes an enum of type `NSLayoutRelation`:

```
let constraint = NSLayoutConstraint(
  ...
  relatedBy: .LessThanOrEqual, // or .GreaterThanOrEqual
  ...
)
```

`NSLayoutAnchor` also contains methods to express those relations:

```
func constraintLessThanOrEqualToAnchor(_:constant:)
func constraintGreaterThanOrEqualToAnchor(_:constant:)
```

As well as the more concise variants in which you can leave off the `constant:`

param when the value is 0:

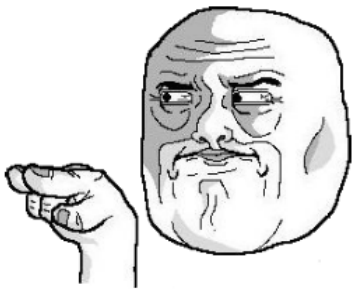
```
func constraintLessThanOrEqualToAnchor(_:)
func constraintGreaterThanOrEqualToAnchor(_:)
```

Including a multiplier

In the old method, there is also a `multiplier` parameter:

```
let constraint = NSLayoutConstraint(
  ...
  multiplier: 1,
  ...
)
```

So, how do you include a multiplier if you need to? If you look at the documentation for `NSLayoutAnchor`, you won't find any methods that contain a `multiplier` parameter.



But `NSLayoutAnchor` *does* have a subclass called `NSLayoutDimension` that has the following methods:

```
func constraintEqualToConstant(_:)
func constraintEqualToAnchor(_:multiplier:)
func constraintEqualToAnchor(_:multiplier:constant:)
```

It also has the `LessThanOrEqual` and `GreaterThanOrEqual` variants:

```
func constraint[Less|Greater]ThanOrEqualToConstant(_:)
func constraint[Less|Greater]ThanOrEqualToAnchor(_:multiplier:)
func constraint[Less|Greater]ThanOrEqualToAnchor(
  _:multiplier:constant:)
```

When would you actually use a multiplier other than 1? Here's an idea: When you want to add a proportional constraint between the width or height of one view to the width or height of another view, like if you wanted the width of user's profile photo to be one-quarter that of its superview.

Effectively, the only anchors that are of type `NSLayoutDimension` are `heightAnchor`

and `widthAnchor`.

This means that you can't accidentally use a multiplier where it doesn't make sense, and since the multiplier-based methods don't exist with anything other than `widthAnchor` and `heightAnchor`, Xcode won't even suggest them to you.

It gets better. `NSLayoutAnchor` has two additional subclasses — `NSLayoutXAxisAnchor` and `NSLayoutYAxisAnchor` — which represent anchors in the horizontal and vertical directions. For example, `bottomAnchor` is of type `NSLayoutYAxisAnchor` and `leadingAnchor` is of type `NSLayoutXAxisAnchor`. So all anchors are actually one of these three specific subclasses of `NSLayoutAnchor`.

The `constraint[Equal|LessThanOrEqual|GreaterThanOrEqual]ToAnchor` family of methods are actually generic methods that, when called from an object of type, `NSLayoutXAxisAnchor`, will only take a parameter of type `NSLayoutXAxisAnchor` and when called from an object of type, `NSLayoutYAxisAnchor` will only take in a parameter of type `NSLayoutYAxisAnchor`. This can prevent you attempting to pin the top of one view to the leading edge of another, for example.

Though this type checking hasn't yet made its way into Swift, it currently works with Objective-C:

```
47
48    [label1.topAnchor constraintEqualToAnchor:label2.leadingAnchor];
49    Incompatible pointer types sending 'NSLayoutXAxisAnchor *' to parameter of type 'NSLayoutAnchor<NSLayoutYAxisAnchor *> *'
50
```

At the time of writing, Swift will still crash at runtime with the message "Invalid pairing of layout attributes", so you'll know pretty quickly if you've made a mistake.

You'll also still get an error in Swift if you try to constrain an `NSLayoutDimension` anchor with a different type of anchor, for example, a `widthAnchor` with a `topAnchor`:

```
108
109     label1.widthAnchor.constraintEqualToAnchor(label2.widthAnchor, multiplier: 1)
110
111     label1.widthAnchor.constraintEqualToAnchor(label2.topAnchor, multiplier: 1)
112       Cannot invoke 'constraintEqualToAnchor' with an argument list of type '(NSLayoutYAxisAnchor, multiplier: Int)'
113
```

To summarize:

- Layout anchors are a quick way of making constraints between different attributes of a view
- There are three different subclasses of `NSLayoutAnchor`
- The compiler will prevent you from creating constraints between the different subclasses of `NSLayoutAnchor`

The specific subclasses of `NSLayoutAnchor` are:

- `NSLayoutXAxisAnchor` for leading, trailing, left, right or center X anchors

- `NSLayoutYAxisAnchor` for top, bottom and center Y anchors
- `NSLayoutDimension` for width and height

Whew, that was a lot to cover. You're probably wondering if you'll ever fix that alignment bug. Of course you will! But first, read through the next section on layout guides — I promise it's much shorter. :]

After that, you'll be fully prepared to dive back into the code and fix that bothersome alignment bug.

Layout guides

A layout guide gives you a new way to position views when you'd previously need to use a dummy view. For example, you might have used spacer views between buttons to space them equally, or a container view to collectively align two labels. But creating and adding a view has a cost, even if it's never drawn.

Think of a layout guide as defining a rectangular region or a frame in your view hierarchy, the edges of which you can use for alignment.

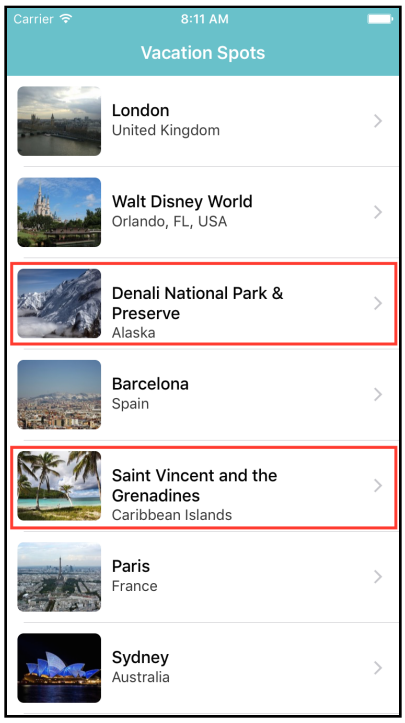
Layout guides don't enable any new functionality, but they do allow you to address these problems with a lightweight solution.

You add constraints to a `UILayoutGuide` in the same way that you add them to a `UIView`, because a layout guide has almost the same layout anchors as a view — dropping just the inapplicable `firstBaselineAnchor` and `lastBaselineAnchor`.

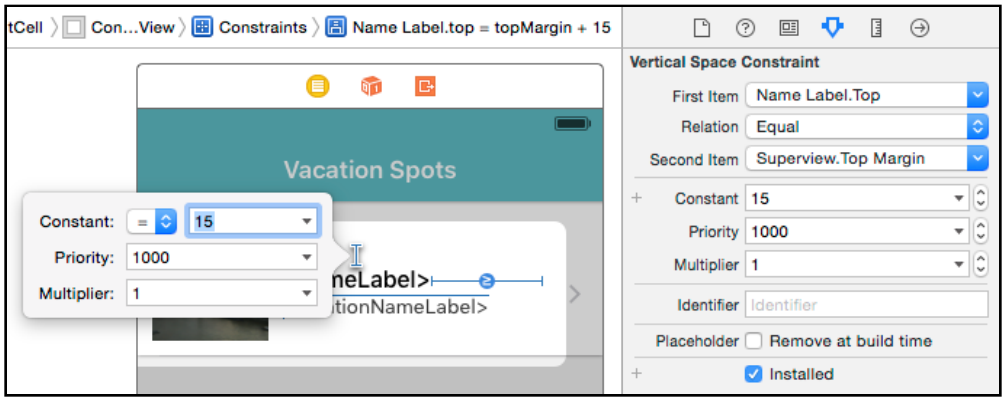
Okay, now it's time to dive back into the project and fix that alignment bug.

Fixing the alignment bug

You'll remember that the whole reason for the dive into layout guides and anchors was so that you could vertically center some of these misaligned labels in the cell by using the new tools available in iOS 9:



They're misaligned because the current constraint specifies that the top of the name label should be a fixed distance from the top margin of the cell's `contentView`:



If the name label was always on a single line, the current constraint would have been fine. But this app has labels that span two lines.

Prior to iOS 9, you'd have centered everything by putting both labels into a container view, and then you would have added a constraint to center the container view vertically within the cell. The only purpose of creating this dummy container view was for the collective alignment of the two labels.

But now you know that you can use a layout guide instead.

Currently, it's only possible to add a layout guide in code. So open **VacationSpotCell.swift** and add the following code to `awakeFromNib()`:

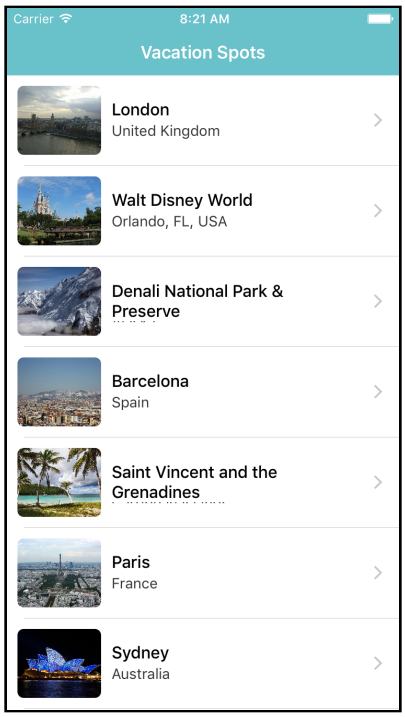
```
// 1
let layoutGuide = UILayoutGuide()
contentView.addLayoutGuide(layoutGuide)

// 2
let topConstraint = layoutGuide.topAnchor
  .constraintEqualToAnchor(nameLabel.topAnchor)

// 3
let bottomConstraint = layoutGuide.bottomAnchor
  .constraintEqualToAnchor(locationNameLabel.bottomAnchor)

// 4
let centeringConstraint = layoutGuide.centerYAnchor
  .constraintEqualToAnchor(contentView.centerYAnchor)

// 5
NSLayoutConstraint.activateConstraints(
  [topConstraint, bottomConstraint, centeringConstraint])
```

Here's the breakdown of the code you just added:

1. Create the `layoutGuide` and use `addLayoutGuide(_:)` to add it to the cell's `contentView`.
2. Pin the top of the layout guide to the top of the `nameLabel`.
3. Pin the bottom of the layout guide to the bottom of the `locationNameLabel`.
4. Add a constraint to center the layout guide vertically within the `contentView`.
5. Activate the constraints.

Note: Using the `activateConstraints(_:)` method on UIView is the recommended way of adding constraints in iOS 8 onwards, as opposed to the old way of using `addConstraints(_:)`.

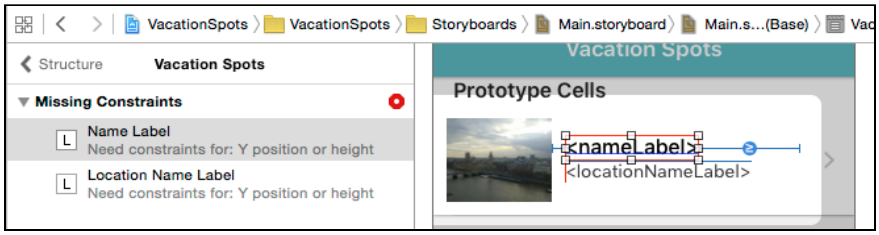
Build and run, you should see the following:



Handling the truncation

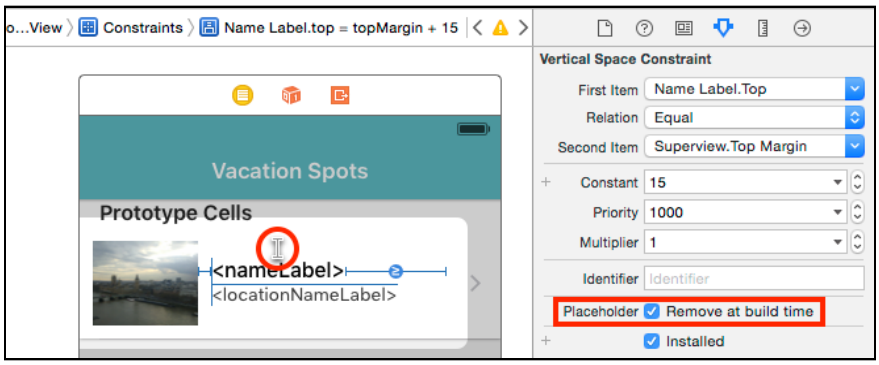
The labels are centered, but when the upper name label has content that causes it to overflow onto two lines, the bottom label has become compressed to the point of almost disappearing. This is because of the constraint that's still in the storyboard.

In order to satisfy that constraint as well as the newly added centering constraint, the bottom label had to compress itself. You can't remove the constraint from the storyboard since you would get the following missing constraint error:

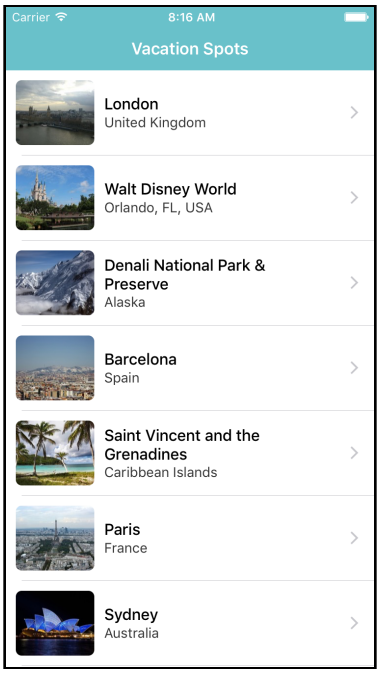


Instead, simply set it as a placeholder constraint. This is a trick to tell Xcode that you'll leave this constraint here for the storyboard, but you want it removed at runtime since you've got it covered in code.

Open **Main.storyboard**, and in the **Vacation Spots scene** click on **<nameLabel>** to select it, and then click on the constraint connecting the top of the label to the top margin of the cell. Place a checkmark next to **Remove at build time**:



Build and run, and you'll see the labels centered correctly!



Where to go from here?

In this chapter, you started learning about stack views and also learned about some of the new features in Auto Layout, such as layout anchors and layout guides.

At this point, you've only scratched the surface. Keep up the momentum and proceed to the next chapter, where you'll continue to learn about stack views in depth. At the end of the next chapter, there will be some additional resources you can use to further your learning, but for now, all you have to do is turn the page!

Chapter 8: Intermediate UIStackView

By Jawwad Ahmad

Welcome back! In the previous chapter, you spent some quality time with stack views and masterfully spaced a row of buttons with a horizontal stack view. Moreover, you also learned about layout guides and layout anchors, and discovered how to use them to vertically center two labels in a table view cell, without the use of dummy container views.

In this chapter, you'll make further improvements to the Vacation Spots app by using — you guessed it — stack views.

Getting started

Open your project from the previous chapter to continue from where you left off, or open the starter project for this chapter.

A recap of your to-do's

Here's a quick recap of the four tasks needed to improve `SpotInfoViewController`; these were laid out for you in the previous chapter.

1. Equally space the bottom row of buttons. **Done!**
2. After pressing the Hide button, the section below it should reposition to occupy the empty space. **Not Done.**
3. Swap the positions of the **what to see** and **weather** sections. **Not Done.**
4. Increase the spacing between sections in landscape mode so that the bottom row of buttons is a comfortable distance from the bottom edge of the view. **Not Done.**

In addition, you'll add some animations to spruce things up a bit.

Converting the sections

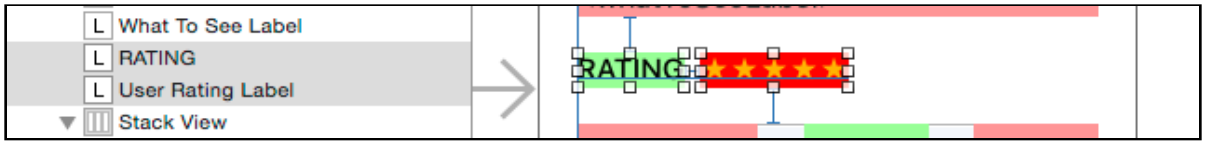
Before you can check off the rest of your to-do's, you need to convert all of the sections in `SpotInfoViewController` to use stack views.

And as you work through this section, you'll learn about the various properties you can use to configure a stack view, such as `alignment`, `distribution` and `spacing`.

Rating section

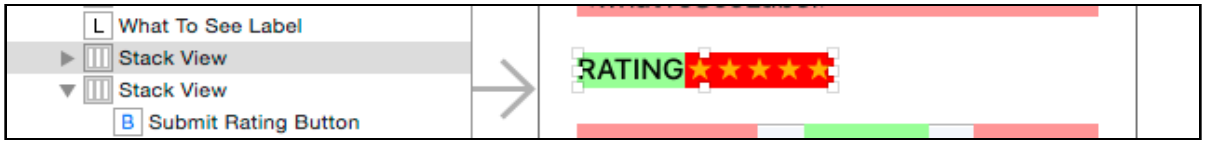
The rating section is the low-hanging fruit here, because it's the simplest one to embed in a stack view.

Open **Main.storyboard** and in the **Spot Info View Controller** scene, select the **RATING** label and the stars label next to it:



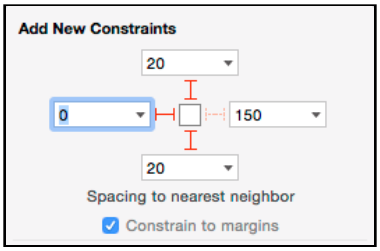
Then click on the **Stack** button to embed them in a stack view. Remember, this button is at the bottom of the storyboard window:

You can also use the menu bar and select **Editor \ Embed in \ Stack View**. Whichever way you go about it, this is the result:

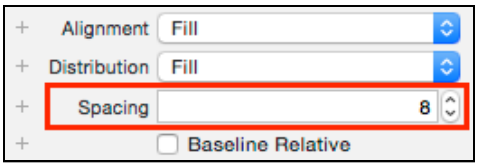


Now click on the **Pin** button — remember that's the square TIE fighter-looking icon that's sitting to the right of the stack button. Place a checkmark in **Constrain to margins** and add the following **three** constraints:

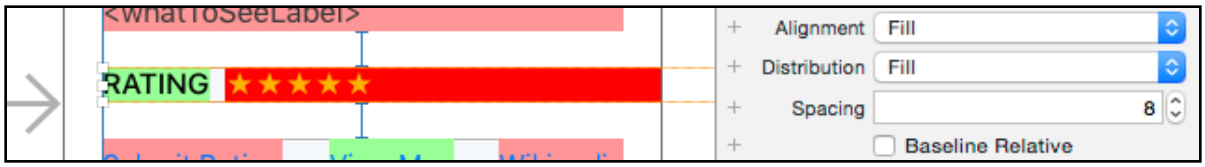
```
Top: 20, Leading: 0, Bottom: 20
```



Now go to the **Attributes inspector** and set the spacing to **8**:



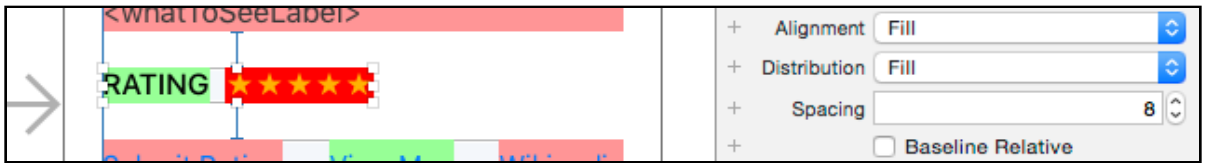
It's possible you may see a *Misplaced Views* warning and see something like this in which the stars label has stretched beyond the bounds of the view:



Sometimes Xcode may temporarily show a warning or position the stack view incorrectly, but the warning will disappear as you make other updates. You can usually safely ignore these.

However, to fix it immediately, you can persuade the stack view to re-layout either by moving its frame by one point and back or temporarily changing one of its layout properties.

To demonstrate this, change the **Alignment** from **Fill** to **Top** and then back to **Fill**. You'll now see the stars label positioned correctly:



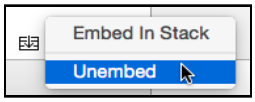
Build and run to verify that everything looks exactly the same as before.

Unembedding a stack view

Before you go too far, it's good to have some basic "first aid" training. Sometimes you may find yourself with an extra stack view that you no longer need, perhaps because of experimentation, refactoring or just by accident.

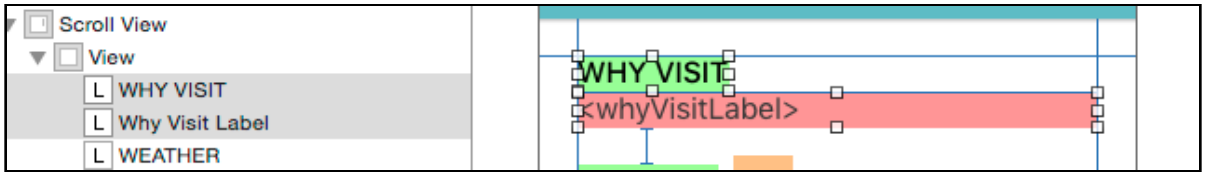
Fortunately, there is an easy way to *unembed* views from a stack view.

First, you'd select the stack view you want to remove. Then from the menu you'd choose **Editor \ Unembed**. Or another way is to hold down the **Option** key and click on the **Stack** button. The click **Unembed** on the context menu that appears:

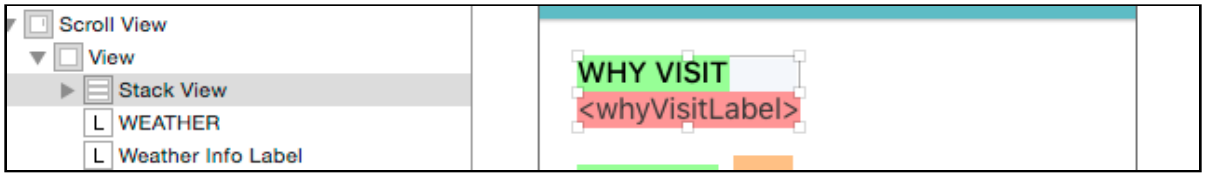


Your first vertical stack view

Now, you'll create your first vertical stack view. Select the **WHY VISIT** label and the **<whyVisitLabel>** below it:



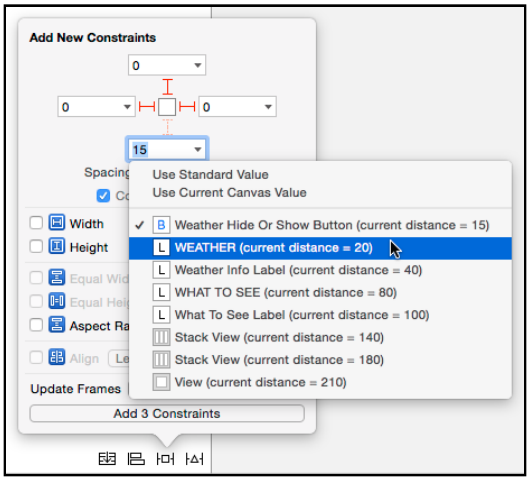
Xcode will correctly infer that this should be a vertical stack view based on the position of the labels. Click the **Stack** button to embed both of these in a stack view:



The lower label previously had a constraint pinning it to the right margin of the view, but that constraint was removed when it was embedded in the stack view. Currently, the stack view has no constraints, so it adopts the intrinsic width of its largest view.

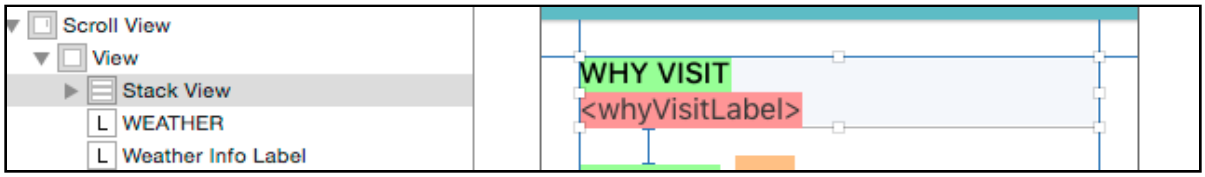
With the stack view selected, click on the **Pin** button. Checkmark **Constrain to margins**, and set the **Top**, **Leading** and **Trailing** constraints to **0**.

Then, click on the dropdown to the right of the bottom constraint and select **WEATHER (current distance = 20)**:



By default, constraints are shown to the nearest neighbor, which for the bottom constraint is the **Hide** button at a distance of 15. You actually needed the constraint to be to the **WEATHER** label below it.

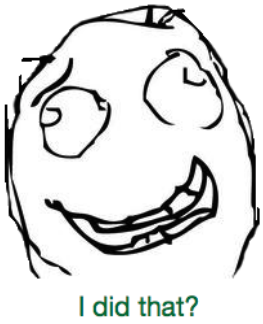
Finally, click **Add 4 Constraints**. You should now see the following:



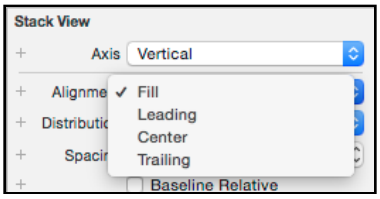
You now have an expanded stack view with its right edges pinned to the right margin of the view. However, the bottom label is still the same width. You'll fix this by updating the stack view's `alignment` — keep reading to discover how!.

Alignment property

Remember how you previously learned that the `distribution` property specifies how a stack view lays out its views *along* its axis? You had set the bottom stack view's `distribution` to *Equal Spacing* to space the buttons within it equally.

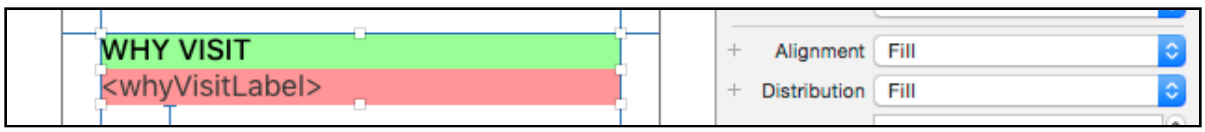


Well, meet `alignment`. It's the property that determines how a stack view lays out its views *perpendicular* to its axis. For a vertical stack view, the possible values are `Fill`, `Leading`, `Center` and `Trailing`.

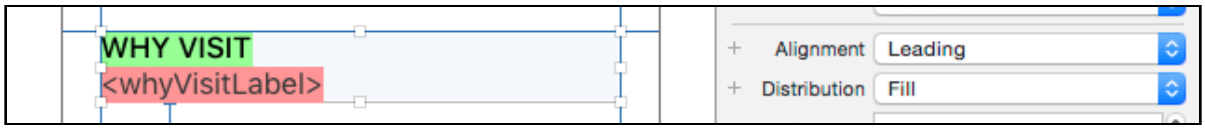


Select each value to see how it affects the placement of the labels in the stack view:

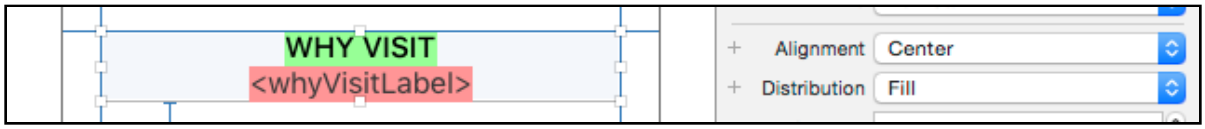
Fill:



Leading:

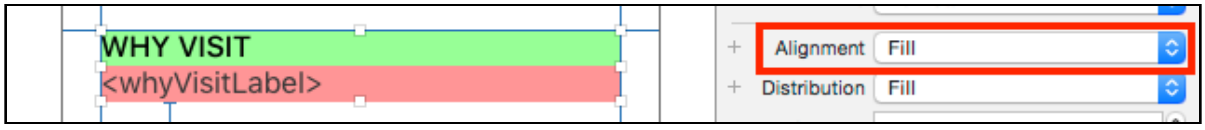


Center:



Trailing:

When you're done testing each value, set the **Alignment** to **Fill**:



Then build and run to verify that everything looks good, and that there are no regressions.

Specifying `Fill` means you want all the views to completely fill the stack view perpendicular to its axis. This causes the **WHY VISIT** label to expand itself to the right edge as well.

But what if you only wanted the bottom label to expand itself to the edge?

For now, it doesn't matter since both labels will have a clear background at runtime, but it'll matter when you're converting the weather section.

You'll learn how to accomplish that with the use of an additional stack view.

Convert the "what to see" section

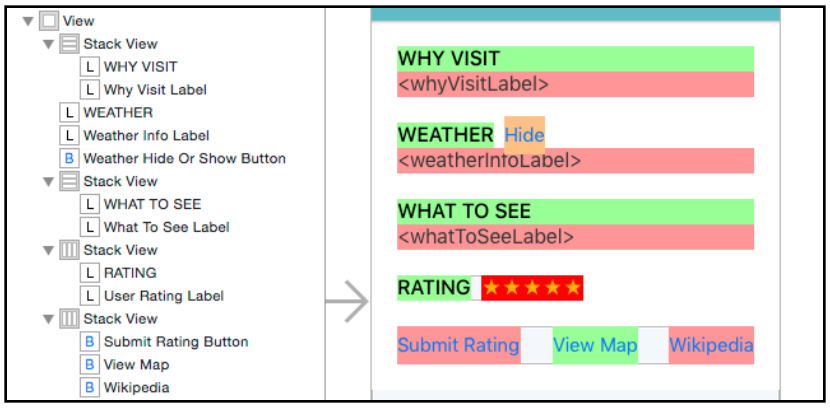
This section is very similar to the previous one, so the instructions here are brief.

1. First, select the **WHAT TO SEE** label and the **<whatToSeeLabel>** below it.
2. Click on the **Stack** button.
3. Click on the **Pin** button.
4. Checkmark **Constrain to margins**, and add the following **four** constraints:

```
Top: 20, Leading: 0, Trailing: 0, Bottom: 20
```

5. Set the stack view's **Alignment** to **Fill**.

Your storyboard should now look like this:



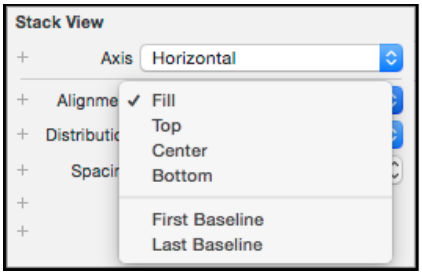
Build and run to verify that everything still looks the same.

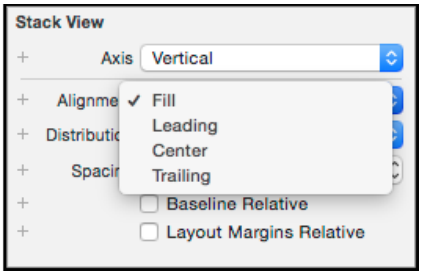
That leaves you with just the **weather** section left. But first, indulge in a quick detour to learn a little more about `alignment`.

Alignment

The `alignment` property is an enum of type `UIStackViewAlignment`. Its possible values in the vertical direction are `.Fill`, `.Leading`, `.Center`, and `.Trailing` which you saw in the previous section.

The possible `alignment` values for a *horizontal* stack view differ slightly:



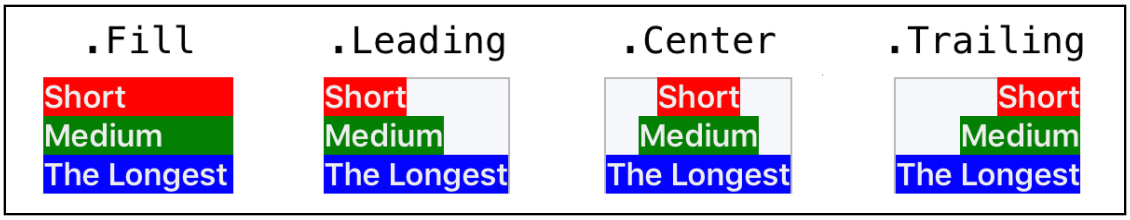


It has `.Top` instead of `.Leading` and has `.Bottom` instead of `.Trailing`. There are also two more properties that are valid only in the horizontal direction, `.FirstBaseline` and `.LastBaseline`.

Here's are some visuals to illustrate how each value works:

Horizontal axis:

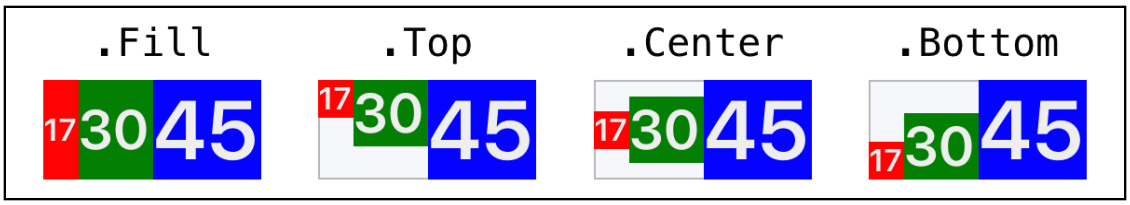
Here labels of different widths are aligned according to each value:



Now if the labels were the same width and the stack view was not stretched beyond its intrinsic width with a constraint, it wouldn't matter what value was chosen since they would all just fill the stack view.

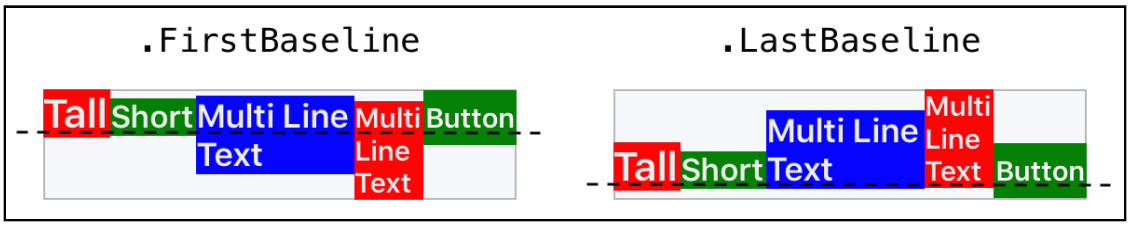
Vertical axis:

Here the labels are configured with different font sizes to give them different intrinsic heights, in order to demonstrate the different values:



FirstBaseline and LastBaseline:

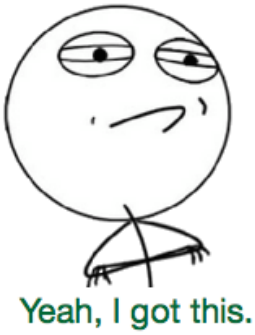
These values are valid only in a horizontal stack view. `FirstBaseline` uses the baseline of the *first* line in multi-line text, and `LastBaseline` uses the baseline of the *last* line in multi-line text.



Convert the weather section

The next task is to add the **weather** section to a stack view. You'll start by adding it to a stack view.

Remember that little **Hide** button? Get ready, because this stack view is a bit more complex due to the inclusion of the **Hide** button.

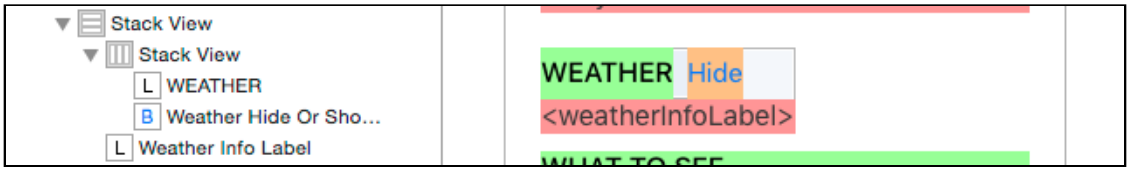


One possible approach

> **Note**: This section explores one possible approach, but don't follow along in Xcode just yet. Consider this first section theoretical.

You could create a nested stack view by embedding the **WEATHER** label and the **Hide** button into a horizontal stack view, and then embed that horizontal stack view and the **<weatherInfoLabel>** into a vertical stack view.

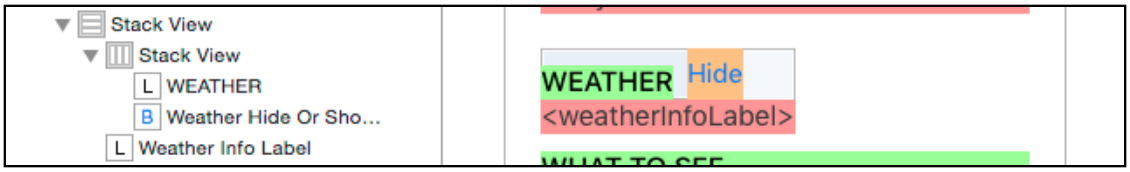
It would look something like this:



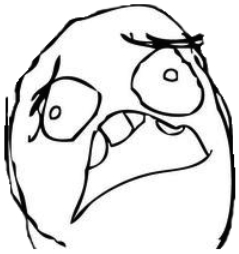
Notice that the **WEATHER** label has expanded to be equal to the height of the **Hide** button. This isn't ideal since this will cause there to be extra space between the baseline of the **WEATHER** label and the text below it.

Remember that `alignment` specifies positioning perpendicular to the stack view. So,

you could set the `alignment` to **Bottom**:



But you really don't want the height of the **Hide** button to dictate the height of the stack view.

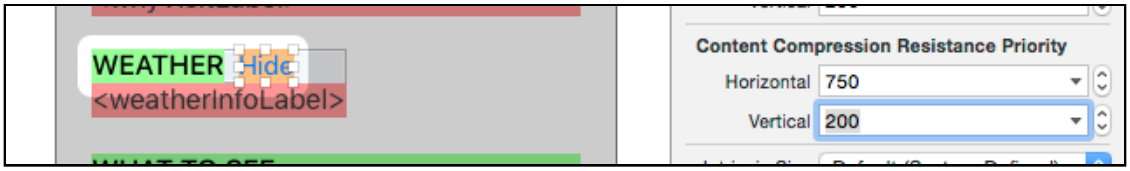


So what is a developer to do?

When the `alignment` of a stack view is set to `fill` and the views are of different sizes in the alignment direction, the stack view determines which views to compress or expand based on the relative content hugging priorities or the content compression resistance priorities of its views.

In your case, the stack view decides to expand the **WEATHER** label because its vertical content hugging priority of 251 is less than the **Hide** button's compression resistance priority of 750.

You *could* decrease the **Hide** button's vertical compression resistance priority to 200 which would cause the stack view to compress the **Hide** button instead:



However, this isn't ideal since it would reduce the size of the tap target of the button.

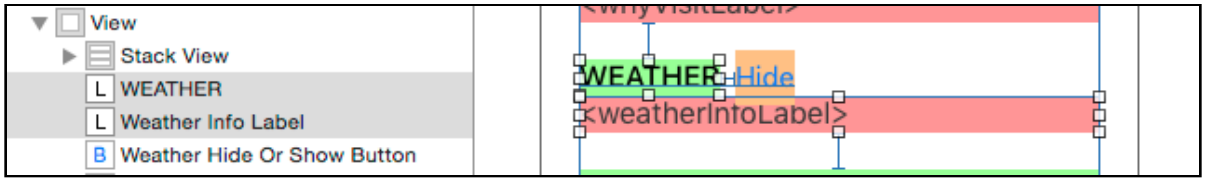
Actual approach

The actual approach you'll take is to have the **Hide** button *not* be in the stack view for the **weather** section, or any other stack view for that matter.

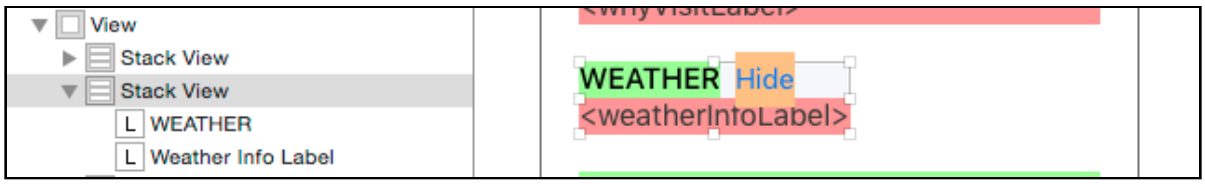
It will remain a subview of the top-level view, and you'll add a constraint from it to the **WEATHER** label — which will be in a stack view. That's right, you'll add a constraint from a button outside of a stack view to a label within a stack view!

Change the weather section – for real

You can once again start following along in Xcode. Select the **WEATHER** label and the **<weatherInfoLabel>** below it:



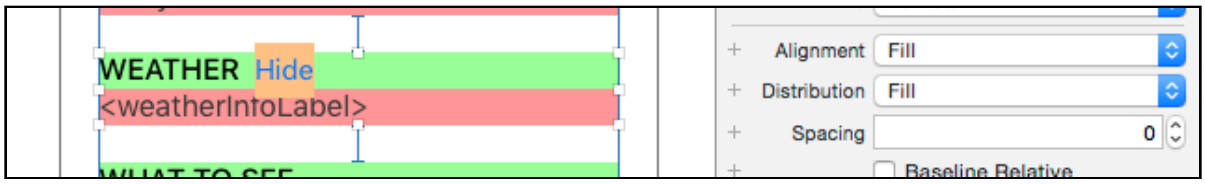
Click on the **Stack** button:



Click on the **Pin** button, checkmark **Constrain to margins** and add the following **four** constraints:

```
Top: 20, Leading: 0, Trailing: 0, Bottom: 20
```

Set the stack view's **Alignment** to **Fill**:

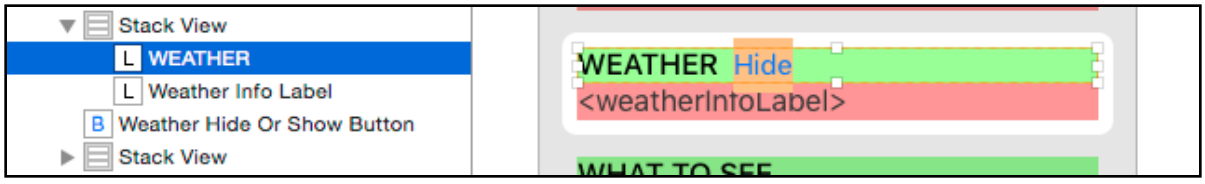


You need a constraint between the **Hide** button's left edge and the **WEATHER** label's right edge, so having the **WEATHER** label fill the stack view won't work.

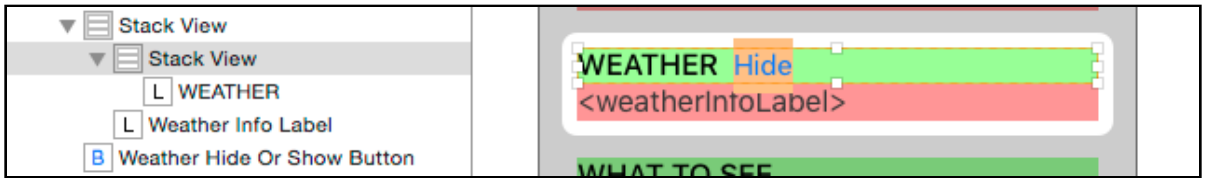
However, you *do* want the bottom **<weatherInfoLabel>** to fill the stack view.

You can accomplish this by embedding just the **WEATHER** label into a vertical stack view. Remember that the `alignment` of a vertical stack view can be set to `.Leading`, and if the stack view is stretched beyond its intrinsic width, its contained views will remain aligned to its leading side.

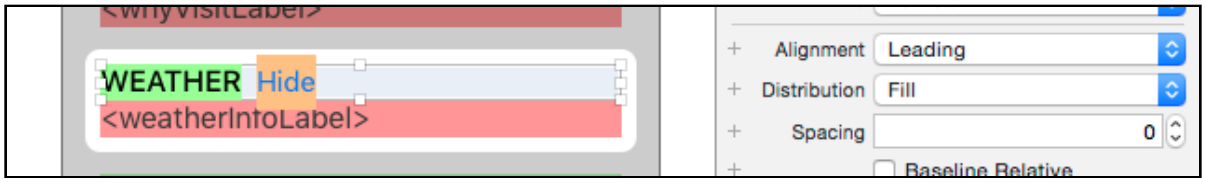
Select the **WEATHER** label using the document outline, or by using the **Control-Shift-click** trick you learned in the previous chapter:



Then click on the **Stack** button:

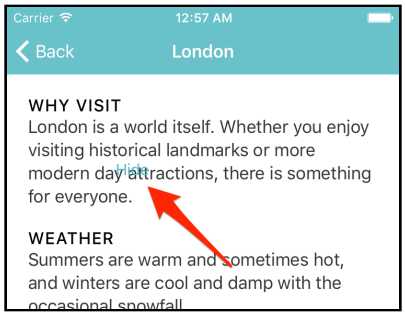


Set **Alignment** to **Leading**, and make sure **Axis** is set to **Vertical**:



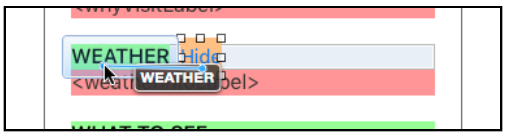
Perfect! You've got the outer stack view stretching the inner stack view to fill the width, but the inner stack view allows the label to keep its original width!

Build and run. Why on earth is the **Hide** button hanging out in the middle of the text?

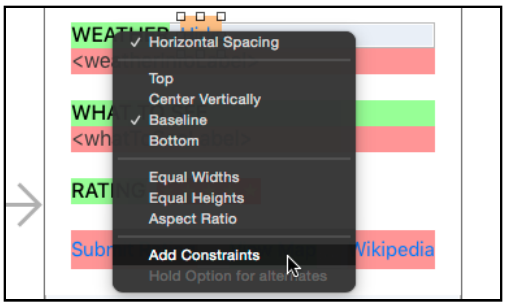


It's because when you embedded the **WEATHER** label in a stack view, any constraints between it and the **Hide** button were removed. So you'll just add them back.

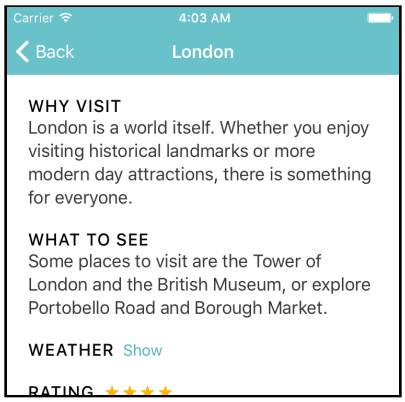
Control-drag from the **Hide** button to the **WEATHER** label:



Hold down **Shift** to select multiple options, and select **Horizontal Spacing** and **Baseline**. Then click on **Add Constraints**:



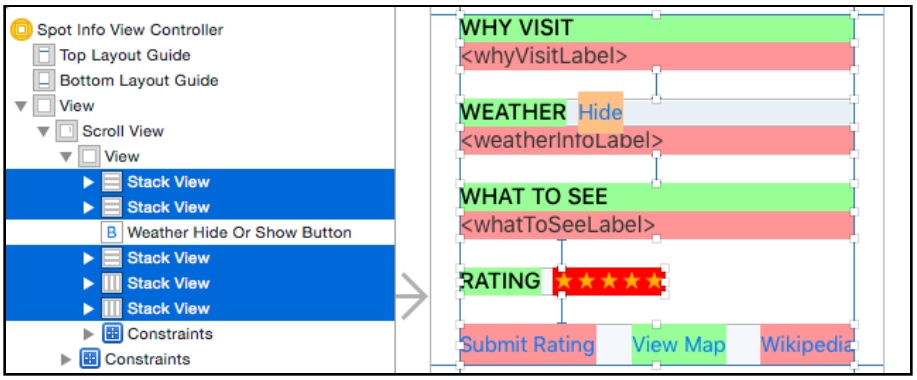
Build and run. The **Hide** button should now be positioned correctly, and since the label that is being set to hidden is embedded in a stack view, pressing **Hide** hides the label, and adjusts the views below it – all without having to manually adjust any constraints.



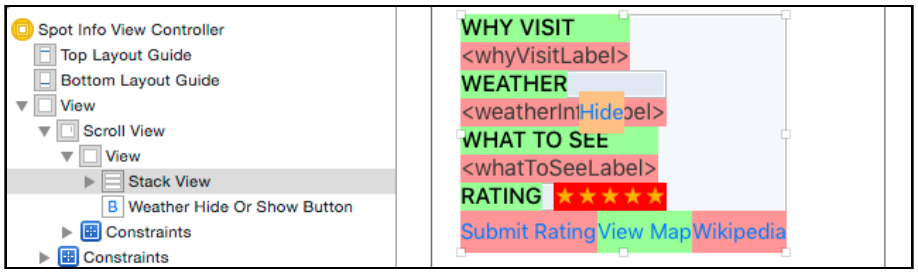
Now that all the sections are in unique stack views, you're set to embed them all into an outer stack view, which will make the final two tasks incredibly simple.

Top-level stack view

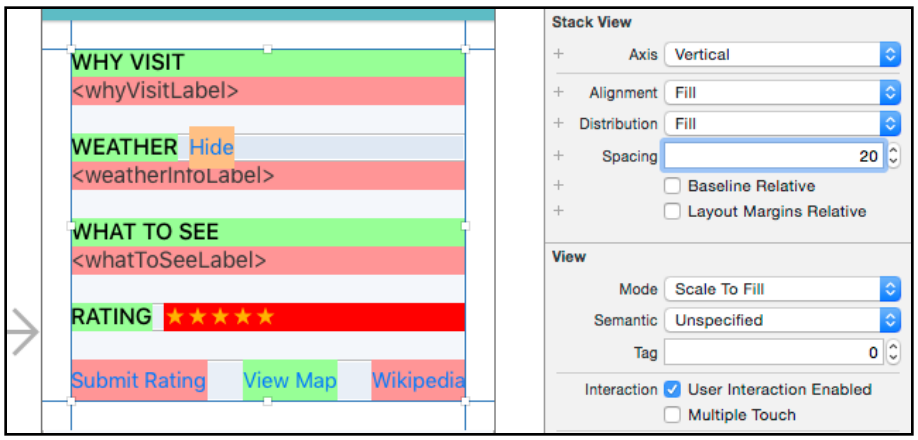
Command-click to select all five top-level stack views in the outline view:



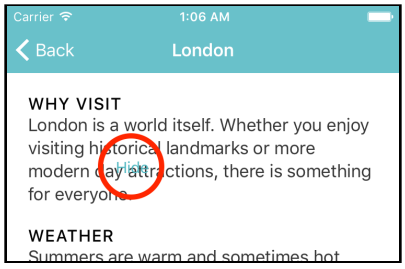
Then click on the **Stack** button:



Click the **Pin** button, checkmark **Constrain to margins** add constraints of **0** to all edges. Then set **Spacing** to **20** and **Alignment** to **Fill**. Your storyboard scene should now look like this:

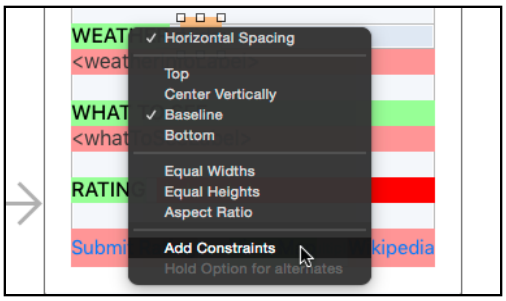


Build and run:



Whoops! Looks like the hide button lost its constraints again when the **WEATHER** stack view was embedded in the outer stack view. No biggie, just add constraints to it again in the same way you did before.

Control-drag from the **Hide** button to the **WEATHER** label, hold down **Shift**, select both **Horizontal Spacing** and **Baseline**. Then click on **Add Constraints**:

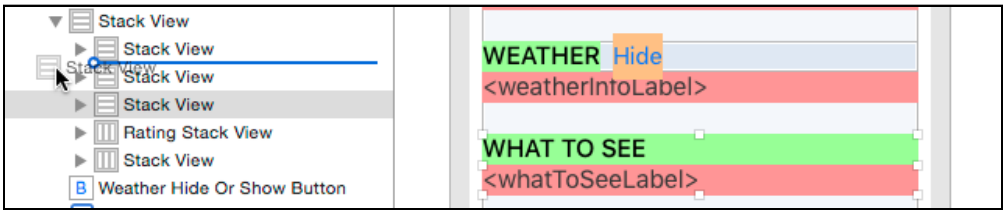


Build and run. The **Hide** button is now behaving itself.

Repositioning views

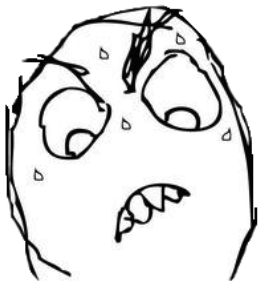
Now that all of the sections are in a top-level stack view, you'll modify the position of the **what to see** section so that it's positioned above the **weather** section.

Select the **middle stack view** from the outline view and **drag it between** the first and second view.



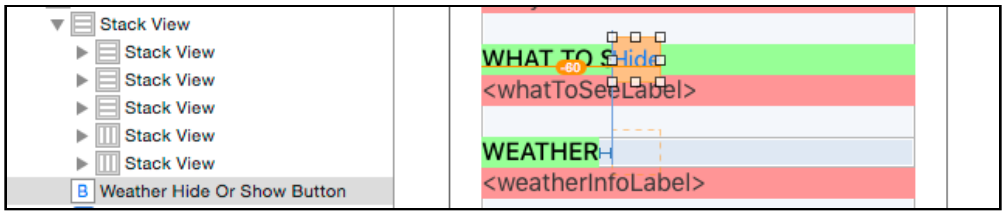
> **Note:** Keep the pointer slightly to the left of the stack views that you're dragging it between, so that it remains a *subview* of the outer stack view. The little blue circle should be positioned at the left edge between the two stack views and not at the right edge:

And now the **weather** section is third from the top, but since the **Hide** button isn't part of the stack view, it won't be moved, so its frame will now be misplaced and the Hide button will look like it's lost its mind again.

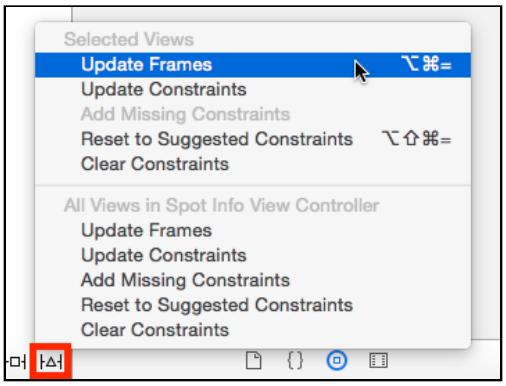


Seriously? Not again.

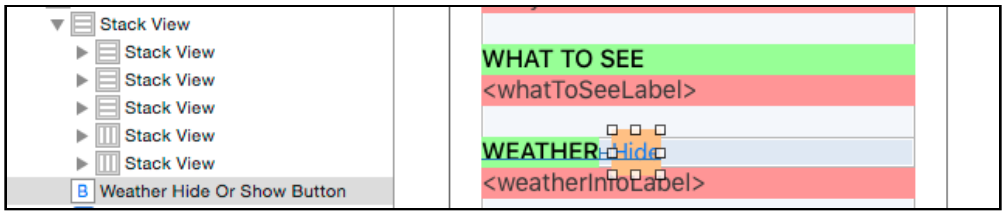
Click on the **Hide** button to select it:



Then click on the **Resolve Auto Layout Issues** triangle shaped button in the Auto Layout toolbar and under the **Selected Views** section, click on **Update Frames**:



The **Hide** button will now be back in the correct position:



Granted, repositioning the view with Auto Layout and re-adding constraints would not have been the most difficult thing you've ever done, but didn't this feel oh-so-much nicer?

Arranged subviews

Okay, back away from Xcode — it's time for some theory!

`UIStackView` has a property named `arrangedSubviews`, and it also has a `subviews` property since it's a subclass of `UIView` — which begs the question about how these two properties differ.

The `arrangedSubviews` array contains the subviews that the stack view lays out as part of its stack. The order in the array represents the ordering within the stack view, whereas the ordering in the `subviews` array represents the front-to-back placement of the subviews, i.e. the z-axis order.

Also, `arrangedSubviews` is always a subset of the `subviews` array. (You can't be an

arranged subview if you're not even a subview!) Anytime a view is added to `arrangedSubviews` it's automatically added to `subviews`, but not vice versa.

The outline view in a storyboard for a stack view actually represents its `arrangedSubviews`. So, it's not possible to add a view only to the stack view's `subviews` using the storyboard; you'd have to do that in code.

When might you want to add a subview to a stack view, but not to its `arrangedSubviews`? One possible case could be to add a background view.

Views can programmatically be added to the stack view, i.e. to `arrangedSubviews`, by using `addArrangedSubview(_:)` or `insertArrangedSubview(_:atIndex:)`.

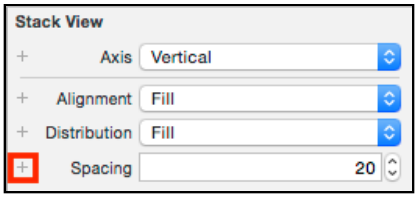
To remove an arranged subview, you can use `removeArrangedSubview(_:)`, however, using this method doesn't remove the view from `subviews`, so it doesn't actually get removed from the view hierarchy until you call `removeFromSuperview()` on the view.

And since it's not possible to have a view in `arrangedSubviews` that's not in `subviews`, you can take the shortcut of just calling `removeFromSuperview()` on the subview, since this will remove it from `arrangedSubviews` as well as from `subviews`.

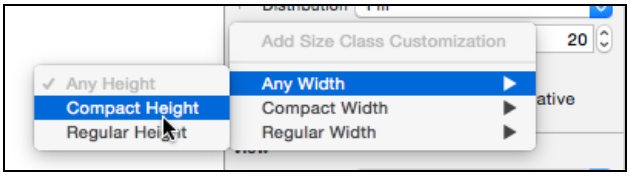
Size class based configuration

Finally you can turn your attention to the one remaining task on your list. In landscape mode, vertical space is at a premium, so you want to bring the sections of the stack view closer together. To do this, you'll use size classes to set the spacing of the top-level stack view to **10** instead of **20** when the vertical size class is compact.

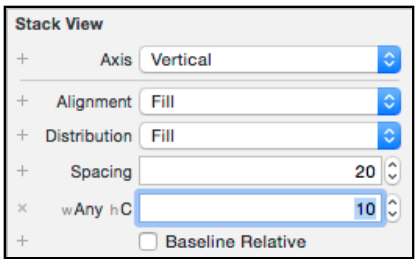
Select the top-level stack view and click on the little **+** button next to **Spacing**:



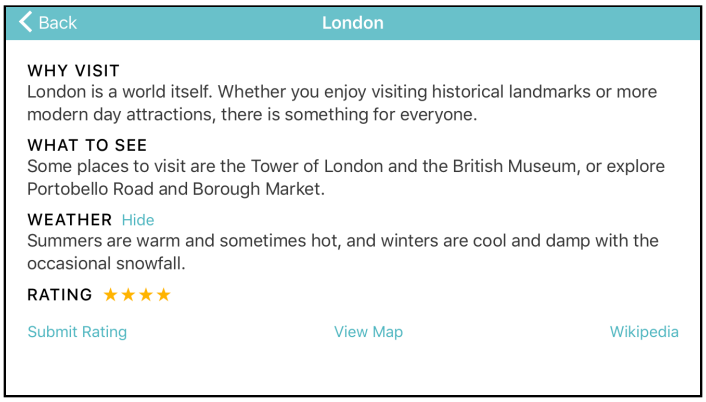
Choose **Any Width** > **Compact Height**:



And set the **Spacing** to **10** in the new **wAny hC** field:



Build and run. Portrait mode should be unchanged, so rotate the simulator (⌘←) to see your handiwork. Note that spacing between the sections has decreased and the buttons now have ample space from the bottom of the view:



If you didn't add a top-level stack view, you still *could* have used size classes to set the vertical spacing to 10 on each of the four constraints that separate the five sections, but isn't it so much better to set it in just a single place?

You have better things to do with your time, like animation!

Animation

Currently, it's a bit jarring when hiding and showing the weather details. It's the perfect place to add some animation to smooth the transition.

Animating hidden

Stack views are fully compatible with the UIView animation engine. This means that animating the appearance/disappearance of an arranged subview, is as simple as toggling its `hidden` property *inside* an animation block.

It's finally time to write some code again! Open **SpotInfoViewController.swift** and take a look at `updateWeatherInfoViews(hideWeatherInfo:animated:)`. When the user taps **Hide**, the current state gets saved. In `viewDidLoad()` this method gets called with `animated: false` and when the button is pressed it gets called with `animated: true`, so the method already receives the `animate` parameter correctly.

You'll see this line at the end of the method:

```
weatherInfoLabel.hidden = shouldHideWeatherInfo
```

Replace it with the following:

```
if animated {
  UIView.animateWithDuration(0.3) {
    self.weatherInfoLabel.hidden = shouldHideWeatherInfo
  }
} else {
  weatherInfoLabel.hidden = shouldHideWeatherInfo
}
```

Build and run, and tap the **Hide** or **Show** button. This looks much nicer, but why not take it a step further by adding some bounce?

Replace the following three lines:

```
UIView.animateWithDuration(0.3) {
  self.weatherInfoLabel.hidden = shouldHideWeatherInfo
}
```

With the following:

```
UIView.animateWithDuration(0.3,
  delay: 0.0,
  usingSpringWithDamping: 0.6,
  initialSpringVelocity: 10,
  options: [],
  animations: {
    self.weatherInfoLabel.hidden = shouldHideWeatherInfo
  }, completion: nil
)
```

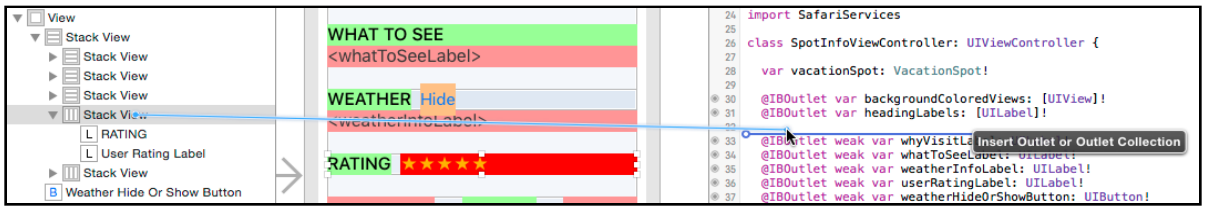
Build and run. You should now see a nice, subtle bounce when you tap the button.

In addition to animating the hidden property on views contained within the stack view, you can also animate properties on the stack view itself, such as `alignment`, `distribution` and `spacing`.

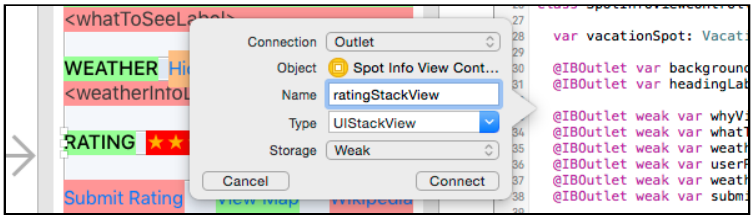
You can even animate the `axis`, and in fact, that's what you'll do next.

Animating the axis

Open **Main.storyboard** and locate the stack view for the **rating** section in the outline view. Open the assistant editor and **Control-drag** to **SpotInfoViewController** to create an outlet:



Name the outlet **ratingStackView** and click on **Connect**:



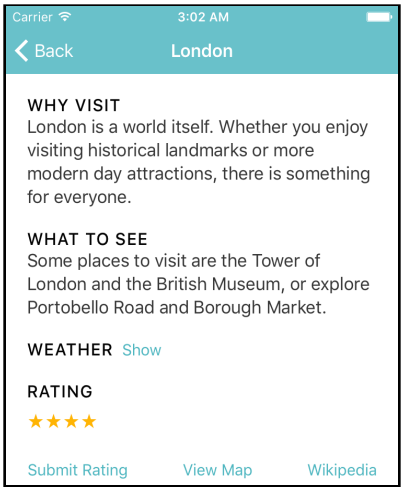
Close the assistant editor and now open **SpotInfoViewController.swift** in the main editor. In `updateWeatherInfoViews(hideWeatherInfo:animated:)` replace the following nil completion block:

```
}, completion: nil
```

With the following code:

```
}, completion: { finished in
  UIView.animateWithDuration(0.3) {
    self.ratingStackView.axis =
      shouldHideWeatherInfo ? .Vertical : .Horizontal
  }
}
```

Once the initial hide or show animation completes, if the weather info was just hidden, then the `axis` of `ratingStackView` animates to horizontal. When the weather is shown again, the `axis` will be set back to vertical.



Add the following lines immediately below the existing line in the `else` clause:

```
} else {
  weatherInfoLabel.hidden = shouldHideWeatherInfo
  ratingStackView.axis =
    shouldHideWeatherInfo ? .Vertical : .Horizontal
}
```

This sets the `axis` of the `ratingStackView` correctly when the view first appears.

Where to go from here?

In this chapter, you continued your dive into stack views and learned about the various properties that a stack view uses to position its subviews. Stack views are highly configurable, and there may be more than one way achieve the same result.

The best way to build on what you've learned is to experiment with various properties yourself. Instead of setting a property and moving on, see how playing with the other properties affects the layout of views within the stack view.

One way to speed up your learning is to test yourself, so before you change a property on a stack view, quiz yourself mentally to see if you can predict the change that will occur, and then see if your expectations match reality.

Stack views are now your new view hierarchy building blocks. Get to know them — and know them well. Really, just make them your new best friend.

Here are some related videos from WWDC 2015 that may be of interest:

- Mysteries of Auto Layout, Part 1 apple.co/1D47aKk
- Mysteries of Auto Layout, Part 2 apple.co/1HTcVJy
- Implementing UI Designs in Interface Builder apple.co/1H5vS84

Chapter 9: What's New in Storyboards?

By Caroline Begbie

Storyboards have been around since iOS 5 and have received lots of upgrades and new features since then, including unwind segues for reverse navigation, universal storyboards for both iPhone and iPad, and live rendering of views designed in code.

Xcode 7 brings new features for storyboards that let you do the following:

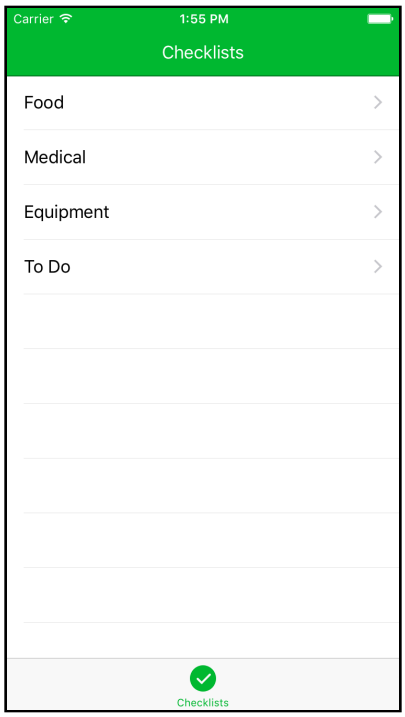
- Refactor a single storyboard into multiple storyboards and link them visually via *storyboard references*.
- Add supplementary views to a view controller using the *scene dock*.
- Add *multiple buttons* to a navigation bar, right from the storyboard itself!

You'll learn how to use the above features as you update an app designed to help you with all those listable moments in life, whether it's grocery shopping, packing your luggage for vacation, or a survival checklist for the impending zombie apocalypse! :]

To get the most out of this chapter you should have some basic storyboard and table view knowledge. Need a quick brush-up? Check out our *Storyboards Tutorial in Swift* at raywenderlich.com/113388.

Getting started

Open the starter project for this chapter and run it in the simulator; tap one of the displayed checklists to view the items contained within, then tap any entry to check it off. Done and done!



Take a quick look at the code to get your bearings.

ChecklistsViewController.swift displays the initial list of checklists, and **ChecklistDetailViewController.swift** displays the items within each list. **Main.storyboard** contains the user interface items.

There are two unused scenes in the storyboard; you'll use those later in the tutorial.

The app is not quite complete; your task in this chapter is to improve it so you can add items to a list, add notes to an item, delete an item and add diary entries to record your zombie survival efforts.

Storyboard references

If you've used storyboards on a large project or as part of a team with other developers, you'll know they can quickly become unwieldy. Merge conflicts, spaghetti-like segue arrows and navigating your way around a wall of scenes is enough to make anybody question whether storyboards are worth the effort.

Although you've always been able to use multiple storyboards in your apps, you've never been able to segue between them using Interface Builder. To present a view controller from a different storyboard, you'd have to instantiate it first and present it in code. But no longer!

With Xcode 7, you can add references between storyboards right in Interface Builder using **storyboard references**, which can either point to specific view

controllers or to the initial view controller within another storyboard. This makes it much easier to divide up storyboards into smaller storyboards, and alleviates many of the issues mentioned above without needing to add any extra code.

Multiple smaller storyboards also make it possible for other team members to work independently on their own storyboards without stepping on each other's toes.

Enough theory — time to put it into practice!

> **Note**: Storyboard references are actually backwards-compatible to iOS 8. However, in iOS 8 you can't use a storyboard reference with a relationship segue, or use it to point to storyboards in external bundles.

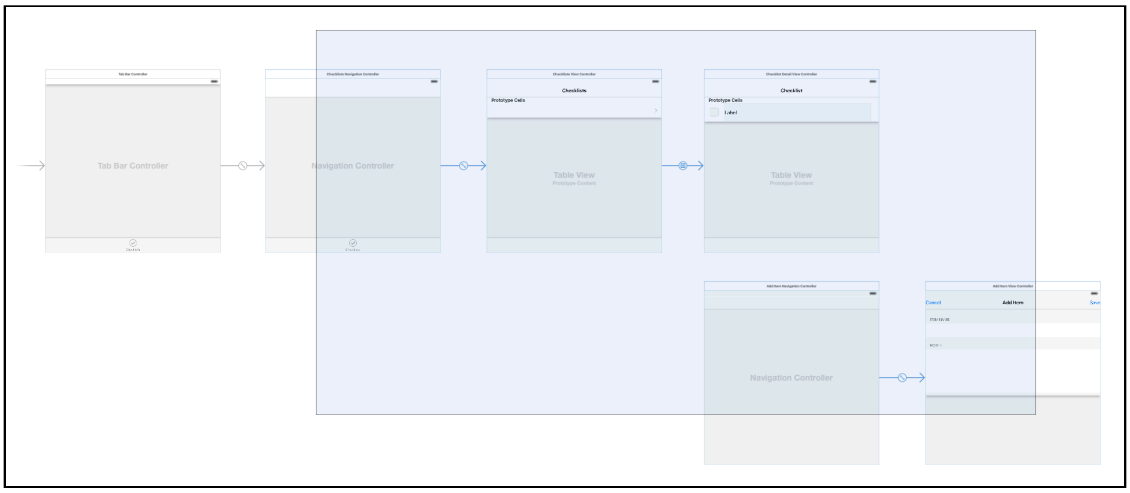
Creating your first storyboard reference

In its current state, **Prepped** is a small app in the early stages of development, but there's enough structure there to discern where to divide up the main storyboard. Container view controllers are a good place to consider splitting out functionality into new storyboards.

Prepped uses a tab bar controller, and in this case it makes sense to separate each tab's children into their own storyboards.

Open **Main.storyboard** and zoom out so you can see all six scenes. Hold **Command** and press **+** to zoom in and **-** to zoom out, or **right-click** on a blank area in the storyboard and choose your zoom level.

Click and drag to highlight all scenes in the storyboard except for the tab bar controller on the left-hand side:

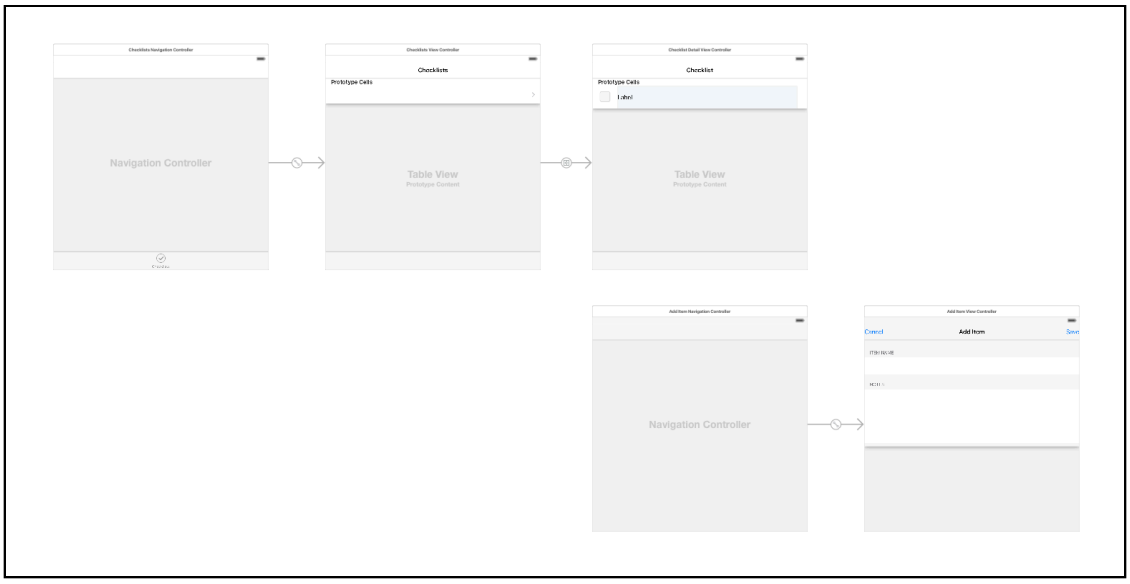


Select **Editor\Refactor to Storyboard** and enter **Checklists.storyboard** as the name of the new storyboard. Set the **Group** to **Checklists**, then click **Save**.

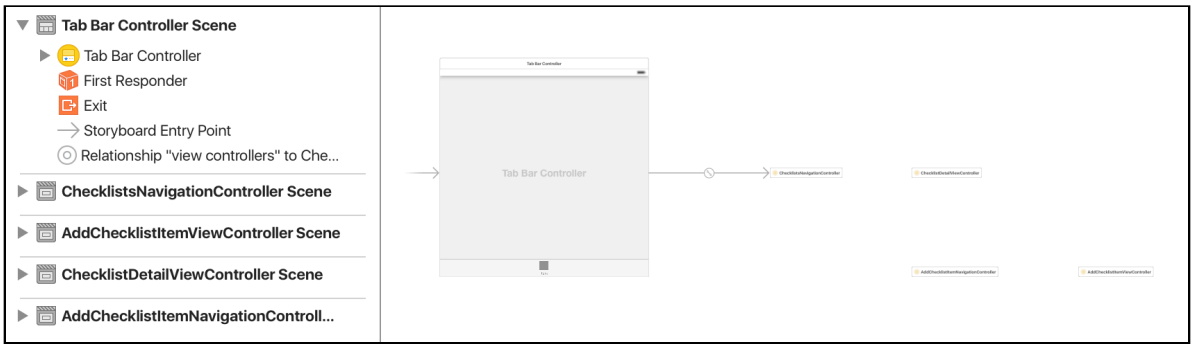
As if by magic, Xcode does the following:

1. Splits out the selected scenes into a new storyboard.
2. Changes the target of the tab bar controller's "view controllers" segue to a storyboard reference that points to the relevant scene in the new storyboard.
3. Takes you to the new storyboard.

You may have to zoom out and reposition the new storyboard to see all of its scenes. The arrangement of the scenes and their segues is exactly like it was in the original storyboard. Here's what the new storyboard should look like:



But what happened to the original storyboard? Open **Main.storyboard** and take a look:



The tab bar controller's "view controllers" segue now points to the storyboard reference for the navigation controller in **Checklists.storyboard**. The storyboard reference uses the navigation controller's storyboard ID to determine which scene to segue to in the new storyboard.

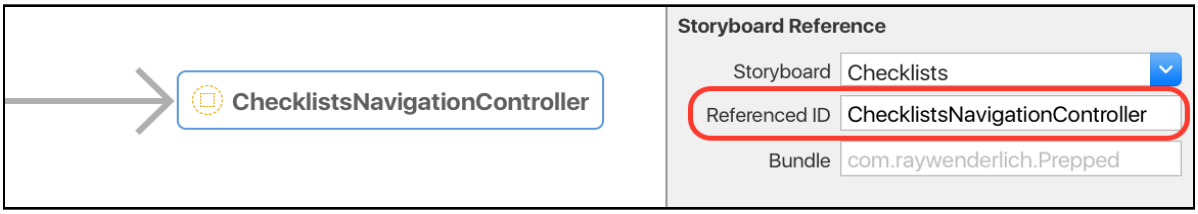
There are a few 'dangling' storyboard references to view controllers that had

storyboard IDs set; you won't need these any longer. Select `ChecklistDetailViewController`, `AddChecklistItemNavigationController` and `AddChecklistItemViewController` and delete them.

> **Note**: If a scene has an empty storyboard ID, the **Refactor to Storyboard** command automatically generates an ugly one, such as *UIViewController-gtY-c7-gYu*. You can change this later, but it's much easier to keep track of things when you explicitly set the storyboard IDs yourself.

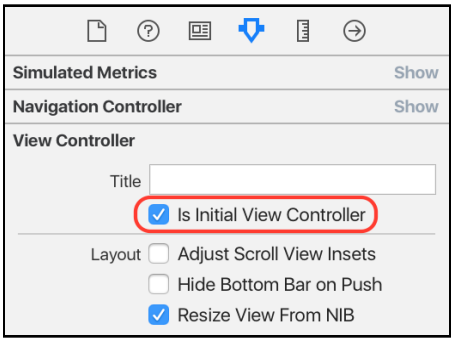
Instead of referencing specific view controllers, storyboard references can simply refer to the initial scene in a storyboard.

Still in **Main.storyboard**, select the new storyboard reference named **ChecklistsNavigationController** and use the **Attributes Inspector** to remove the **Referenced ID**, like so:



The reference now points to the initial view controller in **Checklists.storyboard**, and updates as shown:

Open **Checklists.storyboard** and select the **Checklists Navigation Controller** scene. Use the **Attributes Inspector** to check **Is Initial View Controller**; this indicates this scene should be the entry point for the storyboard.



> **Note**: The initial view controller of a storyboard has an arrow pointing to it from the left-hand side.

Build and run your project; the app performs just as it did when you started. The only difference is that things are a little more organized behind the scenes!

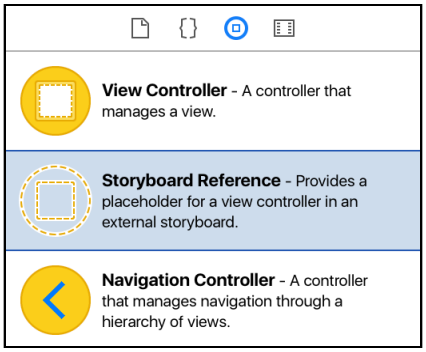
Storyboards within a team

Distributed development of storyboards has always been a challenge; in fact, many developers still avoid storyboards out of fear of the dreaded merge conflict. But storyboard references can help you avoid the complications of team storyboard development.

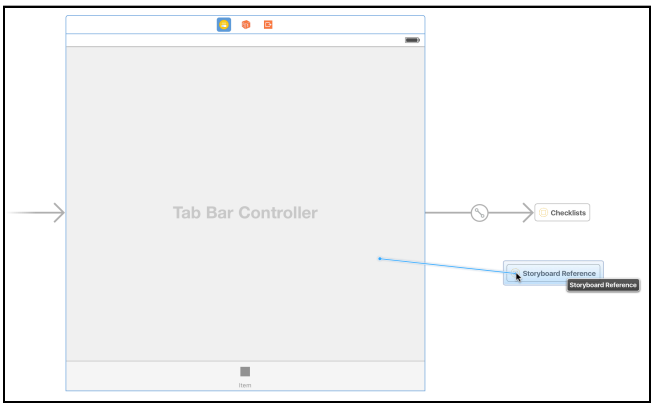
Consider the following scenario: you're writing Prepped with a fellow apocalypse survivor, whose task it is to create the functionality to handle the diary entries. She's built it using a separate storyboard, and now you need to add it to your own storyboard hierarchy...before the zombies descend upon your little enclave.

In the project navigator, select the top level **Prepped group**, located just below the project itself. Click **File\Add Files to "Prepped"**. Navigate to the Prepped folder, and select the **Diary** folder. Ensure that **Copy items if needed** is checked in the dialog box, and that **Added folders** is set to **Create groups**. Ensure that **Add to targets** is ticked for **Prepped**. Click **Add** to add the folder and its contents to the project.

In **Main.storyboard**, drag a **storyboard reference** from the **Object Library** into an empty space on the storyboard:

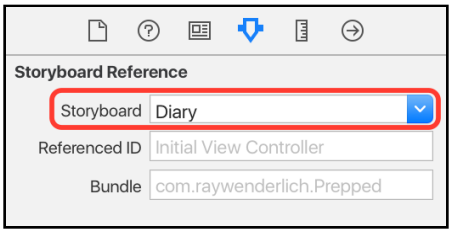


Ctrl-drag from the existing tab bar controller scene to the **storyboard reference**:

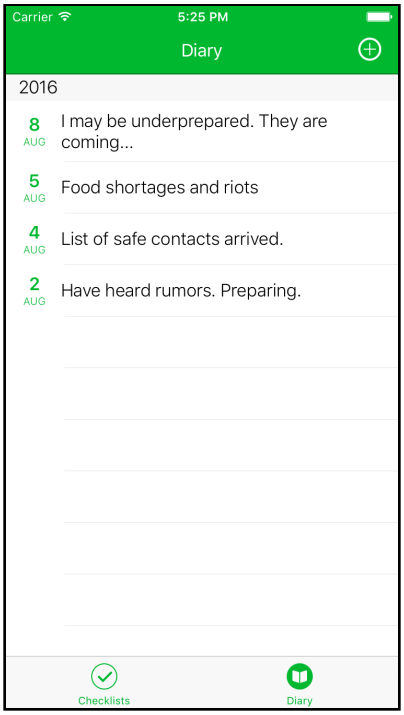


In the pop-up that appears, choose **view controllers** from the **Relationship Segue** section.

Select the **storyboard reference** you just added. In the **Attributes Inspector** set the **Storyboard** to **Diary**:



Build and run your app; you'll see one tab to handle Checklists, and another tab for the Diary entries – the functionality your teammate worked on. You can now add Diary entries using the storyboard scenes and code created by your sister-in-arms:

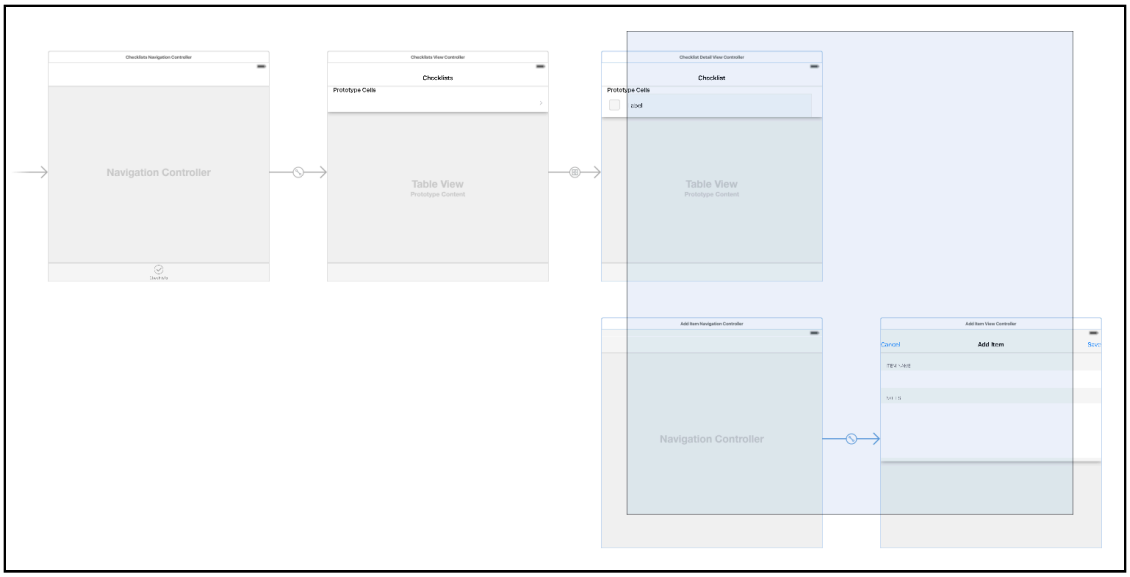


Note: Currently both tabs in the tab bar controller in the storyboard display the title *Item*. The proper title will be loaded at runtime from the Checklists and Diary storyboards. You can change the titles in **Main.storyboard** for your own reference, but it won't make any difference at runtime.

Focusing on a storyboard

Isn't it annoying when you have to tap through a bunch of scenes in your app, when you're just trying to test one single scene buried deep in the stack? With storyboard references you can isolate the scenes you're interested in into their own storyboard and instruct the app to launch straight into that. You'll do that now for the checklist item section.

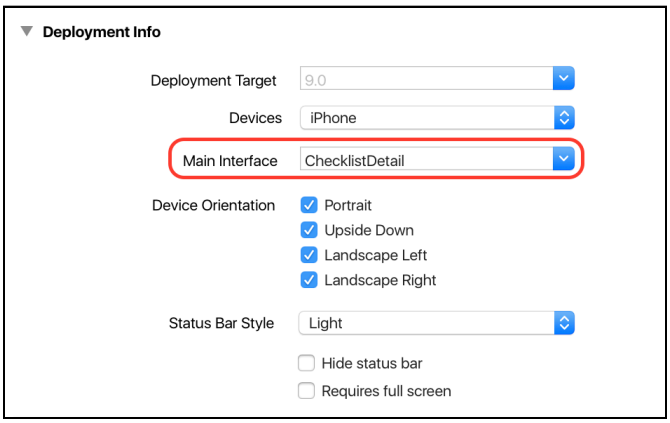
In **Checklists.storyboard** highlight the **Checklist Detail View Controller**, **Add Item Navigation Controller** and **Add Item View Controller** scenes:



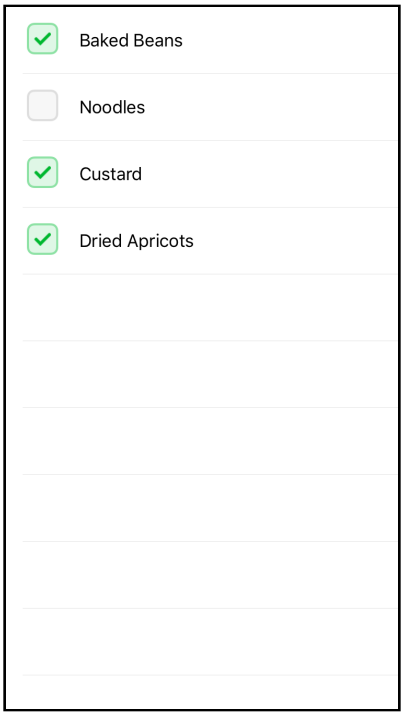
Select **Editor\Refactor to Storyboard** and name the new storyboard **ChecklistDetail.storyboard**. Ensure that the **Group** is still set to **Checklists**.

Just as you did for the Checklists storyboard, select the **Checklist Detail View Controller** scene in **ChecklistDetail.storyboard**, and use the **Attributes Inspector** to check **Is Initial View Controller**. The Checklist Detail View Controller should now have an arrow on its left to indicate it's the first scene in the storyboard.

Click on the **Prepped project** at the top of the project navigator, then click on **Prepped target** and choose the **General** tab. Change **Main Interface** to **ChecklistDetail.storyboard**:



Build and run your app; you'll see the checklist detail scene loads first:

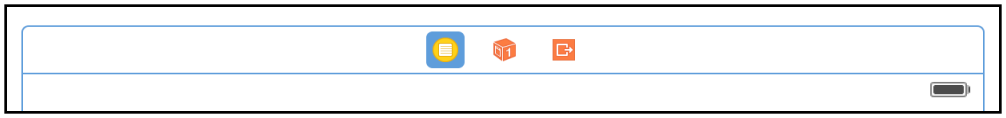


Where are the navigation and tab bar? Since the view controller is no longer embedded in a navigation or tab bar controller, you won't see those two elements while you're working on the items storyboard.

> **Note:** This approach will fail if the initial view controller in the chosen storyboard requires data provided via a segue. In this project, `ChecklistDetailViewController` has already been set up to load initial sample data.

Views in the scene dock

A lesser-known feature of storyboard scenes is the **scene dock**. Most people don't even notice it's there - did you? You'll find it at the top of the currently selected scene in a storyboard:

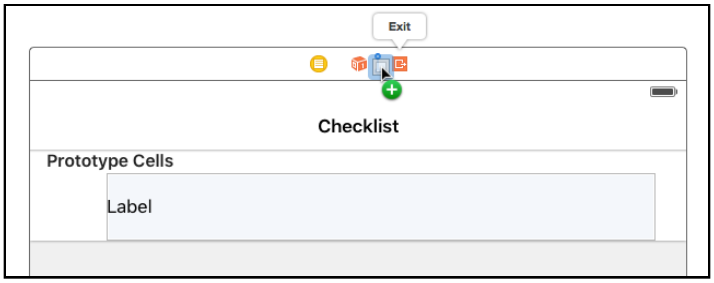


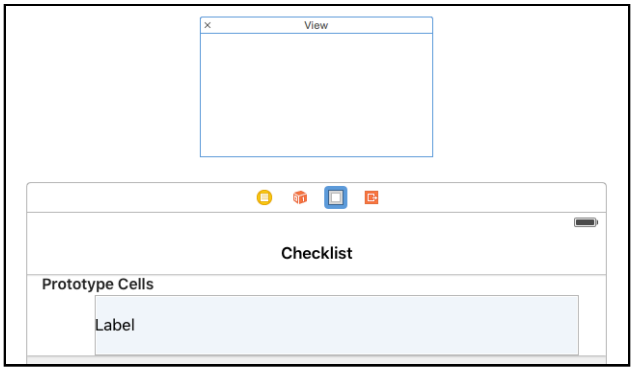
Out of the box, the scene dock contains references to the current view controller, the first responder, and any available unwind segues. But did you know you can add your own views to the scene dock? You've always been able to do so, but Xcode 7 lets you design these attached views within Interface Builder.

Any views you add in the scene dock won't be added to your view controller's initial subviews array; instead, you can add IBOutlets to them and make use of them at runtime.

Selecting a checklist item in Prepped highlights its table row with a boring gray color. You will now perform the amazing feat of changing the color of the selected row with no code at all — thanks to the scene dock!

In **ChecklistDetail.storyboard**, select **Checklist Detail View Controller** and drag a **view** from the Object Library onto the scene dock:



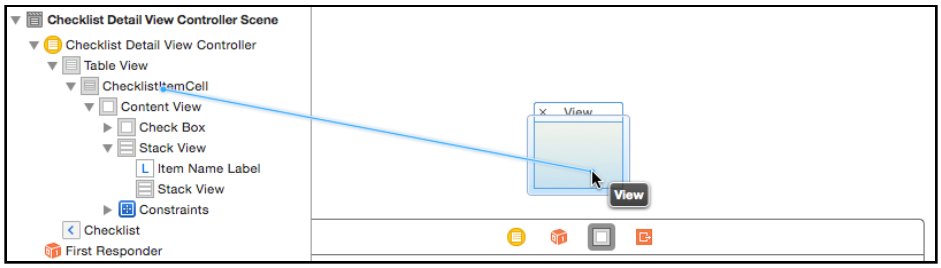


The new view will appear just above the scene dock. You can add subviews and controls to these docked views, just as you would any other view.

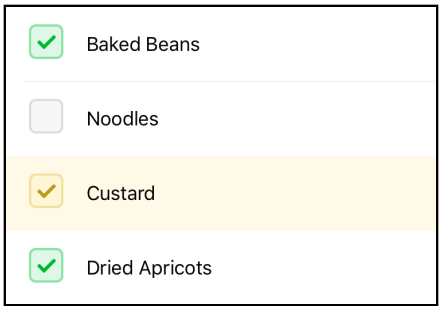
Select the view you added and use the **Attributes Inspector** to change the background color of the view to **#FFFAE8**.

The size of the view in the storyboard doesn't really matter, since it will be stretched automatically when it's used in the cell. However, if you want it to take up less room you can resize it by dragging its top, left and right edges.

In the document outline, **Ctrl-drag** from **ChecklistItemCell** to the new view. Choose **selectedBackgroundView** from the connections pop-up:



Build and run your app; tap any row, and it's highlighted with by your new view. Pretty neat — and without a stitch of code!



Note: This coloring method will only work for table views that don't have multiple selection enabled. Only one instance of the colored view is created, and it's shared between each cell in the table view. As such, it can only be applied to one cell at a time.

Conditional views using the scene dock

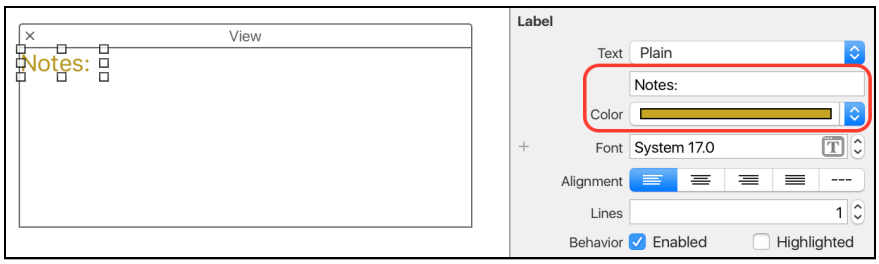
Often, you'll have a view that you only want to show under certain conditions. Designing a view like this amongst all the other views in a view controller was always rather difficult in storyboards. The advantage of having a view in the scene dock is that you can create it visually without interfering with the rest of your view controller's subviews. You can then add it to the view hierarchy in code when it's needed.

The checklist items in Prepped's sample data have notes accompanying them; you're now going to create a view to display an item's note. When you tap the table

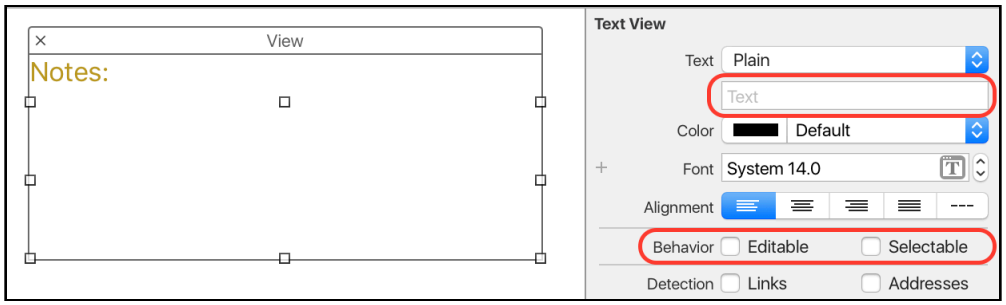
view row for an item, the row will expand to display the associated note. Tapping the row again or tapping a different row collapses the row and removes the note view from that row.

Still in **ChecklistDetail.storyboard**, drag a new **view** onto the scene dock, next to the selected background view you created in the last section. Select the view, and use the **Size Inspector** to set its width to **320** and its height to **128**.

Drag a **label** from the Object Library onto the new view and use the **Attributes Inspector** to change the label text to **"Notes:"**. You may have to resize the label so that the text fits. Change the label's text color to **#BB991E**:

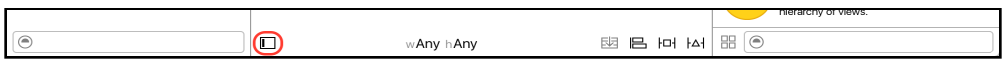


Next, drag a **text view** from the Object Library onto the new view. Remove its default *Lorem ipsum* text using the **Attributes Inspector**. Uncheck **Behavior Editable** and **Selectable**. Resize and rearrange the label and text views so they touch the edges of their container so that it looks like this:



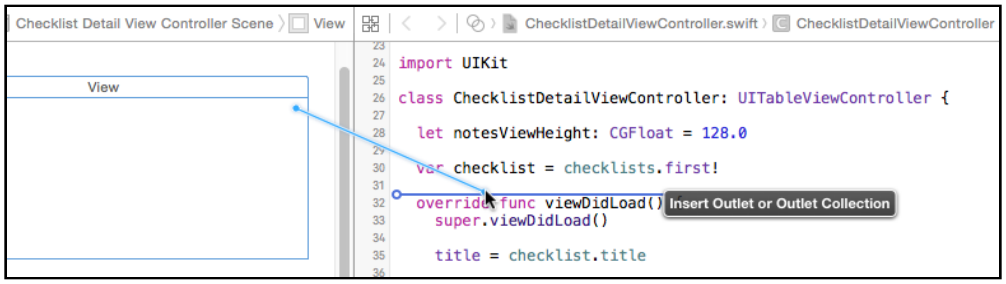
You'll now connect this notes view to an IBOutlet in the view controller. Even though there are multiple cell instances on the screen at one time, there will be only one notes view instance at any time, so it won't be an issue to connect this view to an outlet.

With **ChecklistDetail.storyboard** open in the main editor, open **ChecklistDetailViewController.swift** in the **assistant editor**. You may have to close the **document outline** using the icon beneath the storyboard to get enough space:



Ctrl-drag from the new view to `ChecklistDetailViewController` to create an outlet for the view just below the existing `checklist` property. Ensure that you are dragging from the view's *background*, not from the text view or label. You can also

drag from the view's icon in the scene dock.



Name the outlet `notesView` and click **Connect**. The outlet will appear as a property in `ChecklistDetailViewController`.

Now **Ctrl-drag** from the text view to `ChecklistDetailViewController` to create another outlet just below the one you just made. Name the outlet `notesTextView` and click **Connect**.

Finally, it's time to write some code! :] You'll use another new feature of iOS 9, *UIStackView*, to add and remove the notes view from a cell with an animation.

> **Note:** To learn more about UIStackView, be sure to check out chapter 7, "UIStackView & Auto Layout Changes", and chapter 8, "Intermediate UIStackView".

In **ChecklistDetailViewController.swift**, add the following method to the bottom of the main class implementation:

```
func addNotesViewToCell(cell: ChecklistItemTableViewCell) {
  notesView.heightAnchor
    .constraintEqualToConstant(notesViewHeight)
    .active = true
  notesView.clipsToBounds = true

  cell.stackView.addArrangedSubview(notesView)
}
```

This method ensures Auto Layout defines the the notes view's height, then adds it to the cell's stack view's `arrangedSubviews` collection. It also sets `clipsToBounds` to `true` to prevent the text view from spilling outside of the cell when you perform a swipe-to-delete.

The height needs to be set using Auto Layout since the stack view derives its own height from the heights of its `arrangedSubviews`. If you don't set the height here, the cell won't grow when you add the notes view.

Next, add the following method below `addNotesViewToCell(_:)`:

```
func removeNotesView() {
  if let stackView = notesView.superview as? UIStackView {
    stackView.removeArrangedSubview(notesView)
    notesView.removeFromSuperview()
  }
}
```

This removes the notes view from the stack view's `arrangedSubviews` as well from its set of visible subviews.

Next, you need to put these methods to use. Still in **ChecklistDetailViewController.swift**, find the table view delegate extension for `ChecklistDetailViewController` and add the following code:

```
override func tableView(tableView: UITableView,
  didSelectRowAtIndexPath indexPath: NSIndexPath) {

  // 1
  guard let cell = tableView.cellForRowAtIndexPath(indexPath)
    as? ChecklistItemTableViewCell else { return }

  // 2
  tableView.beginUpdates()
  // 3
  if cell.stackView.arrangedSubviews.contains(notesView) {
    removeNotesView()
  } else {
    addNotesViewToCell(cell)

    // 4
    notesTextView.text = checklist.items[indexPath.row].notes
  }

  // 5
  tableView.endUpdates()
}
```

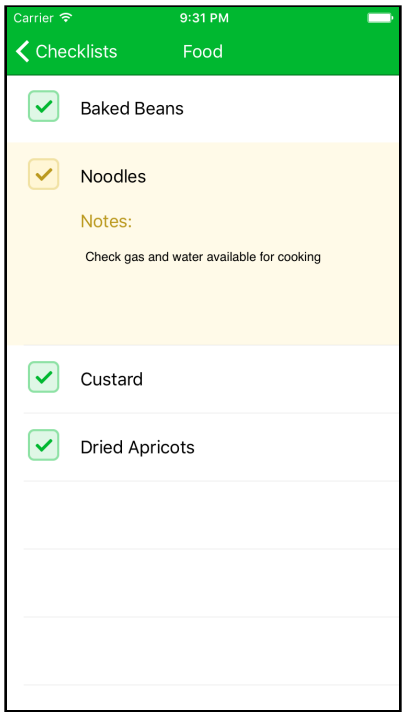
This method does the following:

1. Uses a Swift 2.0 `guard` statement to ensure that there is a valid cell of the right type at the selected index path before continuing.
2. Calls `tableView.beginUpdates()` to animate the changes to the cell's height.
3. Removes the notes view if the cell's stack view already contains it; otherwise, add the notes view.
4. Updates the notes text view to contain the notes for the selected checklist item.
5. Finally, calls `tableView.endUpdates()` to commit the changes.

Finally — don't forget that you changed the project's main interface earlier on. To change the project's main interface back to the main storyboard: click on the

Prepped project in the **project navigator**, click on the **Prepped target** and then click on the **General** tab. Change **Main Interface** to **Main.storyboard**:

Build and run your app; tap any cell and you should see the notes view appear. Using a stack view means you didn't need to set any frames manually or add any constraints to the cell other than the one that defines the height of the notes view. In previous versions of iOS, this would've been rather more tricky to implement.



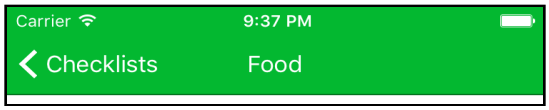
Note: Being able to create a view in the scene dock is useful, but only if it is used solely from one view controller. If the supplementary view is reused throughout the app, you'd be better off using a XIB file that you instantiate in code.

Using multiple bar buttons

The final feature you'll add to your app is the ability to add and delete checklist items. The scene and code for adding a checklist item is already in the starter app, but it's not hooked up to anything yet. That's where you come in.

You need two new buttons on the checklist detail view controller's navigation bar:

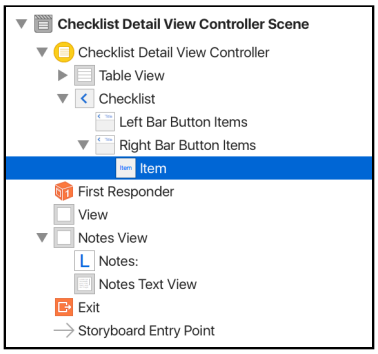
one for Add and one for Edit. Apps often achieve this by having an "Edit" button on the left side of the bar and and an "Add" button on the right of the bar. However, in Prepped the left side of the navigation bar is already being used for the standard navigation back button:



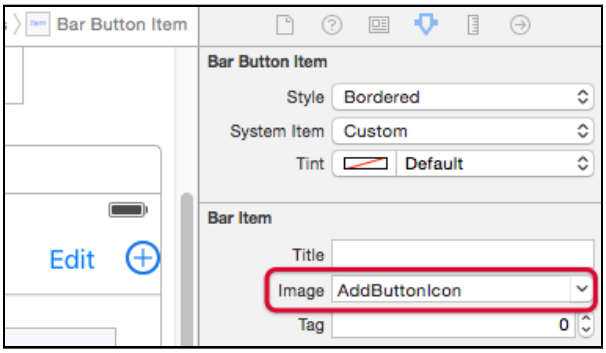
Prior to Xcode 7, you would've had to create a view with multiple buttons and add that view to the navigation bar. Xcode 7 brings another useful new feature to storyboards which makes this extra step unnecessary: the ability to add multiple buttons directly to a navigation bar.

In **ChecklistDetail.storyboard** select **Checklist Detail View Controller** in the document outline. Drag a **bar button item** from the Object Library onto the right hand side of the navigation bar.

The document outline will now show a group for left bar button items and a group for right bar button items:



Drag a second bar button item onto the right side of the navigation bar. Use the **Attributes Inspector** to change the **System Item** of the left of the two buttons to **Edit**. Change the other button's **Image** to **AddButtonIcon**:



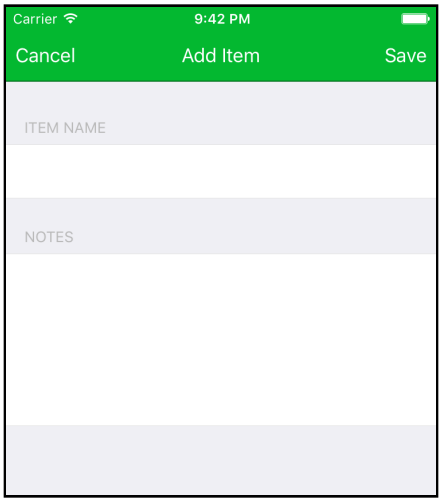
Ctrl-drag from the **Add** button to the **Add Item Navigation Controller** scene and choose **present modally** from the pop-up menu. When the user taps the **Add** button the Add Item scene will appear.

You'll now need to connect a couple of unwind segues to return from the Add Item scene; these unwind methods have already been created for you in **ChecklistDetailViewController.swift**.

Still in **ChecklistDetail.storyboard**, select the **Add Item View Controller** scene in the Document Outline. **Ctrl-drag** from the **Cancel** button on the left side of the navigation bar to **Exit** on the scene dock. Choose `cancelToChecklistDetailViewController:` from the pop-up menu.

Ctrl-drag from the **Save** button on the right side of the navigation bar to **Exit** on the scene dock and choose `saveToChecklistDetailViewController:` from the pop-up menu.

Build and run your app; choose a checklist and try adding items with notes to the list. These won't be saved permanently, because the sample data is currently only held in-memory:



Now you just need to implement the code for the Edit button. First, add the following line to the bottom of `viewDidLoad()` in **ChecklistDetailViewController.swift**:

```
navigationItem.rightBarButtonItems![1] = editButtonItem()
```

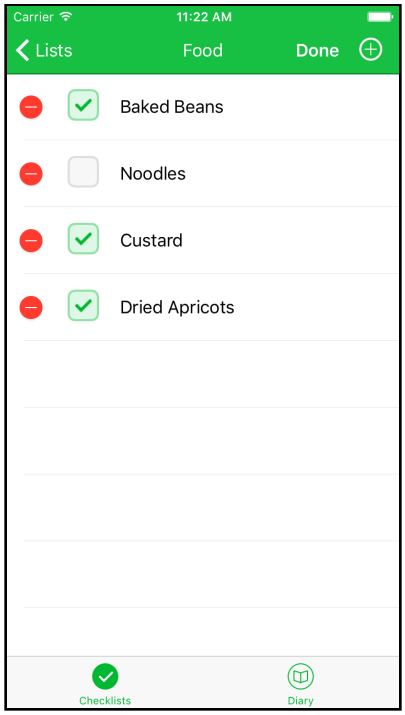
This line replaces the Edit button with the view controller's built-in edit button item. It takes care of animating to and from an 'editing' state and changes the button's text from "Edit" to "Done" and back again as required.

Still in **ChecklistDetailViewController.swift**, find the table view data source extension. Add the following implementation inside the extension, below the existing methods:

```
override func tableView(tableView: UITableView,
  commitEditingStyle editingStyle: UITableViewCellEditingStyle,
  forRowAtIndexPath indexPath: NSIndexPath) {

  if editingStyle == .Delete {
    removeNotesView()

    checklist.items.removeAtIndex(indexPath.row)

    tableView.deleteRowsAtIndexPaths([indexPath],
      withRowAnimation: .Fade)
  }
}
```

This method removes the notes view if it's present, removes the checklist from the view controller's `checklist` array and then tells the table view to delete the row.

Build and run your app; choose a check list, tap the **Edit** button and delete an item from the list. Tap the **Done** button to complete editing.



Where to go from here?

Your app to help you survive the apocalypse is done! All the new features you've covered in this chapter, including storyboard references and an enhanced scene dock should show you there are very few reasons *not* to use storyboards in your own projects.

Storyboards in Xcode 7 also have greater support for custom segues. We've got that covered in chapter 10 of this book: *Custom Segues*. If you decide to make Prepped a universal app, you can read more about supporting multitasking on the iPad in Chapter 5: *Multitasking*.

There are some useful sessions from WWDC 2015 that will help you as well:

- Session 215, What's New In Storyboards: apple.co/1Do4xn7
- Session 407, Implementing UI Designs in Interface Builder: apple.co/1g60D7c

Chapter 10: Custom Segues

By Caroline Begbie

Segues have long been a familiar way to transition between scenes — all the way back to iOS 5. iOS 7 introduced custom view controller transitions to support custom, interactive transitions between views. iOS 9 takes custom transitions even further with custom segues that let you make a complete separation between your transition animation and view controller code.

A small but important change is that segues are now retained during modal or popover presentations of scenes; segues instantiate when presenting a new scene and are held in memory until you dismiss the scene view controller. This means you can move *all* your transition's animation and adaptivity code into a segue class and reuse that segue in any storyboard. When you dismiss the modal scene, the unwind transition will use the presenting segue's transition.

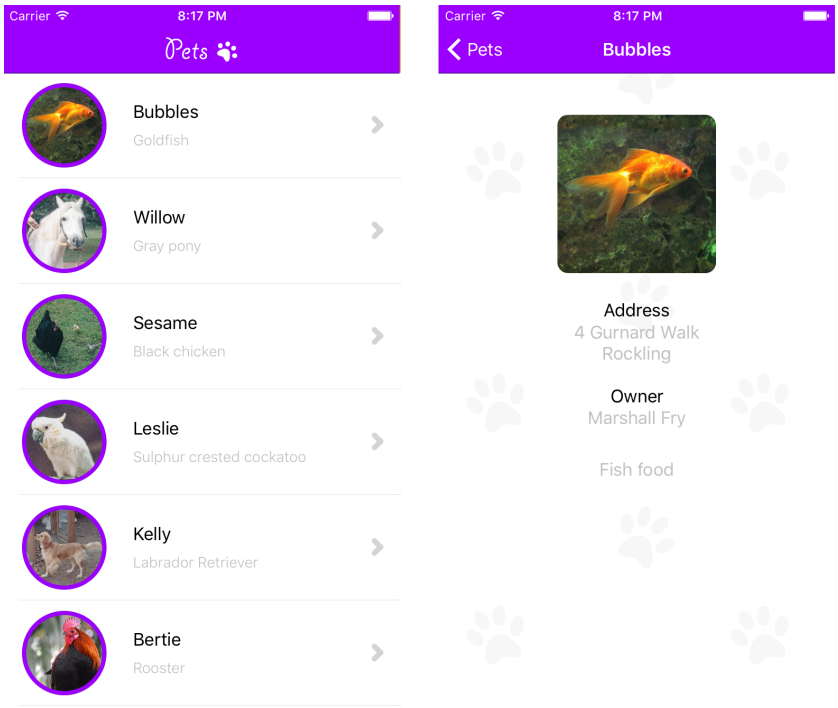
This chapter will show you how to do the following:

- Create a custom segue
- Animate a custom transition within the segue
- Make your segue reusable within navigation and tab controllers

You'll need some basic knowledge of storyboards and segues, but if you understood the previous chapter "What's New In Storyboards?", consider yourself well prepared.

Getting started

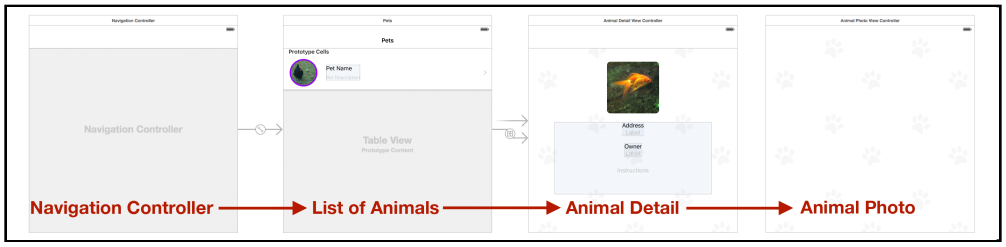
The sample app for this chapter is **PamperedPets**, a simple pet-minding app that, when complete, will display a list of pets to mind and their details:



Explore your starter project for a bit to see how it works. Run the app; you'll see a single scene showing the photo, address and feeding instructions for the star of your show: Bubbles the goldfish.

> **Note**: The project will throw a few warnings related to the Storyboard. Don't panic — you'll hook up the disconnected storyboards later in the chapter.

Have a look at **Main.storyboard**; it has a number of pre-created scenes, but you'll start working with the Animal Detail and Animal Photo scenes:



There aren't any transitions yet - it's your job to add some awesome transitions and make this app shine. And to get you *hooked*, there may be some fish jokes before you *fin*-ish :].

What are segues?

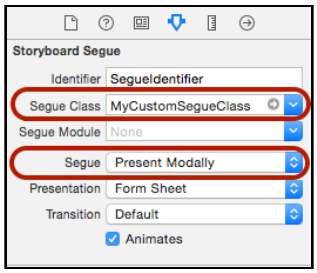
Segues describe transitions between scenes; they show up as the arrows between

view controller scenes. There are several types of segues:

- **Show**: Pushes a scene from a navigation controller.
- **Show Detail**: Replaces a scene detail when in a `UISplitViewController`.
- **Present Modally**: Presents a scene on top of the current scene.
- **Popover**: Presents a scene as a popover on the iPad or full screen on the iPhone.

> **Note**: Relationships between child view controllers embedded in a container view are also shown as arrows on the storyboard, but these types of segues can't be customized.

Segues have always been *either* modal and popover *or* custom. But in iOS 9, you can use the underlying segue type with your custom segue instead of having to define the segue from scratch:



This chapter has you customizing modal segues alone.

A simple segue

Even though you might have used them before, to fully appreciate how segues work you'll first create a basic modal segue, which you will customize later in this chapter.

You'll invoke the segue when the user taps the photo in the `AnimalDetailViewController` scene. The segue will then present the AnimalPhotoViewController scene as a modal controller showing a larger photo.

There are two main parts to this task:

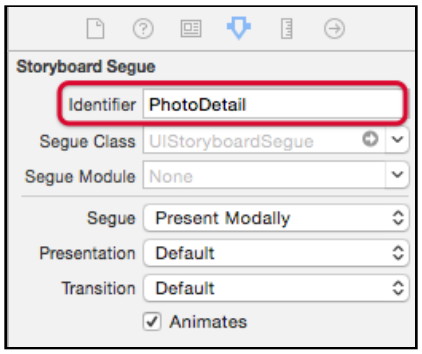
1. Setting up the segue. `prepareForSegue(_:sender:)` triggers when you activate the segue; you'll set up the destination view controller with the necessary data in this method.
2. Performing the destination controller's transition animation. You'll use the default transition initially, but you'll customize it in just a bit.

In **Main.storyboard**, select the **Animal Detail View Controller** scene. Drag a **Tap Gesture Recognizer** from the Object Library onto the **Pet Photo Thumbnail** of Bubbles the fish. This hooks up the tap gesture to the image view.

Next, **Ctrl-drag** from the **Tap Gesture Recognizer** in the document outline to **Animal Photo View Controller**. Choose **present modally** from the popup menu.

That's all it takes to define a segue; now you've just to name the segue and set up `AnimalPhotoViewController` so it shows the correct photo.

Select the segue arrow between the **Animal Detail View Controller** and **Animal Photo View Controller** scenes; use the Attributes Inspector to assign it the identifier **PhotoDetail**:

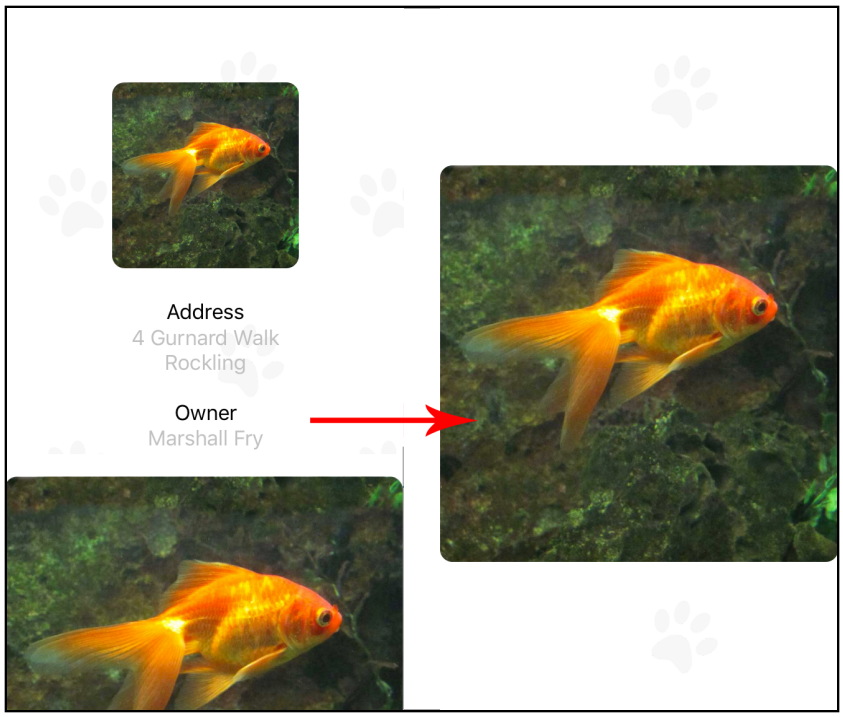


Override `prepareForSegue(_:sender:)` in **AnimalDetailViewController.swift** to set up the destination controller data as shown below:

```
override func prepareForSegue(segue: UIStoryboardSegue,
  sender: AnyObject?) {
  if segue.identifier == "PhotoDetail" {
    let controller = segue.destinationViewController
      as! AnimalPhotoViewController
    controller.image = imageView.image
  }
}
```

Here you give the destination controller — in this case, `AnimalPhotoViewController` — the image to display.

Run the app and tap the photo; you should see a larger photo slide up the screen:



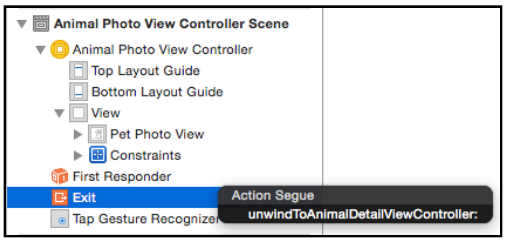
Right now, you have no way to close this screen. You'll need to create a tap gesture to perform an unwind segue.

Add the following method to `AnimalDetailViewController` in **AnimalDetailViewController.swift**:

```
@IBAction func unwindToAnimalDetailViewController(
  segue:UIStoryboardSegue) {
  // placeholder for unwind segue
}
```

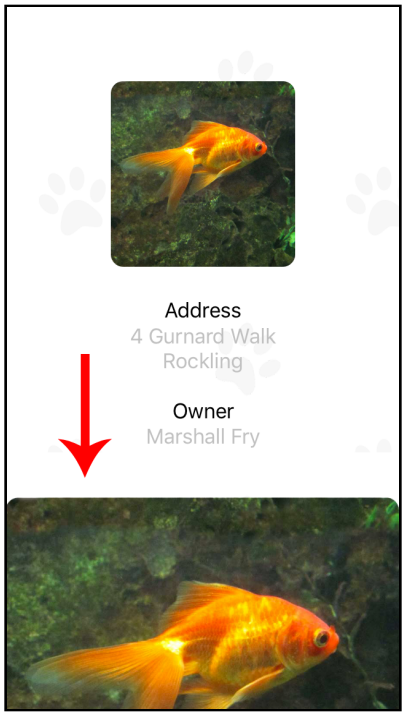
For a simple unwind segue, this method doesn't require any code. Any method with a signature of `@IBAction func methodName(segue:UIStoryboardSegue)` is considered a marker to which a Storyboard segue can unwind.

In **Main.storyboard**, select the **Animal Photo View Controller** scene. Drag a **Tap Gesture Recognizer** from the Object Library onto **Pet Photo View**. Next, **Ctrl-drag** from your new **Tap Gesture Recognizer** in the document outline to **Exit**, then select `unwindToAnimalDetailViewController:` from the popup:

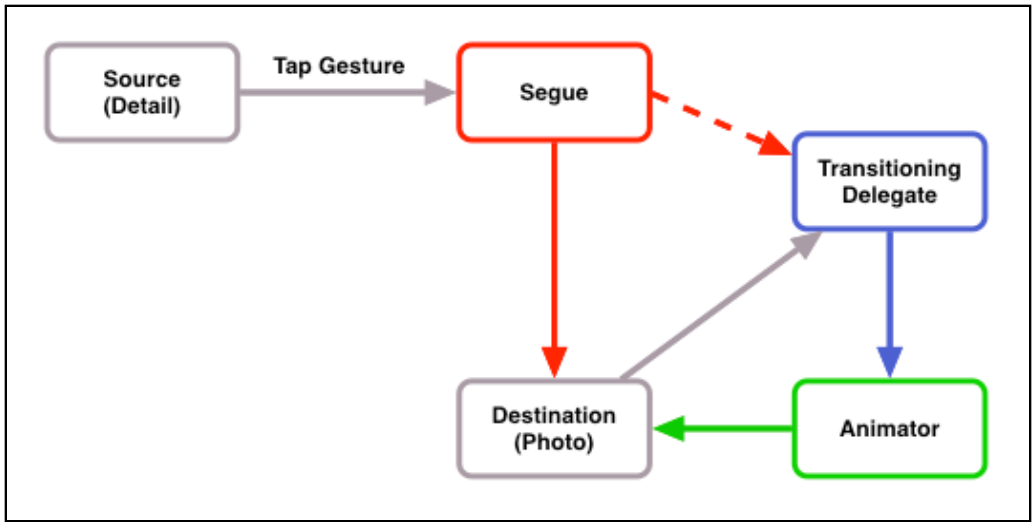


Run the app again and tap the photo; the larger photo appears and a simple tap

unwinds it back to the first scene:



Time to dissect what's happened here. When you tap the thumbnail on the detail view, the tap gesture recognizer initiates a modal segue from `AnimalDetailViewController` to `AnimalPhotoViewController`. `AnimalDetailViewController` is the **source view controller**, while `AnimalPhotoViewController` is the **destination view controller**. The segue holds a reference to *both* the source and destination view controllers:



The segue sets the transitioning delegate of the destination view controller behind the scenes and also sets up its presentation according to the current size class.

The source view controller method `prepareForSegue(_:sender:)` sets up the data for

the destination view controller.

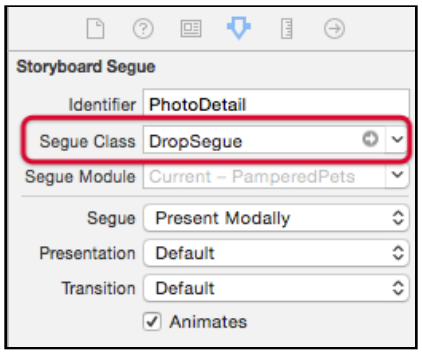
The system then turns control over to the destination view controller. The destination view controller then invokes its transition delegate which sets off the default Cover Vertical animation.

That covers the basic actions behind a segue. Now you can take the working segue and customize it with a segue subclass.

Your custom segue library

A segue exists for the entire duration of a modal or popover presentation, so it's really easy to swap in segues from your library without touching your UIViewController code. The segue can be responsible for both presentation and dismissal transition animations. The starter app contains a pre-made custom segue called `DropSegue` to give you an idea of how easy changing segues can be.

In **Main.storyboard**, select the **PhotoDetail** segue between the Animal Detail and the Animal Photo view controllers. Change **Segue Class** to `DropSegue` as shown below:



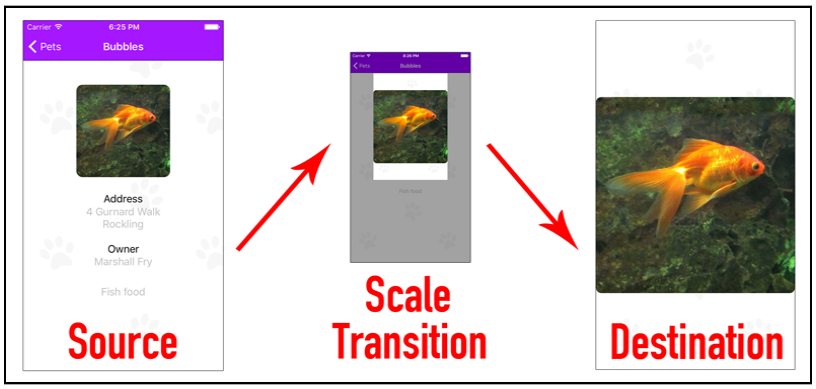
Run the app and tap the photo; you can see the segue and unwind transition animations have changed completely — and you didn't change a single line of code. In fish terms, that was *reely* easy! :]

Want to try another segue modification? Try changing the segue class to `FadeSegue`, which is included in the starter project.

Once you've built up a library of custom segues, you only need to select the desired segue from your library on your storyboard and you're done — no code required.

Creating a custom segue

You'll now create your own custom segue to replace `DropSegue`. As is befitting a fish, you'll create a `Scale` transition animation. :] The image below shows the transition you'll be creating:



The hardest part of creating a custom segue is the terminology. The protocols you'll be working with have rather long names:

- **UIViewControllerTransitioningDelegate**: The custom segue adopts this protocol to vend the animator objects upon presentation and dismissal.
- **UIViewControllerAnimatedTransitioning**: The animator objects adopt this protocol to describe the animations.
- **UIViewControllerContextTransitioning**: This context holds details about the presenting and presented controllers and views; you pass this to the animator objects to provide them the *context* within which to perform the animation.

If you haven't used custom transition animations before, you might be *floundering* a bit at these long method names. :] Once you've used these methods a few times they'll become quite familiar.

Before you start, take a moment to review the steps required to create an animated segue:

1. Subclass `UIStoryboardSegue` and set the segue as the destination controller's transitioning delegate.
2. Create the presenting and dismissing animator classes.
3. Define the animation and its duration to be used in the animators.
4. Instruct the segue which animator classes to use for presentation and dismissal.
5. Finally, use the segue in the storyboard.

The sections below walk you neatly through each step.

Subclass UIStoryboardSegue

You'll first create a new `UIStoryboardSegue` subclass. This segue will adopt the transitioning delegate protocol, allowing it to specify a custom transition animation.

Create a new Cocoa Touch class named **ScaleSegue.swift** that subclasses

UIStoryboardSegue. Add the following `extension` just below the `ScaleSegue` class:

```
extension ScaleSegue: UIViewControllerTransitioningDelegate {

}
```

The `UIViewControllerTransitioningDelegate` protocol lets the segue vend presentation and dismissal animators for use in its transitions. Later, you'll implement a protocol method in this extension that returns the custom animator you're going to build in the next section.

For now, in the `ScaleSegue` class, override `perform()` as follows:

```
override func perform() {
  destinationViewController.transitioningDelegate = self
  super.perform()
}
```

Here you set the destination view controller's transitioning delegate so that `ScaleSegue` will be in charge of vending animator objects. When creating a modal or popover segue as you do here, you must call `perform()` on `super` so that UIKit can take care of the presentation.

In previous iOS versions, you might have put the transition animation in `perform()`, but now you can use this transitioning delegate to decouple the animation from the segue.

Create the animator

Add the following new animator class at the end of **ScaleSegue.swift**:

```
class ScalePresentAnimator : NSObject,
  UIViewControllerAnimatedTransitioning {

}
```

You'll use `ScalePresentAnimator` to present the modal view controller. You'll create a dismissal animator in a bit, but for now your segue will use the default vertical cover transition for the dismissal. Note that Xcode will complain this doesn't yet conform to the `UIViewControllerAnimatedTransitioning` protocol; you're just about to fix that.

> **Note**: It's often easier to keep the animators in the same file as their respective segue as they're usually closely related. If you want to separate the segues from the animators simply move the animators into their own file.

Define the animation

`ScalePresentAnimator` conforms to `UIViewControllerAnimatedTransitioning`. This

protocol requires that you specify both the duration and the animation to be used for the transition.

First, you have to specify the duration of the animation. Add the following method to `ScalePresentAnimator`:

```
func transitionDuration(
  transitionContext: UIViewControllerContextTransitioning?)
  -> NSTimeInterval {
  return 2.0
}
```

Most transitions will have a duration of about 0.3 to 0.5 seconds, but this uses a duration of two seconds so that you can see the effect clearly.

Now for the actual animation: add the following method to `ScalePresentAnimator`:

```
func animateTransition(transitionContext:
  UIViewControllerContextTransitioning) {

  // 1. Get the transition context to- controller and view
  let toViewController = transitionContext
    .viewControllerForKey(
      UITransitionContextToViewControllerKey)!
  let toView = transitionContext
    .viewForKey(UITransitionContextToViewKey)

  // 2. Add the to- view to the transition context
  if let toView = toView {
    transitionContext.containerView()?.addSubview(toView)
  }

  // 3. Set up the initial state for the animation
  toView?.frame = .zero
  toView?.layoutIfNeeded()

  // 4. Perform the animation
  let duration = transitionDuration(transitionContext)
  let finalFrame = transitionContext
    .finalFrameForViewController(toViewController)

  UIView.animateWithDuration(duration, animations: {
    toView?.frame = finalFrame
    toView?.layoutIfNeeded()
  }, completion: {
    finished in
    // 5. Clean up the transition context
    transitionContext.completeTransition(true)
  })
}
```

Taking each numbered comment in turn:

1. You extract the "to" controller and view from the given transition context. Note that the controller is implicitly unwrapped; there will always be a "to" controller, but there may not always be a "to" view. You'll see why later.

2. Add the "to" view to the transition context `containerView` where the animation takes place. Left to its own devices, the framework doesn't add the "to" view to the view hierarchy until the end of the transition. In order to see the new view appear, you need to add it to the hierarchy in your code.
3. The initial state for the "to" view frame is a rectangle of zero size in the top left-hand corner of the screen. When you change the frame of a view, you should always call `layoutIfNeeded()` to update the view's constraints.
4. The animation block is a simple animation from the zero rectangle to the final frame calculated by the transition context.
5. The transition context must always clean up at the end of the animation; calling `completeTransition(_:)` finalizes the view hierarchy and the layout of all views.

Set the animator in the segue

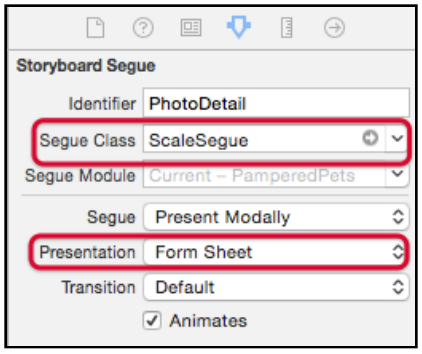
Add the following delegate method to `ScaleSegue`'s `UIViewControllerTransitioningDelegate` extension:

```
func animationControllerForPresentedController(
  presented: UIViewController,
  presentingController presenting: UIViewController,
  sourceController source: UIViewController)
  -> UIViewControllerAnimatedTransitioning? {
  return ScalePresentAnimator()
}
```

This simply tells the segue to use your `ScalePresentAnimator` during presentation.

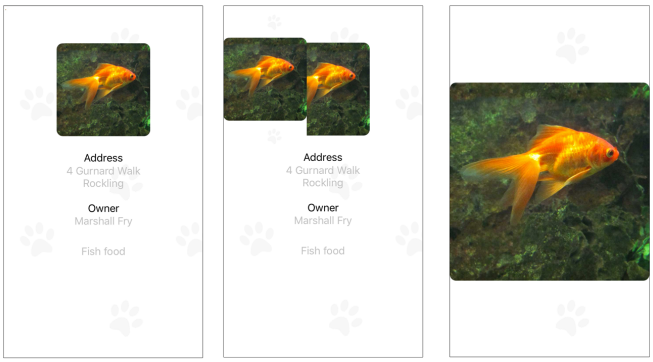
Use the segue in the storyboard

That takes care of all the actual code; all that's left is to set your custom segue in the storyboard. In **Main.storyboard**, locate the **PhotoDetail** segue and change **Segue Class** to `ScaleSegue`. Also, change **Presentation** to **Form Sheet** to improve the appearance of the segue on the iPad:



Run your application and tap the fish; the image will scale from the top left of the screen to take up the full screen on the iPhone, while on the iPad the image scales

up to a form sheet:



Tap the large photo to dismiss it via the standard dismiss animation.

Hey – you've completed your first custom segue! There's a bit of tweaking to do, for sure, but the current state of **ScaleSegue.swift** will serve as a blueprint for all animated segue transitions from here on out. You're going to need a dismissal animator and some more animation code, and your implementation of `animateTransition(_:)` can become quite complicated sometimes, but the basic process will be the same.

Take a moment to browse through example custom segue code in **DropSegue.swift** and **FadeSegue.swift**. Even though the animations are different, the basic structure of both segues is exactly the same as `ScaleSegue`.

Passing data to animators

Most users would expect the small photo to scale directly to a large one when they tap it. But how do you indicate to the animator object which view to scale? You don't have a direct reference to the source image view as everything is decoupled.

Protocols are the perfect tool for this problem. The Animal Detail view controller can adopt a protocol to set which view to scale; the animator object can then use that protocol's scaling view without knowing anything else about the source view controller.

Create the following protocol in **ScaleSegue.swift**:

```
protocol ViewScaleable {
  var scaleView: UIView { get }
}
```

Any view controller can use this segue by adopting `ViewScaleable` and creating a `scaleView` property containing the view to scale.

Add the following extension to the very end of **AnimalDetailViewController.swift**:

```
extension AnimalDetailViewController: ViewScaleable {
  var scaleView: UIView { return imageView }
}
```

`AnimalDetailViewController` now conforms to `ViewScaleable`; this sets the protocol's property to `imageView`, which in this instance is your fish image.

Find the following code in `animateTransition(_:)` of **ScaleSegue.swift**:

```
let toViewController = transitionContext
  .viewControllerForKey(UITransitionContextToViewControllerKey)!
```

Add the following code directly after the above line:

```
let fromViewController = transitionContext
  .viewControllerForKey(
    UITransitionContextFromViewControllerKey)!
let fromView = transitionContext
  .viewForKey(UITransitionContextFromViewKey)
```

This gets references for the "from" view controller and for the "from" view. Again, the view controller is implicitly unwrapped while the "from" view is optional.

Note: Make absolutely sure you use the correct key variables. It's frustratingly easy to use `UITransitionContextToViewControllerKey` instead of `UITransitionContextToViewKey`. Code completion makes it all too easy to pick the wrong variable or to mix up the "from" and "to".

Still in `animateTransition(_:)`, replace:

```
toView?.frame = .zero
```

with the following:

```
var startFrame = CGRect.zero
if let fromViewController = fromViewController
  as? ViewScaleable {
    startFrame = fromViewController.scaleView.frame
} else {
  print("Warning: Controller \(fromViewController) does not " +
    "conform to ViewScaleable")
}
toView?.frame = startFrame
```

Instead of starting the "to" view frame animation at the top left, you start the animation at the "from" view controller's `scaleView` frame property.

Notice again that the animator knows *nothing* about the source view controller, other than that it conforms to the `ViewScaleable` protocol, and therefore has a `scaleView` property. This is a great, decoupled software design!

You'll see a compile warning noting you're not using `fromView`; you'll take care of this in a moment.

Run your app; the segue now scales the original view as expected:

Things are going *swimmingly*! :] There are only a few more tweaks left; the following sections show you how all the views work together.

Working with the view hierarchy

You've been using `transitionContext.viewForKey(_:)` in `animateTransition(_:)` to grab the "to" view. But why didn't you just use the destination controller's `view` property?

The transition context handles presentations differently based on the size class. The modal form sheet for a horizontal, regular-sized display is wrapped in a presentation layer that provides the dimming view and rounds the corners of the form sheet. In contrast, a modal controller on a compact display takes up the full screen.

Therefore, on all iPhones, with the exception of the iPhone 6 Plus in landscape, `viewForKey(UITransitionContextToViewKey)` returns the same view as the destination controller's view because no presentation layer exists. However, the destination controller is wrapped in the presentation layer for iPads and the iPhone 6 Plus in landscape; if you referred to the destination controller's view in this case, you'd scale the destination view — not the presentation layer.

You can try this yourself. Find the following in `animateTranstion(_:)` of **ScaleSegue.swift**:

```
let toView = transitionContext
  .viewForKey(UITransitionContextToViewKey)
```

And modify it as follows:

```
let toView = toViewController.view
```

Run your app on the iPhone 6 in portrait mode, and then run it on an iPad. You won't see any change on the iPhone, but on the iPad the form sheet scales in from the top left and looks very weird — *fishy*, even! :]

Revert the code to its original state:

```
let toView = transitionContext
  .viewForKey(UITransitionContextToViewKey)
```

Similarly, the transition context's "from" view could be different to the source view controller's view. In a compact sized view, the transition context's "from" view will be the same as the source view controller's view, but in a normal-sized view the "from" view would be `nil`.

You can take advantage of this behavior in your transition. The transition on compact-sized screens with full-screen modal views would look better if the "from" view faded out during the scale animation. The transition can remain the same on normal-sized screens since the modal scene is a form sheet and therefore leaves the source view controller in situ in the background.

Find the following in `animateTransition(_:)`, inside the animation block at the end of the method:

```
toView?.frame = finalFrame
toView?.layoutIfNeeded()
```

Add the following code immediately after the code above:

```
fromView?.alpha = 0.0
```

This fades out the "from" view.

Next, find the following in the completion block:

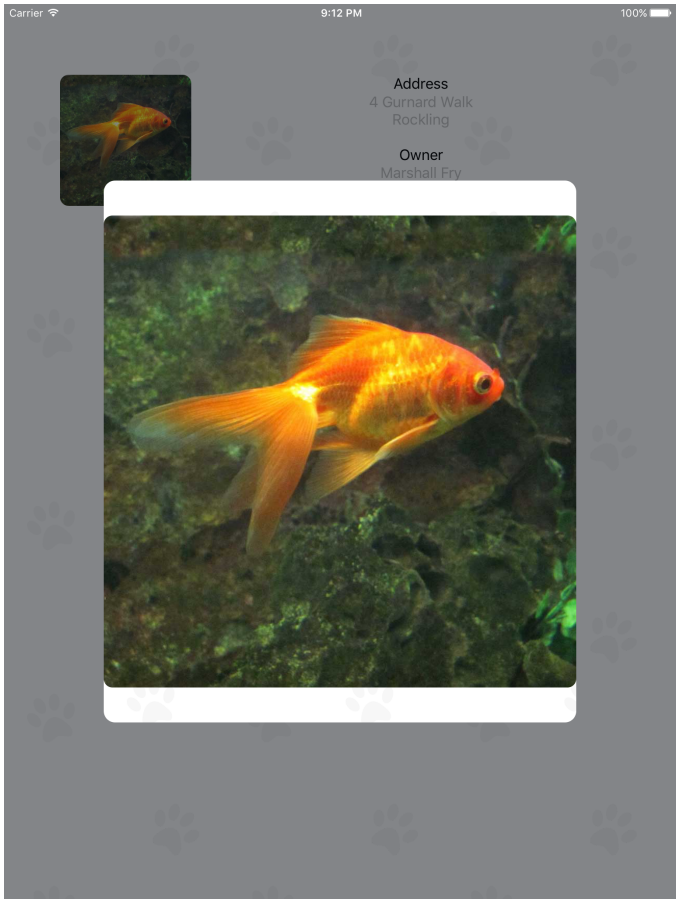
```
transitionContext.completeTransition(true)
```

Add the following code just *before* the code above:

```
fromView?.alpha = 1.0
```

This resets the alpha of the "from" view. If you don't do this, the alpha of the "from" view will remain at 0 and you'll just see a black screen when you when you dismiss the modal scene.

Run the app on both the iPhone 6 and any iPad; the "from" view on the iPad will fade out, but the same view on the iPad won't be affected because `fromView` is `nil`:

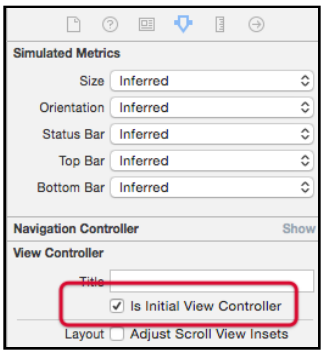


You've created a great-looking custom segue with an animated transition! But don't forget what was promised at the beginning of this chapter — the ability to add multiple pets.

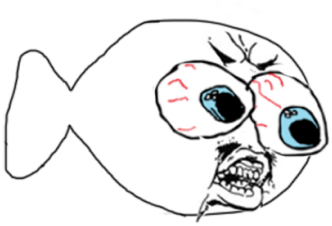
Handling embedded view controllers

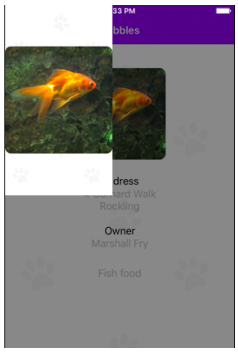
That seems like an easy task; you just need to set the table view of all pets as the initial view controller of the application.

In **Main.storyboard**, select the **Navigation Controller** on the very left of the storyboard. Use the Attributes Inspector to tick **Is Initial View Controller**:



Run your application; you'll see a list of all pets to be minded. Select any pet in the list and tap its photo:





Oh no! The animation scales up from the wrong spot! Look in the debug console and you'll see the debug statement you added earlier indicating the presenting view controller doesn't conform to `ViewScalable`.

That's because the view controller is now embedded within a navigation controller, making that the presenting view controller — not `AnimalDetailViewController`.

Fortunately, this is easy to fix. Simply check whether the presenting view controller is a navigation controller; if so, use the navigation controller's top view controller as the presenting view controller.

Find the following code in the `ScalePresentAnimator` class of **ScaleSegue.swift**, at the top of `animateTransition(_:)`:

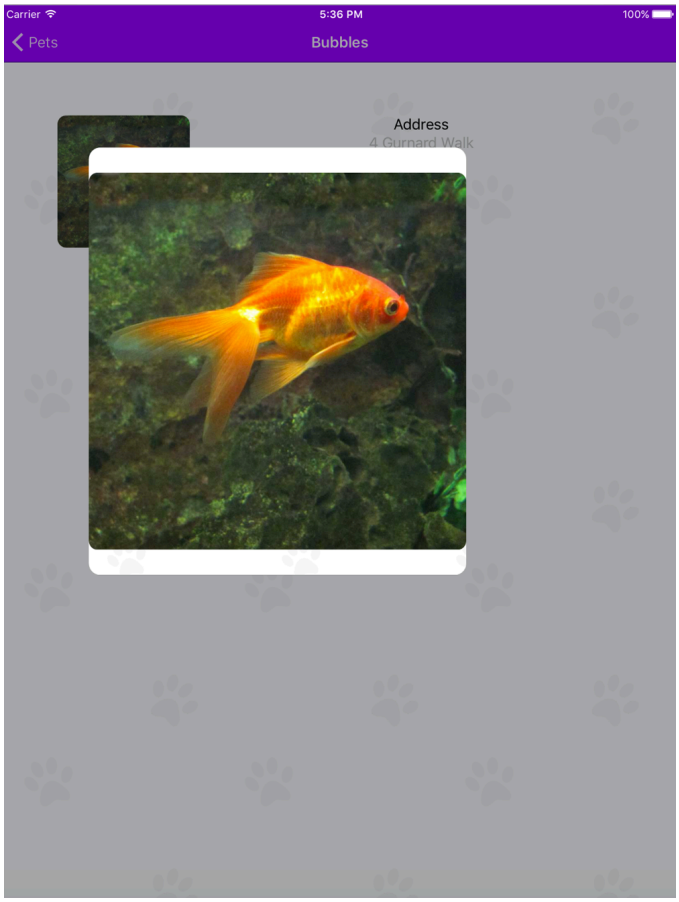
```
let fromViewController = transitionContext
  .viewControllerForKey(
    UITransitionContextFromViewControllerKey)!
```

Replace the previous code with the following:

```
var fromViewController = transitionContext
  .viewControllerForKey(
    UITransitionContextFromViewControllerKey)!
if let fromNC = fromViewController as? UINavigationController {
  if let controller = fromNC.topViewController {
    fromViewController = controller
  }
}
```

Here you replace the transition context's "from" controller *only* if the presenting view controller is a navigation controller.

Run the application again, and the scale transition will work as expected:



When building your reusable segues, expect that they could be used by container controllers. For example, the presenting controller in this case could be a `UITabBarController`, so you could add similar code to handle that case as well.

Now that you're an expert on custom segues, why don't you try these few extra challenges to stretch yourself?

Completing the scale segue dismissal

Your first challenge is to complete the scale segue. You'll create the dismiss animator and set the segue to use it.

The code will look very similar to the presenting animator from this chapter, except that the "from" view controller is now the modal view controller, and the "to" view will be the view from which it was presented. The animator object will need to add the "to" view back to the hierarchy during the transition, so you'll need the code below in order to insert the "to" view at the correct place in the hierarchy:

```
if let fromView = fromView,
  toView = toView {
  transitionContext.containerView()?
    .insertSubview(toView, belowSubview: fromView)
}
```

You can find the full solution to this challenge in the sample code included with this chapter.

Adding a swipe segue

Your next challenge is to create a completely new reusable segue called `SwipeSegue` that uses the default transition to present an up or down swipe to dismiss. The modal scene should slide away in the direction of the swipe.

Here are some tips:

- The `SwipeSegue` class will be almost the same as the `ScaleSegue` class.
- Add a new animator object similar to the Scale dismissal animator that moves the presented frame either off the top of the screen or off the bottom of the screen, depending on which way you swipe.
- Add a new protocol `ViewSwipeable` that stores the swipe direction.
- Add two swipe gestures, one up and one down, to the `AnimalPhotoViewController` image view. Attach an `@IBAction` handler method to the gestures to store the swipe direction in the view controller.
- Add an extension to the view controller for the `ViewSwipeable` protocol that returns the swipe direction.
- Change the existing `PhotoDetail` segue to use your new Swipe segue.

Once again, the solution is in the accompanying sample code.

Where to go from here?

Retaining segues during a modal and popover presentation is a small change that has huge consequences. Previously, view controllers had to know about the transitions to use along with the style to use when presenting. Now the segue can take full responsibility for both elements. You can create any segue with its own transition animation and reuse that segue in any app you wish. You can also easily swap out segues to see which ones look best in your particular app — without changing any code!

You can read more on custom segues in Chapter 3, "Custom View Controller Transitions" of *iOS 7 by Tutorials*. For an excellent reference book on all types of animations in iOS, check out *iOS Animations by Tutorials*.

Congratulations on completing this chapter; with your newly learned transition skills, your apps will be truly *fin*-tastic! :]

Chapter 11: UIKit Dynamics

By Aaron Douglas

iOS applications live in the hands of the people using them. Until somebody taps, swipes and enjoys your work, it sits in suspended animation on a device. Users have come to expect our mobile apps to react to touch and to provide some semblance of "realness". Your app's success depends in part on how much the user enjoys its responsiveness.

iOS 7 introduced the idea of flatness in user interfaces rather than the heavily skeuomorphic concepts we previously experienced. Instead of heavy interfaces, users bond with apps through animations and reactions to touch that mirror real-world physics.

UIKit Dynamics is a 2D physics-inspired animation system designed with a high-level API, allowing you to simulate the physical experiences in your animations and view interactions. Originally introduced in iOS 7, UIKit Dynamics saw very few changes in iOS 8.

iOS 9 is a different matter. With this update we get a bunch of exciting new things like gravity and magnetic fields, non-rectangular collision bounds and additional attachment behaviors.

> **Note**: This chapter will primarily focus on the new features in UIKit Dynamics for iOS 9. Check out chapter 2, "UIKit Dynamics and Motion Effects" of iOS 7 by Tutorials for a full introduction to the original APIs.

Getting started

UIKit Dynamics is definitely a technology you have to learn through playing. Make sure you're using an Xcode Playground to follow along and watch the changes live!

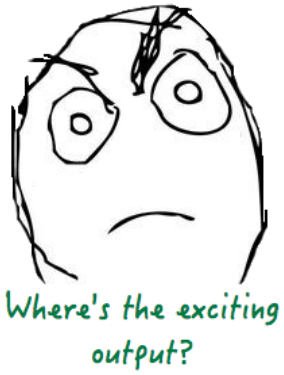
Create the playground

Open Xcode, select **File\New\Playground...** and enter **UIKit Dynamics** for the name and set **Platform** to **iOS**. Click **Next**. Choose a location for your playground and click **Create**.

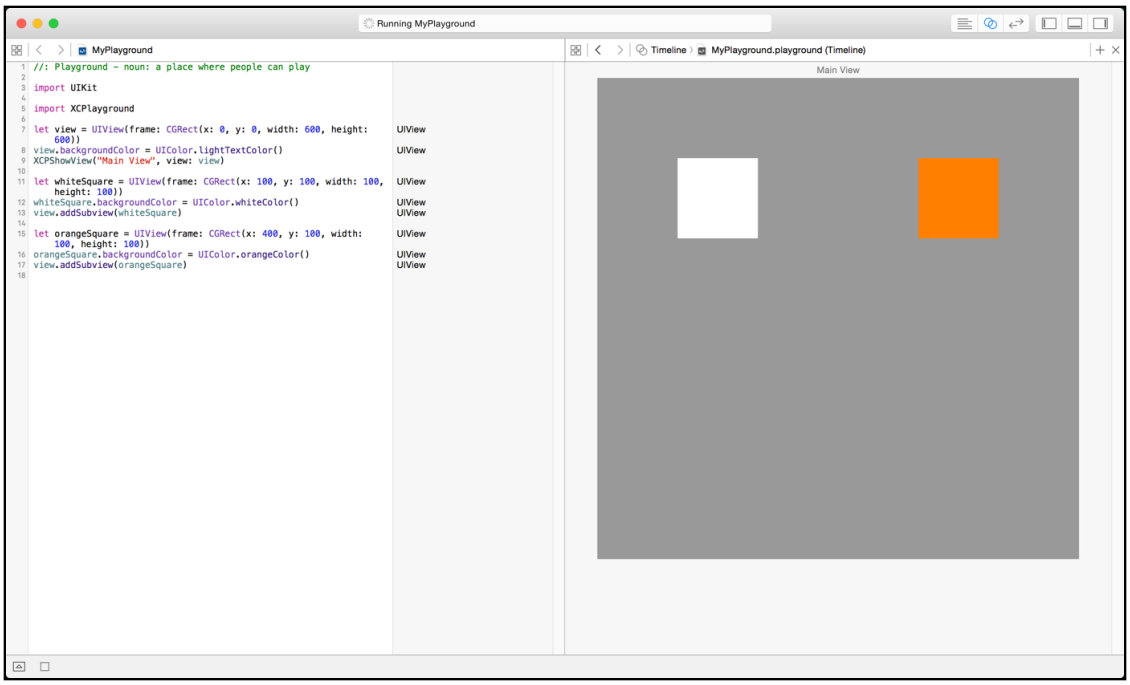
Once the playground opens, replace the contents with:

```
import UIKit
import XCPlayground

let view = UIView(frame: CGRect(x: 0, y: 0,
  width: 600, height: 600))
view.backgroundColor = UIColor.lightTextColor()
XCPShowView("Main View", view: view)

let whiteSquare = UIView(frame: CGRect(x: 100, y: 100,
  width: 100, height: 100))
whiteSquare.backgroundColor = UIColor.whiteColor()
view.addSubview(whiteSquare)

let orangeSquare = UIView(frame: CGRect(x: 400, y: 100,
  width: 100, height: 100))
orangeSquare.backgroundColor = UIColor.orangeColor()
view.addSubview(orangeSquare)
```

You just created a view and added two subviews while giving each a different color, but you don't see anything!



Find it by switching to the assistant editor; simply press **Option + Command + Enter** to bring it up quickly. You should see something like this now:



> **Note**: XCPShowView(_:) is responsible for the magic of rendering your view in the assistant editor. Sometimes Xcode 7 doesn't re-run your Playground after making a change. You can force Xcode to re-run by selecting the menu item **Editor\Execute Playground**.

Add the following line after the second subview:

```
let animator = UIDynamicAnimator(referenceView: view)
```

`UIDynamicAnimator` is where all the physics voodoo happens. The dynamic animator is an intermediary between your dynamic items — UIView subviews in this case — the dynamic behaviors you create, and the iOS physics engine. It provides a context for calculating the animations before rendering.

Dynamic behaviors encapsulate the physics for a particular desired effect like gravity, attraction or bounce. **Dynamic animators** keep track of where all of your items are during the animation process. The `referenceView` you passed in is the canvas where all the animation takes place. All of the views you animate *must* be subviews of the reference view.

Your first behavior

`UIDynamicBehavior` is the base class that describes an effect for one or more dynamic items, like your subviews, and how they take part in the animation. Apple provides a bunch of behaviors, but the easiest one to start with is `UIGravityBehavior`. It's perfect since developers are like cats — we can't help it that we like to see things fall.

Add the following line:

```
animator.addBehavior(UIGravityBehavior(items: [orangeSquare]))
```

This adds a basic gravity behavior to the orange square. See it fall off the screen in the assistant editor?

That took two lines of code. You should be feeling amazed and empowered right now.

Now you'll make the box stop at the bottom of the screen.

```
let boundaryCollision = UICollisionBehavior(items:
  [whiteSquare, orangeSquare])
boundaryCollision.translatesReferenceBoundsIntoBoundary = true
animator.addBehavior(boundaryCollision)
```

Adding a collision behavior and setting `translatesReferenceBoundsIntoBoundary` to `true` makes the border of the reference view turn into a boundary. Now when the orange square falls, it stops and bounces at the bottom of the view.

By default, all dynamic items get a set of behaviors that describe how heavy they are, how much they slow down due to movement, how they respond to collisions and several other physical traits. `UIDynamicItemBehavior` describes these traits.

Change the way the orange square responds to the collision:

```
let bounce = UIDynamicItemBehavior(items: [orangeSquare])
bounce.elasticity = 0.6
bounce.density = 200
bounce.resistance = 2
animator.addBehavior(bounce)
```

A dynamic item's density, along with its size, determines its "mass" when it participates with other behaviors. Elasticity changes how much an item bounces in a collision – the default is 0.0. Resistance represents a frictional force — reducing linear velocity until the item comes to rest.

Take a moment to play around with these values and observe how the animation changes.

Add this line to the end of the playground:

```
animator.setValue(true, forKey: "debugEnabled")
```

This is a new undocumented feature in iOS 9 that turns on a visual debugging mode. It was mentioned in the 2015 WWDC session *What's New in UIKit Dynamics and Visual Effects* (apple.co/1IO1nF3). Although this was described as only being available through the LLDB console, it transpires that you can also enable it via key-value coding using method shown. Debug mode shows cool things like attachments, collision locations and visualizations of field effects.

You'll notice the orange box, when animating, shows a blue border, which visually describes the collision borders for the item.

Leave this debug mode turned on for the remainder of this tutorial.

Behaviors

There are a number of types of behaviors to play around with:

- `UIAttachmentBehavior` – This specifies a connection between two dynamic items or a single item and an anchor point. New to iOS 9 are variants for a sliding attachment, a limit attachment that acts like a piece of rope, a fixed attachment that fuses two items, and a pin attachment that creates the effect of two items connected by a piece of rope hanging over a pin.
- `UICollisionBehavior` – As you've seen already, this behavior declares that an item has a physical interaction with other items. It can also make the reference view turn its border into a collision border with `translatesReferenceBoundsIntoBoundary`.
- `UIDynamicItemBehavior` – This is a collection of physical properties for a dynamic item that are common to multiple behavior types. You've seen friction, density and resistance already. In iOS 9, you can anchor an item to a spot and also

change the charge for an item when it's participating in a magnetic or electric field behavior.

- `UIFieldBehavior` – Totally *new* in iOS 9, this adds a number of physical field behaviors, including electric, magnetic, dragging, vortex, radial and linear gravity, velocity, noise, turbulence and spring fields.
- `UIGravityBehavior` – Adds a gravity field to your views so they react by falling in a particular direction, with constant acceleration.
- `UIPushBehavior` – Applies a force to dynamic items, pushing them around.
- `UISnapBehavior` – Moves a dynamic item to a specific point with a springy bounce-like effect.
- Composite behaviors – You can combine behaviors together for easy packaging and reuse.

MOAR playground

You're probably eager to get lost in the playground with all these new "toys", and now you'll get your chance. Add this code to your playground:

```
let parentBehavior = UIDynamicBehavior()

let viewBehavior = UIDynamicItemBehavior(items: [whiteSquare])
viewBehavior.density = 0.01
viewBehavior.resistance = 10
viewBehavior.friction = 0.0
viewBehavior.allowsRotation = false
parentBehavior.addChildBehavior(viewBehavior)
```

Here you've defined a parent behavior, which doesn't do anything, then you added some physical properties to the white square. Carry on by adding this code:

```
let fieldBehavior = UIFieldBehavior.springField()
fieldBehavior.addItem(whiteSquare)
fieldBehavior.position = CGPoint(x: 150, y: 350)
fieldBehavior.region = UIRegion(size: CGSizeMake(500, 500))
parentBehavior.addChildBehavior(fieldBehavior)
```

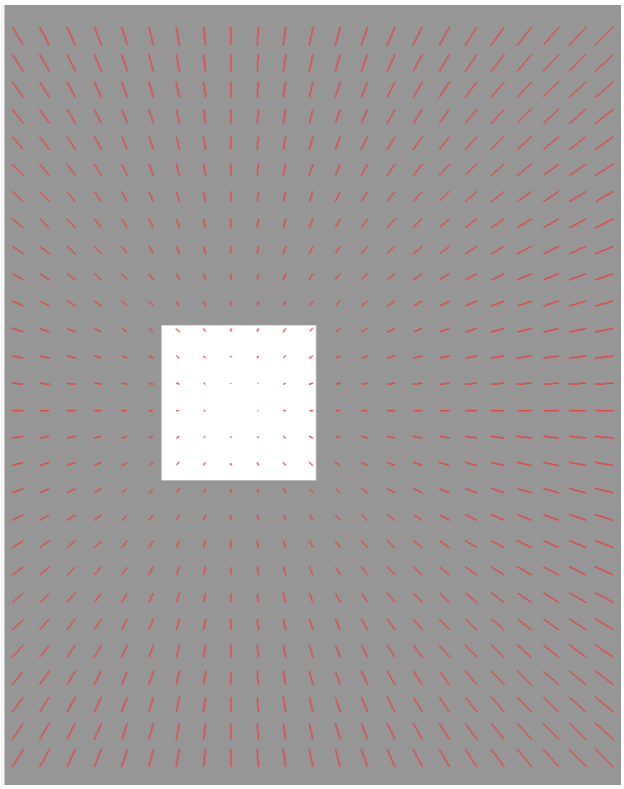
This is one of the new field behaviors. A spring field will drag items caught inside its region to the center. The further out they are, the harder it is to pull them in, so they'll bounce around the center for a while before settling. Note that you've added this spring behavior to the parent behavior.

Now add the composite behavior to the dynamic animator:

```
animator.addBehavior(parentBehavior)
```

Did you see the bouncy snag of the white square? Re-execute the Playground if you didn't. Also, check out the little red lines; this is debug mode showing you the

direction and strength of the spring field:



`UIFieldBehavior` is one of the new behaviors in iOS 9 and arguably the coolest one. Spring fields are great for positioning an element because the region's center draws it in while it bounces into place. Give the white square a little time-delayed push to understand the effect:

```
let delayTime = dispatch_time(DISPATCH_TIME_NOW,
  Int64(2 * Double(NSEC_PER_SEC)))

dispatch_after(delayTime, dispatch_get_main_queue()) {
  let pushBehavior = UIPushBehavior(items: [whiteSquare],
    mode: .Instantaneous)
  pushBehavior.pushDirection = CGVector(dx: 0, dy: -1)
  pushBehavior.magnitude = 0.3
  animator.addBehavior(pushBehavior)
}
```

Now you can really see the power of the spring field! The `UIPushBehavior` gave the white square a nudge upwards and it sprung right back to the center of the field. Push direction is a vector and setting *y* to -1 means up.

The magnitude of the push behavior is set to a small number because the density is set to a small value — the normal push magnitude would kick that box out of the field.

Try removing the magnitude, and you'll notice it does exit the field; however, the collision boundary bounces it back into play.

CHALLENGE: Try attaching the orange box to a point with a `UIAttachmentBehavior` behavior. Use the `init(_:attachedToAnchor:)` method to anchor it to the point. Check out the end of the accompanying playground for the solution. You might like to *play around* with the *playground* (that's kind of the point!) to see what other effects you can create.

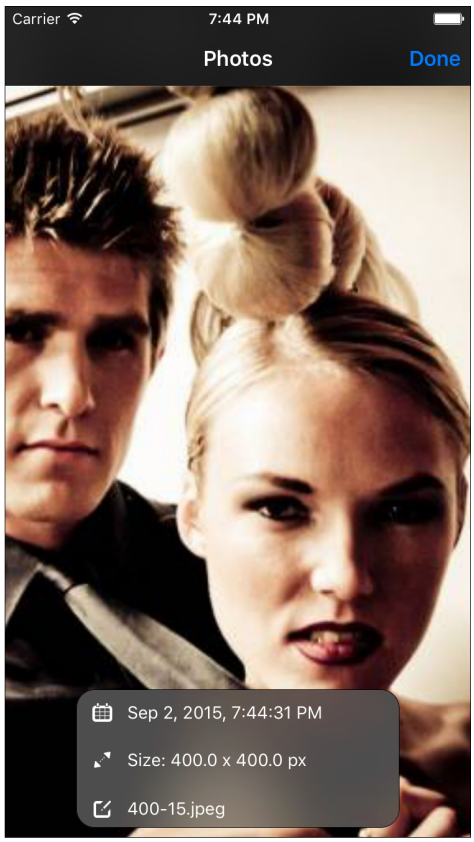
Applying dynamics to a real app

Playing around with UIKit Dynamics in a Playground is fun — but it's not real until it's in an app. UIKit Dynamics is really designed for non-game applications. In reality, your application may only need dynamics in a few key places to give it that extra "pop" you're after. A little goes a long way!

Meet DynamicPhotoDisplay

For this part, you'll work with simple photo viewing application. The user sees a scrolling list of photo thumbnails and taps them to see a full screen version.

You'll find the starter project as well as the final solution in the resources folder for this chapter. Open it in Xcode and build and run it. You should see the following:



You'll notice the full screen view of a photo shows a bit of metadata. The user might encounter a photo where that metadata box obscures a part of the photo.

Your job is to make that box movable, but it should snap into place at the middle-bottom or middle-top of the image and give a cushy feel when it does.

The project's structure is simple. The photos are displayed with a `UICollectionViewController` using a custom `UICollectionViewCell`. When the user taps a cell, standard UIView animations make the corresponding full photo view fall in from the top of the view.

Sticky behavior

You're going to create a new composite behavior to encapsulate the springy-cushiony feel for the metadata box.

Create a new class by clicking on **File\New\File...**, select **Swift File** and name it **StickyEdgesBehavior.swift**. Replace the contents of that file with the following:

```
import UIKit

class StickyEdgesBehavior: UIDynamicBehavior {
  private var edgeInset: CGFloat
  private let itemBehavior: UIDynamicItemBehavior
  private let collisionBehavior: UICollisionBehavior
  private let item: UIDynamicItem
  private let fieldBehaviors = [
    UIFieldBehavior.springField(),
    UIFieldBehavior.springField()
  ]

  init(item: UIDynamicItem, edgeInset: CGFloat) {
    self.item = item
    self.edgeInset = edgeInset

    collisionBehavior = UICollisionBehavior(items: [item])
    collisionBehavior.translatesReferenceBoundsIntoBoundary =
      true

    itemBehavior = UIDynamicItemBehavior(items: [item])
    itemBehavior.density = 0.01
    itemBehavior.resistance = 20
    itemBehavior.friction = 0.0
    itemBehavior.allowsRotation = false

    super.init()

    addChildBehavior(collisionBehavior)
    addChildBehavior(itemBehavior)

    for fieldBehavior in fieldBehaviors {
      fieldBehavior.addItem(item)
      addChildBehavior(fieldBehavior)
    }
  }
```

```
  }
```

The composite behavior starts as a subclass of `UIDynamicBehavior`, which really has no behaviors on its own (make sure you've entered this and **not** `UIDynamicItemBehavior`).

`init` takes the item you're adding the behavior to, as well as an edge inset to make it customizable in the future. Then you create a `UIDynamicItemBehavior` to make the item lighter and more resistant, and also a `UICollisionBehavior` so it can collide with the reference view. Lastly, you add two `UIFieldBehavior` instances, one for the top-middle and one for the bottom-middle.

Add this helper enum just above the class declaration:

```
enum StickyEdge: Int {
  case Top = 0
  case Bottom
}
```

This helps identify the edge in the array of spring fields.

Add the following to the class:

```
func updateFieldsInBounds(bounds: CGRect) {

  //1
  guard bounds != CGRect.zero else { return }
  let h = bounds.height
  let w = bounds.width
  let itemHeight = item.bounds.height

  //2
  func updateRegionForField(field: UIFieldBehavior,
    _ point: CGPoint) {

    let size = CGSize(width: w - 2 * edgeInset,
      height: h - 2 * edgeInset - itemHeight)
    field.position = point
    field.region = UIRegion(size: size)
  }

  //3
  let top = CGPoint(x: w / 2, y: edgeInset + itemHeight / 2)
  let bottom = CGPoint(x: w / 2,
    y: h - edgeInset - itemHeight / 2)

  //4
  updateRegionForField(fieldBehaviors[StickyEdge.Top.rawValue],
    top)
  updateRegionForField(
    fieldBehaviors[StickyEdge.Bottom.rawValue], bottom)
  }
}
```

This function will be called upon initial display of the view or if the view is ever

resized. Its job is to set up the size and position of each of the sticky spring fields. Here's a deeper breakdown:

1. Makes sure the bounds are non-zero or that layout has occurred, and extracts some important values into constants.
2. Defines an inner function to update a particular field, given a location. It centers the field on the location and sizes it so it's inset from the left and right edges and takes up enough vertical space to reach the middle of the screen.
3. Defines the points that will be the center of each field.
4. Updates the top and bottom fields based on the new values.

Next, add the following property to the class:

```
var isEnabled = true {
  didSet {
    if isEnabled {
      for fieldBehavior in fieldBehaviors {
        fieldBehavior.addItem(item)
      }
      collisionBehavior.addItem(item)
      itemBehavior.addItem(item)
    } else {
      for fieldBehavior in fieldBehaviors {
        fieldBehavior.removeItem(item)
      }
      collisionBehavior.removeItem(item)
      itemBehavior.removeItem(item)
    }
  }
}
```

This helper property turns off the behavior in the animator during certain lifecycle events that happen while moving the item.

Finally, add this method:

```
func addLinearVelocity(velocity: CGPoint) {
  itemBehavior.addLinearVelocity(velocity, forItem: item)
}
```

Build your application to make sure it compiles correctly. Disappointingly the app won't look any different yet :[

The method you just added will help snap the metadata box into place with a velocity. Now you need some velocity. To get that, you'll need to add the pan gesture recognizer, which is next.

Open **FullPhotoViewController.swift** and add the following below the existing `@IBOutlet` properties:

```
private var animator: UIDynamicAnimator!
var stickyBehavior: StickyEdgesBehavior!

private var offset = CGPoint.zero
```

Inside of `viewDidLoad()` add the following:

```
let gestureRecognizer = UIPanGestureRecognizer(target: self,
  action: #selector(FullPhotoViewController.pan(_:)))
tagView.addGestureRecognizer(gestureRecognizer)

animator = UIDynamicAnimator(referenceView: containerView)
stickyBehavior = StickyEdgesBehavior(item: tagView,
  edgeInset: 8)
animator.addBehavior(stickyBehavior)
```

This adds the pan gesture recognizer, the dynamic animator to the container view *and* your new sticky behavior to the animator. It also sets the debug flag so you can see what's happening.

Now add the following method to the view controller:

```
override func viewDidLayoutSubviews() {
  super.viewDidLayoutSubviews()

  stickyBehavior.isEnabled = false
  stickyBehavior.updateFieldsInBounds(containerView.bounds)
}
```

Whenever the main view's layout changes, the sticky behavior adjusts its bounds.

Finally, add the following method to the view controller:

```
func pan(pan:UIPanGestureRecognizer) {
  var location = pan.locationInView(containerView)

  switch pan.state {
  case .Began:
    let center = tagView.center
    offset.x = location.x - center.x
    offset.y = location.y - center.y

    stickyBehavior.isEnabled = false

  case .Changed:
    let referenceBounds = containerView.bounds
    let referenceWidth = referenceBounds.width
    let referenceHeight = referenceBounds.height

    let itemBounds = tagView.bounds
    let itemHalfWidth = itemBounds.width / 2.0
    let itemHalfHeight = itemBounds.height / 2.0
```

```
      location.x -= offset.x
      location.y -= offset.y

      location.x = max(itemHalfWidth, location.x)
      location.x = min(referenceWidth - itemHalfWidth, location.x)
      location.y = max(itemHalfHeight, location.y)
      location.y = min(referenceHeight - itemHalfHeight,
        location.y)

      tagView.center = location

    default: ()
    }
  }
```

When the pan gesture begins, the sticky behavior is shut off so the animations won't interfere with the movement. It records the offset of where the user tapped and uses it during the gesture when the location changes. The metadata view's location is updated in the `.Changed` case. The calculations done on the location *x* and *y* limit the movement of metadata view to inside of the container view.

Build and run the application. The metadata box is now draggable, thanks to the pan gesture recognizer, but it just stays where you put it. Seems the sticky behavior is not enabled.

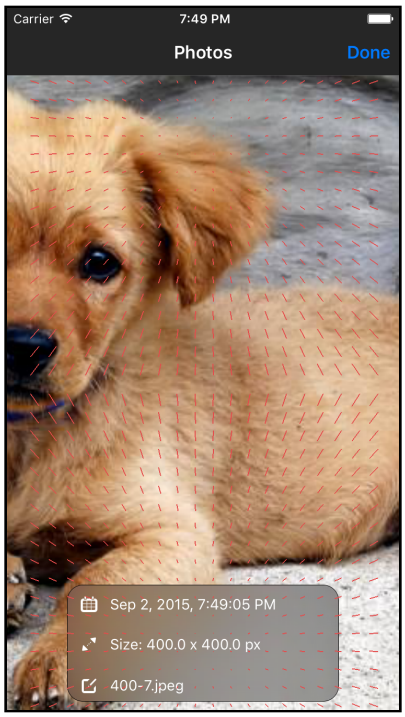
Go back into the `pan` method and add the following case, before the `default:` case:

```
  case .Cancelled, .Ended:
    let velocity = pan.velocityInView(containerView)
    stickyBehavior.isEnabled = true
    stickyBehavior.addLinearVelocity(velocity)
```

Build and run. Now the velocity of your finger as it lifts from the screen will transfer into the sticky behavior, so the view will continue for a moment before being dragged back to the closest field.

For a better understanding of how the behaviors work, turn on debug mode by adding the following to `viewDidLoad()`:

```
  animator.setValue(true, forKey: "debugEnabled")
```



Notice how the lines shorten and nearly disappear in the two zones where the metadata box can live. Seeing is believing!

Full photo with a thud

For your next trick, you're going to update the way the full photo view animates while appearing. Right now, the app uses a UIView animation to animate the bounds change when re-centering the image, you're going to update it to make it feel more dynamic.

You'll effectively do the same action with UIKit Dynamics — animate the change of the center of the view, but this time you'll use gravity and a collision.

Open **PhotosCollectionViewController.swift**, and add the following to the top of the class:

```
var animator: UIDynamicAnimator!
```

Add this line inside of `viewDidLoad()`:

```
animator = UIDynamicAnimator(referenceView: self.view)
```

Now that you've created the animator, swap out the contents of `showFullImageView` with the following:

```
func showFullImageView(index: Int) {
  //1
  let delayTime = dispatch_time(DISPATCH_TIME_NOW,
    Int64(0.75 * Double(NSEC_PER_SEC)))
```

```
    dispatch_after(delayTime, dispatch_get_main_queue()) {
      let doneButton = UIBarButtonItem(barButtonSystemItem: .Done,
        target: self, action:
  #selector(PhotosCollectionViewController.dismissFullPhoto(_:)))
      self.navigationItem.rightBarButtonItem = doneButton
    }

    //2
    fullPhotoViewController.photoPair = photoData[index]
    fullPhotoView.center = CGPoint(x: fullPhotoView.center.x,
      y: fullPhotoView.frame.height / -2)
    fullPhotoView.hidden = false

    //3
    animator.removeAllBehaviors()

    let dynamicItemBehavior = UIDynamicItemBehavior(items:
      [fullPhotoView])
    dynamicItemBehavior.elasticity = 0.2
    dynamicItemBehavior.density = 400
    animator.addBehavior(dynamicItemBehavior)

    let gravityBehavior = UIGravityBehavior(items:
      [fullPhotoView])
    gravityBehavior.magnitude = 5.0
    animator.addBehavior(gravityBehavior)

    let collisionBehavior = UICollisionBehavior(items:
      [fullPhotoView])
    let left = CGPoint(x: 0, y: fullPhotoView.frame.height + 1.5)
    let right = CGPoint(x: fullPhotoView.frame.width,
      y: fullPhotoView.frame.height + 1.5)
    collisionBehavior.addBoundaryWithIdentifier("bottom",
      fromPoint: left, toPoint: right)
    animator.addBehavior(collisionBehavior)
  }
```

Here's a breakdown of that block:

1. Adds the Done button to the nav bar after a short delay. This lets the dynamic animator do most of its animations first before updating the nav bar.

2. Sets the image and repositions the full photo view above the thumbnails view, just off-screen.

3. Removes any existing behaviors from the animator, and then adds the gravity, item and collision behaviors.

Build and run the app. Tap a photo and notice the bounce when the view hits the bottom of the screen. The collision behavior demonstrated here is a bit different from previous examples; instead of using the reference view's boundary, this creates a single line of collision at the bottom. It's positioned just off the screen so the bounce doesn't leave a visible gap.

There are a lot of knobs and levers to change when dealing with behaviors. Play

around with the `UIDynamicItemBehavior` and `UIGravityBehavior` properties to see if you can find a bounce behavior you like!

Where to go from here

You've now played with most of the behaviors available to you in UIKit Dynamics, but there's more properties and options to be explored.

At the time of writing this chapter, Apple hasn't created a guide for UIKit Dynamics, so you'll want to spend some quality time with the documentation on each of the classes to learn more about the finer controls available to you.

Also, check out these videos from the past WWDCs:

- 2013 - #206 - Getting Started with UIKit Dynamics - apple.co/1J1IoNB
- 2013 - #217 - Exploring Scroll Views in iOS 7 - apple.co/1gQGtPM
- 2013 - #221 - Advanced Techniques with UIKit Dynamics - apple.co/1T1N2Qf
- 2014 - #216 - Building Adaptive Apps with UIKit - apple.co/1hoAQbr
- 2015 - #229 - What's New in UIKit Dynamics and Visual Effects - apple.co/1IO1nF3

Challenges

Now it's time for you to take a whack at adding some dynamic goodness to the app. You'll find the solutions in the final version of this app — but give yourself a chance before you go reverse engineering!

Challenge #1

Instead of using UIView animations to dismiss the view after tapping the done button, use UIKit Dynamics. You'll want the view to be pushed off-screen upwards at a slow enough rate for the user to experience it. Replace the contents of `dismissFullPhoto` with the behaviors.

Hints:

- A `UIPushBehavior` behavior will give the view the kick it needs.
- A `UIDynamicItemBehavior` can adjust the photo view's properties so it moves the way you want.
- A `UIAttachmentBehavior` sliding behavior can stop the view once it gets off screen.
- You can use the `UIDynamicAnimator` delegate method `dynamicAnimatorDidPause` to hide the view after it has animated off-screen.

Challenge #2

Add an interaction to the app that allows you to swipe up on the full photo view to dismiss it — it's very similar to the lock screen photo behavior in iOS. You should be able to lift the full photo view up, but it should drop back down if you didn't lift it high enough. A good swipe upwards should fling it off the screen.

Hints:

- You'll need to create a new composite behavior like you did with the `StickyEdgesBehavior` earlier. This should have a dynamic item behavior to give density and elasticity, a collision behavior to stop it from falling off the bottom of the screen, and a gravity behavior to make it drop. You should also allow it to take a linear velocity like the sticky edges behavior did.
- The swipe and drag up behavior will exist in `PhotosCollectionViewController` along with a `UIPanGestureRecognizer`. The setup will look similar to the pan gesture recognizer used to move the metadata view around.
- If the view is moved up less than half way, it should bounce back down. Take the velocity into account as well to determine if there is enough movement to dismiss the view. Try playing around with the camera on the lock screen for an example.
- If you do dismiss the view, you may as well reuse the code you wrote earlier for pushing the view off the screen. Refactor that into a separate method so you can use it in both cases.

Chapter 12: Contacts

By Evan Dekhayser

A long time ago, in an operating system far, far away, developers accessed a user's contacts on their iOS device with a C API and had to deal with the pain of using ancient structs and Core Foundation types in an object-oriented world.

Actually, this wasn't so long ago and far away — this antiquated Address Book framework was still in use as of iOS 8!

Apple deprecated the Address Book framework in iOS 9 — hoorah! In its stead, they introduced two powerful object-oriented frameworks to manage user contacts: **Contacts** and **ContactsUI**. This chapter will show you how to use these frameworks to do the following:

1. Use the ContactsUI framework to display and select contacts.
2. Add contacts to the user's contact store.
3. Search the user's contacts and filter using `NSPredicate`.

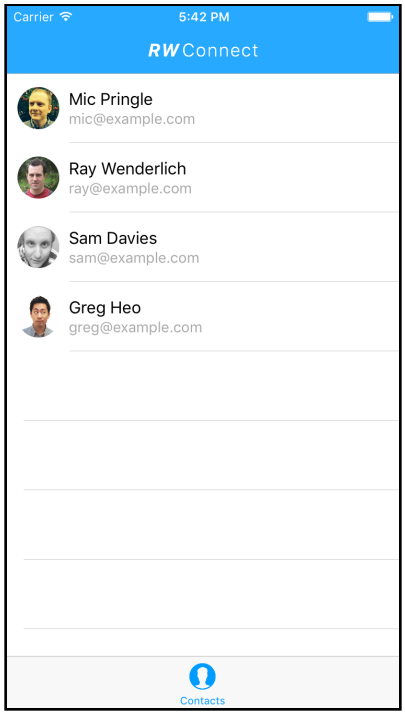
Along the way, you'll learn about best practices for dealing with the user's contacts and how to get the most out of the Contacts framework.

Getting started

In this chapter, you'll create the **RWConnect** app, which is a social network for iOS developers. The app has a friends list to help you keep in touch with all the great developers you know via email.

> **Note**: You should use the simulator instead of a real device to test your app in this chapter; you'll have to reset your device in order to test the app permissions, and you don't want to reset your personal iPhone, do you?

Open the starter project **RWConnect-Starter** and run it on the iPhone 6 Simulator; you'll see a table view with four friends listed, each with a name, picture, and email:



You'll add more features to your app as you progress through the chapter to make your friends list more...friendly! :]

Time to look at the code behind the app. Open **Friend.swift**; it contains a struct `Friend`, which represents each friend in your app. Take note of `defaultContacts()`, which returns the contacts you see in the table.

Now open **FriendsViewController.swift**; you can see that you create the `friendsList` property with the results of `defaultContacts()` to retrieve the sample friends at launch.

The `UITableViewDataSource` methods in the view controller are straightforward; the table has one section with a cell for each friend in `friendsList`. Each cell displays the email address and photo of the respective friend.

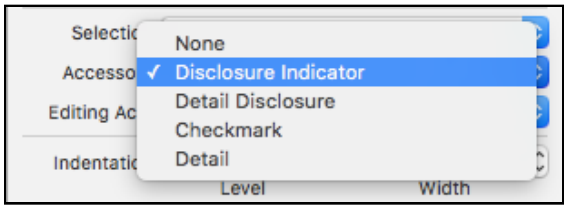
Your first task is to use the ContactsUI framework to display your friends' contact information.

Displaying a contact

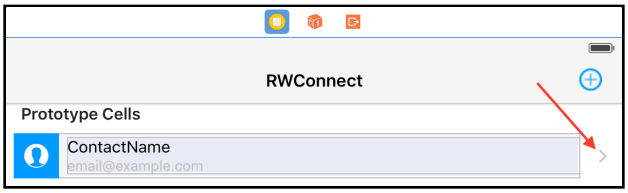
Open **Main.storyboard** and select the table view cell in **FriendsViewController**.

In the **Attributes Inspector**, open the dropdown menu next to **Accessory** and

select **Disclosure Indicator** as shown below:



The table view cell now displays an arrow to indicate there's more information available about this user:



Before you can display the friend to the user, you'll need to convert the `Friend` instance into a `CNContact`.

Convert friends to CNContacts

The Contacts framework represents contacts as instances of `CNContact`, which contain the contact's properties such as `givenName`, `familyName`, `emailAddresses`, and `imageData`.

Open **Friend.swift** and add the following import statement:

```
import Contacts
```

Now, add this extension with the computed property `contactValue`:

```
extension Friend {
  var contactValue: CNContact {
    // 1
    let contact = CNMutableContact()
    // 2
    contact.givenName = firstName
    contact.familyName = lastName
    // 3
    contact.emailAddresses = [
      CNLabeledValue(label: CNLabelWork, value: workEmail)
    ]
    // 4
    if let profilePicture = profilePicture {
      let imageData =
        UIImageJPEGRepresentation(profilePicture, 1)
      contact.imageData = imageData
    }
    // 5
    return contact.copy() as! CNContact
  }
```

```
  }
```

Here's a line-by-line explanation of the code above:

1. You create an instance of `CNMutableContact` with no arguments.
2. Next, you update the contact's properties from the equivalent properties of the `Friend` instance.
3. `emailAddresses` is an array of `CNLabeledValue` objects. This means each email address has a corresponding label. There are many types of labels available for contacts, but in this case you're sticking with `CNLabelWork`.
4. If the `Friend` has a profile picture, set the contact's image data to the profile picture's JPEG representation.
5. Finally, return an immutable copy of the contact.

> **Note**: `CNMutableContact` is the mutable counterpart of `CNContact`. While the properties of `CNContact` are read-only, the properties of `CNMutableContact` can be changed. For this reason, you create a mutable contact, set its properties, then return an immutable copy when done. Note that `CNContact`, like most immutable objects, is thread-safe, while `CNMutableContact` is not.

With this method implemented, you can convert any `Friend` to a `CNContact`. Now that you can convert from one type to another, you can move on to displaying the contact.

Showing the contact's information

Switch to **FriendsViewController.swift** and add the following import statements:

```
import Contacts
import ContactsUI
```

Also, add the following extension to the bottom of the file:

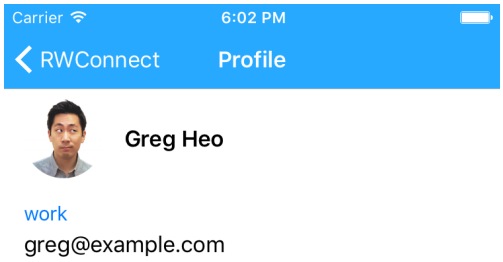
```
//MARK: UITableViewDelegate
extension FriendsViewController {
  override func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
      tableView.deselectRowAtIndexPath(indexPath,
        animated: true)
      // 1
      let friend = friendsList[indexPath.row]
      let contact = friend.contactValue
      // 2
      let contactViewController =
        CNContactViewController(forUnknownContact: contact)
      contactViewController.navigationItem.title = "Profile"
      contactViewController.hidesBottomBarWhenPushed = true
```

```
        // 3
        contactViewController.allowsEditing = false
        contactViewController.allowsActions = false
        // 4
        navigationController?.pushViewController
          (contactViewController, animated: true)
    }
  }
```

Here's what you're doing in the code above:

1. Use the index path of the selected cell to get the selected friend and convert it to an instance of `CNContact`.
2. Instantiate `CNContactViewController`; this is from the ContactsUI framework and displays a contact onscreen. You instantiate the view controller using its `forUnknownContact` initializer because the contact isn't part of the user's contact store. Also, you customize the behaviors of the navigation bar and tab bar to make the app look just right.
3. You set `allowsEditing` and `allowsActions` to `false` so the user can only view the contact's information.
4. Finally, push this view controller onto the navigation stack.

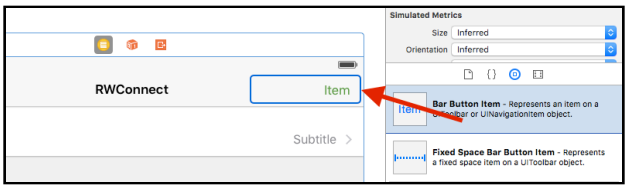
Build and run your app; tap on one of the table view cells and the ContactsUI framework will display the friend's information as shown below:



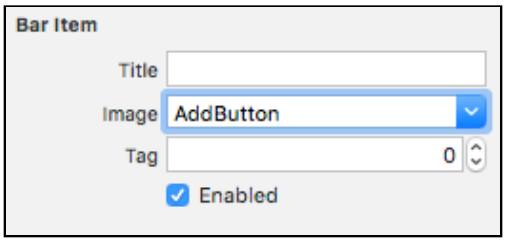
What good is a friends list if you can't add more friends? You can use the ContactsUI class `CNContactPickerViewController` to let your user select contacts to use in the app.

Picking your friends

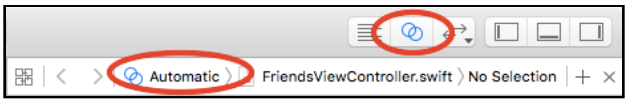
Open **Main.storyboard** and go to **FriendsViewController**. In the **Object library**, drag a **Bar Button Item** to the right side of the navigation bar, as shown below:



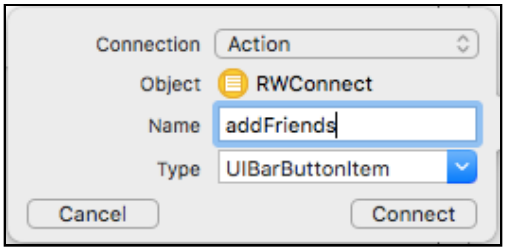
In the **Attributes inspector**, click in the text field next to **Image** and type **AddButton**:



Switch to the **Assistant Editor** and change it to **Automatic**, as shown below:



In the storyboard, select the bar button item you just dragged in, then **Control-Drag** into the `FriendsViewController` implementation in the **Assistant Editor**. Create a new action named **addFriends** that accepts a **UIBarButtonItem** as a parameter:

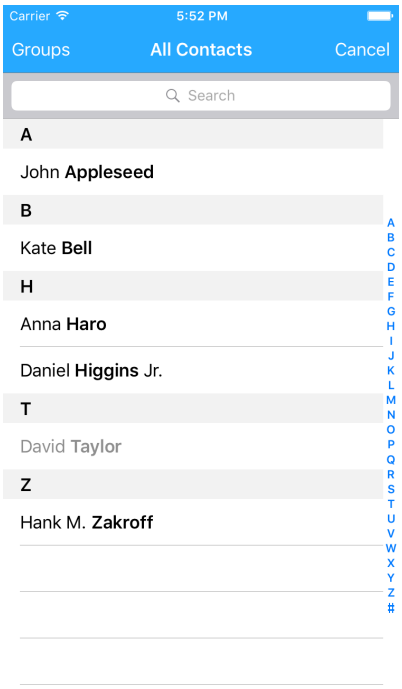


When the user presses the Add button, you'll present the **CNContactPickerViewController** so the user can import their friends from their contacts list.

To do this, add the following code to `addFriends(_:)`:

```
let contactPicker = CNContactPickerViewController()
presentViewController(contactPicker, animated: true,
  completion: nil)
```

Build and run your app; press the **Add** button in the navigation bar and you'll see your **CNContactPickerViewController** appear:



Currently, the user cannot import contacts. When the user selects a contact, the picker view controller just shows more information about the contact.

In order to fix this problem, you need to take advantage of the methods of `CNContactPickerDelegate`.

Conforming to CNContactPickerDelegate

The `CNContactPickerDelegate` protocol has five optional methods, but you'll be interested in `contactPicker(_:didSelectContacts:)`; when you implement this method, `CNContactPickerViewController` knows you want to support multiple selection.

Create the following extension of `FriendsViewController` in **FriendsViewController.swift**:

```
extension FriendsViewController: CNContactPickerDelegate {
  func contactPicker(picker: CNContactPickerViewController,
    didSelectContacts contacts: [CNContact]) {

  }
}
```

`FriendsViewController` now conforms to `CNContactPickerDelegate`. You have an empty implementation of `contactPicker(_:didSelectContacts:)`, which you'll fill with code that does the following things:

1. Creates new `Friend` instances from `CNContacts`

2. Adds the new `Friend` instances to your friends list

To create a `Friend` from a `CNContact`, you'll need a new initializer for `Friend` that takes a `CNContact` instance as a parameter.

Creating a friend from a CNContact

Open **Friend.swift** and add the following initializer in its extension:

```
init(contact: CNContact){
  firstName = contact.givenName
  lastName = contact.familyName
  workEmail = contact.emailAddresses.first!.value as! String
  if let imageData = contact.imageData{
    profilePicture = UIImage(data: imageData)
  } else {
    profilePicture = nil
  }
}
```

When you set `workEmail` you force unwrap the first email address found. Because RWConnect uses email to keep in touch with your friends, all of your contacts must have email addresses. If force unwrapping like this makes you twitchy, don't worry — you'll fix it later! :]

Adding new friends to the friends List

Return to **FriendsViewController.swift** and add the following lines to `contactPicker(_:didSelectContacts:)`:

```
let newFriends = contacts.map { Friend(contact: $0) }
for friend in newFriends {
  if !friendsList.contains(friend){
    friendsList.append(friend)
  }
}
tableView.reloadData()
```

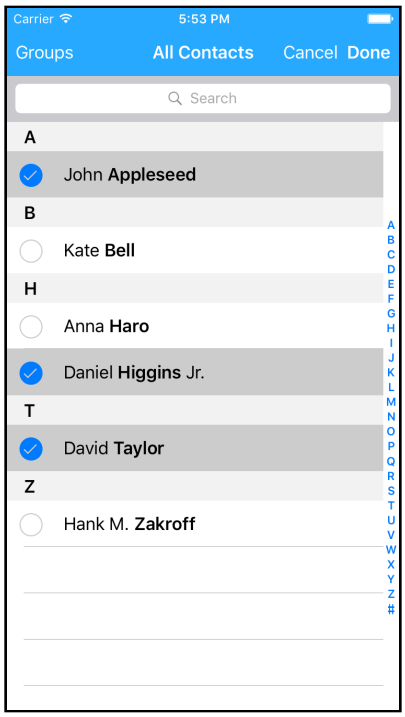
This code transforms each contact the picker returns into a Friend and adds it to the friends list. To show these changes, you simply tell the table view to reload.

Go back to `addFriends(_:)`, and add the following line just before the call to `presentViewController(_:animated:completion:)`:

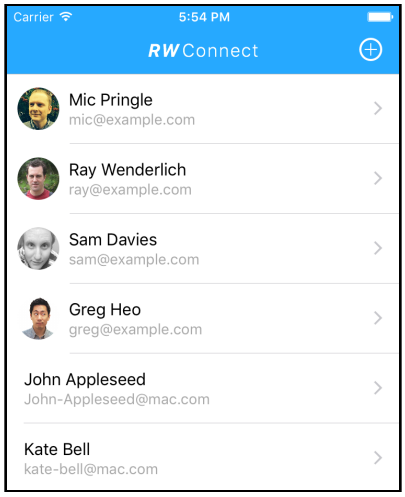
```
contactPicker.delegate = self
```

This oft-forgotten line assigns the friends view controller as at the contact picker's delegate.

Build and run your project; you can now select multiple contacts:



Press the **Done** button, and you'll have a few more friends than you did before!



However, if you select contacts that don't have an associated email address, the app will crash. :[

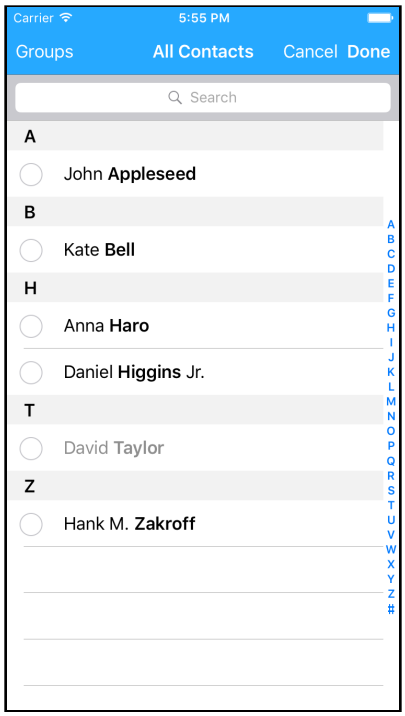
Recall that you force-unwrapped the first email from the contact's email addresses in the `init(contact:)` initializer of `Friend`; a missing email address means the app will crash. Is there a way to make sure that the user can only select contacts with emails? You betcha!

Add the following line before `presentViewController(_:animated:completion:)`:

```
contactPicker.predicateForEnablingContact =
  NSPredicate(format: "emailAddresses.@count > 0")
```

The contact picker's `predicateForEnablingContacts` let you decide which contacts can be selected. In this case, you want to restrict the list of contacts to those with email addresses.

Build and run your app again; press the Add button and you'll see that any contacts without email addresses are grayed out:



Now that you can create friends from your contacts, it's only natural to want to create contacts from your friends! Jump right on to the next section to discover how!

Saving friends to the user's contacts

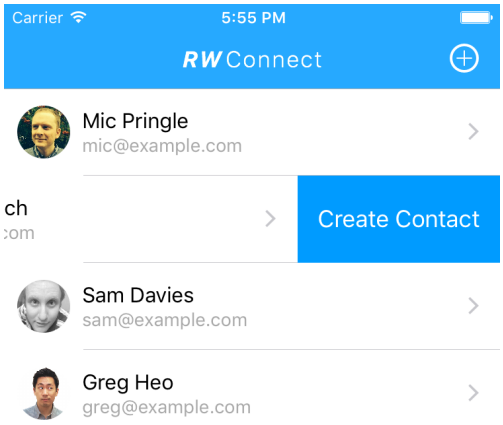
When the user slides left on a table view cell, you'll show a "Create Contact" action to add a friend to the user's contact store.

Add the following code inside the table view delegate extension you added to **FriendsViewController.swift**:

```
override func tableView(tableView: UITableView,
  editActionsForRowAtIndexPath indexPath: NSIndexPath)
  -> [UITableViewRowAction]? {
  let createContact = UITableViewRowAction(style: .Normal,
    title: "Create Contact") { rowAction, indexPath in
    tableView.setEditing(false, animated: true)
    // TODO: Add the contact
  }
  createContact.backgroundColor = BlueColor
  return [createContact]
}
```

The above code creates a single row action for the table view cells named "Create Contact" with a blue background color.

Build and run your app; slide left on a table view cell and you'll see the row action appear like so:



Before you access or modify a user's contacts, it's imperative that you request their permission first; your apps should always respect the user's privacy settings. For this reason, permission functionality is built into the Contacts framework. You can't access the user's contacts without their permission.

Note: When you used `CNContactPickerViewController`, you didn't have to ask for the user's permission. Why? `CNContactPickerViewController` is **out-of-process**, meaning that your app has no access to the contacts shown in the picker, and the user doesn't have to grant permission for this action. If the user selects contacts and presses **Done**, they've given you implicit permission to use their contacts.

Asking for permission

To ask the user for permission, replace the `TODO` comment in the row action you

created earlier with the following code:

```
let contactStore = CNContactStore()
contactStore.requestAccessForEntityType(CNEntityType.Contacts) {
  userGrantedAccess, _ in

}
```

Here you create a `CNContactStore` instance; this represents the user's address book which contains all their contacts. Once you initialize the contact store, you call the instance method `requestAccessForEntityType(:completion:)`.

The completion handler takes a boolean value that indicates whether the user granted permission to access their contacts.

The system presents an alert asking the user for permission the first time you call this method. Each time after that, you call the completion with the user's stored preference. The only way the user can revoke their permission is through the Settings app.

You'll first handle the condition when the user does **not** grant permission. The best practice in this case is to explain the issue to the user and give them the option to open the Settings app.

Add the following method to `FriendsViewController`:

```
func presentPermissionErrorAlert() {
  dispatch_async(dispatch_get_main_queue()) {
    let alert =
      UIAlertController(title: "Could Not Save Contact",
        message: "How am I supposed to add the contact if " +
        "you didn't give me permission?",
        preferredStyle: .Alert)

    let openSettingsAction = UIAlertAction(title: "Settings",
      style: .Default, handler: { alert in
        UIApplication.sharedApplication()
          .openURL(
            NSURL(string: UIApplicationOpenSettingsURLString)!)
    })
    let dismissAction = UIAlertAction(title: "OK",
      style: .Cancel, handler: nil)
    alert.addAction(openSettingsAction)
    alert.addAction(dismissAction)
    self.presentViewController(alert, animated: true,
      completion: nil)
  }
}
```

The above method presents an alert to the user indicating the app can't save the contact. The first UIAlertAction opens the Settings app using the `UIApplicationOpenSettingsURLString` key.

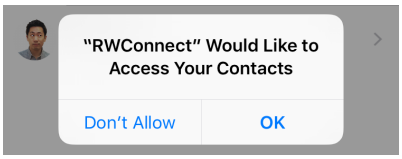
> **Note**: The request access completion block executes on "an arbitrary queue", so you wrap this method in a `dispatch_async` block to ensure the UI code executes on the main thread. The documentation recommends that you work with the contacts store on the handler thread and dispatch to the main thread for UI changes.

Return to the `requestAccessForEntityType(:completion:)` completion handler and add the following code:

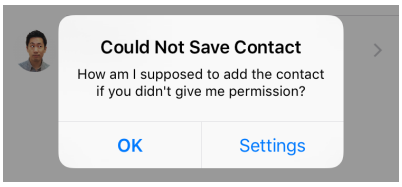
```
guard userGrantedAccess else {
  self.presentPermissionErrorAlert()
  return
}
```

The `guard` statement checks that the user granted permission; if not, you display an alert using `presentPermissionErrorAlert()`.

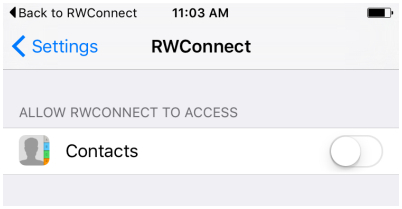
Build and run the app in the simulator; select the **Create Contact** row action for a contact and you'll be prompted for permission to access your contacts:



Select **Don't Allow**; you'll see your custom alert presented like so:



Press **Settings** on your alert and you'll be taken to the Settings app where you can modify your permission if you so desire:



The `guard` statement neatly handles the case where the user does not grant permission. The next logical step is to handle the case where the user grants permission and save the friend to the user's contact store.

Saving friends to contacts

Add the following method to `FriendsViewController`:

```
func saveFriendToContacts(friend: Friend) {
  // 1
  let contact = friend.contactValue.mutableCopy()
    as! CNMutableContact
  // 2
  let saveRequest = CNSaveRequest()
  // 3
  saveRequest.addContact(contact,
    toContainerWithIdentifier: nil)
  do {
    // 4
    let contactStore = CNContactStore()
    try contactStore.executeSaveRequest(saveRequest)
    // Show Success Alert
  } catch {
    // Show Failure Alert
  }
}
```

Taking each numbered comment in turn:

1. First, `addContact` of the Contacts framework expects a mutable contact so you have to convert the Friend parameter into a `CNMutableContact`.
2. Next, create a new `CNSaveRequest`; you use this object to communicate new, updated, or deleted contacts to the `CNContactStore`.
3. Then tell `CNSaveRequest` you want to add the friend to the user's contacts.
4. Finally, try to execute the save request. If the method succeeds, then execution continues and you can assume the save request succeeded; otherwise, you throw an error.

> **Note**: The `catch` block doesn't create a custom alert for every possible thrown error; instead, you use a generic error message. To handle specific error cases, you catch a `CNErrorCode`. These aren't documented, but their declarations exist for your perusal in **CNError.h**.

Now you need to notify the user of the state of their request.

Add the following code to display the alert at `// Show Success Alert`:

```
dispatch_async(dispatch_get_main_queue()) {
  let successAlert = UIAlertController(title: "Contacts Saved",
    message: nil, preferredStyle: .Alert)
  successAlert.addAction(UIAlertAction(title: "OK",
    style: .Cancel, handler: nil))
  self.presentViewController(successAlert, animated: true,
```

```
      completion: nil)
  }
```

Next, add the following code at `// Show Failure Alert:`

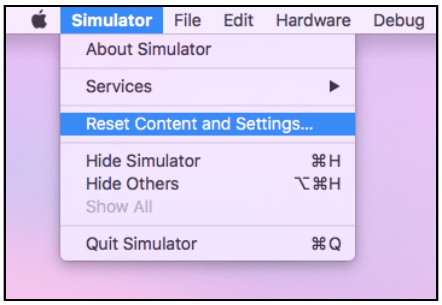
```
  dispatch_async(dispatch_get_main_queue()) {
    let failureAlert = UIAlertController(
      title: "Could Not Save Contact",
      message: "An unknown error occurred.",
      preferredStyle: .Alert)
    failureAlert.addAction(UIAlertAction(title: "OK",
      style: .Cancel, handler: nil))
    self.presentViewController(failureAlert, animated: true,
      completion: nil)
  }
```

Return to `tableView(_:editActionsForRowAtIndexPath:)` and add the following code after the `guard` block:

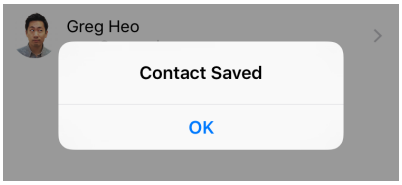
```
  let friend = self.friendsList[indexPath.row]
  self.saveFriendToContacts(friend)
```

Here you pass the friend to add into your save method.

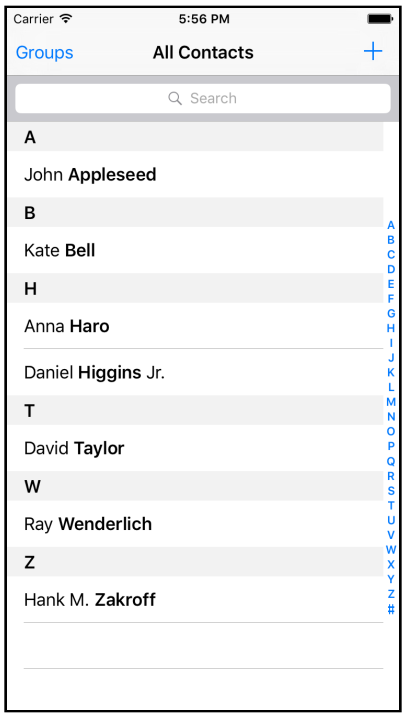
Since you've already granted permission for this app, you'll need to reset the simulator to force the app to prompt for permissions again. This way, you can test all possible permission situations in your app. Select **iOS Simulator/Reset Content and Settings...** as shown below:



Build and run your app; slide left on any contact and tap **OK** when prompted for permission. The app should show the following confirmation:



Now press **Command-Shift-H** in the simulator and open the Contacts app. You'll see your friend appear in your contact list:



Sharp-eyed readers will note you can add the same contact multiple times. You'll add some code to prevent against that.

Checking for existing contacts

Add the following to the top of `saveFriendToContacts(_:)`:

```
//1
let contactFormatter = CNContactFormatter()
//2
let contactName = contactFormatter
  .stringFromContact(friend.contactValue)!
//3
let predicateForMatchingName = CNContact
  .predicateForContactsMatchingName(contactName)
//4
let matchingContacts = try! CNContactStore()
  .unifiedContactsMatchingPredicate(predicateForMatchingName,
    keysToFetch: [])
//4
guard matchingContacts.isEmpty else {
  dispatch_async(dispatch_get_main_queue()) {
    let alert = UIAlertController(
      title: "Contact Already Exists", message: nil,
      preferredStyle: .Alert)
    alert.addAction(UIAlertAction(title: "OK", style: .Cancel,
      handler: nil))
    self.presentViewController(alert, animated: true,
      completion: nil)
  }
  return
}
```

It looks like a lot of detailed code, but it breaks down quite simply:

1. The `CNContactFormatter` class generates locale-aware display names from stored contacts, much as `NSDateFormatter` does with dates.
2. Next, you use the formatter to create the name string based on the contact's given and family name, as well as any titles and suffixes.
3. You then create a predicate for searching the contacts store based on the name string generated in the previous step.
4. `CNContactStore` lets you query the user's contacts for those matching the predicate. In this case, you used `CNContact`'s `predicateForContactsMatchingName(_:)` method to create an `NSPredicate` that finds contacts having a name similar to the provided string.
5. You only save the contact if there aren't any matches; the `guard` statement stops the process in the event of name matches.

> **Note**: `unifiedContactsMatchingPredicate(_:keysToFetch:)` has a `keysToFetch` parameter that you ultimately ignore by passing in an empty array. However, if you were to try to access or modify the fetched contacts, you'd see an error thrown as the keys weren't fetched. For example, if you wanted to access the fetched contacts' first names you'd have to add `CNContactGivenNameKey` to `keysToFetch`.

Build and run your app; try to add a contact that already exists and the app prevents you from doing so.

You're done! You've dramatically improved RWConnect — and learned a ton about the new Contacts framework in the process!

Where to go from here?

At this point you've learned just about everything you need to use the Contacts and ContactsUI frameworks in your own apps. However, there is more to learn about the two frameworks if you want to dig even deeper.

To learn more, be sure to visit the Contact framework guide at apple.co/1LuCodW.

You can also check out the WWDC 2015 Session 223: Introducing the Contacts Framework for iOS and OS X apple.co/1MQVNZV.

Chapter 13: Testing

By Pietro Rea

"How did this feature break *again*?" If you've been writing code for any length of time, you've probably asked yourself that question at least once. As any experienced programmer will tell you, writing code "right" the first time is difficult. And even *if* you get it right the first time, as you add code to your project, you can inadvertently introduce bugs to parts of your app that previously worked flawlessly.

Wouldn't it be great if you could add little "scaffolds" as you code? In theory, these would prevent your code from failing after you've moved on to something else. At the very least, they would give you easy access to important bits of logic and tell you if something broke.

The good news is *you can do this*. You can write tests!

Testing is a proven software engineering practice that helps make software robust and maintainable, and it should be a part of every developer's toolkit. Today it's easy to add unit tests to an iOS project, but it wasn't always that way.

Over the past few years, Apple has made it easier to test iOS apps. Here's a short recap of what's happened in the world of iOS testing since iOS 7:

- With Xcode 5, Apple introduced the first version of the **XCTest** framework, a more modern implementation of the previous SenTestingKit framework. It laid the groundwork for future features and better Xcode integration, including the test navigator.
- When Xcode 6 came out, Apple beefed up XCTest by adding asynchronous and performance testing.
- This year, Xcode 7 introduced code coverage reports and UI testing.

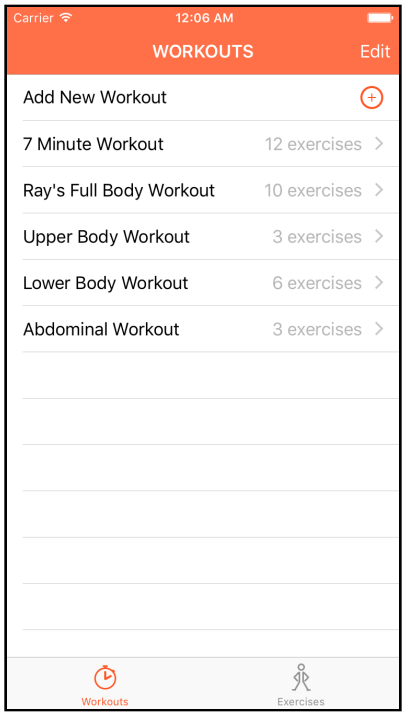
Before Xcode 7, you couldn't use XCTest to test your user interface or write functional tests, you had to turn to third-party libraries such as KIF, Frank or Calabash to fill this gap. Unfortunately, none of these libraries are integrated with Xcode, so you can sometimes find yourself fighting an uphill battle.

This chapter focuses on the latest additions to XCTest. If you're interested in UI testing but don't know much about XCTest, you should first read Chapter 11, "Unit Testing in Xcode 5" in *iOS 7 by Tutorials* and Chapter 29, "What's New with Testing?" in *iOS 8 by Tutorials*.

Swift 2.0 also has an important feature related to testing that you'll read about in this chapter: the addition of `@testable` imports. This solves issues related to tests and access control that many developers faced with earlier versions of Swift. But enough preamble...are you ready to write some tests?

Getting started

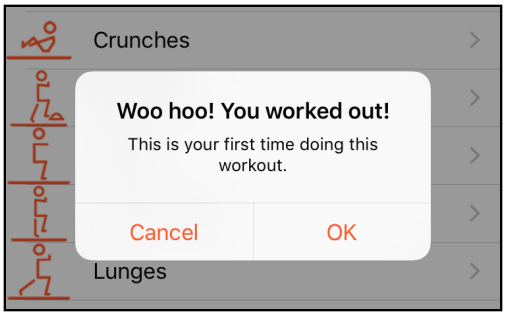
This chapter's sample project is a fitness app for iOS aptly named **Workouts**. Go to this chapter's files and find the **Workouts-Starter** project. Open **Workouts.xcodeproj**, and then build and run the starter project.



The app has two tabs: **Exercises** and **Workouts**. In the exercises tab, you can browse the list of built-in exercises, which includes all-time favorites like push-ups and crunches, or you can make your own. In the Workouts tab, you can also browse built-in workouts or create your own workout. For the purpose of this app, know that a "workout" is simply a sequential list of exercises.

Take a minute to browse through the app. To try it out, tap the **Workouts** tab and select a workout, for example, **Ray's Full Body Workout**. Next, in the workout detail screen scroll to the bottom and tap **Select & Workout**.

Though it's always a good idea to get your heart rate up, you can take a pass on *actually* performing the exercises! You don't need to huff and puff to learn how to test. But you can still revel in some positive reinforcement from the congratulatory alert.



Try creating your own exercises and workouts by tapping **Add New Workout** or **Add New Exercises** in their respective sections of the app.

Now that you've learned your way around the app, head back to Xcode and take a look at the project's files in the project navigator. The starter project is organized into a number of groups. Here's a quick summary of the most important ones:

- **Model:** The project relies on two model objects, `Exercise` and `Workout`. Each holds the data it needs to display on the screen, such as a name, image file name or its duration. Sets of exercises and workouts are stored within an instance of `DataModel`.
- **View Controllers**: `WorkoutViewController` shows you a list of all workouts, both built-in and user-created. When you tap on a specific workout, `WorkoutDetailViewController` displays the workout's information and allows you to perform the workout. `AddWorkoutViewController` lets you add a new workout to the list of workouts.

  Similarly, `ExerciseViewController` displays a list of all exercises in the app. From here, you can add a new exercise or tap into an existing exercise, taking you to `ExerciseDetailViewController`.
- **WorkoutsTests:** The sample app already includes some unit tests in its testing target. **ExerciseTests.swift** and **WorkoutTests.swift** contain unit tests for their corresponding model objects.

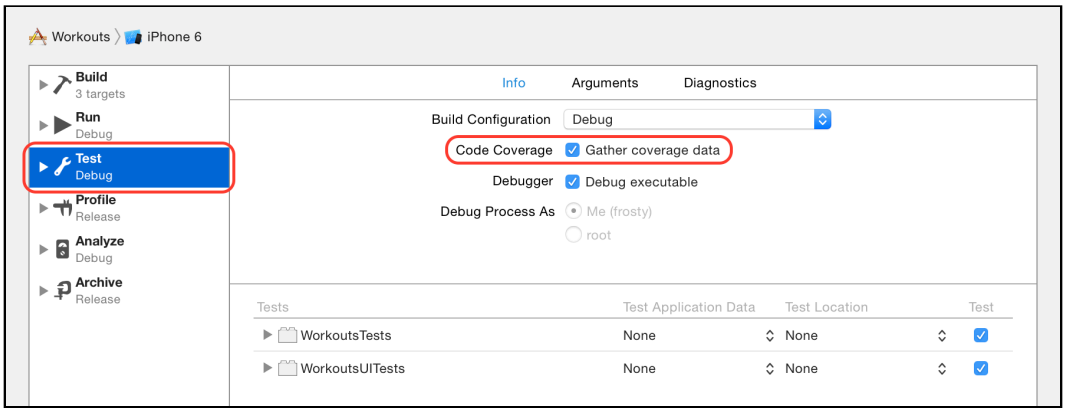
Even though the view controller source files are much longer than the model object source files, notice how they don't have any tests. Talk about a problem! You'll learn how to measure and test this code later in the chapter.

Code coverage

As now know, the starter project already comes with some tests. However, how do you know if you have *enough* tests and whether you're testing the right parts of your project?

That's where code coverage comes in. With Xcode 7 came the ability to get coverage reports that show how much code in any given file is "exercised" by your tests. No pun intended :]

By default, code coverage isn't turned on. Change that now in the starter project by selecting **Product\Scheme\Edit Scheme...**, and then select the **Test** action and click on the **Code Coverage — Gather coverage data** checkbox.

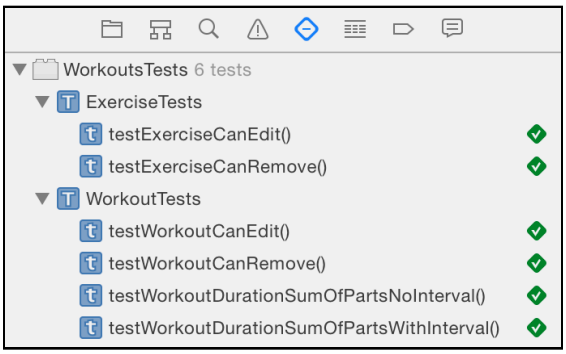


That's all you need to do to turn on code coverage reports. Easy, huh? To check your current levels of code coverage you need to run your test target.

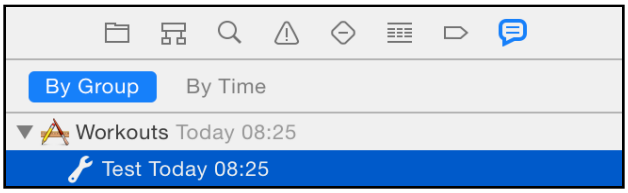
You'll run unit tests constantly in this chapter, so how about a quick refresher? Remember there are three ways to run them in your testing target:

1. Long click Xcode's **Run** button and click on **Test** from the dropdown menu.
2. Select **Product\Test** from Xcode's menu.
3. Use the shortcut **Command+U**.

Use whichever method you like best to run your unit tests. Each tells Xcode to build and launch your app, and then run your tests. In the **test navigator**, you should see a total of six tests:



Now turn your focus towards the code coverage report. Switch to the **report navigator** and click on the latest **test** action.



You're currently seeing the **tests** view of the test report. This shows you a list of your unit tests along with their pass / failure status. From here, click on the **Coverage** tab to switch to the code coverage report:

This report shows you the code coverage for your entire app as well as the code coverage on a per-file basis. For example, the code coverage for the entire app is 37 percent – yikes! – whereas code coverage for **DataModel.swift** is 90 percent.

> **Note**: You'll notice the report doesn't show you the specific coverage percentage right away; you have to hover over the progress indicator until it shows up.

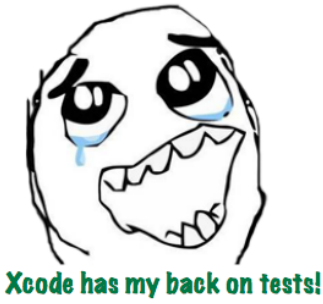
You also have access to code coverage numbers for individual classes and methods. To see them, click on the disclosure indicator to the left of the file name. Xcode can even tell you the coverage of each line in a file. To see this in action, hover over **Workout.swift** and click on the small right-facing arrow that appears to the right of its name:

Doing this takes you to the file you clicked on in Xcode's main editor. Notice that there's a gutter on the right edge of the editor with small numbers on it:

```
var workoutCount: Int {
  return workoutCounter                                    0
}

var canEdit: Bool {                                        7
  return userCreated
}
var canRemove: Bool {                                      2
  return userCreated
}

var duration: NSTimeInterval {                             6
  var totalDuration: NSTimeInterval = 0                    8
  for exercise in exercises {
    totalDuration += exercise.duration
  }                                                        6

  let numIntervals = max(0, exercises.count - 1)
  totalDuration += Double(numIntervals) * restInterval

  return totalDuration
}
```

If granularity is what you want, prepare to be delighted. Xcode's code coverage reports goes beyond the method level. The numbers on the right gutter represents the number of times those lines of code are executed by tests. For example, the getter for `workoutCount` has a 0 next to it because it isn't tested at all.

As you can see, it tells you which lines inside a method are covered and which are not. Now you can identify those edge cases you haven't tested yet without losing your sanity! For instance, if you only test the `if` block in an `if-else` statement, Xcode will pick this up and let you know.



Note: A single code coverage report is simply a snapshot. If you want to know whether your coverage is improving or getting worse, you'll need to see how these numbers change over time. One way to do this is with continuous integration, by using the Xcode server. This chapter won't cover this, but you can learn more about it by catching up on session 410 from WWDC 2015: *Continuous Integration and Code Coverage in Xcode* (apple.co/1J1n1Kd.

@testable imports and access control

As far as test coverage goes, 37 percent is hardly something to brag out. Make it brag worthy by adding more tests. Both **Exercise.swift** and **Workout.swift** have corresponding test files but **DataModel.swift** does not – sounds like a good place to start.

In the project navigator, click the **WorkoutsTests** group. Choose **File\New\File...** and select the **iOS\Source\Unit Test Case Class** template.

Name the class **DataModelTests**, and ensure it's a subclass of **XCTestCase** and that **Swift** is selected. Click **Next** and then **Create**.

As first order of business, add the following `import` at the top of **DataModelTests.swift**:

```
import Workouts
```

Your app and test bundle are separate, so you have to import the app's module before writing any tests against `DataModel`.

Next, delete the entire implementation for the `DataModelTests` class and replace it with the following:

```
class DataModelTests: XCTestCase {
  var dataModel: DataModel!

  override func setUp() {
    super.setUp()

    dataModel = DataModel()
  }

  func testSampleDataAdded() {
    XCTAssert(dataModel.allWorkouts.count > 0)
    XCTAssert(dataModel.allExercises.count > 0)
  }

  func testAllWorkoutsEqualsWorkoutsArray() {
    XCTAssertEqual(dataModel.workouts,
      dataModel.allWorkouts)
  }

  func testAllExercisesEqualsExercisesArray() {
    XCTAssertEqual(dataModel.exercises,
      dataModel.allExercises)
  }

  func testContainsUserCreatedWorkout() {
    XCTAssertFalse(dataModel.containsUserCreatedWorkout)

    let workout1 = Workout()
    dataModel.addWorkout(workout1)
```

```
    XCTAssertFalse(dataModel.containsUserCreatedWorkout)

    let workout2 = Workout()
    workout2.userCreated = true
    dataModel.addWorkout(workout2)

    XCTAssert(dataModel.containsUserCreatedWorkout)

    dataModel
      .removeWorkoutAtIndex(dataModel.allWorkouts.count - 1)
    XCTAssertFalse(dataModel.containsUserCreatedWorkout)
  }

  func testContainsUserCreatedExercise() {
    XCTAssertFalse(dataModel.containsUserCreatedExercise)

    let exercise1 = Exercise()
    dataModel.addExercise(exercise1)

    XCTAssertFalse(dataModel.containsUserCreatedExercise)

    let exercise2 = Exercise()
    exercise2.userCreated = true
    dataModel.addExercise(exercise2)

    XCTAssert(dataModel.containsUserCreatedExercise)

    dataModel
      .removeExerciseAtIndex(dataModel.allExercises.count - 1)
    XCTAssertFalse(dataModel.containsUserCreatedExercise)
  }
}
```

Uh oh... you don't even need to compile to see that errors are popping up everywhere. What's going on?

After adding the new unit tests, Xcode complains every time you reference `DataModel`. The problem is related to Swift access control, which you'll learn about next.

A quick refresher on Swift access control

The concept of access control exists in virtually every programming language, although it bears many names. Whatever it's called, access control allows you to restrict access to sections of code from within other sections of code. In Swift, the access control model is based on the concept of *modules* and *source files*.

A *module* is a single unit of code distribution. This could be an application or a framework. In this example, all the source code in the Workouts app is one module, and all the code in your testing bundle is a separate module. A *source file* is a single Swift source code file within a module (for example **Workout.swift**).

Swift provides three different levels of access:

1. **Public access** enables access to entities in any source file from within their own module, as well as any source file in any other module that imports that module.
2. **Internal access** enables access for entities in any source file from within their own module. Files from outside modules never get access, even if they import the module in question.
3. **Private access** restricts access to entities from anywhere other than the source file where they're defined. This is the most restrictive of all access control levels.

> **Note**: This was a broad overview of Swift's access control model. There's more to it, and if you're interested in learning more, read Apple's documentation on the subject at apple.co/1DH0v9y.

The default access control is **internal**. Now do you see why your unit tests were riddled with errors? All the entities in **DataModel.swift** are internal to the **Workouts** module. You cannot reference them from the testing module, even if you `import` the Workouts module!

@testable imports

Before Xcode 7, you could get around this problem one of two ways:

1. Add every source file you want to test to your testing target. If you were wondering, this is how `WorkoutTests` and `ExercisesTests` currently compile without errors.
2. Make every entity you want to test `public`. Doing this makes your entities visible from your testing module as long as you import your app's main module.

Both options have downsides. With the first option, you know that adding files to both targets is a manual step that's easily forgotten. It also doesn't make much sense conceptually. The testing module should only contain tests!

The second option, marking everything `public`, is even worse. If you have a large project that consists of several modules, there are probably parts of your code that shouldn't be exposed externally. Marking everything `public` for the sake of testability is asking for trouble.

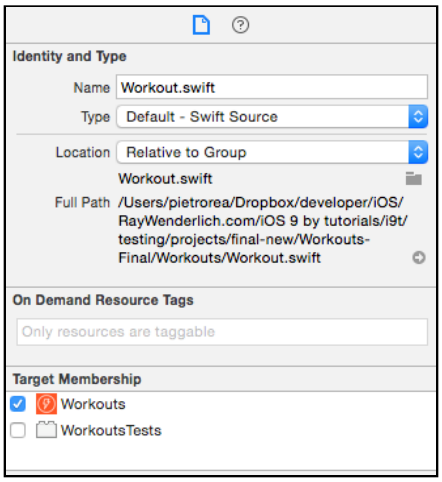
Lucky for you, Swift 2.0 introduces a third option: `@testable` imports.

When you import a module, you can mark it as `@testable`, which changes the way that the default `internal` access control works. Normally you don't have visibility of internal entities from outside modules. With `@testable` you do!

Here's how to implement it in Workouts. First, you'll need to remove `Workout` and

`Exercise` from the tests target.

In Xcode, select **Workout.swift** in the project navigator. In the **File Inspector**, uncheck **WorkoutsTests** in the **Target Membership** section.



Do the same for **Exercise.swift**: click the file in the project navigator, and then uncheck **WorkoutsTests** in the **Target Membership** section of the **File Inspector**.

Now to use `@testable`. Open **DataModelTests.swift**, replace the `import Workouts` line at the top of the file with the following:

```
@testable import Workouts
```

Do the same with the `import` statement at the top of **WorkoutTests.swift** and **ExerciseTests.swift**.

Magic! All your compiler errors disappear. Run your tests again to check that they all pass.

> **Note**: `@testable` has no effect on the *private* access control. As they say in Vegas, what you declare `private` stays `private` :]

Once the tests have finished, head back to the code coverage report in the **report navigator**, select the most recent **test** run, and then click **Coverage** in the main panel. Check out the coverage percentage for **DataModel.swift** — it's now 100 percent! Nice work.

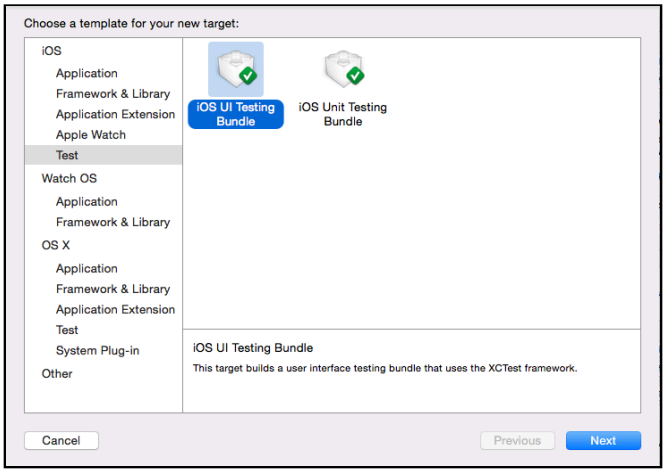
UI testing

So far, you've explored code coverage reports and `@testable` imports. These are great new features — they give you more information and definitely make it easier to test your apps.

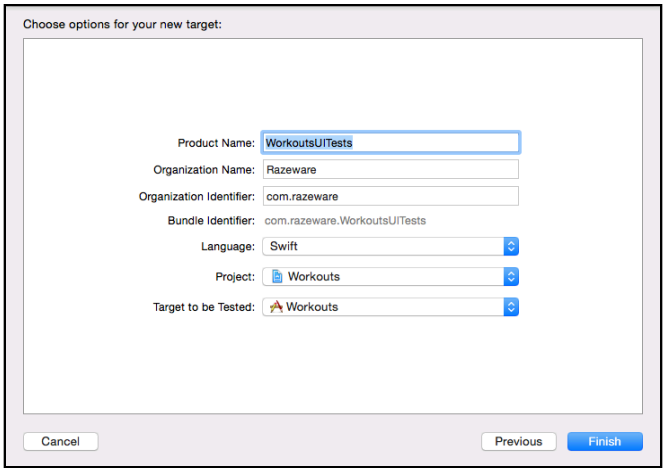
The third and final addition to Xcode's testing capabilities lets you test your app in ways you didn't think possible: through *UI testing*.

Before you can write your first UI test, you have to make sure your project has a UI testing target. The sample project doesn't, so you'll add one right now.

In Xcode's menu, click on **File\New\Target...**, choose the **iOS\Test\iOS UI Testing Bundle** template and click **Next**.



All the default values on the next screen should be correct for this simple sample project. However, if you're working on a project that consists of multiple modules, make sure that the **Target to be Tested** is set to your app's module. Finally, click on **Finish** to create the target.



Voila! Doing this creates a new **WorkoutsUITests** group in Xcode along with a

sample UI testing file called **WorkoutsUITests.swift**.

> **Note**: You'll only have to add a UI testing target if you upgrade an existing project from Xcode 6 or earlier. New projects created with Xcode 7 already come with a UI testing target by default.

Run your first UI test

You're going to validate one important flow in the application: drilling down from the list of workouts into a particular workout's detail page. For this test, the subject will be *Ray's Full Body Workout*.

As a first step, open **WorkoutViewController.swift** and replace `viewDidLoad()` with the following:

```
override func viewDidLoad() {
  super.viewDidLoad()
  tableView.accessibilityIdentifier = "Workouts Table"
}
```

Similarly, open **WorkoutDetailViewController.swift** add the following line to `viewDidLoad()`:

```
tableView.accessibilityIdentifier = "Workout Detail Table"
```

One of the ways the UI testing framework references individual UI components in your app is via accessibility information. Setting this table view's accessibility identifier to `"Workouts Table"` will let you reference this table view later on by its identifier.

> **Note**: If you want to add or improve your existing accessibility information, you can either do it via Interface Builder or the accessibility API. Apple's recommended way is via Interface Builder. However, `UITableView` doesn't have an accessibility panel in Interface Builder, so that's why you did it in code.

Head over to **WorkoutsUITests.swift**. Remove the `tearDown()` and `testExample()` methods and add the following method:

```
func testRaysFullBodyWorkout() {
  let app = XCUIApplication()
  // 1
  let tableQuery = app.descendantsMatchingType(.Table)

  // 2
  let workoutTable = tableQuery["Workouts Table"]
  let cellQuery = workoutTable.childrenMatchingType(.Cell)

  let identifier = "Ray's Full Body Workout"
```

```
    let workoutQuery = cellQuery
      .containingType(.StaticText, identifier: identifier)
    let workoutCell = workoutQuery.element
    workoutCell.tap()

    // 3
    let navBarQuery = app.descendantsMatchingType(.NavigationBar)
    let navBar = navBarQuery[identifier]

    let buttonQuery = navBar.descendantsMatchingType(.Button)
    let backButton = buttonQuery["Workouts"]
    backButton.tap()
  }
```

This test method is fairly small, but it contains classes and concepts you haven't encountered before – you'll read about them shortly. In the meantime, here's what the code does, in broad terms:

1. Get references to all of the tables in the app.
2. Find the Workouts table using the `"Workouts Table"` accessibility identifier you added earlier. After that, you simulate a tap on the cell that contains the static text `"Ray's Full Body Workout"`.
3. Simulate a tap on the back button to go back to the list of workouts. The back button is in the navigation bar and currently says `"Workouts"`.

> **Note**: This test is rather verbose, and as it turns out, can be more concise. You'll get the chance to refactor it shortly.

Go ahead and run this test. To run it in isolation, tap the diamond-shaped icon next to the method declaration:

Once the app has built and launched, the simulator does something you've probably never seen before. It becomes possessed! Specifically, it simulates tapping into Ray's Full Body Workout and then tapping the back button.

If you've written regular XCTest tests before, you know that they usually rely on one or more assertions, such as `XCTAssertTrue`, `XCTAssertFalse` or `XCTAssertEquals`. You may be thinking that this is not a real test since it has no assertions.

Although you can add assertions, you don't have to explicitly assert anything in a UI test. If the test expects to find a specific UI element on the screen, like a button that says *Workouts*, but doesn't find it, the test will fail. In other words, tapping through your app *implicitly* asserts that UI looks and behaves a certain way.

UI test classes

There are three main classes involved in UI testing: `XCUIApplication`, `XCUIElement` and `XCUIElementQuery`. They're difficult to distinguish in `testRaysFullBodyWorkout()` because of Swift's type inference, but they're there! Here's a short description of what they do:

- **XCUIApplication** is a proxy for your application. You use it to launch and terminate the application as you start and end UI tests. Notice that `setup()` in **WorkoutsUITests.swift** launches the app. This means you're launching your `XCUIApplication` before every UI test in the file. `XCUIApplication` is also the root in the element hierarchy visible to your test.
- **XCUIElement** is a proxy for UI elements in the application. Every `UIKit` class you can think of can be represented by an `XCUIElement` in the context of a UI test. How? `XCUIElement` has a type (e.g. `.Cell`, `.Table`, `.WebView`, etc.) as well as an identifier. The identifier usually comes from the element's accessibility information, such as its accessibility identifier, label or value.

  So what can you do with an `XCUIElement`? You can tap, double-tap and swipe on it in every direction. You can also type text into elements like text fields.
- **XCUIElementQuery** queries an `XCUIElement` for sub-elements matching some criteria. The three most common ways to query elements is with `descendantsMatchingType(_:)`, `childrenMatchingType(:_)` and `containingType(_:)`.

Note: Remember that `XCUIApplication` and `XCUIElement` are only *proxies*, not the actual objects. For example, the type `XCUIElementType.Button` can either mean a `UIButton` or a `UIBarButtonItem` or it can be any other button-like UI element!

UI testing convenience methods

Now to add another step to the current test. When the test steps into the workout detail page, it's also going to scroll down and tap the **Select & Workout** button. This brings up an alert controller, so your test will dismiss once you tap **OK**. Finally, it'll return to the workout list screen like before. You'll also refactor the test to make it more concise.

In **WorkoutsUITests.swift**, go to `testRaysFullBodyWorkout()` and replace the entire method with the following:

```
func testRaysFullBodyWorkout() {
  let app = XCUIApplication()

  //1
  let identifier = "Ray's Full Body Workout"

  let workoutQuery = app.tables.cells
    .containingType(.StaticText, identifier: identifier)
  workoutQuery.element.tap()

  //2
  app.tables["Workout Detail Table"].swipeUp()
  app.tables.buttons["Select & Workout"].tap()
  app.alerts.buttons["OK"].tap()

  //3
  app.buttons["Workouts"].tap()
}
```

That's a lot shorter than it was before! Here's what changed in the code:

1. You didn't need to use the accessibility identifier `"Workout Table"` after all. Instead, you get *all* tables in the app and then get all of their cells. Notice that you replaced `descendantsMatchingType(.Table)` with convenience method `tables` and `childrenMatchingType(.Cell)` with convenience method `cells`.

   The element query `descendantsMatchingType(_:)` is so common that Apple provided convenience methods for all the common types. `childrenMatchingType(_:)`doesn't have convenience methods, but using `descendantsMatchingType(_:)` has the same effect in this case.

2. Here's your extra step. Once in the workout detail screen, you find the appropriate table view by its accessibility identifier, scroll downwards by swiping up and tap on **Select & Workout**. Again, notice you don't need to specify *which* table you're talking about. You can drill down from the app to its tables to the tables' buttons, then disambiguate using the button's title. You do the same with the alert's **OK** button, except this time you go through all of the app's *alerts* instead of through all of the app's *tables*.

3. In the previous implementation of this test, you first referenced the navigation bar to get to its back button. Now you directly query the app's buttons and tap on the one identified by the title `"Workouts"`.

Run `testRaysFullBodyWorkout()` one more time. The same sequence of events plays out, except this time, the test taps on **Select & Workout** and dismisses the alert controller.

However, when it tries to return to the list of workouts...splat! The test fails.

```
38  func testRaysFullBodyWorkout() {
39    let app = XCUIApplication()
40
41    //1
42    let identifier = "Ray's Full Body Workout"
43
44    let workoutQuery =
45    app.tables.cells.containingType(.StaticText, identifier: identifier)
46    workoutQuery.element.tap()
47
48    //2
49    app.tables.buttons["Select & Workout"].tap()
50    app.alerts.buttons["OK"].tap()
51
52    //3
53    app.buttons["Workouts"].tap()
54  }
      UI Testing Failure - Multiple matches found:
```

Why did it fail? Xcode gives you the error message **"UI Testing Failure — Multiple matches found"**. You get this error when you're expecting one `XCUIElement`, but instead you get multiple.

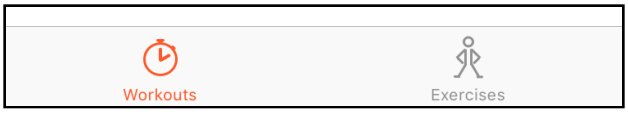
How do you fix this? You have three options when you want to drill down from a set of query results to single `XCUIElement`:

1. You can use **subscripting** if the element you want has a unique identifier. For example, `buttonsQuery["OK"]`.
2. You can use **indexing** if the element you want is located at a particular index. For example, you can use `tables.cells.elementAtIndex(0)` to get the first cell in a table view.
3. If you're *sure* a query resolves down to one element, you can use the `element` property for `XCUIElementQuery`.

If you use any of the three techniques shown above and end up with more than one `XCUIElement`, the test fails. This is because the UI testing framework has no way of knowing which element you actually want to interact with. In this case, `testRaysFullBodyWorkout()` failed because the query in the final line of the test resulted in two buttons with the same identifier:

```
app.buttons["Workouts"]
```

Can you find the duplicates? One is in the top-left, next to the back button – this is the one you meant for the test to tap. The second is inside the **Workouts** tab at the bottom left of the screen:



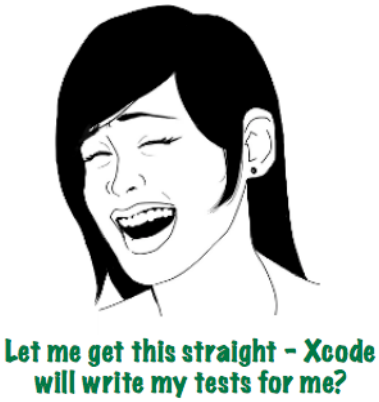
Whoops! Fix the test by replacing the final line of `testRaysFullBodyWorkout()` with the following:

```
app.navigationBars.buttons["Workouts"].tap()
```

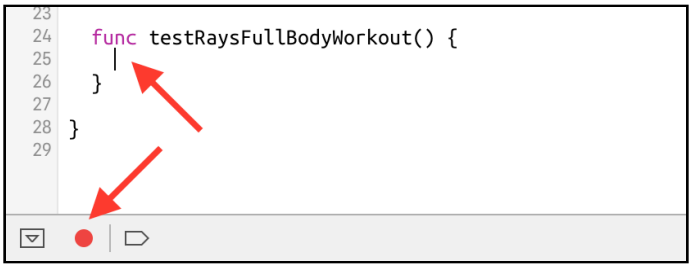
Adding `navigationBars`, which is short for is short for `descendantsMatchingType(.NavigationBar)`, between `app` and `buttons` makes it clear to the UI testing framework that you want the Workouts button located in a navigation bar. Re-run your UI test to verify that it passes now.

UI recording

It's great to know how to write UI tests from scratch, but there is an easier way to get the job done: *UI recording*. With UI recording, you can simply "act out" the steps of your test in the simulator and Xcode will auto-magically translate your actions into UI testing code.



To check it out in action, delete the current contents of `testRaysFullBodyWorkout()`. Place your cursor inside the empty method, then click on the red **Record UI Test** button at the bottom of the editor:



The UI recording button builds and launches your app. Once that's done, "act out" the steps of your tests.

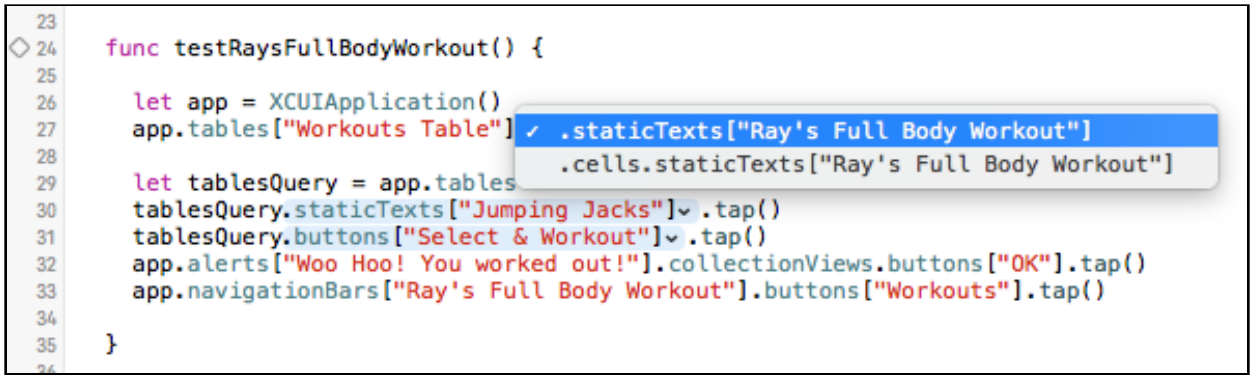
- Tap on **Ray's Full Body Workout**, then scroll down and tap **Select & Workout**.
- Dismiss the alert controller by tapping **OK** and finally tap the **back** button.
- Tap the **record** button again, or Xcode's main **stop** button to stop recording.

Your generated test method should look something like this:

```
func testRaysFullBodyWorkout() {

  let app = XCUIApplication()
  app.tables["Workouts Table"]
    .staticTexts["Ray's Full Body Workout"].tap()

  let workoutDetailTableTable =
    app.tables["Workout Detail Table"]
  workoutDetailTableTable.otherElements["EXERCISES"].swipeUp()
  workoutDetailTableTable.buttons["Select & Workout"].tap()
  app.alerts["Woo hoo! You worked out!"].collectionViews
    .buttons["OK"].tap()
  app.navigationBars["Ray's Full Body Workout"]
    .buttons["Workouts"].tap()
}
```

Magic! Depending on exactly where you swiped and which version of Xcode you're running, the generated code may be different from what you see above. Run the generated test to verify that it simulates your steps one by one.

You'll notice that some of the generated lines of code have **tokens** that contain several options. Click one to see the available options:



There are many ways of querying the same UI elements, and in some cases, Xcode can only make guesses about the steps you want your test to take. With these tokens, Xcode gives you options to help you disambiguate elements.

Once you're happy with a particular path, double-click on the blue **token** to make it final.

> **Note**: Even if you're hardcore and opt to write your tests manually, you can still use UI recording to find out what the testing framework "sees" as you tap around the simulator. It's a good alternative to using the system-provided Accessibility Inspectors.

Run the test again to check it's doing everything you expect. Do a victory dance when you get the desired results.

There are a couple of things to keep in mind as you start writing UI tests: Although they *can* be easier to write, when they fail they can also be harder to debug than regular unit tests. They can also be quite brittle; if you make changes to your UI, you may need to dedicate some time to updating your tests.

Where to go from here?

You've seen how powerful and useful testing can be in Xcode 7. You've explored code coverage reports, taken a look at Swift 2.0's new `@testable` modifier, and you've written and recorded UI tests.

As always, exploring the final project for this chapter is a great way to see how everything came together.

If you ever find yourself fixing the same bug over and over, think back to this chapter! Check the code coverage for the files that contain the bug. If the code coverage is low or incomplete, consider writing more unit tests or even some UI tests that validate the feature.

There are a couple of WWDC sessions from 2015 that are worth looking at to find out more about the topics covered in this chapter:

- *Session 406 — UI Testing in XCode*: apple.co/1N1Eg0I
- *Session 410 — Continuous Integration and Code Coverage in Xcode*: apple.co/1J1n1Kd

You can also read Chapter 29, "What's New with Testing", in our *iOS 8 by Tutorials* book and the "Unit Testing in Xcode 5" chapter in *iOS 7 by Tutorials*.

Chapter 14: Location and Mapping

By Vincent Ngo

Despite a slightly shaky start to its mapping effort in iOS 6, Apple has continued to enhance its mapping and location frameworks every year. iOS 9 is no exception, with a number of great updates to both MapKit and Core Location.

One of the most useful improvements is the addition of transit directions to Apple Maps, with assistance for navigating subways, trains, buses and more. Transit will launch in a small number of cities to start, but will likely roll out on a wider scale as time passes.

This chapter will show you how to take advantage of the following new features:

- New methods to customize the appearance of Maps in your app
- Transit directions in Apple Maps
- Estimated travel times for transit directions
- Single location updates using Core Location

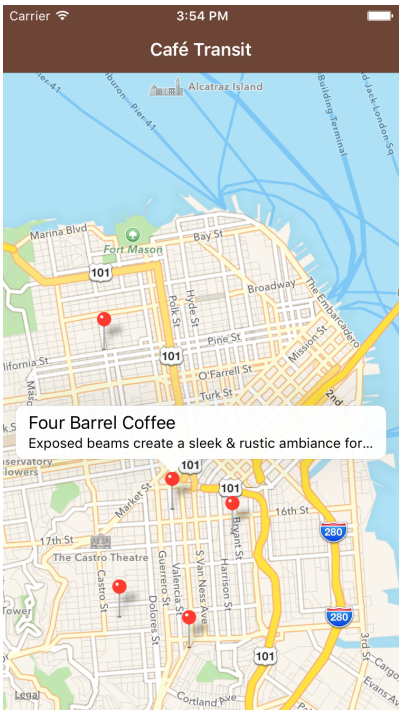
The sample app for this chapter, Café Transit, is for all the coffee aficionados out there. It can help you in your eternal search for good coffee. Currently, it only shows a handful of nearby coffee shops (well, nearby if you're in San Francisco!) and marks them on the map using standard map pins.

By the time you've finished this chapter, your app will show lots of useful information for each coffee shop, including a rating, pricing information and opening hours. Your app will also provide transit directions to a particular coffee shop, and even let you know what time you'll need to leave and when you're likely to arrive.

> **Note**: This chapter will be easier to follow if you have some basic MapKit knowledge. If you need to brush up, take a look at our Getting Started With MapKit tutorial bit.ly/1PrurqE.

Getting started

Open the starter project for this chapter. Run it up and you'll see it's built with a standard MapKit `MKMapView`; tap on a pin to reveal the coffee shop's name and a brief description:

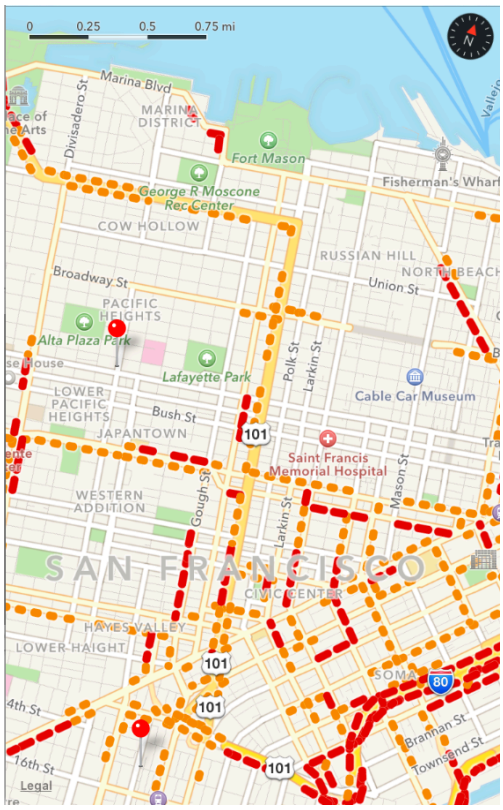


Open **ViewController.swift**; `setupMap()` and `addMapData()` center the map on San Francisco and add an annotation to each coffee shop. The model code for coffee shops is in **CoffeeShop.swift**, which also takes care of loading all of the sample coffee shop data from **sanfrancisco_coffeeshops.plist**.

Finally, take a quick look at **CoffeeShopPinDetailView.swift** and **CoffeeShopPinDetailView.xib**; these define the custom annotations you'll add later to spice up the map view. These annotations will show a rating, price information, and opening hours — everything you need to make your app shine!

Customizing maps

Prior to iOS 9, the only items you could programatically toggle on and off in an `MKMapView` were buildings and places of interest. MapKit in iOS 9 introduces three new boolean properties that let you toggle the map's **compass**, **scale bar** (the small ruler that shows distances on the map) and **traffic** display. You can choose to remove these to clean up your map display, or leave them on the screen to give your users extra information as they navigate:

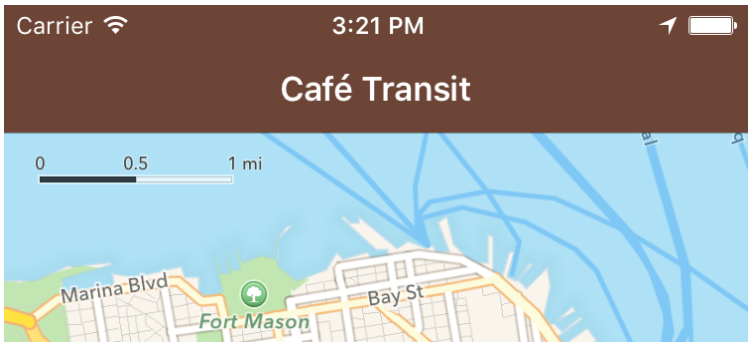


Café Transit would benefit from showing the map's scale — giving the users an idea how far they have to go to get their caffeine fix.

Open **ViewController.swift** and add the following code at the very beginning of `setupMap()`:

```
mapView.showsScale = true
```

Build and run your app; you should see the scale appear in the top left of the map:

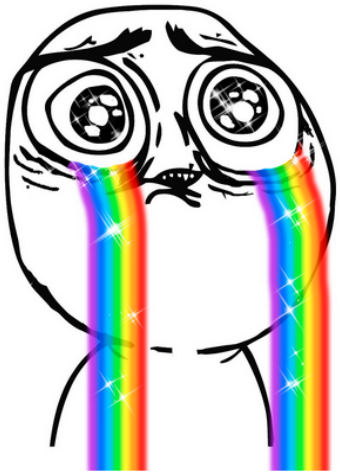


As you pan and zoom around the map, the scale updates itself to match the map's current zoom level.

Customizing map pins

Since iOS 3, MapKit pins have had a `pinColor` property that let you select any color you wanted...as long as it was red, green or purple. But what if you wanted a yellow pin? Or an orange pin? Or a *chartreuse* pin? You were out of luck.

iOS 9 deprecates the `pincolor` property on `MKPinAnnotationView` in favor of the shiny new `pinTintColor`. And get this — you can set it to *any color you like*!



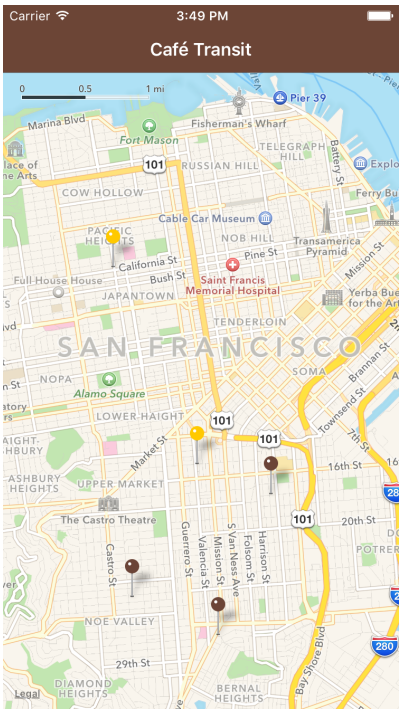
Café Transit currently uses plain old red map pins, which don't really fit in with the coffee aesthetic. They'd look better with the same brown shade used throughout the app. And you could even use a different color to highlight cafés with a 5-star rating.

Add the following code to `mapView(_:viewForAnnotation:)` in **ViewController.swift**, just before the `return` statement at the end:

```
if annotation.coffeeshop.rating.value == 5 {
  annotationView!.pinTintColor =
    UIColor(red:1, green:0.79, blue:0, alpha:1)
} else {
  annotationView!.pinTintColor =
    UIColor(red:0.419, green:0.266, blue:0.215, alpha:1)
}
```

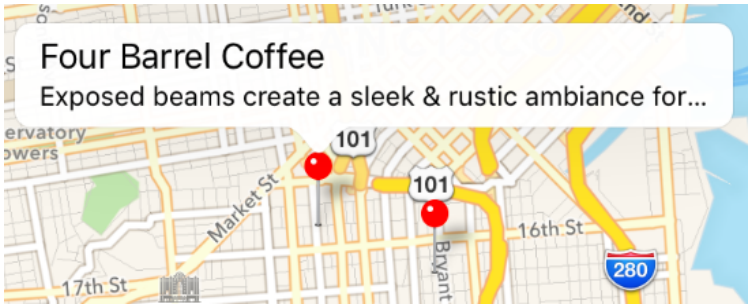
This chooses a `pinTintColor` that depends on the star rating of the coffee shop; gold for a 5-star rating and brown otherwise.

Build and run your app, and check out your fancy new pins:



Customizing annotation callouts

Each map pin (or annotation view) can show a **callout** when you tap it. This is simply a popover that appears above your annotation view on the map, and can provide extra information about a particular location, like so:



Until now, you've been limited in your customization of annotation callouts; you could set a title, subtitle and left and right accessory views. For any other kind of customization, you had to try to add a custom view to the annotation view, which wasn't an easy task.

iOS 9 makes the whole process much simpler, with the new `detailCalloutAccessoryView` property on `MKAnnotationView`, You can set this to any view you like, which gives you almost unlimited customization options for your callouts. You could even use the new `UIStackView`, or some kind of crazy collection view if you so choose!

Managing callout size

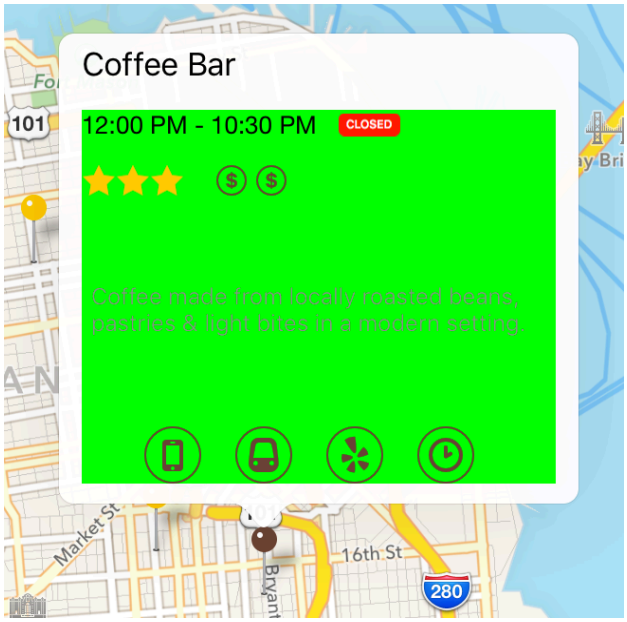
Callouts will use the *intrinsic content size* of your custom view to size themselves appropriately. Your custom callouts can take advantage of this in two ways:

1. Use Auto Layout to lay out your custom view, and let intrinsic content size do its thing.
2. You can override `intrinsicContentSize` within your custom view size and return a size of your choice.

> **Note**: For more information on intrinsic content size and Auto Layout, take a look at Apple's "Implementing a Custom View to Work with Auto Layout" documentation: apple.co/1PHbKA5.

The XIB for `CoffeeShopPinDetailView` uses `UIStackView` and Auto Layout, so you don't have to manually specify `intrinsicContentSize`. Feel free to explore how the XIB makes use of `UIStackView` and constraints.

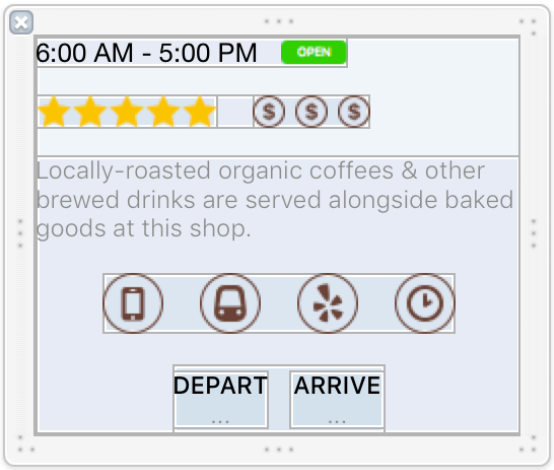
Custom callouts can't fill the entire area of the callout popover, since iOS adds the annotation title and a certain amount of padding. The screenshot below shows the area filled by `detailCalloutAccessoryView` in green:



Keep this in mind when designing your custom callout views, as there's currently no way to modify the padding or title area.

Adding a custom callout accessory view

With that theory out of the way, it's time to add a custom callout of your own. **CoffeeShopPinDetailView.xib** defines the UI for the callout accessory view as shown below:



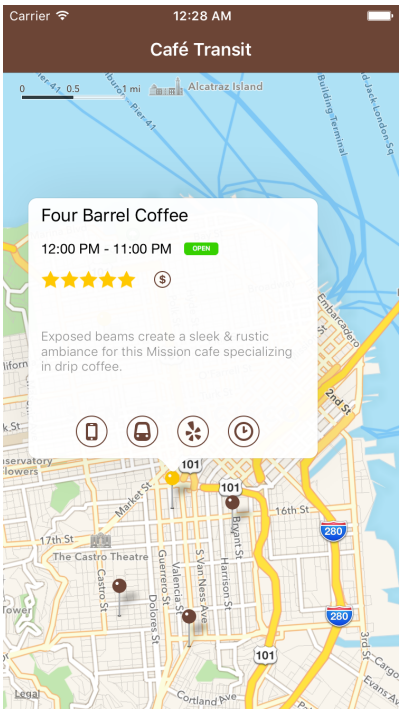
The callout shows the opening hours, star and cost rating and a description of the coffee shop, along with a set of action buttons for such things as phoning the coffee shop or viewing their Yelp page.

Open **ViewController.swift** and add the following code to `mapView(_:viewForAnnotation:)`, just before the `return` statement:

```
let detailView = UIView.loadFromNibNamed(identifier) as!
  CoffeeShopPinDetailView
detailView.coffeeShop = annotation.coffeeshop
annotationView!.detailCalloutAccessoryView = detailView
```

First, you load the `CoffeeShopPinDetailView` from its XIB file. Then you assign a coffee shop to the detail view in order to populate the view's labels and subviews. Finally, you assign the view to the annotation view's `detailCalloutAccessoryView` property.

That's all the code it takes! Build and run your app, tap on one of the pins and you should see your custom annotation at work:

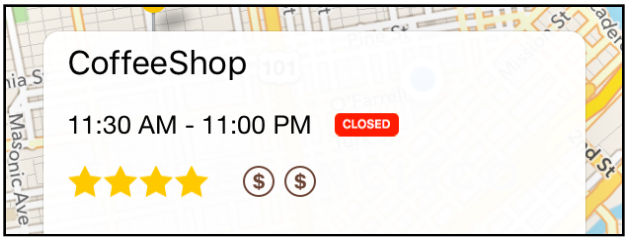


Note: Tapping the phone button in the callout will only work on an actual device.

Tap the Yelp button to open up Safari and load the coffee shop's Yelp review page. Tapping the clock button won't show you any useful information; you'll implement actions for the transit and clock functions later in this chapter.

Supporting time zones

The custom callout view you added in the previous section contains a small image to indicate whether a particular coffee shop is currently open for business:



Open up **CoffeeShop.swift** and find `isOpenNow`; this is a computed property that determines the opened or closed state of the shop:

```
40
41    static var timeZone = NSTimeZone(abbreviation: "PST")!
42
43    /// Calculates whether a coffee shop is currently open for business
44    var isOpenNow: Bool {
45      let calendar = NSCalendar.currentCalendar()
46      let nowComponents = calendar.componentsInTimeZone(CoffeeShop.timeZone, fromDate: NSDate())
47
```

This property uses `NSDate()` to get the current time and then converts it to the time zone of the coffee shop; the shop's opening hours are stored in its local time zone so you have to convert the result of `NSDate()` to the time zone of the shop. You then use this to calculate whether the current local time falls within the range of opening hours.

Easy enough, but take a look at the time zone definition above `isOpenNow`:

```
static var timeZone = NSTimeZone(abbreviation: "PST")!
```

The timezone is hardcoded to PST! Although Café Transit currently only contains some sample coffee shops in San Francisco, it would be nice if the time zone could be inferred from the location of the coffee shop in case you add more in the future.

iOS 9 adds a handy `timeZone` property to both `MKMapItem` and `CLPlacemark`; you can use this to ensure you use the correct time zone no matter where the shop is.

Still in **CoffeeShop.swift**, find `allCoffeeShops`, and replace the `return` statement with the following code:

```
// 1
let shops = array.flatMap { CoffeeShop(dictionary: $0) }
  .sort { $0.name < $1.name }

// 2
let first = shops.first!
let location = CLLocation(latitude: first.location.latitude,
  longitude: first.location.longitude)

// 3
let geocoder = CLGeocoder()
geocoder.reverseGeocodeLocation(location) { (placemarks, _) in
  if let placemark = placemarks?.first, timeZone =
    placemark.timeZone {

    self.timeZone = timeZone
  }
}

return shops
```

This code performs a couple of functions:

1. This is simply the value of the previous `return` statement, but stored in a variable instead.
2. You then create a `CLLocation` that represents the first coffee shop in the list for

use with `CLGeocoder`.

3. Finally, you *reverse geocode* the coffee shop's location using an instance of `CLGeocoder`. This takes the latitude and longitude of the coffee shop and produces a `CLPlacemark` with extra information about the location. This extra information includes the new `timeZone` property, which you then assign to the `timeZone` property of the `CoffeeShop` struct.

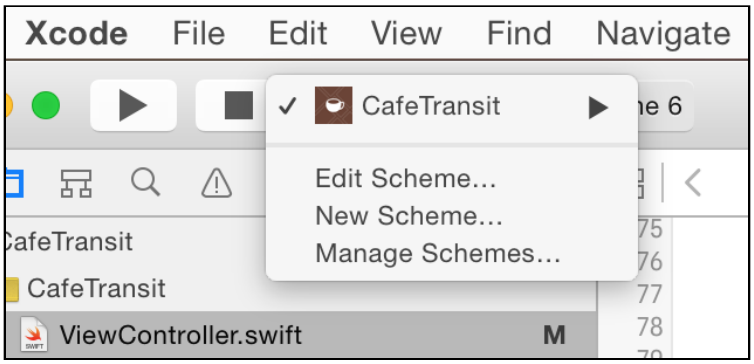
Build and run your app now; check that the opening hours labels are still showing the correct value. Remember, they're based on the current time in San Francisco, not the current time of *your* location!

> **Note**: For the purposes of this chapter, you've simply fetched the time zone for a single coffee shop. In a real project, you'd have to check the time zone for each coffee shop, as they may be spread across different time zones.

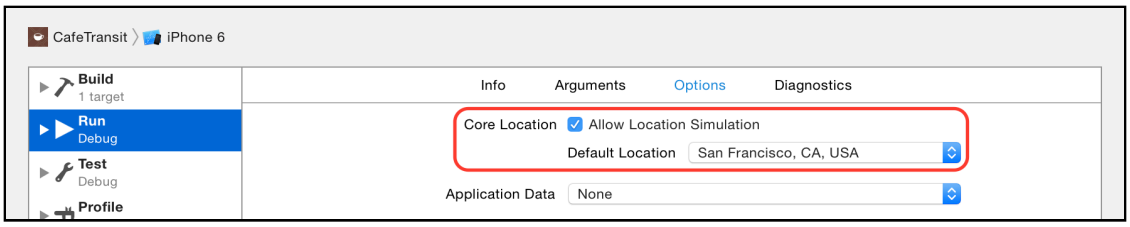
Simulating your location

All of Café Transit's sample coffee shops are based in San Francisco. Statistically, it's quite likely that *you* aren't based in San Francisco. :] The rest of this chapter will make use of the user's location, so it would be pretty useful to at least *pretend* to be there. Xcode lets you simulate your location, which will make testing Café Transit much easier!

With the starter project open, click on the **CafeTransit** scheme and choose **Edit Scheme...**:



Select **Run** in the left pane, and **Options** from the tab bar at the top of the right pane. Enable **Core Location > Allow Location Simulation**, and set your **Default Location** to **San Francisco, CA, USA** as shown below:



Click **Close** to save your location settings; your app now thinks you're in San Francisco. You'll use the simulated location in the next section as you plot the user's location on the map. You'll also request the user's location in order to provide transit directions from the user's current location to a selected coffee shop.

Making a single location request

Before iOS 9, accessing the user's current location was a byzantine process. You had to create a `CLLocationManager`, implement some delegate methods, and then call `startUpdatingLocation()`. This would call the location manager delegate methods repeatedly with updates to the user's location. Once the location reached an acceptable level of accuracy, you then called `stopUpdatingLocation()` to stop the location manager. If you didn't stop it, you could quickly drain the user's battery!

Core Location in iOS 9 collapses this process into a single method call: `requestLocation()`. This still makes use of the existing delegate callback methods, but there's no longer a need to manually start and stop the location manager. You just specify the accuracy you desire, and Core Location will provide the location to you once it's narrowed down the user's position. It only calls your delegate once, and only returns a single location.

Enough theory — you know how your users get when they're deprived of their daily cuppa! Time to add some locating logic.

Adding a location manager

First, open **ViewController.swift** and add the following line just below the class declaration:

```
lazy var locationManager = CLLocationManager()
var currentUserLocation: CLLocationCoordinate2D?
```

The code lazily creates a `CLLocationManager` object the first time it's called. You also create a `CLLocationCoordinate2D` property to store the user's current location.

Next, add the following lines to the end of `viewDidLoad()`:

```
locationManager.delegate = self
locationManager.desiredAccuracy =
  kCLLocationAccuracyHundredMeters
```

Here you set the delegate for location manager, and you determine how accurate you want the coordinates to be. Setting **desiredAccuracy** tells the system to only provide you with the user's location once it's accurate enough for your purposes. In some cases, the system might not reach the level of accuracy you want, and will therefore provide you with a location of a lower accuracy than you requested.

Next, add the following extension to the bottom of **ViewController.swift** to add conformance to the `CLLocationManagerDelegate` protocol:

```
// MARK:- CLLocationManagerDelegate
extension ViewController: CLLocationManagerDelegate {

  func locationManager(manager: CLLocationManager,
didChangeAuthorizationStatus status: CLAuthorizationStatus) {
    if (status == CLAuthorizationStatus.AuthorizedAlways ||
        status == CLAuthorizationStatus.AuthorizedWhenInUse) {
      locationManager.requestLocation()
    }
  }

  func locationManager(manager: CLLocationManager,
    didUpdateLocations locations: [CLLocation]) {

    currentUserLocation = locations.first?.coordinate
  }

  func locationManager(manager: CLLocationManager,
    didFailWithError error: NSError) {

    print("Error finding location: " +
      \(error.localizedDescription)")
  }
}
```

This extension implements three methods of `CLLocationManagerDelegate`: in `locationManager(_:didFailWithError:)`, you simply log errors if they occur; in `locationManager(_:didUpdateLocations:)`, you store the coordinates of the first location returned in `currentUserLocation`; in `locationManager(_:didChangeAuthorizationStatus)`, you check to see if the user has enable you to use their location and if so you request for their location. When you use your new `requestLocation()`, you'll receive the location one time only.

Now you need to *call* `requestLocation()` from somewhere else.

Still in **ViewController.swift**, add the following method to `ViewController`, below `centerMap(_:atPosition:)`:

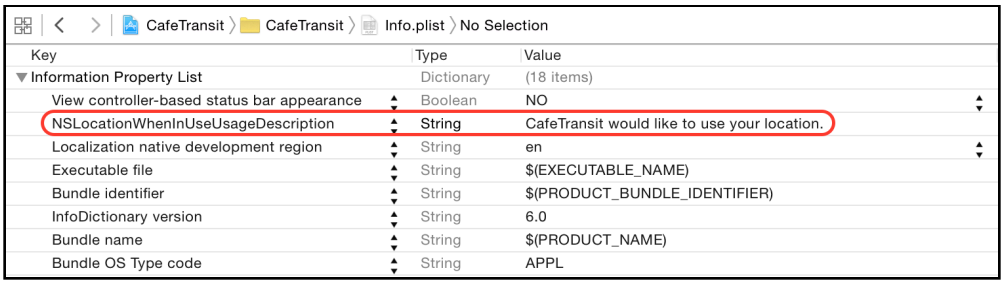
```
private func requestUserLocation() {
  mapView.showsUserLocation = true    // 1
  if CLLocationManager.authorizationStatus() ==
    .AuthorizedWhenInUse { // 2

    locationManager.requestLocation()   // 3
  } else {
```

```
      locationManager.requestWhenInUseAuthorization()    // 4
    }
  }
```

Taking each numbered comment in turn:

1. Show the user's location on the map.
2. Before you can request the user's location, you must first ask for permission to do so. This line checks whether you already have permission.
3. Next, call **requestLocation()** to request the user's current position. When done, this invokes a call to `locationManager(_:didUpdateLocations:)`, which you just implemented.
4. If you don't have permission to use the user's location, prompt for it.

> **Note**: When calling `requestWhenInUseAuthorization()`, you must have your Info.plist file configured with a value for the key `NSLocationWhenInUseUsageDescription` stating *why* you would like access to the user's location. This message will be displayed to the user in the usual permission alert that pops up. To save you time, this setting has already been added to Café Transit's Info.plist as shown below:



Next, add the following implementation underneath **viewDidLoad()**:

```
override func viewDidAppear(animated: Bool) {
  super.viewDidAppear(animated)

  requestUserLocation()
}
```

This calls the `requestUserLocation()` method you just wrote at first launch when the map view appears.

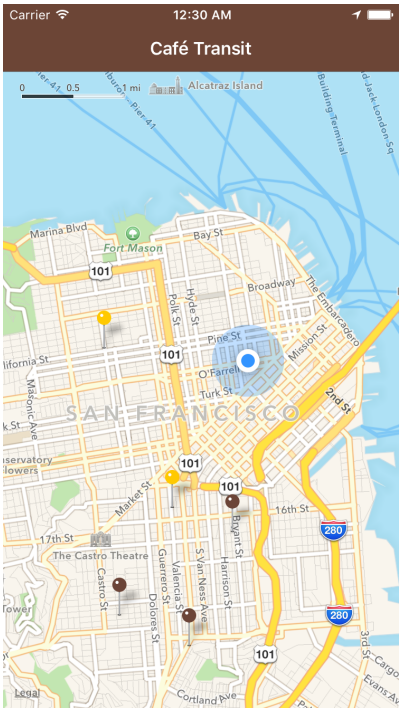
Finally, find the `MKMapViewDelegate` extension near the bottom of **ViewController.swift** and add the following method to it:

```
func mapView(mapView: MKMapView,
  didSelectAnnotationView view: MKAnnotationView) {
  if let detailView = view.detailCalloutAccessoryView
```

```
      as? CoffeeShopPinDetailView {

      detailView.currentUserLocation = currentUserLocation
    }
  }
```

This passes the user's current location into an annotation whenever the annotation appears. You'll need access to this location in the next section to request transit directions.

That was quite a bit of code to get through — you've done well! Build and run your app; you should see a blue dot appear on the map showing your simulated location:



Sure, that doesn't seem like a lot when you consider all the code you wrote, but you're building up to some really cool features in the next section.

Requesting transit directions

Now that you have the user's current location, you're nearly ready to provide transit directions to coffee shops!

Open **CoffeeShopPinDetailView.swift** and add the following method below `// MARK:- Transit Helpers` near the bottom of the file:

```
  func openTransitDirectionsForCoordinates(
    coord:CLLocationCoordinate2D) {

    let placemark = MKPlacemark(coordinate: coord,
```

```
      addressDictionary: coffeeShop.addressDictionary) // 1
    let mapItem = MKMapItem(placemark: placemark)  // 2
    let launchOptions = [MKLaunchOptionsDirectionsModeKey:
      MKLaunchOptionsDirectionsModeTransit]  // 3
    mapItem.openInMapsWithLaunchOptions(launchOptions)  // 4
  }
```

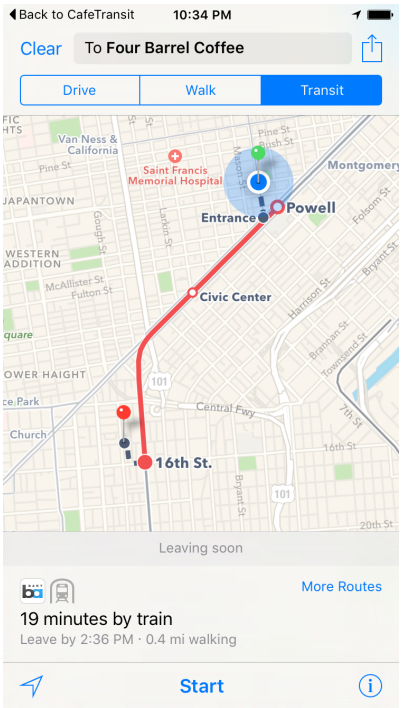
This is a helper method that gives you transit directions to the coordinates you pass in. Here's what the code does:

1. Creates an `MKPlacemark` to store your coordinates. Placemarks usually have an associated address, and the coffee shop model provides a basic one which simply includes the coffee shop's name.
2. Initializes an `MKMapItem` with the placemark.
3. Specifies that you want to launch Maps in **transit** mode.
4. Launches Maps to show transit directions to the requested location.

All you need to do now is replace the `TODO` in `transitTapped()` with a call to your method above and pass in the coffee shop's location:

```
  openTransitDirectionsForCoordinates(coffeeShop.location)
```

Build and run your app; tap a coffee shop and click the train icon in the callout. You'll be launched straight into transit directions to the coffee shop:



Querying transit times

The final new feature of MapKit to add to Café Transit is querying public transit journey information. The `MKETAResponse` class includes the following useful properties:

```
public var expectedTravelTime: NSTimeInterval { get }
@available(iOS 9.0, *)
public var distance: CLLocationDistance { get }
@available(iOS 9.0, *)
public var expectedArrivalDate: NSDate { get }
@available(iOS 9.0, *)
public var expectedDepartureDate: NSDate { get }
@available(iOS 9.0, *)
public var transportType: MKDirectionsTransportType { get }
```

These properties tell you the distance of a trip, the expected duration of travel and the arrival and departure times. This lets you provide some high-level trip information, without pushing the user out to a separate app.

Tap a coffee shop in Café Transit and then tap on the clock icon; the view animates upwards to show you estimated departure and arrival times, but there aren't any yet. That's where you and MapKit will join forces to save your user's coffee crisis! :]

Open **CoffeeShopPinDetailView.swift** and add the following method just after `openTransitDirectionsForCoordinates(_:)`:

```
func requestTransitTimes() {
  guard let currentUserLocation = currentUserLocation else {
    return
  }

  // 1
  let request = MKDirectionsRequest()

  // 2
  let source = MKMapItem(placemark:
    MKPlacemark(coordinate: currentUserLocation,
    addressDictionary: nil))
  let destination = MKMapItem(placemark:
    MKPlacemark(coordinate: coffeeShop.location,
    addressDictionary: nil))

  // 3
  request.source = source
  request.destination = destination
  request.transportType = MKDirectionsTransportType.Transit

  // 4
  let directions = MKDirections(request: request)
  directions.calculateETAWithCompletionHandler {
    response, error in
    if let error = error {
      print(error.localizedDescription)
```

```
      } else {
        // 5
        self.updateEstimatedTimeLabels(response)
      }
    }
  }
```

Here's how you request the transit times:

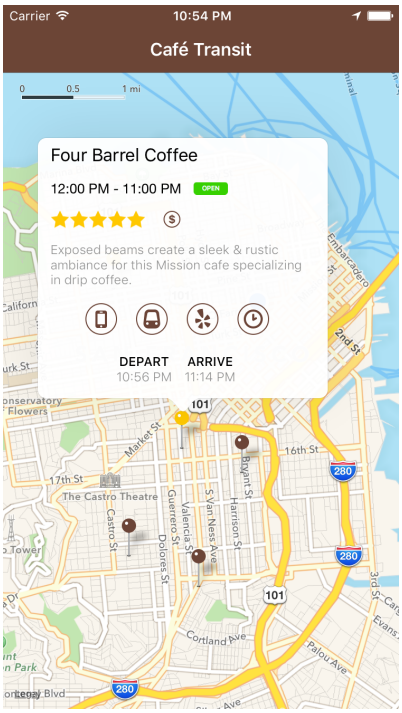
1. Once you ensure a user location's been set, initialize an instance of `MKDirectionsRequest`.
2. Create two instances of `MKMapItem` to represent the user's current location and the coffee shop's location. There's no address dictionary populated here because you only need the latitude and longitude.
3. Configure the `MKDirectionsRequest` object with the source, destination, and type of transport.
4. Create an `MKDirections` object, initialize it with the `MKDirectionsRequest` and instruct it to perform the ETA calculation.
5. If you receive a successful response, update the departure and arrival labels accordingly.

Finally, still in **CoffeeShopPinDetailView.swift**, replace `timeTapped()` with the following:

```
  @IBAction func timeTapped() {
    if timeStackView.hidden {
      animateView(timeStackView, toHidden: false)
      requestTransitTimes()
    } else {
      animateView(timeStackView, toHidden: true)
    }
  }
```

When you tap the clock icon, the time view animates upwards and you send off a request to Apple's servers for the journey's ETA and duration. The time labels on the callout update automagically.

Build and run your app; tap one of the coffee shop pins then tap the clock icon and you'll see an update on when you'll depart and what time you'll arrive! Can't you just smell the beans roasting already? :]



Where to go to from here?

In this chapter you've customized a map view, added a custom callout, requested the user's location, and made use of transit directions and estimated journey times. Awesome stuff!

There are a couple of other MapKit and Core Location updates this chapter didn't cover, including 3D flyovers and a couple of changes to background location updates. For more information about these, check out these related WWDC talks:

- What's New In MapKit: apple.co/1h4r4e7
- What's New in Core Location: apple.co/1EcdPD7

Chapter 15: What's New in Xcode?

By Jawwad Ahmad

The most important tool you use as an iOS developer is Xcode, and each new release brings a variety of features and improvements. In prior chapters, you learned about many of Xcode's new features, such as storyboard references, support for app thinning, improvements to testing and code coverage.

This chapter will introduce you more new features in Xcode, like the new energy gauge and improvements to playgrounds. Along the way, you'll also learn about other features and improvements that will make your time spent in Xcode more productive.

Getting started

In this chapter, you'll work on **Local Weather**, an app that uses your GPS location to show the weather near you.

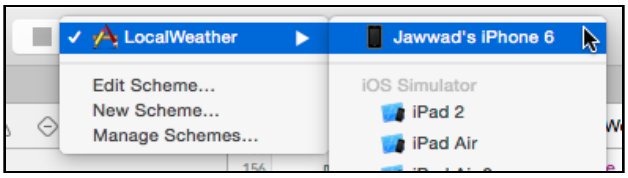
You won't make too many changes to the app, rather you'll use it to explore various new features of Xcode, especially the new energy gauge. Find an iPhone or iPad before you dig in; you'll need to run the app on an actual device since the energy gauge doesn't show on the simulator.

Local Weather uses the *current weather data* API from openweathermap.org, so you'll first need to sign up for an API key at openweathermap.org/register. Don't worry, sign-up is really quick since the only information OpenWeatherMap requires is a username, email address and password.

Once you have signed up, open the starter project for this chapter and in **WeatherViewController.swift** (near the top of the file) replace the text: `"YOUR_API_KEY_HERE"` with your new API key.

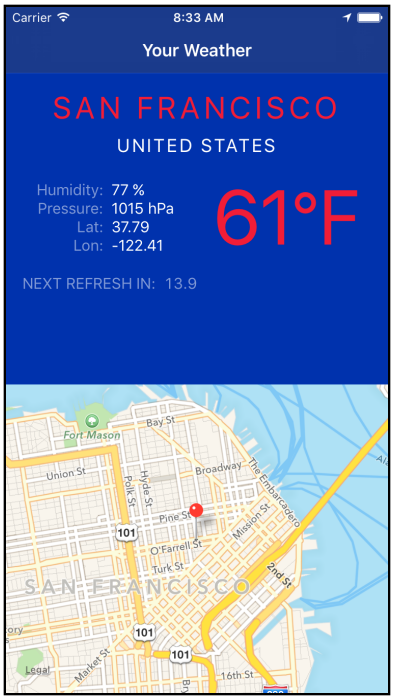
```
let openWeatherMapApiKey = "YOUR_API_KEY_HERE"
```

Now select your device from the destination menu:

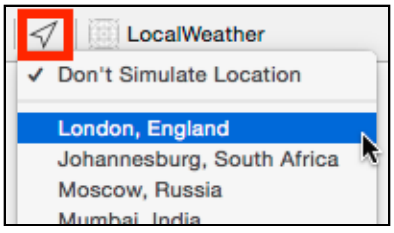


> **Note:** If you can't run the app on your device, it may not be provisioned. See the next section on **free provisioning** to get set up.

After selecting your device, build and run. Then tap **Allow** on the location access request. The location and weather should update fairly quickly:



Although the energy gauge won't show up unless you run the app on a device, you *can* still run it in the simulator. If you do, make sure to select a location using the **Simulate Location** button in the **Debug Area**. You can also use this feature to simulate a different location while running on your device. Perhaps you'd like to find out how cold it is in Moscow or how balmy it is in Maui. No problem, just choose it from the menu.



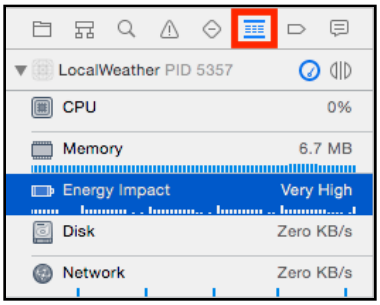
Free provisioning

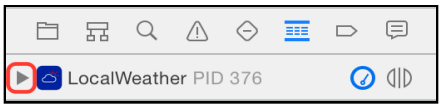
One of the major changes this year is *free provisioning*, which means that anyone with an Apple ID can build and run iOS apps on a physical device without having to join the $99/yr Apple Developer Program. Now you only need to join the paid program if you want to distribute apps on the App Store.

This chapter won't cover the provisioning process, but if your device isn't already provisioned, you can take a look at Apple's documentation to set things up: Launch Your App on Devices Using Free Provisioning (apple.co/1KJ12tJ).

Energy impact gauge

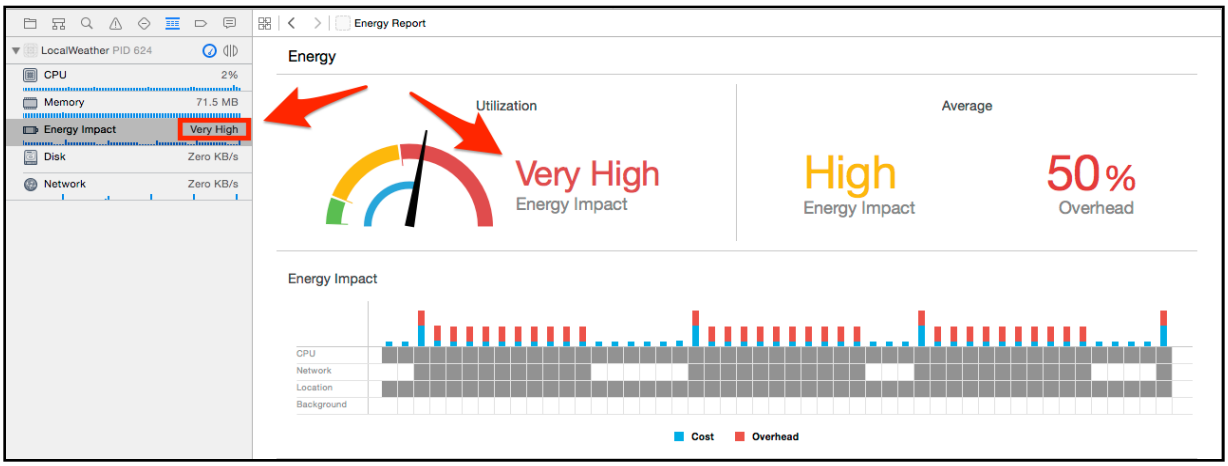
Exploring the new energy gauge will be a major part of this chapter. With the app running on a device, switch to the **Debug navigator** and click on **Energy Impact**:





> **Note:** If you can't see **Energy Impact**, try clicking on the disclosure arrow on the LocalWeather process:

You'll see the new iOS energy gauge, and it shows the app is a bit power hungry!

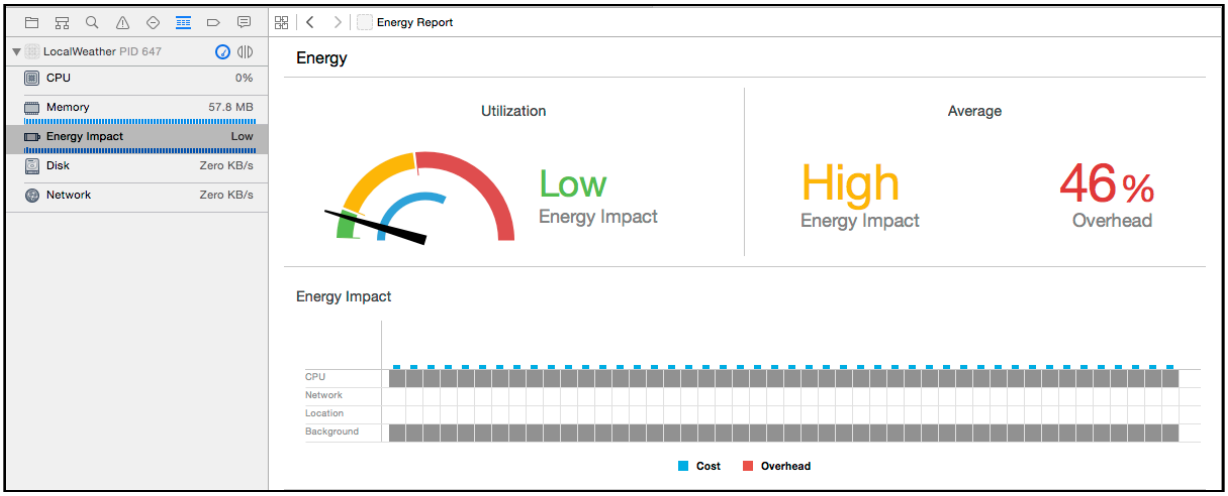


The *Utilization* section on the top left shows you the energy impact at the current moment in time. In the top right section, you see the *average* energy impact, as well as the **overhead**. This includes the energy consumed by system resources to perform work like bringing up radios, as well as the energy used when the resources remain active but idle.

At the bottom, you see four rows of blocks corresponding to CPU, network, location and background. Each block represents a second. If there is any kind of activity during a single second, the box will fill in with a gray block.

Note that the **CPU** and **Location** graphs show continuous activity. The **Network** graph shows an interesting pattern of 10 or 11 seconds of activity, then three to five seconds of inactivity. The **Background** graph is completely clear since the app hasn't been put into the background...yet.

Press the **Home button** to background the app. You'll eventually end up with the following graph that shows the location and network activity have ceased, but you'll have continuous activity for Background and CPU:



Take a look at the console. The log indicates that all of the background tasks

completed in 10.79 seconds, but the background energy graph didn't stop.

```
Method: applicationDidEnterBackground, called at: 2015-08-11 02:53:04 +0000
Background work started at: 2015-08-11 02:53:04 +0000
Doing background work. Task: 1. Completed at: 2015-08-11 02:53:04 +0000
Doing background work. Task: 2. Completed at: 2015-08-11 02:53:05 +0000
Doing background work. Task: 3. Completed at: 2015-08-11 02:53:06 +0000
Doing background work. Task: 4. Completed at: 2015-08-11 02:53:08 +0000
Doing background work. Task: 5. Completed at: 2015-08-11 02:53:09 +0000
Doing background work. Task: 6. Completed at: 2015-08-11 02:53:10 +0000
Doing background work. Task: 7. Completed at: 2015-08-11 02:53:11 +0000
Doing background work. Task: 8. Completed at: 2015-08-11 02:53:12 +0000
Doing background work. Task: 9. Completed at: 2015-08-11 02:53:13 +0000
Doing background work. Task: 10. Completed at: 2015-08-11 02:53:14 +0000
All background work completed at: 2015-08-11 02:53:15 +0000
Background work completed in: 10.79 sec
```

It would appear that Local Weather is a little energy vampire.

Code browsing features

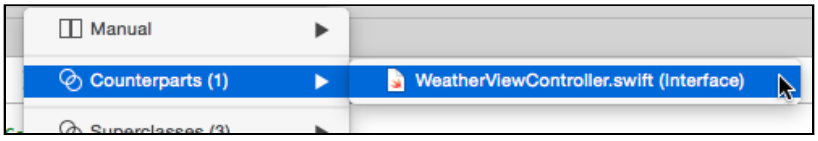
Many of Xcode's new features not only help with development, but also help you get up to speed with a new codebase even faster. In this chapter, you'll use them to get a basic understanding of the app's architecture, and then you'll fix each of the energy issues you just saw.

Interface of Swift classes

If you came to iOS programming in the olden days then you'll remember Objective-C headers. They showed just the public properties and method signatures, and were a great way of getting a quick summary of a class and its capabilities. You'd open up the header file and instantly be able to see what it could do, without having to comb through the implementation code.

Wouldn't it be dreamy if you could have a header file for Swift classes but didn't have to maintain it? Crazy talk you say?

Open **WeatherViewController.swift** and then open the **assistant editor**. Click on the assistant editor menu and choose **Counterparts (1) ▶ WeatherViewController.swift (Interface)**:



> **Note**: You can also select **Generated Interface** instead of **Counterparts ▶ RWHTTPManager.h (Interface)** to view the same result.

Like magic, you'll see the public interface of your class in the assistant editor:

```
internal class WeatherViewController : UIViewController {

    internal let httpManager: RWHTTPManager

    internal var weatherNeedsFetchUpdate: Bool

    internal var networkFetchTimer: NSTimer?

    /// Used to update the countdown label every 0.1 seconds
    internal var countdownUpdateTimer: NSTimer?

    override internal func viewDidLoad()

    /// This method is called every 15 seconds and also upon initial load of the view controller
    internal func requestLocationAndFetchWeather()

    /// Zoom the map to the specified coordinate
    internal func zoomMapToCoordinate(coordinate: CLLocationCoordinate2D)

    /// Requests the weather for a particular location, and then calls `updateViewsWithWeatherData`
    internal func fetchWeatherForCoordinate(coordinate: CLLocationCoordinate2D)

    /// Updates the views and resets the timers
    internal func updateViewsWithWeatherData(weatherData: WeatherData, coordinate: CLLocationCoordinate2D)

    /// Updates `countdownLabel` with the number of seconds until the `networkFetchTimer` will fire.
    internal func updateCountdownLabel()
}
```

However, private variables or methods won't show up in the interface. Take a look at the following part of **WeatherViewController.swift**:

```
  @IBOutlet weak private var countdownLabel: UILabel!
  @IBOutlet weak private var mapView: MKMapView!

  let httpManager = RWHTTPManager(baseURL: NSURL(string: openWeatherMapBaseUrl)!)
  private let locationManager = CLLocationManager()

  // This prevents triggering multiple HTTP requests when locations are received
  var weatherNeedsFetchUpdate = true

  /*
    Used to fetch the weather periodically. Currently every 15 seconds
    */
  var networkFetchTimer: NSTimer?

  /// Used to update the countdown label every 0.1 seconds
  var countdownUpdateTimer: NSTimer?
```

Look at the interface again and note that `locationManager` isn't visible since it's declared `private`, nor are any of the outlets visible since they are all `private`.

> **Note**: Since the class is self-contained, every variable and method could actually have been made `private`. But then there wouldn't have been an interface to demo!

In **WeatherViewController.swift**, delete the `private` access modifier from the following:

```
private let locationManager = CLLocationManager()
```

Press **Command-S** to save the file, and the interface should refresh itself. You'll now see `locationManager` join its friends in the interface:

```
internal class WeatherViewController : UIViewController {

    internal let httpManager: RWHTTPManager

    internal let locationManager: CLLocationManager

    internal var weatherNeedsFetchUpdate: Bool
```

Did you notice that the comment for `countdownUpdateTimer` shows up in the

interface, but the comments for `weatherNeedsFetchUpdate` and `networkFetchTimer` don't? This is because the required documentation comment is nowhere in sight. You need either **///** or **/****.

Add an extra **/** to the comment for `weatherNeedsFetchUpdate` and an extra ***** to the comment for `networkFetchTimer`. **Save** the file, and you'll see both comments show up when the interface refreshes itself:

```
31      /// This prevents triggering multiple HTTP requests when locations are received
32      internal var weatherNeedsFetchUpdate: Bool
33
34      /**
35        Used to fetch the weather periodically. Currently every 15 seconds
36        */
37      internal var networkFetchTimer: NSTimer?
38
39      /// Used to update the countdown label every 0.1 seconds
40      internal var countdownUpdateTimer: NSTimer?
```

Wouldn't it also be cool if you could take an *Objective-C* header file and see what how it would look in Swift? No problem – coming right up!

Generated Swift interface for Objective-C headers

In the **project navigator**, expand the **Helpers** group and you'll see **RWHTTPManager.h** and **RWHTTPManager.m**. The `RWHTTPManager` class is a convenience wrapper around `NSURLSession` and is written in Objective-C.

Click on **RWNetworkHelper.h** to open it in the primary editor:

```
25 @interface RWHTTPManager : NSObject
26
27 @property (nonatomic, strong) NSURL * _Nullable baseURL;
28
29 - (instancetype _Nonnull)initWithBaseURL:(NSURL * _Nonnull)baseURL;
30
31 /// Fetches json at the relativePath (required) and returns JSON or an NSError in the completionBlock
32 - (void)fetchJSONAtPath:(NSString * _Nullable)relativePath
33         completionBlock:(void (^ _Nonnull)(id _Nullable, NSError * _Nullable))completionBlock;
34
35 @end
```

Now take a look at the assistant editor:

```
25 class RWHTTPManager : NSObject {
26
27     var baseURL: NSURL?
28
29     init(baseURL: NSURL)
30
31     /// Fetches json at the relativePath (required) and returns JSON or an NSError in the completionBlock
32     func fetchJSONAtPath(relativePath: String?, completionBlock: (AnyObject?, NSError?) -> Void)
33 }
```

Isn't that just magical? There's an issue though, and it's a bit harder to catch in the RWHTTPManager.h file. Take a look at the Swift interface. In `init(baseURL:)`, the type of `baseURL` is a non-optional `NSURL`, whereas the actual `baseURL` property on the line above it is an optional. The `init` method is correct; `baseURL` should be of type `NSURL` instead of `NSURL?`.

Look at **RWHTTPManager.h** and note that `NSURL` is annotated with `_Nullable`:

```
@property (nonatomic, strong) NSURL * _Nullable baseURL;
```

Change `_Nullable` to `_Nonnull`:

```
@property (nonatomic, strong) NSURL * _Nonnull baseURL;
```

Press **Command-S** to save the file. Once the interface refreshes, you should see that `baseURL` is no longer an optional:

```
var baseURL: NSURL
```

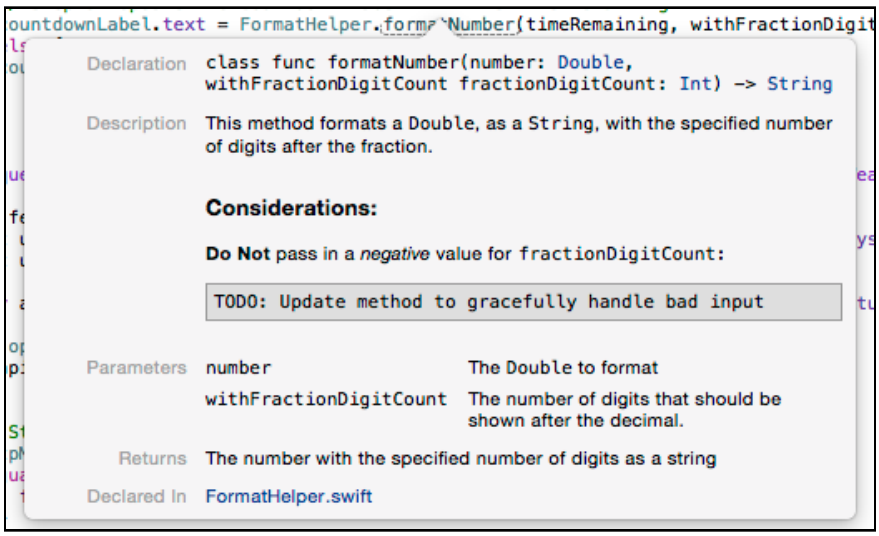
There is actually another small issue. Note that the `relativePath:` parameter takes an optional, `String?`, whereas the comment says `relativePath` is a required parameter. Can you fix this yourself? Go ahead and try it.

Okay, here's the answer. It's upside down so you can't cheat quite so easily:

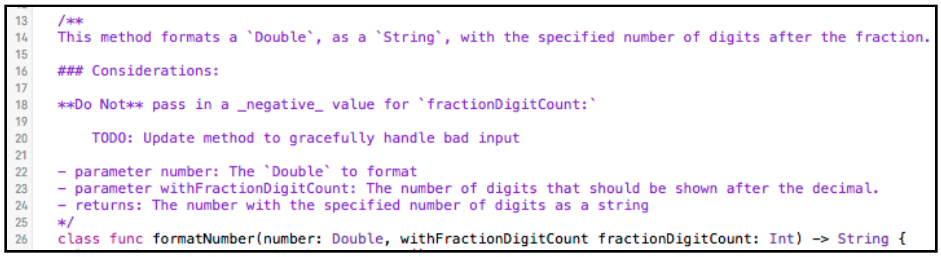
```
- (void)fetchJSONAtPath:(NSString * _Nonnull)relativePath
```

New documentation features

Open **WeatherViewController.swift** and look at **updateCountdownLabel()**. Note the call to `FormatHelper.formatNumber(_:withFractionDigitCount)` right under the **Step 8** comment. **Option-click** on `formatNumber`, and you'll see the documentation appear in rich formatting.



Now **Command-click** on `formatNumber` to jump to the implementation of the method. It's just using basic markdown syntax!



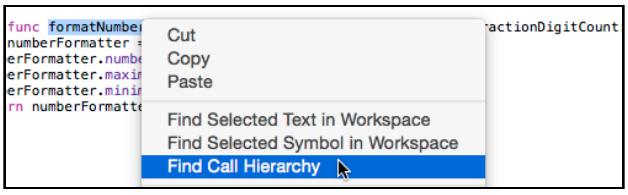
```
/**
This method formats a `Double`, as a `String`, with the specified number of digits after the fraction.

### Considerations:

**Do Not** pass in a _negative_ value for `fractionDigitCount:`

    TODO: Update method to gracefully handle bad input

- parameter number: The `Double` to format
- parameter withFractionDigitCount: The number of digits that should be shown after the decimal.
- returns: The number with the specified number of digits as a string
*/
class func formatNumber(number: Double, withFractionDigitCount fractionDigitCount: Int) -> String {
```

If you want to find out more, NSHipster has a great guide to Swift Documentation, which covers all of the possibilities: bit.ly/1Ltcz0B

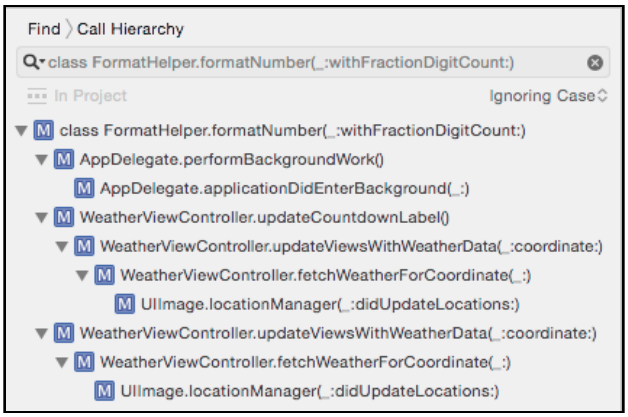
Think of all the beautiful documentation you'll now be able to produce!

Find call hierarchy

Seeing documentation is great, but think about how often you want to see all of the places that use a particular method. **Right-click** or **Control-click** on `formatNumber`, and then click on **Find Call Hierarchy**.



Boom! You'll see all the places that call `formatNumber`. How cool is that? *And* you can also drill down to view the complete call hierarchy:



From here you can quickly jump to any level in the call hierarchy with a click of your trackpad.

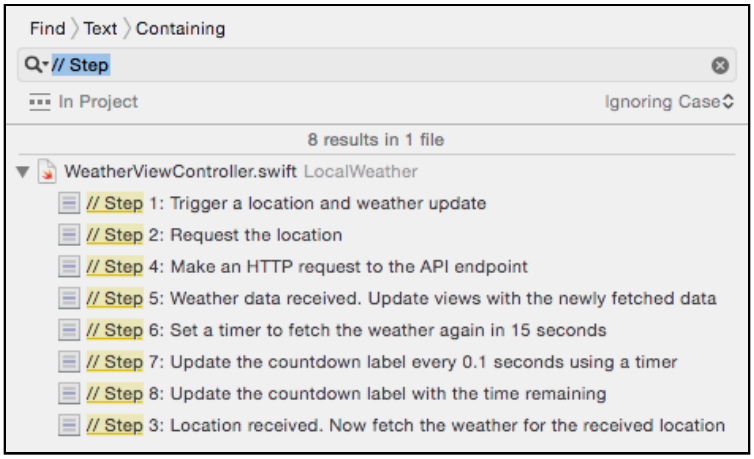
Decreasing energy impact

It's time to get back to fixing those draining energy issues! Before you do, here's a brief summary of the app flow:

- First, it requests the user's current location.
- Once it receives the location, it makes an HTTP request to fetch the weather for that location.
- Once weather data is received, it updates views and schedules and another request for 15 seconds later.

- Every 0.1 seconds, `countdownLabel` refreshes to show the time remaining until the next network request.

That was a quick, high-level overview. To review the app flow in a bit more detail, press **Command-Shift-F** and search for **// Step**:



Do you recall the interesting behavior you saw when there was 10–11 seconds of network activity followed by three to five seconds of non-activity? It happens because any time a network request is made, the network radios stay on for about 10 seconds, even after the network request completes.

It's a recipe for very high overhead with every single network request. While you may have thought you were just using the network radio for about one second every 15 seconds — that's seven percent — you were *actually* using it for 11 seconds out of 15 seconds, or a whopping 73 percent.

This is the first thing you'll fix.

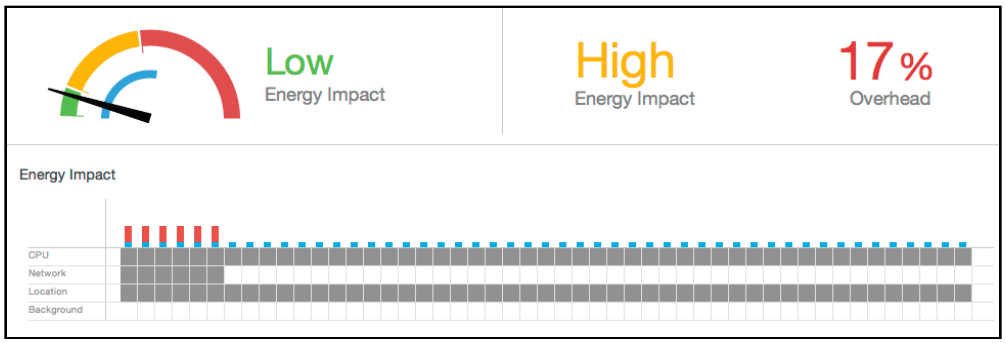
Reducing network energy impact

Weather changes quickly, but not 15 seconds quickly – unless you're out at sea. Hence, there's not a strong use case for a weather app that updates every 15 seconds. A better plan is to make it update whenever the user launches the app.

In the Find results for **// Step** click on **Step 6**, which is in **WeatherViewController.swift**. Comment out the line directly below **Step 6**, which initializes the `networkFetchTimer`:

```
// Step 6: Set a timer to fetch the weather again in 15 seconds
// networkFetchTimer = NSTimer
//  .scheduledTimerWithTimeInterval(15, ...
```

Build and run, switch to the **Debug navigator** and click on the **Energy Impact** row to see the updated energy gauge:



Much better! Now the app will only make a single network request when it starts, saving the user's battery for more important things, like watching videos of pug puppies.

Reducing CPU energy impact

In the previous graph, you see that the CPU is still working even though the app is essentially twiddling its thumbs.

`countdownUpdateTimer`, which updates the `countdownLabel`, is the culprit. It continually sets the remaining time to 0 since there is no longer an active `networkFetchTimer`.

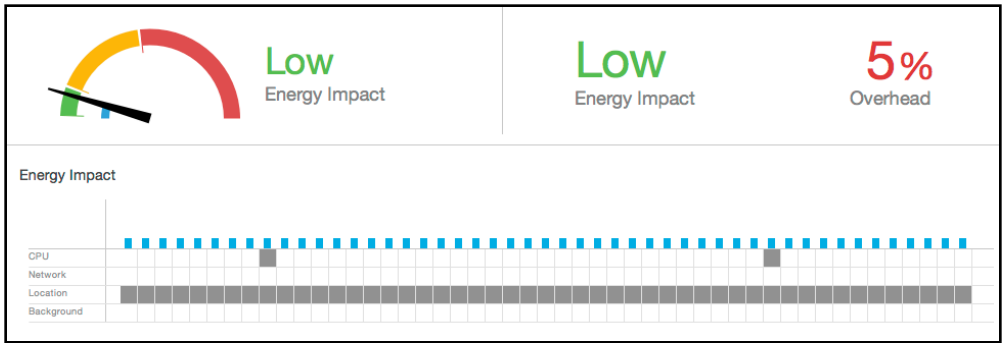
Switch back to the **Find navigator**, and click on **Step 7** to take you to the relevant line in **WeatherViewController.swift**. Comment out the line directly below **Step 7** which initializes `countdownUpdateTimer`:

```
// Step 7: Update the countdown label every 0.1 seconds using a timer
// countdownUpdateTimer = NSTimer.scheduledTimerWithTimeInterval(0.1 ...
```

For good measure, also comment out the line that unhides the `countdownLabelStackView` just above the `UIView` animation block, since you no longer need to see the countdown label:

```
// countdownLabelStackView.hidden = false
```

Build and run, and go back to the **energy gauge**. You may have to wait up to 30 seconds for things to settle down, but this looks much better!



Reducing location energy impact

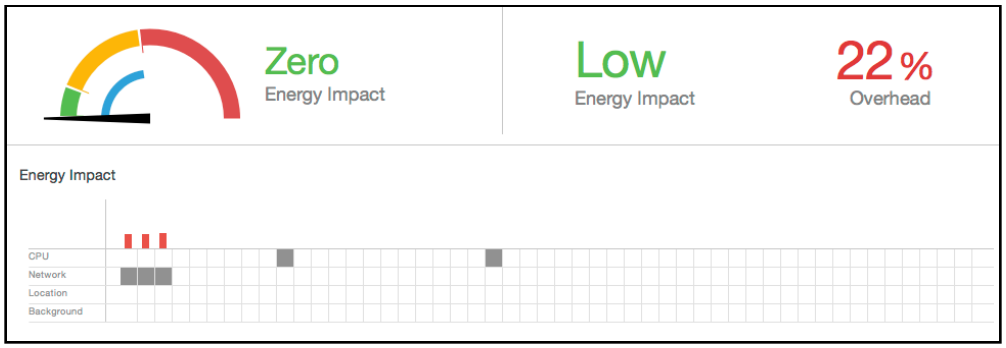
The next thing to do is reduce the location energy impact by turning off GPS when it's not needed. Once the initial location is received, you no longer require location updates, so you should add a call to `locationManager.stopUpdatingLocation()`.

However, in iOS 9 if you only need a single location update, you can just call `requestLocation()` instead of `startUpdatingLocation()`. This new method automatically stops location updates once it's reported the user's location.

In **WeatherViewController.swift**, find **requestLocationAndFetchWeather()** and replace `.startUpdatingLocation()` with `.requestLocation()`:

```
// Step 2: Request the location
locationManager.requestLocation()
```

Build and run, and check the **energy gauge** again. You'll see that after the initial burst of activity, there's little to no activity at all. Nice work!



Reducing background energy impact

Recall that when you pressed the home button, the background graph started registering activity, but didn't stop even though the console log indicated that all background work was complete in about 11 seconds.

Open **AppDelegate.swift** and take a look at **performBackgroundWork()**.

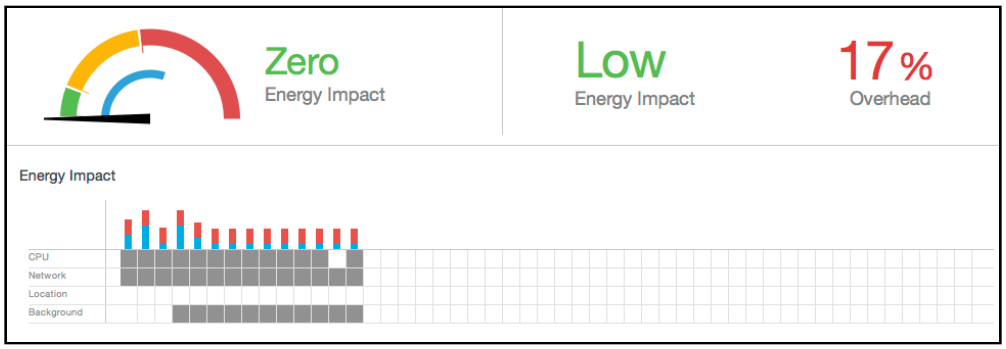
> **Note**: Technically, the method isn't doing any "real" background work, it's just simulating 10 seconds of activity for the purposes of helping you learn how this all works!

The one rule of calling `beginBackgroundTaskWithExpirationHandler(_:)` is that once you're done with any work, you should call `endBackgroundTask(_:)` to let the system know that you no longer need further background execution time.

Add a call to `endBackgroundTask(_:)` at the very end of `performBackgroundWork()` right after the `print` call:

```
print("Background work completed in: \(formattedElapsedTime) " +
  "sec")
UIApplication.sharedApplication().endBackgroundTask(
  backgroundTaskIdentifier)
```

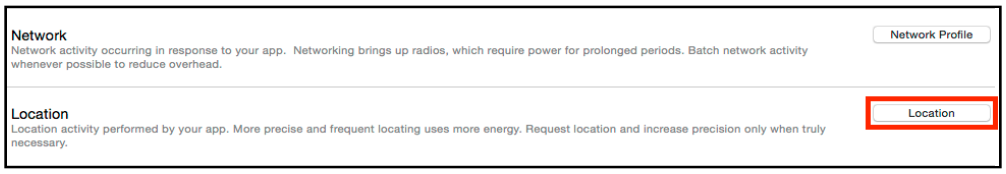
Build and run, go to the **energy gauge** and press the **Home button** to place the app in the background. You'll see that background activity now stops after 10 seconds:



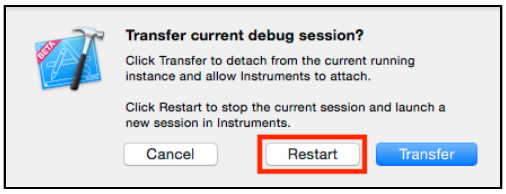
Fantastic! You've fixed all of the energy impact issues using the new energy gauge. Now to take it one step further with the new Core Location instrument in Xcode.

Core Location instrument

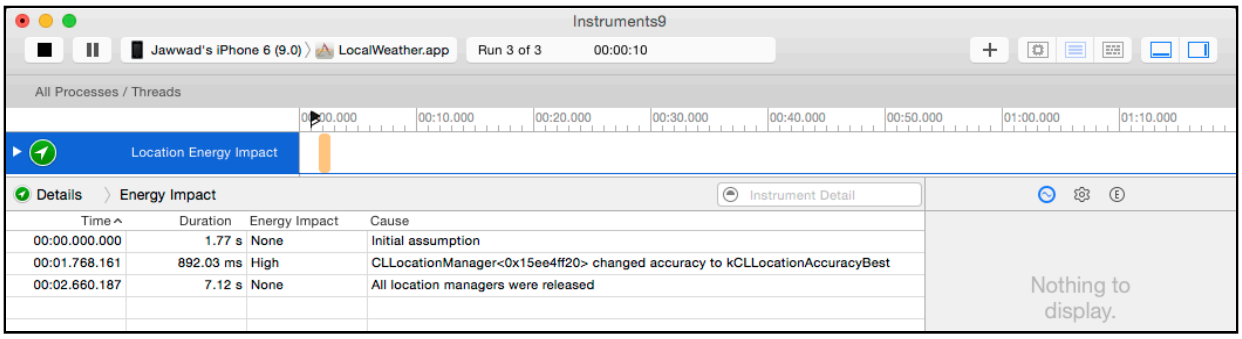
If you want to dig deeper into how the user's location is being accessed, then iOS 9's new Core Location instrument is for you. To use it, build and run the app, go to the **energy gauge** and click on the **Location** button below the gauge:



Click **Restart** on the *Transfer current debug session* prompt that appears:



Instruments will open up and run the Core Location instrument, profiling LocalWeather. The graph will show whenever the app uses Core Location, and the table, shown below, shows extra information about each usage.



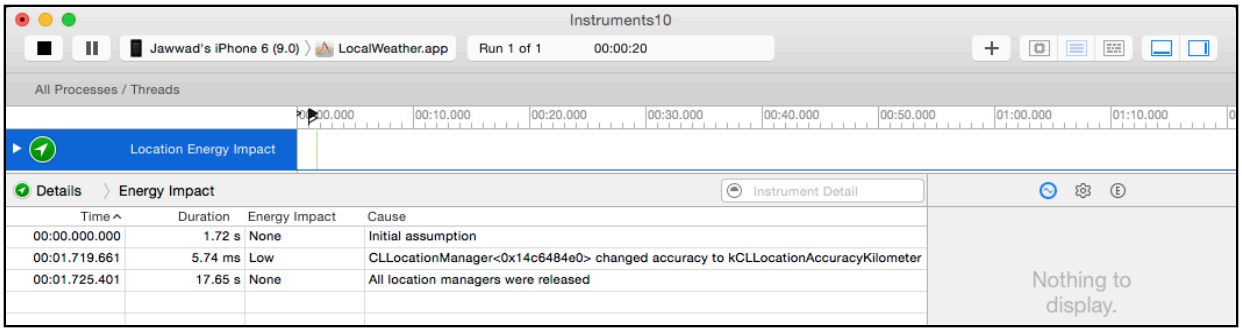
Notice the second row that says "`CLLocationManager` changed accuracy to `kCLLocationAccuracyBest`". The Energy Impact column shows High and the duration shows 892.03 ms.

The app currently uses the default accuracy of `kCLLocationAccuracyBest`. As you can see, it has a high energy impact. A weather app doesn't need to be so precise, so dialing it back is an easy way to improve the app's energy usage.

Click the **stop** button in Instruments, and head back over to Xcode. In **WeatherViewController.swift**, find **viewDidLoad()** and add the following line just before the `locationManager`'s delegate is set:

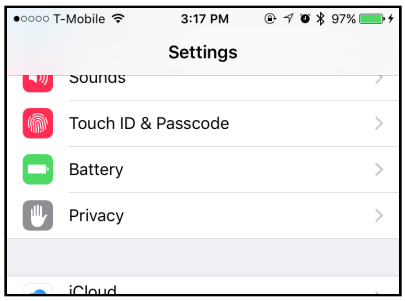
```
locationManager.desiredAccuracy = kCLLocationAccuracyKilometer
```

Build and run, go to the **energy gauge** and click on the **Location** button again. Click **Restart** on the prompt to get back to the Core Location instrument:



The table now says that `CLLocationManager` changed accuracy to `kCLLocationAccuracyKilometer`, which has a low energy impact. It's also much faster, at only *5.74 ms*!

You now have the tools you need to be a good battery citizen on iOS and avoid ending up in the battery hog hall-of-shame. In the iOS 8 settings app, *Battery Usage* was buried 3 levels deep. You had to get to it via Settings \ General \ Usage \ Battery Usage. In iOS 9, it's now on the top-level Settings screen right above Privacy.



It's now easier than ever for a user to keep an eye on the relative battery usage of their apps. And with the new energy impact gauge and Core Location instrument, Apple has made it easier than ever for developers to diagnose and fix energy issues as well.

You're now finished with LocalWeather, so feel free to close the project in Xcode.

Playground improvements

Playgrounds has seen many new features since its debut in Xcode 6 last year. Support for rich authoring, auxiliary source files and inline results were introduced in Xcode 6.3, just 2 months before WWDC 2015. These features are significant, and even though they were initially released in Xcode 6.3, Apple has also chosen to make a note of them in its New Features in Xcode 7 (apple.co/1JSDRMa) document.

Xcode 7 improves upon these features and adds others, such as support for multiple pages and the ability to pause code execution. Playgrounds were already an amazing education tool, and these new features take them to the next level.

Rich playground authoring

In Xcode, choose **File\New\Playground...**, and name it **Xcode7.playground**. Click **Next**, choose a location, and then click **Create**. In the playground, replace the existing "Hello, playground" line with the following:

```
//: # Level 1 Heading

//: ## Level 2 Heading

//: ### Level 3 Heading
```

Reveal the **File inspector** by pressing **Command-Option-1**, or by clicking on the **Utilities** button followed by the **File Inspector** button:

Under **Playground Settings**, place a checkmark in **Render Documentation**:

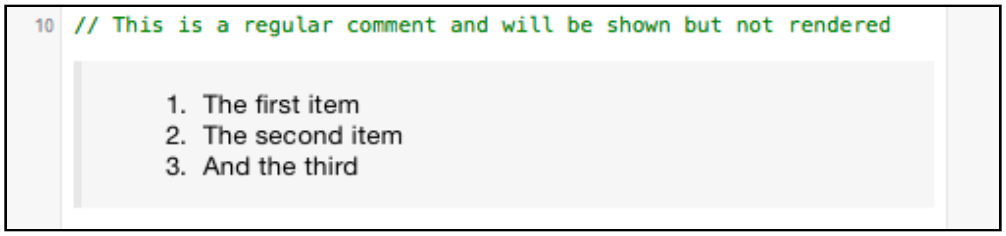
You'll now see the rendered markup:

Unlike Xcode's documentation syntax, which uses an extra comment character, like **///** or **/****, for playgrounds you use a colon instead. So for single-line comments, the syntax is **//:**, and for multiline comments the syntax is **/*:** and ***/**.

Consecutive lines using **//:** are rendered in the same block. Uncheck **Render Documentation** and add the following:

```
// This is a regular comment and will be shown but not rendered
//: 1. The first item
//: 2. The second item
//: 3. And the third
```

Check **Render Documentation** again to see the output:

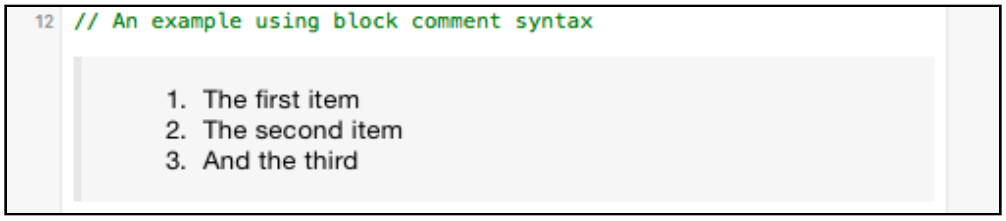


For multiple lines, you can also use the block delimiter **/*:** and ***/** and it will render as a block even, if you have a blank line between any of the lines. Furthermore, anything that you add directly after **/*:** won't render at all.

Uncheck **Render Documentation** and add the following:

```
// An example using block comment syntax
/*: This is a comment and will not be shown at all
1. The first item
2. The second item

3. And the third
*/
```

Check **Render Documentation** once more:

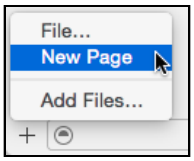


Note that the text after **/*:** does not show at all.

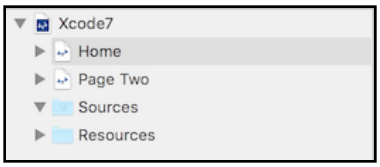
This was just a brief introduction to get you started. For a full reference on syntax, see: Playground Markup Format (apple.co/1IG2eZ9).

Playground pages

You'll now add a second page to the playground. Reveal the **project navigator** by pressing **Command-1**, click the **+** button in the very bottom left, then click on **New Page**:



Name the newly added page **Page Two**, and then rename the original page by **clicking** it once to select it and then again to edit its title. Change its name from **Untitled Page** to **Home**.



Click on **Page Two** and you'll see it already has **Previous** and **Next** links. In the **File inspector**, uncheck **Render Documentation** to see the plain text format of the links:

```
//: [Previous](@previous)

import Foundation

var str = "Hello, playground"

//: [Next](@next)
```

@next and **@previous** are special tokens to get you to the next and previous pages. To add a link to a specific page, you can just use the page name. If the name has a space, replace it with **%20**, as you would in a URL.

Add the following line anywhere in **Page Two**:

```
//: [Jump to Home](Home)
```

And the following line anywhere in **Home**:

```
//: [Jump to Page Two](Page%20Two)
```

Now check **Render Documentation** and try clicking on the links. You should be able to jump between both pages.

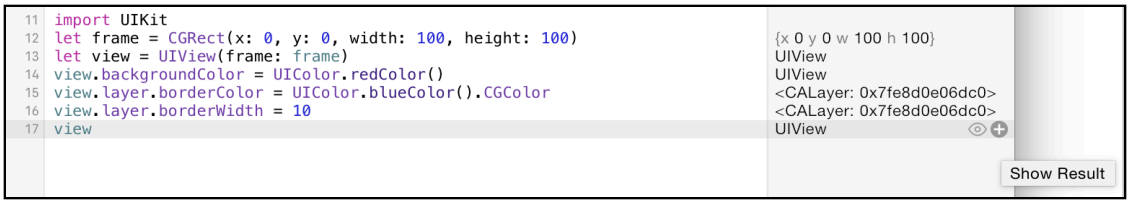
Inline results

Inline results let you see a result directly in the playground itself.

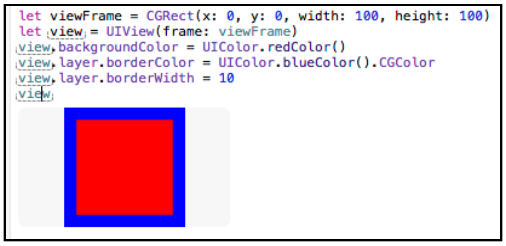
Uncheck **Render Documentation** again, and add the following code to the end of **Page Two**:

```
import UIKit
let frame = CGRect(x: 0, y: 0, width: 100, height: 100)
let view = UIView(frame: frame)
view.backgroundColor = UIColor.redColor()
view.layer.borderColor = UIColor.blueColor().CGColor
view.layer.borderWidth = 10
view
```

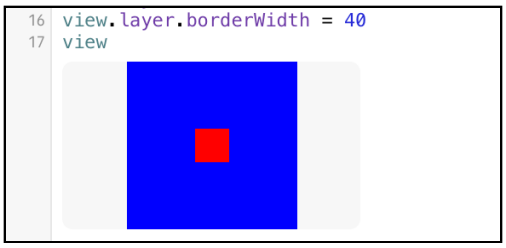
Hover over the `UIView` entry for the last line in the playground **sidebar**, and click on the **Show Result** button that appears:



You'll see the view appear inline right under the `view` variable!

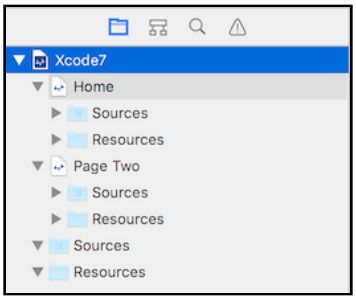


Change the view's layer's `borderWidth` to `40` to see the view update:



Sources and resources

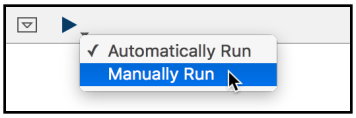
You can now add auxiliary source files and resources to the **Sources** and **Resources** folders in a playground. Moving supporting code to the sources folder allows you to keep the focus on relevant content in the playground page and also speeds up things since the auxiliary files don't need to be recompiled if they don't change.



Also, note that each playground *page* has its own sources and resources folders. If there is an image of the same name in a page's resources folder and also in the top-level resources folder, the image in the page's resources folder will take precedence.

Manually run playgrounds

You can also now choose to *manually* run code in a playground instead of it running automatically as you type; press and hold down the **play** button until a menu appears that lets you select **Manually Run**:



Other improvements

There are many other features and improvements in Xcode 7 – unfortunately, too many to cover in detail in this chapter!

There are many improvements to storyboards and Interface Builder:

- After Control-dragging from one view to another to add a constraint, if you press

Option you'll now also see the *constants* for the constraints that will be added.

- You can now set the layout margins of a view, or the identifier of a constraint in a storyboard.The constraints in the document outline in a storyboard now appear in a much more readable way.

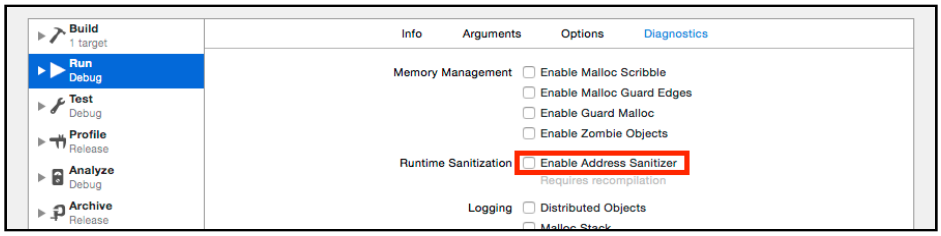
There are also probably some changes that you might not even notice until somebody points them out to you. Like how if you've already implemented a delegate method, Xcode will no longer suggest it to you in the autocomplete menu. And how the Snapshots feature has been removed entirely.

Before this chapter comes to an end, there are two other features worth mentioning briefly...

Address sanitizer

Xcode includes a new tool that will help catch memory corruption errors that may occur when using Objective-C or C. When enabled, Xcode will build your app with additional instrumentation to catch memory errors in the act.

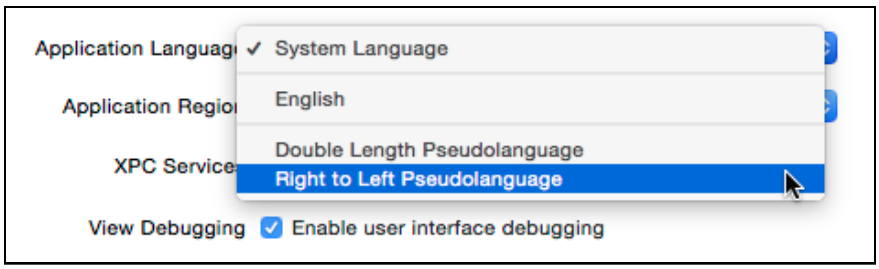
You can enable address sanitizer by going to **Product \ Scheme \ Edit Scheme** and placing a checkmark next to **Enable Address Sanitizer** under **Diagnostics**:



Right-to-left support

iOS 9 contains significant updates for the support of right-to-left languages such as Arabic and Hebrew. For these, the complete view hierarchy will be flipped, and navigation will occur in the opposite direction. If you've been using Auto Layout (and you really should be!), this should mostly "just work".

There is also a new option to test your view hierarchy in this flipped state without having to change your primary language. Edit your scheme and in the **Options** view, under **Application Language**, there is a new **Right to Left Pseudolanguage** option that you can select:



Where to go from here?

Wow. You've learned a *lot* about Xcode in this chapter — from its new documentation and code browsing features to the new energy impact gauge and Core Location instrument that will help make your apps more energy efficient.

You also learned about the various improvements to playgrounds as well as a few other miscellaneous things along the way.

Here are a few links to relevant resources that you might want to check out:

- Playground Markup Format for Comments apple.co/1IG2eZ9
- Energy Efficiency Guide for iOS Apps apple.co/1OEIHMf

And here are a few related WWDC 2015 videos to keep you busy:

- What's New in Xcode apple.co/1M0Fx8e
- Authoring Rich Playgrounds apple.co/1TieQQE
- Debugging Energy Issues apple.co/1gYPZAo
- Achieving All-day Battery Life apple.co/1P2jXxC

Conclusion

We all hope that you've reached the end of this book having had a great deal of fun along the way. If you dipped in and out of the chapters, to cover the topics most relevant to your current projects, then you'll now b e equipped to improve your users' experience with all the new technologies. If you read the book cover-to-cover, well done — you're a coding ninja, and are ready to take on anything that iOS 9 can throw at you!

You now have a lot of real-world hands-on experience with the new iOS 9 technologies and APIs. You've discovered how to support multitasking, 3DTouch and how you can refactor your layouts using stack views to make Auto Layout far more approachable. Your path to creating great iOS apps should now be easier and offer more possibilities — we're looking forward to the innovative ways you use iOS 9.

If you have any questions or comments, please stop by our forums at raywenderlich.com/forums.

Thanks once again for purchasing this book. Your continued support is what makes the tutorials, books, videos and other things we do at raywenderlich.com possible — we all truly appreciate it!

Go forth and use your newfound iOS knowledge for good,

– Jawwad, Soheil, Caroline, Evan, Aaron, James, Vincent, Pietro, Derek, Chris W., Julien, Richard, Wendy, Chris B. and Sam