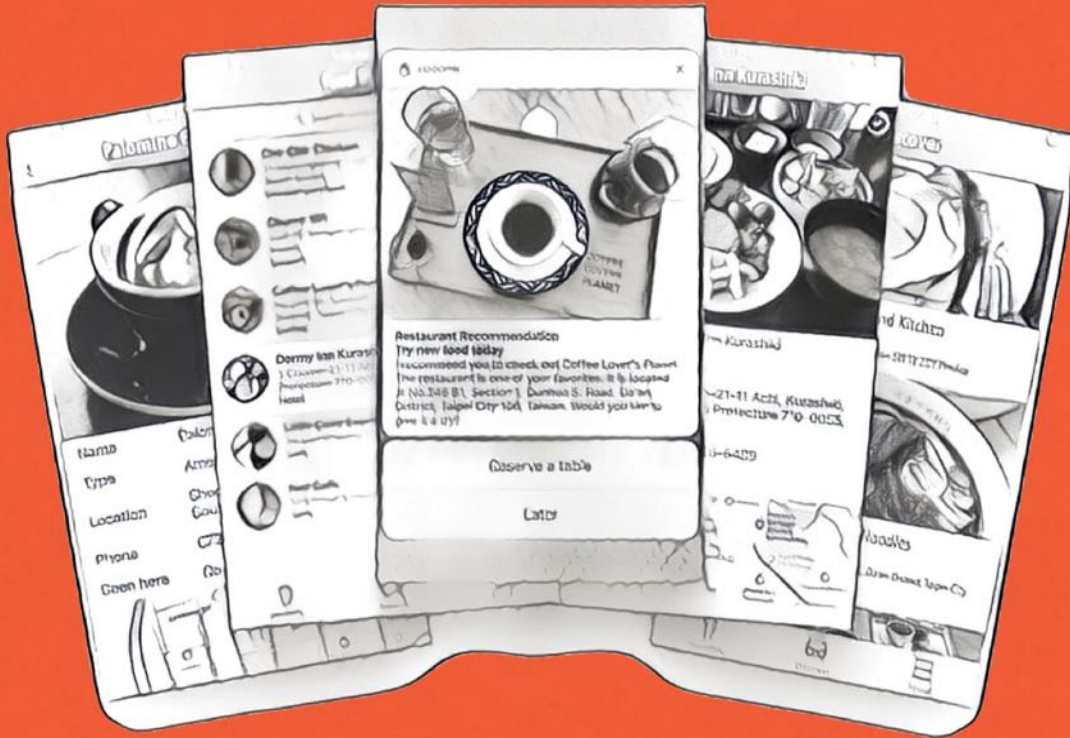




Fully supports Swift 3, Xcode 8 and iOS 10

A Beginner's Guide to Build Your First App from Scratch



Beginning iOS Programming with Swift

by Simon Ng

APPCODA

Table of Contents

1. Preface
2. Getting Started with Xcode 8 Development
3. Swift Playgrounds
4. Build Your First App
5. Hello World App Explained
6. Introduction to Auto Layout
7. Designing UI Using Stack Views
8. Introduction to Prototyping
9. Creating a Simple Table-based App
10. Customize Table Views Using Prototype Cell
11. Interacting with Table View and Using UIAlertController
12. Table Row Deletion, Custom Action Button and MVC
13. Introduction to Navigation Controller and Segue
14. Introduction to Object-Oriented Programming
15. Detail View Enhancement and Navigation Bar Customization
16. Self Sizing Cells and Dynamic Type
17. Basic Animations, Visual Effects and Unwind Segues
18. Working with Maps
19. Introduction to Static Table Views, UIImagePickerController and NSLayoutConstraint
20. Working with Core Data
21. Search Bar and UISearchController
22. Building Walkthrough Screens with UIPageViewController
23. Exploring Tab Bars and Storyboard References
24. Getting Started with WKWebView and SFSafariViewController
25. Exploring CloudKit
26. Localizing Your App to Reach More Users
27. Deploying and Testing Your App on a Real iOS Device
28. Beta Testing with TestFlight
29. Submitting Your App to App Store
30. Adopting 3D Touch

31. [Developing User Notifications in iOS 10](#)
32. [Appendix - Swift Basics](#)

Copyright ©2016 by AppCoda Limited

All right reserved. No part of this book may be used or reproduced, stored or transmitted in any manner whatsoever without written permission from the publisher.

Published by AppCoda Limited

Preface

In June 2014, Apple brought us Swift, a brand new programming language for iOS, macOS, watchOS and tvOS. Fast forward to today, the company already released version 3.0 of the programming language with more features. With over two years of development since its first release, I would not consider Swift as a brand new programming language. Swift is now solidified, more mature and ready for any application development on Apple platforms. If you're planning to create your next iOS app or begin to learn iOS development, Swift is for sure the programming language to check out.

In this book, I will give you an introduction to Swift 3, discuss some of the new features in iOS 10, and most importantly, show you how to build a real world app from scratch in Swift.

As an absolute beginner, you may question if you could learn Swift programming and build a real iOS app.

I have been programming in Swift since it was first announced. Swift is more approachable and it is a lot easier for newbie to learn than Objective-C. Not everyone can become a great developer, but I believe everyone can learn programming and develop an app in Swift. All you need is hard work, determination, and the willingness to take actions.

I launched AppCoda about four years ago and started to publish iOS programming tutorials on weekly basis. Since then, I have published several books on iOS app development. At first, I thought people, who want to learn app development, are those with programming experience and technical background. What's interesting is that people from different backgrounds are passionate to build their own apps. I have a reader from France, who is a surgeon by profession, started from zero programming experience to launching his first app, which allows anyone to share and advertise event information for free. Another reader is a pilot by profession. He began to learn iOS programming a couple years ago and is now building iPhone apps for his own use and other pilots. Boozy is an app for finding Happy Hours, Daily Deals and Brunches. It was built by a law school dropout. The creator of the app could not find a good place for a drink in DC area. So she decided to make an app to meet a real need. Similarly she did not know coding when she came up with the idea. She just got started and learned

along the way.

From time to time, I got emails from people who want to create an app. The emails usually mention something like this: "I have an app idea. Where do I begin? But I have no programming skills. Can I learn from scratch to make one?"

What I learned from these truly inspiring stories is that you don't need to have a degree in Computer Science or Engineering in order to build an app. These readers have one thing in common. They are committed to take actions. They all put their hard work in to make things happen. This is what you need.

So you've got an idea to build an app? I believe you can make one on your own. Remember there is nothing to deter you from learning and get things done if you're really passionate about it. Let me borrow one of my favorite quotes from *Last Lecture* to conclude:

Brick walls are there for a reason: they let us prove how badly we want things.

- Randy Pausch

Lastly, thanks for picking up the book. I hope you will enjoy reading it and launch your first iOS app on App Store. If you'd like to share the story of your first app, drop me an email at simonng@appcoda.com. I would love to hear from you.

Simon Ng Founder of AppCoda

What You Will Learn in This Book

I know many readers have an app idea but don't know where to begin. Hence, this new book is written with this in mind. It covers the whole aspect of Swift programming and you will learn how to build a real world app from scratch. You'll first learn the basics of Swift, then prototype an app, and later add some features to it in each chapter. After going through the whole book, you'll have a real app. During the process, you will learn how to exhibit data in table view, customize the look & feel of a cell, design UI using Stack Views, create animations, work on maps, build an adaptive UI, save data in local database, upload data to iCloud, use TestFlight to arrange beta test, etc.

This new book features a lot of hands-on exercises and projects. You will get the opportunities to write code, fix bugs and test your app. Although it involves a lot of work, it will be a rewarding experience. I believe it will allow you to master Swift 3, Xcode 8, and iOS 10 programming. Most importantly, you will be able to develop an app and release it on App Store.

Audience

This book is written for beginners without any prior programming experience and those who want to learn Swift programming. Whether you are a programmer who wants to learn a new programming language or a designer who wants to turn your design into an iOS app or an entrepreneur who wants to learn to code, this book is written for you.

I just assume you are comfortable using Mac OS X and iOS as a user.

Learning Tips

I have been asked by some of my blog readers about the best way to start learning iOS programming. These are some of the learning tips I shared on my blog and I expect these tips will be useful to you too.

1. **As always, get your hands dirty.** I emphasized this technique at the very beginning when I wrote my first book. You can't learn programming just by reading. You need to take action and write some code. Fire up Xcode, play around with Swift and follow the first few chapters to build your first app. Try your best to understand how the Hello World app works before moving on to the next chapter. If you are able to manage the basics, it will be much easier for you to understand the rest of the materials.
2. **Bugs are your friends when learning programming.** From time to time, you'll hit bugs or errors. You probably want to email me and say the sample app doesn't work. How can I get the bug fixed? Questions are always welcome and I love to help you learn programming. However, I encourage you to attempt to figure out the solutions by yourself first. You may go over the code again and again. Or search the web (stackoverflow.com in particular) for solutions. Just try your best to solve the problem before asking. Like every programmer, I hate bugs especially when facing a project deadline. However it's always

the bugs that help me improve my programming skills.

If you do not have any programming experience, one useful hint for you is that Swift is a case-sensitive language. That means a variable named 'message' is different from that named 'Message'. This is one of the most common errors and frequently asked questions.

3. **"The best way to learn is to teach" is an old saying.** It still works in the modern world, however. You don't need to be an expert to teach. I'm not talking about giving a lecture in a university or teaching a bunch of students in a formal class. Teaching does not always happen that way. It can be as simple as sharing your knowledge with a colleague or a classmate sitting next to you. When you learn something new, try to explain the materials to someone else. For example, after building the HelloWorld app, teach your close friend how it works and how he/she can create an app too. When you become more knowledgeable with the programming language and materials, you may arrange an interest group and share what you've learnt to a larger group of people.

This is one of the most effective ways of learning as I learn so much while publishing tutorials on appcoda.com, as well as, developing my first book.

Sometimes you think you know the materials well. But once you need to explain the concept to someone else and answer questions, you may find that you didn't understand the material thoroughly. And this makes you study the materials even harder. Give this method a shot while you learn iOS programming.

4. **It takes time to become a great programmer.** Be patient. The materials in this book are not magical. You will learn the basics of iOS programming and how to build your own apps. That said, it takes time and lots of practices to become a competent programmer. Don't set your expectations too high for your first app. Just create something simple and fun.
5. **I know money matters.** Some of the people begin learning app development just because of money. There is nothing wrong with that. You may want to build your app business to earn some side income and eventually turn it into a full time business. That's completely understandable. Who doesn't want to live a rich life? However, if money is your primary reason for building apps, you'll be easily discouraged when you hit bugs or errors. And then you give up. Programming is challenging. I find people who successfully

master the language are those who have a strong desire to build apps and are enthusiastic to learn programming. They have an idea in their mind and want to turn it into a real app. Making money is not their number one concern. They know the app can solve their own problems and will be beneficial to others. With such a powerful purpose in mind, they can overcome any obstacles come up. So think again why you want to learn programming.

Chapter 1

Getting Started with Xcode 8

Development



So you want to create your own app? That's great! Creating an app is a fun and rewarding experience. But before we begin to dive into iOS programming, let's go through the tools you need to build an app.

1. Get a Mac

First, you need a Mac. It's the basic requirement for iOS development. To develop an iPhone (or iPad) app, you need to get a Mac with an Intel-based processor running on Mac OS X version 10.10 (or later). If you now own a PC, the cheapest option is to purchase the Mac Mini.

As of this writing, the retail price of the entry model is US\$499. You can hook it up with the monitor of your PC. The basic model of Mac mini comes with a 1.4GHz dual-core Intel Core i5 processor and 4GB memory. It should be good enough to run the iOS development tools smoothly. Of course, if you have a bigger budget, get the higher model or an iMac with better processing power.

2. Register Your Apple ID

You will need an Apple ID to download Xcode, access iOS SDK documentation, and other technical resources. Most importantly, it will allow you to deploy your app to a real iPhone/iPad for testing.

If you have downloaded an app from the App Store, it is quite sure that you already own an Apple ID. In case you haven't created your Apple ID before, you have to get one. Simply go to Apple's website (<https://appleid.apple.com/account>) and follow the procedures for registration.

3. Install Xcode

To start developing iOS apps, Xcode is the only tool you need to download. Xcode is an integrated development environment (IDE) provided by Apple. Xcode provides everything you need to kick start your app development. It already bundles the latest version of the iOS SDK (short for Software Development Kit), a built-in source code editor, graphic user interface (UI) editor, debugging tools and many more. Most importantly, Xcode comes with an iPhone (and iPad) simulator so you can test your app without the real devices.

You have two ways to install Xcode: 1. Download it through Mac App Store. 2. Manually download it from Apple's developer website.

Download Xcode from Mac App Store

To download Xcode, launch the Mac App Store on your Mac. If you're using the latest version of Mac OS, you should be able to open the Mac App Store by clicking the icon in the dock. In case you can't find it, you may need to upgrade your Mac OS.



Figure 1-1. App Store icon in the dock

In the Mac App Store, simply search "Xcode" and click the "Get" button to download it.



Figure 1-2. Download Xcode 8

Once you complete the installation process, you will find the Xcode folder in the Launchpad.

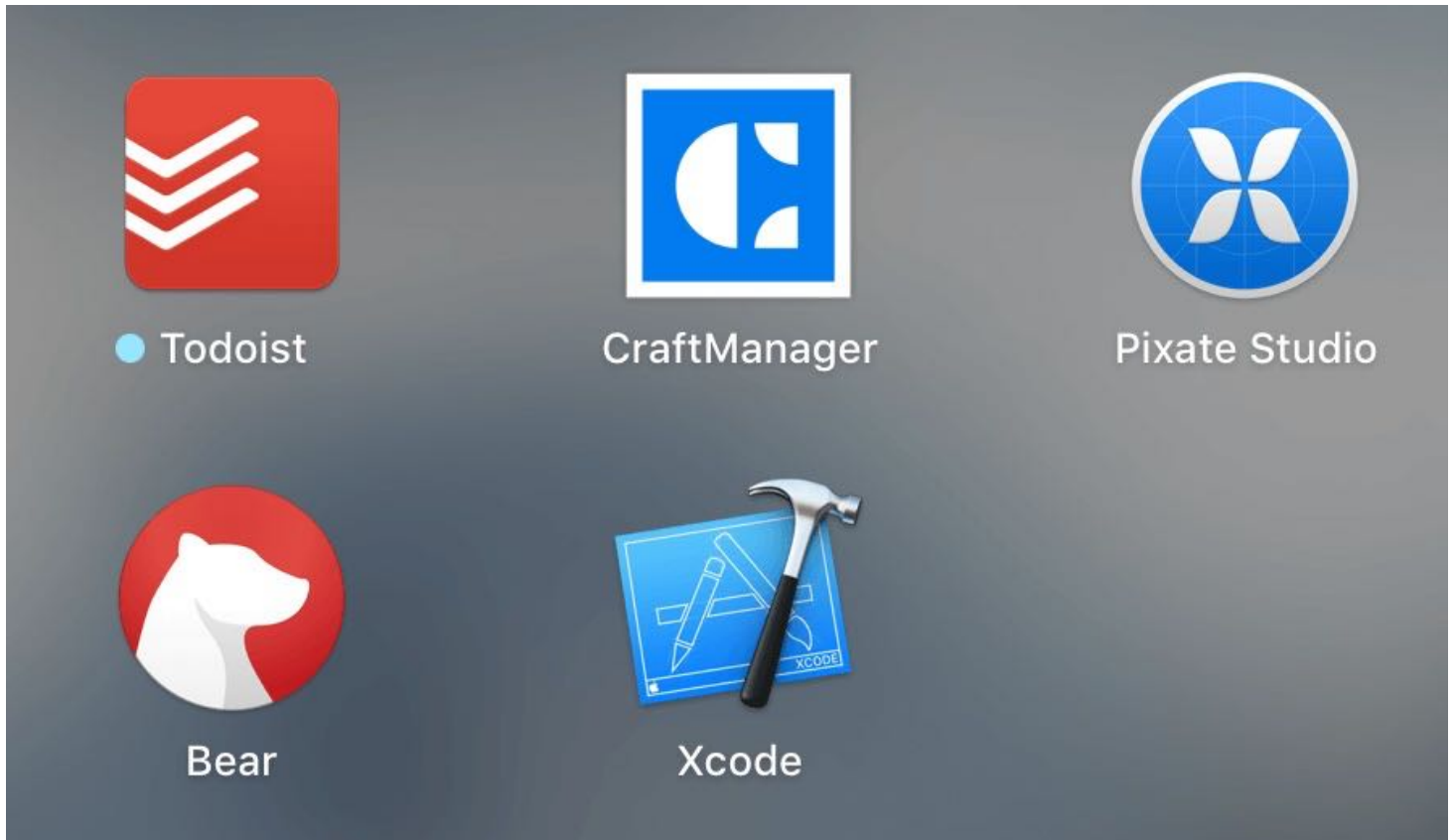


Figure 1-3. Xcode icon in the Launchpad

At the time of this writing, the latest version of Xcode is 8.0. Throughout this book, we will use this version of Xcode to create the demo apps. Even if you have installed Xcode before, I suggest you upgrade to the latest version. This should make it easier for you to follow the tutorials.

Download Xcode 8 from Developer's website

Normally you can download Xcode from the Mac App Store, which is the recommended way for beginners. For any reason you don't want to use Mac App Store, you can download Xcode 8 manually. To get a copy of it, you have to sign into the Apple Developer website (<http://developer.apple.com/register/>). Select Download Tools and then click Download Xcode 8 GM.

Once the file is downloaded, double-click and install it.

4. Enroll in the Apple Developer Program (Optional)

A common question about developing an iOS app is whether you need to enroll in the Apple Developer Program (<https://developer.apple.com/programs/>). The short answer is optional. First, Xcode already includes a built-in iPhone and iPad simulator. You can develop and test out your app on your Mac, without enrolling in the program.

Starting from Xcode 7, Apple has changed its policy regarding permissions required to build and run apps on devices. Before that, the company required you to pay \$99 per year in order to deploy and run your apps on a physical iPhone or iPad. Now, program membership is no longer required. Everyone can test their apps on a real device without enrolling into the Apple Developer Program. Having that said, if you want to try out some advanced features such as in-app purchase and CloudKit, you still need to apply for the program membership. Most importantly, you're not able to submit your app to App Store without paying \$99 annually.

So, should you enroll into the program now? The Apple Developer Program costs US\$99 per year. It's not big money but it's not cheap either. As you're reading this book, you're probably a newcomer and just start exploring iOS development. The book is written for beginners. We will first start with something simple. You are not going to tap into the advanced features yet.

Therefore, even if you do not enroll into the program, you will still be able to build an app and test it on your device. So save your money for now. I will let you know when you need to enroll into the program. At that time, you're encouraged to join the program as you're ready to publish the app to the App Store!

5. What You Need to Learn

Now that you have configured everything you need to start iOS app development, let me answer another common question from beginners before moving on. A lot people have asked about what skills you need to learn in order to develop an iOS app. In brief, it comes down to three things:

- Learn Swift - Swift is now the recommended programming language for writing iOS apps.

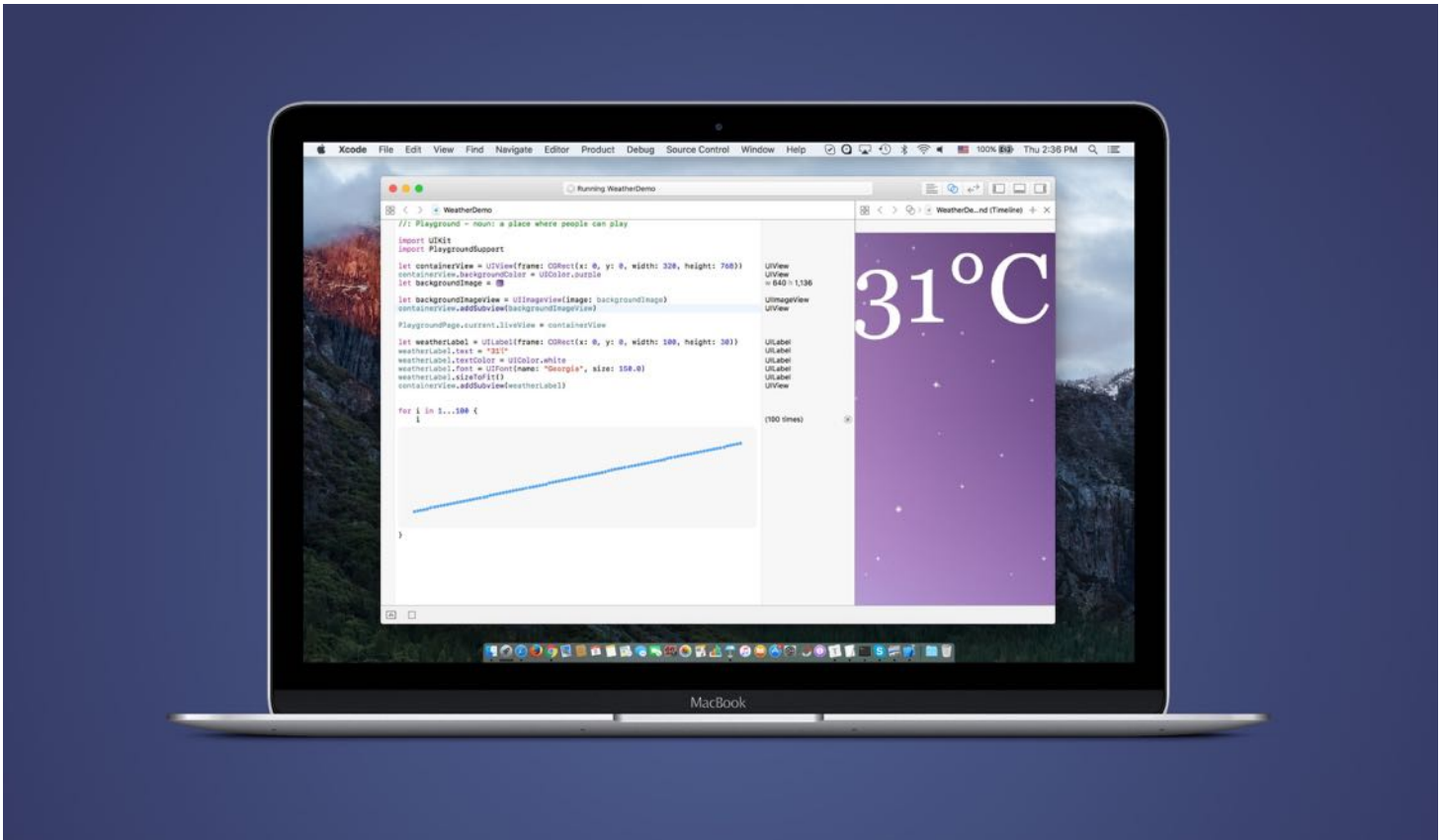
- Learn Xcode - Xcode is the development tool for you to design the app UI, write Swift code, and build your apps.
- Understand the iOS software development kit - Apple provides the software development kit for developers to make our lives simpler. This kit comes with a set of software tools and APIs that empowers you to develop iOS apps. For example, if you want to display a web page in your app, the SDK provides a built-in browser that lets you embed right in your application.

You will have to equip yourself with knowledge on the above three areas. That's a lot of stuff. But no worries. You'll learn the skills as you read through the book.

That's all for the introduction. Take some time to register as an Apple developer and install Xcode 8. When we move onto the next chapter, we will start programming in Swift. So get ready!

Chapter 2

Swift Playgrounds



Just play. Have fun. Enjoy the game.

- Michael Jordan

In the Worldwide Developer Conference 2014, Apple surprised all iOS developers by launching a new programming language called Swift. Swift is advertised as a "fast, modern, safe, interactive" programming language. The language is easier to learn and comes with features to make programming more productive.

Prior to the announcement of Swift, iOS apps were primarily written in Objective-C. The language has been around for more than 20 years and was chosen by Apple as the primary programming language for Mac and iOS development. I've talked to many aspiring iOS developers. A majority of them said Objective-C was hard to learn and its syntax looked weird.

Simply put, the code scares some beginners off from learning iOS programming.

The release of Swift programming language is probably Apple's answer to some of these comments. The syntax is much cleaner and easier to read. I have been programming in Swift since its beta release. It's more than two years for now. I can say you're almost guaranteed to be more productive using Swift. It definitely speeds up the development process. Once you get used to Swift programming, it would be really hard for you to switch back to Objective-C.

It seems to me that Swift will lure more web developers to build apps. If you're a web developer with some programming experience on any scripting languages, you can leverage your existing expertise to gain knowledge on developing iOS apps. It would be fairly easy for you to pick up Swift. Being that said, even if you're a total beginner with no prior programming experience, you'll also find the language friendlier and feel more comfortable to develop apps in Swift.

In June 2015, Apple announced Swift 2, and that the programming language goes open source. This is a huge deal. Since then, developers created some interesting and amazing open source projects using the language. Not only can you use Swift to develop iOS apps, companies like IBM developed web frameworks for you to create web apps in Swift. Probably some days, you can use Swift to develop Android apps. Who knows?!

Earlier this year, Apple introduced Swift 3. The new version of the programming language is integrated into Xcode 8, which is released in Sep, 2016. This is one of the biggest releases since the birth of the language.

With Swift 3, every single API in Swift 3 is new again.

- Olivier Gutknecht, Senior Engineering Manager at Apple Inc.

In brief, there are a lot of changes in Swift 3. APIs are renamed and more features are introduced. All these changes help making the language even better, and enable developer to write more beautiful code. I will not go into the details of the changes here as I think you're pretty new to Swift. But if you have some experience with Swift, and want to know more about the changes, you can check out the appendix. For beginners, just read on. But remember to fire up Xcode and follow me to write some code.

To get a taste of Swift programming language, let's take a look at the following code snippets.

Objective-C

```
const int count = 10;
double price = 23.55;

NSString *firstMessage = @"Swift is awesome. ";
NSString *secondMessage = @"What do you think?";
NSString *message = [NSString stringWithFormat:@"%@@%", firstMessage,
secondMessage];

NSLog(@"%@", message);
```

Swift

```
let count = 10
var price = 23.55

let firstMessage = "Swift is awesome. "
let secondMessage = "What do you think?"
var message = firstMessage + secondMessage

print(message)
```

The first block of code is written in Objective-C, while the second one is written in Swift. Which language do you prefer? I guess you would prefer to programming in Swift, especially if you're frustrated with the Objective-C syntax.

Constants and variables are two basic elements in programming. In the Objective-C world, it's your responsibility to specify the type information when declaring a variable or a constant, be it an integer or a string. For Swift, it introduces a new feature called *Type Inference*. You no longer need to annotate variables/constants with type information. All you need to do is to use `let` to declare a constant and `var` to declare a variable. Swift is intelligent enough to deduce the type by examining the values you provide.

Another difference that you may notice is the omission of the semi-colon. In Objective-C, you have to end each statement with a semicolon. If you forget to do so, you will end up with an error. I know many Objective-C beginners have experienced this error before. Swift should make your developer's life easier.

Swift has added many powerful features to streamline your coding. As you can see from the above example, string manipulation is much simpler. In Objective-C, you have to choose

between `NSString` and `NSMutableString` classes to indicate whether the string can be modified. You do not need to make such choice in Swift. Whenever you assign a string as variable (i.e. `var`), the string can be modified in your code. Concatenating strings is super easy. Just use the `+` operator to combine two strings. Furthermore, Swift allows you to compare strings directly using the `==` operator.

There is no better way to explore coding than actually writing code. Starting from Xcode 6, Apple introduced a new feature called *Playgrounds*. It's an interactive development environment for developers to experiment Swift programming and allows you to see the result of your code in real-time. Xcode 8 makes it even better for beginners to explore Swift. Surprisingly, you can now run Playgrounds on iPad too.

Anyway, we will focus on using Playgrounds in Xcode to learn Swift.

Assuming you've installed Xcode 8 (or up), launch the application (by clicking the Xcode icon in Launchpad). You should see a startup dialog.

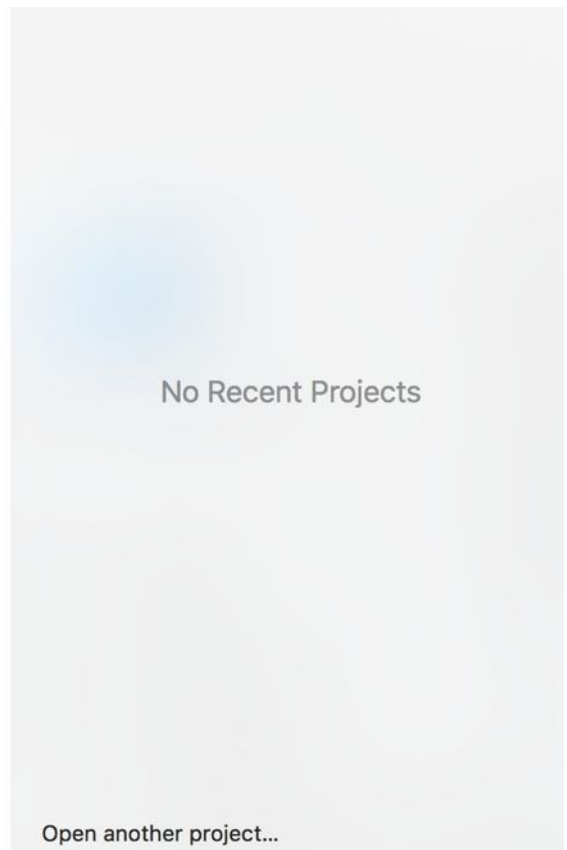


Figure 2-1. The startup dialog

Playground is a special type of Xcode file. You can simply click "Get started with a playground" to get started. You'll then be prompted to fill in a project name and select a platform.

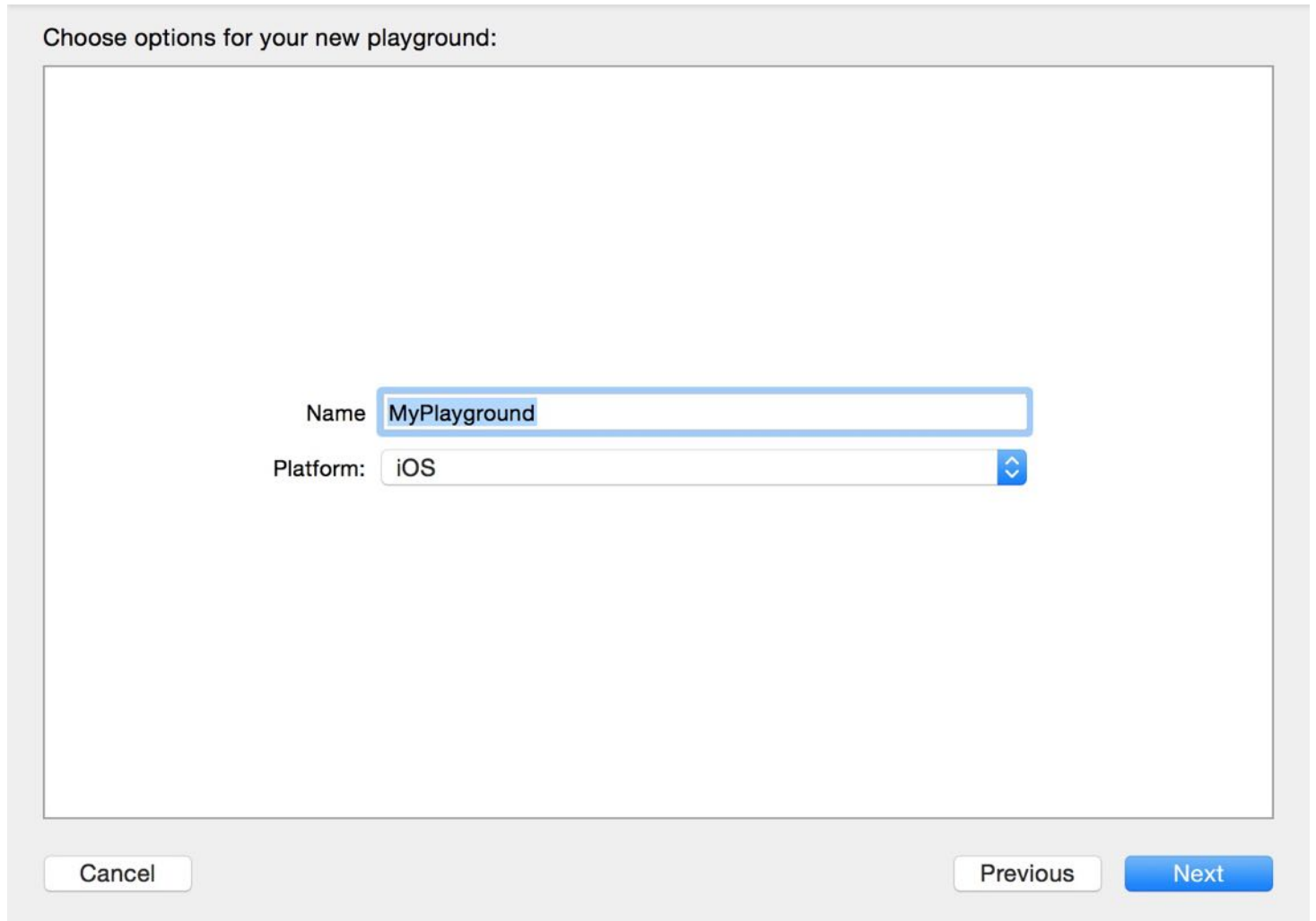
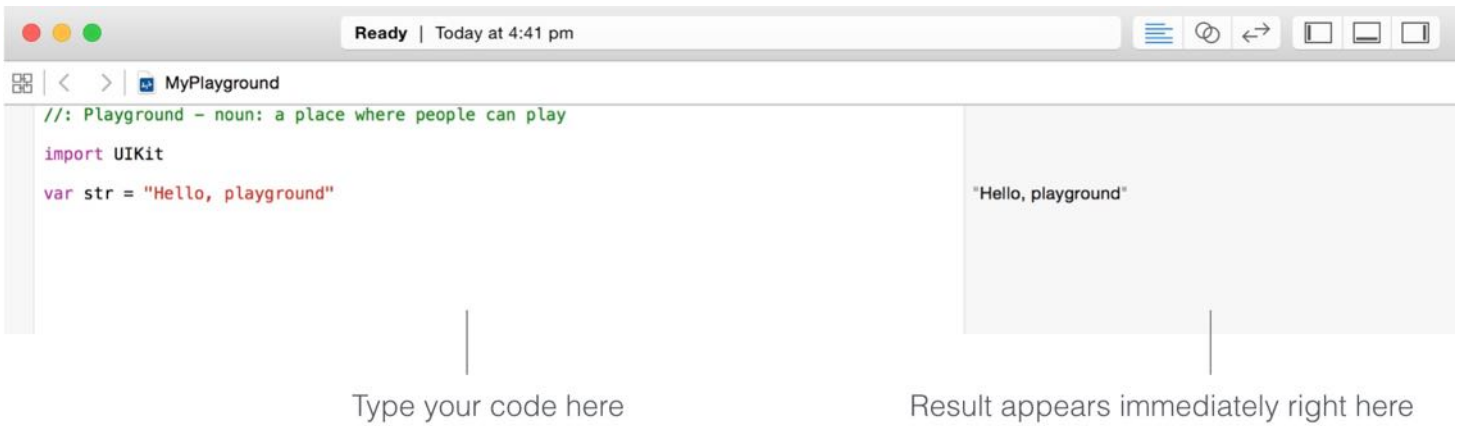


Figure 2-2. Creating a Playground file

Use the default name or choose your own name but remember to select iOS as the platform. Once you confirm to save the file, Xcode opens the Playground interface. Your screen should look like this:



On the left pane of the screen, it is the editor area where you type your code. As you write your code, Playground immediately interprets your code and displays the result on the right pane. By default, Playground includes two lines of code. As you can see, the result of the `str` variable appears immediately on the right pane.

We'll write some code in Playgrounds together. Remember the purpose of this exercise is to let you experience Swift programming and get a better idea of Xcode. I'll try to explain parts of the code as we move along. For now, even if you do not understand any line of the code, that is completely fine. I'm quite sure you'll be confused by some of the terms like *class* and *method*. Forget about their meanings, just relax and play around with Xcode. We'll go over them again in later chapters.

Cool! Let's get started.

First, key in a couple lines of code. Here we declare two more variables:

```
var message1 = "Hello Swift! How can I get started?"
var message2 = "The best way to get started is to stop talking and code."
```

As soon as you insert the above lines of code, you will see the result on the right pane.

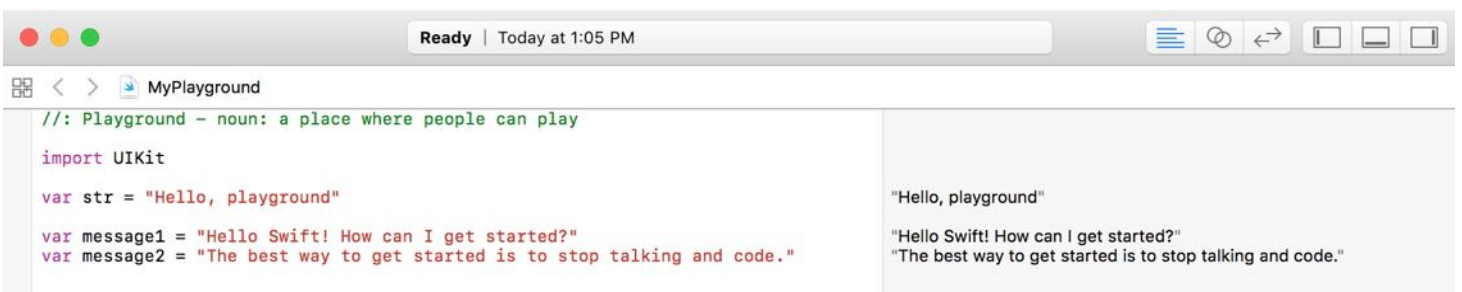
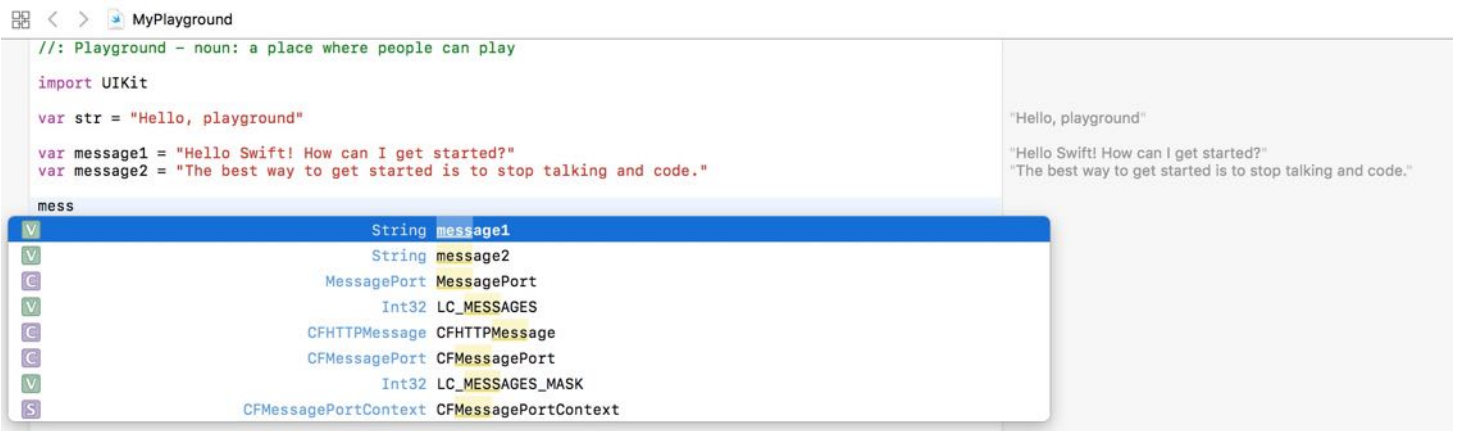


Figure 2-4. Result is shown immediately on the right pane

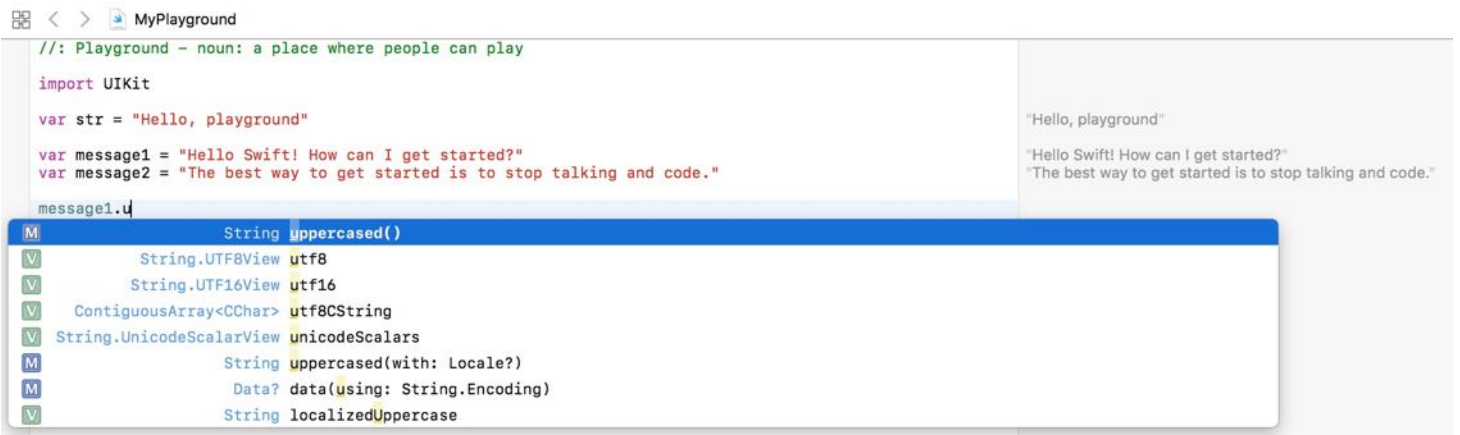
Let's continue to add the following line of code:

```
message1.uppercased()
```

Xcode's editor comes with an auto-complete feature. Auto-complete is a very handy feature to speed up your coding. Once you type `mess`, you'll see an auto-complete window showing some suggestions based on what you have keyed in. All you need to do is to select `message1` and hit enter.



Swift employs the dot syntax for calling methods and accessing the properties of a variable. As you type the dot after `message1`, the auto-complete window pops out again. It suggests a list of methods and properties belonged to the variable. You can continue to type `uppercase()` or select it from the auto-complete window.



Once you complete your typing, you would see the output immediately. When we use `uppercase()` on `message1`, the content of `message1` is converted to upper case automatically.

Continue to type the following line of code:

```
message2.lowercased() + " Okay, I'm working on it "
```

Swift allows you to concatenate two strings with the `+` operator. The line of code converts the content of `message2` into lower case and then concatenated with another string. Interestingly, you can include emoji characters in your code. You may wonder how to type emoji characters in Mac OS. It's easy. Just press control-command-spacebar and an emoji picker will be displayed.

Let's continue to type the following code snippet:

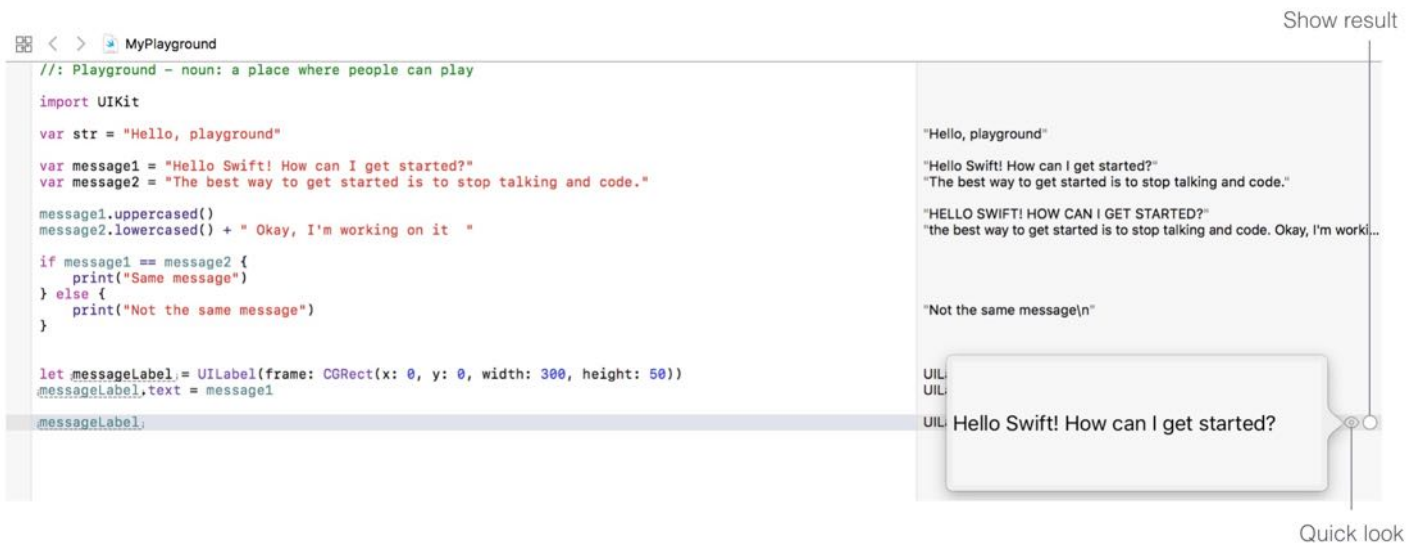
```
if message1 == message2 {  
    print("Same message")  
} else {  
    print("Not the same message")  
}
```

Conditional logic is very common in programming. Sometimes, you want to execute certain part of code only when a certain condition is met. An `if-else` statement is one of the ways in Swift to control program flow. Here we compare the content of `message1` and `message2` variables using the `==` operator. If they're equal, the program prints "Same message". Otherwise, it prints "Not the same message". You should see the following result on your screen.

Let's have some more fun and create a label, which is a common user interface element:

```
let messageLabel = UILabel(frame: CGRect(x: 0, y: 0, width: 300, height: 50))  
messageLabel.text = message1  
messageLabel
```

Here we use a built-in `UILabel` class to create a label and set its size to 300x50 points. We then set its text to `message1`. To preview a UI element in Playground, you can click the Quick Look or Show Result icon. The Quick Look feature displays the label as a pop-over. If you use the Show Result feature, Playground displays the label inline, right below your code.



It's just a plain label. Wouldn't it be great if you can change its color? That's easy. You just need one line of code to customize the color. What's more is that you can easily center the text and change the label to round corner. Type the following lines and click the Show Result icon.

```
messageLabel.backgroundColor = UIColor.orange
messageLabel.textAlignment = NSTextAlignment.center
messageLabel.layer.cornerRadius = 10.0
messageLabel.clipsToBounds = true
messageLabel
```

You'll see a rounded corner label with orange background like this:



Figure 2-8. Label in orange

This is the power of iOS SDK. It comes with tons of pre-built elements and allows developers to customize them with few lines of code.

Don't get me wrong. Typically you do not need to write code to create the user interface. Xcode provides a Storyboard feature that lets you design UI using drag-and-drop. We'll go through it in the next chapter.

You've got a taste of Swift. What do you think? Love it? I hope you find Swift a lot easier to learn and code. Most importantly, I hope it don't scare you away from learning app development. For reference, you can download the Playground file from <http://www.appcoda.com/resources/swift3/MyPlayground.zip>.

What's Next

What's coming up is that I will teach to build your first app. You can now move onto the next chapter. However, if you want to learn more about the Swift programming language, I would recommend you to [download this Playground file](#) and go through the exercise. The Playground file is similar to an interactive book that covers the basics of Swift. You will learn the language syntax, understand functions, optionals, and many more.

But it's not a must.

If you can't wait to build your first app, move onto the next chapter, and check out the Playground file later.

Chapter 3

Hello World! Build Your First App in Swift



Learn by doing. Theory is nice but nothing replaces actual experience.

– Tony Hsieh

By now you should have installed Xcode 8 and some understandings of Swift language. If you haven't done so, check out the first chapter about what you need to begin iOS programming.

We'll use Xcode 8.0 (or up) to work on all exercises in this book.

You may have heard of the "Hello World" program if you have read any programming book before. Hello World is a program for the first-time programmer to create. It's a very simple program that outputs "Hello, World" on the screen of a device. It's a tradition in the programming world. Let's follow the programming tradition and create a "Hello World" app using Xcode.

Despite its simplicity, the "Hello World" program serves a few purposes:

- It gives you an overview of the syntax and structure of Swift, the new programming language of iOS.
- It also gives you a basic introduction to the Xcode environment. You'll learn how to create an Xcode project and lay out your user interface using Interface Builder. Even if you've used Xcode before, you'll learn what's new in the latest version of Xcode.
- You'll learn how to compile a program, build the app and test it using the built-in simulator.
- Lastly, it makes you think programming is not difficult. I don't want to scare you away from learning programming. It'll be fun.

Your First App

Your first app, as displayed in figure 3-1, is very simple and just shows a "Hello World" button. When a user taps the button, the app shows a welcome message. That's it. Extremely simple but it helps you kick off your iOS programming journey.

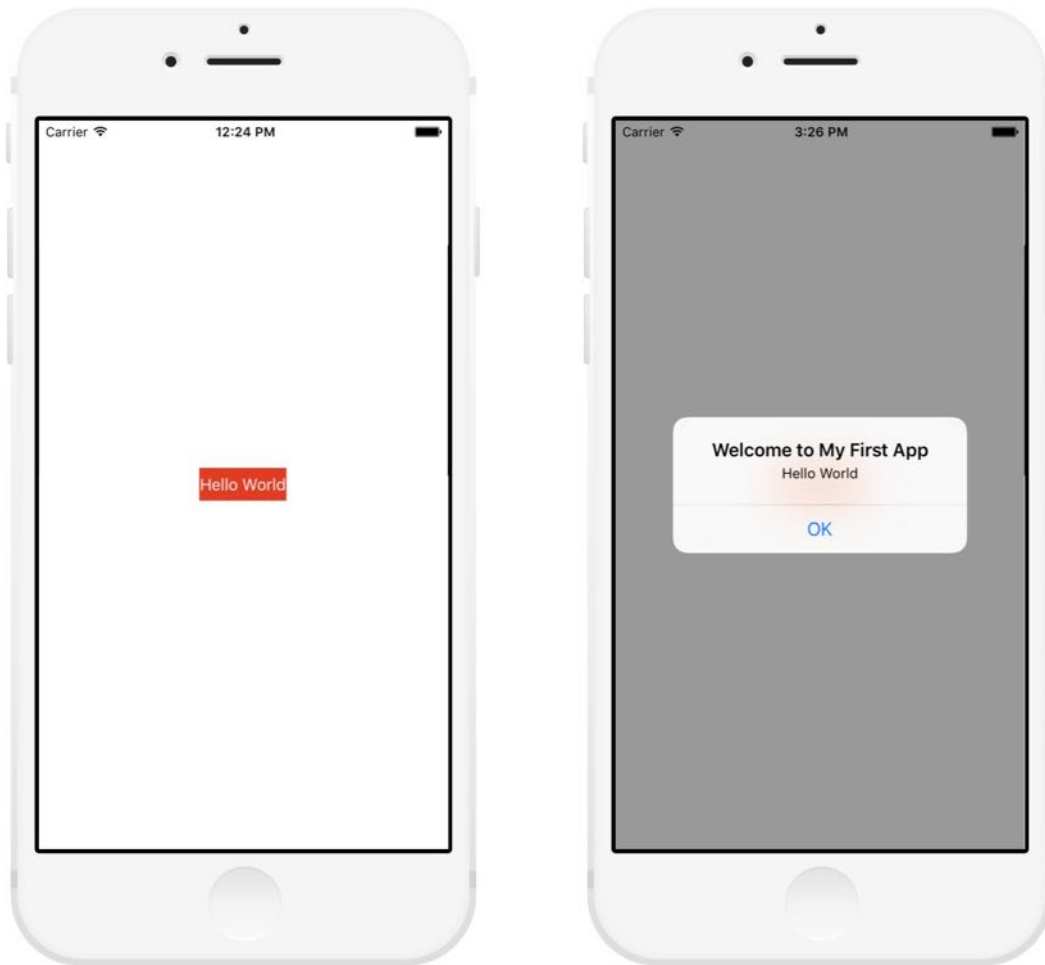


Figure 3-1. HelloWorld App

When you're building your first app, please keep one thing in mind: *forget about the code*. I'm quite sure that you will find some of the Swift code difficult to understand. No worries. Just focus on the exercise, and get yourself familiarized with the Xcode environment. We will talk about the code in the next chapter.

Let's Jump Right Into the Project

First, open Xcode. Once launched, Xcode displays a welcome dialog. From here, choose "Create a new Xcode project" to start a new project:



Welcome to Xcode

Version 8.0 (8A218a)



Get started with a playground

Explore new ideas quickly and easily.



Create a new Xcode project

Create an app for iPhone, iPad, Mac, Apple Watch or Apple TV.



Check out an existing project

Start working on something from an SCM repository.

No Recent Projects

Open another project...

Figure 3-2. Xcode - Welcome Dialog

Xcode shows various project templates for selection. The *Single View Application* template is the most common template for creating an iOS app. So, choose iOS > Single View Application and click *Next*.

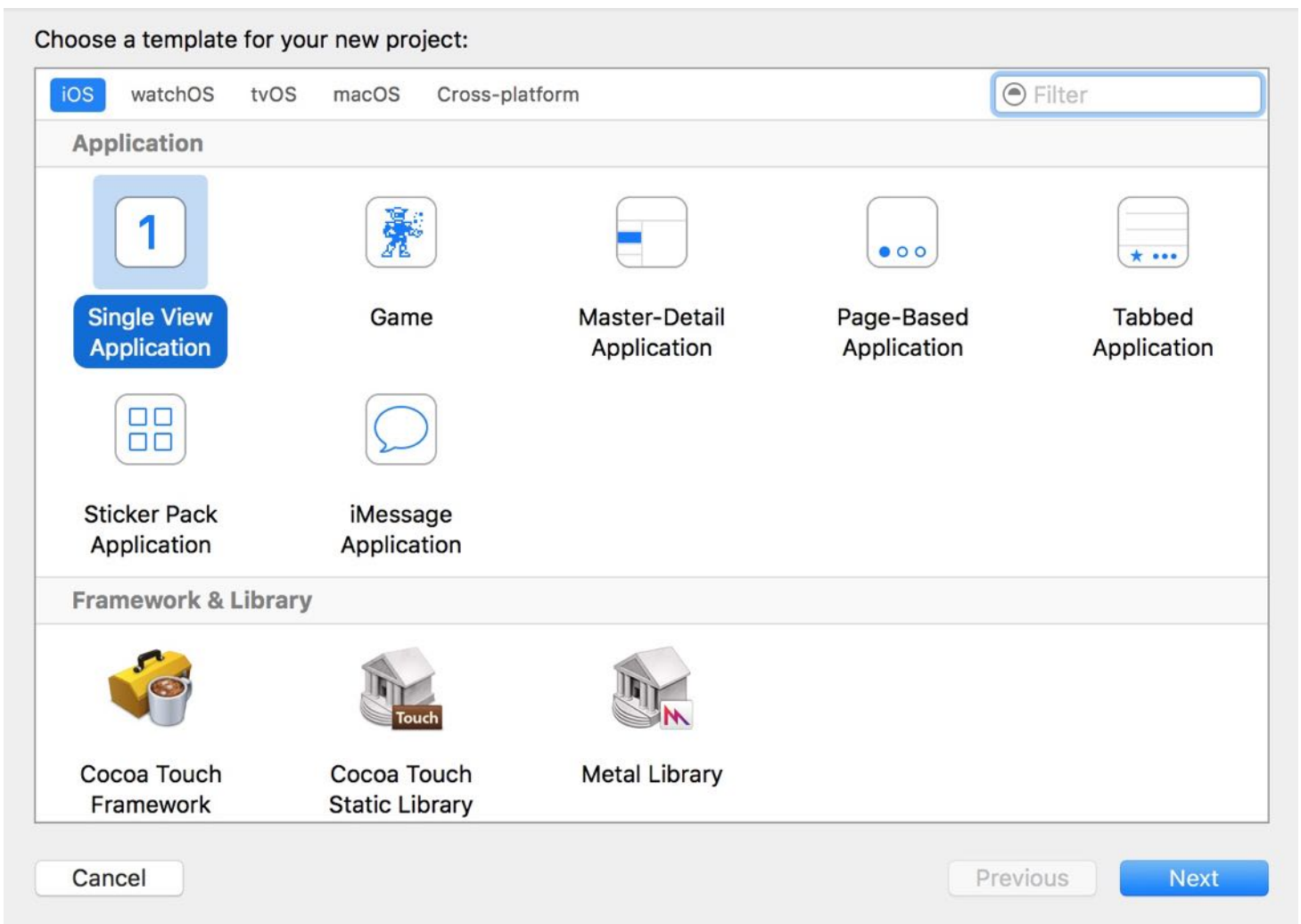


Figure 3-3. Xcode Project Template Selection

This brings you to the next screen to fill in all the necessary options for your project.

Choose options for your new project:

Product Name: HelloWorld

Team: Add account...

Organization Name: AppCoda

Organization Identifier: com.appcoda

Bundle Identifier: com.appcoda.HelloWorld

Language: Swift

Devices: Universal

Use Core Data

Include Unit Tests

Include UI Tests

Cancel Previous Next

Figure 3-4. Options for your Hello World project

You can simply fill in the options as follows:

- **Product Name: HelloWorld** – This is the name of your app.
- **Team:** Just leave it as it is. If you click the button, it will prompt you to sign in with your Apple ID. For your first app, just skip this step.
- **Organization Name: AppCoda** – It's the name of your organization. You may use your name if you're not building an app for your organization.
- **Organization Identifier: com.appcoda** – It's actually the domain name written the other way round. If you have a domain, you can use your own domain name. Otherwise,

you may use "com.appcoda" or just fill in "edu.self".

- **Bundle Identifier: com.appcoda.HelloWorld** - It's a unique identifier of your app, which is used during app submission. You do not need to fill in this option. Xcode automatically generates it for you.
- **Language: Swift** – Xcode 8 supports both Objective-C and Swift for app development. As this book is about Swift, we'll use Swift to develop the project.
- **Devices: iPhone** – Select "iPhone" for this project. Let's keep things simple and build the app for the iPhone device only.
- **Use Core Data: [unchecked]** – Do not select this option. You do not need Core Data for this simple project. We'll explain Core Data in later chapters.
- **Include Unit Tests: [unchecked]** – Do not select this option. You do not need unit tests for this simple project.
- **Include UI Tests: [unchecked]** – Do not select this option. You do not need UI tests for this simple project.

Click "Next" to continue. Xcode then asks you where to save the "HelloWorld" project. Pick any folder (e.g. Desktop) on your Mac. You may notice there is an option for source control. Just deselect it. We do not need to use the option in this book. Click "Create" to continue.

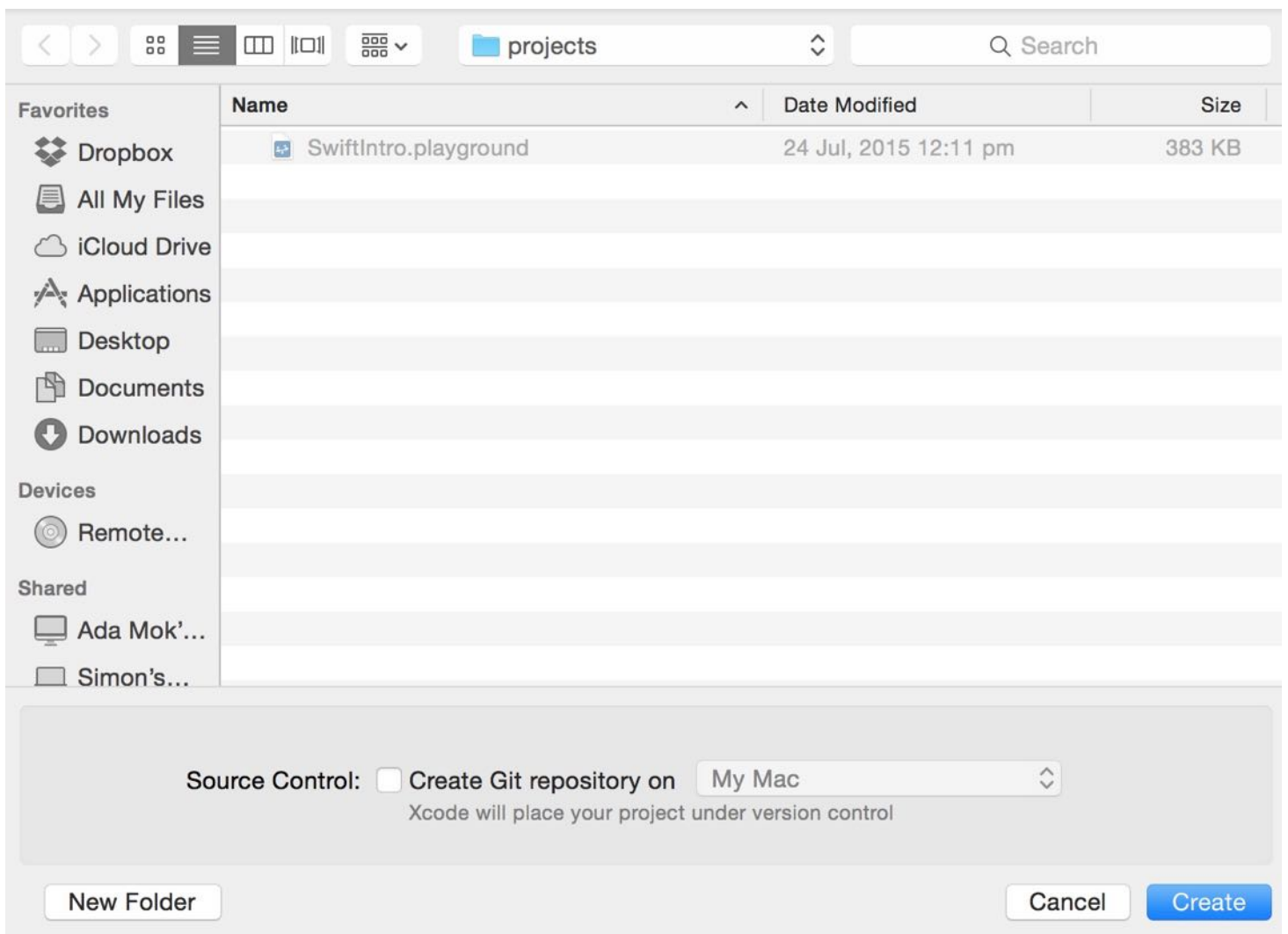
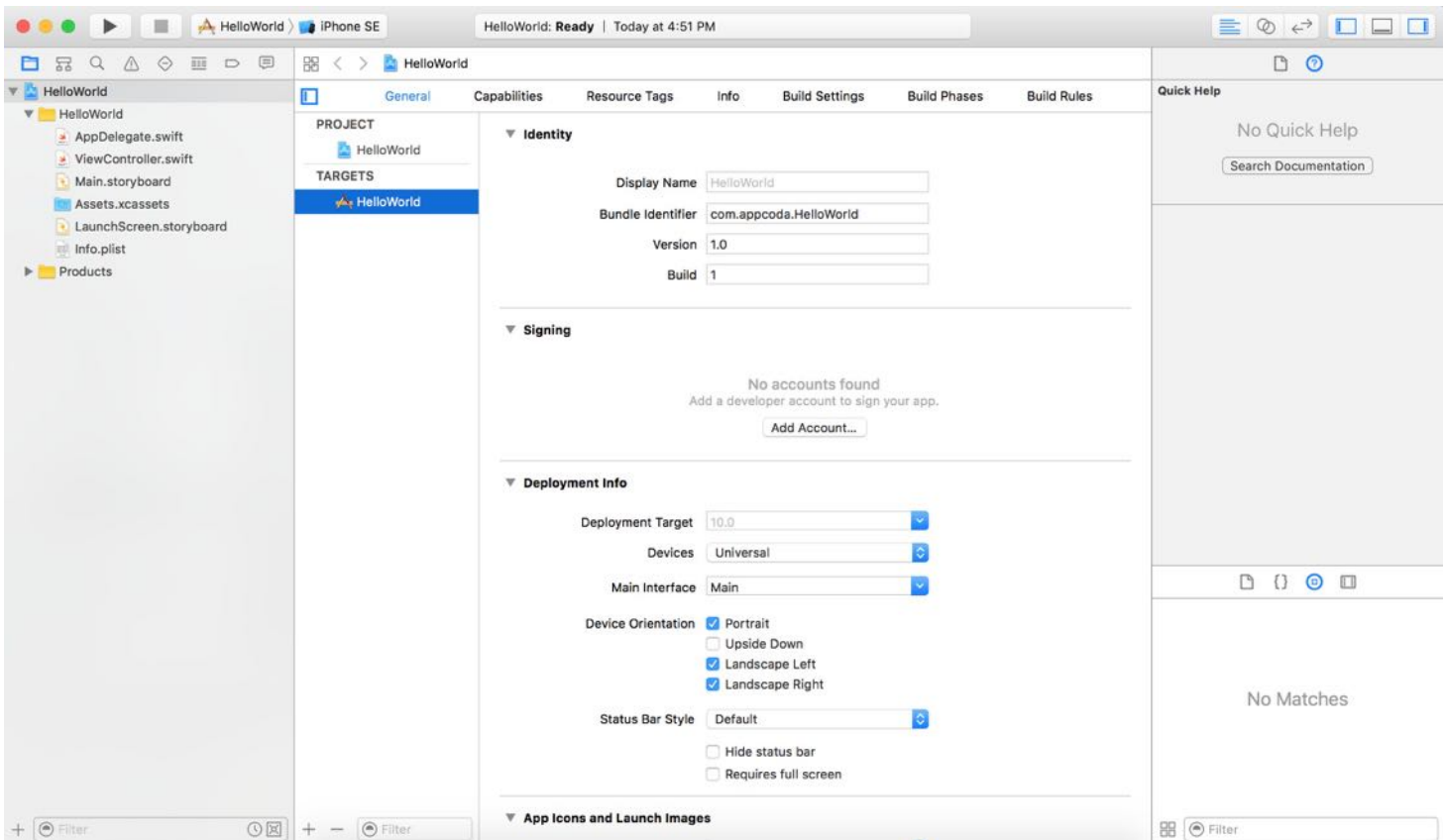


Figure 3-5. Choose a folder and save your project

After you confirm, Xcode automatically creates the "Hello World" project. The screen will look like the screenshot shown in figure 3-6.



Familiarize Yourself with Xcode Workspace

Before we start to write some code, let's take a few minutes to have a quick look at the Xcode workspace environment. In the left pane is the project navigator. You can find all your project files in this area. The center part of the workspace is the editor area. You do all the editing stuff here (such as editing the project setting, source code file, user interface) in this area.

Depending on the type of file, Xcode shows you different interfaces in the editor area. For instance, if you select `viewController.swift` in the project navigator, Xcode displays the source code in the center area (see figure 3-7). If you select `Main.storyboard`, which is the file for storing user interface, Xcode shows you the visual editor for storyboard (see figure 3-8).

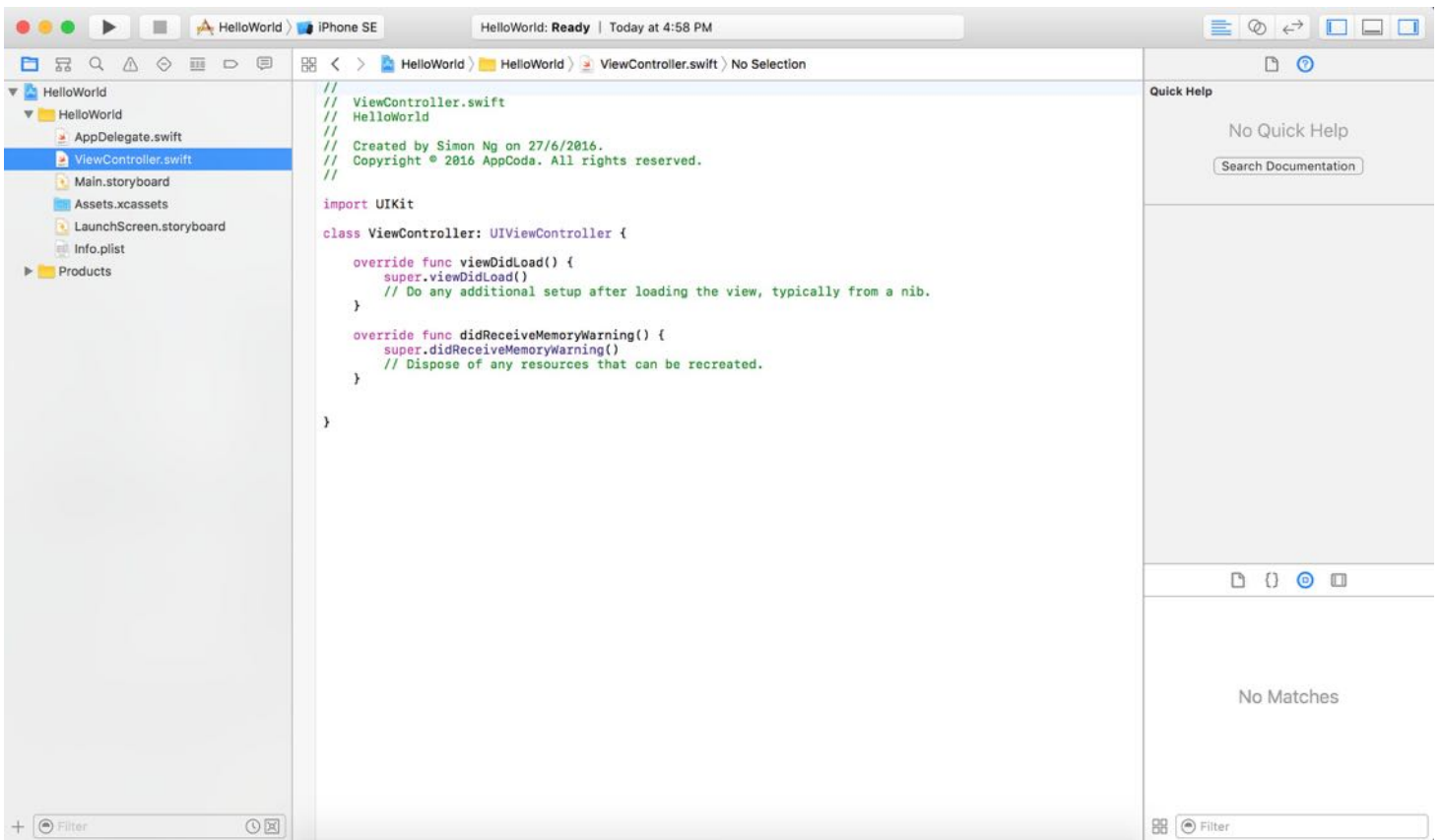


Figure 3-7. Xcode Workspace with Source Code Editor

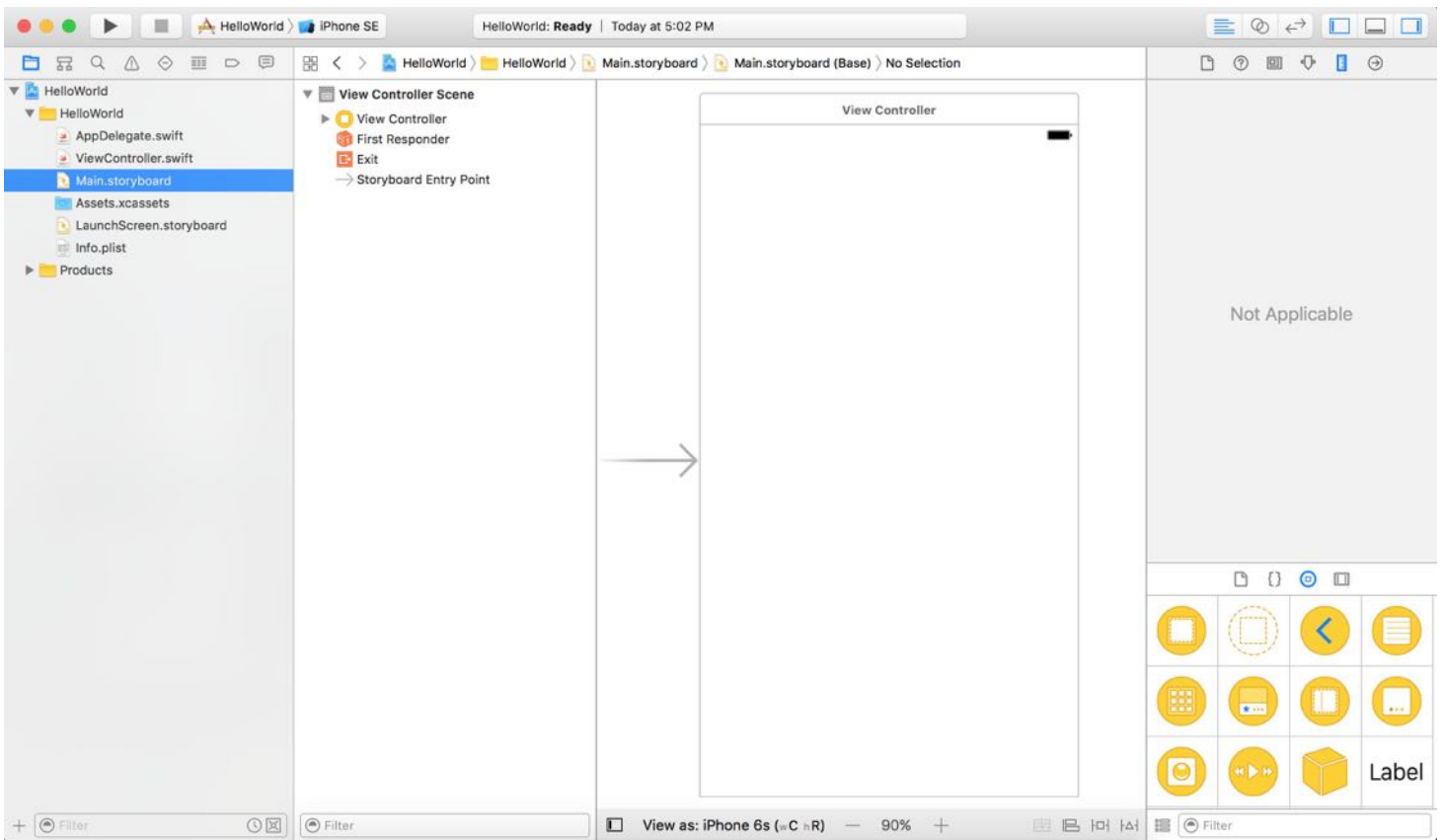


Figure 3-8 . Xcode Workspace with Storyboard Editor

The rightmost pane is the utility area. This area displays the properties of the file and allows you to access Quick Help. If Xcode doesn't show this area, you can select the rightmost button in the toolbar (at the top-right corner) to enable it.

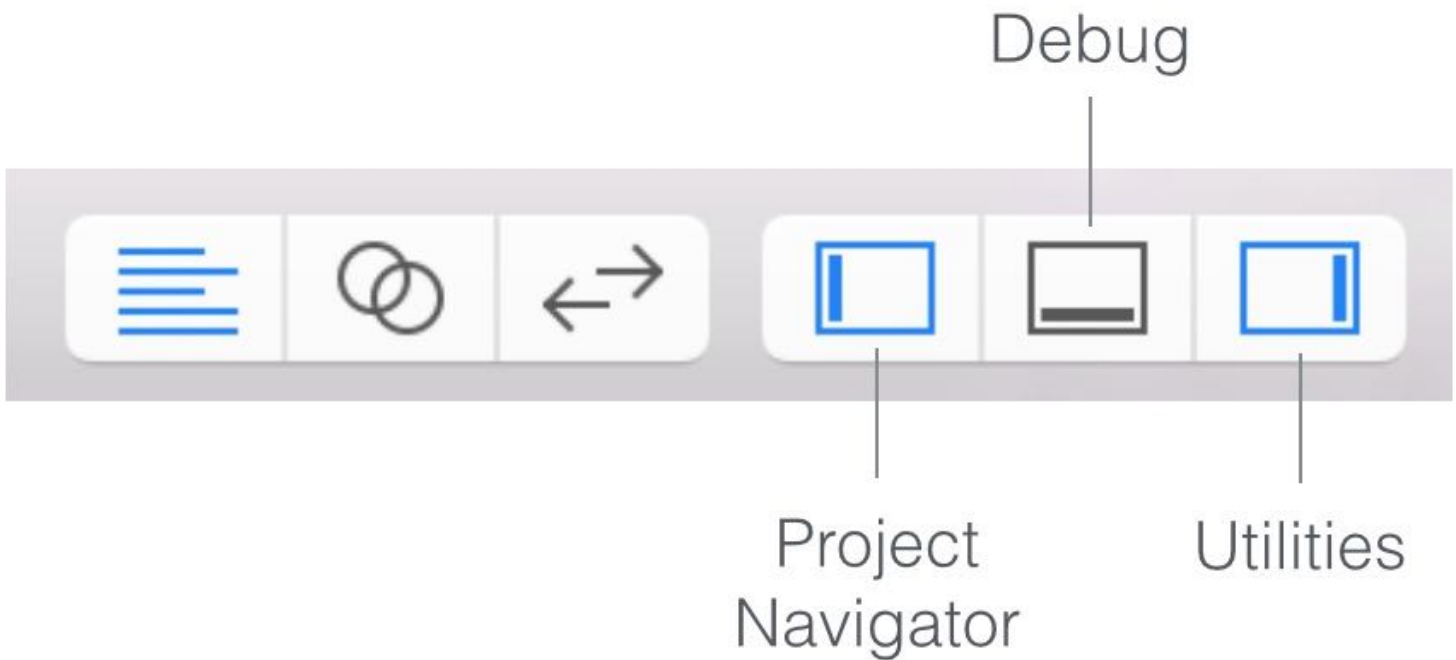


Figure 3-9. Show/hide the content areas of your workspace

The middle view button of the view selector is deselected by default. If you click on it, Xcode displays the debug area right below the editor area. The debug area, as its name suggests, is used for showing debug messages. We'll talk about that in a later chapter, so don't worry if you do not understand what each area is for.

Run Your App for the First Time

Until now, we haven't written a single line of code. Even so, you can run your app using the built-in simulator. This will give you an idea how to build and test your app in Xcode. In the toolbar you should see the Run button.

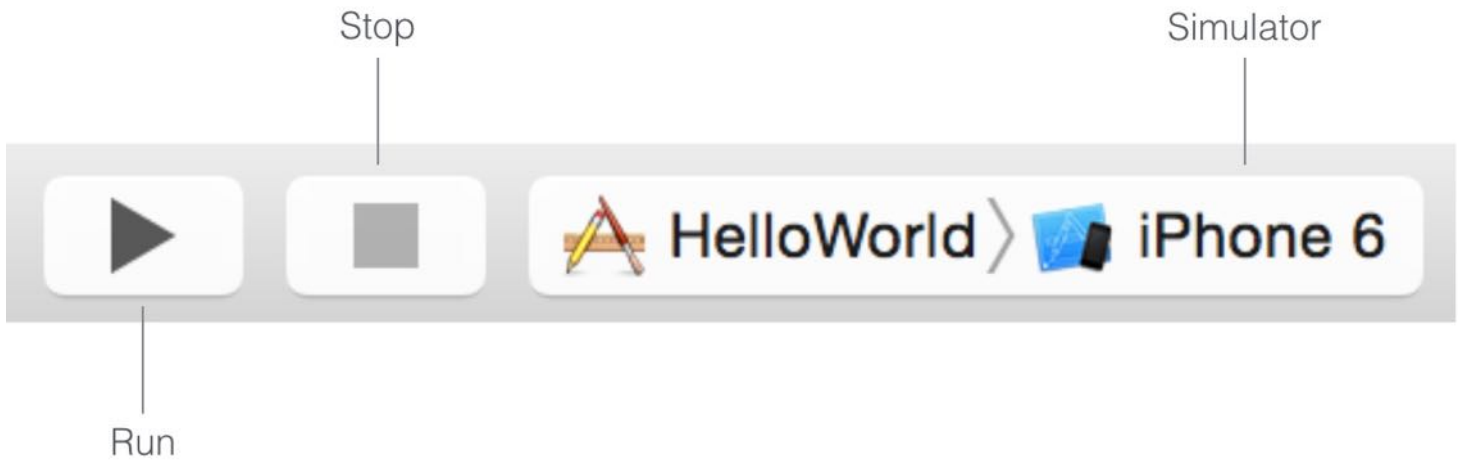


Figure 3-10. Run and Stop Buttons in Xcode

The *Run* button in Xcode is used to build an app and run it in the selected simulator. By default, the Simulator is set to *iPhone SE*. If you click the *iPhone SE* button, you'll see a list of available simulators such as iPhone 4s and iPhone 6s Plus. Let's select *iPhone 6* as the Simulator, and give it a try.

Once selected, you can click the *Run* button to load your app in the Simulator. Figure 3-11 shows the simulator of an iPhone 6.

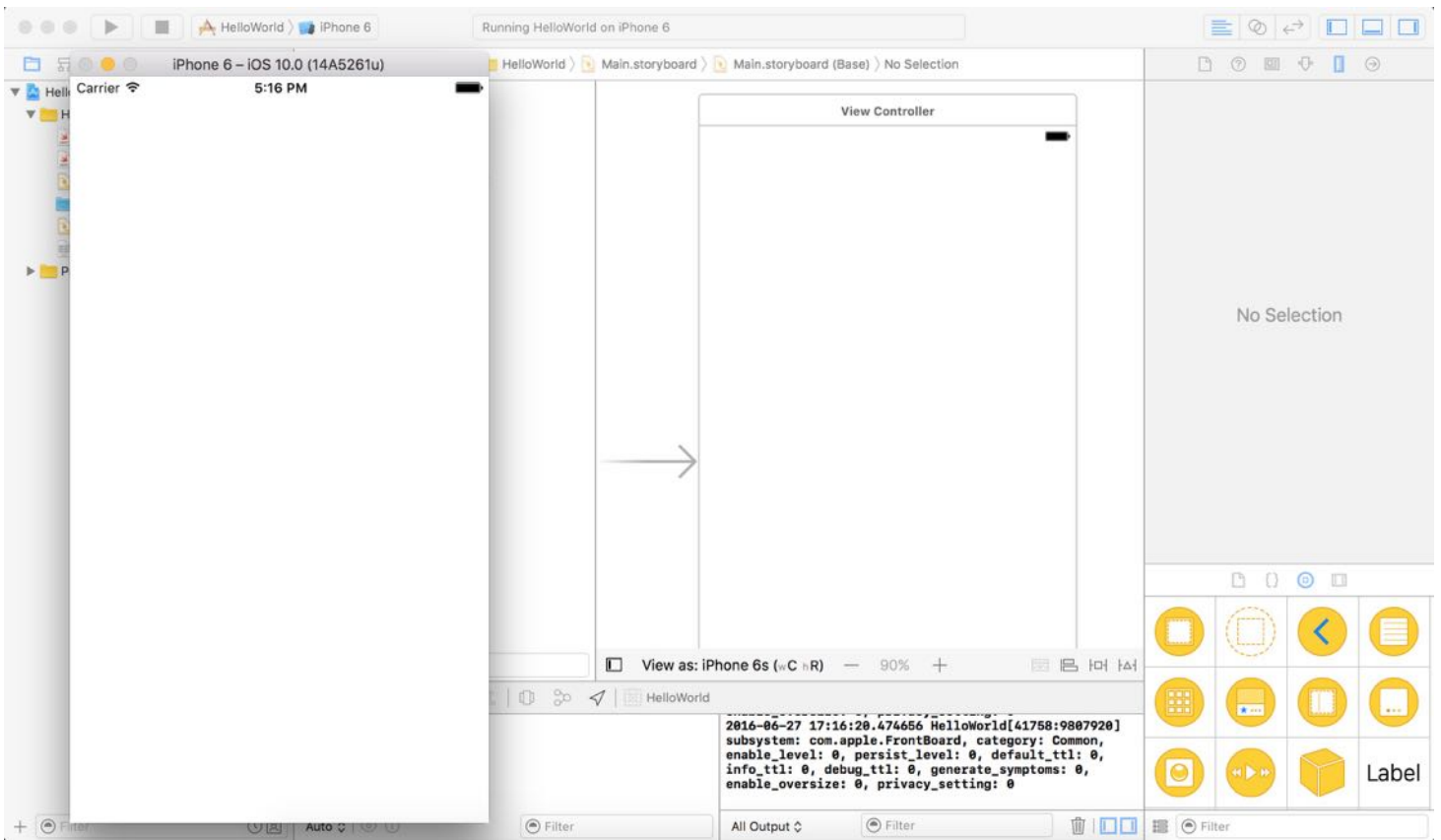


Figure 3-11. The Simulator

Quick tip: On non-retina Mac, it may not be able to show the full simulator window. You can select the simulator, and press `command+2` or `command+3` to scale it down. Alternatively, you can go up to the menu, and choose `Window > Scale`.

A white screen with nothing inside?! That's normal. So far we haven't designed the user interface or written any lines of code. This is why the simulator shows a blank screen. To terminate the app, simply hit the *Stop* button in the toolbar.

Try to select another simulator (e.g. iPhone SE) and run the app. Just play around with it to get yourself familiarize with the Xcode development environment.

A Quick Walkthrough of Interface Builder

Now that you have a basic idea of the Xcode development environment, let's move on and design the user interface of your first app. In the project navigator, select the `Main.storyboard` file. Xcode then brings up a visual editor for storyboards, known as Interface Builder.

The Interface Builder editor provides a visual way for you to create and design an app's UI. Not only can you use it to design individual view (or screen), the Interface Builder's storyboard designer lets you lay out multiple views, and chain them together using different types of transitions to create the complete user interface. All these can be done without writing a line of code.

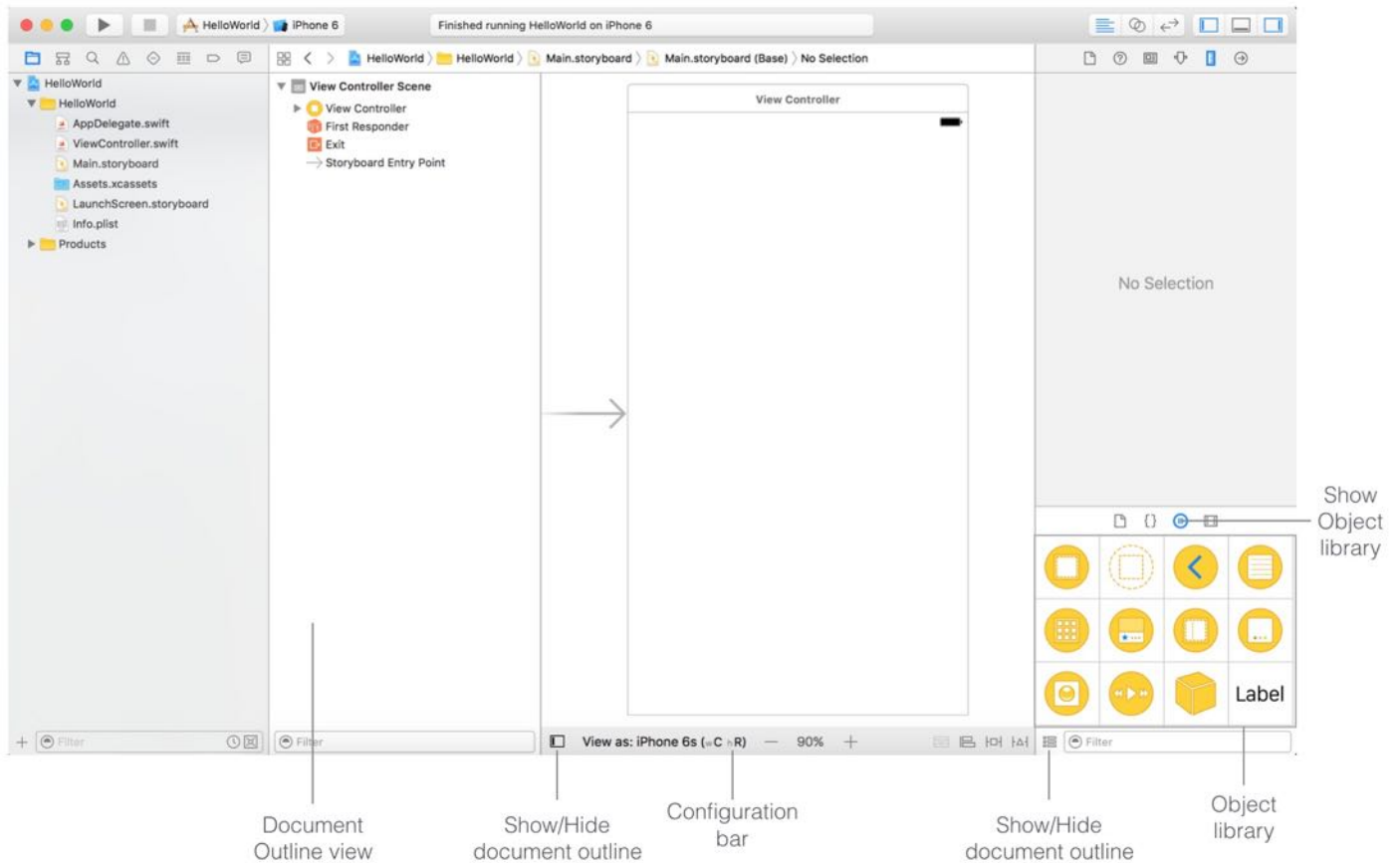


Figure 3-12 . The Interface Builder Editor

The Object library at the lower right pane contains all the available UI objects such as button, label, image view for you to design your user interface. If you can't see the Object library, click the *Show the Object library* button, as displayed in figure 3-12, to reveal it. You can view the Object library in two difference modes: *list view* and *icon view*. I prefer to use icon view in this book. But if you want to change to the list view mode, simply use the toggle button to switch between them.

Since we chose to use the *Single View Application* template during project creation, Xcode

generated a default view controller scene in the storyboard. In your Interface Builder, you should see a view controller in the editor area. This view controller is where you design the app's user interface. Each screen of an app is usually represented by a view controller. Interface Builder allows you to add multiple view controllers to the storyboard and link them up. Later in this book, we will further discuss about that. Meanwhile, focus on learning how to use Interface Builder to lay out the UI for that view controller.

What is a Scene?

A scene in storyboard represents a view controller and its views. When developing iOS apps, views are the basic building blocks for creating your user interface. Each type of view has its own function. For instance, the view you find in the storyboard is a container view for holding other views such as buttons, labels, image views, etc.

A view controller is designed to manage its associated view and subviews (e.g. button and label). If you are confused about the relationship between views and view controllers, no worries. We will discuss how a view and view controller work together in the later chapters.

The Document Outline view of the Interface Builder editor shows you an overview of all scenes and the objects under a specific scene. The outline view is very useful when you want to select a particular object in the storyboard. If the outline view doesn't appear on screen, use the toggle button (see figure 3-12) to enable/disable the outline view.

Lastly, there is a configuration bar in the Interface Builder. To reveal the bar, place the mouse cursor on `View as: iPhone 6s`, and then single-click on it. The configuration bar is a new feature in Xcode 8 that lets you live preview your app UI on different devices. Furthermore, you can use the + and - buttons to zoom in/out the storyboard. We will talk about this new feature later on.

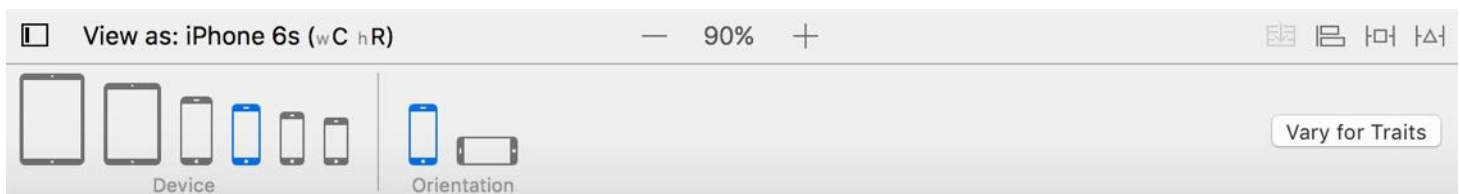


Figure 3-13 . The new configuration bar in Xcode 8

Designing the User Interface

Now we are going to design the app's user interface. First, we will add a Hello World button to the view. At the bottom part of the utility area, it shows the Object library. Here, you can choose any of the UI objects, and drag-and-drop them into the view. If you're in the icon view mode of the Object library, you can click on any of the objects to reveal the detailed description.

Okay, it's time to add a button to the view. All you need to do is drag a Button object from the Object library to the view.

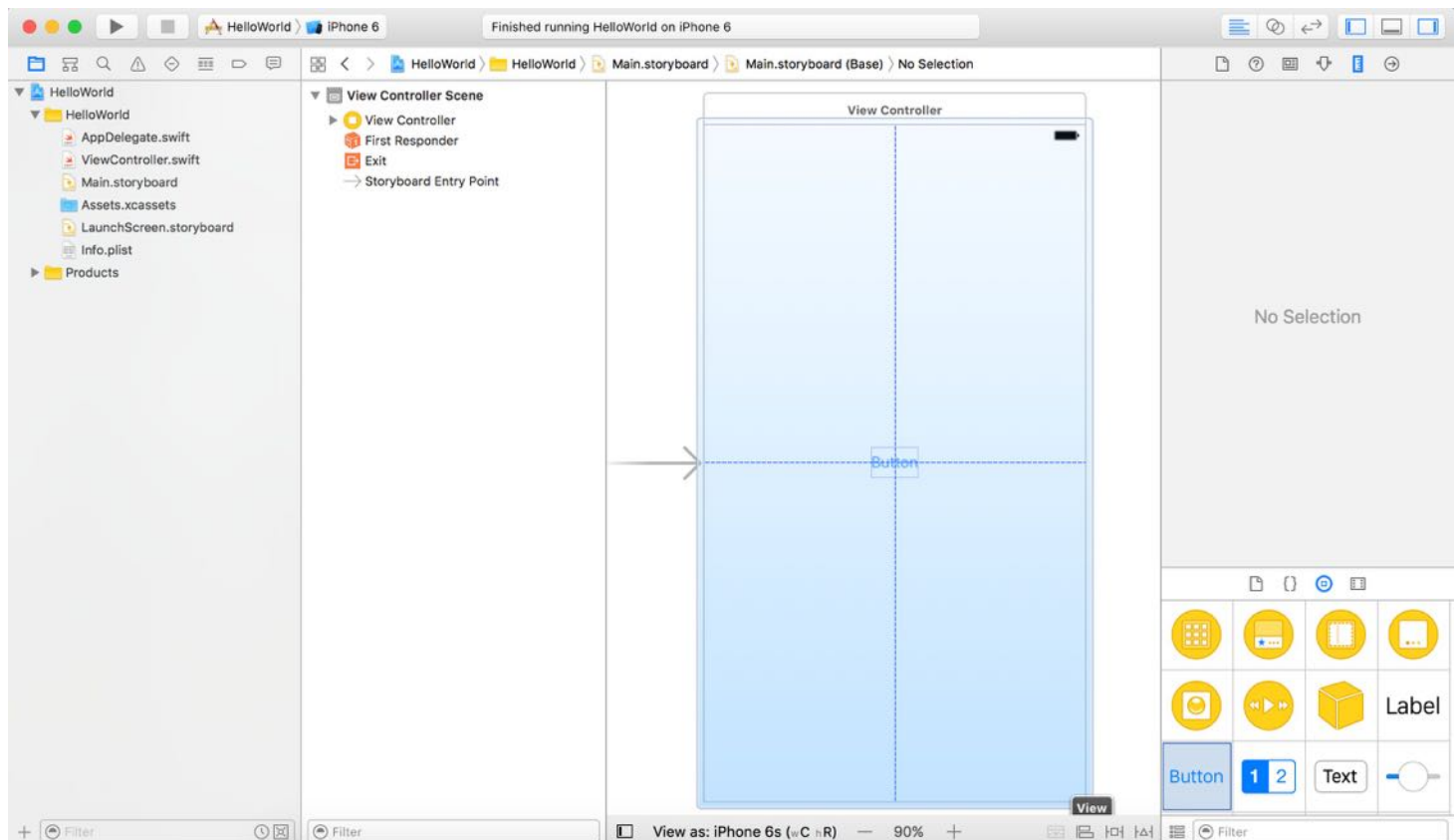


Figure 3-14. Drag the Button to the View

As you drag the Button object to the view, you'll see a set of horizontal and vertical guides if the button is centered. Stop dragging, and release your button to place the Button object there.

Next, let's rename the button. To edit the label of the button, double-click it and name it "Hello

World". After the change, you may need to center the button again.



Figure 3-15. Renaming the button

Great! You're now ready to test your app. Select the iPhone 6/6s simulator and hit the Run button to run the project, you should see a Hello World button in the simulator as shown in figure 3-16. Cool, right?

However, when you tap the button, it shows nothing. We'll need to add a few lines of code to display the "Hello, World" message.

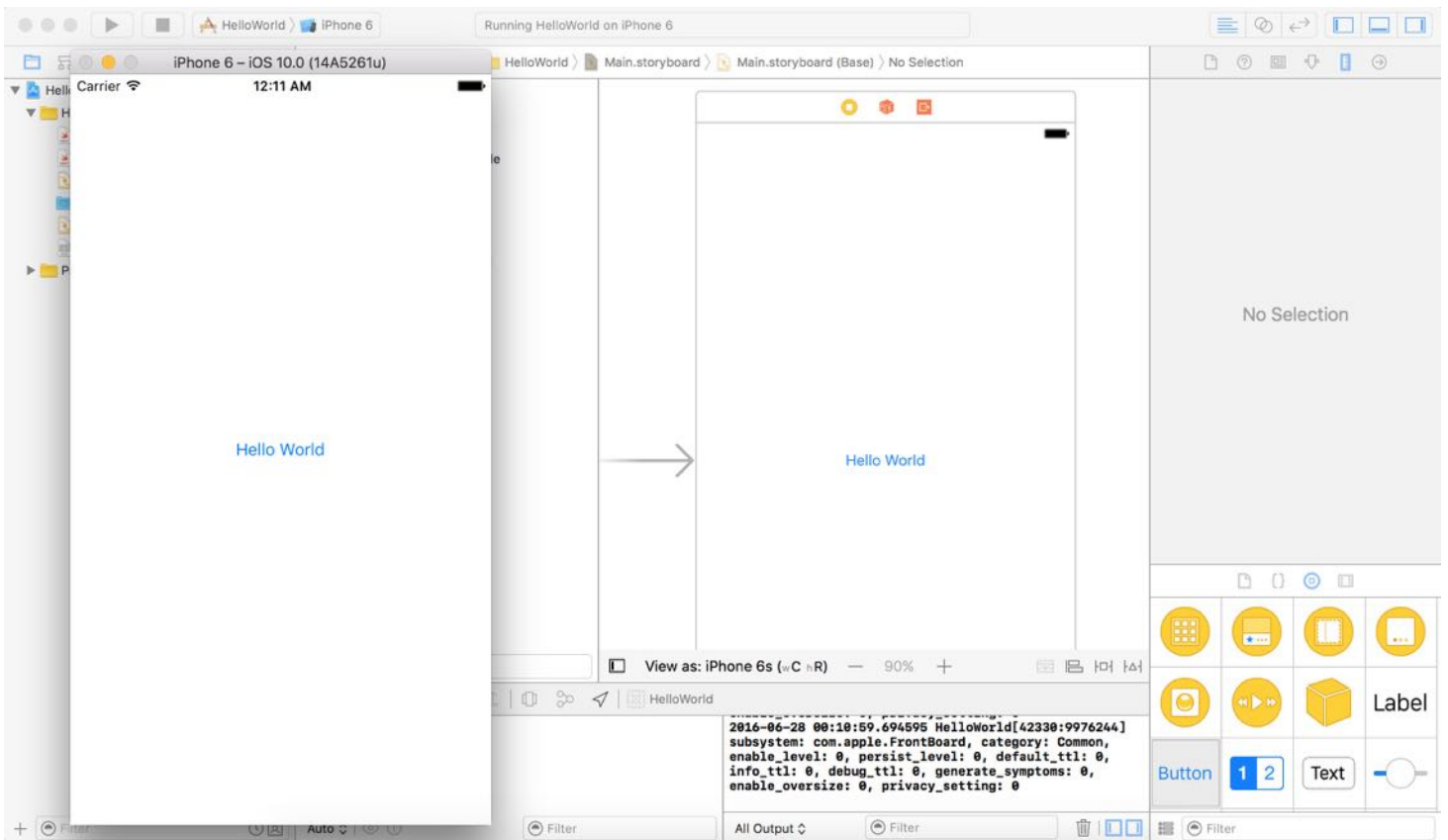


Figure 3-16. Hello World app with a Button

Quick note: This is the beauty of iOS development. The code and user interface of an app are separated. You're free to design your user interface in Interface Builder and prototype an app without writing any lines of code.

Coding the Hello World Button

Now that you've completed the UI of the HelloWorld app, it's time to write some code. In the project navigator, you should find the `ViewController.swift` file. Because we initially selected the *Single View Application* project template, Xcode already generated a `ViewController` class in the `ViewController.swift` file. This file is actually associated with the view controller in the storyboard. In order to display a message when the button is tapped, we'll add some code to the file.

Swift versus Objective-C

If you have written code in Objective-C before, one big change in Swift is the consolidation of header (.h) and implementation file (.m). All the information of a particular class is now stored in a single .swift file.

Select the `ViewController.swift` file and the editor area immediately displays the source code. Type the following lines of code in the `ViewController` class:

```
@IBAction func showMessage() {
    let alertController = UIAlertController(title: "Welcome to My First App",
message: "Hello World", preferredStyle: UIAlertControllerStyle.alert)
    alertController.addAction(UIAlertAction(title: "OK", style:
UIAlertActionStyle.default, handler: nil))
    present(alertController, animated: true, completion: nil)
}
```

Quick note: I encourage you to type the code, rather than copy & paste it.

Your source code should look like this after editing:

```
import UIKit

class ViewController: UIViewController {

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

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

    @IBAction func showMessage() {
        let alertController = UIAlertController(title: "Welcome to My First
App", message: "Hello World", preferredStyle: UIAlertControllerStyle.alert)
        alertController.addAction(UIAlertAction(title: "OK", style:
UIAlertActionStyle.default, handler: nil))
        present(alertController, animated: true, completion: nil)
    }
}
```

What you have just done is added a `showMessage()` method in the `ViewController` class. The Swift code within the method is new to you. I will explain it to you in the next chapter. Meanwhile, just consider the `showMessage()` as an action. When this action is called, the block of code will instruct iOS to display a "Hello World" message on screen.

Connecting the User Interface with Code

I said before that the beauty of iOS development is the separation of code (.swift file) and user interface (storyboards). But how can we establish the relationship between our source code and user interface?

To be specific for this demo, the question is:

- How can we connect the "Hello World" button in the storyboard with the `showMessage()` method in the `ViewController` class?

You need to establish a connection between the "Hello World" button and the `showMessage()` method you've just added, such that the app responds when someone taps the Hello World button.

Now select `Main.storyboard` to switch back to the Interface Builder. Press and hold the control key of the keyboard, click the "Hello World" button and drag it to the View Controller icon. Release both buttons (mouse + keyboard) and a pop-up shows the `showMessage` option under Sent Events. Select it to make a connection between the button and `showMessage` action.

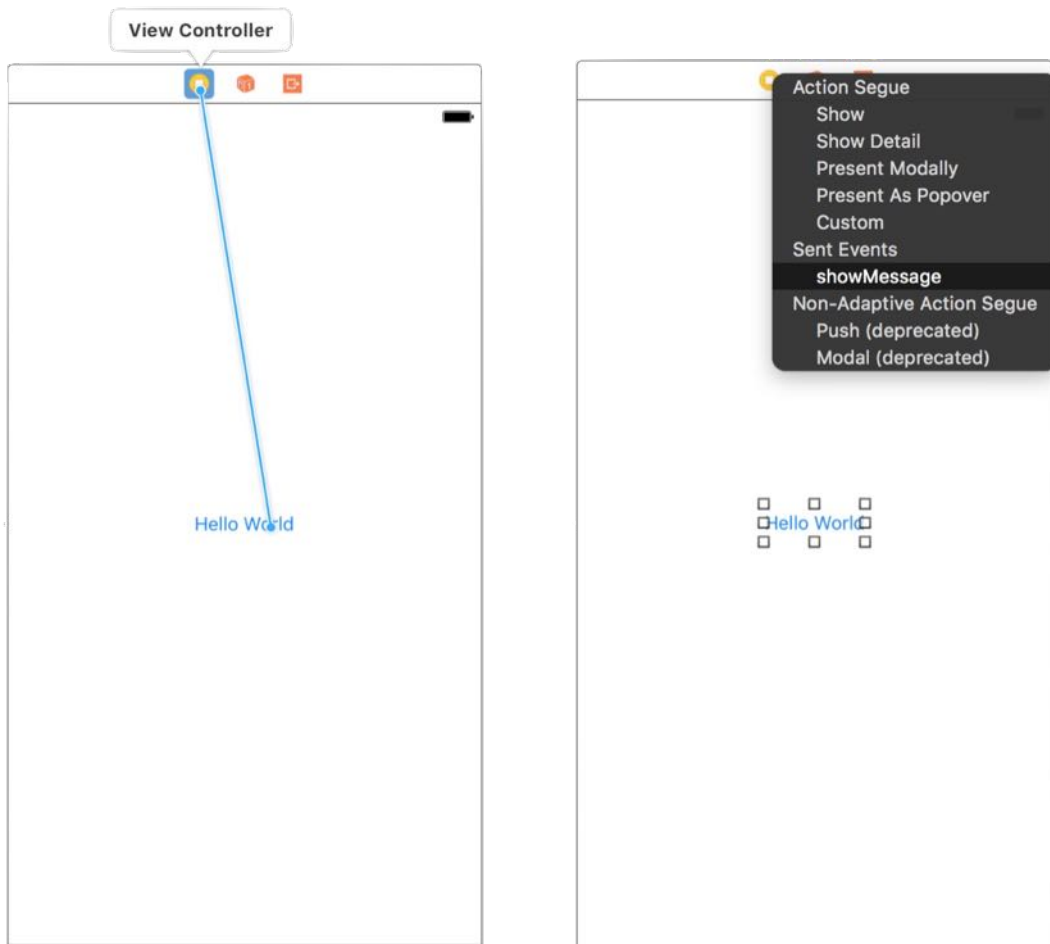


Figure 3-17. Drag to the View Controller icon (left), a pop-over menu appears when releasing the buttons (right)

Test Your App

That's it! You're now ready to test your first app. Just hit the *Run* button. If everything is correct, your app should run properly in the simulator. This time, the app displays a welcome message when you tap the Hello World button.

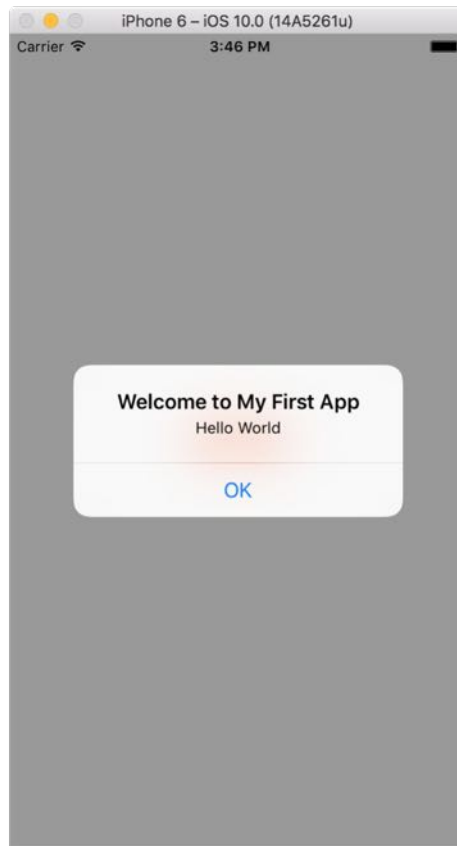


Figure 3-18. Hello World app

Changing the Button Color

There is one more thing I want to discuss with you before ending the chapter. As mentioned before, you do not need to write code to customize a UI control. Here, I want to show you how easy it is to change the properties (e.g. color) of a button.

Select the "Hello World" button and then click the Attributes inspector under the Utility area. You'll be able to access the properties of the button. Here, you can change the font, text color, background color, etc. Try to change the text color (under Button section) to *white* and background (scroll down and you'll find it under View section) to *red* or whatever color you want.

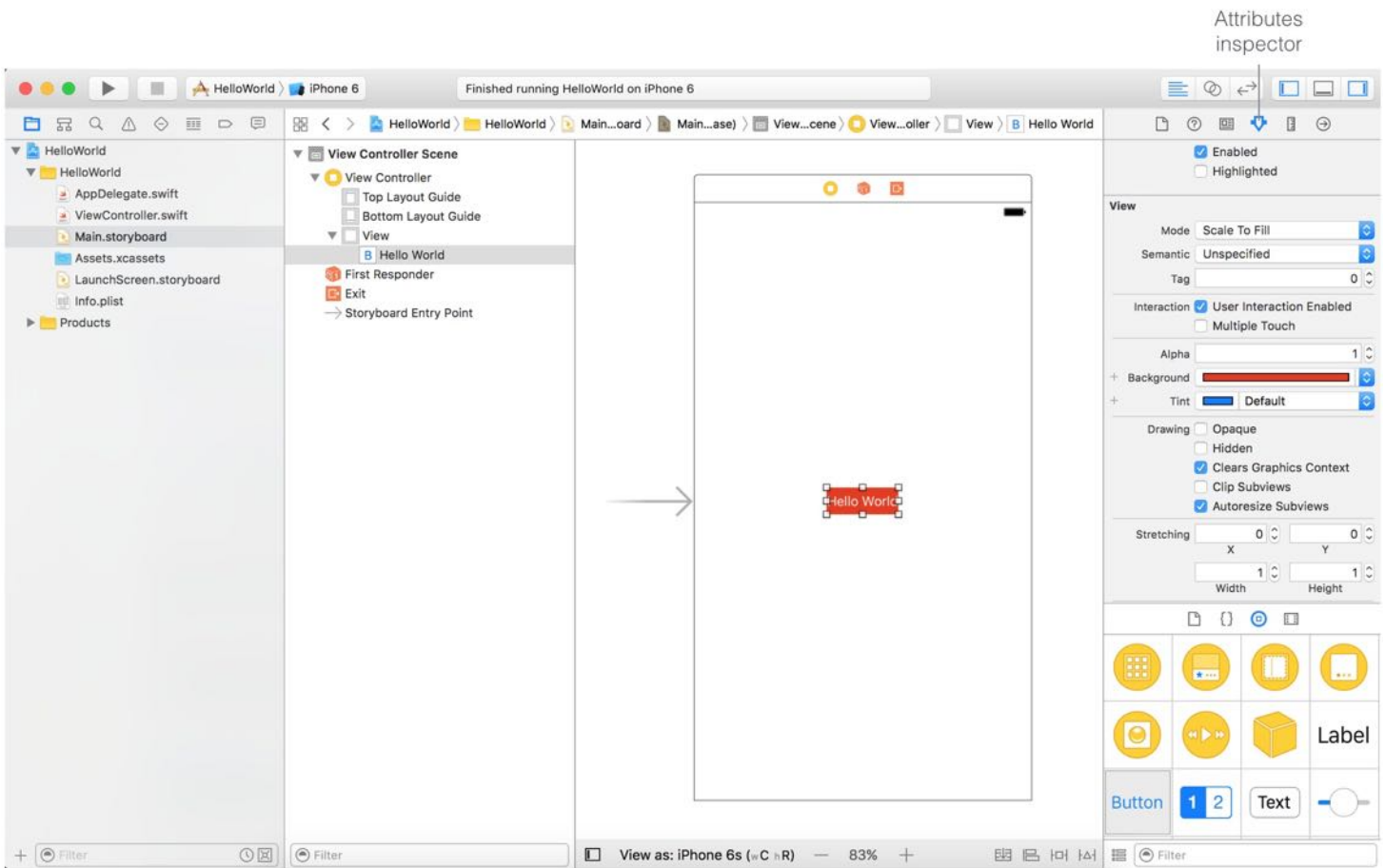


Figure 3-19. Changing the color of the Hello World button

Run the project again and see what you get.

Your Exercise

Not only can you change the color of a button, you can modify the font type and size in the Attributes inspector by setting the *Font* option. Your task is to continue to work on the project and create a user interface like figure 3-20. When a user taps any of the buttons, the app displays the same Hello World message.

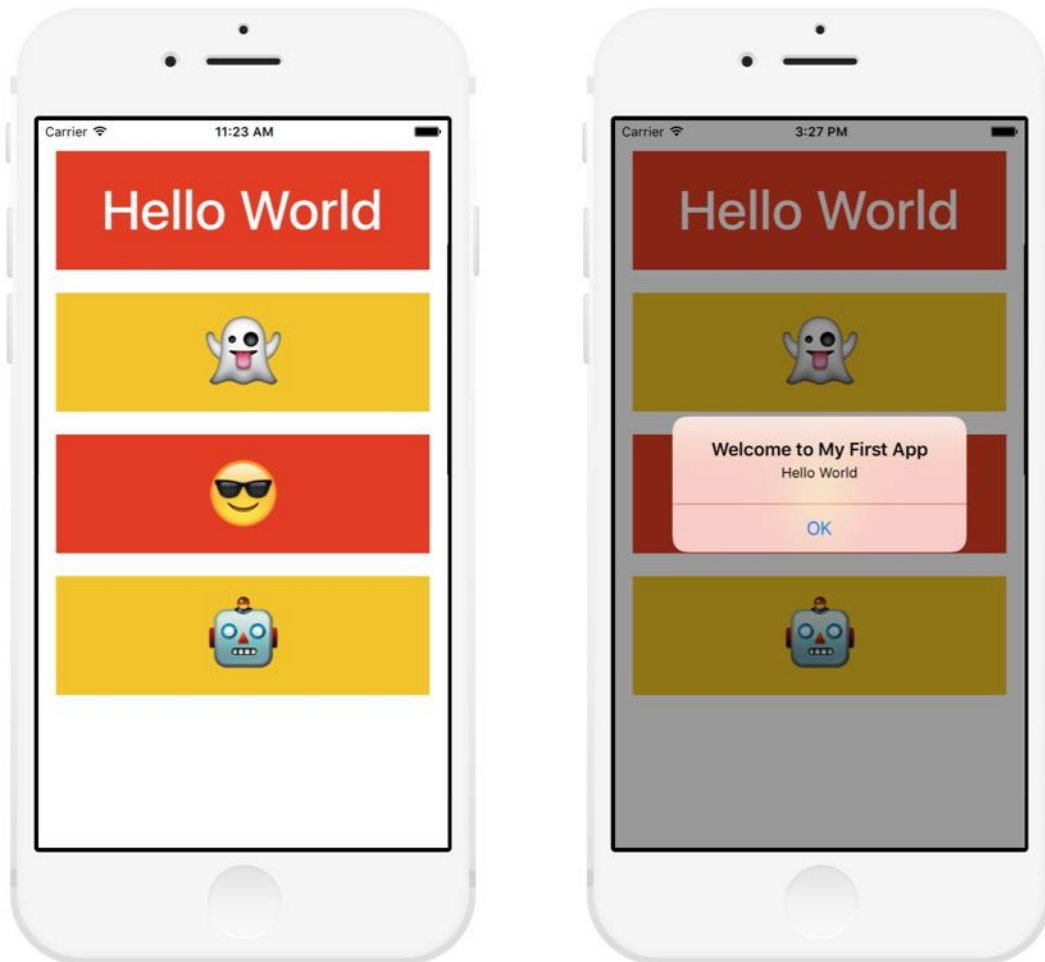


Figure 3-20. Your exercise

To give you some hints, here are the things you will need to:

1. Resize the "Hello World" button and change its font size to *50 points*. Also, set the font style from *Regular* to *Medium*.
2. Add three more buttons. Each of them has an emoji icon as its title. To key in emoji characters, you can hold down control+command and then press spacebar.
3. Establish a connection between the buttons and the `showMessage()` method.

What's Coming Next

Congratulations! You've built your first iPhone app. It's a simple app, but I believe you already have a better understanding of Xcode and understand how an app is built. It's easier than you

thought, right?

In the next chapter, we'll discuss the details of the Hello World app and explain how everything works together.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/HelloWorld.zip>.

For the solution of the exercise, you can download it from <http://www.appcoda.com/resources/swift3/HelloWorldExercise.zip>.

Chapter 4

Hello World App Explained

Any fool can know. The point is to understand.

– Albert Einstein

Isn't it easy to build an app? I hope you enjoyed building your first app.

Before we continue to explore the iOS SDK, let's take a pause here and have a closer look at the Hello World app. It'll be good for you to understand the basics of the Swift language and the inner workings of the app.

So far you followed the step-by-step procedures to build the Hello World app. As you read through the chapter, you may have had a few questions in mind:

- How does the View Controller in the storyboard link up with the `viewController` class in the `viewController.swift` file?
- What does the block of code inside the `showMessage()` method mean? How does it tell iOS to show a Hello World message?
- What does `@IBAction` keyword mean?
- What is behind the "Hello World" button? How can the button detect tap and trigger the `showMessage()` method?
- How does the Run button in Xcode work? What do you mean by "compile an app"?

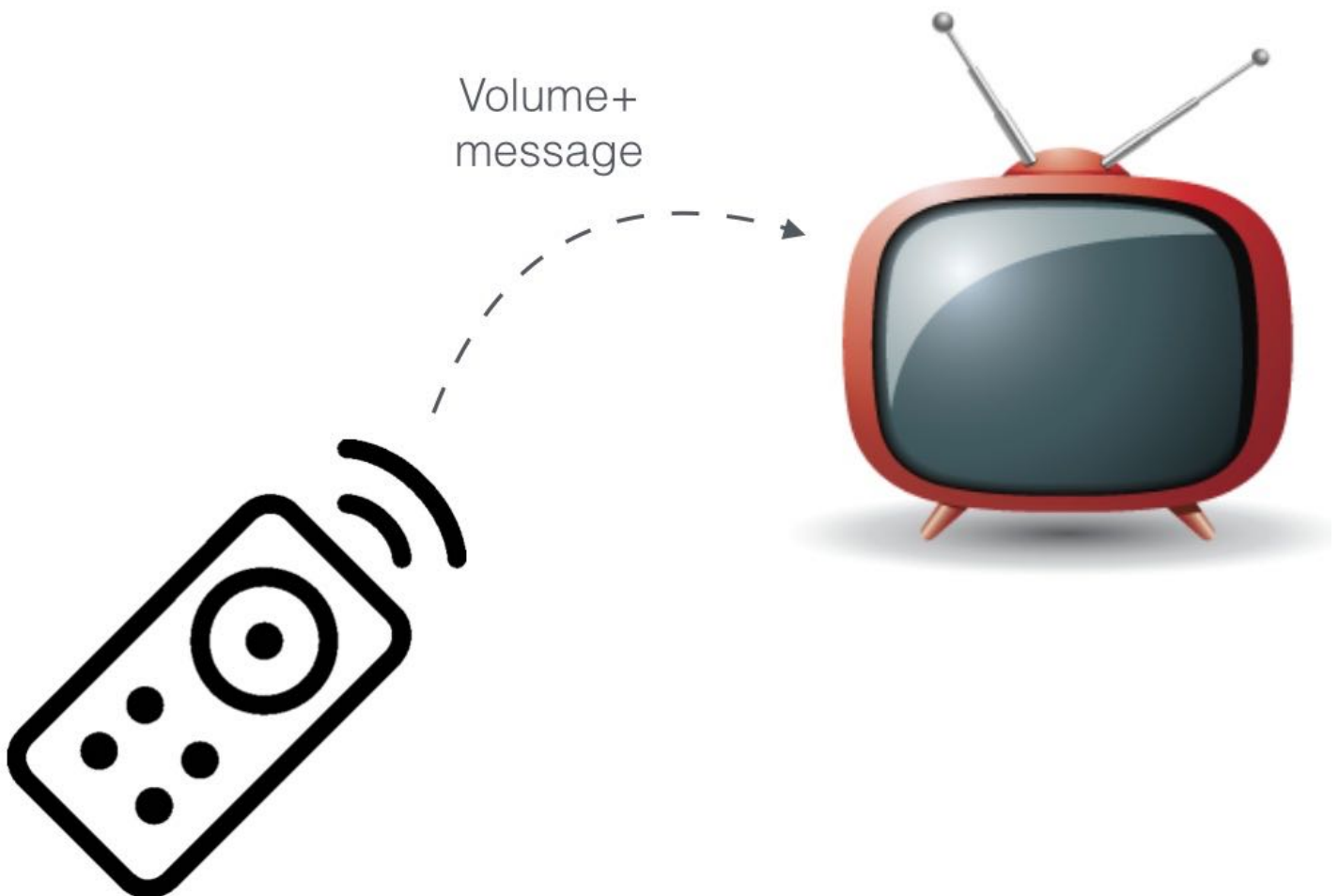
I wanted you to focus on exploring the Xcode environment so I didn't explain any of the above questions. Yet it's essential for every developer to understand the details behind the code and grasp the basic concept of iOS programming. The technical concepts may be a bit hard to understand, in particular, if you have no prior programming experience. Don't worry because this is just the beginning. As you continue to study and write more code in the later chapters, you will gain a better understanding of iOS programming. Just try your best to learn as much as possible.

Before diving into the programming concept, let's take a look at a real life example.

Consider a TV remote control. It's convenient to control the volume of a TV set wirelessly with a remote. To switch TV channels, you simply key in the channel number. To increase the speaker volume, you press the Volume + button.

Let me ask you. Do you know what happens behind the scene when pressing the Volume button or the channel button? Probably not. I believe most of us don't know how a remote control communicates with a TV set wirelessly. You may just think that the remote sends a certain message to the TV and triggers the volume increase or channel switch.

In this example, the button that interacts with you is commonly characterized as the *interface* and the inner details that hide behind the button are referred as the *implementation*. The interface communicates with the implementation via a message.



This concept can also be applied in the iOS programming world. The user interface in storyboard is the *interface*, while the code is the *implementation*. The user interface objects

(e.g. button) communicate with the code via messages.

Specifically, if you go back to the Hello World project, the button you added in the view is the interface. The `showMessage()` method of the `ViewController` class is the implementation. When someone taps the button, it sends a `showMessage` message to `ViewController` by invoking the `showMessage()` method.

What we have just demonstrated is one of the important concepts behind Object Oriented Programming known as *Encapsulation*. The implementation of the `showMessage()` method is hidden from the outside world (i.e. the interface). The Hello World button has no idea how the `showMessage()` method works. All it knows is that it needs to send a message. The `showMessage()` method handles the rest by displaying a "Hello World" message on the screen.

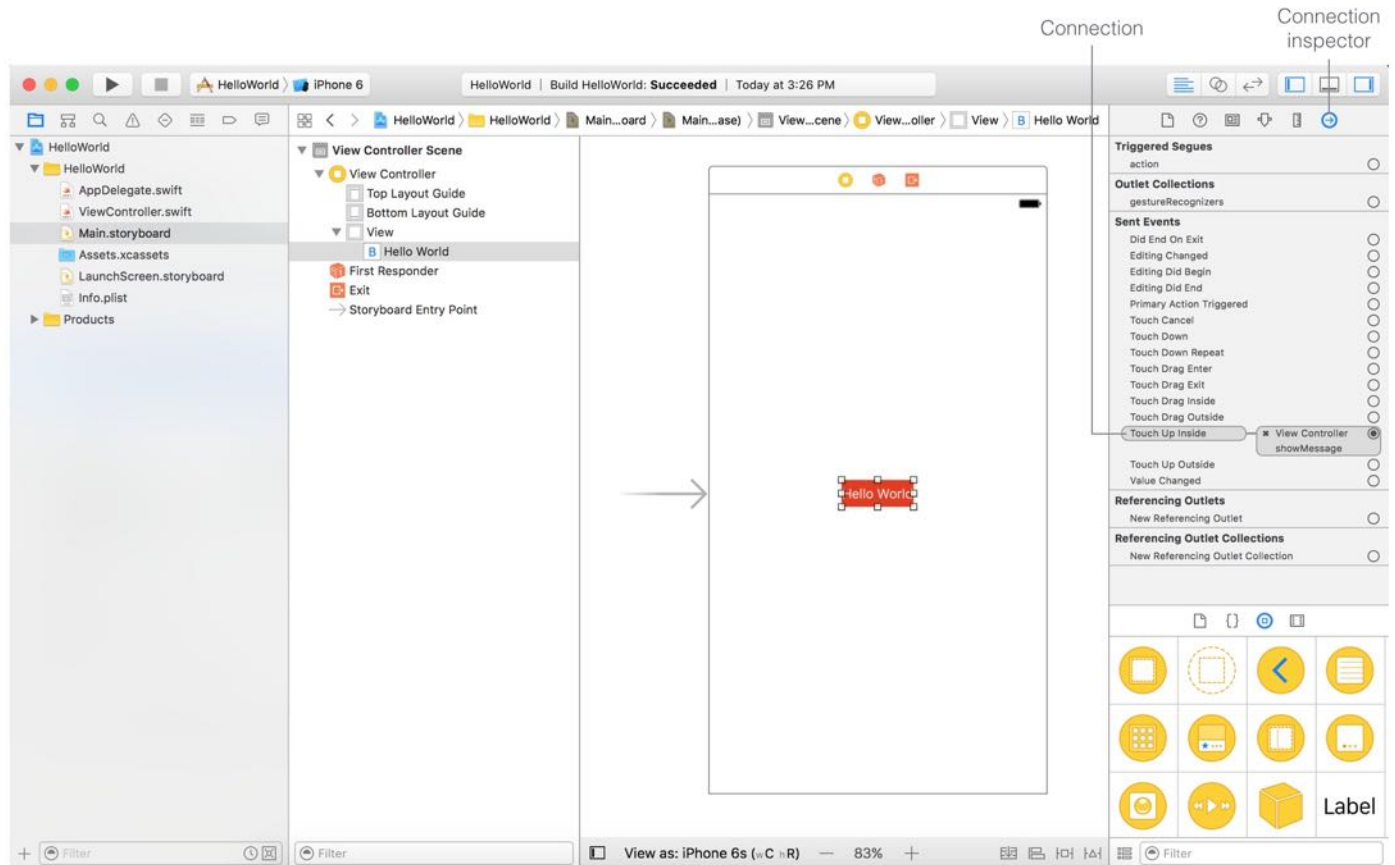
Quick note: Like Objective-C, Swift is an Object-Oriented Programming (OOP) language. Most of the code in an app in some ways deals with objects of some kind. I don't want to scare you away by teaching you the OOP concepts here. Just keep reading. As we go along, you'll learn more about OOP.

Behind the Touch

Now that you should have a basic understanding about how the button in the UI communicates with the code, let's take a look at what actually happens when a user taps the "Hello World" button? How does the "Hello World" button invoke the execution of the `showMessage()` method?

Do you remember how you established the connection between the `Hello World` button and the `showMessage` event in the Interface Builder editor?

Open `Main.storyboard` again and select the "Hello World" button. Click the Connection inspector icon in the Utility area. Under the Sent Events section, you should find a list of available events and its corresponding method to call. As you can see in the below figure, the "Touch Up Inside" event is connected to the `showMessage()` method.



In iOS, apps are based on event-driven programming. Whether it's a system object or UI object, it listens for certain events to determine the flow of the app. For a UI object (e.g. Button), it may listen for a specific touch event. When the event is triggered, the object calls up the preset method that associates with the event.

In the Hello World app, when users lift up the finger inside the button, the "Touch Up Inside" event is triggered. Consequently, it calls up the `showMessage()` method to display the "Hello World" message. We use the "Touch Up Inside" event instead of "Touch Down" because we want to avoid an accidental or false touch. The illustration shown in figure 4-3 sums up the event flow and what I have just described.

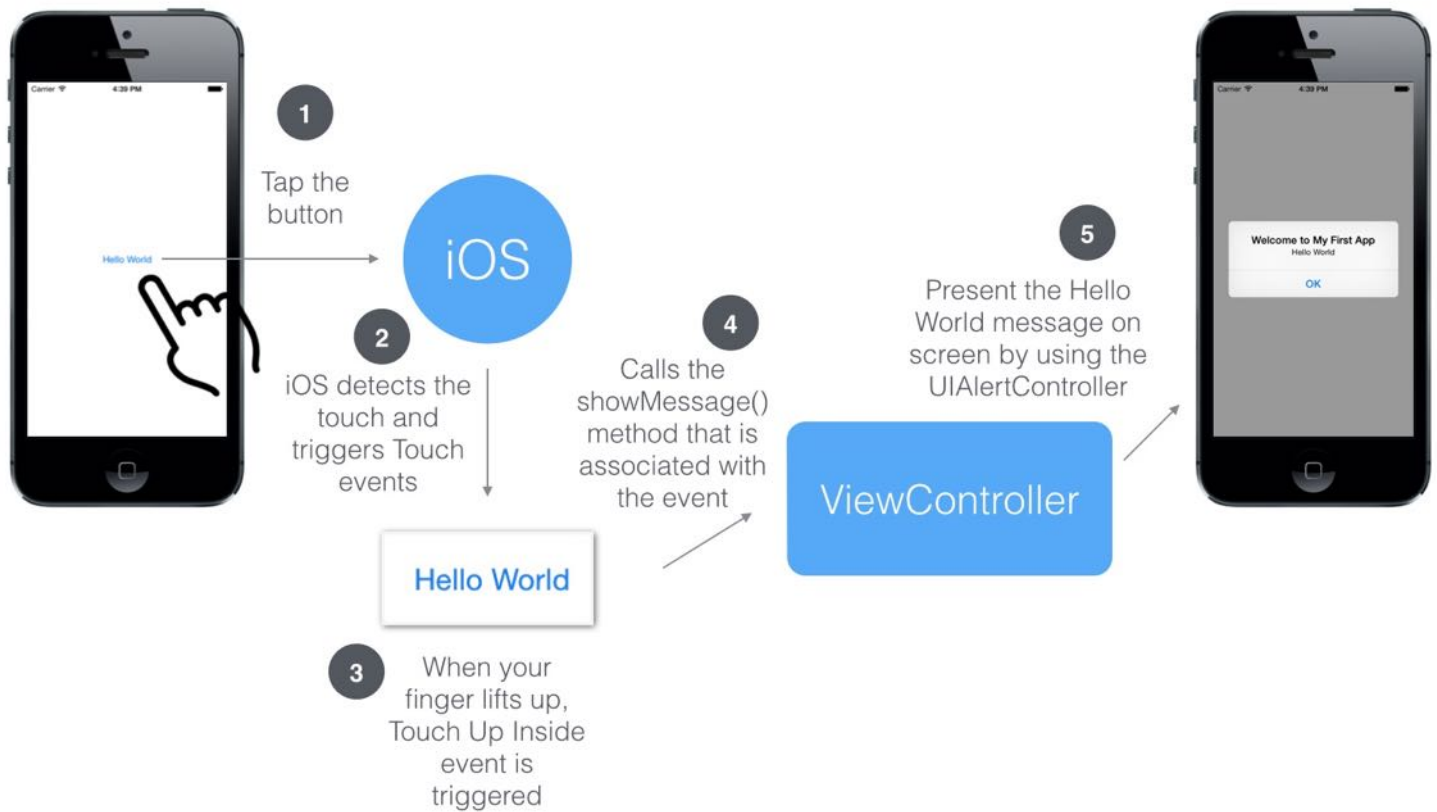


Figure 4-3. Event flow in the Hello World App

Inside the showMessage Method

You should now have a better understanding of iOS programming. But what is the block of code within the `showMessage()` method?

First things first, what is a method? As mentioned before, most of the code in an app in some ways deals with objects of some kinds. Each object provides certain functions and performs specific tasks (e.g. display a message on screen). These functions when expressed in code are known as *methods*.

Now, let us take a closer look at the `showMessage()` method.

Use **import** keyword to access external frameworks (e.g. UIKit)

Use the **func** keyword to declare a method

Method name

Method parameters are defined within parentheses. But in some cases, the method may not have parameters

```
import UIKit

class ViewController: UIViewController {

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

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

    @IBAction func showMessage() {
        let alertController = UIAlertController(title: "Welcome to My First App",
        message: "Hello World", preferredStyle: UIAlertControllerStyle.alert)
        alertController.addAction(UIAlertAction(title: "OK", style: UIAlertActionStyle.default, handler: nil))
        present(alertController, animated: true, completion: nil)
    }
}
```

Expose the method to Interface Builder using **@IBAction**

Methods must be declared in a class. In this example, the class is ViewController.

Quick note: I know it may be a bit hard for you to understand the code. If you're completely new to programming, it'll take some time to get used to Object Oriented Programming. Don't give up, as you'll gain a better understanding of objects, classes and methods as we move along. You can also take a look at the Appendix to learn more about Swift. Or refer to the exercise in the Playgrounds chapter.

In Swift, to declare a method in a class, we use the **func** keyword. Following the **func** keyword is the name of the method. This name identifies the method and makes it easy for the method to be called elsewhere in your code. Optionally, a method can take in parameters as input. The parameters are defined within the parentheses. In our example, however, the method doesn't need any parameter. In this case, we simply write an empty pair of parentheses.

There is one keyword in the method declaration that we haven't discussed. It's the **@IBAction** keyword. This keyword allows you to connect your source code to user interface objects in Interface Builder. When it is inserted in the method declaration, it indicates the method can be exposed to Interface Builder. This is why the **showMessage** event appeared in a pop-over when

you established the connection between the Hello World button and the code in chapter 3. Refer to figure 3-17 if you do not understand what I mean.

Okay, enough for the method declaration. Let's talk about the block of code enclosed in the curly braces. The code block is the actual implementation of the task performed by the method.

If you look at the first line of the code block closely, we make use of `UIAlertController` to construct the Hello World message. What the heck is `UIAlertController`? Where does it come from?

When developing apps in iOS, we don't need to write all functions from scratch. Say, you don't need to learn how to draw the alert box on screen. The iOS SDK, bundled in Xcode, already provides you with tons of built-in functions to make your life easier. These functions are usually known as APIs and organized in the form of *frameworks*. The `UIKit` framework is just one of them, that provides classes and functions to construct and manage your app's user interface. For example, the `UIViewController` and `UIAlertController` are actually come from the `UIKit` framework.

There is one thing to take note. Before you can use any functions from the framework, you have to first import it. This is why you find this statement at the very beginning of

```
ViewController.swift :
```

```
import UIKit
```

Now, let's continue to look into the `showMessage()` method.

The first line of code creates a `UIAlertController` object, and store it in `alertController`. The syntax of constructing an object from a class is very similar to calling a method. You specify the class name, followed by a set of initial values of properties. Here we specify the title, message and style of the alert:

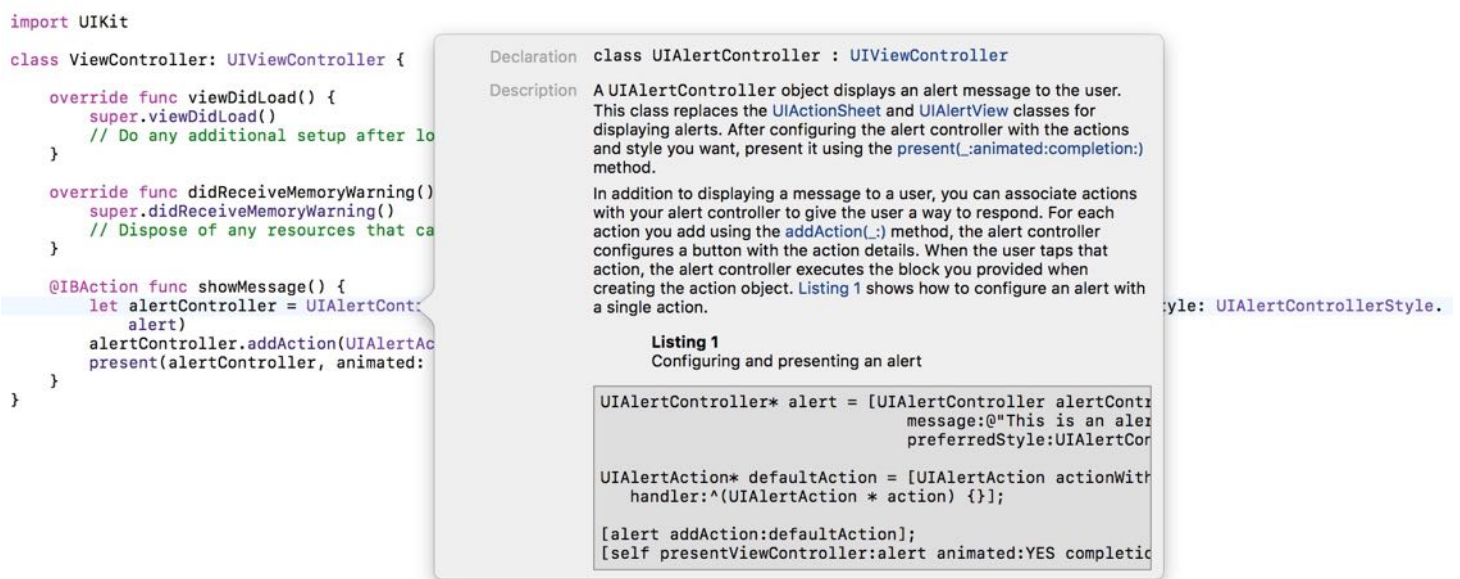
```
let alertController = UIAlertController(title: "Welcome to My First App",  
message: "Hello World", preferredStyle: UIAlertControllerStyle.alert)
```

Right after creating the `UIAlertController` object (i.e. `alertController`), we call the `addAction` method to add an action to the alert so that it displays a "OK" button. When programming in Swift, you call a method by using dot syntax.


```
alertController.addAction(UIAlertAction(title: "OK", style:
UIAlertActionStyle.default, handler: nil))
```

You may wonder how you can find out about the usage and available methods of a class. A simple answer is: *Read the documentation*. You can go up to Google to look up the class. But Xcode provides a convenient way to access the documentation of the iOS SDK.

In Xcode, you can press and hold the option key, point the cursor to the class name (e.g. `UIAlertController`) in your code and then click. A pop-over will appear displaying the class description and sample code. If you need further information, scroll down and click the class reference link. This will bring you to the official documentation of the class.



The screenshot shows a code editor on the left with a Swift class `ViewController` that inherits from `UIViewController`. It has methods `viewDidLoad`, `didReceiveMemoryWarning`, and `showMessage`. The `showMessage` method creates an `UIAlertController` instance, adds an action, and calls `present`. A pop-over window on the right displays the documentation for `UIAlertController`. It includes a declaration, a description, and a listing of sample code for configuring and presenting an alert.

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }

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

    @IBAction func showMessage() {
        let alertController = UIAlertController(
            title: nil, message: nil, preferredStyle: UIAlertControllerStyle.
        )
        alertController.addAction(UIAlertAction(title: nil, style: UIAlertControllerStyle.
        )
    }
}
```

Declaration `class UIAlertController : UIViewController`

Description A UIAlertController object displays an alert message to the user. This class replaces the `UIActionSheet` and `UIAlertView` classes for displaying alerts. After configuring the alert controller with the actions and style you want, present it using the `present(_:animated:completion:)` method.

In addition to displaying a message to a user, you can associate actions with your alert controller to give the user a way to respond. For each action you add using the `addAction(_:)` method, the alert controller configures a button with the action details. When the user taps that action, the alert controller executes the block you provided when creating the action object. Listing 1 shows how to configure an alert with a single action.

Listing 1
Configuring and presenting an alert

```
UIAlertController* alert = [UIAlertController alertControllerWithTitle:@"This is an alert" message:@"This is an alert" preferredStyle:UIAlertControllerStyle.Alert];

UIAlertAction* defaultAction = [UIAlertAction actionWithTitle:@"OK" handler:^(UIAlertAction * action) {}];

[alert addAction:defaultAction];
[self presentViewController:alert animated:YES completion:nil];
```

After the `UIAlertController` object is configured, the last line of the code is for showing the alert message on screen.

```
present(alertController, animated: true, completion: nil)
```

To display the alert, we ask the current view controller to present the `alertController` object with animation.

Sometimes, the above line of code may be written like this:

```
self.present(alertController, animated: true, completion: nil)
```

In Swift, you use the `self` property to refer to the current instance (or object). In most cases, the `self` keyword is optional. So you can omit it.

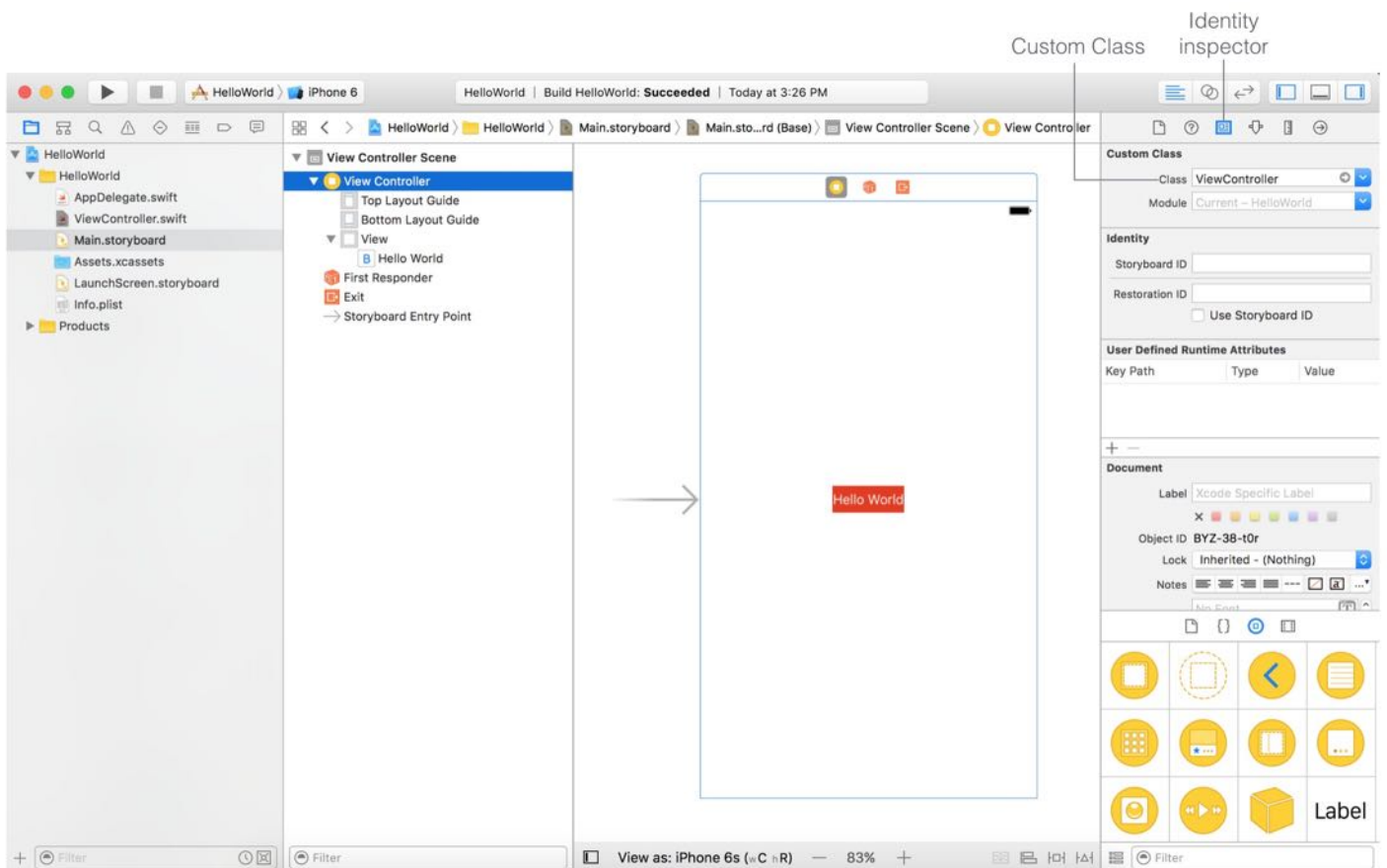
Relationship Between User Interface and Code

How does Xcode know that the View Controller in Interface Builder links up with the `ViewController` class defined in `ViewController.swift` ?

The whole thing seems to be trivial but actually it is not. Do you remember the project template that we chose when creating the Xcode project? It is the *Single View Application* template.

When this project template is used, it automatically creates a default view controller in the Interface Builder editor and generates `ViewController.swift` . On top of that, the view controller is automatically linked with the `ViewController` class defined in the swift file.

Go to `Main.storyboard` , select View Controller in the document outline view. In the Utility area, select the Identity inspector icon and you'll find that `ViewController` is set as the custom class. This is how the objects in Interface Builder associate with the classes in Swift code.

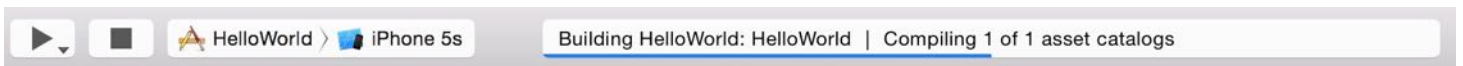


Behind the Scene of the Run Button

One final thing I would like to talk about is the Run button. When you click the Run button, Xcode automatically launches the simulator and runs your app. But what happens behind the scene? As a developer, you have to look at all the pieces.

The entire process can be broken into three phases: *compile*, *package* and *run*.

- **Compile** – You probably think iOS understands Swift code. In reality, iOS only reads machine code. The Swift code is for developer to write and read. To make iOS understand the source code of the app, it has to go through a translation process to translate the Swift code into machine code. This process is referred as "compile". Xcode already comes with a built-in compiler to compile the source code.
- **Package** – Other than source code, an app usually contains resource files such as images, text files, sound files, etc. All these resources are packaged to make up the final app. We used to refer to these two processes as the "build" process.
- **Run** – This actually launches the simulator and loads your app.

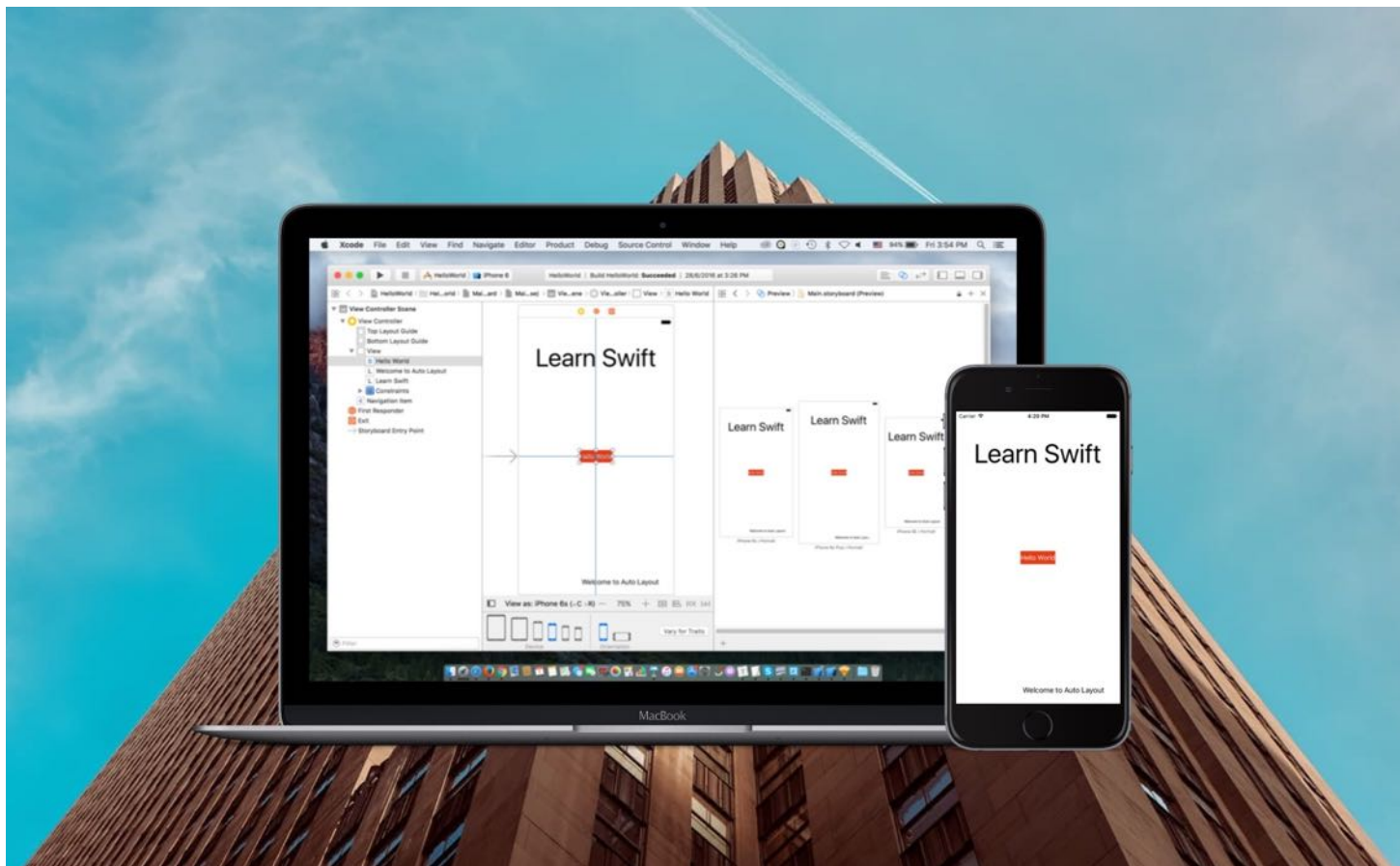


Summary

You should now have a basic understanding of how the Hello World app works. As a beginner without any prior programming experience, it's not easy to understand all the programming concepts we just discussed. No worries. As you write more code and develop a real app in the upcoming chapters, you'll have a better idea about Swift and iOS programming.

Chapter 5

Introduction to Auto Layout



Life is short. Build stuff that matters. – Siqi Chen

Wasn't it fun to create the Hello World app? Before we move onto building a real app, we'll take a look at Auto Layout in this chapter.

Auto layout is a constraint-based layout system. It allows developers to create an adaptive UI that responds appropriately to changes in screen size and device orientation. Some beginners find it hard to learn and avoid using it. Some developers even avoid using it. But believe me, you won't be able to live without it when developing an app today.

With the release of iPhone 6/6s and 6/6s Plus, Apple's iPhones are available in different screen sizes including 3.5-inch, 4-inch, 4.7-inch and 5.5-inch displays. Without using auto layout, it

would be very hard for you to create an app that supports all screen resolutions. And, starting from Xcode 6, it is inevitable to use auto layout to design the user interface. This is why I want to teach you auto layout at the very beginning of this book, rather than jumping right into coding a real app.

In this chapter and the one that follows, I want to help you build a solid foundation on designing an adaptive user interface.

Quick note: Auto layout is not as difficult as some developers thought. Once you understand the basics, you will be able to use auto layout to create complex user interfaces for all types of iOS devices.

Why Auto Layout?

Let me give you an example, and you'll have a better idea why auto layout is needed. Open the HelloWorld project you built in chapter 3. Instead of running the app on iPhone 6/6s simulator, run it using the iPhone 6 Plus or iPhone 5 simulator. You'll end up with the results illustrated in figure 5-1. It turns out that the button isn't centered when running on other iPhone devices, except iPhone 6/6s.

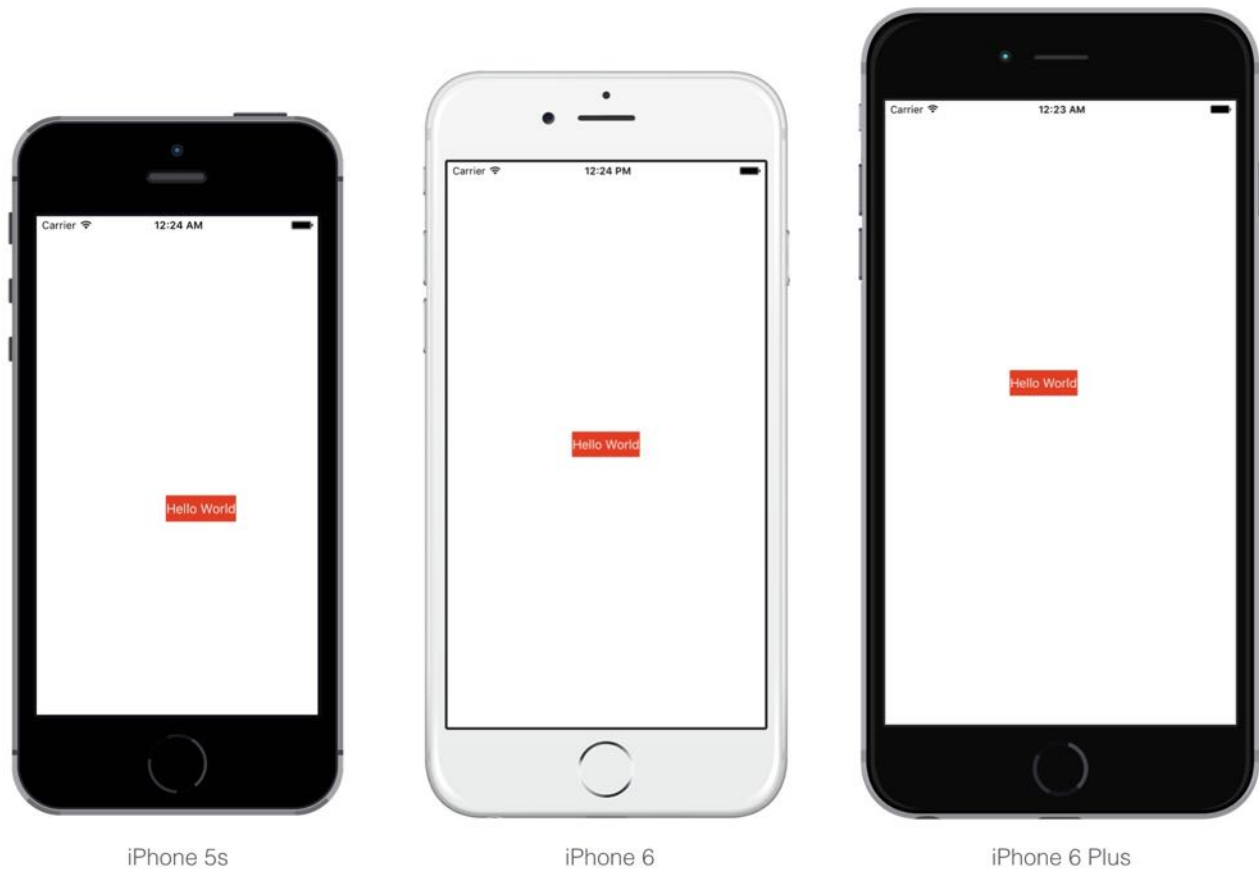


Figure 5-1. The app UI looks different when running on iPhone 5s (4-inch), iPhone 6 (4.7-inch) and iPhone 6 Plus (5.5-inch)

Let's try one more thing.

Click the Stop button and run the app using the iPhone 6/6s simulator. After the simulator launches, go up to the menu and select Hardware > Rotate Left (or Rotate Right) from the menu. This rotates the device to landscape mode. Alternatively, you can press command+left arrow/right arrow to rotate the device sideways. Again, the Hello World button is not centered.

Why? What's wrong with it?

As you know, the iPhone devices have different screen dimensions:

- For iPhone 5/5s, the screen in portrait mode consists of 320 points (or 640 pixels) horizontally and 568 points (or 1136 pixels) vertically.
- For iPhone 6/6s, the screen consists of 375 points (or 750 pixels) horizontally and 667

points (or 1334 pixels) vertically.

- For iPhone 6/6s Plus, the screen consists of 414 points (or 1242 pixels) horizontally and 736 points (or 2208 pixels) vertically.
- For iPhone 4s, the screen consists of 320 points (or 640 pixels) and 480 points (or 960 pixels).

Why Points instead of Pixels?

Back in 2007, Apple introduced the original iPhone with a 3.5-inch display with a resolution of 320x480. That is 320 pixels horizontally and 480 pixels vertically. Apple retained this screen resolution with the succeeding iPhone 3G and iPhone 3GS. Obviously, if you were building an app at that time, one point corresponds to one pixel. Later, Apple introduced iPhone 4 with retina display. The screen resolution was doubled to 640x960 pixels. So one point corresponds to two pixels for retina display.

The point system makes our developers' lives easier. No matter how the screen resolution is changed (say, the resolution is doubled again to 1280x1920 pixels), we still deal with with points and the base resolution (i.e. 320x480 for iPhone 4/4s or 320x568 for iPhone 5/5s). The translation between points and pixels is handled by iOS.

Without using auto layout, the position of the button we lay out in the storyboard is fixed. In other words, we "hard-code" the frame origin of the button. In our example, the "Hello World" button's frame origin is set to (147, 318). Therefore, whether you're using a 3.5-inch or 4-inch or 5.5-inch simulator, iOS draws the button in the specified position. Figure 5-2 illustrates the frame origin on different devices. This explains why the "Hello World" button can only be centered on iPhone 6/6s, and it is shifted away from the screen center on other iOS devices, as well as, in landscape orientation.

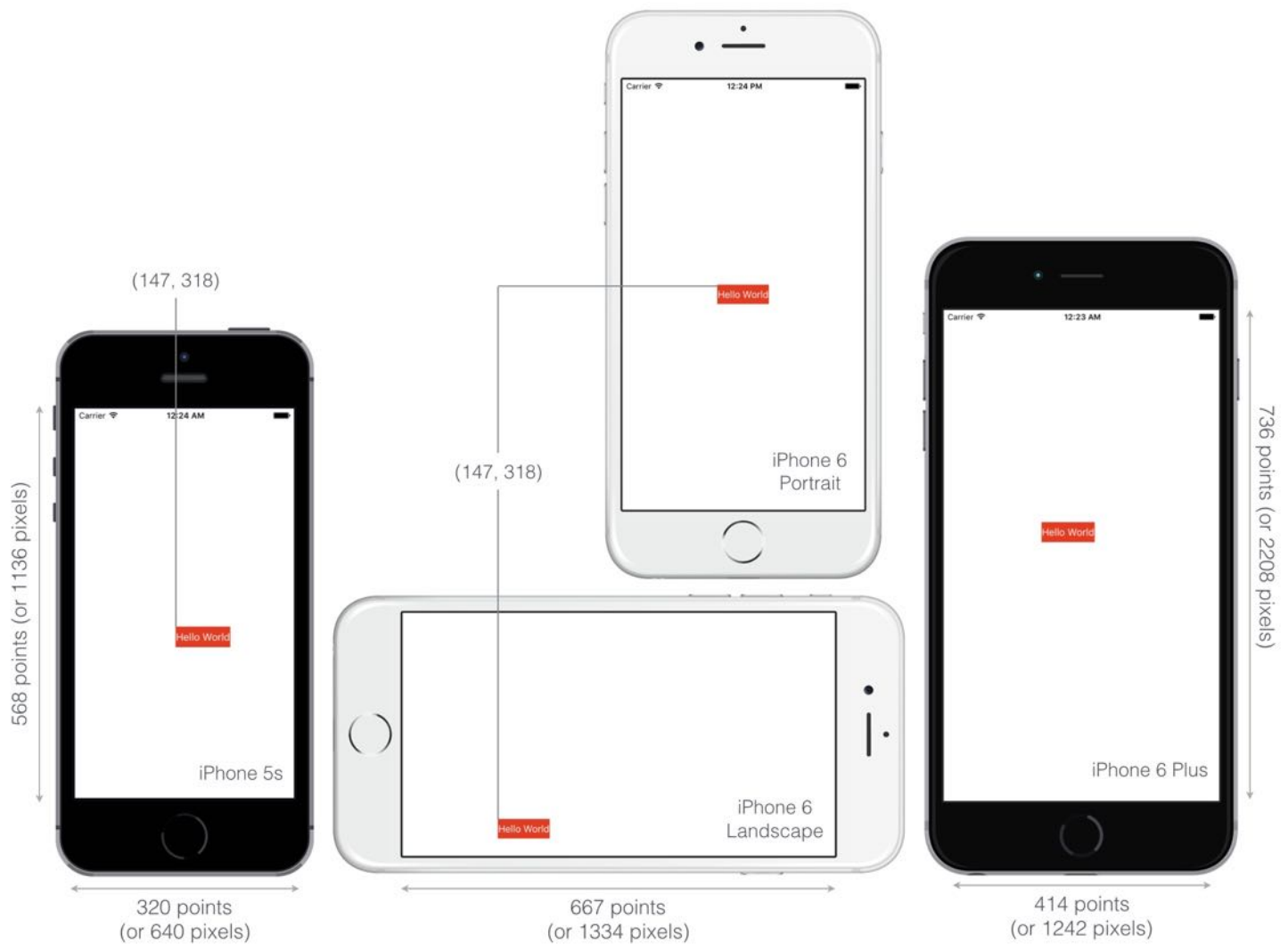


Figure 5-2. How the button is displayed on iPhone 6 Plus, iPhone 6 and iPhone 5s

Obviously, we want the app to look good on all iPhone models, and in both portrait & landscape orientation. This is why we have to learn auto layout. It's the answer to the layout issues, that we have just talked about.

Auto Layout is All About Constraints

As mentioned before, auto layout is a constraint-based layout system. It allows developers to create an adaptive UI that responds appropriately to changes in screen size and device orientation. Okay, it sounds good. But what does the term "constraint-based layout" mean? Let me put it in a more descriptive way. Consider the "Hello World" button again, how do you describe its position if you want to place the button at the center of the view? You would

probably describe it like this:

The button should be centered both horizontally and vertically, regardless of the screen resolution and orientation.

Here you actually define two constraints:

- center horizontally
- center vertically

These constraints express rules for the layout of the button in the interface.

Auto layout is all about constraints. While we describe the constraints in words, the constraints in auto layout are expressed in mathematical form. For example, if you're defining the position of a button, you might want to say "the left edge should be 30 points from the left edge of its containing view." This translates to `button.left = (container.left + 30)`.

Fortunately, we do not need to deal with the formulas. All you need to know is how to express the constraints descriptively and use Interface Builder to create them.

Okay, that's quite enough for the auto layout theory. Now let's see how to define layout constraints in Interface Builder to center the "Hello World" button.

Live Previewing in Interface Builder

First, open `Main.storyboard` of your HelloWorld project (or download it from <http://www.appcoda.com/resources/swift3/HelloWorld.zip>). Before we add the layout constraints to the user interface, let me introduce a handy feature in Xcode 8.

Instead of viewing the app UI in simulators, Xcode 8 provides a configuration bar in Interface Builder for developers to live preview the user interface.

By default, Interface Builder is set to preview the UI on iPhone 6s. To see how your app looks on other devices, click `View as: iPhone 6s` button to reveal the configuration bar and then choose your preferred iPhone/iPad devices to test. You can also alter the device's orientation to see how it affects your app's UI. Figure 5-3 shows a live preview of the Hello World app on

iPhone 6s in landscape orientation.

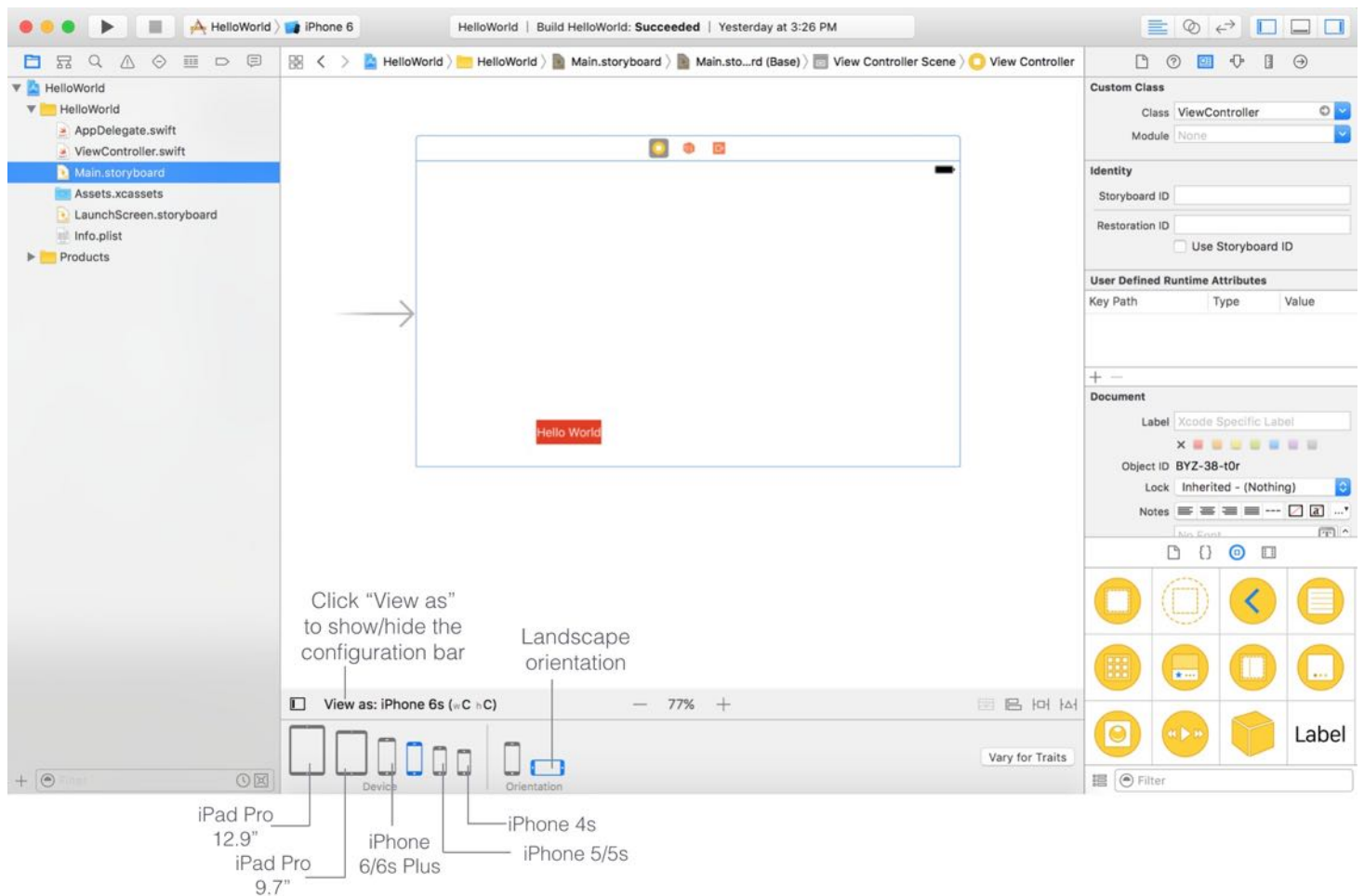


Figure 5-3. Live preview using Xcode 8's new configuration bar

The configuration bar is a great feature in Xcode 8. Take some time to play around with it.

Quick note: You may wonder what "(w C h C)" means. Just forget about it right now and focus on learning auto layout. I will discuss about it in the later chapter.

Using Auto Layout to Center the Button

Now let's continue to talk about auto layout. Xcode provides two ways to define auto layout constraints:

1. Auto layout bar

2. Control-drag

We'll demonstrate both approaches in this chapter. First, we begin with the auto layout bar. At the bottom-right corner of the Interface Builder editor, you should find four buttons. These buttons are from the layout bar. You can use them to define various types of layout constraints.



Each button has its own function:

- **Align** – Create alignment constraints, such as aligning the left edges of two views.
- **Pin** – Create spacing constraints, such as defining the width of a UI control.
- **Issues** – Resolve layout issues.
- **Stack** – Embed views into a stack view. Stack view is a new feature since Xcode 7. We will further discuss about it in the next chapter.

As discussed earlier, to center the "Hello World" button, you have to define two constraints: *center horizontally* and *center vertically*. Both constraints are with respect to the view.

To create the constraints, we will use the Align function. First, select the button in Interface Builder and then click the *Align* icon in the layout bar. In the pop-over menu, check both "Horizontal in container" and "Vertically in container" options. Then click the "Add 2 Constraints" button.

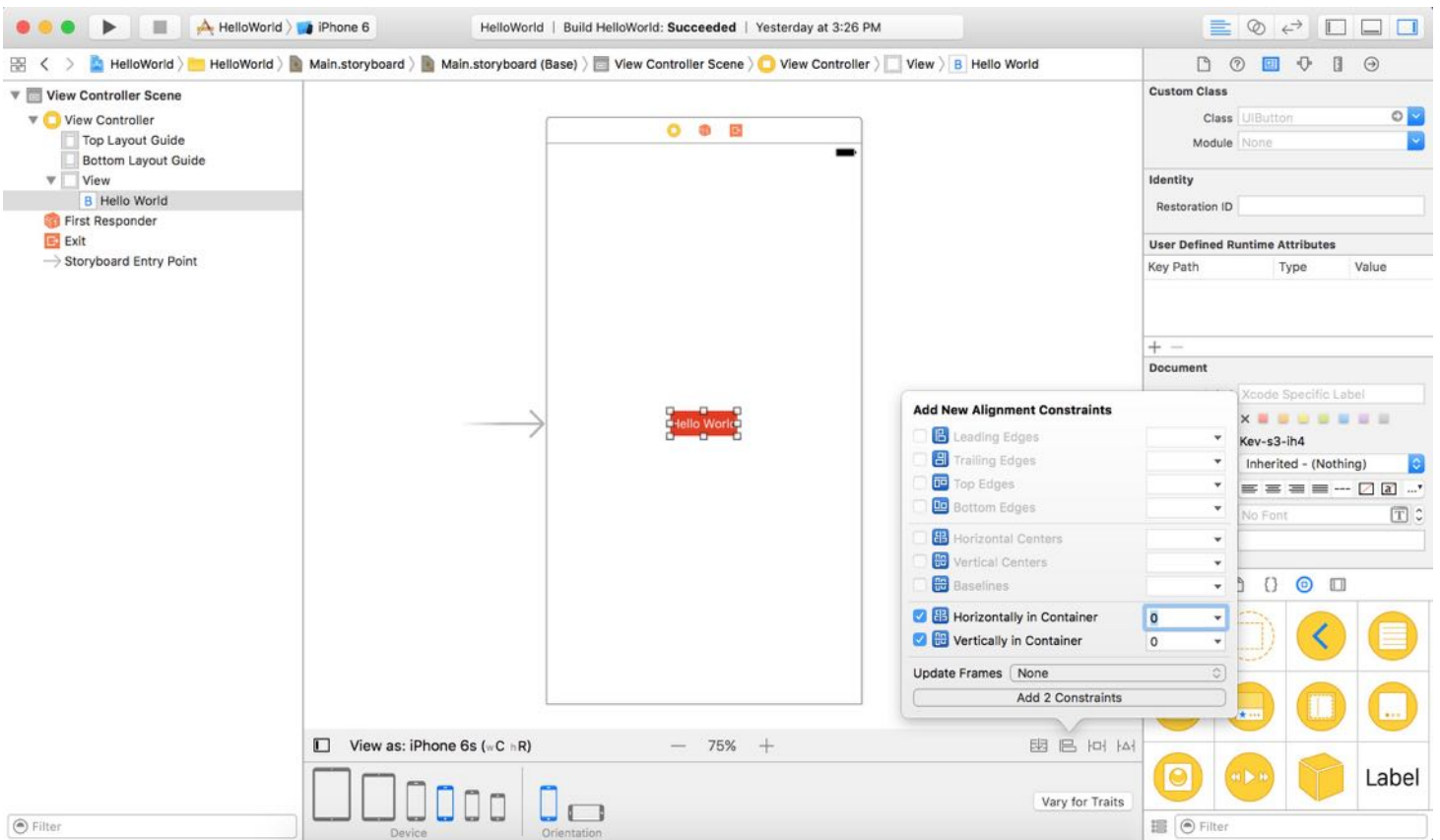


Figure 5-5. Adding constraints using Align button

Quick tip: You can press `command+0` to hide the project navigator. This will free up more screen space for you to work on the app design.

You should now see a set of constraint lines. If you expand the *Constraints* option in the document outline view, you will find two new constraints for the button. These constraints ensure the button is always positioned at the center of the view. Alternatively, you can view these constraints in the Size inspector.

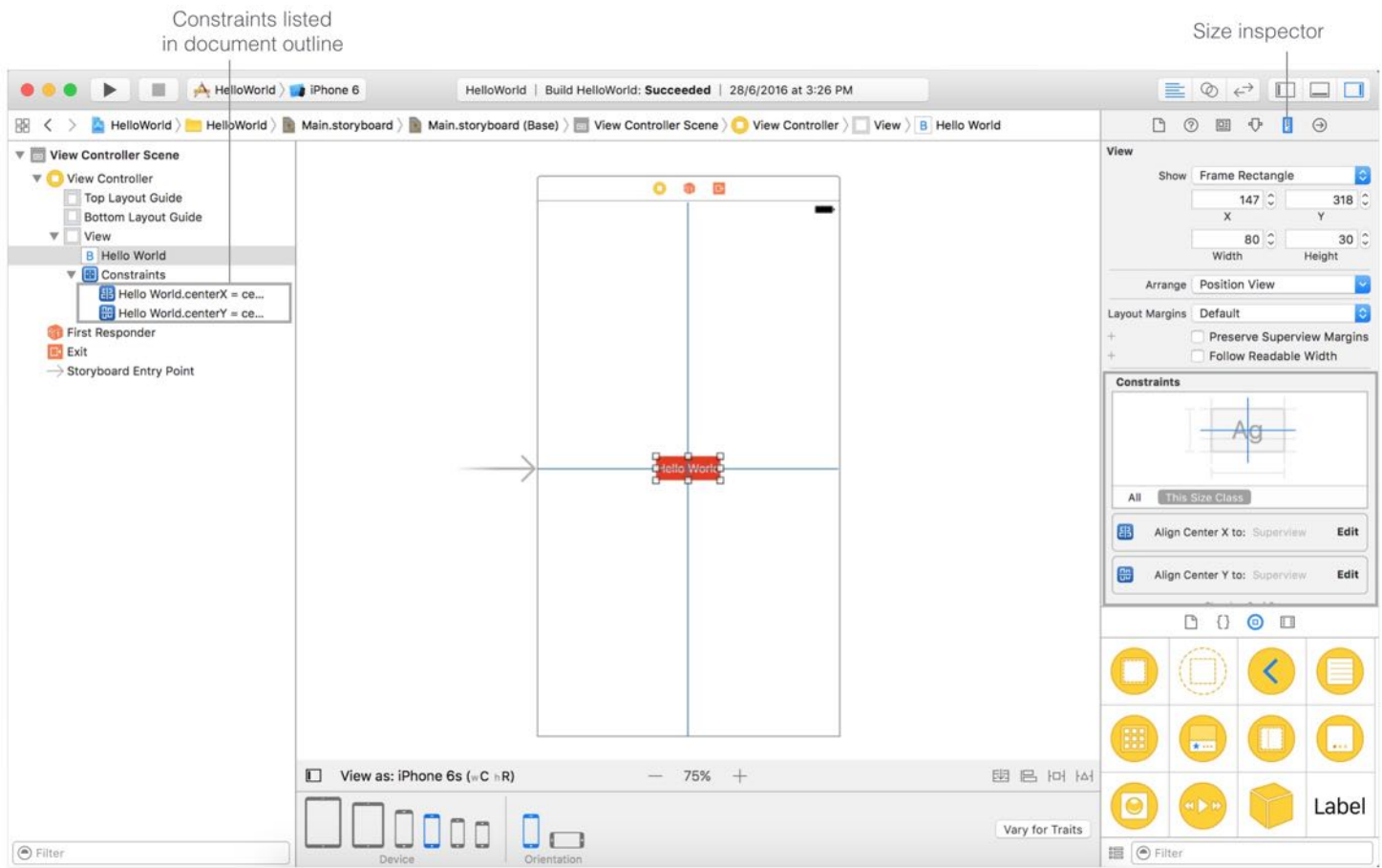


Figure 5-6. View the constraints in Document Outline and Size Inspector

Quick note: When your view layout is being configured correctly and there is no ambiguity, the constraint lines are in blue.

Okay, you're ready to test the app. You can click the Run button to launch the app on iPhone 6/6s Plus (or iPhone 4s). Alternatively, just choose another device or change the device's orientation using the configuration bar to verify the layout. The button should be centered perfectly, regardless of screen size and orientation.

Resolving Layout Constraint Issues

The layout constraints that we have just set are perfect. But that is not always the case. Xcode is intelligent enough to detect any constraint issues.

Try to drag the Hello World button to the lower-left part of the screen. Xcode immediately

detects some layout issues and the corresponding constraint lines turns orange that indicates a misplaced item.

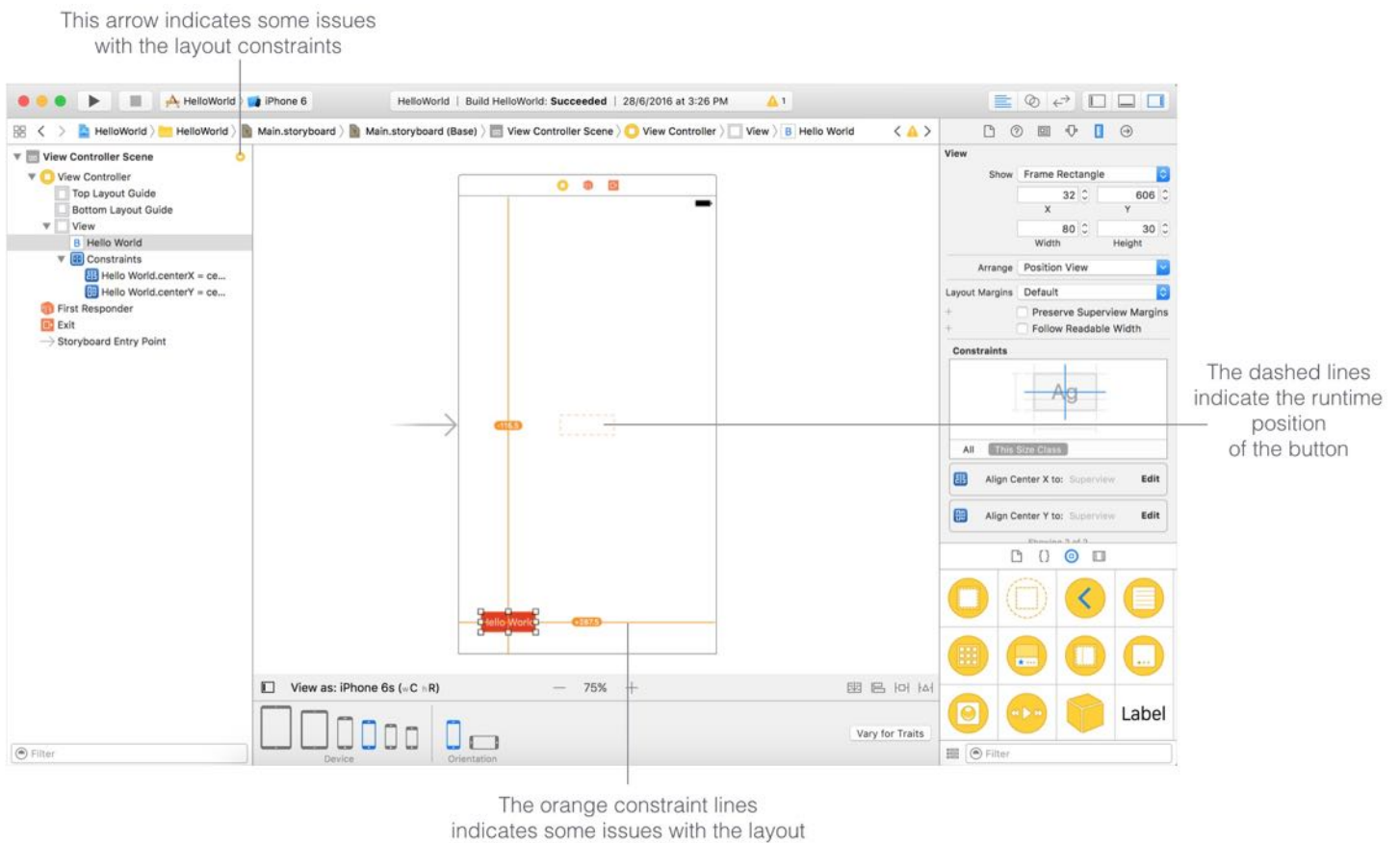
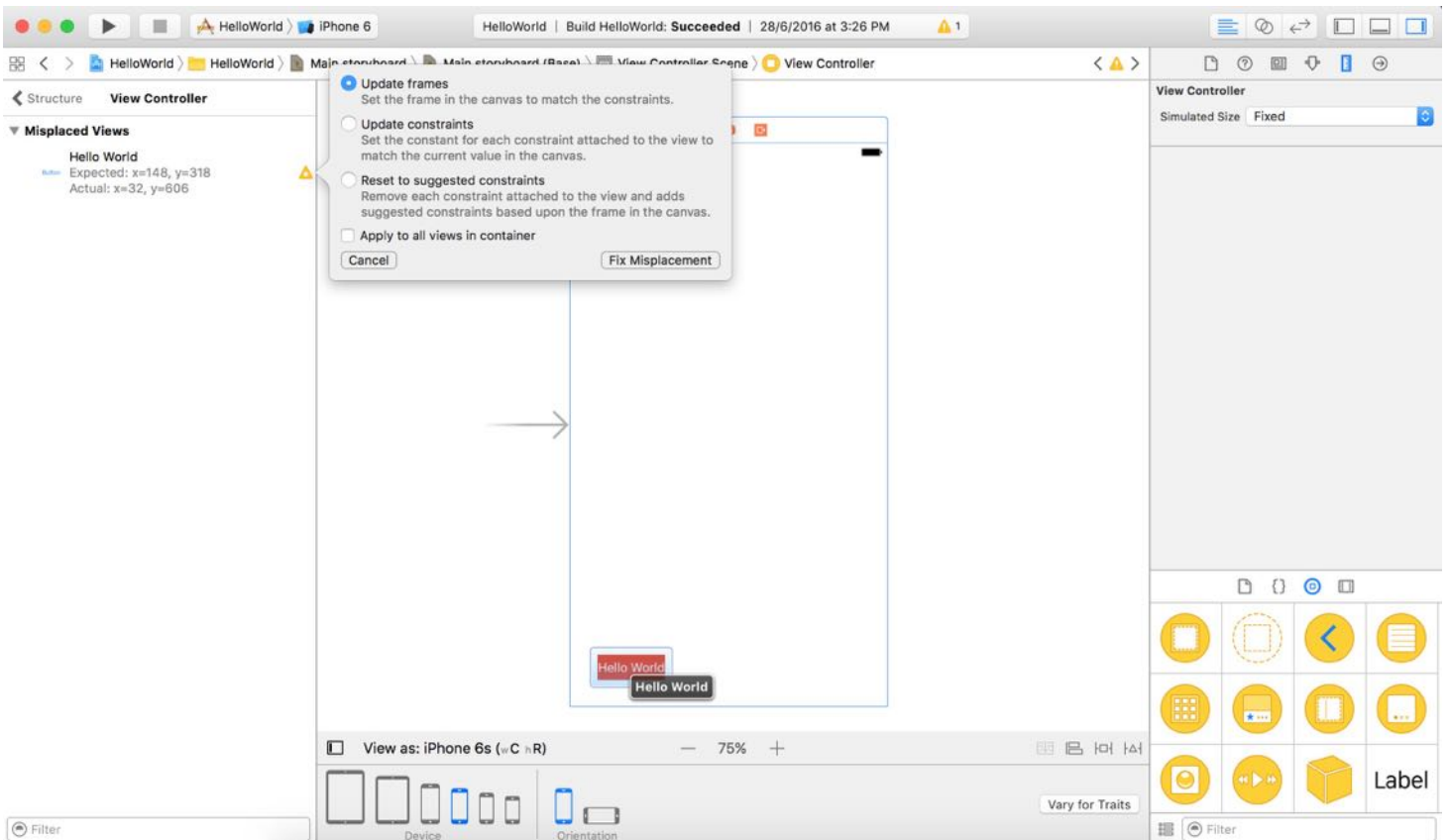


Figure 5-7. Interface Builder uses orange/red lines to indicate Auto Layout issues

Auto layout issues occur when you create ambiguous or conflicting constraints. Here we said the button should be vertically and horizontally centered in the container (i.e. the view). However, the button is now placed at the lower-left corner of the view. Interface Builder found this confusing, therefore it uses orange lines to indicate the layout issues.

When there is any layout issue, the Document Outline view displays a disclosure arrow (red/orange). Now click the disclosure arrow to see a list of the issues. Interface Builder is smart enough to resolve the layout issues for us. Click the indicator icon next to the issue and a pop-over shows you a number of solutions. In this case, select the "Update Frame" option and click "Fix Misplacement" button. The button will then be moved to the center of the view.



This layout issue was triggered manually. I just wanted to demonstrate how to find the issues and fix them. As you go through the exercises in the later chapters, you will probably face a similar layout issue. You should know how to resolve layout issues easily and quickly.

An Alternative Way to Preview Storyboards

While you can use the configuration bar to live preview your app UI, Xcode provides an alternate Preview feature for developers to preview the user interface on different devices simultaneously.

In Interface Builder, open the Assistant pop-up menu > Preview (1). Now press and hold `option` key, then click Main.storyboard (Preview). You can refer to figure 5-9 for the steps.

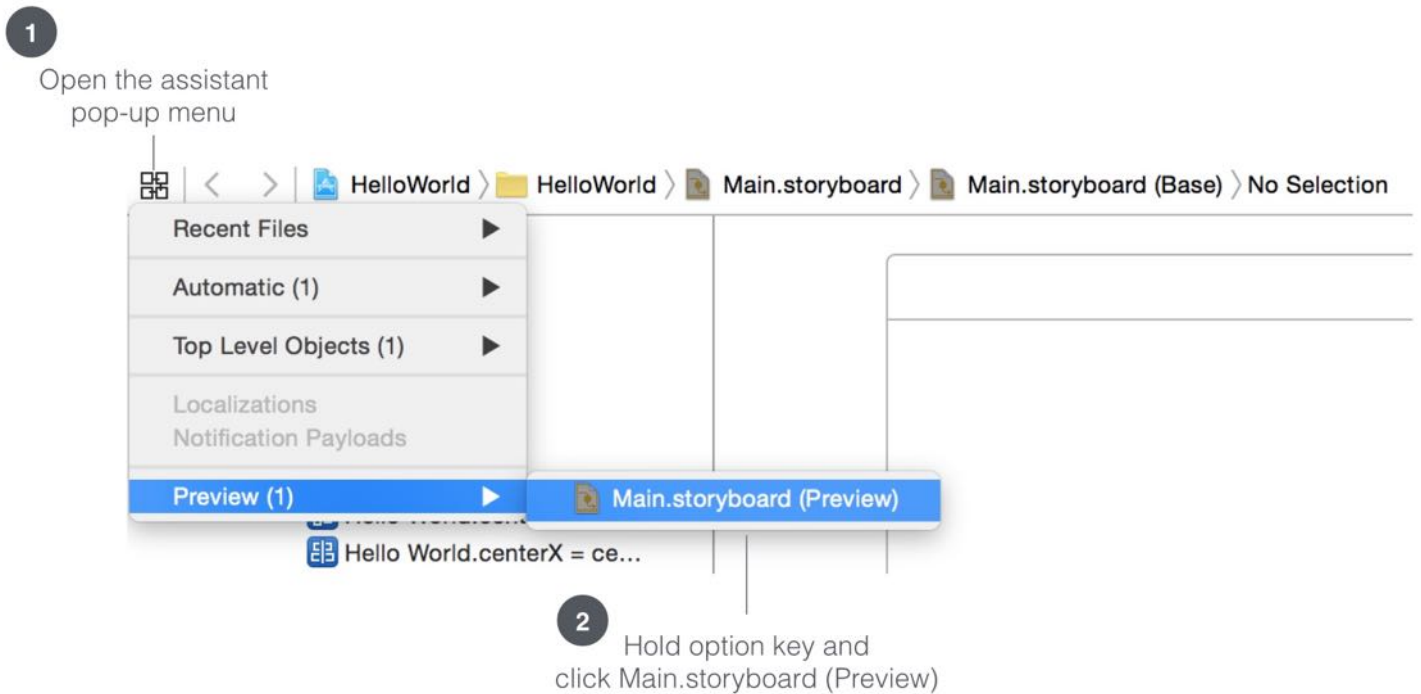
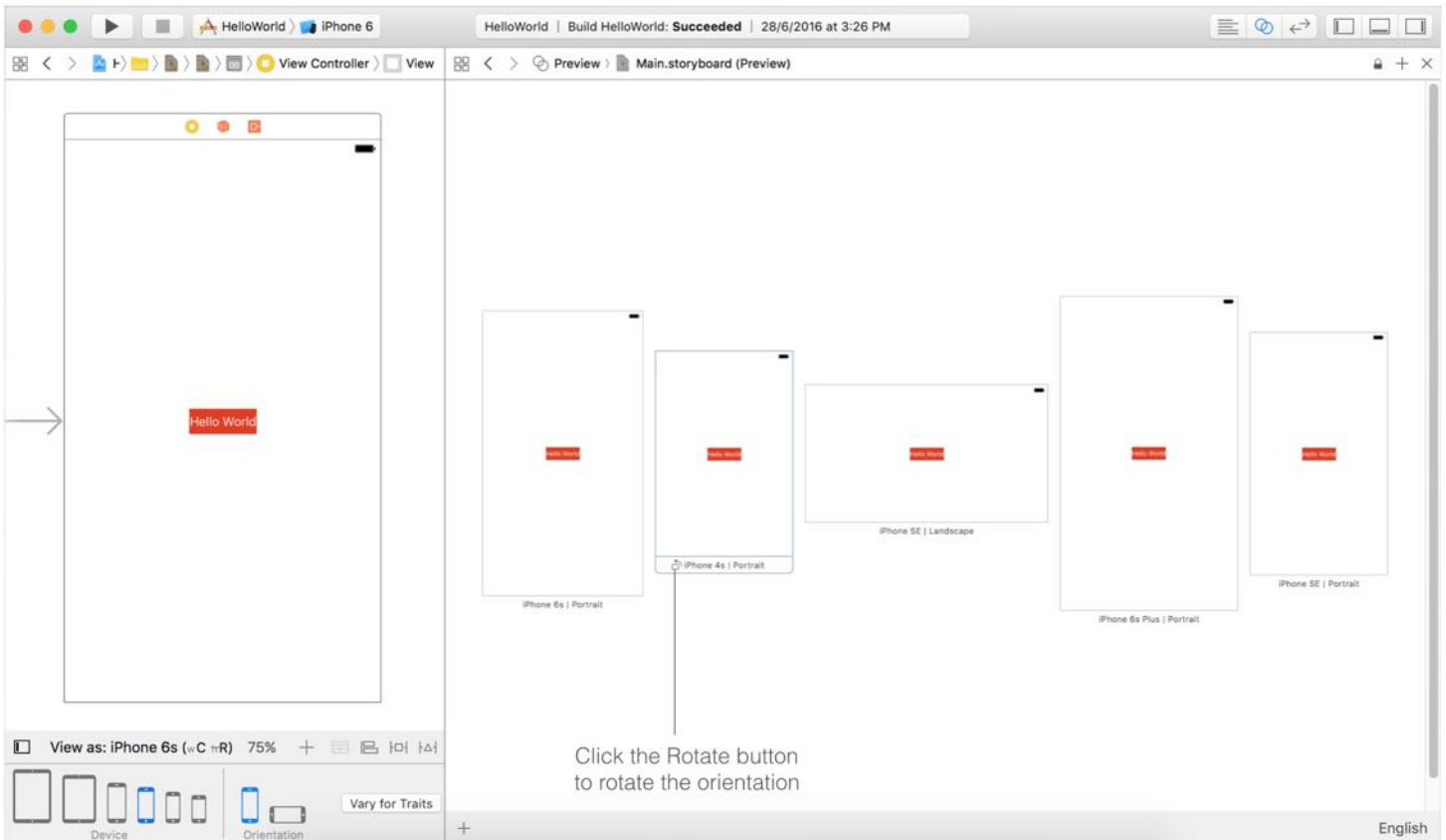


Figure 5-9. Assistant Pop-up Menu

Xcode will display a preview of your app's UI in the assistant editor. By default, it shows you the preview on a iPhone 6s device. You can click the + button at the lower-left corner of the assistant editor to add other iOS devices (e.g. iPhone 4s/SE) for preview. If you want to see how the screen looks like in landscape orientation, simply click the rotate button. The Preview feature is extremely useful for designing your app's user interface. You can make changes to the storyboard (say, adding another button to the view) and see how the UI looks on the chosen devices all at once.



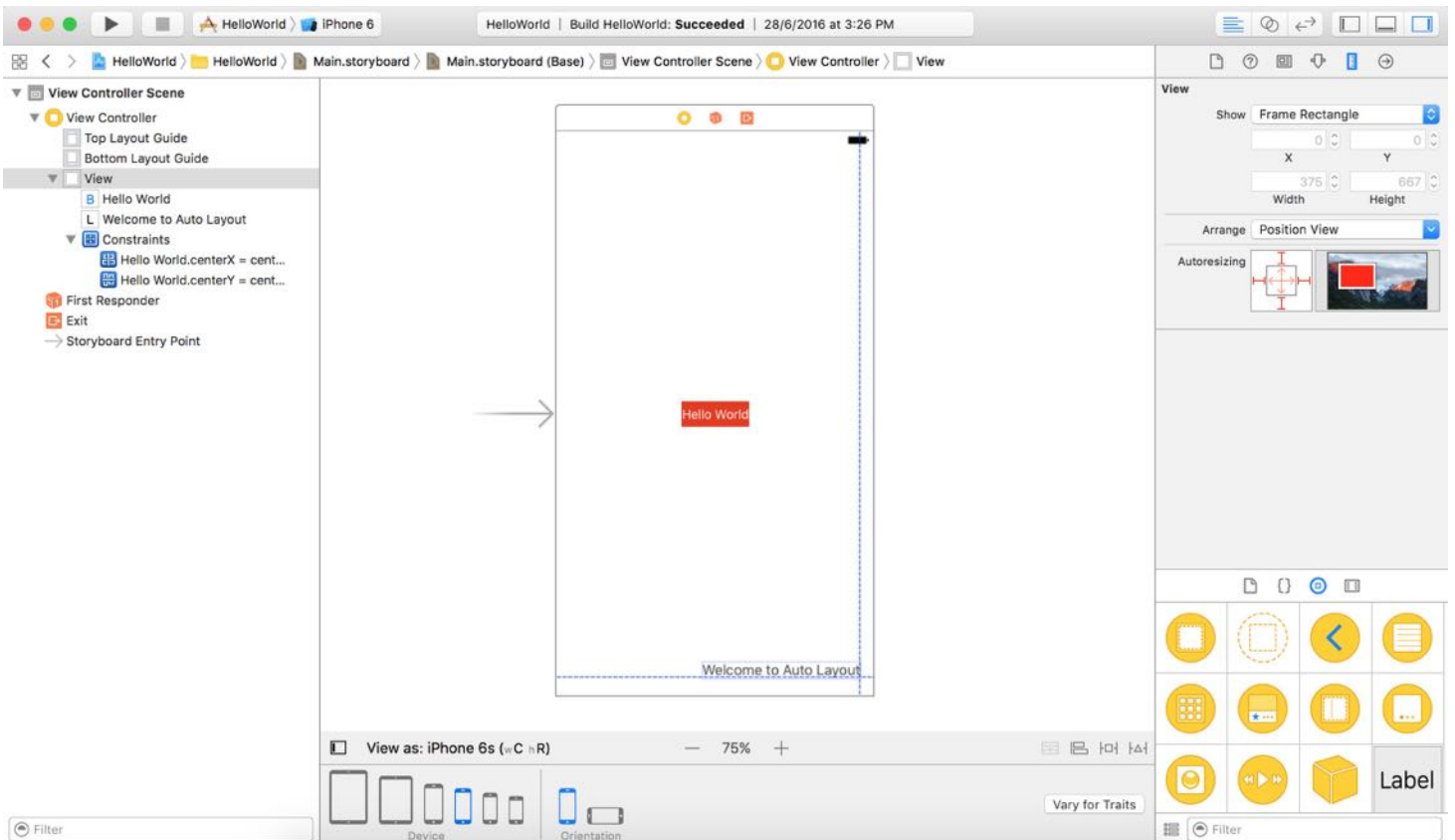
If you want to free up some more screen space for the preview pane, hold both command and option keys and then press o to hide the Utility area.

Quick tip: When you add more devices in the preview assistant, Xcode may not be able to fit the preview of all device sizes into the screen at the same time. If you're using a trackpad, you can scroll through the preview by swiping left or right with two fingers. What if you're still using a mouse with a scroll wheel? Simply hold the shift key to scroll horizontally.

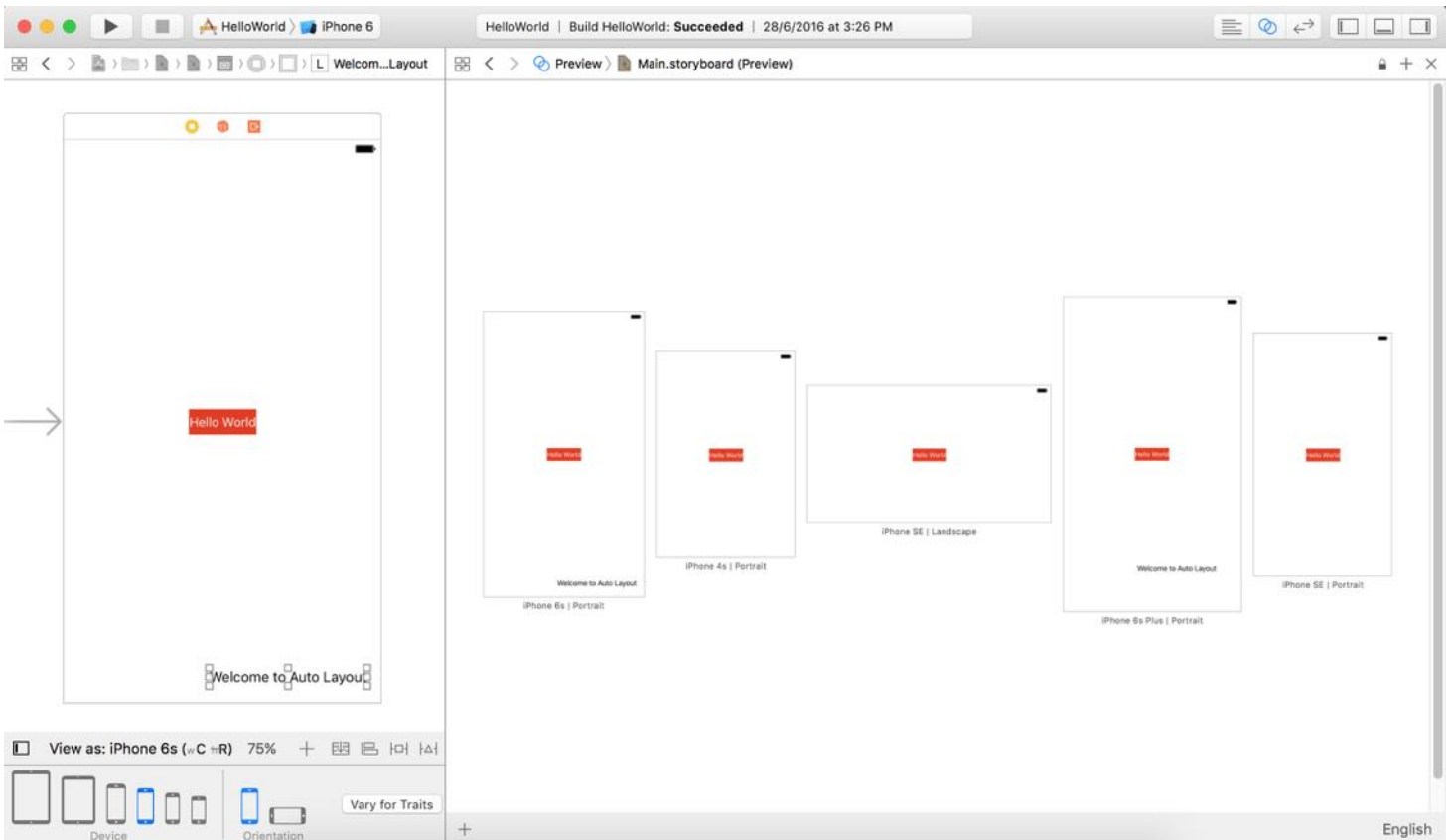
Adding a Label

Now that you have some idea about auto layout and the preview feature, let's add a label to the lower-right part of the view and see how to define the layout constraints for the label. Labels in iOS are usually used for displaying simple text and messages.

In the Interface Builder editor, drag a label from the Object library and place it near the lower-right corner of the view. Double-click the label and change it to "Welcome to Auto Layout" or whatever title you want.



If you opened the preview assistant again, you should see the UI change immediately (see figure 5-12). Without defining any layout constraints for the label, you are not able to display the label on all iPhone devices except iPhone 6s and 6s Plus.



How can you resolve this issue? Obviously, we need to setup a couple of constraints to make it work properly. The question is: *what constraints should we add?*

Let's try to describe the requirement of the label in words. You probably describe it like this:

The label should be placed at the lower-right corner of the view.

That's okay, but not precise enough. A more precise way to describe the location of the label is like this:

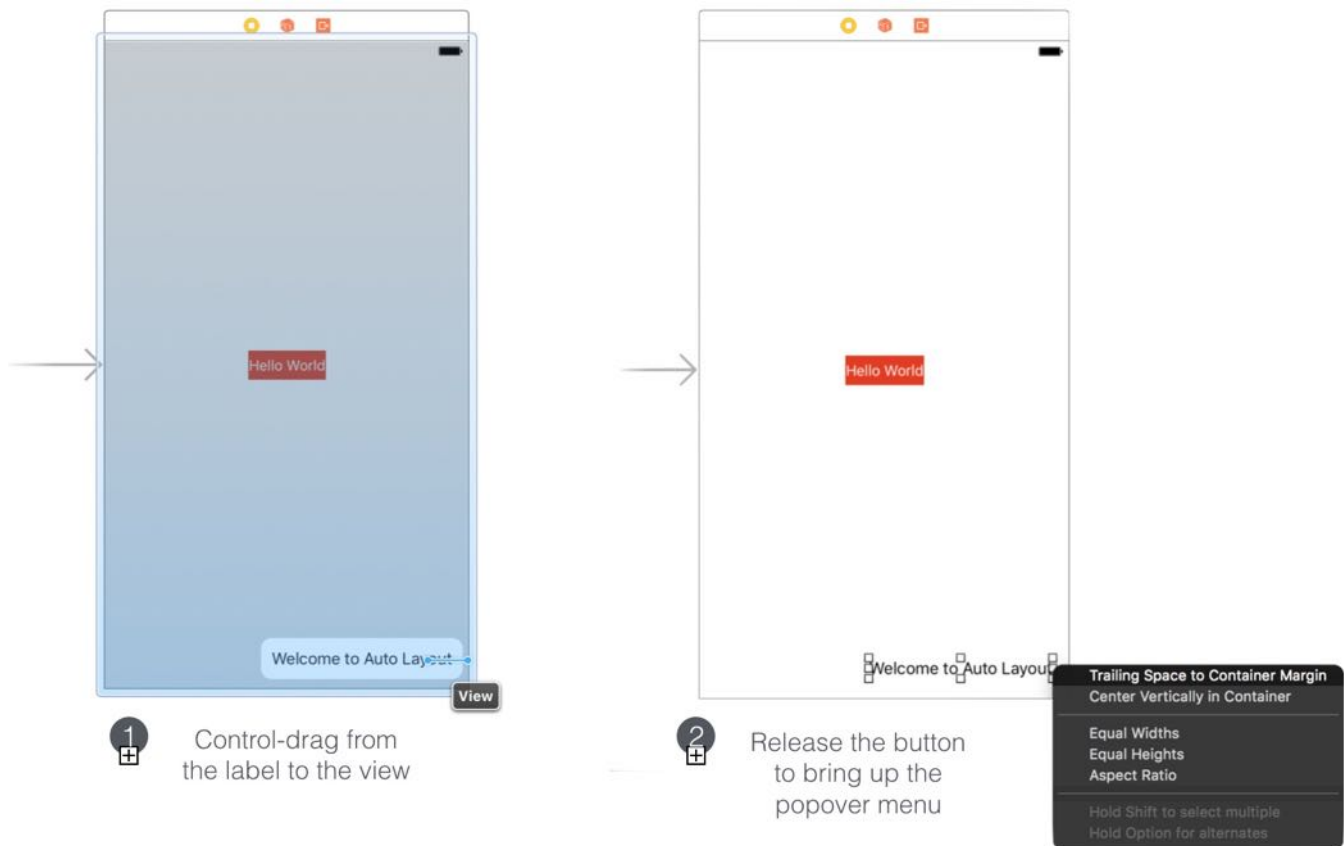
The label is located 0 points away from the right margin of the view and 20 points away from the bottom of the view.

This is much better. When you describe the position of an item precisely, you can easily come up with the layout constraints. Here, the constraints of the label are:

1. The label is 0 points away from the right margin of the view.
2. The label is 20 points away from the bottom of the view.

In auto layout, we refer this kind of constraints as spacing constraints. To create these spacing constraints, you can use the Pin button of the layout button. But this time we'll use the *Control-drag* approach to apply auto layout. In Interface Builder, you can control-drag from an item to itself or to another item along the axis for which you want to add constraints.

To add the first spacing constraint, hold the `control` key and drag from the label to the right until the view becomes highlighted in blue. Now release the button, you'll see a pop-over menu showing a list of constraint options. Select "Trailing space to container margin" to add a spacing constraint from the label to the view's right margin.



In the document outline view, you should see the new constraint. Interface Builder now displays constraint lines in red indicating that there are some missing constraints. That's normal as we haven't defined the second constraint.

Now control-drag from the label to the bottom of the view. Release the button and select "Vertical Spacing to Bottom Layout Guide" in the shortcut menu. This creates a spacing constraint from the label to the bottom layout guide of the view.



Figure 5-14. Using Control-drag to add the second constraint

Once you added the two constraints, all constraint lines should be in solid blue. When you preview the UI or run the app in simulator, the label should display properly on all screen sizes, and even in landscape mode.

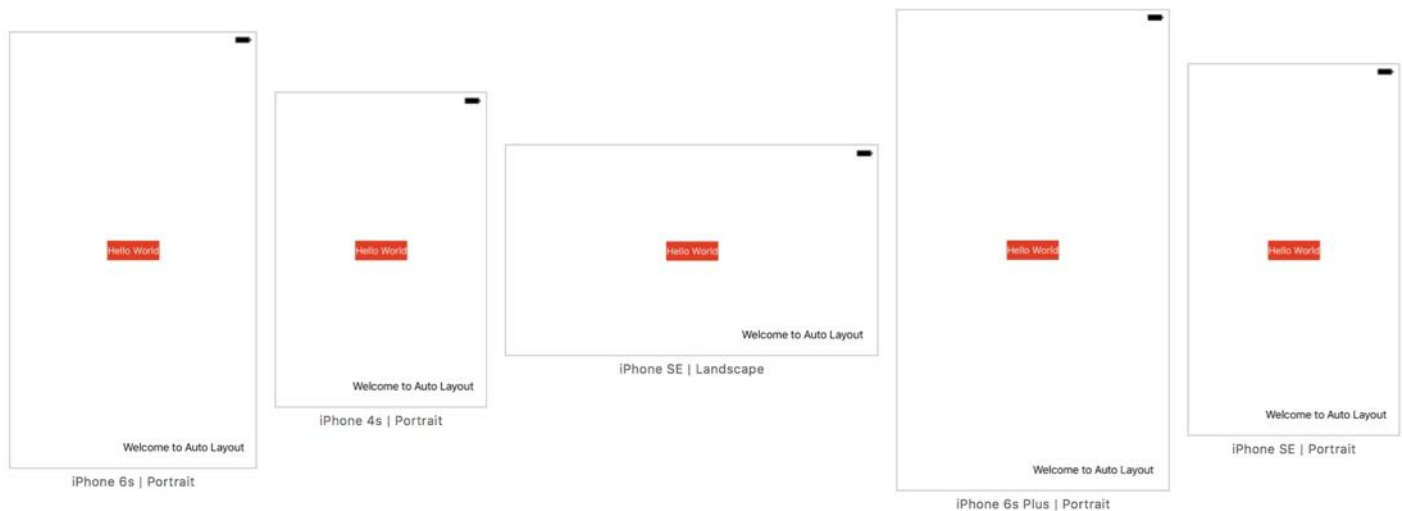


Figure 5-15. The UI now supports all screen sizes

Top and Bottom Layout Guide

You may wonder what the Bottom Layout Guide means. Generally, the Bottom Layout Guide refers to the bottom of the view, like the one shown in the example. Sometimes, the Bottom Layout Guide varies. For example, if there is a tab bar, the Bottom Layout Guide will refer to

the top of tab bar.

For the Top Layout Guide, it sits at 20 points (which is the height of the status bar) from the top of the view.

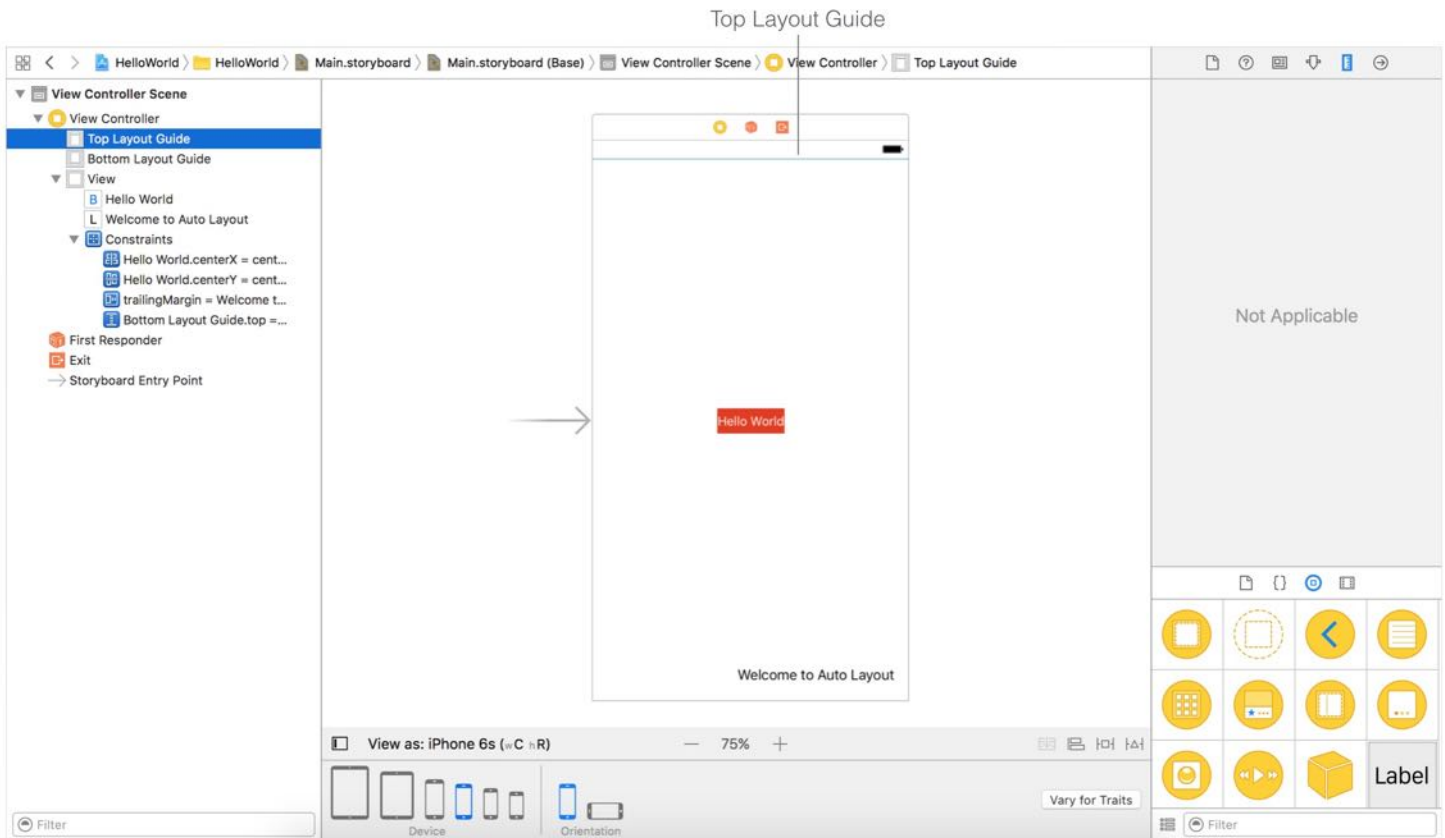


Figure 5-16. Top layout guide

These layout guides are particularly useful for defining layout constraints because they automatically varies its position when the view's layout is changed. For example, if the view contains a navigation bar, the top layout guide sits below the navigation bar (see figure 5-17). Therefore, as long as the UI object is constrained relative to the top/bottom layout guide, your interface will layout correctly even if you add a navigation or tab bar to the interface.



Figure 5-17. Top layout guide with navigation bar

Editing Constraints

The "Welcome to Auto Layout" label is now located 0 points away from the right margin of the view. What if you want to add some space between the label and the right margin of the view? Interface Builder provides a convenient way to edit the constant of a constraint.

You can simply choose the constraint in the document outline view. Here, you should select "trailingMargin" constraint. In the Attributes inspector, you can find the properties of this constraint including relation, constant, and priority. The constant is now set to 0. You can change it to 20 to add some extra space.

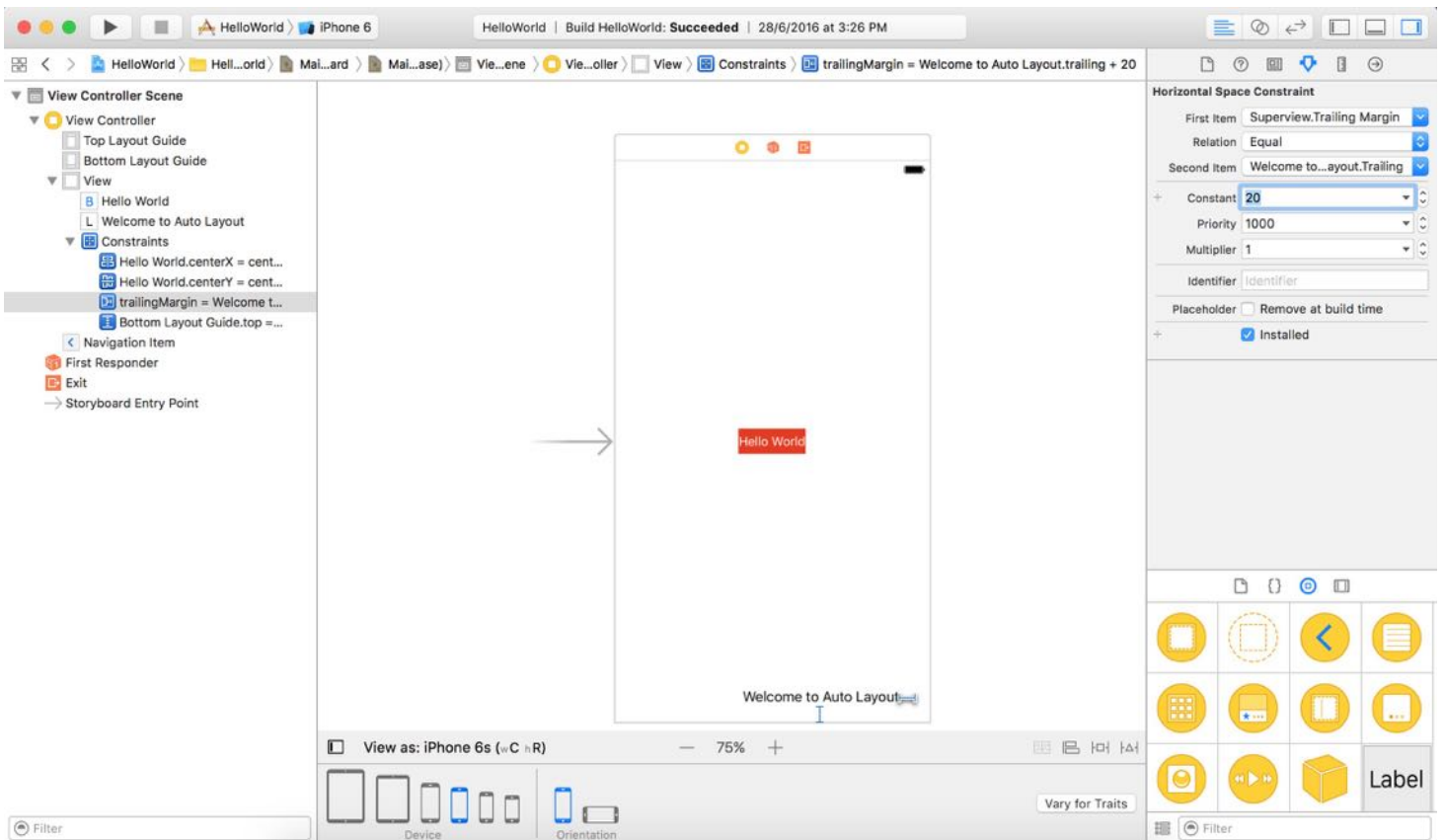


Figure 5-18. Editing a constraint

Your Exercise

By now, I hope you should have some basic ideas about how to lay out your app UI and make it fit for all screen sizes. Let's have a simple exercise before moving on to the next chapter. All I need you to do is add another label named *Learn Swift* to the view with the following constraints:

- The new label should be 40 points away from the top layout guide.
- The new label should be centered horizontally.

Optionally, you can increase the font size of the label to 30 points. To change the font size, you just open the Attributes inspector and edit the *Font* option. Your end result should look like this:

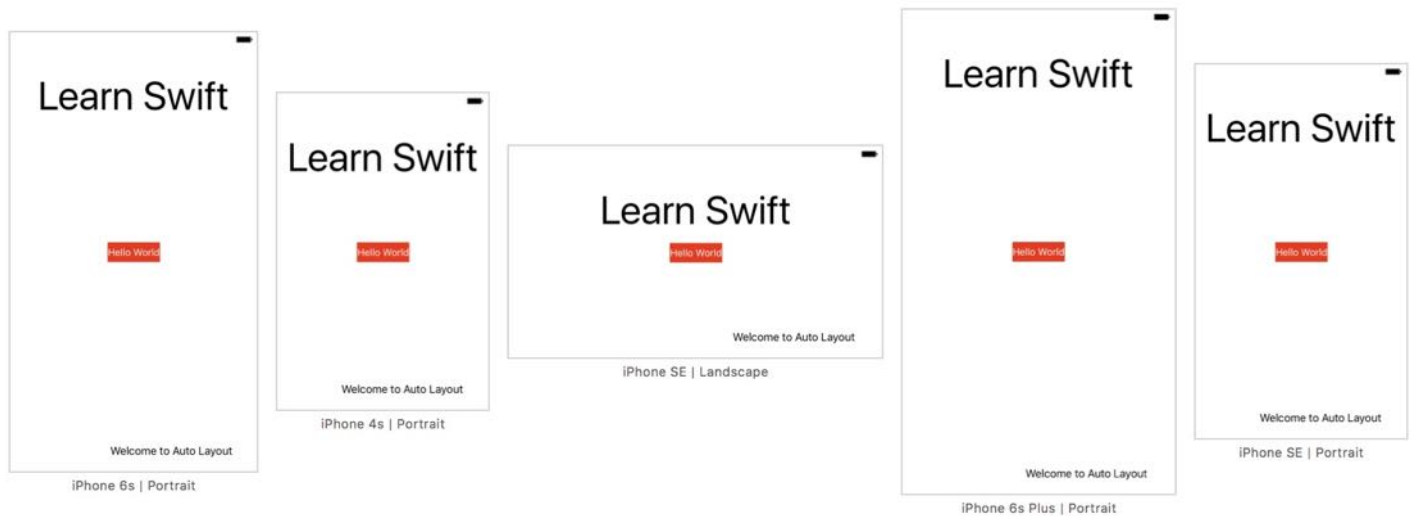


Figure 5-19. The expected app layout

Summary

In this chapter, we went through the basics of Auto Layout. It's just the basics because I don't want to scare you away from learning auto layout. As we dig deeper and create a real app, we'll continue to explore some other features of auto layout.

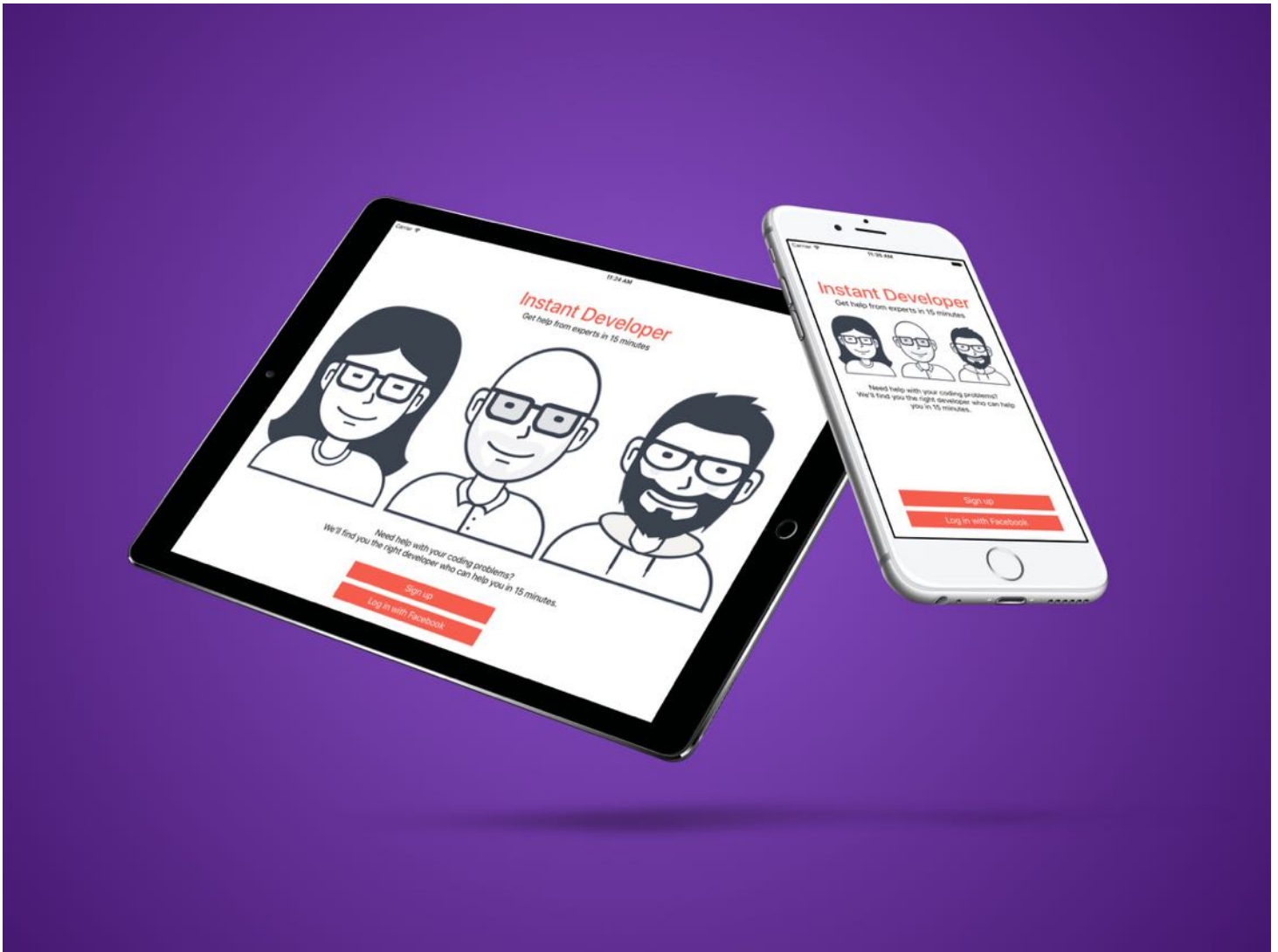
Most of the beginners (and even some experienced iOS programmers) avoid using auto layout because it looks confusing. If you thoroughly understand what I have covered in this chapter, you're on your way to becoming a competent iOS developer. The original iPhone was launched in 2007. Over these years, there have been tons of changes and advancements in the area of iOS development. Unlike the good old days when your apps just needed to run on a 3.5-inch, non-retina device, your apps now have to cater to various screen resolution and sizes. This is why I devoted a whole chapter to Auto Layout.

So take some time to digest the materials.

For reference, you can download the complete Xcode project from <https://www.appcoda.com/resources/swift3/HelloWorldAutoLayout.zip>.

Chapter 6

Designing UI Using Stack Views



To the user, the interface is the product.

- Aza Raskin

I have given you a brief overview of auto layout. The examples that we have worked on were pretty easy. However, as your app UI becomes more complex, you will find it more difficult to define the layout constraints for all UI objects. Starting from iOS 9, Apple introduced a powerful feature called Stack Views that would make our developers' life a little bit simpler.

You no longer need to define auto layout constraints for every UI objects. Stack views will take care of most of that.

In this chapter, we will continue to focus on discussing UI design with Interface Builder. I will teach you how to build a more comprehensive UI, which you may come across in a real-world application. You will learn how to:

1. Use stack views to lay out user interfaces.
2. Use image views to display images.
3. Manage images using the built-in asset catalog.
4. Adapt stack views using Size Classes.

On top of the above, we will explore more about auto layout. You'll be amazed how much you can get done without writing a line of code.

What is a Stack View

First things first, what is a stack view? The stack view provides a streamlined interface for laying out a collection of views in either a column or a row. In Keynote or Microsoft Powerpoint, you can group multiple objects together so that they can be moved or resized as a single object. Stack views offer a very similar feature. You can embed multiple UI objects into one by using stack views. In most cases, for views embedded in a stack view, you no longer need to define auto layout constraints.

Quick note: For views embedded in a stack view, they are usually known as arranged views.

The stack view manages the layout of its subviews and automatically applies layout constraints for you. That means, the subviews are ready to adapt to different screen sizes. Furthermore, you can embed a stack view in another stack view to build more complex user interfaces. Sounds cool, right?

Don't get me wrong. It doesn't mean you do not need to deal with auto layout. You still need to define the layout constraints for the stack views. It just saves you time from creating constraints for every UI element and makes it super easy to add/remove views from the layout.

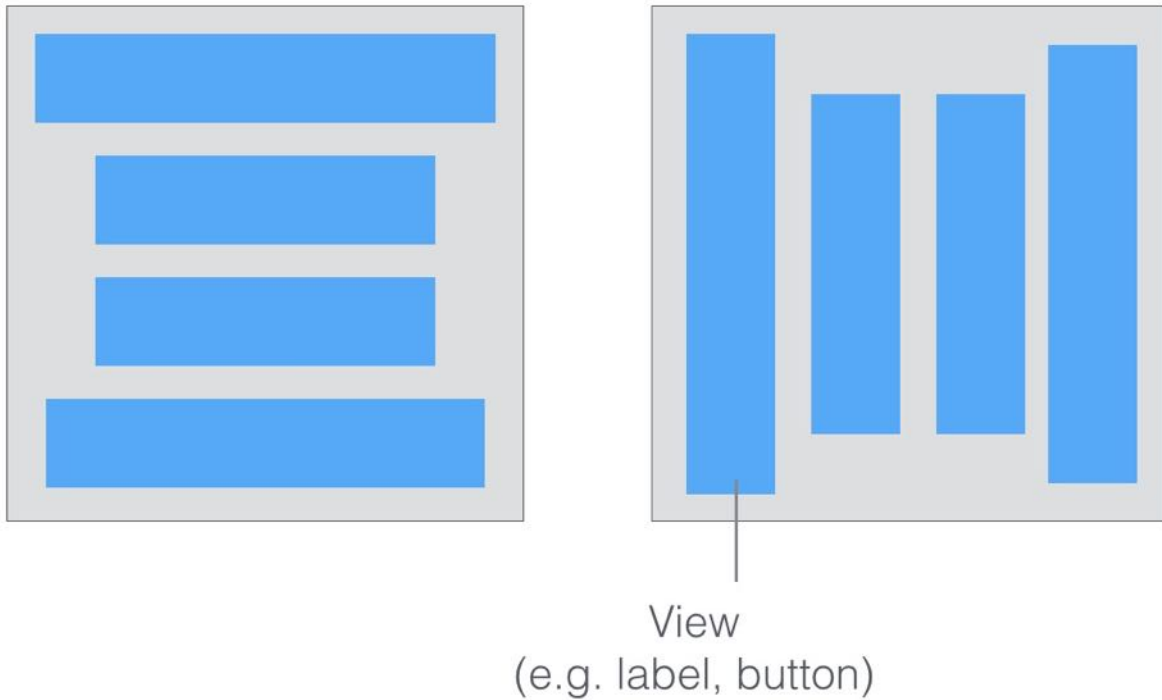


Figure 6-1. Horizontal Stack View (left) / Vertical Stack View (right)

Xcode 8 provides two ways to use stack view. You can drag a Stack View (horizontal / vertical) from the Object library, and put it right into the storyboard. You then drag and drop view objects such as labels, buttons, image views into the stack view. Alternatively, you can use the Stack option in the auto layout bar. For this approach, you select two or more view objects, and then choose the Stack option. Interface Builder then embeds the objects into a stack view and resizes it automatically. If you still have no ideas about how to use a stack view, no worries. We'll go through both approaches in this chapter. Just read on and you'll understand what I mean in a minute.

The Sample App

Let's first take a look at the demo app we're going to build. I will show you how to layout a welcome screen like this using stack views:

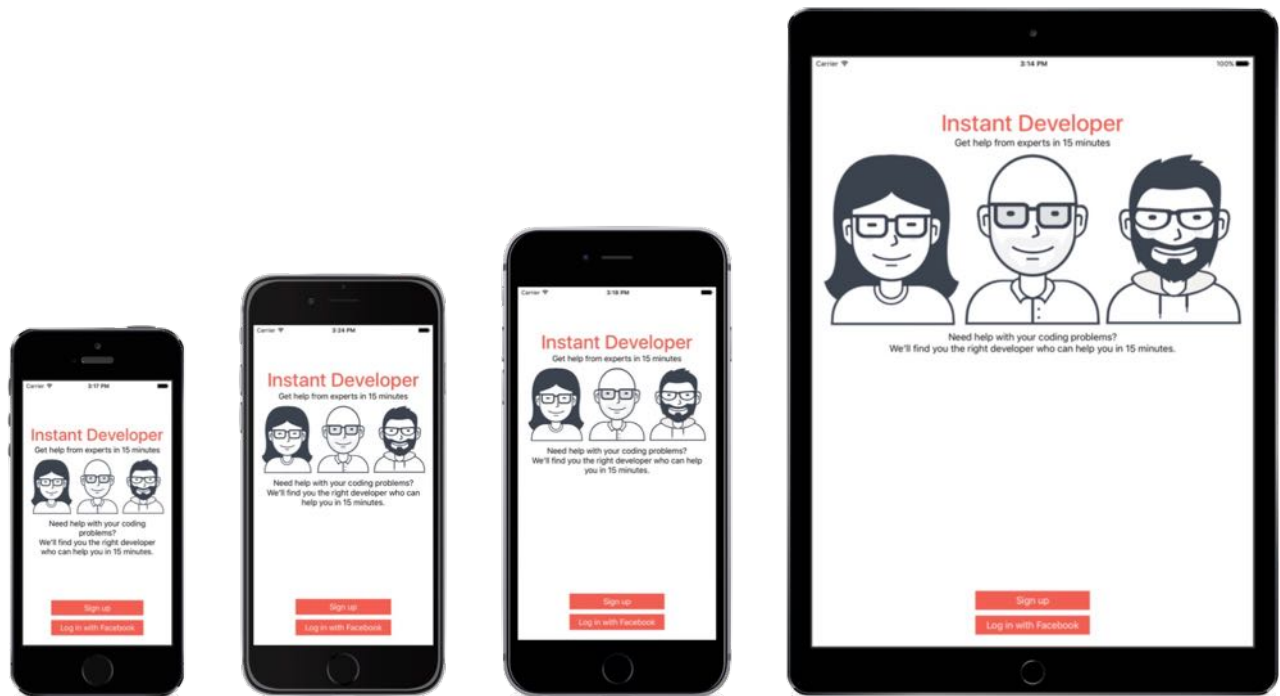


Figure 6-2. The sample app

You can create the same UI without using stack views. But you will soon see how stack views completely change the way how you layout user interfaces. Again, there is no coding in this chapter. We will just focus on using Interface Builder to building an adaptive user interface. This is a crucial skill you will need in app development.

Creating a New Project

Now fire up Xcode and create a new Xcode project. Choose Application (under iOS) > "Single View Application" and click "Next". You can simply fill in the project options as follows:

- **Product Name: StackViewDemo** – This is the name of your app.
- **Team:** Just leave it as it is.
- **Organization Name: AppCoda** – It's the name of your organization.
- **Organization Identifier: com.appcoda** – It's actually the domain name written the other way round. If you have a domain, you can use your own domain * name. Otherwise, you may use "com.appcoda" or just fill in "edu.self".
- **Bundle Identifier: com.appcoda.StackViewDemo** - It's a unique identifier of your

app, which is used during app submission. You do not need to fill in this option. Xcode automatically generates it for you.

- **Language: Swift** – We'll use Swift to develop the project.
- **Devices: Universal** – Select "Universal" for this project. A universal app is a single app that is optimized for iPhone, iPod touch, and iPad devices. For this demo, we'll design a user interface that works on all devices.
- **Use Core Data: [unchecked]** – Do not select this option. You do not need Core Data for this simple project.
- **Include Unit Tests: [unchecked]** – Do not select this option. You do not need unit tests for this simple project.
- **Include UI Tests: [unchecked]** – Do not select this option. You do not need UI tests for this simple project.

Click "Next" to continue. Xcode then asks you where to save the StackViewDemo project. Pick a folder on your Mac. Click "Create" to continue.

Adding Images to the Xcode Project

As you may notice, the sample app shows three images. The question is how can you bundle images in Xcode projects?

In each Xcode project, it includes an asset catalog (i.e. Assets.xcassets) for managing images and icons that are used by your app. Go to the project navigator and select the `Assets.xcassets` folder. By default, it is empty with a blank AppIcon set. We are not going to talk about app icons in this chapter, but will revisit it after building a real-world app.

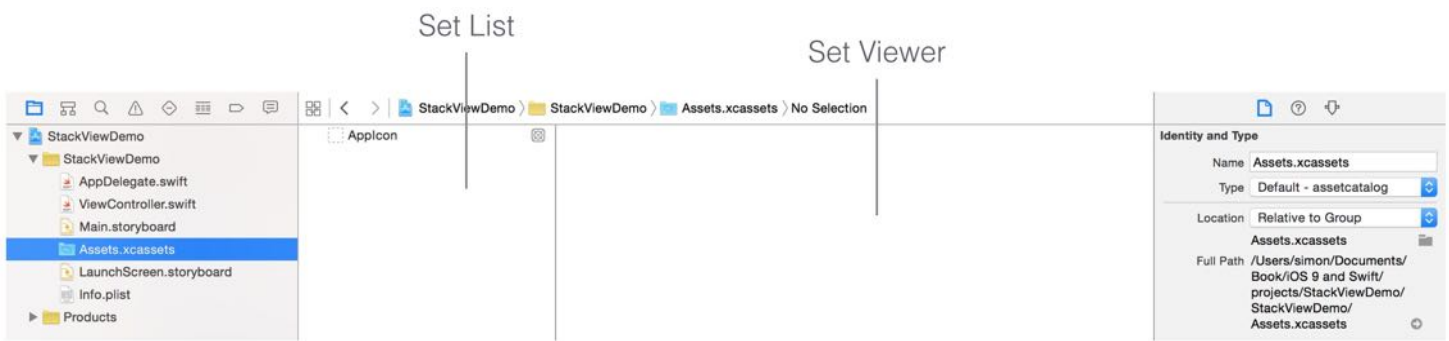


Figure 6-3. Asset Catalog

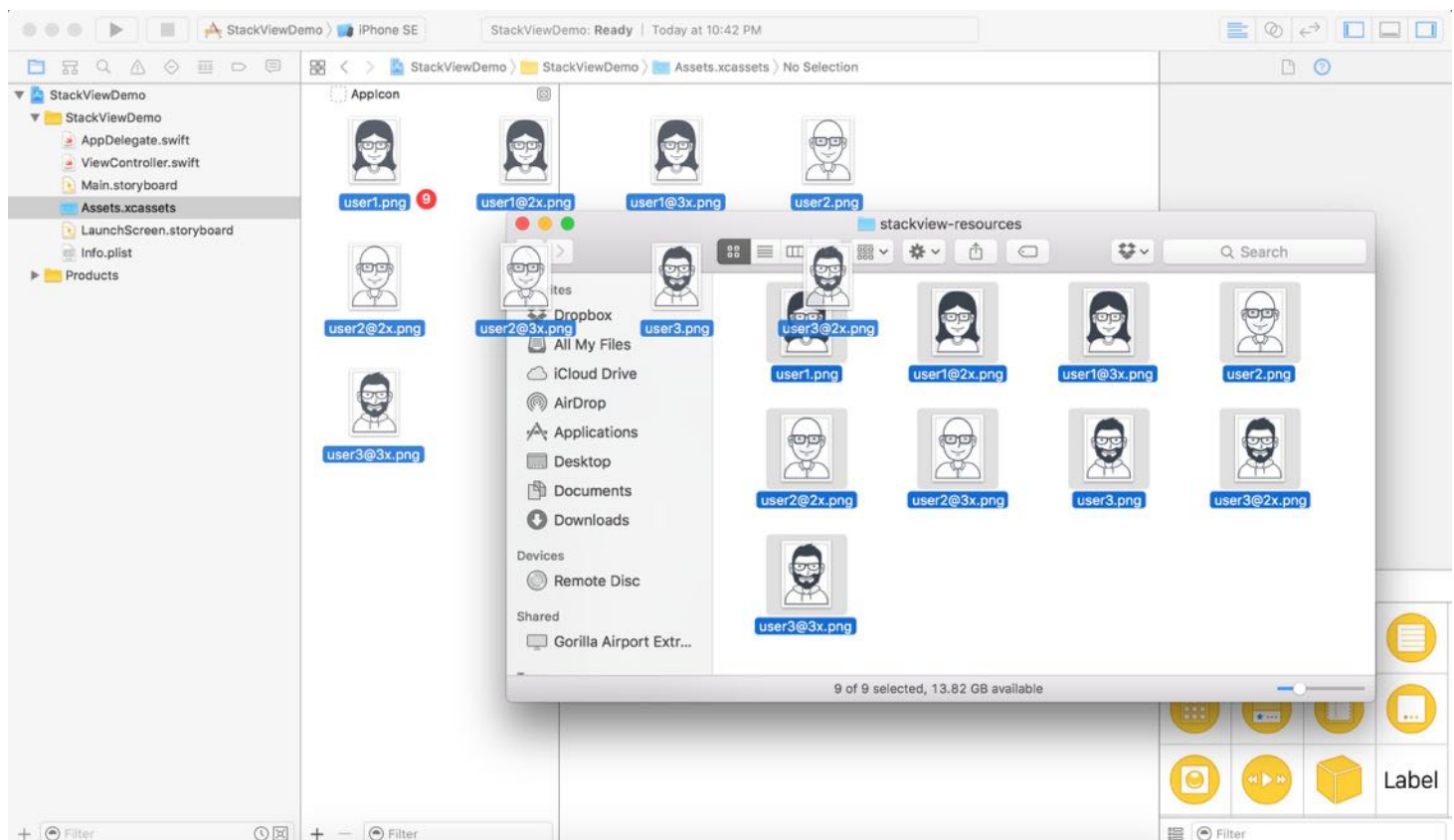
Now download this image set (<https://www.appcoda.com/resources/swift3/stackviewdemo-images.zip>) and unzip it on your Mac. The zipped archive contains a total of 9 image files, but it actually includes three different t-shirt images. Each one of them comes with three different resolutions. Here is an example:

- user1.png
- user1@2x.png
- user1@3x.png

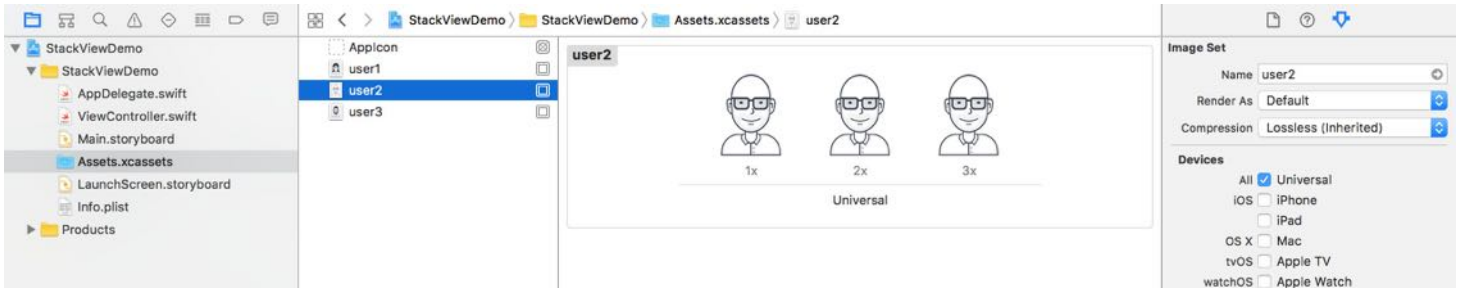
When developing an iOS app, it is recommended to prepare three versions of an image. The one with @3x suffix, which has the highest resolution, is for iPhone 6 Plus. The one with @2x suffix is for iPhone 4/4s/5/5s/6, while the one without the @ suffix is for older devices with non-Retina display (e.g. iPad 2).

Credit: The images are provided by usersinsights.com.

To add the images to the asset catalog, all you need to do is drag the images from Finder, and drop them into the set list or set viewer.



Once you add the images to the asset catalog, the set view automatically organizes the images into different wells. Later, to use the image, you just need to use the set name of a particular image (e.g. user1). You can omit the file extension and you don't have to worry about which version (@2x/@3x) of the image to use. All these are handled by iOS accordingly.



Layout the Title Labels with Stack Views

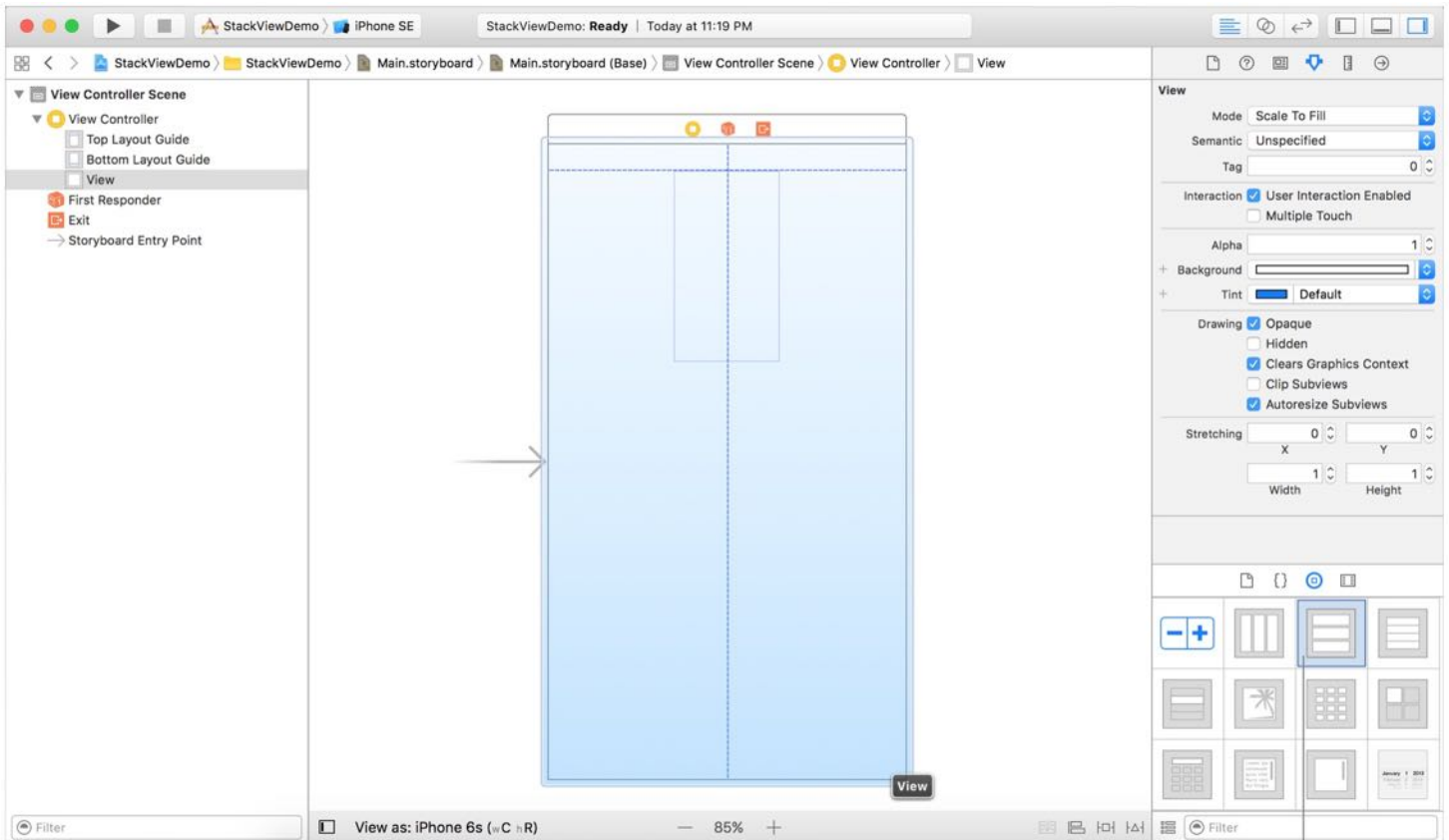
Now that you've bundled the necessary images in the project, let's move onto stack views. First, open `Main.storyboard`. We'll start with the layout of these two labels.

Instant Developer

Get help from experts in 15 minutes

Stack view can arrange multiple views (known as arranged views) in both vertical and horizontal layouts. So first, you have to decide whether you want to use a vertical or horizontal stack view. The title and subtitle labels are arranged vertically. Therefore, vertical stack view is a suitable choice.

From the Object library, drag a Vertical Stack View object to the view controller in the storyboard.



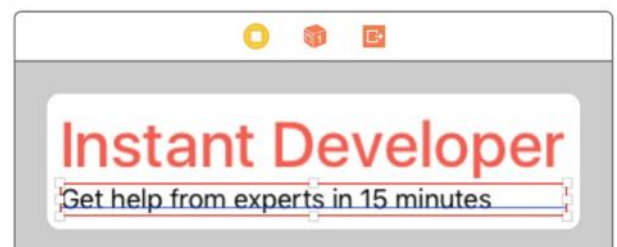
Vertical Stack View

Next, drag a label from the Object library and put it right into the stack view. Once you drop the label into the stack view, the stack view automatically embeds it and resizes itself to fit the label. Double click the label and change the title to "Instant Developer". In the Attributes inspector, increase the font size to 40 points, and font style to *medium*. Optionally, you can change the color to *red*.

Now, drag another label from the Object library to the stack view. As soon as you release the label, the stack view embeds the new label and arranges the two labels vertically like this:



1 Drag a label to the stack view



2 When you let go the button, the new label is added to the stack view automatically

Edit the title of the new label and change it to "Get help from experts in 15 minutes".

Currently, both labels are aligned to the left of the stack view. To change its alignment and appearance, you can modify the built-in properties of the stack view. Select the stack view and you can find its properties in the Attributes inspector.

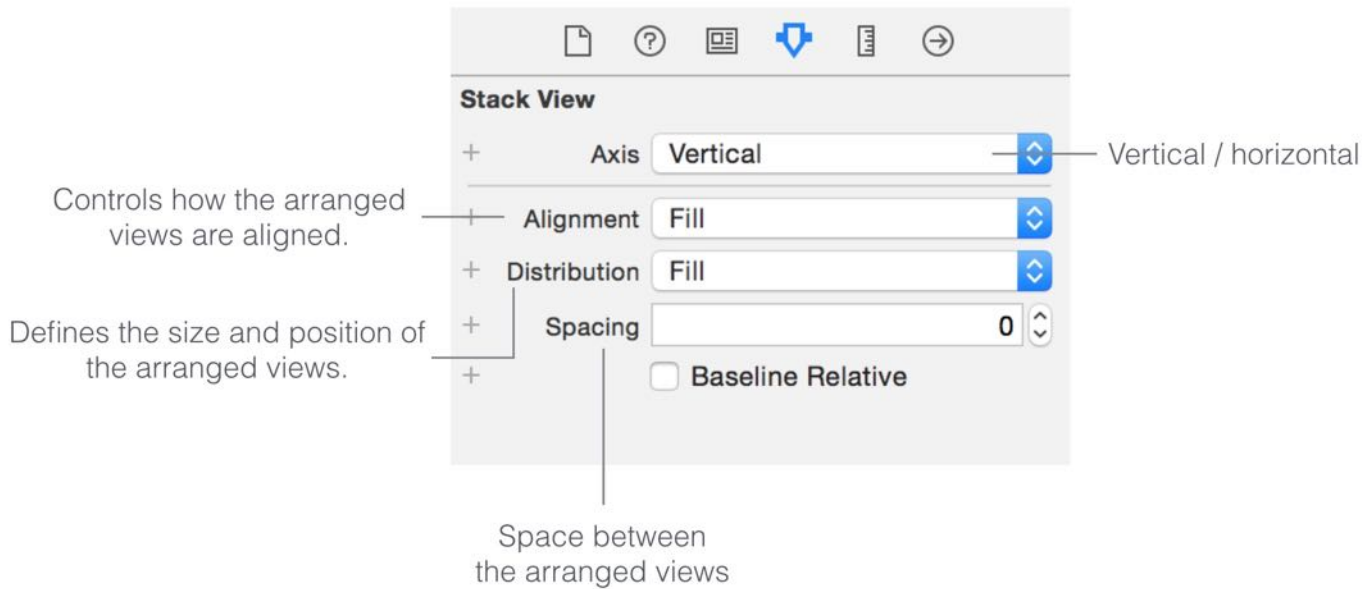


Figure 6-9. Sample properties of a stack view

As a sidenote, if you have any problems selecting the stack view, you can hold the shift key and right click the stack view. Interface Builder will then show you a shortcut menu for selection.

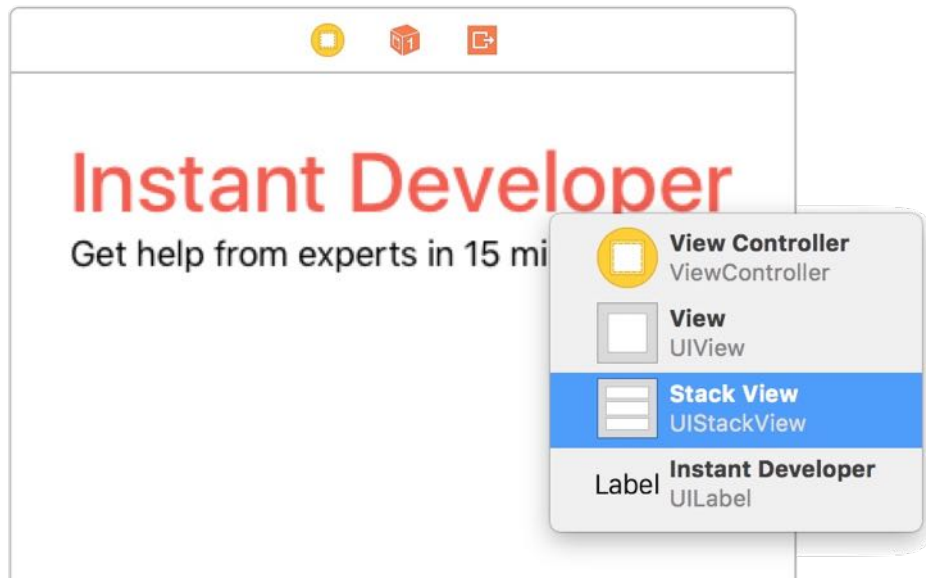


Figure 6-10. Shortcut menu for selection

Let's briefly talk about each property of the stack view:

- The *axis* option indicates whether the arranged views should be layout vertically or horizontally. By changing it from vertical to horizontal, you can turn the existing stack view to a horizontal stack view.
- The *alignment* option controls how the arranged views are aligned. For example, if it is set to *Leading*, the stack view aligns the leading edge (i.e. left) of its arranged views along its leading edge.
- The *distribution* option defines the size and position of the arranged views. By default, it is set to *Fill*. In this case, the stack view tries its best to fit all subview in its available space. If it is set to *Fill Equally*, the vertical stack view distributes both labels equally so that they are all the same size along the vertical axis.

Figure 6-11 shows some sample layouts of different properties.

Axis:



Vertical



Horizontal

Alignment:



Leading



Center



Trailing

Distribution:



Fill



Fill Equally

Figure 6-11. A quick demo of the stack view's properties

For our demo app, we just need to change the Alignment option from *Fill* to *Center* and keep other options intact. This should center both labels.

Before moving onto the next section, make sure you position the stack view correctly. This would help you easier for you to follow the rest of the materials. You can select the stack view and go to the Size inspector. Ensure you set the value of X to 28 and Y to 70 .

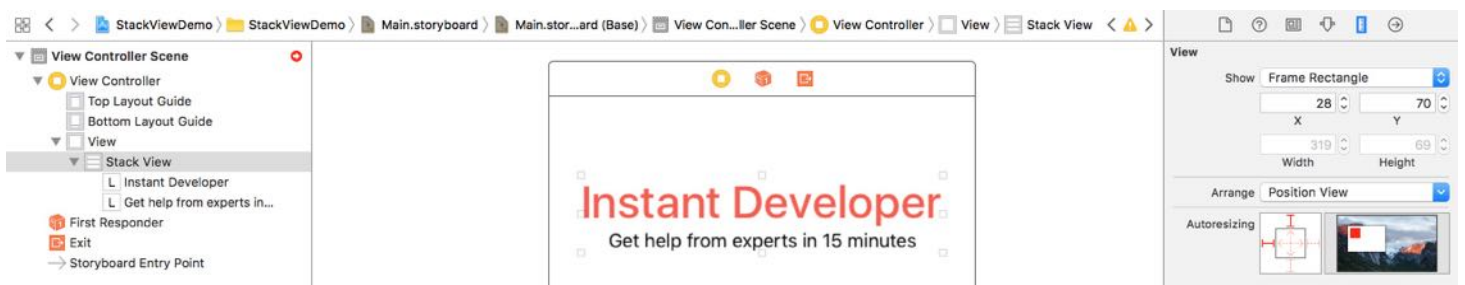


Figure 6-12. Verify the position of the stack view

Layout the Images Using the Stack Button

At the very beginning of this tutorial, I mentioned that there are two ways to use stack views. Earlier, you added a stack view from the object library. Now I will show you another approach.

We're going to lay out the three user images. In iOS, we use image views to display images. From the Object library, look for the image view object, and drag it into the view. Once you added the image view, select the Attributes inspector. The image option already loads the available images from the asset catalog. Simply set it to `user1`, and resize it to make it smaller.

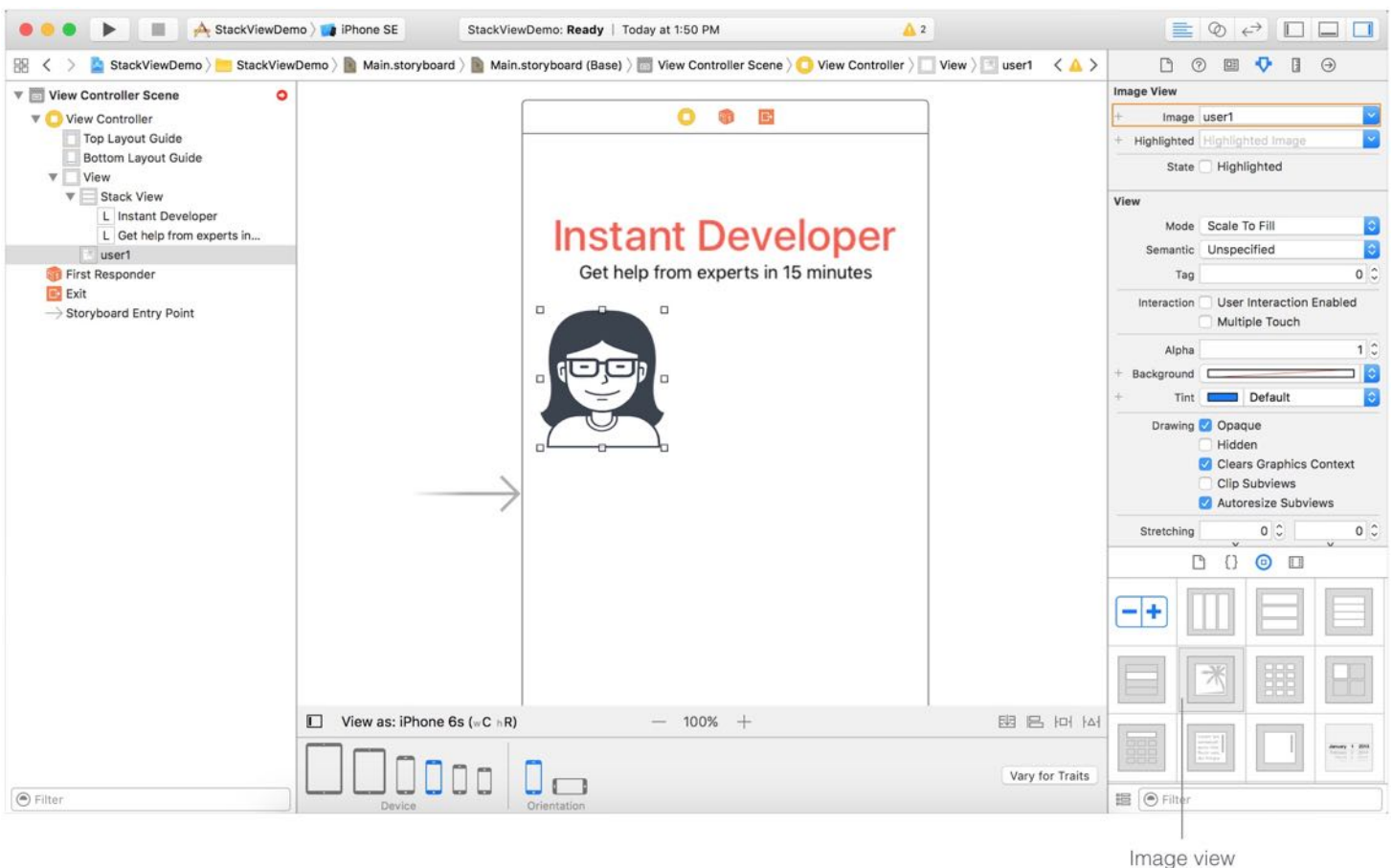


Figure 6-13. Adding an image view for displaying images

Repeat the procedures to add two more image views, and place them next to each other. Set the image of the second and third image view to `user2` and `user3` respectively. Your layout should look like this:

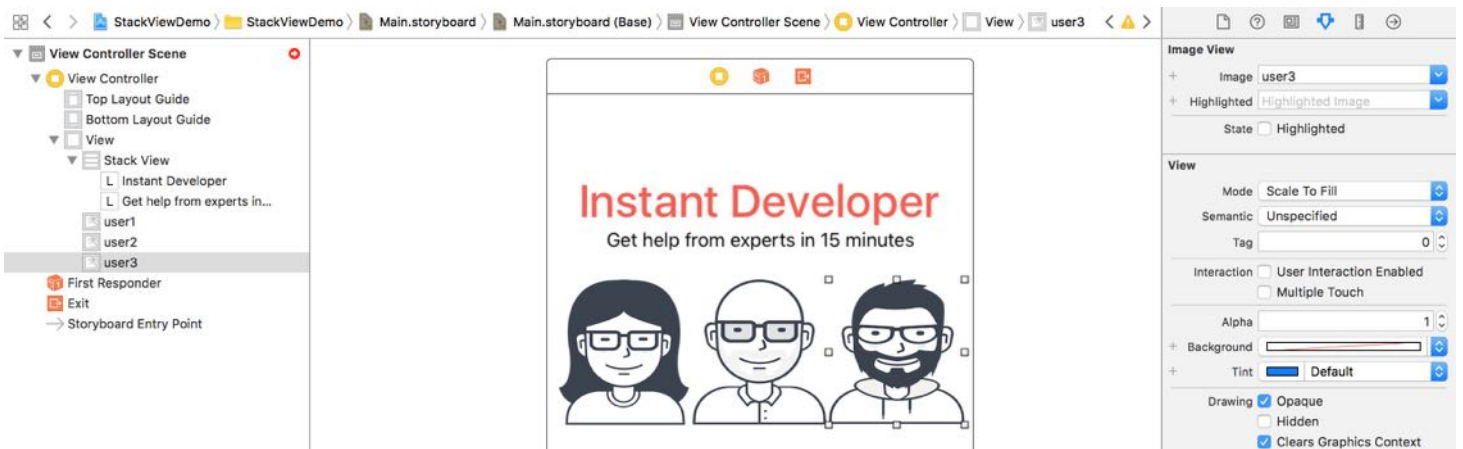


Figure 6-14. Assign the images to the image views

I want to group these three image views together using a stack view, so it is easier to manage. However, we will use an alternative approach to create the stack view.

Hold the command key and click the three image view to select them. Then click the Stack button in the layout bar. Interface Builder automatically embeds the image views in a horizontal stack view.

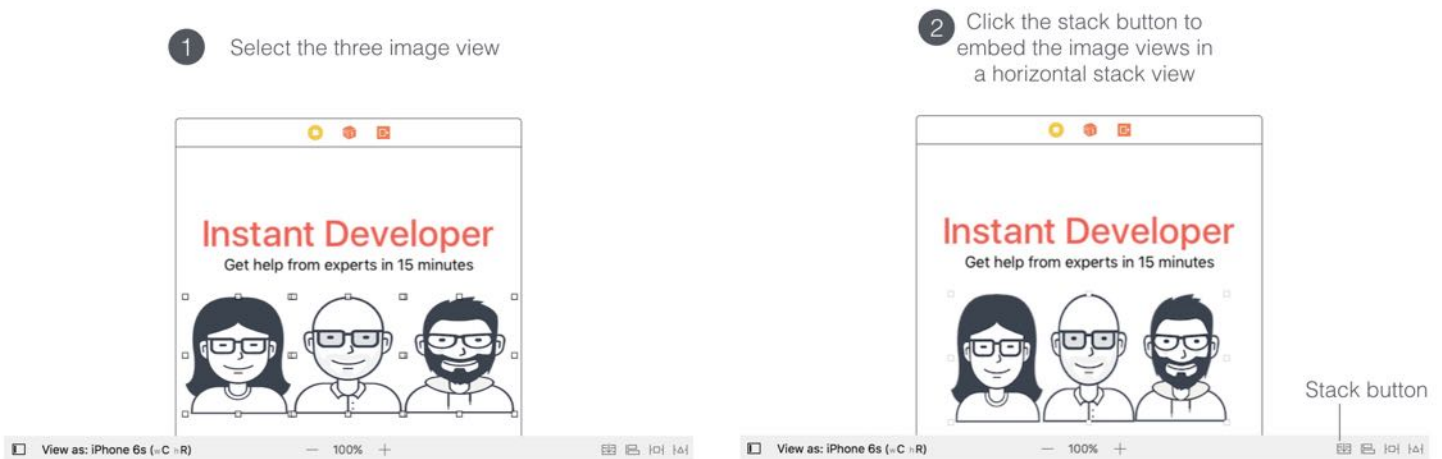


Figure 6-15. Group the image views in a horizontal stack view

To add some spacing between the image views, select the stack view and set the spacing to 10. Also change the Distribution option from *Fill* to *Fill Equally*.

Quick note: It seems that the stack view has already distributed the image views equally. Why do we need to change the distribution to Fill Equally? Remember that you're now designing the UI for all screen sizes. If you do not explicitly set the distribution to Fill Equally, iOS will layout the image views using Fill distribution policy. It may not look good on other screen sizes.

Now you have two stack views: one for the labels, and the other for the image views. If you refer to the final layout (see figure 6-2), these two stack views can actually be combined together for easier management. One great thing about stack views is that you can nest multiple stack views together.

To do that, select both stack views through the document outline view. Then click the Stack button to embed both stack views in a vertical stack view.

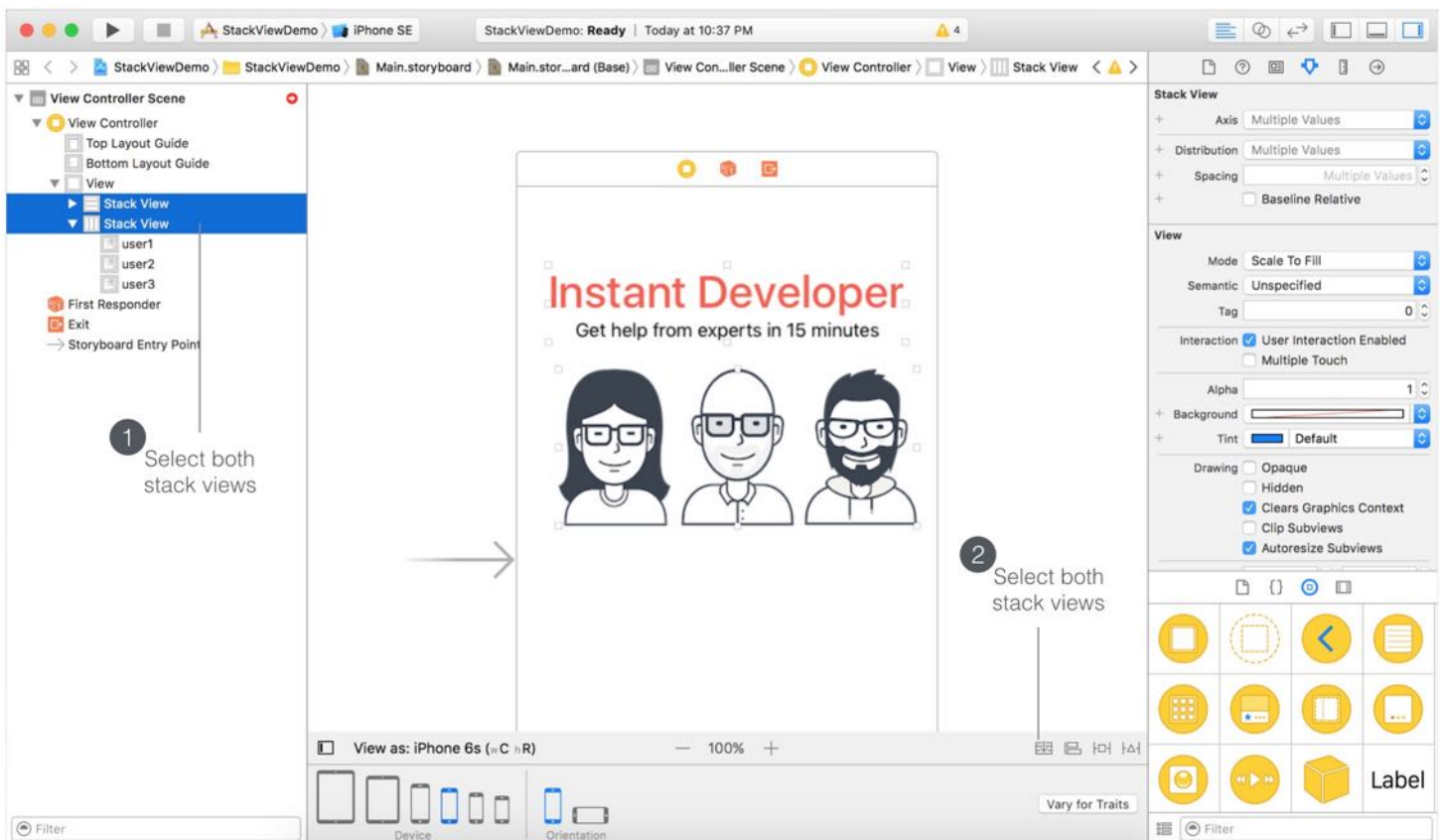


Figure 6-16. Nested stack views

Cool, right? So far the UI looks good on iPhone 6/6s. But if you preview the UI on other devices, it doesn't look as expected. The reason is that we haven't defined the layout

constraints for the stack views. As I mentioned at the very beginning of the chapter, stack views only save you from defining the layout constraints of the arranged views. You still need to create layout constraints for the stack views.

Adding Layout Constraints for the Stack View

For the stack view, we will define the following layout constraints:

- Set a spacing constraint between itself and the top layout guide, such that it is 70 points away from the top layout guide.
- Set a spacing constraint between left side of the stack view and the left margin of the view, such that there is no space (i.e. 0 point) between them.
- Set a spacing constraint between right side of the stack view and the right margin of the view, such that there is no space (i.e. 0 point) between them.

Now click the Pin button in the layout button. Set the space constraints of the top, left and right side to 70, 0 and 0 respectively. When the constraint is enabled, it is indicated by a solid red bar. Then click the "Add 3 Constraints" button to add the constraints.

When the bar is in solid red, it indicates the constraint is selected.

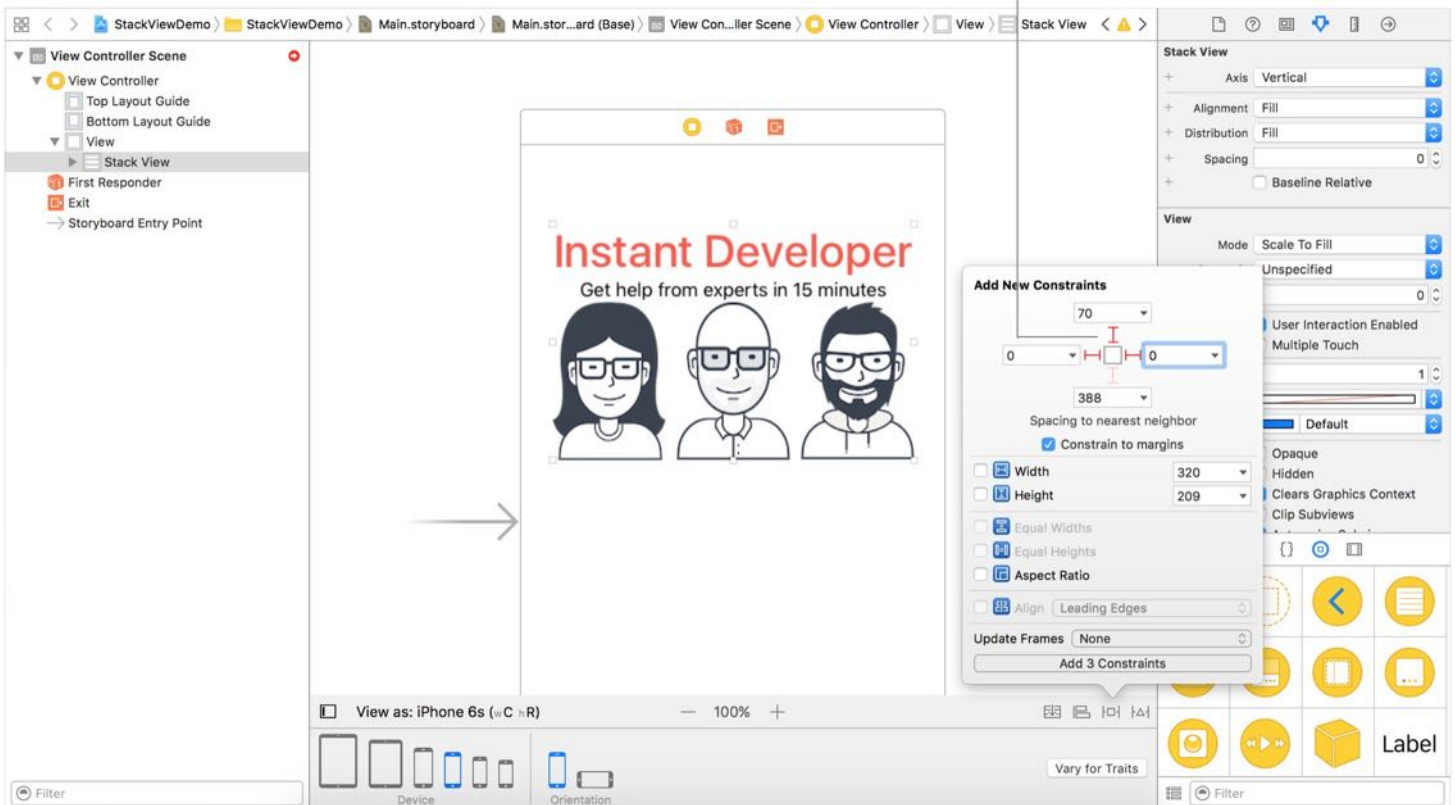


Figure 6-17. Adding multiple constraints using the Pin button

Once you added the constraints, Interface Builder indicates the stack view is not correctly positioned. You can click the issue indicator in the document outline view, followed by clicking another yellow indicator in the issue view. When prompted, select *Update frames* and click *Fix Misplacement*.

Interface Builder automatically re-positions the stack view to the correct position that conforms to your layout constraints.

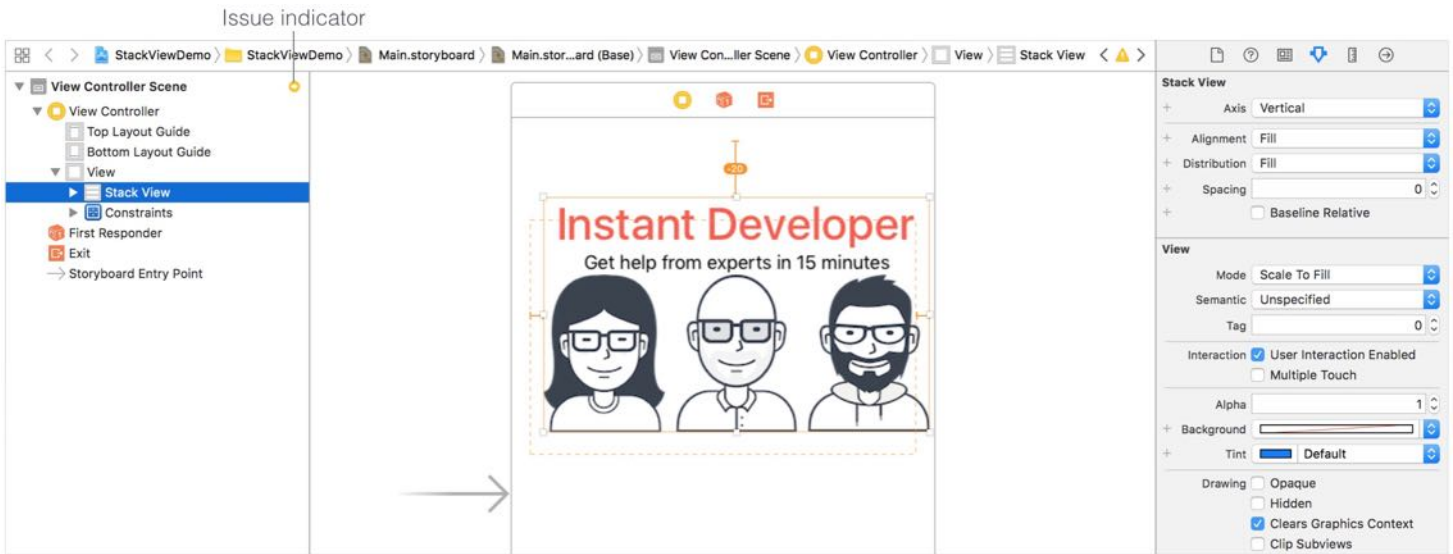


Figure 6-18. Resolving the auto layout issues

Now try to preview the interface on other devices. They should look pretty good. However, you may notice two issues:

- The *Instant Developer* label is truncated for iPhone SE (4-inch) and 4s (3.5-inch).
- The aspect ratio of the images is not retained. All images appear horizontally stretched. When you view the UI on iPad, it looks especially weird.

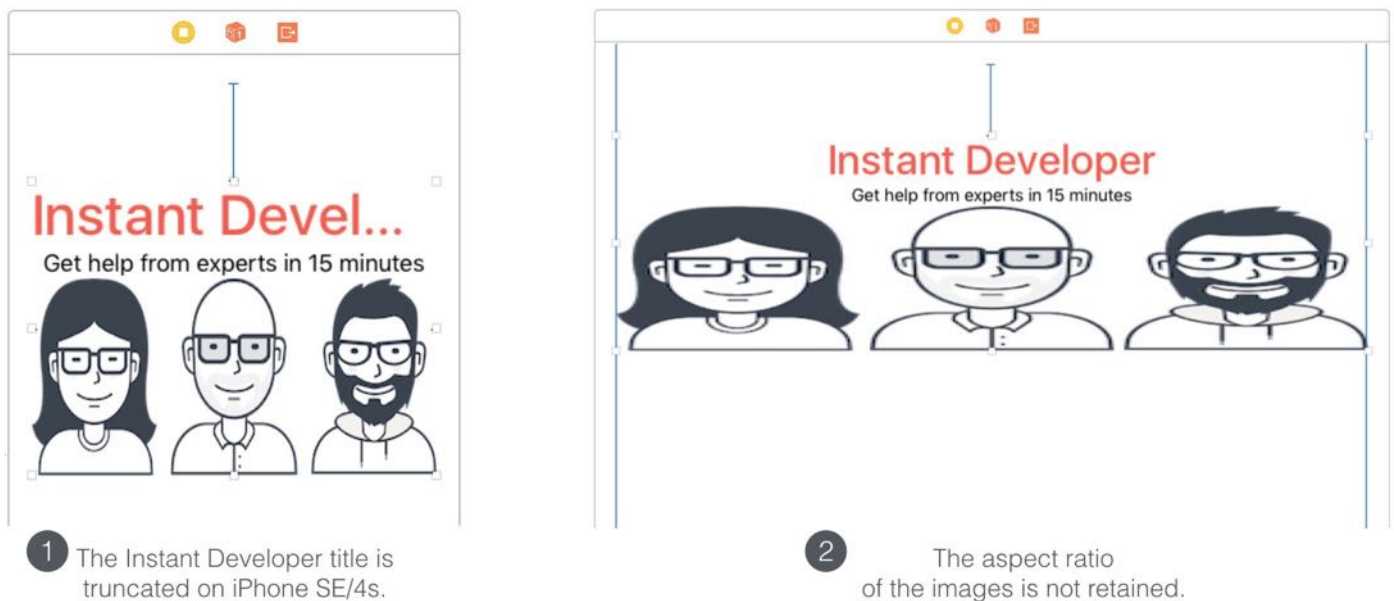


Figure 6-19. Previewing the UI on iPhone SE and iPad

So how can we fix the issues?

For the first issue, it is quite obvious that we can decrease font size of the label. Xcode provides an automatic way to adjust the font size on the fly. Now select the *Instant Developer* label, and go to the Attributes inspector. Set the *Autoshrink* option to *Minimum Font Size*, and the value to 20 .

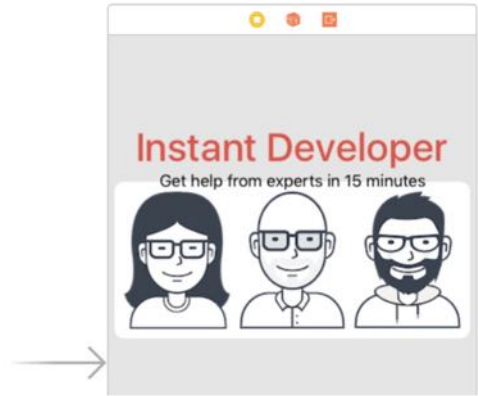


Figure 6-20. Changing the autoshrink option of the label

By doing so, you tell Xcode (or iOS) to determine the suitable font size for the label such that it can perfectly be displayed.

For the second issue about image scaling, we have to define one more constraint and ask the stack view to retain the aspect ratio, regardless of the screen sizes. In the document outline view, control-drag horizontally on the stack view (containing the image views). In the shortcut menu, select `Aspect Ratio` .

1 Control-drag horizontally from the stack view to itself



2 Release the button and choose aspect ratio

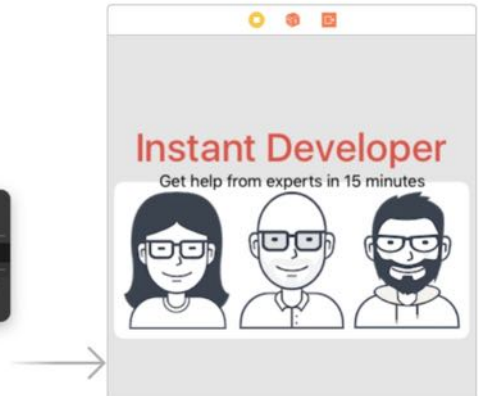
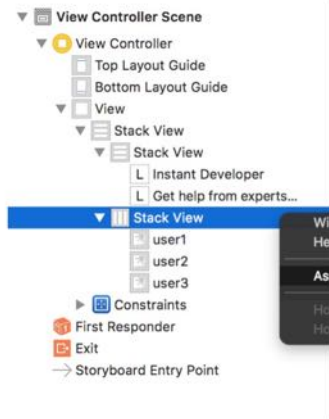


Figure 6-21. Control-drag from the stack view to itself to add a constraint

Now choose iPhone SE/4s or iPad in the configuration bar to test out the UI. The layout should look much better.

Adding a Label Below the Images

There is a label right below the images that we haven't added yet. I intentionally left it out so that I can show you how easy you can add an object to an existing stack view.

In the Object library, drag a label object to the stack view holding the other two stack views. You will see a blue line that indicates the position of the insertion point.

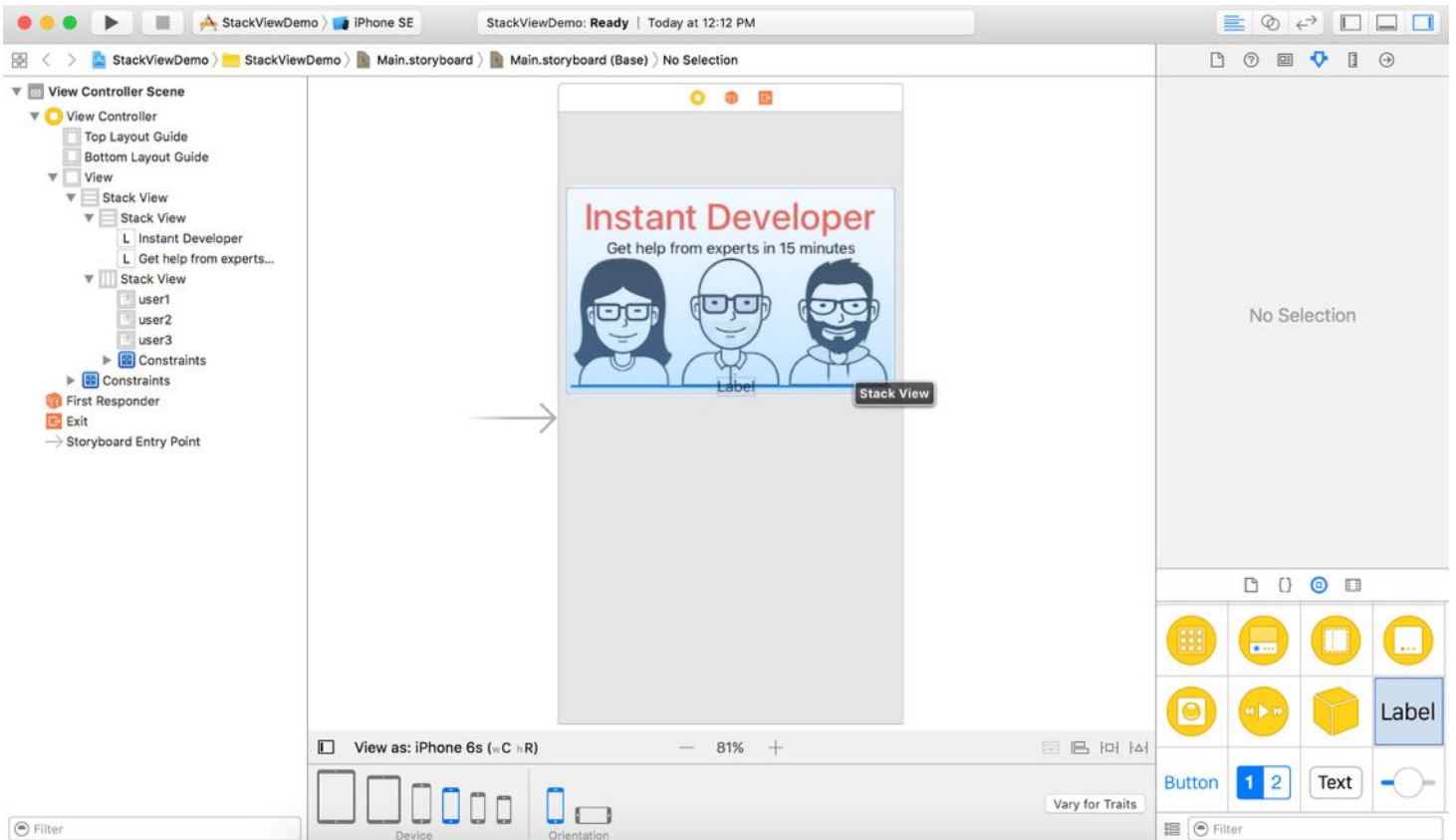


Figure 6-22. Adding a new label to an existing stack view

Next, select the new label. Go to the Attributes inspector, set the text to the following:

Need help with your coding problems? We'll find you the right developer who can help you in 15 minutes.

Quick tip: When you type in the text field, you can press option+return to add a line break.

Furthermore, set the alignment option to center, and lines to 0 to show multiple lines.

Both labels are too close the images. To add some space between the text and images, select the stack view and change the spacing from 0 to 10 in the Attributes inspector.

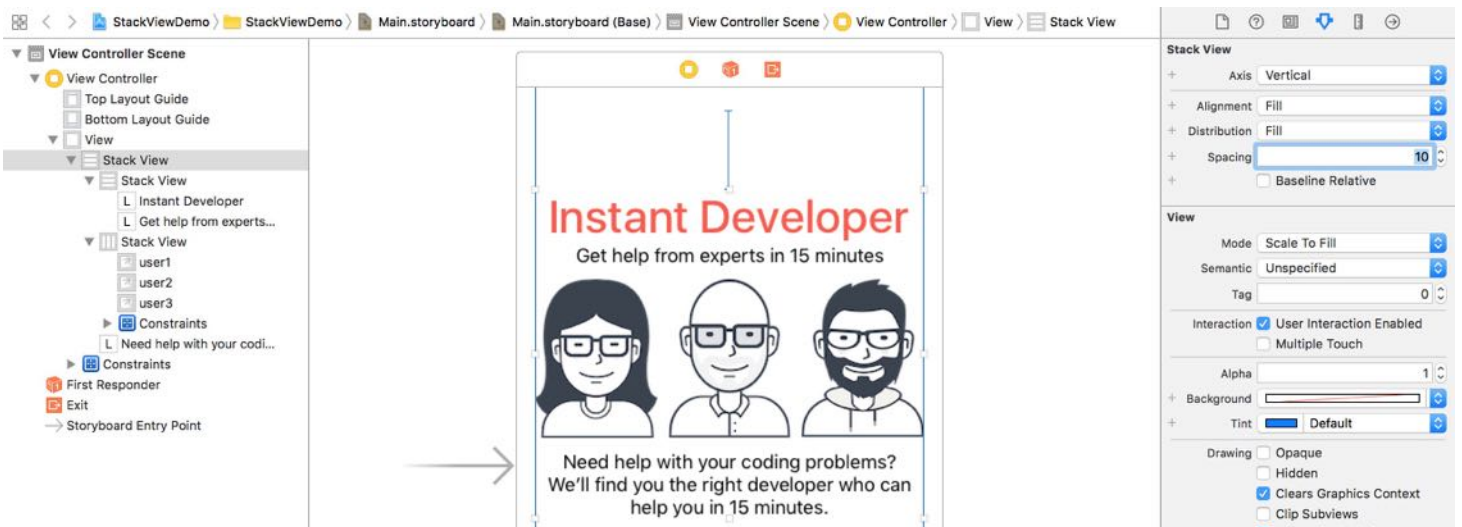


Figure 6-23. Adding some space between the labels and images

Now the UI looks pretty good on all devices. As you can see, stack views save you from defining constraints for the new label. It just inherits the constraints that you have defined earlier.

Layout the Buttons Using a Stack View

We haven't finished yet. Let's continue to layout the two buttons at the bottom of the screen.

First, drag a button from the Object library to the view. Double click the button to name it "Sign in". In the Attributes inspector, change its background color to *red* and text color to *white*. In the Size inspector, set the width to 200. Next, drag another button to the view and name it "Sign up with Facebook". In the Attributes inspector, change its background color to *red* and text color to *white*. In the Size inspector, set the width to 200.

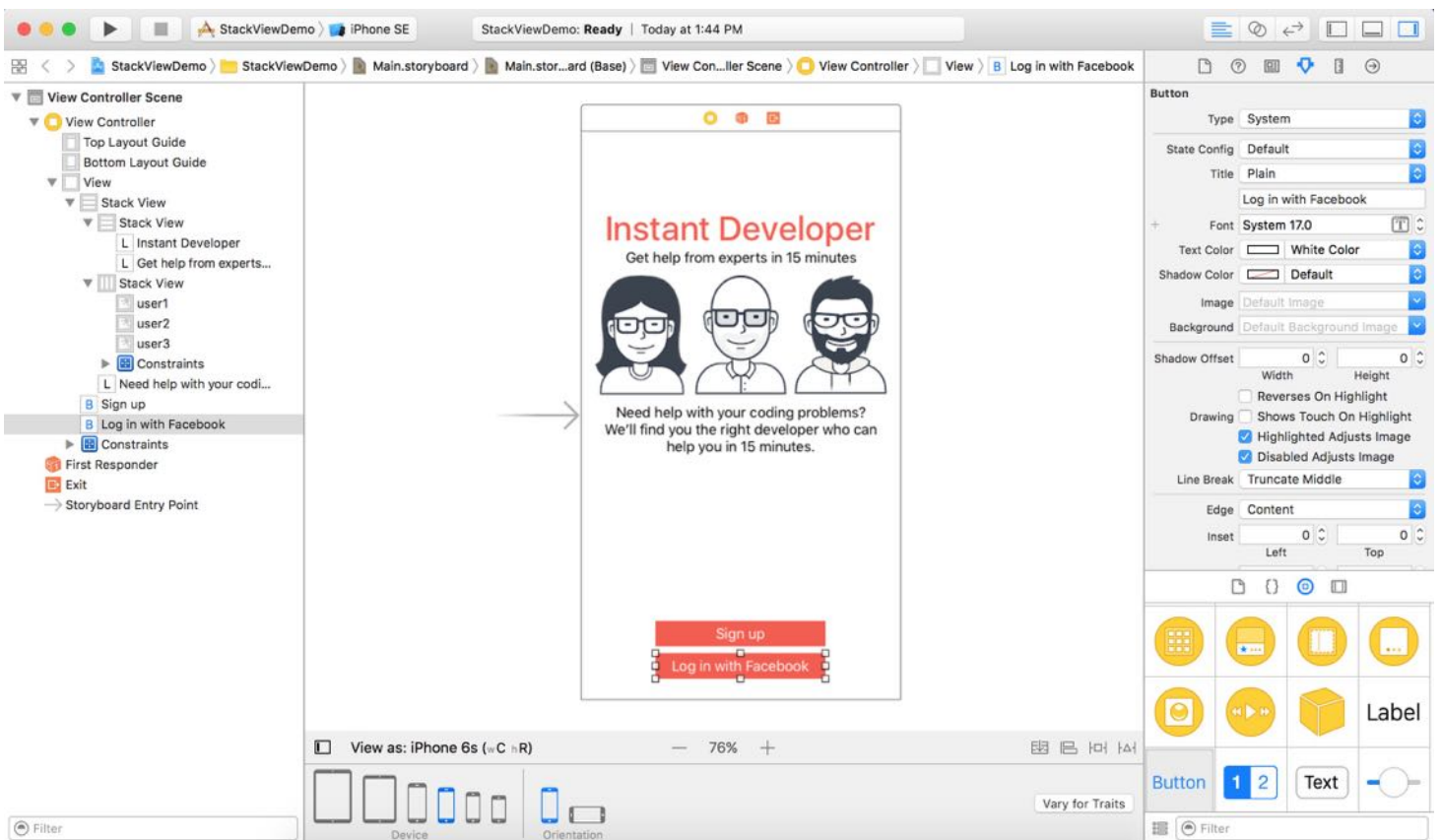


Figure 6-24. Adding two buttons to the view

Again, you do not need to set the layout constraints for these labels. Let the stack view do the magic for you. Hold the command key and select both buttons. Then click the Stack button in the layout bar to group them in a vertical stack view. Next, center the stack view horizontally. To add a space between the buttons, select the stack view. In the Attributes inspector, set the value of *spacing* to 10.

Similarly, we have to define layout constraints for this stack view so that it is positioned close to the bottom of the view. Here are the layout constraints we're going to define:

- Center the stack view horizontally, with respect to the container view.
- Set a spacing constraint so that there is a space between the stack view and the Bottom Layout Guide.

Select the stack view that you just created. Click the Align button in the layout bar. Check the *Horizontally in Container* option and click *Add 1 Constraint*.

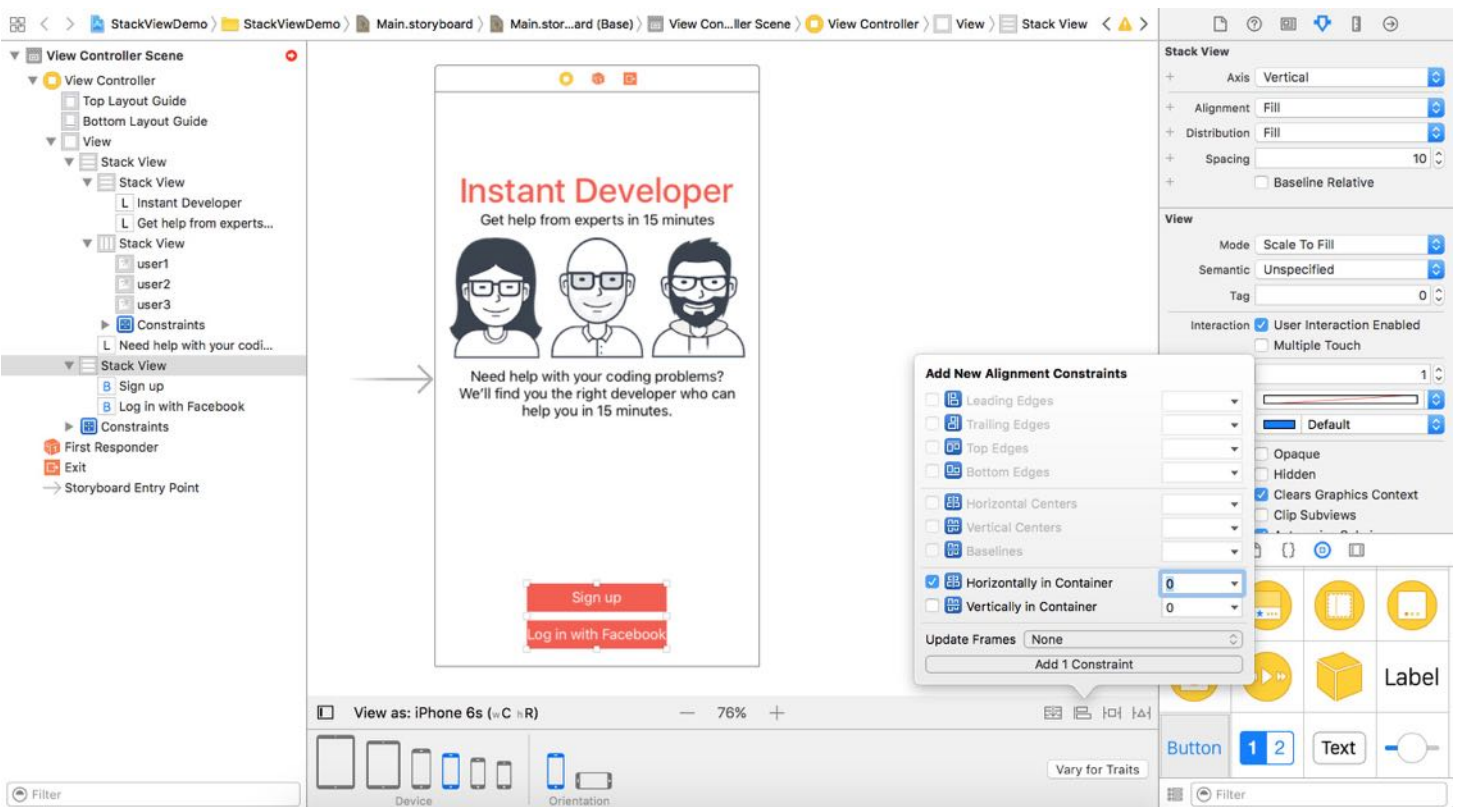


Figure 6-25. Adding a new constraint to center the stack view horizontally

Next, to add a spacing constraint, click the Pin button and set the value of the bottom side to 20 . Make sure the bar is in solid red. Click *Add 1 Constraint* to add the constraint.

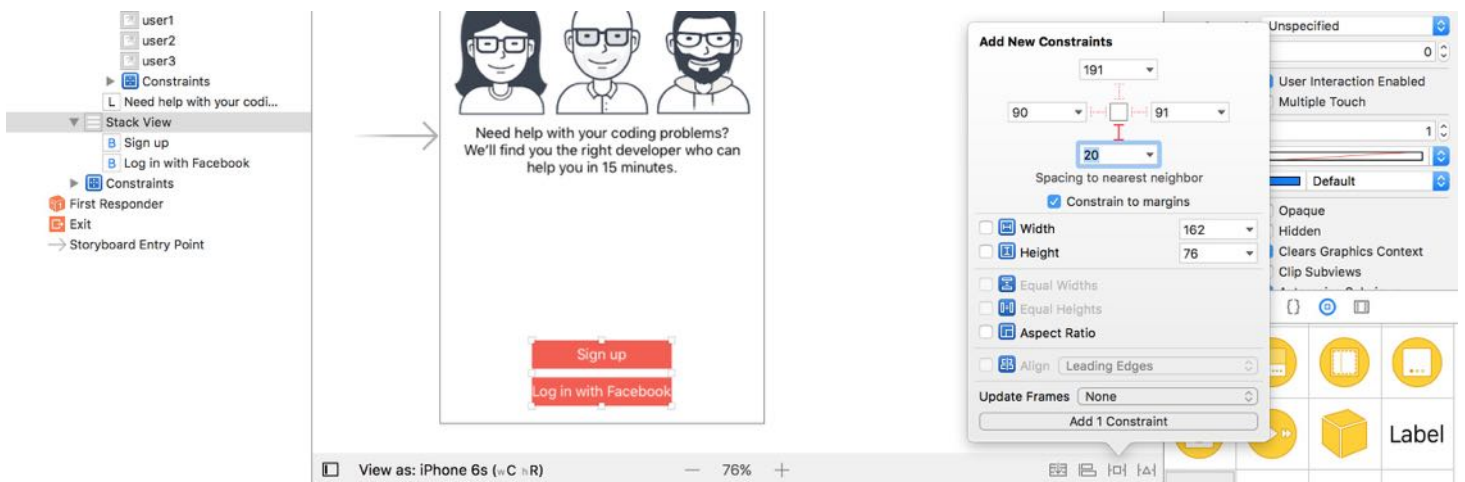


Figure 6-26. Adding a spacing constraint to the stack view

One thing you may notice is that the button is narrower than expected. As you embed the buttons in the stack view, they are resized to its intrinsic size. If you want to set the width to 200, you will need to add a width constraint to the stack view. In the document view, control-drag from the stack view to itself and choose *Width* to add the constraint.

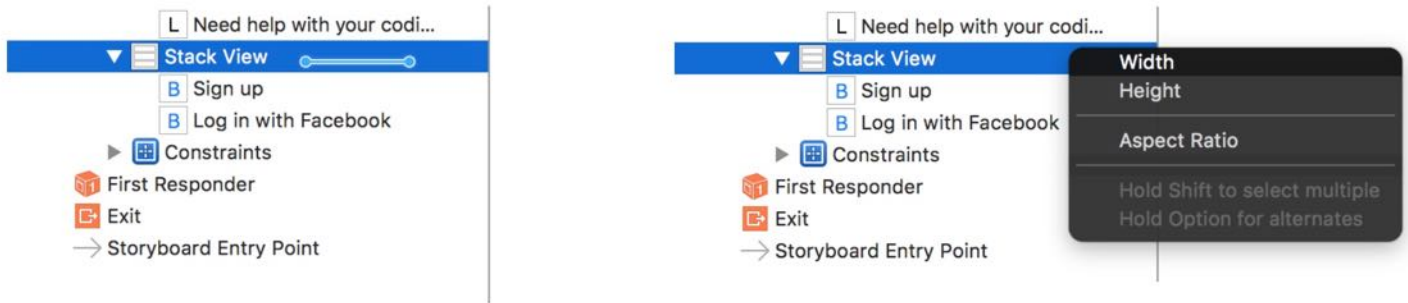


Figure 6-27. Adding the width constraint using control-drag

To change the width to 200 points, select the width constraint of the stack view in the document outline view. In the Attributes inspector, change the constant to 200. The stack view now has a fixed width of 200 points.

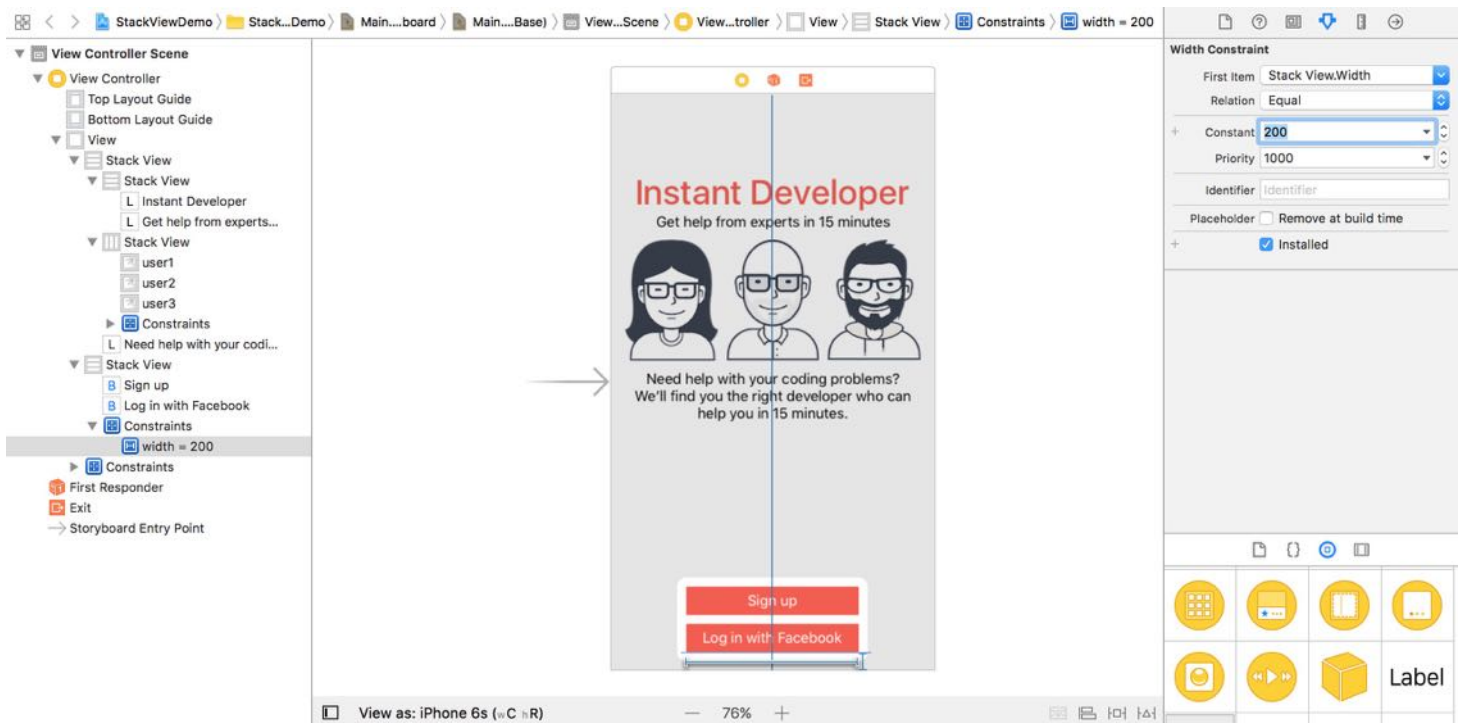


Figure 6-28. Editing the width constraint

It's time to test the app again. You can preview the UI in Interface Builder or run the project on different devices. Your UI should look great on all devices.

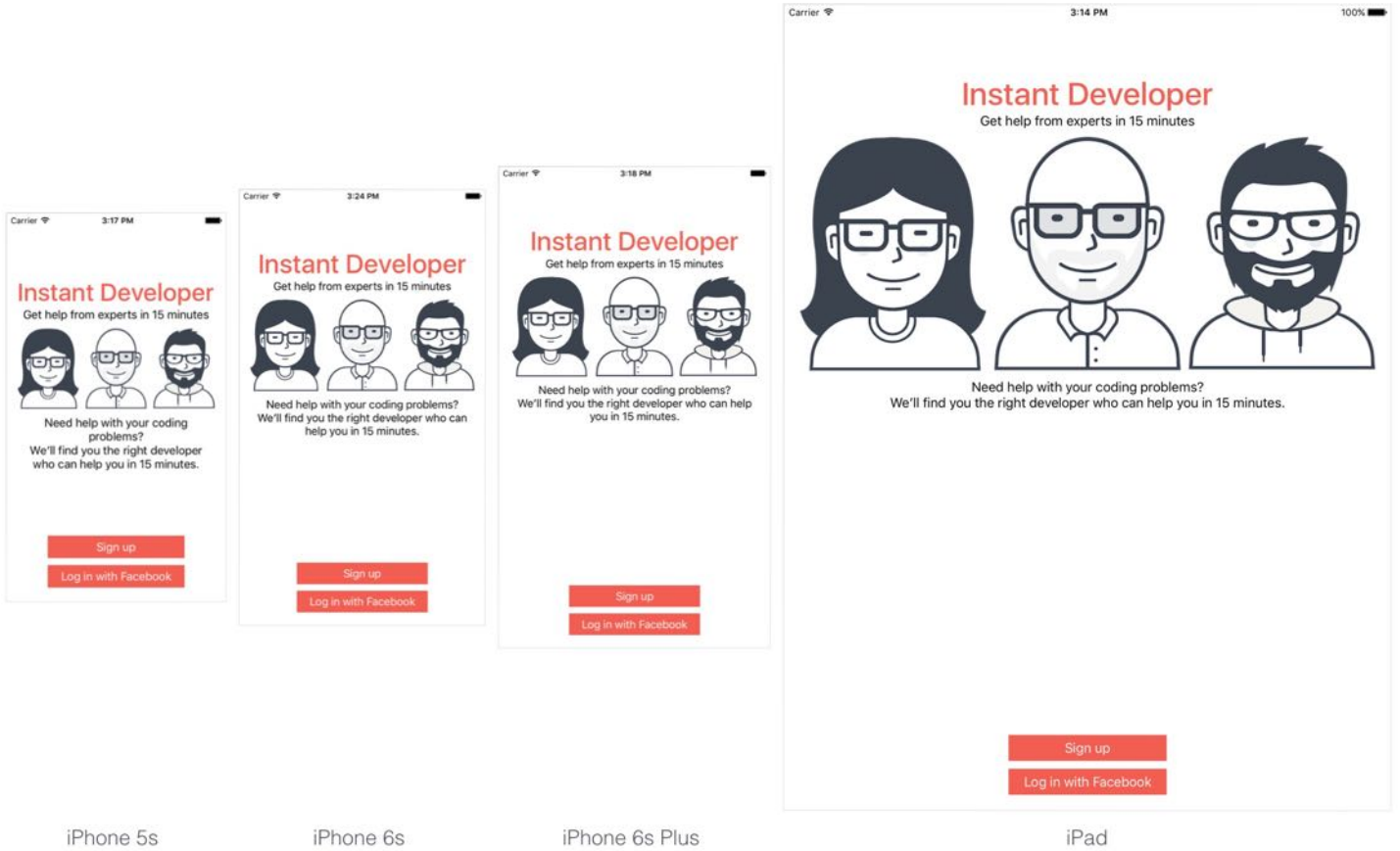


Figure 6-29. Previewing the UI on iPhone and iPad

Adapting Stack Views Using Size Classes

Now that you've already built a nice UI using stack views, have you tested the layout in landscape orientation? If you turn the iPhone simulator sideways, the UI in landscape orientation looks like this:

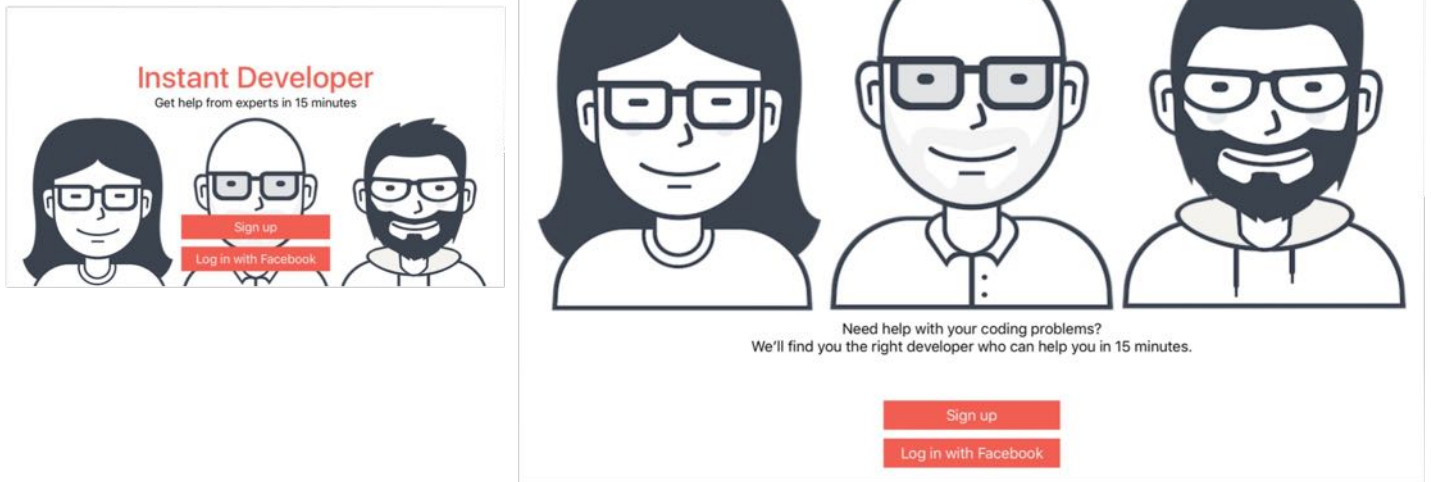


Figure 6-30. The app UI in landscape orientation on iPhone (left) and iPad (right)

It still looks pretty good. To make it look even better, I want to decrease the spacing between the Top Layout Guide and the upper stack view (holding both labels and image views). But this changes only applies to iPhones in landscape orientation.

As you know, we can adjust the spacing by changing the constant of the spacing constraints. The question is: *how can you apply the change for iPhones in landscape orientation only?*

This leads to a new UI design concept known as *Adaptive Layout*, introduced since iOS 8. With adaptive layout, your apps can adapt their UI to a particular device and device orientation. To achieve adaptive layout, Apple introduced a new concept, called **Size Classes**. This is probably the most important aspect which makes adaptive layout possible. Size classes are an abstraction of how a device is categorized depending on its screen size and orientation. You can use both size classes and auto layout together to design adaptive user interfaces.

A size class identifies a relative amount of display space for both vertical (height) and horizontal (width) dimensions. There are two types of size classes: *regular* and *compact*. A regular size class denotes a large amount of screen space, while a compact size class denotes a smaller amount of screen space.

By describing each display dimension using a size class, this will result in four abstract devices: *Regular width-Regular Height*, *Regular width-Compact Height*, *Compact width-Regular Height* and *Compact width-Compact Height*.

The table below shows the iOS devices and their corresponding size classes:

		Horizontal Size Class	
		Regular	Compact
Vertical Size Class	Regular	iPad Portrait iPad Landscape	iPhone Portrait
	Compact	iPhone 6 Plus Landscape	iPhone 4/5/6 Landscape

To characterize a display environment, you must specify both a horizontal size class and vertical size class. For instance, an iPad has a regular horizontal (width) size class and a regular vertical (height) size class. In our case, we want to provide layout specializations for iPhones in landscape orientation. In other words, here are the two size classes we have to deal with:

- Compact width-Compact height
- Regular width-Compact height

You don't need to remember the above table. In Interface Builder, you can find out the current size class in the configuration bar. For example, when iPhone 6s is selected, it shows "View as: iPhone 6s (wC hC)". "wC hC" means compact width and compact height.

With an understanding of size classes, now go back to the `Main.storyboard`. In the configuration bar, set the device to landscape orientation. Next, select the spacing constraint between the Top Layout Guide and the upper stack view. In the Attributes inspector, click the + button next to the *Constant* option. Select Any Width > Compact height > Any Gamut, and then set the value of this size class to 0.

Quick note: Here, any width includes both compact and regular width.

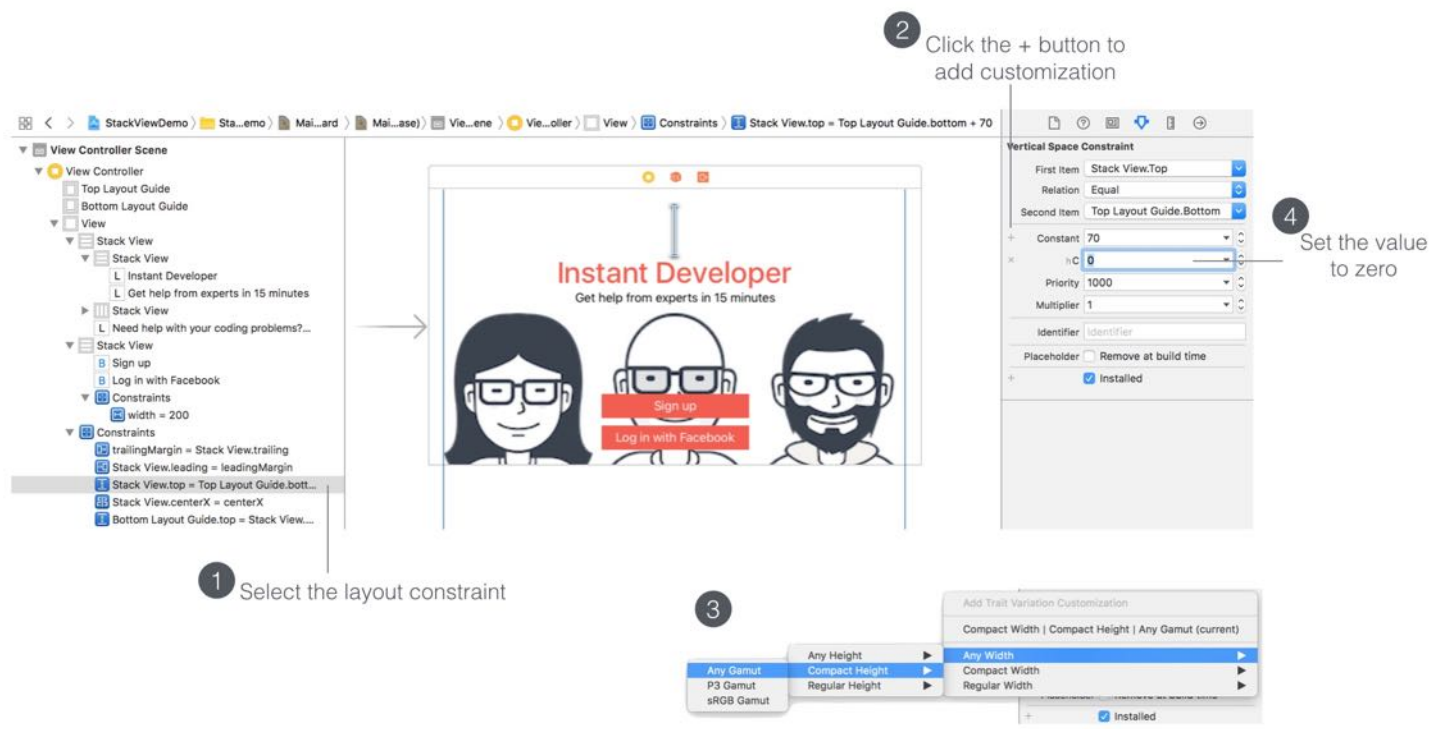


Figure 6-32. Add a Size-Class-specific constraint

By doing that, the space between the stack view and Top Layout Guide will be minimized when your iPhone is turned sideways. Preview the UI in Interface Builder or run the project on various iOS devices to see the result.

If you have run the app on iPhone 6 Plus in landscape orientation, the text under the images is not fully displayed. You can define additional constraints to resolve the issue. But a simple solution is to hide the text for iPhone in landscape orientation. There are multiple ways to hide a label or UI objects. You can write code to achieve that. Here I want to show you how to hide the label through auto layout instead.

In brief, we will define a height constraint for the "Need help with your ..." label. By setting the height to zero, the label will disappear. We will only apply this constraint for a specific size class so the label will disappear when the iPhone device is in landscape orientation.

Let's see how to implement it.

Now go back to Interface Builder and select the "Need help with your ..." label in the document outline view. Control drag from the label to itself and choose *Height* to add the constraint. Then set the constant of the height constraint to 0.

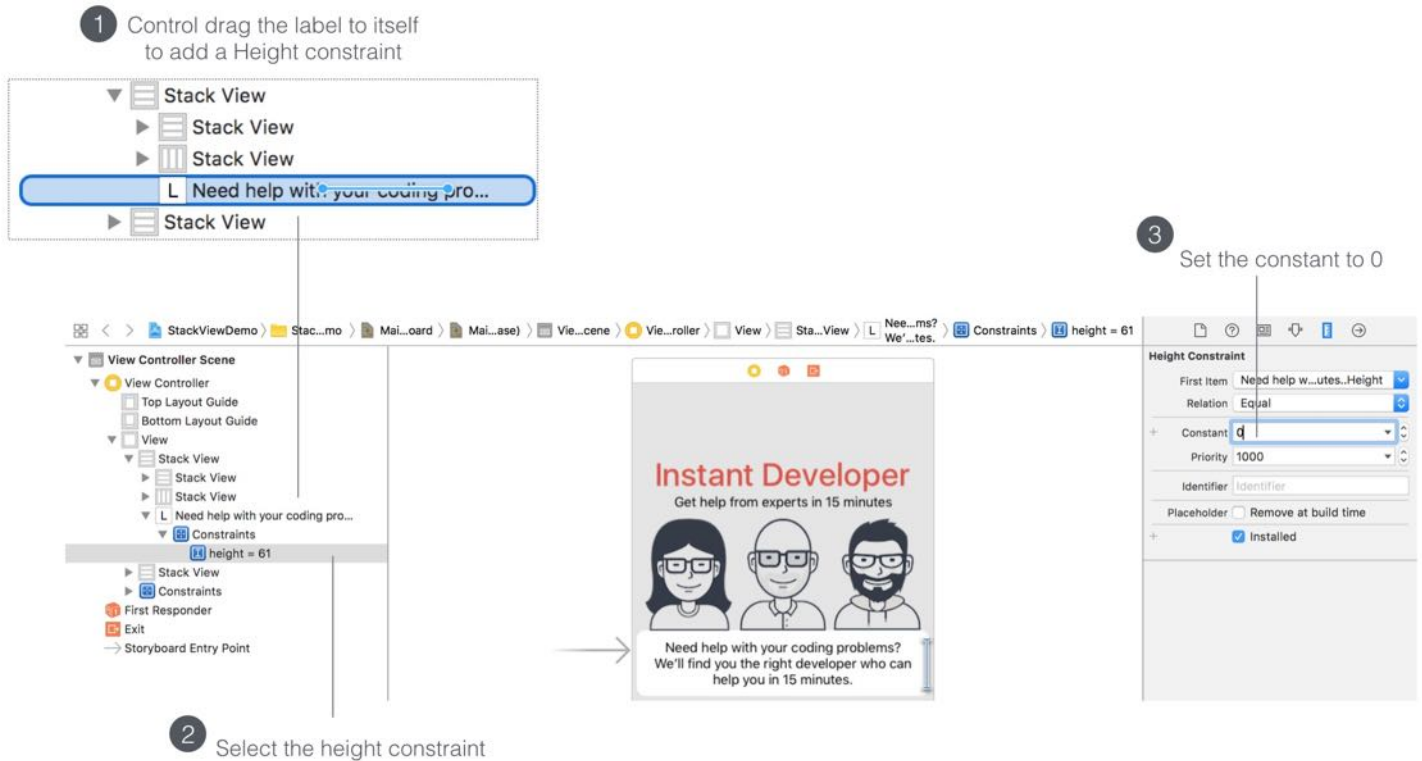


Figure 6-33. Adding a height constraint for the label

When you set the constant to zero, the label diminishes. Now that this height constraint applies to all devices, how can we activate it for iPhone in landscape orientation only?

For each constraint, you should find a *Installed* checkbox in the Attributes inspector. The constraint is active when the checkbox is checked. Now deselect the *Installed* option, the label will appear again.

To enable this constraint for iPhone in landscape orientation only, click the + button and select Any Width > Compact height > Any Gamut. Make sure the *Installed* option for hC is checked.



Figure 6-34. Adding customization for the height constraint

This will hide the label for that particular size class. In other words, it will not display the "Need help with your ..." label on all iPhone devices in landscape orientation. But don't trust my words; try it out yourself and test the app on multiple devices.

Your Exercise

To help you better understand how auto layout and size classes work, let's have another simple exercise. You are required to update some constraints and create another layout specialization for iPhone/iPad. Here are the requirements:

- For all devices, change the width of the "Sign up" and "Log in with Facebook" button from 200 to 300 points.
- For iPhone 6s and older iPhone (in portrait orientation), the font size of the "Need help with ..." label should be set to 15 points.
- For iPad (in portrait orientation), increase the spacing between the images and the labels. Set it to 30 points instead of 10 points.

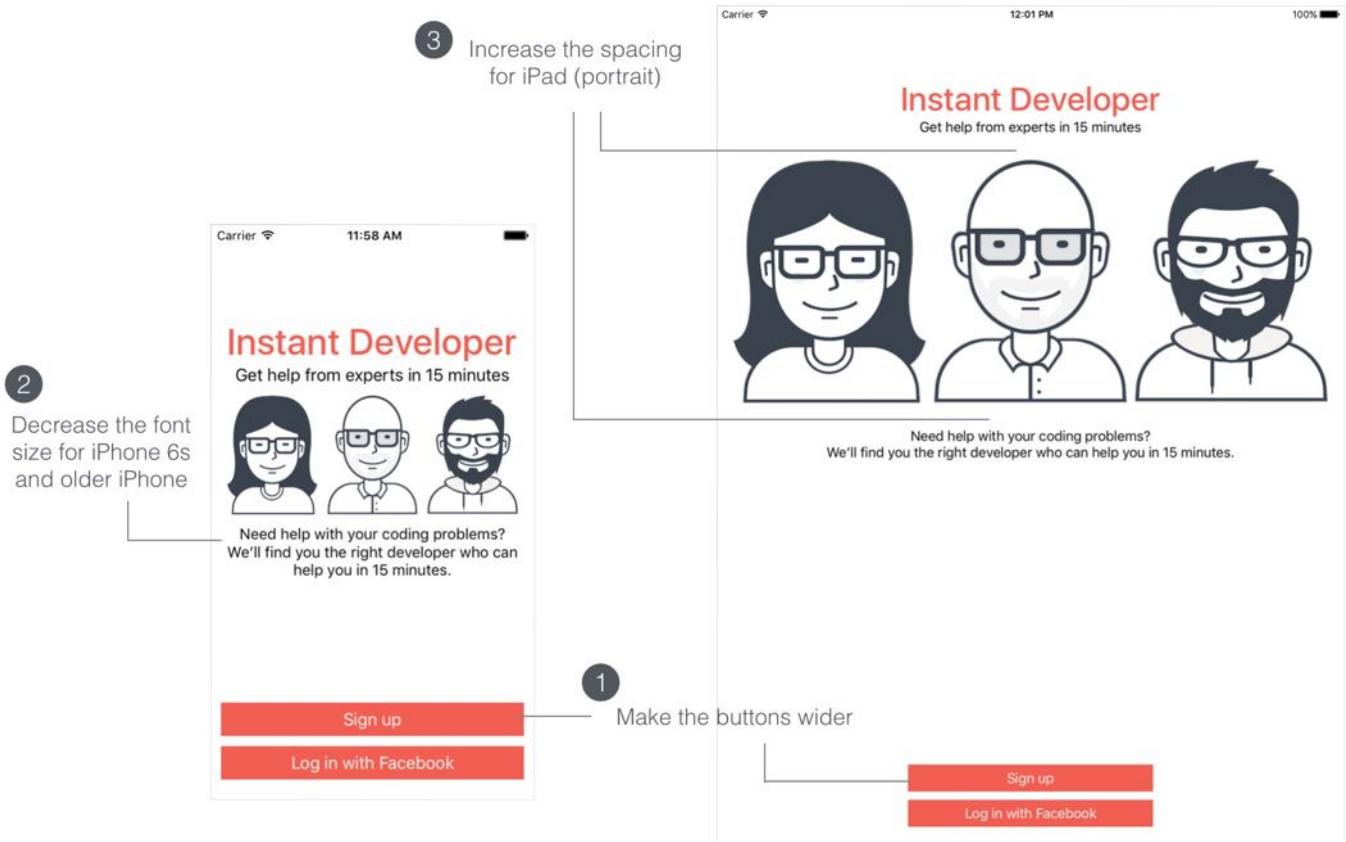


Figure 6-35. UI requirements for your exercise

Hint: When you click the + button to create constraint customization, it shows the current size class (e.g. Compact Width | Regular Height | Any Gamut) as the first option. This is the size class that you need to provide customization.

Summary

Congratulations! You have finished the chapter and learned how to build an adaptive UI using stack views and size classes.

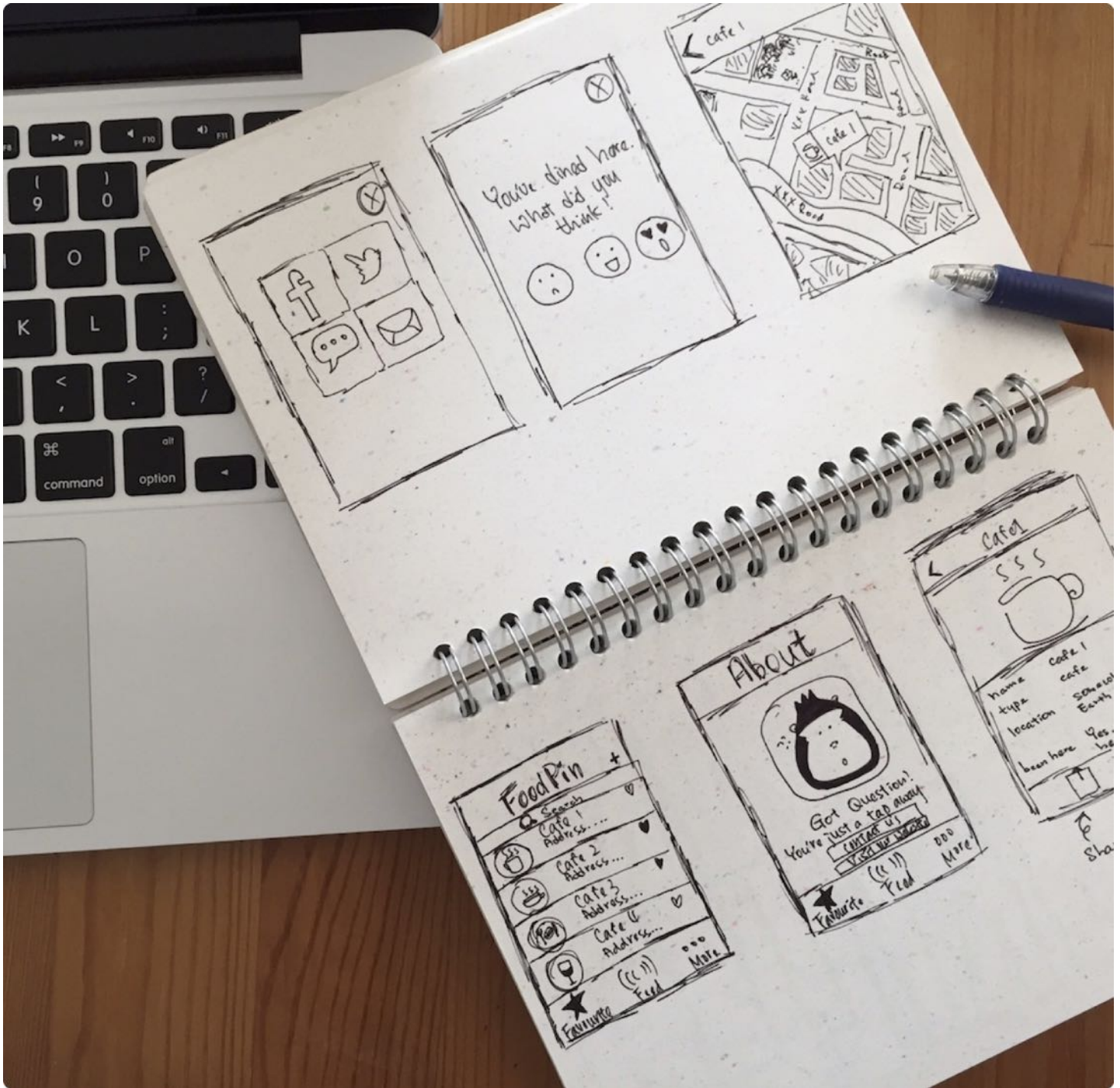
Stack views streamline the way you build user interfaces on iOS, with very minimal constraints. One question you may have is: when should you use stack views? Apple's engineers recommended developers to adopt stack views first, and then only when you need to actually use raw constraints. So throughout this book, we will design the user interfaces using stack views.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/StackViewDemo.zip>. For the solution to the exercise, you can download it from <http://www.appcoda.com/resources/swift3/StackViewExercise.zip>.

Got question? Join our Facebook group (<https://www.facebook.com/groups/appcoda>) to discuss with it with other developers.

Chapter 7

Introduction to Prototyping



If a picture is worth 1000 words, a prototype is worth 1000 meetings.

Now that you have some basic concepts of iOS programming and Interface Builder. As I always said, there is no better way to learn app development than actually creating an app. We will create a real app together in this book. However, we will not rush out to write code. Instead we will build a prototype first.

Every time I mentioned prototype to beginners, two questions pop up:

- What's a prototype?
- Why prototyping?

A prototype is an early model of a product used for testing a concept or visualizing an idea. Prototyping has been used in many industries. Before constructing a building, an architect needs to draw a plan of the building and make a model of the building. An aircraft company builds a prototype of an aircraft to test any design flaws before building and assembling an aeroplane. Software companies also build software prototypes to explore an idea before creating the actual application. In the context of app development, a prototype can be an early sample of an app which is not fully functional and contains basic UI or even sketches.

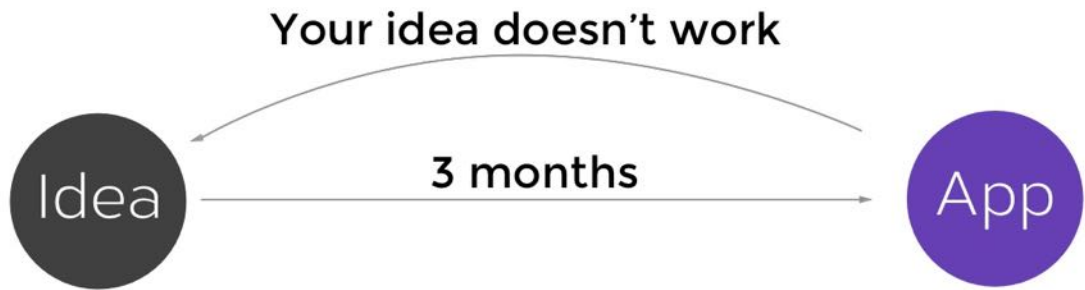
Prototyping is the process of developing a prototype and offers many advantages. First, it helps you visualize your idea and better communicate your idea to your team members and users. If you're the only developer and the only user, you may not need prototyping as you define the requirements and how the app works. Everything is in your head and you know exactly what you want and what you need to build.

However, app development rarely happens like that. You may work in a team of programmers or build an app for a client. Even if you're an indie developer, you're probably developing an app that targets for a particular group of users or all users around the globe. You have to find some ways to communicate your idea or test your idea. You can explain your idea in words, but no one wants to see another boring presentation of ideas. There is no better way than showing your users a working demo.

By creating a prototype, you can involve your users earlier and they will better understand how the app works and figure out what's missing at the early development stage.

Prototyping also allows you to test an idea without building an actual app. You can show your prototype to your potential users and get early feedback before an app is built. This saves you both time and money. Figure 7-1 illustrates the benefits of prototyping.

Without Prototyping



With Prototyping

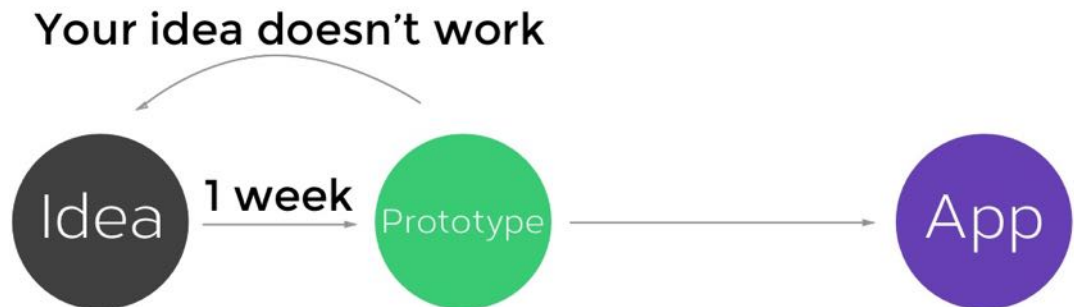


Figure 7-1. Prototyping saves you money and time

Sketching Your App Ideas on Paper

Now that you have an app idea, but how can you create a prototype for your app?

A prototype can take many forms. It can be paper-based or digital-based. I always start from

hand drawn concepts and highly recommend you use paper to sketch out your app design. It's the simplest way to create an app prototype. And paper is still the best way to quickly record all the ideas in your head.

For example, I have an idea for building a food app that allows me to save my favourite restaurants. While Yelp is good, I want to build an app for me to create a personal food guide. The app will have these features:

- List the favorite restaurant at home screen
- Create a restaurant record and import a photo from photo album as the restaurant image
- Save the restaurant locally and share it with other foodies in the world
- Show the location of a restaurant on maps
- View restaurants shared by other foodies

I think people may like this idea too. In order to test my idea, I first draw my design on paper. Some people said they're not good at drawing. You don't have to be an artist to draw your app design. The sketches that are shown in figure 7-2 are good enough to visualize your ideas and explain your app to your friends.



Figure 7-2. Draw your app on paper

Prototyping Your App Using POP

You can illustrate your app on paper. But wouldn't it be great if you can create some screen transitions so that your potential users can interact with the app prototype? There are a

number of tools for developers to prototype their apps. [POP app](#), [Marvel](#), [Proto.io](#), [Flinto](#), [Principle](#) and [InvisionApp](#) are some of the examples. I will use POP to create the app prototype but the other tools work pretty much the same.

POP app turns your hand drawings into a working prototype. It lets you capture your drawings using camera or import them from your photo album. To interact with the images, the app offers various transitions for you to link your screens up. You'll see what I mean in a while. First, install POP app on your iPhone and download the app prototype from <http://www.appcoda.com/resources/swift3/FoodPinSketches.zip>. You can unzip the images and import them to your iPhone.

POP app is very easy to use. When you launch it the first time, you will see your project listing. Click the + icon to create a new project. Give your project a name (e.g. Food Pin). Select the project when it's created. By default, the project is empty. Now click the camera icon and use camera option to capture your sketches. Alternatively, you can import the sketches from your photo album. Figure 7-3 shows a sample POP project.

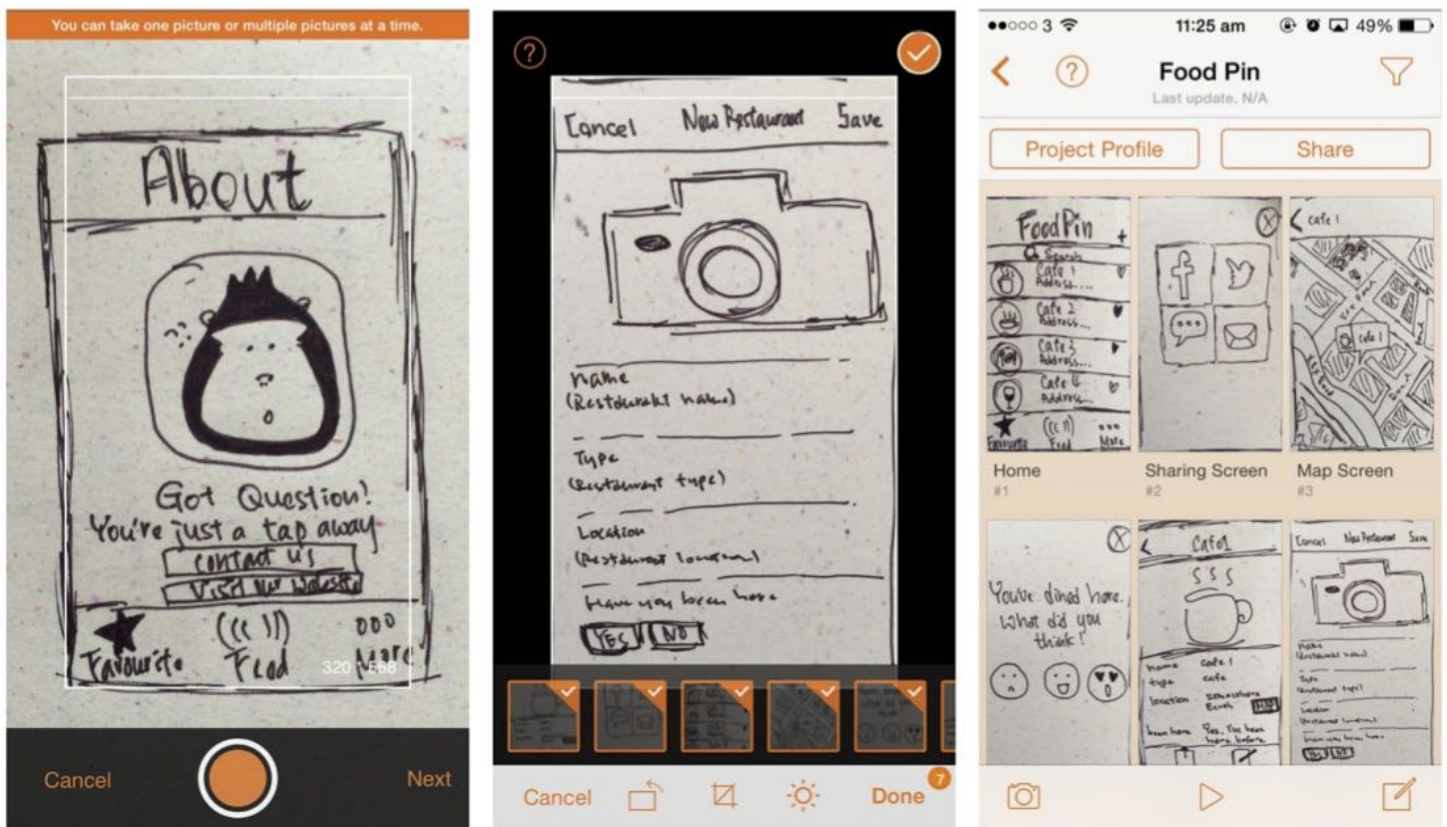


Figure 7-3. Capture your sketches using camera and create a POP project

Start with the home screen of the app and define the screen transition. POP lets you highlight a specific area of the image and specify the target page when that area is tapped. Then define the transition type including fade, next, back, rise and dismiss. Say, for the home screen, the app should navigate to the detail screen when tapping any of the records. So we highlight the records, set the transition to “next” and link it to the detail screen. Once you have made the change, tap the Play button to interact with the prototype. The app will transit to the detail screen when any of the records are tapped.

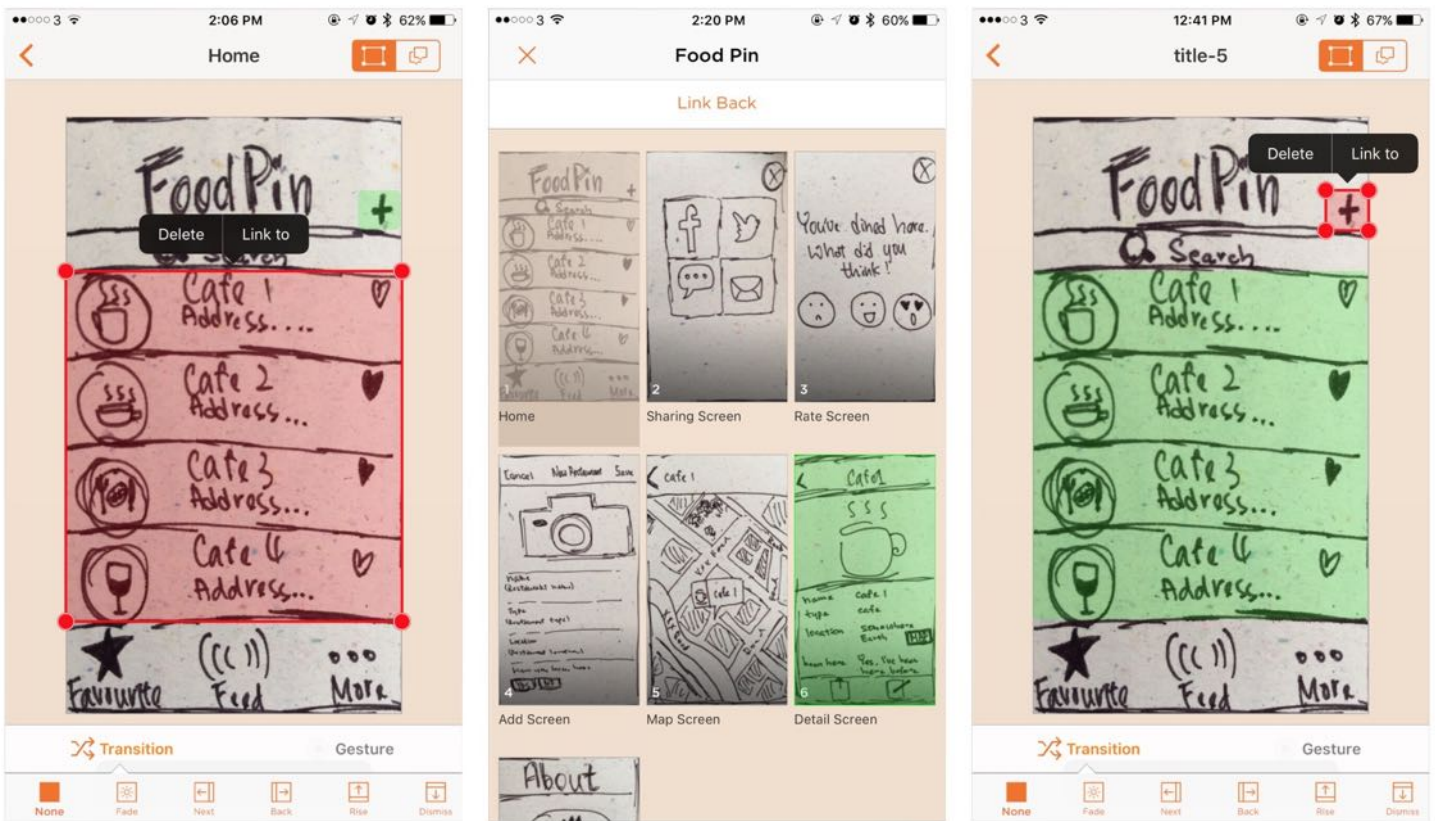


Figure 7-4. Defining screen transitions in POP

You just need to repeat the procedures to define the rest of the screen transitions. When the prototype is finished, you can share it with your team members and potential users using the Share option. Your users can try out the prototype using a web link:

<https://popapp.in/projects/542b9b22fdafo26132c03b74/preview>

This is how you lay out your idea and that allows you to solicit users' feedback as early as possible. If your users don't like the idea or the screen design, this is not a big deal. They're just sketches. We can throw away the sketches and re-create another set. Or you can fine tune some of the areas that don't work and make the prototype better. As you can see, this saves you time and money.

Popular Prototyping Tools

As said before, an app prototype can take many forms. Hand drawing is one of the many ways to create a prototype. If you're a designer, you may use Photoshop or Sketch to design an app UI and then use an app prototyping tool like [Flinto](#) to create the app prototype.

Sketch

I am a big fan of Sketch (<https://www.sketchapp.com>). Even if I am not an app designer, the tool allows me easily turn my paper sketches into a professional app design. The Sketch app comes with iOS app template to help you layout your UI. And, you can find tons of free/paid Sketch resources online (<http://www.sketchappsources.com>) that can elevate your app design. Sketch also has a companion iPhone app so you can easily view the design on a real device. However, it doesn't come with a built-in prototyping feature. What I usually do is export the app screens from Sketch and import them to other prototyping tools (e.g. InvisionApp) to build a working prototype.

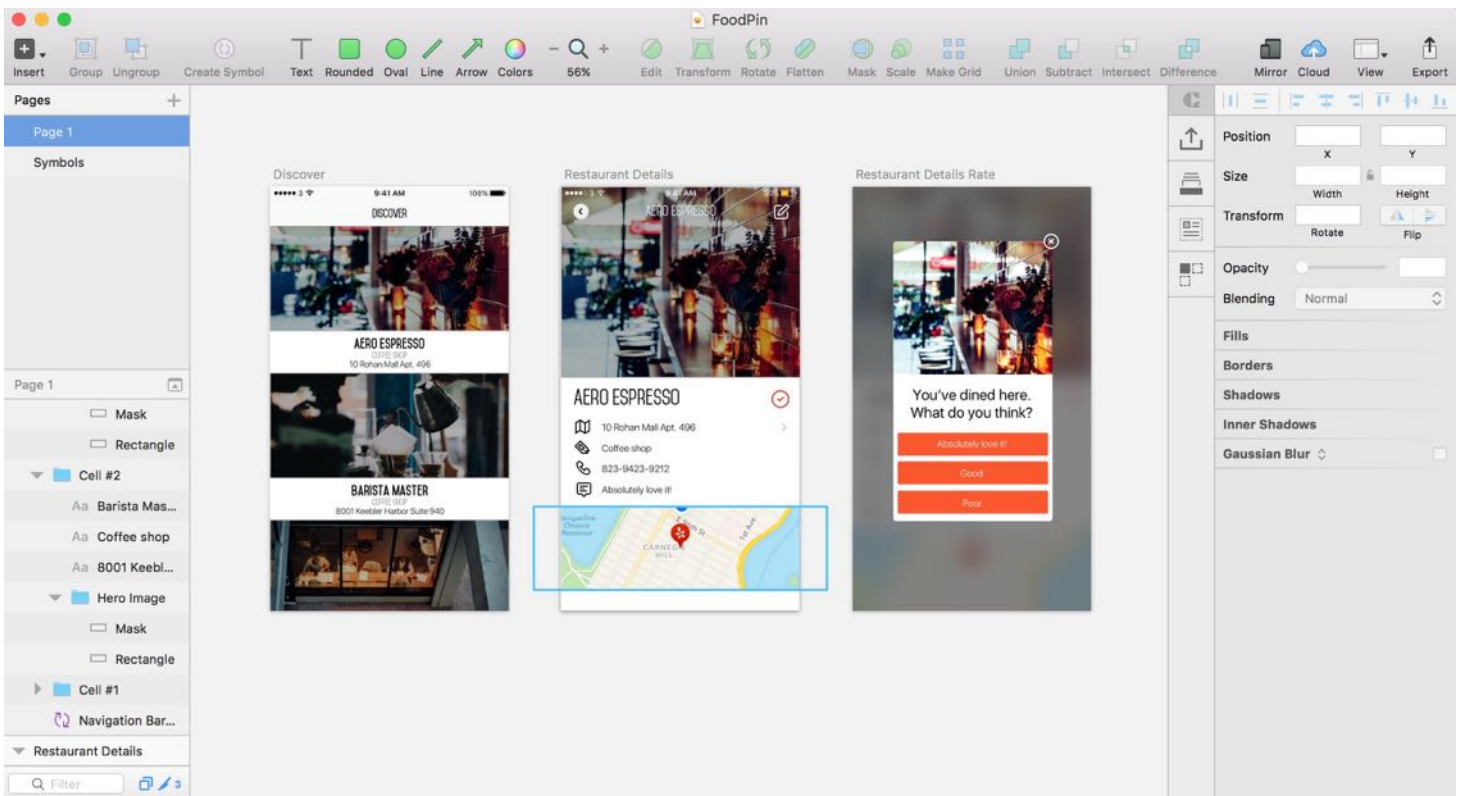


Figure 7-5. Designing the app UI using Sketch

Note: If you want to learn more about Sketch, I highly recommend you to check out Meng To's Design+Code book (<https://designcode.io>).

Adobe Experience Design

Earlier this year, Adobe launched a Sketch competitor called Adobe XD (short for Experience Design) for UI design. The company claimed it as an all-in-one tool for both web and mobile app design projects. At the time of writing, it is still in beta and the preview version is free for download.

If you've used Photoshop before, you will find Adobe XD a lot easier to use. The tool has two modes: *design* and *prototype*. In the Design mode, you can design the app UI with a set of layout tools. At a certain point, you can switch to the Prototype mode to build an interactive prototype with the app screens you just designed. It is pretty easy to connect one screen with the other. By clicking and dragging, you can create an app prototype with simple animations to demonstrate the workflow of your app.

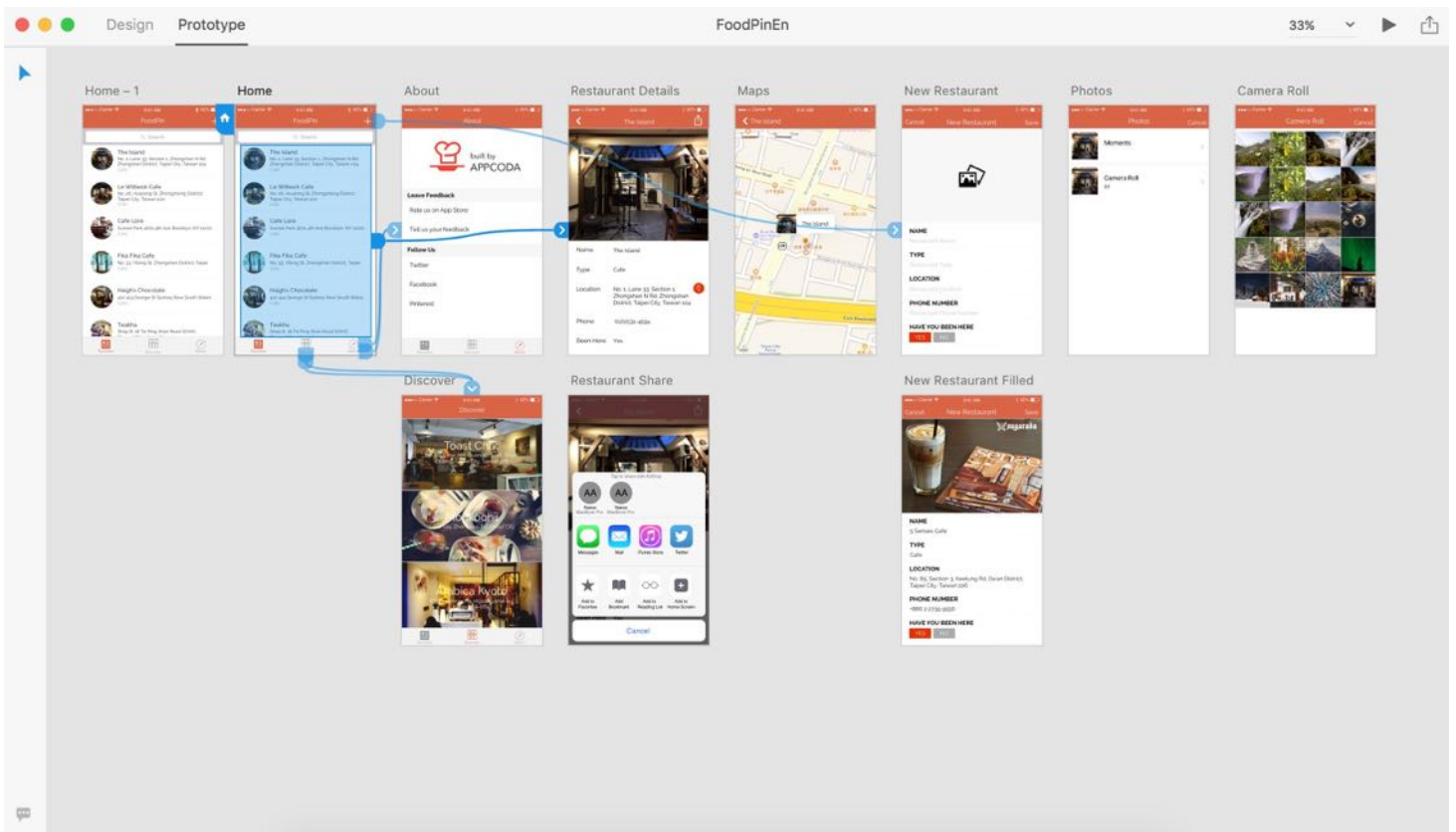


Figure 7-6. Prototyping the FoodPin App using Adobe XD

I have built a prototype demo, which is the app we're going to build in this book, using Adobe XD. You can download the project file from <http://www.appcoda.com/resources/swift3/FoodPinEn.zip>.

Keynote

Keynote! Are you kidding me? You heard me right. Apple's Keynote can also be used to make quick prototypes. Actually Apple's engineers mentioned in WWDC 2016 that they used the presentation tool to create prototype for their app projects.

Keynote shouldn't be new to you if you've used it for creating presentations. The same set of drawing tools built in Keynote allow you to design a simple app UI. [Keynotpia](#) offers mock templates to streamline your sketches. Figure 7-7 shows a sample screen created using Keynote.

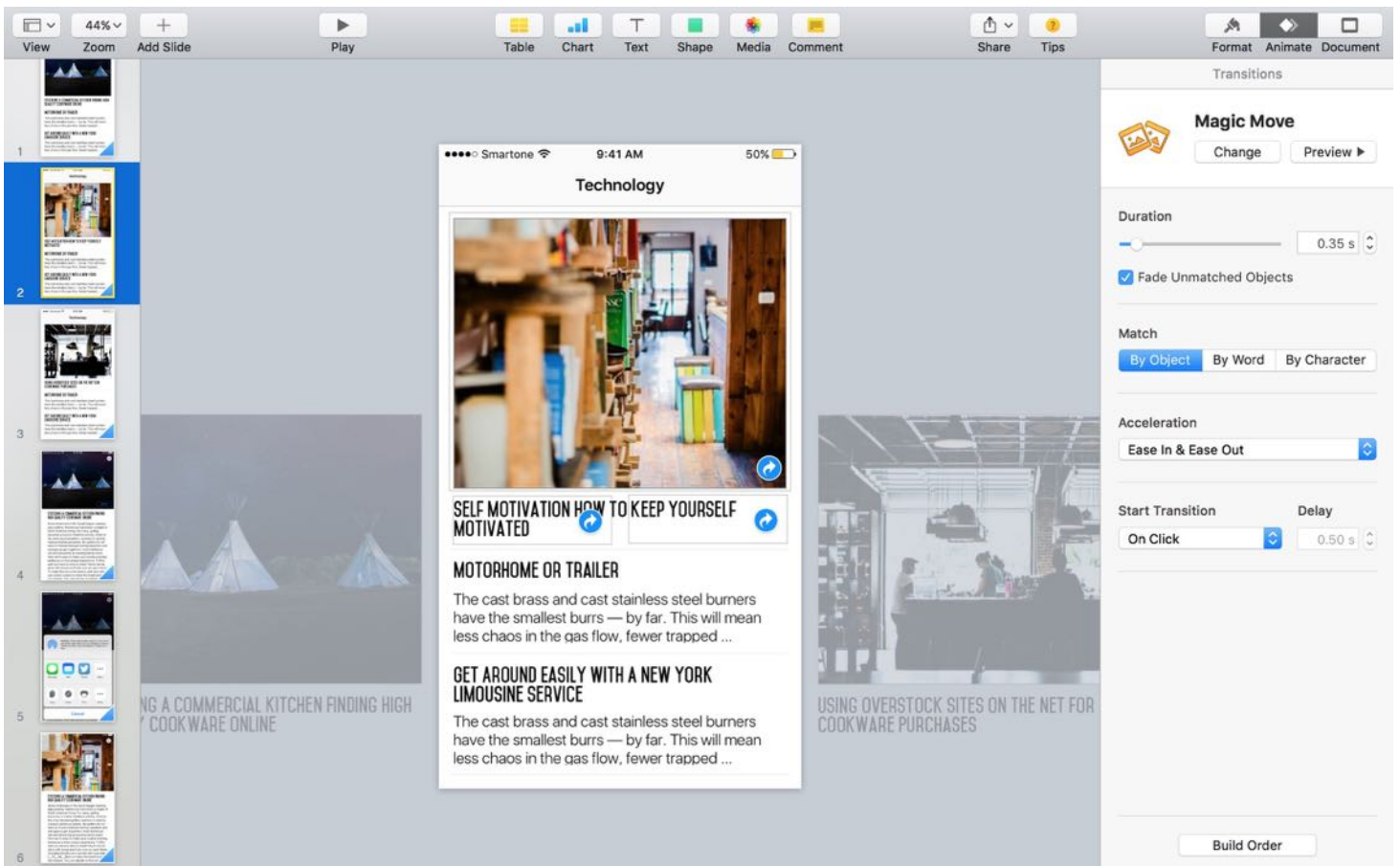


Figure 7-7. Design an app prototype using Keynote

Not only can you design the app UI, interestingly, Keynote makes it very easy to animate the static mockups. The magic move transition lets you animate the screen transition to replicate the feel of native apps. I will not go into the details about how you can create a working prototyping with Keynote. If you want to learn more about how to prototype using Keynote, you can check out this article (<http://webdesign.tutsplus.com/tutorials/how-to-demo-an-ios-prototype-in-keynote--cms-22279>)

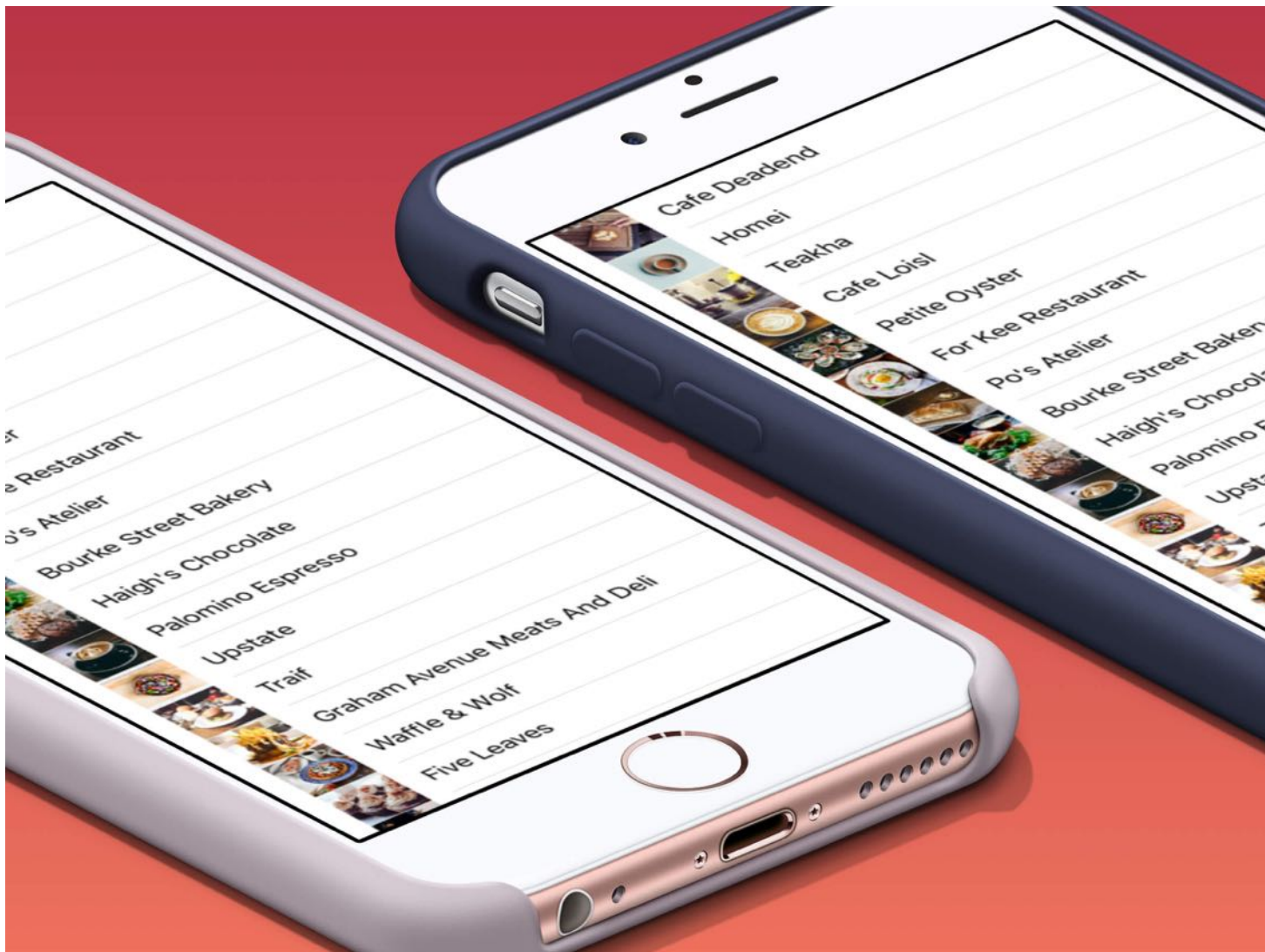
Summary

Prototyping is a common process in app development. It allows you to quickly build something workable and show users. Prototyping is used to test an idea and get feedback as early as possible. If you're building an app for a client, building a prototype lets your client clearly understand the app design.

So, no matter if you are a solo developer or a member of a development team, I want you to start prototyping today. Rather than jump right in to build your app, lay out your idea on paper first and build a simple demo using POP or other prototyping tools. This will save you a lot of time and money creating a product with no appeal.

Chapter 8

Creating a Simple Table Based App



That's been one of my mantras - Focus and Simplicity. Simple can be harder than complex: You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains.

- Steve Jobs

Now that you have a basic understanding of prototyping and our demo app, we'll work on something more interesting and build a simple table-based app using UITableView in this

chapter. Once you master the technique and the table view customization (to be discussed in the next chapter), we'll start to build the Food Pin app.

First of all, what exactly is a table view in an iPhone app? A table view is one of the most common UI elements in iOS apps. Most apps (except games), in some ways, make use of table view to display content. The best example is the built-in Phone app. Your contacts are displayed in a table view. Another example is the Mail app. It uses table view to display your mail boxes and emails. Not only designed for listing textual data, table view allows you to present the data in the form of images. The TED, Google+ and Airbnb are also good examples. Figure 8-1 displays a few more sample table-based apps. Though they look different, all in some ways utilize a table view.

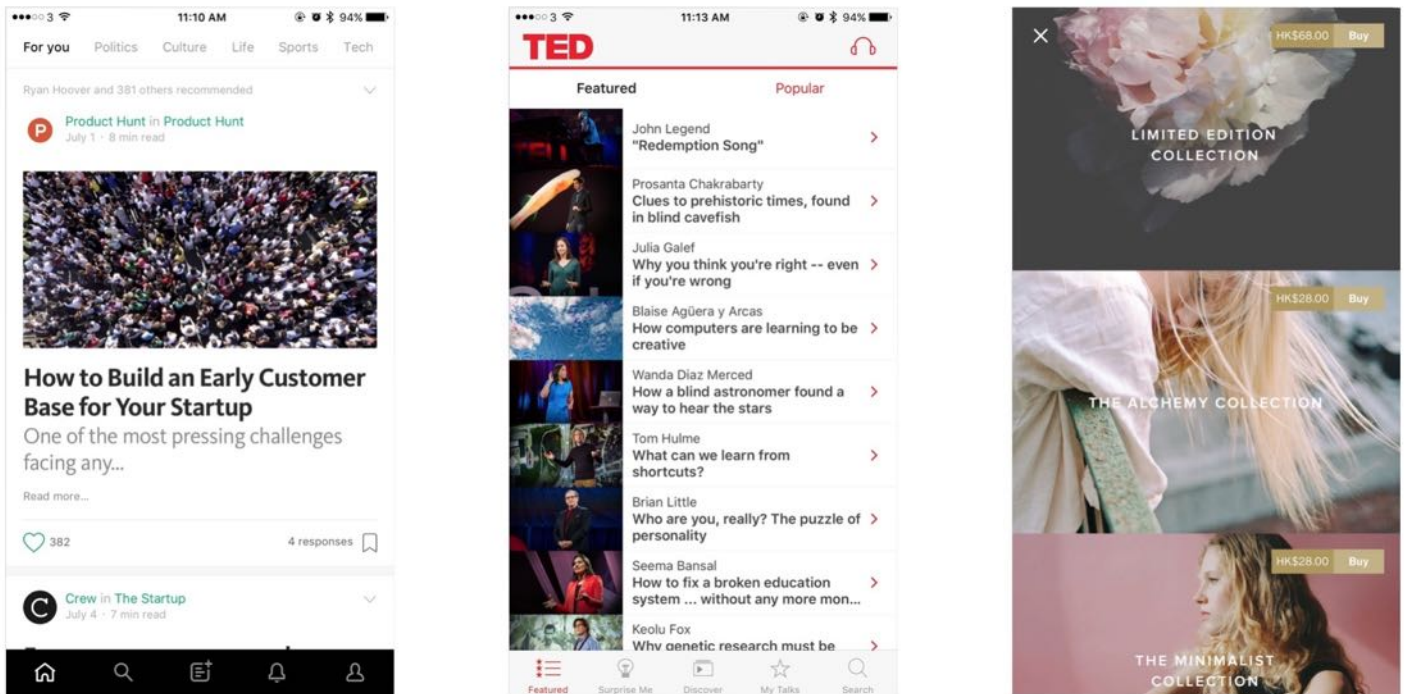


Figure 8-1. Sample table-based apps (left: Product Hunt, middle: TED, right: VSCO)

Creating a SimpleTable Project

Don't just read the book. If you're serious about learning iOS programming, stop reading. Open your Xcode and code! This is the best way to learn programming.

Let's get started and create a simple app. The app is really simple. We'll just display a list of restaurants in a plain table view. We'll polish it in the next chapter. If you haven't fired up Xcode, launch it, and create a new project using the "Single View application" template.

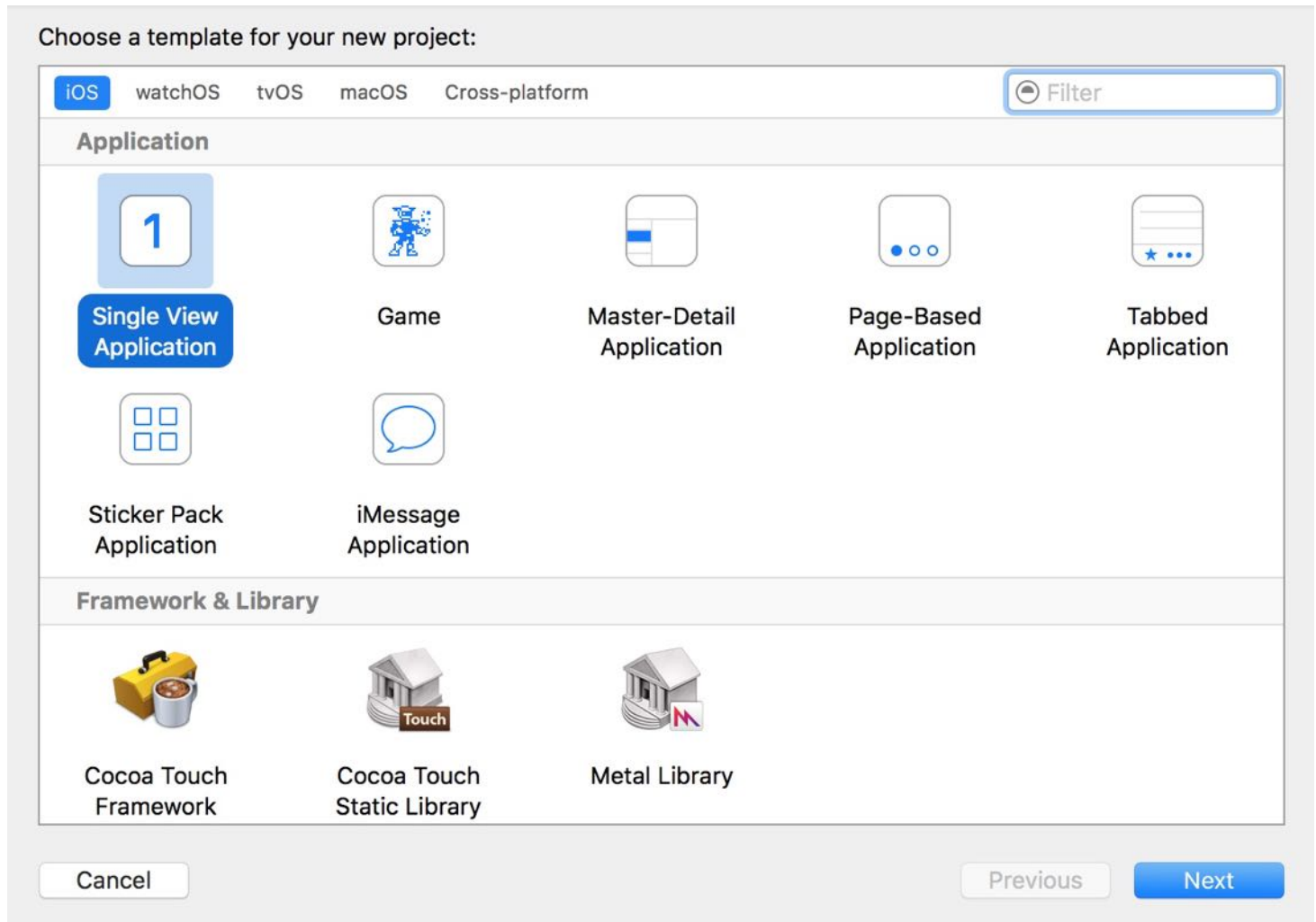


Figure 8-2. Xcode Project Template

Click "Next" to continue. Again, fill in all the required options for the Xcode project:

- **Product Name: SimpleTable** – This is the name of your app.
- **Team:** Just leave it as it is.
- **Organization Name: AppCoda** – It's the name of your organization.
- **Organization Identifier: com.appcoda** – It's actually the domain name written the other way round. If you have a domain, you can use your own domain * name. Otherwise,

you may use "com.appcoda" or just fill in "edu.self".

- **Bundle Identifier: com.appcoda.SimpleTable** - This field should be automatically generated by Xcode.
- **Language: Swift** – We'll use Swift to develop the project.
- **Devices: iPhone** – Select *iPhone* for this project.
- **Use Core Data: [unchecked]** – Do not select this option. You do not need Core Data for this simple project.
- **Include Unit Tests: [unchecked]** – Do not select this option. You do not need unit tests for this simple project.
- **Include UI Tests: [unchecked]** – Do not select this option. You do not need UI tests for this simple project.

Click "Next" to continue. Xcode then asks you where to save the *SimpleTable* project. Pick a folder on your Mac. Click "Create" to continue.

Design the User Interface

To present data in a table in an iOS app, all you need to use is the table view object. First, select `Main.storyboard` to switch to the Interface Builder editor. In the Object library, look for the `Table View` object and drag it into the view.

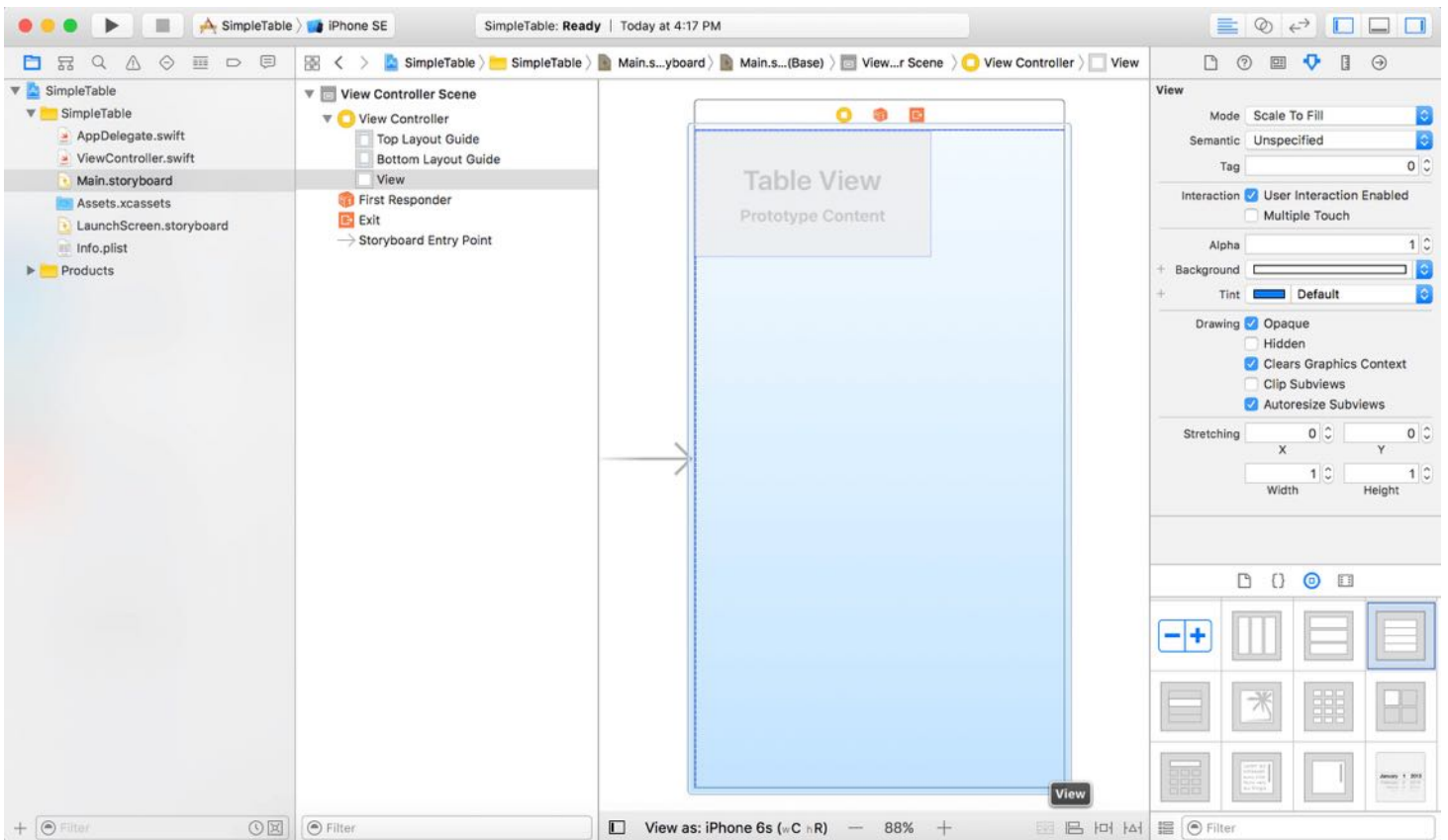


Figure 8-3. Drag a table view from the Object library to the View

Resize the table view to fit the whole view. Next, go to the Attributes inspector (if it doesn't appear in your Xcode, choose View > Utilities > Show Attributes Inspector), change the number of Prototype Cells from 0 to 1. A prototype cell will appear in the table view.

Prototype cells allow you to easily design the layout of your table view cell. You can think of it as a cell template that you can reuse it in all the table cells.

By default, a prototype cell comes with several standard cell styles including basic, right detail, left detail and subtitle. You can create your cell design or simply choose from the default style. In this example, we will use the basic style. For cell customizations, I will leave it to the next chapter.

Select the cell and then open the Attributes inspector. Change the cell style to *Basic*. This style is good enough to display a cell with both text and image. Additionally, set the identifier to *Cell*. This is a unique key for identifying the prototype cell. We'll use it later when writing code.

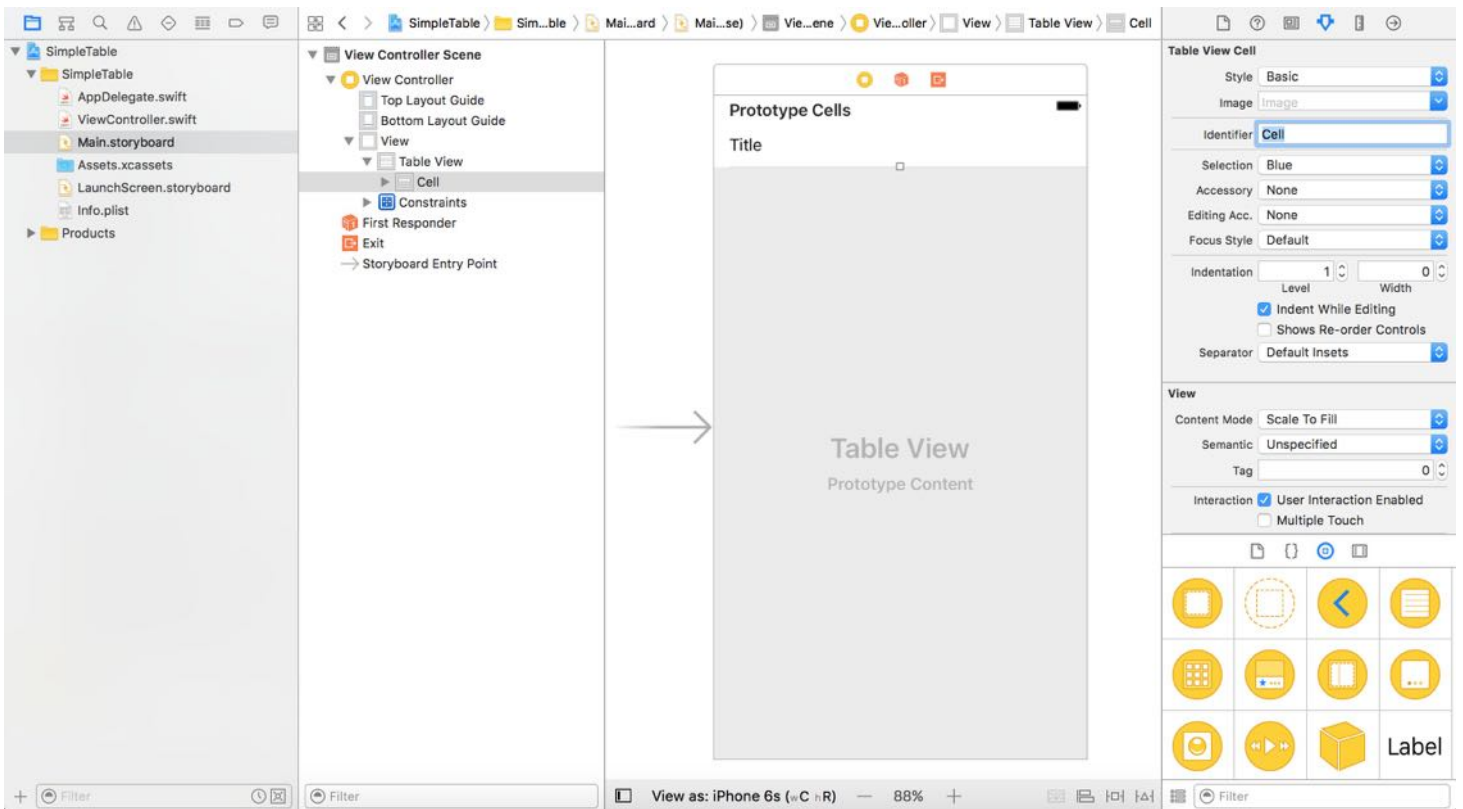


Figure 8-4. Prototype Cell in table view using Basic style

Run Your App to Have a Quick Test

Before moving on, try to run your app on iPhone 6 Plus. The Simulator screen should look like the one in figure 8-5.

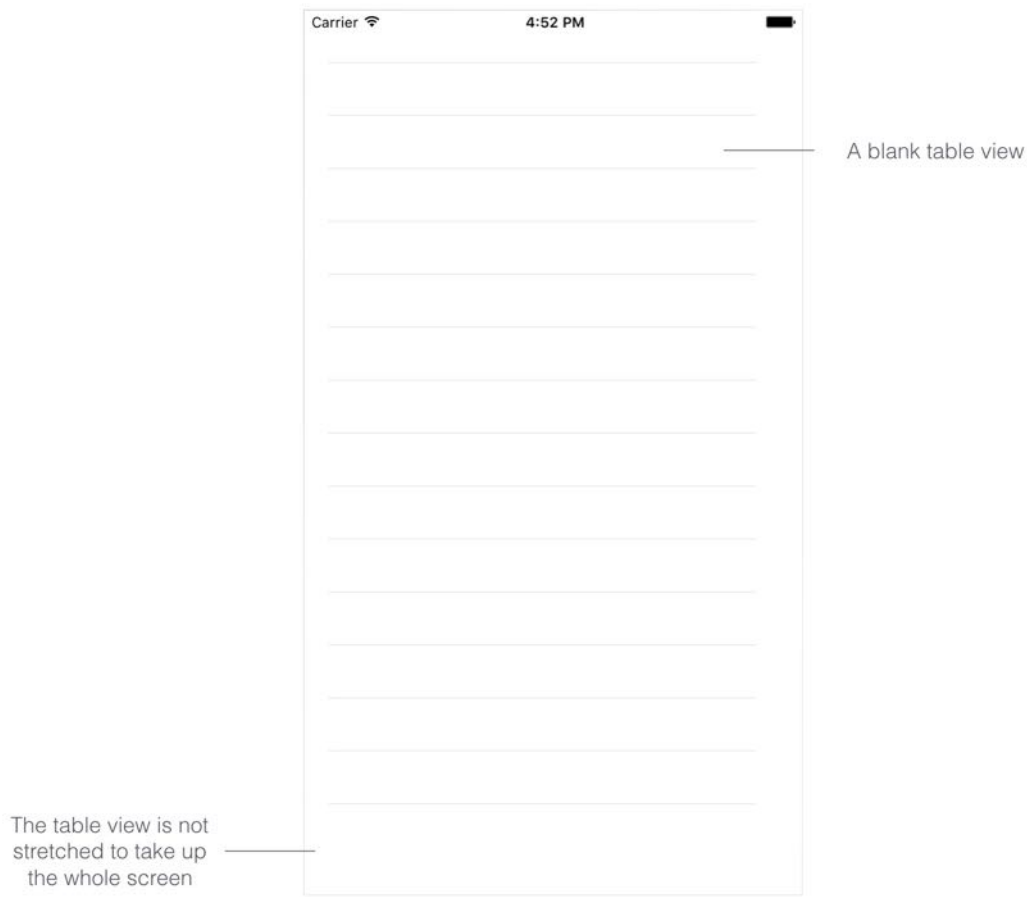


Figure 8-5. A blank table view on iPhone 6 Plus

Easy, right? You have created the table view for your app. Meanwhile, it doesn't display any data. If you take a closer look at the table view, it is not perfectly stretched to take up the whole screen. I believe you should know the reason if you thoroughly understand auto layout.

So far we haven't defined any layout constraints for the table view. This is why it is not displayed properly on every device. Now select the table view in Interface Builder, click the Pin button in the layout bar. Set the spacing of the top, left, right and bottom side to 0.

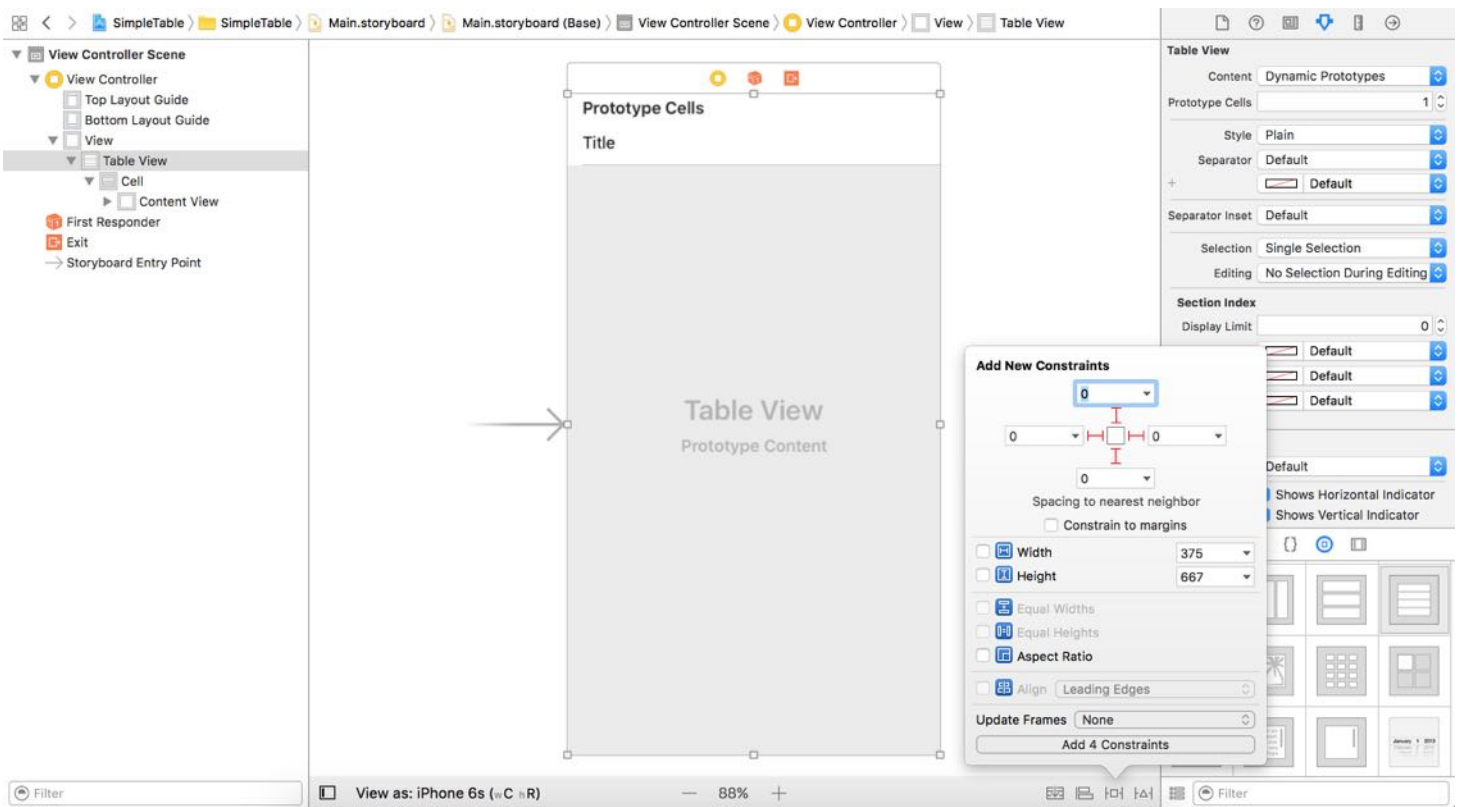


Figure 8-6. Adding layout constraint to the table view

Make sure the bars are in solid red and the *Constrain to margins* option is unchecked. Then click the *Add 4 Constraints* button to add the constraints.

Here we define 4 spacing constraints for each side of the table view. Here, the *Constrain to margins* option is unchecked. In this case, the spacing constraints will be relative to the edge of the view instead of the margin. This ensures that all sides of the table view will stretch to the edge of the view. In other words, your table view will automatically resize to fit all screen sizes.

You can run the project again. The table view should now support all screen sizes. With the UI design ready, let's move onto the core part and write some code to insert the table data.

UITableView and its Protocols

I mentioned before that we deal with foundation classes provided by the iOS SDK. These classes are organized into what-so-called *frameworks*. The UIKit framework is one of the most commonly used frameworks.

It provides classes to construct and manage your app's user interface. All objects listed in the Object library of Interface Builder are provided by the framework. The Button object you used in the HelloWorld app, and the Table View object we're now working on are from the UIKit framework. While we use the term *Table View*, the actual class is *UITableView*. Simply put, every UI component in the Object library has a corresponding class. You can click any item in the Object Library and reveal the actual class name in the pop-over menu.

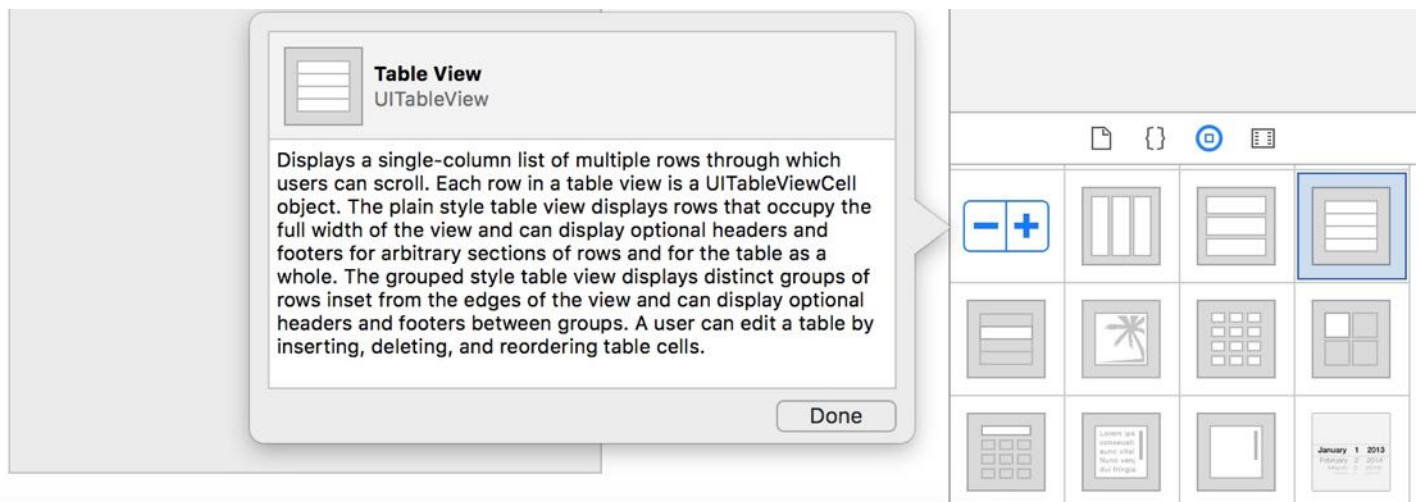


Figure 8-7. Click the object in the Object library to reveal the actual class name in UIKit framework

I intentionally leave out the discussion of classes and methods to later chapters. If you have no ideas what a class is, don't worry. Just think of it as a code template. I will explain it to you in later chapters.

Now that you have an idea about the relationship between Table View and UITableView. We'll write some code to insert the table data. Select `viewController.swift` in the project navigator to open the file in the editor pane. Append `, UITableViewDataSource, UITableViewDelegate` after `UIViewController` to adopt the protocols.

As soon as you insert the code after `UIViewController`, Xcode detects an error. The red exclamation mark indicates that there is a problem. Click the little exclamation mark on the left side of the editor. Xcode will highlight the line of code, and display a message telling you the details of the issue. The message is helpful telling the cause of the problem, but it doesn't show you the solution.



Figure 8-8. Implementing UITableViewDataSource and UITableViewDelegate protocols

So, what does it mean by "Type UIViewController does not conform to protocol UITableViewDataSource"?

The UITableViewDelegate and UITableViewDataSource are known as protocols in Swift. To display data in a table view, we have to conform to a set of requirements, defined in the protocols. Here, the ViewController class is the one that adopts the protocol, and implements all the mandatory methods.

It may be confusing. What are these protocols? Why protocols?

Well, let's say you're starting a new business. You hire a graphic designer to design your company logo. He's a skilled designer capable of creating any kind of logo. But he can't start the logo design right away. In the least, you need to provide him with some requirements such as company name, color preference, business nature before he can create a logo. However, you're busy. You delegate the task to your personal assistant, and let her deal with the designer, providing him with the logo requirements.

In iOS programming, the UITableView class is just like the graphic designer. It's flexible enough to display various kinds of data (e.g. image, text) in table form. You can use it to display a list of countries or contact names. Or in our project, we'll be presenting a list of restaurants with thumbnails.

But before UITableView can display the data for you, it requires someone to provide some basic information like:

- how many rows do you want to display in the table view?
- what is the table data? For example, what do you want to display for row 2? What do you want to display in row 5?

That "someone" is known as delegate object. In the above analogy, the personal assistant is the delegate object. In iOS programming, it also applies the concept of delegation usually known as *delegation pattern*. An object relies on another object to perform a specific task. In our project, the ViewController is the delegate that provides the table data. Figure 8-9 illustrates the relationship of UITableView, the protocols and the delegate object.

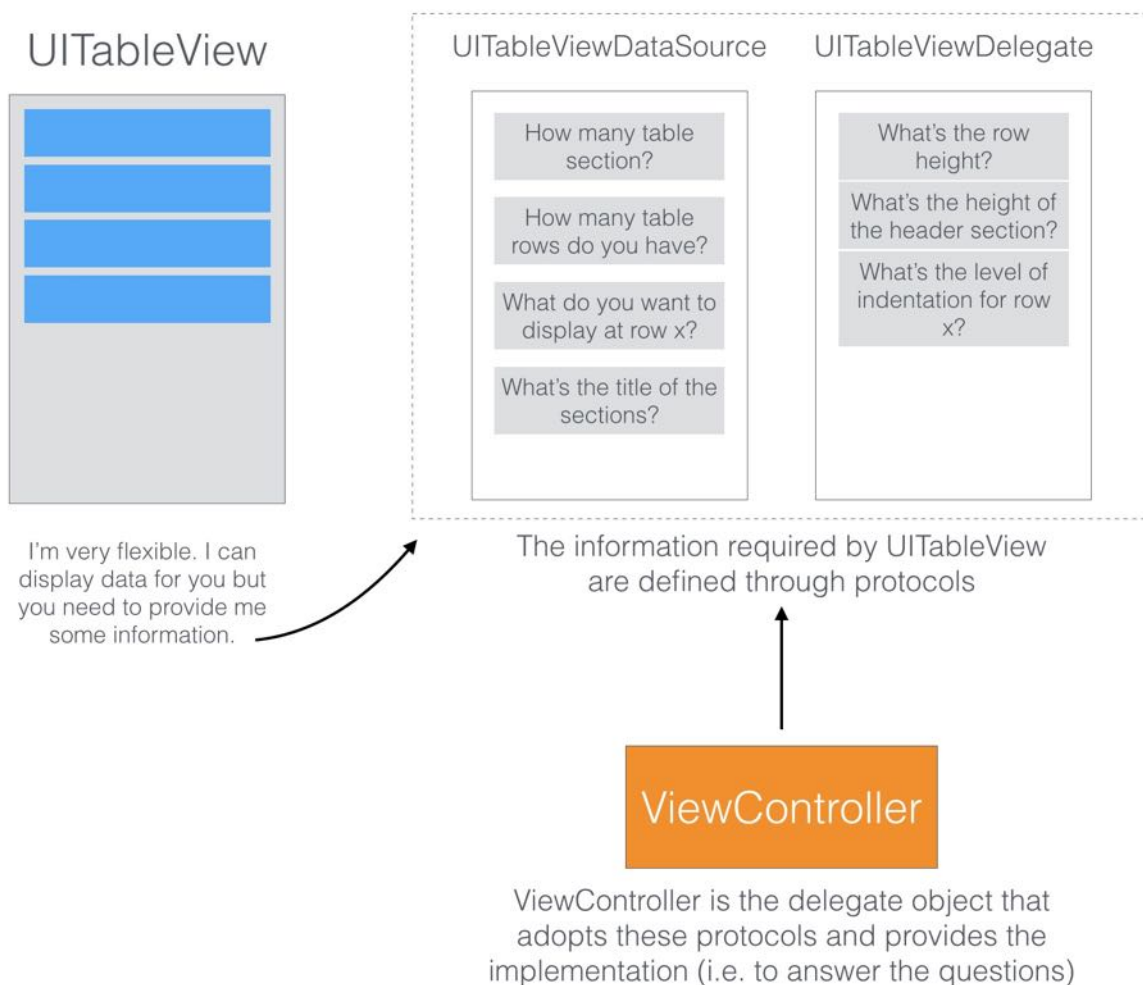


Figure 8-9. The relationship of UITableView, the protocols and the delegate object

So how do you tell UITableView what data to display? The UITableViewDataSource protocol is the key. It's the link between your data and the table view. The protocol defines a list of

methods that you have to adopt. Here are the two required methods for the

`UITableViewDataSource` protocol:

- `tableView(_:numberOfRowsInSection:)`
- `tableView(_:cellForRowAtIndexPath:)`

All you need is to provide an object that implements the above methods, so that `UITableView` knows the total number of rows to display and the data for each row. The protocol also defines some optional methods that we'll not discuss it here.

The `UITableViewDelegate` protocol, on the other hand, deals with the appearance of the `UITableView`. All methods defined in the protocols are optional. They let you manage the height of a table row, configure section headings and footers, re-order table cells, etc. We will not change any of these methods in this example. We will leave that for a later chapter.

With some basic knowledge of protocols, let's continue to code the app. Select

`ViewController.swift` and declare a variable for holding the table data. Name the variable `restaurantNames` and insert the following line of code in the class:

```
var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "Petite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery", "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Avenue Meats And Deli", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional", "Barrafina", "Donostia", "Royal Oak", "CASK Pub and Kitchen"]
```

In this example, we use an array to store the table data. The syntax of declaring an array in Swift is similar to that in Objective C. The values are separated by commas and surrounded by a pair of square brackets.

In Swift, use `let` to make a constant and `var` to make a variable. It's so simple, as compared to Objective-C. All items in an array are of the same type (e.g. `String`). And thanks to Swift's type inference feature, you do not need to specify the array type. It is automatically detected by Swift. Swift can infer that `String` is the type to use for the `restaurantNames` variable.

Arrays for Absolute Beginners

An array is a fundamental data structure in computer programming. You can think of an array as a collection of data elements. Consider the `restaurantNames` array in the above code, it represents a collection of `String` elements. You may visualize the array like this:



Figure 8-10. restaurantNames array

Each of the array elements is identified or accessed by an index. An array with 10 elements will have indices from 0 to 9. That means `restaurantNames[0]` returns the first item of the array.

Let's continue to code. We adopt the two required methods of the `UITableViewDataSource` protocol.

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    // Return the number of rows in the section.
    return restaurantNames.count
}

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

    // Configure the cell...
    cell.textLabel?.text = restaurantNames[indexPath.row]

    return cell
}
```

The first method is used to inform the table view the total number of rows in a section. You can simply call the `count` method to get the number of items in the `restaurantNames` array.

Quick note: A table view can have multiple sections but there is only one by default.

The second method will be called every time a table row is displayed. By using the `indexPath` object, we can get the current row (`indexPath.row`). Here what we did is to retrieve the indexed item from the `restaurantNames` array, and assign it to the text label (`cell.textLabel?.text`) for display.

Okay, but what is `dequeueReusableCell` in the second line of code?

The `dequeueReusableCell` method is used for retrieving a reusable table cell from the queue with the specified cell identifier. The `cell` identifier is the one we defined earlier in Interface Builder.

You want your table-based app to be fast and responsive even when handling thousands of rows of data. If you allocate a new cell for each row of data instead of reusing one, your app will use excess memory and may result in a sluggish performance when user scrolls the table view. Remember every cell allocation has a performance cost, especially when the allocation happens over a short period of time.

The screen real estate of iPhone is limited. Even if your app needs to display 1,000 records, the screen may only be able to fit 10 table cells at most. Therefore, why on earth would the app allocate a thousand table view cells instead of creating 10 table cells and reuse them? This would save tons of memory and make table view work more efficiently. For performance reasons, you should reuse table view cells instead of creating new ones.

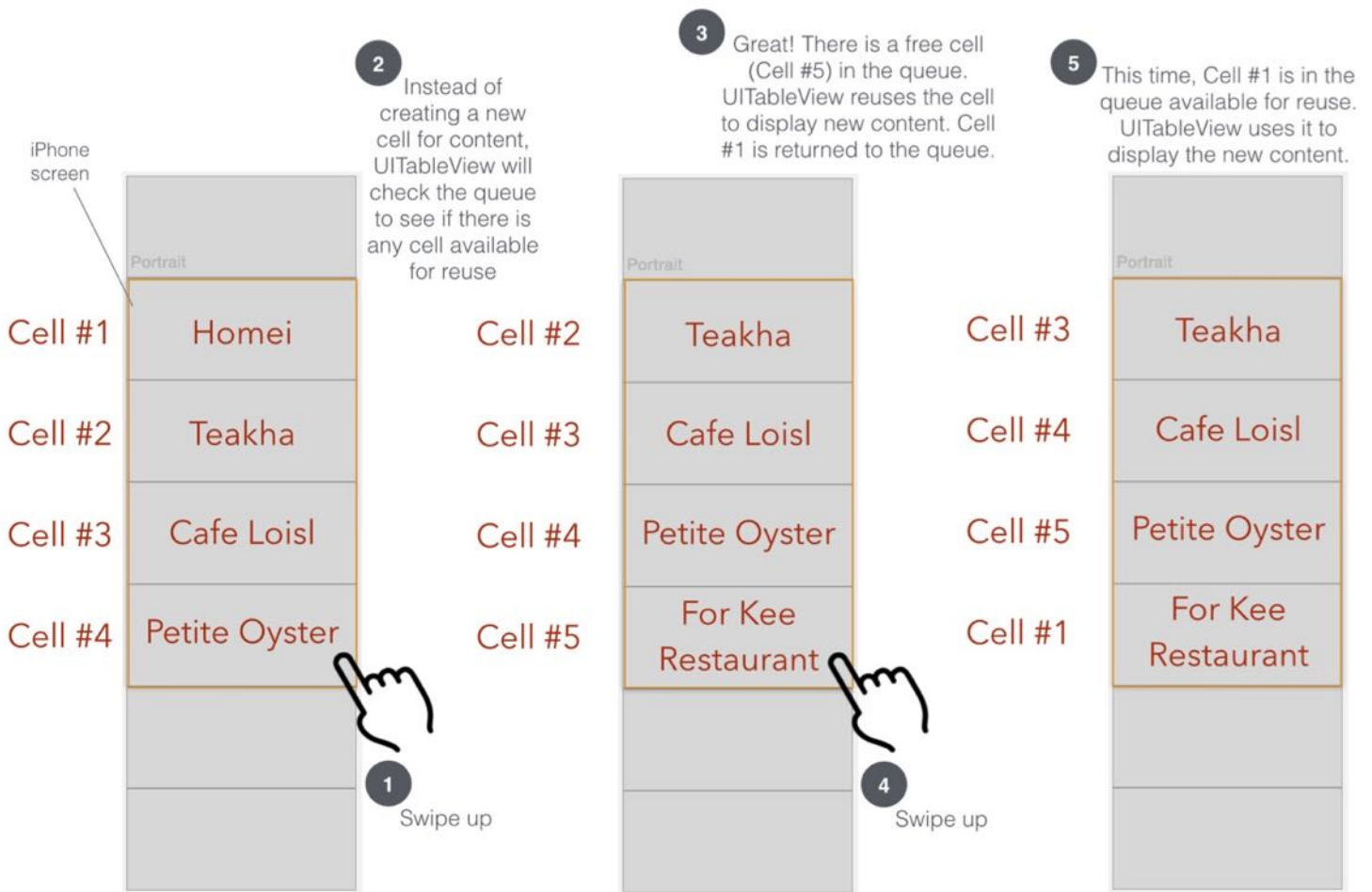


Figure 8-11. Illustration showing how UITableView reuses table view cells

Now, let's hit the "Run" button and test your app. Oops! The app still displays a blank table view just like before.

Why is it the table view still not showing the content as expected? We've written the code for displaying table data and implemented all required methods. But why the table view didn't show the content as expected?

There is still one thing left.

Connecting the DataSource and Delegate

Even though the `viewController` class has adopted the `UITableViewDelegate` and `UITableViewDataSource` protocols, the `UITableView` object in storyboard has no idea about it.

We have to tell the `UITableView` object that `viewController` is the delegate object for the data source.

Go back to `Main.storyboard`. Select the table view. Press and hold the control key, drag from the table view to the View Controller object in the document outline view.

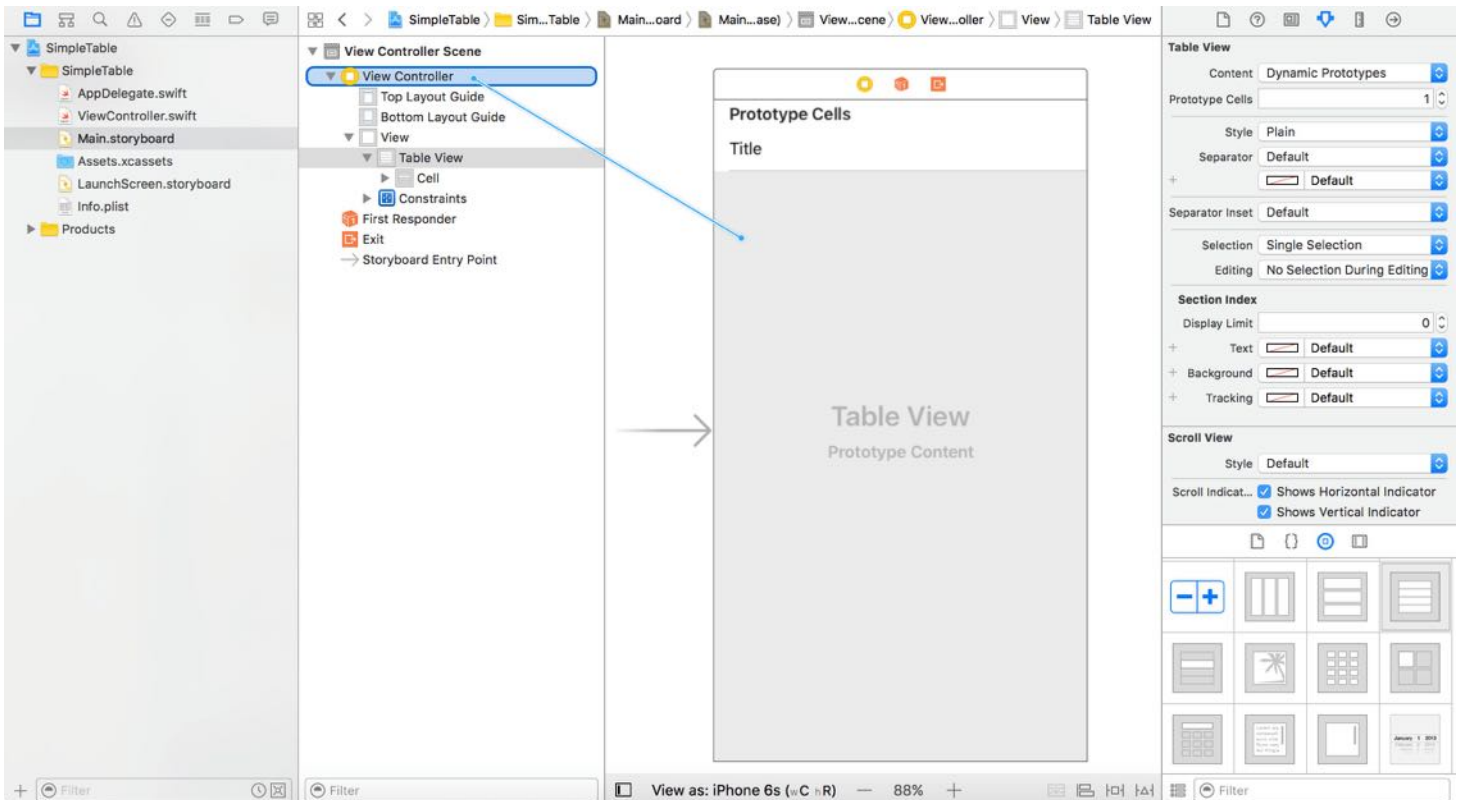


Figure 8-12. Connecting Table View with its Datasource and Delegate

Release both buttons. In the pop-over menu, select `dataSource`. Repeat the above steps and make a connection with `delegate`.

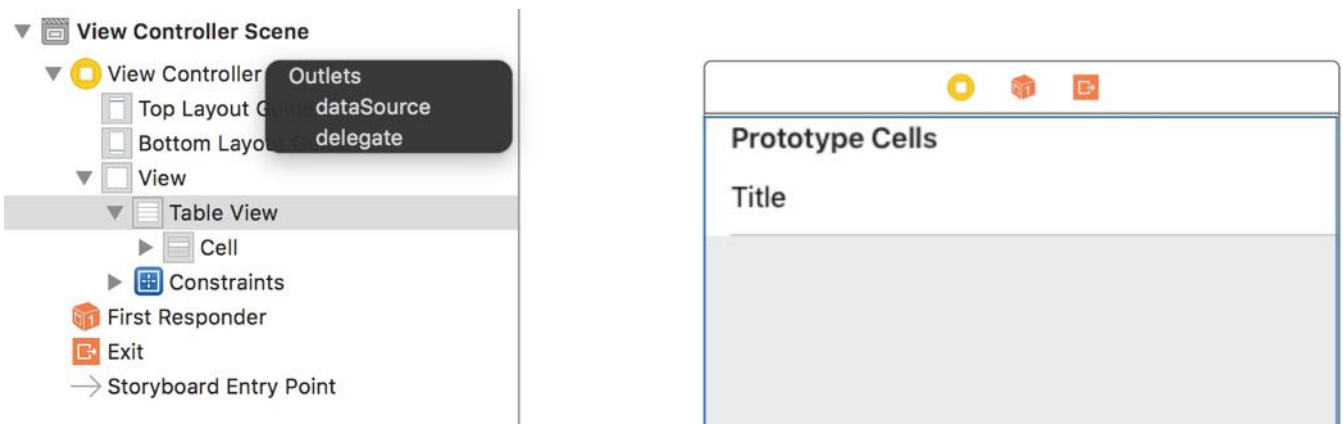


Figure 8-13. Selecting the dataSource and delegate outlets

That's it. To ensure the connections are linked properly, you can select the table view again. Click the Connections Inspector icon in the Utility area to reveal the existing connections. Alternatively, you can right-click the table to reveal the connections as well.

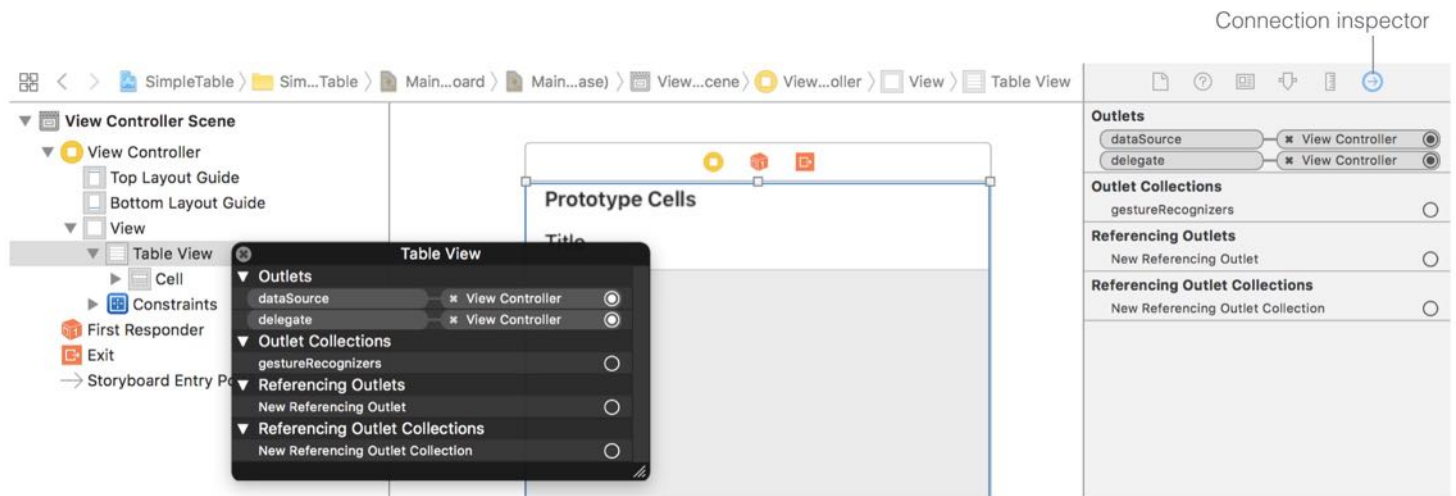


Figure 8-14. Two ways to reveal the connections

Test Your App

Finally, you're ready to test your app. Simply hit the *Run* button and load your app in the simulator. Your app should now show a list of restaurant names.

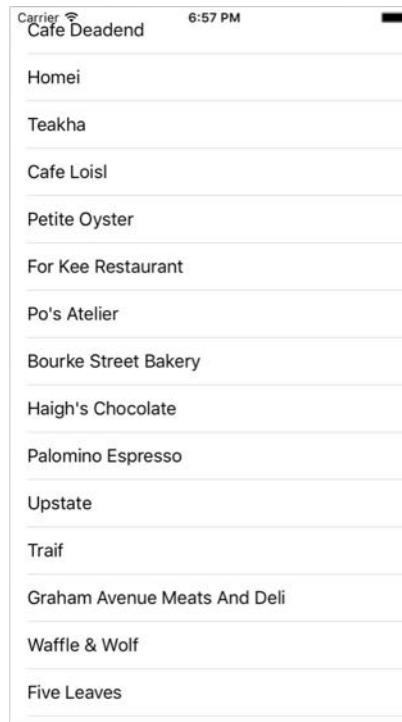


Figure 8-15. SimpleTable App

Quick tip: To scroll through the table in the iOS simulator, simply & hold on the table, drag it up and down to move it.

Add Thumbnail to Table View

Wouldn't it be great to add an image to each row? UITableView makes it extremely easy to do this. You just need to add a line of code to insert a thumbnail in each row.

First, download the sample images from <http://www.appcoda.com/resources/swift3/simpletable-image1.zip>. The zipped archive contains three image files. Unzip the file, and drag the images from Finder to the asset catalog (Assets.xcassets).

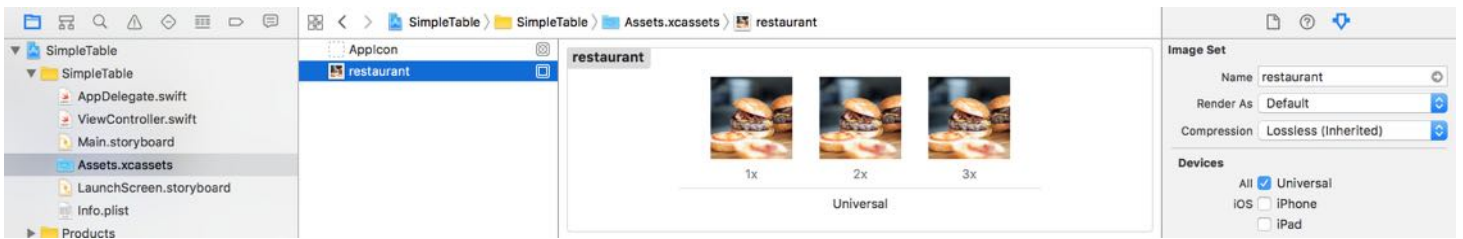


Figure 8-16. Adding images to the asset catalog

Now edit `viewController.swift` and add the following line of code in the `tableView(_:cellForRowAtIndexPath:)` method. Put it right before `return cell`:

```
cell.imageView?.image = UIImage(named: "restaurant")
```

The `UIImage` class provided by the UIKit framework lets you create images from files. It supports various image formats such as PNG, GIF and JPEG. You can pass the name of the image (file extension is optional) and the class will load the image from the asset catalog.

Earlier we selected to use the `Basic` cell style for the table view cell. This type of cell style already comes with a default area for display images or thumbnails. This line of code instructs `UITableView` to load the image and display it in the image view of the table cell. Now, hit the "Run" button again and your SimpleTable app should display the image in each row.

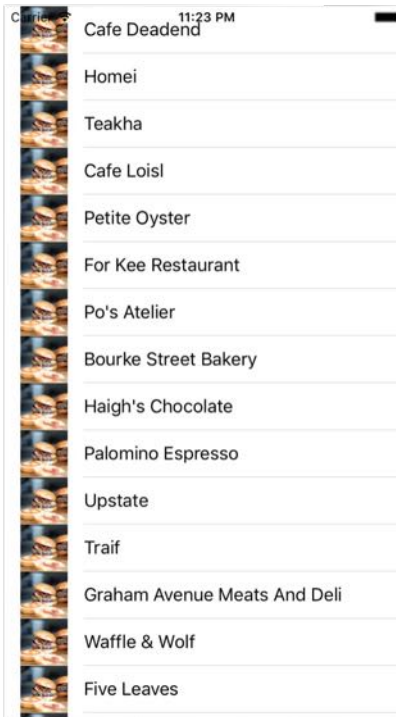


Figure 8-17. SimpleTable app with images

Hide the Status Bar

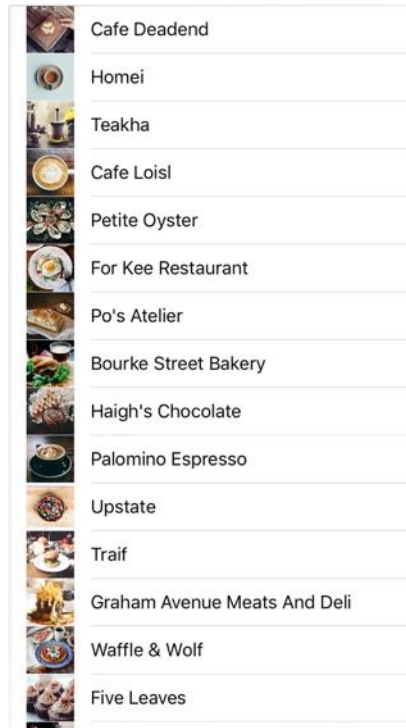
The content of table view now overlaps with the status bar. This doesn't look good. An easy remedy is to hide the status bar. You can control the appearance of the status bar on a per view controller basis. If you don't want to show the status bar in a particular view controller, simply add the following lines of code:

```
override var prefersStatusBarHidden: Bool {  
    return true  
}
```

The `prefersStatusBarHidden` property indicates whether the view controller should hide the status bar. By default, the value is set to `false`. To hide the status, we have to override the default value and set it to `true`. Insert the code to the `ViewController` class and test the app again. You should have a full-screen table view without status bar.

Your Exercise

The demo app displays the same image for all table cells. Try to tweak the app such that it shows a different image in each cell. You can download the image pack from <http://www.appcoda.com/resources/swift3/simpletable-image2.zip>. Figure 8-18 shows the resulting screen.


















	Cafe Deadend
	Homei
	Teakha
	Cafe Loisl
	Petite Oyster
	For Kee Restaurant
	Po's Atelier
	Bourke Street Bakery
	Haigh's Chocolate
	Palomino Espresso
	Upstate
	Traif
	Graham Avenue Meats And Deli
	Waffle & Wolf
	Five Leaves

Figure 8-18. Displaying different images in the demo app

Quick note: In case you do not know how to complete the exercise, no worries. I will go through it with you in the next chapter.

Credit: The images used in this demo project are provided by unsplash.com.

Summary

Table view is one of the most commonly used elements in iOS programming. If you thoroughly understood the materials and built the app, you should have a good idea of how to create your own table view.

I tried to keep everything simple in the demo app. In a real world app, the table data is

generally not 'hard-coded'. Usually, it's loaded from a file, database or somewhere else. We'll talk about that later. Meanwhile, make sure you thoroughly understand how the table view works. Otherwise, go back to the beginning and study the chapter again.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/SimpleTable.zip>.

Chapter 9

Customize Table Views Using Prototype Cell



I think that's the single best piece of advice: Constantly think about how you could be doing things better and questioning yourself.

- Elon Musk, Tesla Motors

In the previous chapter, we created a simple table-based app to display a list of restaurants

using the basic cell style. In this chapter, we'll customize the table cell and make it look more stylish. There are a number of changes and enhancements we are going to work on:

- Rebuild the same app using `UITableViewController` instead of `UITableView`
- Display a distinct image for each restaurant rather than showing the same thumbnail
- Design a custom table view cell instead of using the basic style of table view cell

You may wonder why we need to rebuild the same app. There are always more than one way to do things. Previously, we used `UITableView` to create the table view. In this chapter, we'll use `UITableViewController` to create a table view app in Xcode. Will it be easier? Yes, it's going to be easier. Recalled that we needed to explicitly adopt both `UITableViewDataSource` and `UITableViewDelegate` protocols, `UITableViewController` has already adopted these protocols and established the connections for us. On top of this, it has all the required layout constraints right out of the box.

Starting from this chapter and onwards, you begin to develop a real-world app called *FoodPin*. It's gonna be fun!

Building a Table View App Using `UITableViewController`

First, let's see how to use `UITableViewController` to re-create the same SimpleTable app. Fire up your Xcode, and create a new project using the *Single View application* template. Name the project *FoodPin* and fill in all the required options for the Xcode project, just like what you did in the previous chapter.

Quick note: We're building a real app, so let's give it a better name. Feel free to use other names if you want. Also, make sure you use your own organization identifier.

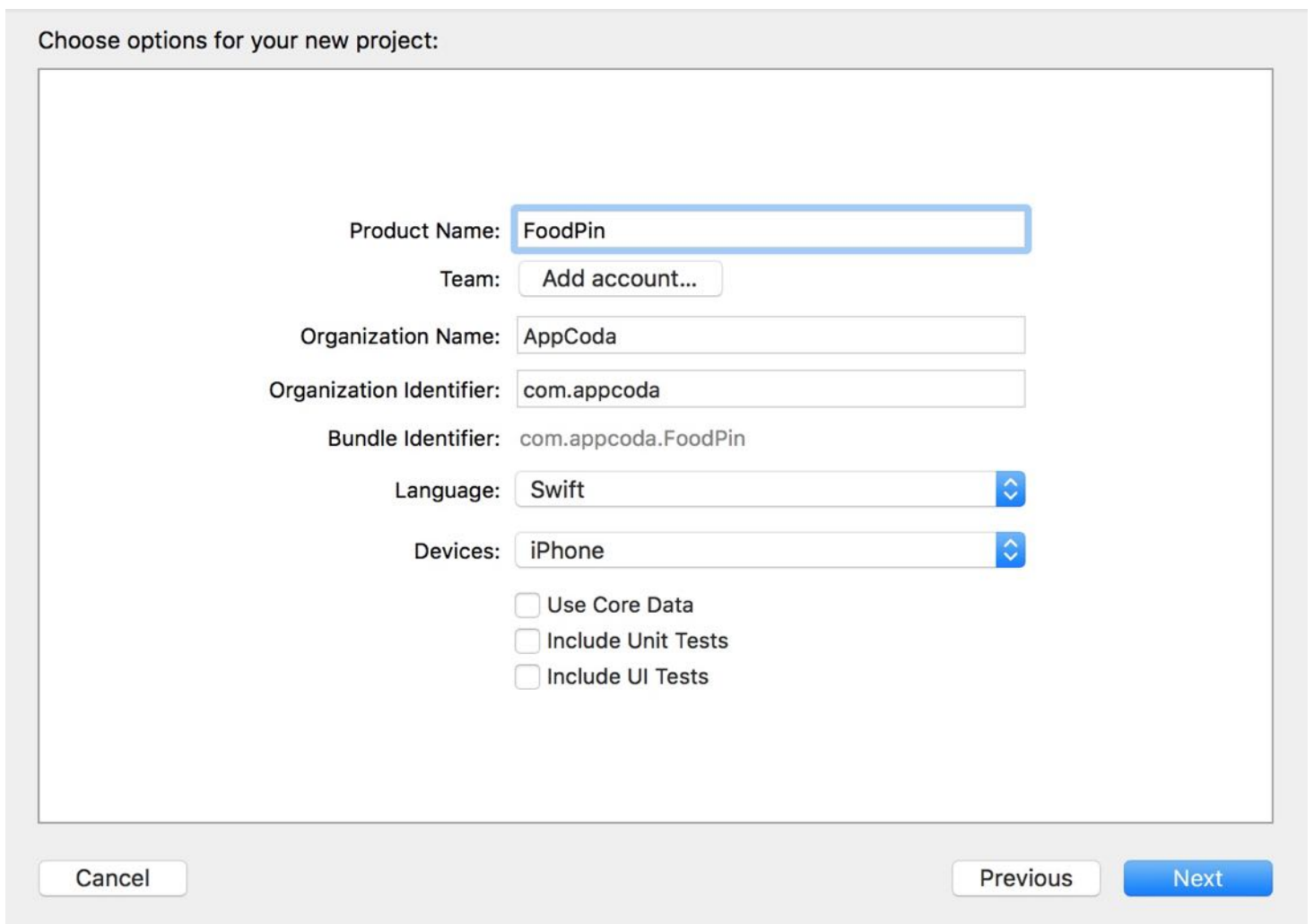


Figure 9-1. Create a FoodPin project

Once you create the Xcode project, select `Main.storyboard` and jump to the Interface Builder editor. As usual, a default view controller is generated by Xcode.

This time, we will not use the default controller. Select the view controller, and press the `delete` key to delete it. The view controller is associated with `ViewController.swift`. We do not need it either. In the Project Navigator, select the file and hit the `delete` button. Select "Move to Trash" when prompted. This will completely delete the file.

Go back to Interface Builder. Drag a Table View Controller (i.e. `UITableViewController`) from the Object library, and place it in the storyboard. You have to designate that controller as the initial view controller. This tells iOS that the table view controller is the first view controller to load. All you need to do is go to the Attributes inspector, and tick the `Is Initial View`

controller option. You'll then see an arrow pointing to the table view controller (see figure 9-2).

Set the table view controller as the initial view controller

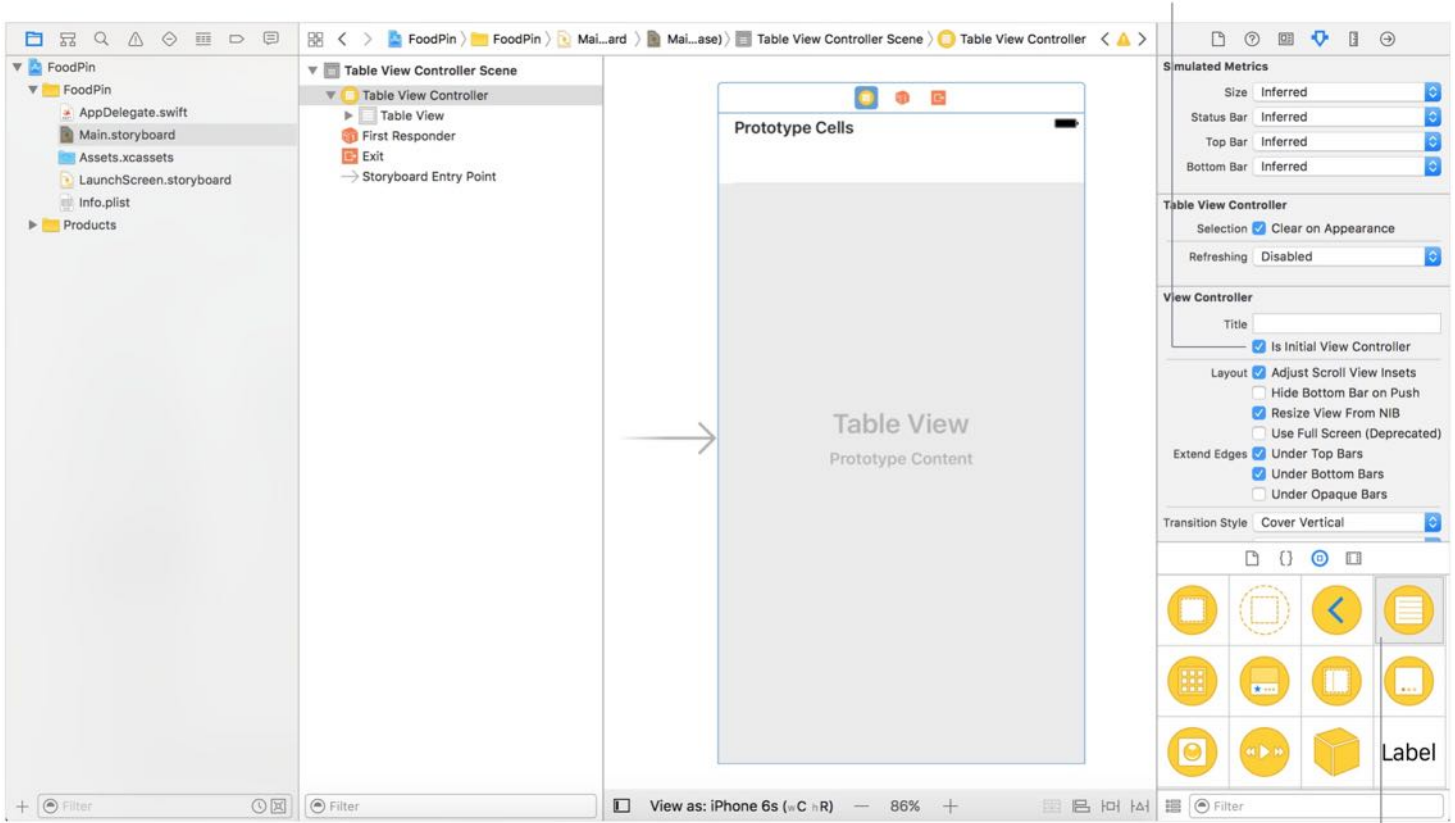


Table view controller

We haven't inserted any data into the table yet. If you compile and run the app now, you'll end up with a blank table.

By default, the table view controller is associated with the `UITableViewController` class (Go to the Identity inspector to take a look). In order to populate our own data, we have to associate it with our own class.

Go back to the Project Navigator and right-click the FoodPin folder. Select "New Files..." to create a new file.

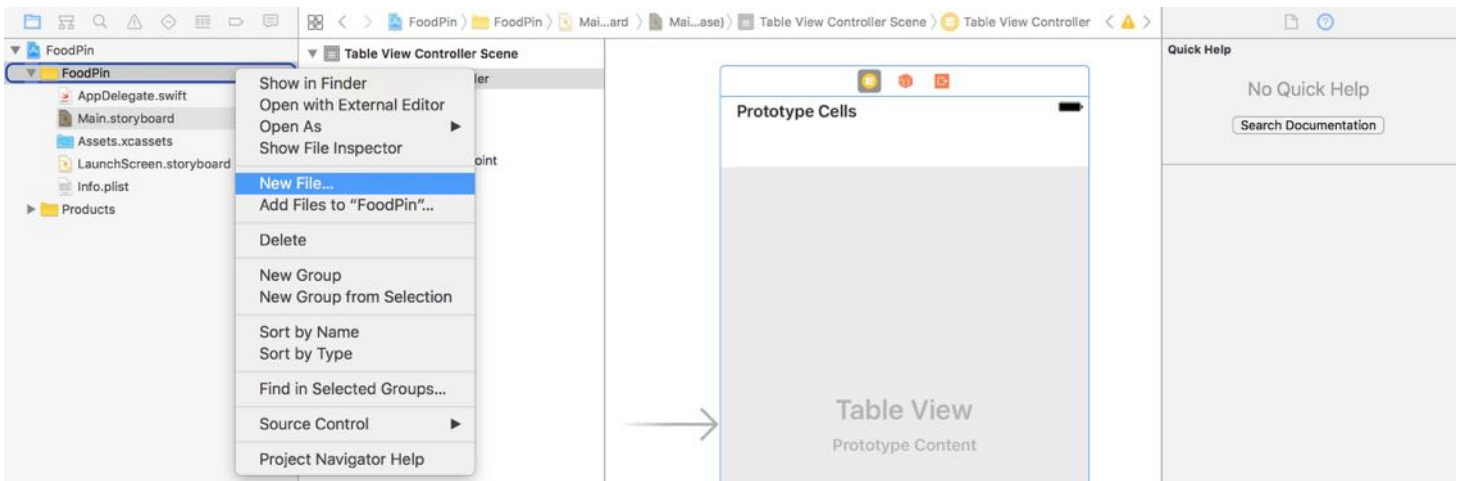
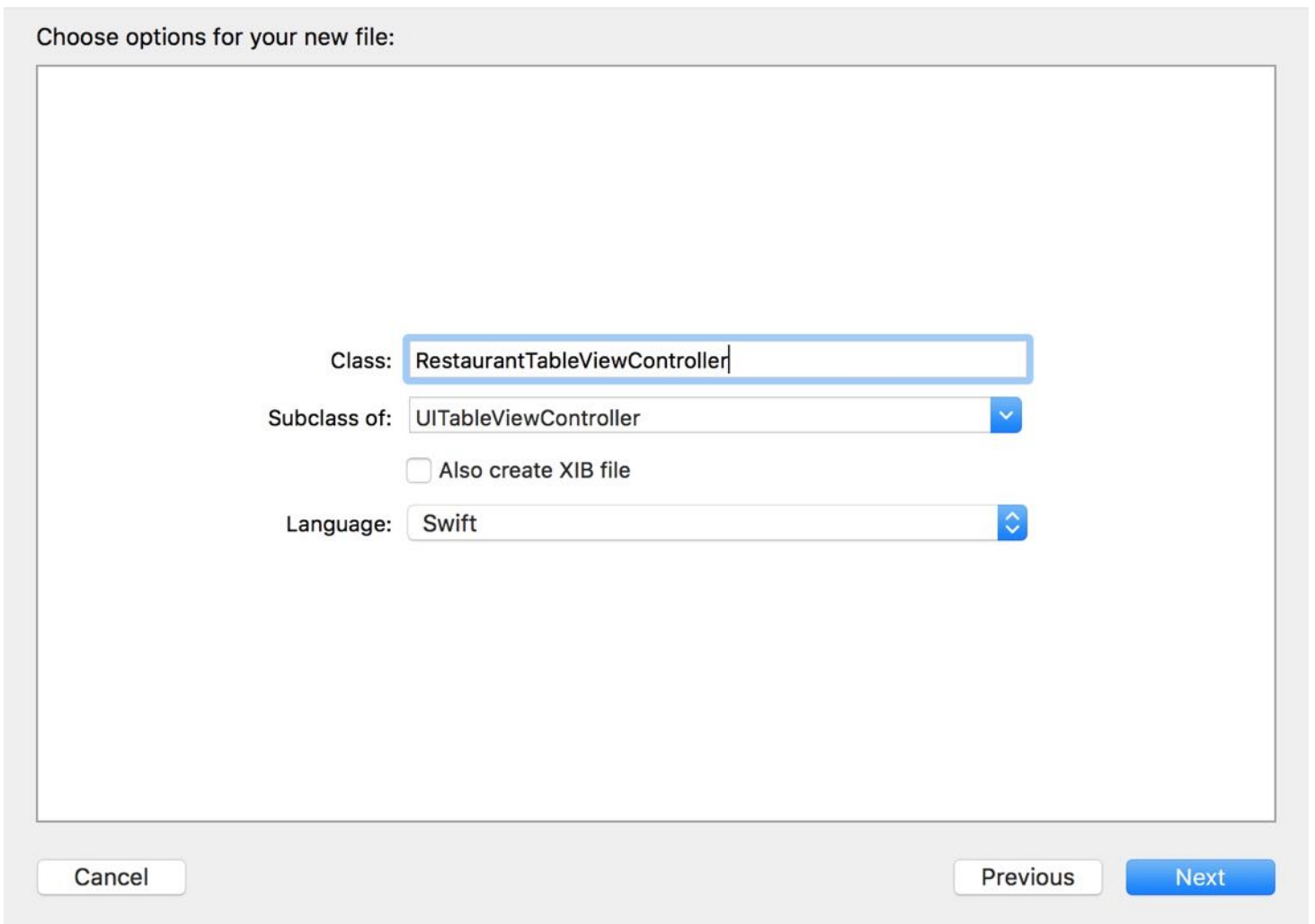


Figure 9-3. Create a new file

Choose *Source (under iOS category) > Cocoa Touch Class* as the template, and then click `Next`. Name the new class `RestaurantTableViewController`. Because we're working with a table view controller, change the value of "Subclass of" to `UITableViewController`. Keep the rest of the values intact, click "Next" and save it to the project folder. You should see the `RestaurantTableViewController.swift` file in the project navigator.



Superclass and Subclass

If you're new to programming, you may wonder what a subclass is. Swift is an object oriented programming (OOP) language. In OOP, a class can be inherited by another class. In the example, the `RestaurantTableViewController` class inherits from the `UITableViewController` class. It inherits all the states and functionalities provided by the `UITableViewController` class. The `RestaurantTableViewController` class is known as a subclass (or child class) of `UITableViewController`. In other words, the `UITableViewController` class is referred as the superclass (or parent class) of `RestaurantTableViewController`.

The table view controller in the storyboard has no idea about the

`RestaurantTableViewController` class. So we have to assign the table view controller with the new custom class. Go to `Main.storyboard` and select the table view controller. In the Identity inspector, set the custom class to `RestaurantTableViewController`. Now we have established a relationship between the table view controller in the storyboard and the new class.

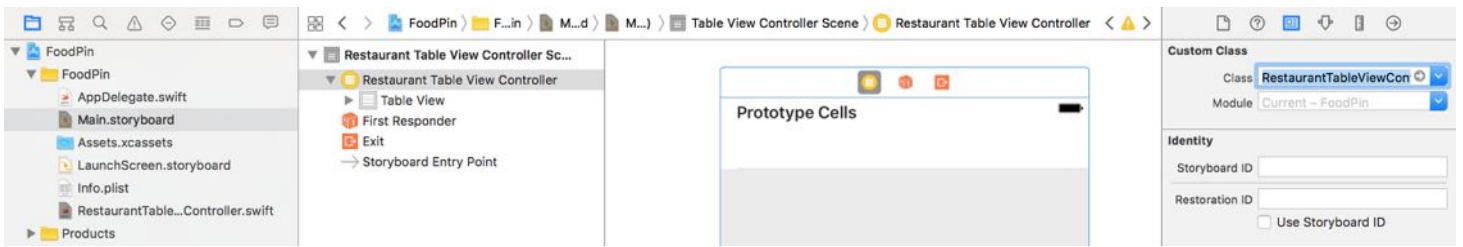


Figure 9-5. Set the custom class of the Table View Controller

There is one more thing to configure for the table view. Select the prototype cell. In the Attributes inspector, change the style to `Basic` and set the identifier to `cell`. This is pretty much the same as what we have done in the previous chapter.

Okay, the user interface is ready. Let's move onto the code. Select the `RestaurantTableViewCellController.swift` file in the Project Navigator and declare an instance variable, which is used for holding the table data.

```
var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "Petite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery", "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Avenue Meats", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional", "Barrafina", "Donostia", "Royal Oak", "CASK Pub and Kitchen"]
```

As said before, the `UITableViewController` class has already adopted the `UITableViewDataSource` and `UITableViewDelegate` protocols. Since the `RestaurantTableViewCellController` class is a subclass of `UITableViewController`, it has adopted these protocols too.

If you're not forgetful, we need to implement the following required methods of the `UITableViewDataSource` protocol in order to provide the table data:

- `tableView(_:numberOfRowsInSection:)`
- `tableView(_:cellForRowAtIndexPath:)`

The `UITableViewController` class has provided a default implementation for these two methods. However, the default methods don't usually fit well in our own apps. For instance, the default methods just show a table without any data.

In most cases, we have to override the default methods and provide our own implementation for displaying data. Let's see how we can do that. Insert the following code in

```
RestaurantTableViewController.swift :
```

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {

    let cellIdentifier = "Cell"
    let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
for: indexPath)

    // Configure the cell...
    cell.textLabel?.text = restaurantNames[indexPath.row]
    cell.imageView?.image = UIImage(named: "restaurant.jpg")

    return cell
}
```

In Swift, to override a method of a superclass, we add the `override` keyword at the very beginning of the method. When you do this, it's just like saying, "Hey, don't use the default method implementation. Use mine."

I will not go into the details of the above code as it is exactly the same as the one we covered in the previous chapter. We just display the restaurant names and images.

Next, change the following code snippets in `RestaurantTableViewController`. The code is generated by Xcode when creating the class file. By default, it returns zero for both the number of section and the number of rows in section. In other words, it's telling the table view that there is no data in the table view. This is what we want. So we have to modify the code like this:

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

override func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
    return restaurantNames.count
}
```

Here we tell the table view that there is only one section and return the total number of restaurants as stored in the array. As a side note, the `numberOfSectionsInTableView` method is optional. If you remove it, the table view still works because the number of sections is set to `1`

by default.

Lastly, download the image pack from

<http://www.appcoda.com/resources/swift3/simpletable-image1.zip> and drag all images to the

`Assets.xcassets` folder. Now, hit the "Run" button and your FoodPin app should look the same as the one we built earlier.

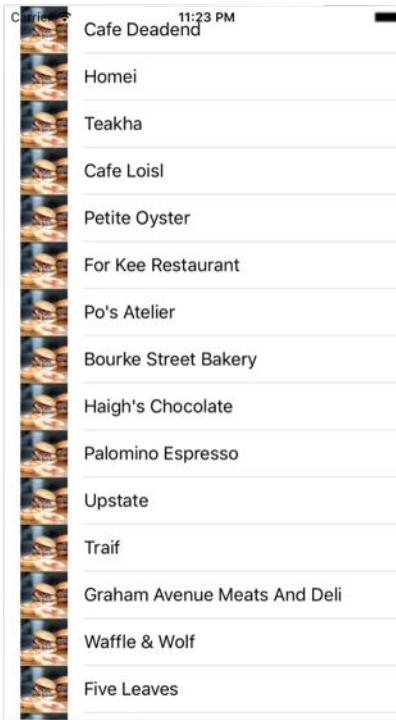


Figure 9-6. SimpleTable App

By now you should understand how to present data in a table view. I've showed you two approaches:

1. By using `UITableView` and View Controller
2. By using `UITableViewController` .

You may wonder which approach you should use. In general, approach #2 is good enough.

`UITableViewController` has configured everything for you. You can simply override some methods to provide the table data. But what you lose is flexibility. The table view, embedded in `UITableViewController` , is fixed. You can't change it. If you want to layout a more complicated

UI using table views, approach #1 will be more appropriate.

Display Different Thumbnails

Did you go through the exercise in the previous chapter? I hope you've put your effort in it. There are many ways to display different thumbnails in each table row. I'll show you the most straightforward way to do. First, download another image pack from <http://www.appcoda.com/resources/swift3/simpletable-image2.zip>, and unzip it. Add all the images into the asset catalog (i.e. Assets.xcasset). The pack includes some food and restaurant images. If you like, you're free to use your own images.

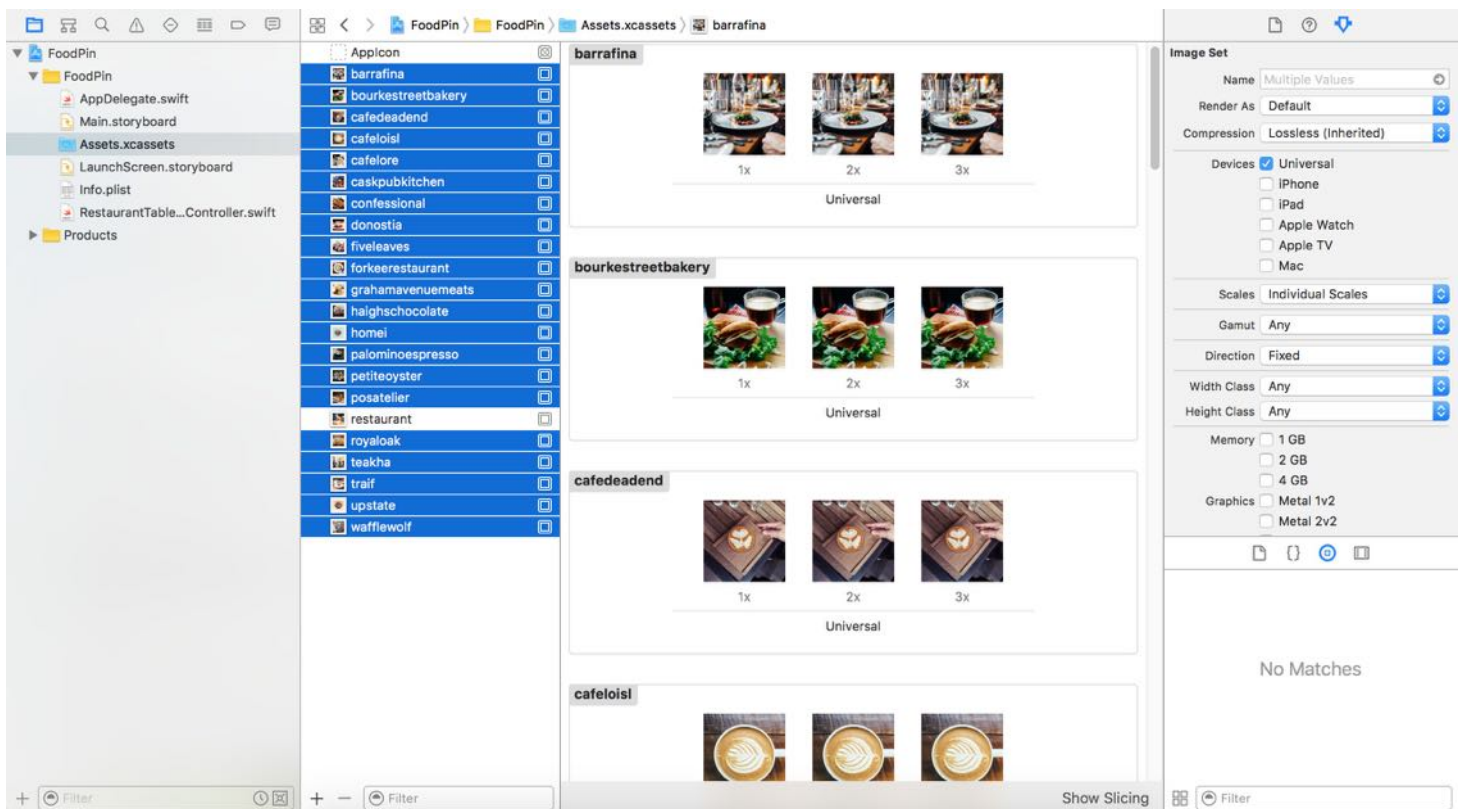


Figure 9-7. Adding restaurant images to the asset catalog

Before we move on to changing the code, let's revisit the code for displaying the thumbnails in table row.

```
cell.imageView?.image = UIImage(named: "restaurant.jpg")
```


The above line code in the `tableView(_:cellForRowAtIndexPath:)` method instructs `UITableView` to display `restaurant.jpg` in each cell. In order to display a different image, we need to alter this line of code.

As explained before, this method is called by iOS every time a particular table row is about to display. The current row number is embedded in the `indexPath` parameter. You can simply use `indexPath.row` to find out which row is now being processed.

Therefore, to display a different image in each row, all we need to do is add a new array to store the file name of thumbnails. Let's name the array `restaurantImages`. Insert the following line of code in the `RestaurantTableViewController` class:

```
var restaurantImages = ["cafedearend.jpg", "homei.jpg", "teakha.jpg",  
"cafeloisl.jpg", "petiteoyster.jpg", "forkeerrestaurant.jpg", "posatelier.jpg",  
"bourkestreetbakery.jpg", "haighschocolate.jpg", "palominoespresso.jpg",  
"upstate.jpg", "traif.jpg", "grahamavenuemeats.jpg", "wafflewolf.jpg",  
"fiveleaves.jpg", "cafelore.jpg", "confessional.jpg", "barrafina.jpg",  
"donostia.jpg", "royaloak.jpg", "caskpubkitchen.jpg"]
```

In the above code, we initialize the `restaurantImages` array with a list of image file names. The order of images are aligned with that of the `restaurantNames`.

In order to load the corresponding image of the restaurant, change the line of code in the `tableView(_:cellForRowAtIndexPath:)` method from:

```
cell.imageView?.image = UIImage(named: "restaurant.jpg")
```

to:

```
cell.imageView?.image = UIImage(named: restaurantImages[indexPath.row])
```

After saving all the changes, try to run your app again. Each restaurant has its own image.
















	Cafe Deadend
	Homei
	Teakha
	Cafe Loisl
	Petite Oyster
	For Kee Restaurant
	Po's Atelier
	Bourke Street Bakery
	Haigh's Chocolate
	Palomino Espresso
	Upstate
	Traif
	Graham Avenue Meats And Deli
	Waffle & Wolf
	Five Leaves

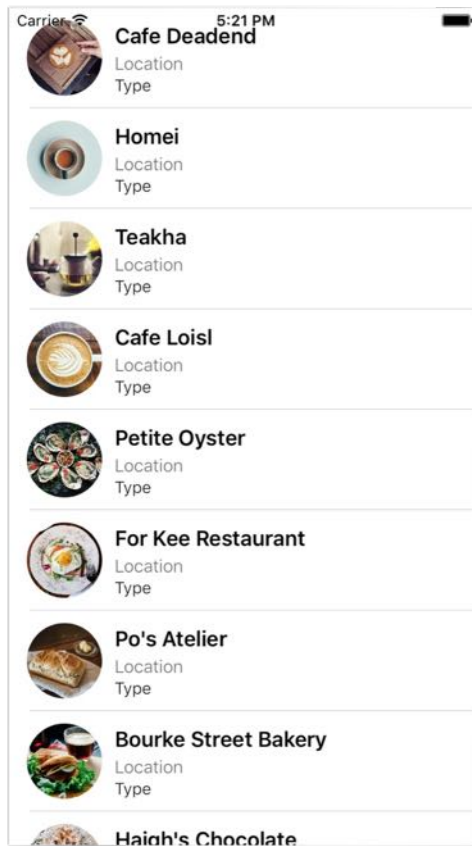
Figure 9-8. SimpleTable App with different images

Customize Table View Cells

Does the app look better? We're going to make it even better by customizing the table cells. So far we utilize the default style of the table view cell. The position and size of the thumbnail are fixed. What if you want to:

- Change the height of the cell
- Make the thumbnail a little bit bigger
- Show more information about the restaurant such as location and type
- Change the font type and size
- Display circular images instead of square images

To give you a better idea about how the cell is customized, take a look at figure 9-9. The cell looks awesome, right?



Designing Prototype Cells in Interface Builder

The beauty of prototype cells is that it allows developers to customize the cell right inside the table view controller. To build a custom cell, you simply add other UI controls (e.g. labels, image views) to the prototype cell.

Let's first change the style of the cell. You can't customize the cell when it's set to the `Basic` style. Select the prototype cell and change the style from `Basic` to `Custom` in the Attributes inspector.

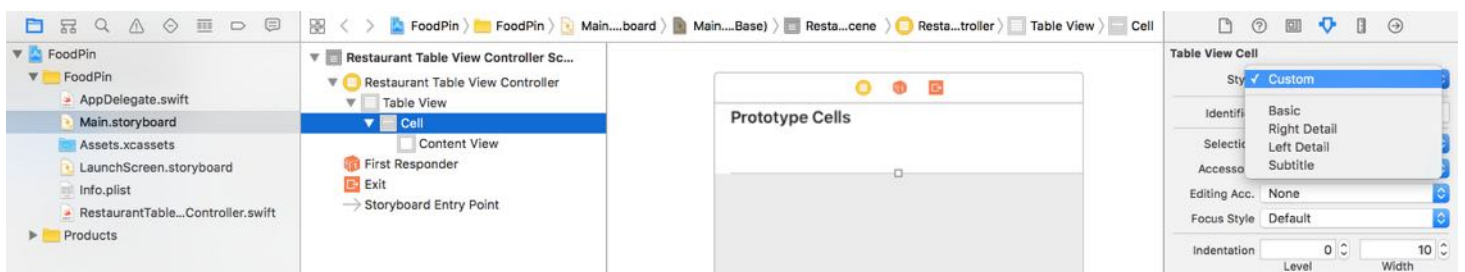
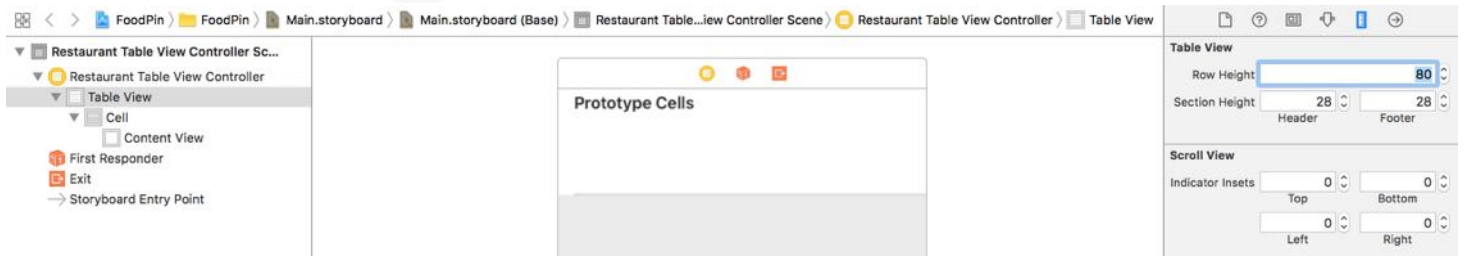
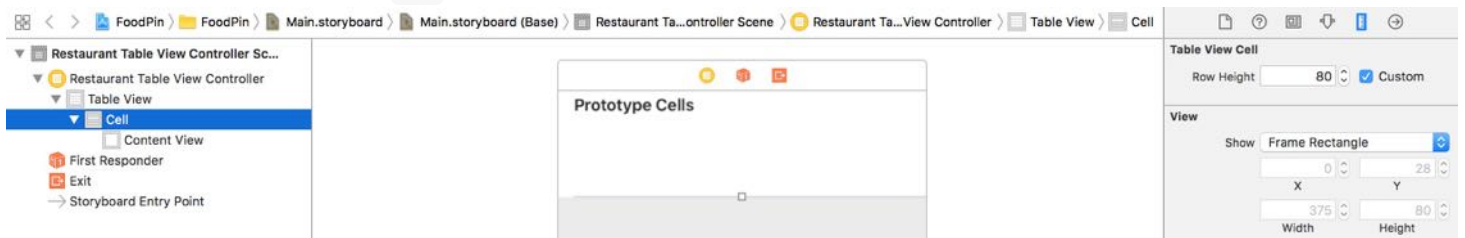


Figure 9-10. Changing the cell style from Basic to Custom

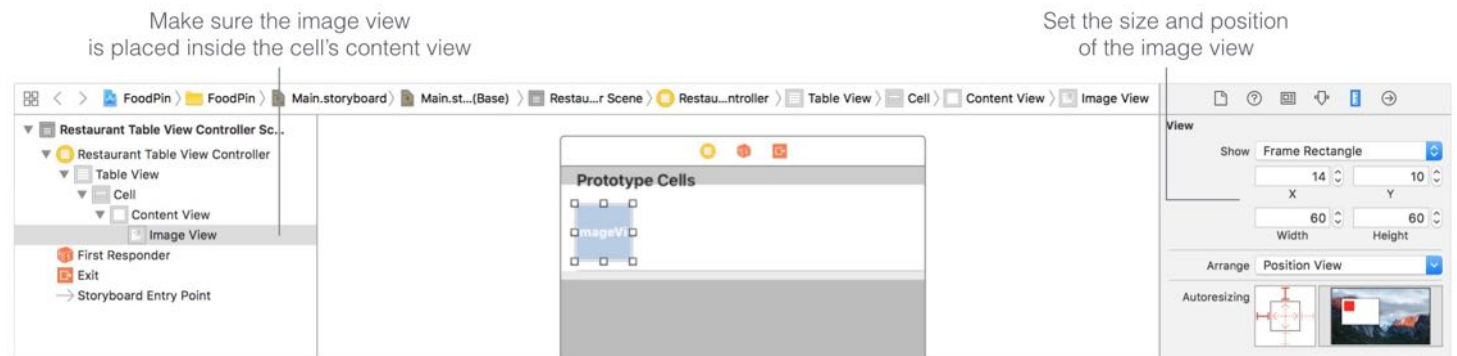
In order to accommodate a larger thumbnail, we have to make the cell a little bit bigger. You'll need to change the row height of table view and prototype cell. Select the table view and change the row height to 80 .



Then select the prototype cell and go to the Size inspector. Check the `custom` checkbox and change the row height to 80 .



After altering the row height, drag an Image View object from the Object library to the prototype cell. You're free to resize the image to fit your need. Figure 6-13 shows my recommended size. You can select the image view, click the Size inspector and change the attribute of *X*, *Y*, *Width* and *Height*.

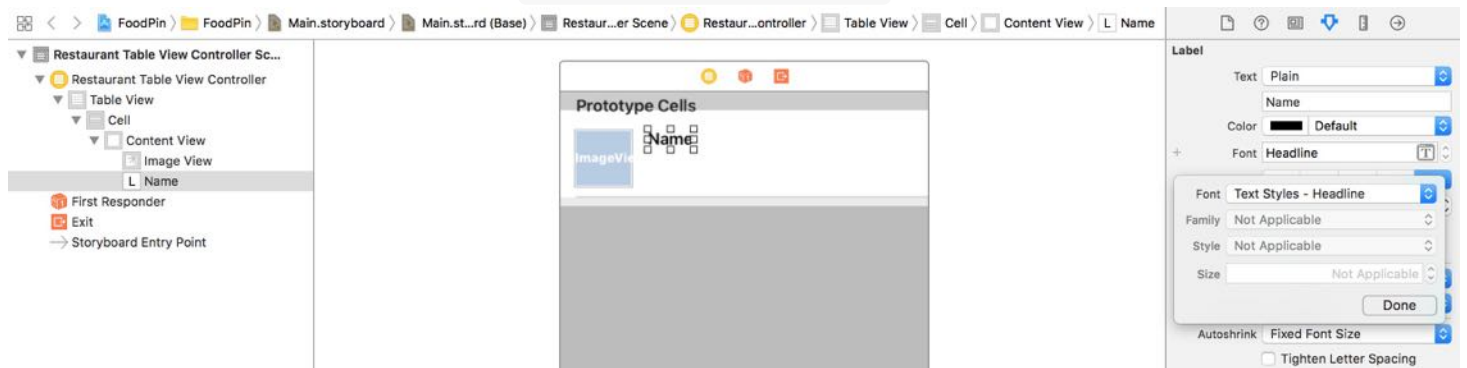


Next, we will add three labels to the prototype cell:

- Name - for restaurant name
- Location - for restaurant location (e.g. New York)
- Type - for restaurant type (e.g. tea house)

To add a label, drag a Label object from the Object library to the cell. Name the first label

Name. Instead of using the system font for the label, we'll use a text style. I will explain to you the difference of a fixed font and a text style in later chapters. For now, just go to the Attributes inspector, and change the font to `Text Styles - Headline`.



Drag another label to the cell and name it *Location*. Change the font style to `Light` and set the font size to `14` points. Also set to the font color to `Dark Gray`. Lastly, create another label and name it *Type*. Similarly, change the font style to `Light` and set the font size to `13` points.

I've covered stack views in chapter 6. Not only can you use stack views in a view controller, you can also apply stack views to layout the components in a prototype cell. Therefore, instead of defining the layout constraints for each of the labels and the image view, we will use stack views to group them together.

First, hold the `command` key, and select the three labels. Click the `stack` button in the layout bar to embed them in a vertical stack view. Go to the Attributes inspector, and change the spacing of the stack view from `0` to `1`. This will add a space between the labels.

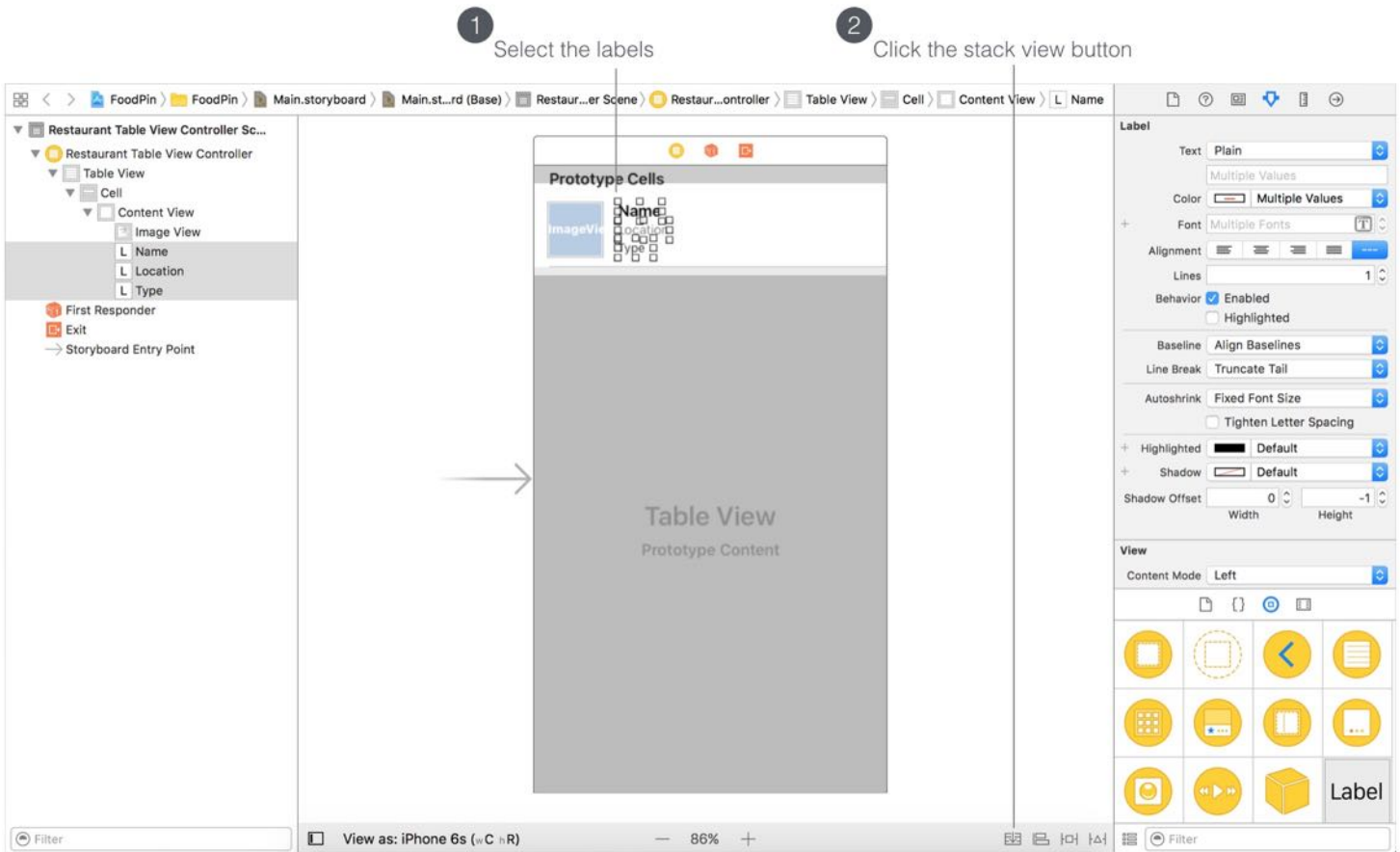


Figure 9-15. Embedding the labels using stack view

Next, select both the image view and the stack view that we have just created. Click the **Stack** button, Interface Builder will embed them into a horizontal stack view. By default, there is no space between the image view and the "Label" stack view. You can go up to the Attribute inspector, and change the spacing option from **0** to **10**. Also set the alignment to **Top**. Awesome, right? You can nest stack views to create complex layouts.

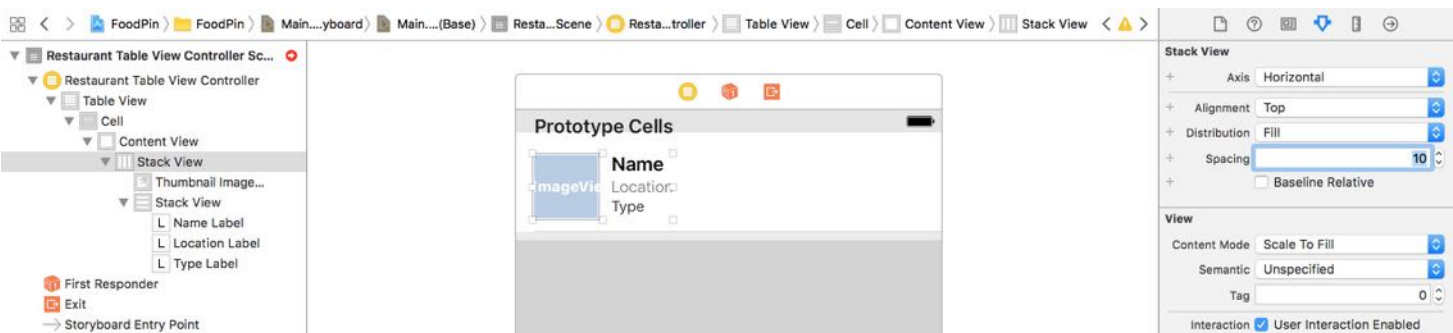


Figure 9-16. Combining the label stack view and the image view into a horizontal stack view

Again, it doesn't mean you do not need to use auto layout. We still need to define the layout constraints for the stack view. Figure 9-17 depicts the layout requirements of the cell.

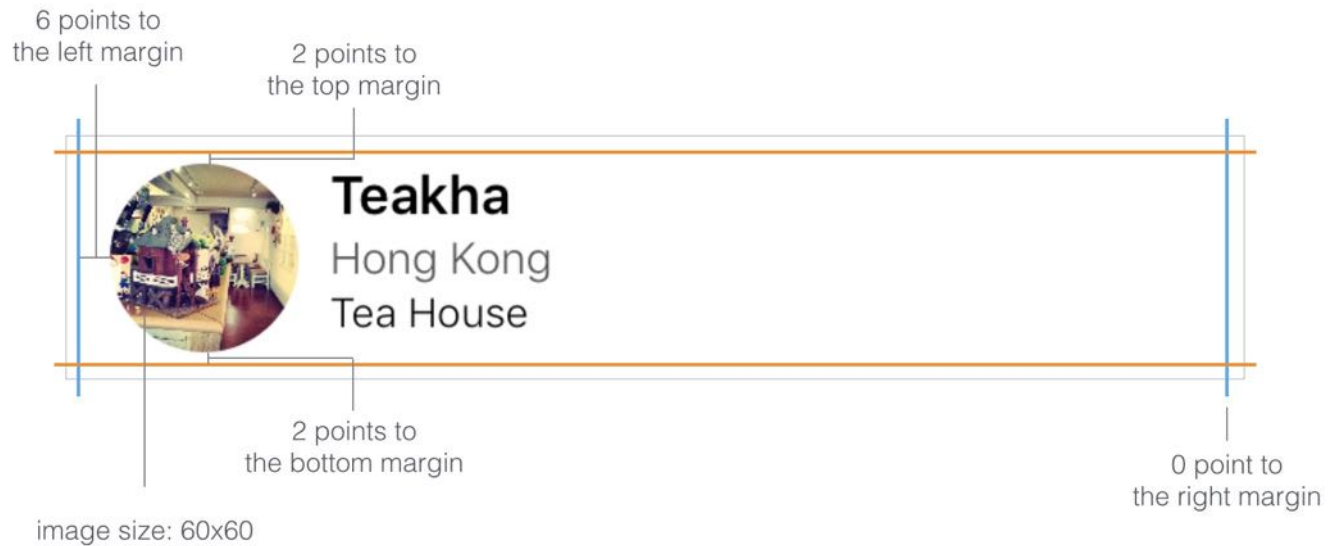


Figure 9-17. The layout requirement of the prototype cell

In brief, we want the cell content (i.e. the stack view) is confined to the viewable area of the cell. This is why we're going to define a spacing constraint for each side of the stack view. On top of this, the size of image view should be fixed to 60x60 points.

Now select the stack view, and click the Pin button in the layout bar. Set the values of the top, left, bottom and right sides to 2, 6, 1.5 and 0 respectively.

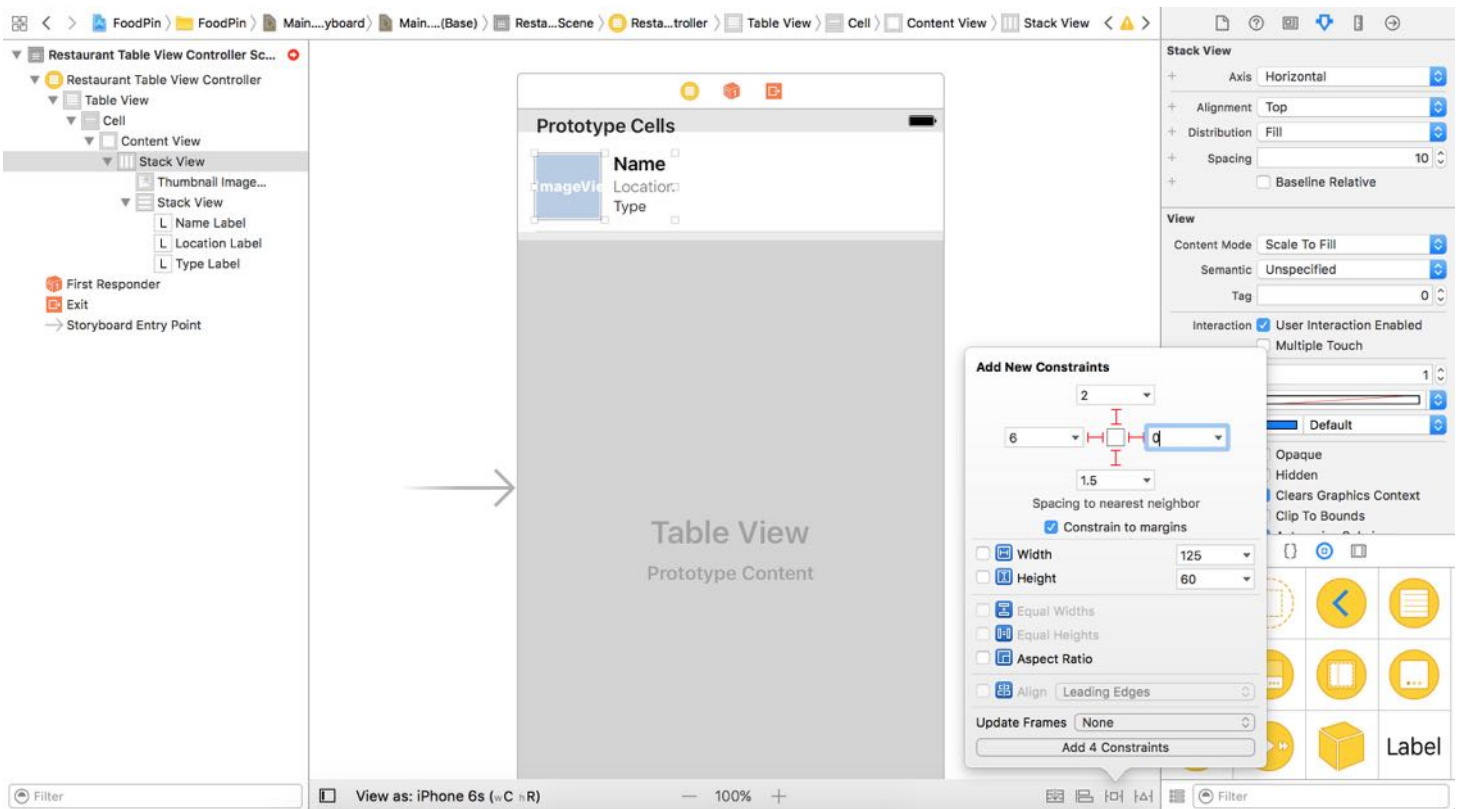


Figure 9-18. Adding spacing constraints for the stack view

Once you add the 4 constraints, the stack view should be resized automatically. Next, in the document outline, drag horizontally from the image view to itself. In the pop over menu, hold shift key and choose both "width" and "height" options. This ensures the size of the image view is fixed.

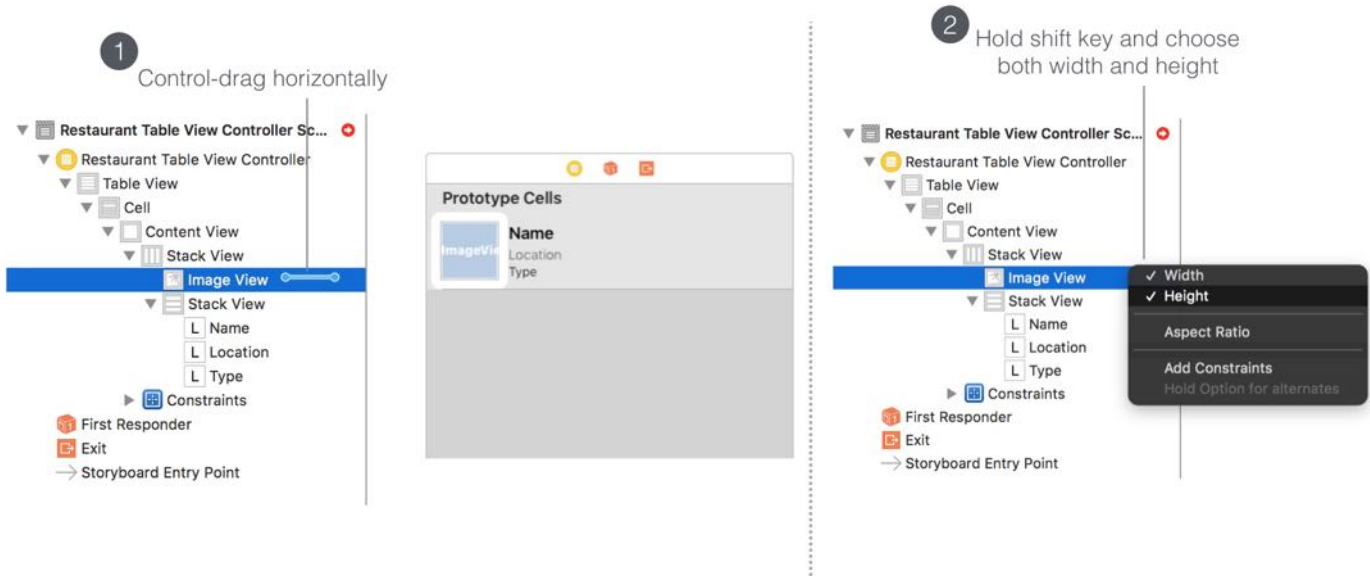


Figure 9-19. Adding the width and height constraints for the image view

Interface Builder detects certain layout constraint issues. Click the red arrow in the document outline view, and then click the red indicator to let Xcode fix the issues for you.

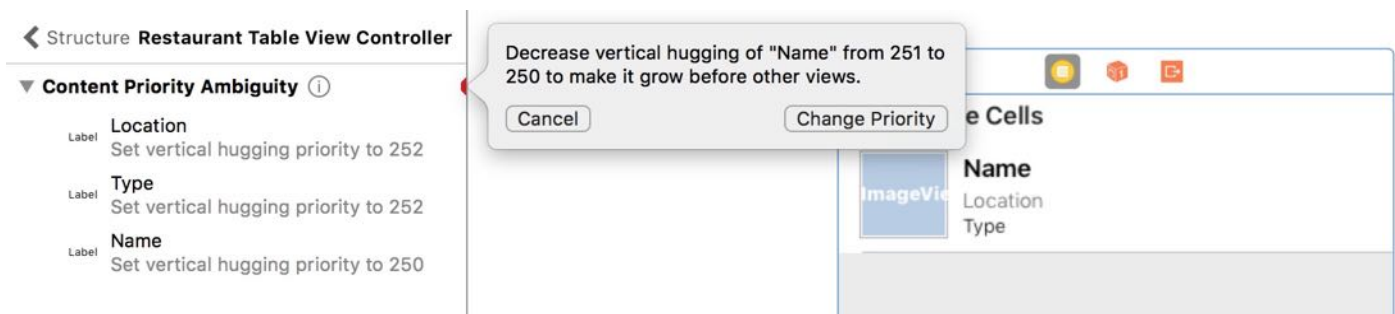


Figure 9-20. Fixing auto layout issues

Cool! You've completed the layout of the prototype cell. Let's move on and write some code.

Creating a Class for the Custom Cell

So far, we've designed the table cell. But how can we change the label values of the prototype cell? These values are supposed to be dynamic. By default, the prototype cell is associated with

the `UITableViewCell` class. In order to update the cell data, we're going to create a new class, which extends from `UITableViewCell`, for the prototype cell. This class represents the underlying data model of the custom cell.

As usual, right click the "FoodPin" folder in Project Navigator and select "New File...". After selecting the option, Xcode prompts you to select a template. As we're going to create a new class for the custom table view cell, select "Cocoa Touch Class" and click "Next". Fill in `RestaurantTableViewCell` as the class name and set the value of "Subclass of" to `UITableViewCell`.

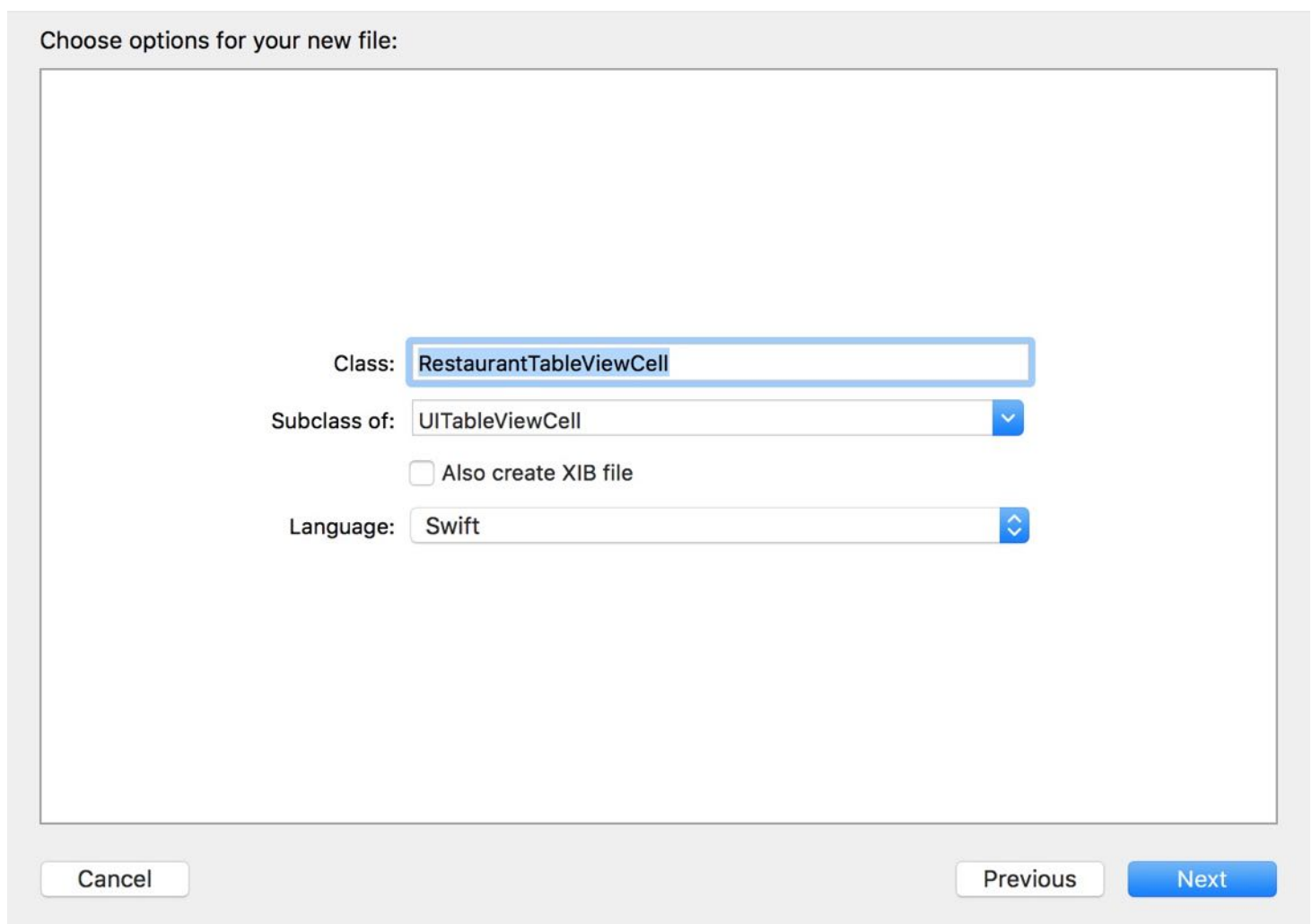


Figure 9-21. Creating a new class for the custom table view cell

Click "Next" and save the file in the FoodPin project folder. Xcode should create a file named `RestaurantTableViewCell.swift` in the Project Navigator.

Next, declare the following outlet variables in the `RestaurantTableViewCell` class:

```
@IBOutlet var nameLabel: UILabel!  
@IBOutlet var locationLabel: UILabel!  
@IBOutlet var typeLabel: UILabel!  
@IBOutlet var thumbnailImageView: UIImageView!
```

The `RestaurantTableViewCell` class serves as the data model of the custom cell. In the cell, we have 4 properties that are changeable:

- thumbnail image view
- name label
- location label
- type label

The data model stores and provides the values for the cell to display. Each of them is required to connect with the corresponding user interface object in Interface Builder. By connecting the source code with the UI objects, we can change the values of UI objects dynamically.

This is a very important concept in iOS programming. Your UI in storyboard and code are separated. You create the UI in Interface Builder, and you write your code in Swift. If you want to change the value or properties of a UI element (e.g. label), you have to establish a connection between them so that an object in your code can obtain a reference to an object defined in a storyboard. In Swift, you use `@IBOutlet` keyword to indicate a property of a class, that can be exposed to Interface Builder. For properties annotated with the `IBOutlet` keywords, we call it *outlets*.

So in the above code, we declare four outlets. Each outlet is going to connect with its corresponding UI object. Figure 9-22 depicts the connections.

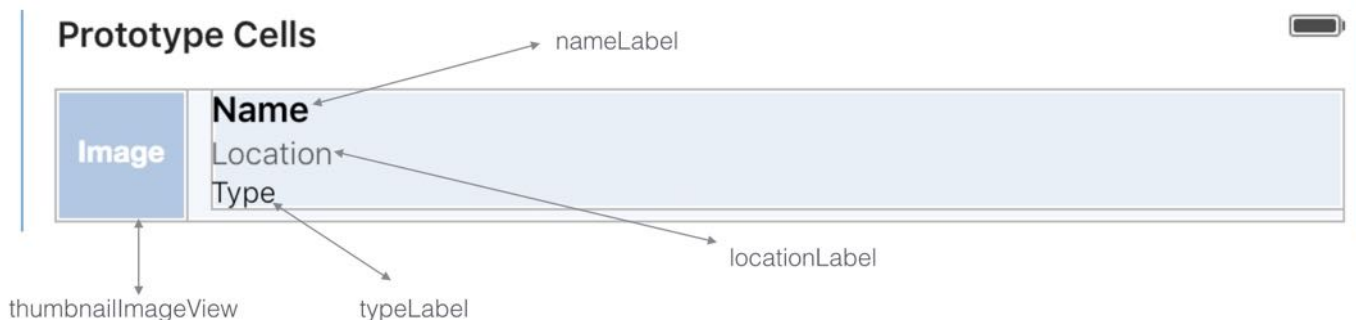


Figure 9-22. Outlet connections

@IBAction vs @IBOutlet

We've used @IBAction to indicate action methods when developing the HelloWorld app. What's the difference between @IBAction and @IBOutlet? @IBOutlet is used to indicate a property that can be connected with a view object in a storyboard. For example, if the outlet is connected with a button, you can use the outlet to change the color or title of the button. On the other hand, @IBAction is used to indicate an action method that can be triggered by a certain event. For example, when a user taps a button, it can trigger an action method to do something.

Before we can establish a connection between the outlets of `CustomTableViewCell` class and the prototype cell in Interface Builder. We have to first set the custom class.

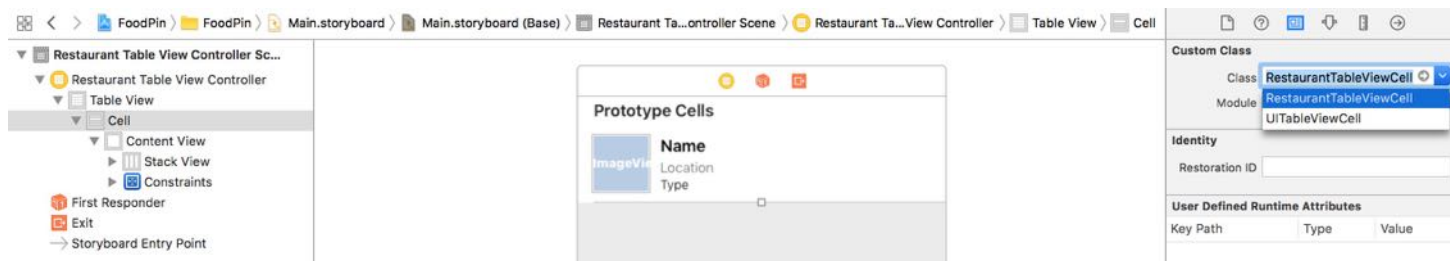


Figure 9-23. Set the custom class of the prototype cell

By default, the prototype cell is associated with the default `UITableViewCell` class. To assign the prototype cell with the custom class, select the cell in storyboard. In the Identity inspector, set the custom class to `CustomTableViewCell` (see figure 9-23).

Establishing the Connection

Next, we'll establish the connections between the outlets and UI objects in the prototype cell. In Interface Builder, right click the cell in the Document Outline view to bring up the Outlets inspector. Drag from the circle (next to thumbnailImageView) to the image view object in the prototype cell (see figure 9-24). Xcode automatically establishes the connection when you release the button.

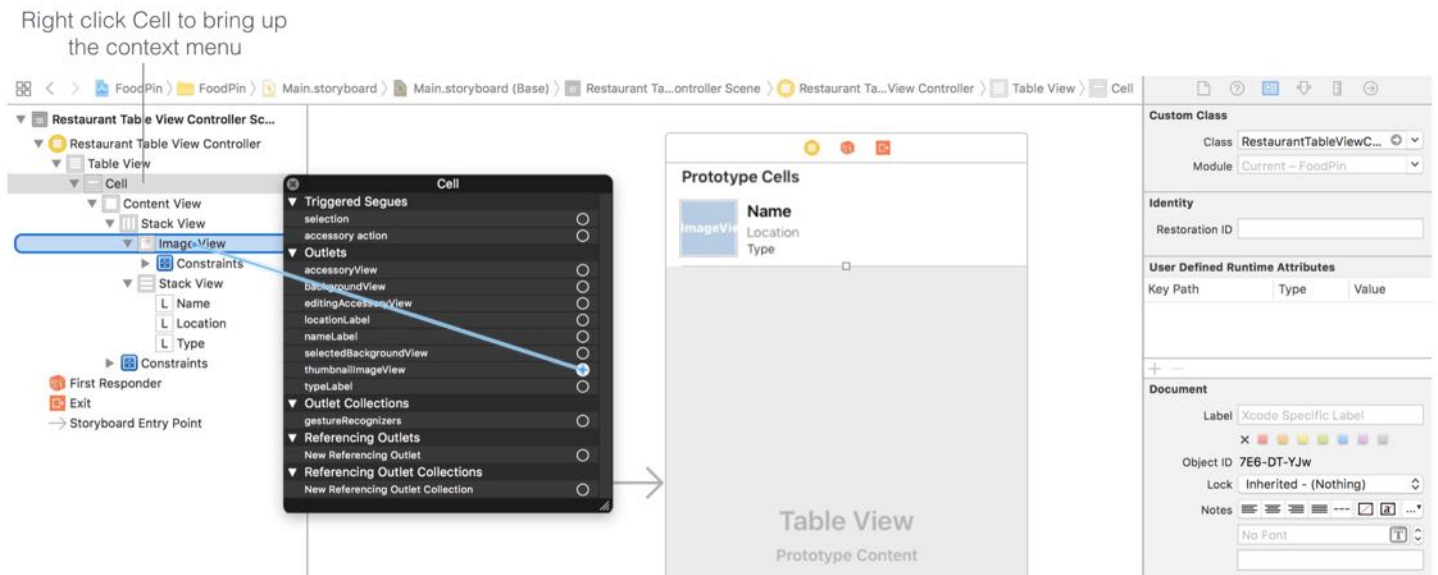


Figure 9-24. Connecting the outlet with the image view

Repeat the above procedures for the following outlets:

- locationLabel - connects to the Location label the cell
- nameLabel - connects to the Name label of the cell
- typeLabel - connects to the Type label of the cell

After you've made all the connections, the UI should look like the screenshot shown in figure 9-25.

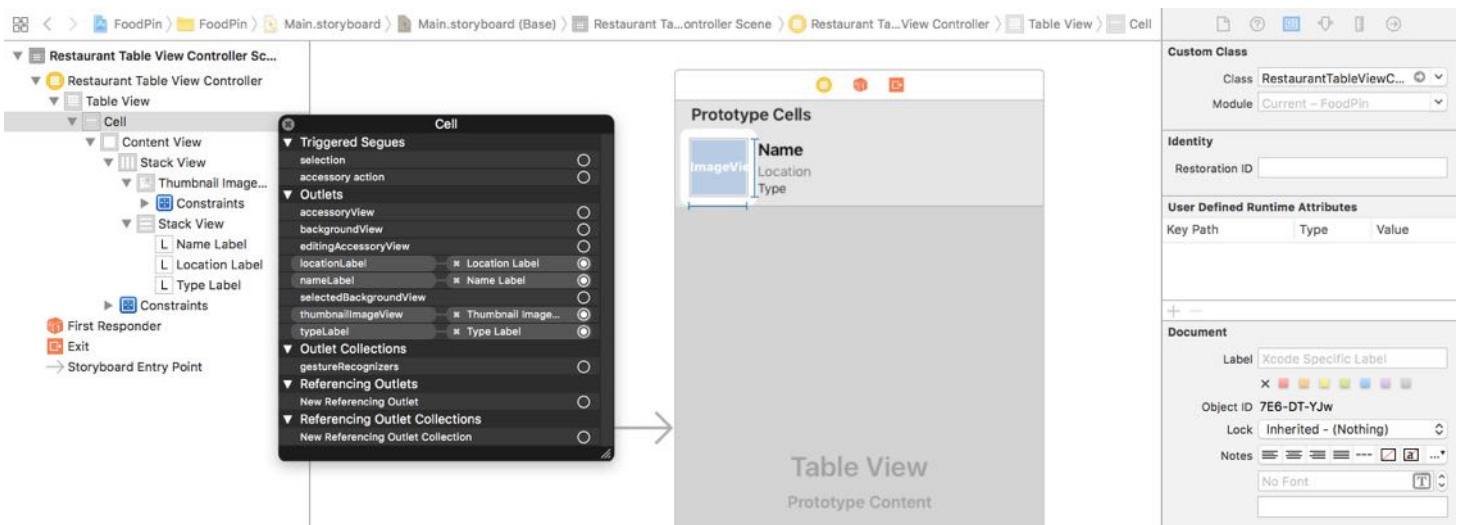


Figure 9-25. The outlet connections

Coding the Table View Controller

Finally, we come to the last part of the change. In the `RestaurantTableViewController` class, we're still using `UITableViewCell` (i.e. the default cell) to display the content. We need to modify a line of code to replace it with our custom cell. If you look into the existing implementation of the `tableView(_:cellForRowAtIndexPath:)` method. The second line of code is:

```
let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for: indexPath)
```

I've explained the meaning of the `dequeueReusableCell` method in the previous chapter. It is flexible enough to return any cell types from the queue. By default, it returns a generic cell of a `UITableViewCell` type. In order to use the `RestaurantTableViewCell` class, it's our responsibility to "convert" the returned object of `dequeueReusableCell` to `RestaurantTableViewCell`. This conversion process is known as downcasting. In Swift, we use the `as!` keyword to perform a forced conversion. Therefore, change the above line of code to the following:

```
let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for: indexPath) as! RestaurantTableViewCell
```


as! and as?

Downcasting allows you to convert a value of a class to its derived class. For example, `RestaurantTableViewCell` is a child class of `UITableViewCell`. The `dequeueReusableCellWithIdentifier` method always returns a `UITableViewCell` object. If a custom cell is used, this object can be converted to the specific cell type (e.g. `RestaurantTableViewCell`). Prior to Swift 1.2, you can just use the `as` operator for downcasting. However, sometimes the object may not be converted to a specified type. Therefore, from Swift 1.2 and onwards, Apple introduced two more operators: `as!` and `as?`. If you're quite sure that the downcasting can perform correctly, use `as!` to perform the conversion. In case you're not sure if the value of one type can be converted to another, use `as?` to perform an optional downcasting. You're required to perform additional checking to see if the downcasting is successful or not.

I know you can't wait to test the app, but we need to change a few more lines of code. The lines of code below set the values of restaurant name and image:

```
// Configure the cell...
cell.textLabel?.text = restaurantNames[indexPath.row]
cell.imageView?.image = UIImage(named: restaurantImages[indexPath.row])
```

Both `textLabel` and `imageView` are properties of the default `UITableViewCell` class. Because we're now using our own `RestaurantTableViewCell`, we need to use the properties of the custom class. Change the above lines to the following:

```
// Configure the cell...
cell.nameLabel.text = restaurantNames[indexPath.row]
cell.thumbnailImageView.image = UIImage(named: restaurantImages[indexPath.row])
```

Now you're ready to go. Hit the Run button and test the app. Your app should look like the one shown in figure 9-26. Try to rotate the simulator. The app also works in landscape orientation.

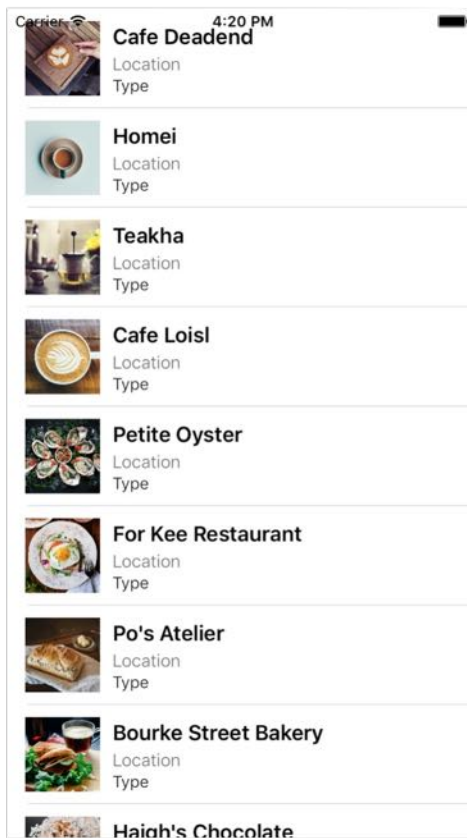


Figure 9-26. FoodPin app with custom table view cells

It is a huge improvement from the previous version of the app. We're going to make it even better by changing the thumbnail into circular image.

Circular Image

Since the release of iOS 7, iOS favors circular image over square image. You can find circular icons or images in stock apps such as Contacts and Phone. Wouldn't it be great to make all the restaurant images circular? You do not need Photoshop to tweak the images. What you need is just two lines of code.

Every view in the UIKit (e.g. `UIView`, `UIImageView`) is backed by an instance of the `CALayer` class (i.e. layer object). The layer object is designed to manage the backing store for the view and handles view-related animations.

The layer object provides various attributes that can be set to control the visual content of the

view such as:

- Background color
- Border and border width
- Shadow color, width, etc
- Opacity
- Corner radius

The corner radius is the attribute, which we use to draw rounded corners. Xcode provides two ways to edit the layer properties. You can directly update its properties through code. Here is the lines of code the following lines of code for changing the corner radius of the image view:

```
cell.thumbnailImageView.layer.cornerRadius = 30.0  
cell.thumbnailImageView.clipsToBounds = true
```

An even easier way is to make the change through Interface Builder. First, select the image view in the stack view. Go to the Identity inspector, click the Add button (+) in the lower left of the user defined runtime attributes editor. A new runtime attribute appears in the editor. Double click on the Key Path field of the new attribute to edit the key path for the attribute. Set the value to `layer.cornerRadius` and hit Return to confirm. Click on the Type attribute and choose `Number`. Lastly, set the value to `30`. To make a circular image from a square image, the radius is set to half the width of the image view. Here, the width of square image is 60 points. Thus, the corner radius is set to `30` points.

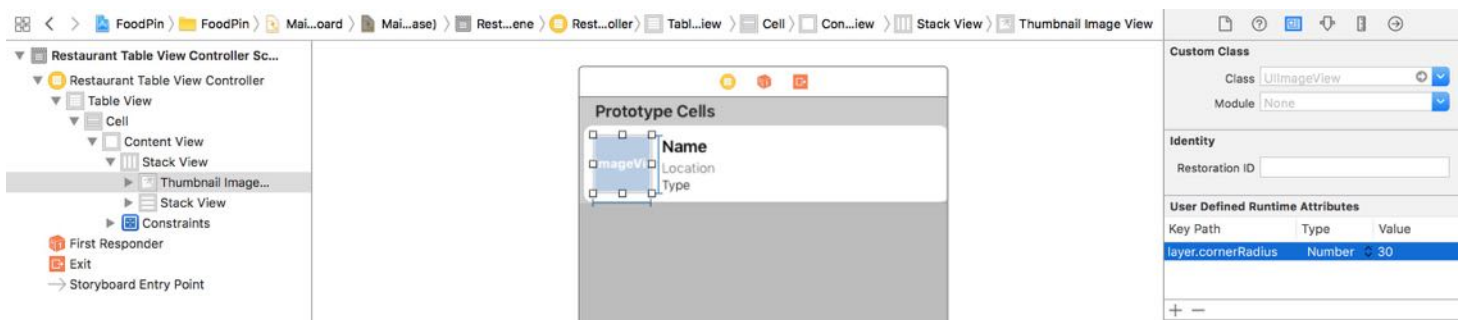


Figure 9-27. Set the runtime attribute to make the corner round

When the image view is initialized, this runtime attribute will be automatically loaded to make the corner round. There is one more thing you have to configure before the circular image

works properly. Select the image view and go to the Attributes inspector. In the Drawing section, enable the *Clip to Bounds* option. This causes the content to be clipped to the rounded corners.

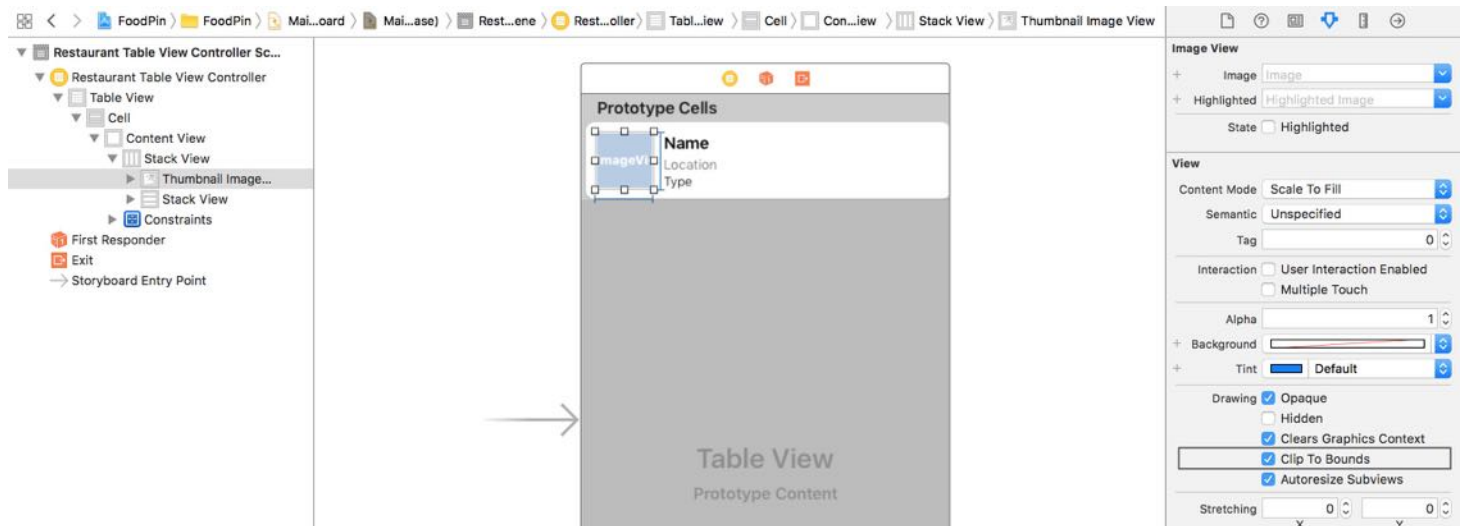


Figure 9-28. FoodPin app with circular images

Now compile and run the app. The UI looks even better, right? Without writing a line of code, we change the square images to circular one.

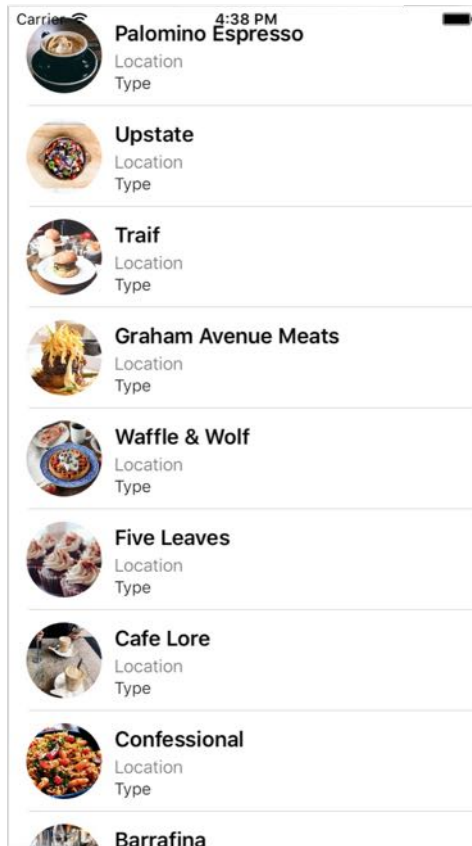


Figure 9-29. FoodPin app with circular images

You're free to alter the value of corner radius. Try to change the corner radius to 10 points and see what you get.

Exercise #1

As of now, the app simply displays "Location" and "Type" for all rows. As an exercise, I'll leave it to you to fix the issue. Try to edit the source code to update the location and type labels.

Below are the two arrays you need:

```
var restaurantLocations = ["Hong Kong", "Hong Kong", "Hong Kong", "Hong Kong",
"Hong Kong", "Hong Kong", "Hong Kong", "Sydney", "Sydney", "Sydney", "New
York", "New York", "New York", "New York", "New York", "New York", "New York",
"London", "London", "London", "London"]

var restaurantTypes = ["Coffee & Tea Shop", "Cafe", "Tea House", "Austrian /
Causal Drink", "French", "Bakery", "Bakery", "Chocolate", "Cafe", "American /
Seafood", "American", "American", "Breakfast & Brunch", "Coffee & Tea", "Coffee
```

& Tea", "Latin American", "Spanish", "Spanish", "Spanish", "British", "Thai"]

Exercise #2

The previous exercise may be too easy for you. Here is another challenge. Try to redesign the prototype cell and see if you create an app like this (see figure 9-30).

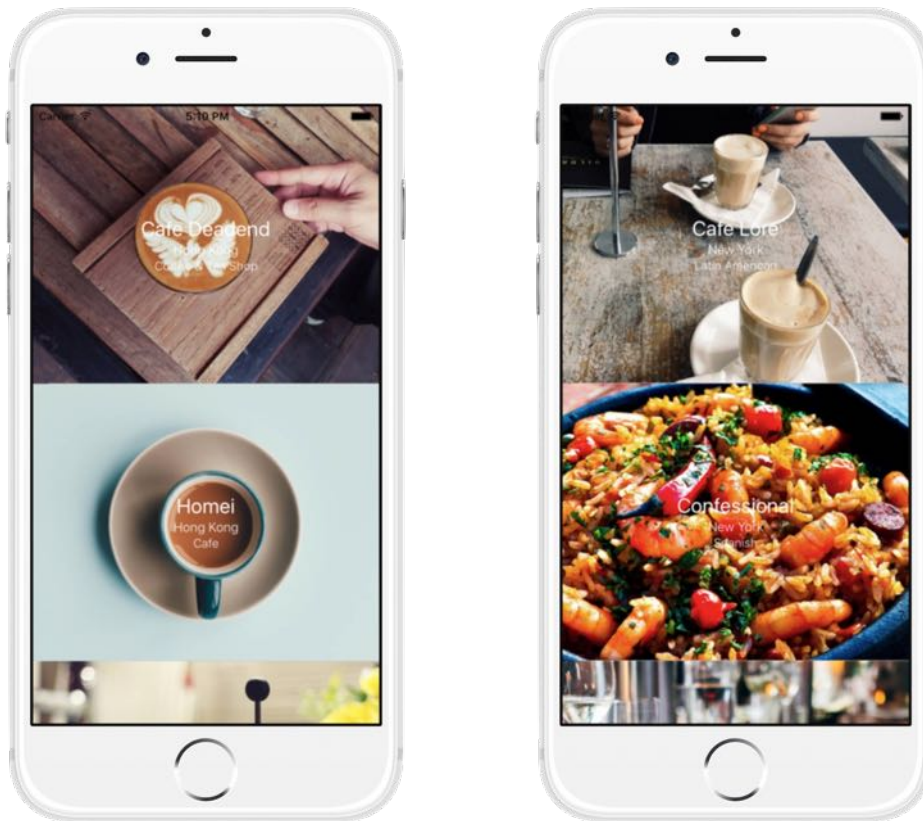


Figure 9-30. Redesigned Custom Table

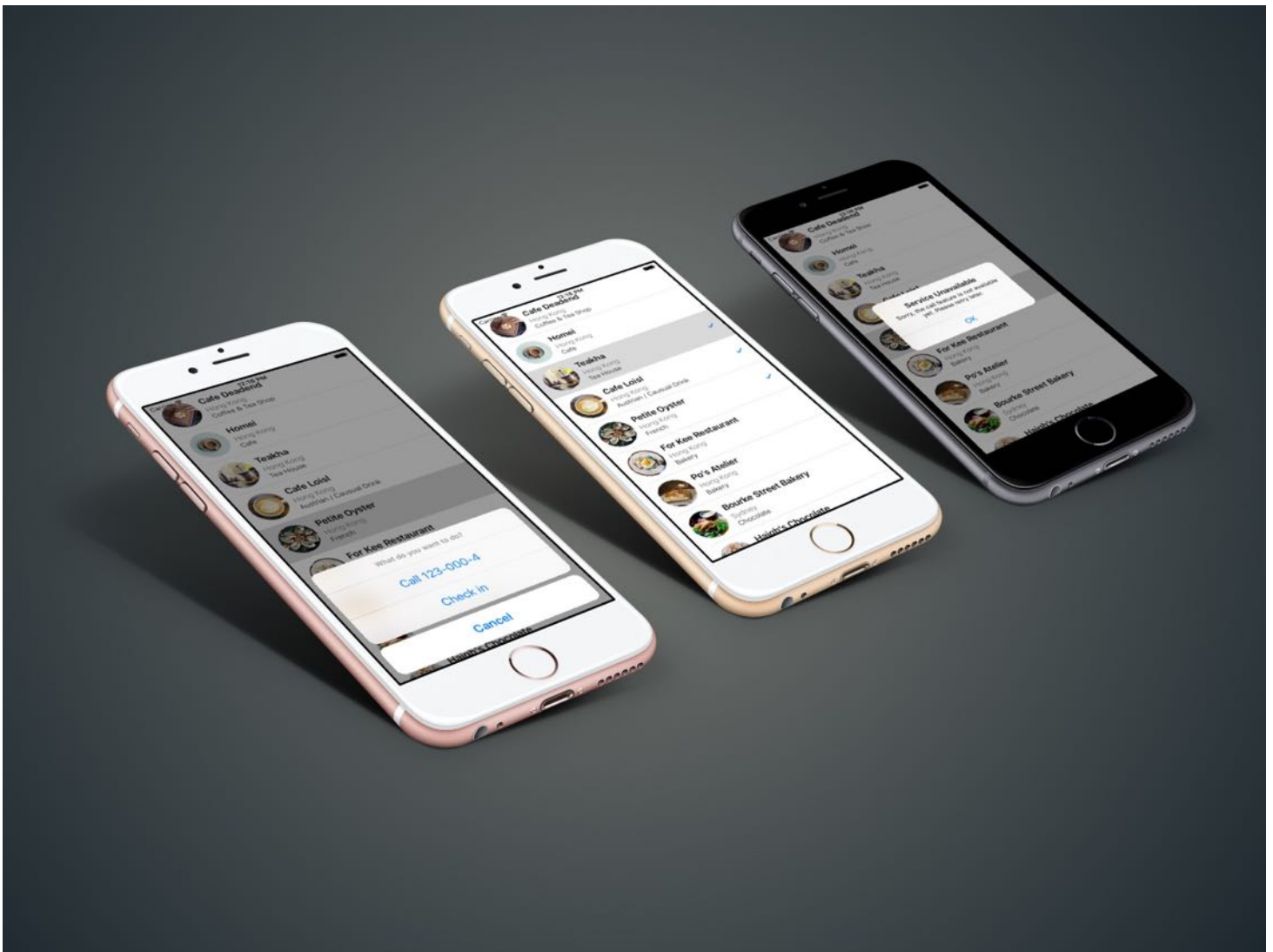
Summary

Congratulations! You've made a huge progress. If you understand the ins and outs of cell customization, you're ready to make some awesome UIs. Table view is the backbone for most iOS apps. Unless you're building a game, you'll probably implement a table view in one way or another when building your own apps. Table view customization may be a bit complex for some of you. So take some time to work on the exercises and play around with the code. Remember "learn by doing" is the best way to learn coding.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinCustomTable.zip>. If you can't figure out how to complete the exercise, you can download the solution from <http://www.appcoda.com/resources/swift3/FoodPinCustomTableExercise.zip>.

Chapter 10

Interacting with Table Views and Using UIAlertController



There is no learning without trying lots of ideas and failing lots of times.

- Jonathan Ive

Up till now, we only focus on displaying data in a table view. I guess you are thinking how we can interact with the table view and detect row selections. This is what we're going to discuss in

this chapter.

We'll continue to polish the FoodPin app, which we have built in the previous chapter (<http://www.appcoda.com/resources/swift3/FoodPinCustomTable.zip>), and add a couple of enhancements:

- Bring up a menu when a user taps a cell. The menu offers two options: *Call* and *Check-in*.
- Display a heart icon when a user selects *Check-in*.

Through implementing these new features, you will also learn how to use `UIAlertController`, which is commonly used to display alerts in iOS apps.

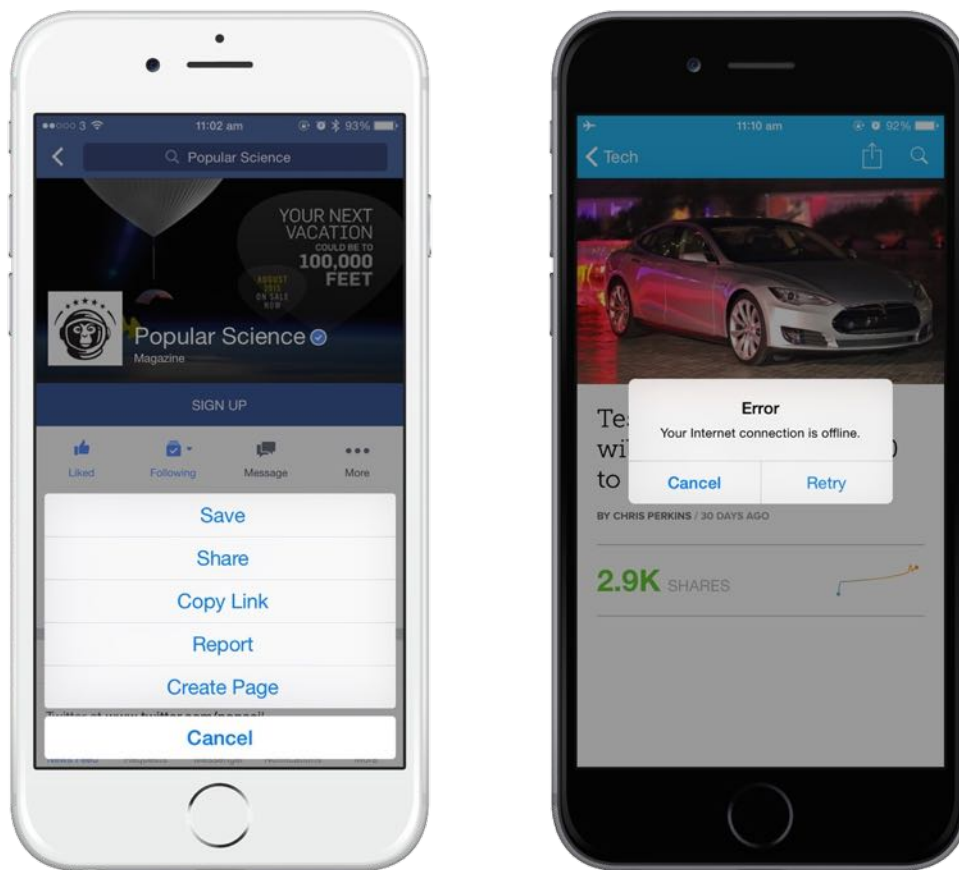


Figure 10-1. Sample alerts in Facebook and Mashable apps

Quick note: This class replaces the `UIActionSheet` and `UIAlertView` classes for displaying alerts in iOS 8 (or up).

Understanding the UITableViewDelegate Protocol

When we first built the SimpleTable app in Chapter 8, we adopted two delegates, `UITableViewDelegate` and `UITableViewDataSource`, in the `RestaurantTableViewController` class. I have discussed with you the `UITableViewDataSource` protocol but barely mentioned about the `UITableViewDelegate` protocol.

As said before, the delegate pattern is very common in iOS programming. Each delegate is responsible for a specific role or task to keep the system simple and clean. Whenever an object needs to perform a certain task, it depends on another object to handle it. This is usually known as "separation of concerns" in software design.

The `UITableView` class applies this design concept. The two protocols are designed for different purposes. The `UITableViewDataSource` protocol defines methods, which are used for managing table data. It relies on the delegate to provide the table data. On the other hand, the `UITableViewDelegate` protocol is responsible for setting the section headings and footers of the table view, as well as, handling cell selections and cell reordering.

To manage the row selection, we will implement some of the methods in the `UITableViewDelegate` protocol.

Reading the Documentation

Before implementing the methods, you may wonder:

How do we know which methods in UITableViewDelegate to implement?

The answer is "Read the documentation". You're granted free access to the Apple's official iOS developer reference (<https://developer.apple.com/library/ios/>). As an iOS developer, you need to get used to reading the API documentation. There is no single book on earth to cover everything about the iOS SDK. Most of the time when we want to learn more about a class or a protocol, we have to look up to the API document. Apple provides a simple way to access the documentation in Xcode. All you need to do is place the cursor over a class or a protocol (e.g. `UITableViewController`) and press `control-command-?`. This brings up a popover showing the details of the class like the protocols it has adopted.

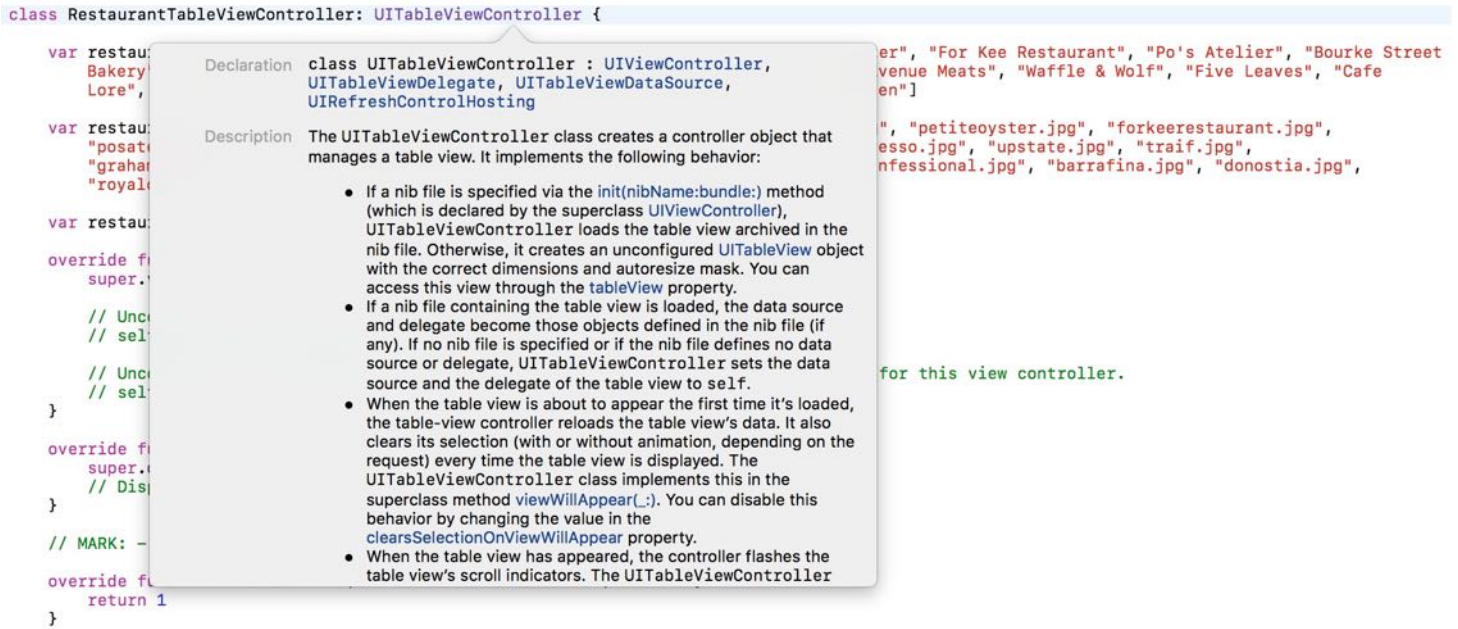


Figure 10-2. Accessing the API documentation by using a shortcut key

Clicking `UITableViewDelegate` would further bring up a documentation browser. From there, you'll find all the methods defined in the protocol.

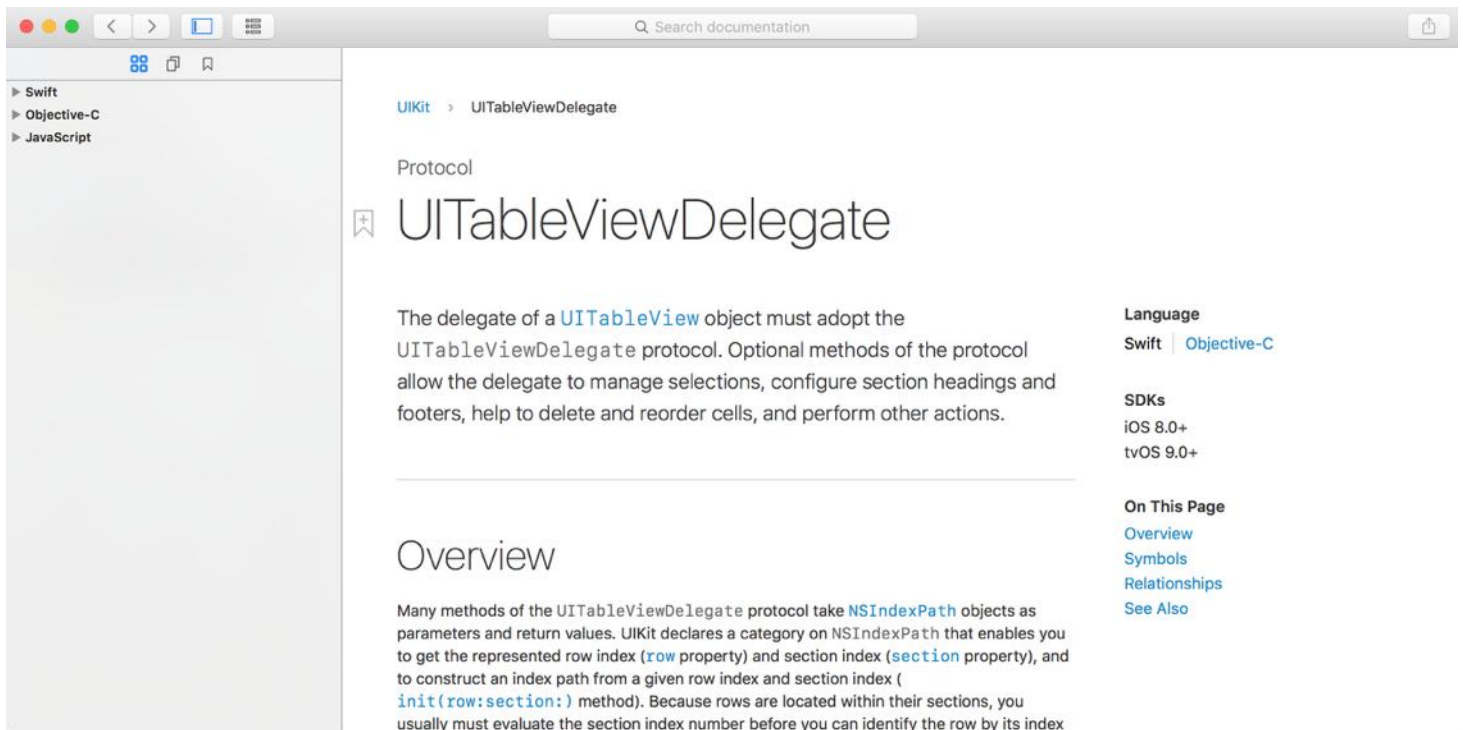


Figure 10-3. UITableViewDelegate Documentation

By glancing through the document, you will find the methods below for managing row selections:

- `func tableView(UITableView, willSelectRowAt: IndexPath)`
- `func tableView(UITableView, didSelectRowAt: IndexPath)`

Both methods are designed for row selections. The only difference is that `tableView(_:willSelectRowAt:)` is called when a specified row is about to be selected. You may use this method to prevent the selection of a particular cell from taking place. Typically, you use the `tableView(_:didSelectRowAt:)` method. It is called after the user selects a row, to handle the row selection. We will implement this method to perform additional tasks (e.g. bringing up a menu) after a row is selected.

Managing Row Selections by Implementing the Protocol

Okay, that's enough for the explanation. Let's move onto the interesting part and write some code. In the FoodPin project, open the `RestaurantTableViewController.swift` file and implement the `tableView(_:didSelectRowAt:)` method in the `RestaurantTableViewController` class:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    // Create an option menu as an action sheet
    let optionMenu = UIAlertController(title: nil, message: "What do you want
to do?", preferredStyle: .actionSheet)

    // Add actions to the menu
    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler:
nil)
    optionMenu.addAction(cancelAction)

    // Display the menu
    present(optionMenu, animated: true, completion: nil)
}
```

The above code creates an option menu by instantiating a `UIAlertController` object. When a user taps any rows in the table view, this method will be called automatically to bring up an action sheet showing a *What do you want to do* message and a *Cancel* button. Try to run the

project to have a quick test. The app should now be able to detect a touch.

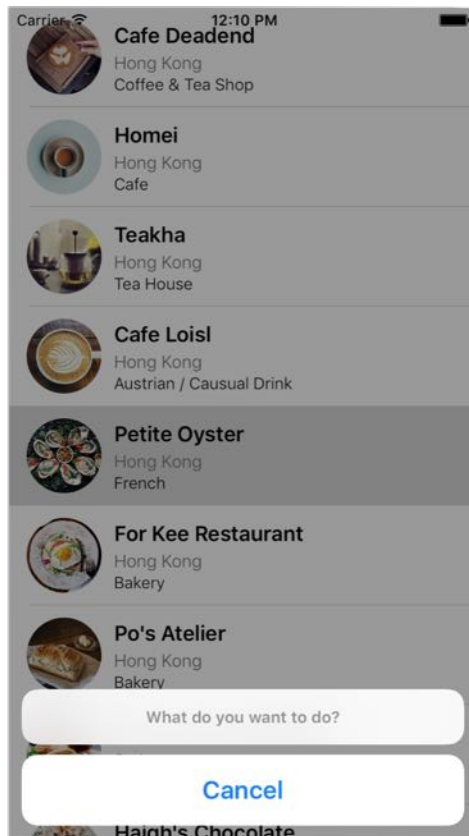


Figure 10-4. Displaying an action sheet

More on UIAlertController

Before we move on, let me talk more about the `UIAlertController` class. The `UIAlertController` class was first introduced in iOS 8 to replace both `UIAlertView` and `UIActionSheet` classes from older versions of iOS SDK. It is designed for displaying alert messages to a user.

Referring to the code snippet in the previous section, you can specify the style of the `UIAlertController` object through the `preferredStyle` parameter. You can either set its value to `.actionSheet` or `.alert`. Figure 10-5 displays the sample alert styles.

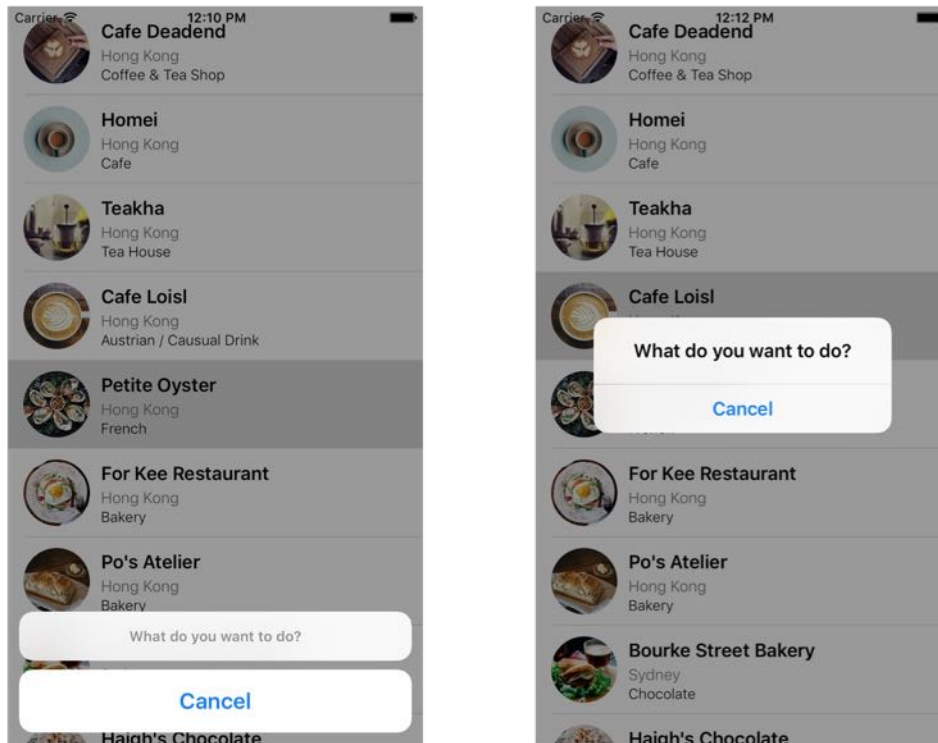


Figure 10-5. ActionSheet (left) and Alert (right)

In addition to displaying a message to users, you can assign actions to the alert controller to give users a way to respond. To do that, you create a `UIAlertAction` object with your preferred title, style, and the block of code to execute for the action. In the code snippet, we create a `cancelAction` object with the title 'Cancel' and `.Cancel` style. There is nothing to perform when a user selects the cancel action. Thus, the handler is set to `nil`. After the `UIAlertAction` object is created, you can assign it to the alert controller by using the `addAction` method.

When the alert controller is configured properly, you can simply present it using the `presentViewController` method.

This is how you use the `UIAlertController` class to present an alert. As a beginner, you may have a couple of questions in mind:

- How do I know the available values of the `preferredStyle` parameter when creating a `UIAlertController` object?
- The dot syntax looks new to me. Shouldn't it be written as `UIAlertControllerStyle.actionSheet` ?

Both are good questions.

For the first question, again the answer is "Refer to the documentation". In Xcode, you can place the cursor over the `preferredStyle` parameter and press `control-command-?`. Xcode will show the method declaration. You can further click `UIAlertControllerStyle` to read the API reference. As you can see in figure 10-6, `UIAlertControllerStyle` is an enumeration, which defines two possible values: `actionSheet` and `alert`.

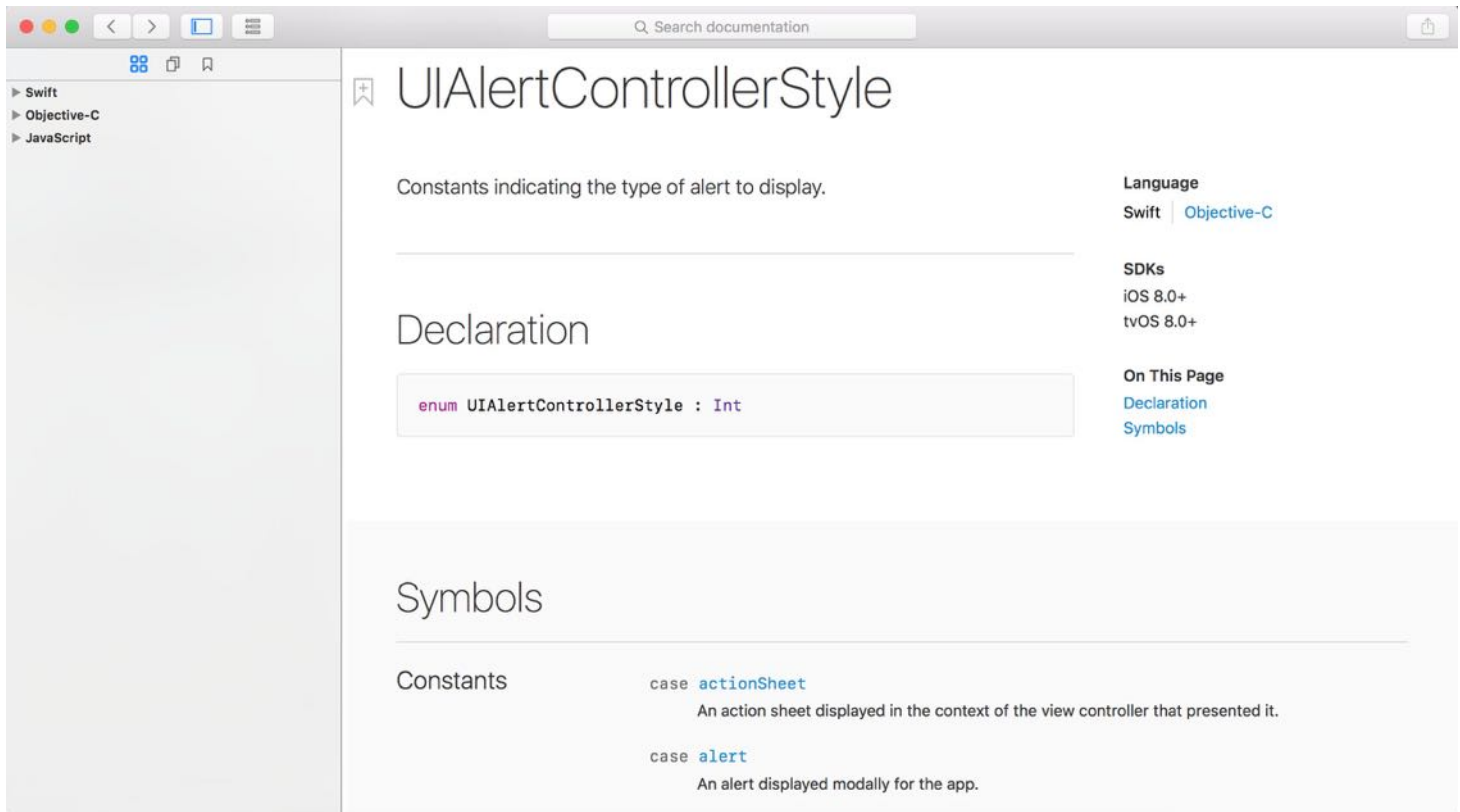


Figure 10-6. UIAlertControllerStyle

Quick note: An enumeration is a common type in Swift that defines a list of possible values for that type. The `UIAlertControllerStyle` is a good example.

We can refer to the values using `UIAlertControllerStyle.actionSheet` or `UIAlertControllerStyle.alert`. So you can write the code like this when creating a `UIAlertController`:

```
let optionMenu = UIAlertController(title: nil, message: "What do you want to do?", preferredStyle: UIAlertControllerStyle.actionSheet)
```

There is nothing wrong with the code above. Swift gives developers a shorthand and helps us type less code. Because the type of the `preferredStyle` parameter is already known (i.e. `UIAlertControllerStyle`), Swift lets you use a shorter dot syntax by omitting `UIAlertControllerStyle`. This is why we instantiate the `UIAlertController` object like this:

```
let optionMenu = UIAlertController(title: nil, message: "What do you want to do?", preferredStyle: .actionSheet)
```

The same applies to `UIAlertActionStyle`. The `UIAlertActionStyle` is an enumeration with three possible values: *default*, *cancel* and *destructive*. When creating the `cancelAction` object, we also use the shorthand syntax:

```
let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
```

Adding Actions to the Alert Controller

Now let's add two more actions to the alert controller:

- Call action - Call the selected restaurant. We will populate a fake phone number and display 'Call 123-000-x', where x is the index of the selected row.
- Check-in action - When selected, this option adds a checkmark to the selected restaurant.

In the `tableView(_:didSelectRowAt:)` method, add the following code for the "Call" action. You can insert the code after the initialization of `cancelAction`:

```
// Add Call action
let callActionHandler = { (action:UIAlertAction!) -> Void in
    let alertMessage = UIAlertController(title: "Service Unavailable", message:
    "Sorry, the call feature is not available yet. Please retry later.",
    preferredStyle: .alert)
    alertMessage.addAction(UIAlertAction(title: "OK", style: .default, handler:
    nil))
    self.present(alertMessage, animated: true, completion: nil)
}

let callAction = UIAlertAction(title: "Call " + "123-000-\(indexPath.row)",
style: .default, handler: callActionHandler)
optionMenu.addAction(callAction)
```

In the above code, you may not be familiar with the `callActionHandler` object. As mentioned

before, you can specify a block of code as a handler when creating a `UIAlertAction` object. The block of code will be executed when a user selects the action. Previously, we specify `nil` for the `cancelAction` object. That means we do not have any follow-up action for the Cancel button.

For the `callAction` object, we assign it with `callActionHandler`. The code block displays an alert, telling the user that the call feature is not yet available.

In Swift, this block of code is known as *Closure*. Closures are self-contained blocks of functionality that can be passed around in your code. It is very similar to *blocks* in Objective-C. Like the example above, one way to provide the action closure is to declare it as a constant or variable with the block of code as the value. The first part of the code block is identical to the definition of the handler parameter of `UIAlertAction`. The `in` keyword indicates that the definition of the closure's parameters and return type has finished, and the body of the closure will begin. Figure 10-7 illustrates the syntax of a closure.

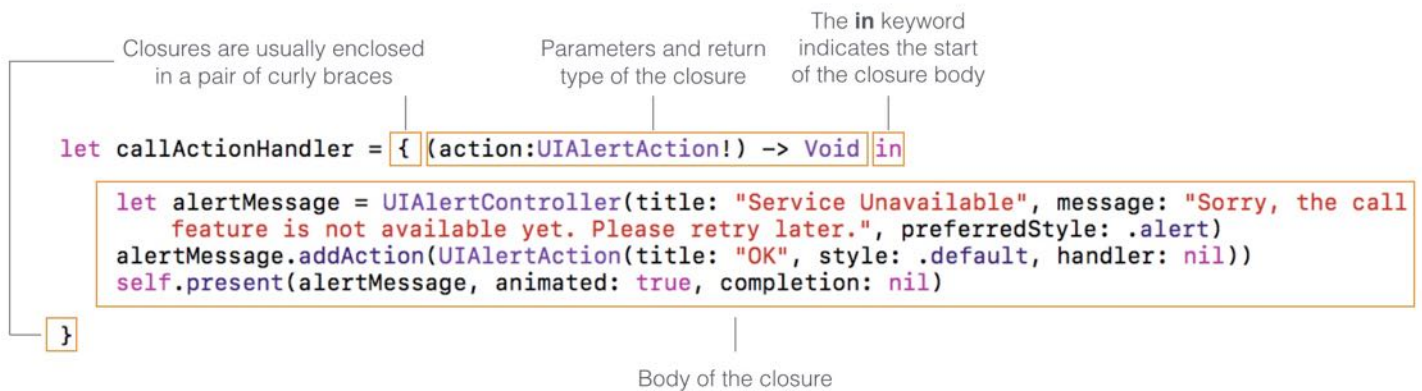


Figure 10-7. Closure

The title of the `callAction` object is a fake phone number. It is generated by concatenating '123-000-' with the selected row index. As you can see from the code, Swift allows developers to concatenate strings using the addition operator (+). In addition, you can create a string from an integer or other literals. All you need is to wrap a pair of parentheses, prefixed with a backslash like this:

```
"Call " + "123-000-\(indexPath.row)"
```

Quick note: String concatenation has been covered in the Playgrounds chapter. If you haven't gone through the exercise in chapter 2, it's time to check it out. Furthermore, you can refer to the appendix for an introduction of Swift.

Following the implementation of the Call action, add the following lines of code for the *Check-in* action:

```
// Check-in action
let checkInAction = UIAlertAction(title: "Check in", style: .default, handler:
{
    (action:UIAlertAction!) -> Void in

    let cell = tableView.cellForRow(at: indexPath)
    cell?.accessoryType = .checkmark
})
optionMenu.addAction(checkInAction)
```

The above code shows you another way to use closure. You can write the closure inline as a parameter of the handler. This is the preferred way as the code is clearer and more readable.

Optionals in Swift

You may wonder what the question mark is for. The cell is known as an optional in Swift. Optional is a new type introduced in Swift. An optional simply means 'there is a value' or 'there isn't a value at all'. The cell returned by `tableView.cellForRow(at:)` is an optional. To access the `accessoryType` property of the cell, you use the question mark. In this case, Swift will check if the cell exists and allow you to set the value of `accessoryType` if the cell exists. In most cases, the autocomplete feature of Xcode automatically adds the question mark for you when accessing a property of an optional. To learn more about Optionals, you can refer to the appendix.

When a user selects the "Check in" option, we add a checkmark to the selected cell to indicate he/she has been to the restaurant. For a table view cell, the right part is reserved for an accessory view. There are four types of built-in accessory views including *disclosure indicator*, *detail disclosure button*, *checkmark* and *detail*. In this case, we use *checkmark* as the indicator.

The first line of the code block retrieves the selected table cell using `indexPath`, which contains the index of the selected cell. The second line updates the `accessoryType` property of the cell with a check mark.

Compile and run the app. Tap a restaurant and choose one of the actions, it'll either show you a check mark or an alert.

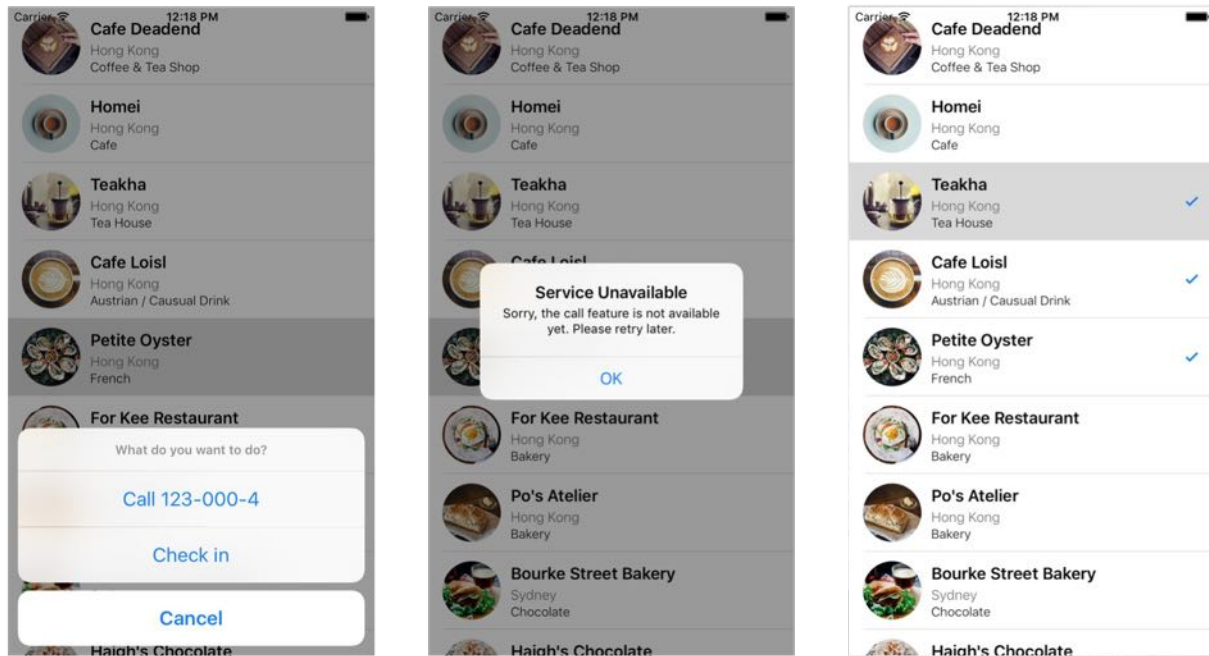


Figure 10-8. Call action and I've been here action

For now, when you select a row, the row is highlighted in gray and stayed as selected. Add the following code at the end of the `tableView(_:didSelectRowAt:)` method to deselect the row.

```
tableView.deselectRow(at: indexPath, animated: false)
```

We Hit a Bug

The app looks great. If you look at it closely, however, there is a bug in the app. Say, you mark the 'Cafe Deadend' restaurant using the *Check-in* action. If you scroll down the table, you'll find that another restaurant (e.g. Palomino Espresso) also contains a checkmark. What's the problem? Why did the app add an extra checkmark?

Like every programmer, I hate bugs especially when facing a project deadline. However it's always the bugs that help me improve my programming skills. You'll hit a lot of bugs too as you continue to learn. Just get used to it.

The problem is due to cell reuse, that we have discussed in the previous chapter. For example, the table view has 30 cells. For performance reason, instead of creating 30 table cells, `UITableView` may just create 10 cells and reuses them as you scroll through the table. In this case, `UITableView` reuses the 1st cell (originally used for Cafe Deadend with a checkmark) for displaying another restaurant. In our code, we only update the image view and labels when the table view reuses the same cell. The accessory view is not updated. Thus, the next restaurant reusing the same cell shares the same accessory view. If the accessory view contains a checkmark, that restaurant will also carry a checkmark.

So how can we resolve the bug?

We have to find another way to keep track of the checked items. How about creating another array to save the checked restaurants? In the `RestaurantTableViewController.swift` file, declare a Boolean array:

```
var restaurantIsVisited = Array(repeating: false, count: 21)
```

`Bool` is a data type in Swift that holds a Boolean value. Swift provides two Boolean values: *true* and *false*. We declare the `restaurantIsVisited` array to hold a collection of `Bool` values. Each value in the array indicates whether the corresponding restaurant is marked as "Check-in". For example, we can look into the value of `restaurantIsVisited[0]` to see if Cafe Deadend is checked or not.

The values in the array are initialized to `false`. In other words, the items are unchecked by default. The above line of code shows you a way to initialize an array in Swift with repeated values. The initialization is the same as the following:

```
var restaurantIsVisited = [false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false, false]
```

We have to make a couple of changes in order to fix the bug. First, we need to update the value of the `Bool` array when a restaurant is checked. Add a line of code in the handler of the `checkInAction` object:

```
let checkInAction = UIAlertAction(title: "Check in", style: .default, handler: {
    (action: UIAlertAction!) -> Void in
```

```
let cell = tableView.cellForRow(at: indexPath)
cell?.accessoryType = .checkmark
self.restaurantIsVisited[indexPath.row] = true
})
```

The code is very straightforward. We update the value of the selected item from `false` to `true`.

Lastly, add a few lines of code to update the accessory view in the `tableView(_:cellForRowAt:)` method before `return cell`:

```
if restaurantIsVisited[indexPath.row] {
    cell.accessoryType = .checkmark
} else {
    cell.accessoryType = .none
}
```

Here we check if the restaurant to be displayed is marked. If the condition is `true`, we display a checkmark in the cell. Otherwise, just display nothing. We now refer to the `restaurantIsVisited` array to see if the restaurant is marked. So even if the cell is reused, we can display the accessory view correctly.

Now, compile and run the app again. Your bug should now be resolved.

As a side note, you can further simplify the above if condition to a single line of code using the ternary conditional operator (`?:`):

```
cell.accessoryType = restaurantIsVisited[indexPath.row] ? .checkmark : .none
```

The ternary conditional operator is an efficient shorthand for evaluating simple condition.

Your Exercise

Presently the app doesn't allow users to uncheck the checkmark. Think about how you can alter the code such that the app can toggle the checkmark. You'll also need to show a different title for the *Undo Check in* button if the selected cell is marked. It's not too hard to make the changes.

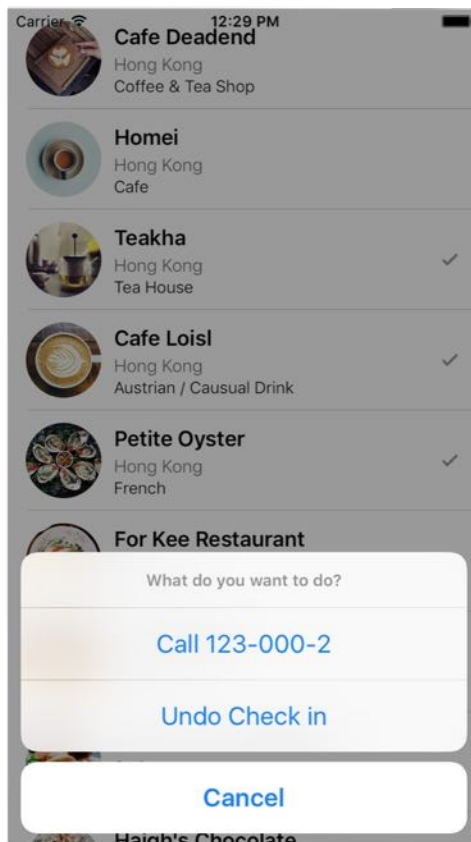


Figure 10-9. Deselecting a restaurant

Take some time to work on the exercise. I'm sure you'll learn a lot.

Summary

At this point, you should have a solid understanding about how to create table view, customize table cells and handle table row selection. You're ready to build a simple table view app on your own (e.g. a simple ToDo app). I always recommend you to create your own project. I don't mean you have to start a big one. If you love travel, create a simple app that displays a list of your favorite destinations. If you love music, create your own app that shows a list of your favorite albums. Just play around with Xcode, make mistake and learn along the way.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinTableSelection.zip>.

In the next chapter, we'll continue to explore table view and see how you can delete a table row.

Chapter 11

Table Row Deletion, Custom Action Buttons, Social Sharing and MVC



If you spend too much time thinking about a thing, you'll never get it done. Make at least one definite move daily toward your goal.

– Bruce Lee

Now you know how to handle table row selection. But how about deletion? How can we delete

a row from a table view?

It's a common question when building a table-based app. Select, delete, insert and update are the basic operations when dealing with data. We've discussed about selection. Let's talk about deletion in this chapter. In addition, we'll go through a couple of new features to the FoodPin app:

1. Adding a custom action button when a user swipes horizontally in a table row. This is usually known as *Swipe for More* action.
2. Adding a social sharing feature to the app, that enables users to share the restaurants on Twitter or Facebook.

There are a lot to learn in this chapter, but it's going to be fun and rewarding. Let's get started.

A Brief Introduction to Model View Controller

Before jumping into the coding part, I would like to give you an introduction of Model-View-Controller (MVC) model, which is one of the most quoted design patterns for user interface programming.

I try to keep this book as practical as possible and seldom talk about the programming theories. That said, you can't avoid from learning Model-View-Controller, especially your goal is to build great apps or become a competent programmer. MVC is not a concept that applies to iOS programming only. You may have heard of it if you've studied other programming languages, such as Java or Ruby. It is a powerful design pattern used in designing a software applications, whether it is a mobile app and a web app.

Understanding Model-View-Controller

At the heart of MVC, and the idea that was the most influential to later frameworks, is what I call Separated Presentation. The idea behind Separated Presentation is to make a clear division between domain objects that model our perception of the real world, and presentation objects that are the GUI elements we see on the screen. Domain objects should be completely self contained and work without reference to the presentation, they should also be able to support multiple presentations, possibly simultaneously. This

approach was also an important part of the Unix culture, and continues today allowing many applications to be manipulated through both a graphical and command-line interface.

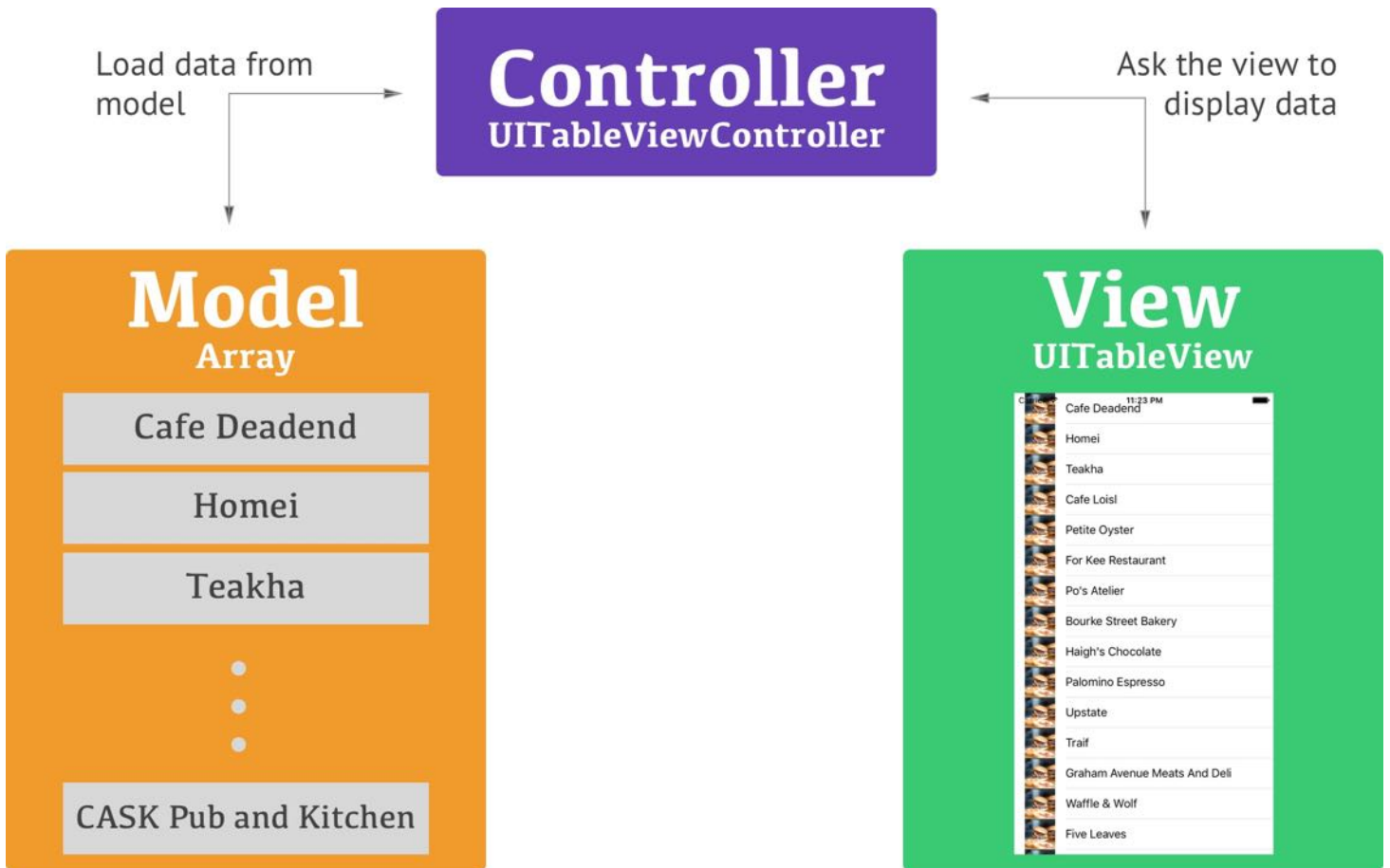
By Martin Fowler

No matter which programming language you learn, one important concept that you need to know is *Separation of Concerns (SoC)*. The concept is pretty simple. Here, the *Concerns* are different aspects of software functionality. This concept encourages developers to break a complicated feature or program into several areas of concern so that each area has its own responsibility. The delegate pattern, that we explained in the earlier chapters, is one of the examples of SoC.

The *model-view-controller (MVC)* concept is another example of SoC. The core idea behind MVC is to separate a user interface into three areas (or groups of objects) that each area is responsible for a particular functionality. As the name suggests, MVC breaks a user interface into three parts:

- **Model** – model is responsible for holding the data or any operations on the data. The model can be as simple as an array object that stores the table data. Add, update and delete are examples of the operations. In business world, these operations are usually known as business rules.
- **View** – view manages the visual display of information. For example, UITableView displays data in a list format.
- **Controller** – controller is the bridge between the model and the view. It translates the user interaction from the view (e.g. tap) into the appropriate action to be performed in the model. For example, a user taps the delete button in the view. Consequently, the controller triggers a delete operation in the model. Once finished, the model requests the view to refresh itself so as to reflect the update of the data model.

To help you better understand MVC, let's use the SimpleTable app (the one that we have built in chapter 8) as an example. The app displays a list of restaurants in the table view. If you turn the implementation into a visual illustration, here is how the table data is displayed:



The `restaurantNames` object, which is an array, is the *Model*. Each table row maps to an element of the `restaurantNames` array. The `UITableView` object is the actual *View* to be seen by a user. It's responsible for all the visuals (e.g. color of the table rows, style of the tableview, separator style, etc). The `UITableViewController` object, which is the *Controller*, acts as the bridge between the table view and the data model. It manages the table view and is responsible to load the data from the model.

Deleting a Row from UITableView

I hope you now have a better understanding of Model-View-Controller. Let's move onto the coding part and see how we can delete a row from a table view. We'll continue to develop the FoodPin app (if you haven't completed the previous exercise, you can start with by downloading the project from <http://www.appcoda.com/resources/swift3/FoodPinTableSelection.zip>) and add the "delete" feature.

If you understand the MVC model, you probably have some ideas on the implementation of

row deletion. There are three main tasks we have to do:

1. Enable the swipe-to-delete feature of the table view so that the user can select the Delete option
2. Delete the corresponding table data from the data model
3. Reload the table view to reflect the change of table data

Enable the Swipe-to-delete feature

In iOS app, users normally swipe horizontally across a table row to reveal the Delete button. Earlier, we have adopted the `UITableViewDataSource` protocol. If you've read the documentation of the protocol, you will find a method called `tableView(_:commit:forRowAt:)`. This is the method that handles the deletion (or insertion) of a specific row.

To enable the swipe-to-delete feature of a table view, all you need to do is implement the method. If the method exists, the table view will automatically display a *Delete* button when the user swipes across a row.

Try to add the following code to the `RestaurantTableViewController.swift` file and put the code inside the `RestaurantTableViewController` class:

```
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellStyle, forRowAt indexPath: IndexPath) {
}
}
```

Now let's have a quick test. Run the app on an iPhone simulator. Though the method is currently without any implementation, you should see the *Delete* button when swiping across a row.

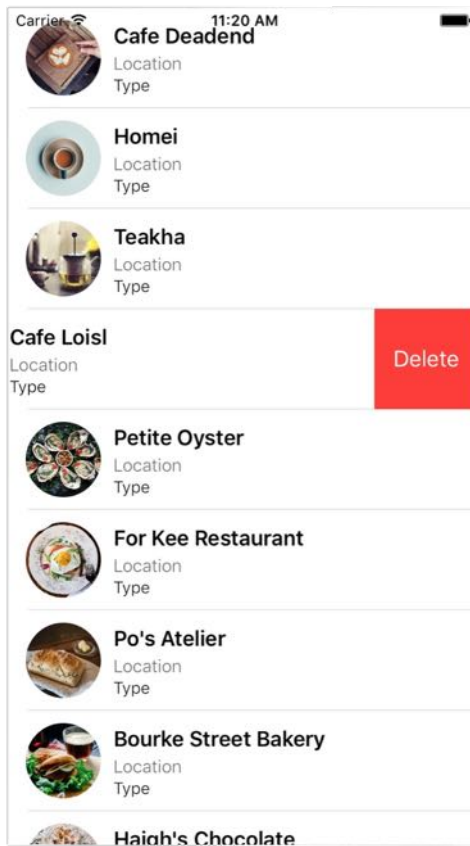


Figure 11-2. Swipe-to-delete feature

Delete Row Data from the Model

The next thing is to implement the method and write the code for removing the actual table data. From the method declaration, the `indexPath` parameter contains the row number of the cell, which is about to delete. You can make use of this information to remove the corresponding element from the data arrays.

In the FoodPin app, the `restaurantNames`, `restaurantLocations`, `restaurantTypes`, and `restaurantIsVisited` are the data model. Obviously, we have to remove the data of the selected restaurant from all of the arrays. To remove an item from an array, you can simply call the `remove(at:)` method of the array object. So update the method with the code below:

```
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellStyle, forRowAt indexPath: IndexPath) {

    if editingStyle == .delete {
```

```

// Delete the row from the data source
restaurantNames.remove(at: indexPath.row)
restaurantLocations.remove(at: indexPath.row)
restaurantTypes.remove(at: indexPath.row)
restaurantIsVisited.remove(at: indexPath.row)
restaurantImages.remove(at: indexPath.row)
}
}

```

The method supports two types of editing styles: *insert* and *delete*. Because we only remove the data when the user taps the Delete button, we first check `editingStyle` before executing the code block.

Now run and test your app again. Oops! The app doesn't work as expected. When you tap the Delete button, the cell is not removed. You may think that the data was not removed properly. Let's do some debugging here. Insert the following lines of code before the end of the method to print out the content of the array:

```

print("Total item: \(restaurantNames.count)")
for name in restaurantNames {
    print(name)
}

```

In Swift, you use the `print` method to output a message to the console. Printing the content of a variable is a very basic way of debugging. The above code prints the total number of item in the `restaurantNames` array, and its content after the cell deletion. By default, the console is hidden in Xcode. Go up to the Xcode menu, select *View > Debug Area > Activate Console* to bring up the console.

Now compile and run the app again. Delete the first row (i.e. Cafe Deadend) from the table view. You'll find the output in the console under the debug area (see figure 11-3). At first, we have 21 restaurant items in the array. After deleting a row, the number reduces to 20. And from the output, "Cafe Deadend" was deleted completely.

As you can see, the app actually deletes the item from the arrays. It looks like the view doesn't show the update. Yes, that's true. We only remove the data from the model but haven't notified the table view to update its content.

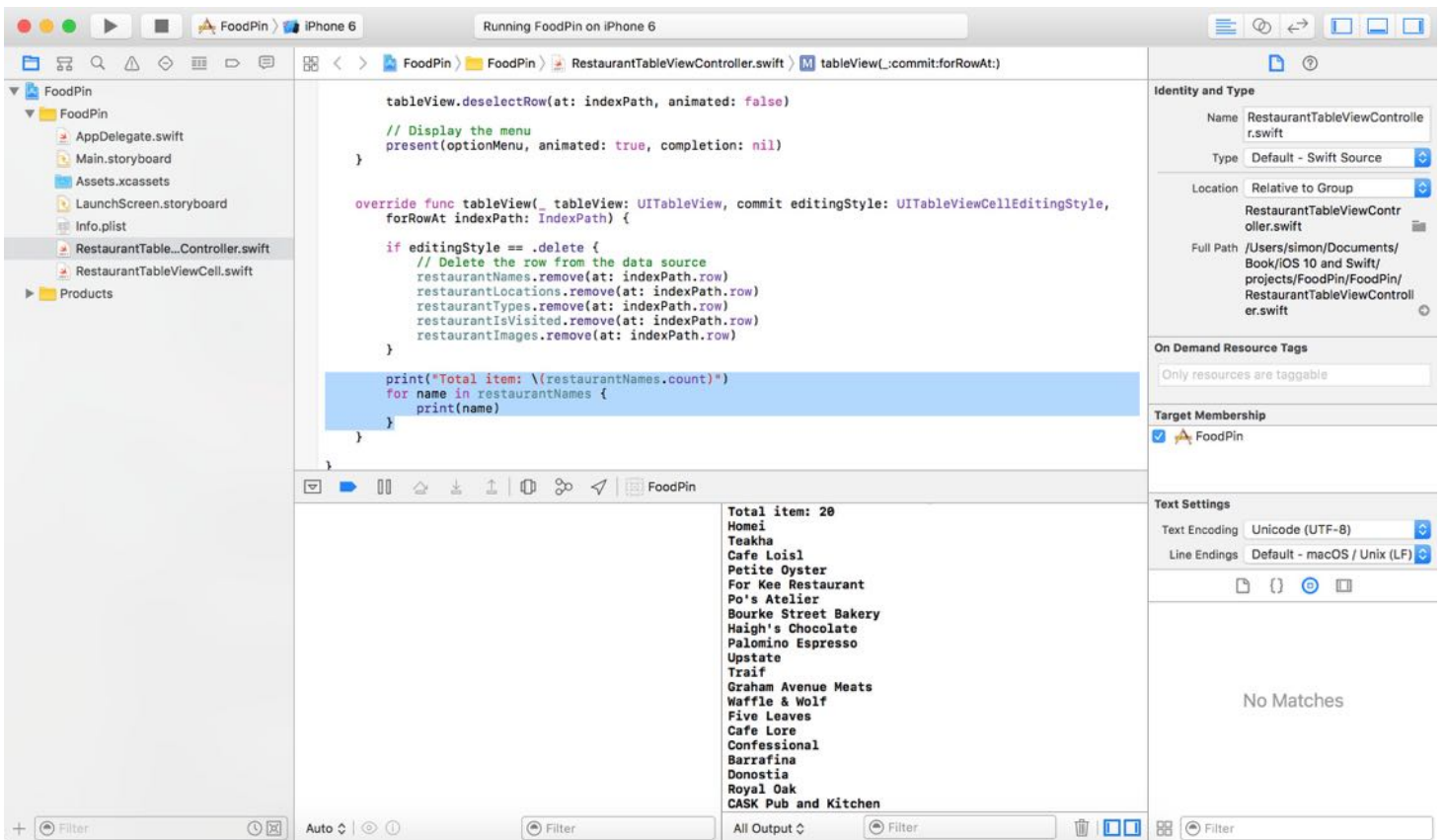


Figure 11-3. Debug area and console

Reloading UITableView

One way to ask the view to reload its content is by calling the `reloadData` method. So insert a line of code in the method to reload the data in the table view:

```
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellStyle, forRowAt indexPath: IndexPath) {

    if editingStyle == .delete {
        // Delete the row from the data source
        restaurantNames.remove(at: indexPath.row)
        restaurantLocations.remove(at: indexPath.row)
        restaurantTypes.remove(at: indexPath.row)
        restaurantIsVisited.remove(at: indexPath.row)
        restaurantImages.remove(at: indexPath.row)
    }

    tableView.reloadData()
}
```

```
print("Total item: \(restaurantNames.count)")
for name in restaurantNames {
    print(name)
}
}
```

When the `reloadData` method is called, the table view clears its content and reload the current data from the restaurant arrays to display. Now compile and test the app again. When you delete a restaurant, the table row should be removed.

Delete a Row from UITableView

The app works, but there is a better way to refresh the table view. Consider that we only need to delete a single row, why don't we just remove that particular row from the table view? You're allowed to use a method called `deleteRows(at:with:)` to delete a specific row (or multiple rows) from the table view. Replace the `reloadData` method with the following line of code:

```
tableView.deleteRows(at: [indexPath], with: .fade)
```

The `deleteRows(at:with:)` method takes in two parameters: *an array of index paths* and *the animation type*. Here we just pass the method with the current index path and specify to use the *fade* animation. The animation type indicates how the deletion is to be animated. `.fade` animation is commonly used. Optionally, you can change it to other animations such as `.right`, `.left` and `.top`. Compile and run the app again. When you confirm to delete a record, the row fades out of the table view.

Swipe for More Actions Using UITableViewRowAction

When you swipe across a table cell in the stock Mail app, you'll see a Trash button, and a More button. The More button will bring up an action sheet providing a list of options such as Reply, Flag, etc.

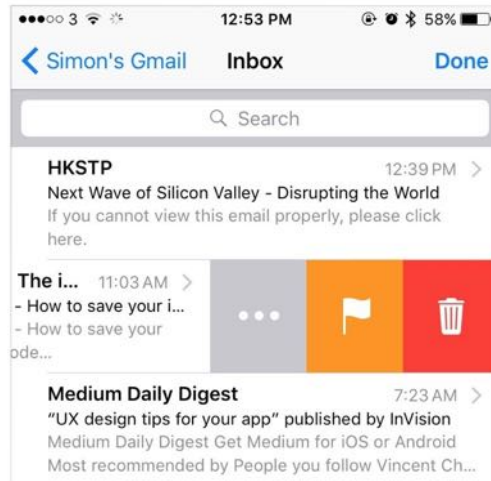


Figure 11-4. Swipe for more in the Mail app

This *swipe-for-more* feature was first introduced in the stock iPhone apps in iOS 7. At that time, Apple didn't make this feature available to developers. Starting from iOS 8, the iOS SDK comes with a new class named `UITableViewRowAction`. You use this class to create custom actions for table rows of any table view. To add custom actions to the table view's row, all you need to do is implement the `tableView(_:editActionsForRowAt:)` method, and set the custom actions as return objects.

Let's see how it works. Insert the following method in the

`RestaurantTableViewController.swift` file:

```
override func tableView(_ tableView: UITableView, editActionsForRowAt
indexPath: IndexPath) -> [UITableViewRowAction]? {

    // Social Sharing Button
    let shareAction = UITableViewRowAction(style:
UITableViewRowActionStyle.default, title: "Share", handler: { (action,
indexPath) -> Void in

        let defaultText = "Just checking in at " +
self.restaurantNames[indexPath.row]
        let activityController = UIActivityViewController(activityItems:
[defaultText], applicationActivities: nil)
        self.present(activityController, animated: true, completion: nil)
    })

    // Delete button
    let deleteAction = UITableViewRowAction(style:
```

```

UITableViewRowActionStyle.default, title: "Delete", handler: { (action,
indexPath) -> Void in

    // Delete the row from the data source
    self.restaurantNames.remove(at: indexPath.row)
    self.restaurantLocations.remove(at: indexPath.row)
    self.restaurantTypes.remove(at: indexPath.row)
    self.restaurantIsVisited.remove(at: indexPath.row)
    self.restaurantImages.remove(at: indexPath.row)

    self.tableView.deleteRows(at: [indexPath], with: .fade)
})

return [deleteAction, shareAction]
}

```

The usage of `UITableViewRowAction` is very similar to that of `UIAlertAction`. You specify the title, the style and the block of code to execute when a user taps the button. In this example, we name the custom action as *Share*. When a user taps the button, it brings up an activity controller for social sharing.

The `UIActivityViewController` class is a standard view controller that provides several standard services, such as copying items to the clipboard, sharing content to social media sites, sending items via Messages, etc. The class is very simple to use. Let's say you have a message for sharing. All you need to do is create an instance of `UIActivityViewController` with the message object, and then present the controller on screen. That is what we have done in the above code snippet.

You may notice that we added a delete action button. Once you override the `tableView(_:editActionsForRowAt:)` method and provide your own implementation, the table view will no longer generate the Delete button for you. This is why we need to create our own Delete button.

The last line of the code is probably the most important part. It returns an array of `UITableViewRowAction` objects (i.e. `deleteAction` and `shareAction`) telling the table view to create the buttons when someone swipes across the cell.

Compile and run the app. Swipe across a table row and it'll show you both Share and Delete buttons. Tapping the Share button will bring up a share menu like the one shown in figure 11-5.

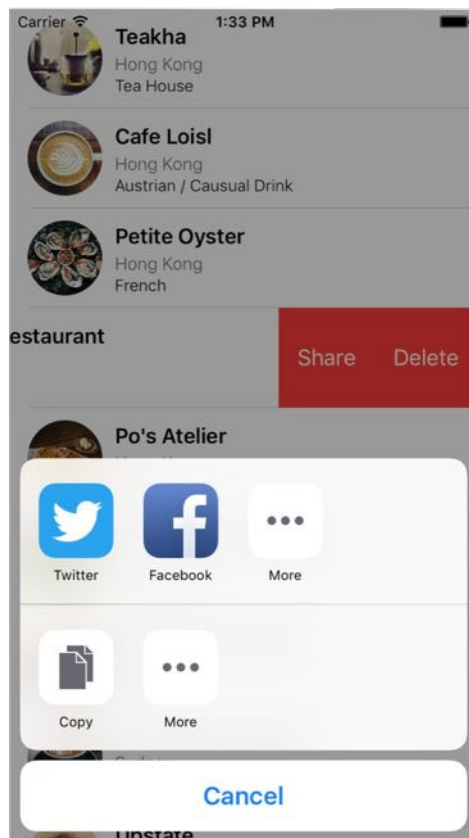


Figure 11-5. Swipe to Share button

The Twitter and Facebook buttons may not appear in your simulator. In this case, press `shift-command-H` to go back to the Home screen. Select Settings > Twitter / Facebook, and sign in with your account. Then re-launch the app. You should be able to share the content on social media sites.

The `UIActivityViewController` class does not limit you from sharing content in text format. If you pass it a `UIImage` object during initialization, your app will allow users to post images to Twitter or Facebook. Modify the `tableView(_:editActionsForRowAt:)` method to the following:

```
override func tableView(_ tableView: UITableView, editActionsForRowAt
indexPath: IndexPath) -> [UITableViewRowAction]? {

    // Social Sharing Button
    let shareAction = UITableViewRowAction(style:
UITableViewRowActionStyle.default, title: "Share", handler: { (action,
indexPath) -> Void in

        let defaultText = "Just checking in at " +
```



```

self.restaurantNames[indexPath.row]

        if let imageToShare = UIImage(named:
self.restaurantImages[indexPath.row]) {
            let activityController = UIActivityViewController(activityItems:
[defaultText, imageToShare], applicationActivities: nil)
            self.present(activityController, animated: true, completion: nil)
        }
    })

    // Delete button
    let deleteAction = UITableViewRowAction(style:
UITableViewRowActionStyle.default, title: "Delete", handler: { (action,
indexPath) -> Void in

        // Delete the row from the data source
        self.restaurantNames.remove(at: indexPath.row)
        self.restaurantLocations.remove(at: indexPath.row)
        self.restaurantTypes.remove(at: indexPath.row)
        self.restaurantIsVisited.remove(at: indexPath.row)
        self.restaurantImages.remove(at: indexPath.row)

        self.tableView.deleteRows(at: [indexPath], with: .fade)
    })

    return [deleteAction, shareAction]
}

```

We just add a couple of lines in the above code to create an `imageToShare` object for sharing. We first load the image by using the `UIImage` class, and then pass it to `UIActivityViewController` during initialization.

```

let activityController = UIActivityViewController(activityItems: [defaultText,
imageToShare], applicationActivities: nil)

```

`UIActivityViewController` will automatically embed the image when the user shares the restaurant to social media sites.

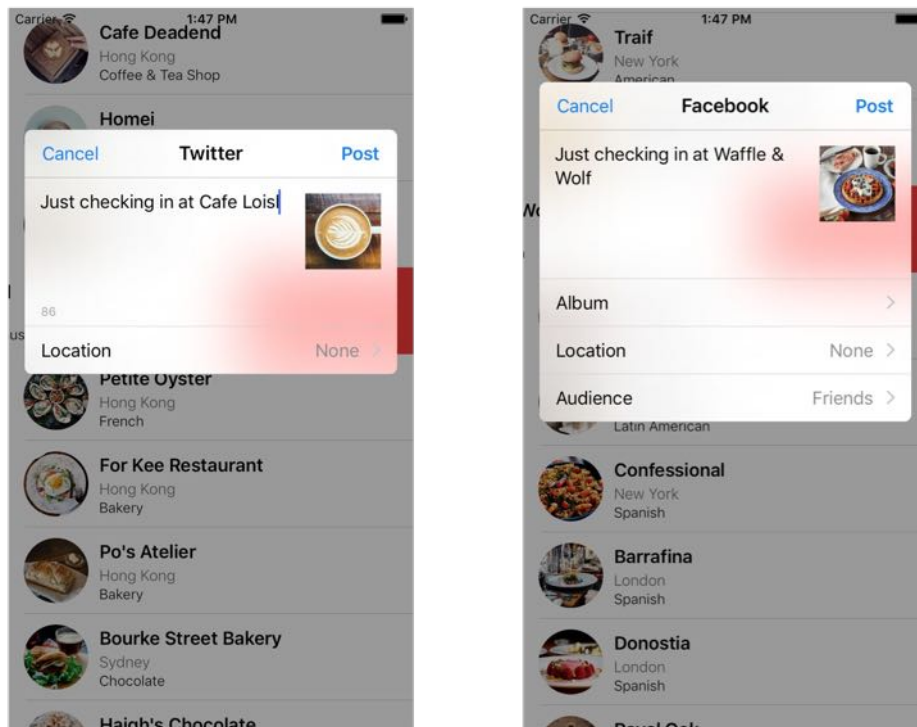


Figure 11-6. Posting content to Twitter and Facebook

if let for Optionals

When loading an image, it is possible that the image is failed to load. This is why the `UIImage` class returns an optional during initialization. In Swift, we use `if let` to verify if an optional contains a value or not. For details, you can refer to the `Optional` section in the appendix.

Customize UITableViewRowAction

By default, the action buttons are in red. The `UITableViewRowAction` class provides an option for developers to customize its background color through the `backgroundColor` property:

```
shareAction.backgroundColor = UIColor(red: 48.0/255.0, green: 173.0/255.0,
blue: 99.0/255.0, alpha: 1.0)
deleteAction.backgroundColor = UIColor(red: 202.0/255.0, green: 202.0/255.0,
blue: 203.0/255.0, alpha: 1.0)
```

The UIKit framework provides a `UIColor` class to represent color. Many methods in UIKit

require you to provide color using a `UIColor` object. The class comes with a number of standard colors such as `UIColor.blueColor()` and `UIColor.redColor()`. If you want to use your own color, you can create your own `UIColor` object using RGB component values. The component values should be within 0 and 1. If you're a web designer or have some experience with graphics design, you know that the RGB values are on a scale of 0 to 255. To conform with the requirement of `UIColor`, you have to divide each of the component values by 255 when creating a `UIColor` object. You can paste the code above before the following line of code in the `tableView(_:editActionsForRowAt:)` method:

```
return [deleteAction, shareAction]
```

Here we go. Test the app again and see if you like the new color. Otherwise, modify the color code and change to your preferred color.

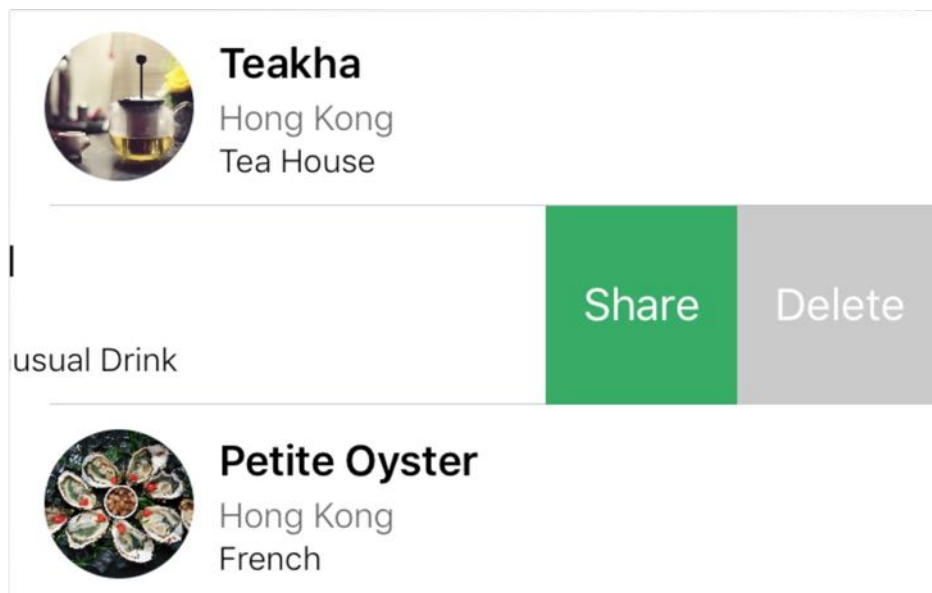


Figure 11-7. Posting content to Twitter and Facebook

Quick note: Like me, if you are not a designer, you may need some color inspiration. You can check out Adobe Color CC (color.adobe.com) and Flat UI Color Picker (flatuicolorpicker.com). You'll find a lot of color combinations for designing your app.

Summary

In this chapter, I gave you a brief overview of MVC, showed you how to manage deletion in table view, and taught you how to create a swipe-for-share button. You also learned how to use `UIActivityViewController` for implementing social sharing feature. The FoodPin app is getting better and better. You should be proud of your achievement so far.

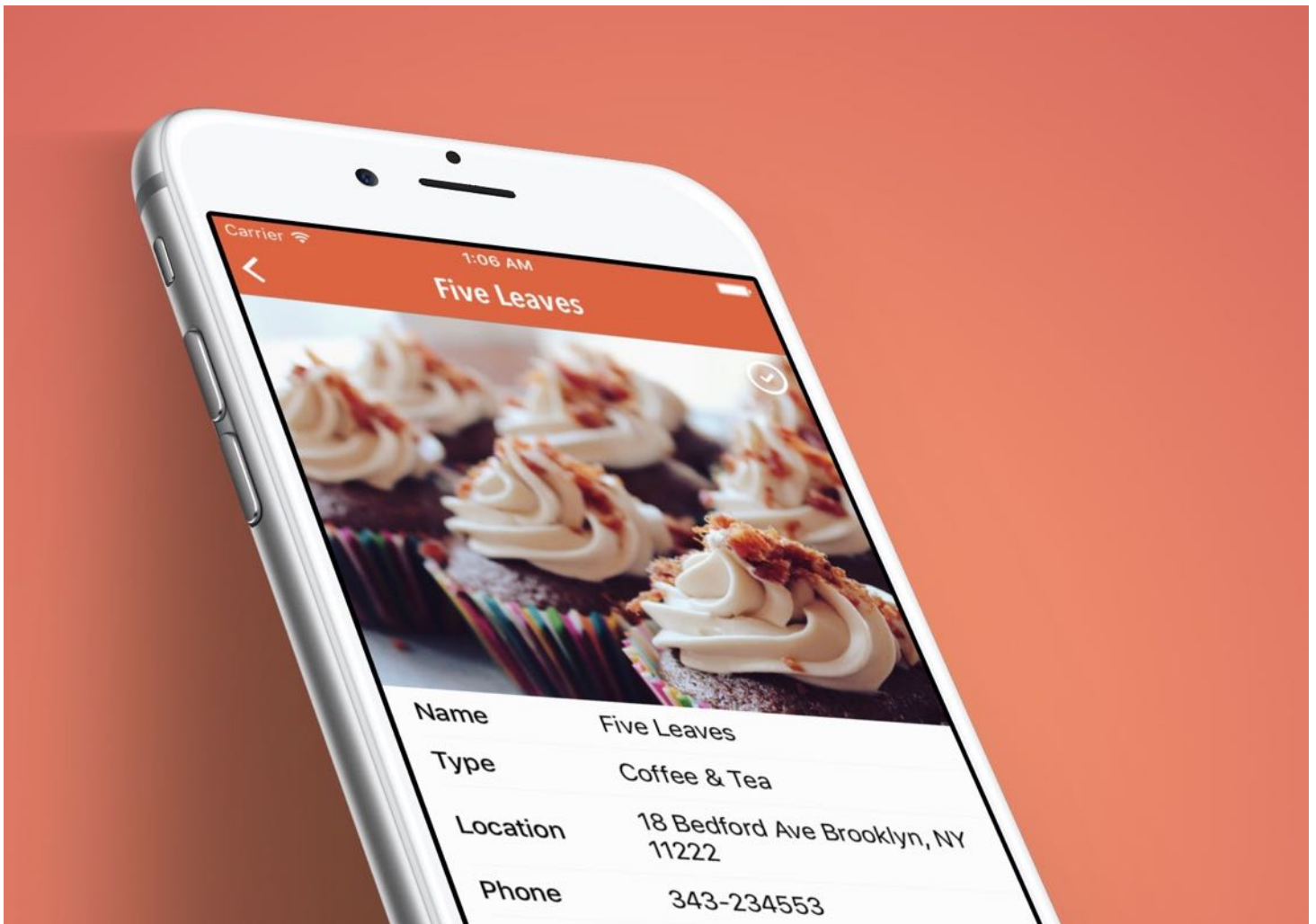
The MVC concept is important. If you're a programming newbie, it may take you some time to fully understand the material. Feeling confused? Just take a break and grab yourself a coffee. After your break, go over the chapter again. Probably you'll find it easier to digest.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinDeleteRow.zip>.

In the next chapter, we'll take a look at something new and create a navigation controller.

Chapter 12

Introduction to Navigation Controller and Segue



Just build something that you'd want to use today, not something you think people would use somehow.

– Paul Graham

First things first, what's navigation controller? Like table views, navigation controllers are another common UI components in iOS apps. It provides a drill-down interface for hierarchical content. Take a look at the built-in Photos app, YouTube, and Contacts. They all

use navigation controllers to display hierarchical content. Generally, you combine a navigation controller with a stack of table view controllers to build an elegant interface for your apps. Being that said, this doesn't mean you have to use both together. Navigation controllers can work with any types of view controller.

Scenes and Segues in Storyboards

Up till now, we just layout a table view controller in the storyboard of the FoodPin app. Storyboarding allows you to do more than that. You can add more view controllers in the storyboard, link them up, and define the transitions between them. All these can be done without a line of code. When working with storyboards, *scene* and *segues* are two of the terms you have to know. In a storyboard, a scene usually refers to the on-screen content (e.g. a view controller). Segues sit between two scenes and represent the transition from one scene to another. *Push* and *Modal* are two common types of transition.

Quick note: Since the release of Xcode 7, you can make storyboards more managable and modular by using a feature called storyboard references. When your project becomes large and complex, you can break a large storyboard into multiple storyboards and link them up using storyboard references. This feature is particularly useful when you are collaborating with your team members to create a storyboard.

Creating Navigation Controller

We'll continue to work on the Food Pin app by embedding the table view controller into a navigation controller. When a user selects any of the restaurants, the app navigates to the next screen to display the restaurant details.

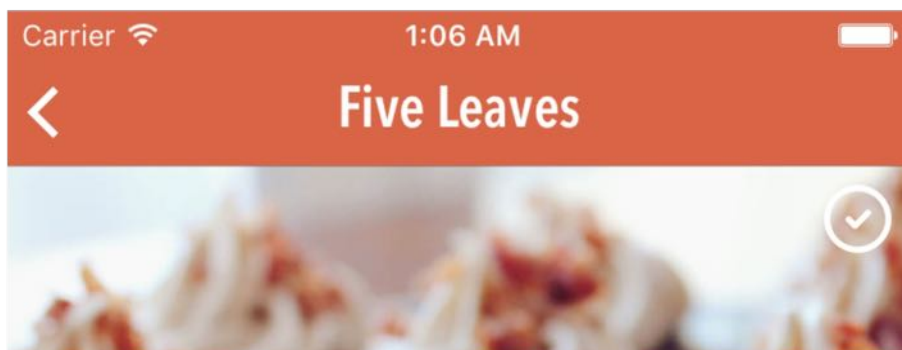


Figure 12-1. FoodPin app with a navigation controller

If you've closed the FoodPin project, it is time to fire up Xcode and open the project again. Select `Main.storyboard` to switch to the Interface Builder editor. Xcode provides an *embed* feature that makes it easy to embed any view controller in a navigation controller. Select the table view controller and click *Editor* in the menu. Choose *Embed in > Navigation Controller*.

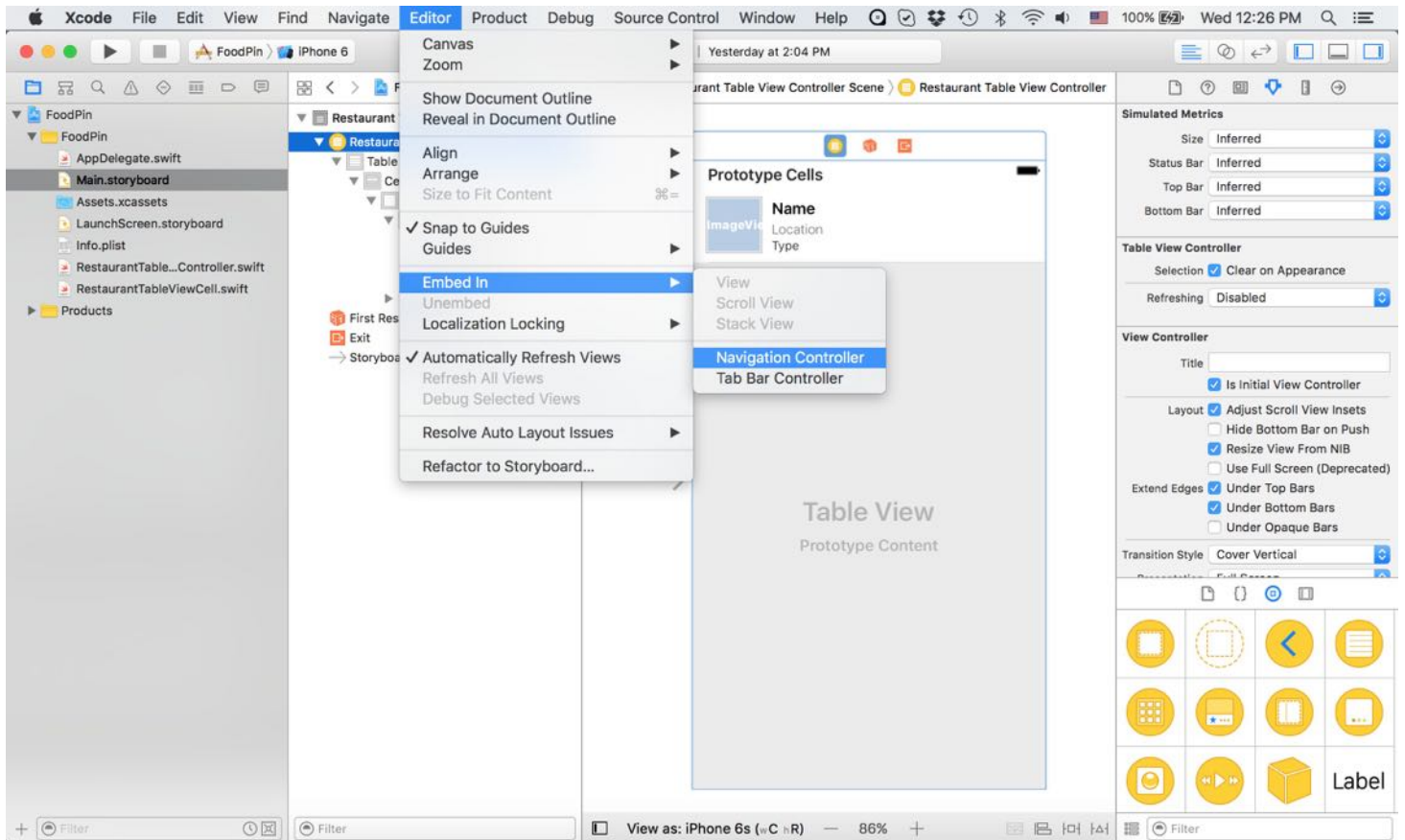


Figure 12-2. Embed in option in Xcode menu

Xcode automatically embeds the controller in a navigation controller. Let's set the title of the navigation bar. Select the navigation bar of the table view controller. Under the Attributes inspector, change the value of the title to *Food Pin*.

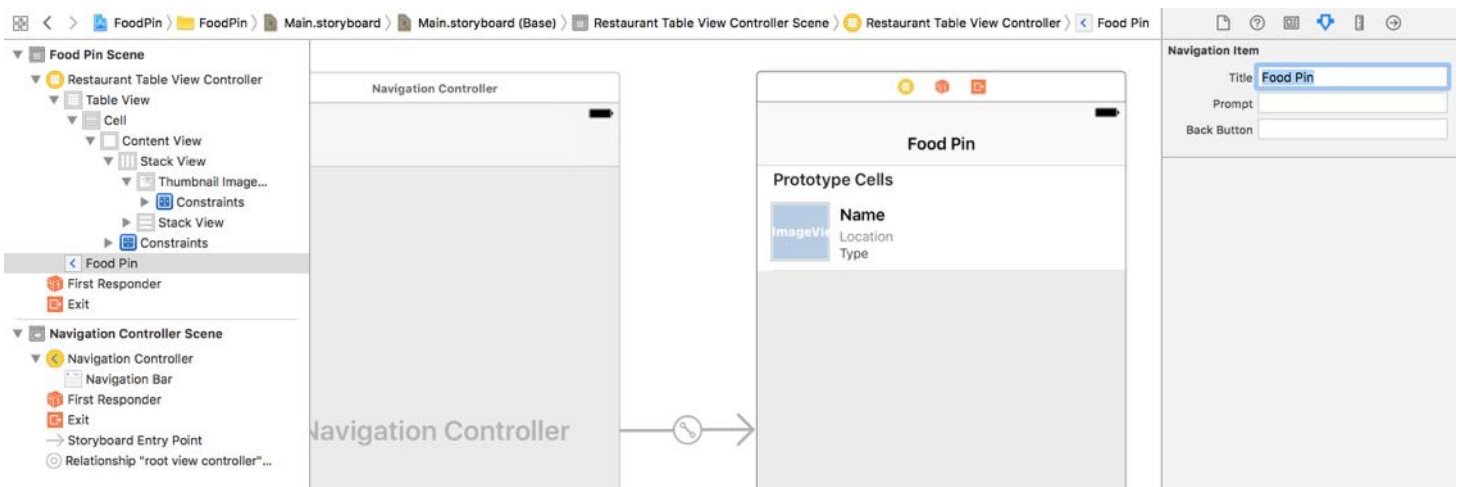


Figure 12-3. Embedding the table view controller in a navigation controller

Now run the app and see how it looks. The app is the same as before, but added with a navigation bar.

Adding a Detail View Controller

Easy, right? With just a few clicks, you've added a navigation bar to your app. What's missing is another view controller that displays the restaurant details. When a user selects a restaurant, the app transits to the detail view controller and displays the restaurant details.

In Interface Builder, drag a view controller from the Object library to create the detail view controller. The primary purpose of this chapter is to show you how to implement navigation controller. We'll keep the detail view as simple as possible. Let's just display the restaurant photo in the detail view. Drag an Image View from Object Library to the view controller. Resize it to fit the view, and add spacing constraints for each side of the image view. To ensure the image is scaled properly, go to the Attributes inspector and change the *mode* from *Scale* to *Fill* to *Aspect Fill*.

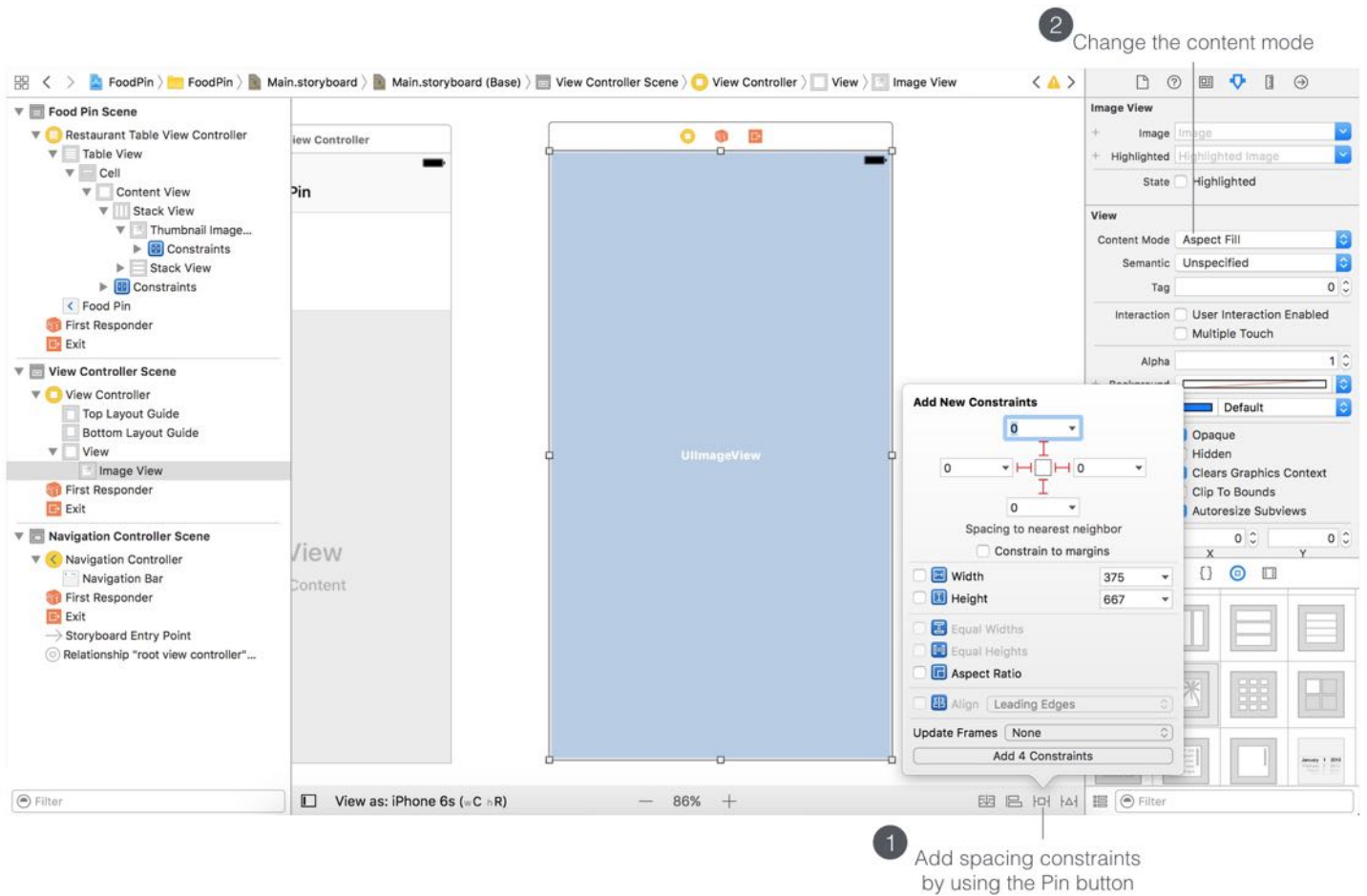


Figure 12-4. Adding an image view to the detail view controller

Now you have two view controllers configured in the storyboard. The question is how can you connect them together? In storyboards, we need to connect them through a segue. In music, a segue is a seamless transition between one piece of music and another. In storyboards, the transition between two scenes is called *segue*.

The table view controller will transit to the detail scene when a user taps a cell. So we will add a segue to connect the prototype cell and the detail scene. It's very straightforward to add a segue object. Press and hold the control key, click on the prototype cell and drag to the Detail View Controller.

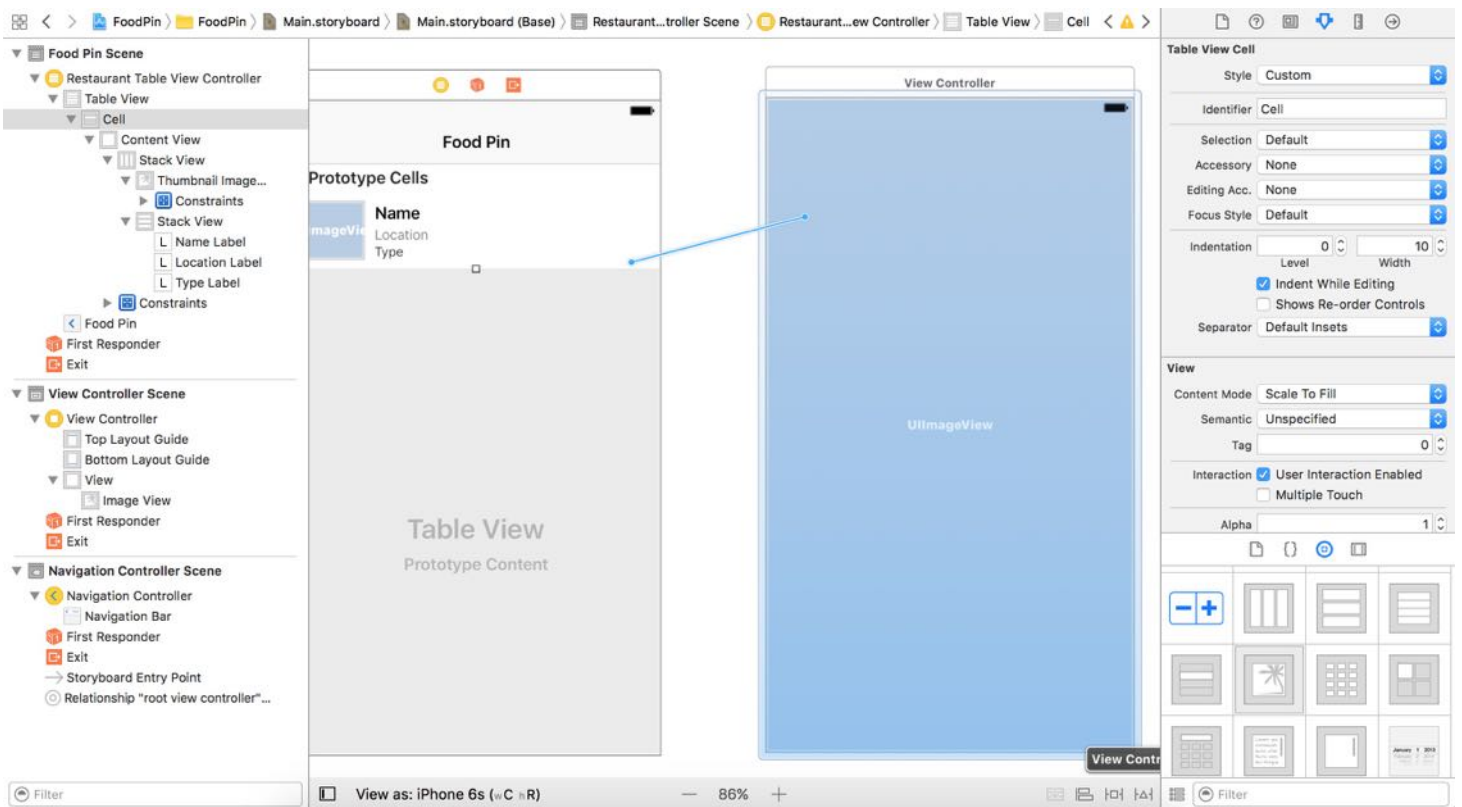
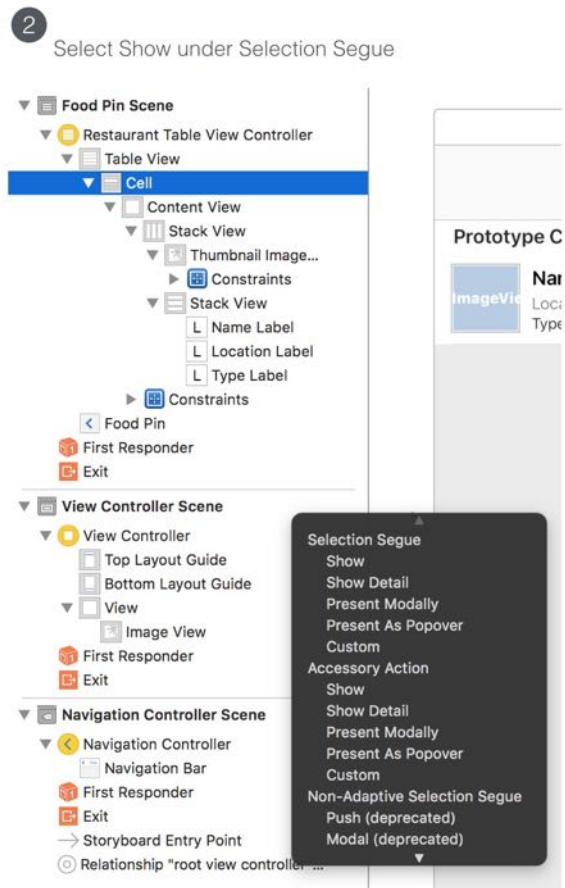
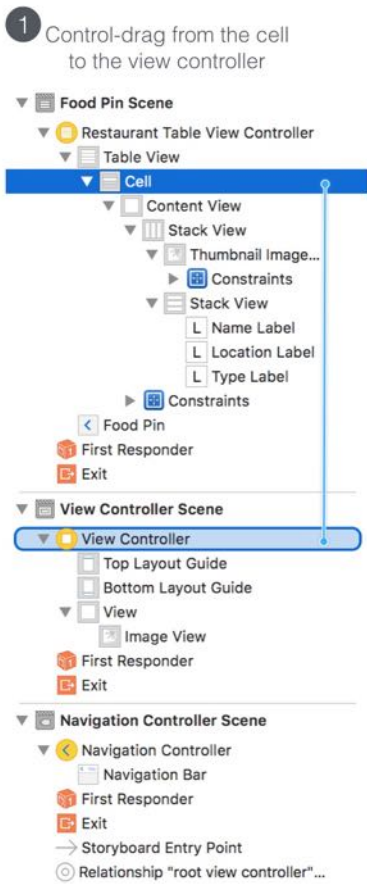


Figure 12-5. Connecting scenes with segue

If you find it difficult to select the prototype cell in the storyboard editor, open the document outline view, and drag from the prototype cell to the view controller. When you release the buttons, a pop-up menu appears for you to choose the style for the segue. Select "Show" for the style, and you'll see a connector between the view controllers.



Once chosen, Xcode automatically connects the prototype cell and the detail view controller with a *show* segue.

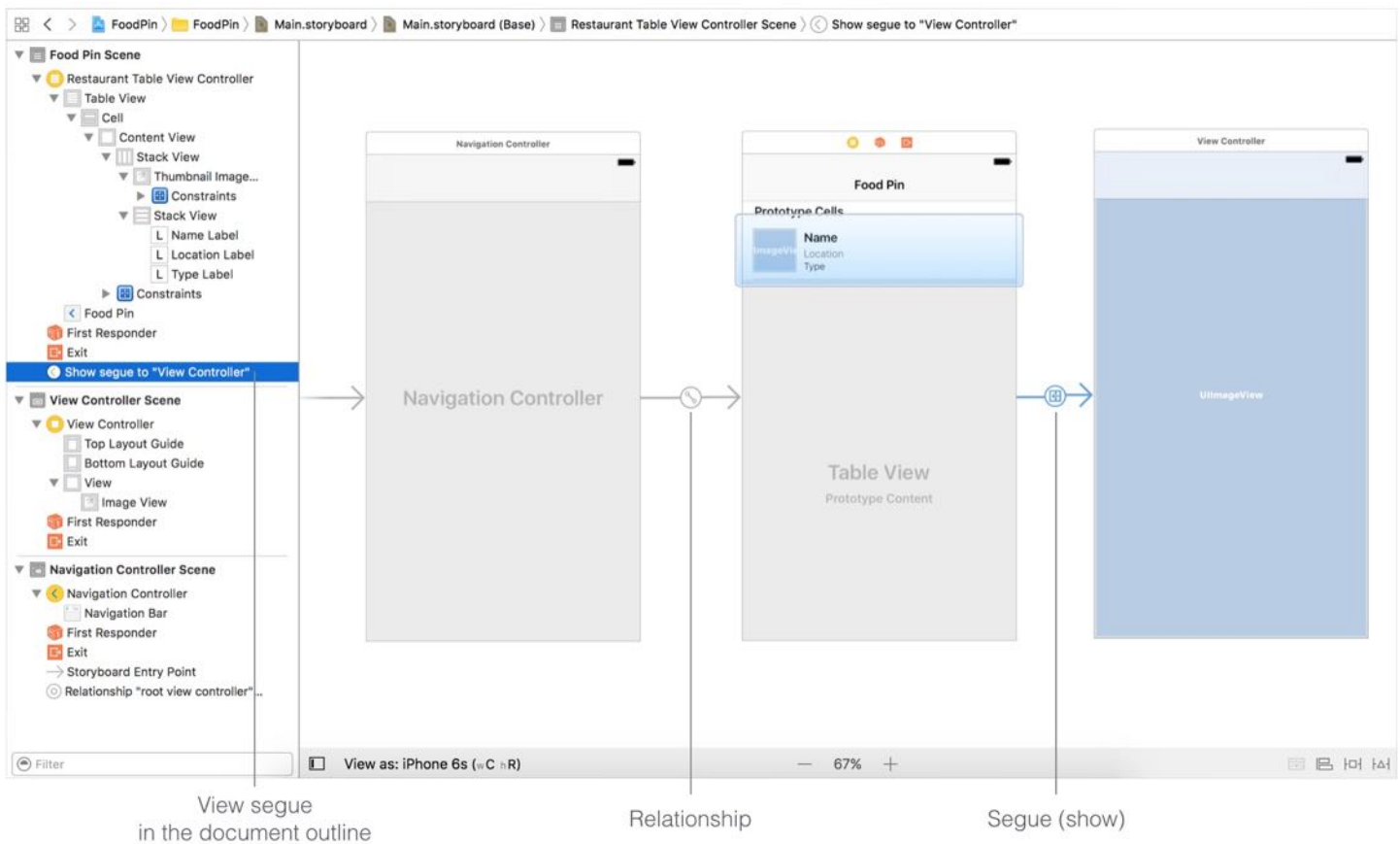


Figure 12-7. A Segue is defined

In iOS 10, it supports the following types of segue:

- **Show** - when the *show* style is used, the content is pushed on top of the current view controller stack. A back button will be displayed in the navigation bar for navigating back to the original view controller. Typically, we use this segue type for navigation controllers.
- **Show detail** - similar to the *show* style, but the content in the detail (or destination) view controller replaces the top of the current view controller stack. For example, if you select *show detail* instead of *show* in the FoodPin app, there will be no navigation bar and back button in the detail view.
- **Present modally** - presents the content modally. When used, the detail view controller will be animated up from the bottom and cover the entire screen on iPhone. A good example of *present modally* segue is the "Add" feature of the built-in Calendar app. When you click the + button in the app, it brings up a "New Event" screen from the bottom.
- **Present as popover** - Present the content as a popover anchored to an existing view.

Popover is commonly found in iPad apps. Figure 12-8 shows you an example. Starting from iOS 8, you can use popover segue on iPhone apps too.

Quick note: These segue types are deprecated since the release of iOS 8: Push, Modal, Popover, Replace.

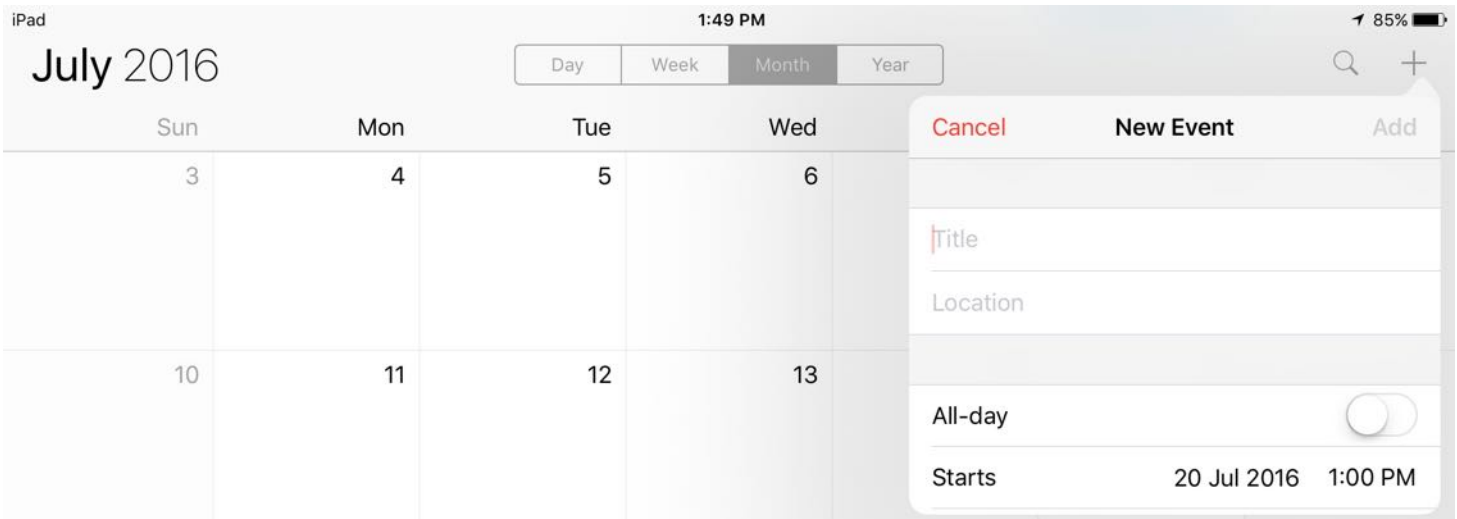


Figure 12-8. A sample popover segue in the stock Calendar app

Now, you're ready to run the app. Once launched, select a restaurant and the app should navigate to the detail view controller. Meanwhile, the detail view controller just shows a blank screen. The good news is that you already created a navigation interface.

Without writing a single line of code, you've added a navigation controller in your app. However, I guess you got a couple of questions in your mind:

- How can you pass the restaurant information from `RestaurantTableViewController` to the detail view controller?
- How can you display the photo of the selected restaurant in the detail view controller?

We'll look into them one by one in a minute.

Before moving on to the next section, let's make a little tweak. When you ran the app and select a cell, it now navigates to the blank screen and show an action sheet that we have implemented in chapter 10. We no longer need that action sheet. Later we'll add the same function in the

detail view controller. Therefore, remove the `tableView(_:didSelectRowAt:)` method from `RestaurantTableViewController.swift`.

Quick tip: Sometimes you may just want to comment out a block of code instead of deleting them. Xcode provides a shortcut key to comment multiple lines of code. First, select the block of code and press `command+slash`. Xcode automatically marks the code block with comment indicators. To remove the comment, do the same procedures again.

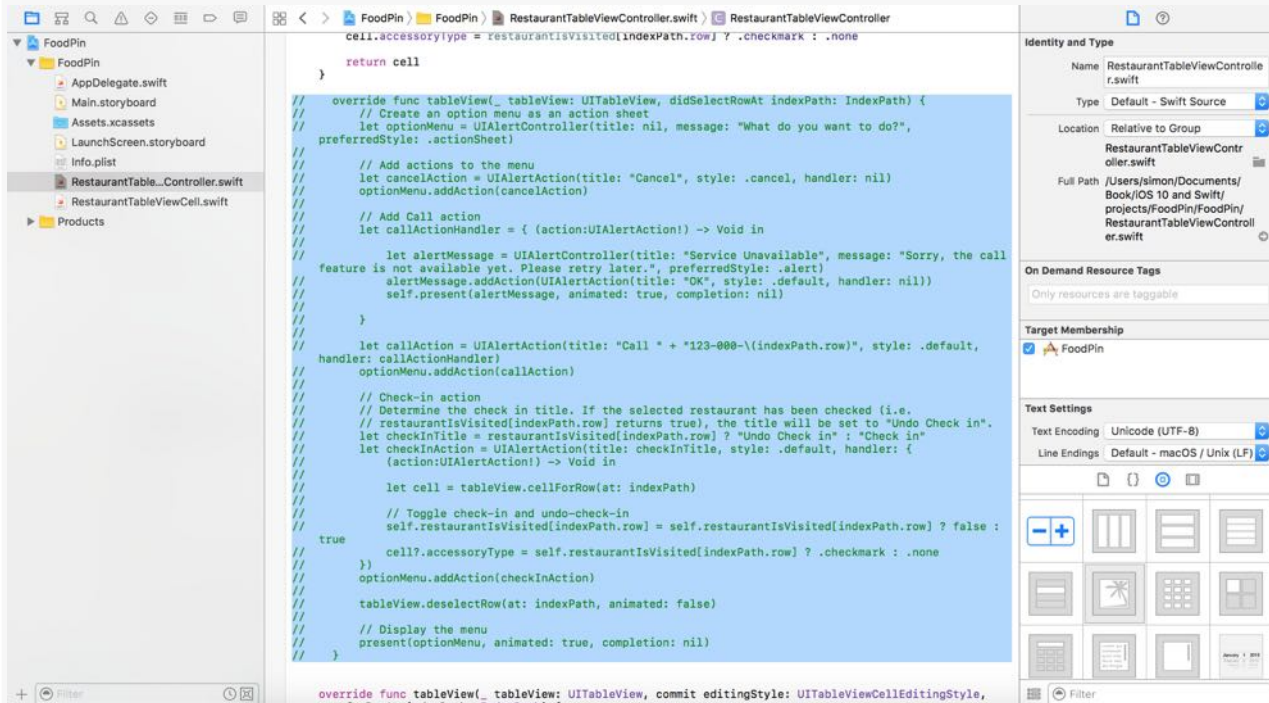


Figure 12-9. Use a shortcut key to comment a block of code quickly

Creating a New Class for the Detail View Controller

Okay, let's go back to the detail view controller. Our goal is to update the image view in the view controller with the selected restaurant. The view controller is now associated with the `UIViewController` class by default. The fact is the `UIViewController` class only provides the fundamental view management model. There is no variable for storing the restaurant image. Obviously, we have to extend `UIViewController` to create our own class so that we can add a new variable for the image view.

In the Project Navigator, right-click the *FoodPin* folder and select *New File...* Choose *Cocoa Touch Class* as the class template. Name the class `RestaurantDetailViewController` and set it as a subclass of `UIViewController` . Click *Next* and save the file in your project folder.

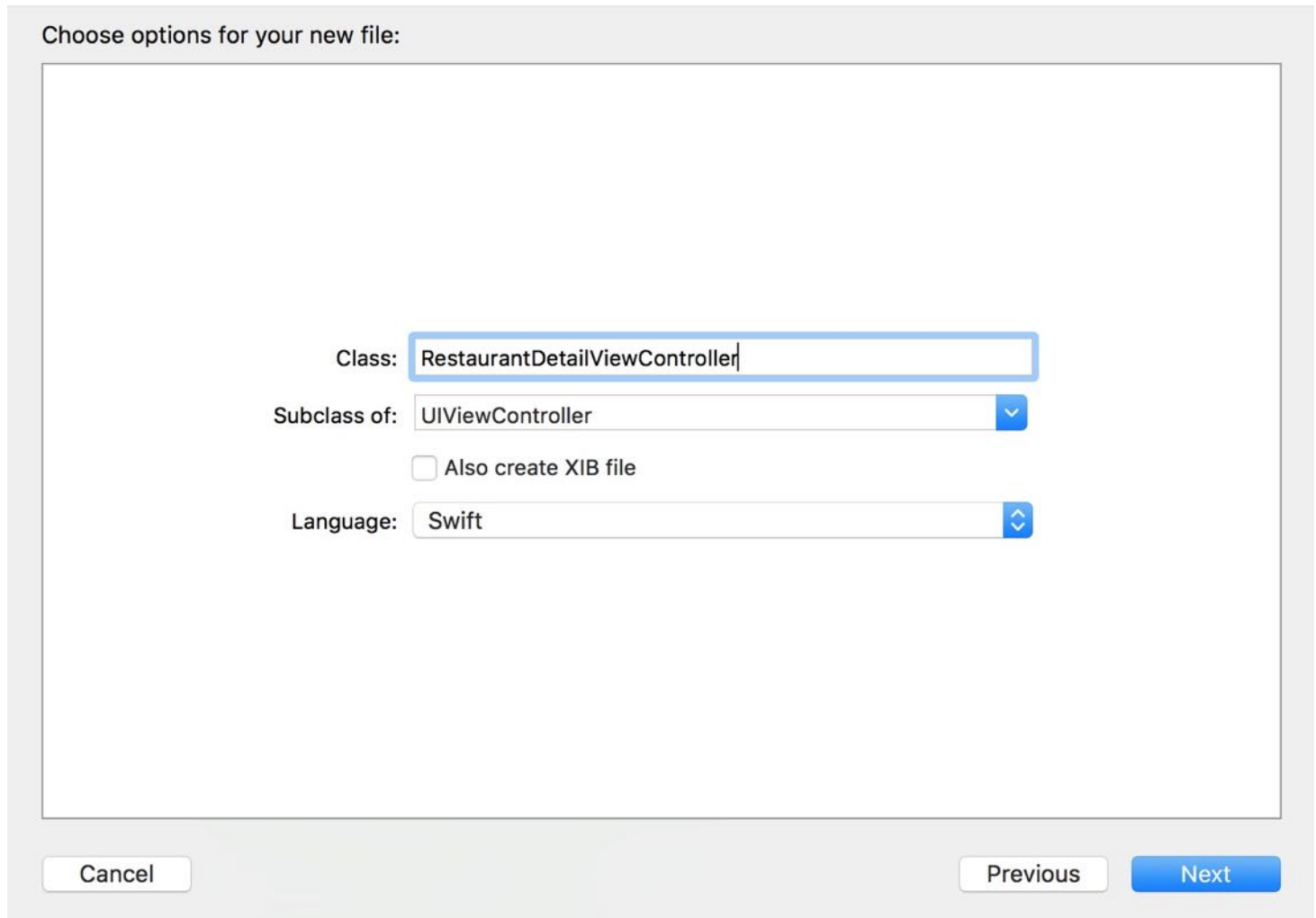


Figure 12-10. Create a DetailViewController class

First, we want to establish a relationship between the view controller in Interface Builder and the new class. Go to `Main.storyboard` , and assign the `DetailViewController` class to the detail view controller. In Interface Builder, select the detail view controller and open the Identity inspector. Change the custom class to `RestaurantDetailViewController` .

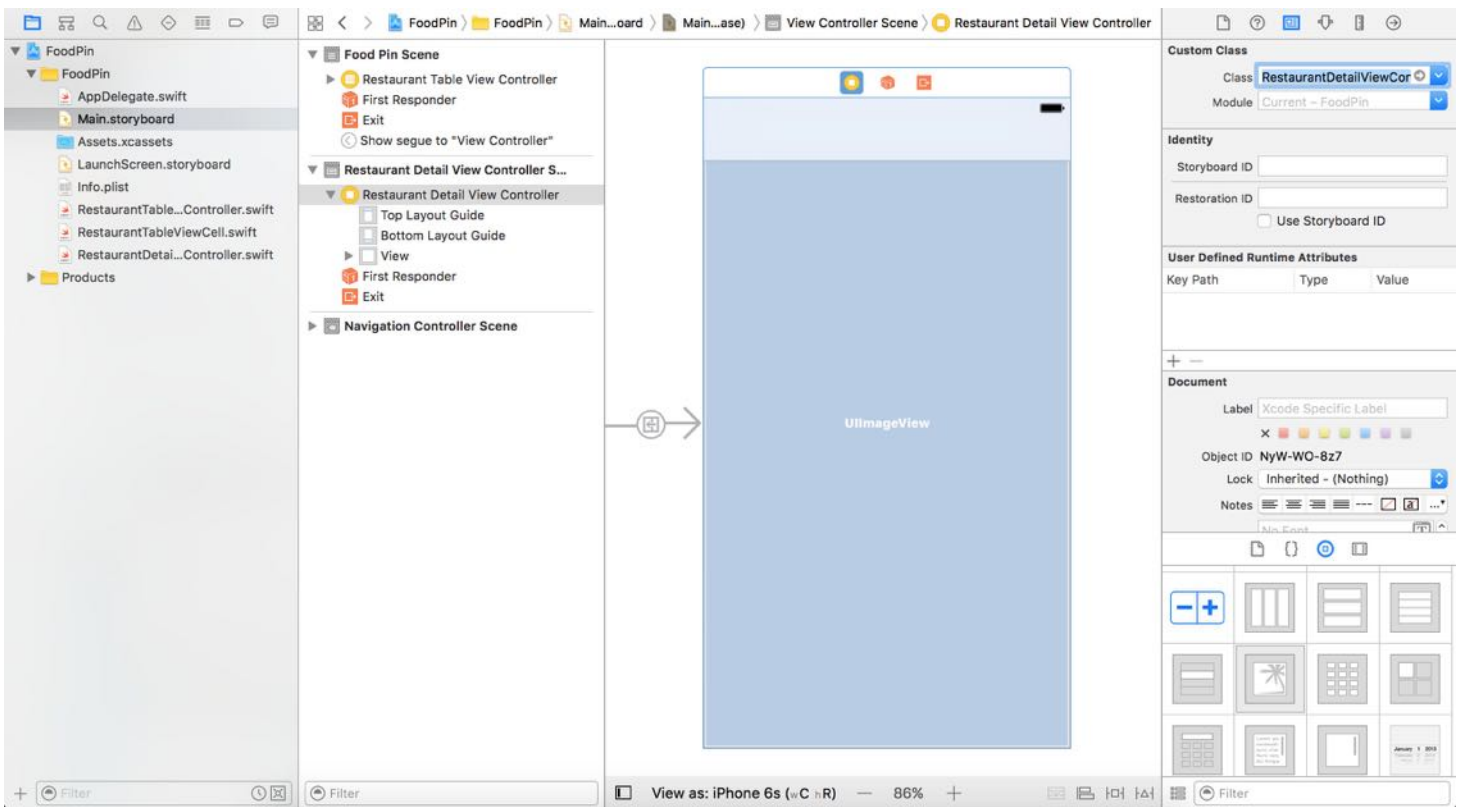


Figure 12-11. Assign a custom class for the detail view controller

Adding Variables to the Custom Class

There are a couple of things we have to add in the custom class:

- Create a variable named `restaurantImage` – when user selects a restaurant in the table view controller, there must be a way to pass the image name to the detail view. This variable will be used for data passing.
- Create an outlet called `restaurantImageView` for the image view – we need a reference to update the image view of the detail view controller, so we have to create an outlet.

Okay, add the following code to the `RestaurantDetailViewController` class:

```
@IBOutlet var restaurantImageView:UIImageView!
var restaurantImage = ""
```

Next, establish a connection between the `restaurantImageView` variable and the image view of the detail view controller. Go back to `Main.storyboard`. Right click the Restaurant Detail View

Controller object in the document outline. In the popover menu, connect the `restaurantImageView` outlet with the image view of the controller.

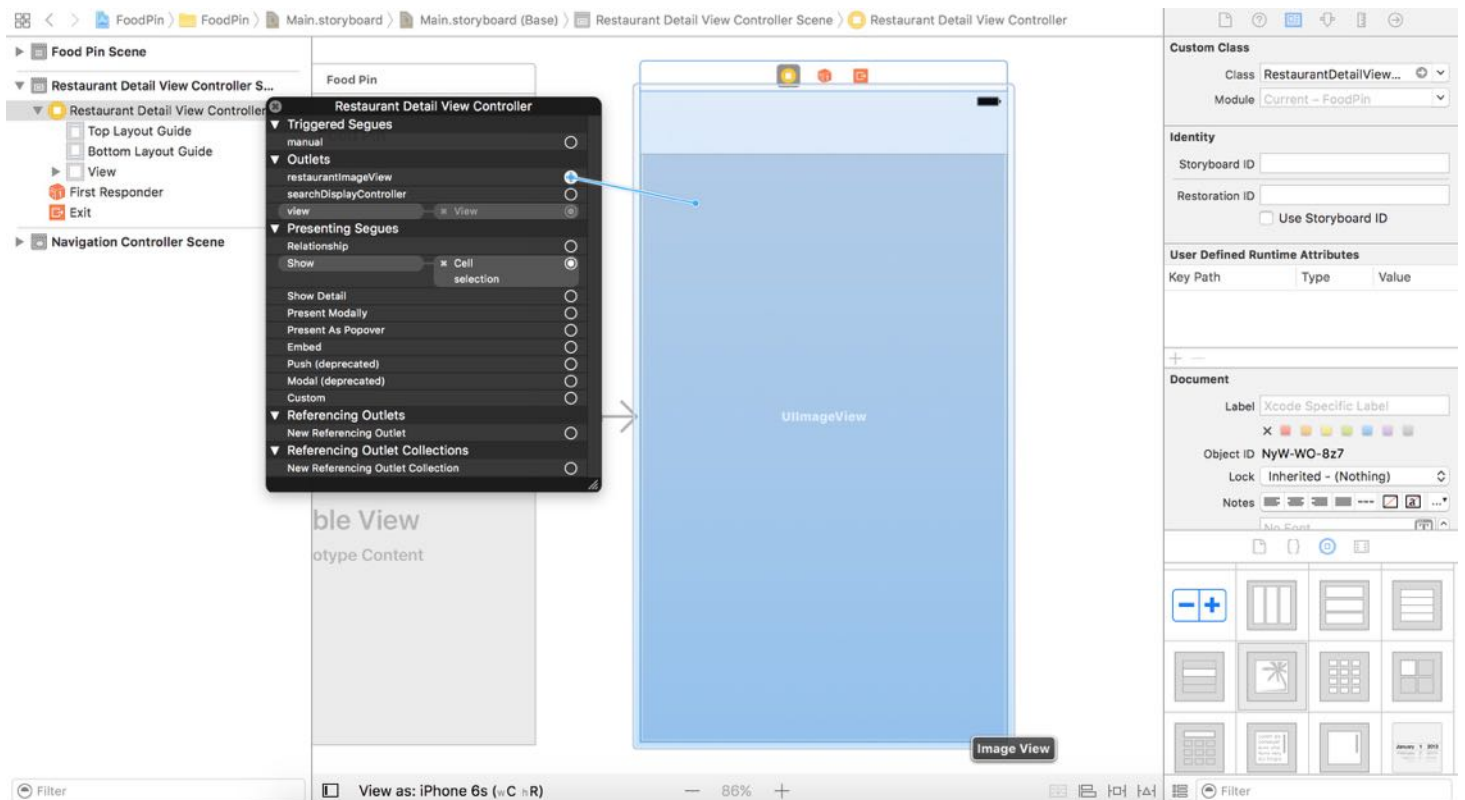


Figure 12-12. Establish the connection between the image view and the outlet

Now that you've linked up the outlet variable with the image view object in the storyboard, but there is still one thing left. You haven't provided the image yet. The image view should display the selected restaurant image. In the `viewDidLoad` method of the `RestaurantDetailViewController` class, add a line of code. Your method should look like this:

```
override func viewDidLoad() {
    super.viewDidLoad()

    // Do any additional setup after loading the view.
    restaurantImageView.image = UIImage(named: restaurantImage)
}
```

The `viewDidLoad` method is called when the view is loaded into memory. You can provide additional customization of the view in this method. In the above, we simply set the image view's image to the selected restaurant image.

Try to compile and run your app. Oops! The detail view is still blank after selecting a restaurant. We still miss one thing. We haven't passed the image name of the selected restaurant from the table view controller to the detail view controller. This is why the `restaurantImage` variable is not assigned with any value.

Passing Data Using Segues

This comes to the core part of this chapter about data passing with segues. A segue manages the transition between view controllers, and contains the view controllers involved in the transition. When a segue is triggered, before the visual transition occurs, the storyboard runtime notifies the source view controller (i.e. `RestaurantTableViewController`) by calling the `prepare(for:sender:)` method. The default implementation of the `prepare(for:sender:)` method does nothing. By overriding the method, you can pass any relevant data to the new controller, which is `RestaurantDetailViewController` in our project.

Segues can be triggered by multiple sources. As your storyboard becomes more complex, it is very likely that you'll have more than one segue between view controllers. Therefore, the best practice is to give each segue a unique identifier. This identifier is a string to distinguish one segue from another. To assign an identifier for a segue, select the segue in the storyboard editor, and then go to the Attributes inspector. Set the value of the identifier to `showRestaurantDetail`.

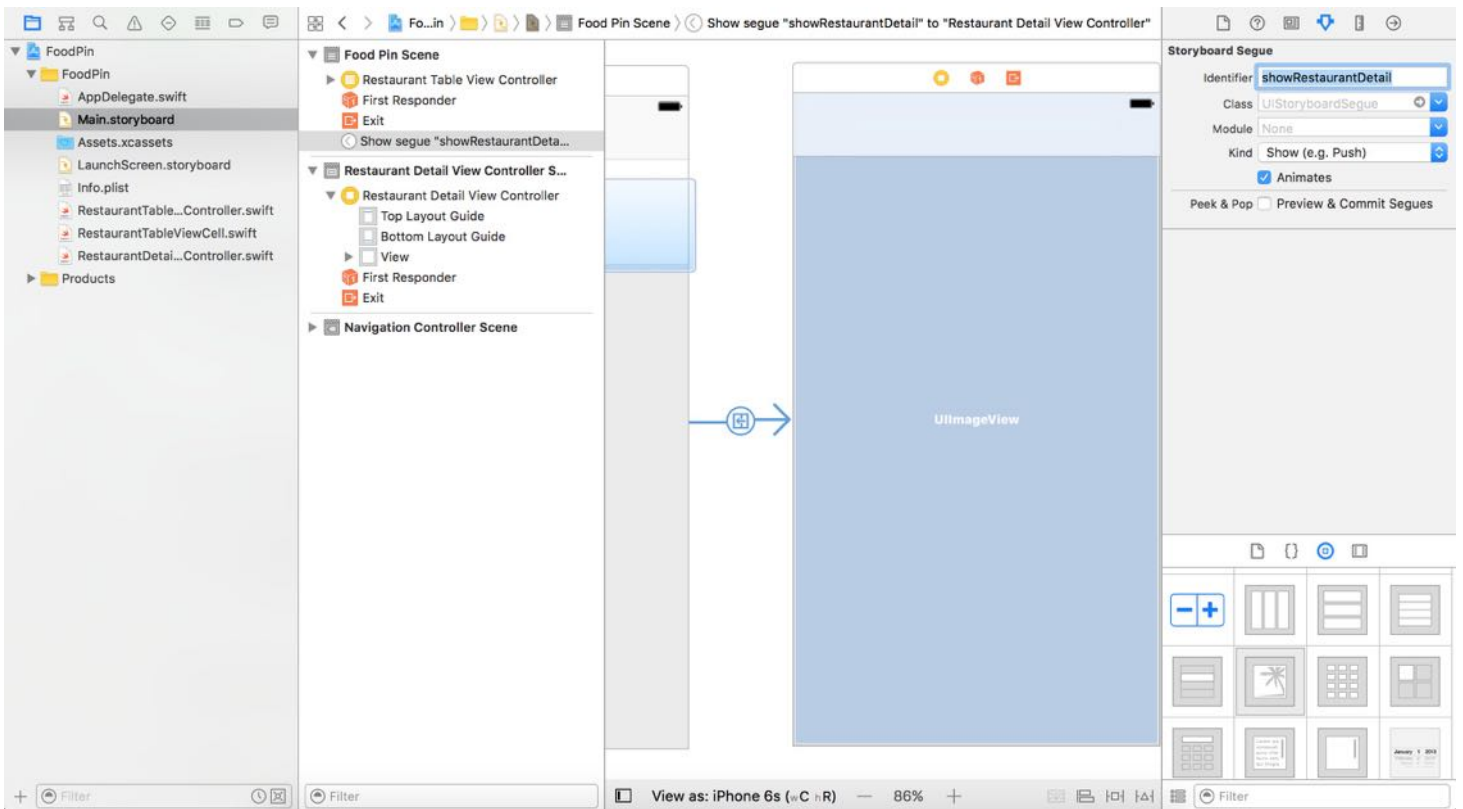


Figure 12-13. Adding a segue identifier

With the segue configured, insert the following code in the `RestaurantTableViewController` class to override the default implement of the `prepareForSegue` method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "showRestaurantDetail" {
        if let indexPath = tableView.indexPathForSelectedRow {
            let destinationController = segue.destination as!
RestaurantDetailViewController
            destinationController.restaurantImage =
restaurantImages[indexPath.row]
        }
    }
}
```

The first line of code is used to check the segue's identifier. The block of code is only executed for the `showRestaurantDetail` segue. In the code block, we first retrieve the selected row by accessing `tableView.indexPathForSelectedRow`. The `indexPath` object should contain the selected cell.

A segue object contains both the source and destination view controllers involved in the transition. You use `segue.destination` to retrieve the destination controller. In this case, the destination controller is the `RestaurantDetailViewController` object. This is why we have to downcast it by using the `as!` operator. Lastly, we pass the image name of the selected restaurant to the destination controller.

Ready to Test

Now, it's ready to test the app. Hit the Run button to compile and run the app. This time, your app should work as expected. Select a restaurant in the table view, the detail view should display the image of the selected item.

Your Exercise

Wouldn't it be better to show more restaurant information in the detail view? In this exercise, you're required to add a few labels for displaying the name, type and location of the selected restaurant. Your resulting screen should look figure 12-14. If you understand how data passing works, it shouldn't be difficult for you to make these changes.



Figure 12-14. The detail view with more restaurant information

I highly recommend you to complete this exercise before moving onto the next chapter. Not only it helps you better understand the concept of segue and data passing, it will revise your knowledge on stack views and auto layout.

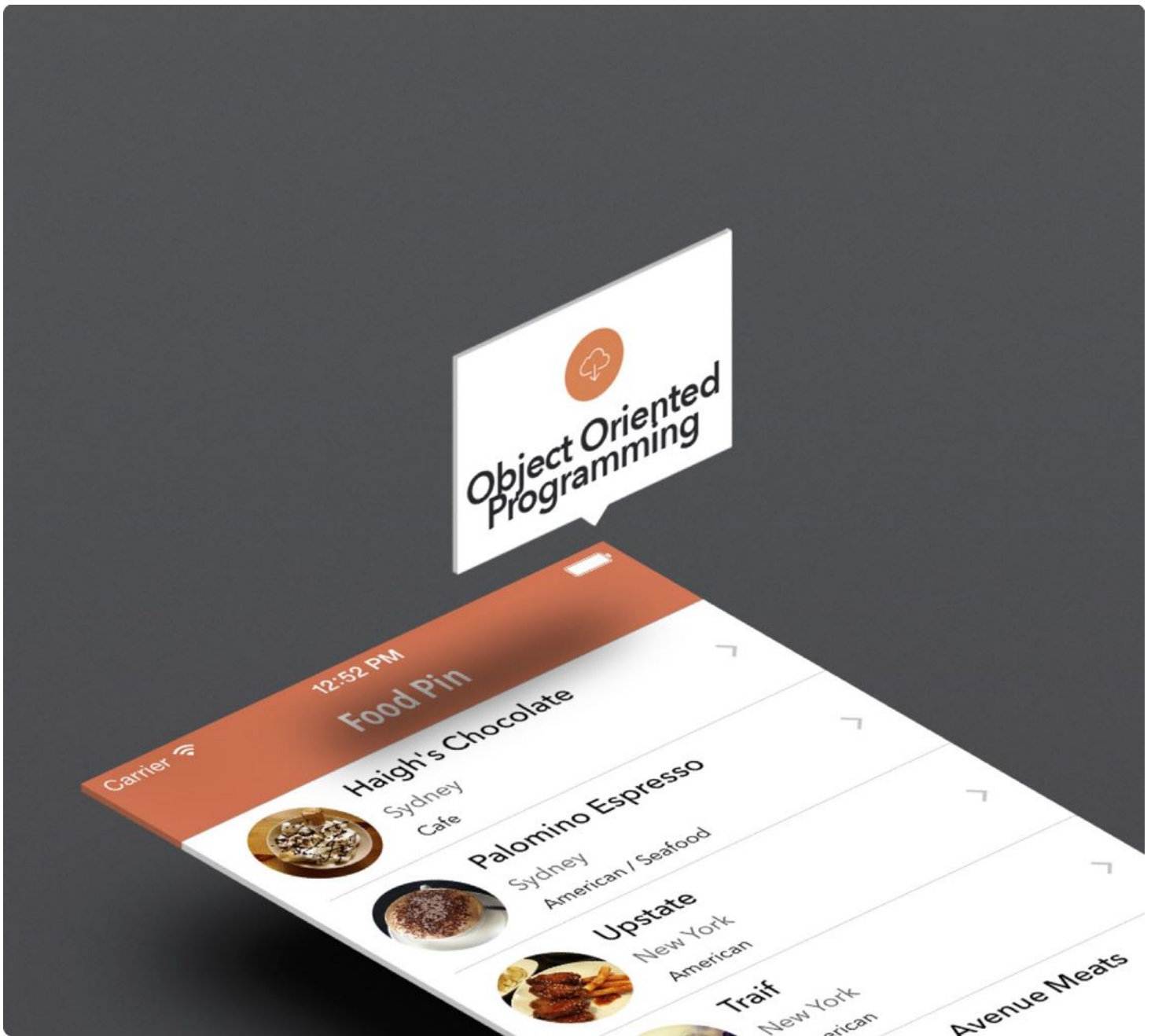
Summary

In this chapter, I have walked you through the basics of navigation controllers and segues. What we have built is really simple. We only pass the restaurant image from one view controller to another. But, by now you should know how to handle data passing between view controllers through segues.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinNavigationController.zip>.

Chapter 13

Introduction to Object Oriented Programming



Most good programmers do programming not because they expect to get paid or adulation by the public, but because it is fun to program.

If you read from the very beginning of the book and have worked through all the projects, you've gone pretty far. By now, you should be able to build an iPhone app with navigation controllers and table views using storyboards. We'll further enhance the FoodPin app, and improve the detail view. Did you manage to complete the previous exercise and develop your own detail view? This shouldn't be difficult to implement if you understand the materials and I intentionally left out that part for you as an exercise.

Anyway, we'll revisit it and show you how to improve the detail screen. But before that, I have to introduce you the basics of *Object Oriented Programming*. In the next chapter, we'll build on top of what we'll learn in this chapter and enhance the detail view screen.

Don't be scared by the term "Object Oriented Programming" or OOP in short. It's not a new kind of programming language, but a programming concept. While some programming books start out by introducing the OOP concept, I purposely left it out when I began writing the book. I want to keep things simple and show you how to create an app. I don't want to scare you away from building apps, just because of a technical term or concept. Having said that, I think it's time to discuss OOP. As you're still reading the book, I believe you're determined to learn iOS programming, and you want to take programming skills to the next level.

Okay, let's get started.

The Basic Theory of Object Oriented Programming

Like Objective-C, Swift is known as an Object Oriented Programming (OOP) language. OOP is a way of constructing software application composed of objects. In other words, the code written in an app in some ways deals with objects of some kinds. The `UIViewController`, `UIButton`, `UINavigationController`, and `UITableView` objects that you have used are some sample objects that come with the iOS SDK. Not only can you use the built-in objects, you have already created your own objects in the project, such as `DetailViewController` and `RestaurantTableViewCell`.

First, why OOP? One important reason is that we want to decompose a complex software into smaller pieces (or building block) which are easier to develop and manage. Here, the smaller

pieces are the objects. Each object has its own responsibility, and objects coordinate with each other in order to make the software work. That is the basic concept of OOP. Take the Hello World app, that we've built at the very beginning, as an example. The `UIViewController` object is responsible for controlling the view of the app, and its view object is used for holding the Hello World button. The `UIButton` (i.e. Hello World button) object implements a standard iOS button on the touch screen and listens to any touch events. On the other hand, the `UIAlertController` object displays an alert message to a user. After all, these objects work together to create the Hello World app.

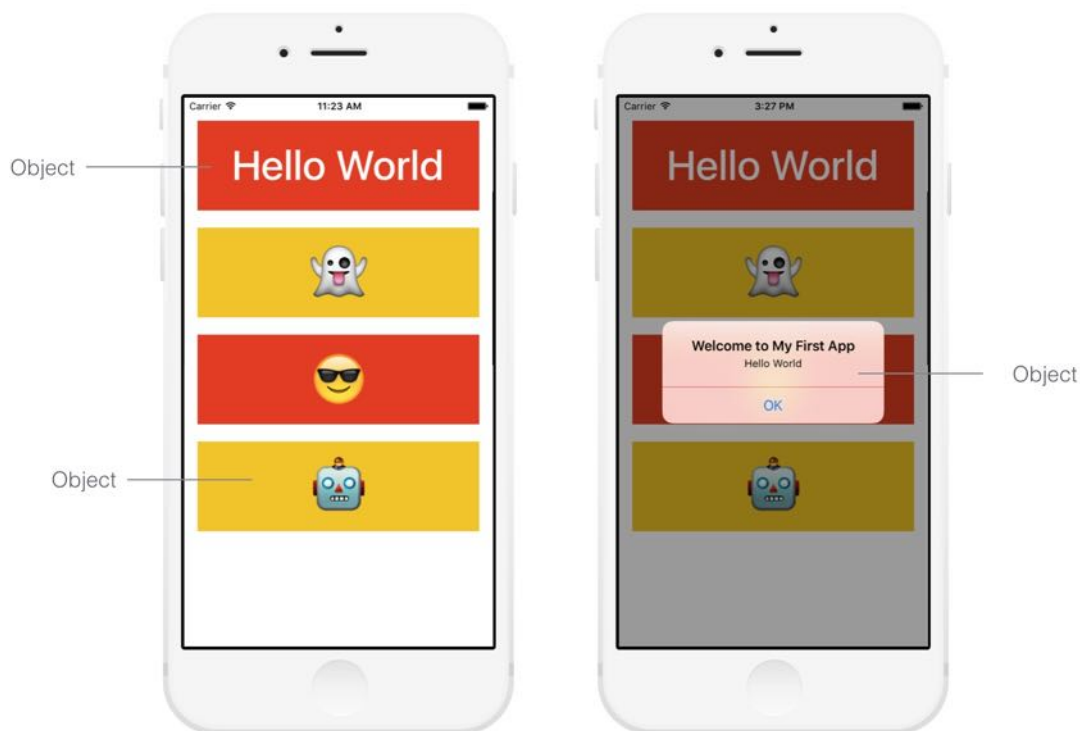


Figure 13-1. Sample Objects in Hello World App

In Object Oriented Programming, an object shares two characteristics: *properties* and *functionalities*. Let's consider a real world object – Car. A car has its own color, model, top speed, manufacturer, etc. These are the properties of a car. In terms of functionalities, a car should provide basic functions, such as accelerate, brake, steering, etc.

Software objects are conceptually similar to real-world objects. If we go back to the iOS world, let's take a look at the properties and functionalities of the `UIButton` object in the Hello World app:

- **Properties** – Background, size, color and font are examples of the `UIButton` 's properties
- **Functionalities** – When the button is tapped, it recognizes the tap event. The ability to detect a touch is one of the many functions of `UIButton` .

In earlier chapters, you always come across the term *method*. In Swift, we create methods to provide the functionalities of an object. Usually a method corresponds to a particular function of an object.

Classes, Objects and Instances

Other than method and object, you have come across terms like *instance* and *class* . These are also the common terms in OOP. Let me give you a brief explanation.

A class is a blueprint or prototype from which objects are created. Basically, a class consists of properties and methods. Let's say, we want to define a `Course` class. A `Course` class contains properties, such as *name*, *course code* and *total number of students*. This class represents the blueprint of a course. We can use it to create different courses like iOS Programming course with course code *IPC101*, Cooking course with course code *CC101*, etc. Here, the "iOS Programming course" and "Cooking course" are known as the *objects* of the `Course` class. We typically refer a single course as an *instance* of the `Course` class. For the sake of simplicity, the term *instance* and *object* are sometimes interchangeable.

A blueprint for a house design is like a class description. All the houses built from that blueprint are objects of that class. A given house is an instance.

Source: <http://stackoverflow.com/questions/3323330/difference-between-object-and-instance>

Revisit the FoodPin Project

So why do we cover OOP in this chapter? There is no better way to explain the concept than showing you an example. Take a look at the FoodPin project (<http://www.appcoda.com/resources/swift3/FoodPinNavigationController.zip>) again.

In the `RestaurantTableViewController` class, we created multiple arrays to store the names,

types, locations and images of the restaurants.

```
var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "Petite Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery", "Haigh's Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Avenue Meats", "Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional", "Barrafina", "Donostia", "Royal Oak", "CASK Pub and Kitchen"]

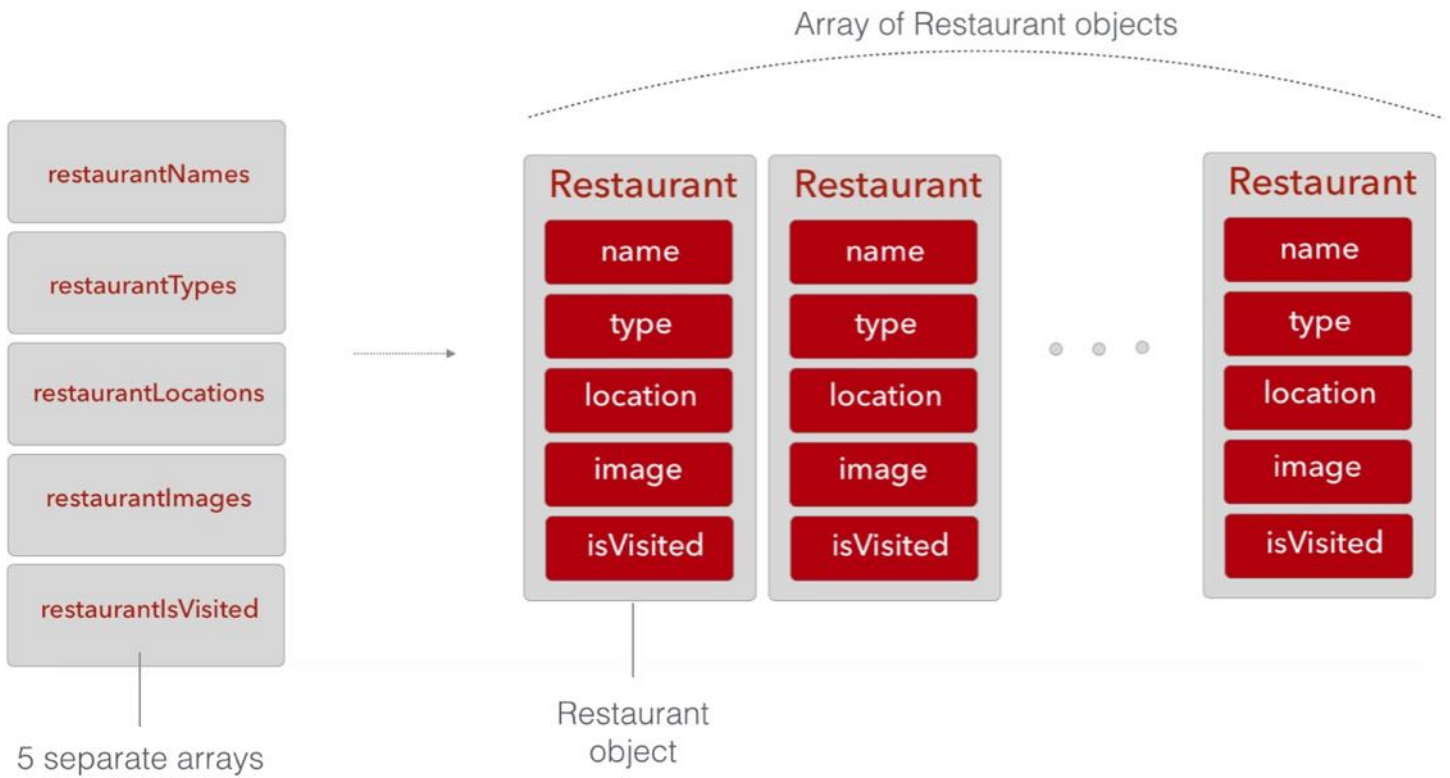
var restaurantImages = ["cafedeadend.jpg", "homei.jpg", "teakha.jpg", "cafeloisl.jpg", "petiteoyster.jpg", "forkeerrestaurant.jpg", "posatelier.jpg", "bourkestreetbakery.jpg", "haighschocolate.jpg", "palominoespresso.jpg", "upstate.jpg", "traif.jpg", "grahamavenuemeats.jpg", "wafflewolf.jpg", "fiveleaves.jpg", "cafelore.jpg", "confessional.jpg", "barrafina.jpg", "donostia.jpg", "royaloak.jpg", "caskpubkitchen.jpg"]

var restaurantLocations = ["Hong Kong", "Hong Kong", "Hong Kong", "Hong Kong", "Hong Kong", "Hong Kong", "Hong Kong", "Hong Kong", "Sydney", "Sydney", "Sydney", "New York", "New York", "New York", "New York", "New York", "New York", "New York", "London", "London", "London", "London"]

var restaurantTypes = ["Coffee & Tea Shop", "Cafe", "Tea House", "Austrian / Causal Drink", "French", "Bakery", "Bakery", "Chocolate", "Cafe", "American / Seafood", "American", "American", "Breakfast & Brunch", "Coffee & Tea", "Coffee & Tea", "Latin American", "Spanish", "Spanish", "Spanish", "British", "Thai"]

var restaurantIsVisited = Array(repeating: false, count: 21)
```

In Object Oriented Programming, these data can be characterized as the properties of a restaurant. Therefore, instead of storing these data in separate arrays, we can create a `Restaurant` class to store these data in an array of `Restaurant` objects.



Now let's tweak the FoodPin project, create the `Restaurant` class, and convert the data into `Restaurant` objects.

Creating a Restaurant Class

First, we'll start with the `Restaurant` class. In the project navigator, right click on the `FoodPin` folder and select "New File...". Choose the "Swift File" template under Source and click "Next". Name the file `Restaurant.swift` and save it in the project folder.

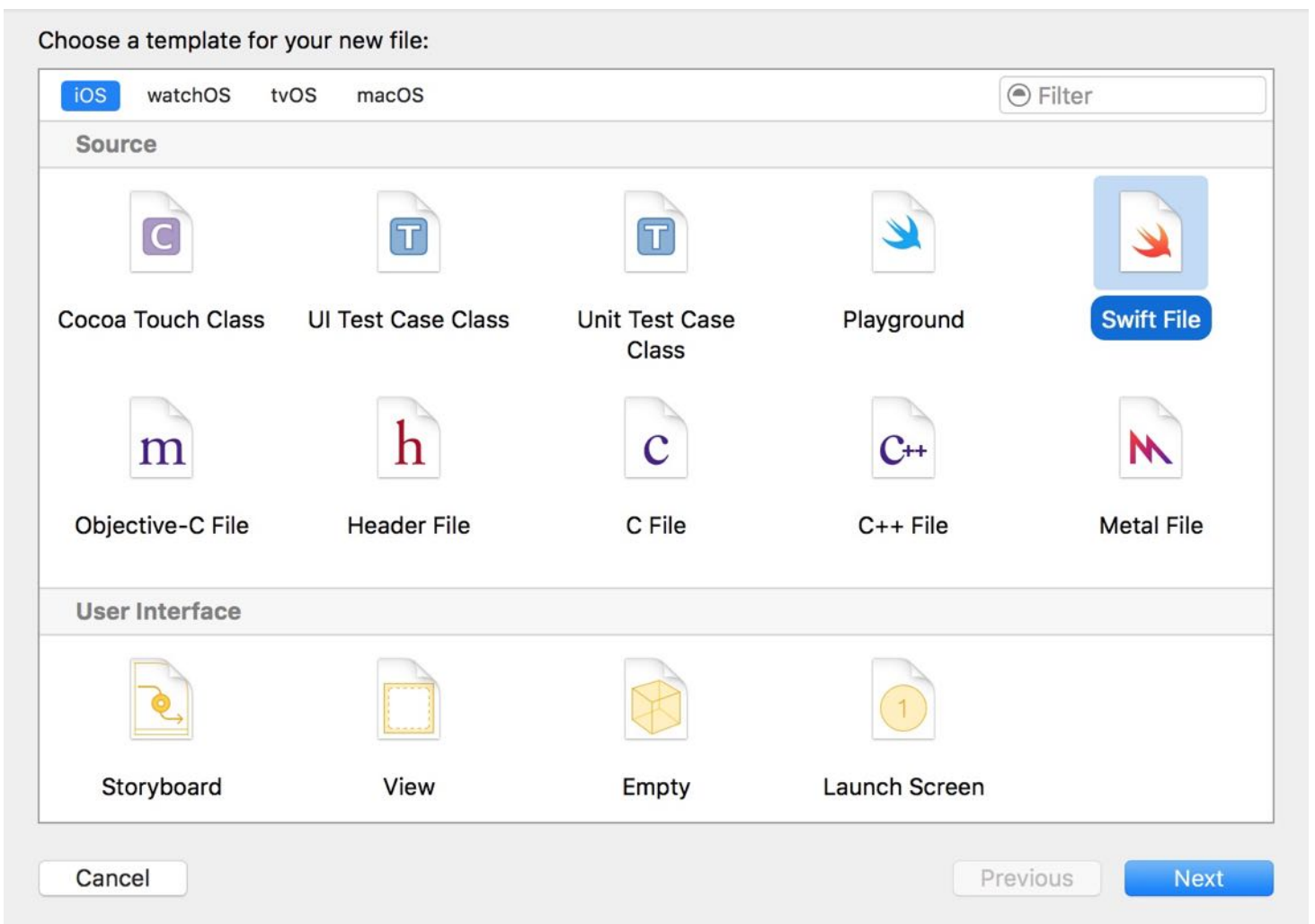


Figure 13-3. Creating a new class using the Swift File template

Once completed, define the `Restaurant` class in the `Restaurant.swift` file using the below code:

```
class Restaurant {
    var name = ""
    var type = ""
    var location = ""
    var image = ""
    var isVisited = false

    init(name: String, type: String, location: String, image: String,
isVisited: Bool) {
        self.name = name
        self.type = type
        self.location = location
    }
}
```



```
        self.image = image
        self.isVisited = isVisited
    }
}
```

You use the `class` keyword to define a class. The above code defines a `Restaurant` class with five properties including `name`, `type`, `location`, `image` and `isVisited`. Except for `isVisited`, which is a boolean, the rest of the properties are strings. All properties are initialized with a default value.

Use of self

In Swift, you use `self` to differentiate between property names and arguments in initializers. Because the arguments have the same name as the properties, we use `self` to refer to the property of the class.

```
class Restaurant {
    var name = ""
    var type = ""
    var location = ""
    var image = ""
    var isVisited = false

    init(name: String, type: String, location: String, image: String, isVisited: Bool) {
        self.name = name
        self.type = type
        self.location = location
        self.image = image
        self.isVisited = isVisited
    }
}
```

Figure 13-4. The use of self keyword

In Swift, all variables should be initialized with some values. Or you declare the variable as an optional. For instance, if you do not have the initial value for the location properties, you should declare the property like below. The question mark indicates the location variable may or may not contain a value.

```
var location:String?
```

Quick note: If you want to learn more about Optionals in Swift, refer to the appendix.

Initialization is the process of preparing an instance of a class. When you create an object, the initializer is called for setting an initial value for each stored property on that instance and performing any extra setup, before the instance is ready to use. You use the `init` keyword to define an initializer. In its simplest form, it looks like this:

```
init() {  
  
}
```

You can also customize an initializer to take input parameters, just like the one we have defined in the `Restaurant` class. Our initializer has five parameters. Each of them is given a name and explicitly specified with a type. In the initializer, it initializes the values of the property with the given values.

To create an instance of the `Restaurant` class, the syntax is like this:

```
Restaurant(name: "Thai Cafe", type: "Thai", location: "London", image:  
"thaicafe.jpg", isVisited: false)
```

Using the Array of Restaurant Objects

With a basic understanding of class and object initializations, let's go back to the FoodPin project and combine multiple arrays into an array of `Restaurant` objects. First, delete the restaurant-related arrays from the `RestaurantTableViewController` class:

```
var restaurantNames = ["Cafe Deadend", "Homei", "Teakha", "Cafe Loisl", "Petite  
Oyster", "For Kee Restaurant", "Po's Atelier", "Bourke Street Bakery", "Haigh's  
Chocolate", "Palomino Espresso", "Upstate", "Traif", "Graham Avenue Meats",  
"Waffle & Wolf", "Five Leaves", "Cafe Lore", "Confessional", "Barrafina",  
"Donostia", "Royal Oak", "CASK Pub and Kitchen"]  
  
var restaurantImages = ["cafedeadend.jpg", "homei.jpg", "teakha.jpg",  
"cafeloisl.jpg", "petiteoyster.jpg", "forkeerestaurant.jpg", "posatelier.jpg",  
"bourkestreetbakery.jpg", "haighschocolate.jpg", "palominoespresso.jpg",  
"upstate.jpg", "traif.jpg", "grahamavenuemeats.jpg", "wafflewolf.jpg",  
"fiveleaves.jpg", "cafelore.jpg", "confessional.jpg", "barrafina.jpg",  
"donostia.jpg", "royaloak.jpg", "caskpubkitchen.jpg"]  
  
var restaurantLocations = ["Hong Kong", "Hong Kong", "Hong Kong", "Hong Kong",  
"Hong Kong", "Hong Kong", "Hong Kong", "Sydney", "Sydney", "Sydney", "New  
York", "New York", "New York", "New York", "New York", "New York", "New York",  
"London", "London", "London", "London"]
```

```

var restaurantTypes = ["Coffee & Tea Shop", "Cafe", "Tea House", "Austrian /
Causal Drink", "French", "Bakery", "Bakery", "Chocolate", "Cafe", "American /
Seafood", "American", "American", "Breakfast & Brunch", "Coffee & Tea", "Coffee
& Tea", "Latin American", "Spanish", "Spanish", "Spanish", "British", "Thai"]

var restaurantIsVisited = Array(repeating: false, count: 21)

```

Instead of using the above array, replace them with a new array of `Restaurant` objects:

```

var restaurants:[Restaurant] = [
    Restaurant(name: "Cafe Deadend", type: "Coffee & Tea Shop", location: "Hong
Kong", image: "cafedeadend.jpg", isVisited: false),
    Restaurant(name: "Homei", type: "Cafe", location: "Hong Kong", image:
"homei.jpg", isVisited: false),
    Restaurant(name: "Teakha", type: "Tea House", location: "Hong Kong", image:
"teakha.jpg", isVisited: false),
    Restaurant(name: "Cafe loisl", type: "Austrian / Causal Drink", location:
"Hong Kong", image: "cafeloisl.jpg", isVisited: false),
    Restaurant(name: "Petite Oyster", type: "French", location: "Hong Kong",
image: "petiteoyster.jpg", isVisited: false),
    Restaurant(name: "For Kee Restaurant", type: "Bakery", location: "Hong
Kong", image: "forkeerestaurant.jpg", isVisited: false),
    Restaurant(name: "Po's Atelier", type: "Bakery", location: "Hong Kong",
image: "posatelier.jpg", isVisited: false),
    Restaurant(name: "Bourke Street Bakery", type: "Chocolate", location:
"Sydney", image: "bourkestreetbakery.jpg", isVisited: false),
    Restaurant(name: "Haigh's Chocolate", type: "Cafe", location: "Sydney",
image: "haighschocolate.jpg", isVisited: false),
    Restaurant(name: "Palomino Espresso", type: "American / Seafood", location:
"Sydney", image: "palominoespresso.jpg", isVisited: false),
    Restaurant(name: "Upstate", type: "American", location: "New York", image:
"upstate.jpg", isVisited: false),
    Restaurant(name: "Traif", type: "American", location: "New York", image:
"traif.jpg", isVisited: false),
    Restaurant(name: "Graham Avenue Meats", type: "Breakfast & Brunch",
location: "New York", image: "grahamavenuemeats.jpg", isVisited: false),
    Restaurant(name: "Waffle & Wolf", type: "Coffee & Tea", location: "New
York", image: "wafflewolf.jpg", isVisited: false),
    Restaurant(name: "Five Leaves", type: "Coffee & Tea", location: "New York",
image: "fiveleaves.jpg", isVisited: false),
    Restaurant(name: "Cafe Lore", type: "Latin American", location: "New York",
image: "cafelore.jpg", isVisited: false),
    Restaurant(name: "Confessional", type: "Spanish", location: "New York",
image: "confessional.jpg", isVisited: false),
    Restaurant(name: "Barrafina", type: "Spanish", location: "London", image:
"barrafina.jpg", isVisited: false),
    Restaurant(name: "Donostia", type: "Spanish", location: "London", image:

```

```

"donostia.jpg", isVisited: false),
    Restaurant(name: "Royal Oak", type: "British", location: "London", image:
"royaloak.jpg", isVisited: false),
    Restaurant(name: "CASK Pub and Kitchen", type: "Thai", location: "London",
image: "caskpubkitchen.jpg", isVisited: false)
]

```

Once you replaced the original arrays with the `restaurants` array, you'll end up with a few errors in Xcode because some of the code still refer to the old arrays which have been removed. We have to modify the code to use the new `restaurants` array. First, update the `tableView(_:numberOfRowsInSection:)` method:

```

override func tableView(_ tableView: UITableView, numberOfRowsInSection: Int) -> Int {
    return restaurants.count
}

```

Instead of using the `restaurantNames` array, we get the count from the new `restaurants` array.

Secondly, update the `tableView(_:cellForRowAtIndexPath:)` method:

```

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

    let cellIdentifier = "Cell"
    let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier,
for: indexPath) as! RestaurantTableViewCell

    // Configure the cell...
    cell.nameLabel.text = restaurants[indexPath.row].name
    cell.thumbnailImageView.image = UIImage(named:
restaurants[indexPath.row].image)
    cell.locationLabel.text = restaurants[indexPath.row].location
    cell.typeLabel.text = restaurants[indexPath.row].type

    cell.accessoryType = restaurants[indexPath.row].isVisited ? .checkmark :
.none

    return cell
}

```

All the restaurant information are now retrieved from the `restaurants` array. We first locate the restaurant object from the `restaurants` array. Then we use the dot-syntax to access the property value.

Other than that, replace the `tableView(_:commit:forRowAt:)` method with the following code:

```
override func tableView(_ tableView: UITableView, commit editingStyle:
UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {

    if editingStyle == .delete {
        // Delete the row from the data source
        restaurants.remove(at: indexPath.row)
    }

    tableView.deleteRows(at: [indexPath], with: .fade)
}
```

And, update the `tableView(_:editActionsForRowAt:)` method with the code below:

```
override func tableView(_ tableView: UITableView, editActionsForRowAt
indexPath: IndexPath) -> [UITableViewRowAction]? {

    // Social Sharing Button
    let shareAction = UITableViewRowAction(style:
UITableViewRowActionStyle.default, title: "Share", handler: { (action,
indexPath) -> Void in

        let defaultText = "Just checking in at " +
self.restaurants[indexPath.row].name

        if let imageToShare = UIImage(named:
self.restaurants[indexPath.row].image) {
            let activityController = UIActivityViewController(activityItems:
[defaultText, imageToShare], applicationActivities: nil)
            self.present(activityController, animated: true, completion: nil)
        }
    })

    // Delete button
    let deleteAction = UITableViewRowAction(style:
UITableViewRowActionStyle.default, title: "Delete", handler: { (action,
indexPath) -> Void in

        // Delete the row from the data source
        self.restaurants.remove(at: indexPath.row)

        self.tableView.deleteRows(at: [indexPath], with: .fade)
    })

    shareAction.backgroundColor = UIColor(red: 48.0/255.0, green: 173.0/255.0,
blue: 99.0/255.0, alpha: 1.0)
    deleteAction.backgroundColor = UIColor(red: 202.0/255.0, green:
```

```
202.0/255.0, blue: 203.0/255.0, alpha: 1.0)

    return [deleteAction, shareAction]
}
```

For the `RestaurantDetailViewController` class, instead of declaring variables for each restaurant property, we now define a `Restaurant` object to save the restaurant information. Replace the following variables:

```
var restaurantImage = ""
var restaurantName = ""
var restaurantType = ""
var restaurantLocation = ""
```

with:

```
var restaurant:Restaurant!
```

Quick note: As mentioned before, you have to give a default value when declaring a variable. If not, you either use a question mark (?) or an exclamation mark (!) in the declaration. In general, it is preferable to define the variable as an optional by using a question mark (?). But if you are quite sure that the variable will be assigned with a valid value, you can declare it as an implicitly unwrapped optional through an exclamation mark (!).

And, update the `viewDidLoad` method with the following code:

```
override fun viewDidLoad() {
    super.viewDidLoad()

    // Do any additional setup after loading the view.
    restaurantImageView.image = UIImage(named: restaurant.image)
    restaurantNameLabel.text = restaurant.name
    restaurantTypeLabel.text = restaurant.type
    restaurantLocationLabel.text = restaurant.location
}
```

Finally, change the `prepare(for:sender:)` method to the code below. We now pass the `Restaurant` object directly to the `RestaurantDetailViewController` class.

```
override fun prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "showRestaurantDetail" {
        if let indexPath = tableView.indexPathForSelectedRow {
            let destinationController = segue.destinationViewController as!
```

```
RestaurantDetailViewController
    destinationController.restaurant = restaurants[indexPath.row]
    }
}
}
```

You can now run your app. The look & feel of the app is exactly the same as before. However, we have refactored the code to use the new `Restaurant` class. By combining multiple arrays into one, the code is now more readable.

Summary

Congratulations if you made it this far. I hope you're not bored by the chapter. What I have covered is the basics of Object Oriented Programming. There are a lot more about the OOP concepts, such as polymorphism. However, we do not have time to discuss in-depth in this book. If you want to become a professional iOS developer, check out the references to learn more. It'll take you a lot of practices to truly pick up OOP. Anyhow, if you manage to finish this chapter, this is a good start.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinOOP.zip>. In the next chapter, based on what we've learned, you'll continue to tweak the detail view screen of the FoodPin app. It's going to be fun!

Further References

Swift Programming Language - Classes and Structures

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ClassesAndStructures.html

Swift Programming Language - Initialization

https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html

Swift Programming Language - Inheritance

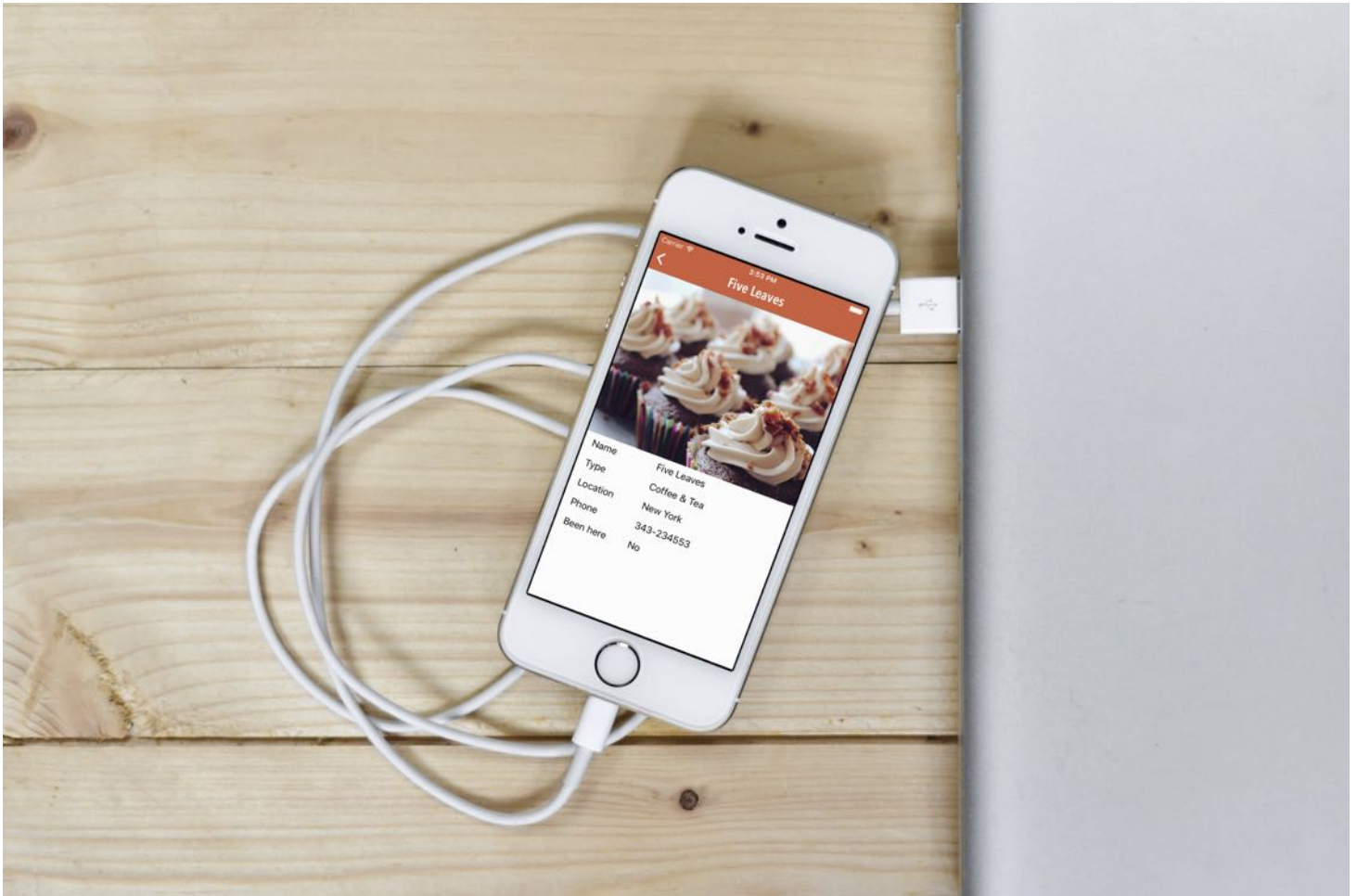
https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Inheritance.html

Object Oriented Programming from MIT Open Courseware

<http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spring-2011/unit-1-software-engineering/object-oriented-programming/>

Chapter 14

Detail View Enhancement and Navigation Bar Customization



To create something exceptional, your mindset must be relentlessly focused on the smallest detail.

- Giorgio Armani

The detail view is a bit primitive. Wouldn't it be great to improve the detail view to the one shown above? In this chapter we'll focus on two areas:

- Improve the detail view of the FoodPin app.

- Customize the appearance of the navigation bar.

Okay, let's see how we can tweak the detail view first.

What we're going to do is to list the restaurant information including name, type and location in a table view. The restaurant image will be used as the header of the table.

To begin, first download the FoodPin project from <http://www.appcoda.com/resources/swift3/FoodPinDetailViewStarter.zip>. You can use your own project if you've followed every step covered in the earlier chapters. However, you have to delete everything in the detail view controller because we're going to redesign it from the ground up.

Open the project and go to `Main.storyboard`. Select a table view object (*note: it's a table view instead of a table view controller*) from the Object library and drag it to the detail view controller. Resize it to fit the whole view, and the spacing constraints for each side of the table view. Your screen should look like that in figure 14-1.

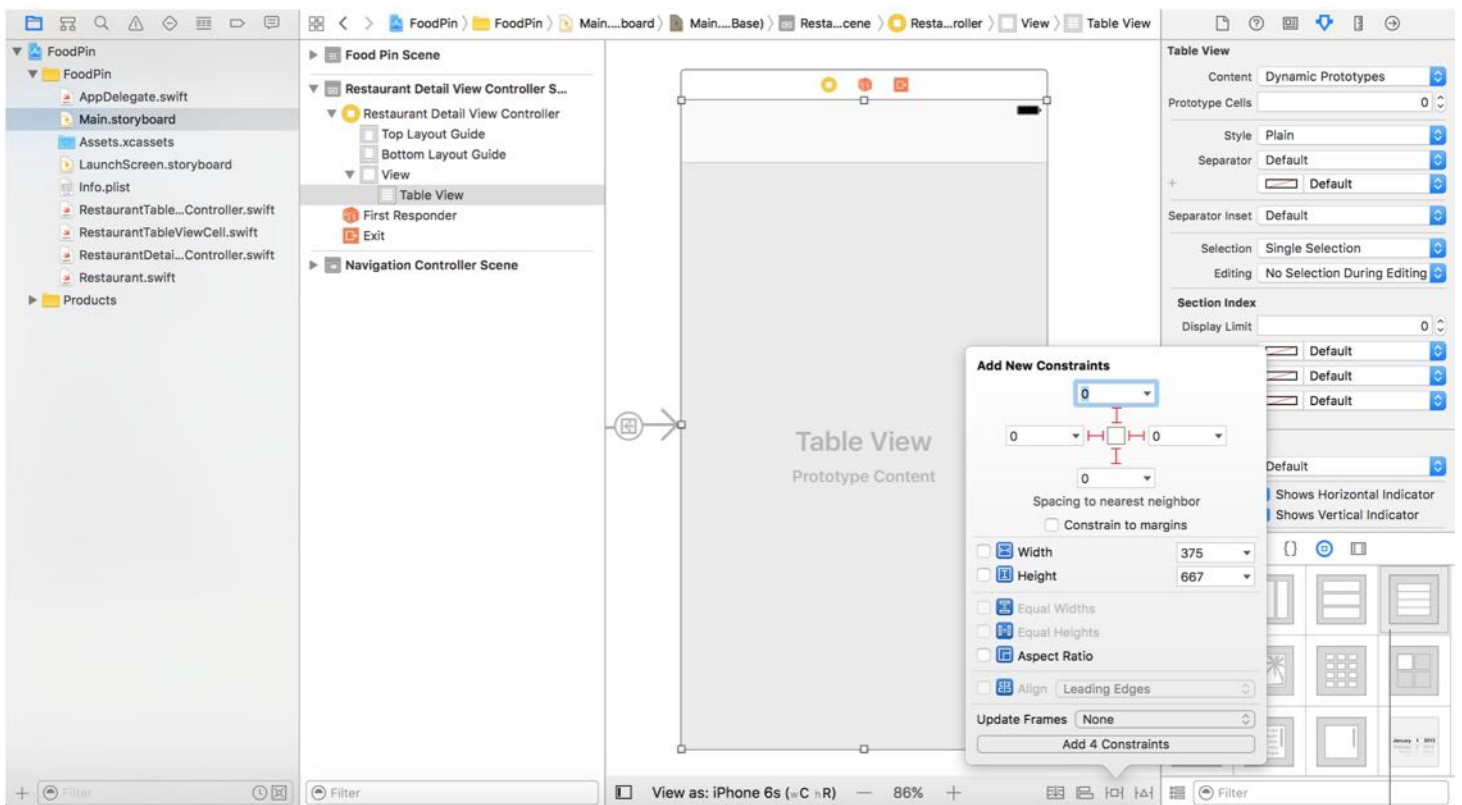


Figure 14-1. Adding a table view to the detail view controller

Select the table view and set the prototype cells to `1` in the Attributes inspector. This adds a prototype cell in the table view. Select the table view cell and set the identifier to `cell1` in the Attribute inspector. Also set the row height to `36` points.

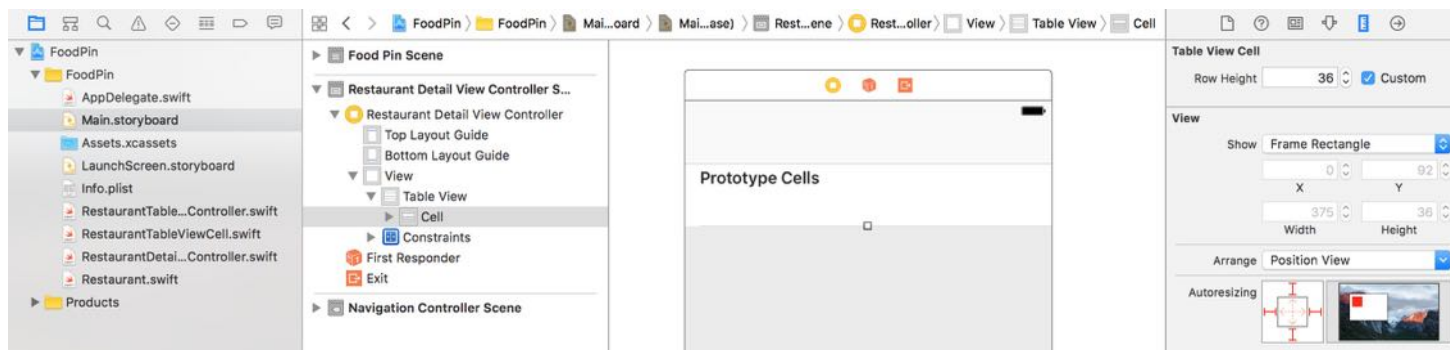


Figure 14-2. Adding a prototype cell in the table view

Next, we will add an image view to the table view's header. In the document outline view, drag the image view to table view and place it right above the table view cell. Change the height of the image view to 300 points in the Size inspector. Once done, the design of your detail view should look like that in figure 14-3.

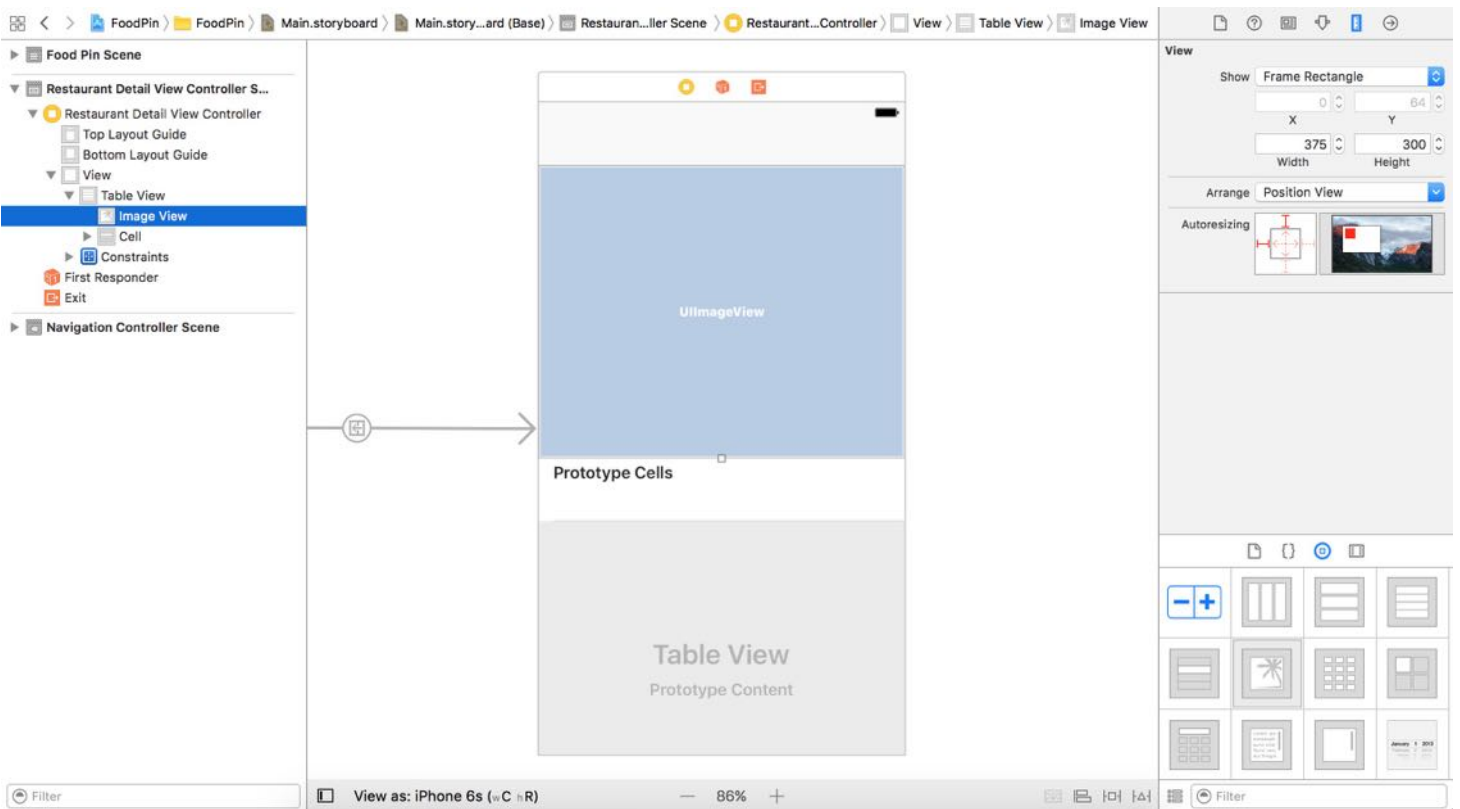


Figure 14-3. Image view added to the table view's header

Don't forget to establish a connection between the image view and the `restaurantImageView` outlet. Right-click `Restaurant Detail View Controller` in the document outline, and connect the outlet with the image view.

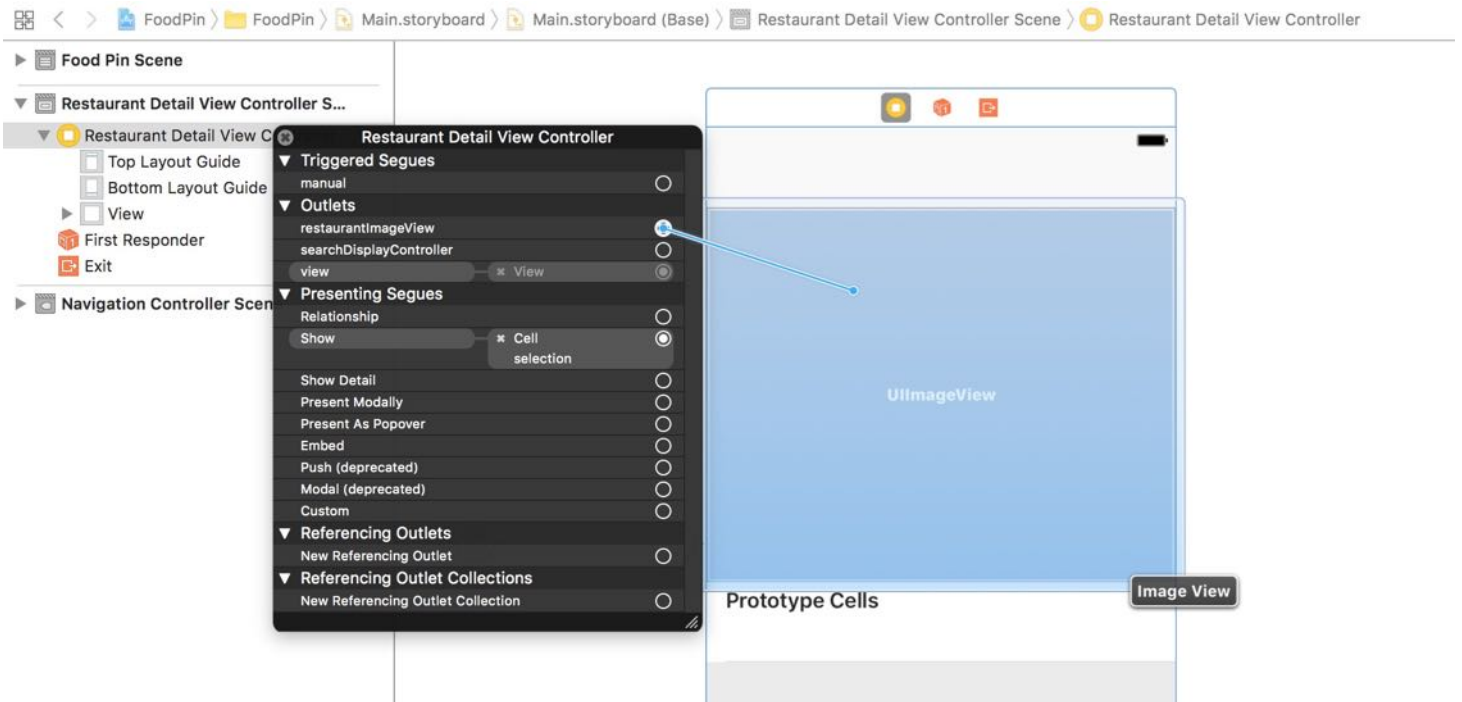


Figure 14-4. Establish a connection between the image view and the `restaurantImageView` outlet

If you can't wait to test the app, run the project and have a quick look. The image is now displayed in the table view's header (right above the rest of the table rows).

Image View Scaling

Before continuing the development of the detail view, I would like to sidetrack a little bit and talk about the scaling mode of `UIImageView`. If you select the image view, you should find the *mode* option in the Attributes inspector. The `UIImageView` class is a subclass of `UIView`, which provides a content mode property to specify how a view adjusts its content. Here are the three common scaling modes you usually use:

- Scale to Fill (default)
- Aspect Fit
- Aspect Fill

By default, the `UIImageView` object is set to *Scale To Fill*. In this mode, the image view scales its image to fit the view's size. The aspect ratio of the image may be changed due to the scaling.

If you run the FoodPin app and select *For Kee Restaurant*, you may aware that the image doesn't keep the original aspect ratio.

If you change the mode to *Aspect Fit*, the aspect ratio of the image will be maintained. However, it may leave some extra spaces on both sides of the image.

The *Aspect Fill* mode is the best fit for the FoodPin app. In this mode, the image view scales the image to fill the size of the view. The aspect ratio is maintained, though some parts of the image may not be displayed. Please make sure you enable "Clip to Bounds" in the drawing section when using this mode. Otherwise, the image may extend to the table view area. Figure 14-5 illustrates the results of the three content modes.

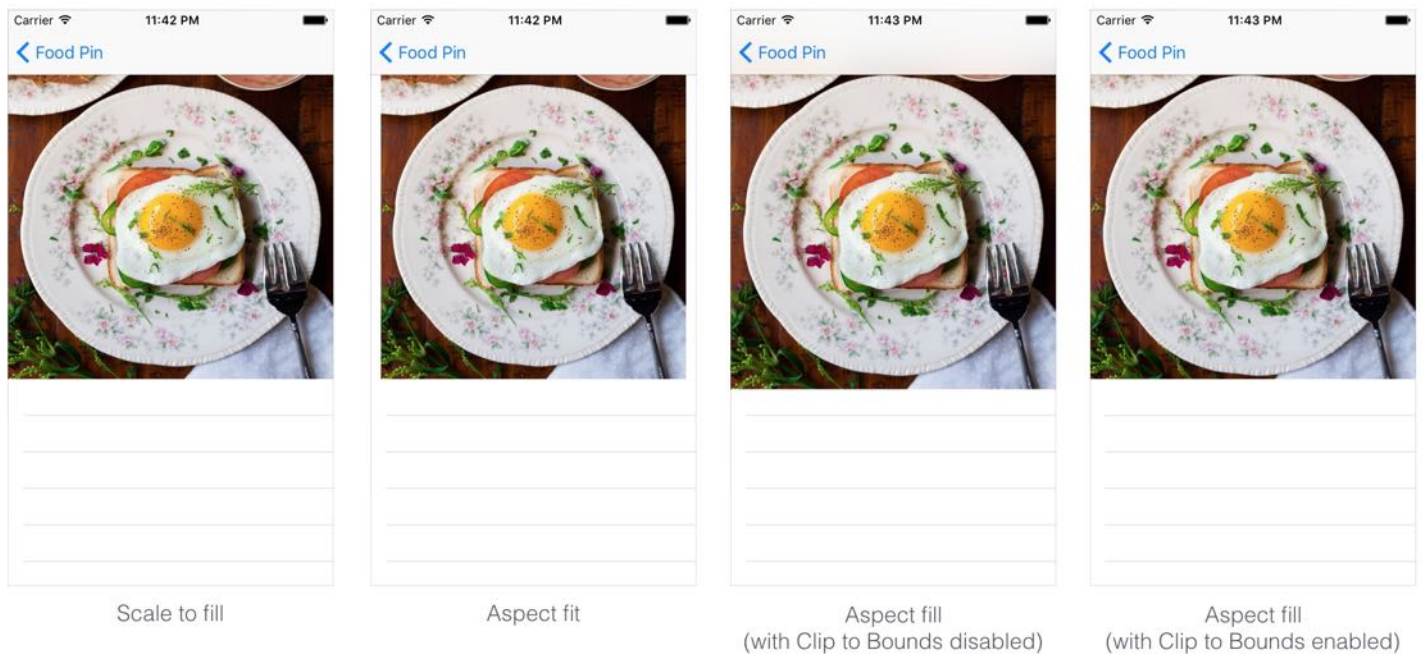


Figure 14-5. Image View Scaling Mode

Customizing the Prototype Cell

If you read the book from the very beginning, you should know how to customize the prototype cell. We'll add two labels to the cell. One is for the field name and the other one is for the field value.

So drag a Label object from the Object library and place it in the prototype cell. Name the label *Field* and set the font style to *Medium* (or whatever font style you want). Drag another label to the cell and set the name to *Value*. Again, we use stack view to layout these two buttons. Select both labels and click the *Stack* button in the layout bar to embed them in a horizontal stack view.

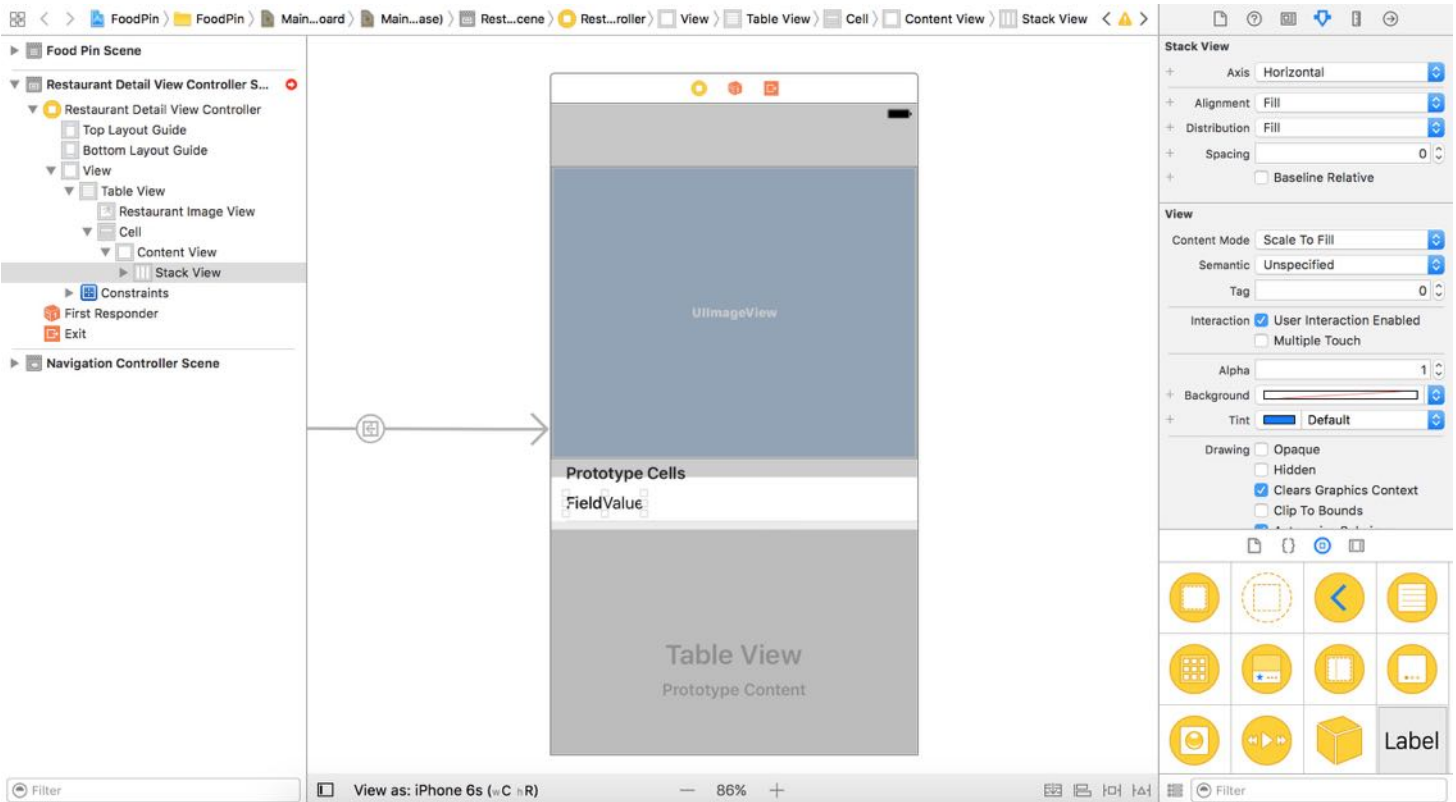


Figure 14-6. Customizing the prototype cell

Next, define the auto layout constraints for the stack view. The resulting layout of the cell will look like figure 14-7.



Figure 14-7. Cell layout

To achieve the layout, you have to define a few constraints for the stack view and the labels:

- Add a constraint such that the stack view is vertically centered in the cell.
- Add spacing constraints for the left and right sides of the stack view. For the spacing constraint of the trailing side, the constant should be set to zero.
- Add another constraint for the labels such that the width of the *Field* label is 50% of the *Value* label.

You can follow the procedures illustrated in figure 14-8 and figure 14-9 to define the first two constraints.

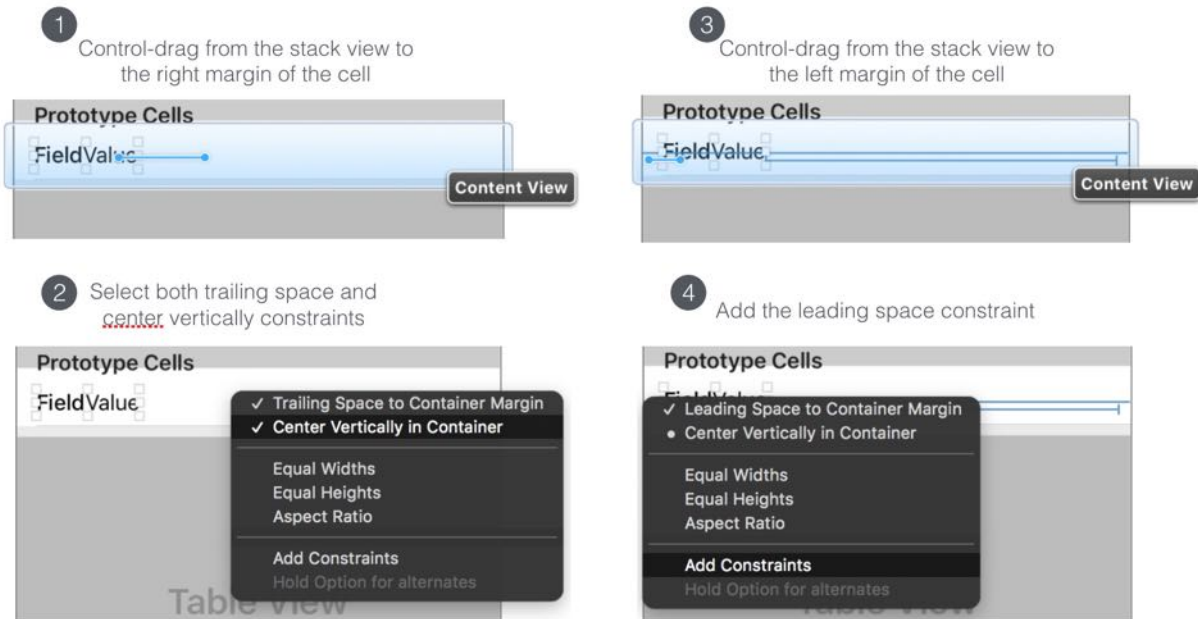


Figure 14-8. Center the stack view vertically and add spacing constraints

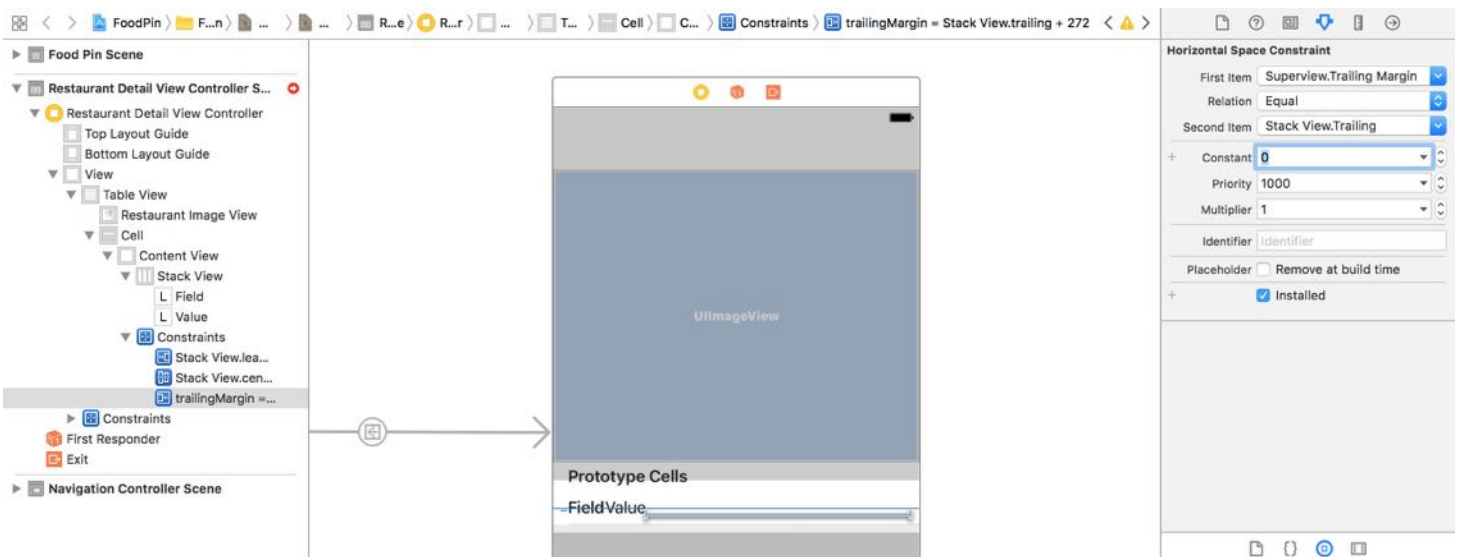


Figure 14-9. Change the constant of the trailing spacing constraint to zero

The procedures should be very similar to you as we have covered them in earlier chapters. After defining the constraints, the cell layout should look like that in figure 14-10.

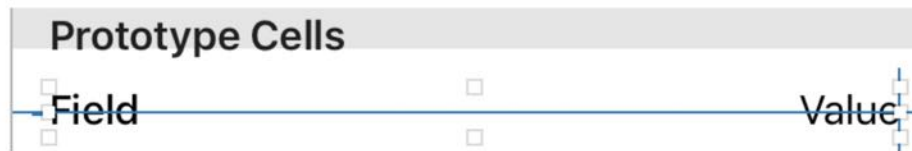


Figure 14-10. Change the constant of the trailing spacing constraint to zero

Content Hugging Priority

As a sidenote, let's me introduce you a layout property called *Content Hugging Priority*. So far we only define the layout constraints of the stack view, we let the stack view handle the layout of the labels. In order to fulfill the layout requirements, the stack view either needs to expand the *Field* label or the *Value* label.

The question is: why did the stack view choose to expand the *Field* label instead of the *Value* label?

If you select the *Field label* and go to the Size inspector, you will find the

content hugging priority of the label in both horizontal and vertical direction. By default, it is set to 251. The stack view relies on this priority to determine if it should expand the *Field* label or the *Value* label. A view with higher content hugging priority resists being made larger and remains its intrinsic size. Because both labels have the same priority, the stack view simply expands the *Field* label.

Now if you change the horizontal content hugging priority of the *Field* label from 251 to 261. Once the priority is increased, the *Field* label is resized to its intrinsic size, while the *Value* label is expanded.

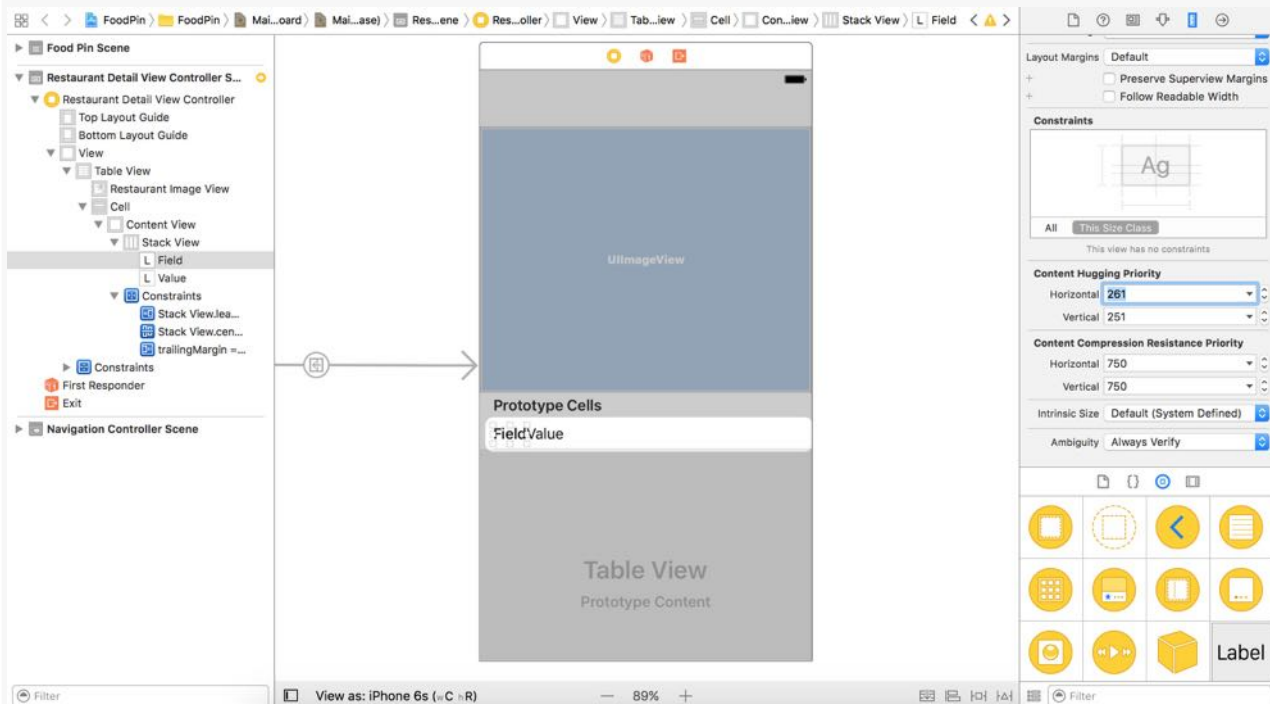


Figure 14-11. The cell layout changes as the content hugging priority is updated

Now that you've defined the first two layout constraints, how can you define a constraint such that the width of the *Field* label is 50% of the *Value* label?



Figure 14-12. Expected layout for both labels

To do so, control-drag horizontally from the *Field* label to the *Value* label. In the shortcut menu, select *Equal Widths* . By setting this option, both labels have the same width.

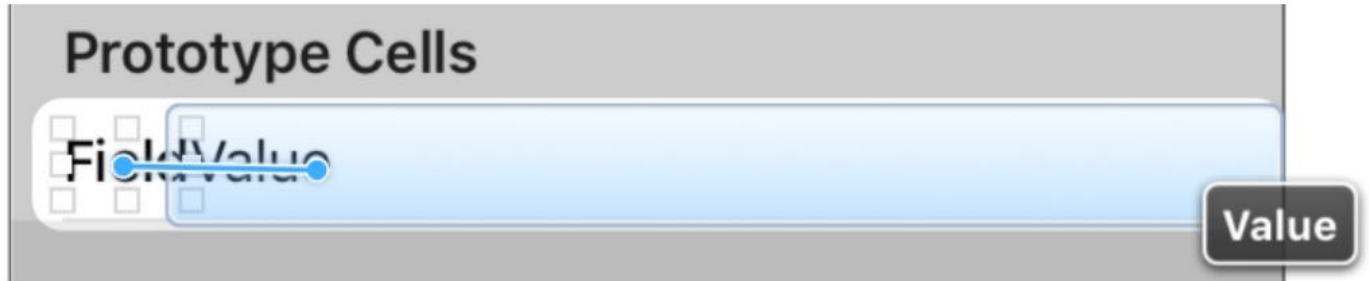


Figure 14-13. Control-drag horizontally from the *Field* label to the *Value* label

Now select the "Field.width = Value.width" constraint and go to the Size inspector. Change the value of *Multiplier* from 1 to 0.5 . The width of the *Field* label is changed so that it is 50% of that of the *Value* label.

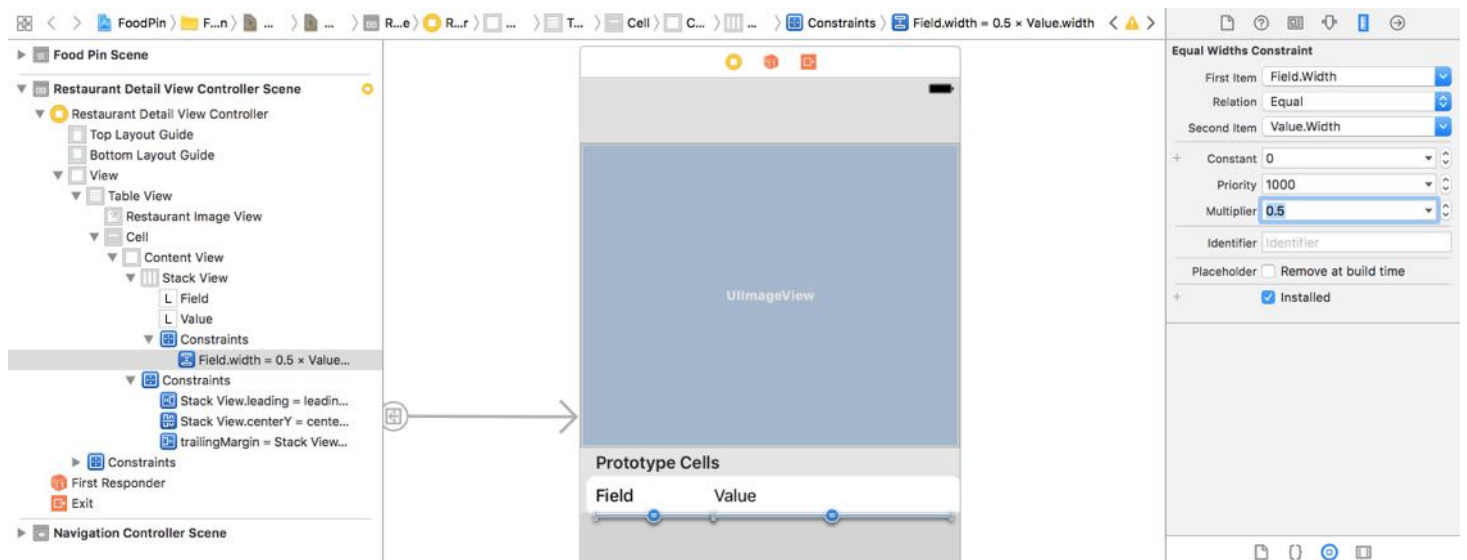


Figure 14-14. Change the multiplier to 0.5

Cool! You've completed the design of the prototype cell.

In order to use it, as you know, we have to create a custom class to pair with it. Right click the FoodPin folder in the project navigator, and then click "New File...". Choose the "Cocoa Touch Class" template and proceed. Name the class `RestaurantDetailTableViewCell` and set it as a subclass of `UITableViewCell`. Click Next and save the file.

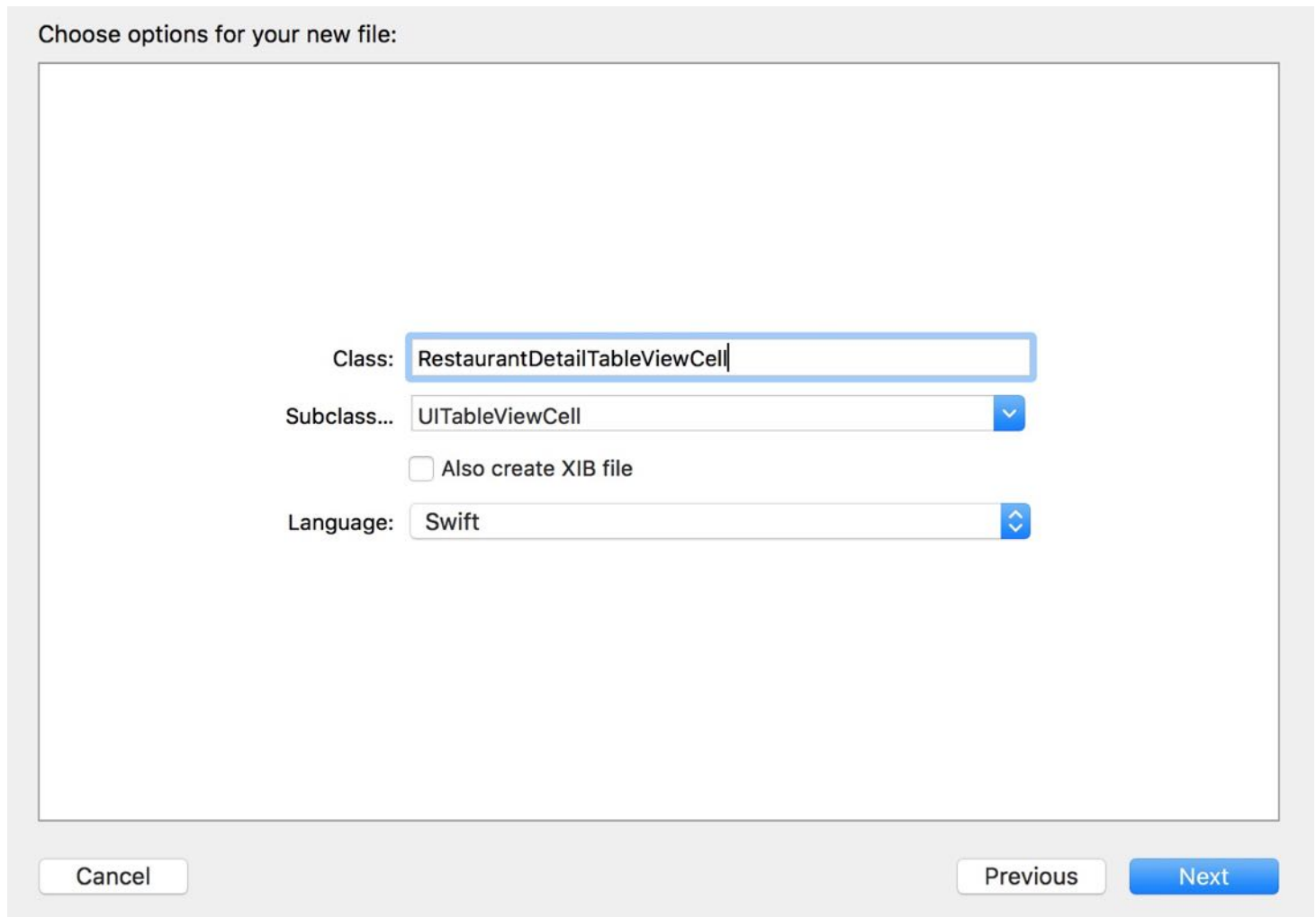


Figure 14-15. Creating a new class for the prototype cell

Open the `RestaurantDetailTableViewCell.swift` file and add two outlet variables for both *Field* and *Value* labels:

```
@IBOutlet var fieldLabel:UILabel!  
@IBOutlet var valueLabel:UILabel!
```

The rest of the procedures should be very familiar to you. Go back to the Interface Builder and

select the prototype cell. In the Identity inspector, set the custom class to `RestaurantDetailTableViewCell`.

Now establish the connections between the outlet variables and the label objects in the storyboard. Right click the Cell in the document outline view to bring up the context menu. Connect `fieldLabel` with the *Field* label object and `valueLabel` with the *Value* label object respectively.

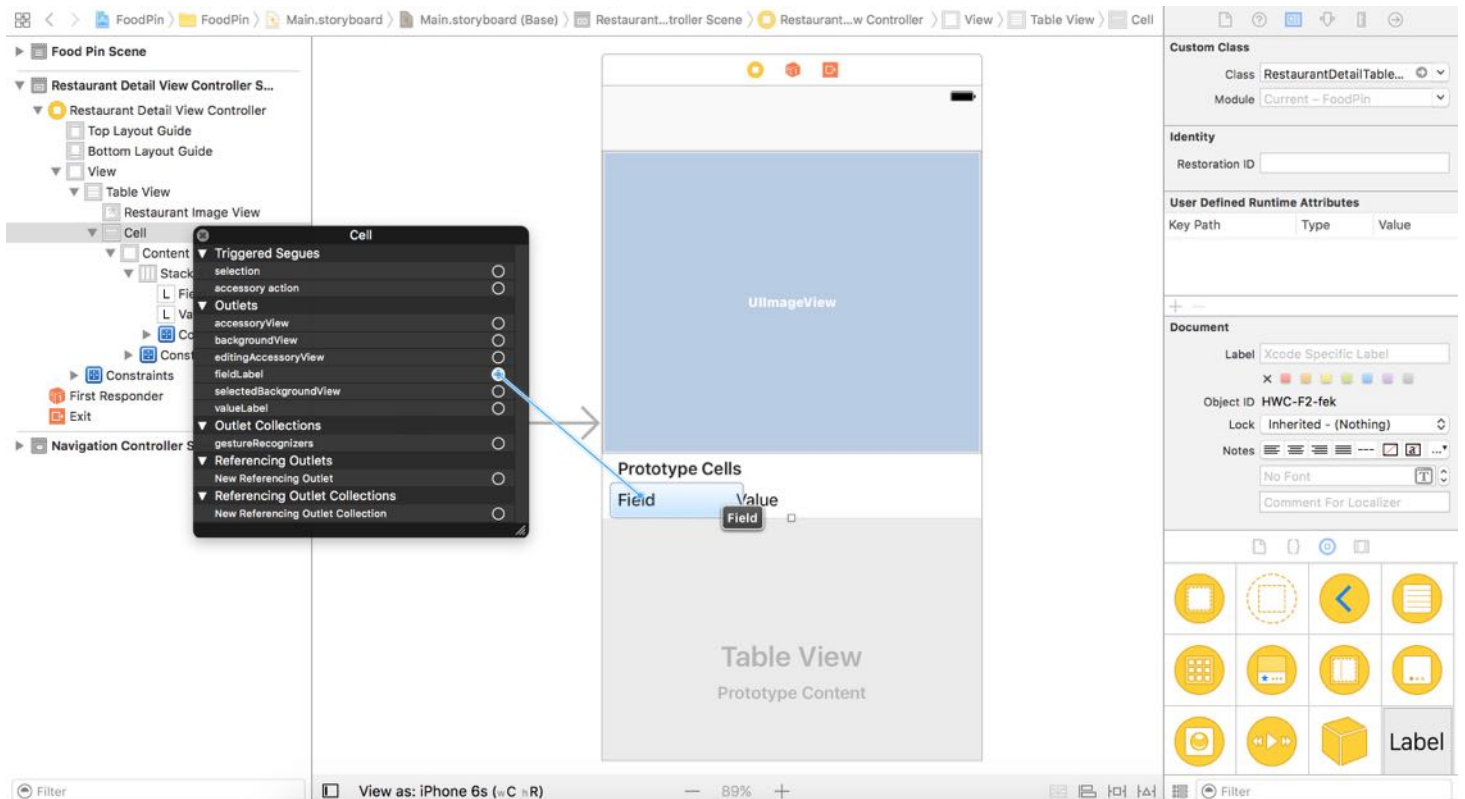


Figure 14-16. Connecting the outlets

Updating the RestaurantDetailViewController Class

The `RestaurantDetailViewController` class is a subclass of `UIViewController`. As you've learned in chapter 8, we have to adopt both the `UITableViewDataSource` and `UITableViewDelegate` protocols for displaying content in the table view.

In `RestaurantDetailViewController.swift`, update the class declaration to adopt the protocols:


```
class RestaurantDetailViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate {
```

We then populate the restaurant information by implementing the required methods of the `UITableViewDataSource` protocol:

```
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) ->
Int {
    return 4
}

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

    // Configure the cell...
    switch indexPath.row {
    case 0:
        cell.fieldLabel.text = "Name"
        cell.valueLabel.text = restaurant.name
    case 1:
        cell.fieldLabel.text = "Type"
        cell.valueLabel.text = restaurant.type
    case 2:
        cell.fieldLabel.text = "Location"
        cell.valueLabel.text = restaurant.location
    case 3:
        cell.fieldLabel.text = "Been here"
        cell.valueLabel.text = (restaurant.isVisited) ? "Yes, I've been here
before" : "No"
    default:
        cell.fieldLabel.text = ""
        cell.valueLabel.text = ""
    }

    return cell
}
```

The above code is very straightforward. The app displays 4 rows of restaurant information including *name*, *type*, *location* and *been here*. Other than `if` statement, you can use `switch` to control the program flow and execute different branches of code. If you've studied other programming languages before, the `switch` statement in Swift has been significantly improved and is more powerful than its counterparts. One highlight is the cases of `switch` statement do not "fall through" to the next case. In other words, the entire `switch` statement finishes its

execution when a case is matched, without requiring an explicit break statement. You can learn more about switch statement in the appendix.

Okay, the code of the `RestaurantDetailViewController` class is ready. However, we haven't established the connection with the table view in storyboard.

In the document outline of the Interface Builder editor, right click the Table View object and connect the `dataSource` outlet with Restaurant Detail View Controller. Repeat the same procedure to connect the `delegate` outlet.

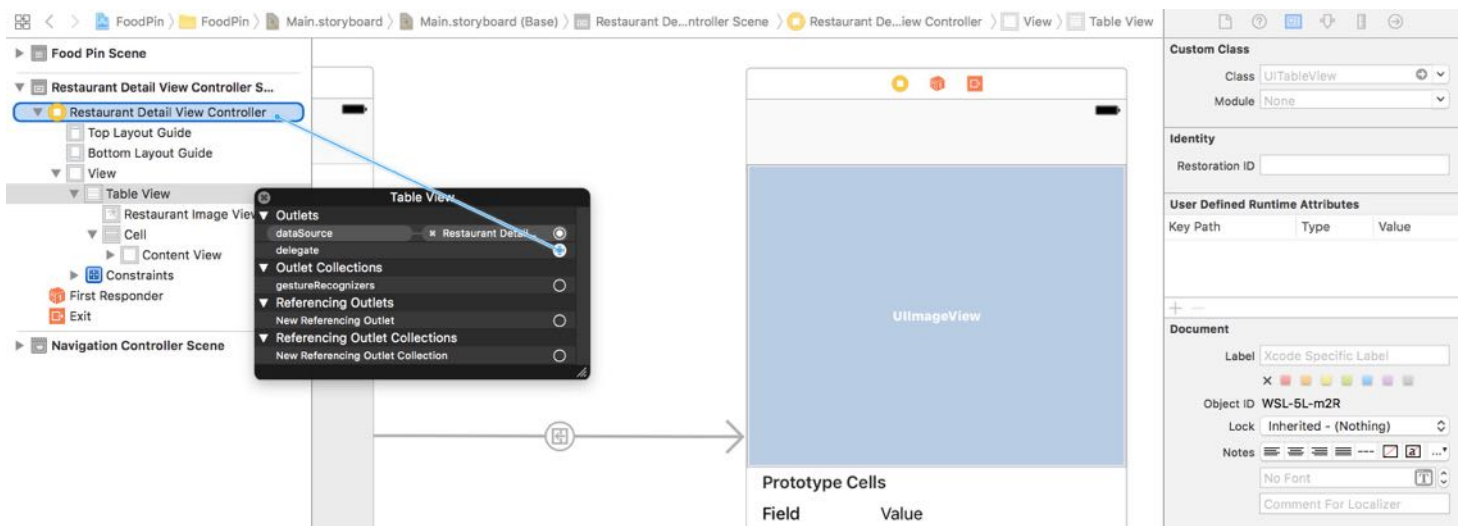


Figure 14-17. Connecting the data source and delegate of the table view

Ready to Test

Hit the Run button and test your app. The detail view now displays more information about the selected restaurant.

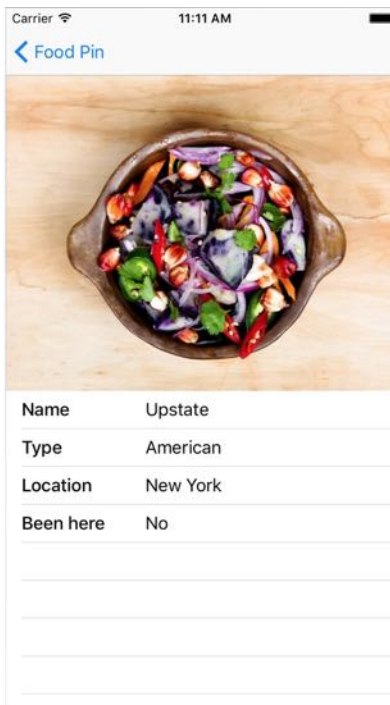


Figure 14-18. Detail view of a sample restaurant

Customizing the Table View Appearance

The table view is pretty nice. But there are a number of things we can implement to make it even better:

- Change the background color of the table view
- Remove the separators of the empty rows
- Change the color of the separator

Before we dive into the customization, add an outlet variable in the

`RestaurantDetailViewController` class so that we can establish a connection with the table view object in the storyboard.

```
@IBOutlet var tableView:UITableView!
```

Go to Interface Builder. Right click Table View and drag the + icon of the New Reference Outlet to `Restaurant Detail View Controller`. Select `tableView` when prompted.

To change the background color of the table view, set a new `UIColor` object using the `backgroundColor` property. Insert the following line in the `viewDidLoad` method:

```
tableView.backgroundColor = UIColor(red: 240.0/255.0, green: 240.0/255.0, blue: 240.0/255.0, alpha: 0.2)
```

This will change the table background to light gray. Feel free to change it to your preferred color. If you run the app, the background color of the table view is partially changed.

Quick tip: The color change is subtle. If you're not aware of the color change, you may change it to a more vibrant color:

```
tableView.backgroundColor = UIColor(red: 0.0/255.0, green: 240.0/255.0, blue: 240.0/255.0, alpha: 0.2)
```

For those cells with content, the background color is still in white. You'll need to insert the following line of code in the `tableView(_:cellForRowAt:)` method. Put the line of code right before `return cell`:

```
cell.backgroundColor = UIColor.clear
```

This makes the cells transparent, so that the background color of the table view can be seen.

The second item we want to tweak is to remove the separators of the empty rows. It can be achieved by setting the footer of the table view to blank. Add the following line to the `viewDidLoad` method:

```
tableView.tableFooterView = UIView(frame: CGRect.zero)
```

Lastly, we change the color of the separators for content rows. Again, it can be done by a line of code. Put the following code in the `viewDidLoad` method:

```
tableView.separatorColor = UIColor(red: 240.0/255.0, green: 240.0/255.0, blue: 240.0/255.0, alpha: 0.8)
```

Now it's ready to test the app. You'll find a slight color change for the background and separator. If you find the color change is too subtle, feel free to change to whatever color as you like.

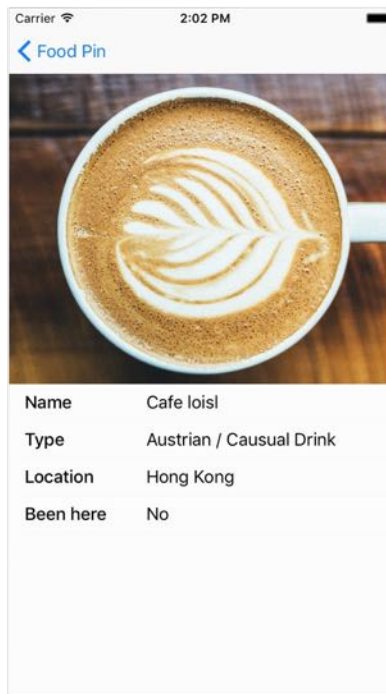


Figure 14-19. Restaurant Detail View after table view customizations

Customizing the Appearance of Navigation Bar

Now that you know how to customize the appearance of table view. The next thing I'm going to show you is to customize the appearance of navigation bar. Here are a few changes we'll make:

- Changing the background color of navigation bar
- Changing the font of navigation bar title
- Customizing the color of back button
- Changing the title of back button

Quick tip: User will pay a premium for a great looking app. Even if you're not an app designer, try your best to make your app stand out. If you need some inspirations, check out pptrns.com, inspired-ui.com and mobile-patterns.com. You will find a tons of iPhone and iPad user interface patterns.

Apple provides the Appearance API for developers to customize the visual appearance of the UIKit controls across the entire application, through an `appearance` proxy of a specific class. For example, to customize the appearance of navigation bar, you use `appearance()` to get the appearance proxy of the class:

```
UINavigationController.appearance()
```

To modify the background color of navigation bar, set the `barTintColor` property to your preferred color:

```
UINavigationController.appearance().barTintColor = UIColor(red: 216.0/255.0, green: 74.0/255.0, blue: 32.0/255.0, alpha: 1.0)
```

The title style can be changed by setting the `titleTextAttributes` property:

```
if let barFont = UIFont(name: "Avenir-Light", size: 24.0) {
    UINavigationController.appearance().titleTextAttributes =
    [NSForegroundColorAttributeName:UIColor.white, NSFontAttributeName:barFont]
}
```

You can specify the font, text color, text shadow color, and text shadow offset for the title in the text attributes dictionary. In the above code, we specify `Avenir-Light` as the font and set the color to `white`.

Where to find the name of iOS font?

You may wonder how to find out the name of iOS font. The easiest way is to check out <http://iosfonts.com>. The site lists out all the available fonts in iOS.

The `tintColor` property controls the color of navigation items and bar button items. You use it to change the color of back button.

```
UINavigationController.appearance().tintColor = UIColor.white
```

The `AppDelegate` is sort of the entry point of the application. The class is generated by Xcode when your project is created from a project template. Typically we put these customization code in the `AppDelegate` class as the changes apply to the entire application.

Insert the following code in the `application(_:didFinishLaunchingWithOptions:)` method:

```
UINavigationController.appearance().barTintColor = UIColor(red: 216.0/255.0, green: 74.0/255.0, blue: 32.0/255.0, alpha: 1.0)
UINavigationController.appearance().tintColor = UIColor.white
```

```
if let barFont = UIFont(name: "Avenir-Light", size: 24.0) {
    UINavigationController.appearance().titleTextAttributes =
[NSForegroundColorAttributeName:UIColor.white, NSFontAttributeName:barFont]
}
```

That's it! You can now run your app and have a look. With the Appearance API, it's pretty easy to customize the navigation bar.

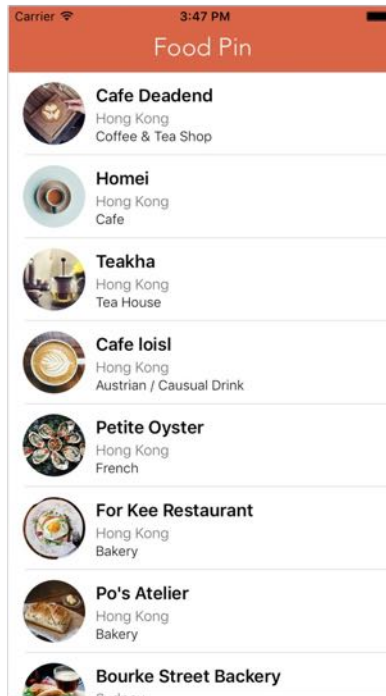


Figure 14-20. Styling the navigation bar

Presently, the back button of the navigation bar shows the app's title. What if you want to remove the back button title and just keep the back arrow? By default, the title of back button is pulled from the title of the source controller in a segue.

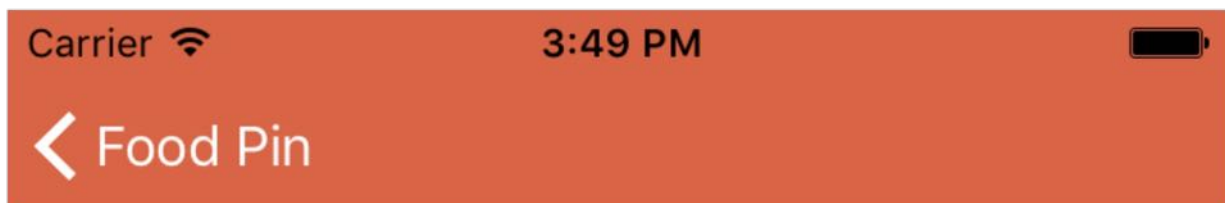


Figure 14-21. Back button with title

To remove the title, all you need to do is update the `backBarButtonItem` property of the navigation item and assign it with a bar button item with a blank title. But remember this update should be applied in the `viewDidLoad` method of the source view controller. In this case, it's the `RestaurantTableViewController` class. So update the `viewDidLoad` method like this:

```
override func viewDidLoad() {
    super.viewDidLoad()

    // Remove the title of the back button
    navigationItem.backBarButtonItem = UIBarButtonItem(title: "", style:
.plain, target: nil, action: nil)
}
```

Let's make one more change. The navigation bar title in the detail view is currently empty. I want to display the restaurant name in the navigation bar. It's super easy to do so. You can simply set the title of the detail view to the name of restaurant. Insert the following line of code in the `viewDidLoad` method of the `RestaurantDetailViewController` class:

```
title = restaurant.name
```

If you test the app again, the navigation bar in the detail view should look like this.

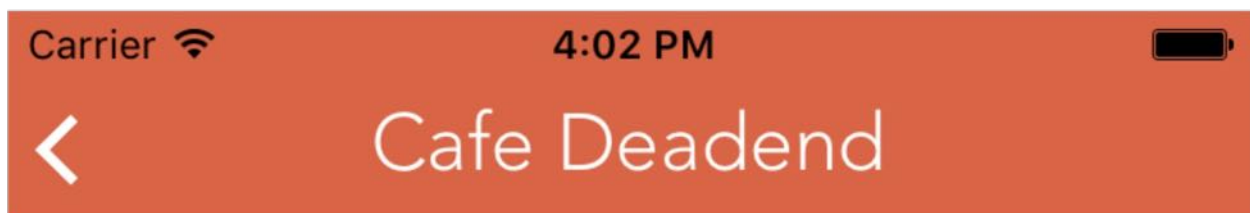


Figure 14-22. Back button without title

Hiding a Navigation Bar on Swipe

Since iOS 8, Apple has introduced a nifty feature which allows you to hide a navigation bar on swipe or tap. This feature shouldn't be new to you. When scrolling down a web page in mobile Safari, the address bar is condensed and the toolbar goes away. Similar feature can be found in some of the popular apps such as Indiegogo.

Prior to iOS 8, you need to develop your own solution if you want to hide a navigation on swipe. Starting from iOS 8, Apple opened up the feature for all iOS developers. You'll need a click or a line of code to enable this feature.

In the Interface Builder editor, select the navigation controller in the navigation controller scene. In the Attribute inspector, there is an option named `On Swipe` under the navigation controller section. When enabled, your app will hide the navigation bar, as well as, toolbar (if any) when a user scrolls the content view. The bars appear again when the user swipes back up.

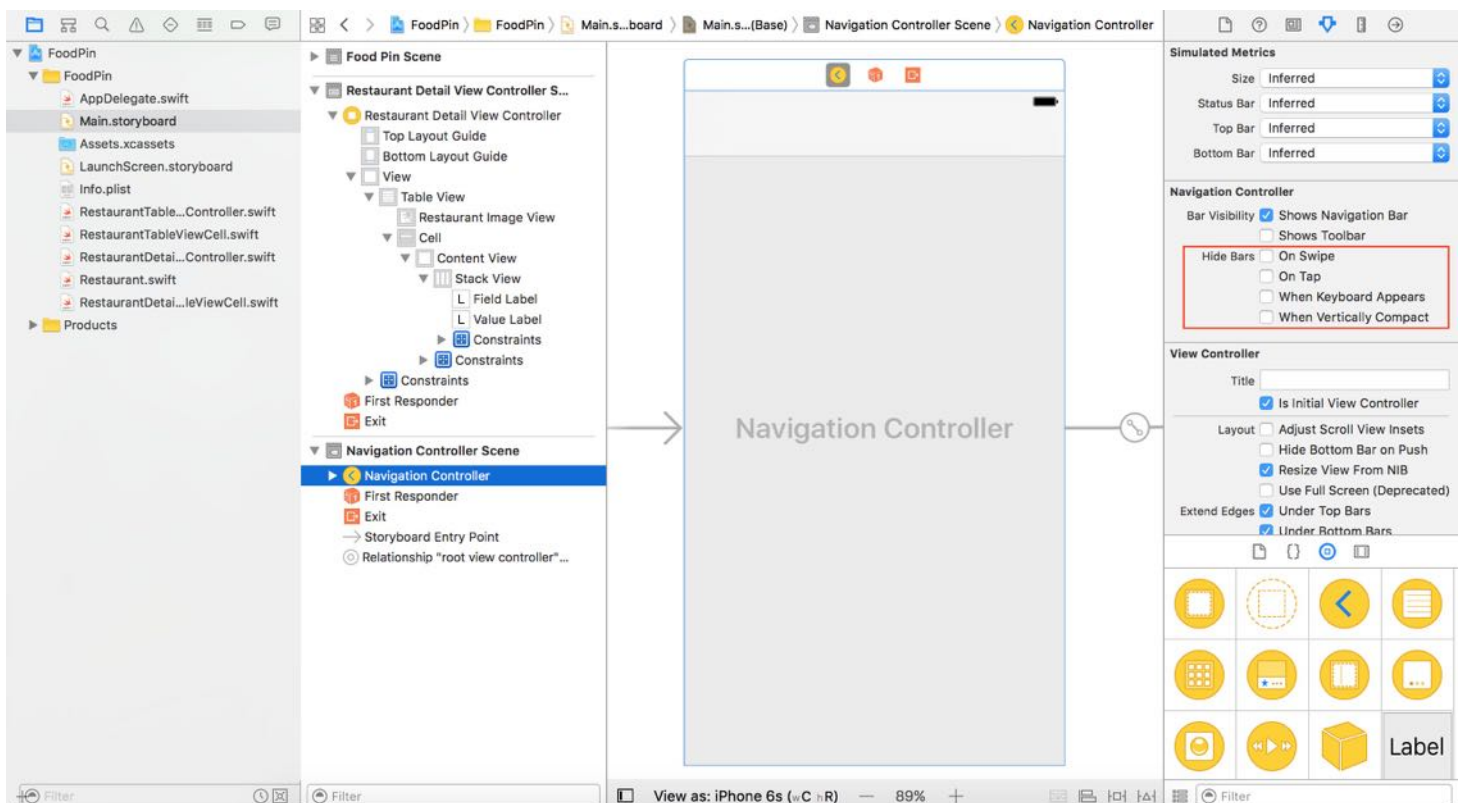


Figure 14-23. On Swipe option for hiding the bars

This change applies to all navigation bars of the entire app. That means, both the restaurant

table view controller and the detail view controller are automatically enabled with the feature. Now what if we just want to hide the navigation bar for the restaurant table view controller? It would be better for us to do this in code.

Quick note: If you've enabled the On Swipe option in storyboard, please disable it before moving on.

To hide the bars on swipe, you set the `hidesBarsOnSwipe` property to `true` using the code below:

```
navigationController?.hidesBarsOnSwipe = true
```

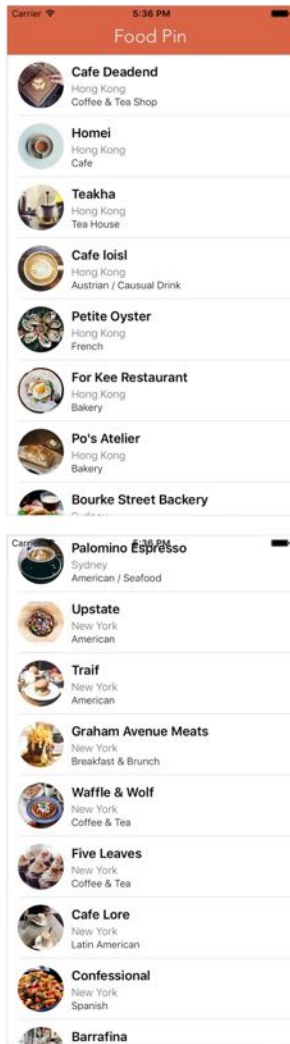
Let's add the above line of code in the `viewDidLoad` method of `RestaurantTableViewController`. After that, run the app to have a quick test. Swipe up the table view and the navigator bar will be hidden. It works pretty good.

However, if you select a restaurant to navigate to the detail view, this swipe-to-hide feature is enabled too. Okay, we want to disable the swipe-to-hide feature in the detail view. Let's add the following code in the `viewDidLoad` method of `RestaurantDetailViewController` to disable it:

```
navigationController?.hidesBarsOnSwipe = false
```

Run the app again to test it. Does it fix the issue?

At first, it looks like we have fixed the navigation bar issue. But if you test more thoroughly, you will find other issues as illustrated in figure 14-24.



→
←

Problem #1
Cannot hide the navigation bar after navigating back to the Restaurant table view controller

→

Problem #2
Missing the navigation bar in the detail view

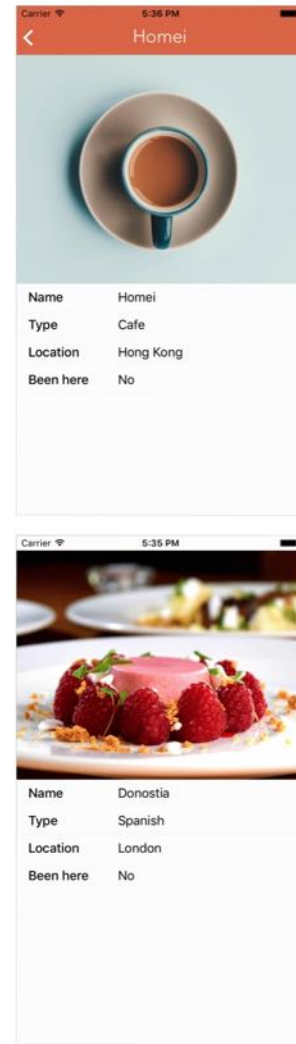


Figure 14-24. Possible issues when setting the `hidesBarsOnSwipe` property in the `viewDidLoad` method

First, keep in mind that the `viewDidLoad` method is called only once when the view is first loaded. After that, it won't be called again. In other words, if you navigate to the detail view, the `hidesBarsOnSwipe` property is set to `false`. And the value will keep intact even when you navigate back to the table view controller. This is the reason why the navigation bar won't hide (Problem #1).

For Problem #2, the hidden navigation bar is carried over to the detail view. Even if we manage to set the `hidesBarsOnSwipe` property back to `false` in the `viewDidLoad` method, it won't display the navigation bar. We have to explicitly tell the app to re-display the navigation bar.

Fortunately, the `UIViewController` class provides several methods that respond to view events. Unlike the `viewDidLoad` method, these methods are called every time when a view is displayed or removed. When a view is displayed, both `viewWillAppear` and `viewDidAppear` methods are invoked. The `viewWillAppear` method is called when a view is about to display, whereas the `viewDidAppear` method is called when a view is displayed on screen.

Apparently, the `viewWillAppear` method fits our need. Insert the following code in the `RestaurantTableViewController` class:

```
override fun viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    navigationController?.hidesBarsOnSwipe = true
}
```

For the `RestaurantDetailViewController` class, add the following code snippet:

```
override fun viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    navigationController?.hidesBarsOnSwipe = false
    navigationController?.setNavigationBarHidden(false, animated: true)
}
```

As the `viewWillAppear` method is called every time when a view is displayed, we can toggle the `hidesBarsOnSwipe` property perfectly. The extra line of code in the `RestaurantDetailViewController` class explicitly tells the app to unhide the navigation bar. This resolves problem #2. One thing to take note is that you must call the `super` method at some point of the implementation. After the changes, run the app and have fun!

Change the Style of Status Bar

The style of the status bar doesn't go well with the color of the navigation bar. Wouldn't it be great if we can change the color of status bar to white? This is the last thing I want to discuss with you in this chapter.

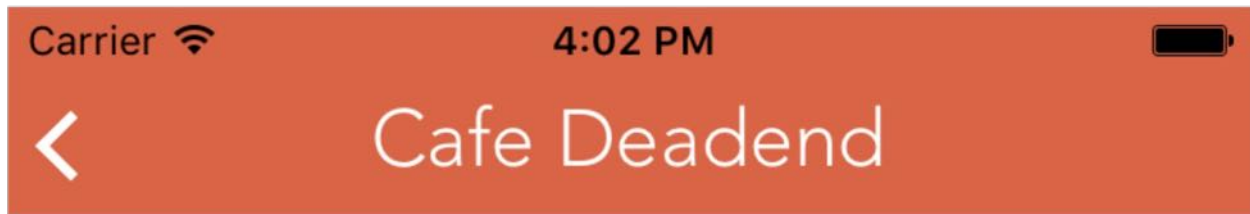


Figure 14-25. Default style of status bar

In older versions of iOS, the status bar was always in black and there is not much you can change. From iOS 7 and onwards, you're allowed to change the appearance of the status bar per view controller. You can use a `UIStatusBarStyle` constant to specify whether the status bar content should be dark or light. By default, the status bar displays dark content. In other words, items such as time, battery indicator and Wi-Fi signal are displayed in dark color, just like the one shown in figure 14-25.

You may want to change the style of status bar from dark to light to make the app look better. There are two ways to do this. First, you're allowed to control the style of the status bar in any view controllers by overriding the `preferredStatusBarStyle` method:

```
override func preferredStatusBarStyle() -> UIStatusBarStyle {  
    return .lightContent  
}
```

Alternatively, you can set it through the `statusBarStyle` property of `UIApplication`. This lets you change the style of the status bar for the entire app. By default, however, the Xcode project is enabled to use "View controller-based status bar appearance". This means you can control the appearance of the status per view controller. If you want to change the style of the status globally, you need to first opt out the *View controller-based status bar appearance*.

Select the `Info.plist` in project navigator. Right click any blank area and select *Add Row*. Insert a new key named `view controller-based status bar appearance` and set the value to `NO`. Refer to figure 14-26 for details.

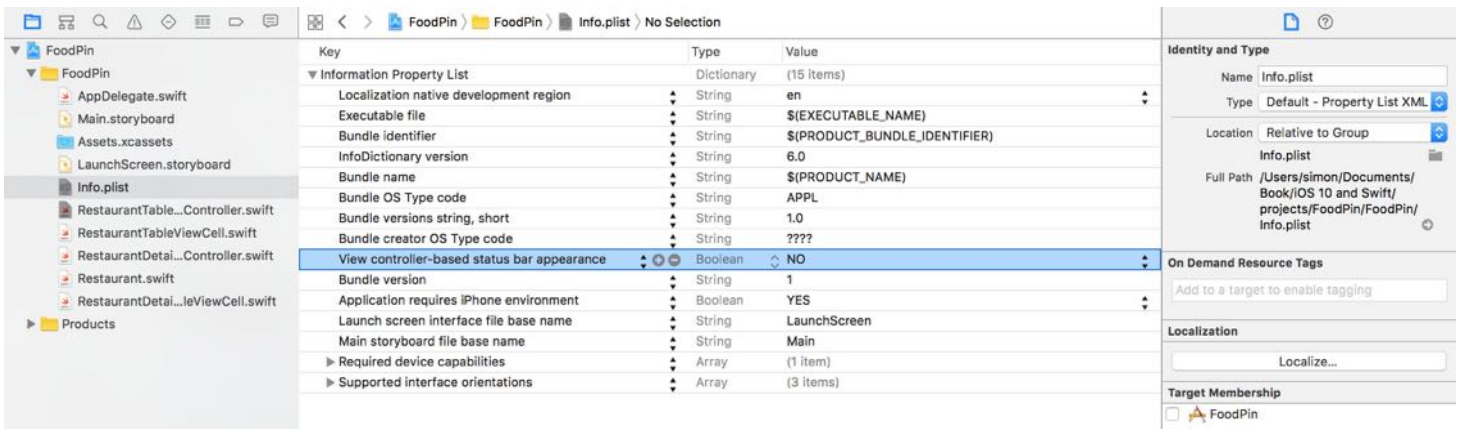


Figure 14-26. Adding a key to disable view controller-based status bar appearance

By disabling *View controller-based status bar appearance*, you can change the appearance of the status bar by using the following code:

```
UIApplication.shared.statusBarStyle = .lightContent
```

Normally you put the above code in the `application(_:didFinishLaunchingWithOptions:)` method of the `AppDelegate` class. If you apply the change using one of the approaches, the status bar will be updated to the light style.

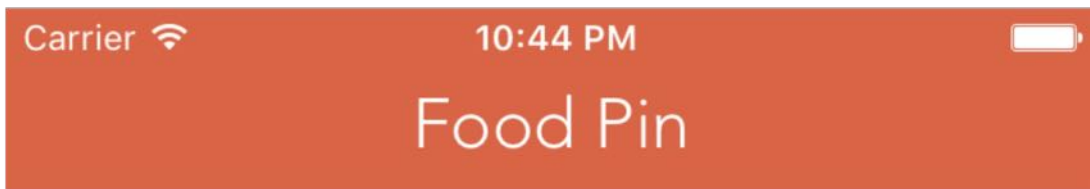


Figure 14-27. Status bar in light style

Exercise #1

This exercise will help you refresh your memory of Object Oriented Programming. Tweak the app and see if you can display a Phone field in the detail view.

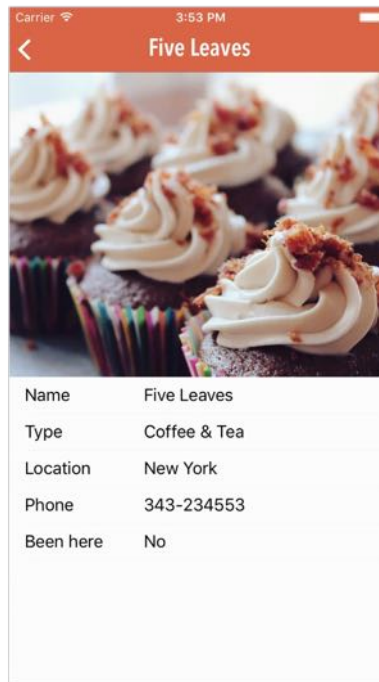


Figure 14-28. Adding phone number to the detail view

Hint: You'll need to add a phone number property in the Restaurant class and modify the initialization of the `restaurants` variable in `RestaurantTableViewController`.

Exercise #2

Instead of using the Avenir font for the navigation bar header, your user prefers to use the Avenir Next Condensed font with Demi Bold style. Please make the change accordingly. Your final result should look like this:

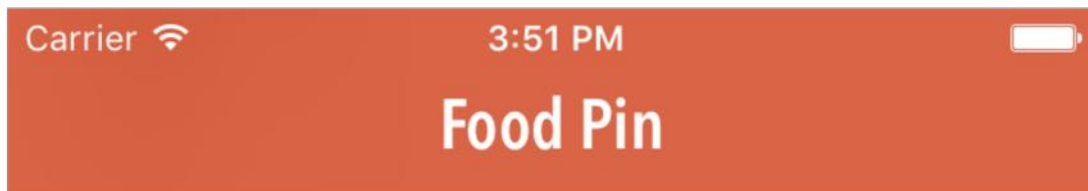


Figure 14-29. Changing the font for the navigation bar

Hint: Go to iosfonts.com to look up the actual font name on iOS.

Summary

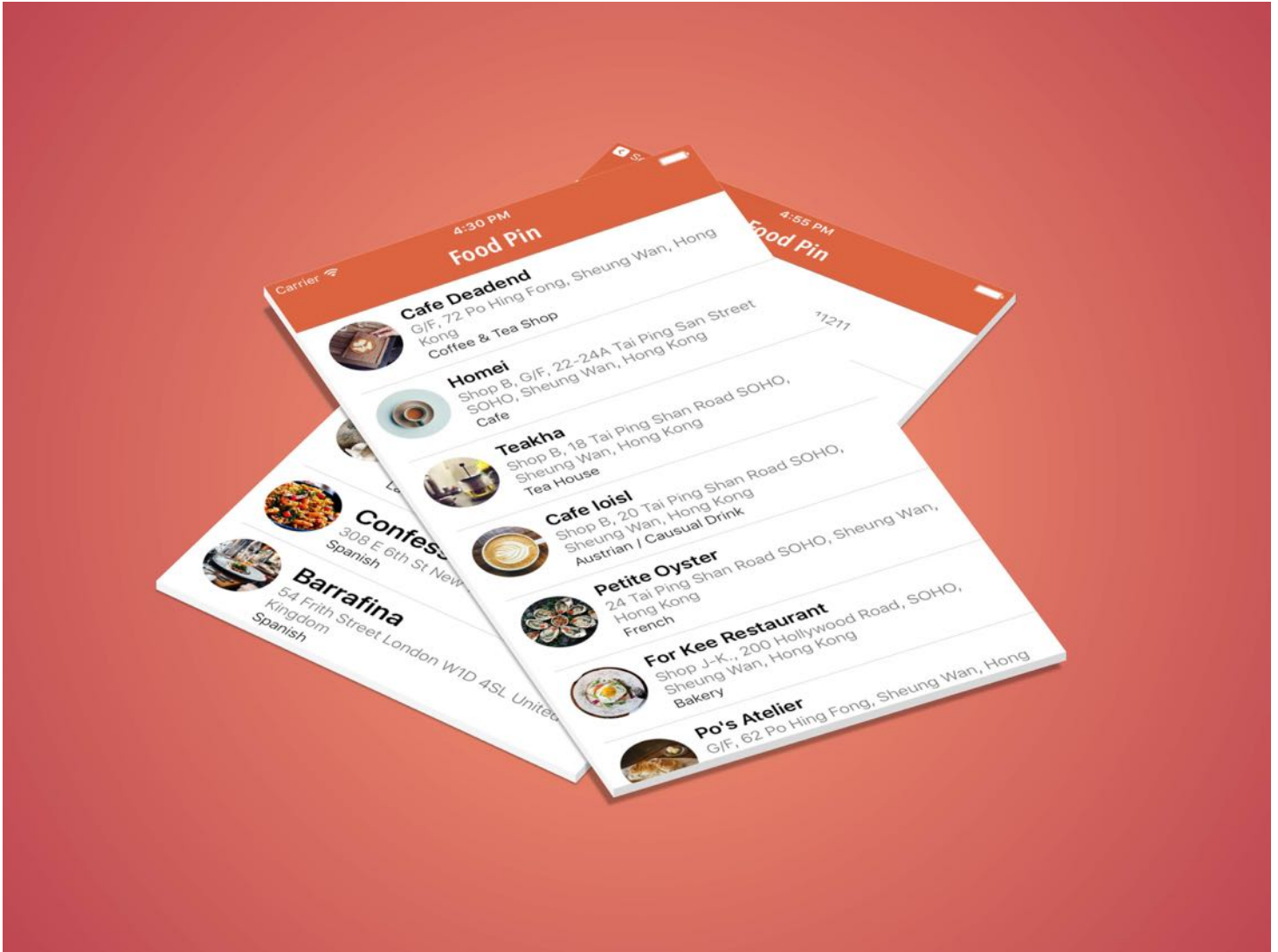
I hope you love this chapter and love the app you created. Congratulations again if you made this far. You've built a polished Restaurant app. It's not a complex app but you've managed to use some common components such as table view and navigation controller in iOS. What's more you've learned how to customize navigation bar and table view. Remember you should try your best to make your app stand out from others.

For reference, you can download the complete project from <http://www.appcoda.com/resources/swift3/FoodPinDetailView.zip>.

In the next chapter, we'll explore a great feature known as *Self Sizing Cell*.

Chapter 15

Self Sizing Cells and Dynamic Type



Fun is one of the most important and underrated ingredients in any successful venture. If you're not having fun, then it's probably time to call it quits and try something else.

- Richard Branson

In iOS 8, Apple introduces a new feature for `UITableView` known as *Self Sizing Cells*. This is seriously one of the most useful features of the SDK. Prior to iOS 8, if you want to display dynamic content in a table view with multiple rows, you have to calculate the row height of a

cell manually. Now iOS is capable to determine the row height for you. All you need to do is write a couple lines of code and make sure you define all the layout constraints for the prototype cell (that we have already done).

Self sizing cells are particularly useful to support Dynamic Type. By using Dynamic Type, your app can respond to the text change and allow users to customise the text size to fit their own needs. Dynamic Type is another topic which will be covered in this chapter.

Let's start with self sizing cells. In summary, here are the procedures to implement self sizing cells:

- Add auto layout constraints in your prototype cell
- Specify the `estimatedRowHeight` property of your table view
- Set the `rowHeight` property of your table view to `UITableViewAutomaticDimension`

If we express the last two points in code, it looks like this:

```
tableView.estimatedRowHeight = 36.0
tableView.rowHeight = UITableViewAutomaticDimension
```

With just two lines of code, you instruct the table view to calculate the cell's size, matching its content and render it dynamically. This self sizing cell feature should save you tons of code and time. You're gonna love it. There is no better way to learn a new feature than using it. We'll continue to develop the FoodPin app and turn the table view cell in the detail view into self sizing cell.

So far, the cell content fits perfectly without any truncation because the data didn't contain any lengthy text. To test out self sizing cells, we will first update the `restaurants` array and change each of the restaurant location to a full address.

For your convenience, you can download the starter project from <http://www.appcoda.com/resources/swift3/FoodPinSelfSizingCellStarter.zip>. The project is exactly the same as the one you work on in the previous chapter, except that I have updated the `restaurants` array to the following:

```
var restaurants:[Restaurant] = [
    Restaurant(name: "Cafe Deadend", type: "Coffee & Tea Shop", location: "G/F,
```

72 Po Hing Fong, Sheung Wan, Hong Kong", phone: "232-923423", image: "cafedeadend.jpg", isVisited: false),
Restaurant(name: "Homei", type: "Cafe", location: "Shop B, G/F, 22-24A Tai Ping San Street SOHO, Sheung Wan, Hong Kong", phone: "348-233423", image: "homei.jpg", isVisited: false),
Restaurant(name: "Teakha", type: "Tea House", location: "Shop B, 18 Tai Ping Shan Road SOHO, Sheung Wan, Hong Kong", phone: "354-243523", image: "teakha.jpg", isVisited: false),
Restaurant(name: "Cafe loisl", type: "Austrian / Casual Drink", location: "Shop B, 20 Tai Ping Shan Road SOHO, Sheung Wan, Hong Kong", phone: "453-333423", image: "cafeloisl.jpg", isVisited: false),
Restaurant(name: "Petite Oyster", type: "French", location: "24 Tai Ping Shan Road SOHO, Sheung Wan, Hong Kong", phone: "983-284334", image: "petiteoyster.jpg", isVisited: false),
Restaurant(name: "For Kee Restaurant", type: "Bakery", location: "Shop J-K., 200 Hollywood Road, SOHO, Sheung Wan, Hong Kong", phone: "232-434222", image: "forkeerrestaurant.jpg", isVisited: false),
Restaurant(name: "Po's Atelier", type: "Bakery", location: "G/F, 62 Po Hing Fong, Sheung Wan, Hong Kong", phone: "234-834322", image: "posatelier.jpg", isVisited: false),
Restaurant(name: "Bourke Street Bakery", type: "Chocolate", location: "633 Bourke St Sydney New South Wales 2010 Surry Hills", phone: "982-434343", image: "bourkestreetbakery.jpg", isVisited: false),
Restaurant(name: "Haigh's Chocolate", type: "Cafe", location: "412-414 George St Sydney New South Wales", phone: "734-232323", image: "haighschocolate.jpg", isVisited: false),
Restaurant(name: "Palomino Espresso", type: "American / Seafood", location: "Shop 1 61 York St Sydney New South Wales", phone: "872-734343", image: "palominoespresso.jpg", isVisited: false),
Restaurant(name: "Upstate", type: "American", location: "95 1st Ave New York, NY 10003", phone: "343-233221", image: "upstate.jpg", isVisited: false),
Restaurant(name: "Traif", type: "American", location: "229 S 4th St Brooklyn, NY 11211", phone: "985-723623", image: "traif.jpg", isVisited: false),
Restaurant(name: "Graham Avenue Meats", type: "Breakfast & Brunch", location: "445 Graham Ave Brooklyn, NY 11211", phone: "455-232345", image: "grahamavenuemats.jpg", isVisited: false),
Restaurant(name: "Waffle & Wolf", type: "Coffee & Tea", location: "413 Graham Ave Brooklyn, NY 11211", phone: "434-232322", image: "wafflewolf.jpg", isVisited: false),
Restaurant(name: "Five Leaves", type: "Coffee & Tea", location: "18 Bedford Ave Brooklyn, NY 11222", phone: "343-234553", image: "fiveleaves.jpg", isVisited: false),
Restaurant(name: "Cafe Lore", type: "Latin American", location: "Sunset Park 4601 4th Ave Brooklyn, NY 11220", phone: "342-455433", image: "cafelore.jpg", isVisited: false),
Restaurant(name: "Confessional", type: "Spanish", location: "308 E 6th St New York, NY 10003", phone: "643-332323", image: "confessional.jpg", isVisited: false),

```

Restaurant(name: "Barrafina", type: "Spanish", location: "54 Frith Street
London W1D 4SL United Kingdom", phone: "542-343434", image: "barrafina.jpg",
isVisited: false),
Restaurant(name: "Donostia", type: "Spanish", location: "10 Seymour Place
London W1H 7ND United Kingdom", phone: "722-232323", image: "donostia.jpg",
isVisited: false),
Restaurant(name: "Royal Oak", type: "British", location: "2 Regency Street
London SW1P 4BZ United Kingdom", phone: "343-988834", image: "royaloak.jpg",
isVisited: false),
Restaurant(name: "CASK Pub and Kitchen", type: "Thai", location: "22
Charlwood Street London SW1V 2DY Pimlico", phone: "432-344050", image:
"caskpubkitchen.jpg", isVisited: false)
]

```

The values of the `location` property are updated with the full address. If you run the app and navigate to the detail view, you should notice that the location is truncated. We will modify the app so that the cell height can be adjusted automatically to fit its content.

Name	Haigh's Chocolate
Type	Cafe
Location	412-414 George St Sydney Ne...

Adding Auto Layout Constraints for the Prototype Cell

You cannot use self sizing cells without applying auto layout. iOS relies on the layout constraints to calculate the size of the cell. The very first step in implementing self sizing cells is to define the layout constraints for the prototype cell. For our FoodPin project, we have already defined the constraints in the earlier chapter. So let's enable self sizing cells and see if it works.



Figure 15-2. Our prototype cell has been defined with the necessary layout constraints

Enabling Self Sizing Cells

To enable self sizing cells for the detail view, add the following code in the `viewDidLoad` method of the `RestaurantDetailViewController` class:

```
tableView.estimatedRowHeight = 36.0
tableView.rowHeight = UITableViewAutomaticDimension
```

As explained earlier, the first line of code sets the estimated row height of the cell. That's the height of the existing prototype cell. The second line changes the `rowHeight` property to `UITableViewAutomaticDimension`, which is the default row height in iOS 10.

Setting the Label Lines to Zero

If you run the app now, the cell is not resized. The reason is that we have explicitly set the "number of lines" of the *Value* label to `1`. In other words, the label can only display a single line of content.

Now go to the Interface Builder editor and select the *Value* label. In the Attributes inspector, change the value of the *Lines* option to `0`. The label will automatically adjust the number of lines to fit the content.

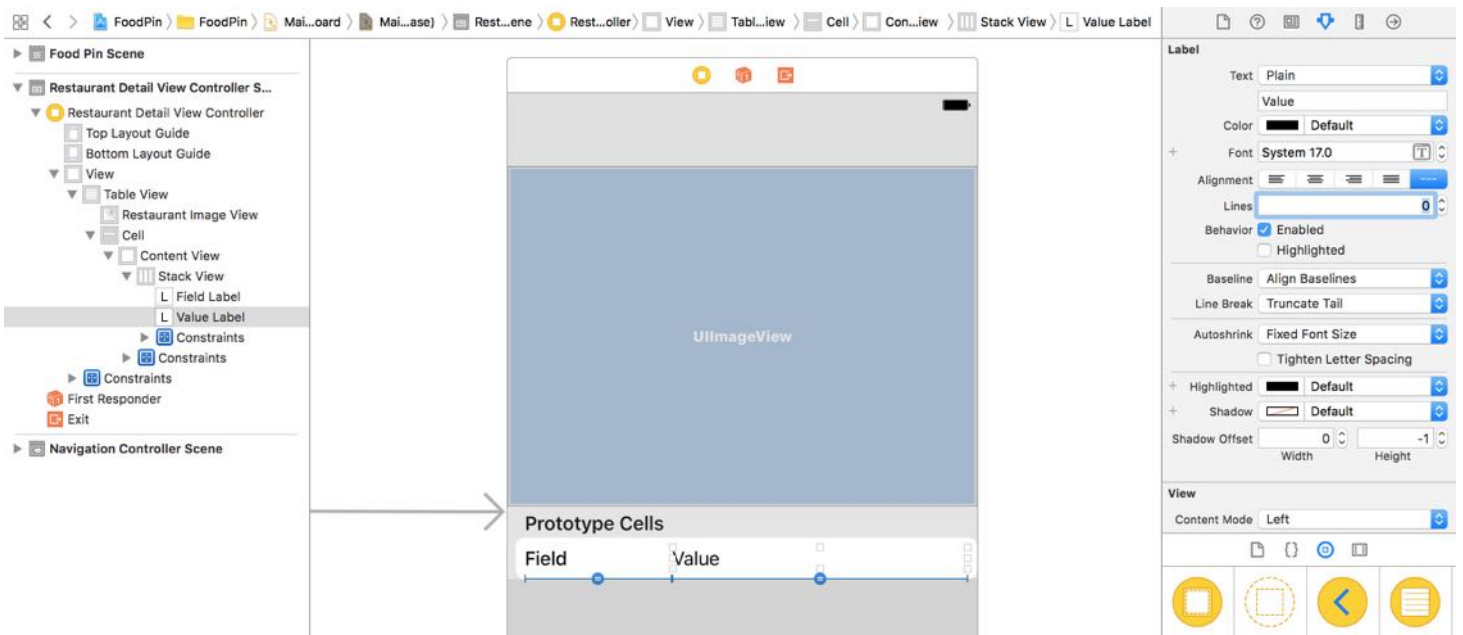


Figure 15-3. Changing the Lines option of the Value label

That's it. You can now test the app. The cell now resizes dynamically even in landscape mode.



Figure 15-4. Self sizing cells in the FoodPin app

Adding Spacing Constraints

Everything looks great. But if you look into the console, you may notice the following warning:

```
Warning once only: Detected a case where constraints ambiguously suggest a height of zero for a tableview cell's content view. We're considering the collapse unintentional and using standard height instead.
```

On top of that, if you navigate to the *Homei* restaurant, the location cell is not correctly resized. It seems iOS can't calculate the correct row height based on the existing constraints. To resolve the issues, we will add the spacing constraints for the top and bottom sides of the stack view. This will give iOS more information about the cell's layout.



Figure 15-5. The location field is not correctly resized for address that takes up more than three rows

Now select the stack view in the prototype cell. Click the Pin button to add a couple of spacing constraints. Select the bar of the top and bottom side. Make sure the bar turns into solid red and then click `Add 2 Constraints` to add the spacing constraints.

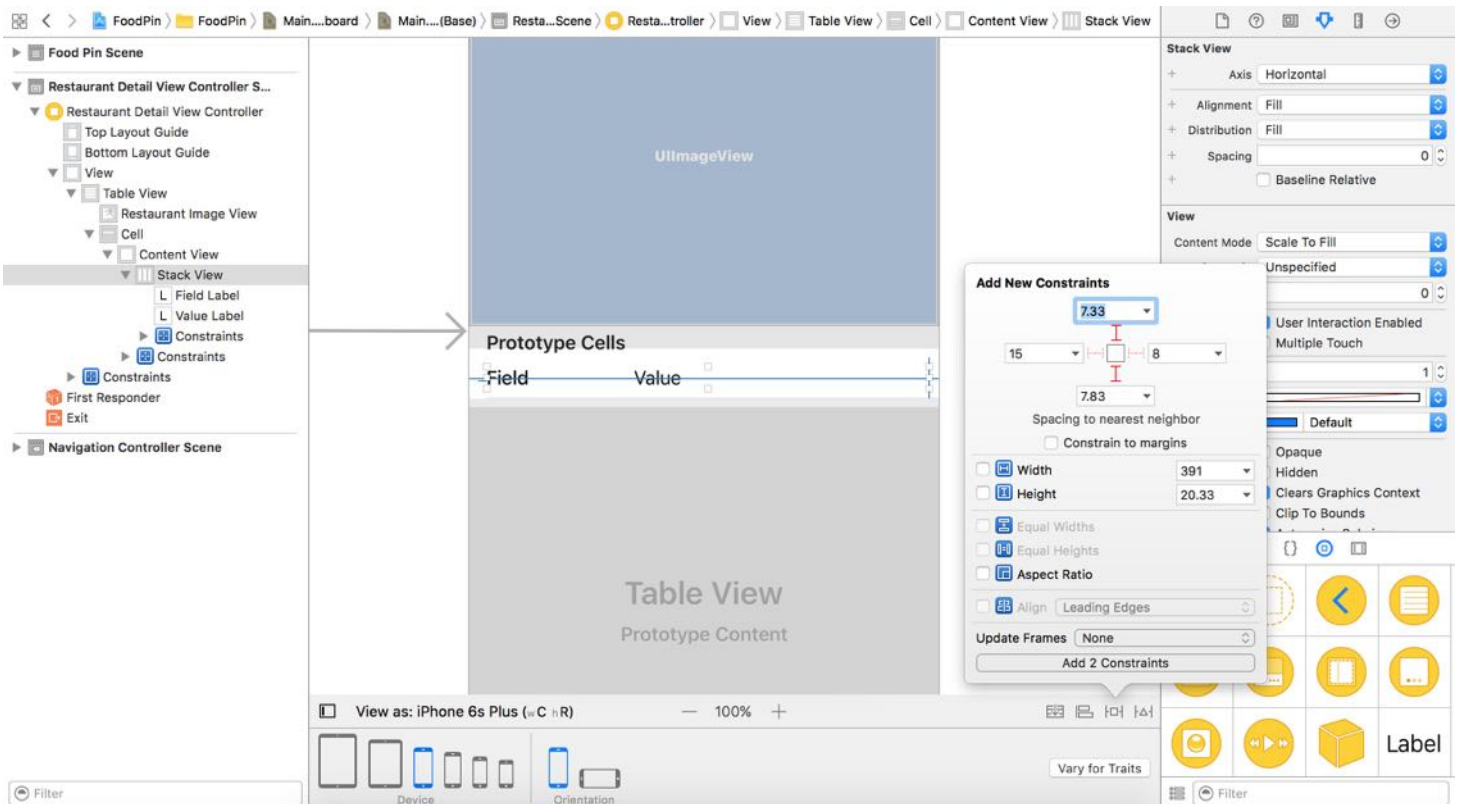


Figure 15-6. Adding spacing constraints for the top and bottom sides of the stack view

It's time to test out the changes and see if it resolves the issue. Run the project in a iPhone simulator, and choose the *Homei* restaurant. Unfortunately, it doesn't work yet. In the console, it shows some errors related to layout constraints:

```
[LayoutConstraints] Unable to simultaneously satisfy constraints.
    Probably at least one of the constraints in the following list is one you
don't want.
    Try this:
    (1) look at each constraint and try to figure out which you don't
expect;
    (2) find the code that added the unwanted constraint or constraints and
fix it.
(
    "<NSLayoutConstraint:0x60000028d700 UIStackView:0x7fa763f122d0.centerY ==
UITableViewControllerContentView:0x7fa763f12bd0.centerY (active)>",
    "<NSLayoutConstraint:0x60000028d7a0 V:|-(7.33)-[UIStackView:0x7fa763f122d0]
(active, names: '|':UITableViewControllerContentView:0x7fa763f12bd0 )>",
    "<NSLayoutConstraint:0x60000028d7f0 V:[UIStackView:0x7fa763f122d0]-(7.83)-|
(active, names: '|':UITableViewControllerContentView:0x7fa763f12bd0 )>"
)
```

The app failed to fulfill all the layout constraints when rendering the stack view of the table view cell. Let's take a look at the above constraint errors. It mentioned three layout constraints of the stack view:

1. First, it is the alignment constraint. The stack view is required to center vertically.
2. Secondly, it is the spacing constraint of the top side. Earlier, we defined a constraint such that the top side of the stack view should be 7.33 points away from the margin of the cell.
3. Lastly, it is the spacing constraint of the bottom side. The bottom side should be 7.83 points away from the margin of the cell.

The system complained that it was impossible to satisfy all the above layout constraints simultaneously. In other words, it can't center the stack view, and at the same time, keep the required spacing between the top and bottom sides.

It looks like we have made some mistakes when defining the spacing constraints. If we want to center the stack view, the value of both spacing constraints should be equal. Let's modify the value and see if it will fix the issue.

Go to the `main.storyboard`. In document outline, select the `Stack View.top` constraint. In the Attributes inspector, set the constant to `7`. Next, select the `Stack View.bottom` constraint. Again, set the constant to `7`.

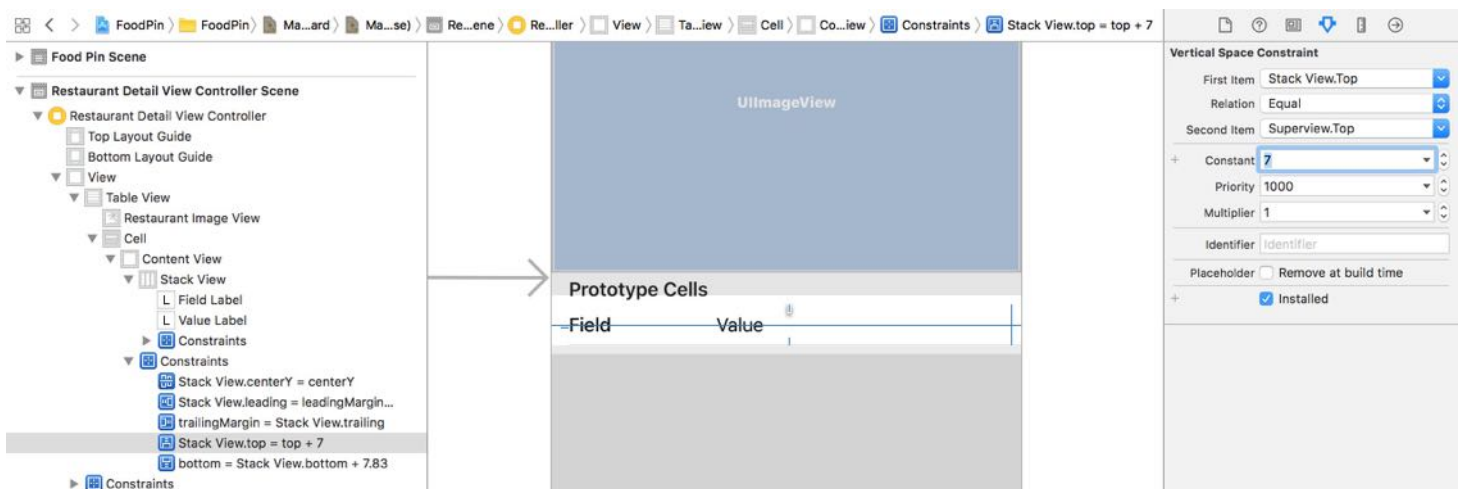


Figure 15-7. Removing the alignment constraint of the stack view

Now run the app again. It should be able to display the address properly without any layout errors.



Figure 15-8. The app can now display the location field properly

Exercise

For the home screen, the cells of the table view have not been converted to self sizing cells. Your exercise is to add the necessary layout constraints (if any) and apply self sizing for the cells. Your resulting screen will be similar to that shown in figure 15-9.

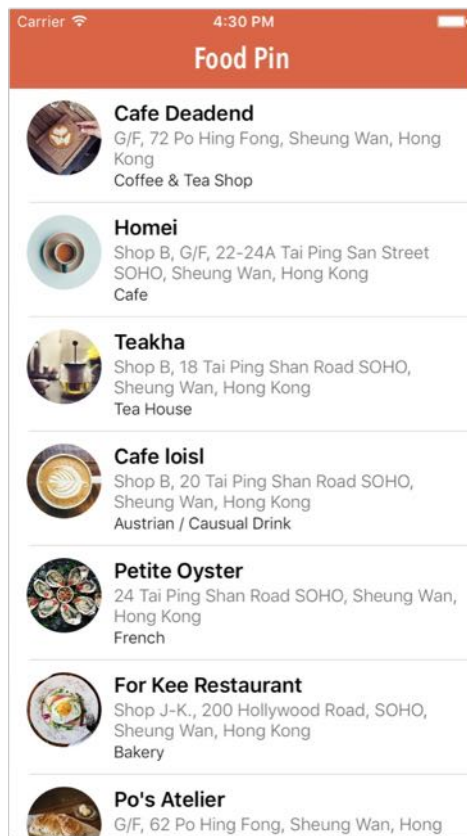


Figure 15-9. Self sizing cells in the home screen

Dynamic Type

Self sizing cells are particularly useful to support Dynamic Type. You may not have heard of Dynamic Type but you should have seen the setting screen (*Settings > General > Accessibility > Larger Text* or *Settings > Display & Brightness > Text Size*) shown in figure 15-10.

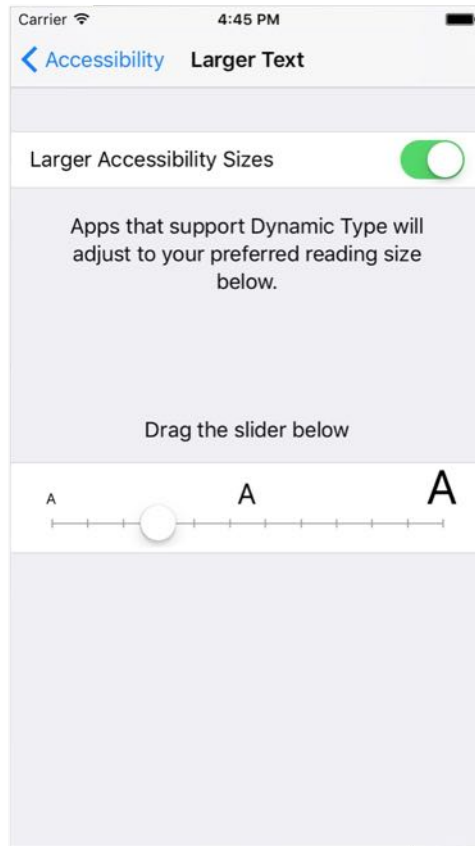


Figure 15-10. Larger Text Setting

Dynamic Type was first introduced in iOS 7 - it allows users to customise the text size to fit their own needs. However, only apps that adopt dynamic type respond to the text change. I believe most of the users are not aware of this feature because only a fraction of third-party apps have adopted the feature.

Starting from iOS 8 and now iOS 10, Apple encourages developers to adopt Dynamic Type, so users can choose their own text size.

So how can you adopt Dynamic Type? Once the cell can be self-sized, adopting Dynamic Type is just a piece of cake. Recalled that we configured the name label in the main screen to use a text style - `Headline` in chapter 9, this is what you need to adopt Dynamic Type - use a text style instead of a fixed font type.

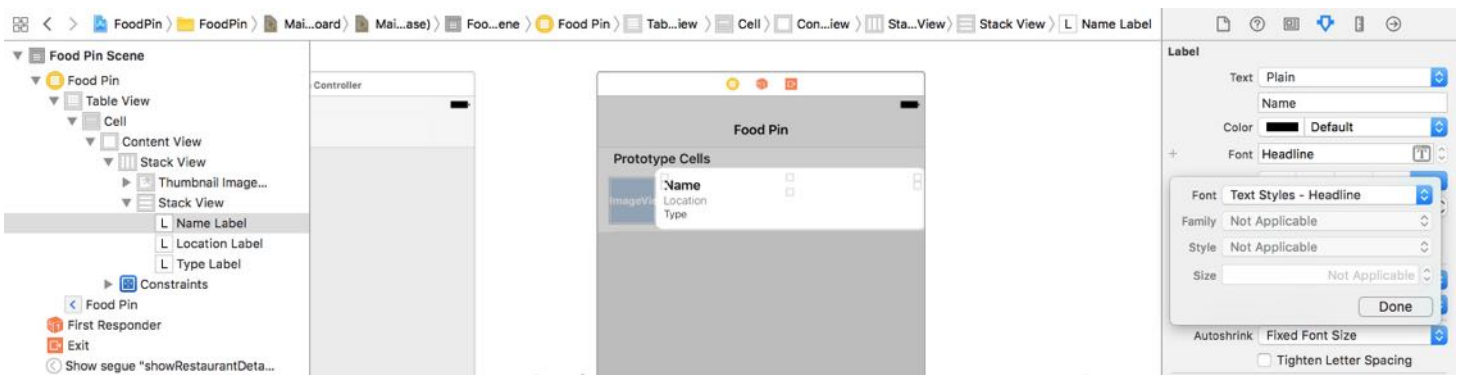


Figure 15-11. The name label is configured with a text style

The rest of the labels are still configured to use a fixed system font. Now let's open up the iPhone simulator and go to Settings > General > Accessibility > Larger Text. Enable *Larger Accessibility Sizes* and drag the slider to your right to enlarge the font.

Run the project in the simulator, the name label should be automatically scaled up. For the rest of the label, their sizes are kept intact because they're using a fixed sized font. In summary, to adopt Dynamic Type, all you need to do is configure the labels with a text style. That's it.

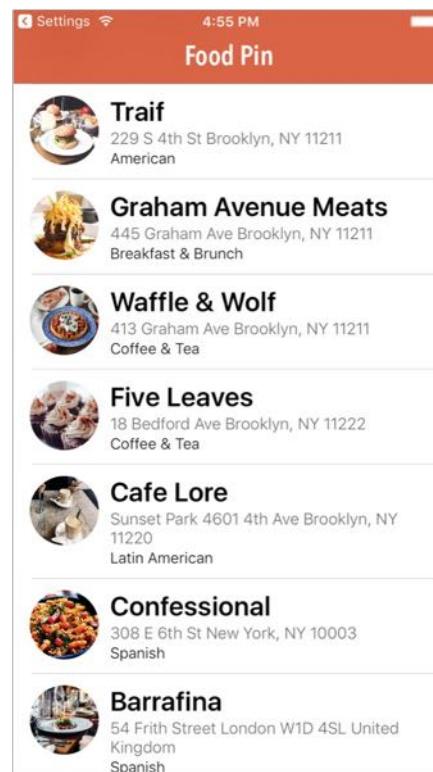
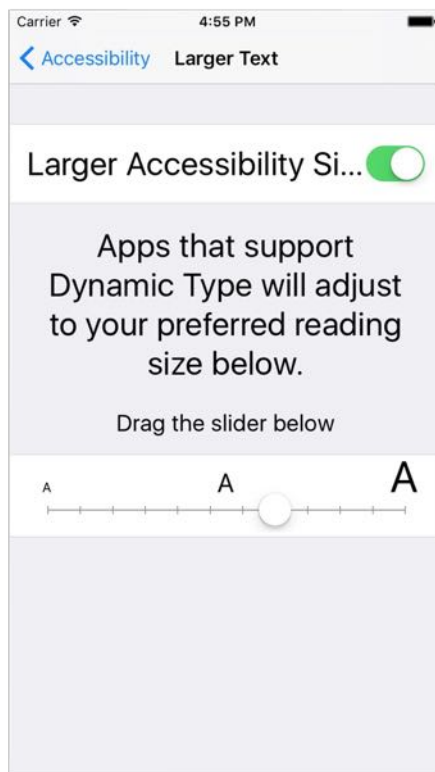


Figure 15-12. Enlarging the font in Settings (left), The name label of the main screen is enlarged according (right)

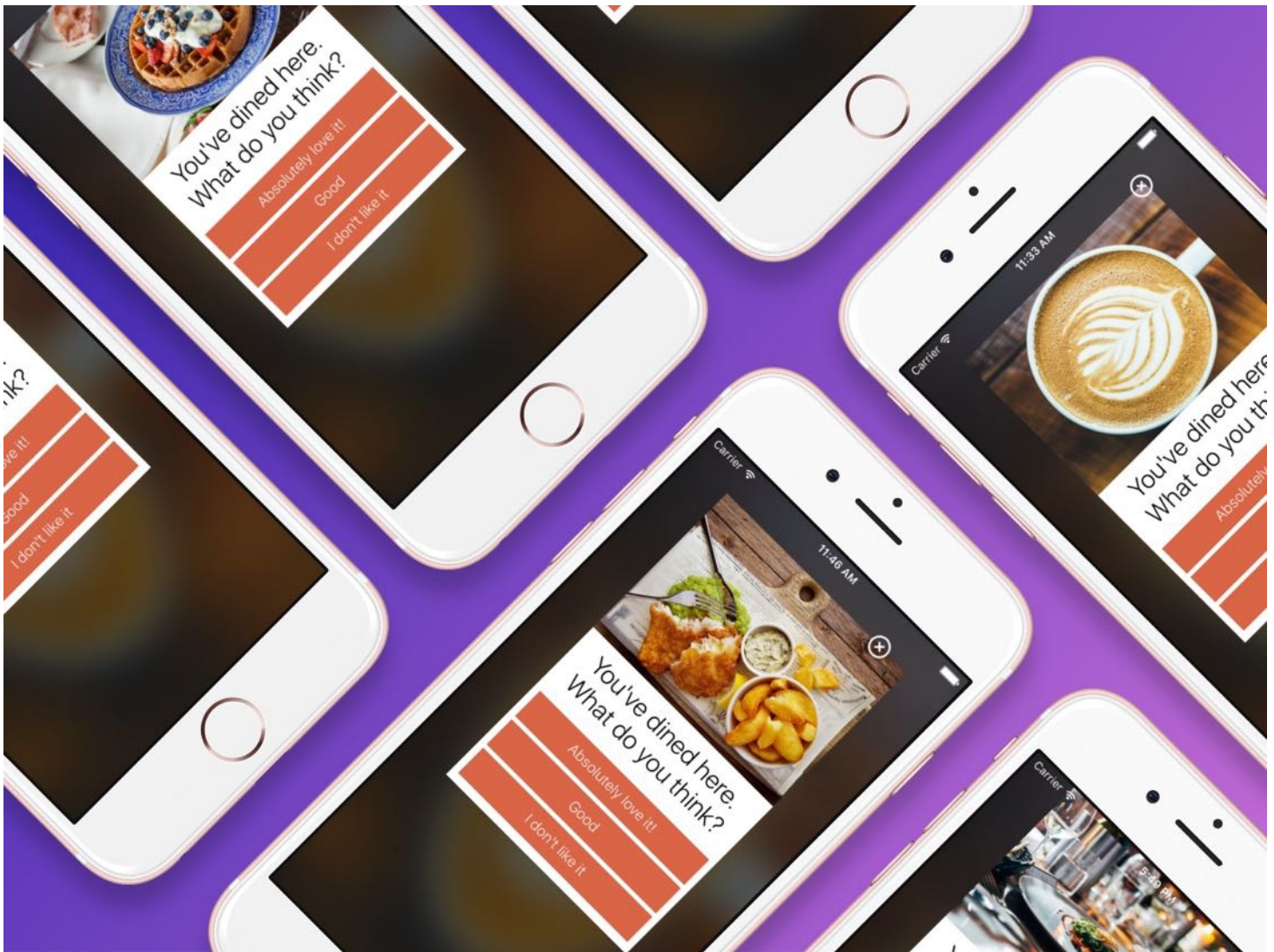
Summary

In this chapter, I walked you through one of the most useful features in the iOS SDK - self sizing cells. By combining auto layout, you just need two lines of code to achieve a dynamic cell layout with variable row height. This definitely makes app development easier to support different device orientation and dimensions. I have also covered Dynamic Type, which allows your users to choose their preferred font size. Once you implement self sizing cells, it's very easy to adopt Dynamic Type. Apple encourages all iOS developers to adopt this technology in their apps. If you're going to develop your next app, consider to adopt Dynamic Type.

For your reference, you can download the Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinSelfSizingCells.zip>.

Chapter 16

Basic Animations, Visual Effects and Unwind Segues



Animation can explain whatever the mind of man can conceive. This facility makes it the most versatile and explicit means of communication yet devised for quick mass appreciation.

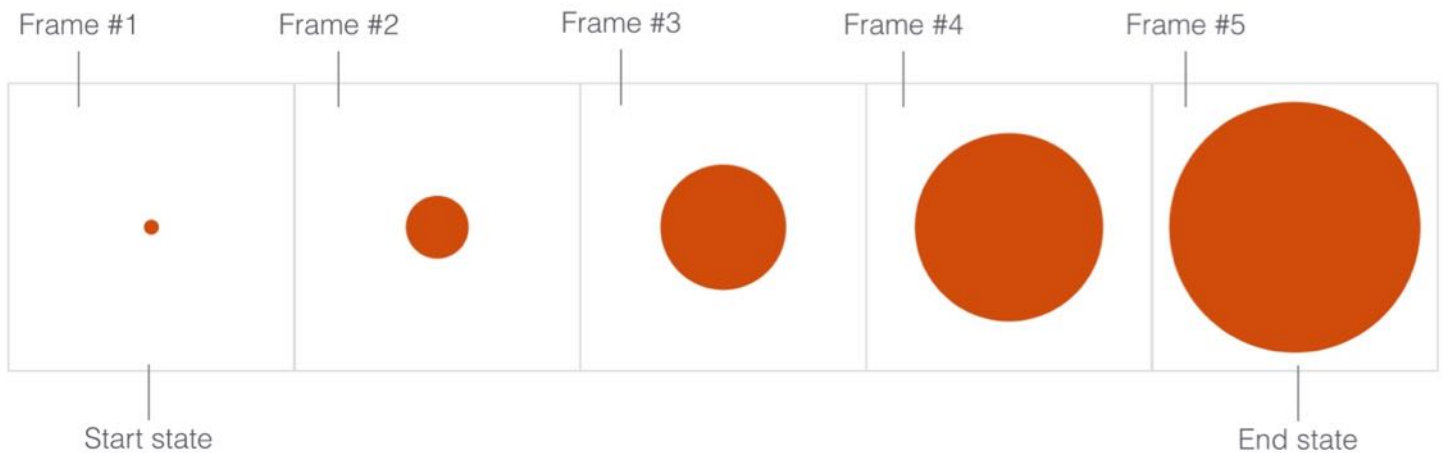
– Walk Disney

In iOS, creating sophisticated animations does not require you to write complex code. All you need to know is a single method in the `UIView` class:

```
UIView.animateWithDuration(1.0, animations)
```

There are several variations of the method that provide additional configuration and features. This is the basis of every view animation.

First things first, what's an animation? How is an animation created? Animation is a simulation of motion and shape change by rapidly displaying a series of static images (or frames). It is an illusion that an object is moving or changing in size. For instance, a growing circle animation is actually created by displaying a sequence of frames. It starts with a dot. The circle in each frame is a bit larger than the one before it. This creates an illusion that the dot grows bigger and bigger. Figure 16-1 illustrates the sequences of static images. I keep the example simple so the figure displays 5 frames. To achieve a smooth transition and animation, you'd need to develop several more frames.



Now that you have a basic idea of how animation works, how will you create an animation in iOS? Consider our growing circle example. You know the animation starts with a dot (i.e. start state) and ends with a big red circle (i.e. end state). The challenge is to generate the frames between these states. You may need to think of an algorithm and write hundreds of lines of code to generate the series of frames in between. `UIView` animation helps you compute the frames between the start and end state resulting in a smooth animation. You simply specify the start state and tell `UIView` the end state by calling the `UIView.animateWithDuration` method. The rest is handled by iOS. Sounds good, right?

There is no better way to understand the technique than by working on a real example. We will add some basic animations to our FoodPin app. Here is what we're going to do:

- Add a check-in button in the detail view
- When a user taps the button, it brings up a review view controller, in which the buttons are animated, for the user to rate the restaurant.

Through building the review view controller, I will show you how to create basic animations using `UIView`.

Adding a Rating button

Before creating the animated views, we'll add a rating button to the detail view controller.

First, download this image pack

(<http://www.appcoda.com/resources/swift3/FoodPinButtons.zip>) and add the icons to

`Assets.xcassets`.

Credit: The icons are made by [Cosmin Negoita](#) and delivered through geticonjar.com.

Go to `Main.storyboard` and drag a Button object from the Object library to the detail view controller. Place it at the top-right corner of the image view. In the Attributes inspector, set the title to blank. Change the *image* option to `check` and its type to `System`. Next, scroll down to set *tint* to `white`. By changing the button type to *System*, you can alter the value of tint to change the button's color.

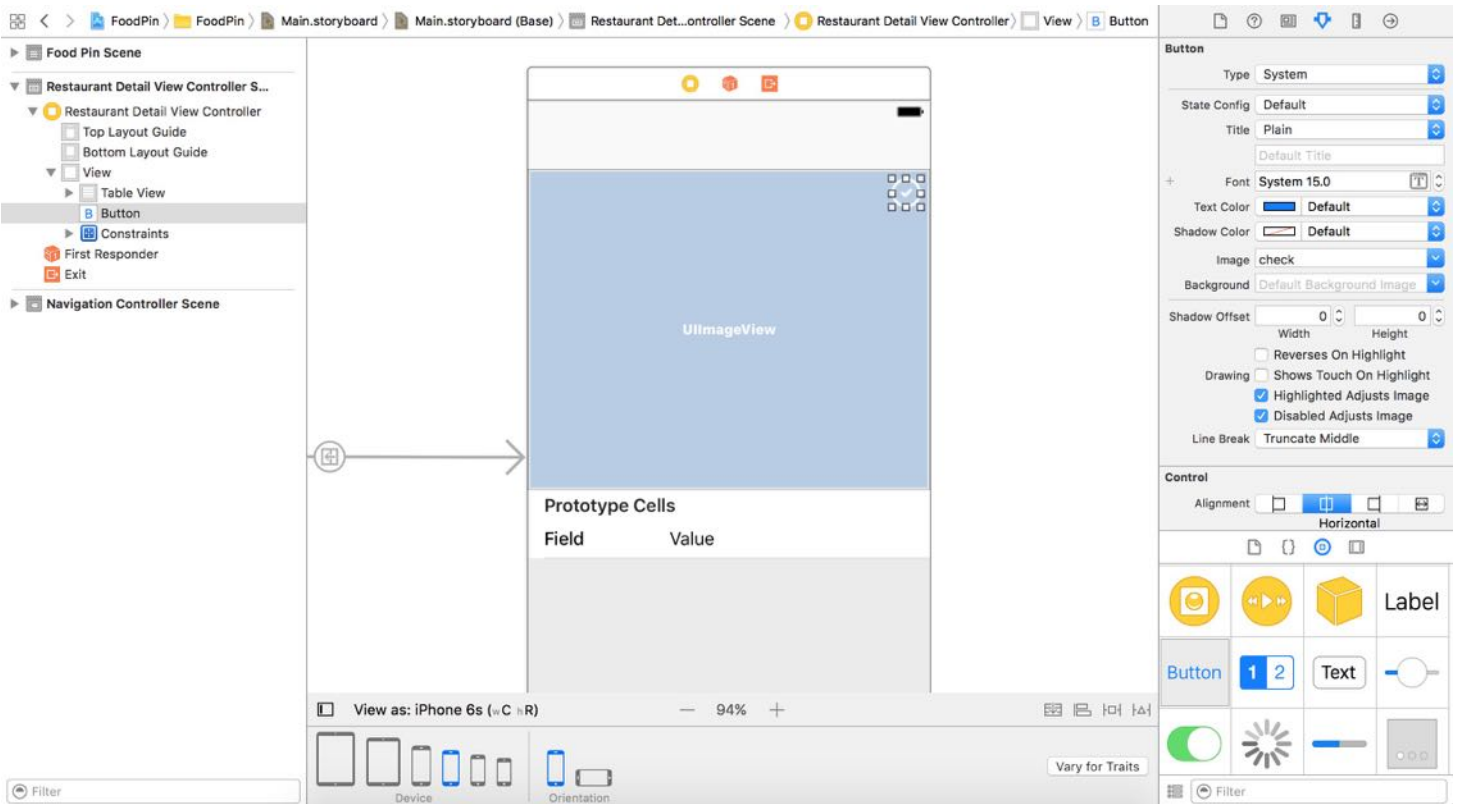


Figure 16-2. Adding a review button in the detail view

As usual, we need to add some layout constraints for the button. Select the button and click the Pin button in the layout bar. Refer to figure 16-3 to add four layout constraints (two spacing constraints for the top & right sides and two constraints to control the width & height of the button).

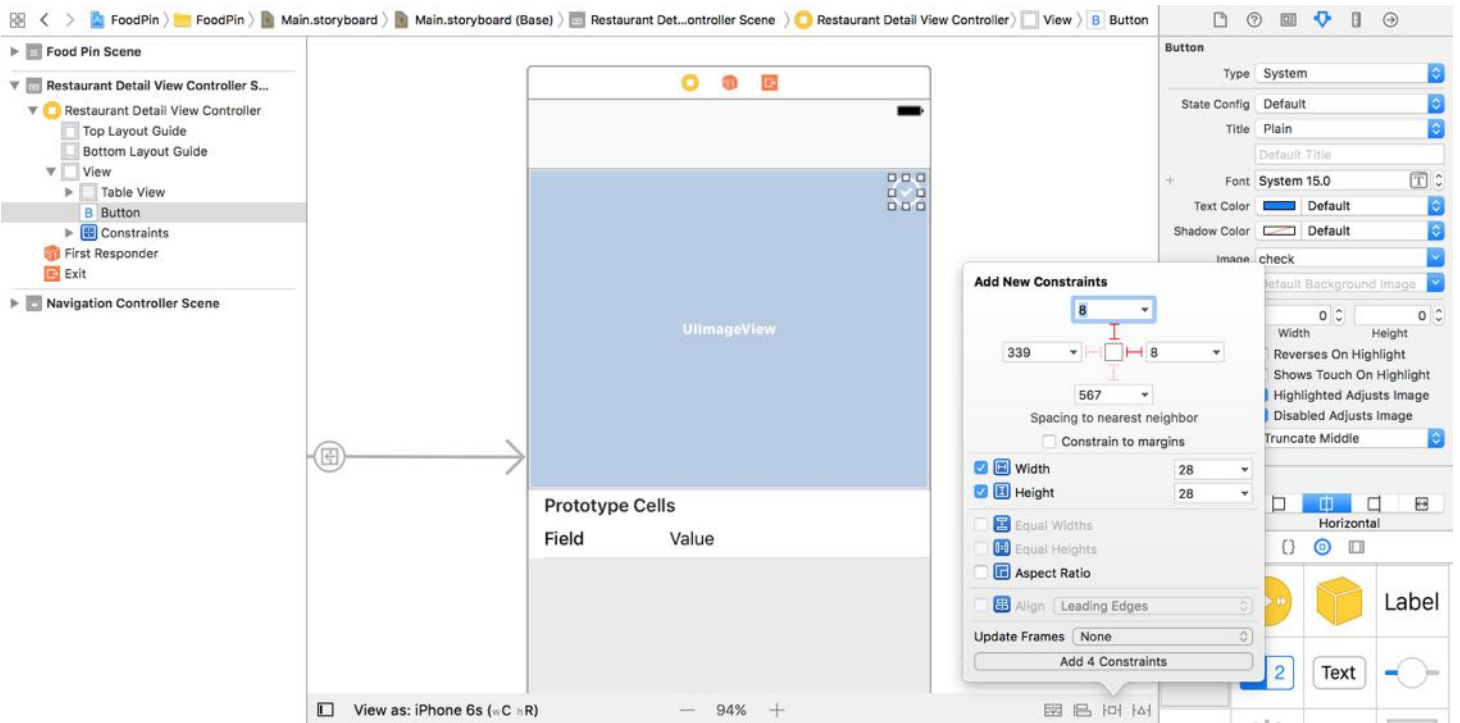


Figure 16-3. Adding layout constraints for the button

Once done, run the app to have a quick test. Your detail view should look very similar to that in figure 16-4.

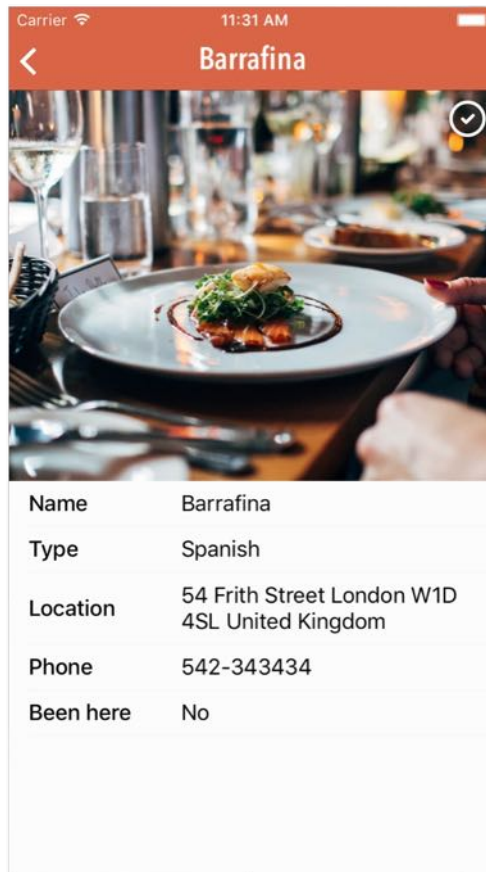


Figure 16-4. The check-in button in the detail view

Create a View Controller for Restaurant Review

When a user taps the *Check-in* button, we want to bring up a modal view for the user to give a rating for the restaurant. In Interface Builder, drag a new view controller from the Object library to the storyboard. Add an image view to the view and set the image to `cafe1ois1`. Then select the "Resolve Auto Layout Issues" button in the layout bar and choose *Add Missing Constraints*. Xcode automatically adds the layout constraints for you.

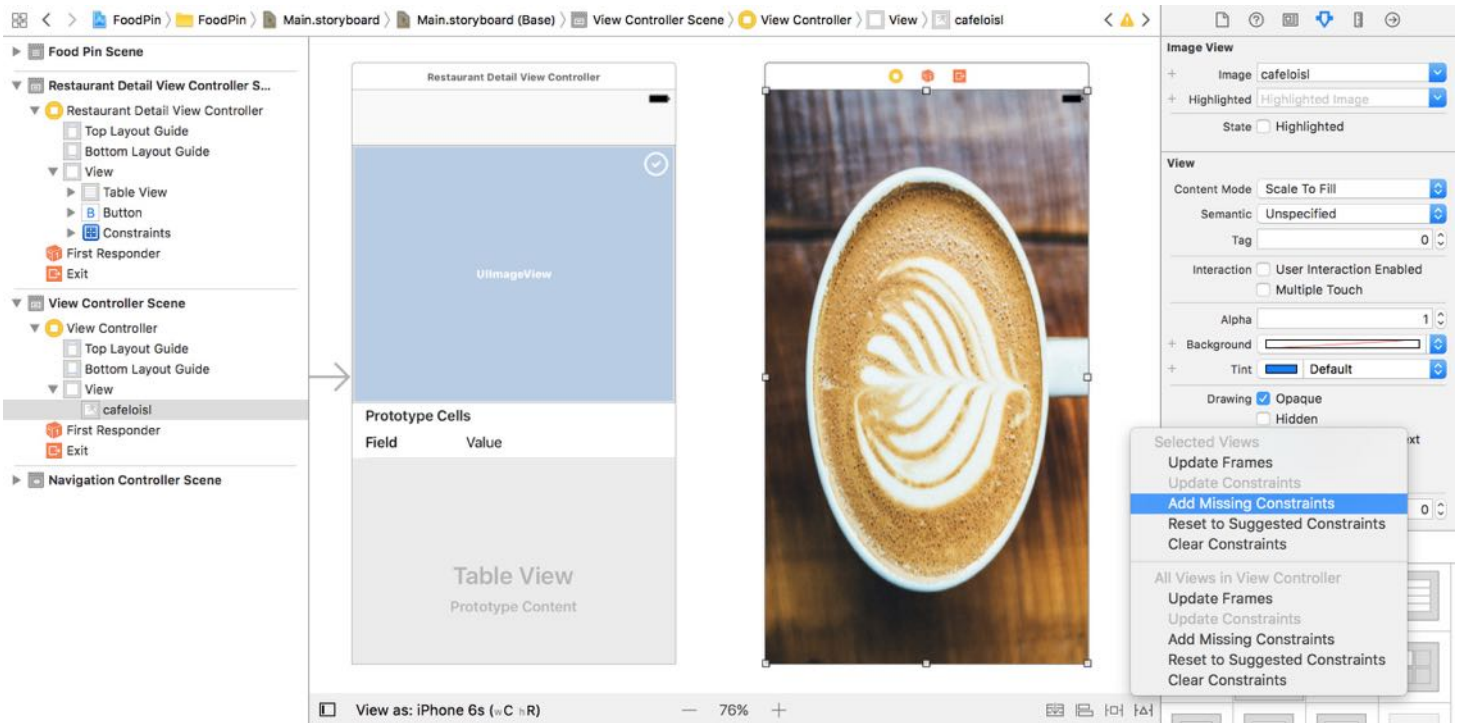


Figure 16-5. Adding missing constraints

Quick note: You may wonder why I do not show you this amazing option for defining layout constraints. First, I want you to understand auto layout thoroughly, so you know how to define the constraints. Secondly, the "Add Missing Constraints" option doesn't always work. It mostly works on simple layouts. But for more complicated ones, you have to handle it on your own.

This image is used as the background image of the view. Later I'll show you how to apply a blurring effect to the image.

Now, we will design the UI of the review view. The resulting view is displayed in figure 16-6.

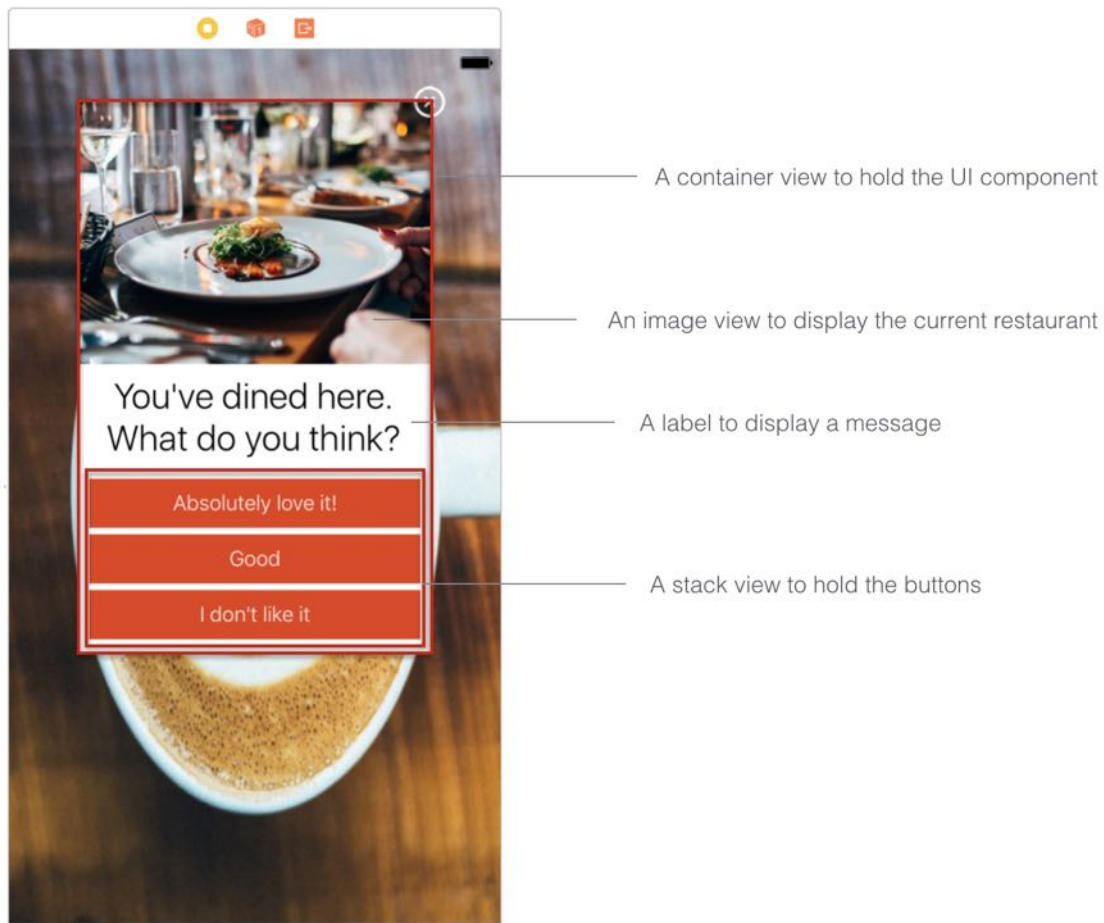


Figure 16-6. UI design of the review view

First, drag a view object to the image view. In the Size inspector, set X to 53 , Y to 40 , width to 269 and height to 420 . This view serves as a container view for holding other views and UI components.

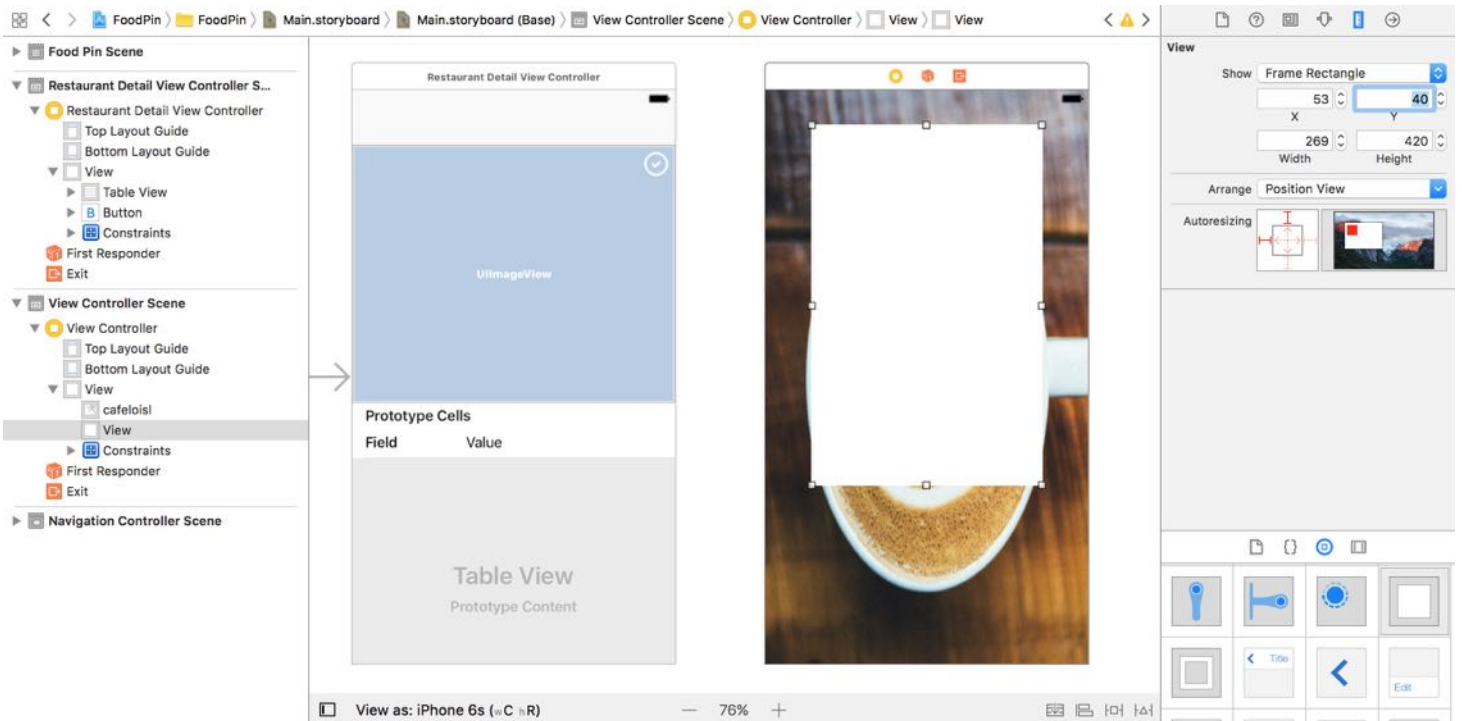


Figure 16-7. Adding a view to the view controller

Next, drag an image view from the Object library and put it inside the view you just created. Change its width and height to 269 and 200 points respectively. Set the image to `barrafina` (or other restaurant image).

Then add a label to the view and put it right below the image. Change the text to `You've dined here. What do you think? .` Set the font style to *Light* and size to 27 points. Furthermore, change the alignment option to *center*, and set the lines option to 2. If you've made everything right, your UI should look similar to figure 16-8.

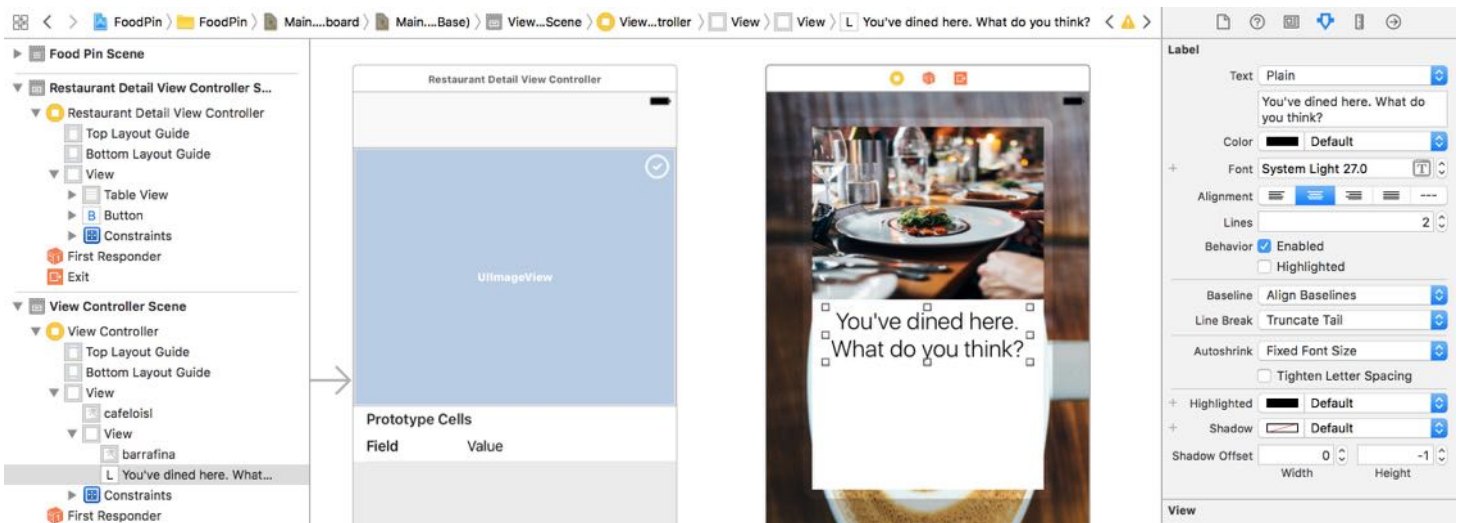


Figure 16-8. Adding an image view and a label to the container view

Next, drag three buttons to the image view and change the title for each of the image:

- Set the title of the first button to `Absolutely love it!`.
- Set the title of the second button to `Good`.
- Set the title of the third button to `I don't like it`.

You can change the background to `red` and set the tint to `white`. Set the font style to *Light* and size to `16` points. Your buttons should be similar to that in figure 16-9.

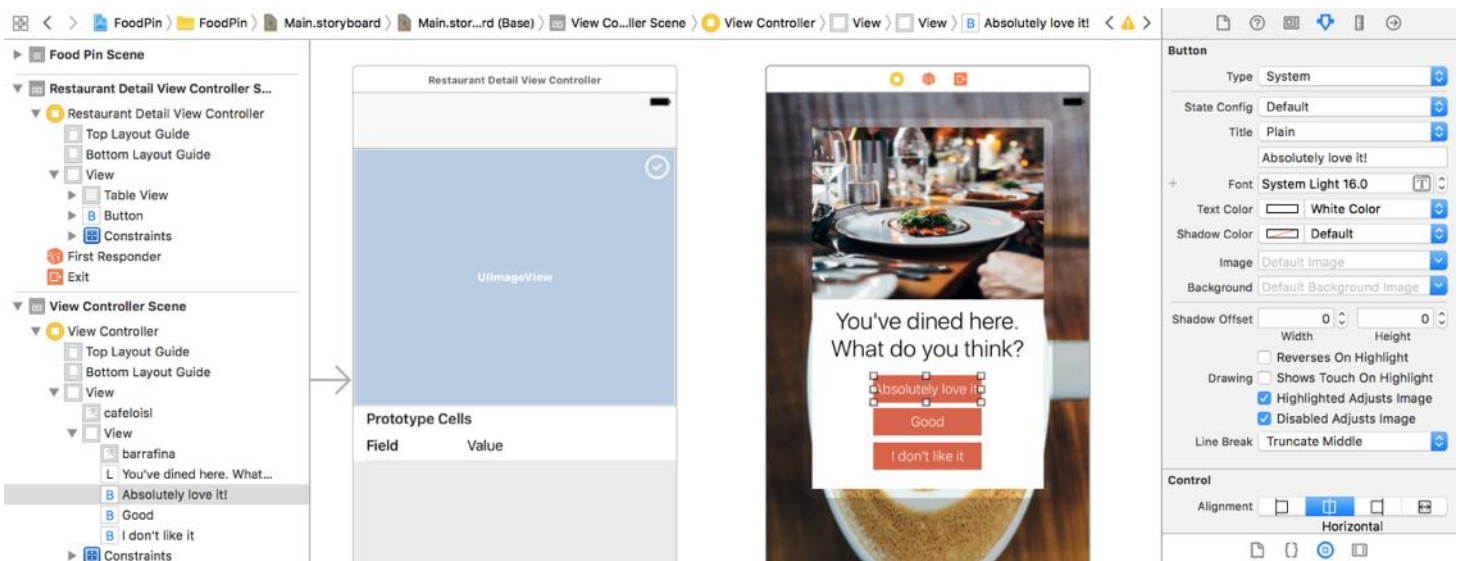


Figure 16-9. Rating buttons

Again, it's time to define the layout constraints for the UI objects.

First, let's embed the buttons in a stack view, and then define the necessary constraints. Select the three buttons and click the *Stack* button in the layout bar to embed them in a vertical stack view. In the Attributes inspector, change the distribution option to *Fill Equally* and spacing to 5 points. The stack view tries its best to fit all the buttons. Meanwhile, it doesn't look good, as compared to the ones shown in figure 16-6. No worries. It'll get better after we define the layout constraints for the stack view and the rest of the components.

Let's first start with the view. Select the view, and click the Pin button. Set the spacing of the top, left and right sides to 20 , 37 and 37 points. We also want to fix the height of the view. So tick the height checkbox. Then click `Add 4 Constraints` to add the constraints.

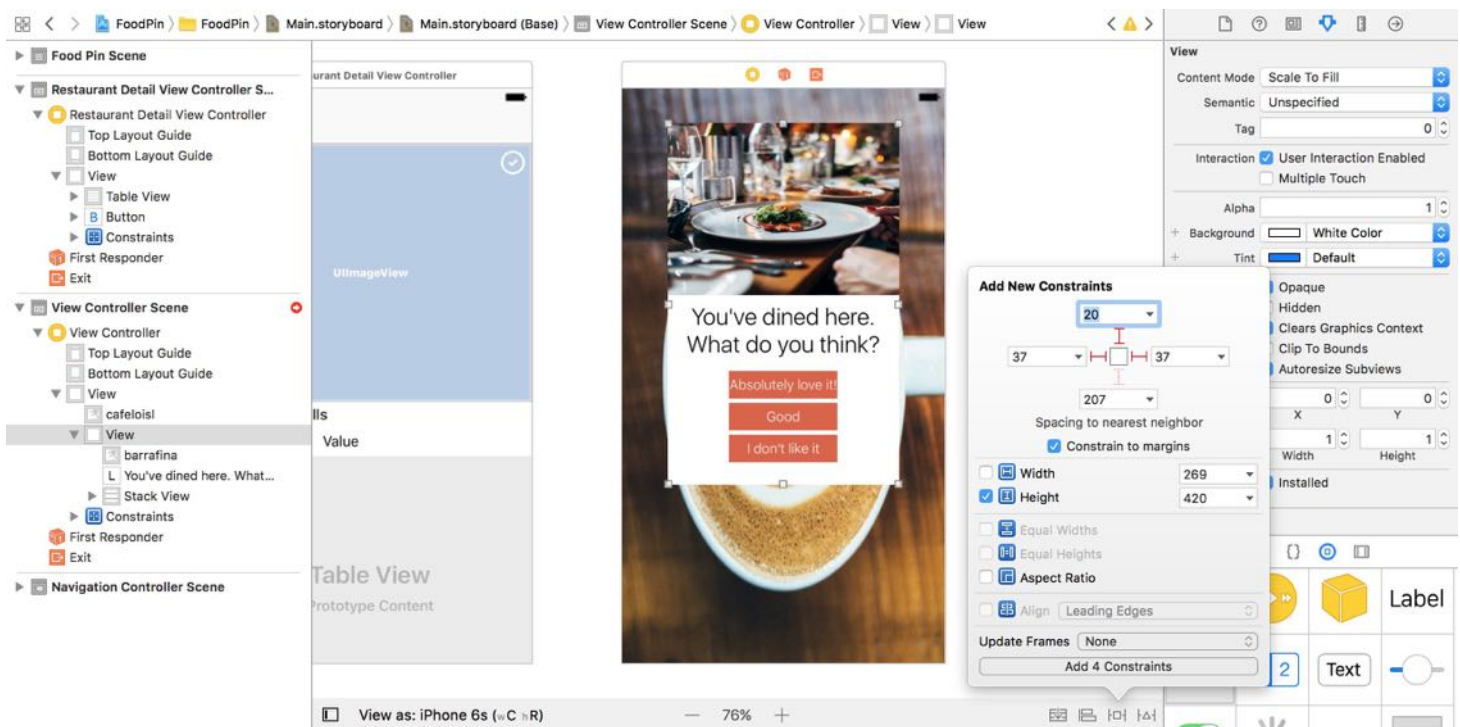


Figure 16-10. Adding layout constraints for the container view

Next, select the image view and click the Pin button. Refer to figure 16-11 to assign the layout constraints.

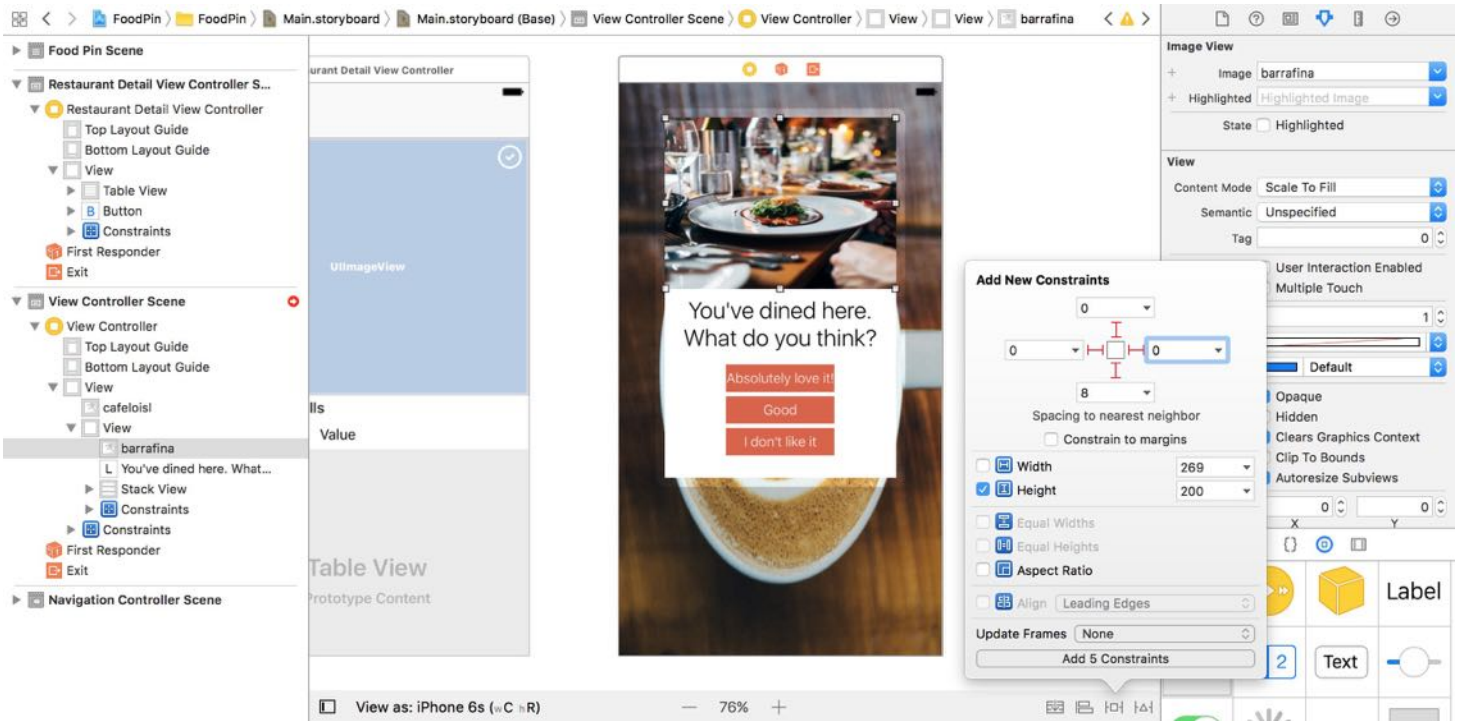


Figure 16-11. Layout constraints for the image view

Now let's move onto the label. Select it and click the Pin button to add three spacing constraints. For the left and right sides of the label, they should be 15 points away from the edge of the view. For the bottom side, it should be 15 points away from the stack view.

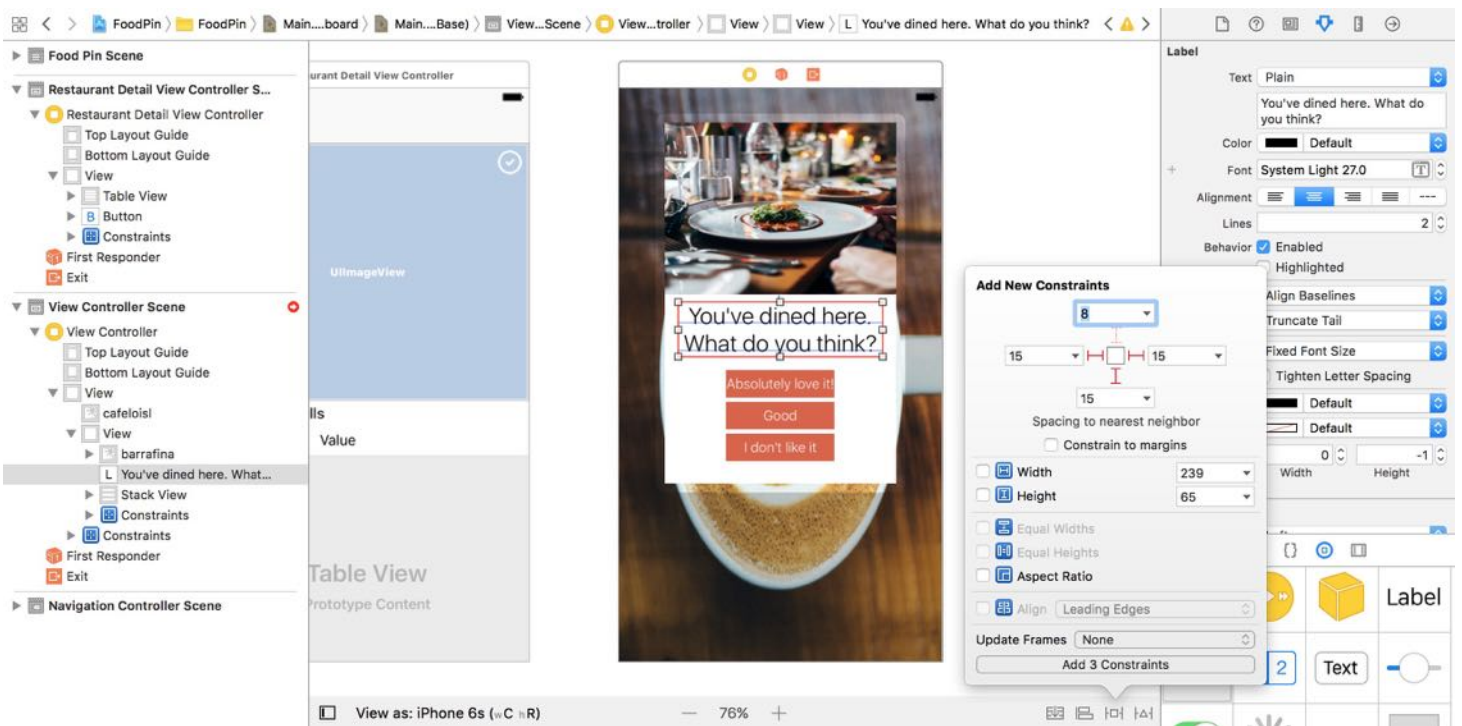


Figure 16-12. The resulting stack view

Lastly, let's define the required layout constraints for the stack view. Set the spacing value of the left, right and bottom sides to 8 , 8 , and 10 points respectively. After adding the constraints, you will see a yellow indicator in the document outline. Click it and follow the instructions to update the frame position.

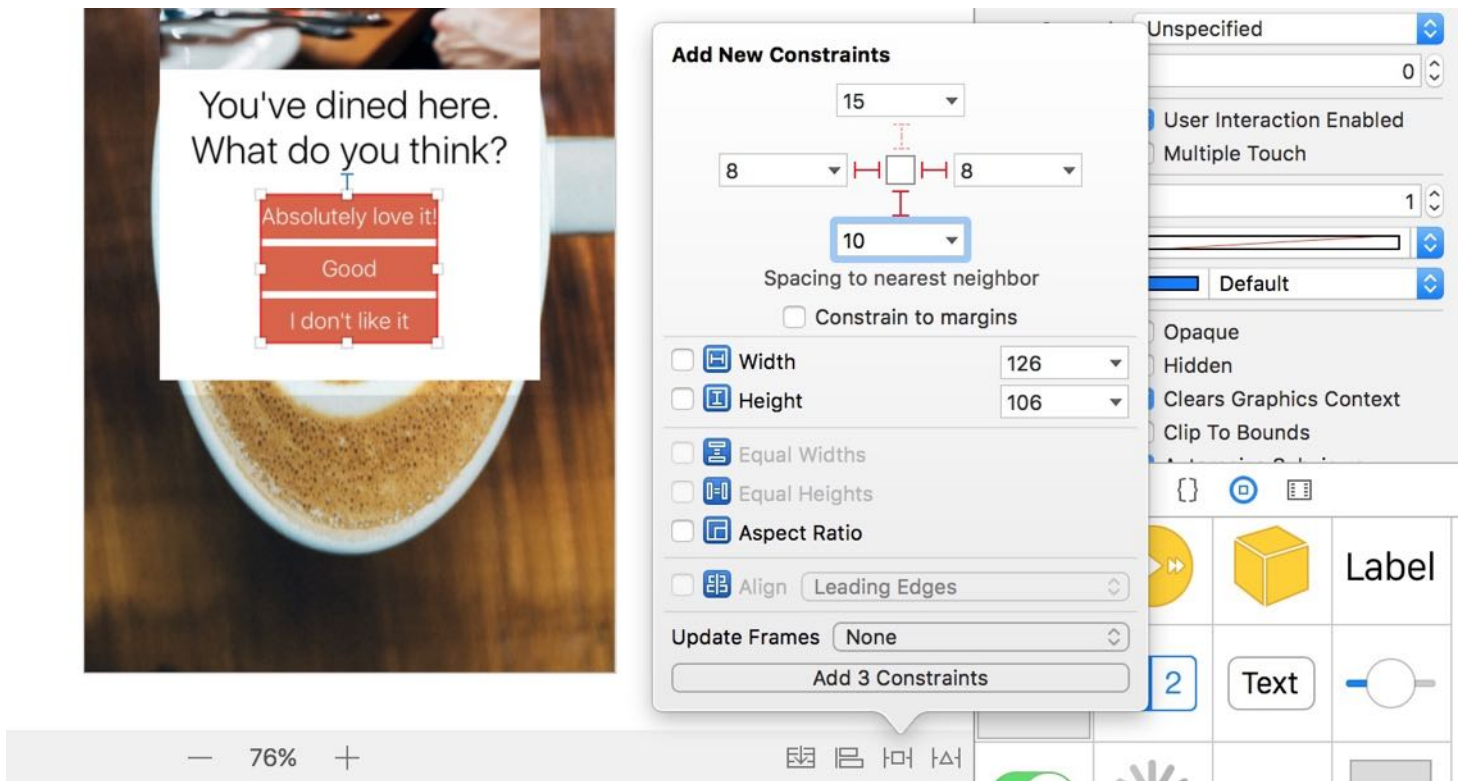


Figure 16-13. Adding layout constraints for the close button

That's it. You've defined the layout constraints.

Create a Segue for the Modal View

To bring up the review view modally, we have to connect the *Check-in* button with the review view controller with a segue. Hold the control key and drag from the *Check-in* button to the review view controller. Release the buttons and select *Present modally* as the segue type. Once the segue is created, select it and set the identifier of the segue to `showReview` under the Attribute inspector.

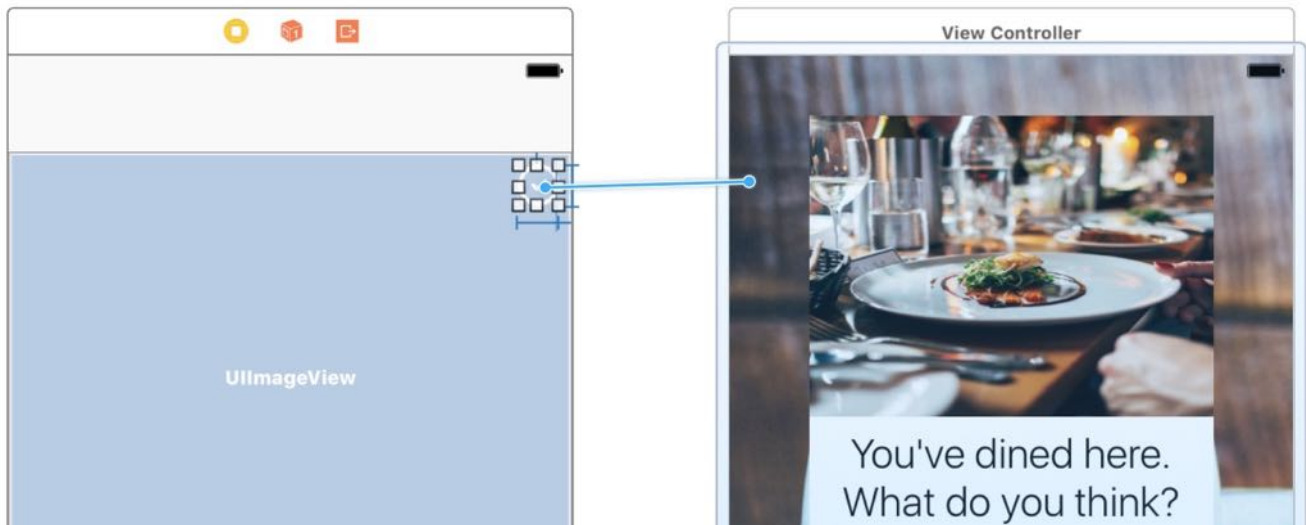


Figure 16-14. Create a segue to present the review view controller modally

Now let's have a quick test. Run the app and go to the detail view. Tapping the review button should now bring up the review view.

Defining an Exit for the Review View Controller

Presently, there is no way to dismiss the view to return to the previous screen (i.e. detail view controller). What we are going to do is to define a so-called *unwind segue*. An unwind segue can be used to navigate back through a modal or push segue. In this example, we can use it to dismiss the modal view.

Before we create the unwind segue, let's first add a close button. Drag a button from the Object library to the view. Make sure you put the button inside the container view, and place it at the top-right corner. In the Attributes inspector, set the button title to *blank* and image to *cross*. Similarly, you will need to a few layout constraints. Click the Pin button and refer to figure 16-15 to set the constraints accordingly.

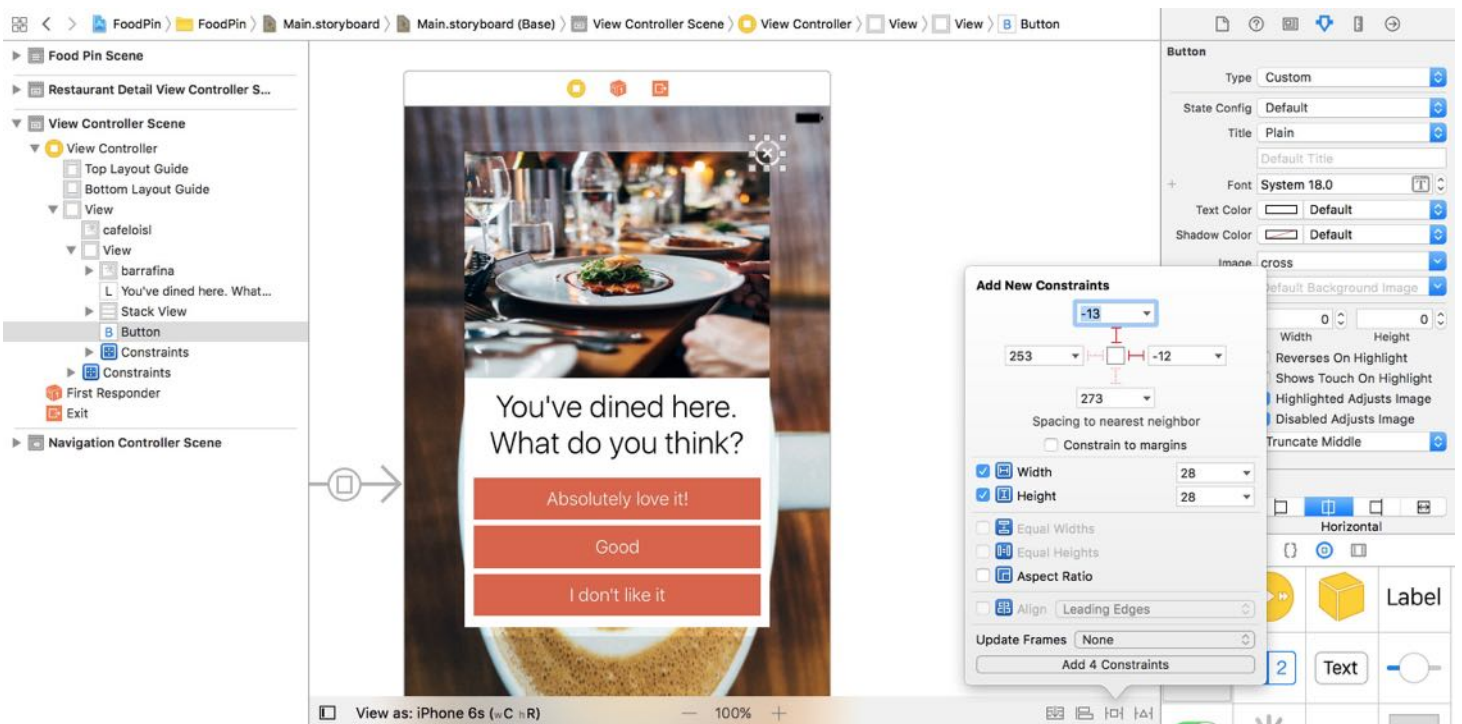


Figure 16-15. Adding a close button

Note: If you have any problems designing the UI, download this Xcode project (<http://www.appcoda.com/resources/swift3/FoodPinReviewUI.zip>) to take a look.

To use an unwind segue, you need to do two things. First, declare a method in the destination view controller. In this case, it is the `RestaurantDetailViewController` class. Add the following method in `RestaurantDetailViewController.swift`:

```
@IBAction func close(segue:UIStoryboardSegue) {
}
```

Before you can begin adding unwind segues in Interface Builder, you must define at least one unwind action. This action method tells Xcode that it can be unwound. Optionally, you can implement additional logic in the method. Meanwhile, an empty method is good enough for us to configure an unwind segue. Go back to Interface Builder. Press and hold control key and drag from the *Close* button to the Exit icon of the scene dock (see figure 16-16). When prompted, select `closeWithSegue:` for the action segue.

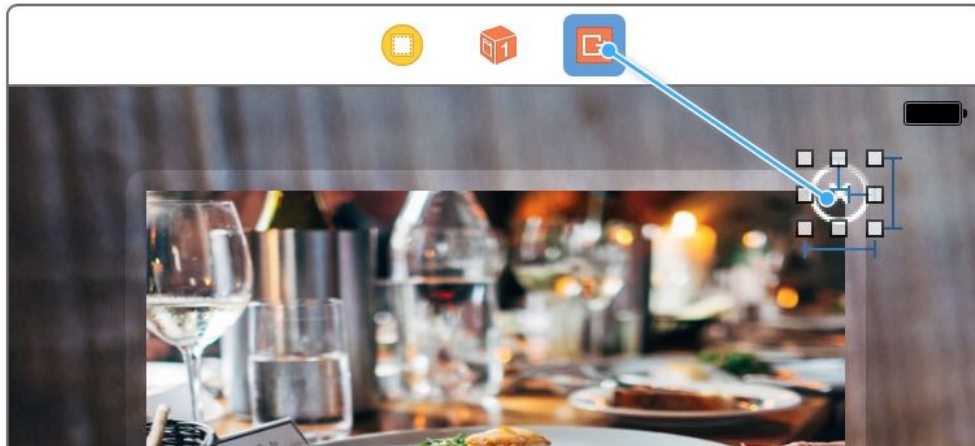


Figure 16-16. Adding an unwind segue for the close button

You can run the project again. Now when the user taps the close button, the modal view will be dismissed.

Applying a Blurring Effect to the Background Image

In iOS 8, Apple introduced a new class called `UIVisualEffectView` that lets developers apply visual effects to a view. Combining with the `UIBlurEffect` class, you can easily apply a blurring effect to an image view.

Now let's see how we can blur the background image. First, create a new class for the review view controller. In the project navigator, right click the `FoodPin` folder and select "New File...". Select the *Cocoa Touch Class* template to proceed. Name the class `ReviewViewController` and set it as a subclass of `UIViewController`. Then save the file in the `FoodPin` folder.

Select the `ReviewViewController.swift` file you've just created. Because we are going to apply a blurring effect to the image view, let's add an outlet variable for it:

```
@IBOutlet var backgroundImageView: UIImageView!
```

In the `viewDidLoad` method, add the following code:

```
let blurEffect = UIBlurEffect(style: UIBlurEffectStyle.dark)
let blurEffectView = UIVisualEffectView(effect: blurEffect)
```



```
blurEffectView.frame = view.bounds
backgroundImageView.addSubview(blurEffectView)
```

To apply a blurring effect to the background image view, all you need to do is create a `UIVisualEffectView` object with the blurring effect, followed by adding the visual effect view to the background image view. The `UIBlurEffect` class offers three different styles: *dark*, *light*, and *extra light*. I like the dark style but it is up to you to choose your own style. The above lines of code are all you need to blur the background image.

Now, go to Interface Builder and select the review view controller. In the Identify inspector, set the custom class to `ReviewViewController`. Finally, establish a connection between the background image view and the `backgroundImageView` outlet.

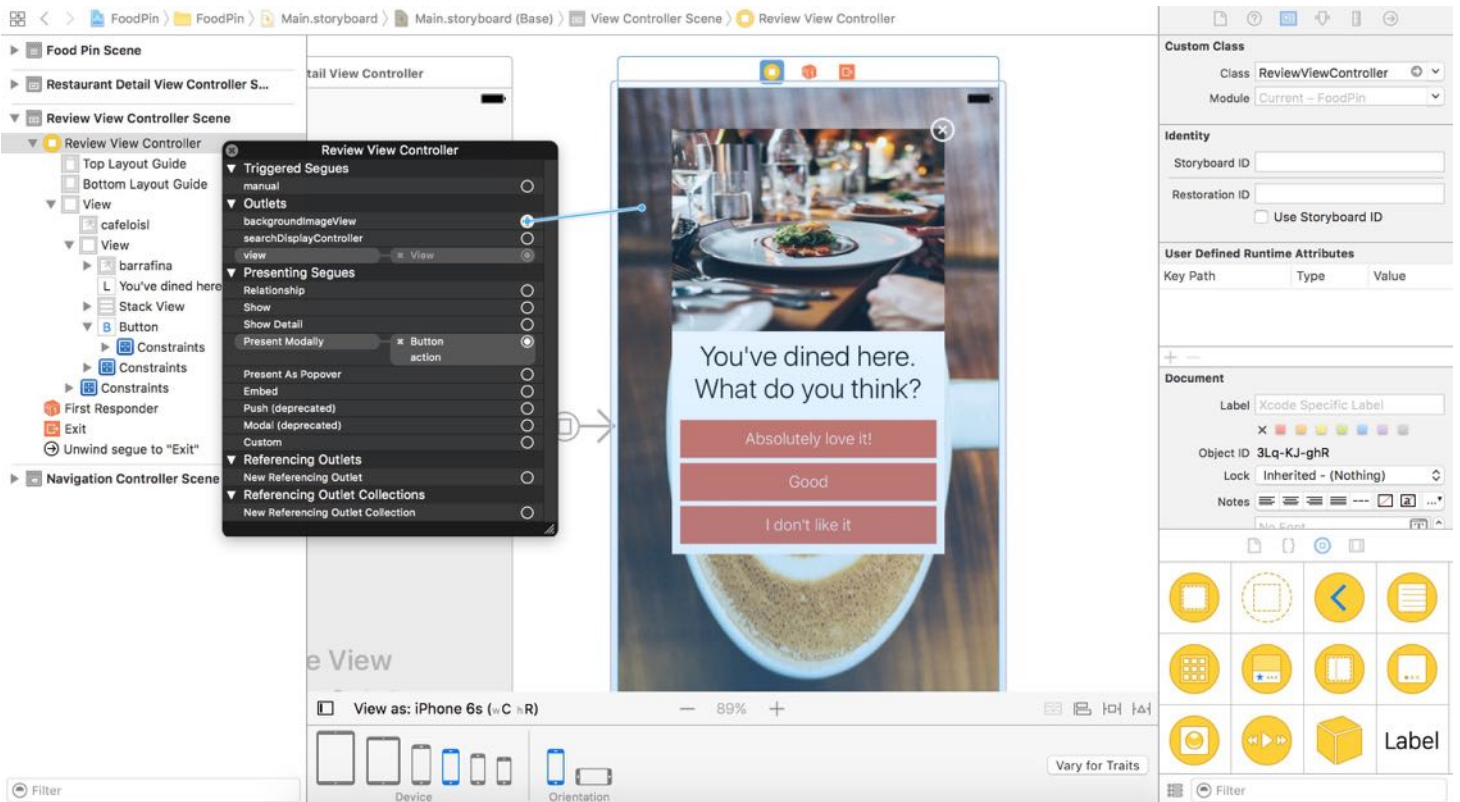


Figure 16-17. Establishing a connection between the outlet variable and the image view

You're ready to go. Figure 16-18 displays the resulting screen, depending on your choice of blur style.

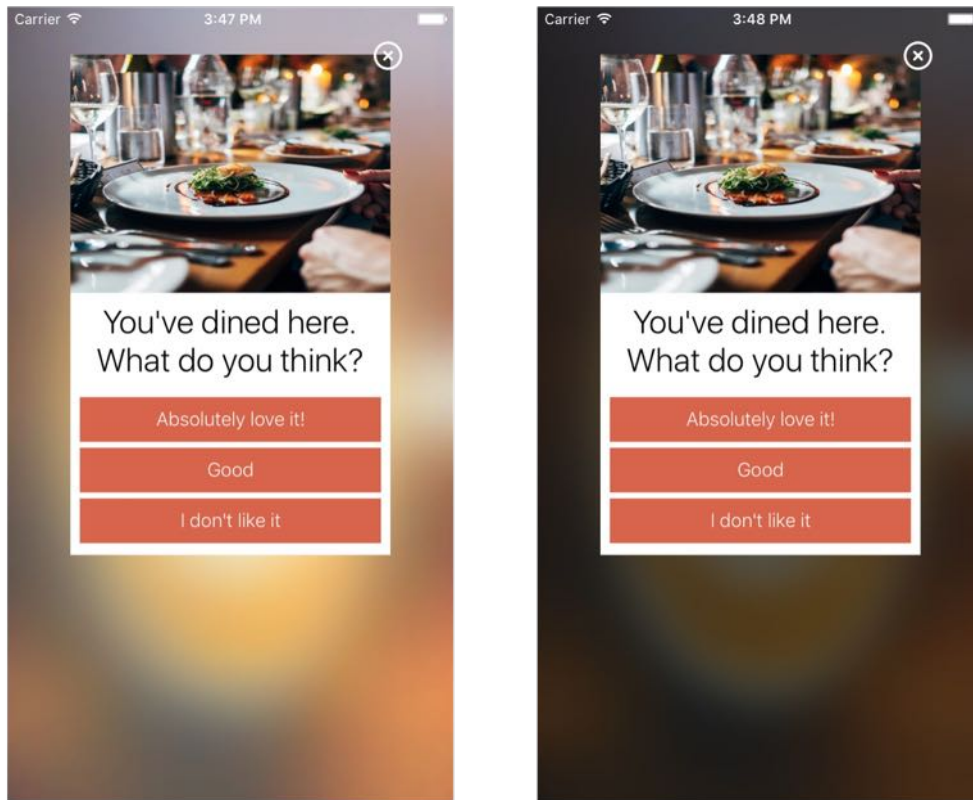


Figure 16-18. Light blurring style (left), Dark blurring style (right)

Animating Dialog View Using UIView Animation

After all the preparation, finally we come to the core part of the chapter: *UIView animation*. We're going to add a growing animation for the container view. The growing effect is very similar to the growing circle animation we talked about earlier. The container view does not appear when the modal view is first displayed, but the dialog view starts to grow till it reaches its actual size. Figure 16-19 gives you a better idea of the animation.

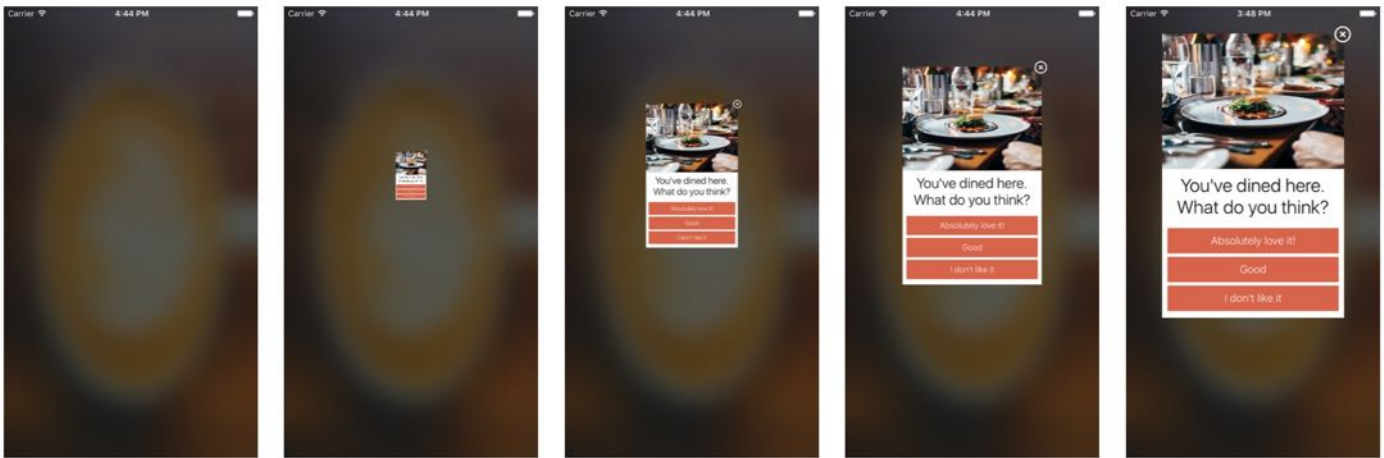


Figure 16-19. Growing animation

As explained before, you just need to provide two states (start and end state) of an animation. `UIView` will then generate the animation for you. For the growing animation, these are the start and end states:

- Start state - the container view is in zero size
- End state - the container view is in its regular size

At this point you might be trying to figure out how to resize a view. iOS provides a structure called `CGAffineTransform` for you to scale, rotate, and move a view. To scale a view, all you need to do is create an affine transformation and set it to the `transform` property of a `UIView` object. Here is the line of code for creating a scaling transform that can be used to change a view to zero size:

```
CGAffineTransform.init(scaleX: 0, y: 0)
```

Since we're going to manipulate the container view, the very first thing is to establish the connection. In the `ReviewViewController` class, add another outlet variable for the container view:

```
@IBOutlet var containerView: UIView!
```

Go back to Interface Builder and establish the connection between the container view and the outlet variable.

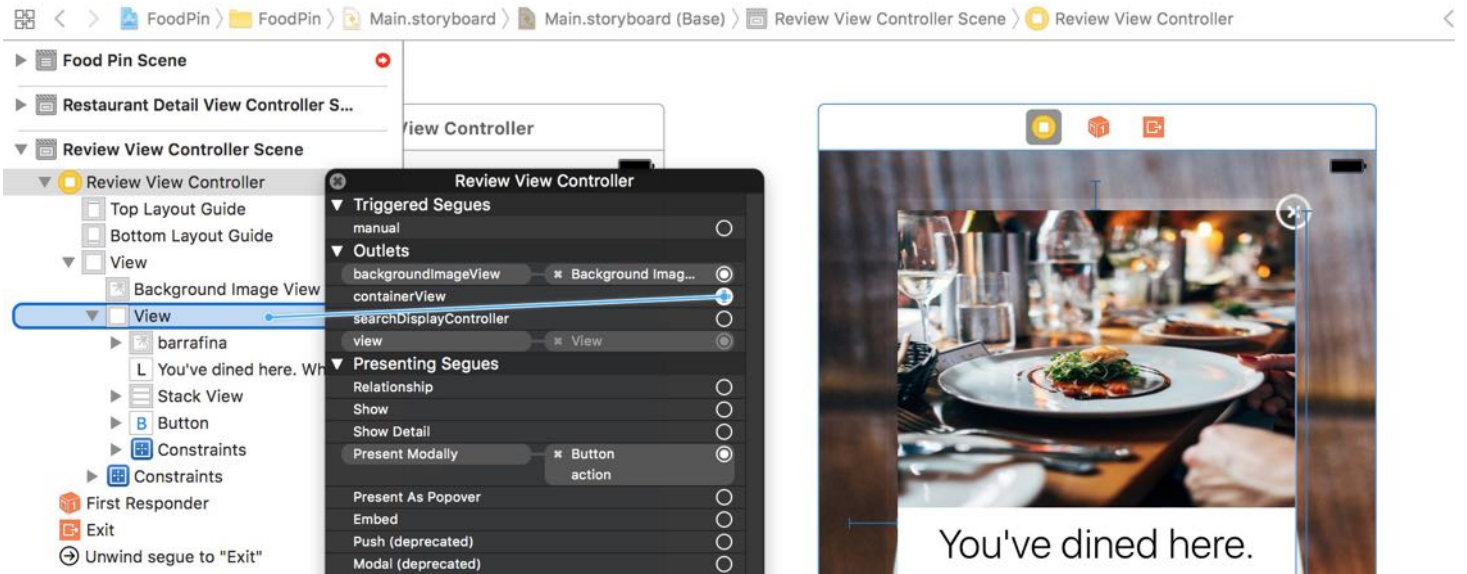


Figure 16-20. Establish a connection between the stack view and the outlet variable

In the `viewDidLoad` method of the `ReviewViewController` class, add the following line of code to set the initial state of the container view:

```
containerView.transform = CGAffineTransform.init(scaleX: 0, y: 0)
```

This scales down the container view when it is first loaded. To create the growing effect, we use the `UIView.animate(withDuration:animations:)` method to animate the size change. Insert the following code in `ReviewViewController`:

```
override func viewWillAppear(_ animated: Bool) {
    UIView.animate(withDuration: 0.3, animations: {
        self.containerView.transform = CGAffineTransform.identity
    })
}
```

The duration of the animation is set to 0.3 seconds. In the animations closure, we define the final state of the dialog view. In this case, we simply scale it to the original size.

`CGAffineTransform.identity` is a constant for resetting a view to its original size and position. Figure 16-21 depicts how this code works.

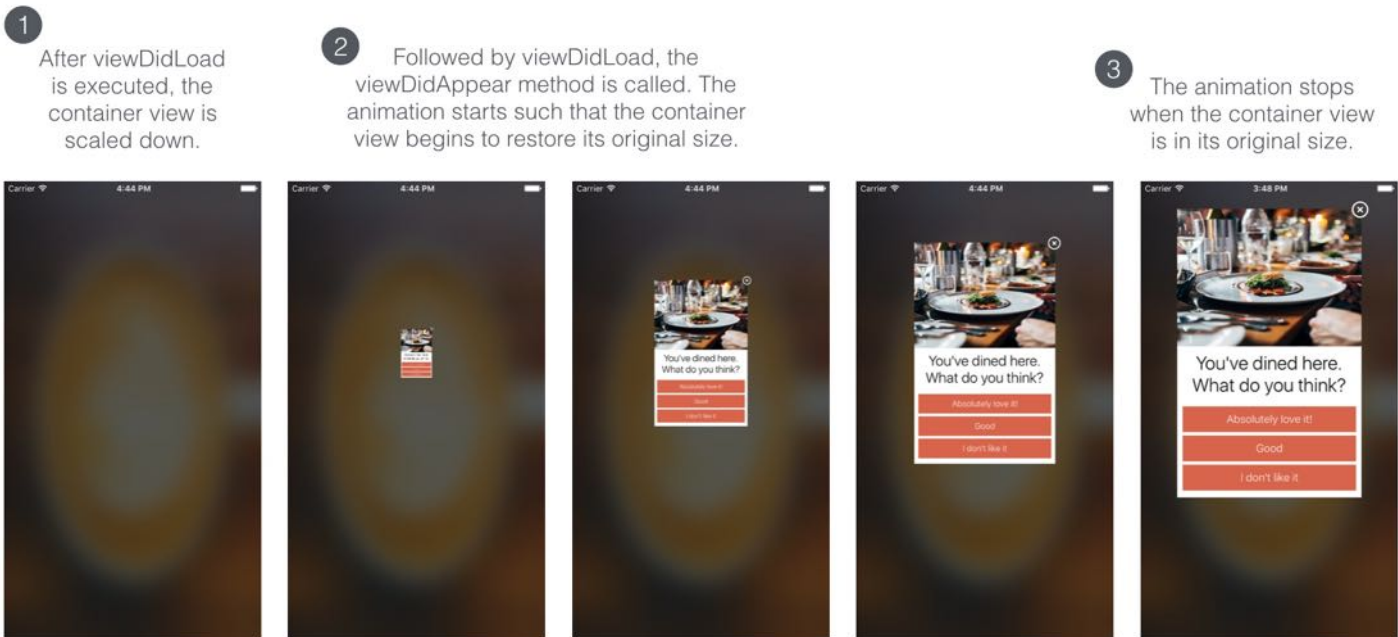


Figure 16-21. How the growing animation works

That's it. `UIView` automatically renders the animations. Run the app for a quick test and enjoy the animation.

Quick note: You can change the duration parameter from `0.3` to other value (e.g. `2.0`) to slow down or speed up the animation. Alternatively, you can enable slow animations in the iPhone simulator. Go up to the simulator menu, select `Debug > Slow animations`.

Spring Animation

The animation is cool, right? Let me introduce a variation of the `UIView` animation known as *spring animation*. Spring animation is very common in iOS 7 or later. One example is the animation during app launch. To take advantage of spring animation in your app, here is the single method call you need:

```
UIView.animate(withDuration: 0.4, delay: 0.0, usingSpringWithDamping: 0.3,
initialSpringVelocity: 0.2, options: .curveEaseInOut, animations: {
    self.containerView.transform = CGAffineTransform.identity
}, completion: nil)
```

The method should look familiar to you, but it adds the `damping` and `initialSpringVelocity` parameters. Damping takes a value from 0 to 1, and controls how much resistance the spring has when it approaches the end state of an animation. If you want to increase oscillation, set to a lower value. The `initialSpringVelocity` property specifies the initial spring velocity. Try to replace the existing animation code with the above code and see how the spring animation works.

Slide-Down Animation

`UIView.animate(withDuration:animations:)` is one of the common transforms for scaling a view. Let's use another affine transform to create a slide-down animation:

```
CGAffineTransform.init(translationX: 0, y: -1000)
```

`CGAffineTransform` allows you to change the view's position. For a slide-down animation, we first move the container view off screen and then bring it back to its original position. In the `viewDidLoad` method, replace `containerView.transform` with the following line of code:

```
containerView.transform = CGAffineTransform.init(translationX: 0, y: -1000)
```

The line of code will move the container view off screen (top). The final state is to restore the container view to the original position. So we can keep the `viewDidAppear` method intact.

Now run the app again and the slide-up animation should work.

- 1 After `viewDidLoad` is executed, the container view is moved off the screen.
- 2 Followed by `viewDidAppear`, the `viewWillAppear` method is called. The animation starts such that the container view begins to restore its original position.
- 3 The animation stops when the container view is in its original position.

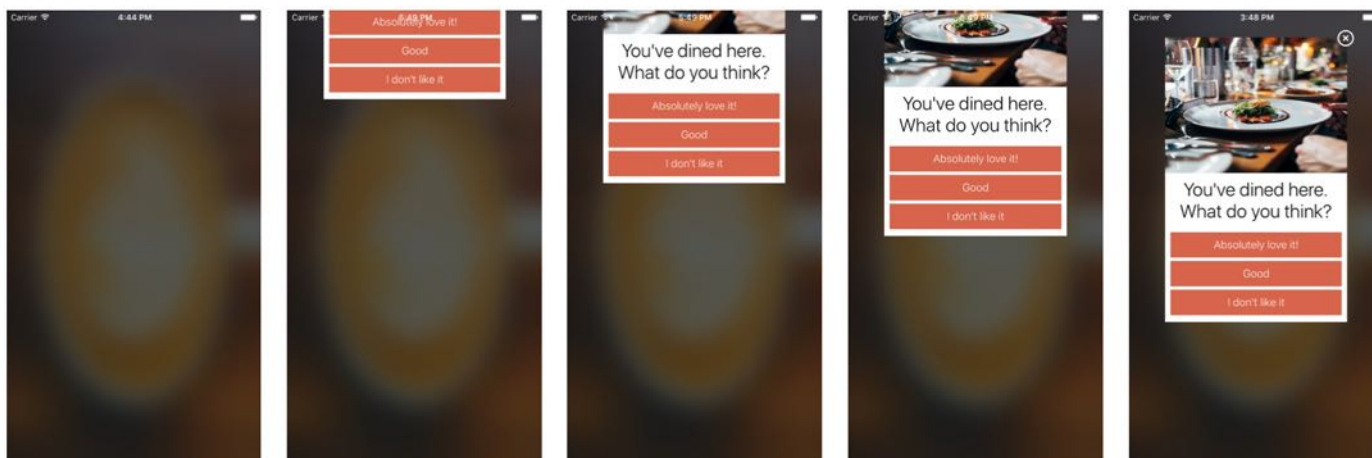


Figure 16-22. Slide-down animation

Combining Two Transforms

What's interesting about affine transform is that you can concatenate one transform with another. This is the function you need to remember:

```
transform1.concatenating(transform2)
```

Now that we have implemented the *translate* transform for the container view. Wouldn't it be great if we can add a scale transform to it? In the `viewDidLoad` method, replace the following code:

```
containerView.transform = CGAffineTransform.init(translationX: 0, y: -1000)
```

with:

```
let scaleTransform = CGAffineTransform.init(scaleX: 0, y: 0)
let translateTransform = CGAffineTransform.init(translationX: 0, y: -1000)
let combineTransform = scaleTransform.concatenating(translateTransform)
containerView.transform = combineTransform
```

The code is very straightforward. We combine the two transforms by calling the `concatenating` function. Again, we can keep the `viewDidAppear` method unchanged.

That's it! You're ready to go and see how the combined animation works.

Unwind Segues and Data Passing

Earlier, we used an unwind segue to "go back" to the detail view, when a user taps the *close* button. How about the *rating* buttons? How can we pass the selected rating from the review view controller to the detail view controller?

We will make use of unwind segues to facilitate the data passing. Here are what we are going to implement:

- We will add another unwind action method in the `ReviewViewController` class - when any of the buttons (dislike/good/great) is tapped, the method will be called.
- In the method, we determine which button is tapped, and save the corresponding rating.
- Then we can update the *Been here* field of the restaurant detail view to display the rating.

It sounds complicated, but the implementation is quite similar to the one we just implemented for the close button.

Let's first start with the unwind action method. Declare the following method in the `RestaurantDetailViewController` class:

```
@IBAction func ratingButtonTapped(segue: UIStoryboardSegue) {  
}
```

Now go to `Main.storyboard` to connect each of the rating buttons with the unwind action method. Control-drag from the *Absolutely love it!* button to the exit icon. Release the button and select `ratingButtonTappedWithSegue:`. Once the unwind segue is created, select it in the document outline. Set its identifier to "great" in the Attributes inspector.

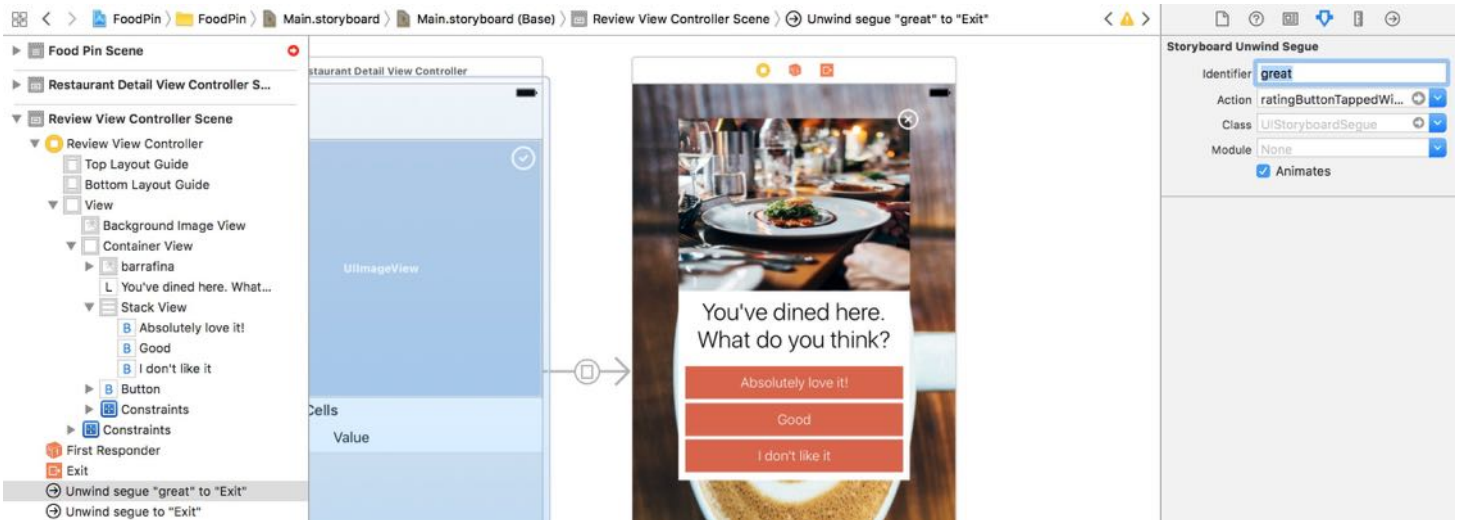


Figure 16-23. Adding an identifier for the unwind segue

Repeat the same step to connect the other two buttons with the `ratingButtonTappedWithSegue:` method. For the Good button, set the unwind segue's identifier to "good". For the "I don't like it" button, please use "dislike" as the identifier of the unwind segue.

Do you know why we set these identifiers? Each of the rating buttons is connected to the same unwind action method. We need to have a way to find out which button is tapped. This is the reason why we set these segue identifiers.

Now go to `Restaurant.swift` and add a `rating` property:

```
var rating = ""
```

Later we will store the restaurant rating in this property.

Next, go back to `RestaurantDetailViewController.swift` to implement the unwind action method. Update the `ratingButtonTapped(segue:)` method like this:

```
@IBAction func ratingButtonTapped(segue: UIStoryboardSegue) {
    if let rating = segue.identifier {
        restaurant.isVisited = true

        switch rating {
        case "great": restaurant.rating = "Absolutely love it! Must try."
        }
    }
}
```



```

        case "good": restaurant.rating = "Pretty good."
        case "dislike": restaurant.rating = "I don't like it."
        default: break
    }
}

tableView.reloadData()
}

```

The above code is pretty straightforward. We first retrieve the unwind segue's identifier. When a user rates a restaurant, it means he/she has dined there. So we update the `isVisited` property to `true`. Depending on the segue's identifier (great/good/dislike), we set the rating of the restaurant accordingly. Finally, we reload the table view to refresh the table.

If you run the app now and rate a restaurant, you will notice that the `Been here` field is updated to "Yes, I've been here before."

To make the app even better, let's display the rating in the same field. Change the following line of code in `tableView(_:cellForRowAt:)` from:

```
cell.valueLabel.text = (restaurant.isVisited) ? "Yes, I've been here before" : "No"
```

to:

```
cell.valueLabel.text = (restaurant.isVisited) ? "Yes, I've been here before. \ (restaurant.rating)" : "No"
```

That's it! You've passed the selected rating from one controller to another. Hit the Run button to test the app. Choose a restaurant, rate it and the detail view's *Been here* field should be updated accordingly.

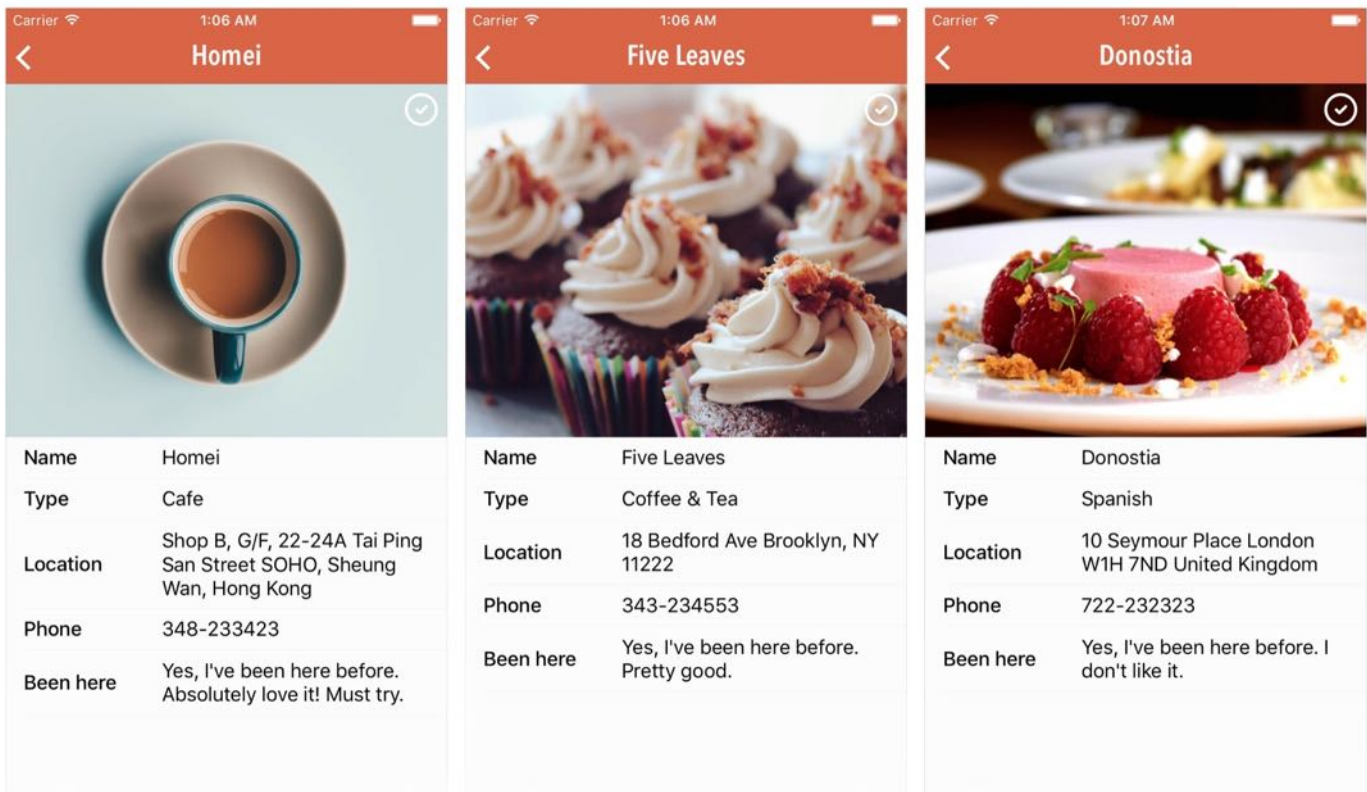


Figure 16-24. Sample rating in the Been here field

Exercise #1

This time, I have two exercises for you. The first one is designed to test your knowledge of data passing. Now that we display a static image in the container view of the review view controller, would it be great if we can display the image of the selected restaurant? Try to update the project to make this happen.

Hint: You can add a restaurant property in the ReviewViewController class and pass the restaurant object from detail view controller by implementing `prepare(for:sender:)`.

Exercise #2

The next exercise is related to animation. Your exercise is to animate the close button in the Review View Controller. Try to animate each button and create a move-in effect (right to left) similar to that shown in figure 16-25.

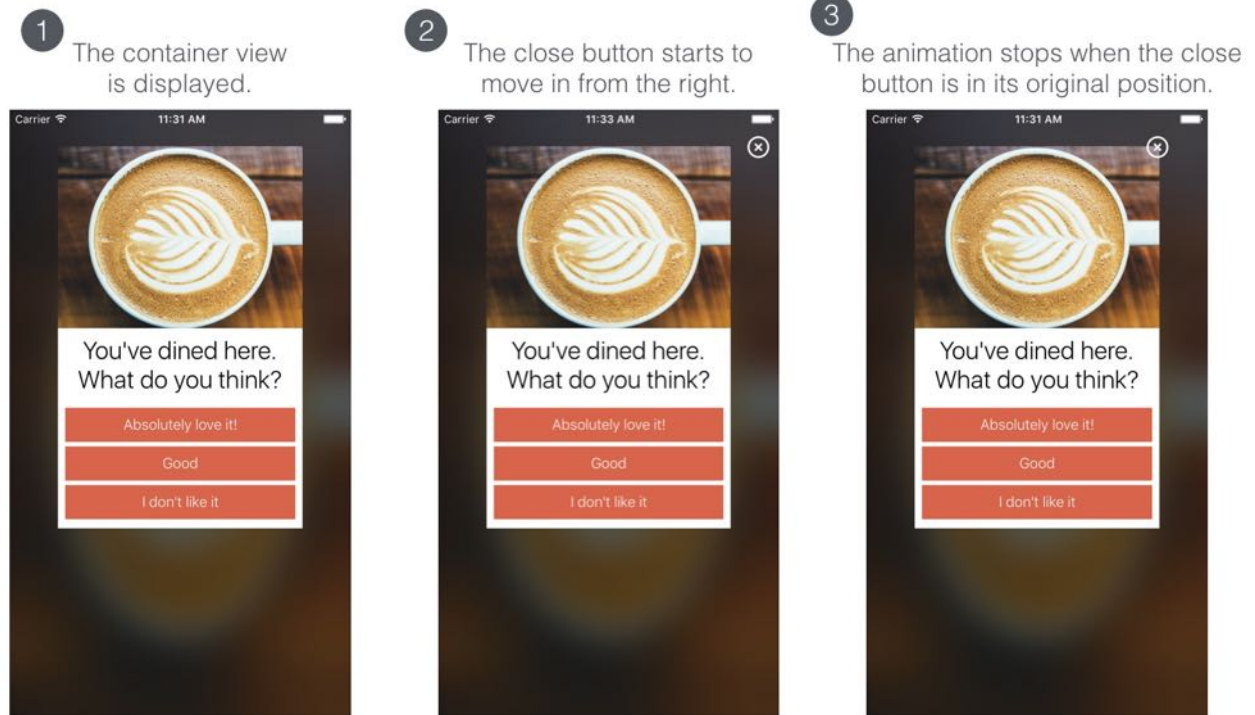


Figure 16-25. Animating the close button

Hint: You will need to create an outlet for the close button and create a translate transform for the button.

Summary

This is another huge chapter that covers UIView animation, visual effects, and unwind segues. I hope you love all the techniques you've learned about UIView animations and visual effects. As you can see, it is super easy to animate a view, whether it's a button or a view. I would encourage you to play around with the parameters (like damping, initial spring velocity and delay) and see what animations you can create. And, don't forget to take some time to complete the exercises.

For your reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinAnimation.zip>. For the exercise, you can download the solution from <http://www.appcoda.com/resources/swift3/FoodPinAnimationExercise.zip>.

Next up we'll see how to add a map to your app.

Chapter 17

Working with Maps



The longer it takes to develop, the less likely it is to launch.

-Jason Fried, Basecamp

The MapKit framework provides APIs for developers to display maps, navigate through maps, add annotations for specific locations, add overlays on existing maps, etc. With the framework

you can embed a fully functional map interface into your app without any coding.

In iOS 9 or later, the MapKit framework allows developers to provide pin customization, transit and flyover support. We will go over some of these features with you. In particular, you will learn a few things about the framework:

- How to embed a map in a view and table view footer
- How to translate an address into coordinates using Geocoder
- How to add and customize a pin (i.e. annotation) on map
- How to customize an annotation

To give you a better understanding of the MapKit framework, we will add a map feature in the FoodPin app. After the change, the app will bring up a map when a user taps the address of a restaurant, and pin its location on the map.

Cool, right? It's gonna be fun. Let's get started.

Using MapKit Framework

By default, the MapKit framework is not bundled in the Xcode project. To use it, you have to first add the framework and bundle it in your project. But you don't need to do it manually. Xcode has a capability section that lets you configure frameworks for various Apple technologies such as Maps and iCloud.

In the project navigator, select the FoodPin project and then select the FoodPin target. You can then enable the `Maps` feature under the Capabilities section. Just flip the switch to ON, and Xcode automatically configures your project to use the MapKit framework.

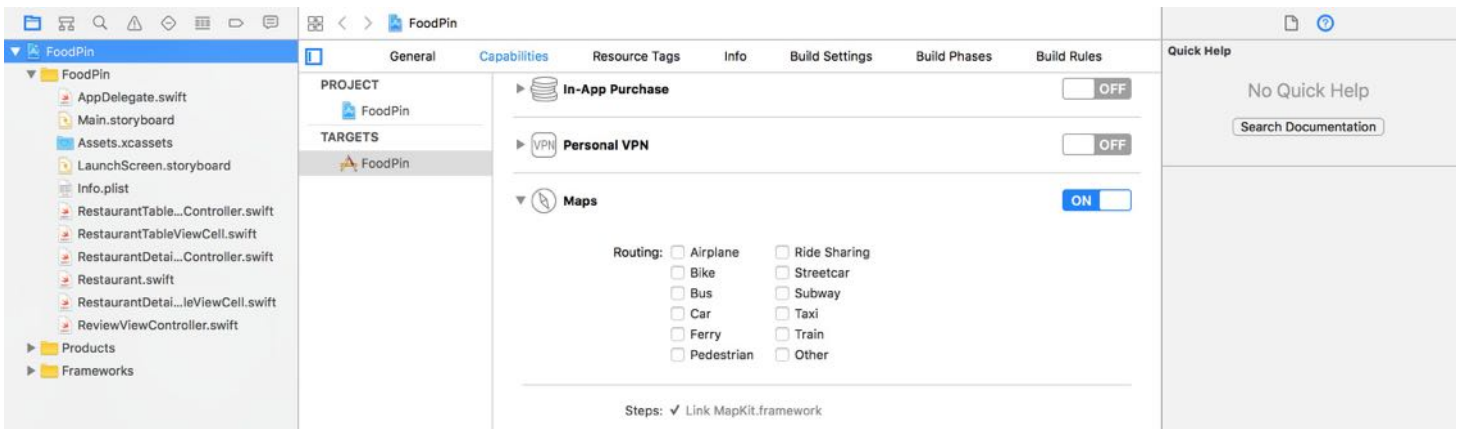


Figure 17-1. Enabling Maps in your Xcode project

Adding a Map Interface to Your App

What we're going to do is to add a non-interactive map to the footer of the restaurant detail view. When a user taps the map, the app navigates to a map view controller showing a full screen map of the restaurant location. Figure 17-2 displays the resulting UI of the app.

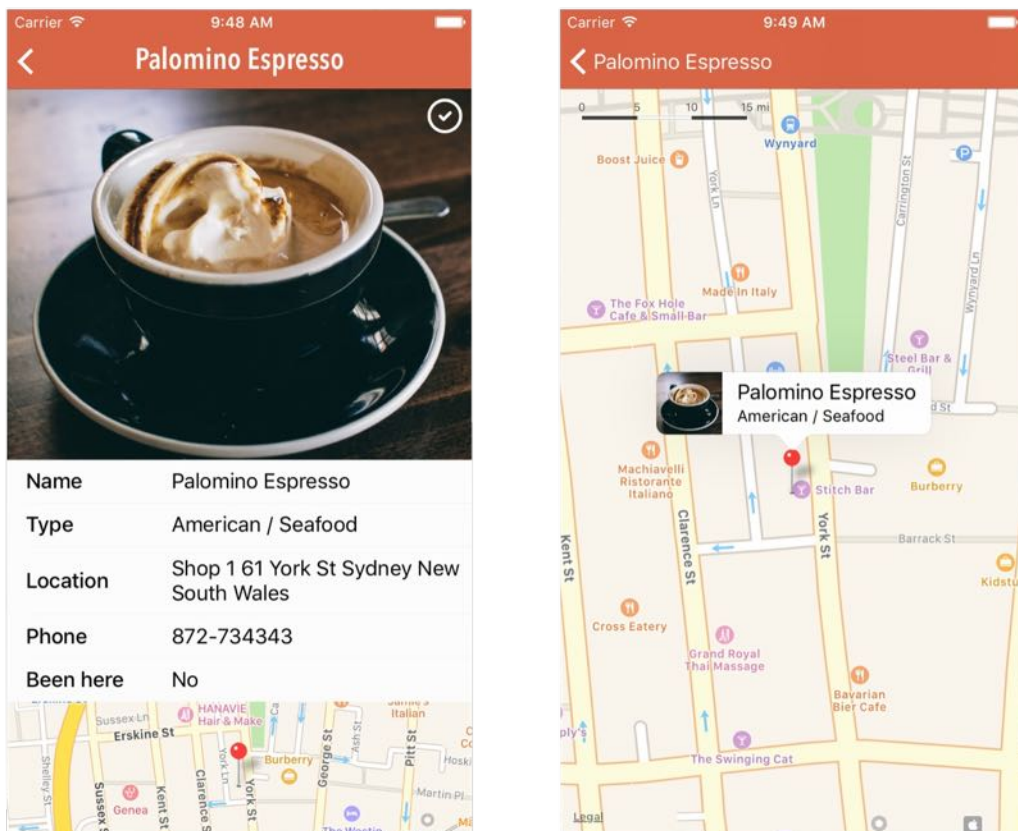


Figure 17-2. An embedded map in the table view footer (left), A full screen map in a view controller (right)

Now open `Main.storyboard` and drag a map view from the Object library to the footer of the table view in the detail view controller. After adding the map, it should appear right below the prototype cell.

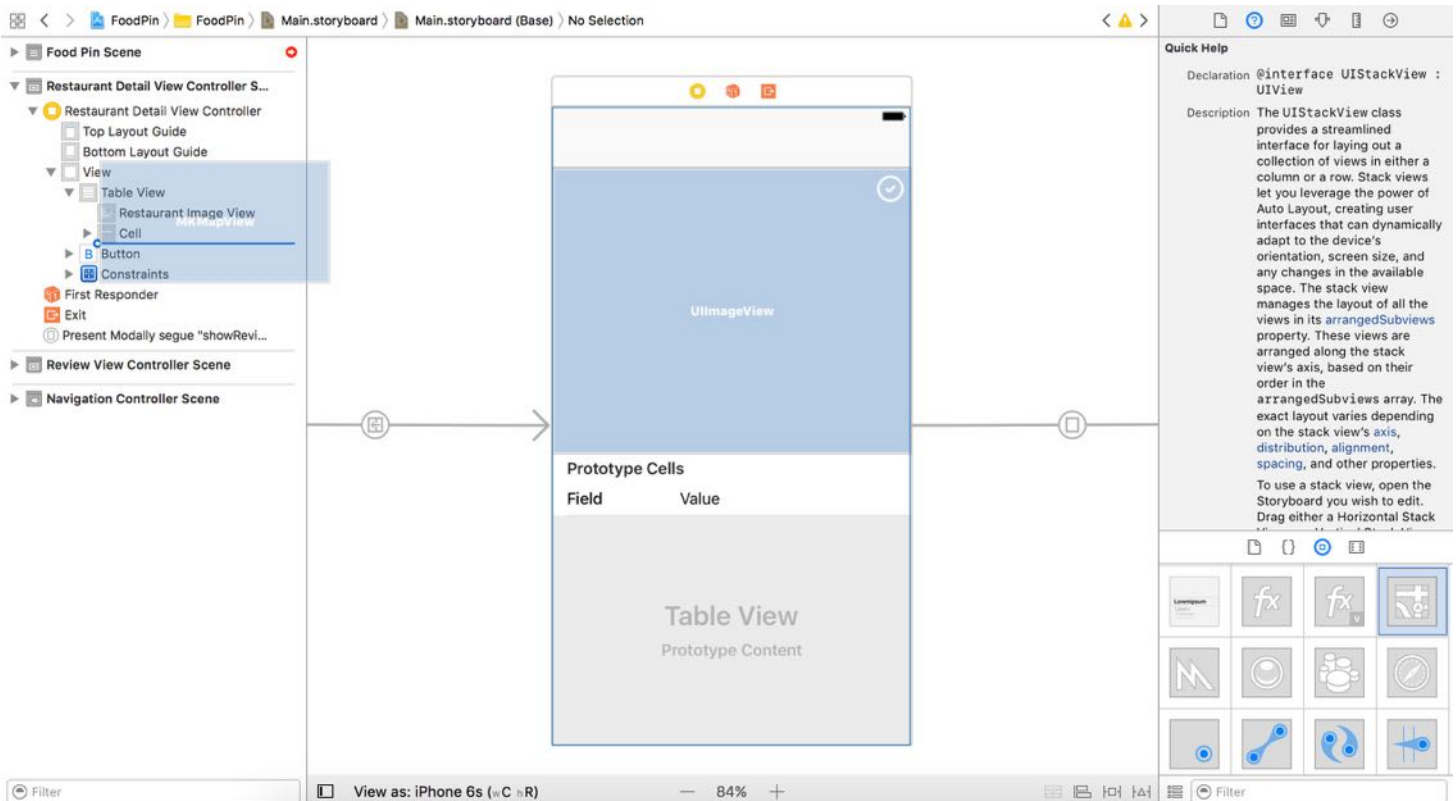


Figure 17-3. Adding a map view to the table view footer

Resize the map view to make it a little bit bigger. You can set its height in the Size inspector to 135 points.

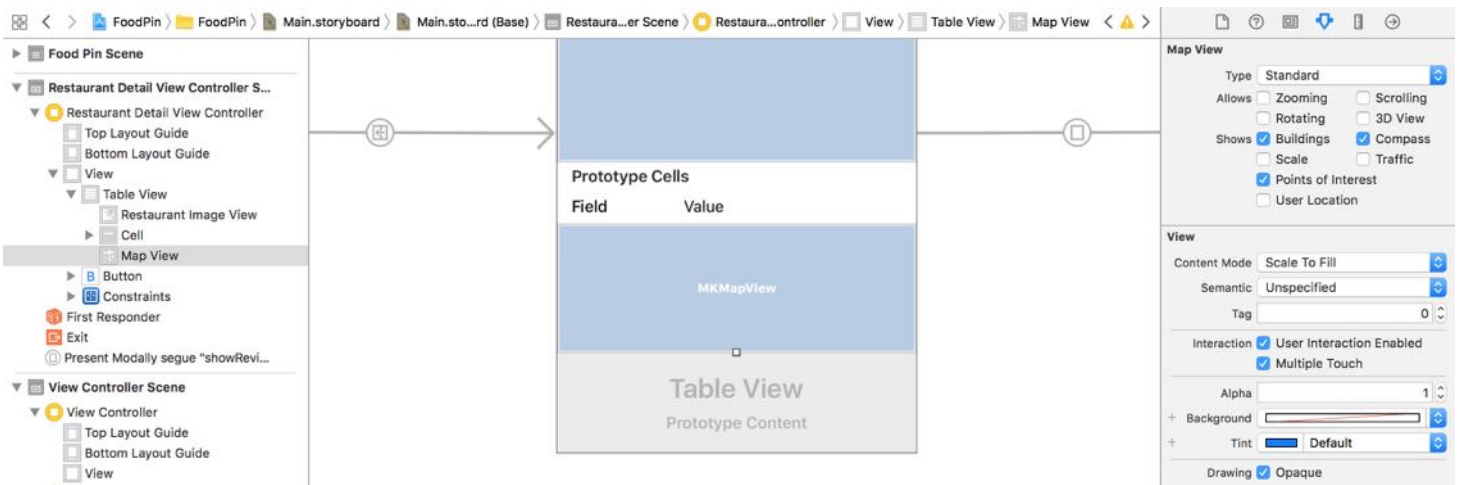


Figure 17-4. Resize the map view

The map view provides various options such as zooming and scrolling for customizing the map features. In Interface Builder, select the map view and you can edit the options by going to the Attributes inspector. As we want this map to be non-interactive, uptick the zooming, scrolling, and rotating checkboxes.

I know you can't wait to test out the change. Let's hit the Run button to have a quick test. When you select a restaurant to navigate to the detail view, the UI is still the same as before. Where is the map?

In the earlier chapter, we have written a line of code to clear the table view footer. This is why the map is not displayed properly. Now go to `RestaurantDetailViewController.swift` and delete the following line of code in the `viewDidLoad()` method:

```
tableView.tableFooterView = UIView(frame: CGRect.zero)
```

Once you made this change, the app should display the map in the table footer. Cool, right? This is the power of MapKit. Without writing a line of code, you already embed a map within your app, though it just displays a default map.

Before that, however, let's implement the UI of the full screen map.

For the full screen map, we will implement it as a separate view controller. So drag a view controller from the Object library to the storyboard. Resize the map view to fit the view, and

then click "Resolve Auto Layout Issues" > "Add Missing Constraints" to add the required layout constraints.

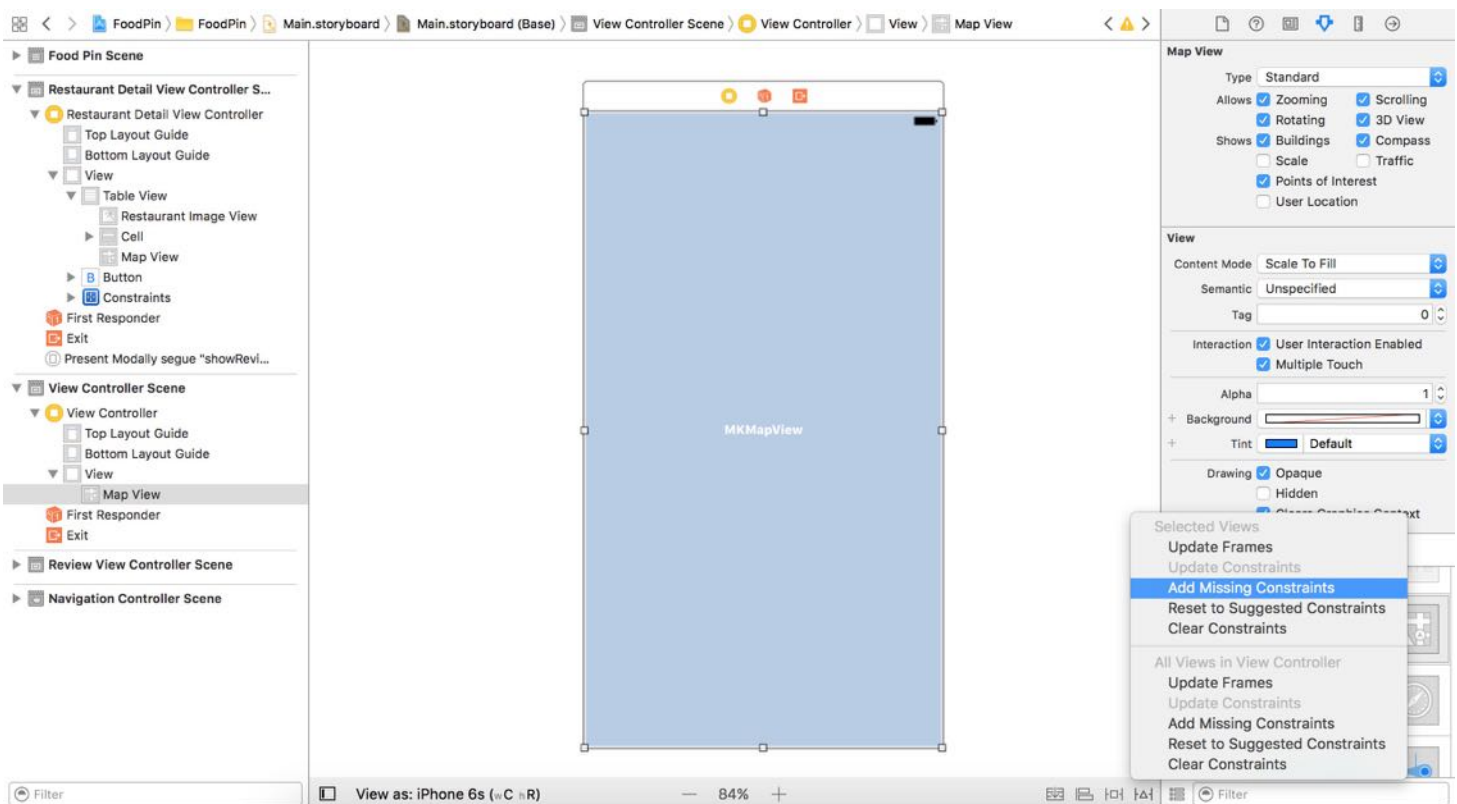


Figure 17-5. Defining layout constraints for the map view

To bring up a full screen map when a user taps the non-interactive map in the table footer, we have to set up a segue between the controllers.

Now hold control key and drag from the Restaurant Detail View Controller to the new map view controller to create a segue. Select *show* as the segue type. For the segue we just created, set the identifier to `showMap` under the Attribute inspector.

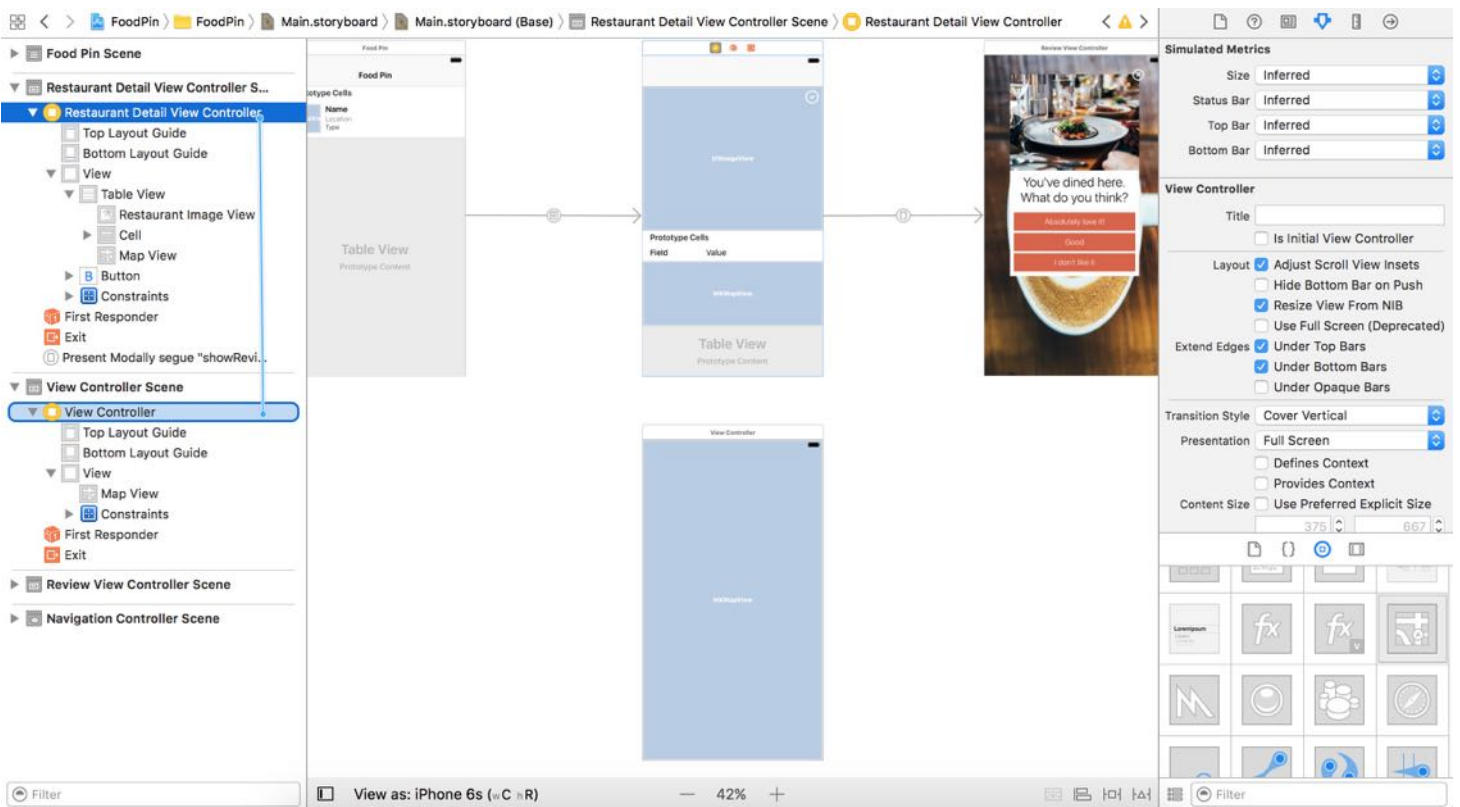


Figure 17-6. Creating a segue between the view controllers

You may wonder why we didn't create a segue between the map view (in the table footer) and the map view controller. Both table view header and footer are not selectable. Therefore, we can't drag from the map view to the map view controller. In this case, we set up the segue between the view controllers. Later we will trigger the transition programmatically.

As said, the map in the table footer can't detect touches by default. We have to implement our own solution to detect the tap. The iOS SDK provides a set of gesture recognizers to detect common gestures such as taps. We will use a class called `UITapGestureRecognizer` to handle the tap gesture.

Now open `RestaurantDetailViewController.swift`. Recalled that we have configured the MapKit framework in the project, you have to add an `import` statement in order to use it. So first add a line of code at the very beginning to import the MapKit framework:

```
import MapKit
```

Next, declare an outlet variable for the mapView:

```
@IBOutlet var mapView: MKMapView!
```

To use `UITapGestureRecognizer` for detecting the tap gesture, you will need to initialize an `UITapGestureRecognizer` object and attach it to the map view. Insert the following code in the `viewDidLoad()` method:

```
let tapGestureRecognizer = UITapGestureRecognizer(target: self, action:
#selector(showMap))
mapView.addGestureRecognizer(tapGestureRecognizer)
```

During initialization, we tell `UITapGestureRecognizer` the target method to call, which is `showMap`, when a tap gesture is detected. We haven't implemented the `showMap()` method yet. So add these code to the `RestaurantDetailViewController` class:

```
func showMap() {
    performSegue(withIdentifier: "showMap", sender: self)
}
```

The method is simple. We programmatically trigger the `showMap` segue by calling `performSegue` with the identifier we configured earlier. Now when a user taps the map view in the table footer, the `showMap()` method will be invoked. Consequently, the `showMap` segue is triggered.

Lastly, go back to `Main.storyboard` and establish a connection between the map view (in the table view footer) and the `mapView` outlet.

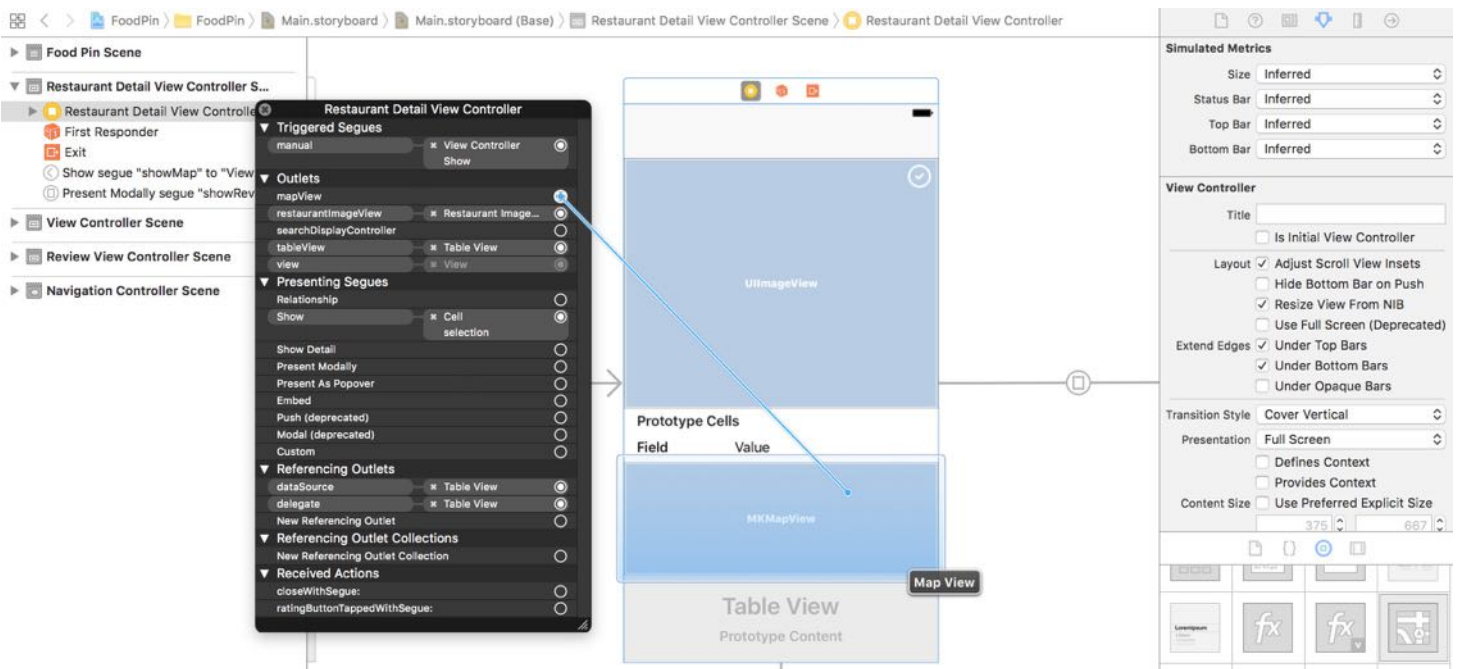


Figure 17-7. Connecting the map view outlet

If you compile and run the app, tap the map in the detail view. The app should navigate to a fully functional map.

Converting an Address into Coordinates Using Geocoder

Now that you understand how to embed a map in your app, but how can you pin a location on maps?

To highlight a location on the map, you cannot just use a physical address. The MapKit framework doesn't work like that. Instead, the map has to know the geographical coordinates expressed in terms of the latitude and longitude of the corresponding point on the globe.

The framework provides a `Geocoder` class for developers to convert a textual address, known as *placemark*, into global coordinates. This process is usually referred to *forward geocoding*. Conversely, you can use `Geocoder` to convert latitude and longitude values back to a *placemark*. This process is known as *reverse geocoding*.

To initiate a forward-geocoding request using the `CLGeocoder` class, all you need do is create an instance of `CLGeocoder`, followed by calling the `geocodeAddressString` method with the

address parameter. Here is an example:

```
let geoCoder = CLGeocoder()
geoCoder.geocodeAddressString("524 Ct St, Brooklyn, NY 11231",
completionHandler: { placemarks, error in

// Process the placemark

})
```

There is no designated format of an address string. The method submits the specified location data to the geocoding server asynchronously. The server then parses the address and returns you an array of placemark objects. The number of placemark objects returned greatly depends on the address you provide. The more specific the address information you have given, the better the result. If your address is not specific enough, you may end up with multiple placemark objects.

With the `placemark` object, which is an instance of `CLPlacemark` class, you can easily get the geographical coordinate of the address using the code below:

```
let coordinate = placemark.location?.coordinate
```

The completion handler is the code block to be executed after the forward-geocoding request completes. Operations like annotating the placemark will be done in the code block.

A Quick Overview of Map Annotations

Now that you have a basic idea of `Geocoder` and understand how to get the global coordinates of an address, we will look at how you can pin a location on maps. To do that, the MapKit framework provides an annotation feature for you to pinpoint a specific location.

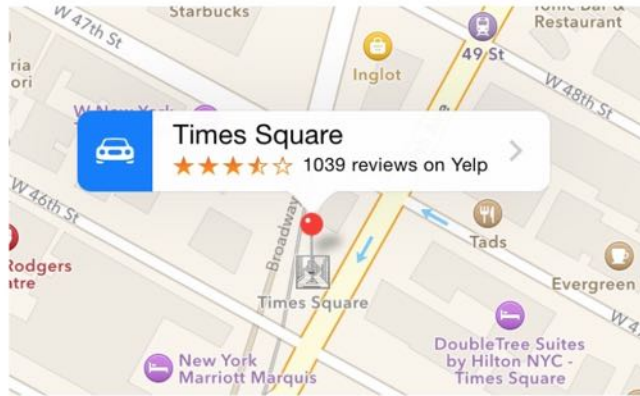


Figure 17-8. An annotation

An annotation can appear in many forms. The pin annotation that you usually see in a Maps app is an example of annotations. Typically an annotation consists of an image (e.g. pin) and a callout bubble that displays additional information about the location.

From a developer's point of view, an annotation actually consists of two different objects:

- an annotation object - which stores the data of an annotation such as the name of the placemark. The object should conform to the `MKAnnotation` protocol as defined in the Map Kit.
- an annotation view - which is the actual object for the visual representation of the annotation. The pin image is an example. If you want to display the annotation in your own form (say, use pencil instead of pin), you'll need to create to your own annotation view.

The MapKit framework comes with a standard annotation object and an annotation view. So you do not need to create your own, unless you want to customize the annotation.

In simplest form, a standard annotation appears as a pin without images and callout bubbles. To add a pin on a map, you just need a few lines of code:

```
let annotation = MKPointAnnotation()
if let location = placemark.location {
    annotation.coordinate = location.coordinate
    mapView.addAnnotation(annotation)
```

```
}
```

The `MKPointAnnotation` class is standard class, which adopts the `MKAnnotation` protocol. By specifying the coordinates in the annotation object, you can call the `addAnnotation` method of the `mapView` object to put a pin on the map.

Adding an Annotation to the Non-interactive Map

After introducing you the basics of annotations and geocoding, let's get back to the FoodPin project. We first add a pin annotation to the non-interactive map in the table footer.

In `RestaurantDetailViewController.swift`, insert the following code in the `viewDidLoad()` method:

```
let geoCoder = CLGeocoder()
geoCoder.geocodeAddressString(restaurant.location, completionHandler: {
    placemarks, error in
        if error != nil {
            print(error)
            return
        }

        if let placemarks = placemarks {
            // Get the first placemark
            let placemark = placemarks[0]

            // Add annotation
            let annotation = MKPointAnnotation()

            if let location = placemark.location {
                // Display the annotation
                annotation.coordinate = location.coordinate
                self.mapView.addAnnotation(annotation)

                // Set the zoom level
                let region =
MKCoordinateRegionMakeWithDistance(annotation.coordinate, 250, 250)
                self.mapView.setRegion(region, animated: false)
            }
        }
    })
```

I'll not go into the above code line by line as we've discussed the usage of `Geocoder` and

annotation earlier. In brief, we first convert the address of the selected restaurant (i.e. `restaurant.location`) into coordinates using `Geocoder`. In most cases, the `placemarks` array should contain a single entry. So we just pick the first element from the array, followed by displaying the annotation on the map view.

The last two lines of code are new to you. Here we use the `MKCoordinateRegionMakeWithDistance` function to adjust the initial zoom level of the map to 250m radius.

Note: You can still display a pin without these two lines of code. Try to remove these lines of code and see how the map differs.

If you run the app now, it should correctly display the restaurant location on the map.

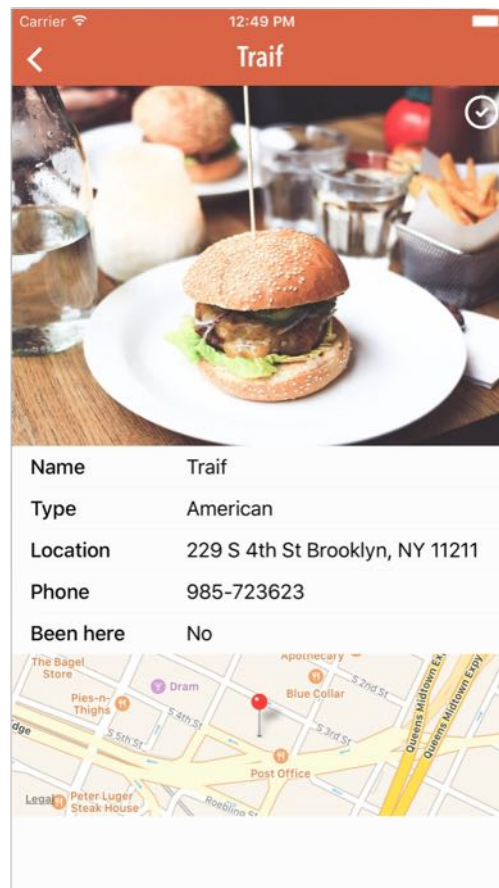


Figure 17-9. Pin the restaurant location on a map

Adding an Annotation to the Full Screen Map

Okay, let's move onto the implementation of the full screen map.

As usual, we first create a custom class for the map view controller. In the project navigator, right click the `FoodPin` folder and select "New File...". Create the new class using the `Cocoa Touch class` template and name the class `MapViewController`. Make sure you set it as a subclass of `UIViewController`, and save the file.

Again we first import the MapKit framework. Insert the following line of code at the very beginning of the `MapViewController.swift` file:

```
import MapKit
```

Next, declare the following outlet variable for the map view and another variable for the selected restaurant:

```
@IBOutlet var mapView: MKMapView!  
  
var restaurant:Restaurant!
```

The outlet variable is used for establishing a connection with the map view in the storyboard. Go to Interface Builder and select the map view controller. Under the Identity inspector, set the custom class to `MapViewController`. Then establish a connection between the outlet variable and the map view.

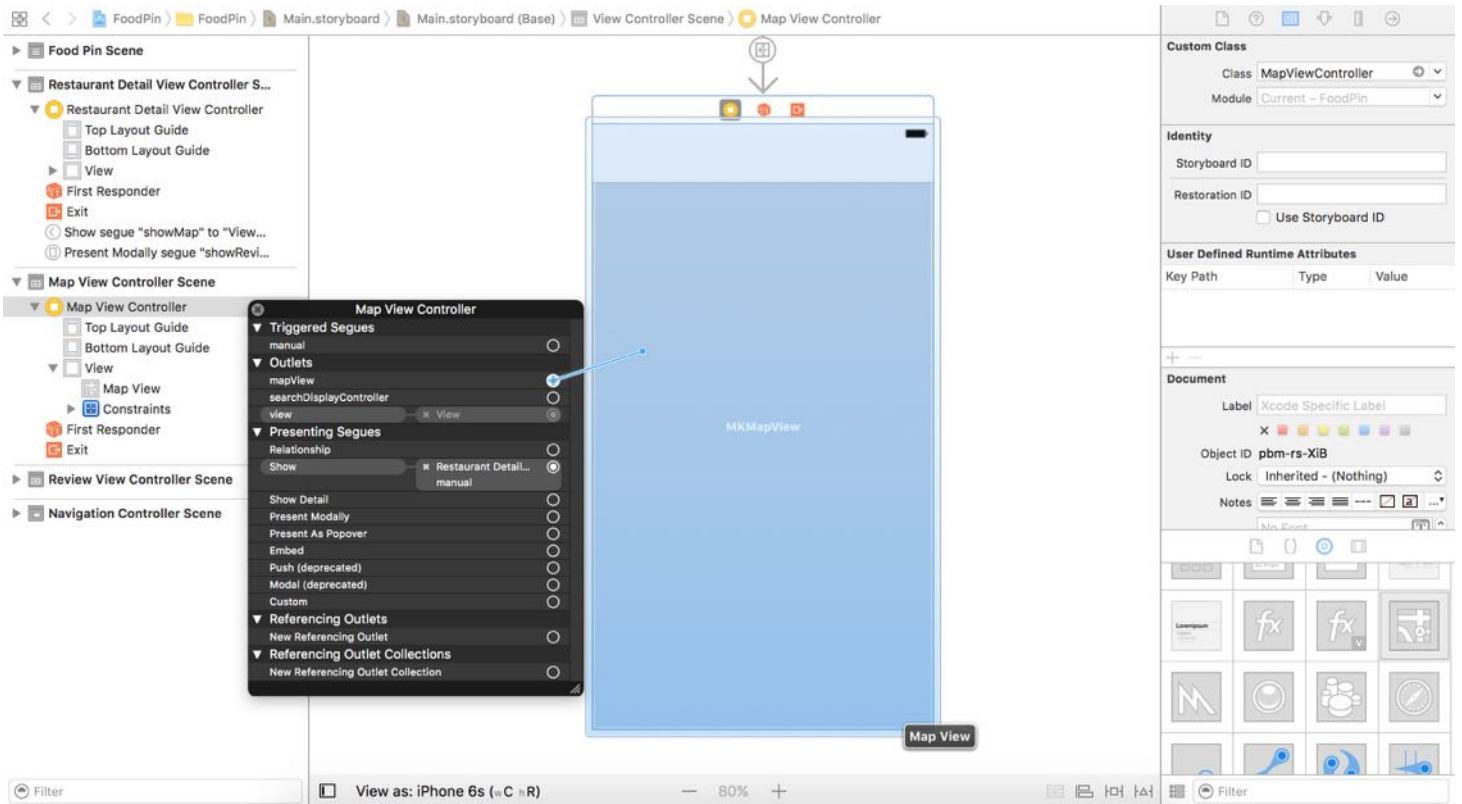


Figure 17-10. Establish a connection between MKMapView and the outlet variable

To add an annotation on map, update the `viewDidLoad` method to the following code:

```
override func viewDidLoad() {
    super.viewDidLoad()

    // Convert address to coordinate and annotate it on map
    let geoCoder = CLGeocoder()
    geoCoder.geocodeAddressString(restaurant.location, completionHandler: {
    placemarks, error in
        if error != nil {
            print(error)
            return
        }

        if let placemarks = placemarks {
            // Get the first placemark
            let placemark = placemarks[0]

            // Add annotation
            let annotation = MKPointAnnotation()
            annotation.title = self.restaurant.name
            annotation.subtitle = self.restaurant.type
        }
    })
}
```

```

        if let location = placemark.location {
            annotation.coordinate = location.coordinate

            // Display the annotation
            self.mapView.showAnnotations([annotation], animated: true)
            self.mapView.selectAnnotation(annotation, animated: true)
        }
    }
}

```

The above code is very similar to the one we have just discussed, so I'll not go into the above code line by line. We convert the address into a coordinate for annotation. Once again, we use `MKPointAnnotation` to add a pin to the map. But this time we assign it with a title and a subtitle, and use the `showAnnotations` method of the `mapView` object to put a pin on the map. The method is smart enough to determine the best fit region for the annotation. By default, the callout bubble is not shown on the map until the user taps the pin. Here we also invoke the `selectAnnotation` method to display the callout bubble.

There is still one thing left before testing the app. We haven't passed the selected restaurant to the map view controller. In the `RestaurantDetailViewController` class, update the `prepare(for:sender:)` method like this:

```

override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showReview" {
        let destinationController = segue.destination as! ReviewViewController
        destinationController.restaurant = restaurant
    } else if segue.identifier == "showMap" {
        let destinationController = segue.destination as! MapViewController
        destinationController.restaurant = restaurant
    }
}

```

We simply get the selected restaurant and pass it to the destination view controller. In this case, it's the `MapViewController` class. Okay, let's compile and run the app. Tap the Map button in the detail view and you'll see a pin on the map.



Figure 17-11. Tapping the map button now shows a map with the restaurant's location

Adding an Image to the Annotation View

Wouldn't it be great if we can show the restaurant thumbnail in the callout bubble?

As mentioned in the beginning of this chapter, an annotation view controls the visual part of an annotation. In order to add a thumbnail or image in the annotation, you have to modify the annotation view. To do that, you have to adopt the `MKMapViewDelegate` protocol, which defines a set of optional methods that you can use to receive map-related update messages. The map view uses one of these methods to request annotation. Every time when the map view needs an annotation view, it calls the `mapView(_:viewFor:)` method:

```
optional func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView?
```

So far we haven't adopted the protocol and provide our own implementation for the method.

This is why a default annotation view is displayed. We're going to implement the method and create our own annotation view to embed the restaurant image.

Go back to the `MapViewController.swift` file and adopt the `MKMapViewDelegate` protocol:

```
class MapViewController: UIViewController, MKMapViewDelegate
```

In the `viewDidLoad` method, add the following line of code to define the delegate of `mapView`:

```
mapView.delegate = self
```

Quick note: Here we define `MapViewController` as the delegate object of `mapView`. The delegation pattern is among the most common patterns in iOS development. By now, I assume you have a good understanding of the delegate pattern. If not, read chapter 7 again.

Next implement the `mapView(_:viewForAnnotation:)` method using the following code:

```
func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) ->
MKAnnotationView? {
    let identifier = "MyPin"

    if annotation.isKind(of: MKUserLocation.self) {
        return nil
    }

    // Reuse the annotation if possible
    var annotationView:MKPinAnnotationView? =
mapView.dequeueReusableAnnotationView(withIdentifier: identifier) as?
MKPinAnnotationView

    if annotationView == nil {
        annotationView = MKPinAnnotationView(annotation: annotation,
reuseIdentifier: identifier)
        annotationView?.canShowCallout = true
    }

    let leftIconView = UIImageView(frame: CGRect.init(x: 0, y: 0, width: 53,
height: 53))
    leftIconView.image = UIImage(named: restaurant.image)
    annotationView?.leftCalloutAccessoryView = leftIconView

    return annotationView
}
```

Let's go through the above code line by line.

Other than placemark, the user's current location is also a kind of annotation. The map view will also call this method when annotating the user's location. As you may know, the current location is displayed as a blue dot in maps. Even though we haven't enabled the app to display the current location, we don't want to change its annotation view. At the very beginning of the code, we verify if the annotation object is a kind of `MKUserLocation`. If yes, we simply return `nil` and the map view will keep displaying the location using a blue dot.

For performance reasons, it is preferred to reuse an existing annotation view instead of creating a new one. The map view is intelligent enough to cache unused annotation views that it isn't using. Similar to `UITableView`, we can call up the `dequeueReusableAnnotationView(withIdentifier:)` method to see if any unused views are available. We then downcast it to `MKPinAnnotationView`.

If there are no unused views available, we create a new one by instantiating a `MKPinAnnotationView` object with the `canShowCallout` property set to `true`. We still use the standard pin as the annotation view. But we add a thumbnail of the selected restaurant to `leftCalloutAccessoryView`, which is a view shown on the left side of the callout bubble.

Cool! We're done. Press the Run button and launch the app. Pick a restaurant and tap the Map button. You should see a thumbnail in the callout bubble.

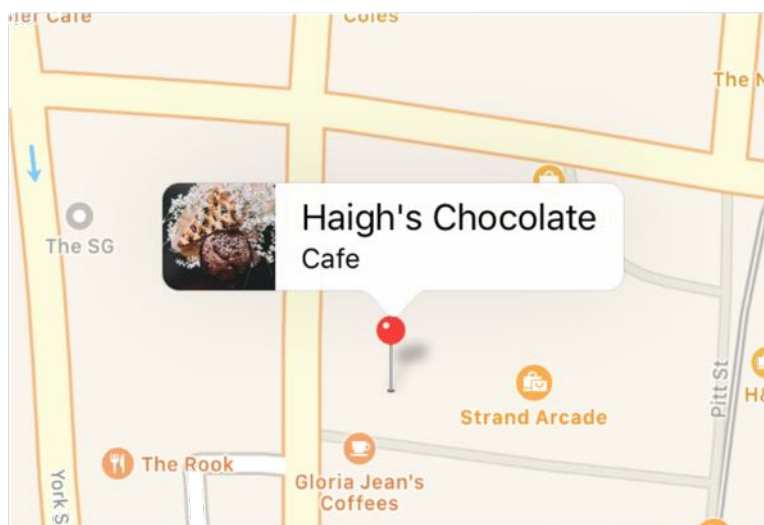


Figure 17-12. An annotation with a thumbnail

Customizing the Pin Color

In iOS 9 or later, you can customize the color of the pin to any color you want. Apple introduces a new property called *pinTintColor* for `MKPinAnnotationView`. To change the pin color, all you need to do is assign the *pinTintColor* property with a `UIColor` object.

```
annotationView?.pinTintColor = UIColor.orange
```

Map Customizations

Starting from iOS 9, Apple also lets developers control what goes on the map view. Here are the three new properties for you to control the content of a map view:

- **showTraffic** - shows any high traffic on your map view
- **showScale** - shows a scale on the top-left corner of your map view
- **showCompass** - displays a compass control on the top-right corner of your map view

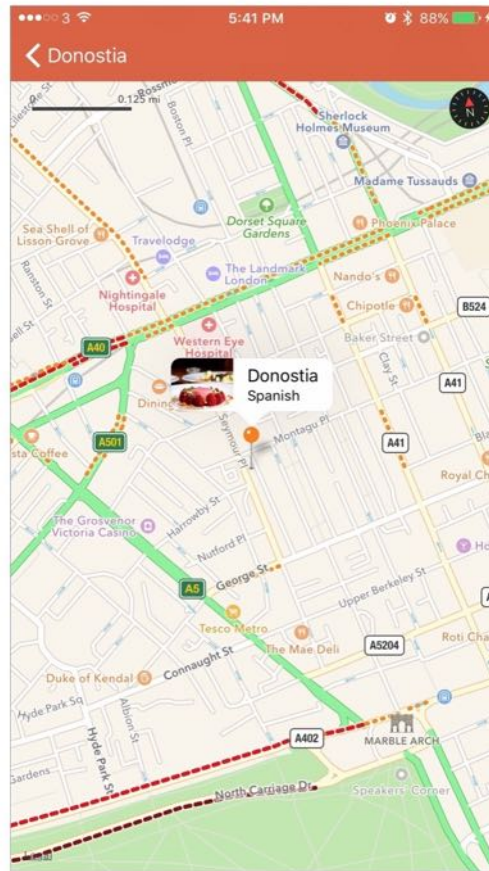


Figure 17-13. Show traffic, scale and compass in the map

As a demo, you can insert the following lines of code in the `viewDidLoad` method to give it a try:

```
mapView.showsCompass = true
mapView.showsScale = true
mapView.showsTraffic = true
```

Summary

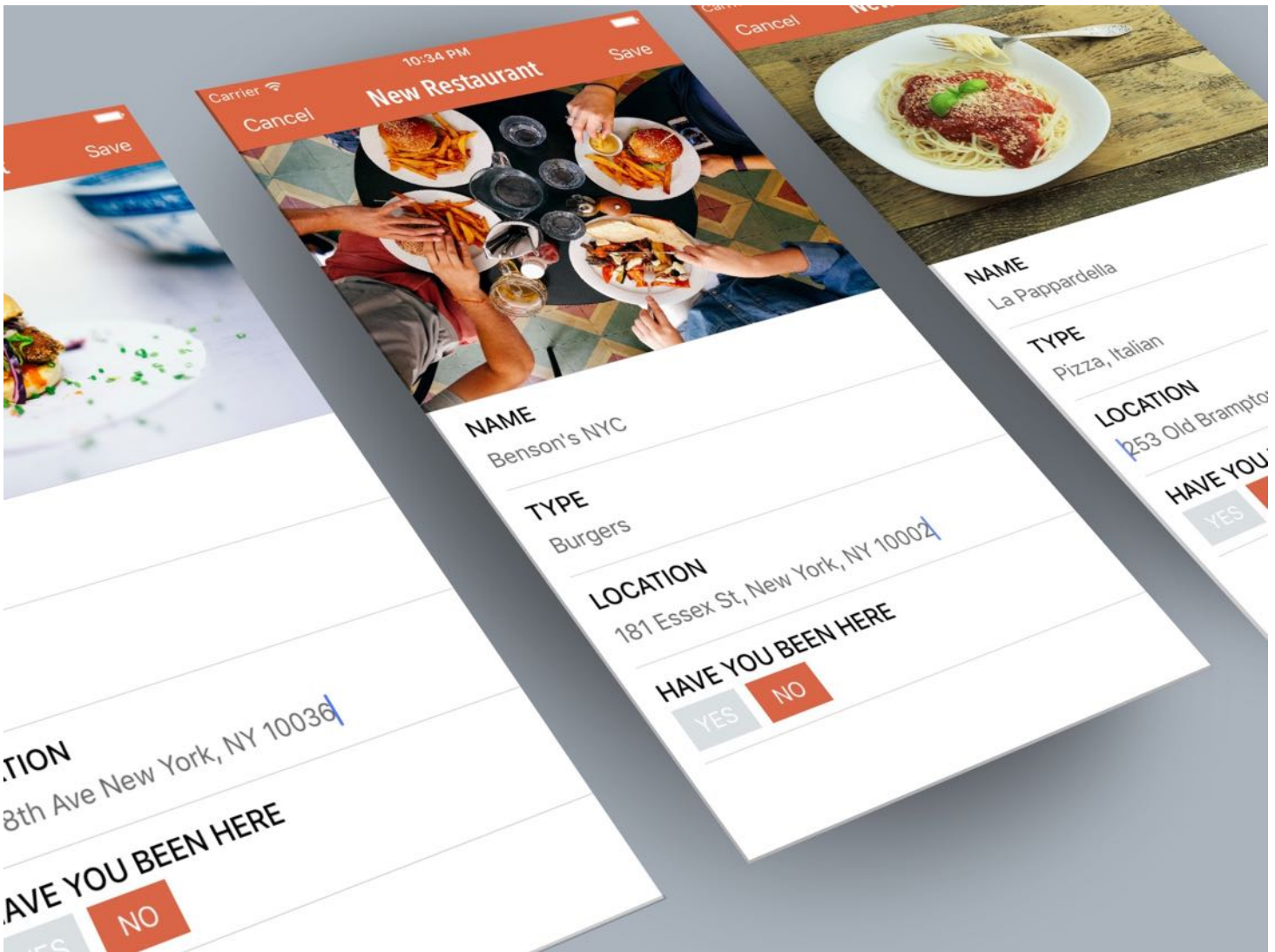
In this chapter, I've walked you through the basics of the MapKit framework. By now, you should know how to embed a map in your app and add an annotation. But this is just a beginning. There is a lot more you can do from here. One thing you can do is further explore [MKDirection](#). It provides you with route-based directions data from Apple servers to get travel-time information or driving or walking directions. You can take the app further by

showing directions.

For your reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinMaps.zip>.

Chapter 18

Introduction to Static Table Views, UIImagePickerControllerController and NSLayoutConstraint



My biggest motivation? Just to keep challenging myself. I see life almost like one long University education that I never had. Everyday I'm learning something new.

- Richard Branson

Up till now, the FoodPin app is only capable of displaying content. We need to find a way for users to add a new restaurant. In this chapter, we will create a new screen that displays an input form for collecting restaurant information. In the form, it will let users pick a restaurant photo from the built-in photo library. You'll learn a number of stuff:

- How to create a form using a static table view
- How to use `UIImagePickerController` to select photo from the built-in photo library and take photos
- How to define auto layout constraints programmatically using `NSLayoutConstraint`

In the first few chapters of the book, we have gone through the basics of table views. The table views that I covered are dynamic in nature. Usually you create a prototype cell and populate it with dynamic content. However, table views are not limited to present dynamic content. Sometimes, you may just want to use a table view to present a form or a setting screen. In this case, a static table view is what you need. Static table views are ideal for situations where there are a pre-defined number of data items to be displayed.

Xcode allows developers to create static table views with minimal code. To illustrate how easy you can use storyboard to implement a static table view, we will build a new screen for adding a new restaurant.

Let's get started.

Adding a New Table View Controller

Go to `Main.storyboard` and drag a table view controller from the Object library to the storyboard. Select the table view in the document outline. In the Attribute inspector, change the *Content* option from `Dynamic Prototypes` to `Static Cells`. Once changed, you will have a table view with three empty static cells.

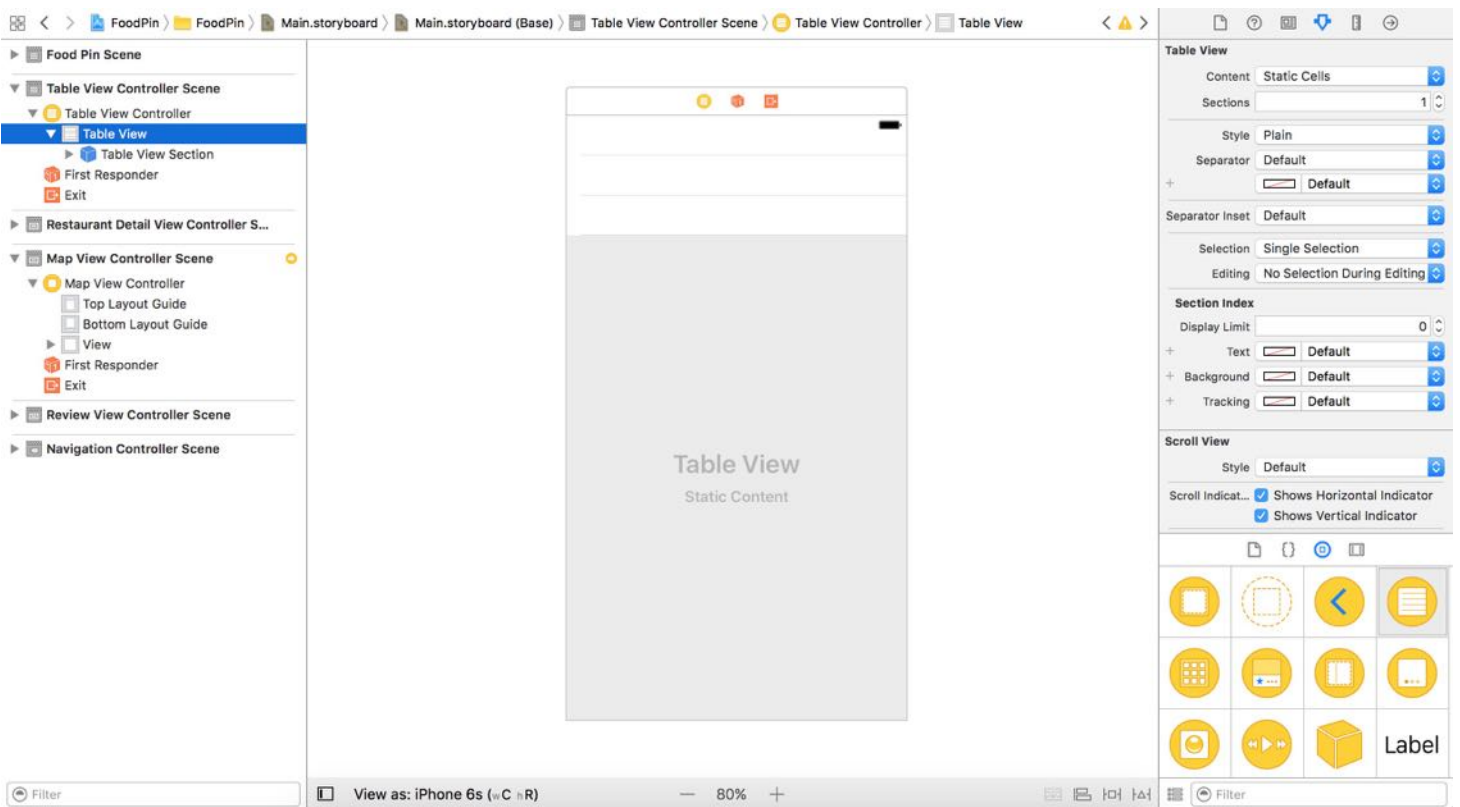


Figure 18-1. A static table view

For our input form, we need a total of 5 table view cells:

- Cell #1 - an image view for displaying the restaurant image
- Cell #2 - a *Name* label and a text field
- Cell #3 - a *Type* label and a text field
- Cell #4 - a *Location* label and a text field
- Cell #5 - a *Have You Been Here* label and two buttons for Yes/No

You can easily increase the number of cells by selecting the Table View Section in the document outline. In the Attribute inspector, you can change the *Rows* option from 3 to 5.



Figure 18-2. Changing the number of rows from 3 to 5

Now we'll layout each of the table view cells. First, download this image pack (<http://www.appcoda.com/resources/swift3/photoicons.zip>) and unzip it. Add the images to `Assets.xcassets`.

Credit: Icons made by [Freepik](http://www.freepik.com) from www.flaticon.com and is licensed by [CC BY 3.0](https://creativecommons.org/licenses/by/3.0/)

For the first cell, change its height to 250 (or whatever value you prefer), and change the background color to *light gray*. Also set the Selection option to *None*.

Then drag an image view from the Object library to it. In the Attribute inspector, set the image to `photoalbum`. Resize the image to 64x64 points and place it right at the center of the cell. Click *Resolve Auto Layout Issues > Add missing constraints* to add the required layout constraints.

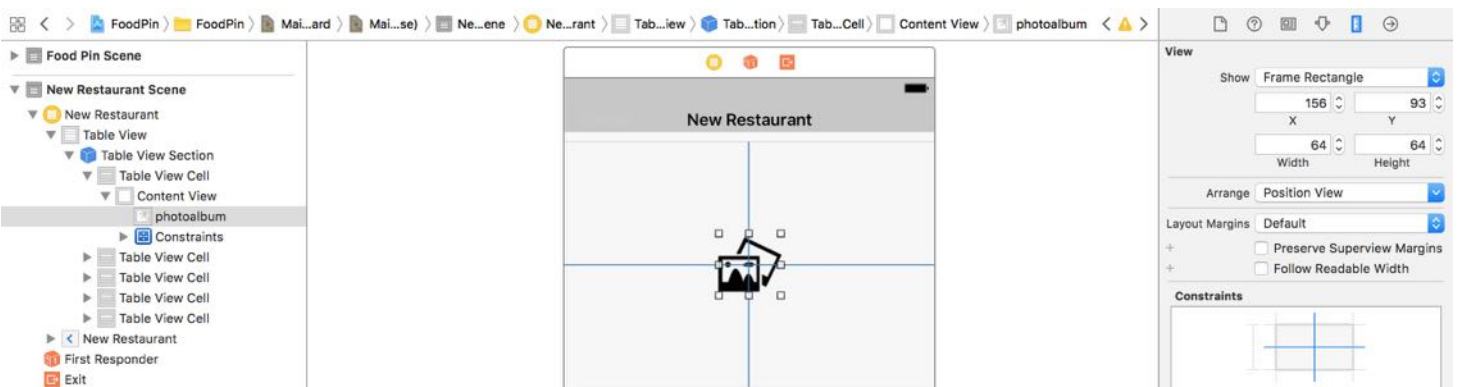


Figure 18-3. Add an image view to the first cell

For the second cell, change its height to 72 . Drag a label to the cell and change its title to NAME . Then, drag a text field to the cell and put it under the name label. The text field is a control for capturing user input and displaying editable text. Typically you use it to gather small amounts of text from users. In the Attribute inspector, set the placeholder value to Restaurant Name and set the border style to None . A placeholder is displayed when there is no other text in the text field. Set the width of the text field to 339 points. Figure 18-4 shows the sample design of the cell. You are not required to follow the design strictly and free to implement your own.



Figure 18-4. The cell layout for capturing user's input

Again, you need to define the layout constraints. Select both the *Name* label and the text field. In the layout bar, click *Resolve Auto Layout Issues > Add missing constraints* to add the required layout constraints.

For the third and fourth cells, repeat the same procedures, but set the label to *TYPE* and *LOCATION* respectively.

For the fifth cell, set its height to 72 points. Then drag a label to the cell and name it *Have You Been Here*. Then add two buttons to the cell. Name one button *YES* and the other one *NO*. For the *YES* button, change its background color to red , and set the background color of the *NO* button to gray . Also, set the text color to white for both buttons. Again, select the labels and the buttons, followed by adding the missing constraints. Below shows a sample design of the field.

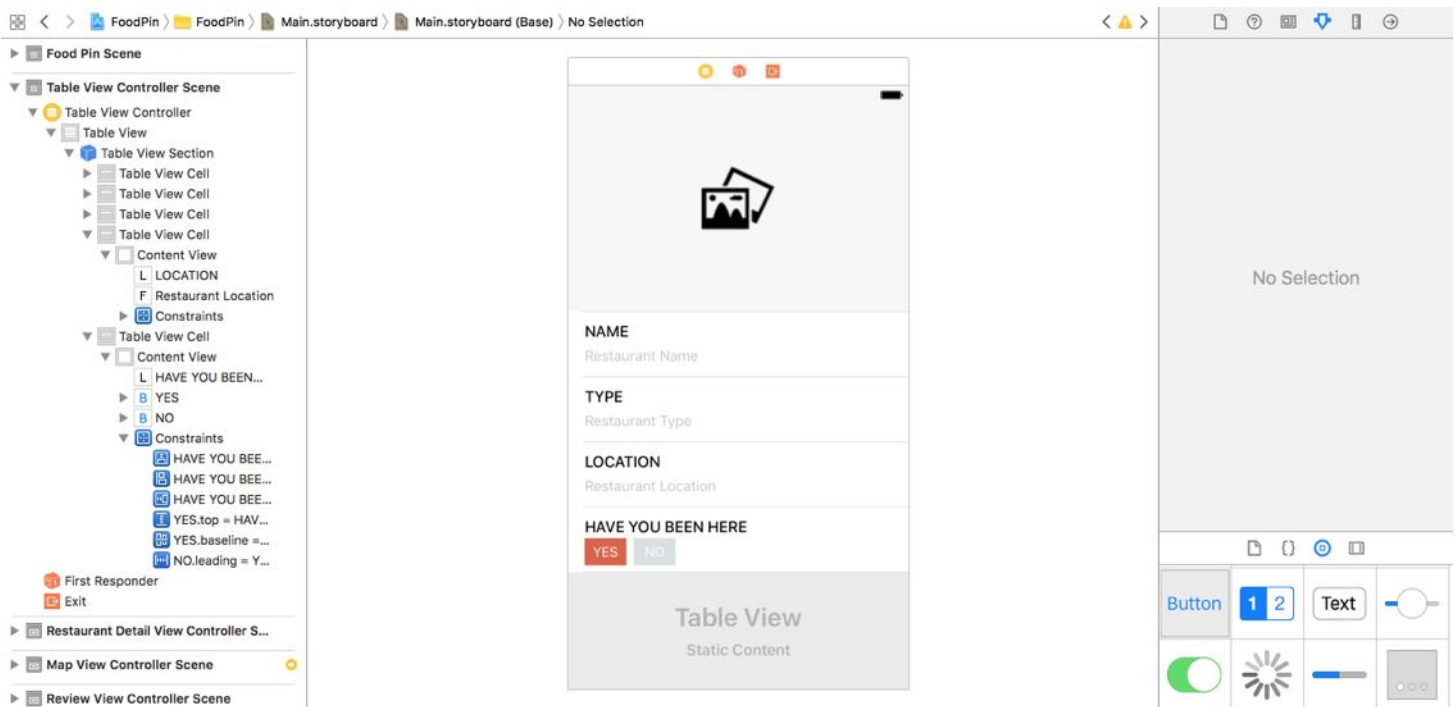


Figure 18-5. Sample cell design

To finish the layout of the screen, embed the table view controller in a navigation controller. Go up to the Xcode menu, and select the table view controller and select Editor > Embed in > Navigation Controller. Set the title of navigation bar to `New Restaurant`.

Without writing a single line of code, you create a form by using a static table view. You're not limited to create forms with static table views. You can apply the same technique to create screens like Settings.

Note: If you find it hard to create the UI design, you can download the starter project from <http://www.appcoda.com/resources/swift3/FoodPinStaticTableStarter.zip> (<http://www.appcoda.com/resources/swift3/FoodPinStaticTableStarter.zip>) for reference.

Adding a Segue

So far we just created a standalone table view controller. We expect to bring up this controller when a user taps a `+` button in the top-right corner of the main view. Obviously, we need to connect the button with the "New Restaurant" controller using a segue. In the Interface

Builder editor, first drag a bar button item from the Object library to the navigation bar of the Food Pin controller. In the Attribute inspector, change the identifier to `Add`, and you'll see a `+` icon.

Note: A bar button (`UIBarButtonItem`) is very similar to a standard button (`UIButton`). However, a bar button is specifically designed for navigation bars and toolbars.

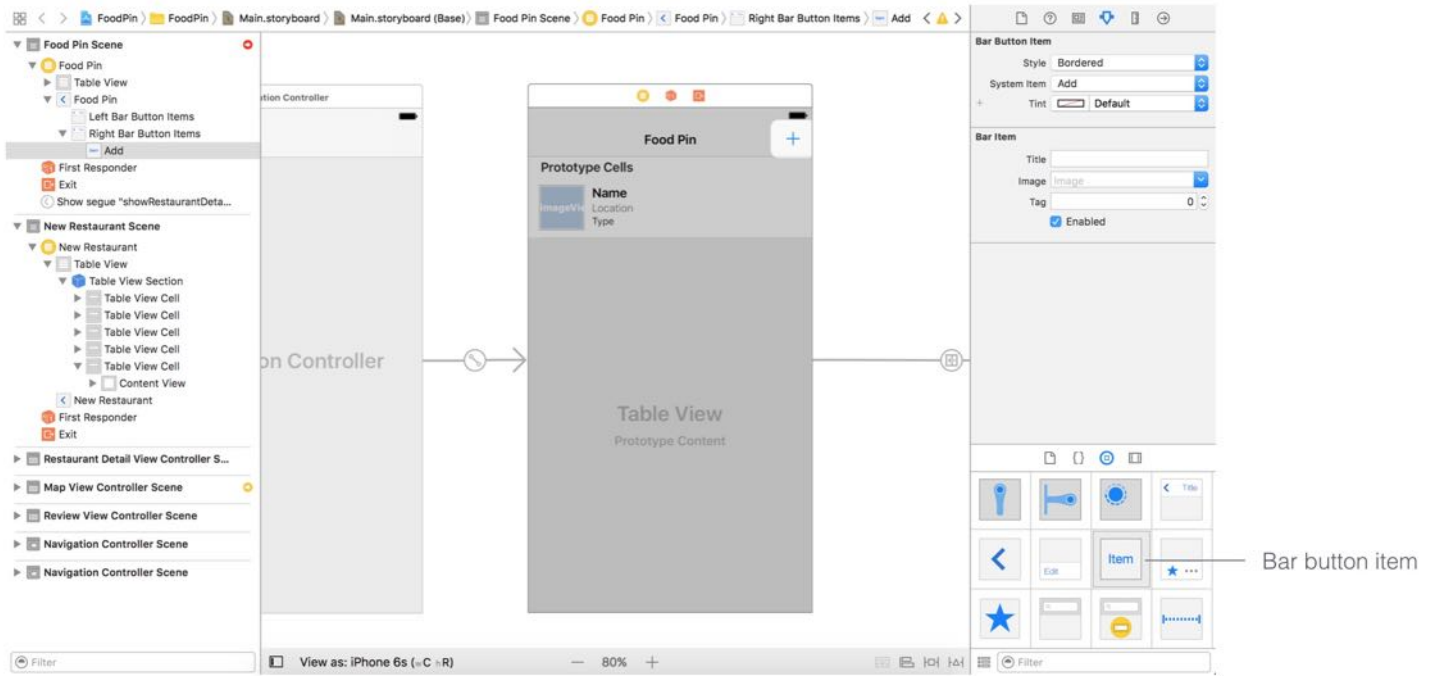


Figure 18-6. Add a bar button item in the navigation bar of the main screen

Next, hold the control key, and drag from the `+` icon to the navigation controller that embeds the *New Restaurant* view controller. Release the buttons and select *present modally* for the segue type. Set the identifier of the segue to `addRestaurant` in the Attribute inspector.

Quick tip: You can right-click any empty area on the storyboard, and select the zoom out option to have an overview of the storyboard.

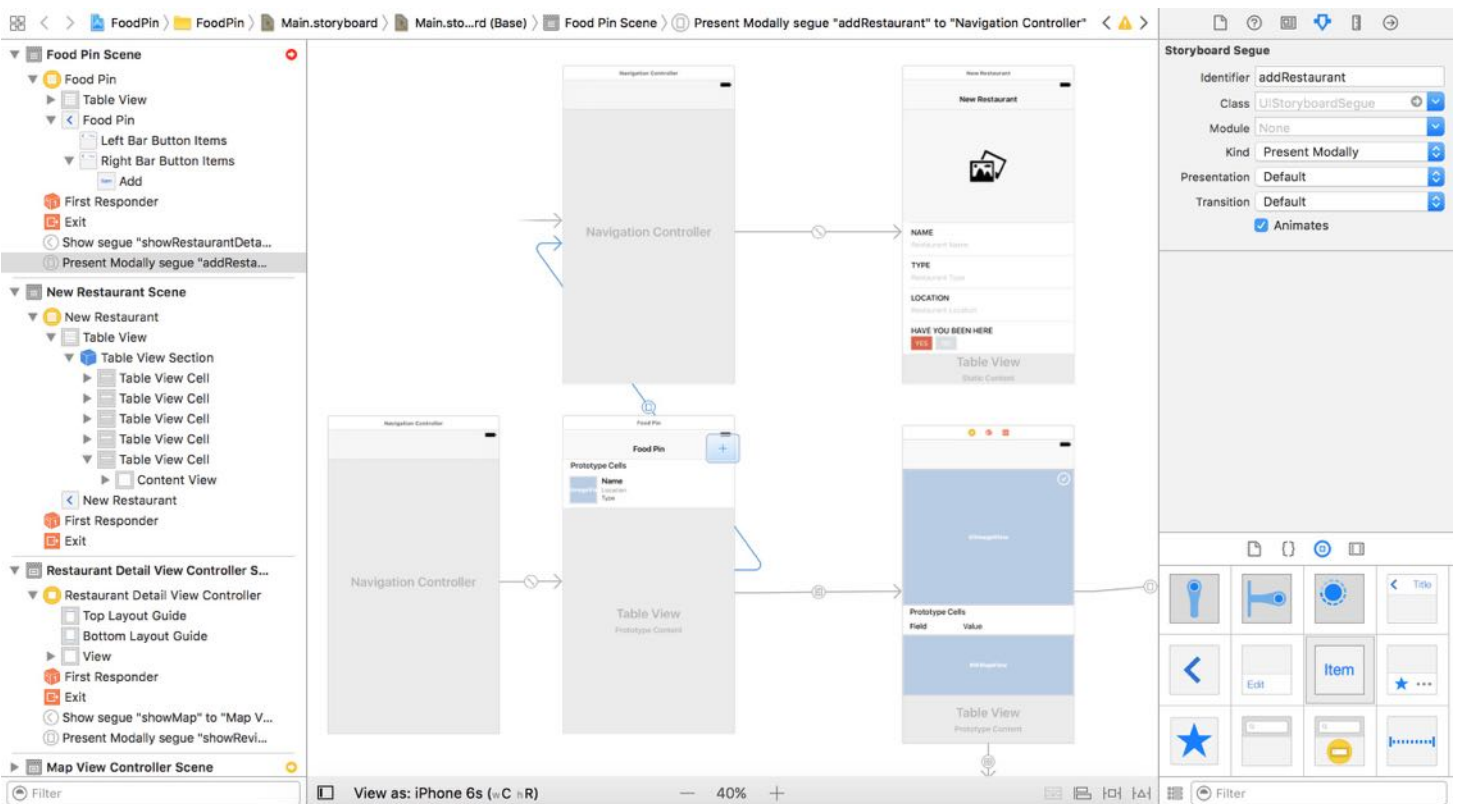


Figure 18-7. Adding a segue for the + icon

Similar to the modal view controller we built before, we need to provide a way for users to dismiss it. In the navigation bar of the *New Restaurant* view controller, add a bar button item to the top-left corner. In the Attribute inspector, set its identifier to `cancel` and tint to `white`.

When a user taps the *Cancel* button, the modal view will be dismissed. In order to do that, we'll define an unwind segue. Select the `RestaurantTableViewController.swift` file and define the following unwind action:

```
@IBAction func unwindToHomeScreen(segue:UIStoryboardSegue) {
}
```

Once you add this method, Interface Builder can recognize the unwind action. Now, go back to `Main.storyboard`. Hold control key and drag from the *Cancel* button to the Exit icon of the scene dock (see figure 18-8). In the popover menu, select the `unwindToHomeScreenWithSegue:` option for the action segue.

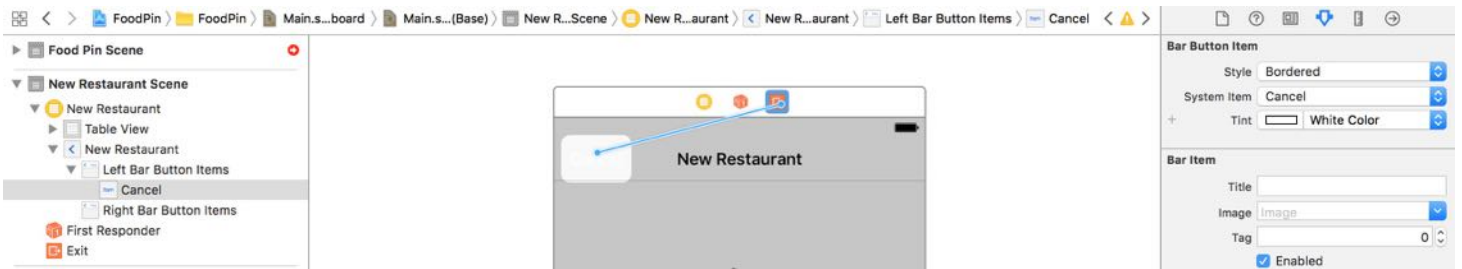


Figure 18-8. Adding an unwind segue for the Cancel button

Now let's have a quick test. Compile and run the project. After launching the app, tap the + icon. You should see the New Restaurant screen.

Displaying the Photo Library Using UIImagePickerController

When the first table view cell (i.e. the one with camera) is tapped, we want to bring up the built-in photo library and let users choose a photo. The UIKit framework provides a convenient API called `UIImagePickerController` for loading photos from the photo library. What's great is that the same API can be used to bring up a camera interface for taking a picture.

The simulator doesn't support the camera feature. If you want to test an app that utilizes the built-in camera, you'll need a real iOS device.

Note: Starting from Xcode 7, you no longer need to enroll into the Apple Developer Program before you can test your app on a real iOS device. I will show you how to deploy an app to the device for testing in chapter 25.

To keep things simple, we will only use `UIImagePickerController` for choosing saved images. First, create a new class named `AddRestaurantController` and set it as a subclass of `UITableViewController`. In the storyboard, select the Add Restaurant controller and set its custom class to `AddRestaurantController` in the Identity inspector.

In the `AddRestaurantController.swift` file, remove the following generated methods because we do not need them for the static table view:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    // #warning Incomplete implementation, return the number of sections
```



```

    return 0
}

override fun tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    // #warning Incomplete implementation, return the number of rows
    return 0
}

```

Next, add the following method to detect the touch and load the photo library:

```

override fun tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    if indexPath.row == 0 {
        if UIImagePickerController.isSourceTypeAvailable(.photoLibrary) {
            let imagePicker = UIImagePickerController()
            imagePicker.allowsEditing = false
            imagePicker.sourceType = .photoLibrary

            present(imagePicker, animated: true, completion: nil)
        }
    }
}

```

As we've discussed before, the `tableView(_:didSelectRowAt:)` method is called when a cell is selected. In this case, we only want to bring up the photo library when the first cell is selected. Therefore, we have a conditional check at the very beginning. To load up the photo library, all you need to do is create an instance of `UIImagePickerController` and set its `sourceType` to `.photoLibrary`. You then call up the `present(_:animated:completion:)` method to bring up the photo library.

That's it. Easy, right? Sometimes, the user may not allow you to access the photo library. As a good practice, you should always use the class method `isSourceTypeAvailable` to verify if a particular media source is available.

If you compile and run the app, you'll end up with an error when tapping the photo cell like this:

```

[access] This app has crashed because it attempted to access privacy-sensitive data without a usage description. The app's Info.plist must contain an NSPhotoLibraryUsageDescription key with a string value explaining to the user how the app uses this data.

```

In iOS 10 or later, for privacy reasons, you have to explicitly describe the reason your app accesses the user's photo library. If you fail to do so, you will see the above error.

To fix that, you need to add a key (`NSPhotoLibraryUsageDescription`) in the `Info.plist` file and provide your reason.

Now select `Info.plist` in the project navigator. Right click any blank area in the editor and select *Add Row*. Choose "Privacy - Photo Library Usage Description" for the key and set the value to:

You need to grant the app access to your photo library so you can pick your favorite restaurant photo.

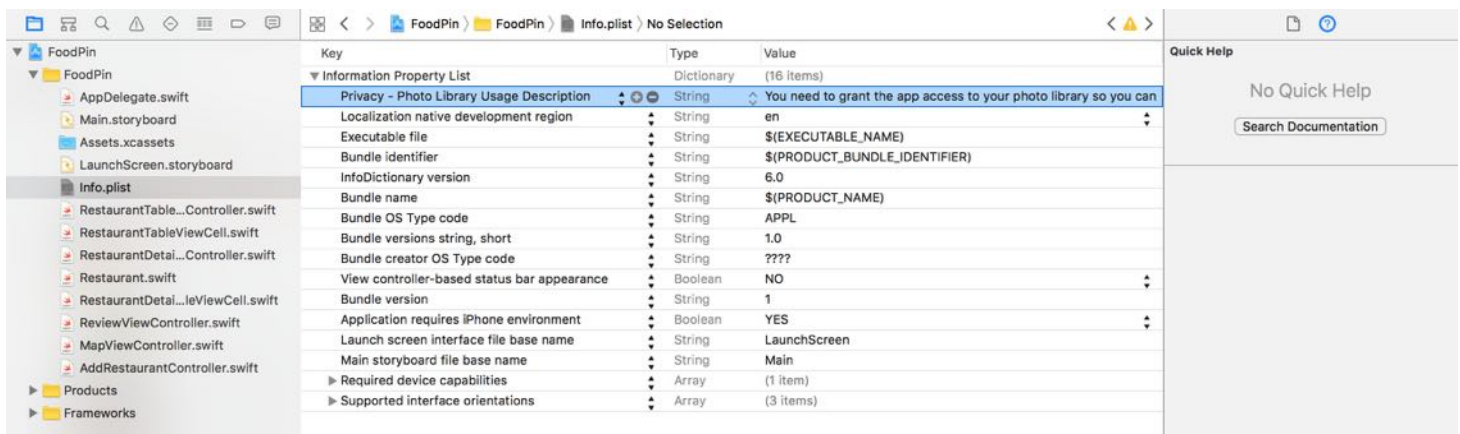


Figure 18-9. Add a key to specify the reason your app accesses the photo library

Now compile and run the app. Tapping the photo cell should bring up the built-in photo library. When prompted, remember to tap OK to enable your app to access the photo library. The simulator already comes with several sample photos. If you want to add your own photos, drag them from Finder to the simulator. This will automatically add the photos to the Photos app of the simulator.

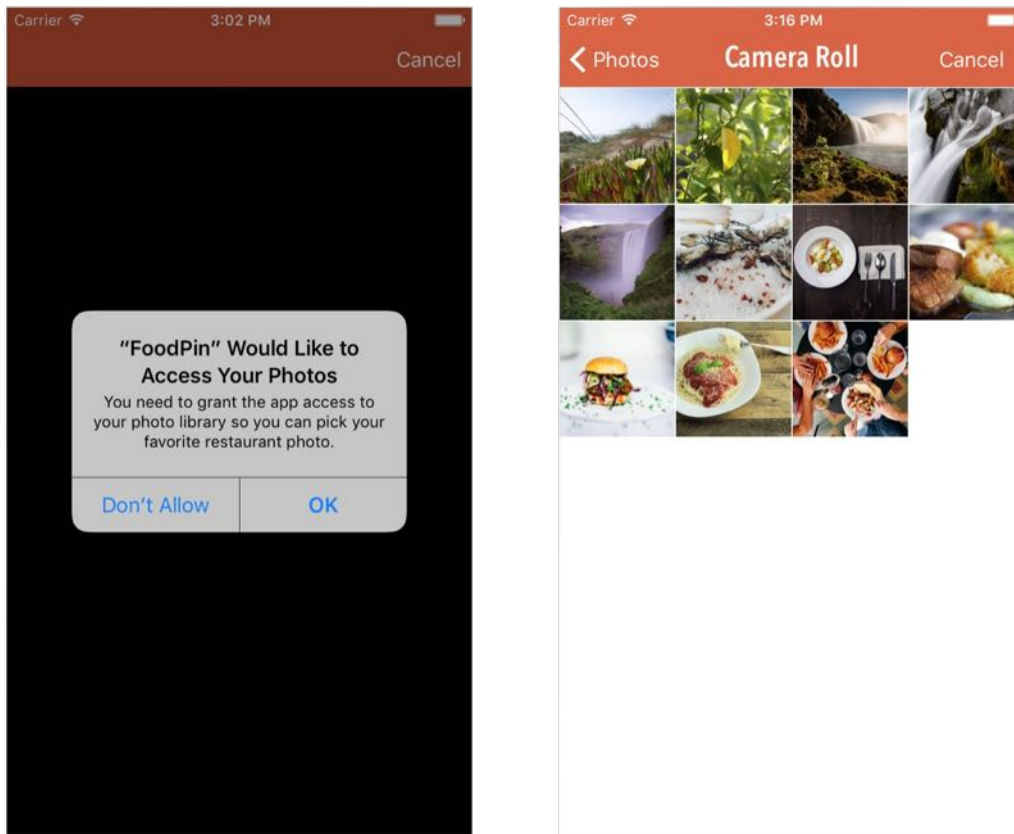


Figure 18-10. Loading the built-in photo library in FoodPin app

What if you want to let your user take picture?

If you want to allow the user to take a photo instead of choosing a saved image, simply set the `sourceType` to `.camera`. In this case, `UIImagePickerController` will show up a camera interface for taking picture.

Adopting the `UIImagePickerControllerDelegate` Protocol

If you select a photo from the photo library, it wouldn't show up in the image view. How do you know which photo the user selects? To interact with the image picker interface, the

`AddRestaurantController` class has to adopt two delegates: `UIImagePickerControllerDelegate` and `UINavigationControllerDelegate`.

```
class AddTableViewCell: UITableViewController,
UIImagePickerControllerDelegate, UINavigationControllerDelegate
```

When a user chooses a photo from the photo library, the

`imagePickerController(_:didFinishPickingMediaWithInfo:)` method of the delegate is called.

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : Any]) {

}
```

By implementing the method, we can get the selected photo from the method's parameter. Before implementing the method, let's declare an outlet variable for the image view. Later we can set the image view with the chosen image.

```
@IBOutlet var photoImageView: UIImageView!
```

Go to the Interface Builder and connect the image view of the cell with the `photoImageView` outlet variable.

Next, implement the delegate method as follows:

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : Any]) {

    if let selectedImage = info[UIImagePickerControllerOriginalImage] as?
UIImage {
        photoImageView.image = selectedImage
        photoImageView.contentMode = .scaleAspectFill
        photoImageView.clipsToBounds = true
    }

    dismiss(animated: true, completion: nil)
}
```

When the method is called, the system passes you an `info` dictionary object that contains the selected image. `UIImagePickerControllerOriginalImage` is the key of the image selected by the user.

The above code assigns the image view with the selected image. We also change the content mode so that the image is displayed in *aspect fill* mode. Lastly, we call the `dismiss` method to dismiss the image picker.

Don't forget to add this line of code in the `tableView(_:didSelectRowAt:)` method, immediately

after the declaration of `imagePicker` :

```
imagePicker.delegate = self
```

Now you're ready to test the app. Try to select a photo from the album and it should be displayed right in the image view.

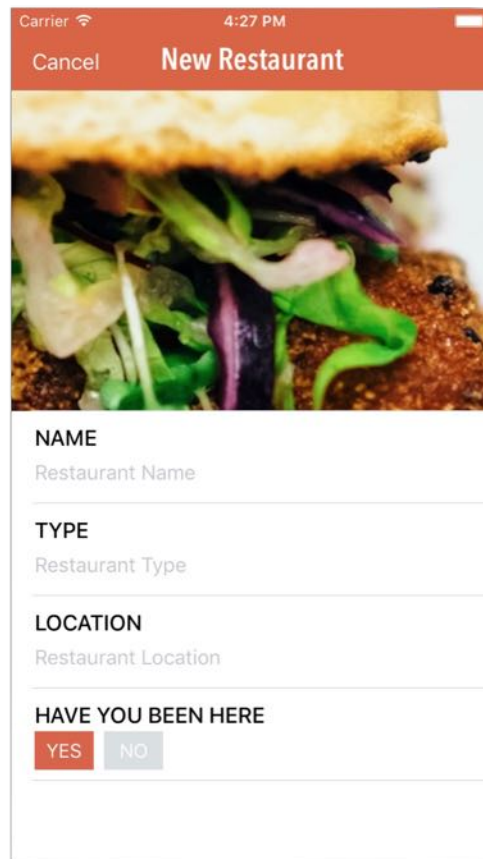


Figure 18-11. Displaying the selected image in the image view

Defining Auto Layout Constraints Programmatically

Unfortunately, the image was not perfectly displayed. It is because some layout constraints are missing. Let's first revisit the layout constraints of the image view. If you go to `Main.storyboard` and take a look at the image view, we have just defined two constraints to center the image in both vertical and horizontal directions.

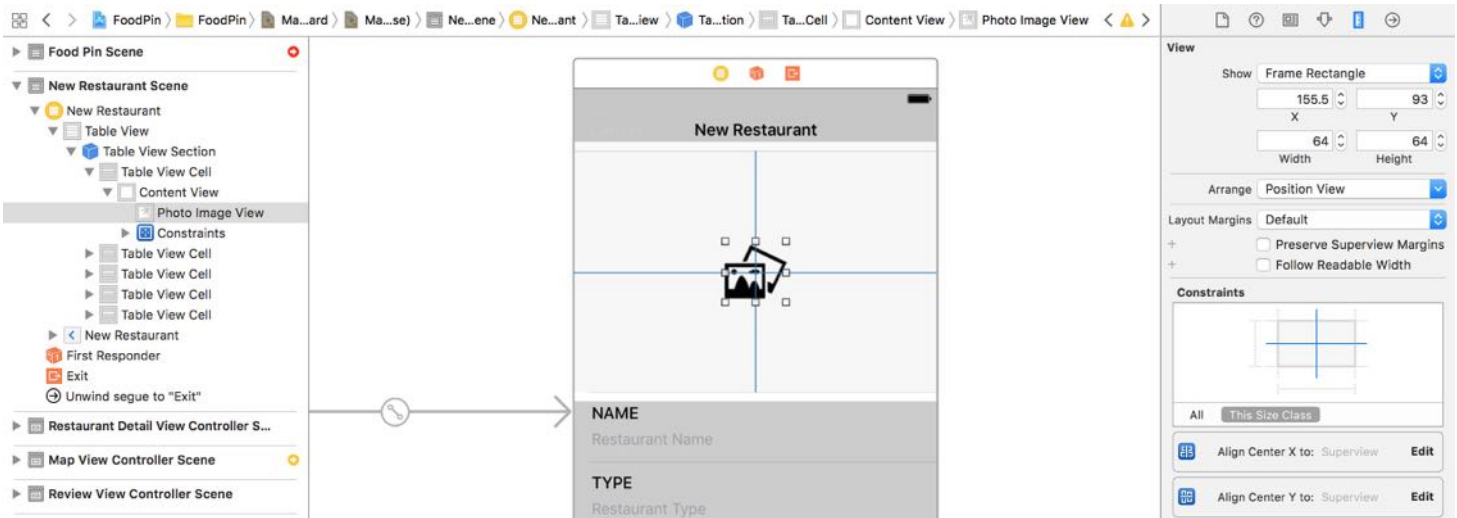


Figure 18-12. Layout constraints for the image view

When the app loads the selected image from the photo library, it tries to satisfy the constraints and center the image in the cell. But if the image is larger than the viewable area of the image view, it will not be fully displayed, just like the one shown in figure 18-11.

To fix the issues, we have to add a few layout constraints so that the image is bound within the viewable area. Instead of defining the layout constraints visually through the Interface Builder editor, you can add the constraints by using an API called `NSLayoutConstraint`. This time let us see how to create them programmatically.

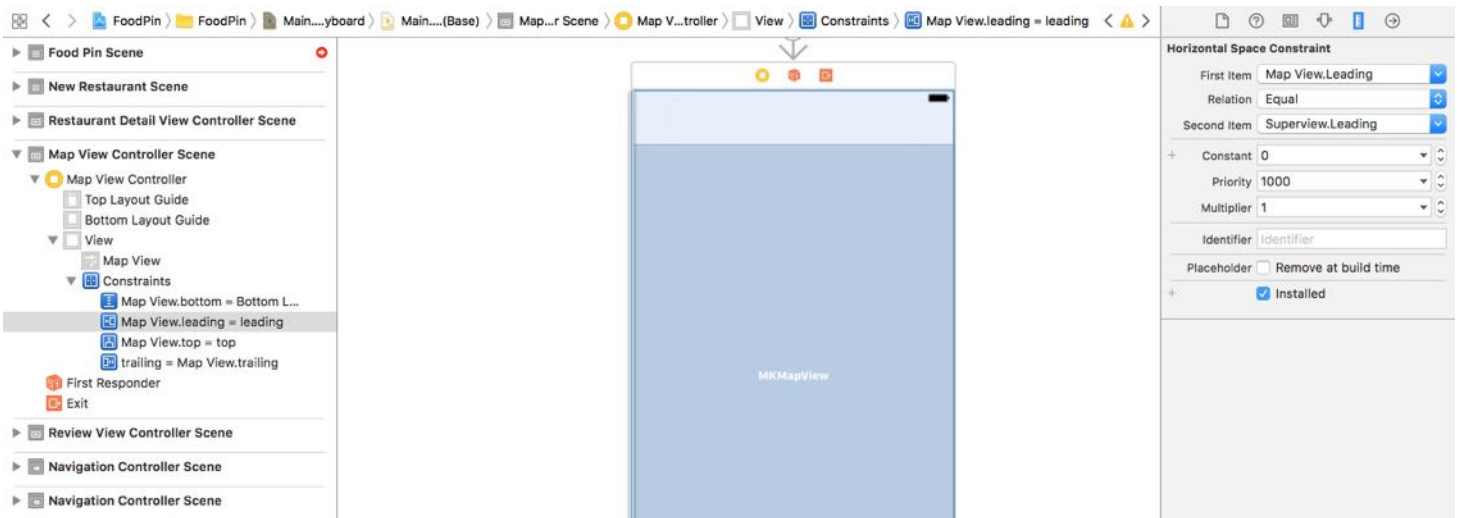


Figure 18-13. Sample spacing constraints for the map view

Figure 18-13 shows a sample layout constraint that we have created before using Interface Builder. The same constraint can be written using `NSLayoutConstraint` like this:

```
let leadingConstraint = NSLayoutConstraint(item: mapView, attribute:
NSLayoutConstraintAttribute.leading, relatedBy: NSLayoutConstraintRelation.equal, toItem:
mapView.superview, attribute: NSLayoutConstraintAttribute.leading, multiplier: 1,
constant: 0)
leadingConstraint.isActive = true
```

A layout constraint actually defines a relationship between two user interface objects. If you compare the constraint's properties defined in the Attribute inspector with those defined in the code, you should reveal their relationship. The `item` parameter corresponds to *First item*, `attribute` corresponds to the item after the dot in *First item*, `relatedBy` corresponds to *Relation*, `toItem` corresponds to *Second Item*, `attribute` corresponds to the item after the dot in *Second item*, `multiplier` corresponds to *Multiplier*, and `constant` corresponds to *Constant*.

By default, the constraint is not activated after instantiation. You have to set its `isActive` property to `true` in order to activate it.

Now back to the image view, we have to add four layout constraints for the top, bottom, leading (i.e. left), and trailing (i.e. right) sides of the image view. Update the delegate method with the following code:

```
func imagePickerController(_ picker: UIImagePickerController,
didFinishPickingMediaWithInfo info: [String : AnyObject]) {

    if let selectedImage = info[UIImagePickerControllerOriginalImage] as?
UIImage {
        photoImageView.image = selectedImage
        photoImageView.contentMode = .scaleAspectFill
        photoImageView.clipsToBounds = true
    }

    let leadingConstraint = NSLayoutConstraint(item: photoImageView, attribute:
NSLayoutConstraintAttribute.leading, relatedBy: NSLayoutConstraintRelation.equal, toItem:
photoImageView.superview, attribute: NSLayoutConstraintAttribute.leading, multiplier: 1,
constant: 0)
    leadingConstraint.isActive = true

    let trailingConstraint = NSLayoutConstraint(item: photoImageView,
```



```

attribute: NSLayoutConstraint.trailing, relatedBy: NSLayoutConstraint.equal,
 toItem: photoImageView.superview, attribute: NSLayoutConstraint.trailing,
 multiplier: 1, constant: 0)
    trailingConstraint.isActive = true

    let topConstraint = NSLayoutConstraint(item: photoImageView, attribute:
NSLayoutConstraint.top, relatedBy: NSLayoutConstraint.equal, toItem:
photoImageView.superview, attribute: NSLayoutConstraint.top, multiplier: 1,
constant: 0)
    topConstraint.isActive = true

    let bottomConstraint = NSLayoutConstraint(item: photoImageView, attribute:
NSLayoutConstraint.bottom, relatedBy: NSLayoutConstraint.equal, toItem:
photoImageView.superview, attribute: NSLayoutConstraint.bottom, multiplier: 1,
constant: 0)
    bottomConstraint.isActive = true

    dismiss(animated: true, completion: nil)
}

```

Here we programmatically define and activates four layout constraints for the image view so that the image is bound within the cell. Now let's run the app again. The app should now be able to display the selected image properly.

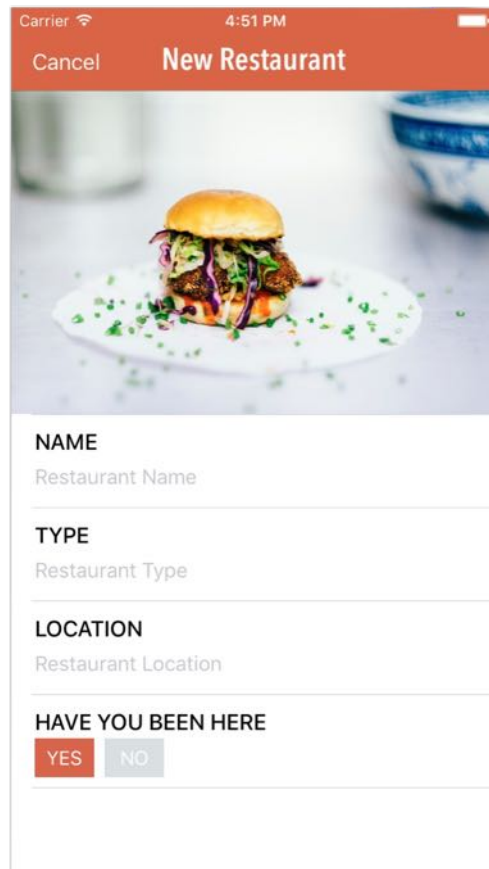


Figure 18-14. The image is now scaled and displayed perfectly

Your Exercise

So far, we only handle the image cell of the input form. Your exercise is to add a *Save* button in the top-right corner of the navigation bar. When the user taps the button, you get the values of the text fields, validate them and print them out to the console.

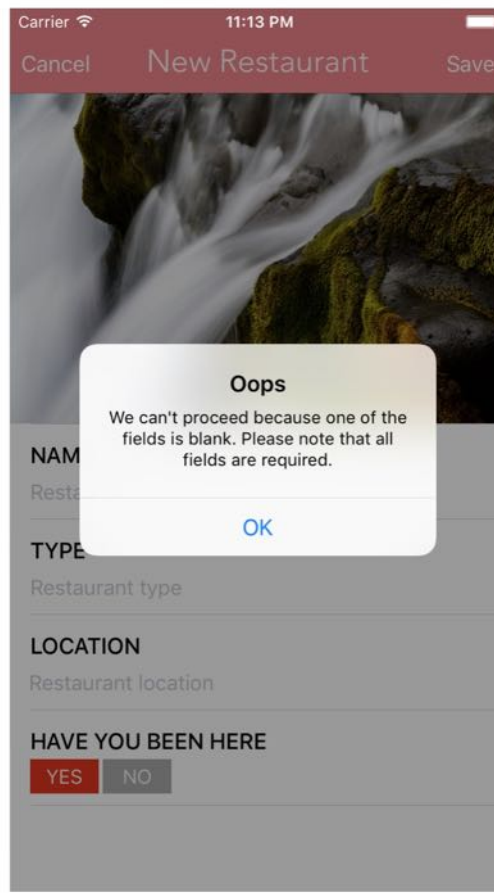
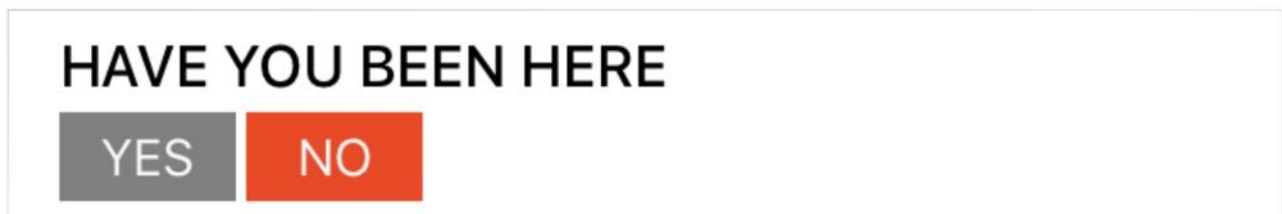


Figure 18-15. Sample alert for the form validation

On top of that, you need to handle the selection of Yes/No button in the *Have You Been Here* field. When the user selects No, the background color of the No button will be changed to red. Conversely, the background color of the Yes button will be changed to gray.



This exercise is more challenging than the earlier ones, so I will give you five hints. The first hint is that you will need to declare the following outlet variables in the `AddRestaurant` class and connect each of them with the corresponding UI component:

```
@IBOutlet var nameTextField:UITextField!
```

```
@IBOutlet var typeTextField:UITextField!  
@IBOutlet var locationTextField:UITextField!  
@IBOutlet var yesButton:UIButton!  
@IBOutlet var noButton:UIButton!
```

The next hint is that you will need to refer to the documentation of `UITextField` (https://developer.apple.com/library/ios/documentation/UIKit/Reference/UITextField_Class/). By reading the API document, you will find out which attribute is used for storing the text of the text field.

Thirdly, you have to declare a `save` action method in the `AddRestaurant` class. When the *Save* button is tapped, this action method will be called. So you will have to connect the button with the action method in Interface Builder. In the method, you will write code to perform a couple of things:

1. Form validation - check if the input text is blank. Otherwise, use `UIAlertController` to present an alert prompt.
2. Print out the form values to console. Here is a sample output:

```
Name: Optional("For Kee Restaurant")  
Type: Optional("Hong Kong Style")  
Location: Optional("Hong Kong")  
Have you been here: true
```

Next, you can set an identifier for the unwind segue and execute it programmatically. This allows you to dismiss the Add Restaurant view controller and return to the home screen.

```
performSegue(withIdentifier: "unwindToHomeScreen", sender: self)
```

Alternatively, you can also invoke the following method to dismiss the controller:

```
dismiss(animated: true, completion: nil)
```

You will have to create another action method to toggle the color of the *YES* and *NO* buttons. As the last hint, here is the skeleton of the action method:

```
@IBAction func toggleBeenHereButton(sender: UIButton) {  
    // Yes button clicked  
    if sender == yesButton {  
        isVisited = true  
    }  
}
```

```
        // Change the backgroundColor property of yesButton to red
        // Change the backgroundColor property of noButton to gray
    } else if sender == noButton {
        isVisited = false
        // Change the backgroundColor property of yesButton to gray
        // Change the backgroundColor property of noButton to red
    }
}
```

I have included the solution for you. You can find the download link at the end of the chapter. However, I highly recommend you to work on the exercise. It's gonna be fun to figure out your own implementation.

Summary

In this chapter, we demonstrate how to create a static table view using Interface Builder. While you can use a table view to display dynamic data from data source, a static table view provides a great way to display a finite quantity of data that is already known beforehand. You also learned how to access the built-in photo library. `UIImagePickerController` makes it so simple for developer to access the built-in camera and photo library.

Furthermore, we walked through `NSLayoutConstraint` together. You should now have some ideas about how to create layout constraints programmatically. You commonly use Interface Builder to define the layout constraints. That's good enough for most layouts. However, when you need to build a dynamic UI that changes within the application, you will need to modify the layout constraints at runtime. I only give you a brief introduction to `NSLayoutConstraint`. You can refer to the official documentation (https://developer.apple.com/library/ios/documentation/AppKit/Reference/NSLayoutConstraint_Class/) for details.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinStaticTableView.zip>. For the solution to the exercise, you can download it from <http://www.appcoda.com/resources/swift3/FoodPinStaticTableViewExercise.zip>.

Next up, we will talk about Core Data and see how to save the restaurant data in database.

Chapter 19

Working with Core Data



Learn not to add too many features right away, and get the core idea built and tested.

– Leah Culver

Congratulations on making it this far! By now you've already built a simple app for users to list their favorite restaurants. If you've worked on the previous exercise, you should understand the fundamentals of how to add a restaurant; I've tried to keep things simple and focus on the basics of UITableView. Up to this point, all restaurants have been predefined in the source

code and stored in an array. If you want to save a restaurant, the simplest way is to add a new restaurant to the existing restaurants array.

However, if you do it this way, you can't save the new restaurant permanently. Data holding in memory (e.g. array) is volatile. Once you quit the app, all the changes are gone. We need to find a way to save the data in a persistent manner.

To save the data permanently, we'll need to save in a persistent storage-like file or database. By saving the data to a database, for example, the data will be safe even if the app quits or crashes. Files are another way to save data, but they are more suitable for storing small amounts of data that do not require frequent changes. For instance, files are commonly used for storing application settings. If you open the *Support Files* folder in project navigator, you'll find the Info.plist file. This property file is used for storing your project settings.

The FoodPin app may need to store thousands of restaurant records. Users may also add or remove the restaurant records quite frequently. In this case, a database is a suitable way to handle a large set of data. In this chapter, I will walk you through the Core Data framework and show you how to use it to manage data in database. You will make a lot of changes to your existing FoodPin project, but after going through this chapter your app will allow users to save their favorite restaurants persistently.

If you haven't done the previous exercise, you can download this project's templates (<http://www.appcoda.com/resources/swift3/FoodPinStaticTableViewExercise.zip>) to start with.

What is Core Data?

When we talk about persistent data, you probably think of databases. If you are familiar with Oracle or MySQL, you know that a relational database stores data in the form of tables, rows and columns; your app talks to the database by using an SQL (Structured Query Language) query. However, don't mix up Core Data with databases. Though SQLite database is the default persistent store for Core Data on iOS, Core Data is not exactly a relational database - it is actually a framework that lets developers interact with database (or other persistent storage) in an object-oriented way.

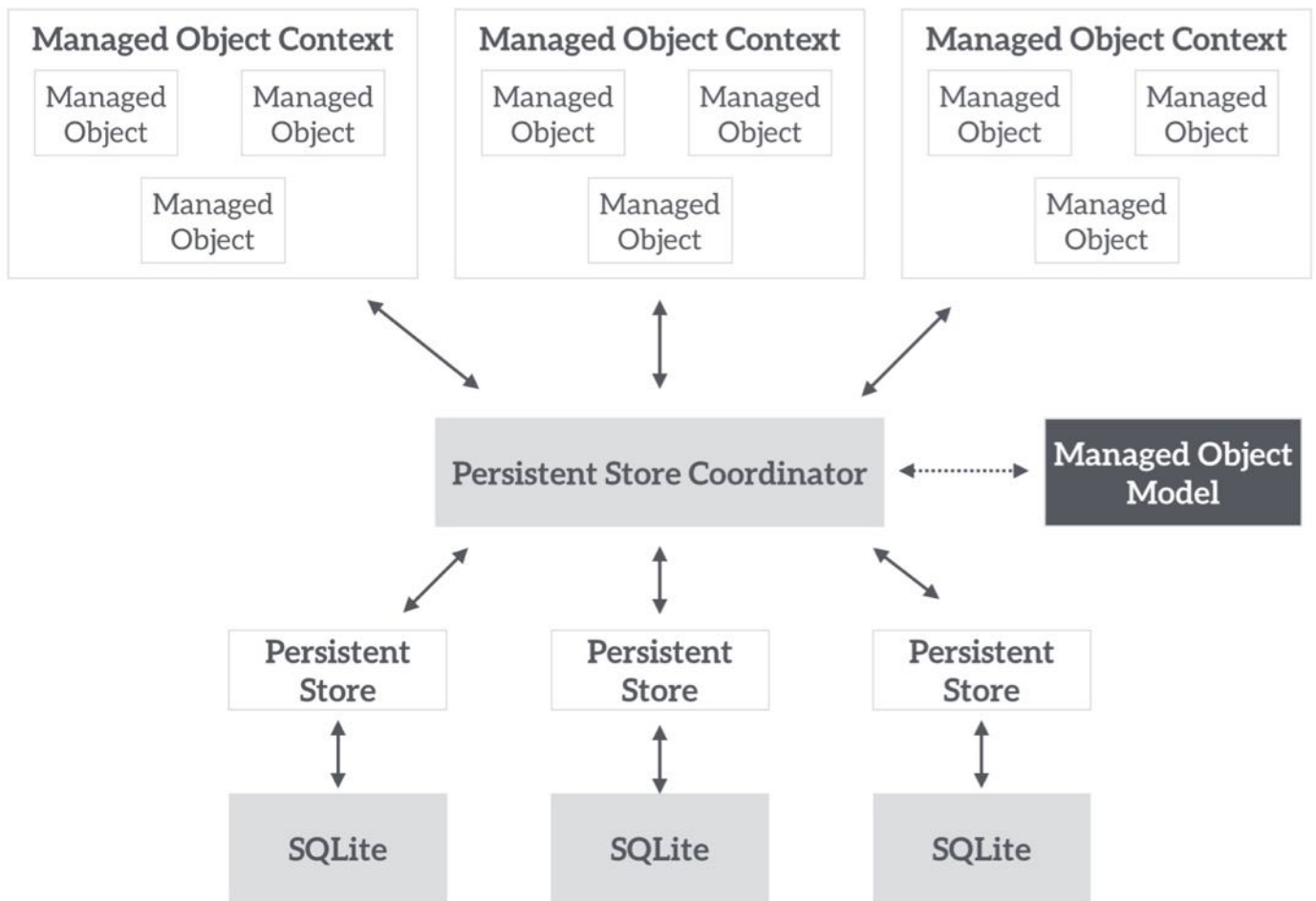
Take the FoodPin app as an example. If you want to save the data to database, you are responsible for writing the code to connect to the database and retrieve or update the data using SQL. This would be a burden for developers, especially for those who do not know SQL.

Core Data provides a simpler way to save data to a persistent store of your choice. You can map the objects in your apps to the table in the database. Simply put it allows you to manage records (select/insert/update/delete) in the database without even knowing any SQL.

Core Data Stack

Before we start working on the project, you need to first have a basic understanding of the Core Data Stack (see figure 19-1):

- **Managed Object Context** – Think of it as a *scratch pad* or temporary memory area containing objects that interact with data in persistent store. Its job is to manage objects created and returned using Core Data framework. Among the components in the Core Data stack, the managed object context is the one you'll work directly with most of the time. In general, whenever you need to fetch and save objects in the persistent store, the context is the first component you'll interact with.
- **Persistent Store Coordinator** – SQLite is the default persistent store in iOS. However, Core Data allows developers to set up multiple stores containing different entities. The persistent store coordinator is the party responsible for managing different persistent object stores and saving the objects to the stores. Forget about it if you don't understand what it is; you won't interact with the persistent store coordinator directly when using Core Data.
- **Managed Object Model** – This describes the schema that you use in the app. If you have some background in databases, think of this as the database schema. However, the schema is represented by a collection of objects (also known as entities). For example, a collection of model objects can be used to represent the collection of restaurants in the FoodPin app. In Xcode, the managed object model is defined in a file with the extension `.xcdatamodeld`. You can use the visual editor to define the entities and their attributes and relationships.
- **Persistent Store** - This is the repository in which your data is actually stored. Usually it's a database, and SQLite is the default database. But it can also be a binary or XML file.



That looks complicated, right? Definitely. Therefore, iOS 10 introduces a new class called `NSPersistentContainer` to simplify the management of Core Data stack in your apps. `NSPersistentContainer` is the class you will deal with for saving and retrieving data.

Feeling confused? No worries. You will understand what I mean as we convert the FoodPin app from arrays to Core Data.

Using Core Data Template

The simplest way to use Core Data is to enable the Core Data option during project creation. Xcode will generate the required code in `AppDelegate.swift` and create the data model file for Core Data.

Choose options for your new project:

Product Name: CoreDataDemo

Team: None

Organization Name: AppCoda

Organization Identifier: com.appcoda

Bundle Identifier: com.appcoda.CoreDataDemo

Language: Swift

Devices: iPhone

Use Core Data

Include Unit Tests

Include UI Tests

Cancel Previous Next

If you create a *CoreDataDemo* project with Core Data option enabled, you will see the following variables and method generated in the `AppDelegate` class:

```
// MARK: - Core Data stack

lazy var persistentContainer: NSPersistentContainer = {
    /*
     The persistent container for the application. This implementation
     creates and returns a container, having loaded the store for the
     application to it. This property is optional since there are legitimate
     error conditions that could cause the creation of the store to fail.
     */
    let container = NSPersistentContainer(name: "CoreDataDemo")
    container.loadPersistentStores(completionHandler: { (storeDescription,
error) in
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error
            appropriately.
            // fatalError() causes the application to generate a crash log and
            terminate. You should not use this function in a shipping application, although
```

it may be useful during development.

```
        /*
        Typical reasons for an error here include:
        * The parent directory does not exist, cannot be created, or
disallows writing.
        * The persistent store is not accessible, due to permissions or
data protection when the device is locked.
        * The device is out of space.
        * The store could not be migrated to the current model version.
        Check the error message to determine what the actual problem was.
        */
        fatalError("Unresolved error \(error), \(error.userInfo)")
    }
})
return container
}()

// MARK: - Core Data Saving support

func saveContext () {
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
            try context.save()
        } catch {
            // Replace this implementation with code to handle the error
appropriately.
            // fatalError() causes the application to generate a crash log and
terminate. You should not use this function in a shipping application, although
it may be useful during development.
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

The generated code provides a variable and a method:

- The variable `persistentContainer` is an instance of `NSPersistentContainer`, and has been initialized with a persistent store named "CoreDataDemo". Later, you will use this variable to interact with the Core Data stack.
- The `saveContext()` method provides data saving. When you need to insert/update/delete data in the persistent store, you will call up this method.

If you've used Core Data in older version of Xcode, you should find that the generated code has

been greatly simplified. `NSPersistentContainer` encapsulates the Core Data stack, and simplifies the way of using Core Data.

The question is how we can use this code template in our existing Xcode project. You can simply copy and paste the code into the `AppDelegate.swift` file of your Food Pin project, but you will need to make a minor change.

```
let container = NSPersistentContainer(name: "CoreDataDemo")
```

The original code template was generated for the `CoreDataDemo` project. Xcode names the SQLite and data model file using the project name. For the FoodPin project, we use the name *FoodPin* instead. So change the above line of code to the following:

```
let container = NSPersistentContainer(name: "FoodPin")
```

Finally, add the `import` statement at the beginning of the `AppDelegate` class to import the Core Data framework:

```
import CoreData
```

Quick note: For reference, you can also download this project template (<http://www.appcoda.com/resources/swift3/FoodPinCoreDataTemplate.zip>) to continue.

Creating the Data Model

Now that you've prepared the code for accessing the Core Data stack, let's move on to create the data model. In the project navigator, right-click the `FoodPin` folder and select `New File...`. Choose Core Data and select Data Model.

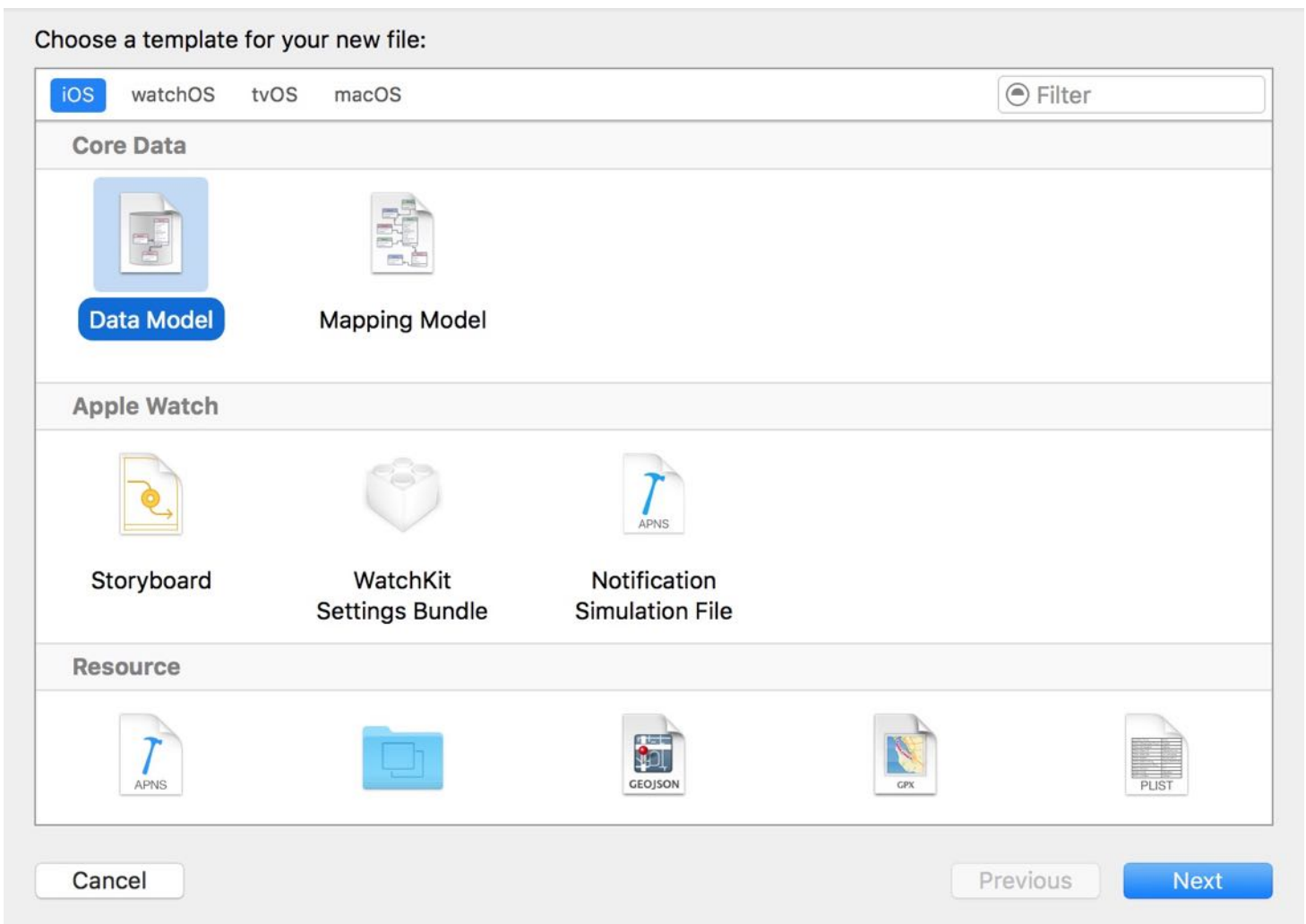


Figure 19-3. Creating the data model using Data Model template

Name the model `FoodPin` and click `create` to create the data model. Once created, you should find a file named `FoodPin.xcdatamodeld` in the project navigator. Select it to open the data model editor. From here, you can create entities for your data model.

As we would like to store the `Restaurant` object in database, we will create a `Restaurant` entity that matches the `Restaurant` class in our code. To create an entity, click the *Add Entity* button at the bottom of the editor pane and name the entity `Restaurant`.

In order to save the data from the `Restaurant` object to the database, we will add several attributes for the entity that align with the attributes of the object. Simply click the `+` button under the attributes section to create a new attribute. Add six attributes for the `Restaurant` entity including *name*, *type*, *location*, *image*, *isVisited*, and *rating*. Refer to figure 19-4 for

details.

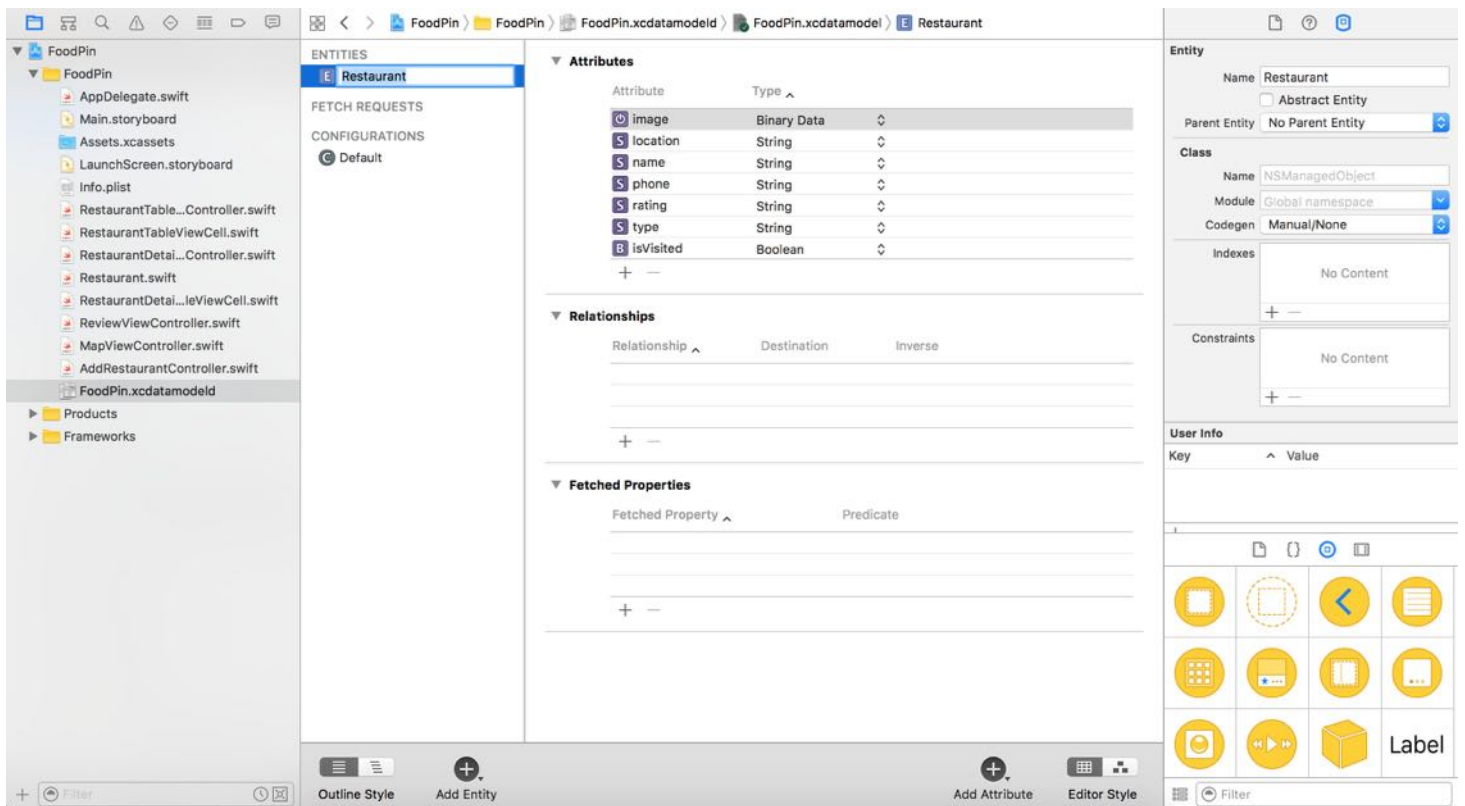


Figure 19-4. Adding attributes to the Restaurant entity

The attribute types of *name*, *type*, *location*, *phoneNumber*, *isVisited*, and *rating* are trivial, but why do we set the attribute type of *image* to Binary Data?

Presently, the restaurant images are bundled in the app, and managed by the asset catalog. This is why we can load an image by passing `UIImage` with the set name. When user creates a new restaurant, the image is loaded from an external source whether it's from the built-in photo library or taken from camera. In this case, we can't just store the file name. Instead, we save the actual data of the image into database. Binary data type is used for this purpose.

If you select a particular attribute, you can further configure its properties in the Data Model inspector. For example, the *name* attribute is a required attribute. You can uncheck the *Optional* checkbox to make it mandatory. For the FoodPin project, you can set the *name*, *type*, and *location* as required.

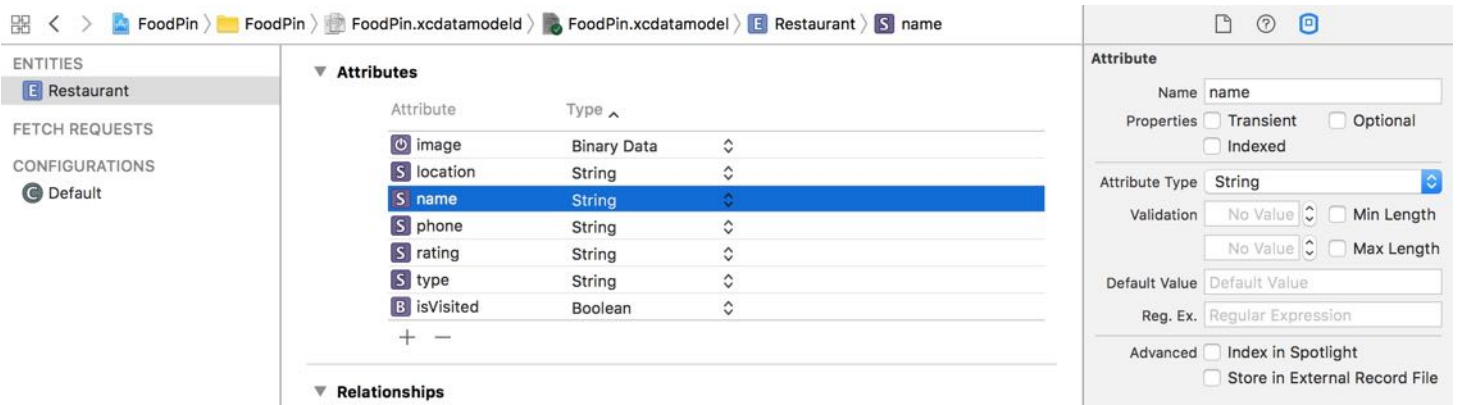


Figure 19-5. Editing attribute properties in the Data Model inspector

Creating Managed Objects

Model objects that tie into in the Core Data framework are known as *managed objects*, which are at the heart of any Core Data application. Now that you've created the managed object model, the next thing is to create the managed objects.

For the FoodPin project, you can manually convert the `Restaurant` class to a managed object class. In Xcode 8, however, you can let the development tool do it for you. By referencing the entity model, Xcode can automatically generate the managed object class for you.

Note: You're probably wondering why we have to create a managed object class. Do you remember the relationship between outlet variables and UI objects in Interface Builder? By updating the value of outlets, we can modify the content of UI objects. Here the manage objects is very similar to outlet variables. You can modify the entity's content by updating the managed object.

RestaurantMO

- name
- type
- location
- image
- isVisited
- rating
- phone

1 Create a RestaurantMO object and set its properties

2 Map class properties to the entity's attributes and store the data in database

Restaurant Entity

name Upstate

type Western

location New York

image 

isVisited true

rating good

phone 2323-223

Figure 19-6. Sample relationship between model object and entity

Now select the `Restaurant` entity and go to the Data Model inspector. You should see the Class section. Set the class name to `RestaurantMO` and the Codegen option to `Class Definition`.

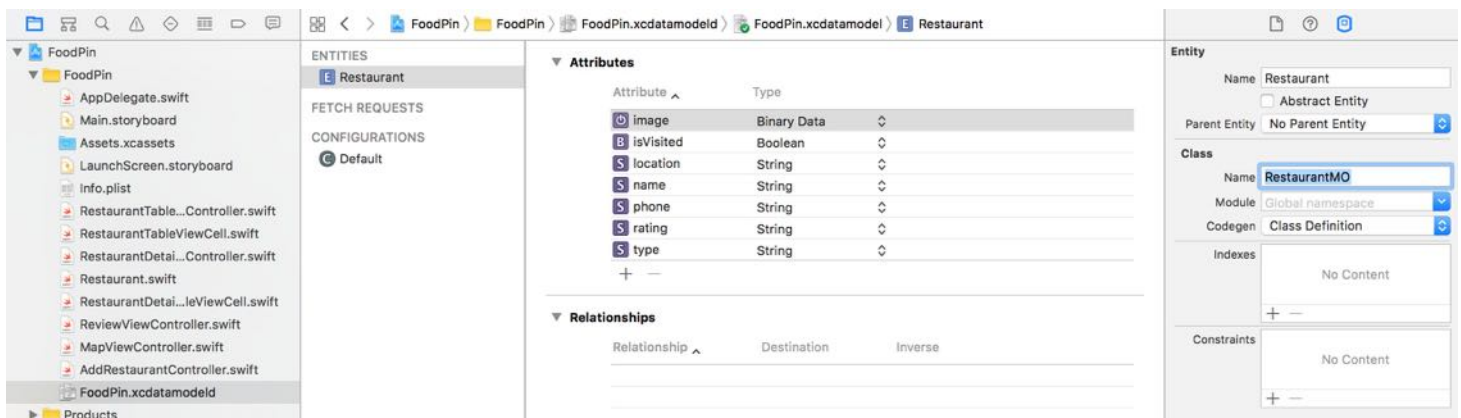


Figure 19-7. Assign the entity with the Restaurant class

That's it. Xcode will generate the class for you when you build the project. You can simply run the app or press command-B to build the project.

You will not find the generated class in the project navigator. It's stored somewhere else in the project folder. But you're now ready to use in your code.

Since we are going to save the restaurant object into the database, we now have to replace the original `Restaurant` class with the new `RestaurantMO` class.

Let's start with the `RestaurantTableViewController.swift` file. We no longer need to initialize the `restaurants` array with default values as we will pull the data from database. Therefore, declare the `restaurants` object like this:

```
var restaurants:[RestaurantMO] = []
```

Once you make the change, you will see quite a number of errors in Xcode. First, in `RestaurantTableViewController.swift`, you no longer use file name to load an image.

```
cell.thumbnailImageView.image = UIImage(named:  
restaurants[indexPath.row].image)
```

The image is now stored as a `Data` object. To load the image, instead of passing the image through the `named` parameter, initialize the `UIImage` object using the `data` parameter:

```
cell.thumbnailImageView.image = UIImage(data: restaurants[indexPath.row].image  
as! Data)
```

The same applies to the line of code in `tableView(_:editActionsForRowAt):`

```
if let imageToShare = UIImage(data: self.restaurants[indexPath.row].image as!  
Data) {
```

Also, the properties of `RestaurantMO` are now generated as an optional. So you will have to replace the `defaultText` variable with this line of code in order to unwrap the optional:

```
let defaultText = "Just checking in at " +  
self.restaurants[indexPath.row].name!
```

Next, let's move onto `RestaurantDetailViewController.swift`.

First, we have to change the type of the restaurant variable from `Restaurant` to `RestaurantMO` :

```
var restaurant:RestaurantMO!
```

Again, you will see some errors after changing the code. In the `viewDidLoad()` method, initialize the restaurant image using data instead of name:

```
restaurantImageView.image = UIImage(data: restaurant.image as! Data)
```

Also, you will need to unwrap the restaurant location when calling `geocodeAddressString` :

```
geoCoder.geocodeAddressString(restaurant.location!, completionHandler: {  
    placemarks, error in
```

For the `MapViewController` class, we will apply the similar change. Modify the type of the restaurant variable to `RestaurantMO` :

```
var restaurant:RestaurantMO!
```

Then fix any errors related to the change. Here is an example:

```
leftIconView.image = UIImage(data: restaurant.image as! Data)
```

Lastly, change the type of the `restaurant` variable in `ReviewViewController.swift` :

```
var restaurant:RestaurantMO?
```

In the `viewDidLoad()` method, modify the line of code that loads the image like this:

```
restaurantImageView.image = UIImage(data: restaurant.image as! Data)
```

Quick note: I encourage you to fix the errors on your own. But for reference, you can download the Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinCoreDataTemplate2.zip>.

As there is no restaurant data, your app should now display a blank table when launched. Next up, we'll implement the `AddTableViewController` class and save a new restaurant to database.

Working with Managed Objects

Now that we have converted our FoodPin project to use managed objects, the next question is how can we use the objects to save data into the database?

With the introduction of `NSPersistentContainer` in iOS 10, the complexity involved in setting up a Core Data stack had been dramatically reduced. Common operations, such as inserting a record to database, are a lot simpler. To save a restaurant, you basically need to handle a couple of things:

1. Create a `RestaurantMO` object with the view context of the persistent container and then set its properties as usual.

```
restaurant = RestaurantModel(context:
AppDelegate.persistentContainer.viewContext)
restaurant.name = "Upstate"
restaurant.type = "Cafe"
restaurant.location = "New York"
```

2. Next, you call the `saveContext()` method in `AppDelegate` to save the data to the database:

```
AppDelegate.saveContext()
```

That's it. As you can see, Core Data shields you away from the underlying logics of database management. You do not need to understand how to insert a record into the database using SQL. All is done by using the Core Data APIs.

Saving a New Restaurant to the Database

With some basic understanding about managed objects, let's update the `AddTableViewController` class to save a new restaurant to the database.

First, add the following `import` statement at the very beginning of

`AddTableViewController.swift` so that the class can utilize the Core Data framework:

```
import CoreData
```

Declare a restaurant variable in the `AddTableViewController` class:

```
var restaurant:RestaurantMO!
```

In the `save` method, apply what you've just learned to save the `restaurant` object into the persistent store. Insert the following code before calling the `dismiss` method:

```
if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
    restaurant = RestaurantMO(context:
appDelegate.persistentContainer.viewContext)
    restaurant.name = nameTextField.text
    restaurant.type = typeTextField.text
    restaurant.location = locationTextField.text
    restaurant.isVisited = isVisited

    if let restaurantImage = photoImageView.image {
        if let imageData = UIImagePNGRepresentation(restaurantImage) {
            restaurant.image = NSData(data: imageData)
        }
    }

    print("Saving data to context ...")
    appDelegate.saveContext()
}
```

The above code is pretty much the same as what we discussed in the previous section. However, there are a few lines of code that are new to you. First, it's the following line of code:

```
if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
```

The `persistentContainer` variable is declared in `AppDelegate.swift`. To access the variable, we have to first get a reference to `AppDelegate`. In iOS SDK, you can use

```
UIApplication.shared.delegate as? AppDelegate
```

 to get the `AppDelegate` object.

Next, it's related to the `image` property.

```
if let imageData = UIImagePNGRepresentation(restaurantImage) {
    restaurant.image = NSData(data: imageData)
}
```

Recalled that the image type of the Restaurant Entity is set to Binary Data, in this case, the image property of the generated `RestaurantMO` has a type of `NSData`.

Therefore, when we set the value of the image property, we have to retrieve the data of the selected image and convert it to an `NSData` object. The UIKit framework provides a set of built-

in functions for graphics operations. The `UIImagePNGRepresentation` function allows us to get the data of a specified image in PNG format. We then create the `NSData` object using the image data.

This is the piece of code you need to add a new restaurant to database. If you run the app now and save a new restaurant, the app should be able to save the record into database without any errors. However, your app is not ready to display the restaurant just added. That's what we're going to do next.

Fetching Data Using Core Data

To fetch data using Core Data, the simplest way is to create a fetch request and then use the `fetch` method provided by the view context:

```
if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
    let request: NSFetchedRequest<RestaurantMO> = RestaurantMO.fetchRequest()
    let context = appDelegate.persistentContainer.viewContext
    do {
        restaurants = try context.fetch(request)
    } catch {
        print(error)
    }
}
```

The generated `RestaurantMO` class has a built-in `fetchRequest()` method. When called, it returns you an `NSFetchedRequest` object that specifies the search criteria and which entity to search (here, it is the `Restaurant` entity).

With the fetch request, we can then call the `fetch` method of `viewContext` to retrieve data from a persistent store (here, it's the database).

Note: Swift comes with an exception-like model using try-throw-catch keywords. You use do-catch statement to catch errors and handle them accordingly. As you may notice, we put a try keyword in front of the method call. With the introduction of the new error handling model in Swift 2.0, some methods can throw errors to indicate failures. When we invoke a throwing method, you will need to put a try keyword in front of it. For details of error handling, please refer to the appendix.

You can put the above code in the `viewWillAppear` method to load the latest restaurant objects

from database. However, we're not going to use the convenient method for fetching the records. Instead, I'll introduce you another API called `NSFetchedResultsController`.

You may wonder why we don't just use the simple method - the primary reason is for performance. Every time a user adds a new record or removes a record from database, we load all restaurant records from database and re-display them in the table view. This is not an efficient way to manage the data. A better way to do that is like this:

- When adding a new record, we add a new row in the table view.
- When removing a record, we just remove that row from the table view.

`NSFetchedResultsController` is specially designed for managing the results returned from a Core Data fetch request, and providing data for a table view. It monitors changes to objects in the managed object context and reports changes in the result set to its delegate.

Let's see how to use `NSFetchedResultsController` to retrieve the restaurants.

In `RestaurantTableViewController.swift`, first import the CoreData framework:

```
import CoreData
```

Then adopt the `NSFetchedResultsControllerDelegate` protocol:

```
class RestaurantTableViewController: UITableViewController,  
NSFetchedResultsControllerDelegate
```

The `NSFetchedResultsControllerDelegate` protocol provides methods to notify its delegate whenever there are any changes in the controller's fetch results. Later we'll implement the methods. For now, declare an instance variable for the fetched results controller:

```
var fetchResultsController: NSFetchedResultsController<RestaurantMO>!
```

And add the following code in the `viewDidLoad` method:

```
// Fetch data from data store  
let fetchRequest: NSFetchedResultsController<RestaurantMO> = RestaurantMO.fetchRequest()  
let sortDescriptor = NSSortDescriptor(key: "name", ascending: true)  
fetchRequest.sortDescriptors = [sortDescriptor]
```

```

if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
    let context = appDelegate.persistentContainer.viewContext
    fetchResultController = NSFetchedResultsController(fetchRequest:
fetchRequest, managedObjectContext: context, sectionNameKeyPath: nil,
cacheName: nil)
    fetchResultController.delegate = self

    do {
        try fetchResultController.performFetch()
        if let fetchedObjects = fetchResultController.fetchedObjects {
            restaurants = fetchedObjects
        }
    } catch {
        print(error)
    }
}

```

We first get the `NSFetchRequest` object from `RestaurantMO` and specify the sort order using an `NSSortDescriptor` object. `NSSortDescriptor` lets you describe how the fetched objects are sorted. Here we specify that the `RestaurantMO` objects should be sorted in ascending order using the *name* key.

After creating the fetch request, we initialize `fetchResultController` and specify its delegate for monitoring data changes. Lastly, we call the `performFetch()` method to execute the fetch request. When complete, we get the `RestaurantMO` objects by accessing the `fetchedObjects` property.

If you compile and run the app now, it should display the restaurants that you previously added. However, if you try to add another new restaurant, the table does not refresh with the new record.

There are still something left.

When there is any content change, the following methods of the `NSFetchedResultsControllerDelegate` protocol will be called:

- `controllerWillChangeContent(_:)`
- `controller(_:didChange:at:for:newIndexPath:)`
- `controllerDidChangeContent(_:)`

Here is the implementation. Insert the following code in the `RestaurantTableViewController`

class:

```
func controllerWillChangeContent(_ controller:
NSFetchedResultsController<NSFetchRequestResult>) {
    tableView.beginUpdates()
}

func controller(_ controller: NSFetchedResultsController<NSFetchRequestResult>,
didChange anObject: Any, at indexPath: IndexPath?, for type:
NSFetchedResultsControllerChangeType, newIndexPath: IndexPath?) {

    switch type {
    case .insert:
        if let newIndexPath = newIndexPath {
            tableView.insertRows(at: [newIndexPath], with: .fade)
        }
    case .delete:
        if let indexPath = indexPath {
            tableView.deleteRows(at: [indexPath], with: .fade)
        }
    case .update:
        if let indexPath = indexPath {
            tableView.reloadRows(at: [indexPath], with: .fade)
        }
    default:
        tableView.reloadData()
    }

    if let fetchedObjects = controller.fetchedObjects {
        restaurants = fetchedObjects as! [RestaurantMO]
    }
}

func controllerDidChangeContent(_ controller:
NSFetchedResultsController<NSFetchRequestResult>) {
    tableView.endUpdates()
}
```

The first method is called when `NSFetchedResultsController` is about to start processing the content change. We tell the table view, "Hey, we're going to update the table. Get ready for it." This is done by calling `tableView.beginUpdates()`.

When there is any content change in the managed object context (e.g. a new restaurant is saved), the second method is automatically called. Here we determine the type of operation and proceed with the required operation accordingly. For instance, we insert new rows when

the type is set to `.Insert`. Because the objects in the fetched results controller are changed, we sync them with the `restaurants` array at the end of the method.

After `NSFetchedResultsController` completes the change, it calls the `controllerDidChangeContent` method. Here, we need to tell the table view that we've completed the update and it will animate the change accordingly.

That's it. Now run the app again and create a few restaurants. The app should respond to the change instantaneously.

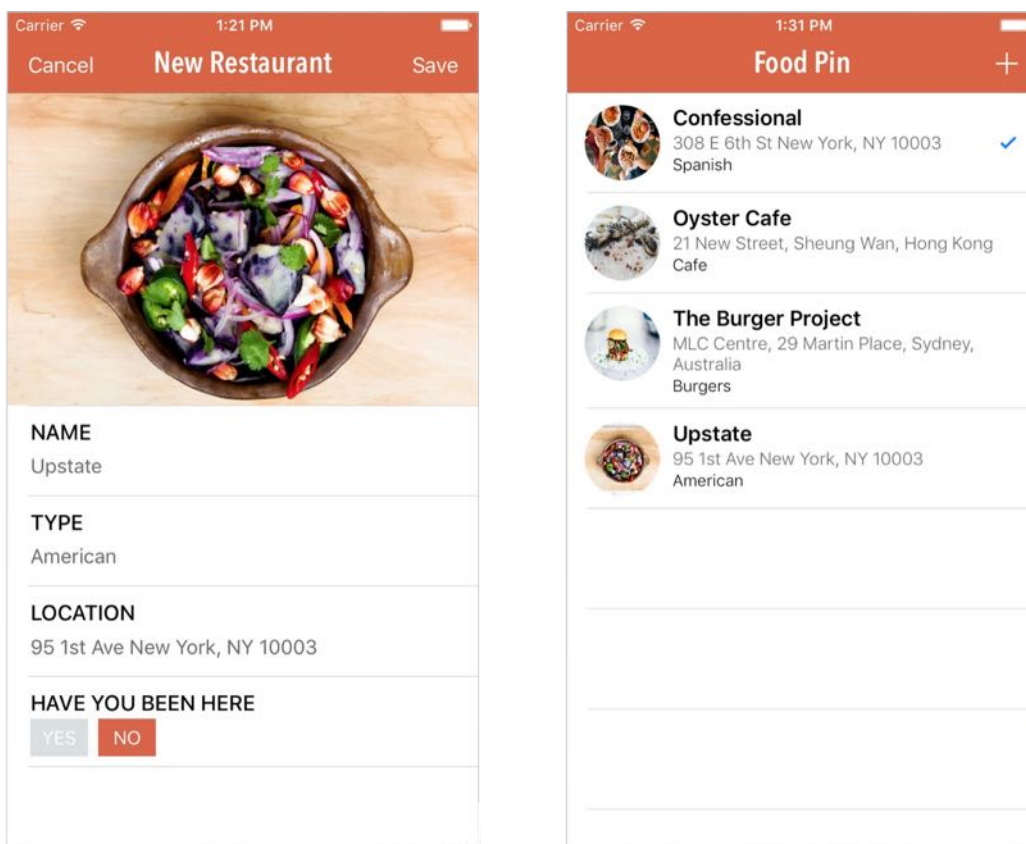


Figure 19-8. Saving a new restaurant

Deleting Data Using Core Data

To delete a record from the persistent data store, you just need to call a method named `delete` with the managed object to delete. Then, you call the `saveContext` method to apply the

changes. Here is a sample code snippet:

```
let context = appDelegate.persistentContainer.viewContext
context.delete(objectToDelete)
appDelegate.saveContext()
```

To remove the selected restaurant from database, you have to update the `deleteAction` variable in the `tableView(_:editActionsForRowAt:)` method like this:

```
let deleteAction = UITableViewRowAction(style:
UITableViewRowActionStyle.default, title: "Delete", handler: { (action,
indexPath) -> Void in

    if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
        let context = appDelegate.persistentContainer.viewContext
        let restaurantToDelete = self.fetchResultController.object(at:
indexPath)
        context.delete(restaurantToDelete)

        appDelegate.saveContext()
    }
})
```

In the above code, we get the selected `RestaurantMO` object from `fetchResultController` and then call the `delete` method to delete the item. Finally we call the `saveContext()` method to apply the changes.

Now compile and run the app again. At this point, if you delete a record it should remove completely from the database.

Updating a Managed Object

What if we need to update the rating of an existing restaurant? How can we update the record in database?

Similar to creating a new restaurant, you can update a restaurant record in the persistent store by updating the corresponding managed object and then call `saveContext()` to apply the changes.

For example, to save the rating of a restaurant, you can update the `close` method of the

RestaurantDetailViewController class like this:

```
@IBAction func ratingButtonTapped(segue: UIStoryboardSegue) {
    if let rating = segue.identifier {

        restaurant.isVisited = true

        switch rating {
        case "great": restaurant.rating = "Absolutely love it! Must try."
        case "good": restaurant.rating = "Pretty good."
        case "dislike": restaurant.rating = "I don't like it."
        default: break
        }
    }

    if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
        appDelegate.saveContext()
    }

    tableView.reloadData()
}
```

The code is exactly the same as before, except that we add a couple line of code to call up `appDelegate.saveContext()` to save the changes to database.

Your Exercise

I intentionally left out the phone number field for your implementation. Currently, there is no way for users to input a phone number. Your exercise is to add the *Phone Number* field in the New Restaurant screen, and save the information by using Core Data.

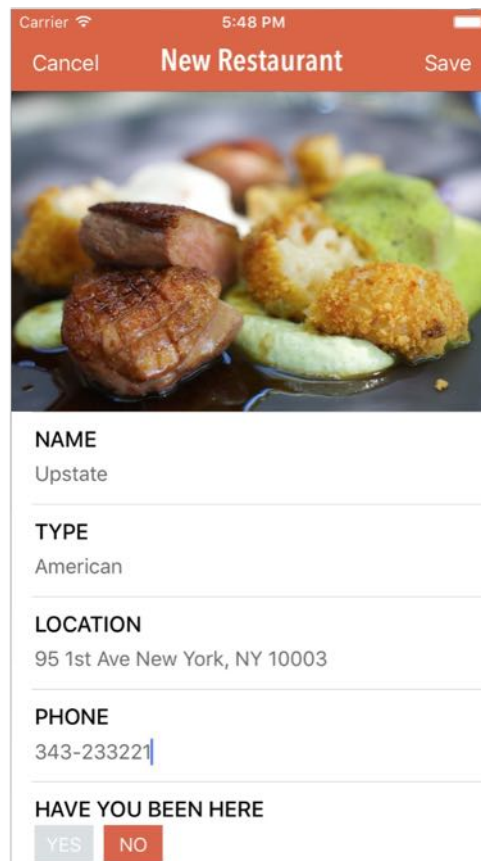


Figure 19-11. Adding a Phone Number field in the New Restaurant screen

Summary

Congratulate yourself on making an app using Core Data. At this point, you should know how to retrieve and manage data in a persistent data store. Core Data is a powerful framework for working with persistent data especially for those who do not have any database knowledge. The new class `NSPersistentContainer`, introduced in iOS 10, encapsulates the whole Core Data stack and makes working with Core Data a breeze, especially for beginners.

This chapter gives you a brief overview of Core Data. I hope you understand the basics of Core Data, and how to use it to store persistent data. Though we will not further discuss Core Data in this book, don't stop learning and exploring. You can check out Apple's official reference (<https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html>) to learn more.

For reference, you can download the complete Xcode from
<http://www.appcoda.com/resources/swift3/FoodPinCoreDataFinal.zip>.

Are you ready to further improve the app? I hope you're still with me. Let's move on and take a look how to add a search bar.

Chapter 20

Search Bar and UISearchController



I knew that if I failed I wouldn't regret that, but I knew the one thing I might regret is not trying.

– Jeff Bezos

For most of the table-based apps, it is common to have a search bar at the top of the screen. How can you implement a search bar for data searching? In this chapter, we will add a search bar to the FoodPin app. With the search bar, we will enhance the app to let users search through the available restaurants.

In iOS 8, a new class called `UISearchController` was introduced to replace the `UISearchDisplayController` API that has been around since iOS 3. The old API is now deprecated. If you have some experience with iOS 7 or older version of SDK, remember to use `UISearchController` instead.

The `UISearchController` API simplifies the way to create a search bar and manage search results. You're no longer limited to embed search in table view controller but can use it in any view controller like collection view controller. Even more, it offers developers flexibility to influence the search bar animation through a custom animator object.

With `UISearchController`, adding a search bar to your app is quite an easy task. Let's get started to implement a default search bar and see how we can filter the restaurant data.

Using UISearchController

In general, to add a search bar in a table-based app, it essentially comes down to the following lines of code:

```
searchController = UISearchController(searchResultsController: nil)
searchController.searchResultsUpdater = self
tableView.tableHeaderView = searchController.searchBar
```

The first line of code creates an instance of `UISearchController`. If you pass a `nil` value, that means the search results would be displayed in the same view that you're searching. Optionally, you can specify another view controller for displaying the search result.

You may wonder when you need to define another view controller. Take the FoodPin app as an example. If `nil` is used, the search results will be displayed in the table view. Figure 20-1 shows the format of search results. As you can see, the display style is exactly the same as that of the table view. If you want to display the search results in a different format, you will need to create another view controller and specify it during the initialization of `UISearchController`.

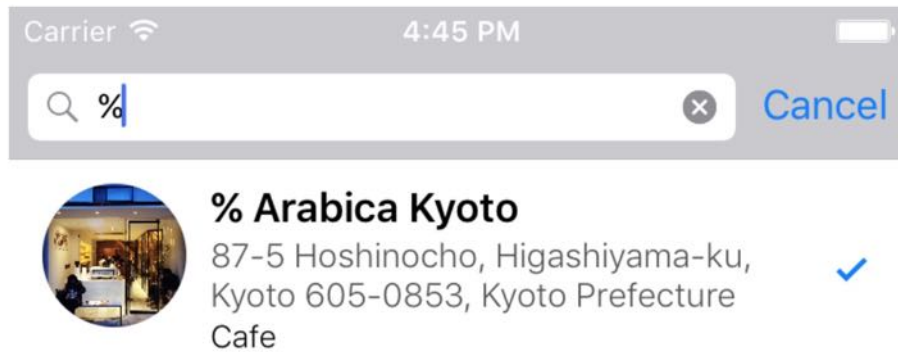


Figure 20-1. A standard search bar in iOS

The second line of code tells the search controller which object is responsible for updating the search result. It can be any object in your application or simply the current one.

The last line of code adds the search bar to the header view of the table view.

Adding a Search Bar

Now let's try to add a search bar in the FoodPin app. Open

`RestaurantTableViewController.swift`, declare the `searchController` variable:

```
var searchController:UISearchController!
```

Then add the following lines of code in the `viewDidLoad` method:

```
searchController = UISearchController(searchResultsController: nil)
tableView.tableHeaderView = searchController.searchBar
```

As I have explained the code before, I will not go over it again. But as you can see, you can add a default search bar with just two lines of code. If you compile and run the app now, you should find a search bar below the navigation bar. However, it doesn't work yet because we haven't implemented the search logic.

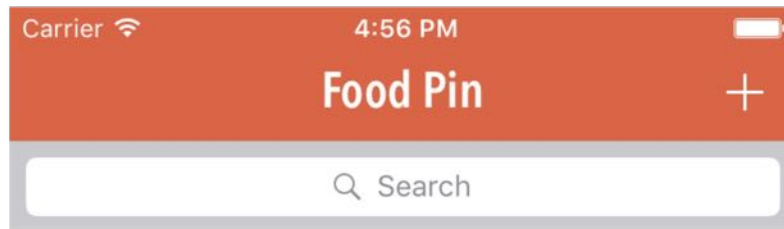


Figure 20-2. Search bar added in the header of table view

Filtering Content

The search controller doesn't provide any default functions to filter your data. It's your responsibility to provide your own implementation to filter the content. For the FoodPin app, it will allow users to do a search against the name of a restaurant. In order to implement such a search feature, first declare a new variable in the `RestaurantTableViewController` class to store the search results:

```
var searchResults:[RestaurantMO] = []
```

Add the following code to create a new method for content filtering:

```
func filterContent(for searchText: String) {
    searchResults = restaurants.filter({ (restaurant) -> Bool in
        if let name = restaurant.name {
            let isMatch = name.localizedCaseInsensitiveContains(searchText)
            return isMatch
        }
        return false
    })
}
```

In Swift, there is a built-in method called `filter` for filtering an existing array. You use `filter` to loop over a collection and return a new array containing those items that match the specified condition. For example, the new array can only contain restaurants with the name starts with "up".

The `filter` method takes in a block of code in which you provide the filtering rule. For those

elements to be included, you indicate with a return value of `true` . Otherwise, `false` is returned and that element will be excluded.

In the above code, we use the `localizedCaseInsensitiveContains` method to see if the restaurant name contains the search text, regardless of the case of the string (i.e. case-insensitive). If the search text is found, the method returns `true` , indicating the restaurant name should be included in the new array. Otherwise, a `false` is returned to exclude the item.

Updating Search Results

Now that we have implemented the search logic, how can we update and display the search results on screen? To update the search result, you first need to adopt the

`UISearchResultsUpdating` protocol:

```
class RestaurantTableViewController: UITableViewController,  
NSFetchedResultsControllerDelegate, UISearchResultsUpdating
```

The protocol defines a method called `updateSearchResults(for:)` . When a user selects the search bar or key in a search keyword, the method will be called. By implementing the method, we can instruct the search controller to display the search results. Insert the following code snippet in the `RestaurantTableViewController` class:

```
func updateSearchResults(for searchController: UISearchController) {  
    if let searchText = searchController.searchBar.text {  
        filterContent(for: searchText)  
        tableView.reloadData()  
    }  
}
```

The code is very straightforward - we get the search text as entered by the user and pass it to the `filterContent(for:)` method. Finally, reload the table data. As mentioned before, we use the same table view to display the search results. This is why we call `tableView.reloadData()` to load the search results..

`RestaurantTableViewController` is now responsible for displaying a full list of restaurants and the search results. The question is how can we figure out when it is used to display the search result and when it is used to display all restaurant data?

Obviously, the app should display the search results only when the search controller is active. The search controller provides a convenient property to verify its state. When the user taps the search field, the search interface will be brought up. Its `isActive` property is then set to `true`.

We can use this property to determine if the table view controller is used to display a full list of restaurants or the search results.

Now update the `tableView(_:numberOfRowsInSection:)` method like this:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    if searchController.isActive {
        return searchResults.count
    } else {
        return restaurants.count
    }
}
```

When the search controller is active, we return the number of the search results. In case that the search controller is inactive, we return the count of the full restaurant list.

The code in the `tableView(_:cellForRowAt:)` method should also be updated with an additional condition:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

    let cellIdentifier = "Cell"
    let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for: indexPath) as! RestaurantTableViewCell

    // Determine if we get the restaurant from search result or the original array
    let restaurant = (searchController.isActive) ? searchResults[indexPath.row] : restaurants[indexPath.row]

    // Configure the cell...
    cell.nameLabel.text = restaurant.name
    cell.thumbnailImageView.image = UIImage(data: restaurant.image as! Data)
    cell.locationLabel.text = restaurant.location
    cell.typeLabel.text = restaurant.type

    cell.accessoryType = restaurant.isVisited ? .checkmark : .none
}
```



```
    return cell
}
```

Again, we check if the search controller is active. If the user is doing a search, retrieve the restaurant from the search result rather than the restaurants array. Here is the line of code that does the magic:

Recall that the app displays two action buttons (Share & Delete) when a user swipes the cell; you probably don't want to show the buttons in the search results. You can simply implement the following method and indicate the cell is non-editable when the search controller is active.

```
override func tableView(_ tableView: UITableView, canEditRowAt indexPath:
IndexPath) -> Bool {
    if searchController.isActive {
        return false
    } else {
        return true
    }
}
```

Similarly, update the following line of code in the `prepare(for:)` method:

```
destinationController.restaurant = (searchController.isActive) ?
searchResults[indexPath.row] : restaurants[indexPath.row]
```

When passing the selected restaurant to the detail view controller, it's important to check if the search is active and pass the correct restaurant.

You're almost done. The very last thing is to add the following lines of code in the `viewDidLoad` method:

```
searchController.searchResultsUpdater = self
searchController.dimsBackgroundDuringPresentation = false
```

The first line assigns the current class as the search results updater. The `dimsBackgroundDuringPresentation` property controls whether the underlying content is dimmed during a search. Because we are presenting the search results in the same view, the property should be set to `false`.

Cool! You're ready to fire up your app and test out the search function. What's great is that you

can navigate to the restaurant details by tapping the search results. Everything in the original table view controller is reused.

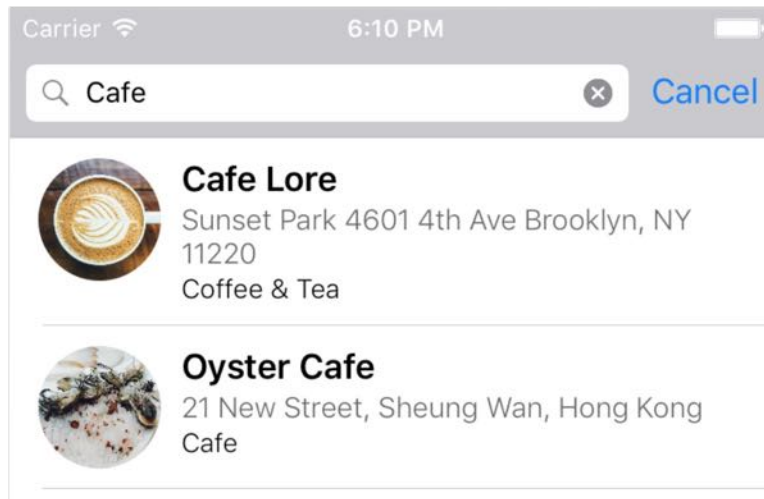


Figure 20-3. Search results

Customizing the Appearance of the Search Bar

`UISearchBar` provides several options for customizing the appearance of a search bar. You can access its properties by using the following line of code:

```
searchController.searchBar.tintColor
```

Here are some of the common properties for the customization.

- **placeholder** - you can use the `placeholder` property to set the default text when there is no other text in the text field.
- **prompt** - the `prompt` property allows you to display a single line of text at the top of the search bar.
- **barTintColor** - set the background color of the search bar.
- **tintColor** - set the tint color of the key elements in the search bar. For example, you can use the property to change the color of the Cancel button in the search bar.
- **searchBarStyle** - specify the search bar's style. By default, it is set to `.prominent`. When this style is set, the search bar has a translucent background, and the search field is

opaque. Alternatively, you can change it to `.minimal` to remove the background and make the search field translucent.

As an example, you can insert the following lines of code at the end of the `viewDidLoad` method of the `RestaurantTableViewController` class:

```
searchController.searchBar.placeholder = "Search restaurants..."
searchController.searchBar.tintColor = UIColor.white
searchController.searchBar.barTintColor = UIColor(red: 218.0/255.0, green:
100.0/255.0, blue: 70.0/255.0, alpha: 1.0)
```

Figure 20-4 shows the custom search bar after customization. You can further refer to the official documentation

(https://developer.apple.com/library/ios/documentation/UIKit/Reference/UISearchBar_Classes/) for the full set of customizable properties.

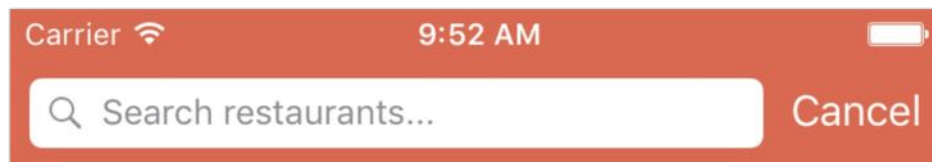
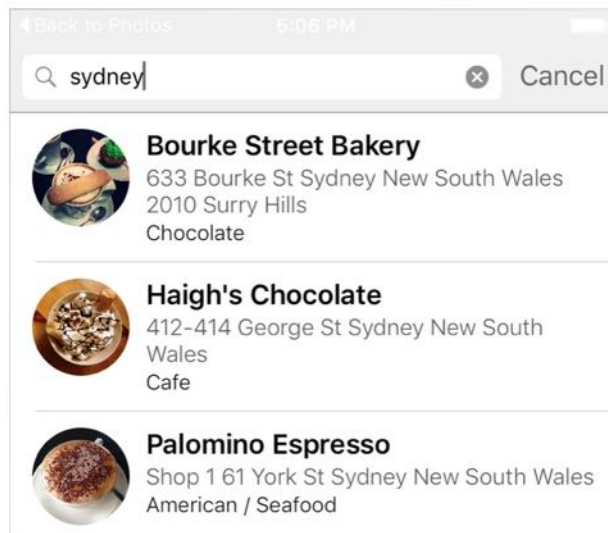


Figure 20-4. Sample search bar after customization

Your Exercise

Now the app only allows users to search restaurants by name. Your exercise is to enhance the search feature so that it supports location search too. For example, if your user keys in *Sydney* in the search field, the app searches through the restaurant list and shows you the restaurants that are either located in Sydney or have "Sydney" in the restaurant name.



Hint: You just need to modify the `filterContent(for:)` method to support location search.

Summary

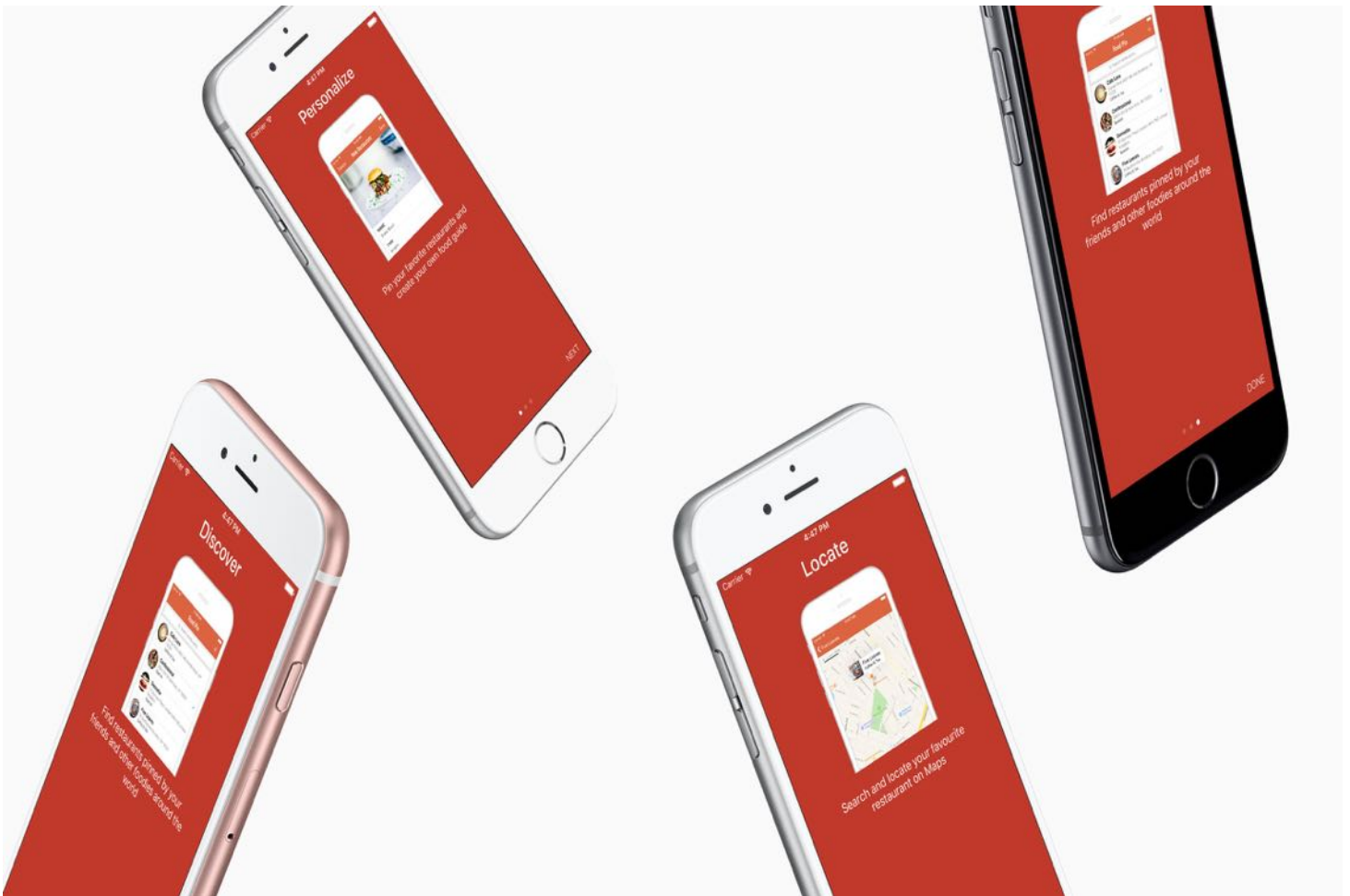
By now, you should know how to implement a search bar in an iOS app. We've made the FoodPin app even better by enhancing it with a search feature; this search feature is particularly important when you have a large amount of information to display. If you still don't fully understand the search bar feature, go back and work through this chapter a second time, step by step, before moving on.

In this chapter we only covered the basics of `UISearchController`. However, there are numerous other ways to use a search bar in your app. For instance, you may want to display the search bar only when the user taps a search button. To explore new ways of search bar integration, I encourage you to check out the UICatalog demo from Apple (<https://developer.apple.com/library/prerelease/ios/samplecode/UICatalog/Introduction/Intro.html>).

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinSearch.zip>. The solution of the exercise is included.

Chapter 21

Building Walkthrough Screens with UINavigationController



If you're interested in the living heart of what you do, focus on building things rather than talking about them.

- Ryan Freitas, About.me

For the very first time launching an app, you probably find a series of walkthrough (or tutorial) screens. Lately it's a common practice for mobile apps to step users through multi-screen tutorial where all the features are demonstrated. Some said your app design probably fails if your app needs walkthrough screens. Personally I don't hate walkthrough screens and mostly

find them pretty useful. Just make sure you keep it short and don't take it too far to include long and boring tutorials. Here I'm not going to argue whether you should or should not include walkthrough screens in your app. I just want to show you how. In this chapter, we'll discuss how to use `UIPageViewController` to create walkthrough screens.

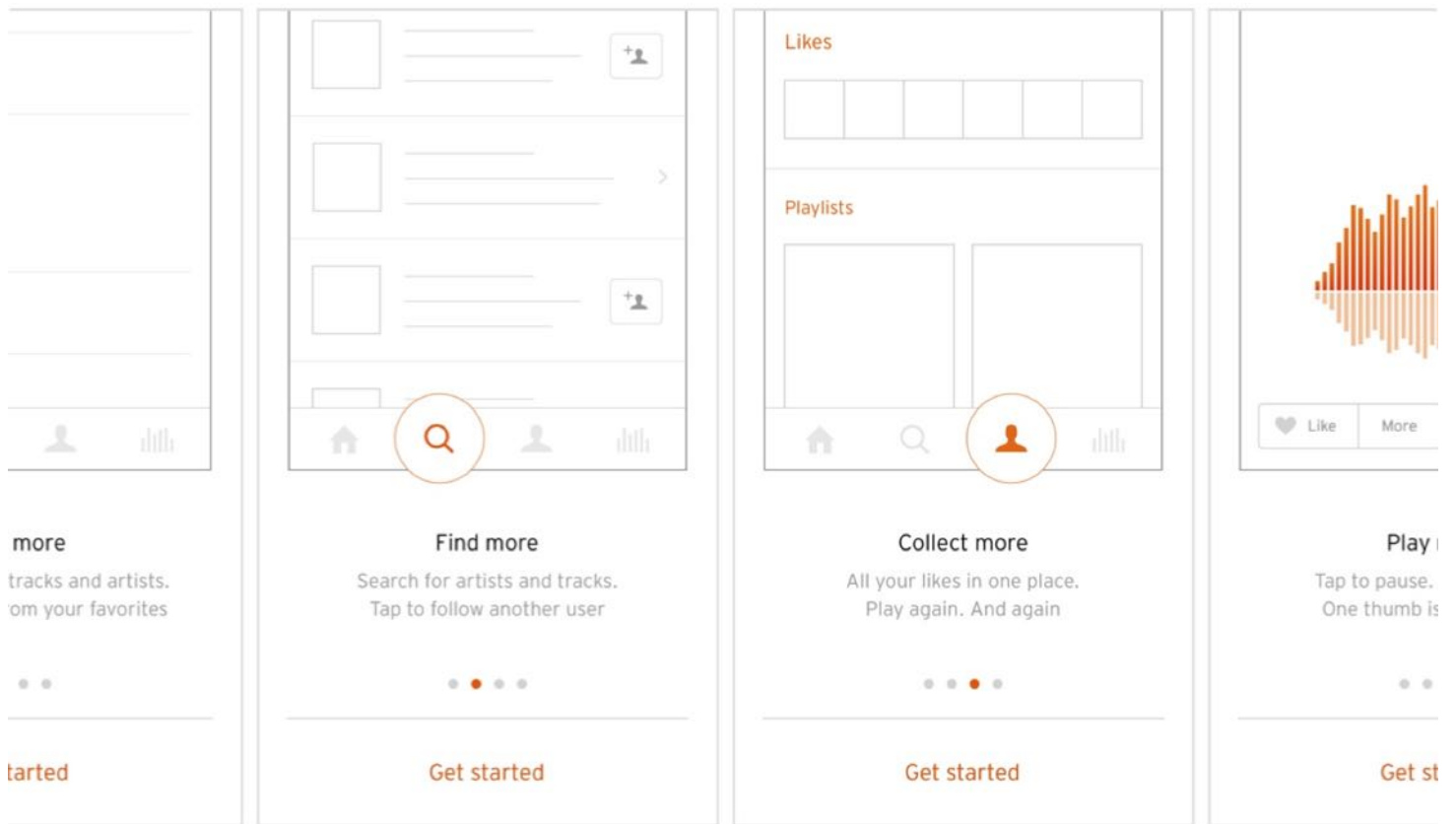


Figure 21-1. Sample Walkthrough Screens from SoundCloud

The `UIPageViewController` class was first introduced in iOS 5 SDK. It lets developers build pages of content, for which each page is managed by its own view controller. The class was further improved in iOS 6 to support the scrolling transition. With `UIPageViewController`, users can easily navigate between multiple pages through simple gestures. The page view controller is not limited to creating walkthrough screens. You can find examples of page view implementation in games like Angry Birds (the page that shows all available levels) or in book apps (the page that displays the table of contents).

The `UIPageViewController` class is a highly configurable class. You're allowed to define:

- the orientation of the page views – vertical or horizontal
- the transition style – page curl transition style or scrolling transition style
- the location of the spine – only applicable to page curl transition style
- the space between pages – only applicable to scrolling transition style to define the inter-page spacing

We'll add a simple walkthrough for the FoodPin app. By implementing this new feature, you'll learn how `UIPageViewController` works along the way. That said, we'll not demonstrate every option of `UIPageViewController` ; we'll just use the scrolling transition style to display a series of walkthrough screens. With the basic understanding of the `UIPageViewController`, however, I believe you should be able to explore other features in the page view controller.

Let's get started.

A Quick Look at the Walkthrough Screens

Let's have a quick look at the walkthrough screens. The app will display a total of three screens during the walkthrough. The user will be able to navigate between pages by swiping through the screen or tapping the arrow icon. In the last screen of the walkthrough, it displays a *Get Started* button. When the user taps the button, the walkthrough screen will be dismissed and never be shown again. Figure 21-2 shows the screenshots of the walkthrough.

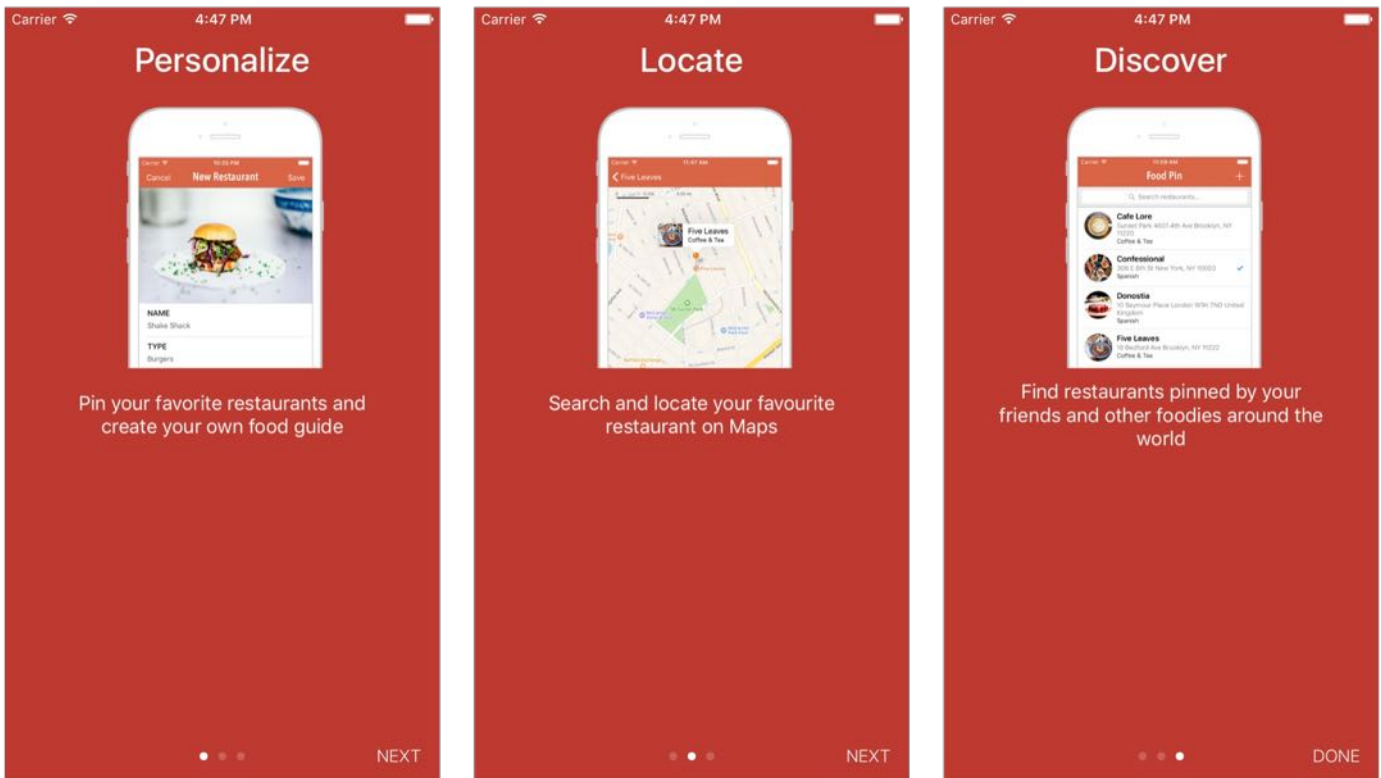


Figure 21-2. Walkthrough screens for the FoodPin app

Adding UINavigationController in Storyboard

Open the FoodPin project and jump to `Main.storyboard`.

Note: You can download the FoodPin project from <http://www.appcoda.com/resources/swift3/FoodPinSearch.zip>

In the Object library, drag a page view controller to the storyboard. As you can see, there are multiple options for configuring the behavior of the controller under the Attribute inspector. You can customize the navigation style (horizontal/vertical), transition style, page spacing, and spine location.

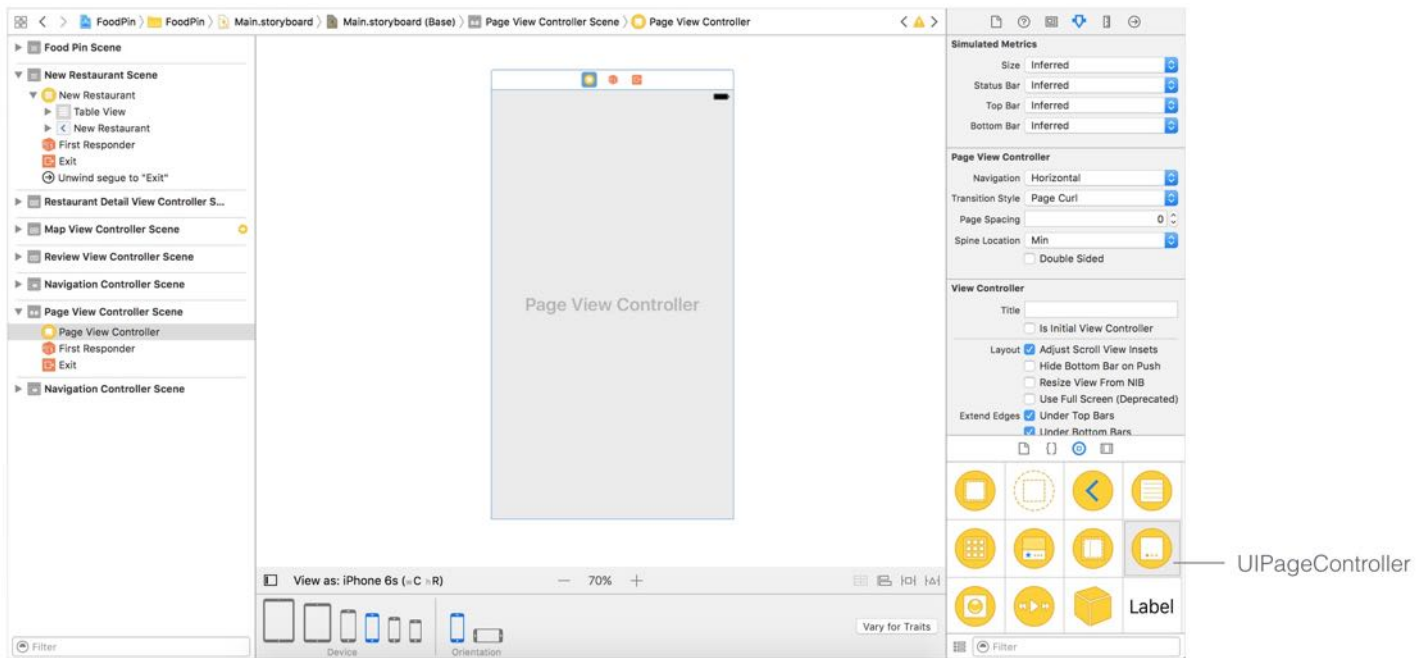


Figure 21-3. Page View Controller in Interface Builder

By default, the transition style of the page view controller is set as `PageCurl`. This style is perfect for book apps. For walkthrough screens, we prefer to use scrolling style. In the Attributes inspector, change the transition style to `Scroll`.

Next, assign a storyboard ID to the page view controller. In the Identity inspector, set the storyboard ID of the page view controller to `walkthroughController`. Later we will use the ID for creating the controller programmatically. This storyboard ID is optional, but if you need to use it in your code. You have to give the controller a storyboard ID.

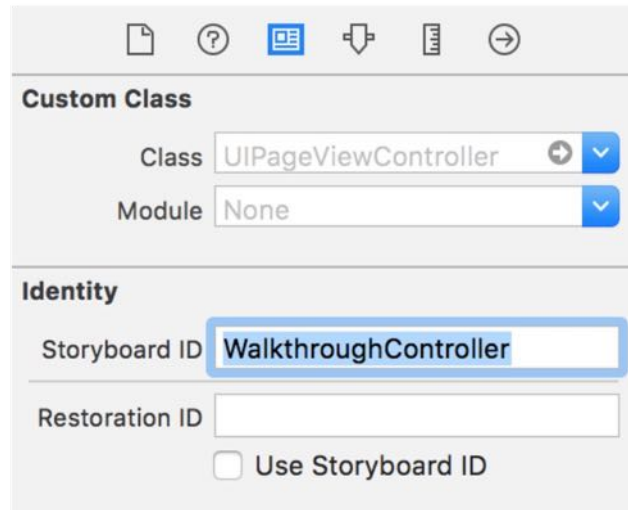


Figure 21-4. Set a Storyboard ID for the Page View Controller

Understanding Page View Controllers

For now, you can't layout the walkthrough screens directly on the page view controller. Before I show you how, let's see how the page view controller works.

Like `UINavigationController`, the `UIPageViewController` class is classified as a container controller. The container controller is designed to hold and manage multiple view controllers shown in an app, as well as, control how one view controller switches to another. Here `UIPageViewController` is the container controller that lets users navigate from page to page. Each page is actually managed by its own view controller known as page content view controller. Figure 21-5 depicts the relationship between the page view controller and the page content view controller.

UIPageViewController manages multiple content view controllers.

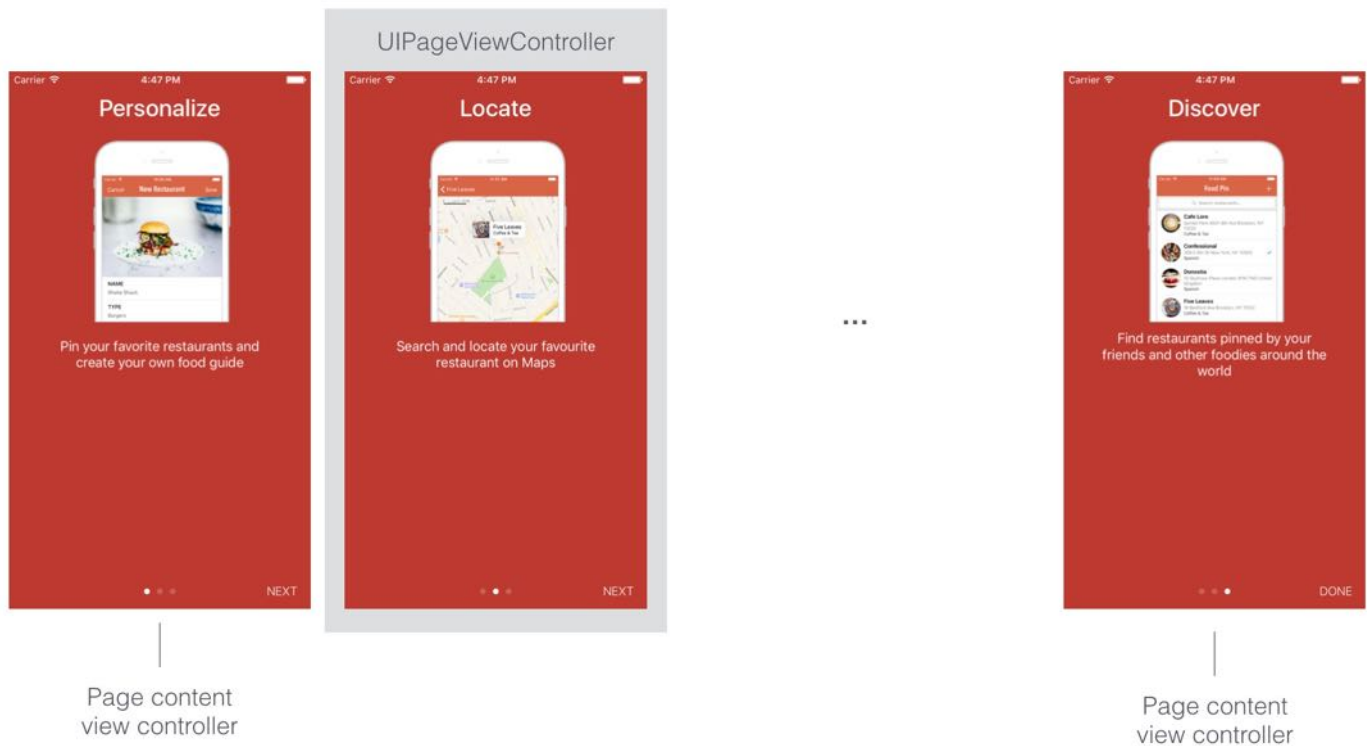


Figure 21-5. Relationship between the page view controller and the page content view controller

Therefore, to create the walkthrough screens, we need to add a view controller to the storyboard. You may be curious why we just add a single view controller for three pages of content. Shouldn't we add three view controllers?

Take a look figure 21-5 again. All walkthrough screens look very similar. Instead of building a view controller for each screen, it's better to reuse the same view controller and create different screens programmatically.

Designing the Walkthrough Screens

First, download the image pack from (<http://www.appcoda.com/resources/swift3/walkthrough.zip>) and add the images into `Assets.xcasset`. Next drag a view controller from the Object library to the storyboard. Follow these procedures to layout the view:

- Set the background color of the view controller to red (#C0392B).
- Add a label object and name it `Personalize`. Change to your preferred font and font size. Set the alignment to `center`.
- Add an image view object to the view controller, put it right below the `Personalize` label. Set the width to `300` points and the height to `232` points.
- Add another label object and name it `Pin your favorite restaurants and create your own food guide`. Again change the font if you like and set the alignment to `center`. For the number of lines, set it to zero. Resize the label to `282` by `64` points so it is big enough to accommodate three lines of text.

You'll end up with a screen similar to that shown in figure 21-6.

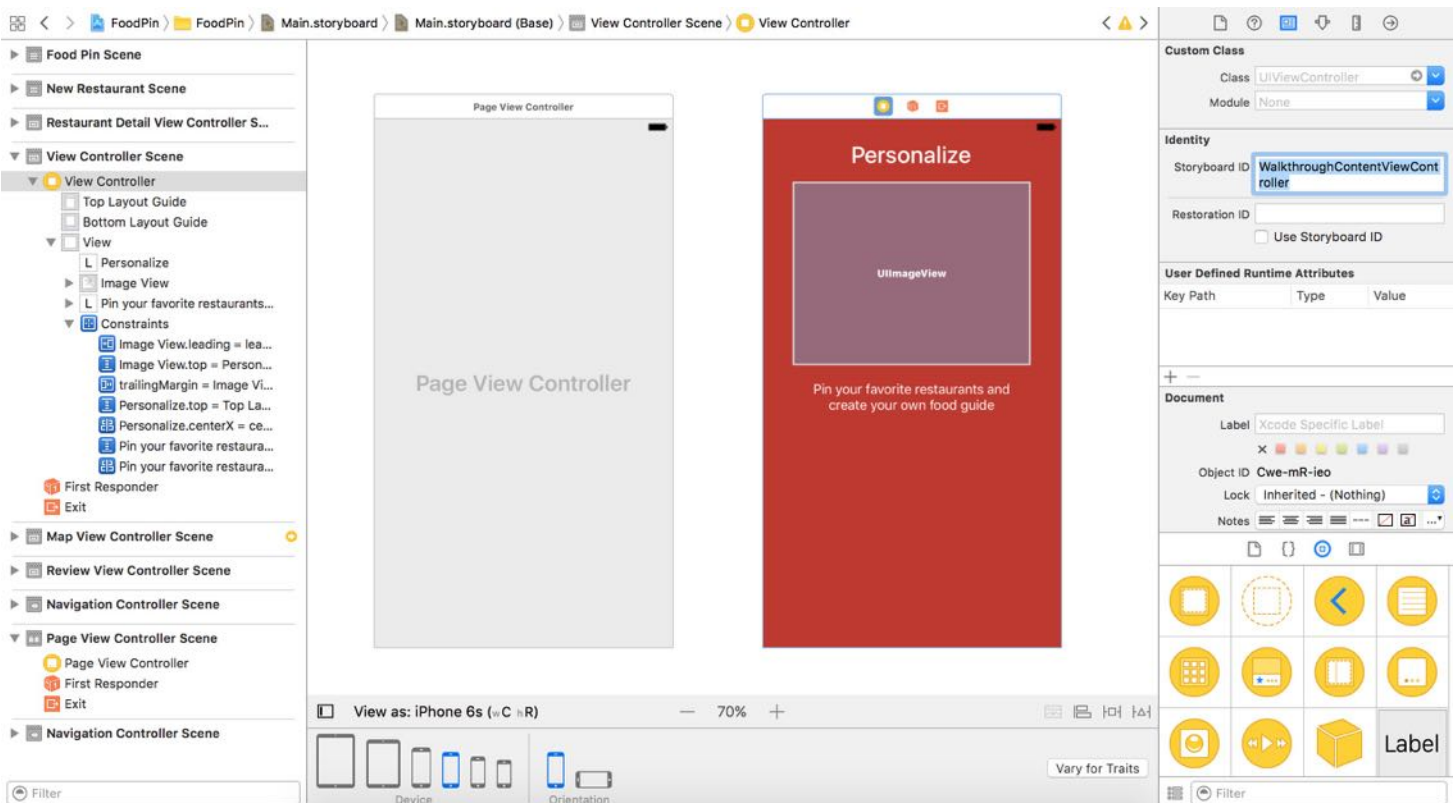


Figure 21-6. Designing the View Controller for walkthrough screens

As usual, you have to define the layout constraints for the components:

- For the first label object, add spacing constraints for the top side and define a constraint to center it horizontally.

- For the image view, add spacing constraints for top, left, right and bottom. Also add an `Aspect ratio` constraint.
- For the second label object, define a constraint to center it horizontally. Also add constraints to limit its width and height.

Lastly in the Identity inspector, set the storyboard ID of view controller to `walkthroughContentViewController` . Later, we'll use this ID later in our code.

Note: By now you should be able to use auto layout, so I left out a lot of details. If you have any difficulties defining the constraints, you can refer to this Xcode project template <http://www.appcoda.com/resources/swift3/FoodPinWalkthrough1.zip> for reference.

Creating WalkthroughContentViewController Class

It's very straightforward to implement the page content view controller. Right-click the FoodPin folder in the project navigator and select `New File...` . Name the class `walkthroughContentViewController` and set it as a subclass of `UIViewController` .

In the `walkthroughContentViewController.swift` file, declare the following outlet and instance variables in the class:

```
@IBOutlet var headingLabel: UILabel!  
@IBOutlet var contentLabel: UILabel!  
@IBOutlet var contentImageView: UIImageView!  
  
var index = 0  
var heading = ""  
var imageFile = ""  
var content = ""
```

We will use this class to support multiple walkthrough screens. The index variable is used to store the current page index. For instance, the first walkthrough screen will have the index value of `0` . The view controller is designed to display an image, heading and content. So we create three variables for data passing.

Next, change the `viewDidLoad()` method:

```
override func viewDidLoad() {  
    super.viewDidLoad()
```

```
headingLabel.text = heading
contentLabel.text = content
contentImageView.image = UIImage(named: imageFile)
}
```

Here we initialize the labels and image view. As you should know, the next thing we have to do is to establish a connection between the UI components and the outlet variables.

Go to `Main.storyboard` and select the content view controller you've created. In the Identity inspector, set the custom class to `walkthroughContentViewController`. Next, right-click the page content view controller in the document outline and establish the following connections:

- Connect the `headingLabel` outlet with the `Personalize` label.
- Connect the `contentLabel` outlet with the description label.
- Connect the `contentImageView` outlet with the Image View object.

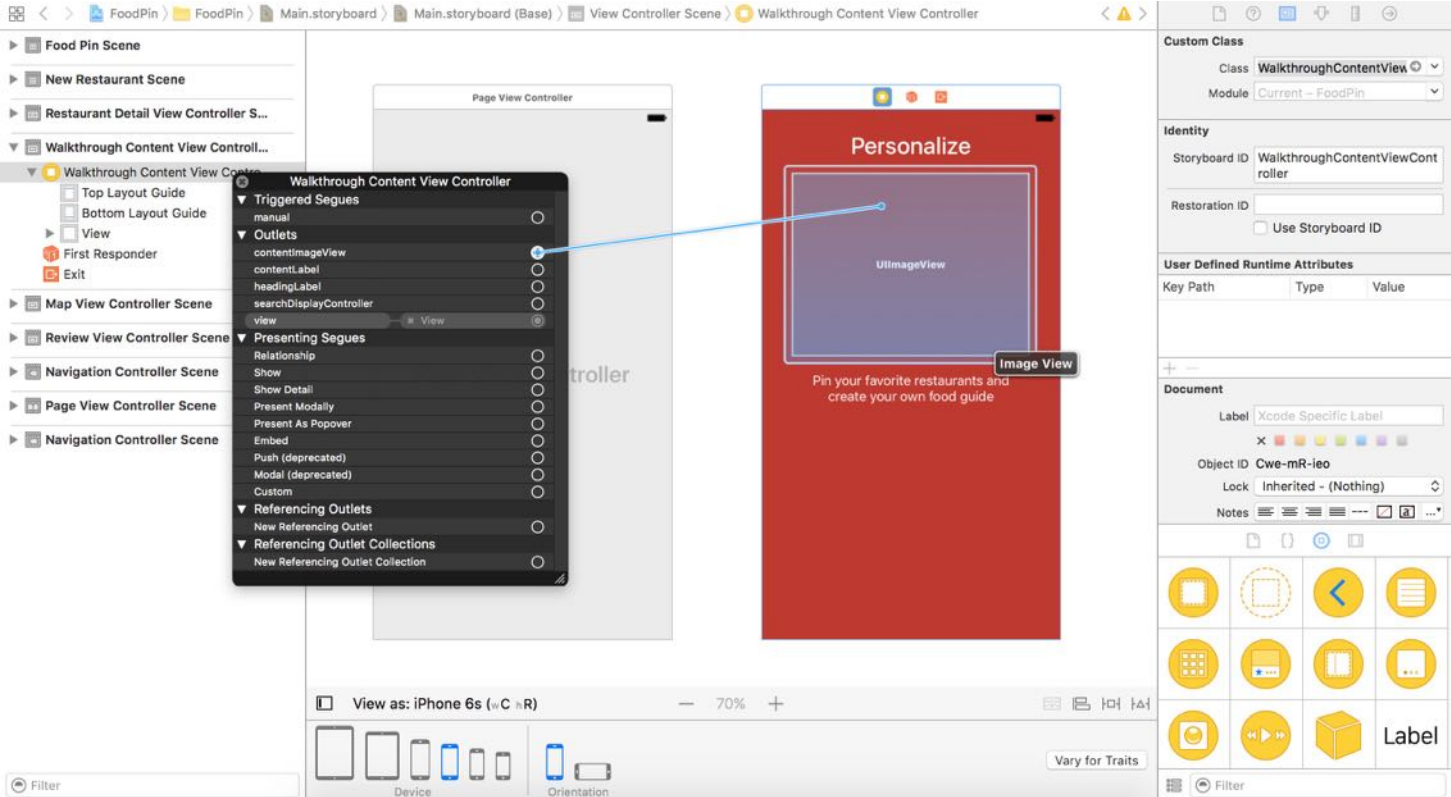


Figure 21-7. Establishing connections with the outlet variables

Implementing UIPageViewController

Now that we've prepared the content view controller, the next step is to create each of the content view controller and add it into the `UIPageViewController` so that the user can navigate between them. You have two ways to tell `UIPageViewController` what to display; you either provide the content view controllers one at a time or on-demand using a data source. If you just want to display a certain view controller in `UIPageViewController`, you can call `setViewControllers(_:direction:animated:completion:)` method with the view controller to be displayed.

As our app supports gesture-based navigation for the walkthrough, it's required to use the on-demand approach. For this approach, you assign a data source object serving as the provider of the content view controllers. Every time when a user navigates from one page to another, `UIPageViewController` asks its data source, "*Hey, what content view controller should be displayed? Please pass it to me.*" The data source object then returns the corresponding content view controller.

The data source object should conform to the `UIPageViewControllerDataSource` protocol and implement the following required methods:

- `pageViewController(_:viewControllerBefore:)`
- `pageViewController(_:viewControllerAfter:)`

Both methods may be called when a user navigates between pages. When called, the method is passed with a specific view controller. Your job is to determine and return a content view controller for display before/after the given controller. Let's say, if you have a content view controller with the index value of `1`. The `UIPageViewController` object will ask:

- Hey, what is the *next* view controller? In this case, you should return a content view controller with index `#2`.
- Hey, what is the *before* view controller? In this case, you should return a content view controller with index `#0`.

Now that you have some ideas about how `UIPageViewController` works, we'll proceed to create a new class for the page view controller. Right-click the `FoodPin` folder and select `New File...`. Name the class `walkthroughPageViewController` and set it as a subclass of

```
UIPageViewController .
```

Once created, open `walkthroughPageViewController.swift` and adopt the `UIPageViewControllerDataSource` protocol:

```
class WalkthroughPageViewController: UIPageViewController,
UIPageViewControllerDataSource
```

Declare and initialize the heading, content, and image variables, which are used when creating the content view controllers:

```
var pageHeadings = ["Personalize", "Locate", "Discover"]
var pageImages = ["foodpin-intro-1", "foodpin-intro-2", "foodpin-intro-3"]
var pageContent = ["Pin your favorite restaurants and create your own food
guide",
                  "Search and locate your favourite restaurant on Maps",
                  "Find restaurants pinned by your friends and other foodies
around the world"]
```

Next, implement the two required methods of the `UIPageViewControllerDataSource` protocol:

```
func pageViewController(_ pageViewController: UIPageViewController,
viewControllerBefore viewController: UIViewController) -> UIViewController? {

    var index = (viewController as! WalkthroughContentViewController).index
    index -= 1

    return contentViewController(at: index)
}

func pageViewController(_ pageViewController: UIPageViewController,
viewControllerAfter viewController: UIViewController) -> UIViewController? {

    var index = (viewController as! WalkthroughContentViewController).index
    index += 1

    return contentViewController(at: index)
}
```

The above methods are very straightforward. First, we get the current page index of the given view controller. Depending the method, we simply increase/decrease the index number and return the view controller to display.

As you may notice, we haven't created the `contentViewController(at:)` method. This is a helper method that is designed to create the page content view controller on demand. It takes in the `index` parameter and creates the corresponding page content controller.

Now add the helper method in the `WalkthroughPageViewController` class:

```
func contentViewController(at index: Int) -> WalkthroughContentViewController?
{
    if index < 0 || index >= pageHeadings.count {
        return nil
    }

    // Create a new view controller and pass suitable data.
    if let pageContentViewController =
        storyboard?.instantiateViewController(withIdentifier:
            "WalkthroughContentViewController") as? WalkthroughContentViewController {

        pageContentViewController.imageFile = pageImages[index]
        pageContentViewController.heading = pageHeadings[index]
        pageContentViewController.content = pageContent[index]
        pageContentViewController.index = index

        return pageContentViewController
    }

    return nil
}
```

Recall that we have set a storyboard ID for the view controller when designing the user interface. The ID is used as a reference for creating the view controller instance. To instantiate a view controller in storyboard, you call the `instantiateViewController(withIdentifier:)` method with a specific storyboard ID. The method returns an optional corresponding to the storyboard ID. This is why we use `as?` to downcast the object to `WalkthroughContentViewController`. Following the instantiation, we assign the content view controller with specific image, heading, content, and index.

Lastly, update the `viewDidLoad` method to the following:

```
override func viewDidLoad() {
    super.viewDidLoad()

    // Set the data source to itself
    dataSource = self
}
```

```

// Create the first walkthrough screen
if let startingViewController = contentViewController(at: 0) {
    setViewControllers([startingViewController], direction: .forward,
animated: true, completion: nil)
}
}

```

In the above code, we set the data source of `UIPageViewController` to itself and create the first content view controller when the page view controller is first set up.

With the class configured, go to storyboard and select the page view controller. In the Identity inspector, change the custom class from `UIPageViewController` to `WalkthroughPageViewController`.

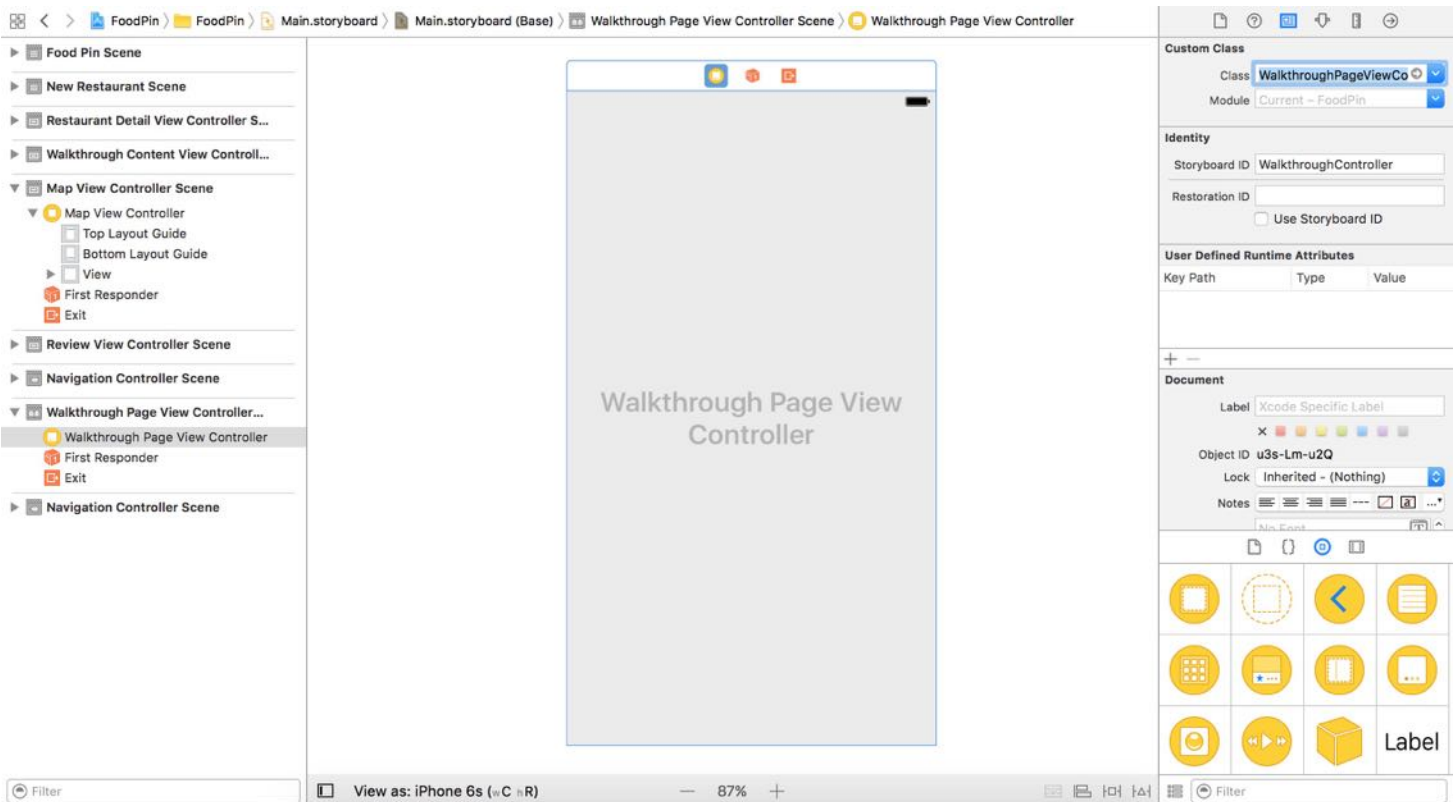


Figure 21-8. Setting the custom class

Displaying the Walkthrough Screens

You're almost ready to test out the walkthrough screens. Because we want to bring up the page

view controller when a user first launches the app. We have to instantiate the controller in the `RestaurantTableViewController` class. Simply add the following method:

```
override func viewDidLoad(_ animated: Bool) {
    if let pageViewController =
        storyboard?.instantiateViewController(withIdentifier: "WalkthroughController")
        as? WalkthroughPageViewController {
        present(pageViewController, animated: true, completion: nil)
    }
}
```

The code is self-readable. We just instantiate the `walkthroughPageViewController` object and present it in a modal way.

Now you're ready to hit the Run button and test the app. As soon as you launch the app, it brings up the walkthrough screens and you can navigate through the screens using gestures.

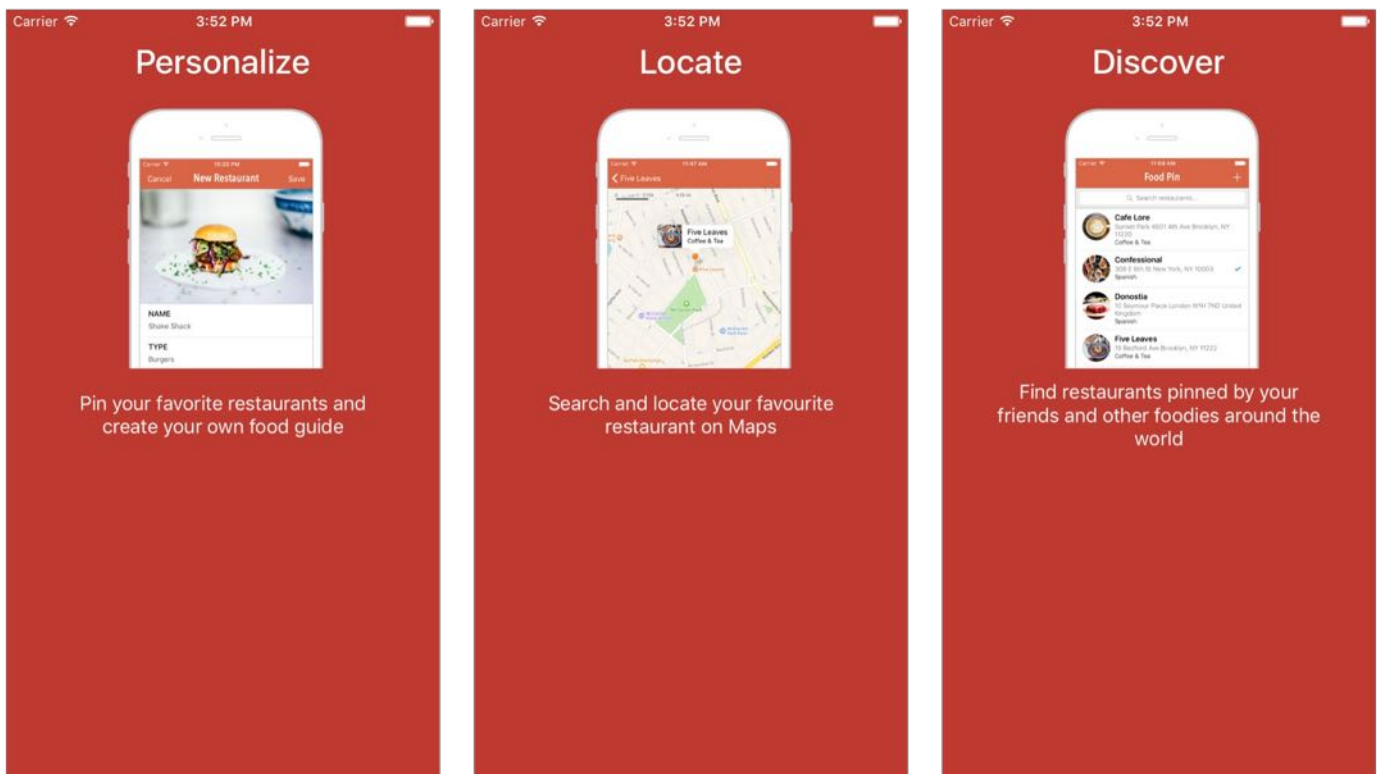


Figure 21-9. Walkthrough screens without page indicator

Adding Default Page Indicator

The walkthrough screens work great, right? But the page indicator is now missing in the screens. How can you add a page indicator?

The `UIPageViewControllerDataSource` protocol provides the following method to support a standard page indicator:

- `presentationCount(for:)` - implements this method to return the total number of dots (or pages) to be shown in the indicator
- `presentationIndex(for:)` - implements this method to return the index of the selected item

All you need is to implement these two methods in the `WalkthroughPageViewController` class:

```
func presentationCount(for pageViewController: UIPageViewController) -> Int {
    return pageHeadings.count
}

func presentationIndex(for pageViewController: UIPageViewController) -> Int {
    if let pageContentViewController =
        storyboard?.instantiateViewController(withIdentifier:
            "WalkthroughContentViewController") as? WalkthroughContentViewController {
        return pageContentViewController.index
    }
    return 0
}
```

If you compile and run the app again, you should find a page indicator shown at the bottom of the walkthrough screens.

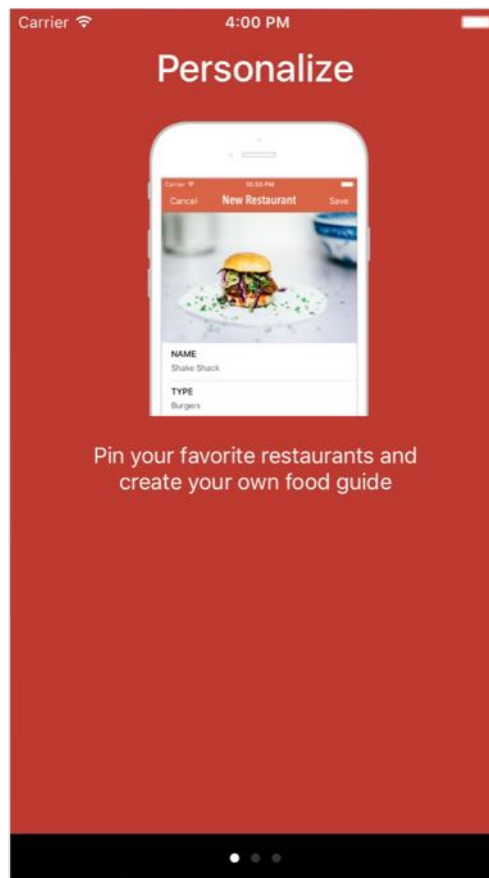


Figure 21-10. A page indicator shown at the bottom of the screens

Adding a Custom Page Indicator

It's very easy to add the default page indicator. However, there is a catch - you can't change its position. If you want to place it at other positions and add other customizations, you will need to use a custom `UIPageControl`.

First, comment out or remove the code added in the previous section. We no longer need them because we're going to implement a custom page control.

Go to Interface Builder and drag a page control object from the Object library to the walkthrough content view controller. Place it right below the content label. In the Attributes inspector, set the number of pages to `3`. You can also customize the tint color and background if you like. I prefer to keep the default color. Your storyboard should look like this:

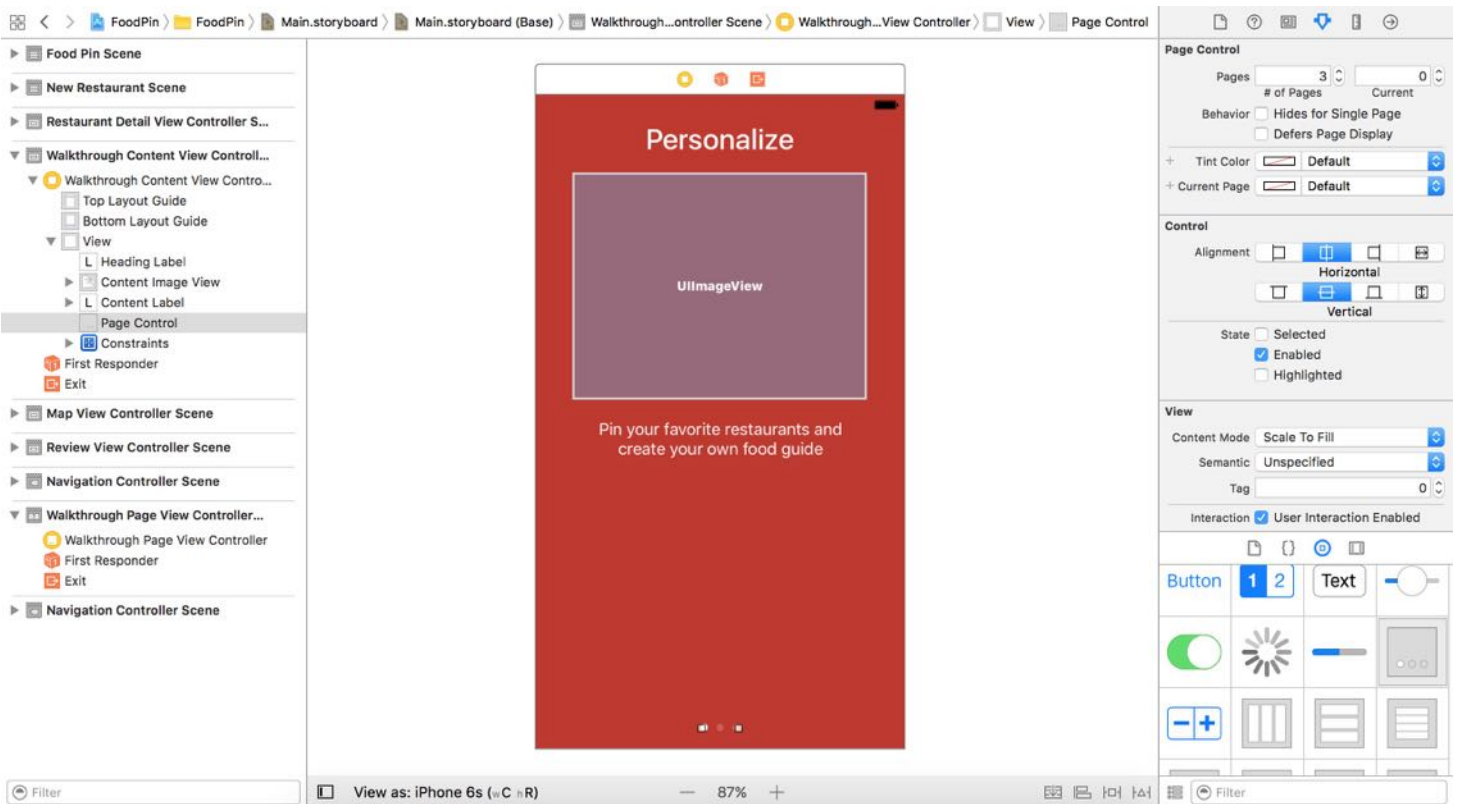


Figure 21-11. Adding page control

Control-drag from the page control to the view. Once released the buttons, add *Vertical Spacing to Bottom* layout guide and *Center Horizontally* constraints.



Figure 21-12. Adding page control

Now the page control is ready to use. You can run the project to have a quick test. The app should display a custom page control. But as you navigate through the walkthrough screens, the white dot, that indicates the current page, does not change accordingly.

To make it work, we have to write a couple lines of code. First, create an outlet variable for the

page control in the `WalkthroughContentViewController` class:

```
@IBOutlet var pageControl: UIPageControl!
```

In the `viewDidLoad` method, add the following line of code to update the `currentPage` property of the page control:

```
pageControl.currentPage = index
```

Go to Interface Builder and right-click the walkthrough content view controller in the document outline. Then establish a connection with the `pageControl` outlet. That's it! You just create your own page control. When you run the app, it should show the page indicator at the bottom of the view.

Adding Next and Get Started Buttons

The walkthrough screens work pretty great, but there is no way to dismiss it. What we're going to do next is to add a *DONE* button at the last walkthrough screen. On top of that, we want to provide an alternative way for user to navigate through the pages. We will add a *NEXT* button to the first two pages of the walkthrough. When a user taps the button, it forwards to the next screen with an animated transition.

Go to `Main.storyboard` and drag a button to the walkthrough content view controller. Name the button *NEXT*. I put the button at bottom right corner of the view and change the text color to white. It's up to you to customize the button's color and font.

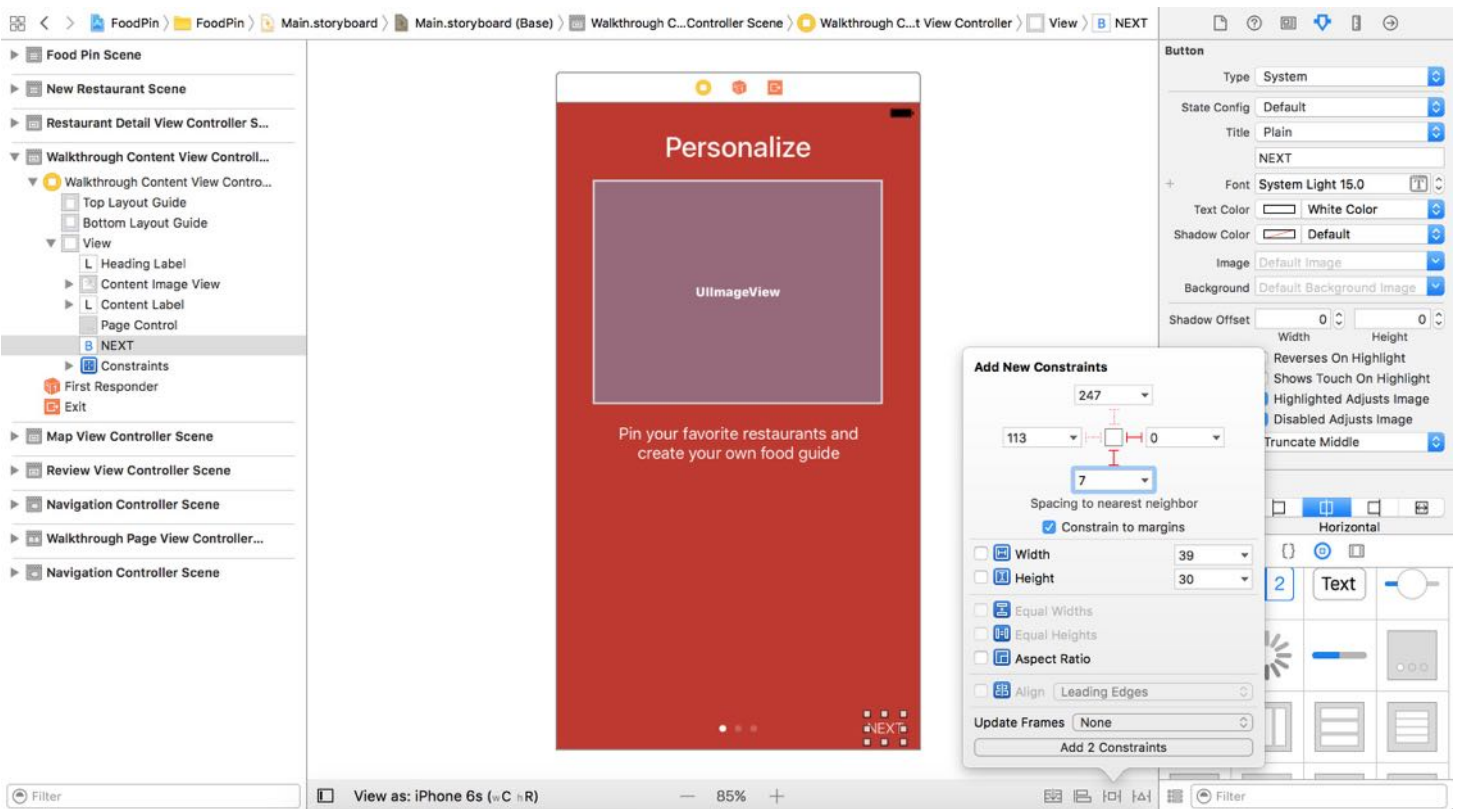


Figure 21-13. Sample design of the Next button

As mentioned before, the *NEXT* button is shown in the first two pages of the walkthrough, while the *DONE* button is displayed at the last screen. To do that, we will change the button's title based on the index of the walkthrough page. First, let's add an outlet variable in the `WalkthroughContentViewController` class:

```
@IBOutlet var forwardButton: UIButton!
```

In the `viewDidLoad` method, add the following lines of code:

```
switch index {
case 0...1: forwardButton.setTitle("NEXT", for: .normal)
case 2: forwardButton.setTitle("DONE", for: .normal)
default: break
}
```

The code should be self-explanatory. We change the button's title based on the page index. For the first two page, we set the title to *NEXT*. For the last walkthrough page, we set the title to *DONE*.

if case in Swift 2.0

In Swift 2, it introduces the `if case` keyword. You can rewrite the code snippet like this:

```
if case 0...1 = index {  
  
    forwardButton.setTitle("NEXT", for: .normal)  
} else if case 2 = index {  
    forwardButton.setTitle("DONE", for: .normal)  
}
```

Next, add an action method to handle the button tap:

```
@IBAction func nextButtonTapped(sender: UIButton) {  
  
    switch index {  
    case 0...1:  
        let pageViewController = parent as! WalkthroughPageViewController  
        pageViewController.forward(index: index)  
  
    case 2:  
        dismiss(animated: true, completion: nil)  
  
    default: break  
  
    }  
}
```

Depending on the button type, the method performs different operation. If it's the *NEXT* button, it will forward to the next page content view controller. In the above code, we call the `forward` method of the `WalkthroughPageViewController` object with the current page index. We haven't implemented the `forward` method yet, so Xcode shows an error. For the *DONE* button, we simply dismiss the page view controller and show the main screen of the app.

The `forward` method is a helper method in `WalkthroughPageViewController.swift`. We haven't implemented it yet. So add the following code in the `WalkthroughPageViewController` class:

```
func forward(index: Int) {  
    if let nextViewController = contentViewController(at: index + 1) {  
        setViewControllers([nextViewController], direction: .forward, animated:  
true, completion: nil)  
    }  
}
```

```
}
```

The method takes in a page index and creates the next content view controller. If the controller can be created, we call the built-in `setViewControllers` method and navigate to the next view controller.

With the action method and outlets set up, the last thing is to connect them with the *NEXT* button in storyboard. Right-click the walkthrough content view controller in the document outline and establish the connections for the `forwardButton` outlet. Then control-drag the *NEXT* button to the walkthrough content view controller. Release both buttons and select `nextButtonTappedWithSender:` event.

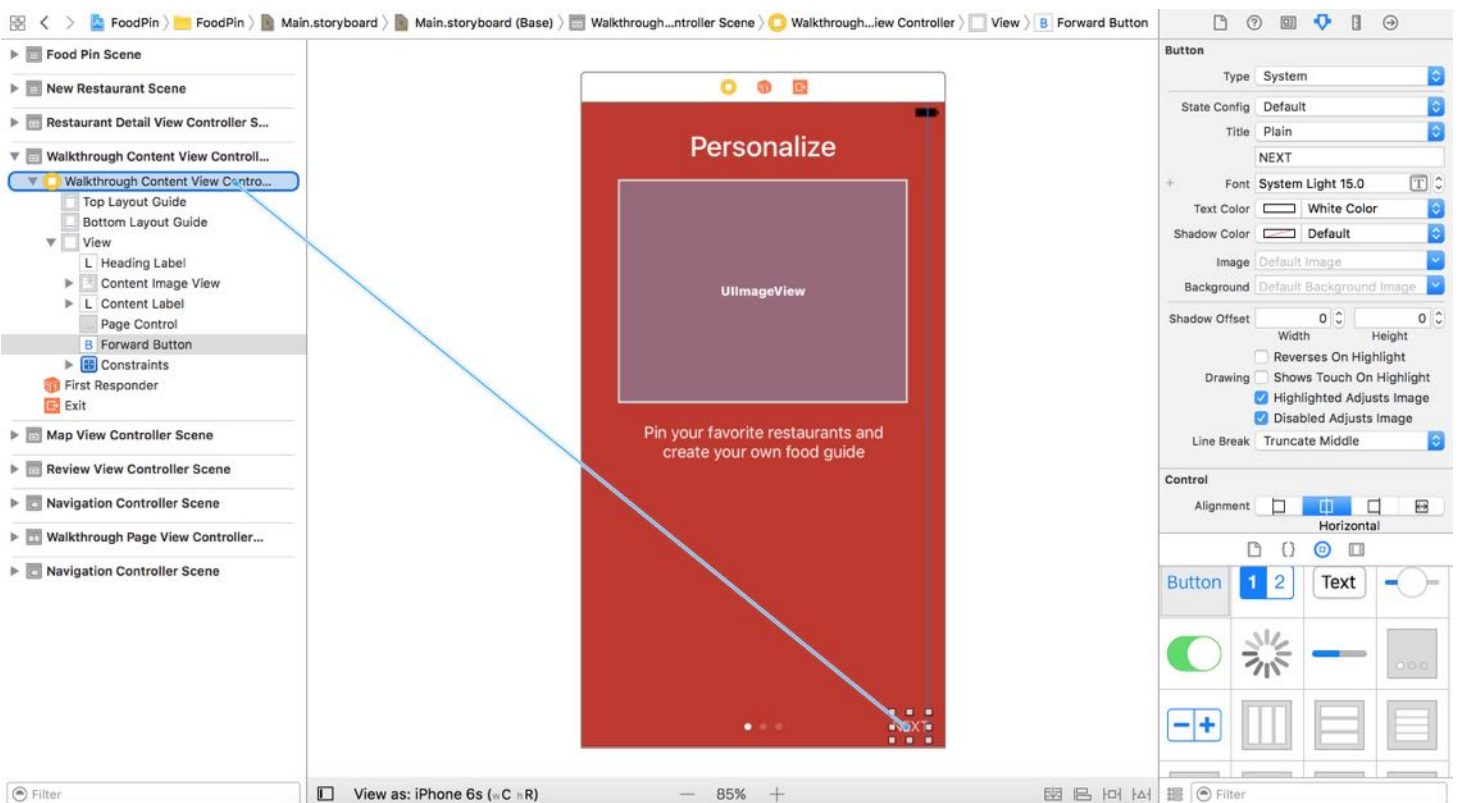


Figure 21-14. Connecting the action method with the button

Now you're ready to go. Your app should display a walkthrough screen as shown in figure 21-15 that supports both gesture-based and button-based navigation. When you tap the *NEXT* button, it navigates to the next screen with an animated transition.

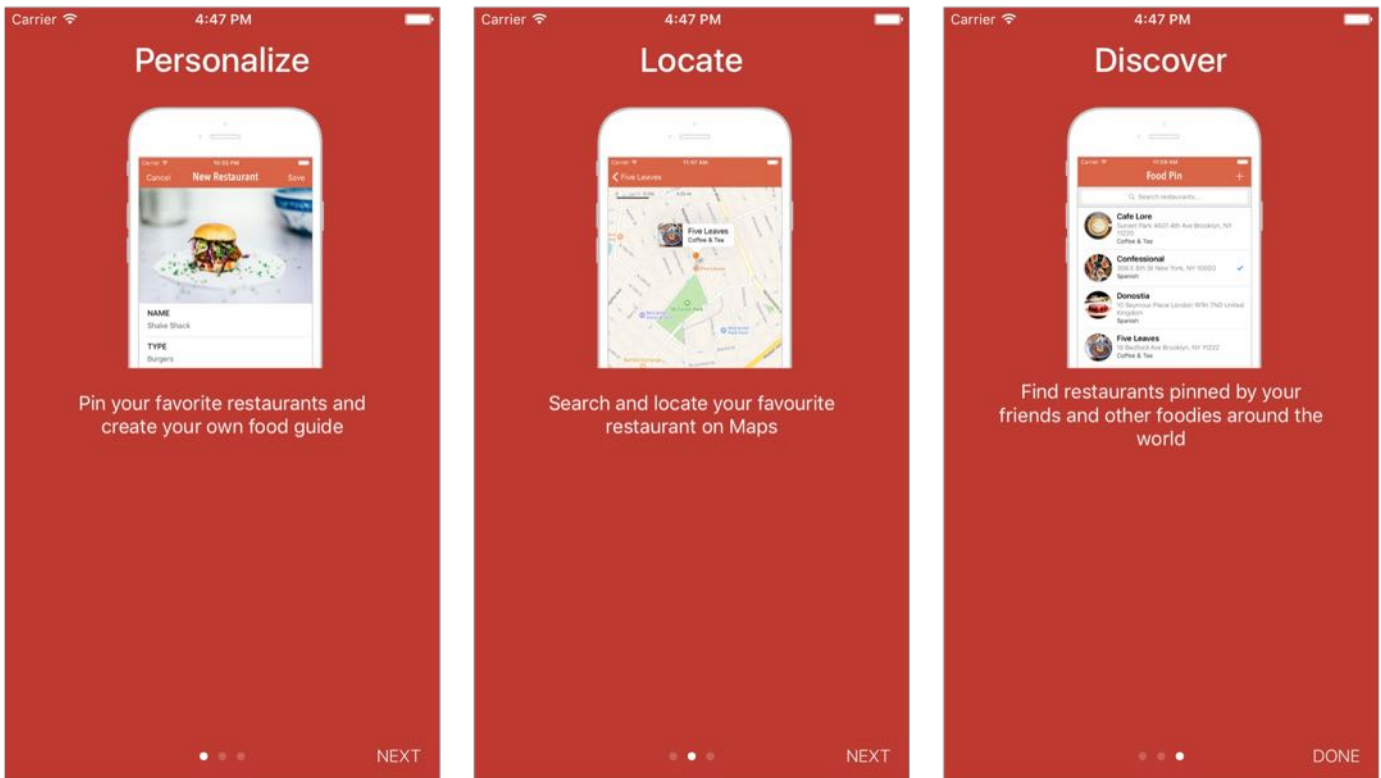


Figure 21-15. Walkthrough screens with the custom page indicator and next button

Now the walkthrough is up and running. However, it's displayed again and again, even you dismiss it. It is because the `viewDidAppear` method, where we present the page view controller, is called every time when the main screen appears. We haven't implemented any logic to control the reappearance of the walkthrough screens.

Generally, walkthrough screens are displayed only when a user launches the app for the very first time. In order to do so, we need to find a way to save a status that indicates whether the user has viewed the walkthrough.

You've learned Core Data, you probably want to save the status in database. While this is an option, there is a simpler way to save the application and user settings.

Introducing UserDefaults

The iOS SDK comes with an `UserDefaults` class for managing application and user-related

settings. For example, you can save a status to indicate whether the user has gone through the walkthrough. Or if you've enhanced the app to display the average price of a basic meal and want to allow users to configure the default currency, you can also use `UserDefaults` to save the user's preference.

The `UserDefaults` class provides a programmatic interface for interacting with the defaults system. The defaults system is created automatically and available for all code across your app. Data stored in the defaults system is persistent. In another words, you can still access the data even if the user quits the app or the app crashes.

To use `UserDefaults`, all you need is to get the shared defaults object:

```
UserDefaults.standard
```

Depending on the type of objects to retrieve, you can use one of the following methods to retrieve the setting:

- `array(forKey:)`
- `bool(forKey:)`
- `data(forKey:)`
- `dictionary(forKey:)`
- `float(forKey:)`
- `integer(forKey:)`
- `object(forKey:)`
- `stringArray(forKey:)`
- `string(forKey:)`
- `double(forKey:)`
- `url(forKey:)`

To save a setting in the defaults system, you set the value with a specific key. Here is an example:

```
UserDefaults.standard.set(true, forKey: "hasViewedWalkthrough")
```

Working with UserDefaults

If you understand how `UserDefaults` works, you may know how to use it to save the walkthrough status. After a user taps the *DONE* button, we save a status in the user defaults to indicate that the user has gone through the tutorial. Open

`WalkthroughContentViewController.swift` and update the `nextButtonTapped` method to the following:

```
@IBAction func nextButtonTapped(sender: UIButton) {

    switch index {
    case 0...1: // Next Button
        let pageViewController = parent as! WalkthroughPageViewController
        pageViewController.forward(index: index)

    case 2: // Done Button
        UserDefaults.standard.set(true, forKey: "hasViewedWalkthrough")
        dismiss(animated: true, completion: nil)

    default: break
    }
}
```

When the *DONE* button is tapped, we add a `hasViewedWalkthrough` key to the user defaults and set it to `true`.

Now open `RestaurantTableViewController.swift` and add a simple logic in the `viewDidAppear` method to determine if it should present the page view controller. Update the method like this:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    if UserDefaults.standard.bool(forKey: "hasViewedWalkthrough") {
        return
    }

    if let pageViewController =
        storyboard?.instantiateViewController(withIdentifier: "WalkthroughController")
        as? WalkthroughPageViewController {

        present(pageViewController, animated: true, completion: nil)
    }
}
```

The change should be very straightforward. We just retrieve the `hasViewedWalkthrough` key

from the user defaults and check its value. The walkthrough controller will only be presented when the value is set to `false` .

That's it. Run the project to have a quick test. Now once you tap the *DONE* button, the app will no longer display the walkthrough screens.

Summary

In this chapter, we covered the basics of `UIPageViewController` and `UserDefaults` . We demonstrated how to use both classes to implement walkthrough screens when an app is first launched.

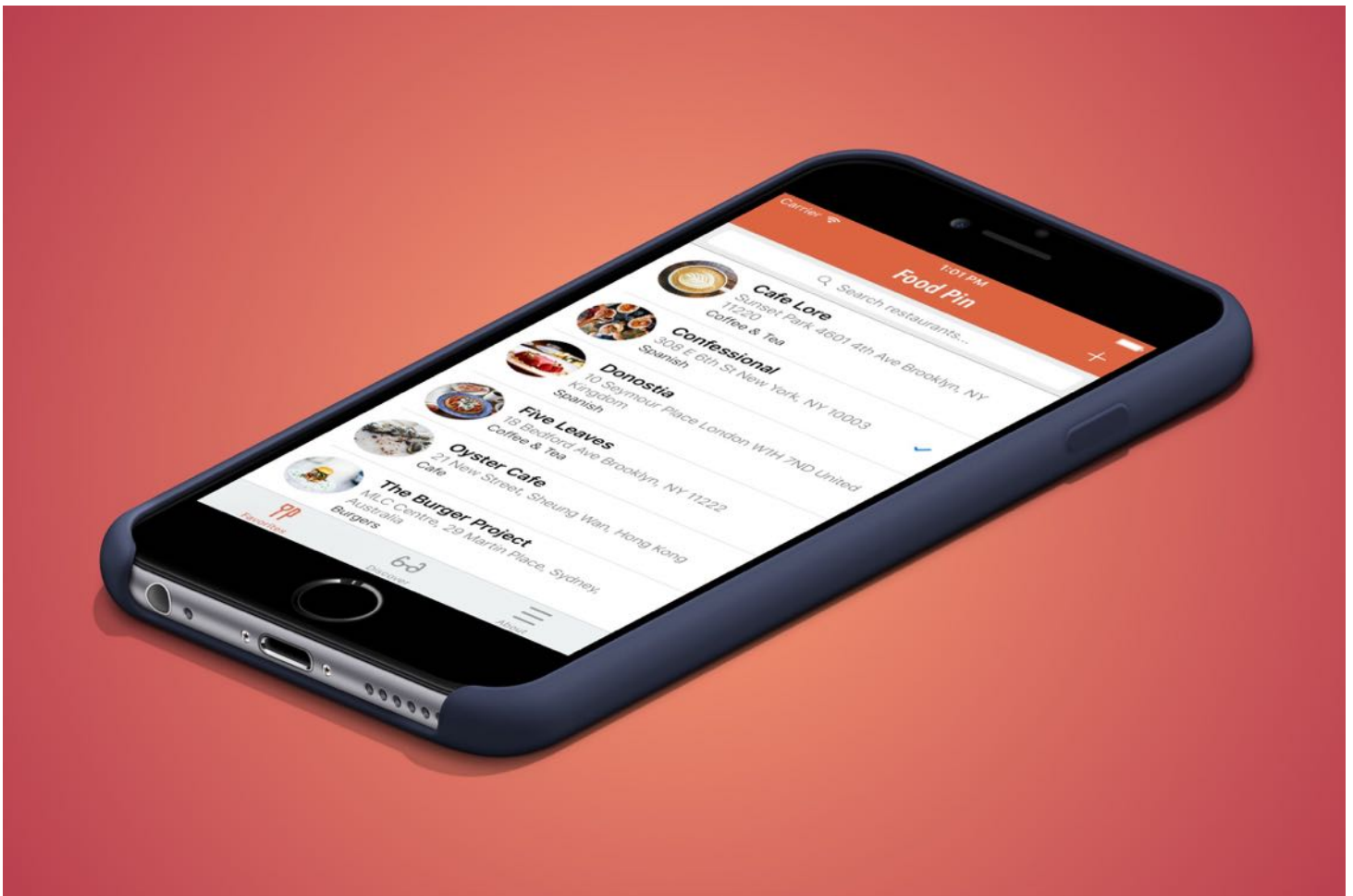
`UIPageViewController` is a very handy class for implementing walkthrough or tutorial screens. That said, the usage of `UIPageViewController` is unlimited. You can use it to display whatever information you like such as pages of web view.

So far we just use the *scroll* transition style. Don't you know that you can easily use `UIPageViewController` to build a simple book app? Simply change the transition style from *scroll* to *page curl* and see what you'll get.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinWalkthroughFinal.zip>.

Chapter 22

Exploring Tab Bar Controller and Storyboard References



If you're trying to achieve, there will be roadblocks. I've had them; everybody has had them. But obstacles don't have to stop you. If you run into a wall, don't turn around and give up. Figure out how to climb it, go through it, or work around it.

- Michael Jordan

The tab bar is a row of persistently visible buttons at the bottom of the screen that open different features of the app. Once a less-prominent UI design in the mainstream, the tab bar design becomes popular again in 2015.

One drawback of tab bars is that you sacrifice a bit of screen estate. At the time where there were only 3.5-inch and 4-inch iPhones, this was really an issue. As Apple rolled out the iPhone 6 and 6 Plus with larger screen sizes in late 2014, app developers start to replace the existing menus of their apps with tab bars. The Facebook app has switched from a sidebar menu design to a tab bar. Other popular apps like Whatsapp, Twitter, Quora, Instagram, and Apple Music also use tab bars for navigation.

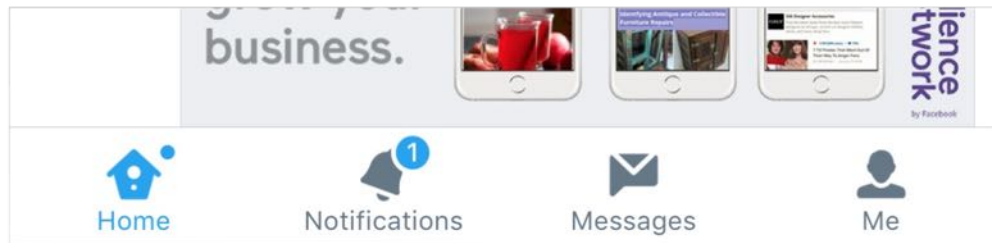


Figure 22-1. A tab bar in the Twitter app

Tab bars allow users to access the core features of an app quickly, with just a single tap. Though it takes up a bit of screen estate, it is worth it.

While navigation controllers let users navigate hierarchical content by managing a stack of view controllers, tab bars manage multiple view controllers that don't necessarily have a relation to one another. Normally a tab bar controller contains at least two tabs and you're allowed to add up to five tabs depending on your needs.

In this chapter, I will walk you through the implementation of tab bars and see how we can customize its appearance. We will also take a look at a rather new feature of Xcode called *storyboard references*.

Building a Tab Bar Controller

First, let's open the FoodPin project. We're going to create a tab bar with three items:

- Favorites - this is the restaurant list screen.
- Discover - this is a new screen to discover favorite restaurants recommended by your friends or other foodies in the world. We will implement this tab in the iCloud chapter.

- About - this is the "About" the about screen of the app. Once again we'll keep this page blank until the next chapter.



Creating a tab bar is easy and doesn't require to write a line of code. All you need to do is embed a set of view controllers in a tab bar controller using Interface Builder.

Open the `Main.storyboard` file and select the Navigation Controller, which is the initial controller of the app. This is the controller we want to embed in a tab bar controller. Now go up to the Xcode menu, select Editor > Embed in > Tab Bar Controller.

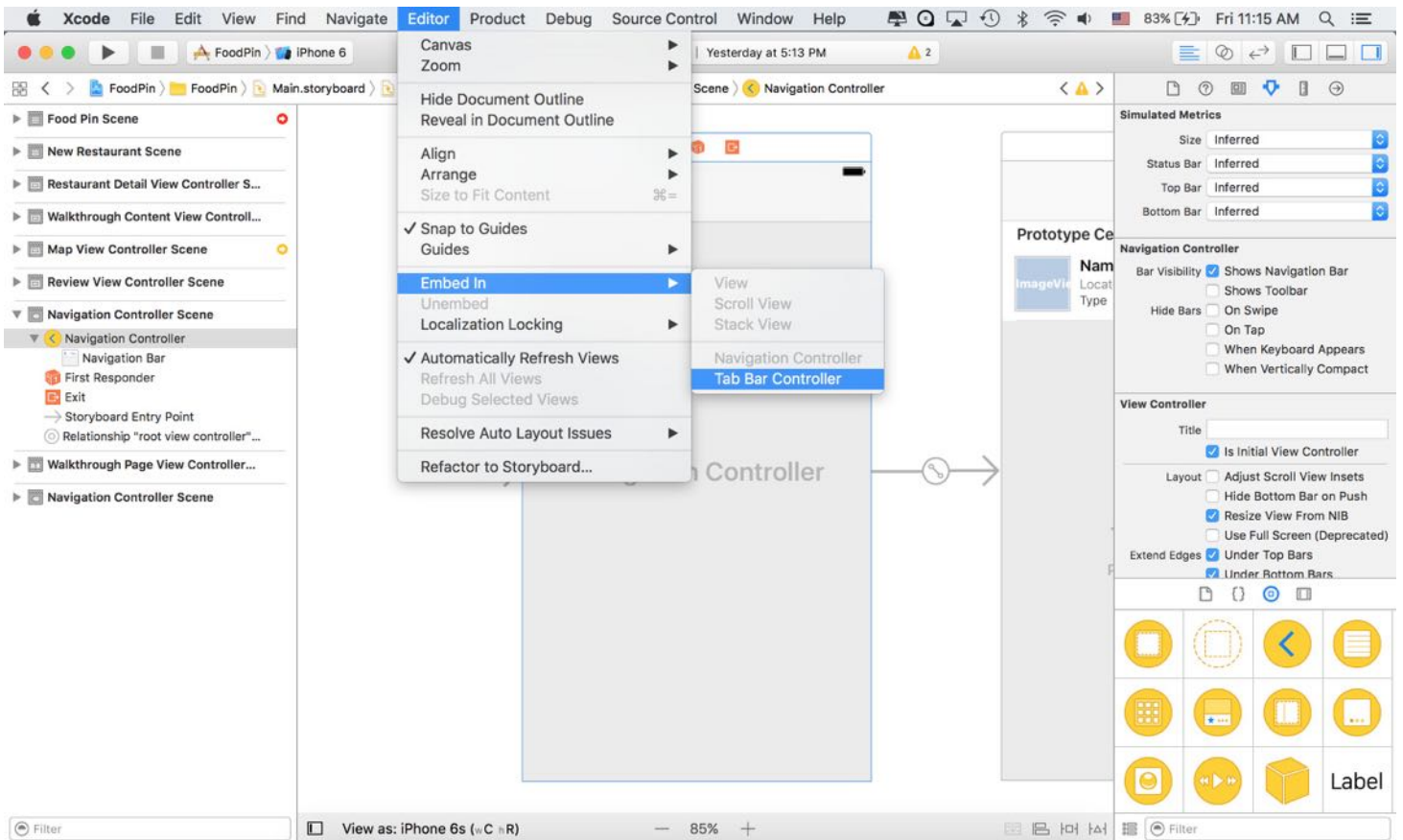


Figure 22-3. Embed the navigation controller in a tab bar controller

Interface Builder automatically puts the navigation controller inside a tab bar controller. Easy, right? You can then click the tab item in the navigation controller and change its properties in the Attribute inspector. You can use your own images to create a custom tab item. For simplicity, we just use the system items. Change the *System Item* option to *Favorites*. The tab bar item should be updated like the one shown in figure 22-4.

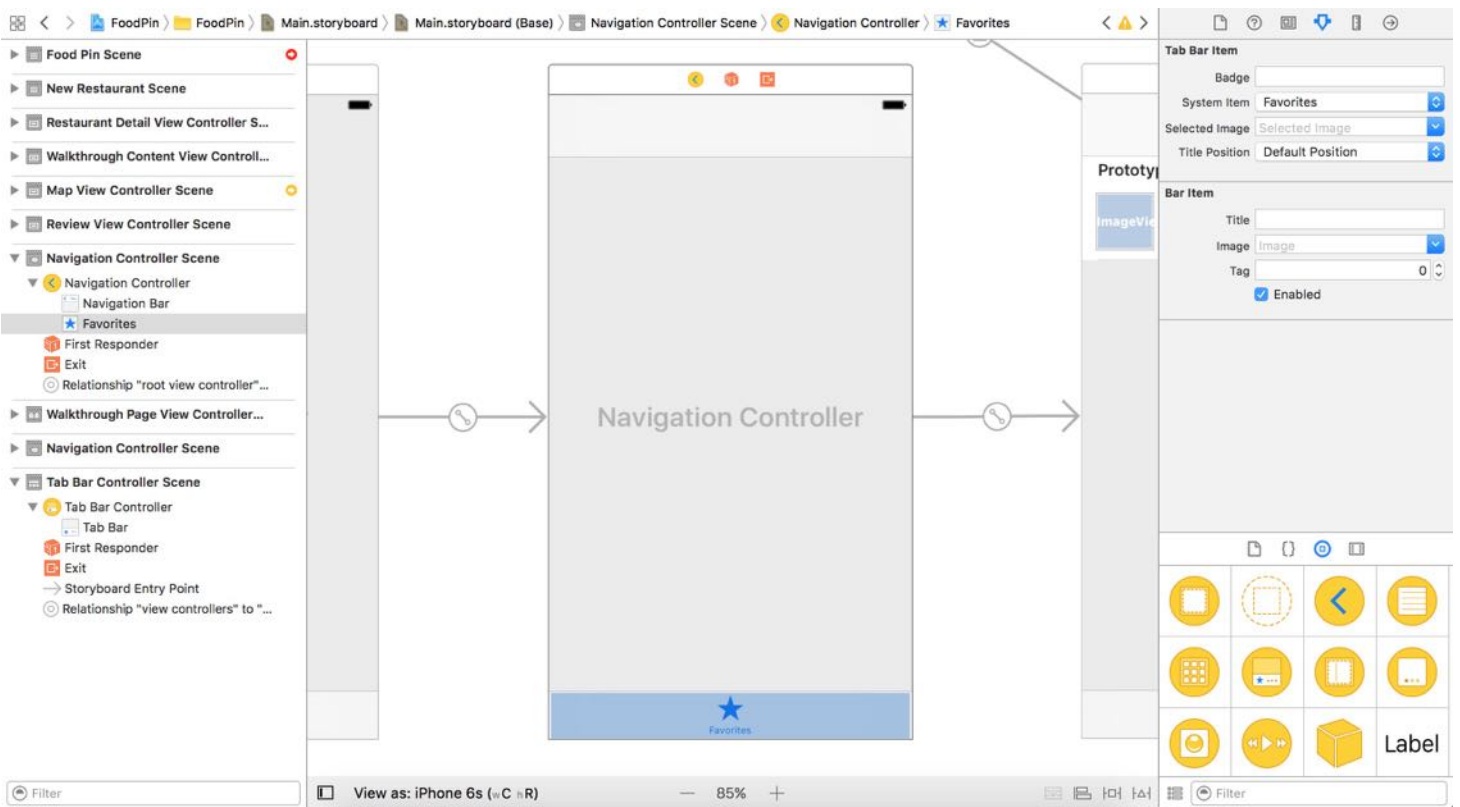


Figure 22-4. Setting the tab bar item

It's now ready to test the app. Hit the Run button and see how it looks. The app should now have a tab bar with the *Favorites* tab.

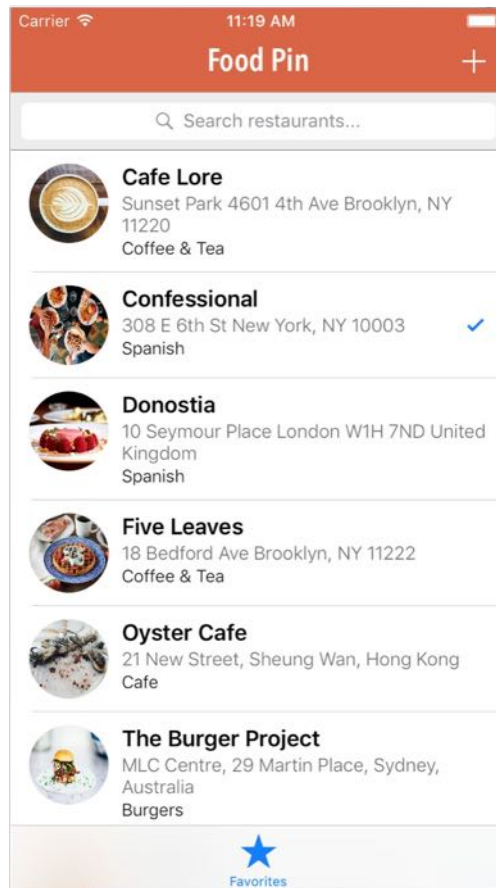


Figure 22-5. The FoodPin app now has a tab bar

Hide Tab Bar When Pushed

Everything we implemented in the earlier chapters are now put right inside a tab bar controller. If you select a restaurant, the app navigates to the detail view, and the tab bar is still there. You may want to hide the tab bar in the detail view or any other views along the navigation hierarchy. iOS provides a simple way to hide a tab bar when a view controller is pushed on a navigation controller.

For example, to hide the tab bar in the detail view, you can select the detail view controller in storyboard and enable the *Hide Bottom Bar on Push* option in the Attribute inspector. The tab bar will be hidden when the detail view appears.

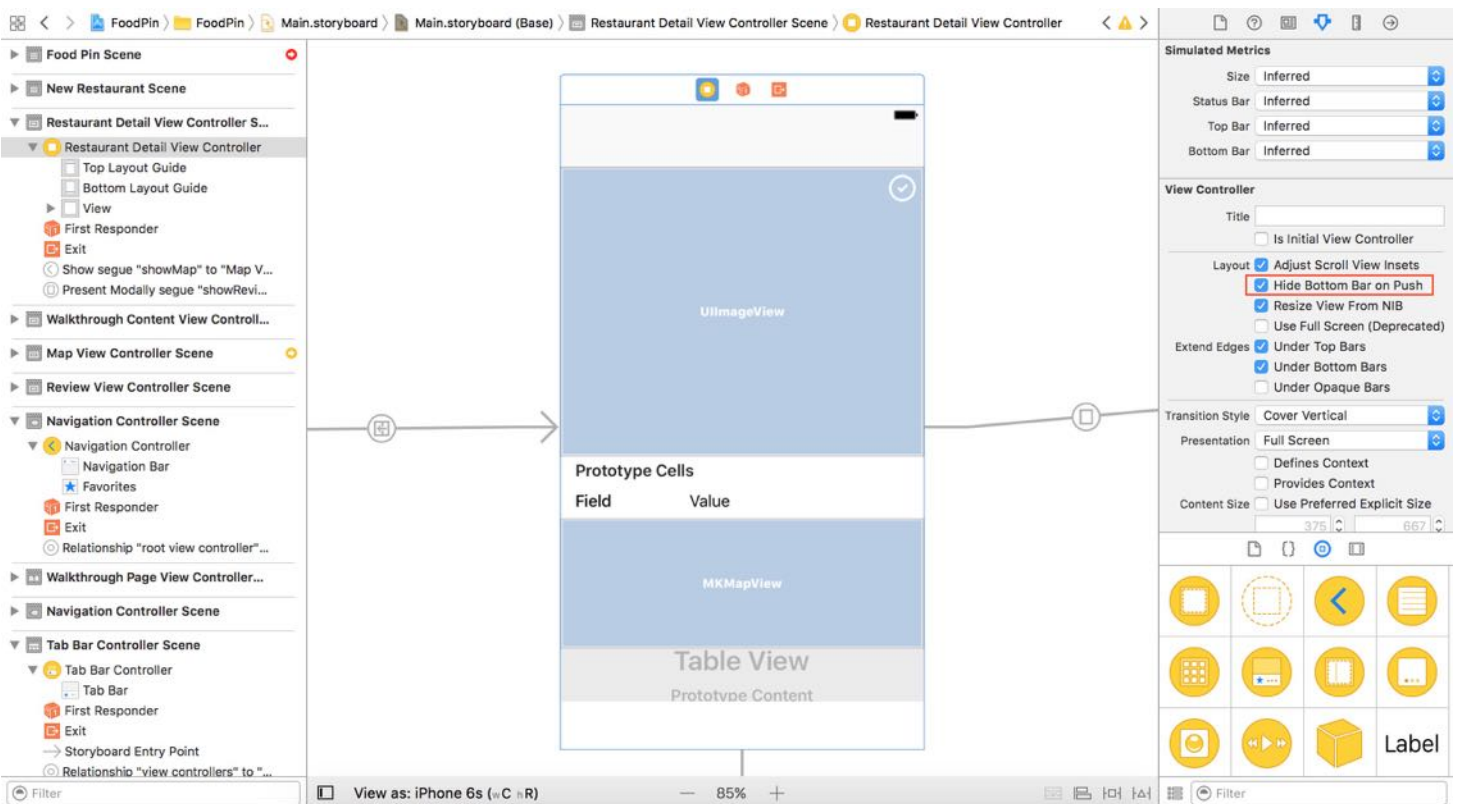


Figure 22-6. Enabling the Hide Bottom Bar on Push option for the detail view controller

Alternatively, if you want to do it in code, you can add the following line of code in the `prepare(for:)` method of the `RestaurantTableViewViewController` class:

```
destinationController.hidesBottomBarWhenPushed = true
```

Try to test the app again. The tab bar should now disappear when you navigate to the detail view.

Adding New Tabs

There is no reason why an app needs a tab bar for displaying a single tab item. You use a tab bar interface to organize your app into distinct modes of operation. Each tab opens a specific feature. In general, there are at least two tabs in an app when a tab bar is used. We're going to create two more tab items: *Discover* and *About*.

Drag a Navigation Controller object into the storyboard. The default navigation controller is

associated with a table view controller. Simply change the title of the navigation bar to *Discover*. For now keep everything else intact. We will deal with the implementation of this tab in later chapters.

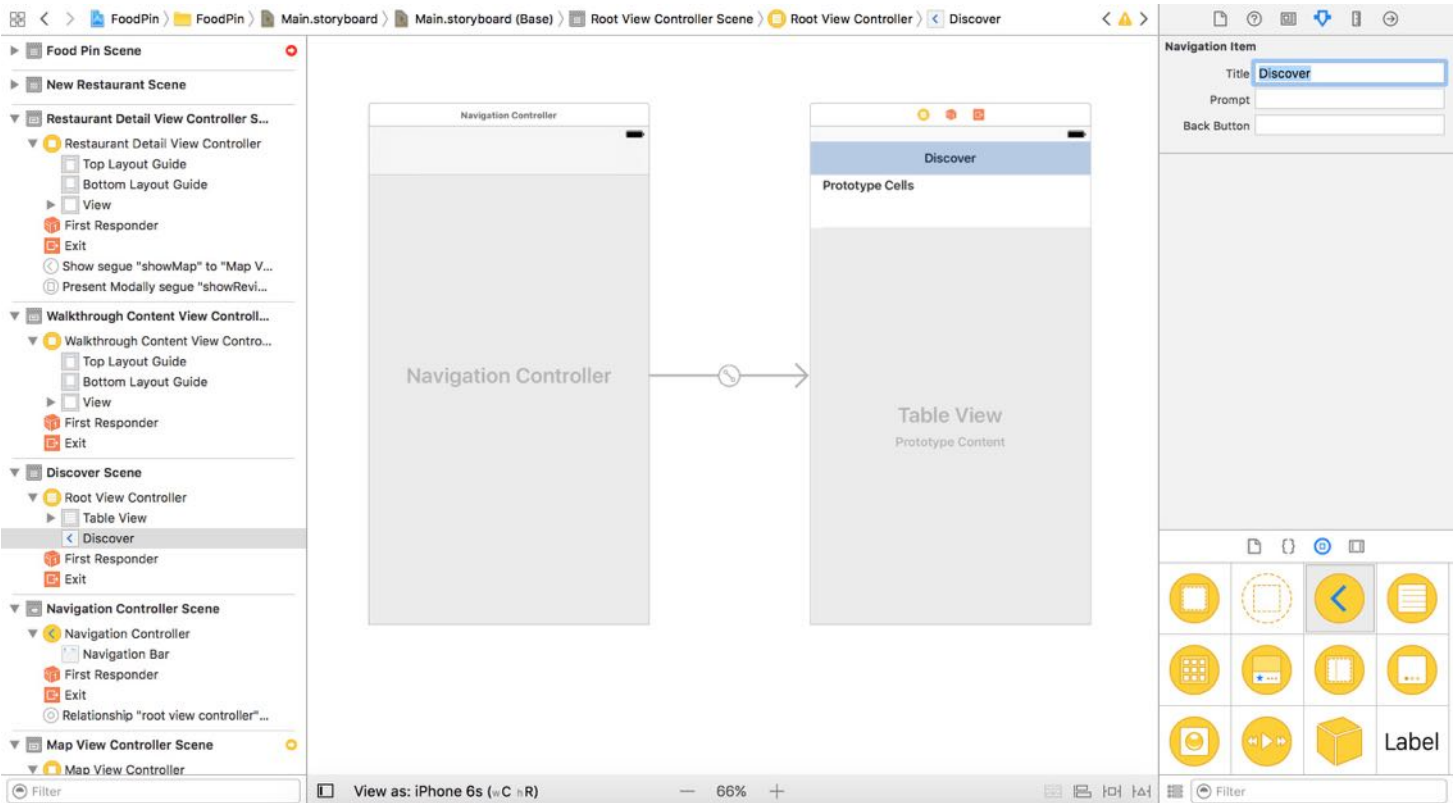


Figure 22-7. Navigation controller and table view controller in the Discover tab

Next, we want to make this set of controllers to be a part of the tab bar controller. All you have to do is establish a relationship between the new navigation controller and the existing tab bar controller. Press and hold the control key, drag from the tab bar controller to the new navigation controller. Release both buttons, and select the `Relationship Segue - View controllers` option. This tells Xcode that the new navigation controller is a part of the tab bar controller.

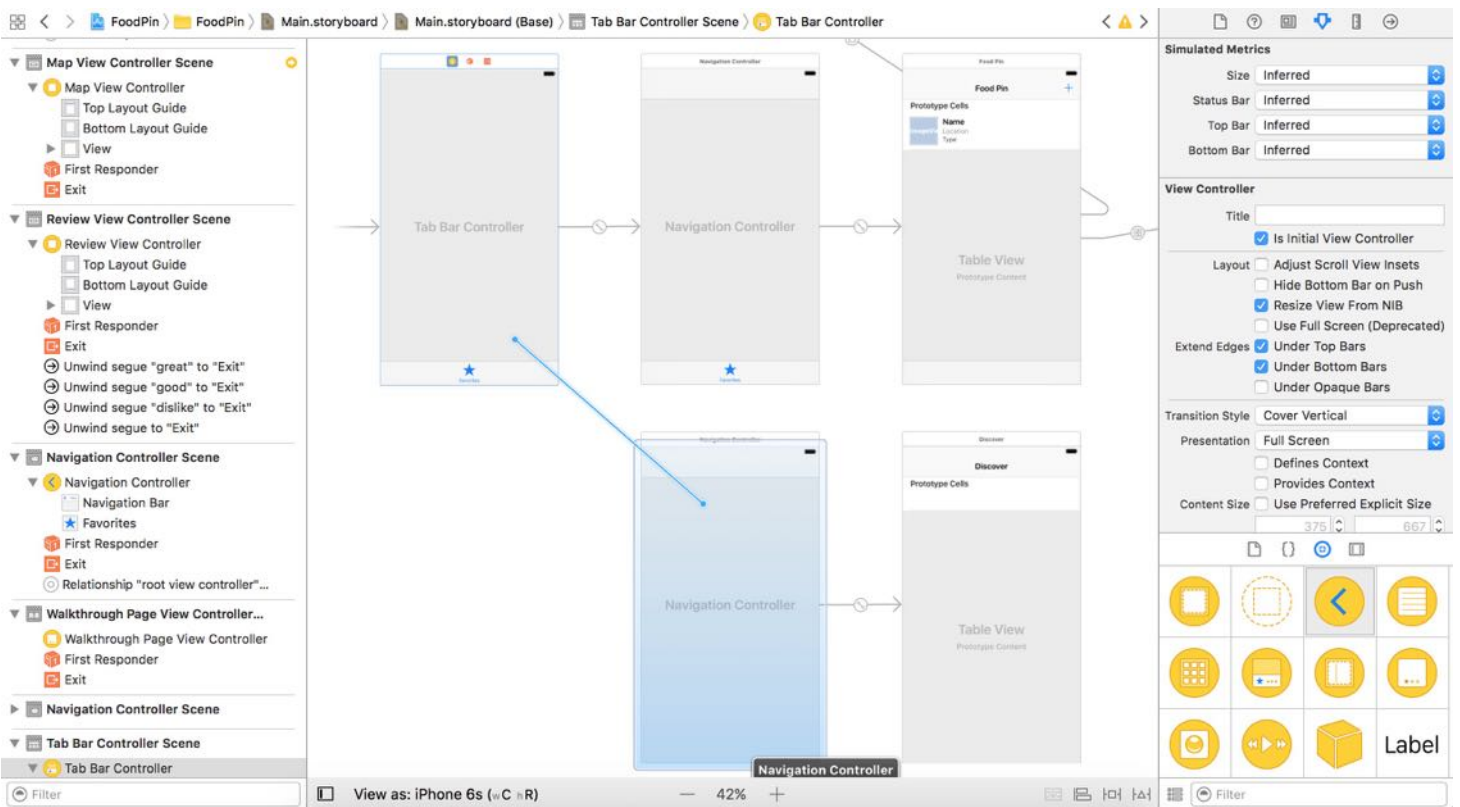


Figure 22-8. Link up the tab bar controller and the new navigation controller

As soon as the relationship is established, the tab bar controller automatically adds a new tab and associates it with the new navigation controller. Change the tab bar item to *Recent*. Save the storyboard and run the app again. You should now see the new *Recents* tab.

Quick note: For the sake of simplicity, we just use the system item. You may wonder how to use your own title and image for the tab item. I will show you how later in this chapter.

Repeat the same procedures to create another tab for *About*. Drag another navigation controller and name the title of the table view controller to *About*. Then establish a relationship with the tab bar controller. This will add another new tab in the tab bar controller. Change the tab bar item of the new navigation controller to the system item *More*. The connections in your storyboard for the two new controllers should be similar to that in figure 22-9.

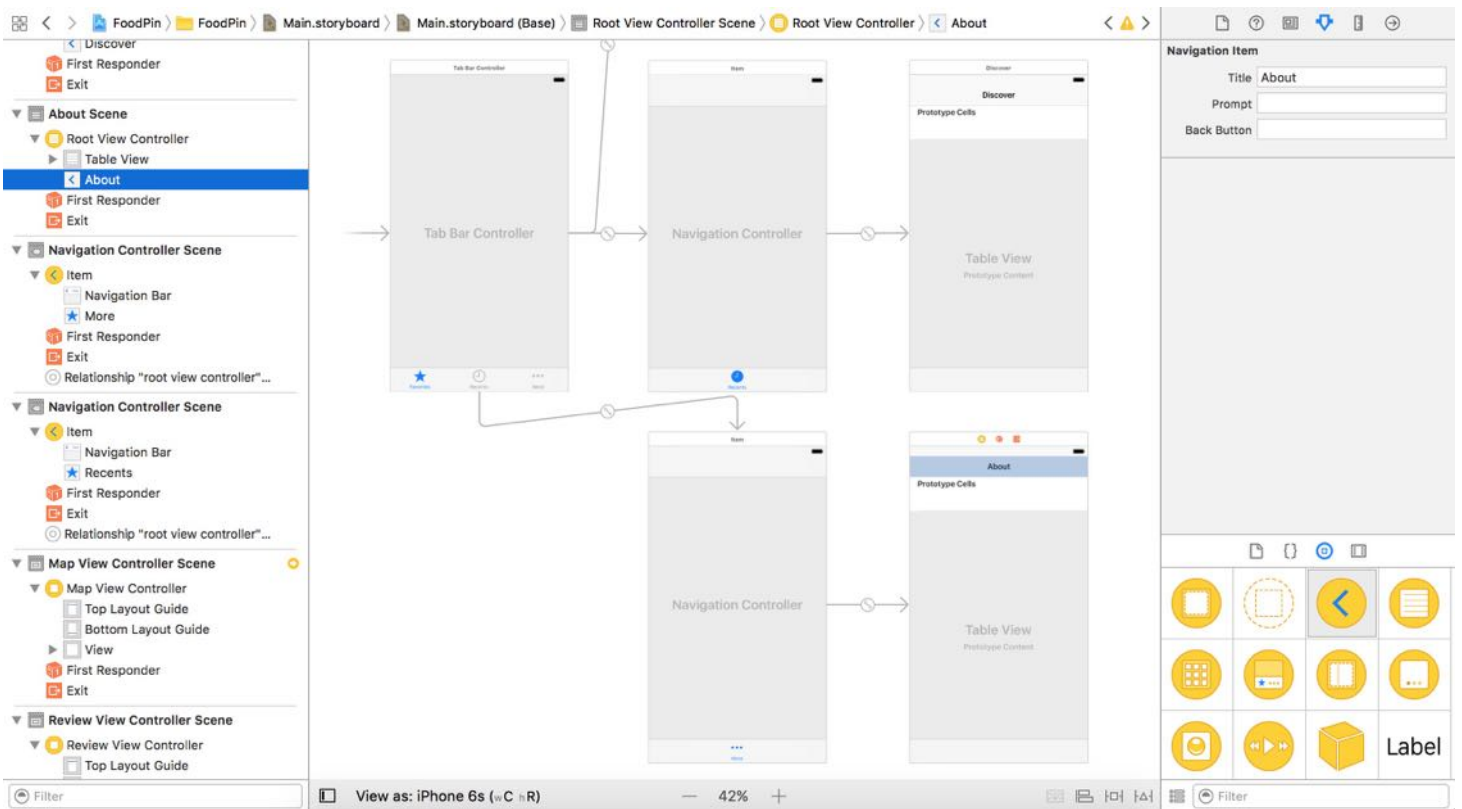


Figure 22-9. The tab bar controllers are now added with two new tab items

Now you're ready to run to the app and have a quick test. The FoodPin app should display a tab bar with three tab items - Favorites, Recent, and More. You're allowed to switch between tabs, though the Discover and About screens are blank.

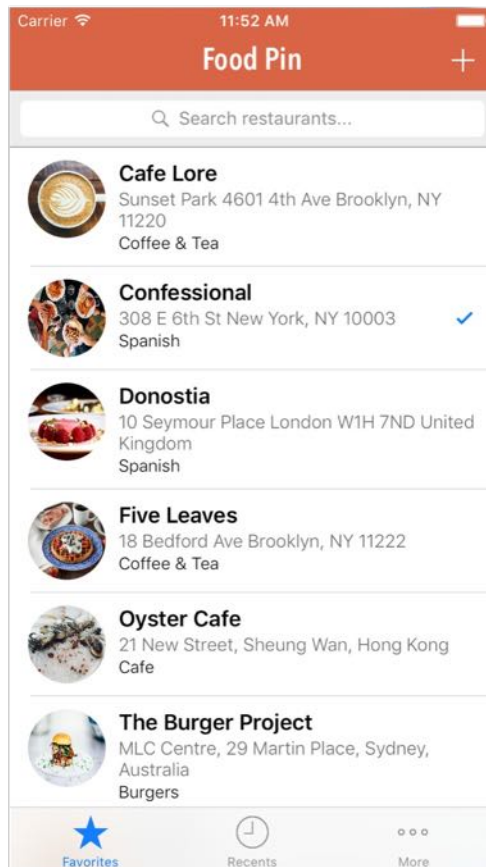


Figure 22-10. The FoodPin app now has three tabs

Customizing the Appearance of the Tab Bar

Up till now, we used the built-in tab items for the tab bar. Similar to the navigator bar, you can use the Appearance API to customize the tab bar's appearance. Here are a couple of the customization properties:

- **tintColor** - this property allows you to change the tint color of the tab bar item.

```
UITabBar.appearance().tintColor = UIColor(red: 235.0/255.0, green: 75.0/255.0, blue: 27.0/255.0, alpha: 1.0)
```

- **barTintColor** - this property lets you change the tint color of the tab bar background. The code below changes the background color to black:

```
UITabBar.appearance().barTintColor = UIColor.blackColor()
```

- **backgroundImage** - this property lets you define a background image of the tab bar.

```
UITabBar.appearance().backgroundImage = UIImage(named: "tabbar-background")
```

Like before, you can put the customization code in the

`application(_:didFinishLaunchingWithOptions:)` method of the `AppDelegate` class. Figure 22-11 shows a sample custom tab bar.

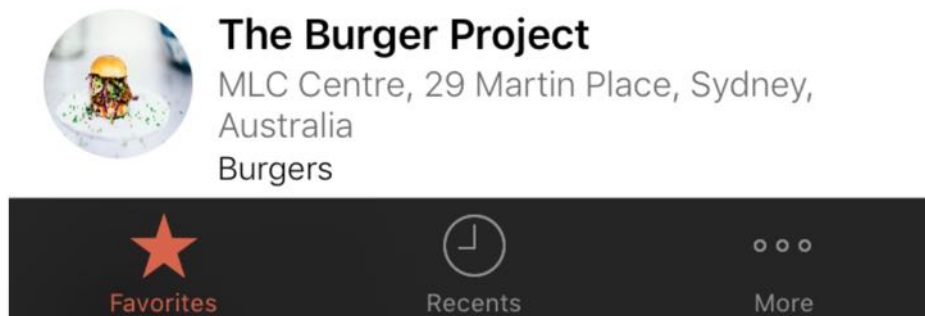


Figure 22-11. Sample tab bar after customization

Quick note: You can further refer to the official documentation https://developer.apple.com/library/ios/documentation/UIKit/Reference/UITabBar_C CH3-SW4 about the appearance properties of the tab bar.

Changing the Tab Bar Item Image

Instead of using the system item, you're allowed to use your own image and title for the tab bar item. You can download these tab bar icons from

<http://www.appcoda.com/resources/swift3/tabbaricons.zip> and import them into

```
Assets.xcasset .
```

To change the tab bar item, select the item in storyboard. In the Attribute inspector, change System Item Option to `Custom` and set the Title/Image to your own values. For the *Recent* tab, set the title to `Discover` and image to `discover` . For the *More* tab, change the title to `About` and image to `about` . Optionally, you can also change the *Favorite* tab to use our own image.

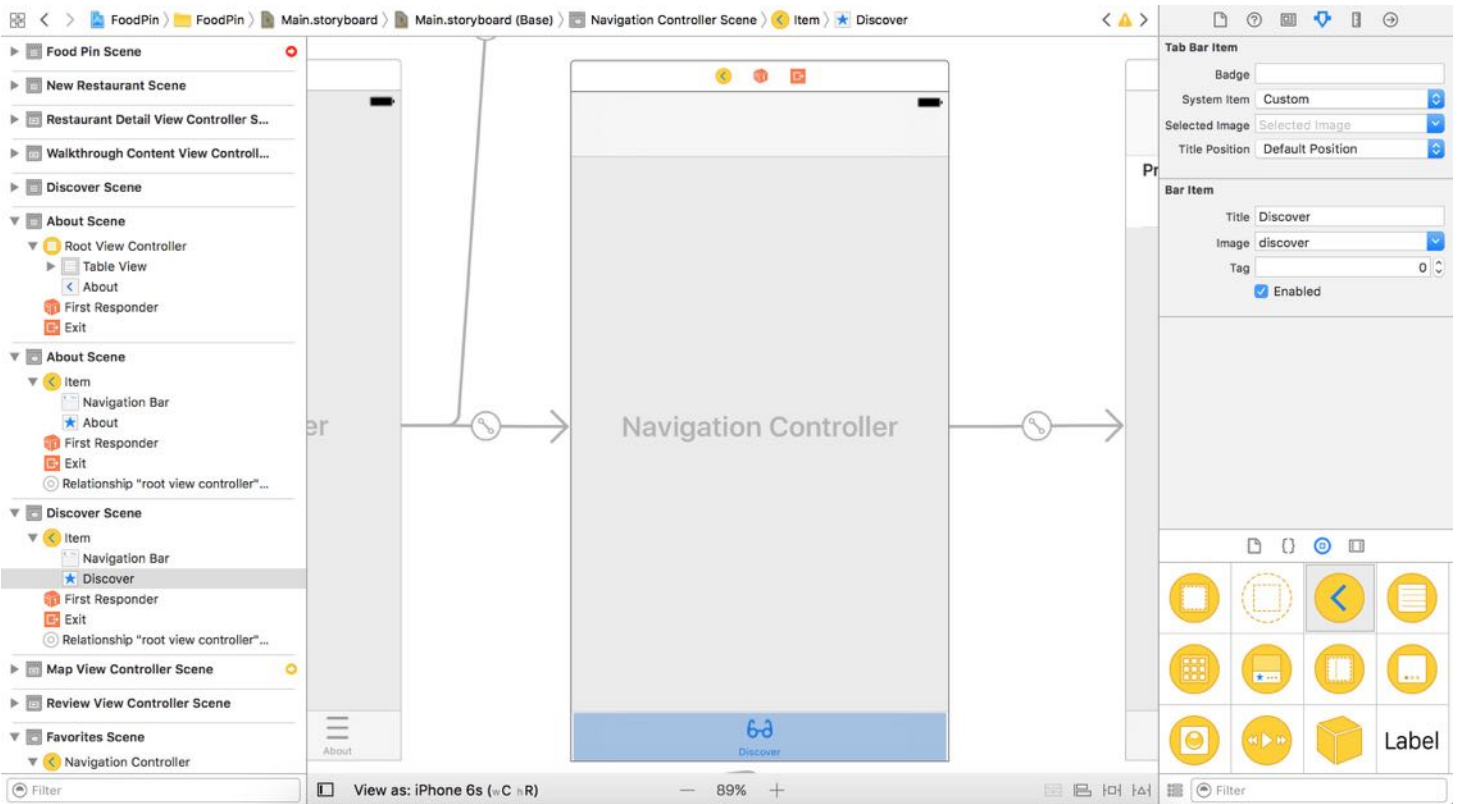


Figure 22-12. Setting the title and image of a custom tab

If you test the app again, the tab bar will be like that shown in figure 22-13.



Figure 22-13. Change the image and title of a tab bar item

Note that I have set `barTintColor` to light gray. Here is the code:

```
UITabBar.appearance().barTintColor = UIColor(red: 236.0/255.0, green:
```

```
240.0/255.0, blue: 241.0/255.0, alpha: 1.0)
```

Changing the Selection Indicator Image

Optionally, you can specify a selection indicator image which is drawn behind the bar item icon when a tab bar item is selected. You can simply set the `selectionIndicatorImage` property to your own image.

```
UITabBar.appearance().selectionIndicatorImage = UIImage(named: "tabitem-  
selected")
```

You can create a simple background image and assign it using the above code. Below shows a sample selection indicator.



Figure 22-14. Sample selection indicator image

Storyboard References

Storyboards are great that allow you to layout the app's UI visually. One complaint is that storyboards are less manageable as your project grows. Developers working in a large team are reluctant to use storyboards. For most app projects, there is a single storyboard. This makes collaboration really difficult.

Starting from Xcode 7, Apple rolled out a feature called *storyboard references* to make storyboards more manageable. Prior to Xcode 7, you can develop your own solution to break a storyboard into smaller pieces, and link them up using code. Now this feature allows you to do the same thing visually, right in Interface Builder.

Let's use the FoodPin project as an example to see how storyboard references work. The app now has three tabs: *Favorites*, *Discover* and *About*. Each of these tabs is implemented with its own UI navigation controller. The view controllers under each tab are mostly independent of the rest of the scenes in the storyboard. Suppose that you're working on the project with two other members. Each team member is responsible for one of the tabs. It makes sense to pull each tab out to its own storyboard.

To do that, go to `Main.storyboard` and select those view controllers that belong the *About* tab. Go up to the editor menu and choose refactor to storyboard.

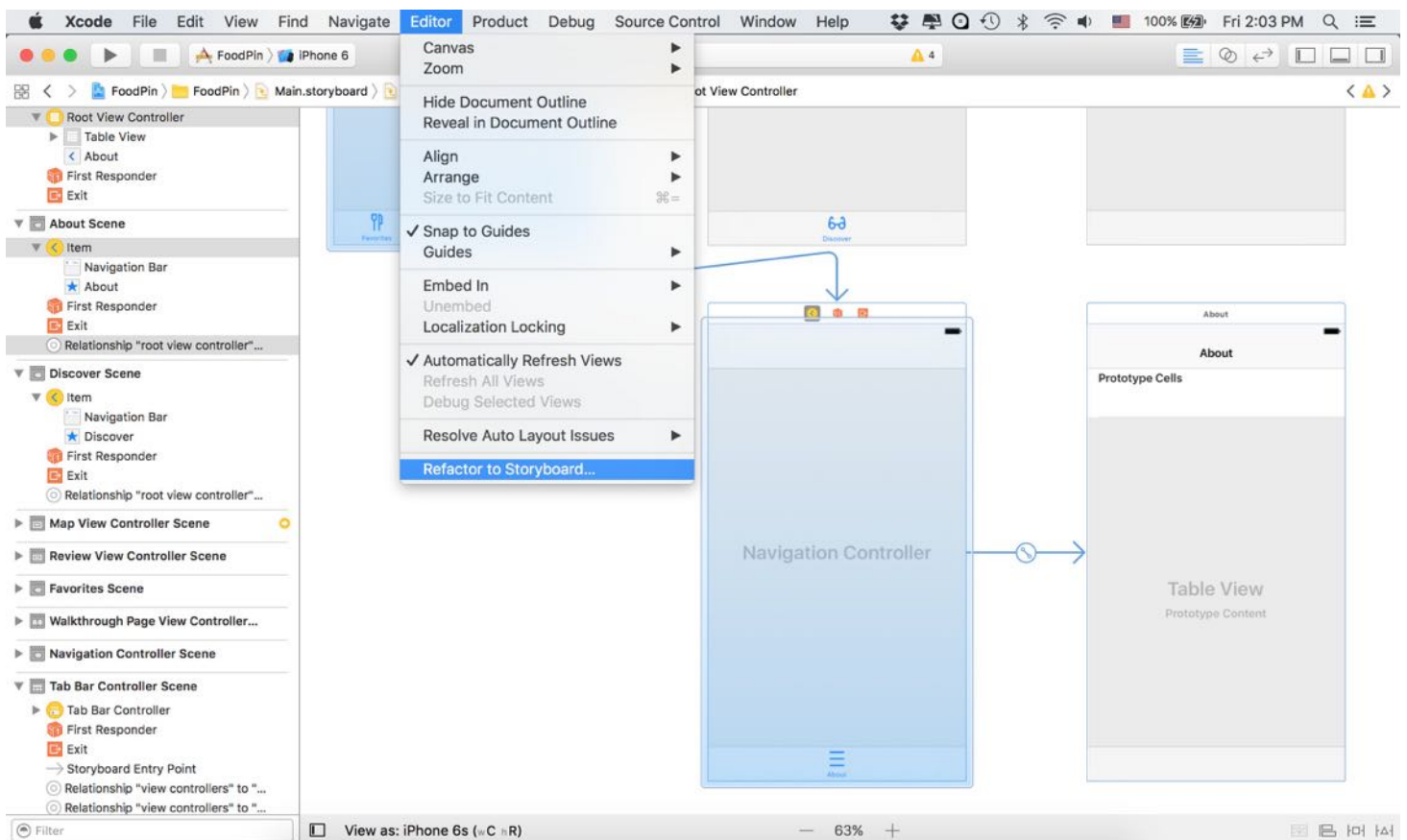


Figure 22-15. Refactor the About view controllers into a new storyboard

When prompted, give the new storyboard a name. Let's name it `about.storyboard`. Once you confirm the change, Xcode pulls out the selected view controllers, and place them in the `about.storyboard` file. In `Main.storyboard`, it shows a storyboard reference. If you double-click on the storyboard reference, Xcode will take you to the corresponding storyboard and

show you the view controllers that the main storyboard is referencing.

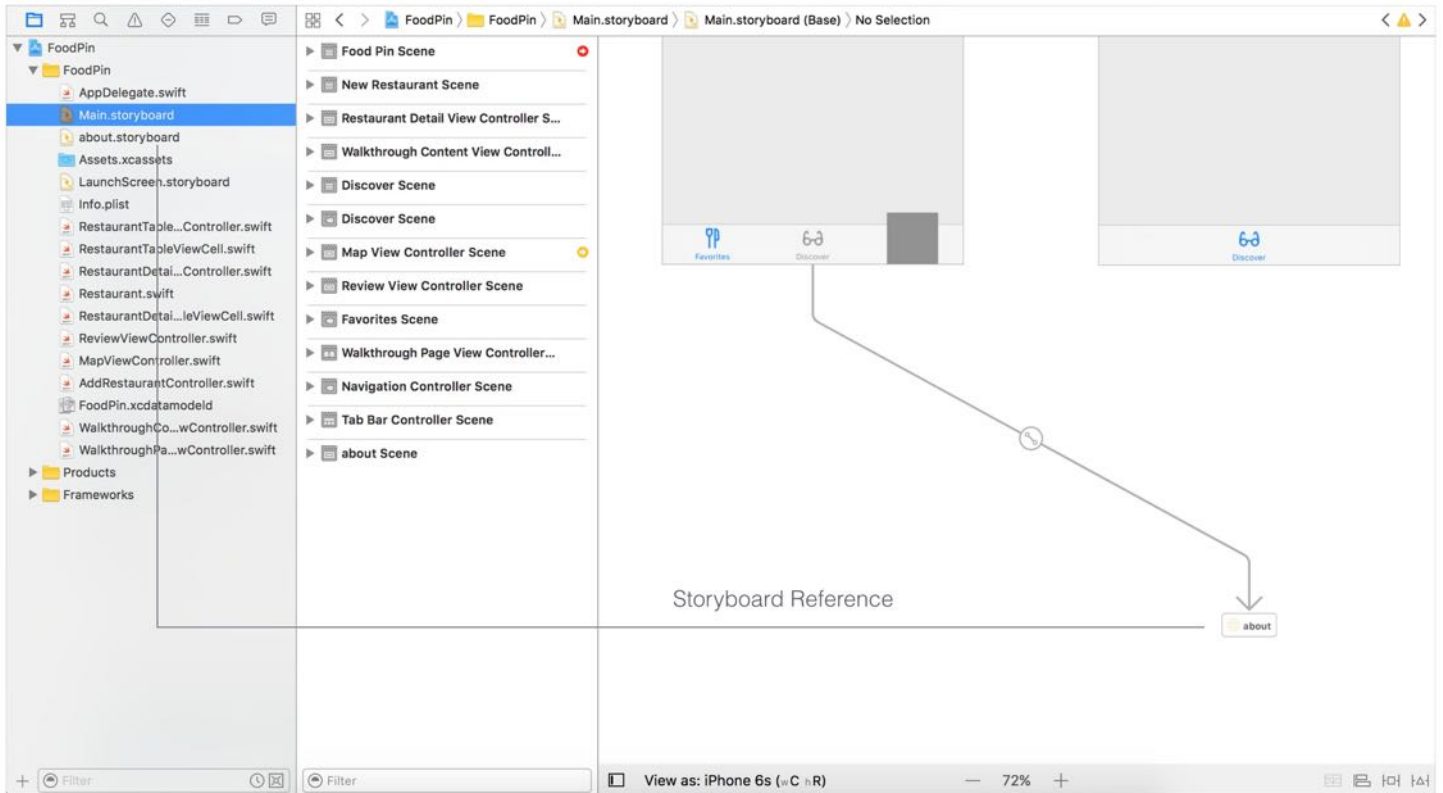


Figure 22-16. A storyboard reference

Repeat the same procedures to refactor the *Discover* tab, and name the new storyboard `discover.storyboard`.

Summary

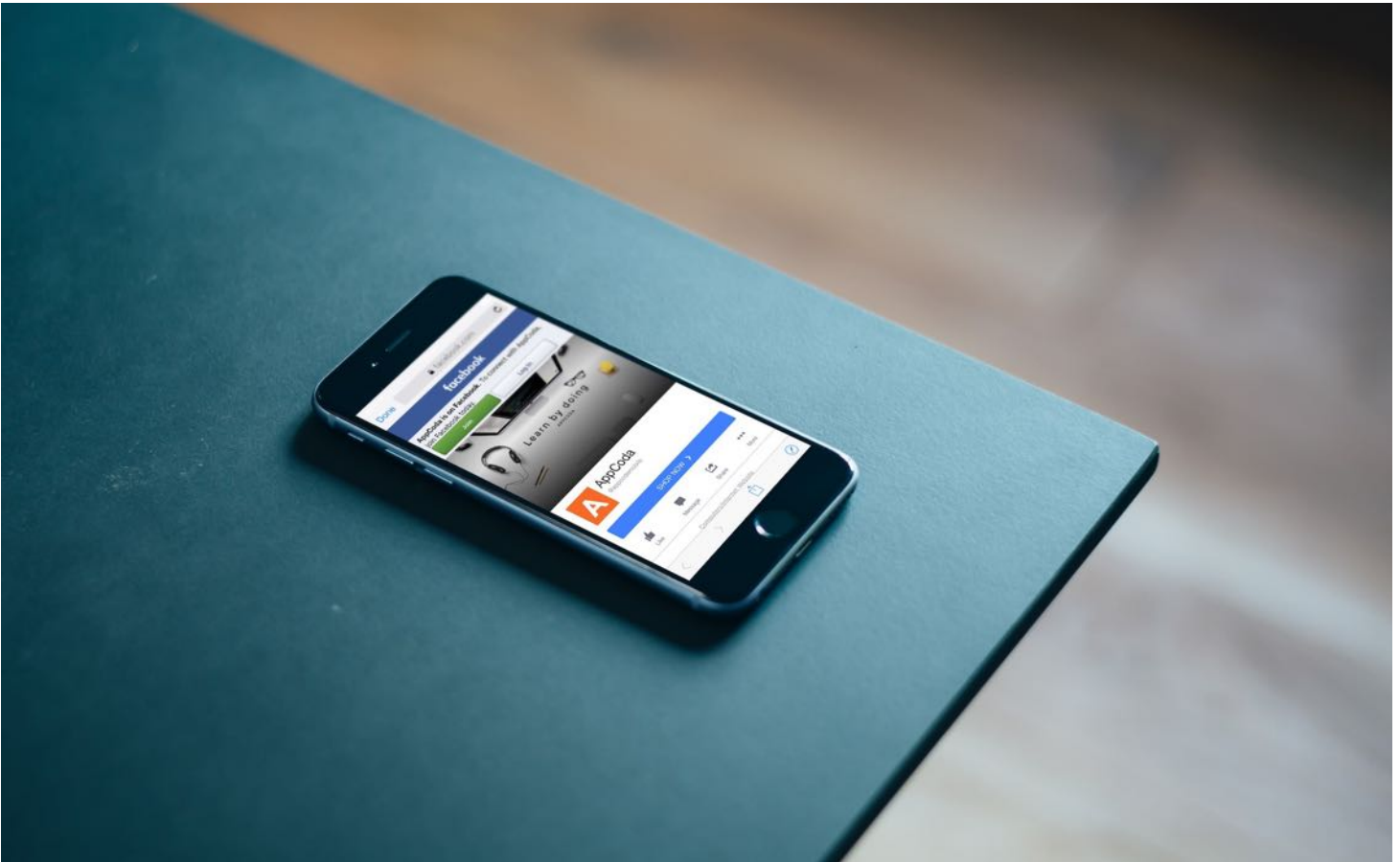
By now you should know how to create a tab bar controller and add new tab bar items. Interface Builder makes it very easy to embed any view controllers in a tab bar controller. Tab bars provide users a quick access to different features of your app.

We also went over a feature of Xcode called storyboard references. As your storyboard becomes more complex, you can break down a storyboard into multiple pieces to stay organized. This feature is especially useful for teams. If you're reluctant to use storyboards, it is time to think again and adopt storyboards in your app projects.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinTabbar.zip>.

Chapter 23

Getting Started with WKWebView and SFSafariViewController



I've got a theory that if you give 100% all of the time, somehow things will work out in the end.

- Larry Bird

It is very common you need to display web content in your apps. From iOS 9 and onward, it provides three options for developers to show web content:

- **Mobile Safari** - the iOS SDK provides APIs for you to open a specific URL in the built-in Mobile Safari browser. In this case, your users temporarily leave the application and

switch to Safari.

- **UIWebView / WKWebView** - Before the release of iOS 9, this is the most convenient way to embed web content in your app. You can think of `UIWebView` as a stripped-down version of Safari. It is responsible to load a URL request and display the web content. `WKWebView`, introduced in iOS 8, is an improved version of `UIWebView`. It has the benefit of the Nitro JavaScript engine and offers more features. If you just need to display a specific web page, `UIWebView` or `WKWebView` is the best option for this scenario.
- **SFSafariViewController** - this is a new controller introduced in iOS 9. While `UIWebView` allows you embed web content in your apps, you have to build a custom web view to offer a complete web browsing experience. For example, `UIWebView` doesn't come with the Back/Forward button that lets users navigate back and forth the browsing history. In order to provide the feature, you have to develop a custom web browser using `UIWebView`. In iOS 9, Apple introduced `SFSafariViewController` to save developers from creating our own web browser. By using `SFSafariViewController`, your users can enjoy all the features of Mobile Safari without leaving your apps.

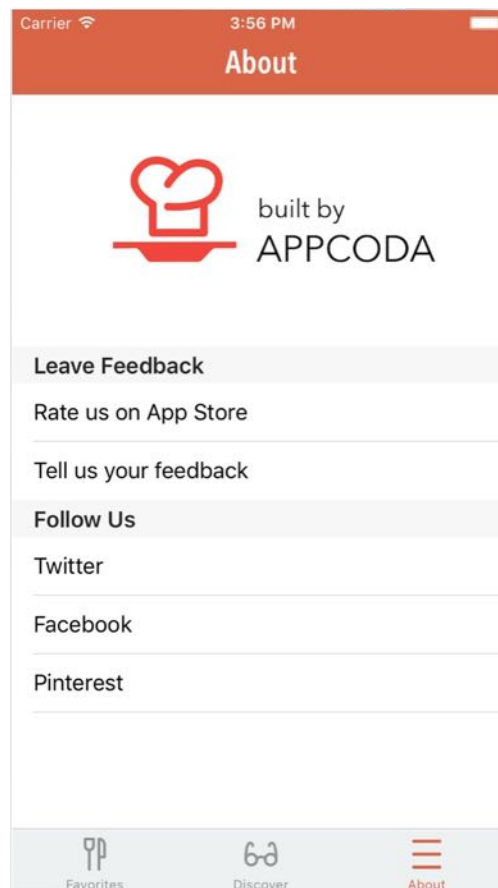


Figure 23-1. The About Screen

In this chapter, I will walk you through all the options and show you how to use them to display web content. We have create `about.storyboard` in the previous chapter. However, we didn't provide any implement yet. Take a look at figure 23-1. That is the *About* screen we're going to create, and here are how each of the rows works:

- **Rate us on App Store** - when selected, we will load a specific iTunes link in Mobile Safari. Users will leave the current app and switch to the App Store.
- **Tell us your feedback** - when selected, we will load a *Contact Us* web page using `WKWebView`.
- **Twitter / Facebook / Pinterest** - each of these items has its own link for the corresponding social profile. We will use `SFSafariViewController` to load these links.

Sounds interesting, right? Let's get started.

Designing the About View

First, download the image pack from <http://www.appcoda.com/resources/swift3/about-logo.zip> and import the images into `Assets.xcasset`.

Open `about.storyboard` to switch to Interface Builder. For the table view controller, drag an image view to the header view of the table view (just like what we have done before when designing the Add Restaurant controller). Change its height to `190` points. In the Attribute inspector, set the image to `about-logo` and mode to `Aspect fit`. Next, select the table view cell. In the Attributes inspector, set the cell's identifier to `cell` and style to `Basic`. Your design should look like figure 23-2.

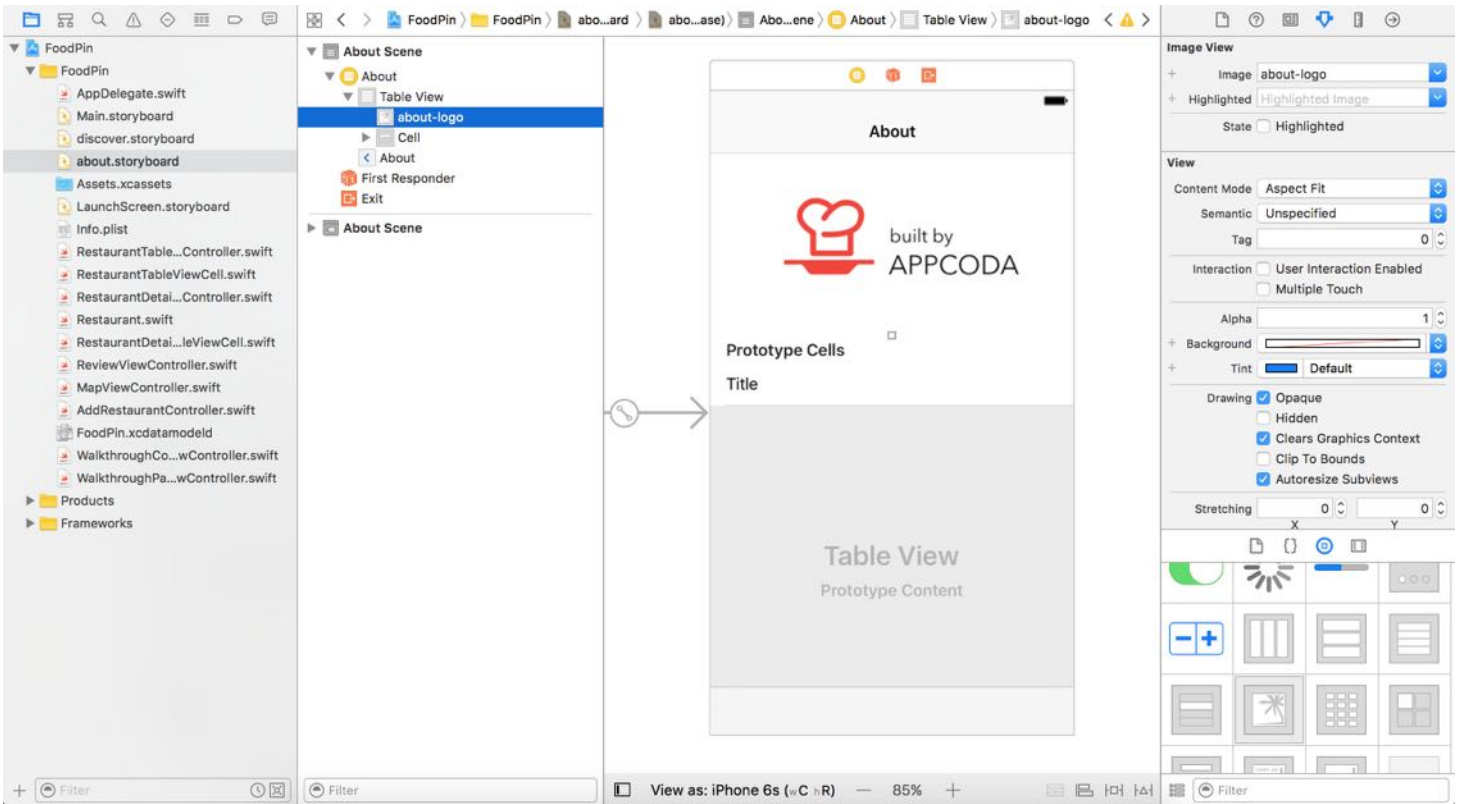


Figure 23-2. The layout of the About view

Creating a Custom Class for the About View Controller

As usual, we need a class to associate with the table view controller in `about.storyboard`. Right-click the `FoodPin` folder and select `New File...`. Name the class `AboutTableViewController` and set its subclass to `UITableViewController`. In `about.storyboard`, select the table view controller and set the custom class to `AboutTableViewController` in the Identity inspector.

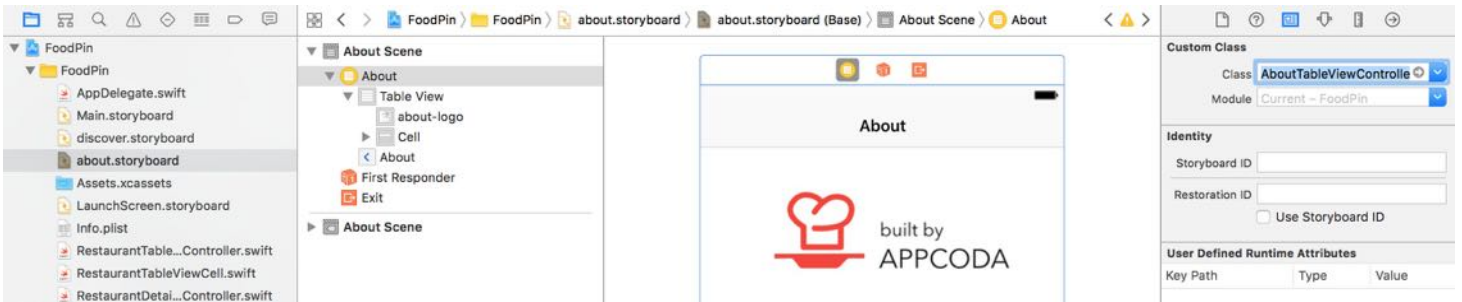


Figure 23-3. Setting the custom class

You should be very familiar with the implementation of table view. But this time, we have to create two sections in the table view. Let's see how to do it.

First, declare three array variables that store the table content:

```
var sectionTitles = ["Leave Feedback", "Follow Us"]
var sectionContent = ["Rate us on App Store", "Tell us your feedback"],
    ["Twitter", "Facebook", "Pinterest"]
var links = ["https://twitter.com/appcodamobile",
    "https://facebook.com/appcodamobile", "https://www.pinterest.com/appcoda/"]
```

The first variable stores the section title. The `sectionContent` array keeps the table row item for each of the section. Lastly, it's the `links` array which holds the URLs of the social profiles.

To create a table view with two sections, all you need to do is return `2` (or `sectionTitles.count`) in the `numberOfSections(in:)` method, and return the corresponding number of rows in the `tableView(_:numberOfRowsInSection:)` method:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    return sectionTitles.count
}

override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

    return sectionContent[section].count
}
```

In other words, the `tableView(_:numberOfRowsInSection:)` method will return `2` for the first section and `3` for the section section.

The `UITableViewDataSource` protocol provides an optional method called `tableView(_:titleForHeaderInSection:)`. To present the section title, you need to implement the method and return the corresponding title for the specified section. Insert the following code in the `AboutTableViewController` class:

```
override func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
    return sectionTitles[section]
}
```

```
}
```

Lastly, implement the `tableView(_:cellForRowAt:)` to set the text label and return the cell:

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

    // Configure the cell...
    cell.textLabel?.text = sectionContent[indexPath.section][indexPath.row]

    return cell
}
```

Before moving onto the next section, run the app and take a quick look. You have already created the user interface of the About screen.

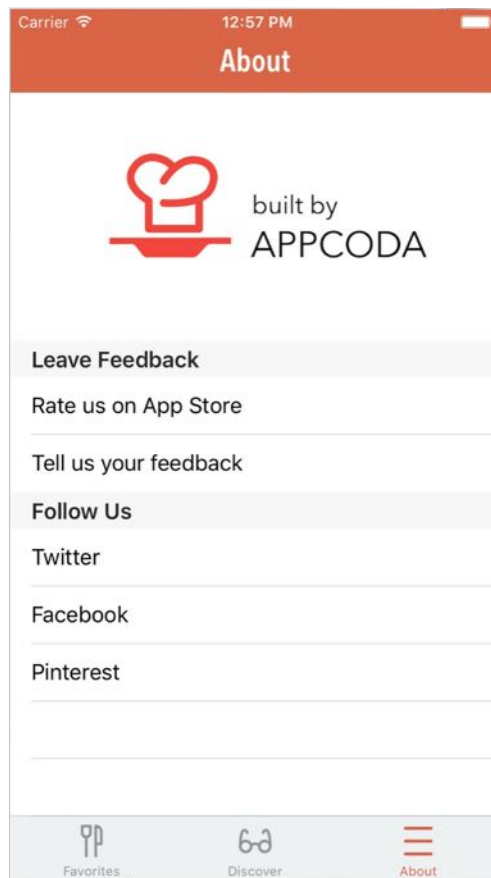


Figure 23-4. The About screen is now populated with content

If you want to remove the separator for those blank table rows after the second section, you can just insert the following line of code in the `viewDidLoad` method:

```
tableView.tableFooterView = UIView(frame: CGRect.zero)
```

Opening Web Content in Mobile Safari

Now that we've prepared the user interface of the About view, let's start to explore the first option of loading web content. When the *Rate us on App Store* option is selected, the app will switch to Mobile Safari and load the content.

To open a link in the Safari browser, you just need to call the

`open(_:options:completionHandler:)` method of `UIApplication` with a specific URL:

```
UIApplication.shared.open(url)
```

When the method is invoked, your user will leave the current application and switch to Safari to load the web content. To handle table cell selection, as you know, we need to override the

`tableView(_didSelectRowAtIndexPath:)` method. Insert the following code snippet:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    switch indexPath.section {
    // Leave us feedback section
    case 0:
        if indexPath.row == 0 {
            if let url = URL(string: "http://www.apple.com/itunes/charts/paid-
apps/") {
                UIApplication.shared.open(url)
            }
        }

    default:
        break
    }

    tableView.deselectRow(at: indexPath, animated: false)
}
```

Here we use a `switch` statement to check the section number. For the first row of the first section (i.e. Rate us on App Store), we call `UIApplication.shared.open(url)` to open the Safari

browser. Typically we specify the iTunes link of the app in the call. However, that link is not ready yet. I simply put an arbitrary link for demo purpose. Later when you're ready to release your app, you can change it accordingly.

Now run the app in the simulator. In the About tab, tap the *Rate us on App Store* item. You'll be switched to Mobile Safari and the web content should be loaded properly.

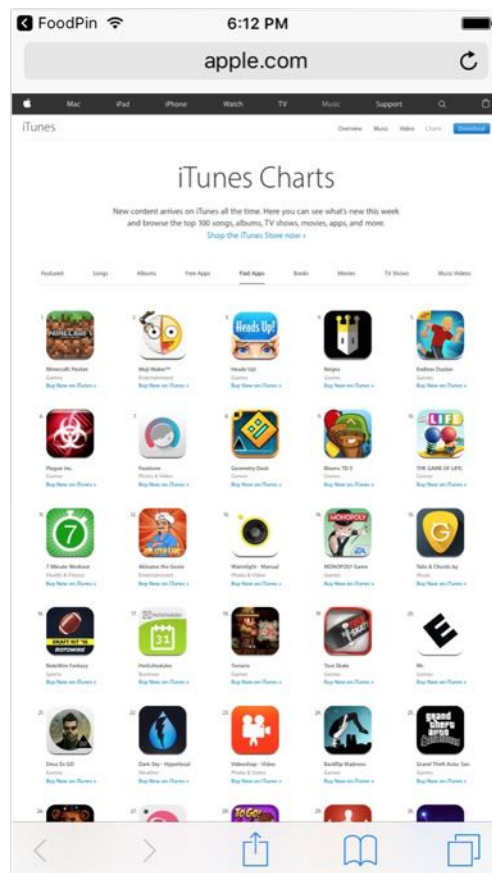


Figure 23-5. Displaying web content in Safari

Loading Web Content Using WKWebView

It's very easy to open a web page in Mobile Safari. To load web content using `WKWebView`, it will take you a few more steps.

The `WKWebView` class is a successor of `UIWebView`, which is capable of loading remote web content. If you bundle an HTML file in your app, you can also use the class to load the web

page locally.

In general, for apps that supports iOS 8 or later, it is recommended to use `WKWebView` instead of `UIWebView` because `WKWebView` outperforms its predecessor in terms of page loading speed.

Practically, you need to create an `URL` object, followed by an `URLRequest` object, and then load the request by calling the `load` method of `WKWebView`. Here is a sample code snippet:

```
if let url = URL(string: "http://www.appcoda.com/contact") {
    let request = URLRequest(url: url)
    webView.load(request)
}
```

The above code instructs the web view (i.e. the instance of `WKWebView`) to load web content remotely. As mentioned before, you're allowed to load a local web page. Let's say, there is a HTML file bundled in your app. You can initialize an `URL` object using the `fileURLWithPath` parameter:

```
let url = URL(fileURLWithPath: "about.html")
let request = URLRequest(url: url)
webView.load(request)
```

`WKWebView` will then load the web page locally, which works even without an Internet connection.

With a basic understanding of the `WKWebView` class, let's start to implement it in the app. In `about.storyboard`, drag a View Controller object from the Object library to the storyboard. Your view controller should look like figure 23-6.

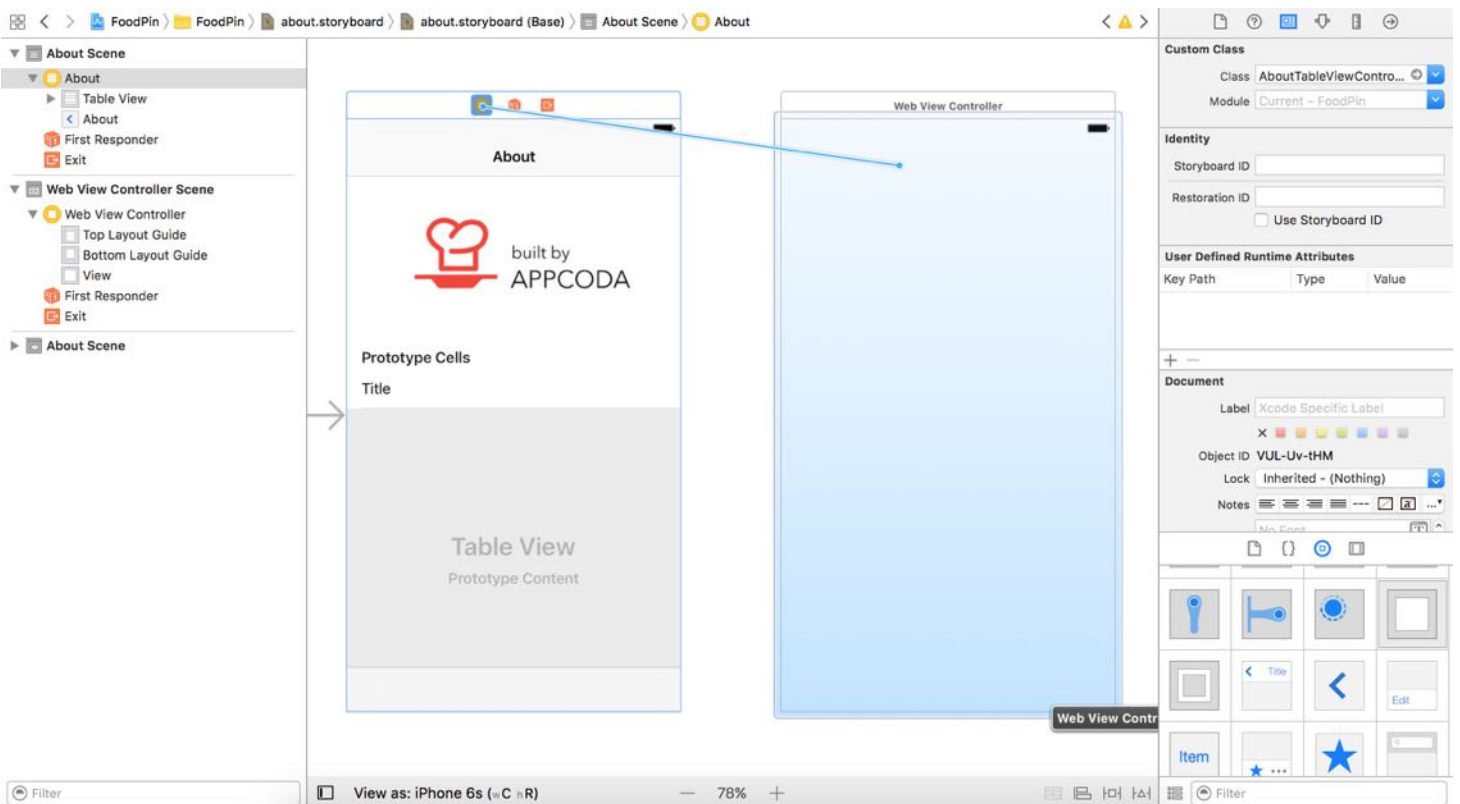


Figure 23-6. Embedding a web view in a view controller

When the *Tell us your feedback* button is tapped, the app will transit to the new view controller to display the web content. To do that, we have to define a segue between the About view controller and the web view controller. Drag from the view controller button in the scene dock to the new view controller. Release the buttons and select *Show* as the segue type. This creates a segue between the two controllers. Select the segue and set the identifier to `showWebView` in the Attributes inspector.

As usual, the next step is to create a custom class the web view controller. Right-click the FoodPin folder in the project navigator, and select `New File...`. Choose the *Cocoa Touch Class* template and name the class `WebViewController`. Set it as a subclass of `UIViewController`. Click `Next` and save the file.

In the `WebViewController.swift` file, declare a variable for the web view:

```
var webView: WKWebView!
```

As `WKWebView` is in the `WebKit` framework, you have to insert the following import statement at the very beginning of the file:

```
import WebKit
```

For the `viewDidLoad` method, update it with the following line of code to load a web page. You're free to change the URL to a website of your choice:

```
override func viewDidLoad() {
    super.viewDidLoad()

    if let url = URL(string: "http://www.appcoda.com/contact") {
        let request = URLRequest(url: url)
        webView.load(request)
    }
}
```

We haven't created the `WKWebView` object yet. What we're going to do is instantiate the `webView` object and replace the `view` of the view controller with `webView`.

In order to achieve this, we will override the `loadView` method with the following code:

```
override func loadView() {
    webView = WKWebView()
    view = webView
}
```

The `loadView` method is called before the `viewDidLoad` method to create the view of the view controller. Here we instantiate the `webView` object and set it as the view.

Now go back to the `AboutTableViewCellController` class and update the `tableView(_didSelectRowAtIndexPath:)` method like this:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    switch indexPath.section {
    // Leave us feedback section
    case 0:
        if indexPath.row == 0 {
            if let url = URL(string: "http://www.apple.com/itunes/charts/paid-
apps/") {
                UIApplication.shared.open(url)
            }
        }
    }
}
```

```

    }
  } else if indexPath.row == 1 {
    performSegue(withIdentifier: "showWebView", sender: self)
  }

  default:
    break
}

tableView.deselectRow(at: indexPath, animated: false)
}

```

We just add another `if` clause for `indexPath.row == 1`, which is the *Tell us your feedback* cell. When the cell is selected, we simply call `performSegue(withIdentifier:sender:)` method to trigger the `showWebView` segue and transit to the web view controller.

Now run the project to test out the changes. You should be able to load the web view controller, but `WKWebView` can't load the URL, showing you a blank page.

What's the problem? It's due to the non-HTTPS URL. Starting from iOS 9, Apple introduced a feature called App Transport Security or ATS for short. The purpose of this feature is to improve the security of connections between an app and web services by enforcing some of the best practices. One of them is the use of secure connections. With ATS, all network requests should now be sent over HTTPS. If you make a network connection using HTTP, ATS will block the request.

To resolve the issue, you should load a network request over HTTPS instead HTTP. That is what Apple wants you to do. However, if the websites that you're talking to is out of your control and that doesn't support HTTPS, how can you work around this restriction?

Apple provides an option for developers to disable ATS restrictions for content loaded inside of web views. To do so, you need to set the value of the specific key named

`NSAllowsArbitraryLoadsInWebContent` in your app's `Info.plist`.

The `Info.plist` file contains essential configuration information for your app. To edit the file, select `Info.plist` in the project navigator to display the content in a property list editor. To add a new key, right click the editor and select *Add Row*. For the *key* column, enter `App Transport Security Settings`. Set the *type* to `Dictionary`. Next, follow the procedures in figure 23-7 to add the `Allow Arbitrary Loads in Web Content` key.

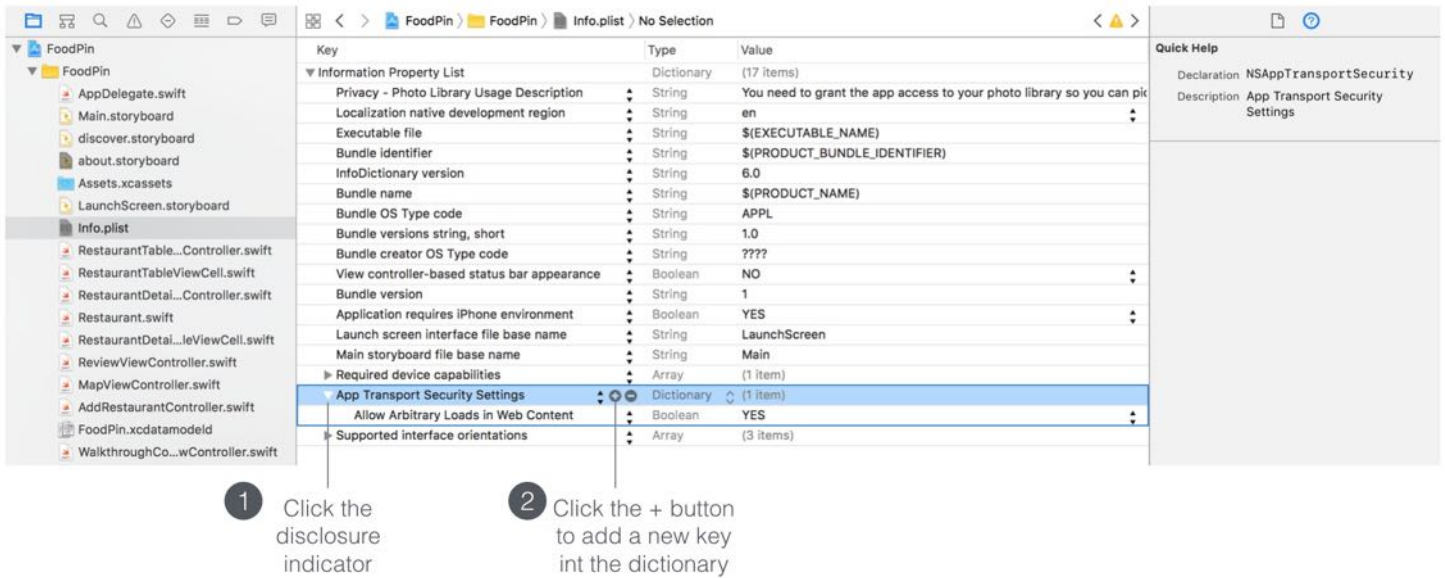


Figure 23-7. Adding the `NSAllowsArbitraryLoads` key

By setting the key to `YES`, you explicitly disable App Transport Security. Now run the app again. It should be able to load the web page.

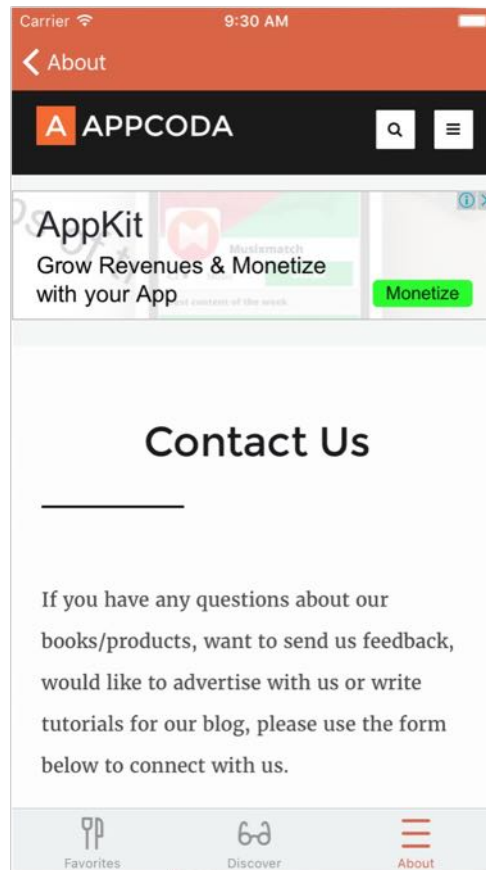


Figure 23-8. Loading a web page using `UIWebView`

Loading Web Content Using `SFSafariViewController`

Finally, let's talk about `SFSafariViewController`. As discussed earlier, this new controller allows developers to embed a Safari browser in your app. It shares many great features of Safari such as Safari AutoFill and Safari Reader. According to Apple, if your app displays web content in a standard browser, it is recommended to use `SFSafariViewController`.

To embed `SFSafariViewController` in your app, all you need is a couple lines of code:

```
let safariController = SFSafariViewController(url: url)
present(safariController, animated: true, completion: nil)
```

You first create the `SFSafariViewController` object with a specified URL. Optionally, you can set `entersReaderIfAvailable` to `true` to enable Safari Reader. The next step is to present the

controller by calling the `present(_:animated:completion:)` method.

Now open the `AboutTableViewController.swift` file. We'll update the code so that the app displays the social links (Twitter/Facebook/Pinterest) using `SFSafariViewController`. First, insert the following import statement at the very beginning of the file:

```
import SafariServices
```

You have to import the `SafariServices` framework before using `SFSafariViewController`. Next, update `tableView(_:didSelectRowAtIndexPath:)` to the following:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath:
IndexPath) {
    switch indexPath.section {
    // Leave us feedback section
    case 0:
        if indexPath.row == 0 {
            if let url = URL(string: "http://www.apple.com/itunes/charts/paid-
apps/") {
                UIApplication.shared.open(url)
            }
        } else if indexPath.row == 1 {
            performSegue(withIdentifier: "showWebView", sender: self)
        }

    // Follow us section
    case 1:
        if let url = URL(string: links[indexPath.row]) {
            let safariController = SF SafariViewController(url: url)
            present(safariController, animated: true, completion: nil)
        }

    default:
        break
    }

    tableView.deselectRow(at: indexPath, animated: false)
}
```

In the above code, we add a new case (`case 1`) to handle the social profile item. The code is very straightforward. We just load the link from the `links` array, and display the web content with `SFSafariViewController`.

It's ready to test the app! Once launched, select *About* and tap a cell in the *Follow us* section.

The app will open the link in a Safari-like browser.

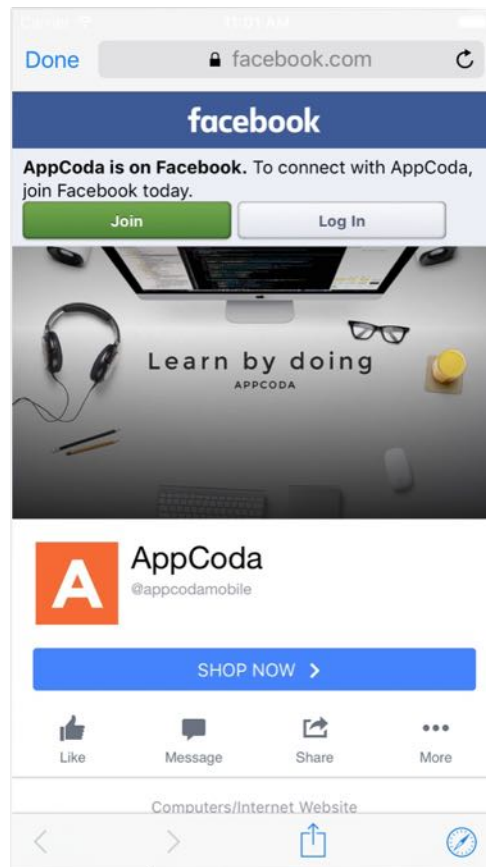


Figure 23-9. Loading a web page using UIWebView

Summary

We have just explored three options for displaying web content. The introduction of `SFSafariViewController` in iOS 9 provides developers a simple way to embed a web browser in your app. With just a few lines of code, you can offer a first class browsing experience for your users. If your app displays web content, the Safari view controller should save you a lot of time from creating your own version of web browser.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinWebView.zip>.

Chapter 24

Exploring CloudKit



The most impressive people I know spent their time with their head down getting shit

down for a long, long time.

- Sam Altman

Let's start with some history. When Steve Jobs unveiled iCloud to complement iOS 5 and OS X Lion at Apple's annual Worldwide Developers Conference (WWDC) in 2011, it gained a lot of attention but came as no surprise. Apps and games could store data on the cloud and have it automatically synchronize between Macs and iOS devices.

But iCloud fell short as a cloud server.

Developers are not allowed to use iCloud to store public data for sharing. It is limited to sharing information between multiple devices that belong to the same user. Take our Food Pin app as an example - you can't use the classic version of iCloud to store your favorite restaurants publicly and make them available for other app users. The data, that you save on iCloud, can only be read by you.

If you wanted to build a social app to share data amongst users at that time, you either came up with your home-brewed backend server (plus server-side APIs for data transfer, user authentication, etc) or relied on other cloud server providers such as Parse.

Note: Parse was a very popular cloud service at the time. But Facebook announced the demise of the service on January 28, 2016.

In 2014, the company reimagined iCloud functionality and offered entirely new ways for developers, as well as, users to interact with iCloud. The introduction of CloudKit represents a big improvement over its predecessor and the offering is huge for developers. You can develop a social networking app or add social sharing features easily using CloudKit.

What if you have a web app and you want to access the same data on iCloud as your iOS app? Apple further takes CloudKit to the next level by introducing CloudKit web services or CloudKit JS, a JavaScript library. You can develop a web app with the new library to access the same data on iCloud as your app.

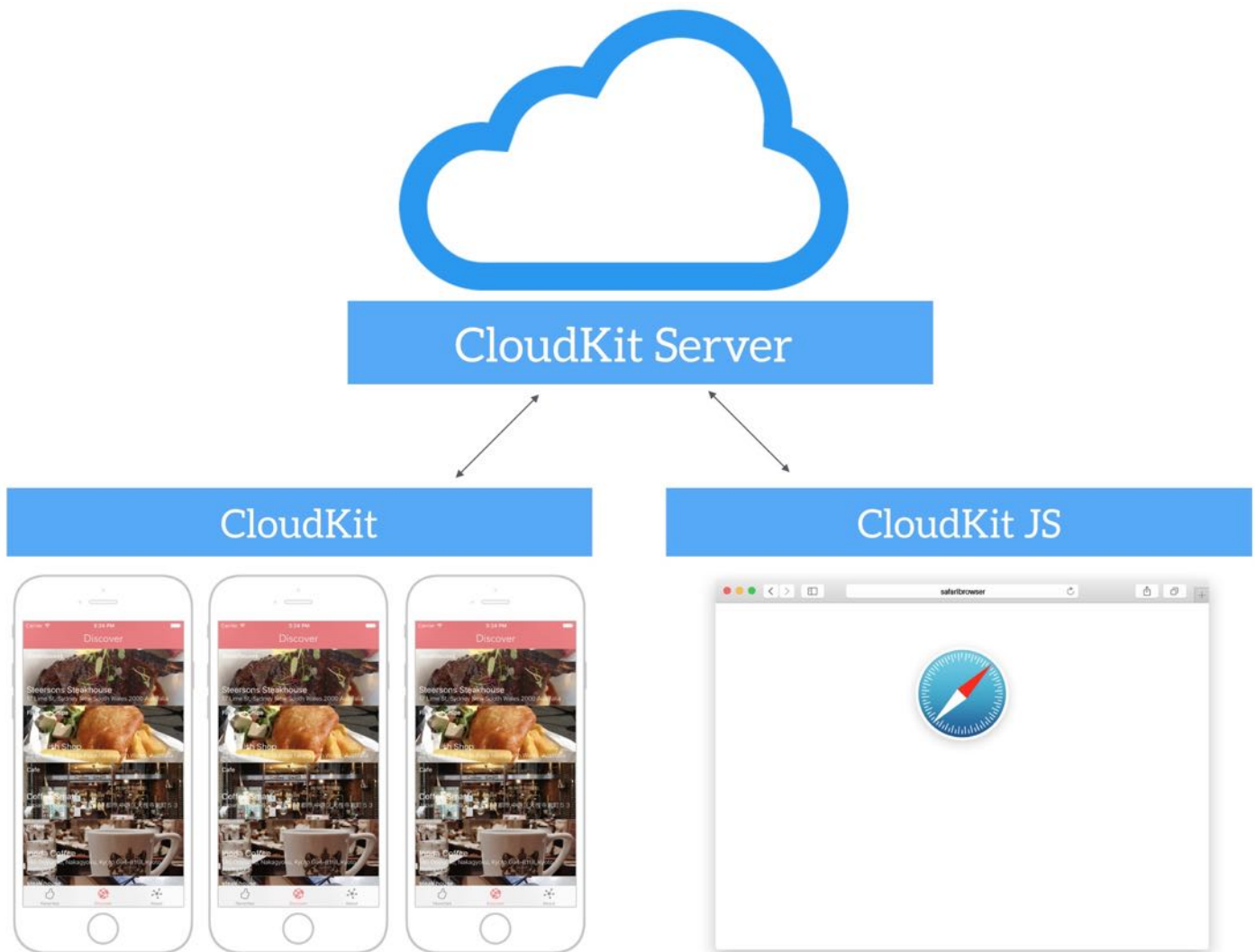


Figure 24-1. Storing your data to the cloud

In WWDC 2016, Apple announced the introduction of Shared Database. Not only can you store your data publicly or privately, CloudKit now lets you store and share the data with a group of users.

CloudKit makes developers' lives easier by eliminating the need to develop our own server solutions. With minimal setup and coding, CloudKit empowers your app to store data, including structured data and assets, in the cloud.

Best of all, you can get started with CloudKit for free (with limits). It starts with:

- 10GB for assets (e.g. image)

- 100MB for database
- 2GB for data transfer

As your app becomes more popular, the CloudKit storage grows with you and adds an additional 250MB for every single user. For each developer account, you can scale all the way up to the following limits:

- 1PB assets
- 10TB database
- 200TB data transfer

That's a massive amount of free storage and is sufficient for the vast majority of apps. According to Apple's [iCloud calculator](#), the storage should be enough for about 10 million free users.

With CloudKit, we were able to focus on building our app, and even squeeze in a few extras.

- Hipstamatic

In this chapter, I will walk you through the integration of iCloud using the CloudKit framework. But we will only focus on the Public database.

As always, you will learn the APIs by implementing a feature of the FoodPin app. We will enhance the app to let users share their favorite restaurants anonymously, and all users can view other's favorites in the *Discover* tab. It's going to be fun.

There is a catch, however. You have to enroll in Apple Developer Program (USD99/year). Apple opens up the CloudKit storage for paid developers only. If you're serious about creating your app, it's time to enroll in the program and build some CloudKit-based apps.

Understanding CloudKit Framework

CloudKit is not just about storage. Apple provides the CloudKit framework for developers to interact with iCloud. The CloudKit framework offers services for managing the data transfer to and from iCloud servers. It's a transfer mechanism that takes your user's app data from the

device and transfers it to the cloud.

Importantly, CloudKit doesn't provide any local persistence and it only provides minimal offline caching support. If you need caching or to persist the data locally, you should develop your own solution.

Containers and databases are the fundamental elements of CloudKit framework. Every app has its own container for managing its content. By default, one app talks to one container. The container is exposed as the `CKContainer` class.

Inside of a container, it contains a public database, a shared database and a private database for storing data. As the name suggests, the public database is accessible by all users of the app and is designed to store shared data. Data stored in the private database is visible to a single user, while data stored in the shared database (introduced in iOS 10) can be shared among a group of users.

Apple lets you choose the type of database that best fits your app. For example, if you're developing an Instagram-like app, you can use the public database to save photos uploaded by users. Or if you're creating a To-do app, you probably want to use the private database to store the to-do items per user. The public database doesn't require users to have an active iCloud account unless you need to write data to the public database. Users need to log into the iCloud before accessing its private database. In the CloudKit framework, the database is exposed as the `CKDatabase` class.

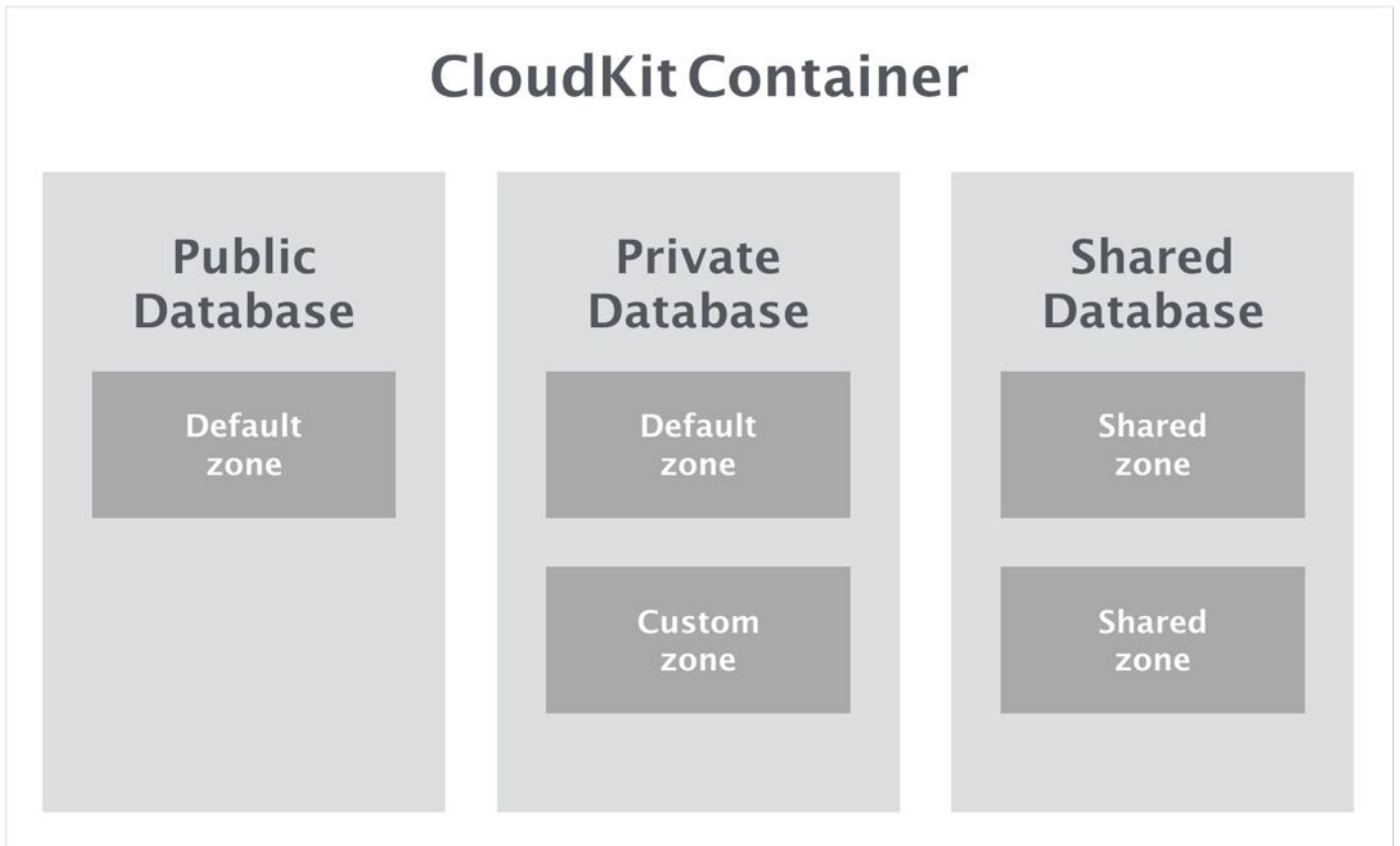


Figure 24-2. Visual Representation of Containers and Databases

Navigating further down the hierarchy is *Record Zone*. CloudKit does not store data loosely. Instead, records of data are partitioned in different *Record Zones*. Depending on the type of database, it supports different types of record zone. Both *private* and *public* databases have a default zone. It is good enough for most scenarios. That said, you're allowed to create custom zones if needed. Record Zone is exposed as the `CKRecordZone` class in the framework.

At the heart of the data transaction is *Record*, represented by the `CKRecord` class. Record is essentially a dictionary of key-value pairs. The key represents a record field. The associated value of a key is the value of a specific record field. Each record has a record type. The record type is defined by developers in the CloudKit dashboard. Meanwhile, you may be confused about all these terms. No worries. You will understand what they mean after going through a working demo.

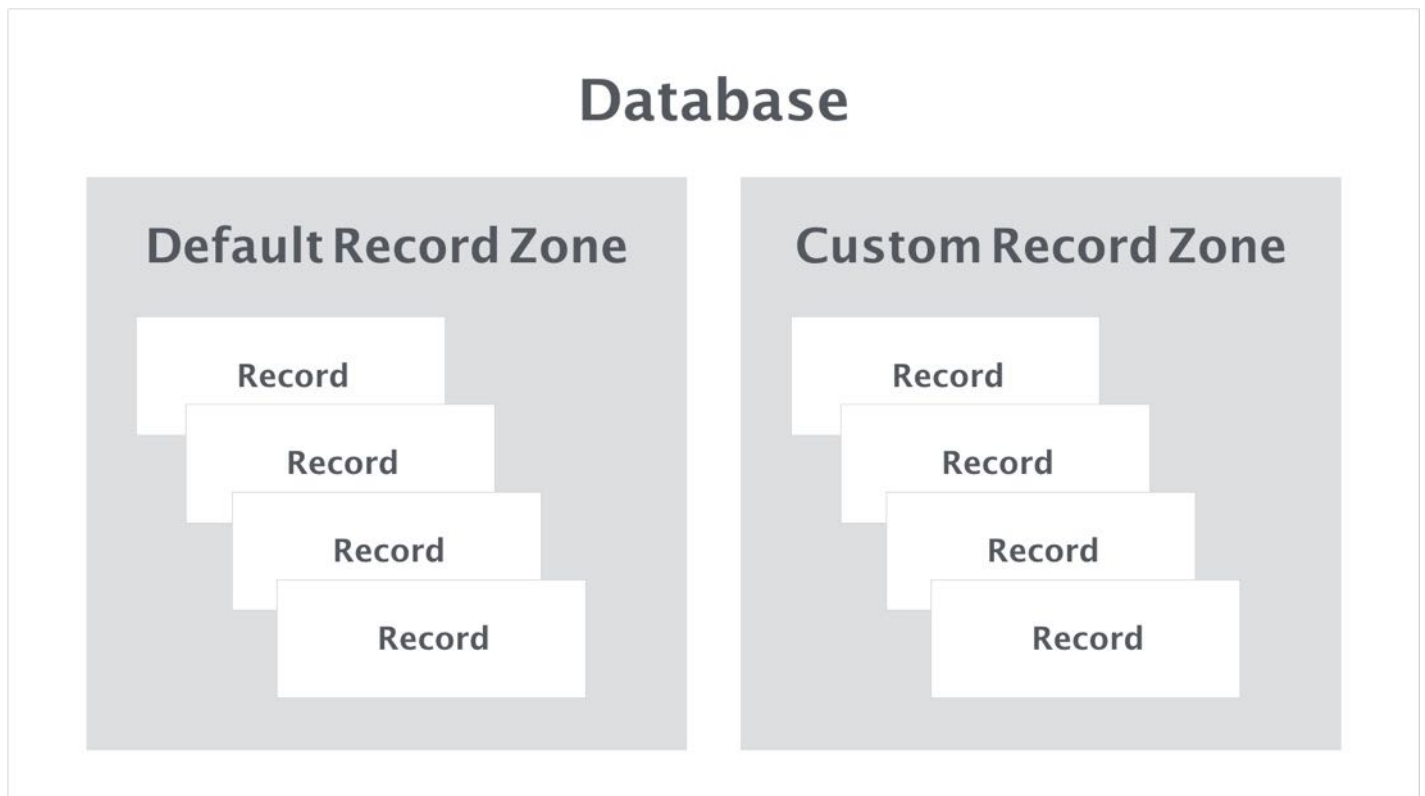


Figure 24-3. Record zones and Records

Now that you have some ideas about the CloudKit framework, let's get started and build the *Discover* tab. By integrating the app with CloudKit, you'll learn:

- How to enable CloudKit in your app
- How to use CloudKit Dashboard to create your records in the cloud
- How to fetch records asynchronously from iCloud using the convenience API
- How to fetch records from iCloud servers using the operational APIs
- How to improve app performance using lazy loading
- How to cache images using NSCache
- How to save data to iCloud servers

Enabling CloudKit in Your App

Assuming you have enrolled in the Apple Developer Program, the first task is to register your account in the Xcode project.

Note: In the project navigator, select the FoodPin project and then select FoodPin under targets. If you are using `com.appcoda.FoodPin` as the bundle identifier, you will need to change it to something else. Say, `.FoodPin`. If you don't own a domain, you may use `.FoodPin`. Later, CloudKit will use the bundle identifier to generate the container. Because the name space of containers is global to all developers, you have to ensure the name is unique.

Under the General tab, if you haven't assigned a developer account in the *Identity* section, simply click the dropdown box of the *Team* option. Select *Add an account*, you'll be prompted to log in with your developer account. Follow the procedures and your developer account will appear in the *Team* option.

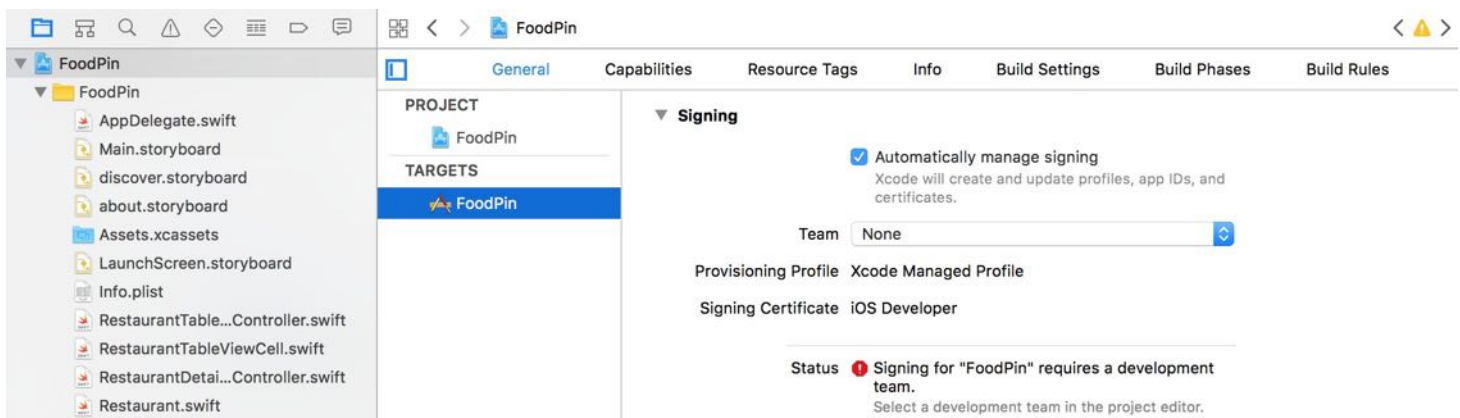


Figure 24-4. Assigning a developer account for your Xcode project

Quick note: For this demo, I will use `com.appcoda.FoodPinCloud` as the bundle identifier.

Assuming you have the identity and bundle identifier configured, navigate to the *Capabilities* tab. To enable CloudKit, all you need to do is flip the iCloud switch to **ON**, and select `cloudKit` for the services option. For containers, just leave it to use the default container. As soon as you enable the iCloud option, Xcode automatically creates the default container on the CloudKit server, and adds the necessary frameworks in the project.

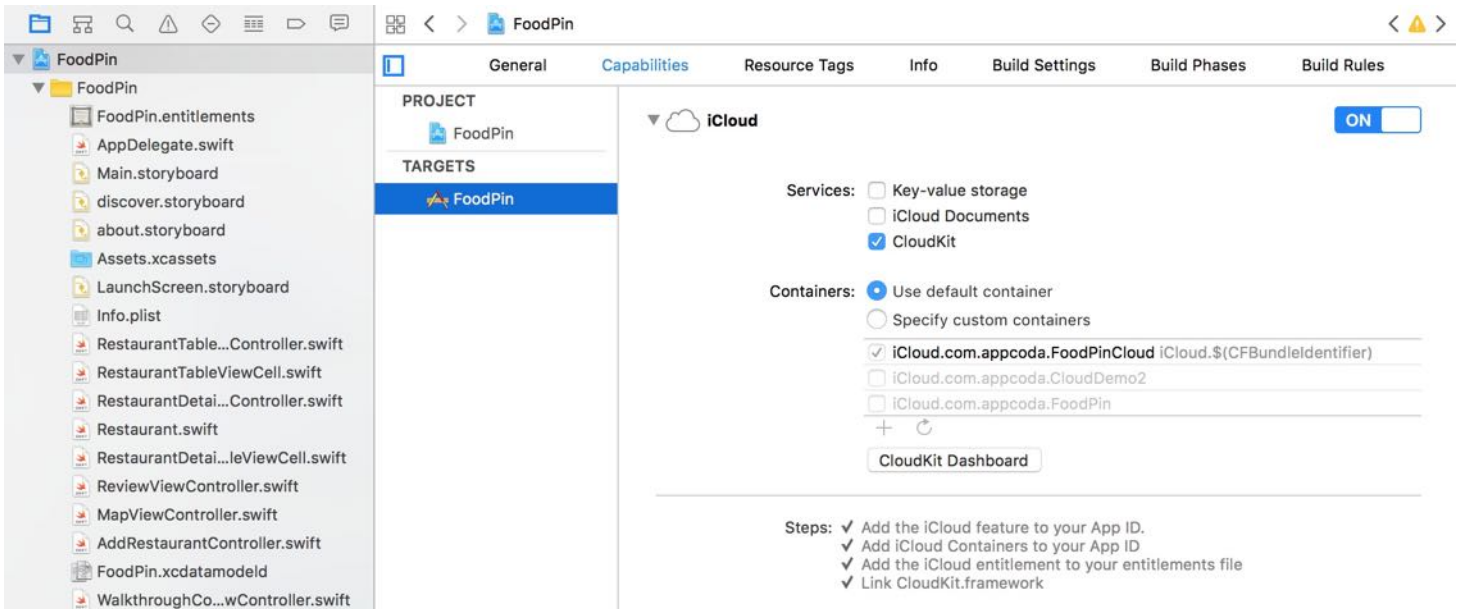


Figure 24-5. Enabling CloudKit for your app

Quick tip: If you experience the error "An App ID with identifier is not available. Please enter a different string.", you may need to choose another bundle identifier.

Managing Your Record in CloudKit Dashboard

You can click the *CloudKit Dashboard* button to open a web-based dashboard (see figure 24-6). The dashboard lets you manage your container and perform operations like adding record types and removing records. Before your app can save restaurant records to iCloud, you first need to define a record type. Do you remember that we created a `Restaurant` entity when working with Core Data? A record type in CloudKit is equivalent to an entity in Core Data.

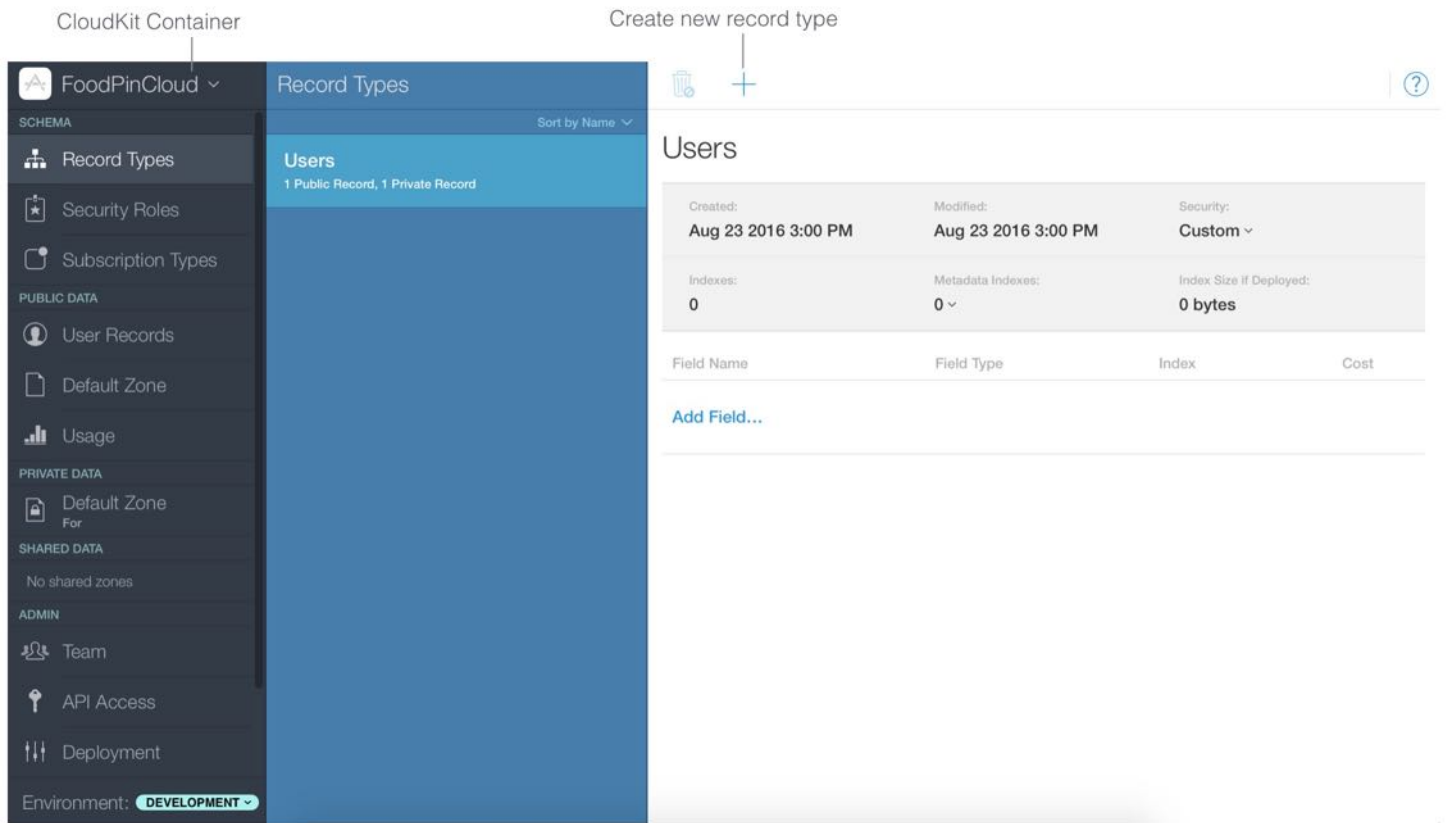


Figure 24-6. CloudKit Dashboard

In the left pane of the dashboard, select *Record Types* and click the + icon to create a new record type. Name the record type `Restaurant`. In the record type, define the field name and field type. CloudKit supports various attribute types such as `String`, `Data/Time`, `Double` and `Location`. If you need to store binary data like image, you use the `Asset` type. Now add the following field names and types for the `Restaurant` record type:

Field Name	Field Type
name	String
type	String
location	String
phone	String
image	Asset

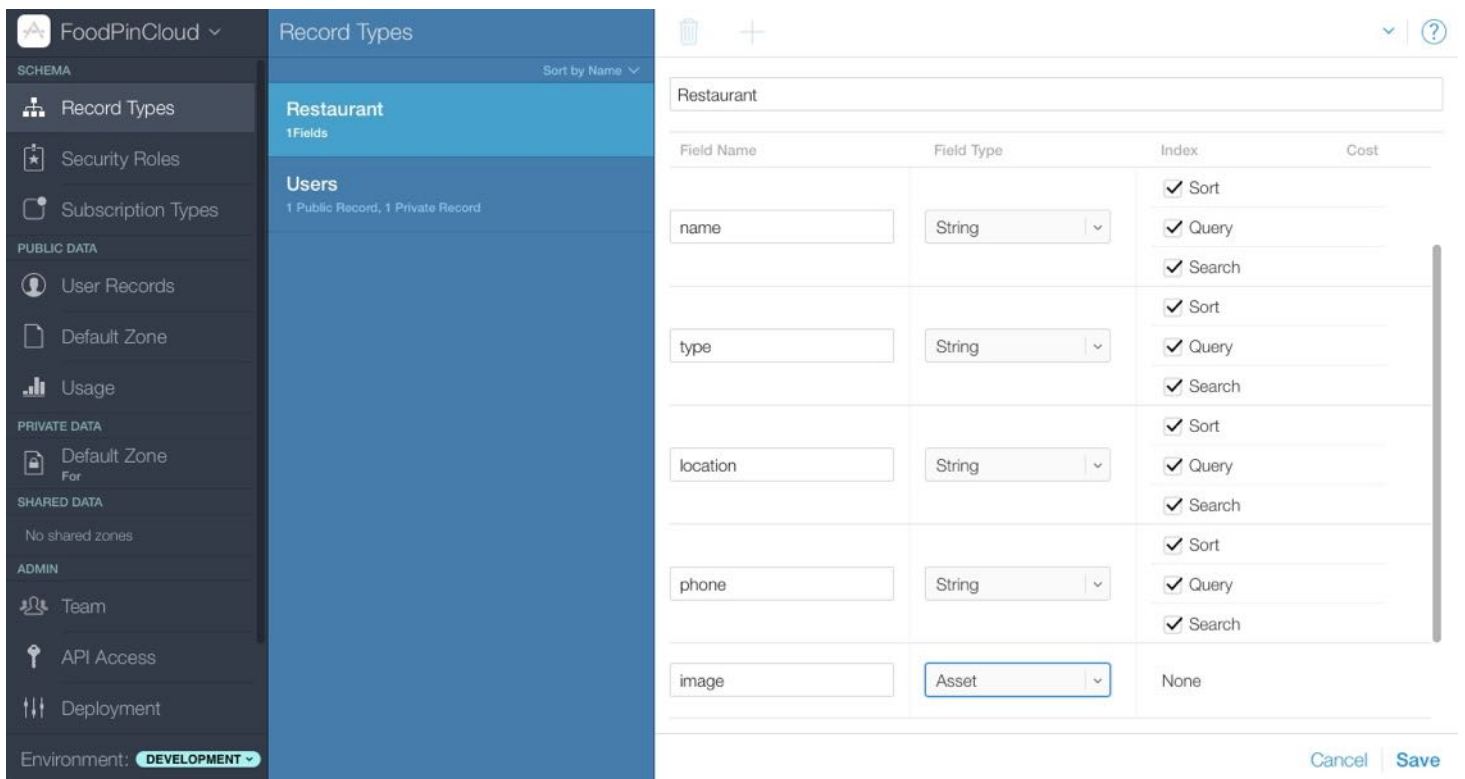


Figure 24-7. Creating the Restaurant record type

Note: CloudKit uses asset objects to incorporate external files such as image, sound, video, text, and binary data files. An asset is exposed as the CKAsset class and associated with a record. When saving an asset, CloudKit only saves the asset data. It does not save the filename. Other than image, you can configure the sort, query and search options for the rest of the fields.

Don't forget to click the *Save* button to save the change. With the record type configured, it's ready for your app to save the restaurant records to iCloud. You have two ways to add records to the database:

- You either create the records through the CloudKit APIs.
- Or you add the records via the CloudKit dashboard.

Let's try to populate some records using the dashboard. In the left pane, select *Default Zone* in the *Public Data* section. This is the default record zone of your public database. By default, the zone doesn't contain any records. You can click the + icon or New Record button to create one. Simply key in the name, type, location, phone, and specify the image location. Then click *Save* to save the record. Figure 24-8 shows a sample new record.

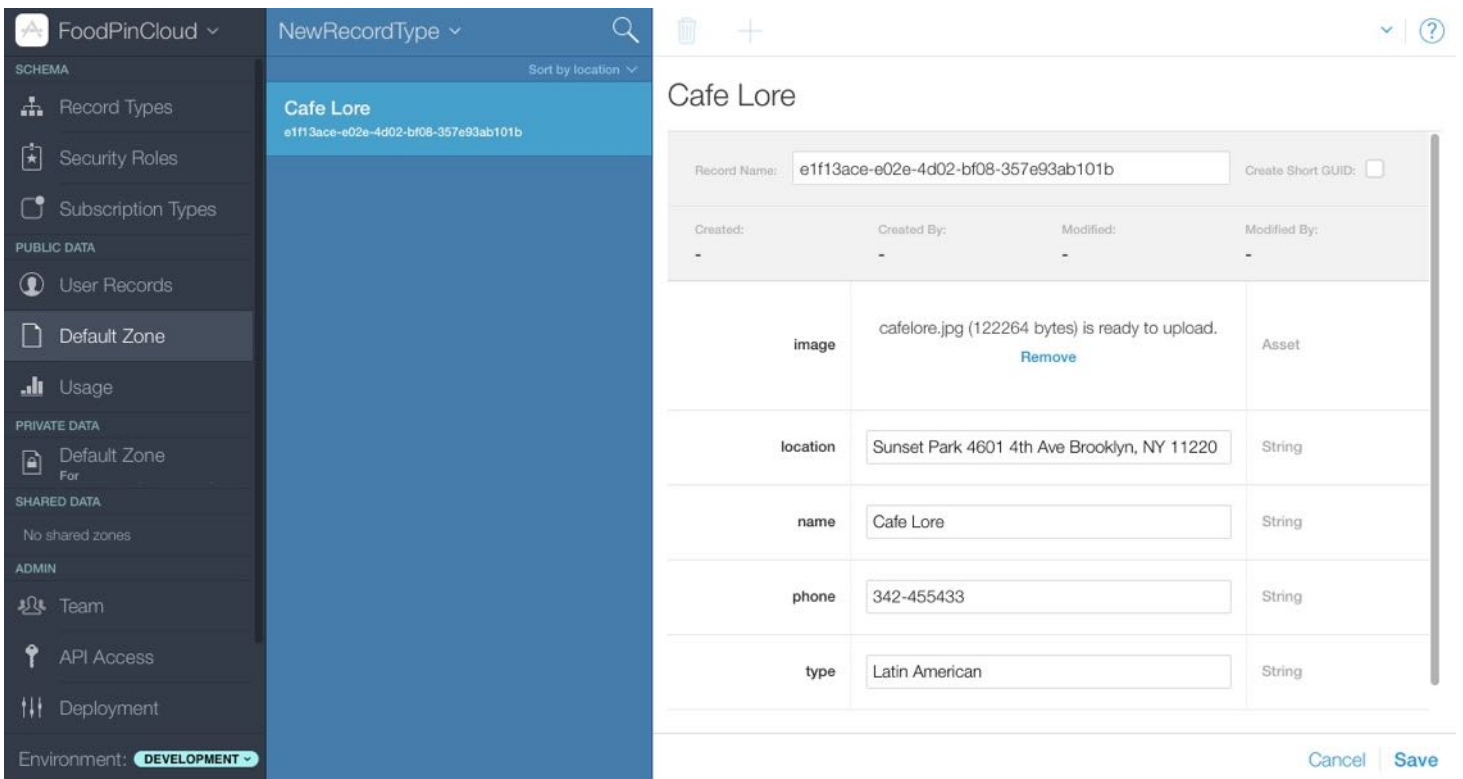


Figure 24-8. Adding a new record in CloudKit

Now you have created a Restaurant record in the cloud. Repeat the same procedures and create around 10 records; we'll use them later.

Fetching Data from a Public Database Using Convenience API

The CloudKit framework provides two types of APIs for developers to interact with iCloud. They are known as the convenience API and the operational API. Both APIs let you save and fetch data from iCloud asynchronously. In other words, the data transfer is executed in the background. We will first go over the convenience API and use it to implement the *Discover* tab. After that, we will discuss the operational API.

As its name suggests, the convenience API allows you interact with iCloud with just a few lines of code. In general, you just need the following code to fetch the `Restaurant` records from the cloud:

```

let cloudContainer = CKContainer.default()
let publicDatabase = cloudContainer.publicCloudDatabase
let predicate = NSPredicate(value: true)
let query = CKQuery(recordType: "Restaurant", predicate: predicate)
publicDatabase.perform(query, inZoneWith: nil, completionHandler: {
    (results, error) -> Void in

// Process the records

})

```

The above code is fairly straightforward. We first get the default CloudKit container of the app, followed by obtaining the default public database. To retrieve the `Restaurant` records from the public database, we construct a `CKQuery` object with the `Restaurant` record type and the search criteria (i.e. `predicate`).

`Predicate` may be new to you. The iOS SDK provides a foundation class called `NSPredicate` for developers to specify how data should be filtered. If you have some database background, you may think of it as the `WHERE` clause in SQL. You can't perform a `CKQuery` without a predicate. Even if you want to query the records without any filtering, you still need to specify a predicate. In this case, we initialize a predicate that always evaluates to `true`. This means we do not perform any sorting on the query result.

Lastly, we call the `performQuery` method of `CKDatabase` with the query. CloudKit then searches and returns the results as an array of `CKRecord`. The search and data transfer are executed in the background. When the records are ready, it will invoke the complete handler for further processing.

Simple, right? Now let's go back to our FoodPin project and implement the *Discover* tab.

First, create a `DiscoverTableViewController` class for the table view controller. In the project navigator, right-click the FoodPin folder and select `New Files...`. Name the class `DiscoverTableViewController` and set it as a subclass of `UITableViewController`.

Once created, go to `discover.storyboard`. Select the table view controller of the *Discover* tab and set its custom class to `DiscoverTableViewController` in the Identity inspector. For the prototype cell, just use the default style and set the identifier to `cell`.

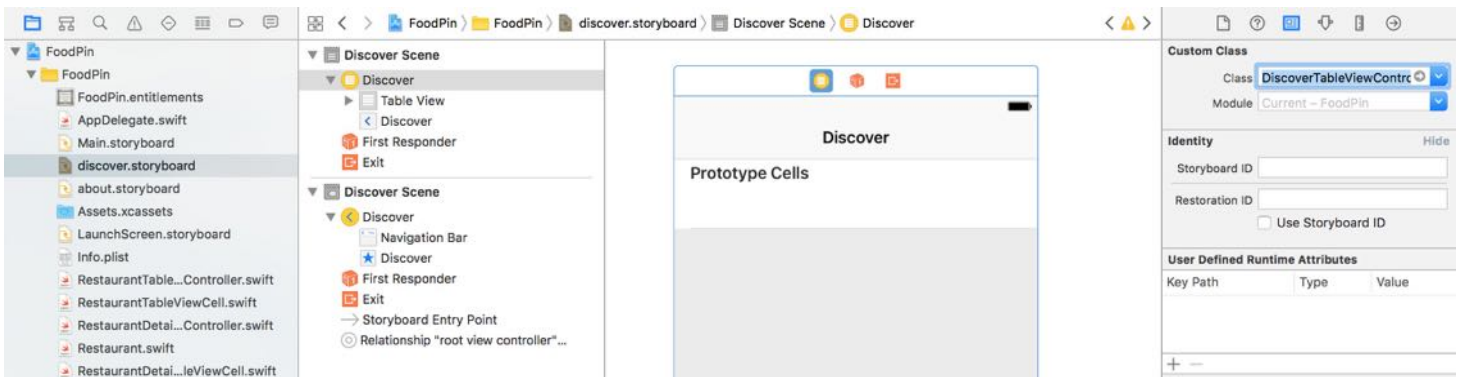


Figure 24-9. Set the custom class for the table view controller

Now you are ready to implement `DiscoverTableViewController` to fetch records from the cloud. To use CloudKit, you have to first import the CloudKit framework. Insert the following line of code at the very beginning of `DiscoverTableViewController.swift` :

```
import CloudKit
```

Next, declare a `restaurants` variable that stores an array of `CKRecord` objects. Initially the array is empty. Later we use it to store the records fetched from the cloud.

```
var restaurants:[CKRecord] = []
```

When the *Discover* tab is loaded, the app will start to fetch the records via CloudKit. In the `viewDidLoad` method, insert the following line of code:

```
fetchRecordsFromCloud()
```

Next, insert the `fetchRecordsFromCloud` method in the `DiscoverTableViewController` class:

```
func fetchRecordsFromCloud() {
    // Fetch data using Convenience API
    let cloudContainer = CKContainer.default()
    let publicDatabase = cloudContainer.publicCloudDatabase
    let predicate = NSPredicate(value: true)
    let query = CKQuery(recordType: "Restaurant", predicate: predicate)
    publicDatabase.perform(query, inZoneWith: nil, completionHandler: {
        (results, error) -> Void in

        if error != nil {
```



```

        print(error)
        return
    }

    if let results = results {
        print("Completed the download of Restaurant data")
        self.restaurants = results
        self.tableView.reloadData()
    }
})
}

```

The block of code is nearly the same as the one we discussed before. In the complete handler, we check if there is any errors. If there is no error, we then get the `results` array and save it to the `restaurants` array. Finally, we reload the table view with the updated restaurants.

Now that you've obtained the records, the next step is to implement the required methods of the `UITableViewDataSource` protocol to display the record in the table view. Add/update the following methods in the class:

```

override func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}

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

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

    // Configure the cell...
    let restaurant = restaurants[indexPath.row]
    cell.textLabel?.text = restaurant.object(forKey: "name") as? String

    if let image = restaurant.object(forKey: "image") {
        let imageAsset = image as! CKAsset

        if let imageData = try? Data.init(contentsOf: imageAsset.fileURL) {
            cell.imageView?.image = UIImage(data: imageData)
        }
    }
}

```

```
    return cell
}
```

These methods should be very familiar to you. Let's focus on the last method. Note that the `restaurants` variable contains an array of `CKRecord` objects. As mentioned before, a `CKRecord` object is a dictionary of key-value pairs that you use to fetch and save the data of your app. It provides the `object(forKey:)` method to retrieve the value of a record field (e.g. name field of the Restaurant). When creating the `Restaurant` record type in the CloudKit dashboard, we use `Asset` as the type of the image field. Thus, the object returned is a `CKAsset` object.

When downloading a record containing an asset, CloudKit saves the asset data in the local file system and you can retrieve the asset data with the URL in the `fileURL` property. So we can load the image by accessing the `fileURL` property of the image asset.

```
if let imageData = try? Data.init(contentsOf: imageAsset.fileURL) {
    cell.imageView?.image = UIImage(data: imageData)
}
```

We discussed how to load an image with binary data. But what is the `try?` keyword? When you initialize a `Data` with the file URL, `Data` may fail to load the file and throws you an error. You can create a do-catch statement to handle the error. But in the above code we use `try?` instead. In this case, if an error is thrown, the method returns an optional without a value. If the image file is successfully loaded, the data is bind to the `imageData` variable. `try?` is particularly useful if you do not care about the error message.

That's it. Before testing the app in the simulator, you have to configure the iCloud setting. Otherwise, you can't fetch the data from iCloud. In the Simulator, click command-shift-H to go back to Home screen. Select Setting > iCloud and sign in with your Apple ID. Now you're ready to run the app. Select the *Discover* tab and the app should be able to fetch the restaurants through CloudKit. It may take over 10 seconds before the restaurants are displayed. I will explain why it takes so long to download the records in the next section. Meanwhile, just be patient. Figure 24-10 shows a sample screenshot of the *Discover* tab.

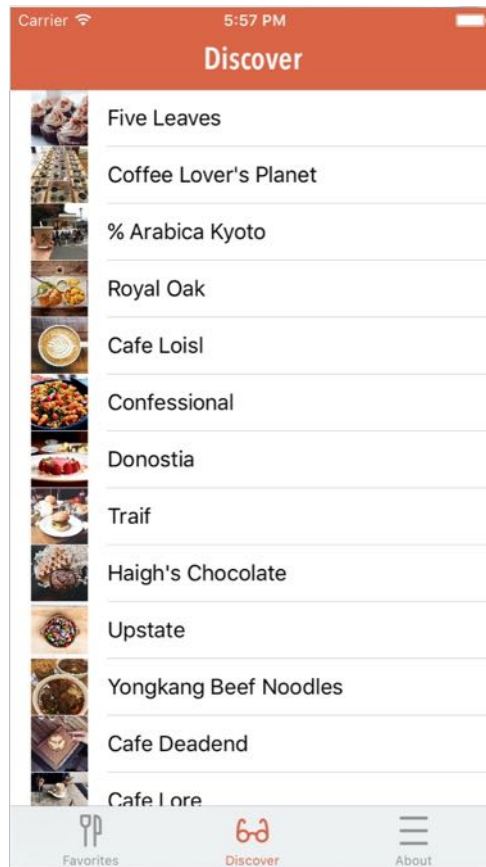


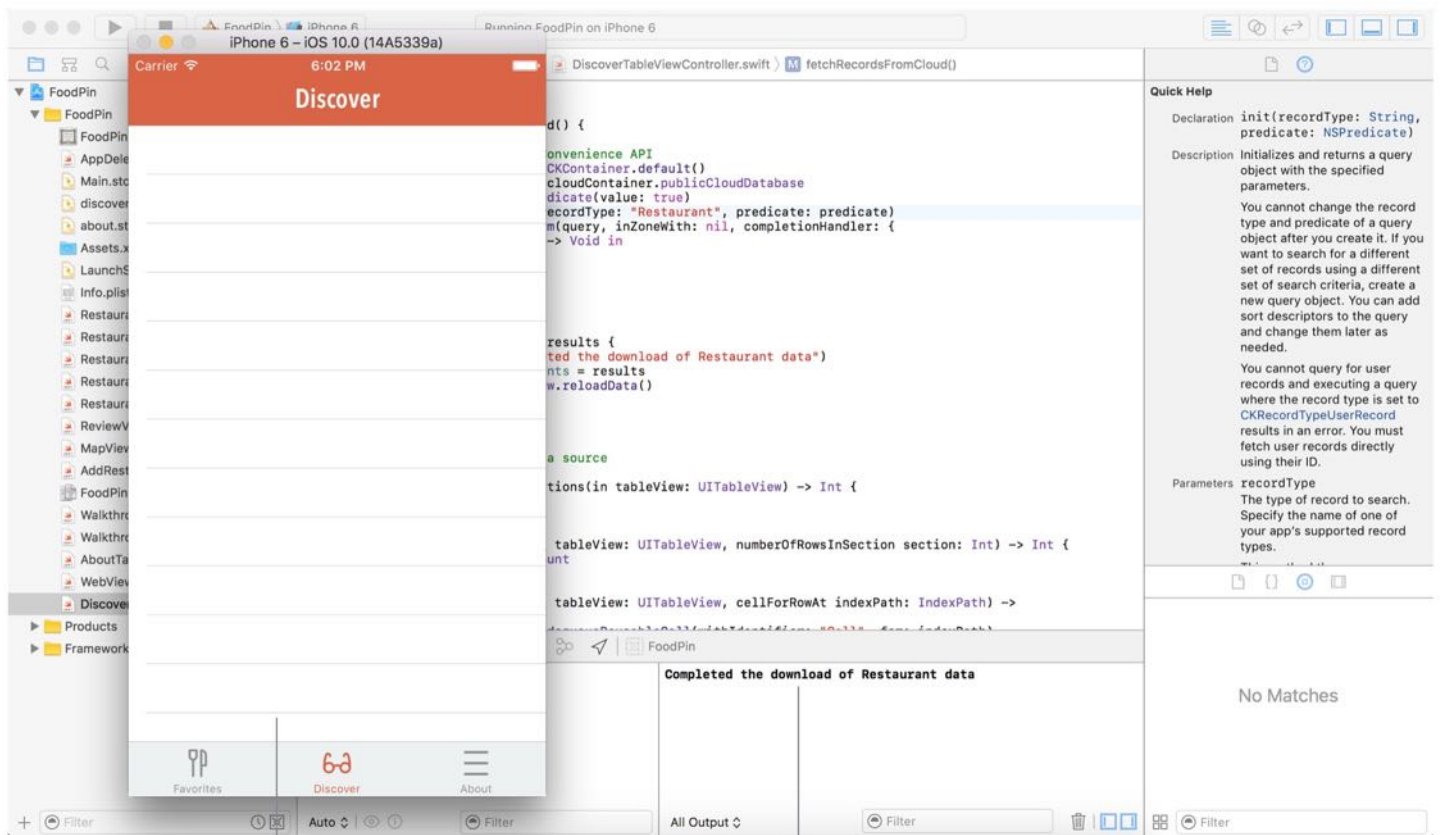
Figure 24-10. Set the custom class for the table view controller

Note: Occasionally you may end up with an error in the console. In this case, click the Stop button to quit the app and launch it again.

Why So Slow?

Why did it take so long to load the restaurants from iCloud? Was it a network issue? Or was it due to the size of images?

Let's look into the issue together. If you look into the code block of the `fetchRecordsFromCloud` method, we print a message `Completed the download of Restaurant data` to the console when the query completes. Try to run your app again and take a closer look at the console.



2 But the table is still blank. 1 The console prints the status message.

Figure 24-11. Console message after the query completes

It only takes a couple of seconds and the message appears in the console. In other words, it doesn't take long to fetch the restaurant records. Thus, the root cause of the slow response is not related to network performance. But why does it take 10-20 seconds before the records show up? It looks like there is something wrong with the display of table cells. Let's look at the `perform(_:inZoneWith:completionHandler:)` method call again.

```
publicDatabase.perform(query, inZoneWith: nil, completionHandler: {
    (results, error) -> Void in

    if error != nil {
        print(error)
        return
    }

    if let results = results {
        print("Completed the download of Restaurant data")
    }
})
```

```
        self.restaurants = results
        self.tableView.reloadData()
    }
})
```

Once we get the restaurants from iCloud, we call the `reloadData` method to reload the table view. For some reasons, it seems the reload is not performed immediately.

As mentioned before, all queries in CloudKit are asynchronous. CloudKit automatically executes the query in background. In fact, the completion handler is executed in a background thread as well.

If you're new to programming, you may not be familiar with the term *thread*. However, you should be very familiar with this situation:

- You need to do your homework,
- but you also want to watch TV,
- and check your Facebook, Twitter, etc.

You're doing multiple tasks at the same time. You spend a little bit of time on your homework and then a little bit of time on TV, and then give some attention to your Facebook feed.

In computing, the same situation happens all the time. An app always needs to handle multiple tasks in parallel. Say, it needs to update the user interface while waiting for user input or downloading file from the Internet. For each of these tasks, the app spawns a thread to handle it. iOS allocates a little time on each thread and switches between them. The time allocated for each thread varies depending on its priority. The one with higher priority usually executes first and iOS may spend more time on it.

Now that you've got some background knowledge of thread, let's go back to our code. CloudKit creates a background thread for the search query and we also reload the table view in this background thread.

In iOS, UI updates like table reload should be executed in the main thread. The main thread is given with a higher priority to ensure a responsive UI. If we execute a UI-related update in a background thread, the UI change will not be executed immediately; this is why it takes so long before the downloaded records appear on screen.

So how can you fix the issue and improve the performance?

It's a simple fix. You just need to put the table reload in the main thread. In Swift, you can use `OperationQueue.main` to get the operation queue of the main thread, and then add your task in the code block like below. The code block will be executed in the main thread.

```
OperationQueue.main.addOperation {  
    self.tableView.reloadData()  
}
```

Change the code accordingly, and test the app again. The app now shows the restaurant records as soon as the data download completes. For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinCloudKit-1.zip>. Make sure you change the bundle identifier to your own ID before compiling the project.

Fetching Data from Public Database Using Operational API

The convenience API is good for simple query. However, it does have some limitations. The `perform(_:inZoneWith:completionHandler:)` method is only suitable for retrieving a small amount of data. If you have a few hundred records (or even more), the convenience API may not fit. You also can't do additional tweaking with the convenience API. When you call the `perform` method, it retrieves all restaurant records. For each record, it downloads the whole records including the image and every other fields. However, as you may notice, the app just displays the restaurant names and the images. We can actually leave out those fields like phone number, type and location to save some bandwidth.

So how can you tell CloudKit to merely retrieve the name field of all restaurant records? You can't do that via the convenience API. In this case, you'll need to explore the operational API.

The usage of operational API is similar to that of the convenience API but it offers more flexibility. Let's jump right into the code. Replace the `fetchRecordsFromCloud` method with the following code snippet:

```
func fetchRecordsFromCloud() {  
  
    // Fetch data using Convenience API
```

```

let cloudContainer = CKContainer.default()
let publicDatabase = cloudContainer.publicCloudDatabase
let predicate = NSPredicate(value: true)
let query = CKQuery(recordType: "Restaurant", predicate: predicate)

// Create the query operation with the query
let queryOperation = CKQueryOperation(query: query)
queryOperation.desiredKeys = ["name", "image"]
queryOperation.queuePriority = .veryHigh
queryOperation.resultsLimit = 50
queryOperation.recordFetchedBlock = { (record) -> Void in
    self.restaurants.append(record)
}

queryOperation.queryCompletionBlock = { (cursor, error) -> Void in
    if let error = error {
        print("Failed to get data from iCloud - \
(error.localizedDescription)")
        return
    }

    print("Successfully retrieve the data from iCloud")
    OperationQueue.main.addOperation {
        self.tableView.reloadData()
    }
}

// Execute the query
publicDatabase.add(queryOperation)
}

```

The first few lines of code are exactly as before. We get the default container and the public database, followed by creating the query for retrieving the restaurant records.

Instead of calling the `perform` method to fetch the records, we create a `CKQueryOperation` object for the query. This is why Apple called it *operational API*. The query operation object provides several options for your configuration. The `desiredKeys` property lets you specify the fields to fetch. You use this property to retrieve only those fields that you need for the app. In the above code, we tell the query operation object that we only need the `name` and `image` fields of the records.

Other than the `desiredKeys` property, you can use the `queuePriority` property to specify the execution priority of the operation and `resultsLimit` property to set the maximum number of

records at any one time.

The operation object will be executed in the background. It reports the status of the query operation through two callbacks. One is `recordFetchedBlock` and the other is `queryCompletionBlock`. The block of code within `recordFetchedBlock` will be executed every time a record returned. In the code snippet, we simply append each of the returned records to the `restaurants` array.

On the other hand, `queryCompletionBlock` allows you to specify the code block that executes after all records are fetched. In this case, we ask the table view to reload and display the restaurant records.

More about `queryCompletionBlock`

The `queryCompleteBlock` provides a cursor object to indicate if there are more results to fetch. Recalled that we use the `resultsLimit` property to control the number of the fetched records, the app may not be able to fetch all data in a single query. In this case, a `CKQueryCursor` object indicates there are more results to fetch. Additionally, it marks the stopping point of the query and the starting point for retrieving the remaining results. For example, let's say you have a total of 100 restaurant records. For each search query, you can get a maximum of 50 records. After the first query, the cursor would indicate that you have fetched record 1 to 50. For your next query, you should start from the 51st record. The cursor is very useful if you need to get your data in multiple batches.

Lastly, we call the `add` method of the `CKDatabase` class to execute the query operation. You can now compile and run the app again. The result should be the same as before. Internally, however, you have built a custom query to fetch only those data you need.

Optimizing the Performance

Up until now, both approaches achieve the same performance. It is not too slow as the size of data set is relatively small. However, as your data set grows, it will take even longer to download and display the records to users. Performance is critical in mobile app development. Your users expect your app to be fast and responsive. A sluggish performance of an app may mean losing a customer. So how can we improve the performance?

Optimizing the server performance? No, that's out of our control. We can only focus on the things that we can optimize.

Reducing the size of the images? This is a good suggestion. You can further optimize the images to achieve a faster download. But from a user point of view, the improvement would not be very significant as the sizes of original images were moderately optimized.

Tip: tinypng.com is a great website for image optimization.

When we talk about performance optimization, sometimes we're not talking about optimizing the real performance but rather perceived performance. Perceived performance refers to how fast your user thinks your app is. Let me give you an example. Say, after a user taps the *Discover* tab, it takes 10 seconds to load the restaurant records. You then optimize the image sizes and reduce the loading time to 6 seconds. The real performance is improved by 40%. You think that's a huge improvement. But the perceived performance is still sluggish. To the user, your app is slow because it can't respond instantaneously. When it comes to performance optimization, sometimes the technical statistics do not matter. Rather, it's about optimizing the perceived performance to make users feel that your app is speedy.

Activity Indicator

One way to improve the perceived performance is to make your app responsive. Currently, when a user selects the *Discover* tab, nothing shows up. The user expects an instantaneous response. This doesn't mean you have to show the remote content immediately. However, your app should at least display something even it's waiting for the data. Adding an animated spinner or an activity indicator will do the trick. The spinner gives a real-time feedback and tells the user that something is happening. Research has also shown that the feeling of waiting can be reduced by keeping a user's attention occupied.



The UIKit framework provides a class called `UIActivityIndicatorView` for showing a spinner to indicate a task is in progress. This is a sample code snippet to create an activity indicator:

```
let spinner:UIActivityIndicatorView = UIActivityIndicatorView()  
spinner.activityIndicatorViewStyle = .gray  
spinner.center = view.center  
spinner.hidesWhenStopped = true  
view.addSubview(spinner)  
spinner.startAnimating()
```

You can put the above code in the `viewDidLoad` method to create the activity indicator. But let's use storyboards to add the indicator. At the same time, I can demonstrate you a new feature of Storyboards in Xcode known as *Extra Views*.

Go to `discover.storyboard` and drag an Activity Indicator View object to the scene dock of the table view controller.

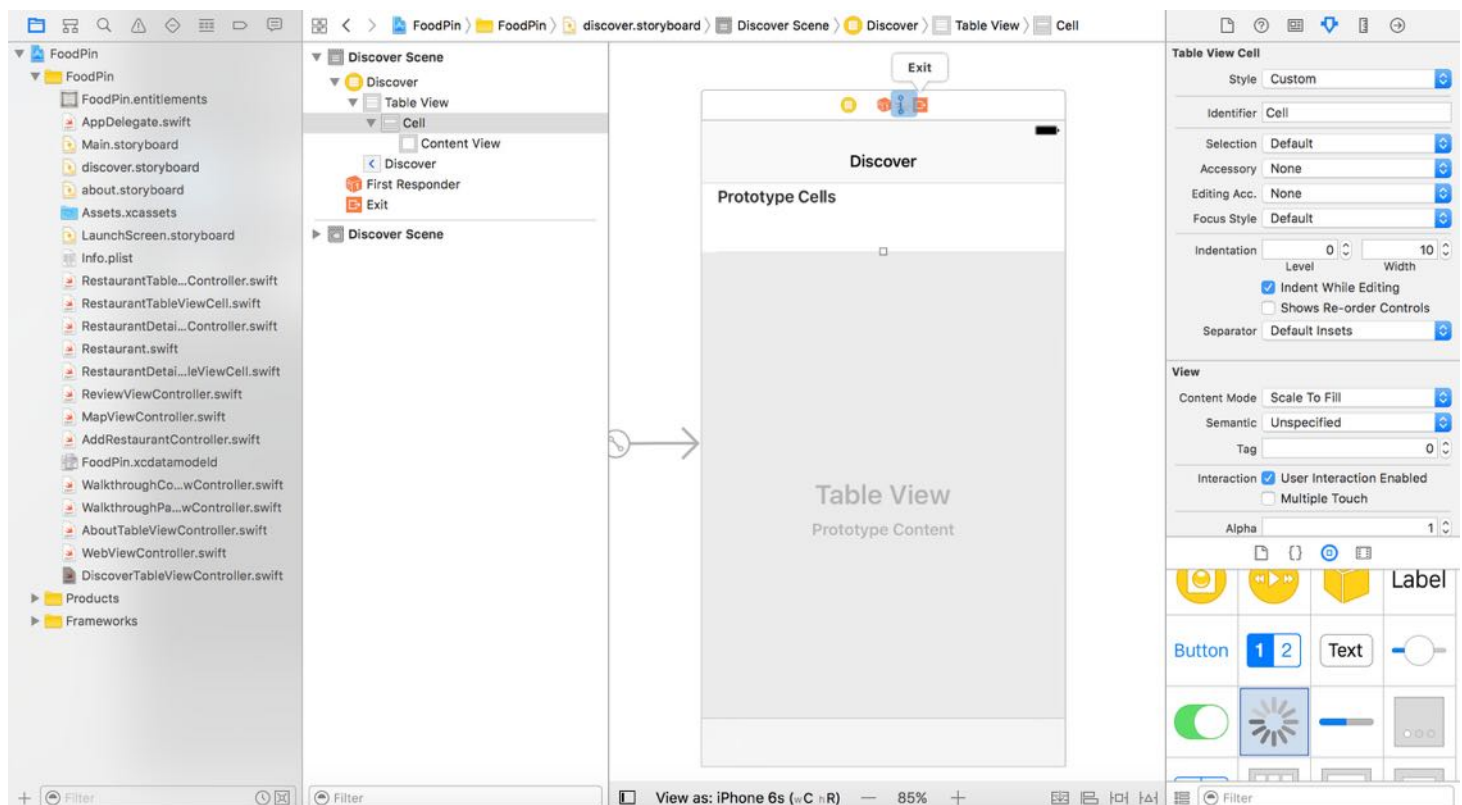


Figure 24-13. Adding an activity indicator view in the storyboard

Once you added the view, it appears as an extra view in the scene dock, and it shows up in its

own editor above the view controller. The *Extra Views* feature allows you to put a view alongside a view controller if you don't want that view to initially be a part of the view hierarchy.

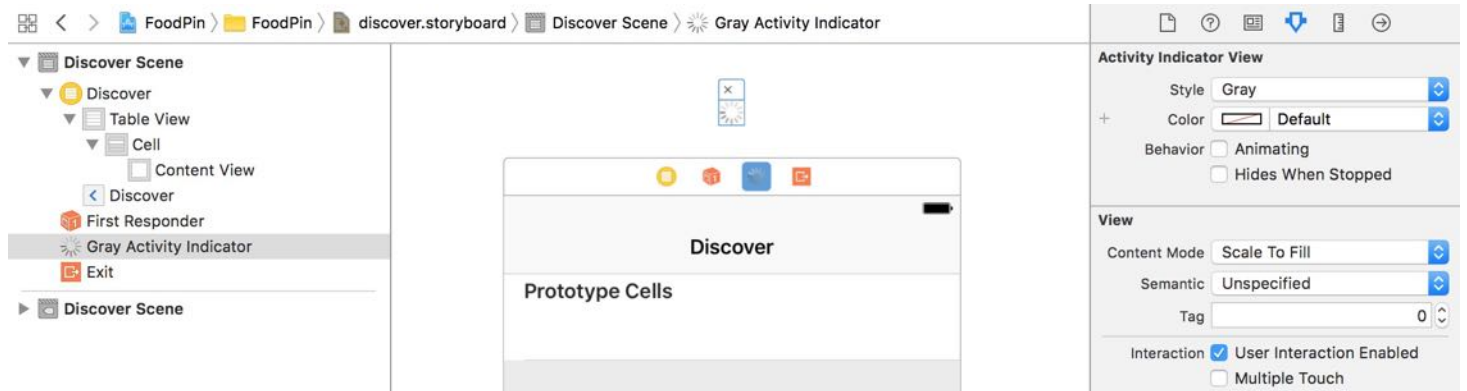


Figure 24-14. The activity indicator view has its own editor

Now add an outlet variable in the `DiscoverTableViewController` class:

```
@IBOutlet var spinner: UIActivityIndicatorView!
```

Go back to `discover.storyboard`. You can establish a connection between the outlet and the activity indicator view like before. To display the indicator, all you need to do is insert the following code in the `viewDidLoad` method of the `DiscoverTableViewController` class:

```
spinner.hidesWhenStopped = true  
spinner.center = view.center  
tableView.addSubview(spinner)  
spinner.startAnimating()
```

When the spinner is first initialized, its position is set to the top-left of a view. Here we use the `center` property to place the indicator at the center of the view. The `hidesWhenStopped` property controls whether the indication is hidden when the animation is stopped. Once the indicator is configured, you can add it to the current view and call the `startAnimating` method to start the animation.

That's it! If you run the app, the activity indicator will appear when you select the *Discovery* tab. However, it doesn't hide itself even when the restaurant records are fully loaded.

The indicator has no idea when it should be hidden. When the data download completes, you have to explicitly call the `stopAnimating` method to stop the animation, right before reloading the table view. Because the `hidesWhenStopped` property is set to true, the activity indicator will then be hidden automatically.

```
OperationQueue.main.addOperation {  
    self.spinner.stopAnimating()  
    self.tableView.reloadData()  
}
```

Lazy Loading Images

Cool! The spinner made the app more responsive. Are there any other ways to optimize the performance? Let's take a look at how we fetch the data from iCloud. Currently the app displays the records on screen only when all restaurant records are downloaded completely. This includes the image download. Obviously this download operation is one of the bottlenecks. It takes time to download all images. Wouldn't it be great if we can retrieve the restaurant names and display them in the table view first? The size of the restaurant names is significantly smaller than the size of the images. It will take a fraction of the total time to download the data.

This sounds great! But what about the images? We will employ a well-known technique called *lazy loading*. Simply put, we defer the download of the images. At first, we just display a local image, bundled in the app, in the table view cell. Then we start another thread in background to download the remote images. When the image is ready, the app updates the cell's image view. Figure 24-15 illustrates the lazy loading technique.



Figure 24-15. The lazy loading technique illustrated

With lazy loading in place, your user will be able to view the restaurant data almost instantly. Though the images are not ready when the data is first loaded, it improves the responsiveness of your app.

Let's see how we can implement it in the app. Because we defer the image download, we have to update the `desiredKeys` property of the query operation in the `fetchRecordsFromCloud` method like this.

From:

```
queryOperation.desiredKeys = ["name", "image"]
```

To:

```
queryOperation.desiredKeys = ["name"]
```

Now we only retrieve the restaurant names and leave out the images. In the `tableView(_:cellForRowAt:)` method, we modify two things to implement lazy loading:

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

```

indexPath)

// Configure the cell...
let restaurant = restaurants[indexPath.row]
cell.textLabel?.text = restaurant.object(forKey: "name") as? String

// Set the default image
cell.imageView?.image = UIImage(named: "photoalbum")

// Fetch Image from Cloud in background
let publicDatabase = CKContainer.default().publicCloudDatabase
let fetchRecordsImageOperation = CKFetchRecordsOperation(recordIDs:
[indexPath.restaurant.recordID])
fetchRecordsImageOperation.desiredKeys = ["image"]
fetchRecordsImageOperation.queuePriority = .veryHigh

fetchRecordsImageOperation.perRecordCompletionBlock = { (record, recordID,
error) -> Void in
    if let error = error {
        print("Failed to get restaurant image: \
(error.localizedDescription)")
        return
    }

    if let restaurantRecord = record {
        OperationQueue.main.addOperation() {
            if let image = restaurantRecord.object(forKey: "image") {
                let imageAsset = image as! CKAsset

                if let imageData = try? Data.init(contentsOf:
imageAsset.fileURL) {
                    cell.imageView?.image = UIImage(data: imageData)
                }
            }
        }
    }
}

publicDatabase.add(fetchRecordsImageOperation)

return cell
}

```

First, we set a default image (photoalbum.png) for each cell. As this image is bundled in the app, it can be displayed instantly when the *Discover* tab is loaded. Secondly, we spawn a background thread to download the image asynchronously. Once the image is retrieved, the default image will be replaced by the image just downloaded.

The code is very similar to that of the `fetchRecordsFromCloud` method except that we use `CKFetchRecordsOperation` to fetch a specific record. Every record on the cloud has its own ID. To fetch the image of a specific restaurant record, we create a `CKFetchRecordsOperation` object with the ID of that particular restaurant record. Similar to `CKQueryOperation`, you can assign a code block (`perRecordCompletionBlock`) to execute when the record is available. In the code block, we just load the downloaded image in the cell's image view.

After the modification, you can test the app again. You'll see a great improvement in performance as the restaurant records should appear with a little (or even no) delay. The thumbnails of the restaurants will get loaded in background. Figure 24-16 shows a sample screenshot.

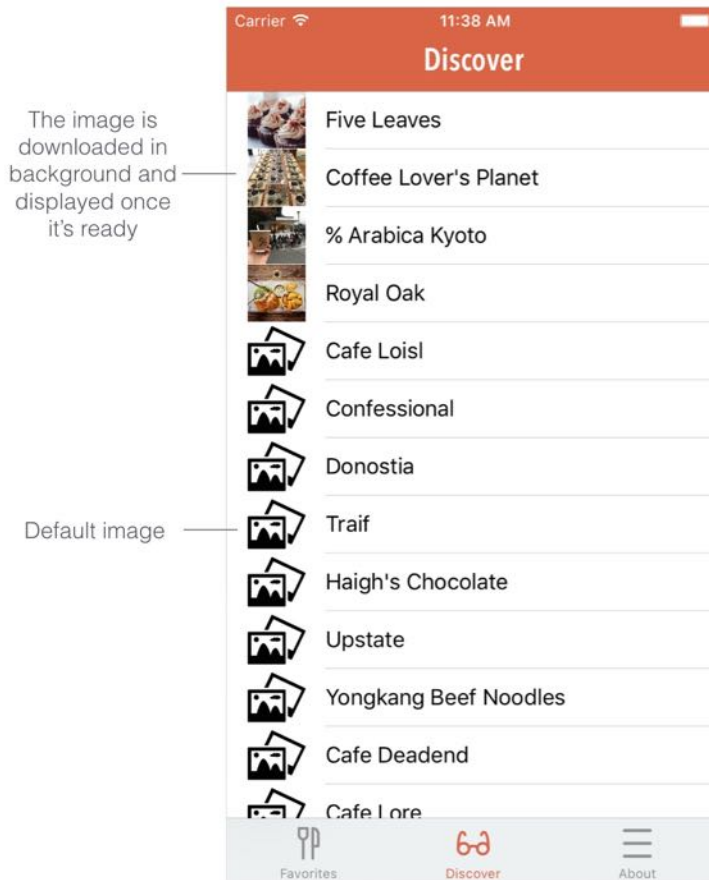


Figure 24-16. Lazy loading image in action

Caching Images Using NSCache

Every time a table view cell goes off the screen (as you scroll up/down the table), the cell is removed from view. Whenever the same cell comes back on screen, it goes up to the cloud and re-downloads the restaurant image. This is inefficient and wastes Wi-Fi/cellular data on duplicated downloads. Why don't we cache the image for later reuse?

Caching is one of the common approaches to improve app performance. iOS SDK provides an `NSCache` class for developers to implement simple caching. `NSCache` behaves like a dictionary and caches data in key-value pairs. You use `setObject(_:forKey:)` method to add an object to the cache using a specific key. To obtain an object of a given key from the cache, you use `object(forKey:)` method. That's much the same as accessing items in a dictionary.

However, unlike a dictionary, it incorporates various auto-removal policies to clean up the cached objects. This ensures that it does not use too much of the system's memory.

So what do we need to do to cache the restaurant objects?

As mentioned earlier, CloudKit automatically downloads the images and saves them in the local file system. The images are then temporarily available for offline reading. You use the `fileURL` property of the `CKAsset` class to retrieve the image's location. Thus, what we need to cache are the file URLs of the images.

To implement caching in the *Discover* tab, we'll make the following modifications:

- If the image is downloaded for the first time, we cache its file URL using `NSCache` with the record ID as key.
- When a table view cell loads an image, we first check if the image's file URL is stored in the cache. If yes, we get the URL from the cache and load it directly. Otherwise, we'll retrieve it from iCloud.

To create a cache object using `NSCache`, insert the following line of code in the `DiscoverTableViewController` class:

```
var imageCache = NSCache<CKRecordID, NSURL>()
```

In Swift 3, `NSCache` is a generic. When initializing it, you have to provide the type of the key and value pair in angle bracket. In this case, the key is of type `CKRecordID` and the value is of

type `NSURL` .

In order to handle the image caching, we need to add a conditional block to check if the image's URL is available in the cache in the `tableView(_:cellForRowAt:)` method. Update the method like this:

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

    // Configure the cell...
    let restaurant = restaurants[indexPath.row]
    cell.textLabel?.text = restaurant.object(forKey: "name") as? String

    // Set the default image
    cell.imageView?.image = UIImage(named: "photoalbum")

    // Check if the image is stored in cache
    if let imageURL = imageCache.object(forKey: restaurant.recordID) {
        // Fetch image from cache
        print("Get image from cache")
        if let imageData = try? Data.init(contentsOf: imageURL as URL) {
            cell.imageView?.image = UIImage(data: imageData)
        }
    }
} else {
    // Fetch Image from Cloud in background
    let publicDatabase = CKContainer.default().publicCloudDatabase
    let fetchRecordsImageOperation = CKFetchRecordsOperation(recordIDs:
[restaurant.recordID])
    fetchRecordsImageOperation.desiredKeys = ["image"]
    fetchRecordsImageOperation.queuePriority = .veryHigh

    fetchRecordsImageOperation.perRecordCompletionBlock = { (record,
recordID, error) -> Void in
        if let error = error {
            print("Failed to get restaurant image: \
(error.localizedDescription)")
            return
        }

        if let restaurantRecord = record {
            OperationQueue.main.addOperation() {
                if let image = restaurantRecord.object(forKey: "image") {
                    let imageAsset = image as! CKAsset
                }
            }
        }
    }
}
```

```

        if let imageData = try? Data.init(contentsOf:
imageAsset.fileURL) {
            cell.imageView?.image = UIImage(data: imageData)
        }

        // Add the image URL to cache
        self.imageCache.setObject(imageAsset.fileURL as NSURL,
forKey: restaurant.recordID)
    }
}
}

publicDatabase.add(fetchRecordsImageOperation)
}

return cell
}

```

Basically we have made a couple of changes:

1. If the URL can be found in the cache, we simply get the URL and load it right away.
2. For images that are downloaded for the first time, we need to cache the image's URL. Therefore, we add a line of code to call `setObject` of the image cache. We use the record ID of the restaurant record as key and add the URL to the cache.

That's it. If you run the app and scroll through the records in the *Discover* tab, the images should be loaded from the cache if it is available. You will see the message "Get image from cache" in the console.

Pull to Refresh

After all the tweaks, the *Discover* tab should be working much better. However, there is a limitation. Once the restaurant records are loaded, there is no way to get an update.

Most modern iOS apps allow users to refresh their content through a feature called *pull-to-refresh*. The pull-to-refresh interaction for updating data of a table view was originally created by Loren Brichter. Since its invention, an endless number of apps, including Apple's Mail app, have adopted the design for content updates.

Prior to iOS 6, adding such a feature to your app was not an easy task. You either implemented

your own solution or relied on third-party libraries. Both solutions took you considerable time and effort.

Thanks to the popularity of the pull-to-refresh feature. Apple have made a standard pull-to-refresh control in the iOS SDK. With the built-in control, it is very simple to add the pull-to-refresh feature to your app.

`UIRefreshControl` is a standard control for implementing the pull-to-refresh feature. You can simply associate a refresh control with a table view controller, which will automatically add the control to the table view. Let's see how it works.

In the `DiscoverTableViewController` class, insert the following lines of code in the `viewDidLoad` method:

```
// Pull To Refresh Control
refreshControl = UIRefreshControl()
refreshControl?.backgroundColor = UIColor.white
refreshControl?.tintColor = UIColor.gray
refreshControl?.addTarget(self, action: #selector(fetchRecordsFromCloud), for:
UIControlEvents.valueChanged)
```

`UIRefreshControl` is very simple to use. You can instantiate it and assign it to the table view controller's `refreshControl` property. That's it. The table view controller handles the task of adding the control to the table's visual appearance. Similar to other view objects, you can configure the background color and tint color using the `backgroundColor` and `tintColor` properties.

The refresh control doesn't initiate the refresh operation when it is first created. Instead, when the table view is pulled down sufficiently, the refresh control triggers the

`UIControlEvent.valueChanged` event. We have to assign an action method to this event and use it to fetch the restaurant records. To capture the event and specify the follow-up action, you can use the `addTarget` method of the refresh control object to register the

`UIControlEvent.valueChanged` event. In the method call, you specify the target object (i.e. `self`) and the action method (i.e. `fetchRecordsFromCloud`) to handle the event. When the event is triggered, the refresh control will invoke the `fetchRecordsFromCloud` method to refresh the restaurant records.

Note: Starting from Swift 2.2 (or Xcode 7.3), Apple introduces a new expression

```
`#selector` that allows developers to build a selector from a reference to a method.
```

If you compile and run the app, the pull-to-refresh feature should work. Try to pull down the table until it triggers the refresh. If you've added new restaurants using CloudKit dashboard, the table should show the update. However, there's still a problem; the refresh control appears even after the table content is loaded.

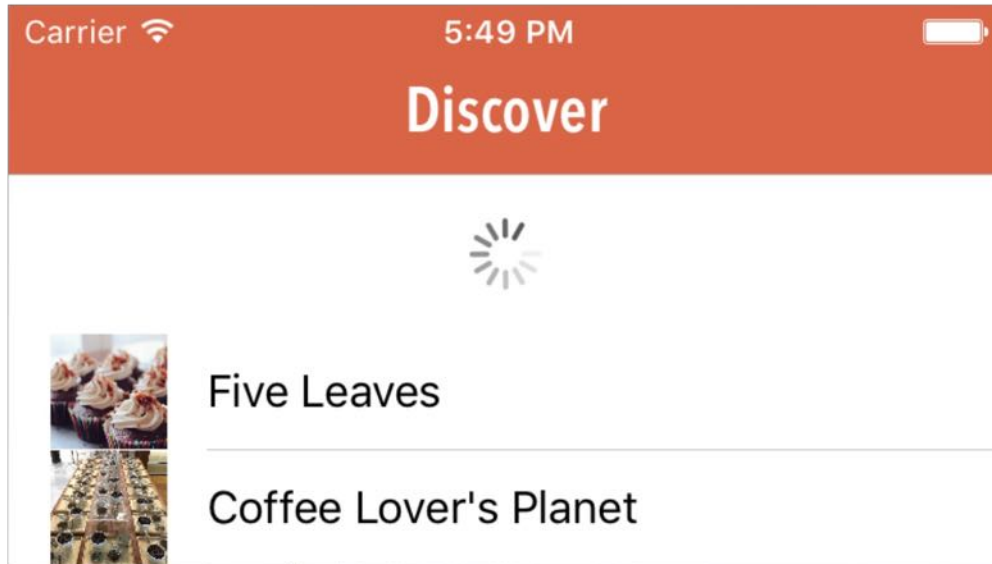


Figure 24-17. Pull to Refresh

So how can we hide the refresh control? When the data is ready, you can simply call the `endRefreshing` method to hide the control. Apparently, we can update `queryCompletionBlock` in the `fetchRecordsFromCloud` method by adding adding a few lines of code:

```
if let refreshControl = self.refreshControl {  
    if refreshControl.isRefreshing {  
        refreshControl.endRefreshing()  
    }  
}
```

You can place the code after `self.tableView.reloadData()`. Here we check if the refresh control is still refreshing. If yes, we call `endRefreshing()` to end the animation and hide the control.

Now run the app again. The pull-to-refresh control should disappear when the data transfer

completes.

Currently, there is a bug when you refresh the data from cloud. Some of the records are duplicated because the existing records are not removed before loading the new ones. To fix it, insert the following lines of code at the beginning of the `fetchRecordsFromCloud` method:

```
// Remove existing records before refreshing
restaurants.removeAll()
tableView.reloadData()
```

We're now going to discuss a new topic but make sure you understand the material covered so far. For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinCloudKit-2.zip>.

Saving Data Using CloudKit

Now that we have discussed data query, let's further explore the CloudKit framework and see how you can save data to the cloud. It all comes down to this convenience API provided by the `CKDatabase` class:

```
func save(_ record: CKRecord, completionHandler: @escaping (CKRecord?, Error?)
-> Void)
```

The `save(_:completionHandler:)` method takes in a `CKRecord` object and uploads it to iCloud. When the operation completes, it reports the status by calling the completion handler. You can examine the error and see if the record is saved successfully.

To demonstrate the usage of the API, we'll tweak the Add Restaurant function of the FoodPin app. When a user adds a new restaurant, in addition to saving it to the local database, the record will also be uploaded to iCloud.

I'll go straight into the code and walk you through the logic along the way. In the `AddTableViewController` class, import CloudKit:

```
import CloudKit
```

and add the following method:


```

func saveRecordToCloud(restaurant:RestaurantMO!) -> Void {

    // Prepare the record to save
    let record = CKRecord(recordType: "Restaurant")
    record.setValue(restaurant.name, forKey: "name")
    record.setValue(restaurant.type, forKey: "type")
    record.setValue(restaurant.location, forKey: "location")
    record.setValue(restaurant.phone, forKey: "phone")

    let imageData = restaurant.image as! Data

    // Resize the image
    let originalImage = UIImage(data: imageData)!
    let scalingFactor = (originalImage.size.width > 1024) ? 1024 /
originalImage.size.width : 1.0
    let scaledImage = UIImage(data: imageData, scale: scalingFactor)!

    // Write the image to local file for temporary use
    let imagePath = NSTemporaryDirectory() + restaurant.name!
    let imageFileURL = URL(fileURLWithPath: imagePath)
    try? UIImageJPEGRepresentation(scaledImage, 0.8)?.write(to: imageFileURL)

    // Create image asset for upload
    let imageAsset = CKAsset(fileURL: imageFileURL)
    record.setValue(imageAsset, forKey: "image")

    // Get the Public iCloud Database
    let publicDatabase = CKContainer.default().publicCloudDatabase

    // Save the record to iCloud
    publicDatabase.save(record, completionHandler: { (record, error) -> Void
in
        // Remove temp file
        try? FileManager.default.removeItem(at: imageFileURL)
    })
}

```

The `saveRecordToCloud` method takes in a `Restaurant` object, which is the object to save. We first transform the `Restaurant` object to a `CKRecord` object by instantiating a `CKRecord` of the `Restaurant` record type and setting its name, type and location values.

The restaurant image requires a bit of work. First, we don't want to upload a super-high resolution photo. We would like to scale it down before uploading. The `UIImage` class allows us to create an object with a certain scaling factor. In this case, any photo with width larger than 1024 pixels will be resized.

As you know, you use `CKAsset` object to represent an image on the cloud. To create the `CKAsset` object, we have to provide the file URL of the scaled image. So we save the image in the temporary folder. You can use the `NSTemporaryDirectory` function to get the path of the temporary directory. By combining the path with the restaurant name, we have the temporary file path of the image. We then use `UIImageJPEGRepresentation` function to compress the image data and call the `write` method to save the compressed image data as a file.

With the scaled image ready for upload, we can create the `CKAsset` object using the file URL. Lastly, we get the default public database and save the record to the cloud using the `save` method of `CKDatabase`. In the complete handler, we clean up the temporary file just created.

The `saveRecordToCloud` method is now ready. As the app uploads the restaurant record whenever a user saves a new restaurant, we will call the method in the `save` method of the `AddTableViewCellController` class. Simply insert the following line of code in the `save` method and place it right before `dismiss(animated:completion:)`:

```
saveRecordToCloud(restaurant)
```

You're ready to go! Hit the Run button and test the app. Click the + button to add a new restaurant. Once you save the restaurant, go to the *Discover* tab and you should find the new restaurant there. If it doesn't appear, wait for a few seconds and pull-to-refresh the table again. Alternatively, you can go up to CloudKit Dashboard to reveal the new record.

Sorting the Result by Creation Date

One problem of the *Discover* feature is that the restaurants are not in any order. As a user, you may want to view the new restaurants shared by other app users. That means, we need to arrange the results in reverse chronological order.

Sorting has been built into the `CKQuery` class. Here is the code snippet of the `fetchRecordsFromCloud` method in `DiscoverTableViewCellController`:

```
// Prepare the query
let predicate = NSPredicate(value: true)
let query = CKQuery(recordType: "Restaurant", predicate: predicate)
```

The `CKQuery` class provides a property named `sortDescriptor`. Earlier, we didn't set the sort

descriptor. In this case, CloudKit simply returns an unordered collection of restaurant records. To request CloudKit to sort the records in reverse chronological order, you need to insert a line of code:

```
query.sortDescriptors = [NSSortDescriptor(key: "creationDate", ascending: false)]
```

This creates an `NSSortDescriptor` object using the `creationDate` key and set the order to descending. When CloudKit performs the search query, it will order the results by creation date. You can run now the app again and add a new restaurant. Once saved, go to the *Discover* tab and the restaurant just added should appear first.

Your Exercise

Presently, the cell in the *Discover* tab only displays the name and thumbnail of a restaurant. Modify the project so that the cell shows the restaurant's location and type. Figure 24-18 displays a sample screenshot.

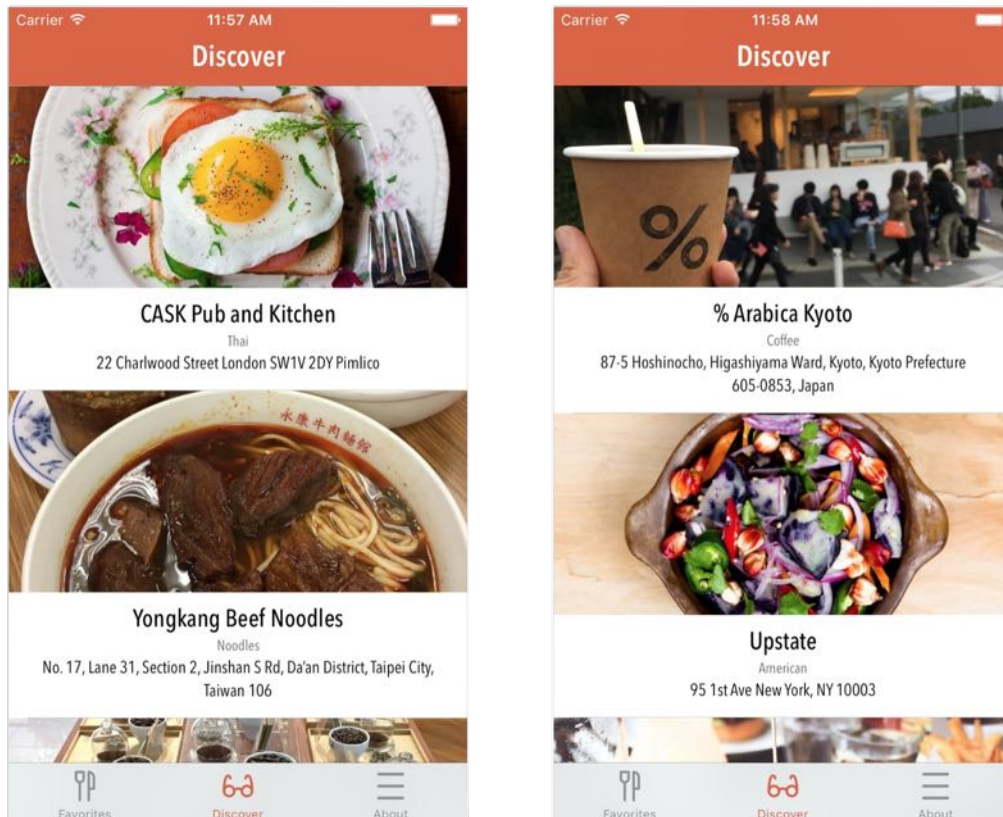


Figure 24-18. A modified Discover tab

Note: This exercise is designed to refresh your knowledge of table cell customization and self sizing cells.

Summary

Wow! You've made a social network app for sharing restaurants. This is a huge chapter; by now you should understand the basics of CloudKit. The introduction of CloudKit could change the landscape of cloud providers. Apple has made it so easy for iOS developers to integrate their apps with iCloud. The service is completely free. With the release of CloudKit JS, you are able to build a web app for users to access the same containers as your iOS app. This is a huge deal for developers.

That said, CloudKit isn't perfect. CloudKit is an Apple product. I don't see any possibilities that the company would open up the service to other platforms. If you want to create a cloud-based app for both iOS and Android, CloudKit may not be your first choice. You may want to explore Google's Firebase, Parse server, Microsoft's Azure, Kinvey, or BassBox. If your primary focus is on iOS platform, CloudKit holds a lot of potential for both you and your users. I encourage you to adopt CloudKit in your next app.

For your reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinCloudKit.zip>.

For the solution to the exercise, you can download the project from <http://www.appcoda.com/resources/swift3/FoodPinCloudKitExercise.zip>.

Chapter 25

Localizing Your App to Reach More Users



Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" Improve the code and then document it to make it even clearer.

- Steve McConnell

In this chapter, let's talk about localization. The iOS devices including iPhone and iPad are available globally. The App Store is available in 150 countries around the world. Your users are from different countries and speak different languages. To deliver a great user experience and reach a global audience, you would want to make your app available in multiple languages. The process of adapting an app to support a particular language is usually known as *localization*.

Xcode has the built-in support for localization. It's fairly easy for developers to localize an app through the localization feature and a few API calls.

You may have heard of localization and internationalization. You may think that both terms refer to the process of translation; that's partially correct. In iOS development, internationalization is considered a milestone of building a localized app. Before your app can be adapted to different languages, you design and structure the app to be language and region independent. This process is known as *internationalization*. For instance, your app displays a price field. As you may know, some countries use a dot to indicate decimal place (e.g. \$1000.50), while many other countries use a comma instead (e.g. \$1000,50). The internationalization process involves designing the price field so that it can be adapted to different regions.

Localization is the process of adapting an internationalized app for different languages and regions. This involves translating static and visible text to a specific language and adding country-specific elements such as images, videos and sounds.

In this chapter, we'll localize the FoodPin app into Chinese and German. However, don't expect me to translate all the text in the app - I just want to show you the overall process of localization using Xcode.

Internationalizing Your App

The first step of building a localized app is *internationalization*. In this section, we will modify the app so that it can be easily adapted to various languages.

First, let's talk about the user-facing text of the app. There is tons of user-facing text in the source code. For convenience, we simply specify the strings in the source code. Figure 25-1 displays a few user-facing text in the `RestaurantTableViewController` class. When it comes to

localization, these hardcoded strings are not localizable. We have to internationalize them first.

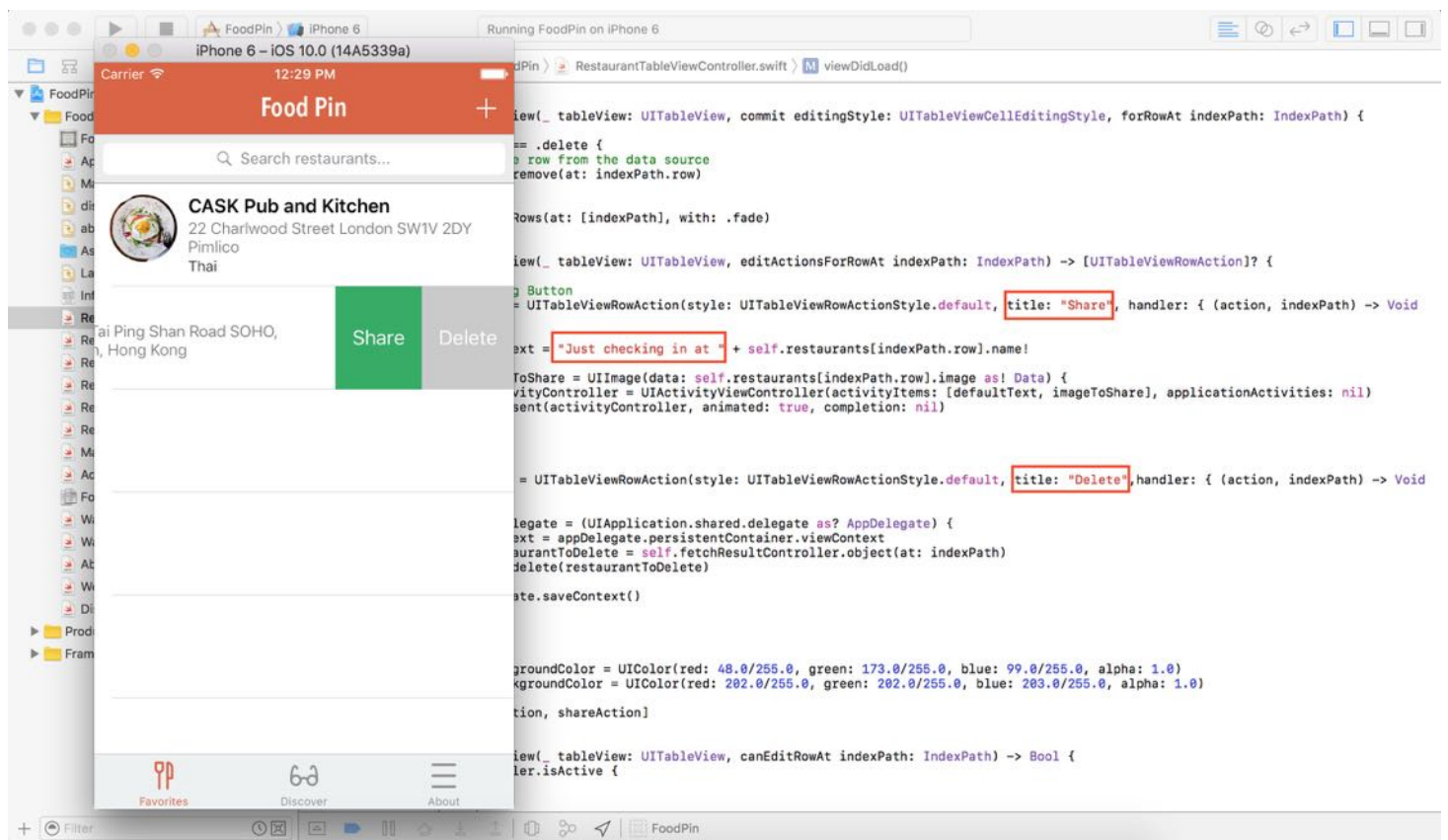


Figure 25-1. Sample user-facing text

At the heart of string internationalization is the `NSLocalizedString` macro. The macro allows you to internationalize the user-facing strings with ease. It has two arguments:

- `key` - the string to be localized
- `comment` - the string that is used to provide additional information for translation.

The macro returns a localized version of `String`. To internationalize the user-facing strings in your app, one thing that you need to do is wrap the existing strings with `NSLocalizedString`. Let's do some code changes so you can fully understand the macro. Take a look at the below code snippet from the `RestaurantDetailViewController` class:

```
switch indexPath.row {
case 0:
    cell.fieldLabel.text = "Name"
```



```

    cell.valueLabel.text = restaurant.name
case 1:
    cell.fieldLabel.text = "Type"
    cell.valueLabel.text = restaurant.type
case 2:
    cell.fieldLabel.text = "Location"
    cell.valueLabel.text = restaurant.location
case 3:
    cell.fieldLabel.text = "Phone"
    cell.valueLabel.text = restaurant.phone
case 4:
    cell.fieldLabel.text = "Been here"
    cell.valueLabel.text = (restaurant.isVisited) ? "Yes, I've been here
before. \(restaurant.rating!)" : "No"
default:
    cell.fieldLabel.text = ""
    cell.valueLabel.text = ""
}

```

All the field labels are displayed in English and are non-localizable. To make them language independent, you need to wrap the strings with `NSLocalizedString`:

```

switch indexPath.row {
case 0:
    cell.fieldLabel.text = NSLocalizedString("Name", comment: "Name Field")
    cell.valueLabel.text = restaurant.name
case 1:
    cell.fieldLabel.text = NSLocalizedString("Type", comment: "Type Field")
    cell.valueLabel.text = restaurant.type
case 2:
    cell.fieldLabel.text = NSLocalizedString("Location", comment:
"Location/Address Field")
    cell.valueLabel.text = restaurant.location
case 3:
    cell.fieldLabel.text = NSLocalizedString("Phone", comment: "Phone Field")
    cell.valueLabel.text = restaurant.phone
case 4:
    cell.fieldLabel.text = NSLocalizedString("Been here", comment: "Have you
been here Field")
    cell.valueLabel.text = (restaurant.isVisited) ? NSLocalizedString("Yes,
I've been here before. \(restaurant.rating!)", comment: "Yes, I've been here
before") : NSLocalizedString("No", comment: "No, I haven't been here")
default:
    cell.fieldLabel.text = ""
    cell.valueLabel.text = ""
}

```

Xcode actually stores the localized strings in `Localizable.strings` files. Each language has its own `Localizable.strings` file. Say, your user's device is using German as the default language, `NSString` looks up the German version of the `Localizable.strings` file and returns the string in German.

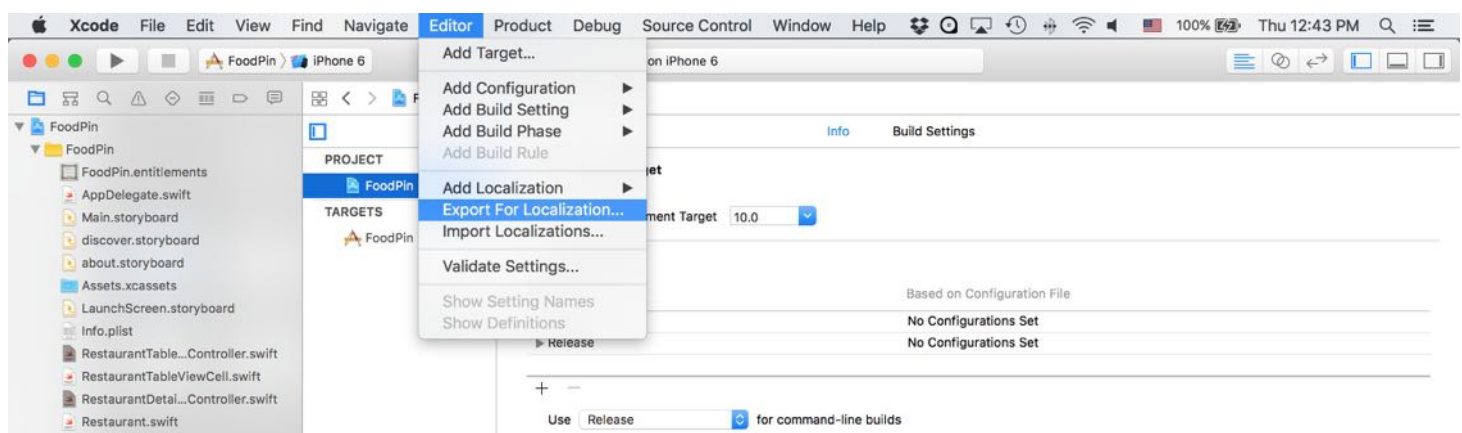
Tip: If you don't know how an app picks a language, it actually refers to the language settings of iOS (General > International > Language). An app refers to that setting and access the corresponding localized resources.

That's all we need to do to internationalize the user-facing strings. In the coming section, we will discuss about how to create the `Localizable.strings` file, and show you how to localize those labels in the storyboard.

Export for Localization

Starting from version 6, Xcode comes with an export feature to streamline the translation process. The export feature is intelligent enough to extract all localizable strings from your source code and Interface Builder/Storyboard files. The extracted strings are stored in an XLIFF file. If you haven't heard of the term, XLIFF stands for XML Localization Interchange File, which is a well-known and globally-recognized file format for localization.

To use the export feature, select the FoodPin project in the project navigator. Then head up to the Xcode menu and select Editor > Export For Localization. When prompted, choose a folder to save the XLIFF file. Xcode then examines all your files and generates the XLIFF file.



If you open Finder and go to your selected folder, you should find the `en.xliff` file. Open it with a text editor, you should find something like this in the file:

```

<trans-unit id="Been here">
  <source>Been here</source>
  <note>Have you been here Field</note>
</trans-unit>
<trans-unit id="Location">
  <source>Location</source>
  <note>Location/Address Field</note>
</trans-unit>
<trans-unit id="Name">
  <source>Name</source>
  <note>Name Field</note>
</trans-unit>
<trans-unit id="No">
  <source>No</source>
  <note>No, I haven't been here</note>
</trans-unit>
<trans-unit id="Phone">
  <source>Phone</source>
  <note>Phone Field</note>
</trans-unit>
<trans-unit id="Type">
  <source>Type</source>
  <note>Type Field</note>
</trans-unit>
<trans-unit id="Yes, I've been here before. (restaurant.rating!)">
  <source>Yes, I've been here before. (restaurant.rating!)</source>
  <note>Yes, I've been here before</note>
</trans-unit>

```

Xcode has extracted the strings that we have just wrapped with the `NSLocalizedString` macro. Other than the above, the file should have something like this:

```

<file original="FoodPin/Base.lproj/LaunchScreen.storyboard" source-
language="en" datatype="plaintext">
  <header>
    <tool tool-id="com.apple.dt.xcode" tool-name="Xcode" tool-version="8.0"
build-num="8S201h"/>
  </header>
  <body/>
</file>
<file original="FoodPin/Base.lproj/Main.storyboard" source-language="en"
datatype="plaintext">
  <header>
    <tool tool-id="com.apple.dt.xcode" tool-name="Xcode" tool-version="8.0"
build-num="8S201h"/>
  </header>
  <body>
    <trans-unit id="5Ik-ha-Kpq.normalTitle">

```

```

    <source>NEXT</source>
    <note>Class = "UIButton"; normalTitle = "NEXT"; ObjectID = "5Ik-ha-
Kpq";</note>
  </trans-unit>
  <trans-unit id="87e-VN-sYe.title">
    <source>New Restaurant</source>
    <note>Class = "UINavigationController"; title = "New Restaurant"; ObjectID =
"87e-VN-sYe";</note>
  </trans-unit>
  <trans-unit id="BgK-EL-iP5.text">
    <source>LOCATION</source>
    <note>Class = "UILabel"; text = "LOCATION"; ObjectID = "BgK-EL-iP5";
</note>
  </trans-unit>

```

The export feature automatically examines the storyboard and extracts all the localizable strings, including labels and button titles. Typically you pass the file to a professional translator for translation. The translator then uses an XLIFF-enabled tool to add all the missing translations. Or, like me, you can use Google Translate to do the translation and Xcode editor to edit the file. You only need to translate the text enclosed by the tag. Content inside the tag contains comments that we (or Xcode) added, so you do not need to translate them. The translated text is placed inside the tag. Below is a part of the Traditional Chinese translation file:

```

<file original="FoodPin/Base.lproj/Main.storyboard" source-language="en"
target-language="zh-Hant" datatype="plaintext">
  <header>
    <tool tool-id="com.apple.dt.xcode" tool-name="Xcode" tool-version="8.0"
build-num="8S201h"/>
  </header>
  <body>
    <trans-unit id="5Ik-ha-Kpq.normalTitle">
      <source>NEXT</source>
      <target>下一步</target>
      <note>Class = "UIButton"; normalTitle = "NEXT"; ObjectID = "5Ik-ha-
Kpq";</note>
    </trans-unit>
    <trans-unit id="87e-VN-sYe.title">
      <source>New Restaurant</source>
      <target>新增餐廳</target>
      <note>Class = "UINavigationController"; title = "New Restaurant"; ObjectID =
"87e-VN-sYe";</note>
    </trans-unit>
    <trans-unit id="BgK-EL-iP5.text">
      <source>LOCATION</source>

```

```

<target>地址</target>
  <note>Class = "UILabel"; text = "LOCATION"; ObjectID = "BgK-EL-iP5";
</note>
</trans-unit>
<trans-unit id="BuA-j5-8a5.placeholder">
  <source>Restaurant Phone</source>
  <target>餐廳電話</target>
  <note>Class = "UITextField"; placeholder = "Restaurant Phone"; ObjectID
= "BuA-j5-8a5";</note>
</trans-unit>
<trans-unit id="DgX-19-5vc.placeholder">
  <source>Restaurant Name</source>
  <target>餐廳名字</target>
  <note>Class = "UITextField"; placeholder = "Restaurant Name"; ObjectID
= "DgX-19-5vc";</note>
</trans-unit>

```

The first line of the XML code specifies the original file and the target language of the translation. The target language specifies the language code of the translation. `zh-Hant` is the language code of Chinese. For German, the language code is `de`.

Import Localizations

Assuming your translator has completed the translations and passed you the translation files (download them from <http://www.appcoda.com/resources/swift3/translations.zip>), you just need a few clicks to import the translations.

Head up to the Editor menu and select *Import Localizations*. When prompted, select the translation file (e.g. `zh-Hant.xliff`). Xcode then automatically compares the translation with the existing files and shows you the differences (see figure 25-3).

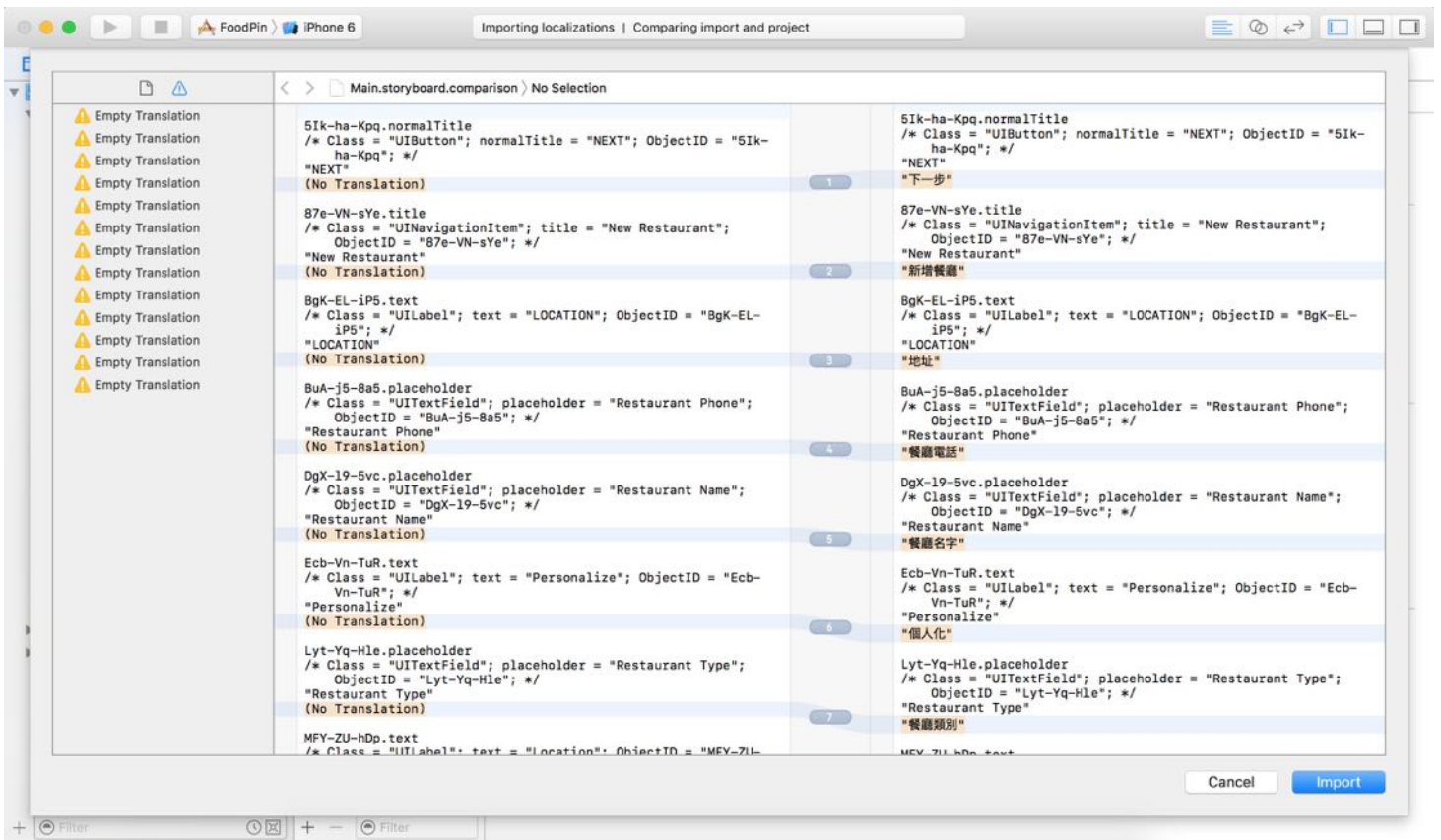


Figure 25-3. Import localizations

If you confirm the translation is correct, click `Import` button to import it right away. If you have other translations (e.g. de.xliff), repeat the same procedures to add them. When finished, Xcode displays the available localizations in the project screen. You should notice that Xcode adds the localized resource files in the project navigator.

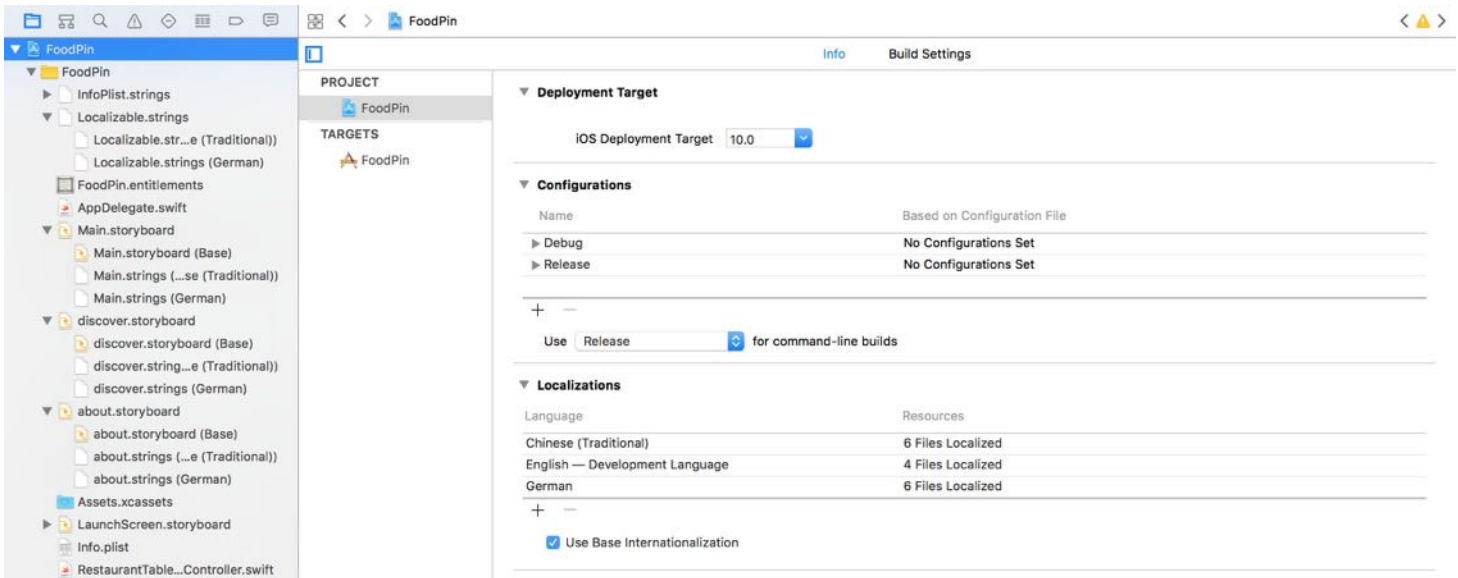


Figure 25-4. German and Chinese localizations added to the Xcode project

Before we move on to testing the localized app, let me give you some more information about base internationalization and the localized resource files. In the project navigator, you should find the `Localizable.strings` files. These files are automatically generated after the import process. The localized strings in source code (i.e. those wrapped with `NSLocalizedString`) are stored in a language-dependent `Localizable.strings` file. As we have imported the `zh-Hant` and `de` translations, there are two versions of `Localizable.strings` files.

If you look into any of the `Localizable.strings` file (say, the Chinese version), you will find the translation of the localizable strings. At the beginning of each entry, it is the comment you've put in the source code. On the left side of the equal sign, it is the key you specify when using `NSLocalizedString` macro.

```
/* Have you been here Field */
"Been here" = "有來過這裡嗎";

/* Facebook */
"Facebook" = "臉書";

/* Follow Us */
"Follow Us" = "追蹤我們";

/* Leave Feedback */
"Leave Feedback" = "給我們反饋";
```



```
/* Locate */  
"Locate" = "定位";  
  
/* Location/Address Field */  
"Location" = "位置 / 地址";
```

Note: While you can edit the translations directly, I would recommend you to use the export feature as I have just demonstrated.

The second thing I want to talk about is *base internationalization*. As you can see from figure 25-4, `Main.storyboard` is now split into three files: the base file, the Chinese translation file and the German translation file. But what is `Main.storyboard (Base)` ?

The concept of base internationalization was first introduced in Xcode 4.5. Prior to Xcode 4.5, there was no concept of base internationalization. Xcode replicates the whole set of storyboards for each localization. For example, let's say if your app is localized into 5 languages. Xcode generates 5 sets of storyboards for localization purposes. There is a major drawback of this design. When you need to add a new UI control in the storyboard, you'll need to add the same element in each localized storyboard. This is a tedious process, which is why Apple introduced base internationalization in Xcode 4.5.

With base internationalization, an Xcode project has only one set of storyboards that is set to the default language. This storyboard is known as the base internationalization. Whenever the storyboard is localized into another language, Xcode only generates a `.strings` file containing all the text of the base storyboard. This efficiently separates the UI design from the translations.

Testing the Localized App

One way to test the localization is to change the language preference of the simulator and then run the localized app on it. Alternatively, you can utilize a preview feature of Xcode that lets you test your app in different languages and regions, both at runtime and in Interface Builder. I will go through them with you one by one.

Xcode 8 supports preview at runtime. You can enable this feature by editing the scheme sheet and set your preferred language in the dialog box.

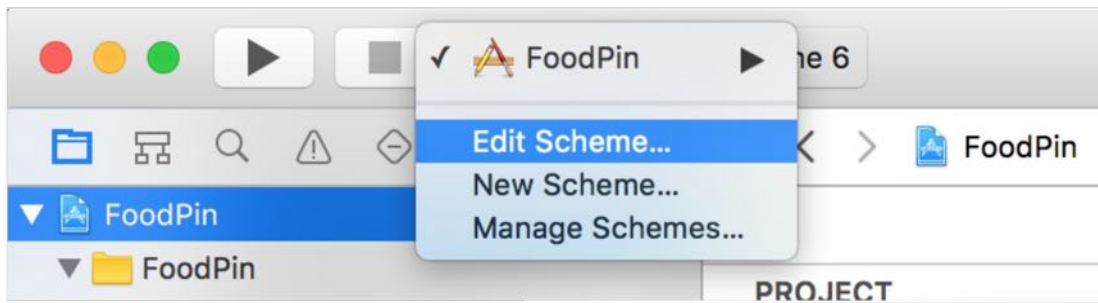


Figure 25-5. Edit scheme option

In the dialog box, select *Options* and change the application language to your preferred language. For example, Chinese (Traditional). Click the Close button to save the setting.

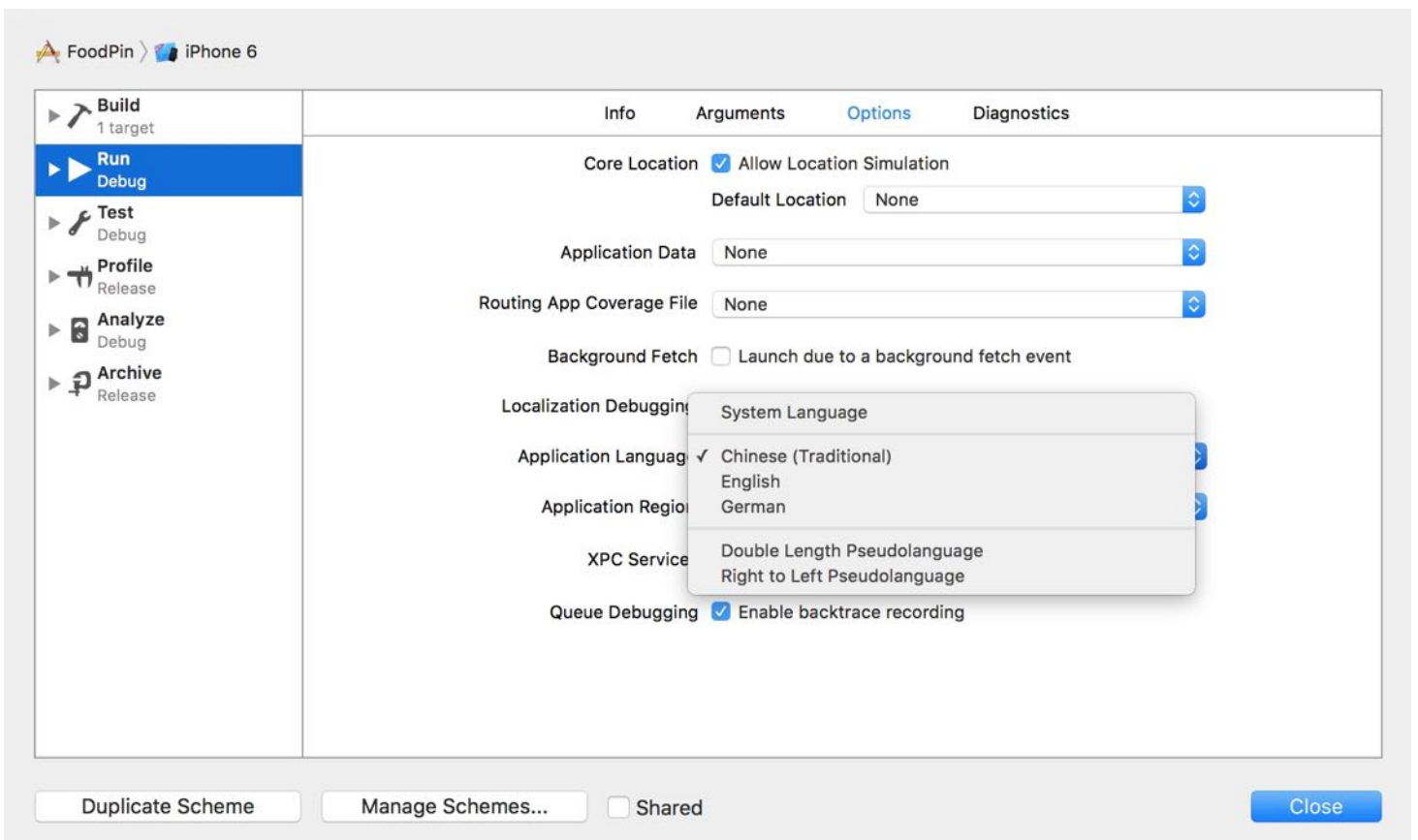


Figure 25-6. Changing the application language to your preferred language

Now click the Run button to launch the app; the language of the simulator should set to your

preferred language. If you've set it to Chinese, your app should look like the screenshot shown in figure 25-7.

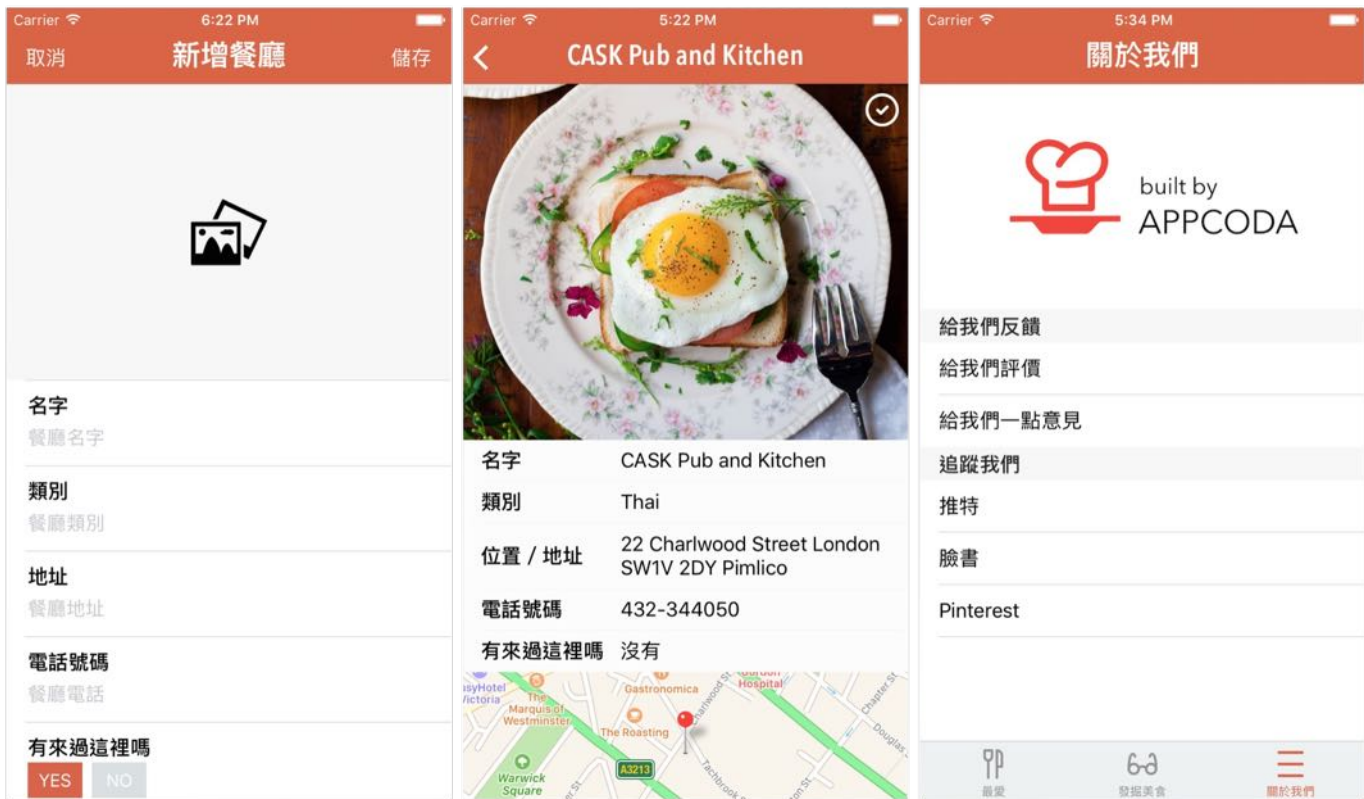


Figure 25-7. FoodPin App in Traditional Chinese

Alternatively, Xcode allows you to preview the localized app UI in the preview assistant. To launch the preview assistant, select `Main.storyboard` and click the button at the top-left corner of the Interface builder. Scroll down to `Preview (1)`. Hold the option key and click `Main.storyboard (Preview)`.

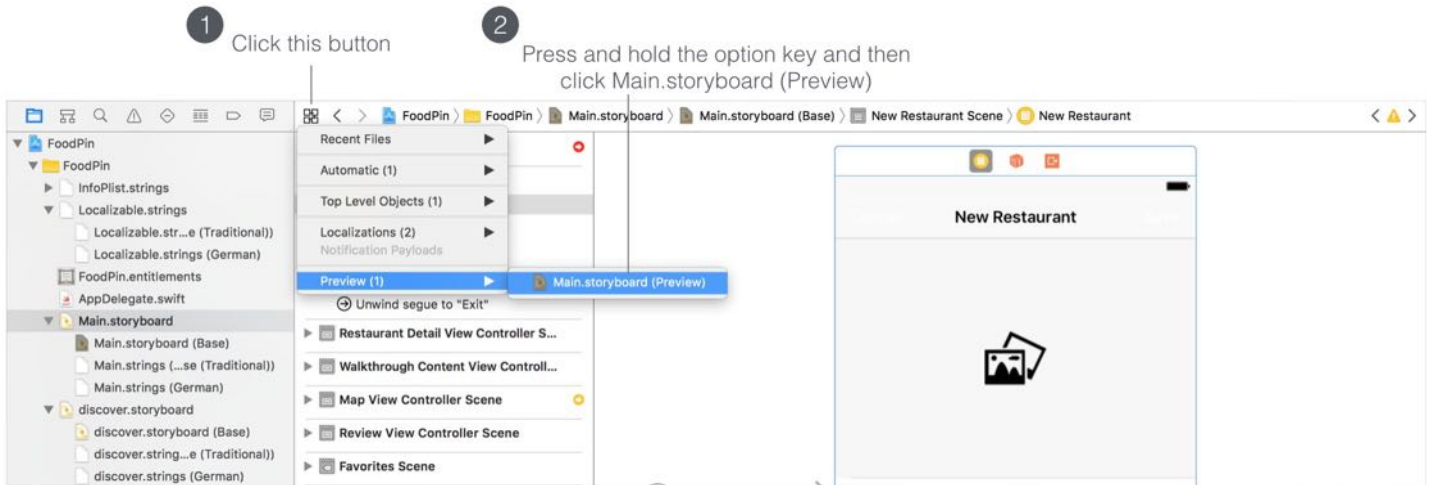


Figure 25-8. Open the preview assistant

Xcode then opens the preview assistant. You can preview the localized UI by selecting the language from the language pop-up at the lower right corner of the preview window.

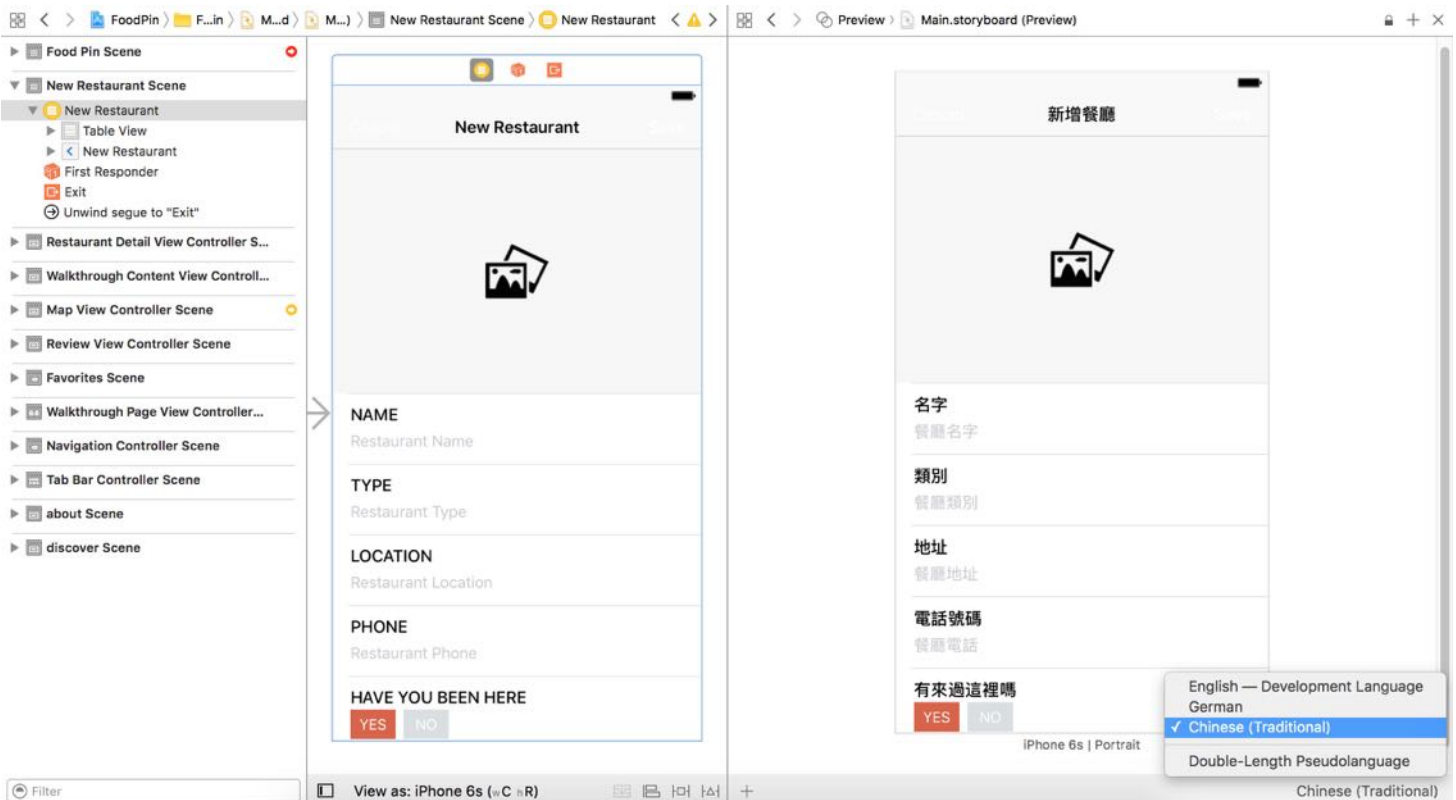


Figure 25-9. Previewing your localized app UI using Preview Assistant

Localizing Resources

Other than text, you may want to localize resources such as images, videos and sounds. For example, you can display a locale-specific image in the About tab. Figure 25-10 shows the German and Chinese version of the image.



Figure 25-10. German (left) and Chinese (right) version of the image in About tab

From the very beginning, we used the asset catalog to manage the project's images. Unfortunately, asset catalogs do not support localization. To localize an image in your app, one way is to import the file directly into the project navigator.

To demonstrate how it works, first download this image pack from <http://www.appcoda.com/resources/swift3/aboutfoodpin.zip>. After unzipping the file, you will find three versions of the about logo.

Now, right click on the FoodPin folder in the project navigator and select `New Group`. Name the group `images`. This step is optional but creating a new group allows you to better organize your project resources. Next, drag `aboutfoodpin.png` (under the `base` folder) from Finder to the project navigator. Make sure that `Copy item if needed` is enabled when prompted. The image is then copied to the project folder. Now, select the file and bring up the File inspector. In the localization section, you should find the `Localize...` button. Click the button, choose `Base` and click the `Localize` button to confirm.

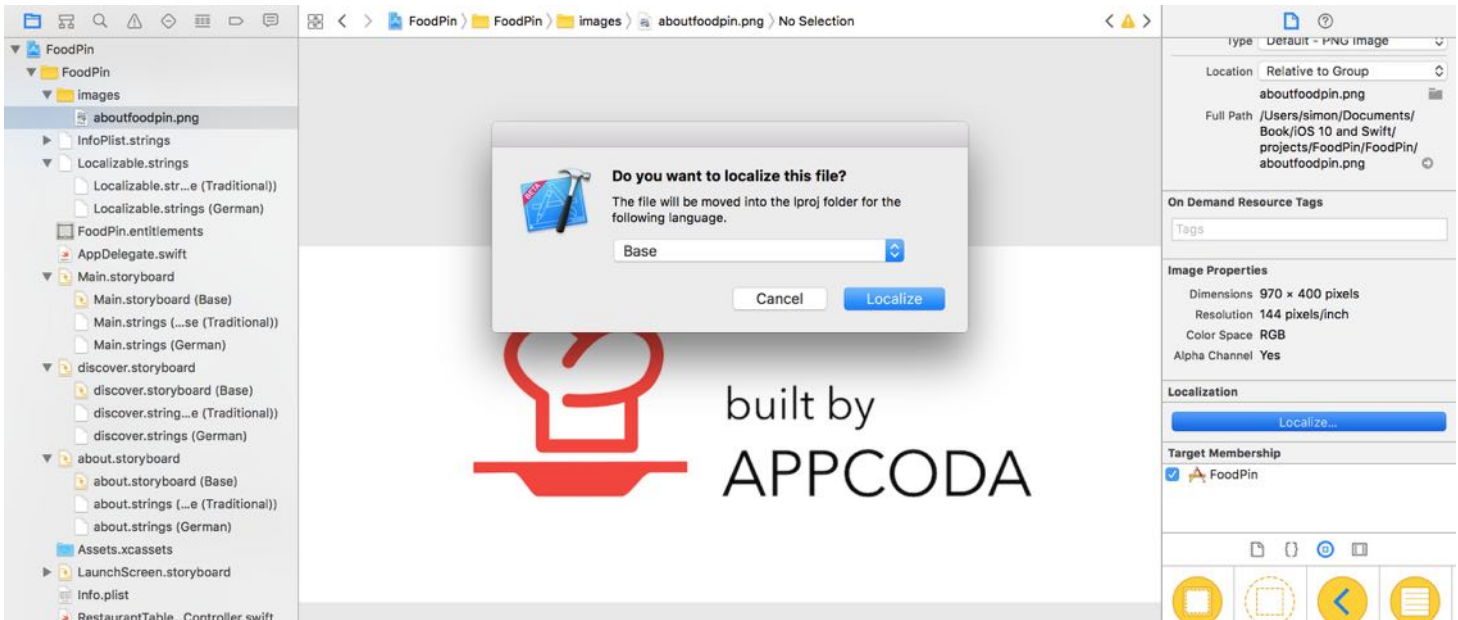


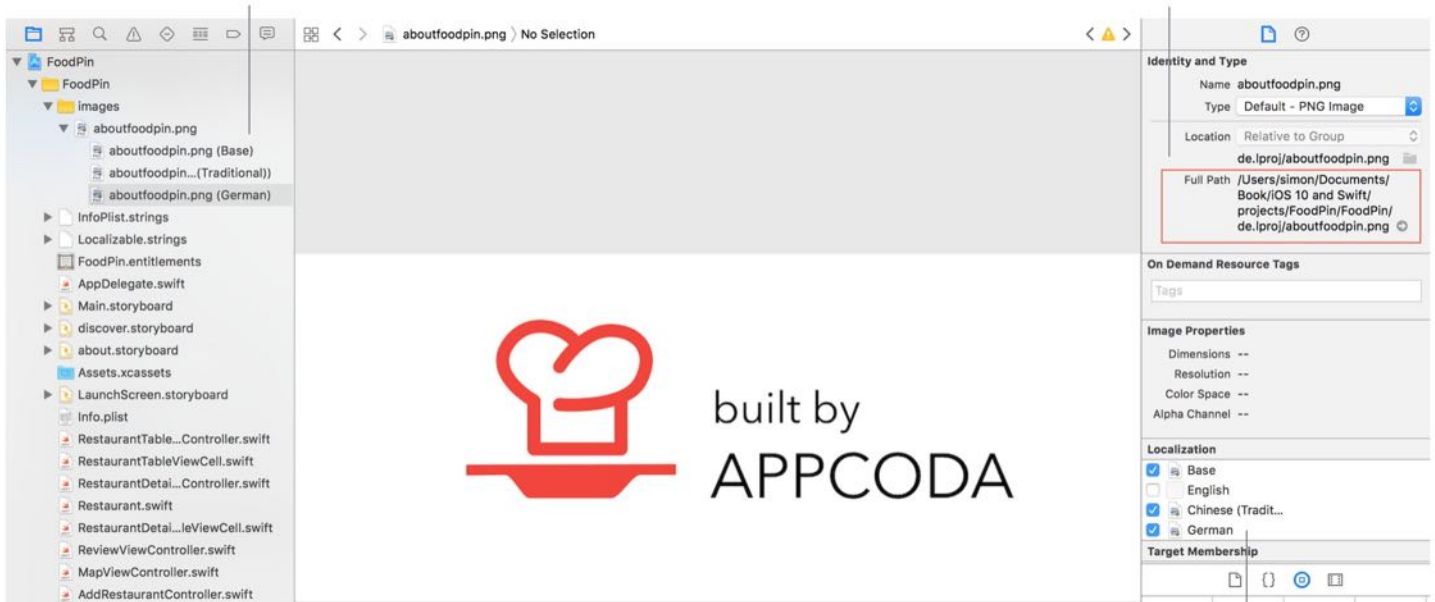
Figure 25-11. Localize the about logo

After that, you'll see the Chinese (Traditional) and German options in the localization section. Select both languages and Xcode code will generate three versions of `aboutfoodpin.png` in project navigator.

2

Xcode adds two versions of aboutfoodpin.png

File path of the localized images



1

Enable localization for Chinese and German

Figure 25-12. Enable the German and Chinese (Traditional) localization

Now that you've created the localized versions of the image, they are stored in separate folders. You can locate the folder by selecting the localized version of image to reveal the full path in the File Inspector. The German version of the image is stored under the `de.lproj` folder, while the Traditional Chinese version is stored under the `zh-Hant.lproj` folder.

By default, the image generated in the localized folders is the same as the original. To localize the image, what you need to do is to replace the images under `de.lproj` and `zh-Hant.lproj` with their own localized versions.

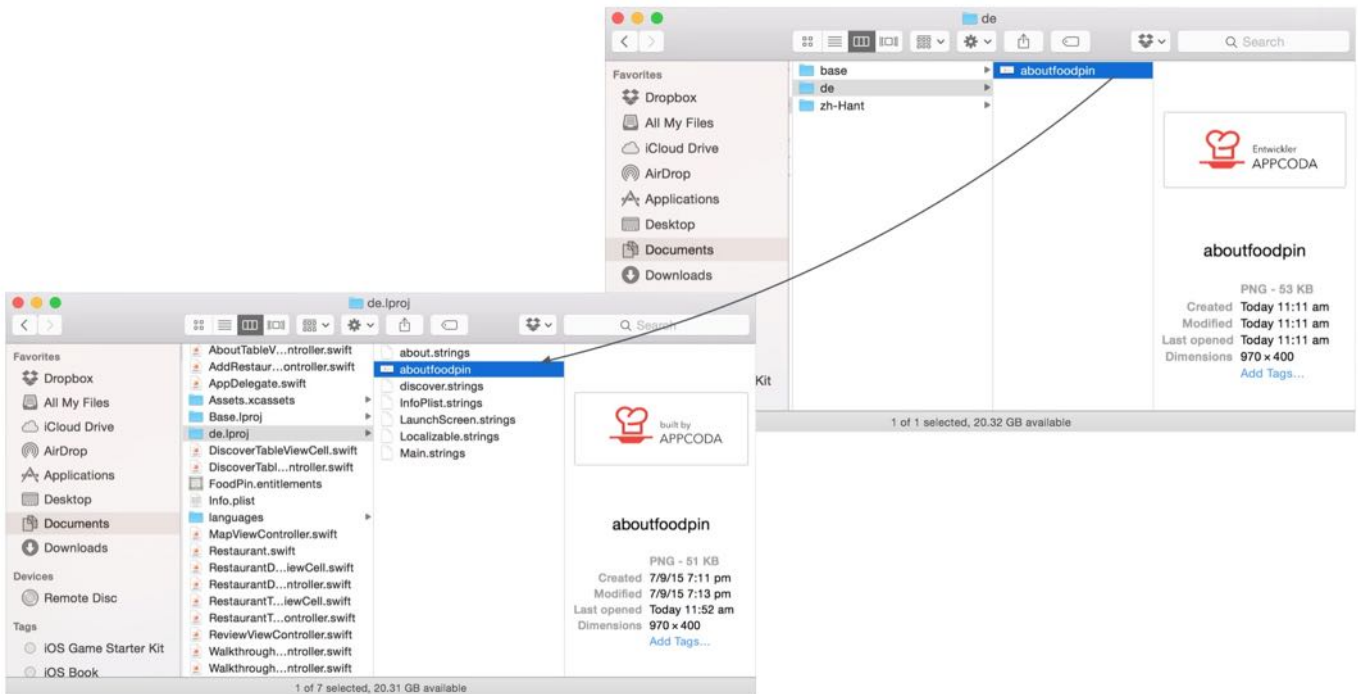


Figure 25-13. Copy your own localized images to replace the original image

Lastly, remember to change the image of the About view controller from about to `aboutfoodpin.png` in `about.storyboard`, because we no longer use the about image from the asset catalog.

Cool! It's time to test the localized UI. If you got everything right, here are what you'll see for both German and Traditional Chinese versions.



Figure 25-14. Food Pin app in German (left) and Traditional Chinese (right)

Summary

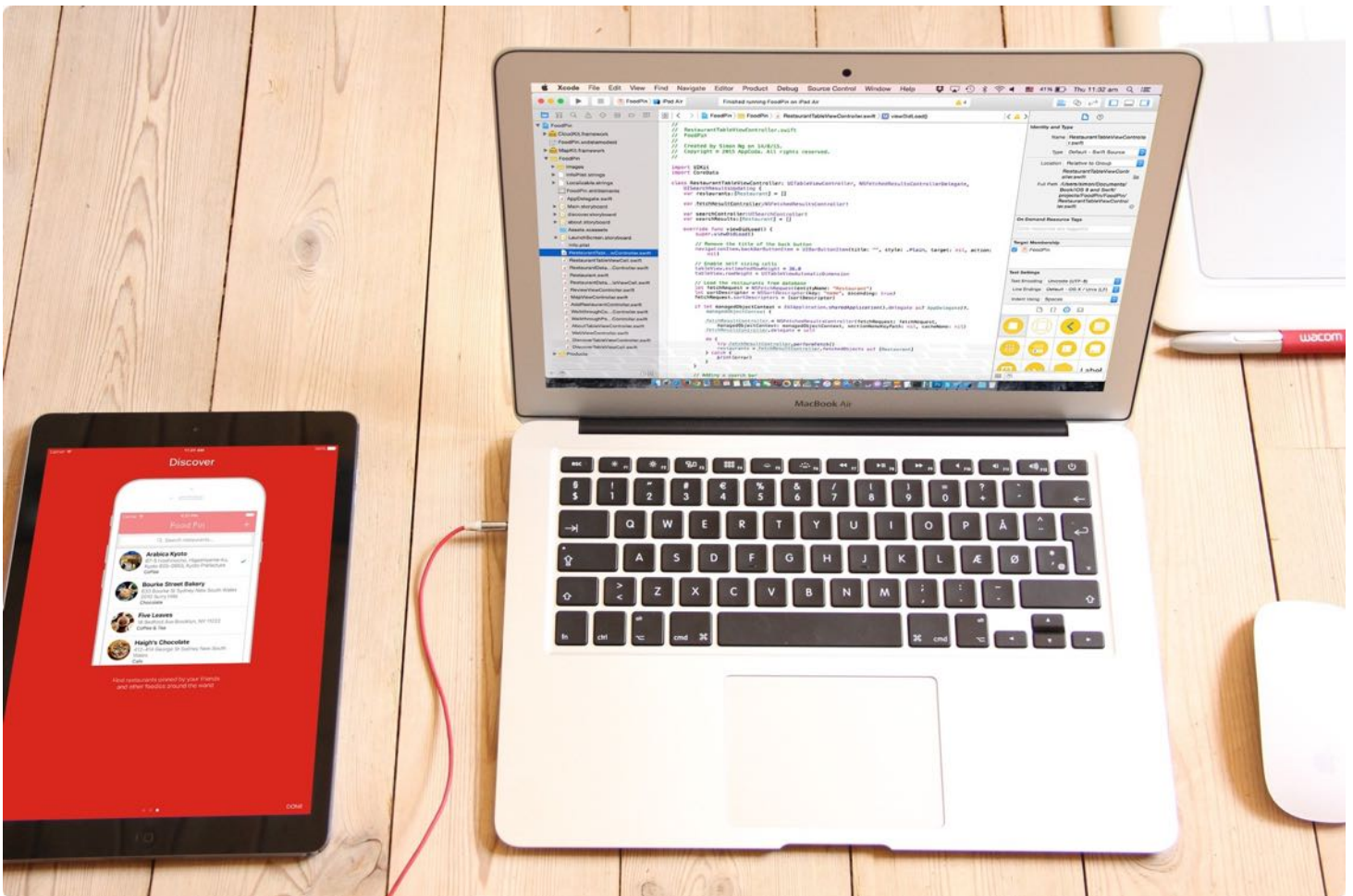
In this chapter, I have walked you through the localization process in Xcode 8. By using the export feature, it is pretty easy to localize an iPhone app.

You've learned how to localize text and images in storyboards using the built-in localization support of Xcode. You should also know how to localize strings using the `NSLocalizedString` macro. Remember that your apps are marketed to a global audience - users prefer to have an app UI in their own language. By localizing your app, you'll be able to provide a better user experience and attract more downloads.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinLocalization.zip>.

Chapter 26

Deploying and Testing Your App on a Real iOS Device



If we want users to like our software, we should design it to behave like a likable person: respectful, generous and helpful.

- Alan Cooper

Up till now, you have been running and testing your app on the built-in simulator. The simulator is a great companion for app development especially if you do not own an iPhone. While the simulator is good, you can never rely completely on the simulator. It's not recommended to submit your app to App Store without testing it on a real device. Chances are

there are some bugs that only shows up when your app runs on a physical iPhone or over the cellular network. If you're serious about building a great app, this is a must to test your app on a real device before releasing it to your users.

One great news, especially for aspiring iOS developers, is that Apple no longer requires you to enroll in the Apple Developer Program before you can test your app on an iOS device. You can now simply sign in with your Apple ID, and your app is ready to run on your iPhone or iPad. To some of you, I think this is one of the coolest features in Xcode 7/8. However, if your app makes use of the services like CloudKit and Push Notifications, you still need to enroll in the Apple Developer Program, which costs \$99 per year. I know, for some, this is a significant amount of money. But if you read the book from the very beginning and are still with me, I believe you have demonstrated a strong determination to build an app and deploy it to your audience. It's not easy to make it this far! So why stop here? If you're not on a tight budget, I highly recommend you enroll in the program so that you can continue to learn the rest of the materials and most importantly, submit your app to App Store.

To test an app on a physical device, you will need to perform a few configurations:

- Request your development certificate
- Create an App ID for your app
- Configure your device for development
- Create a provisioning profile for your app

In the old days of iOS development, you had to manage the above configurations all on your own through the iOS Provisioning Portal (or Member Center). Xcode 8 automates the whole signing and configuration processes by introducing a new feature called *Automatic Signing*. This makes your life a lot easier. You will see what I mean shortly.

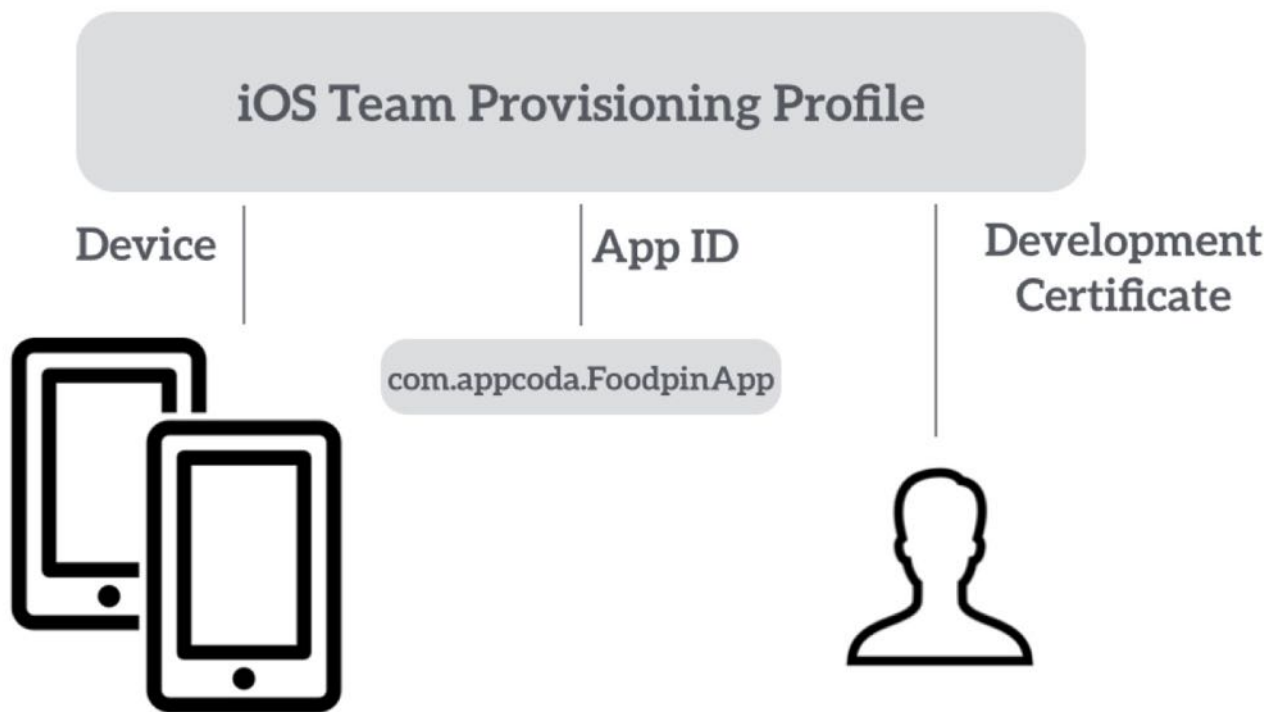
Understanding Code Signing and Provisioning Profiles

You may wonder why you need to request a certificate and go through all these complicated procedures just to put an app on a device. Not to say, it's your own app and device. The answer is simply based around security. Every app deployed on an iOS device is code signed. Xcode uses both your private key and digital certificate to sign the code. By signing your code, Apple knows you're the rightful owner of the app. As you know, Apple only issues digital certificate to

legitimate developers who have joined the developer program. This also prevents non-authorized developer to deploy apps on the app store.

Apple issues two types of digital certificates: *Development* and *Distribution*. To run an app on a real device, you just need the development certificate. Later when you submit an app to App Store, you have to sign your app using a distribution certificate. Again, this is automatically taken care by Xcode 8. We will discuss app submission in the last chapter. For now, let's focus on deploying your app on a device.

Provisioning profile is another term you come across in app deployment. Provisioning refers the process of preparing and configuring an app to launch on devices. A team provisioning profile allows your app to be signed and run by all team members (if you're an indie developer, you're the only member in the team) on their devices. Figure 26-4 shows an illustration of a sample provisioning profile.



Reviewing Your Bundle ID

Before moving on, I would like you to review your bundle ID. As discussed before, a bundle ID identifies a single app, so it must be unique. You must not use `com.appcoda.FoodPin` because it has been used by the demo app. In this case, change the bundle ID to whatever you like. Just

make sure it is unique and in reverse-DNS format (e.g. `edu.self.newfoodapp`).

Note: After your app is published on the app store, you will not be able to change the bundle ID.

Automatic Signing in Xcode 8

Automatic signing is a new feature in Xcode 8. As mentioned before, you will need to manage the following tasks before you can deploy and run your app on a real device:

- Request your development certificate
- Create an App ID for your app
- Create a provisioning profile for your app

With automatic signing, you no longer need to create your development certificate or provisioning profile manually. Xcode 8 takes care of all these tasks for you.

When creating an Xcode project, automatic signing is enabled by default. If you select the FoodPin project in the project navigator and go to the FoodPin target, you should find the *Automatically manage signing* option is on.

By now, the *Team* option is set to *None*. To use automatic signing, you have to add an account to your project. Click the *Team* option and choose *Add an account*. Fill in your Apple ID/password, and click `Add` to proceed.

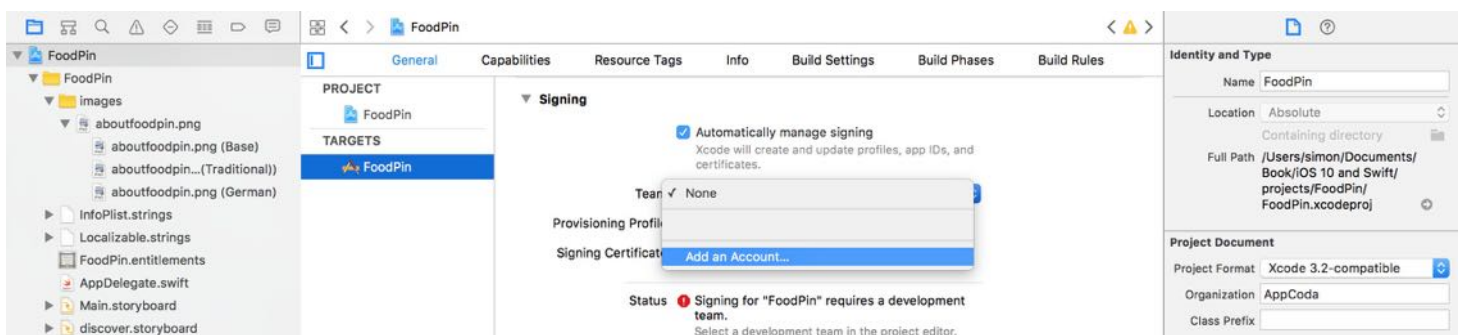


Figure 26-2. Adding your iOS developer account in Xcode

Even if you haven't enrolled in the Apple Developer Program, you can still sign in with your

Apple ID.

Once you add your account, Xcode 8 will generate the development certificate and create the provisioning profile automatically. In the signing section, you will find your signing certificate and provisioning profile. You can further click the info icon (next to the provisioning profile) to reveal the details.

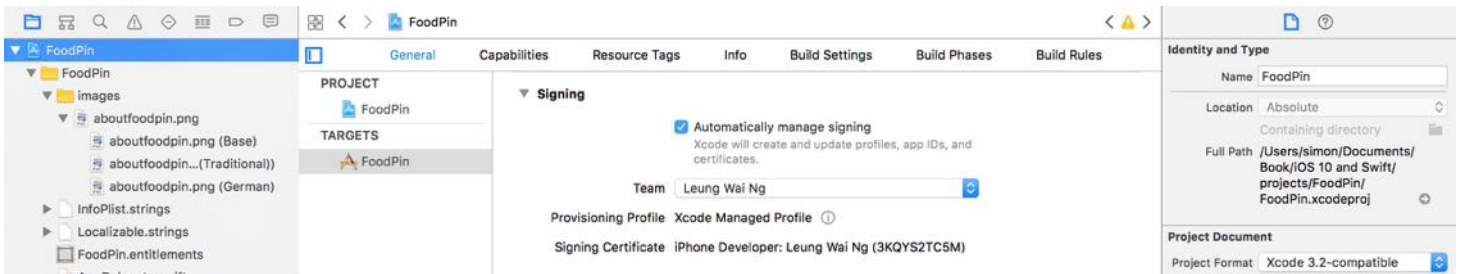


Figure 26-3. Xcode creates the provisioning profile and signing certificate for you

You can always review your App ID and provisioning profile in the Member Center. Open Safari and access <https://developer.apple.com/membercenter/>. Sign in with your developer account and you'll see the below screen.

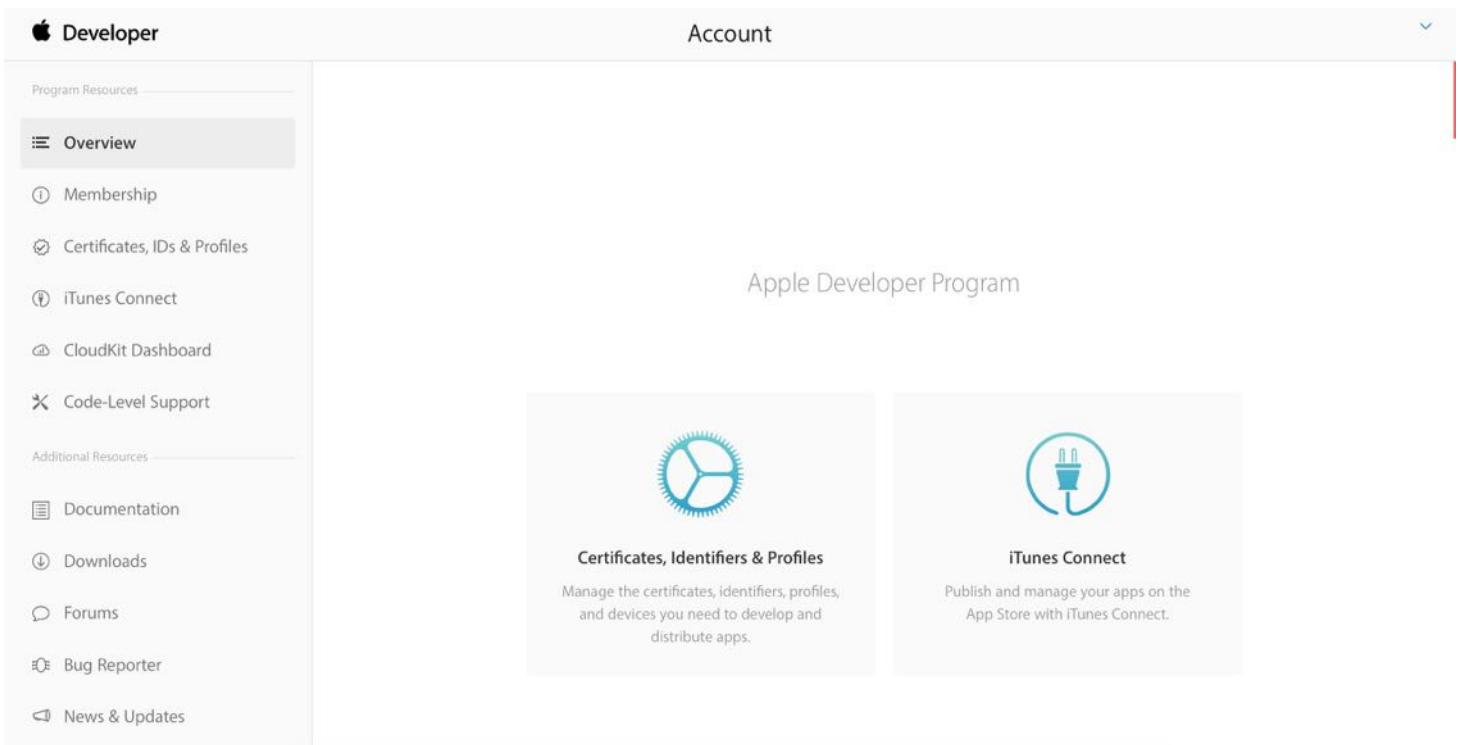


Figure 26-4. Member Center for iOS Developer

Note: The "Certificates, identifiers & Profiles" will only appear if you have enrolled in the Apple Developer Program.

Select "Certificates, identifiers & Profiles" and you'll navigate to another screen for managing certificates, identifiers, devices and provisioning profiles. Choose identifiers under iOS Apps and you will find the App ID (which is the same as your bundle ID) of your app. Figure 26-5 shows a sample App ID.

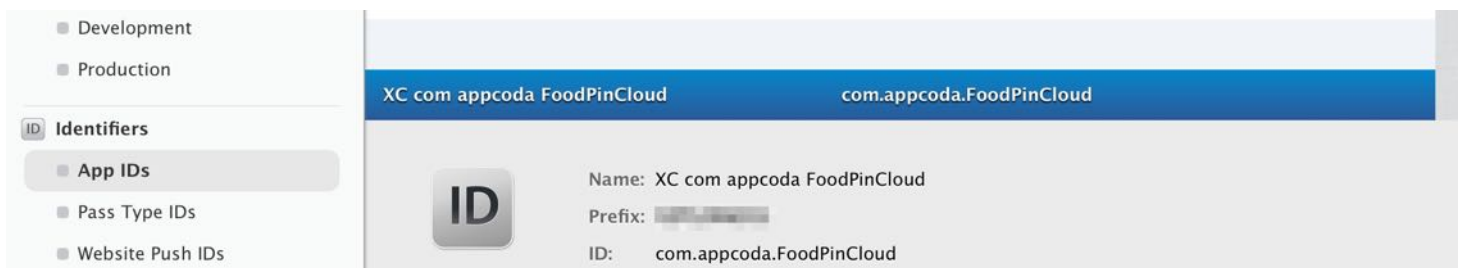


Figure 26-5. Sample App ID

To review the provisioning profile, you select Provisioning Profiles and then Development. Your provisioning profile name is like "iOS Team Provisioning Profile: ".

Launching the App on Your Device

Now connect your iOS device to your Mac. If it is eligible for deployment, you can select it from the Scheme pop-up menu.

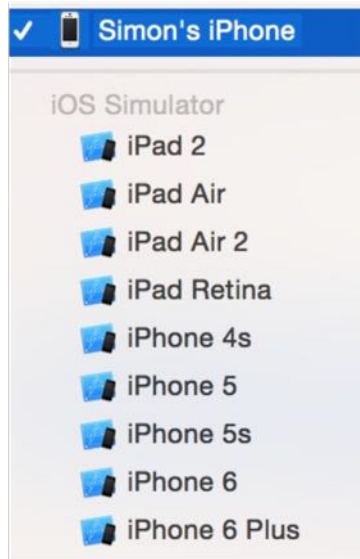


Figure 26-6. iOS device in the Scheme pop-up menu

Quick note: If your iOS device is ineligible, fix the issue before continuing. For example, if the device doesn't match the deployment target, upgrade the version of iOS on the device.

You're now ready to deploy and run the app on your device. Make sure you select your iPhone or iPad in the Scheme pop-up menu, and then hit the Run button.

Note: Remember to unlock your iPhone before deploying the app. Otherwise, you won't be able to launch it.

For the very first time running your app, Xcode may prompt you with a security error. Even though your app was deployed on your device, due to a security restriction, Xcode could not launch it. Even if you try to launch it manually, you will end up with an error prompt (see figure 26-7). To fix the error, go to Settings > General > Profiles. Select the developer profile and click the `Trust <email>` option. It should prompt you for confirmation. Simply click `Trust` to proceed. Once you trust the developer, you will be able to launch the app.

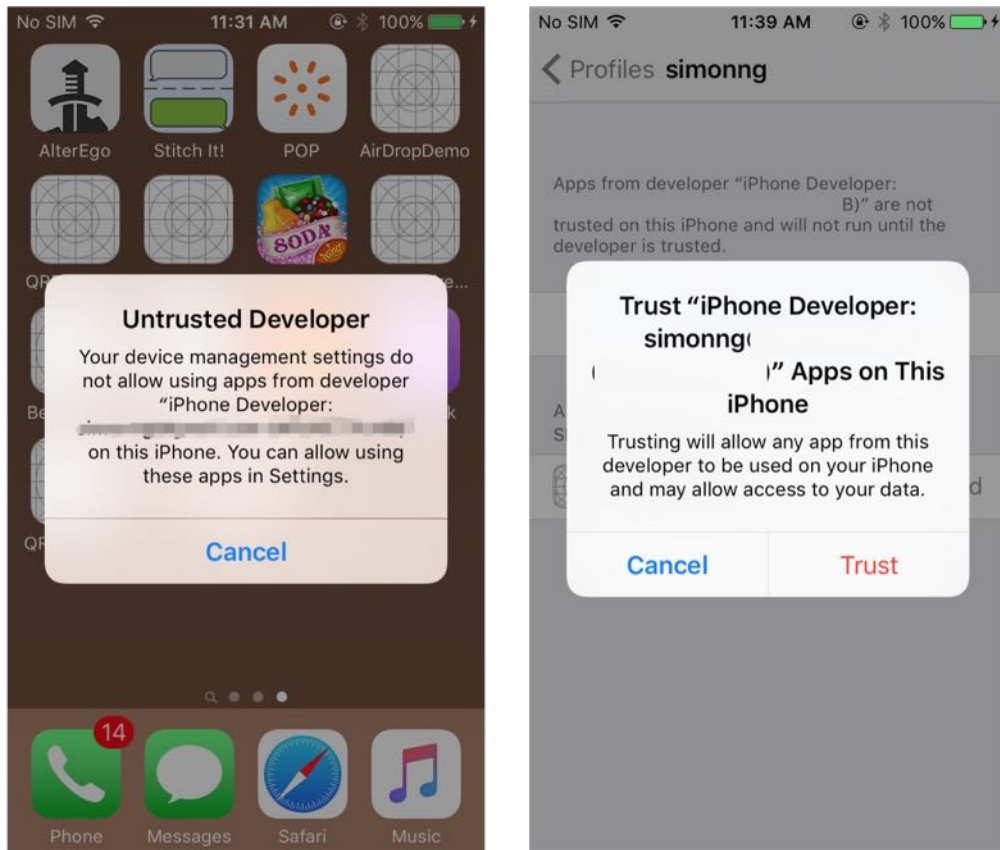


Figure 26-7. Prompt to trust the app developer (left), Trust the developer in Settings (right)

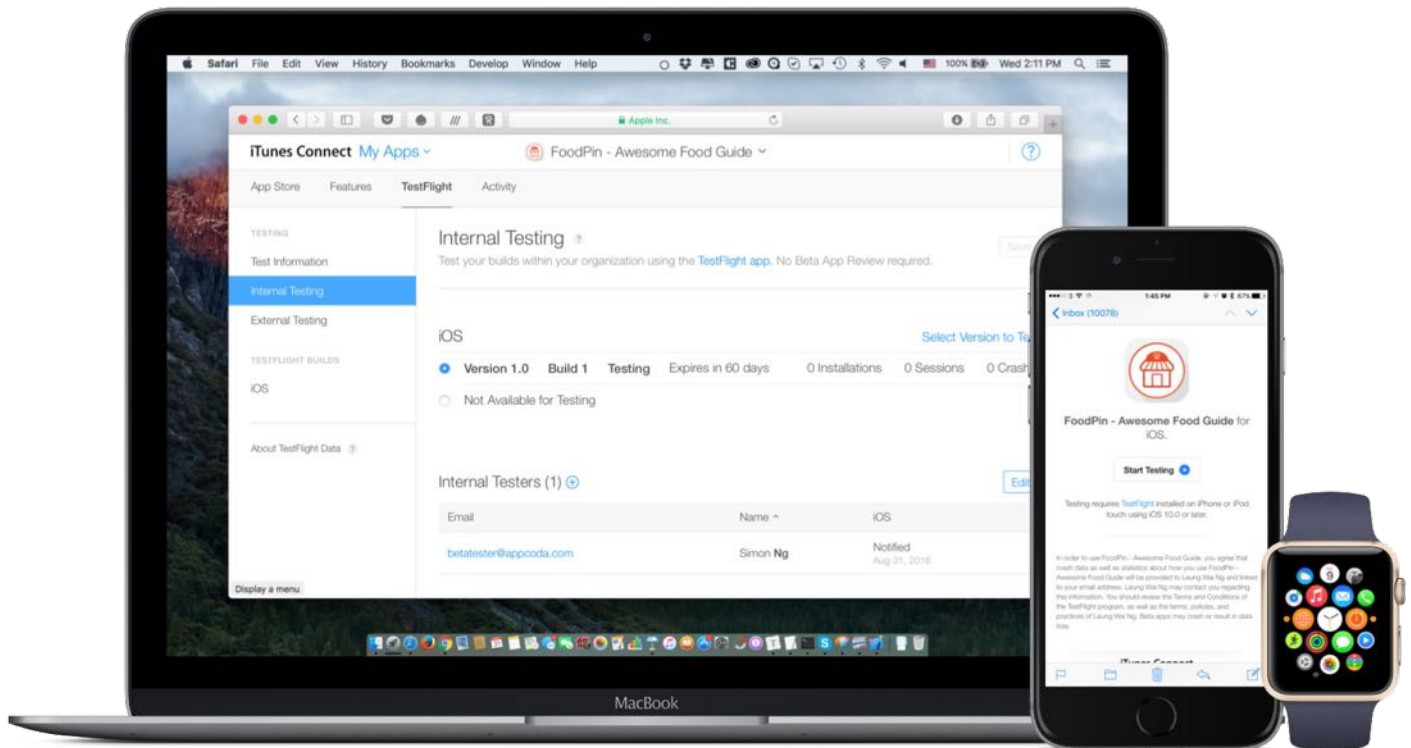
Summary

Xcode 8 makes your developer's life easier by simplifying the way to create a provisioning profile, app ID creation and developer certificate generation. With a few clicks, you will be able to test your app on your devices.

Next up, I will show you how to beta testing your app using TestFlight.

Chapter 27

Beta Testing with TestFlight



If you are not embarrassed by the first version of your product, you've launched too late.

- Reid Hoffman, LinkedIn

Now that you have completed the testing of your app on a real device, what's next? Submit your app directly to App Store and make it available for download? Yes, you can if your app is a simple one. If you're developing a high quality app, don't rush to get your app out, as I suggest you beta test your app before the actual release.

A beta test is a step in the cycle of a software product release. I know you have tested your app using the built-in simulator and on your own device. Interestingly, you may not be able to uncover some of the bugs, even though you're the app creator. By going through beta test, you

would be amazed at the number of flaws discovered at this stage. Beta testing is generally opened to a select number of users. They may be your potential app users, your blog readers, your Facebook followers, your colleagues, friends or even family members. The whole point of beta testing is to let a small group of real people get their hands on your app, test it, and provide feedback. You want your beta tester to discover as many bugs as possible in this stage so that you can fix them before rolling out your app to the public.

You may be wondering how can you conduct a beta test for your app, how beta testers run your app before it's available on App Store and how testers report bugs?

In iOS 8, Apple released a new tool called **TestFlight** to streamline the beta testing. You may have heard of TestFlight before. It has been around for several years as an independent mobile platform for mobile app testing. In February 2014, Apple acquired TestFlight's parent company, *Burstly*. Now TestFlight is integrated into iTunes Connect and iOS that allows you to invite beta testers using just their email addresses.

TestFlight makes a distinction between beta testers and internal users. Conceptually, both can be your testers at the beta testing stage. However, TestFlight refers internal users as members of your development team who have been assigned the Technical or Admin role in iTunes Connect. You're allowed to invite up to 25 internal users to test your app. A beta tester, on the other hand, is considered as an external user outside your team and company. You can invite up to 2,000 users to beta test your app.

There is a catch, though. Your app must be approved by Apple before you can invite beta testers for testing. This restriction doesn't apply to internal users. Your internal users can begin beta testing once you upload your app to iTunes Connect.

Note: You have to enroll in the Apple Developer Program before you can access TestFlight.

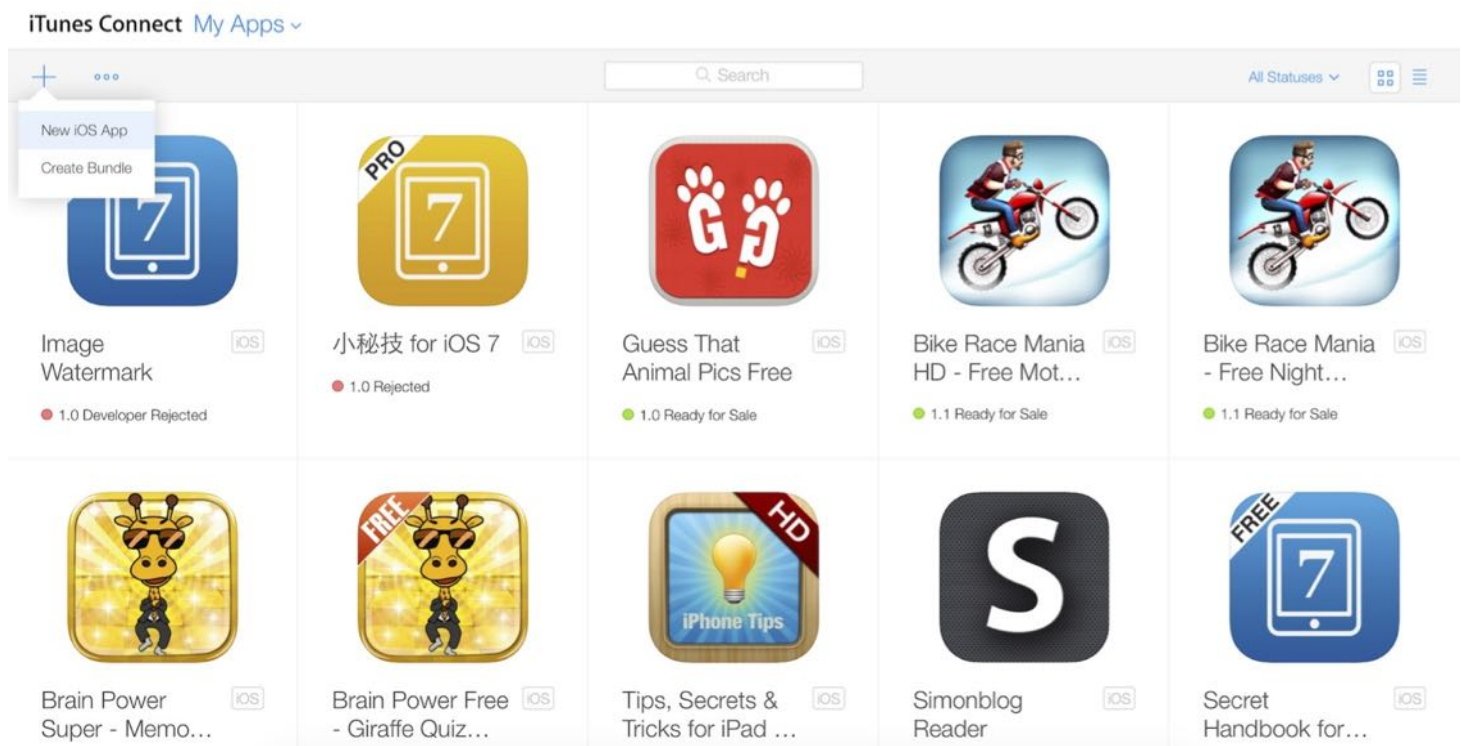
In this chapter, I will walk you through the beta test process using TestFlight. In general, we will go through the below tasks to distribute an app for beta testing:

- Create an app record on iTunes Connect.
- Update the build string.
- Archive and validate your app.

- Upload your app to iTunes Connect.
- Manage beta testing in iTunes Connect.

Creating an App Record on iTunes Connect

Firstly, you need an app record on iTunes Connect before you can upload an app. iTunes Connect is a web-based application for iOS developers to manage their apps sold on App Store. If you have enrolled in the iOS Developer Program, you can access iTunes Connect at <http://itunesconnect.apple.com>. Once signed into iTunes Connect, select **My Apps** and the **+** icon to create a new iOS app.



You'll be prompted to complete the following information:

- Platform - choose app platform, which is iOS.
- App name - your app name appears on App Store.
- Primary language - the primary language of your app such as English.
- Bundle ID - the bundle ID that was created in the previous chapter.
- SKU - stands for Stock Keeping Unit. It can be anything you like. For example, your app

name is "Awesome Food App". You can use "awesome_food_app" as the SKU. You can use letters, numbers, hyphens, periods and underscores except space.

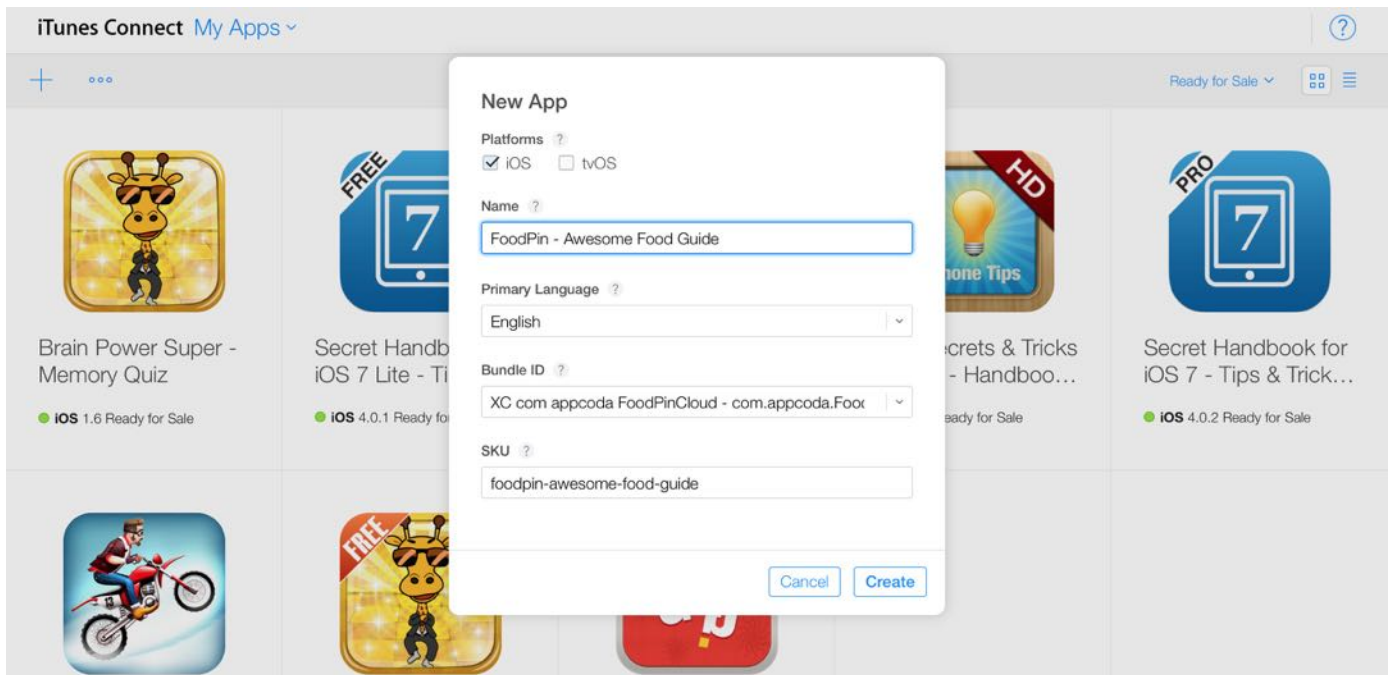


Figure 27-2. Fill in your app information

Once you click the Create button, you'll proceed to another screen to fill in the details of your app.

App Information

For the app information section, it displays the app details that you have just filled in. Optionally, you can provide the app information in other languages. Simply click the *English (U.S.)* button to choose the language. There is one option you have to set - choose your app's primary category. This is the category your app will be listed on the app store. Pick the category that best fits the app. For example, you can choose the *Food & Drink* category for the FoodPin app.

Pricing and Availability

In the side menu, you should also see the Pricing and Availability section. This is where you set

the price of your app. By default, your app will be made available globally. In case you want to limit it to certain countries, you can click the *Edit* button under the availability section, and choose your preferred countries. Remember to save the changes before you go to the next section.

Prepare for Submission

Other than the basic information, you will need to provide additional information such as screenshots, app description, app icon, contact information, etc.

Choose the *Prepare for Submission* option in the side menu to get started.

1. App Preview and Screenshots

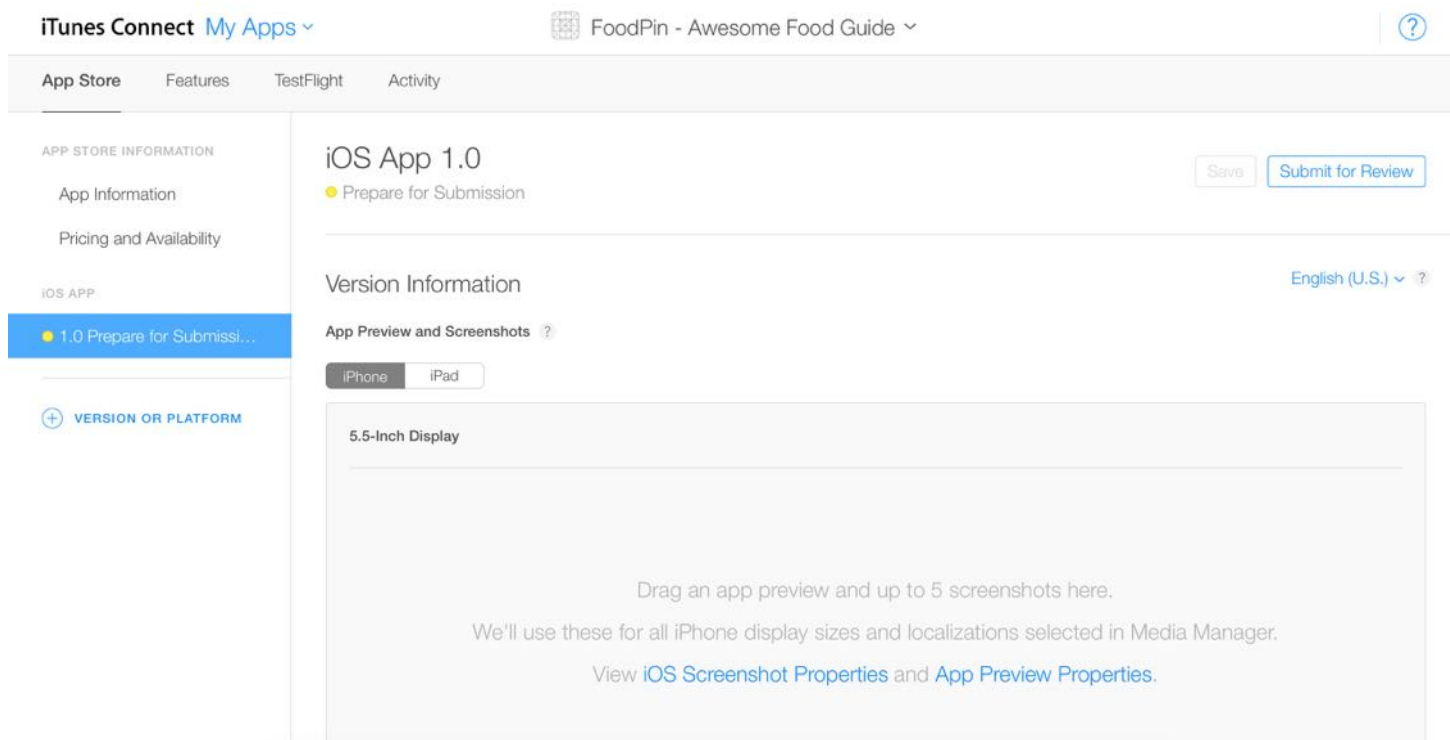


Figure 27-3. Prepare the app screenshots

These are the preview screens for your app. For the preview screens, you need to provide at least one screenshot for 5.5-inch (1242x2208 pixels) devices. iTunes Connect then automatically generates the screenshots for all iPhone display sizes.

To add the screenshots, you can click *Choose File* or simply drag the file to the box. Optionally, you can incorporate an app video in the preview.

Note: You can further refer to Apple's iTunes Connect Developer guide for details (<https://developer.apple.com/library/ios/documentation/LanguagesUtilities/Concepts>)

2. App Description and URL

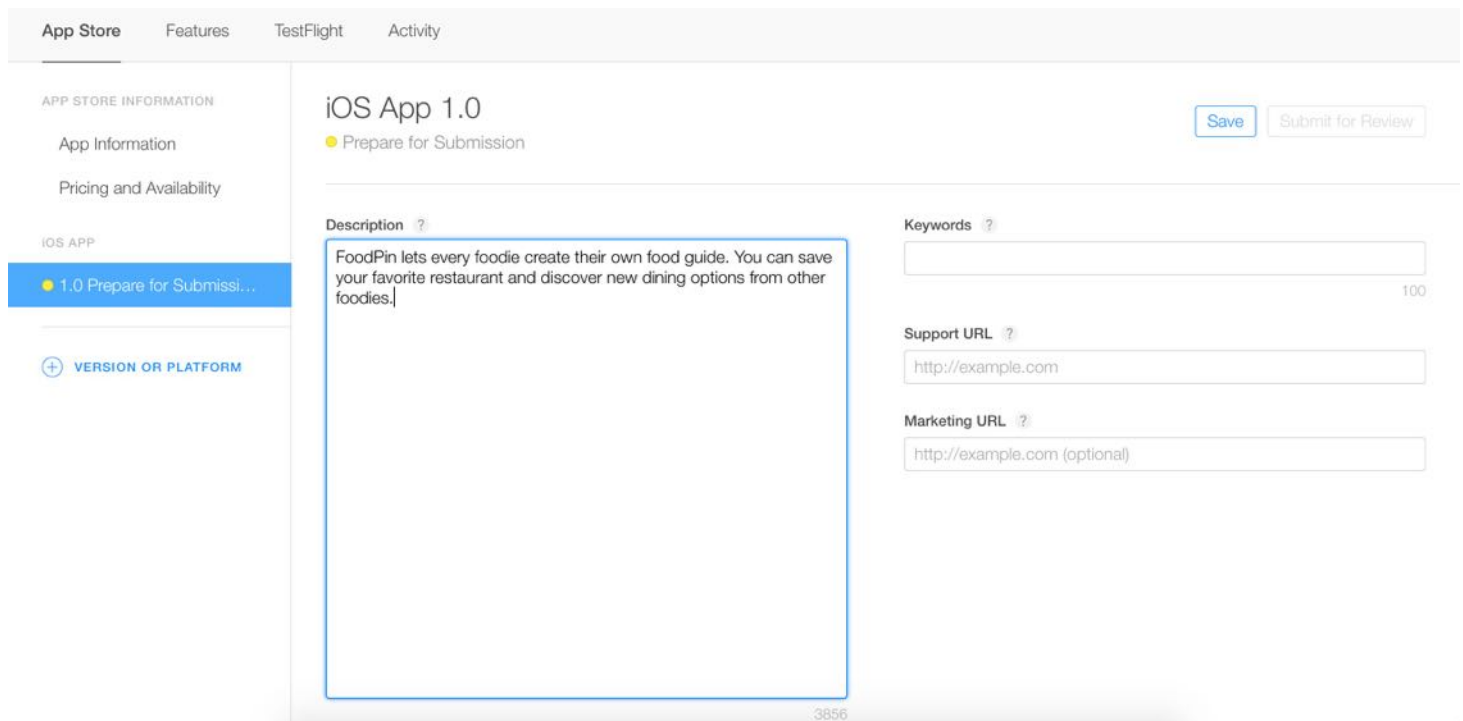


Figure 27-4. Your app description

Next, fill in your app description and at least one keyword that describes your app. If you have more than one keyword, separate them by comma. For example, "food,restaurant,recipe". Keywords are referenced by App Store for app search. It's one of the most important elements impacting your app download. You may have heard of App Store Optimization (ASO). Keyword optimization is a part of ASO. I will not go into keyword optimization here. If you want to learn more about keyword optimization, you can refer to [this article](#) or just google ASO.

The support URL is mandatory. You can fill in the URL of your website or blog. If you don't

have one, register a website at wordpress.com.

3. General App Information

You can skip the Build section and go straight to the General App Information section. This section is all about the general information of your app. You need to upload the app icon. Remember the app icon (1024x1024 pixels) must be not contained any transparency element. Here is a sample of the app icon.



Figure 27-5. Sample large app icon

The icon has a square shape. After you upload the app icon to iTunes Connect, the icon will be converted to appear with rounded corners. Next, fill in the version number (e.g. 1.0) and pick a category that best describes your app.

You need to give a rating for your app. Just click the Edit button next to Rating and complete the form. iTunes Connect generates a rating for your app based on your answers.

For the copyright field, you can just fill in your name or company preceded by the year the rights were obtained (e.g. 2016 AppCoda Limited). If you want to display addition information with your app on Korean App Store, you need to provide the Trade Representative Contact Information.

App Store Features TestFlight Activity

APP STORE INFORMATION

- App Information
- Pricing and Availability

IOS APP


- 1.0 Prepare for Submissi...

+ VERSION OR PLATFORM

iOS App 1.0 Save Submit for Review

● Prepare for Submission

General App Information

App Icon ? 

Version ?

Rating [Edit](#)
Ages 4+
[Additional Ratings](#)

Copyright ?

Trade Representative Contact Information ?
 Display Trade Representative Contact Information on the Korean App Store.

Leung Wai Ng

Figure 27-6. Sample app information

4. App Review Information

Simply fill in your contact information in this section. The demo account field is optional. It is for those apps that require login.

5. Version Release

You're allowed to release your app automatically or manually right once it has been approved by App Review. Just set it to `Automatically release this version`. Finally click the `Save` button on the top right corner to save the changes.

If you didn't leave out any required information, the `Submit for Review` button should be enabled. That means your app record was successfully created on iTunes Connect.

Update Your Build String

Now go back to Xcode. We're going to build the app and upload to iTunes Connect. But before

that, review your project and make sure the version number matches what you entered in iTunes Connect. In the project navigator, select the project and the target to display the project editor. Under the General tab, review the version field under the Identity section. As this is the first build, set the Build field to 1.

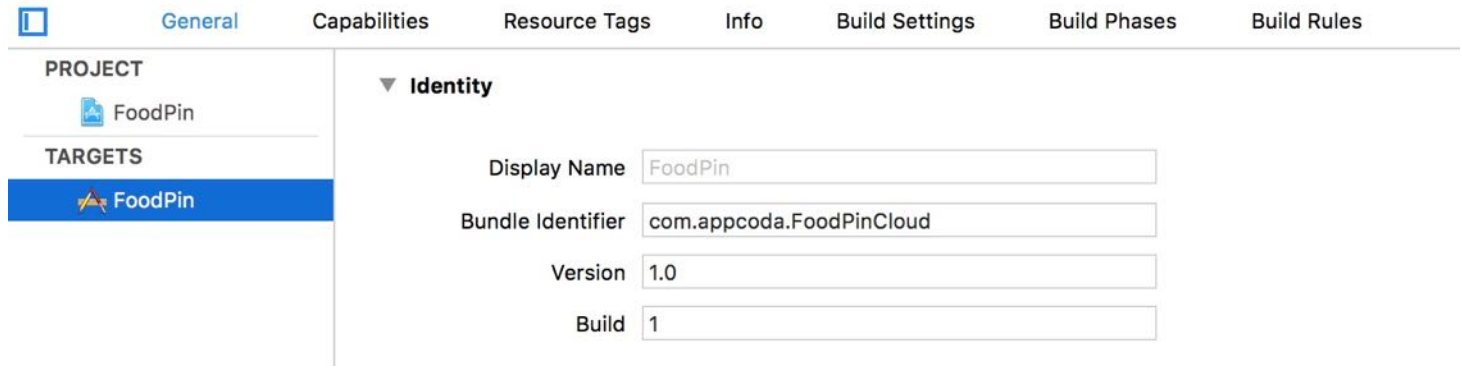


Figure 27-7. Review the version and build in project editor

Prepare Your App Icon and Launch Image

Before submitting your app, make sure you include the app icon and launch screen. The app icon is managed by the asset catalog. You should find the `AppIcon` set in `Asset.xcassets`. To add an icon to the set, select an app icon in the Finder and drag it to the appropriate image well in the set viewer. You will need to provide various sizes of app icons to fit for different devices. Just follow the instructions accordingly. For example, your Spotlight icon should be in 80x80 pixels (40pt x 2).

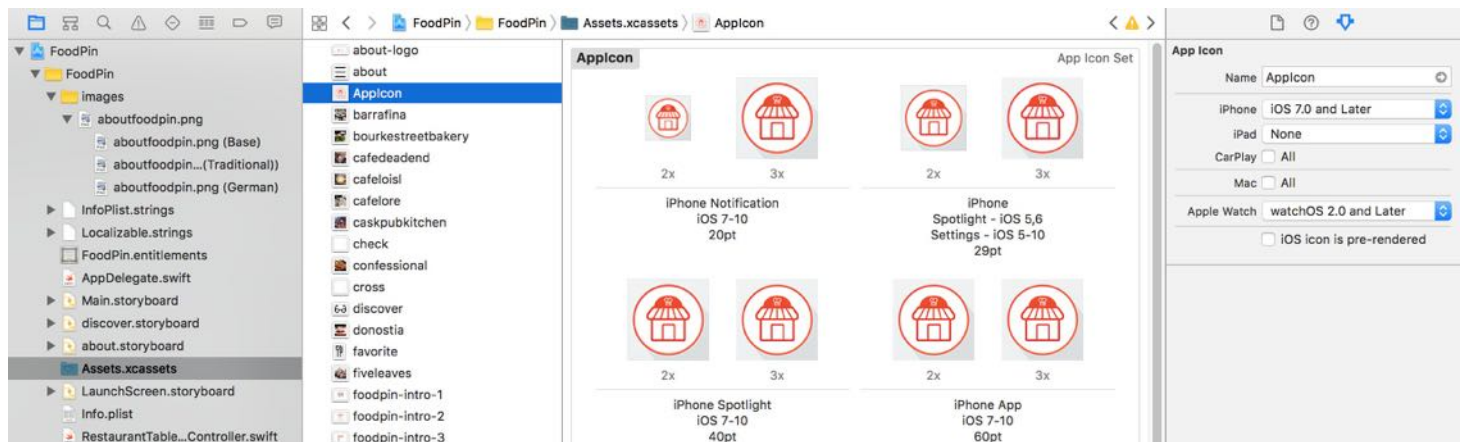


Figure 27-8. A sample AppIcon set

We have briefly mentioned about the launch screen at the very beginning of the book. Xcode 8 allows developers to use a storyboard or Interface Builder file to design a launch screen. In the FoodPin project, you should have a `LaunchScreen.storyboard` file, which is set as the default launch screen file in the project editor.

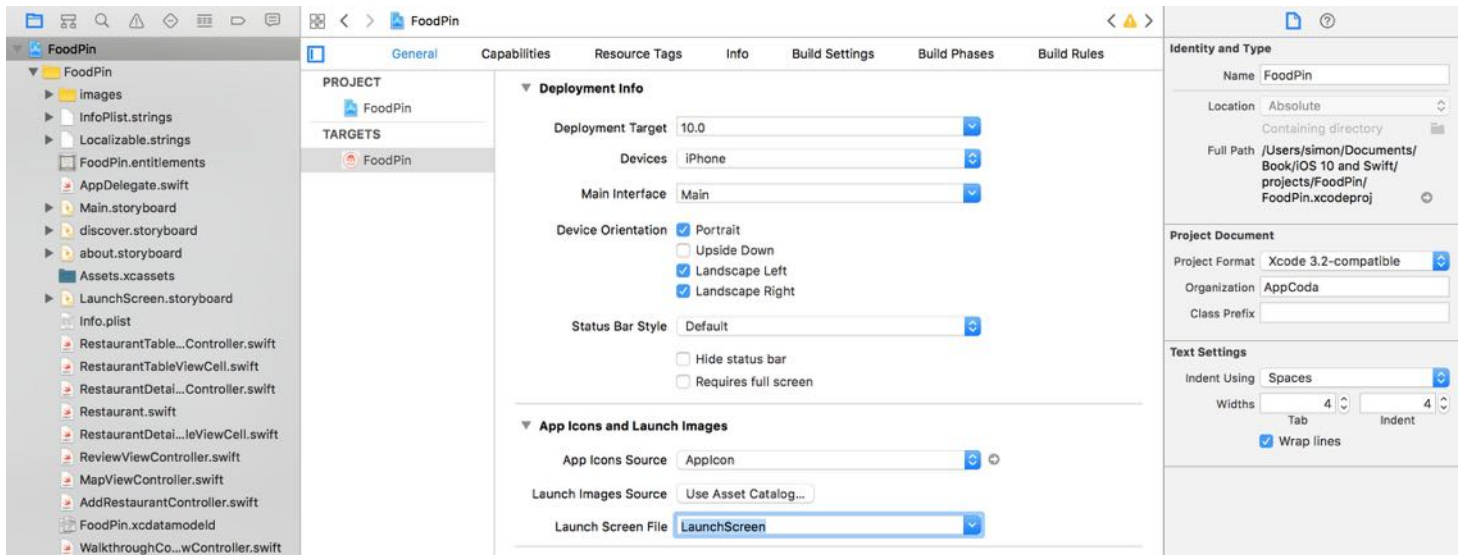


Figure 27-9. LaunchScreen is set as the default launch screen file

Quick tip: If you do not have a launch screen, you can right click the FoodPin folder (or your project folder) and select `New File...`. Just use the Launch Screen template under User Interface to create one.

In the `LaunchScreen.storyboard` file, you will find an empty view controller. When the app is launched, this view controller will be loaded as the startup screen. You're free to design its interface just like what you did for other view controllers. For reference, you can refer to [Apple's Human Interface Guideline](#) about the best practice of a launch screen. Figure 27-10 displays a sample launch screen design.

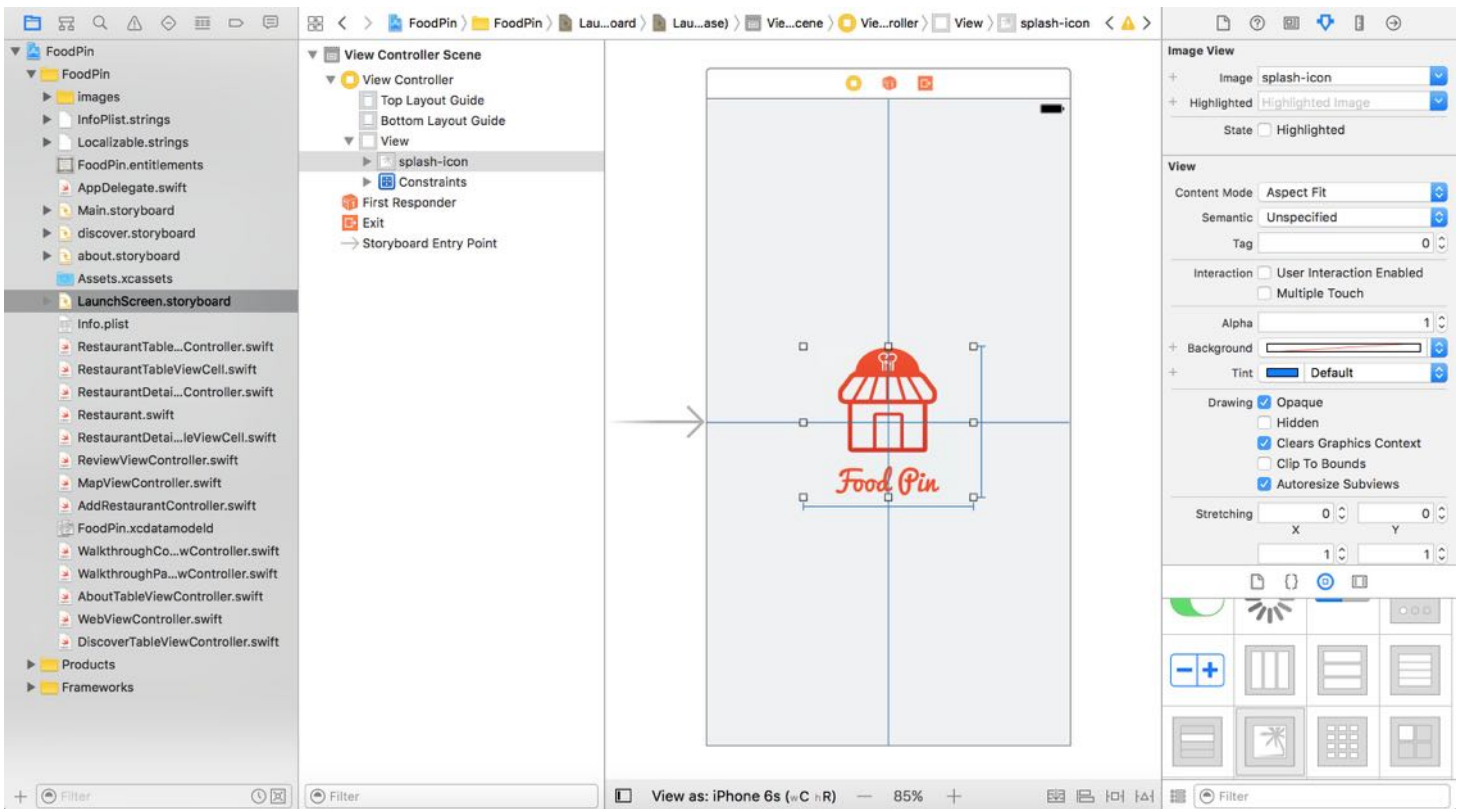


Figure 27-10. LaunchScreen is set as the default launch screen file

Archiving and Validating Your App

Before uploading an app to iTunes Connect, you will need to create an app archive. It's pretty easy to archive your app in Xcode 8. First, review the Archive scheme settings and ensure the build configuration sets to Release (instead of Debug). Go up to the Xcode menu. Choose *Product > Scheme > Edit Scheme*. Select Archive scheme and review the Build Configuration setting. The option should be set to *Release*.

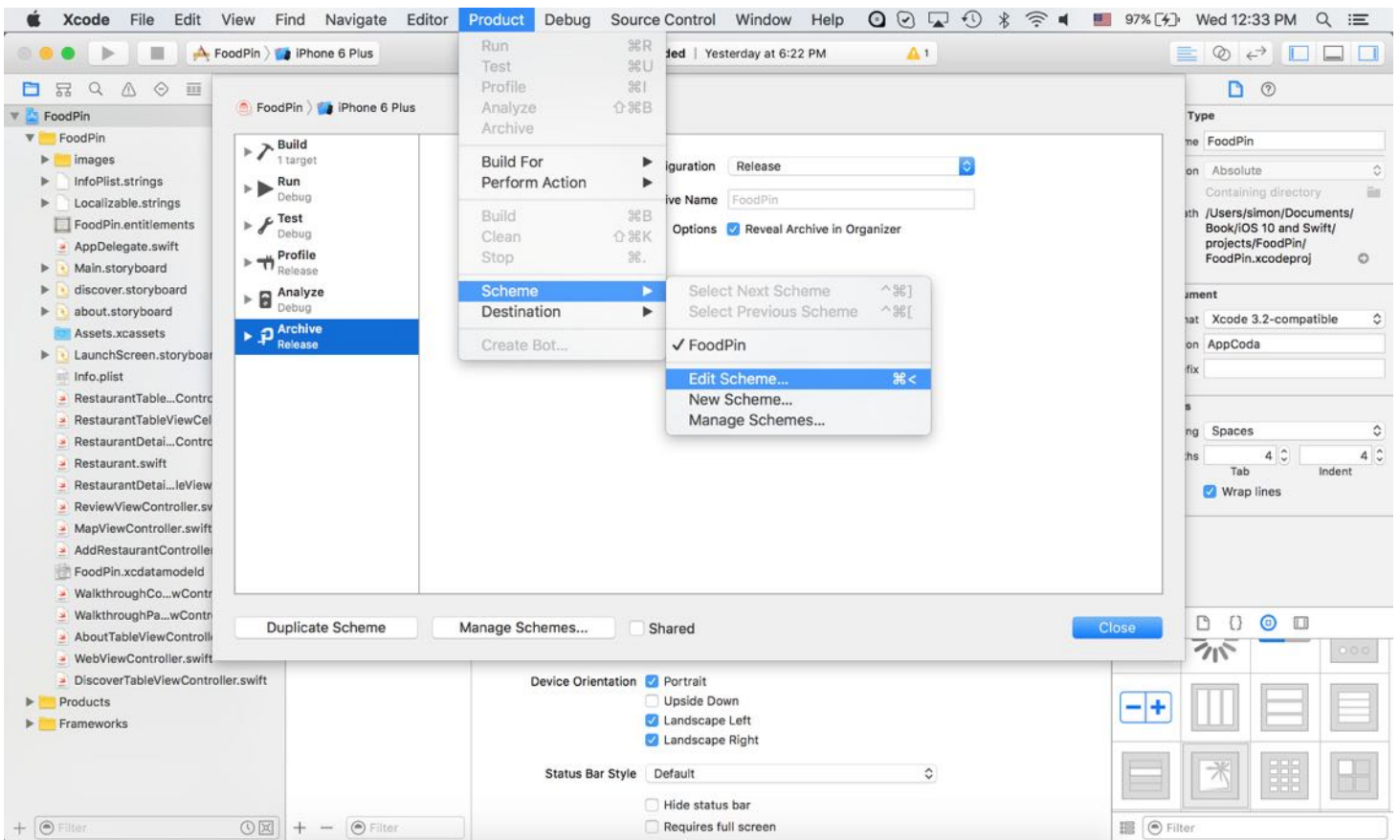


Figure 27-11. Review the Archive scheme setting

Now you're ready to archive your app. The Archive feature is disabled if you're using a simulator. So first select Generic iOS Device or your device name (if you have connected a device to your Mac) from the Scheme pop-up menu. Then go up to the Xcode menu and choose Product > Archive.

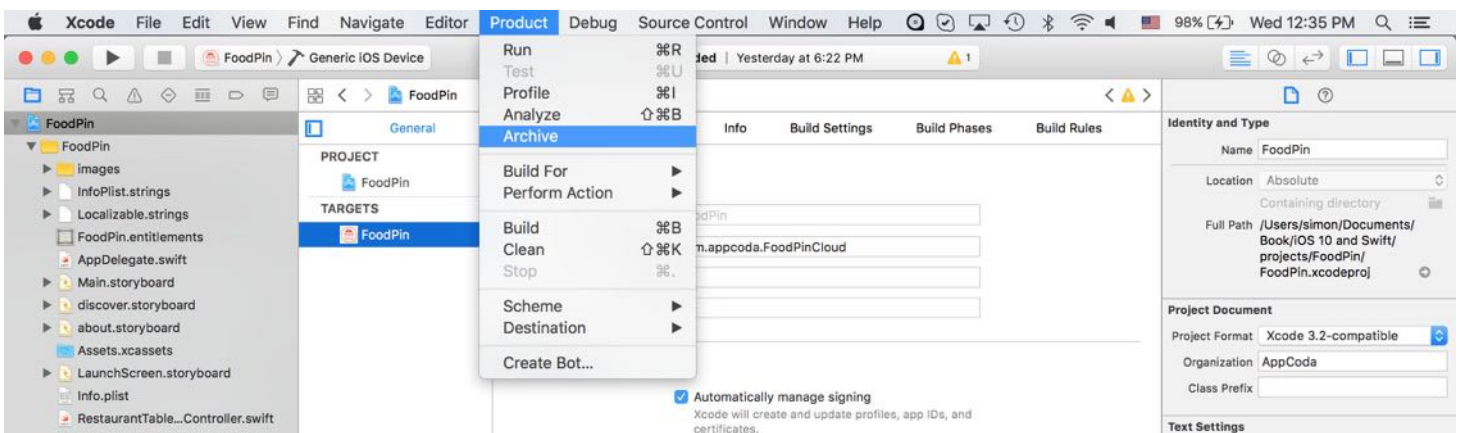


Figure 27-12. Archive your app

After archiving, your archive will appear in the Organizer. It's ready to upload to iTunes Connect. But it's best to go through the validation process to see if there are any issues. Just click the `Validate...` button and then select your developer account. Xcode will then validate your archive.



Figure 27-13. Your app archive appears in Organizer

Upload Your App to iTunes Connect

If the validation is successful, you can click the `Upload to App Store...` button to upload the archive to iTunes Connect.

You will be prompted to provide a development team account during the upload process. Just choose the appropriate account and proceed with the upload. The whole process will take several minutes before you see the "Upload Success" message.



Figure 27-14. Upload your app archive to App Store

Manage Internal Testing

Now that you have uploaded your build to iTunes Connect, let's see how you can roll out your app for internal testing.

Go back to <https://itunesconnect.apple.com>. Select My Apps and then your app. In the menu, choose *TestFlight*. Here you can fill in your test information including feedback email, privacy policy and marketing URL. Note that these information can be viewed by your testers.

Once you fill in the required information, select *Internal Testing* from the side menu. Click the *Select Version to Test* button and choose the version of the build for internal testing.

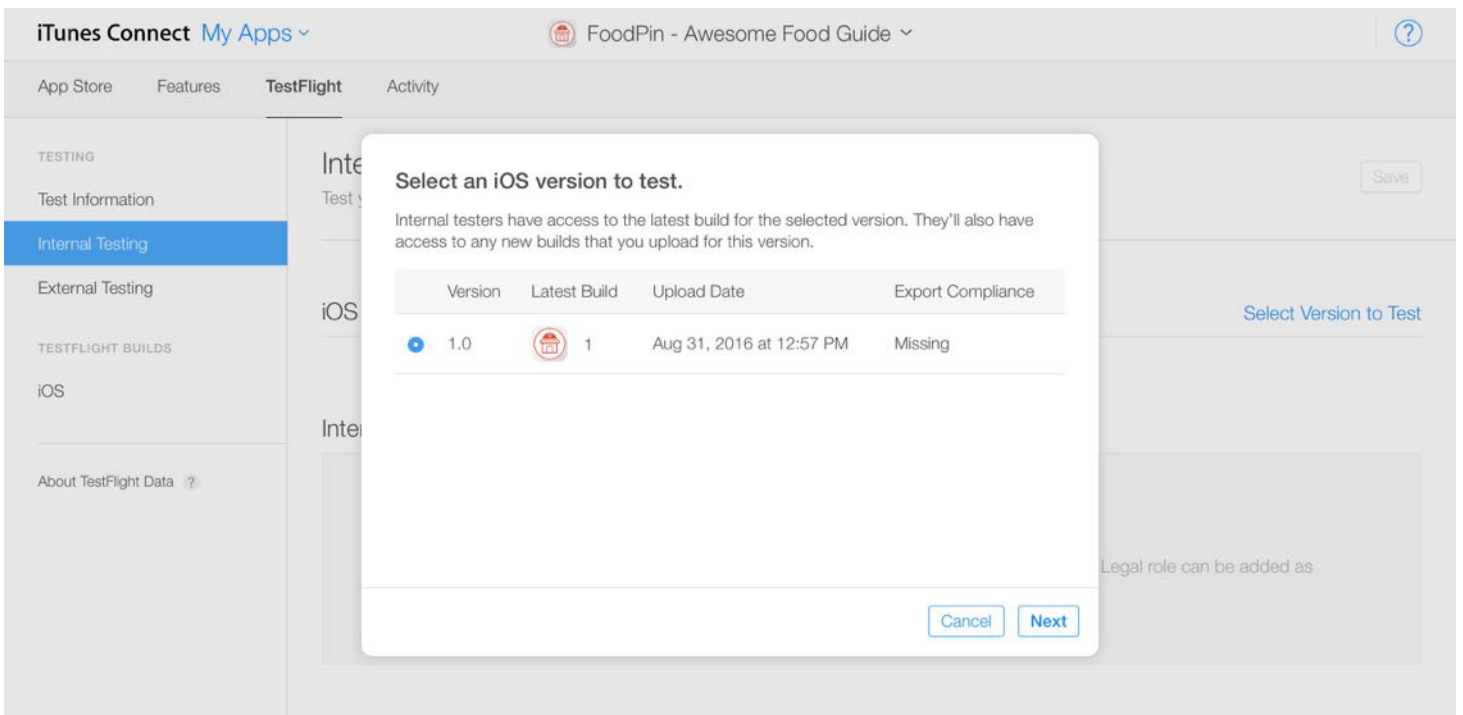


Figure 27-15. Choose a version to test

When you hit *Next*, you will be asked if your app is designed to use cryptography. For this app, answer *No* to proceed.

The selected version should now be ready to test. The next thing you have to do is add at least one internal test. Click the + button and select any of the iTunes Connect users as your internal testers.

Finally hit the *Start Testing* button to send email invitations to the testers.

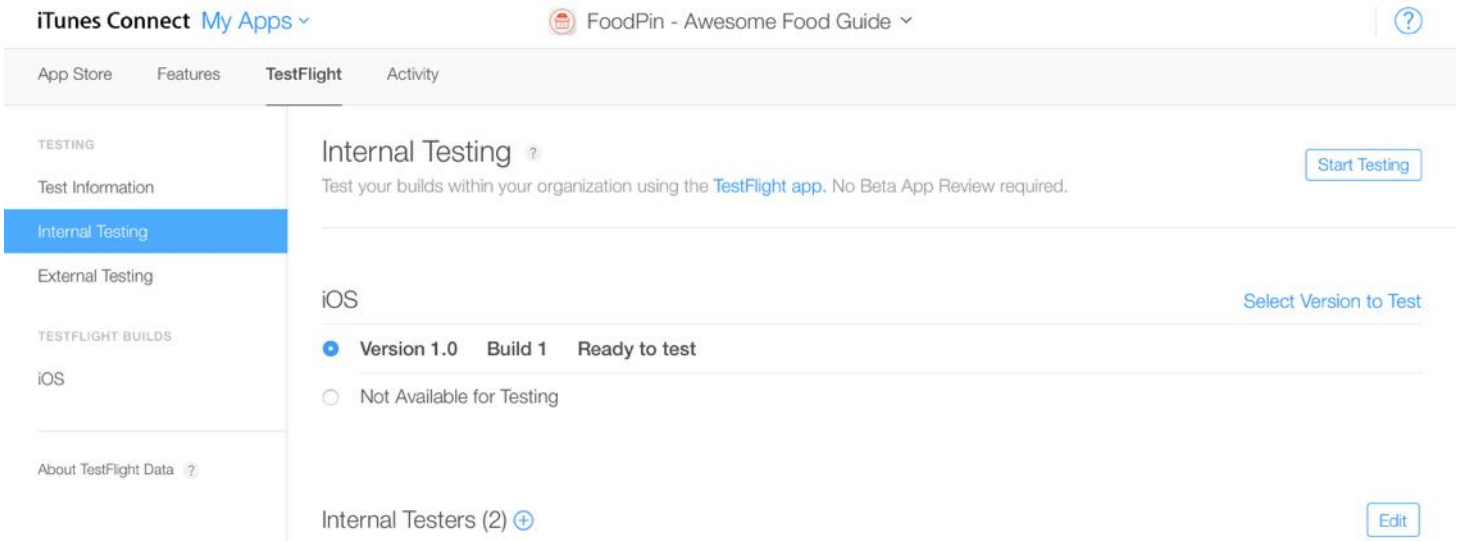


Figure 27-16. Click Start Testing to notify your internal testers

Note: Here internal users refer to those who are part of your iTunes Connect team with the Admin, Legal, or Technical role. If you want to add more users or change their roles, select iTunes Connect > Users and Roles, and then click + to add the user.

When a tester receives the email notification, he/she just needs to click the Open TestFlight button. iOS automatically opens the TestFlight app. The tester can then install your app for internal testing.

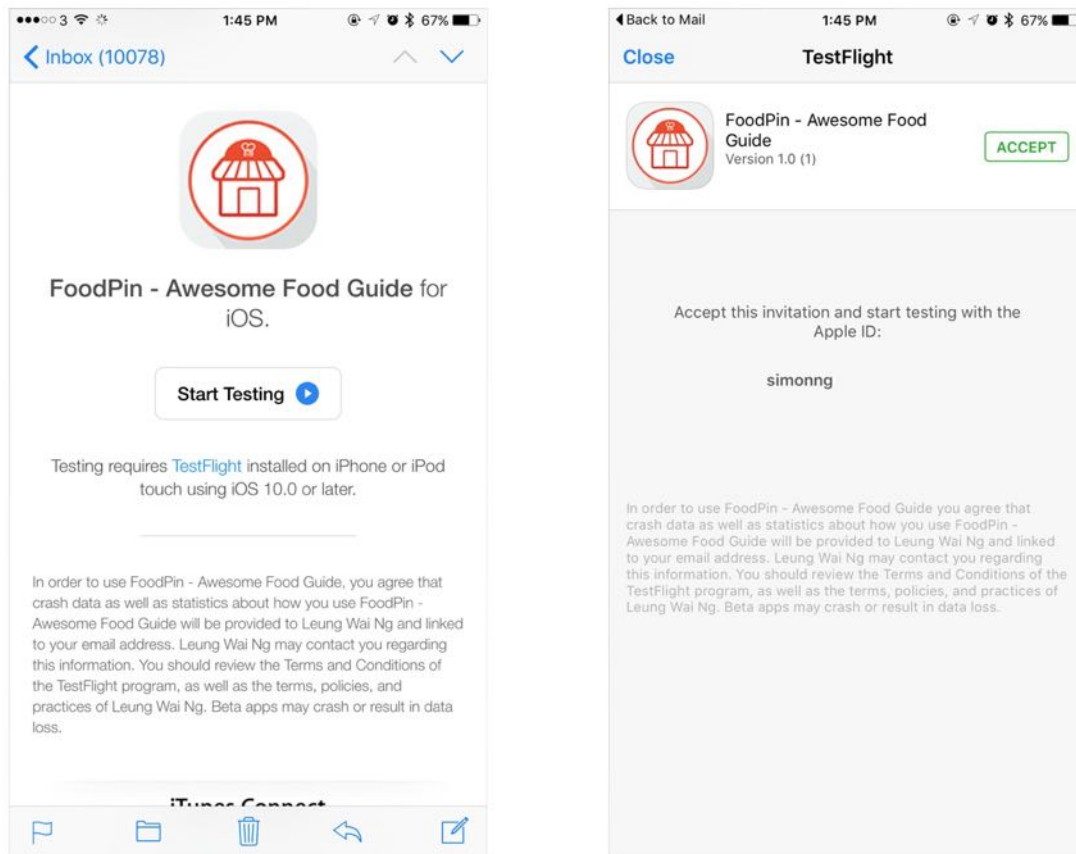


Figure 27-17. Send invitation to your testers

If your tester does not have the TestFlight app installed, he/she will need to install it first. For any future update of your beta app, your internal testers will always get the most recent build you uploaded.

Manage Beta Testing with External Users

You're allowed to invite up to 25 internal users for testing. Once your app lives up to the expectation of your users, you can invite more users to test out your app. TestFlight lets you invite up to 2,000 users to be external testers. All you need is an email address of each of the testers so that you can send them invite.

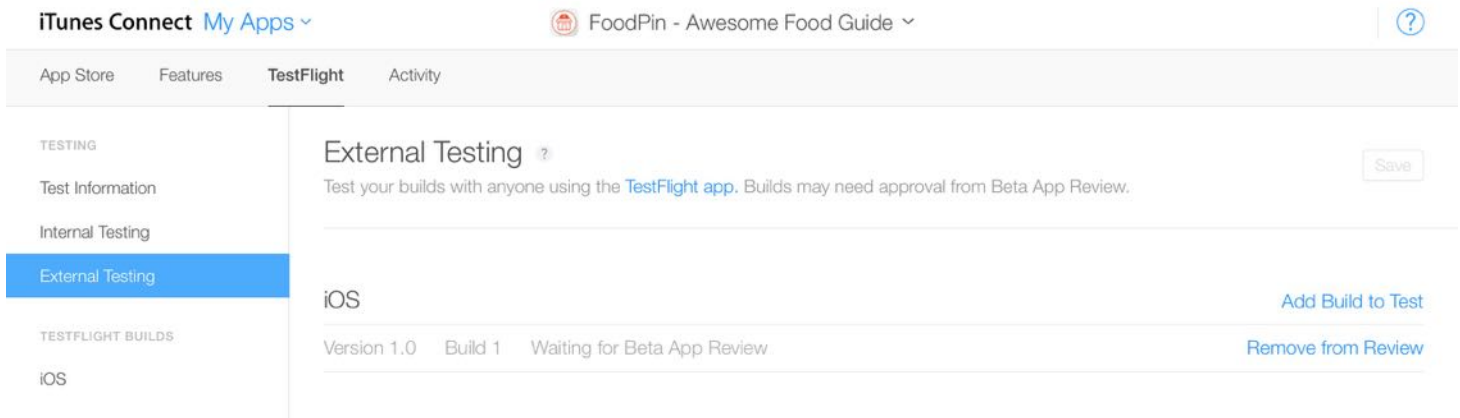


Figure 27-18. Submitting your app for beta app review

There is a catch, however. Your app must be approved by Beta App Review before you can send out your invitations. To submit the app for Beta App Review, go to *External Testing*. Add the build for beta app review. You will need to provide your beta app description, review notes and contact information. Finally you can click the *Submit* button to submit your app for review.

Once your app is submitted for review, the build status will change to *Waiting for Beta App Review*. Normally it'll take less than two days before your app is approved.

After Apple approves your beta app, you can add your external testers and notify them for beta testing.

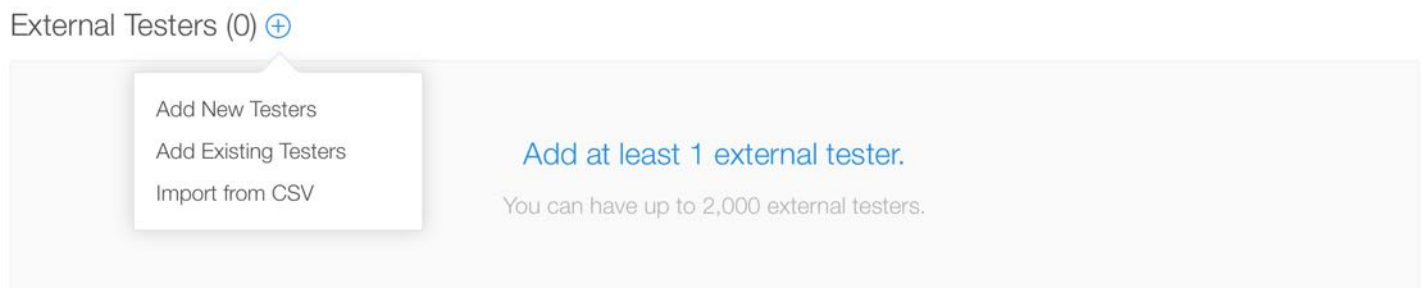


Figure 27-19. Adding external testers

Summary

TestFlight provides us with a powerful tool to easily beta test our apps. In this chapter, I have walked you through the basics of TestFlight Beta Testing. If you're building your next app, use the tool to invite your friends and beta users to test out your app before the official release. This is an important step to build a high quality app.

Chapter 28

Submit Your App to App Store



Don't let people tell you your ideas won't work. If you're passionate about an idea that's stuck in your head, find a way to build it so you can prove to yourself that it doesn't work.

- Dennis Crowley, FourSquare

Congratulations! You have probably worked weeks or months before coming to the last step of app development. After beta testing and numerous bug fixing, your app is finally ready for submission.

You already uploaded your app binary to iTunes Connect in the previous chapter, so it is quite

straightforward to submit your app to App Store. Once you submit your app, it will be reviewed by Apple's App Review team before publishing it onto App Store. For a first-time app developer, submitting an app to the app store can be a nightmare. You may need to submit your app multiple times before Apple approves it.

In this chapter, I will walk you through the app submission process and give you some guidelines to minimize the possibility of app rejections.

Get Prepared and Well-Tested

From a developer's point of view, the app approval process is a black box. I still remembered how frustrated I felt when I submitted my first app to App Store and the app was rejected by Apple. Since then, I've learned that there are a few things you can do to minimize the likelihood of rejection.

1. **Test your app thoroughly** - Have you tested your app before submission? If you followed the previous two chapters, you should have tested your app on a real device and invited a group of beta testers to try it out. But let me emphasize again the importance of testing. You shouldn't just test your app using the built-in simulator - you have to test it at least on one real iOS device. If the app is a universal app, remember to test it on both iPhone and iPad. If your app integrates with other services like iCloud, make sure you test it on both cellular and Wi-Fi networks. Whenever possible, follow the instructions of the previous chapter to beta test your app. If you submit an app that can't be run properly or crashes unexpectedly, Apple will reject it. Also, remember to test your app under both normal and exceptional scenarios. Let's say, your app requires Internet connection to work properly. What happens if the device has no signal? Will the app crash? Or it just displays a nice error message? This is something you have to take care of, and make sure your app works in all situations.
2. **Follow the App Store review guidelines** - Even though we said the app store process is like a blackbox, Apple does provide a review guideline for developers' reference. You can access the guideline at <https://developer.apple.com/appstore/resources/approval/guidelines.html>. Although I recommend you read it thoroughly, here are a few main points:

- Apps that crash will be rejected
- Apps that exhibit bugs will be rejected
- Apps that do not perform as advertised by the developer will be rejected
- Apps that include undocumented or hidden features inconsistent with the description of the App will be rejected
- Apps that use non-public APIs will be rejected

You should also take note when your app title includes any of Apple's trademarks. I once created a book app about iOS and named it *iPhone Handbook*. The app was rejected immediately by Apple. Later I changed the app title to *Handbook for iPhone* and the app was approved. Apple allows the use of Apple trademarked phrases and words when used with a referential phrase such as *for*.

For details, you can check out Apple's guideline at:

<http://www.apple.com/legal/trademark/guidelinesfor3rdparties.html>.

1. **Meet the User Interface requirement** - the app UI should be clean and user friendly. Otherwise, Apple may reject your app due to substandard UI design. You can refer to [UI Design Dos and Don'ts](#) for details.
2. **Broken links** - All links embedded in your app must be functional. No broken links are allowed.
3. **Finalize all images and text** - All your images and text in your app should be finalized before submitting your app for review. Your app will be rejected if it contains any placeholder content.

For more information about the common pitfalls that cause app rejections, you can refer to <https://developer.apple.com/app-store/review/rejections/>. Furthermore, don't forget to check out the App Store Review guidelines (<https://developer.apple.com/app-store/review/guidelines/>).

Submit Your App to App Store

If you haven't read the previous chapter, please go back to read about the procedures of creating an app record on iTunes Connect and uploading an app archive.

Now go to <http://itunesconnect.apple.com> and select My Apps. Then select the FoodPin app. Assuming you have completed all of the required information in App Information, click *Pricing and Availability* in the side menu. If you have set your price before, this is a good chance to review your settings again. The latest version of iTunes Connect provides a new feature for developers to plan a price change. Let's say, you want to offer your app for free when it is first released. After a certain period of time, you change it to a paid app. The *Plan a Price Change* option can help you plan ahead for future price changes.

The screenshot shows the 'Pricing and Availability' section of the iTunes Connect interface. The left sidebar contains navigation options: 'APP STORE INFORMATION' (with sub-items 'App Information' and 'Pricing and Availability'), 'iOS APP' (with a sub-item '1.0 Prepare for Submiss...'), and 'VERSION OR PLATFORM'. The main content area is titled 'Pricing and Availability' and includes a 'Saved' button. Below the title is a 'Price Schedule' section with a link for 'All Prices and Currencies'. A table displays the price schedule with columns for Price, Start Date, and End Date. The table contains two rows: one for a free tier (HKD 0) and one for a paid tier (HKD 8). A 'Plan a Price Change' link is located below the table.

Price ?	Start Date ?	End Date ?
HKD 0 (Free)	Aug 31, 2016	Sep 29, 2016
HKD 8 (Tier 1)	Sep 29, 2016	No End Date

Figure 28-1. Fill in the availability date and price tier

Next, go to the *Prepare for Submission* option. Scroll down to the Build section. Click + to add a build and select the one you want to submit.

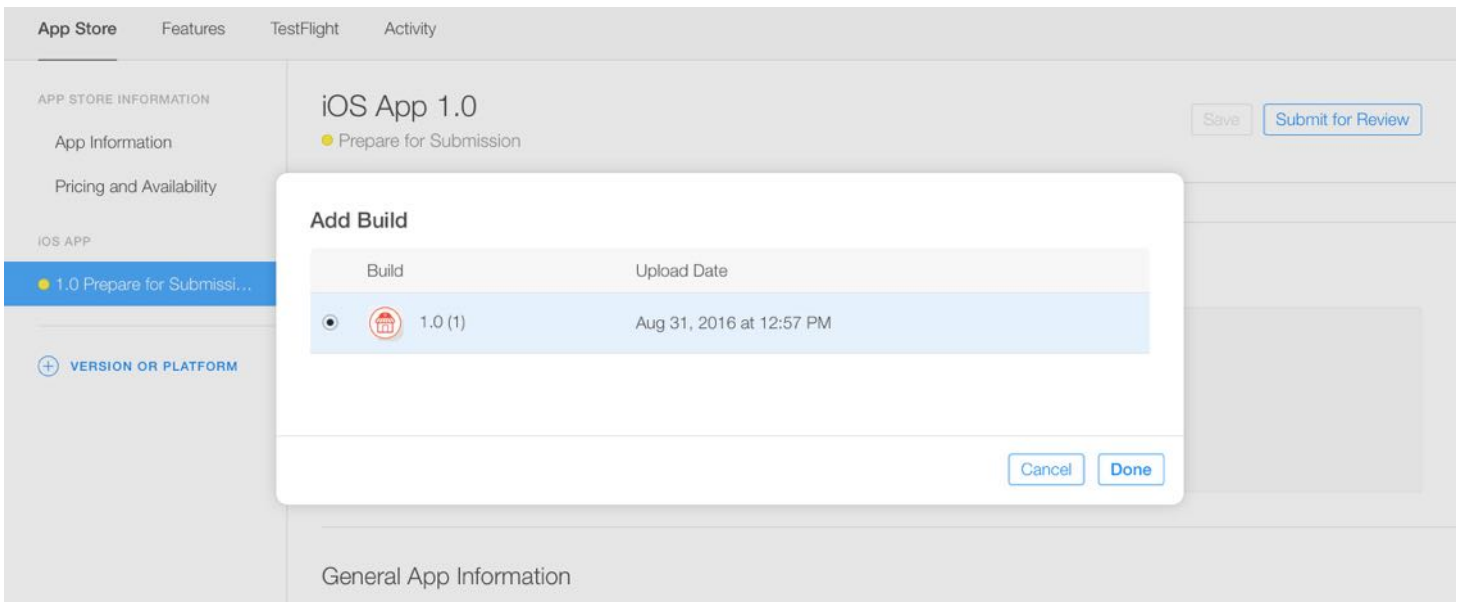


Figure 28-2. Pick a build for submission

Finally, save the changes and click the `Submit for Review` button to submit your app. Once you fill in the Export Compliance, Content Rights, and Advertising Identifier, your app is then submitted for review.

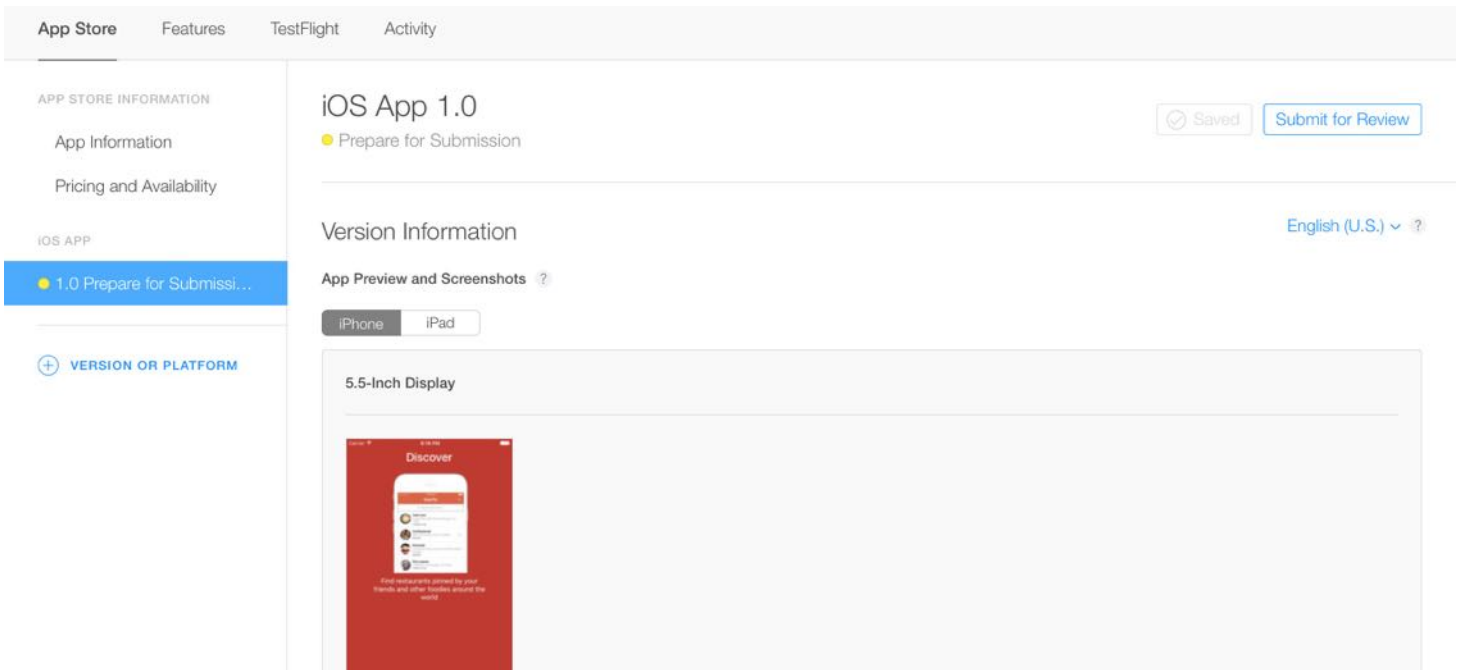


Figure 28-3. Click Submit for Review button to submit your app

After you successfully submit your app, the status will be changed to Waiting for Review.

Now what's next? That's it. There are not much thing you can do right now, but just wait. In the past, you would have to wait for around 7 days before your app was approved (or rejected). Starting from May 2016, Apple started to cut the approval time from more than a week to less than two days.

On average, 50% of apps are reviewed in 24 hours and over 90% are reviewed in 48 hours.

<https://developer.apple.com/support/app-review/>

So be patient and wait. You will receive an email notification when your app is approved or rejected.

Summary

Once again, congratulations! You have built a real app and learned how to submit it to the App Store. I hope your app will be approved by Apple the first time you submit it. Even if it's rejected, don't get frustrated - many iOS developers share the same experience. Just fix the issue and re-submit your app again.

This isn't the end. I still got a couple of things to share in the next chapter.

Chapter 29

Adopting 3D Touch



With the iPhone 6s and 6s Plus, Apple introduced us an entirely new way to interact with our phones known as 3D Touch. It literally adds a new dimension to the user interface and offers a new kind of user experience. Not only can it sense your touch, the new iPhone can now sense how much pressure you apply to the display.

With 3D Touch, you now have three new ways to interact with the iPhones: Quick Actions, Peek, and Pop. Quick actions are essentially shortcuts for your applications. When you press an app icon a little harder, it shows a set of quick actions, each of which allows you to jump straight to a particular part of an app. It simply saves you a few "taps".

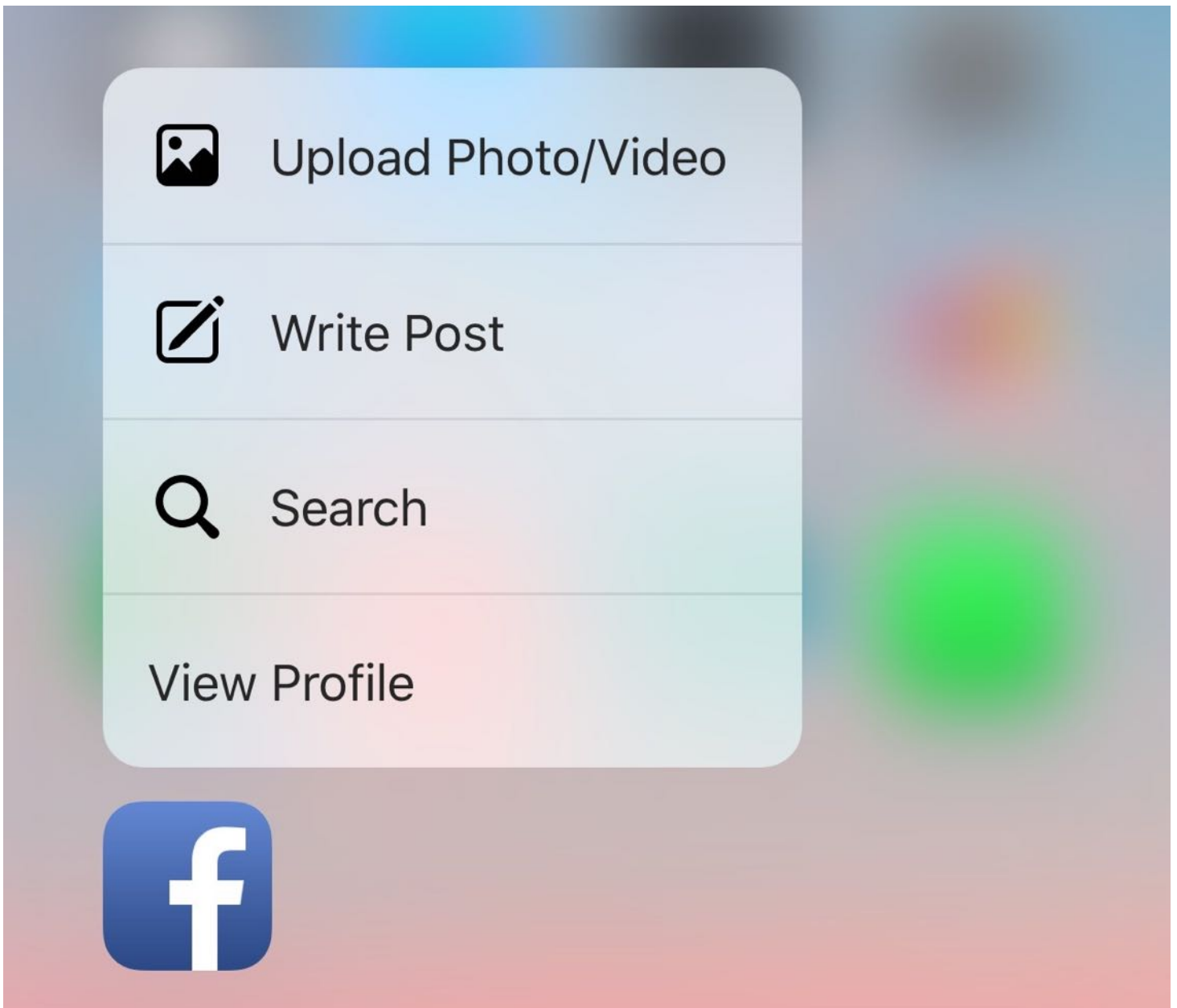


Figure 29-1. Sample Quick Actions

Peep and Pop purposely want to give users a quicker access of the app's contents. Take the built-in Photos app as an example. When you press a little harder on a photo, the app "Peeks" the photo in a pop-up. If the preview is good enough for you, simply release your finger and you're back to the photo album. But if you want more than a preview, just press a bit harder to "Pop" into a full view.

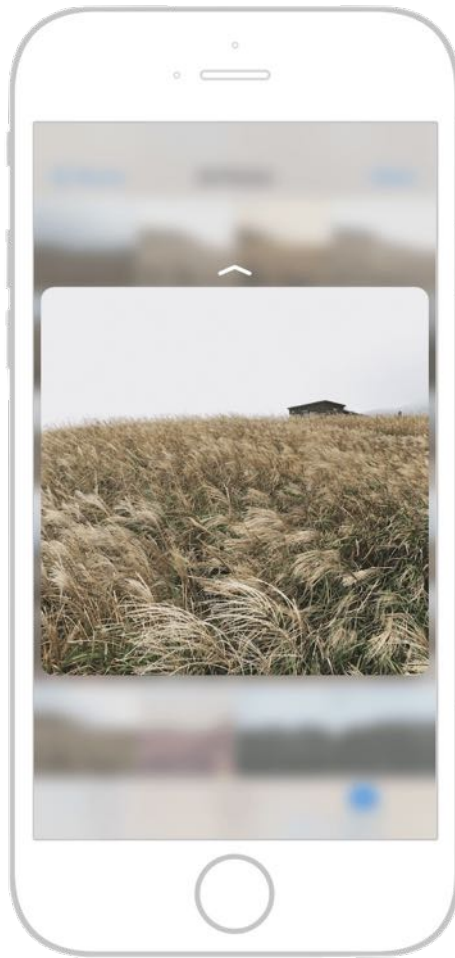


Figure 29-2. Peek and Pop in Photos app

Starting from iOS 9, Apple provides a new set of APIs for developers to work with 3D Touch. In this chapter, I will go through some of the new APIs with you. More specifically, we will add Quick Actions, Peek, and Pop features to the FoodPin app.

Home Screen Quick Actions

First, let's talk about Quick Actions. Apple offers two types of quick actions: *static* and *dynamic*. Static quick actions are hardcoded in the `Info.plist` file. Once the user installs the app, the quick actions will be accessible, even before the first launch of the app. As the name suggests, dynamic quick actions are dynamic in nature. The app creates and updates the quick actions at runtime. Take the News app as an example. Its quick actions show some of the most frequently accessed channels. They must be dynamic because these channels will change over time.

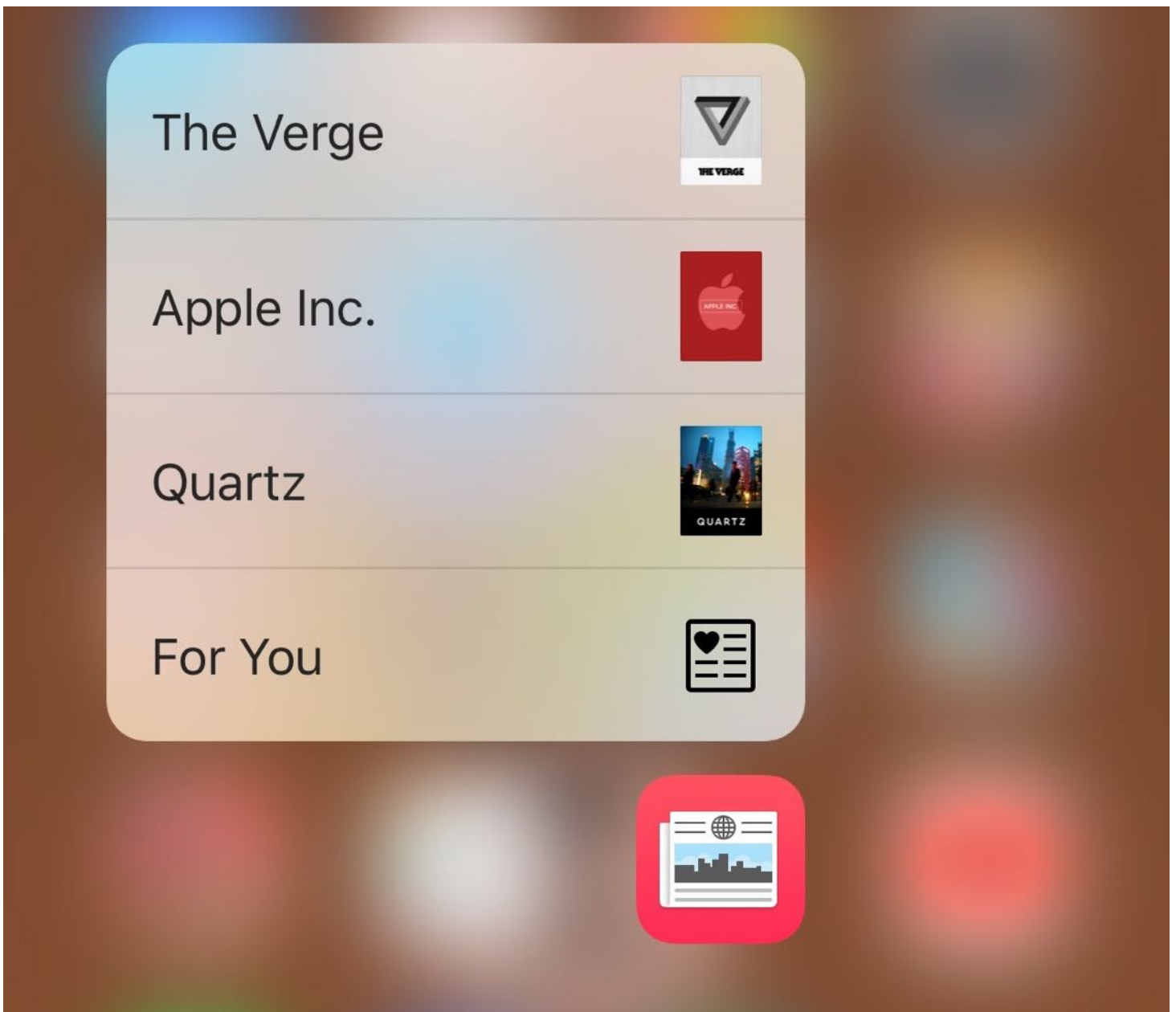


Figure 29-3. Quick actions are changeable in News app

But one thing they have in common is that you can create at most 4 quick actions, no matter you're using static or dynamic quick actions.

It's pretty simple to create static quick actions. All you need to do is edit the `Info.plist` file and add a `UIApplicationShortcutItems` array. Each element of the array is a dictionary containing the following properties:

- `UIApplicationShortcutItemType` (required) - a unique identifier used to identify the quick action. It should be unique across all apps. So a good practice is to prefix the identifier with the app bundle ID (e.g. `com.appcoda.`).
- `UIApplicationShortcutItemTitle` (required) - the name of the quick action visible to the user.
- `UIApplicationShortcutItemSubtitle` (optional) - the subtitle of the quick action. It is an optional string displayed right below the title of the quick action.
- `UIApplicationShortcutItemIconType` (optional) - an optional string to specify the type of an icon from the system library. Refer to [this document](#) for the available icon type.
- `UIApplicationShortcutItemIconFile` (optional) - if you want to use your own icon, specify the icon image to use from the app's bundle. Alternatively, specify the name of the image in an asset catalog. These icons should be square, single colour with sizes 35x35 (1x), 70x70 (2x) and 105x105 (3x).
- `UIApplicationShortcutItemUserInfo` (optional) - an optional dictionary containing some extra information you want to pass. For example, one use for this dictionary is to pass the app version.

If you want to add some static quick actions, here is an example of the

`UIApplicationShortcutItems` array, which creates a "New Restaurant" shortcut:

▼ <code>UIApplicationShortcutItems</code>	▲	Array	(1 item)
▼ Item 0	▲	Dictionary	(3 items)
<code>UIApplicationShortcutItemSubtitle</code>		String	Creates a new restaurant
<code>UIApplicationShortcutItemType</code>		String	<code>com.appcoda.NewRestaurant</code>
<code>UIApplicationShortcutItemTitle</code>		String	New Restaurant

Figure 29-4. Sample Info.plist for static quick actions

Now that you should have some ideas about static quick actions, let's talk about the dynamic ones. As a demo, we will modify the FoodPin project, and add three quick actions to the app:

- **New Restaurant** - go to the New Restaurant screen directly
- **Discover restaurants** - jump right into the Discover tab
- **Show Favorites** - jump right into the Favorites tab

First things first, why do we use dynamic quick actions? A simple answer is that I want to show you how to work with dynamic quick actions. But the actual reason is that I only want to enable these quick actions after the user goes through the walkthrough screens.

To create a quick action programmatically, you just need to instantiate a

`UIApplicationShortcutItem` object with the required properties and then assign it to the `shortcutItems` property of `UIApplication`. Here is an example:

```
let shortcutItem = UIApplicationShortcutItem(type: "com.appcoda.NewRestaurant",
localizedTitle: "New Restaurant", localizedSubtitle: nil, icon:
UIApplicationShortcutIcon(type: .add), userInfo: nil)
UIApplication.shared.shortcutItems = [shortcutItem]
```

The first line of code defines a shortcut item with the quick action type

`com.appcoda.NewRestaurant` and system icon `.Add`. The title of the quick action is set to `New Restaurant`. The second line of code initializes an array with the shortcut item, and set it to the `shortcutItems` property.

Do you still remember how we indicate a user has gone through the walkthrough screens? We set the key named `hasViewedWalkthrough` to `true` into the user defaults, once the user completes the walkthrough. This line of code can be found in the `nextButtonTapped` method of the `walkthroughContentViewController` class:

```
defaults.setBool(true, forKey: "hasViewedWalkthrough")
```

To create the quick actions when the user completes the walkthrough, insert the code after that line of code:

```
// Add Quick Actions
if traitCollection.forceTouchCapability == UIForceTouchCapability.available {
    let bundleIdentifier = Bundle.main.bundleIdentifier
    let shortcutItem1 = UIApplicationShortcutItem(type: "\
(bundleIdentifier).OpenFavorites", localizedTitle: "Show Favorites",
localizedSubtitle: nil, icon: UIApplicationShortcutIcon(templateImageName:
"favorite-shortcut"), userInfo: nil)
    let shortcutItem2 = UIApplicationShortcutItem(type: "\
(bundleIdentifier).OpenDiscover", localizedTitle: "Discover restaurants",
localizedSubtitle: nil, icon: UIApplicationShortcutIcon(templateImageName:
"discover-shortcut"), userInfo: nil)
    let shortcutItem3 = UIApplicationShortcutItem(type: "\
(bundleIdentifier).NewRestaurant", localizedTitle: "New Restaurant",
```

```
localizedSubtitle: nil, icon: UIApplicationShortcutIcon(type: .add), userInfo:
nil)
    UIApplication.shared.shortcutItems = [shortcutItem1, shortcutItem2,
shortcutItem3]
}
```

In the above code, we first check if the device is 3D Touch capable. From iOS 9 and onwards, the `traitCollection` class comes with a property called `forceTouchCapability`, which indicates if the device is enabled with 3D Touch.

The rest of the code is pretty much the same with what I have covered before. We create three quick action items. Each of which has its own identifier, title and icon. For the *Discover restaurant* and *Show favorites* shortcuts, we use our own icons. So please download this icon pack (<http://www.appcoda.com/resources/swift3/QuickActionIcons.zip>) and put the icons into the asset catalog.

Now you're ready to go. Hit the Run button and deploy the app to your iPhone 6s/6s Plus. After launching the app once and going through the walkthrough screens, you will be able to see the quick actions.

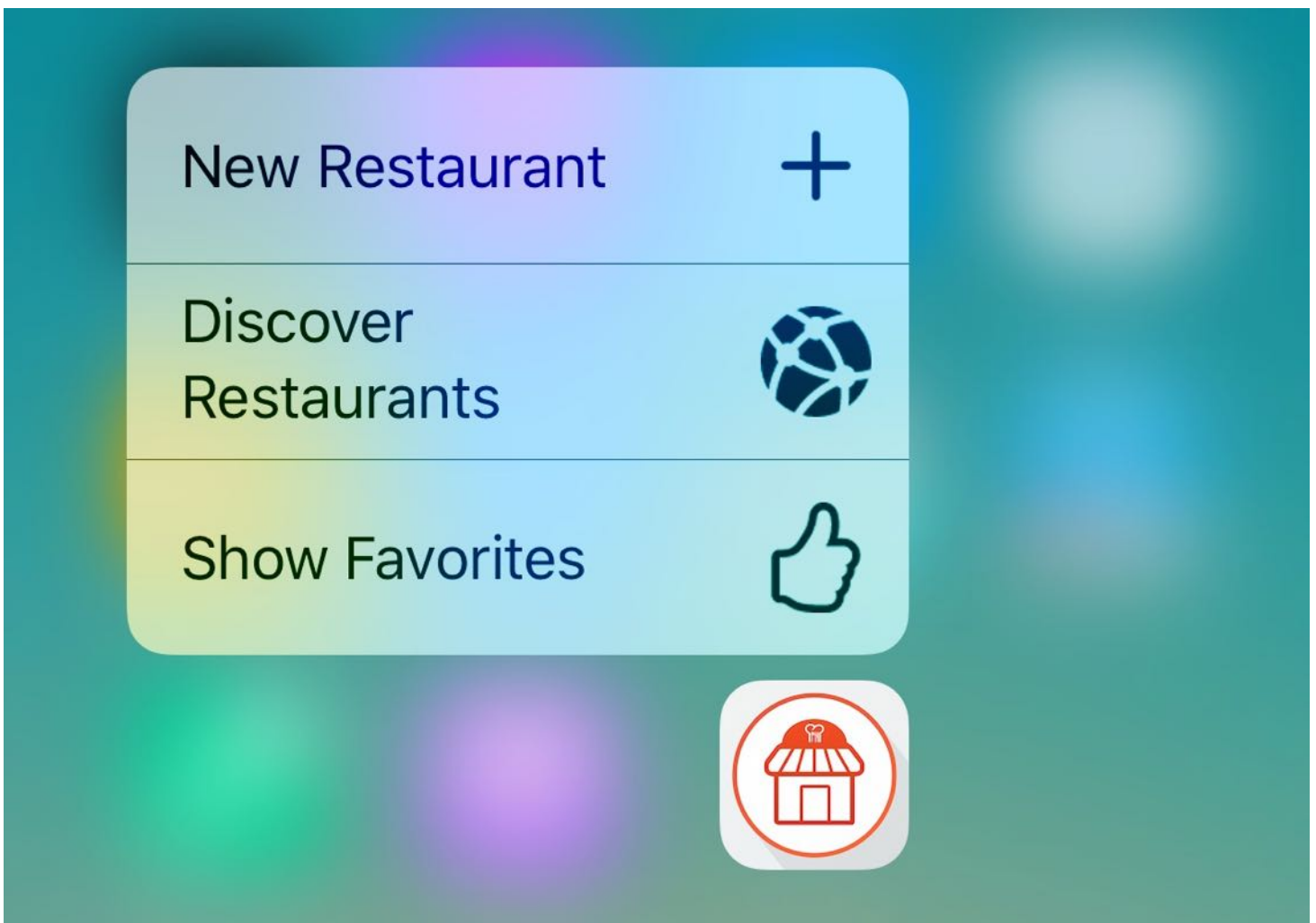


Figure 29-5. Quick actions in the FoodPin app

The quick actions are not ready to work yet because we haven't implemented the required methods to launch the quick actions. In iOS 9/10, there is a method called `application(_:performActionFor:completionHandler:)` defined in the `UIApplicationDelegate` protocol. When the user selects a quick action, this method is called. So we will implement the method in `AppDelegate.swift`.

But let's first declare an enum for the quick actions in the app delegate:

```
enum QuickAction: String {
    case OpenFavorites = "OpenFavorites"
    case OpenDiscover = "OpenDiscover"
    case NewRestaurant = "NewRestaurant"

    init?(fullIdentifier: String) {
```

```

        guard let shortcutIdentifier = fullIdentifier.components(separatedBy:
".").last else {
            return nil
        }

        self.init(rawValue: shortcutIdentifier)
    }
}

```

The Enumeration type in Swift is particularly useful for defining a group of related values. In the `QuickAction` enum, we define the available quick actions in each case. We also create an initialization method that turns a full identifier (e.g. `com.appcoda.NewRestaurant`) into the corresponding enumeration case (e.g. `NewRestaurant`).

Now implement the `application(_:performActionFor:completionHandler:)` method in the `AppDelegate` class like this:

```

func application(_ application: UIApplication, performActionFor shortcutItem:
UIApplicationShortcutItem, completionHandler: @escaping (Bool) -> Void) {

    completionHandler(handleQuickAction(shortcutItem: shortcutItem))
}

private func handleQuickAction(shortcutItem: UIApplicationShortcutItem) -> Bool
{

    let shortcutType = shortcutItem.type
    guard let shortcutIdentifier = QuickAction(fullIdentifier: shortcutType)
else {
        return false
    }

    guard let tabBarController = window?.rootViewController as?
UITabBarController else {
        return false
    }

    switch shortcutIdentifier {
    case .OpenFavorites:
        tabBarController.selectedIndex = 0
    case .OpenDiscover:
        tabBarController.selectedIndex = 1
    case .NewRestaurant:
        if let navController = tabBarController.viewControllers?[0] {
            let restaurantTableViewController =

```

```

navController.childViewControllers[0]
    restaurantTableViewController.performSegue(withIdentifier:
"addRestaurant", sender: restaurantTableViewController)
    } else {
        return false
    }
}

return true
}

```

When a quick action is activated, the `application(_:performActionFor:completionHandler:)` method is called. The selected shortcut (quick action) is passed as a parameter. We created a separate method, which we will discuss later, for handling the quick action. When the quick action is complete, you are expected to call the completion handler with an appropriate boolean value, depending on the success/failure of the quick action.

Let's now talk about the `handleQuickAction` method. It takes in a shortcut item and has a `switch` statement to perform the action accordingly:

1. For `.openFavorites` and `.openDiscover`, we simply change the selected index of the tab bar controller, so the app jumps right into the particular screen.
2. It takes a little more work for handling the New Restaurant action. We need to first retrieve the restaurant table view controller from the tab bar controller, and then programmatically invoke the `addRestaurant` segue. This will open the New Restaurant screen directly.

You can now run the app on your iPhone 6s/6s Plus. The quick actions should now work.

Peek and Pop

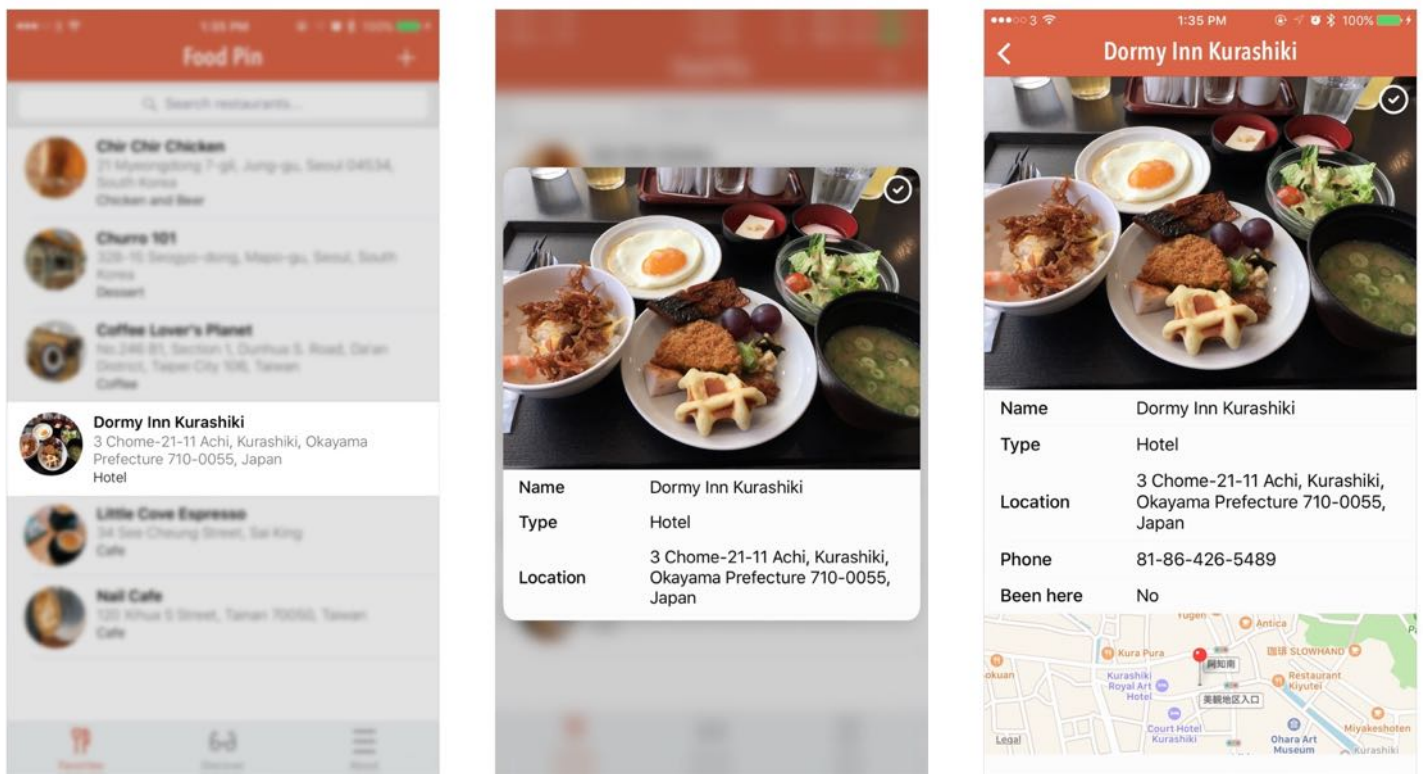
Peek and Pop allows your users preview content without opening the actual screen. As the user presses more deeply in a view controller with **Peek and Pop** enabled, the app first responds the user press to indicate that content preview is available, and then displays a preview of the content. When the user presses even harder, it pops into the full view of the content.

In iOS 9 or later, you can configure view controllers to support *peek and pop* by using predefined methods in the `UIViewController` class, and adopting a view controller protocol

named `UIViewControllerPreviewingDelegate`. Basically, here are the tasks you need to do:

1. Register the view controller for peek and pop by calling the `registerForPreviewing` method of `UIViewController`.
2. Adopt the two required methods of the new view controller protocol `UIViewControllerPreviewingDelegate`. By implementing the methods, you provide a preview view controller and a commit view controller to respond to the user's press.

Now that you have some basic understanding of the implementation, we will modify the FoodPin app to support peek and pop in the *Favorites* tab. Figure 29-6 illustrates how it works.



1 Press a little harder to peek at a restaurant. The selected restaurant remains visually sharp, while the rest of the content blurs

2 Press deeper to display a preview of the restaurant

3 Press even harder to pop

Figure 29-6. Peek and Pop in the FoodPin app

Registering the View Controller for Peek and Pop

The `RestaurantTableViewController` class is the source controller of peek and pop. To allow users to peek its content, the very first thing is to register the class for peek and pop. Insert the following lines of code in the `viewDidLoad` method:

```
if(traitCollection.forceTouchCapability == .available){
    registerForPreviewing(with: self as UIViewControllerPreviewingDelegate,
sourceView: view)
}
```

We first check if the device is capable of 3D Touch, and then call `registerForPreviewing` to itself to participate with 3D Touch preview (peek) and commit (pop).

Handling Peek and Pop

To handle peek and pop, we have to first adopt the `UIViewControllerPreviewingDelegate` protocol. Simply add the name of the protocol to the class declaration:

```
class RestaurantTableViewController: UITableViewController,
NSFetchedResultsControllerDelegate, UISearchResultsUpdating,
UIViewControllerPreviewingDelegate
```

You have to implement two required methods as stated in the protocol:

- `previewingContext(_:viewControllerForLocation:)` - this method is called when the user has pressed the source view deep enough to start a peek. In this case, it is the view of `RestaurantTableViewController`. You implement this method to retrieve the location of the touch and prepare the previewing view controller accordingly.
- `previewingContext(_:commitViewController:)` - this method is called to let you prepare for the display of the commit (pop) view.

Let's start to implement the first method. Add the following code snippet in the

`RestaurantTableViewController` class:

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

    guard let indexPath = tableView.indexPathForRow(at: location) else {
        return nil
    }
}
```

```

guard let cell = tableView.cellForRow(at: indexPath) else {
    return nil
}

guard let restaurantDetailViewController =
storyboard?.instantiateViewController(withIdentifier:
"RestaurantDetailViewController") as? RestaurantDetailViewController else {
    return nil
}

let selectedRestaurant = restaurants[indexPath.row]
restaurantDetailViewController.restaurant = selectedRestaurant
restaurantDetailViewController.preferredContentSize = CGSize(width: 0.0,
height: 450.0)

previewingContext.sourceRect = cell.frame

return restaurantDetailViewController
}

```

When the method is invoked, the location of the touch is passed as a parameter of the type `CGPoint`. From this piece of information, we can deduce the selected table view cell.

`UITableView` provides a method called `indexPathForRow(at:)`, that takes in a point and returns you an index path identifying the row (and section) of that given point. With the index path, we can easily find out which cell has been selected.

The `RestaurantDetailViewController` class has been designed to display the restaurant details. Therefore we simply use it as the previewing view controller. First, we instantiate the view controller from storyboard using the `RestaurantDetailViewController` identifier. Like what we have done in the earlier chapter, we assign the selected restaurant to the detail view controller. But as the preview only takes up certain part of the screen, we want to limit the height to 450 points. This line of code is optional. If you do not set the preferred content size, it will still work. iOS just displays the previewing view controller with a default size.

Lastly, we set the `sourceRect` property of the preview context. Before the app previews (peeks) the content, the selected cell remains visually sharp while the rest of the content blurs. The last line of code simply tells the preview context that it should keep the cell visually sharp.

Now that you've created the method for handling peek, let's move onto the implementation of pop. Insert the following method in the class:


```
func previewingContext(_ previewingContext: UIViewControllerPreviewing, commit
viewControllerToCommit: UIViewController) {
    show(viewControllerToCommit, sender: self)
}
```

It's very straightforward to present the commit (pop) view controller. In this case, it's the detail view controller we have created earlier. As the detail view is actually embedded in a navigation controller, we simply call `showViewController` to present it.

That's it! However, before testing the changes on your device, remember to set the identifier of the Restaurant Detail View Controller to `RestaurantDetailViewController`. Otherwise, you will not be able to instantiate the controller programmatically. Go to `Main.storyboard` and select Restaurant Detail View Controller. Under the Identity inspector, set the storyboard ID to `RestaurantDetailViewController`.

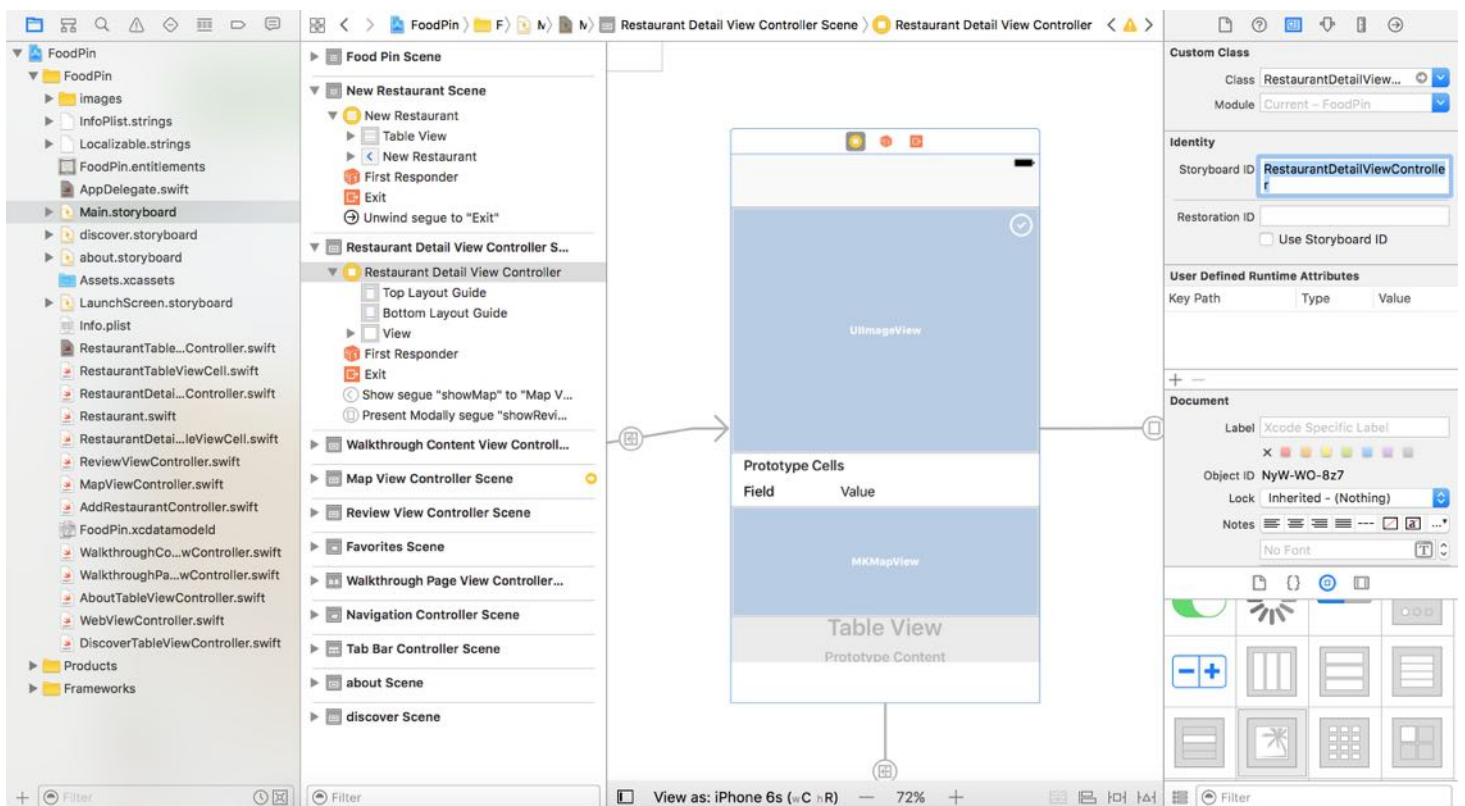


Figure 29-7. Assign an identifier for the Restaurant Detail View Controller

You're good to go. Plug your iPhone 6s/6s Plus to your Mac and run the app on the device. The FoodPin app should now let you peek and pop a restaurant.

Summary

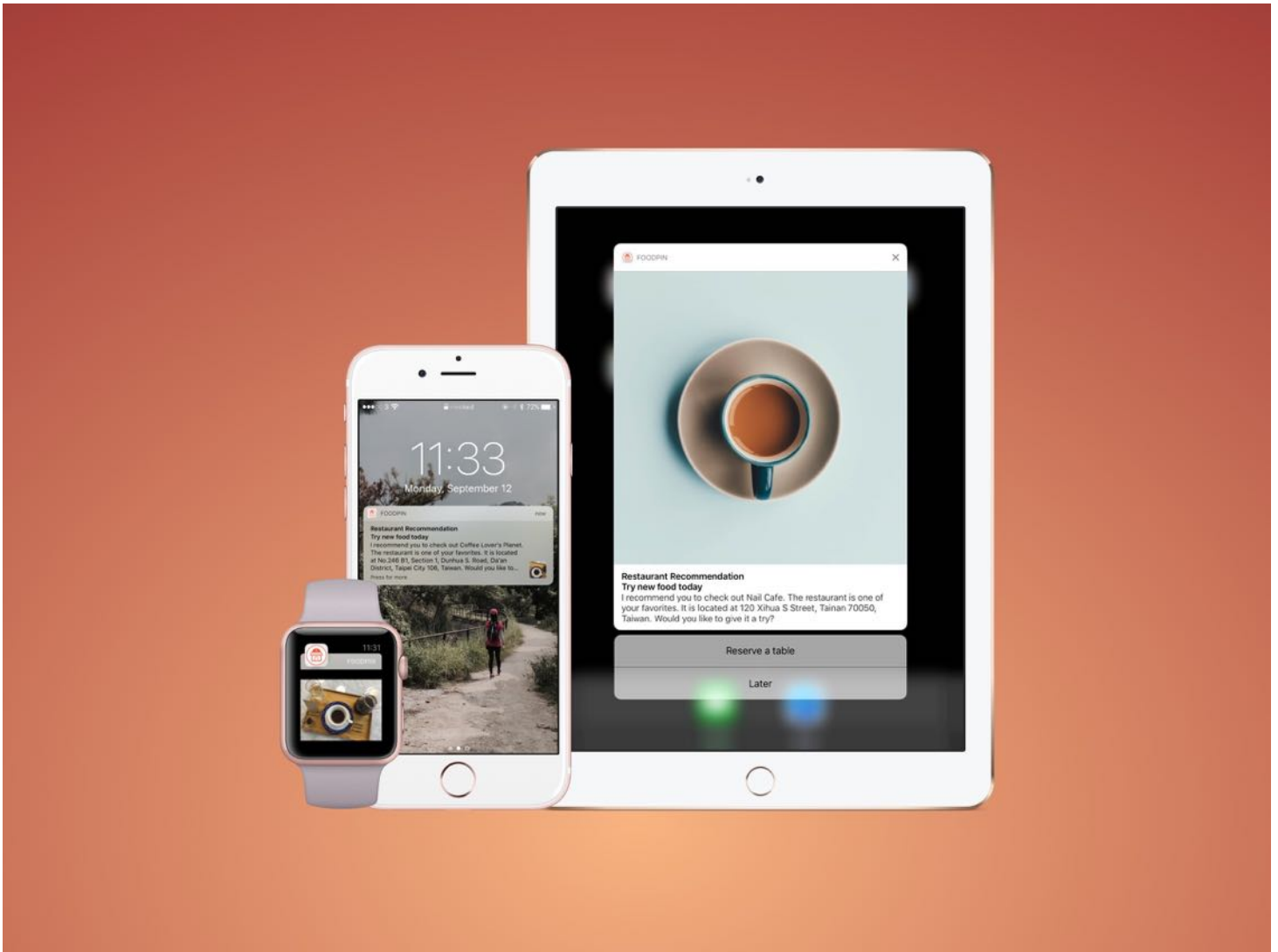
In this chapter, I have walked you through some basic APIs of 3D Touch. As you can see, the APIs are pretty easy to use. And you can add quick actions without writing a line of code.

3D Touch provides an entirely new way for users to interact with their phones. As an app developer, it is your responsibility to deliver a great user experience for your users. Now is the best time to start thinking about how to make use of this new technology to further improve your apps.

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPin3DTouch.zip>.

Chapter 30

Developing User Notifications in iOS 10



iOS 10 introduces a number of new frameworks. Aside from SiriKit, the User Notifications framework has also drawn a lot of attention.

Prior to iOS 10, user notifications are plain and simple. No rich graphics or media. It is just in text format. Depending on the user's context, the notification can appear on lock screen or home screen. If the user misses any of the notifications, they can bring up the Notification Center to reveal all pending notifications.

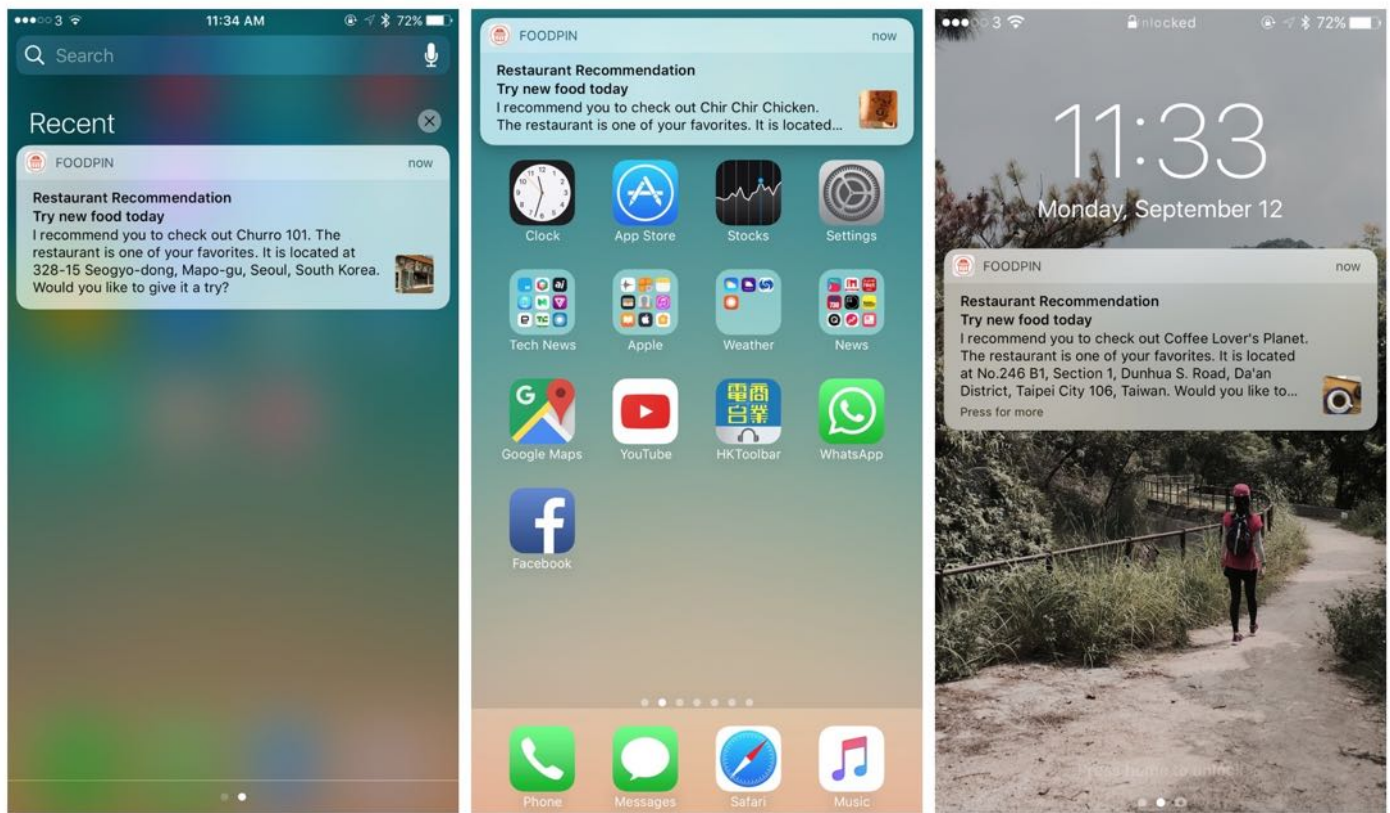


Figure 30-1. User notifications in lock screen, home screen, and notification center

In iOS 10, Apple is revamping the notification system to support user notifications in rich content, and custom notification UI. By rich content, it means you can include static images, animated GIFs, videos, and audios in the notifications. Figure 30-2 gives you an idea of the new notifications.

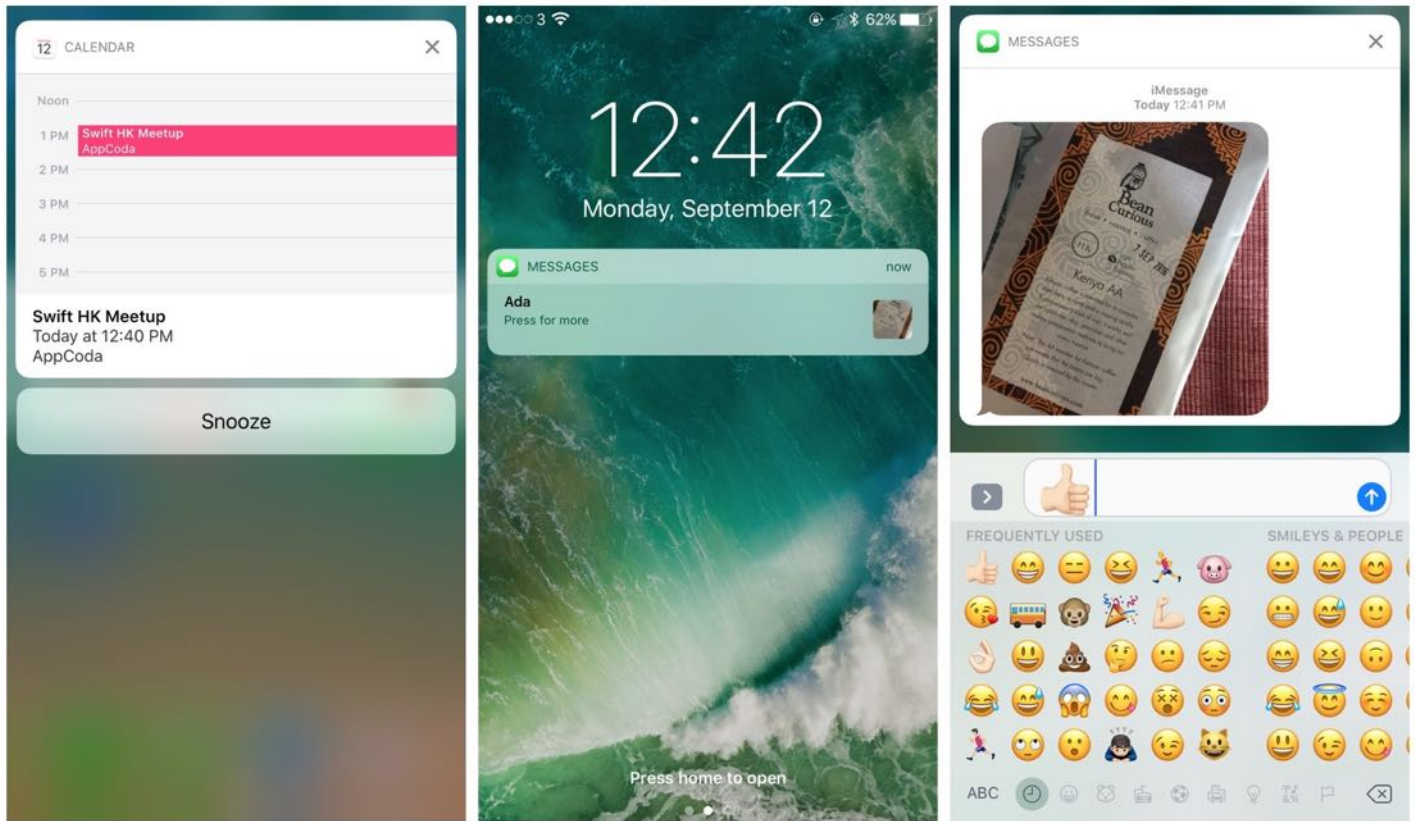


Figure 30-2. Sample user notifications in rich content

You may have heard of push notifications, which have been widely adopted in messaging apps. Actually user notifications can be classified into two types: *local notifications* and *remote notifications*. Local notifications are triggered by the application itself and contained on the user's device. For example, a location-based application will send users a notification when they are in a particular area. Or a to-do list app displays a notification when an item is close to the due date.

Remote notifications are usually initiated by server side applications that reside on remote servers. When the server application wants to send messages to users, it sends a notification to Apple Push Notification Service (or APNS for short). The service then forwards the notification to users' devices.

We're not going to talk about the implementation of remote notifications in this chapter. Instead, we will focus on discussing local notifications, and show you how to use the new User Notifications framework to implement the rich-content notifications.

User Notifications in the FoodPin App

So what features are we going to add in the FoodPin app? Using local notifications is a great way to remind your users about your app. A recent study revealed that only less than 25% of people will use an app for more than one time. In other words, over 75% of users download an app, open it once and then never return to it.

More than 75% of App Downloads Open an App Once And Never Come Back

by Erin Griffith

<http://fortune.com/2016/05/19/app-economy/>

With two million apps available in the App Store, it is hard to get people notice and download your app. Yet it is even harder to keep people using it. A smart use of user notifications can help you retain your users, and improve the user experience of your app.

The User Notifications framework provides different triggers to initiate a local notification:

- **Time-based trigger** - triggers a local notification after a specific amount of time (say, after 10 minutes).
- **Calendar-based trigger** - triggers a local notification at a specified date and time.
- **Location-based trigger** - triggers a local notification when the user reaches a specific location.

The FoodPin app is designed for food lovers to save their favorite restaurants. Wouldn't it be great if the app suggests the user his/her favorite restaurants when he/she reaches a particular location? For example, you've saved a few restaurants in Tokyo. At the time you arrive Tokyo, the app triggers a notification showing a list of your favorite restaurants in the city.

Or you can use calendar-based trigger to initiate notifications at festival times (say, 10 days before Christmas). The notification can be something like this:

"Hey, Christmas is around the corner. It's time to plan for the holidays and try some good food with your friends. Here are some of your favorite restaurants you can check out."

These are a couple of sample use cases. It's more likely users will return to the app after seeing the notifications.

To keep things simple in this beginner book, we would not implement the above triggers. Instead, I will show you how to use time-based triggers to trigger local notifications. That said, it doesn't mean that the notifications are useless or spammy. And once you understand the basics of the User Notifications framework, it is not too difficult for you to implement other types of triggers.

What we are going to do is that we will notify the user and recommend him/her a restaurant after a certain period of time (say, 24 hours) since he/she last used the app. Furthermore, we will allow users to interact with the notification. When the user sees the notification, he/she will be given an option to reserve a table. If the user taps the button, it will directly make a call to the restaurant. Figure 30-3 illustrates a sample notification.

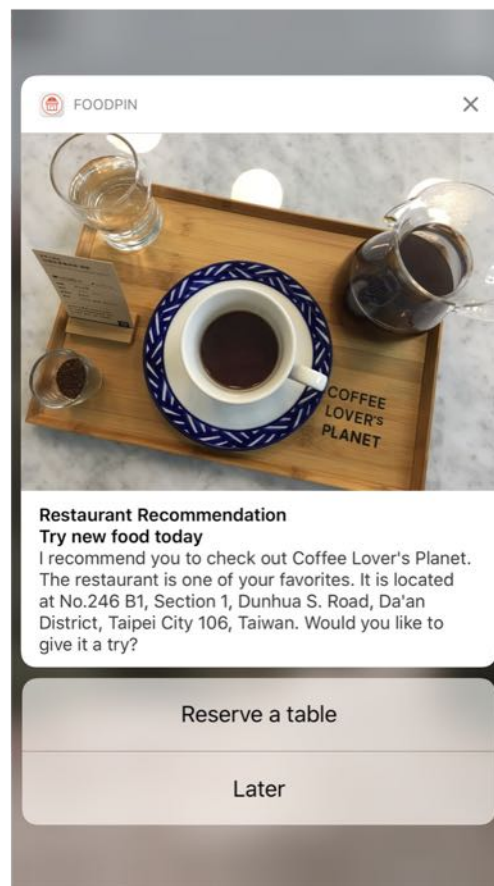


Figure 30-3. The FoodPin app recommends users a restaurant through local notifications

Does it look great? Let's get started and see how you can make the notifications happen in your app.

Using the User Notifications Framework

The User Notifications framework is a brand new framework, introduced in iOS 10, for managing and scheduling notifications. To implement user facing notifications, the very first thing to do is import the framework in your code, so you can access the APIs bundled in the framework.

Later we will implement user locations in both `AppDelegate.swift` and `RestaurantTableViewController.swift`. Insert the following line of code in both files:

```
import UserNotifications
```

Asking for User Permission

Regardless of the type of notifications, you have to ask for the user's authorization and permission before you can send notifications to the user's device.

If you have used iOS for some time, this kind of authorization request shouldn't be new to you. The request is usually prompted when you first launch an app.

So in code, we usually implement the authorization request in the `AppDelegate.swift` file. Insert the following code snippet in the `application(_:didFinishLaunchingWithOptions:)` method:

```
UNUserNotificationCenter.current().requestAuthorization(options: [.alert,
.sound, .badge]) { (granted, error) in
    if granted {
        print("User notifications are allowed.")
    } else {
        print("User notifications are not allowed.")
    }
}
```

It is very simple to prompt users an authorization request. As you can see from the above code,

you simply call `requestAuthorization` on the `UNUserNotificationCenter` object associated with the application. And we request the ability to display alerts, play sounds, and update the app's badge.

Now run the project to test it out. When the app is launched, you should see the authorization request. Once you accept it, the app is allowed to send notifications to the device. To verify the notification settings, you can go to `Setting > FoodPin > Notifications`.

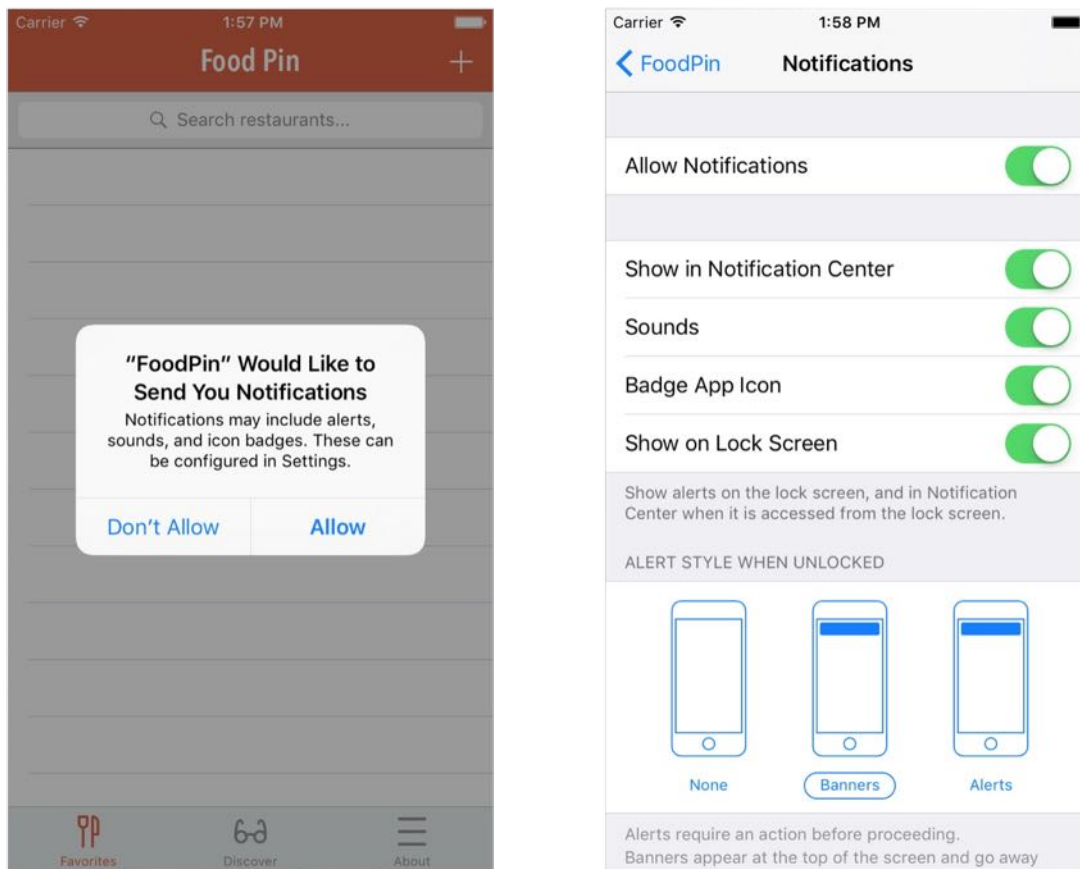


Figure 30-4. Ask for the user's permission and authorization

Creating and Scheduling Notifications

Now the FoodPin app is ready to send notifications to the users. Let's first check out the basic look & feel of a notification in iOS 10. At the top, it is the title of the notification. The next line is the subtitle, followed by the body of the message. Figure 30-5 is an example.

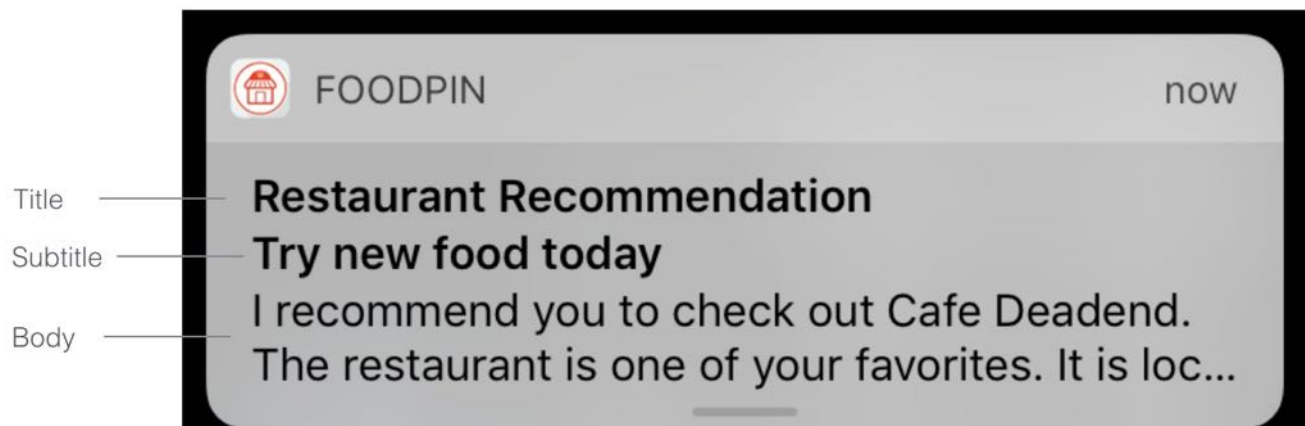


Figure 30-5. A standard notification in iOS 10

The content of a user notification is represented by `UNMutableNotificationContent`. To create the content, you instantiate an `UNMutableNotificationContent` object and set its properties to the appropriate data. Here is an example:

```
let content = UNMutableNotificationContent()
content.title = "Restaurant Recommendation"
content.subtitle = "Try new food today"
content.body = "I recommend you to check out Cafe Deadend."
```

If you want to play a sound when the notification is triggered, you can also set the `sound` property of the content:

```
content.sound = UNNotificationSound.default()
```

Scheduling the notification is as simple as creating the `UNNotificationRequest` object with your preferred trigger, and then adding the request to `UNUserNotificationCenter`. Take a look at the code snippet below. This is the code you need to schedule a notification.

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10, repeats:
false)
let request = UNNotificationRequest(identifier: "foodpin.restaurantSuggestion",
content: content, trigger: trigger)

// Schedule the notification
UNUserNotificationCenter.current().add(request, withCompletionHandler: nil)
```

As said before, we want to trigger the notification after a specific period of time. We create a

`UNTimeIntervalNotificationTrigger` object and set the time interval to a specific value (e.g. 10 seconds). Then we construct a `UNNotificationRequest` object with the notification content and the trigger. You have to assign the request with a unique identifier. Later if you want to remove or update the notification, you use this identifier to identify the notification. Finally, you call the `add` method of `UNUserNotificationCenter` with the notification request to schedule the notification.

Now that you should have some ideas about how to create and schedule the notifications, let's implement the Restaurant Recommendation notification. Open

`RestaurantTableViewController.swift` and insert the following method:

```
func prepareNotification() {
    // Make sure the restaurant array is not empty
    if restaurants.count <= 0 {
        return
    }

    // Pick a restaurant randomly
    let randomNum = Int(arc4random_uniform(UInt32(restaurants.count)))
    let suggestedRestaurant = restaurants[randomNum]

    // Create the user notification
    let content = UNMutableNotificationContent()
    content.title = "Restaurant Recommendation"
    content.subtitle = "Try new food today"
    content.body = "I recommend you to check out \(suggestedRestaurant.name!).
The restaurant is one of your favorites. It is located at \
(suggestedRestaurant.location!). Would you like to give it a try?"
    content.sound = UNNotificationSound.default()

    let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 10, repeats:
false)
    let request = UNNotificationRequest(identifier:
"foodpin.restaurantSuggestion", content: content, trigger: trigger)

    // Schedule the notification
    UNUserNotificationCenter.current().add(request, withCompletionHandler: nil)
}
```

To better manage the code, we create the `prepareNotification()` method for handling user notifications. We have gone through most of the code before. But the first line of code is new to you. Here we want to randomly pick a restaurant from the favorites and recommend it to the

user. This line of code is used to generate a random number:

```
let randomNum = Int(arc4random_uniform(UInt32(restaurants.count)))
```

Say, you have 10 restaurants in the favorites. The function will generate a number between 0 and 9. With the random number, we can choose a suggested restaurant from the array, and create the notification content.

One thing you have to take note is that we now set the time interval to 10 seconds. This is for demo purpose and ease of testing. In reality, this is too short. You may want to trigger the notification after 24 hours (24 60 60 seconds) of use or even longer:

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 86400, repeats: false)
```

Now that the `prepareNotification` method is ready, you can call it in the `viewDidLoad` method. Insert this line of code before the end of the method:

```
prepareNotification()
```

Cool! Let's run the project and have a quick test. The notification will not be presented in-app. Therefore, once you launch your app, make sure you go back to home screen or lock screen. Wait for 10 seconds and you should see the notification.

Note: If you test the app using the iPhone simulator, press shift-command-H to switch to the home screen. To access the lock screen, press command-L.

If the notification appears on the lock screen, you can slide the notification to return to the FoodPin app. And when the device is unlocked, the notification rolls down from the top as a banner. You can swipe it further down to reveal the full content or simply tap it to jump back to the app.

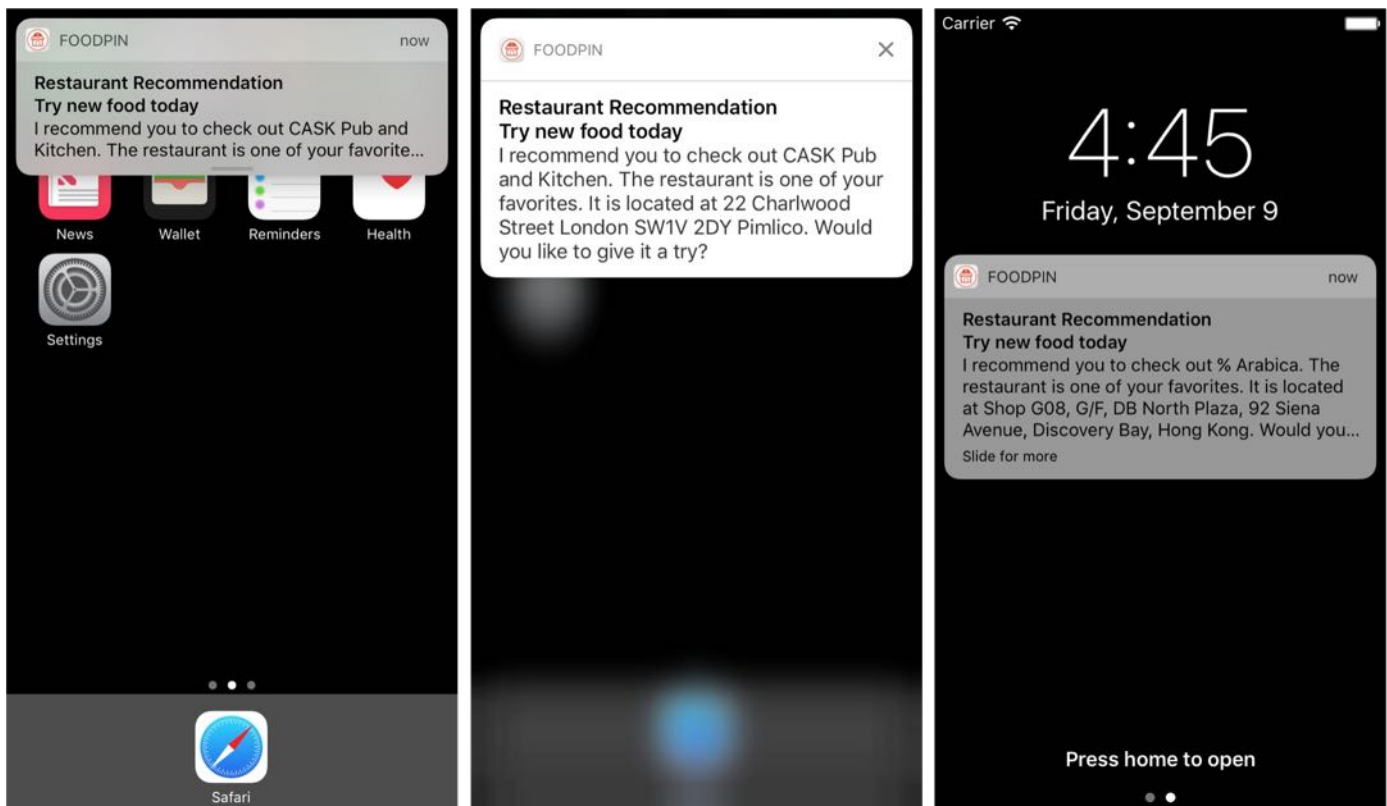


Figure 30-6. User notifications on home screen and lock screen

Adding Images to the Notification

We've talked about notifications with rich content from the beginning of the chapter. So far the notification we have created is in plain text. How can we bundle the image of the suggested restaurant in the notification?

It is as simple as setting the `attachment` property of the `UNMutableNotificationContent` object:

```
content.attachments = [attachment]
```

The `attachments` property accepts an array of `UNNotificationAttachment` objects to display with the notification. Attachment can be images, sounds, audio, and movie files.

Note that you should provide the file URL of the attachment. In our case, it's the image file of the suggested restaurant.

If you're not forgetful, you probably remember that the `image` property of `RestaurantMO` is of type `NSData`. So how can we create the image file from the image data in order to create the attachment object?

Let's first check out the code for creating attachment. You can insert the following in the `prepareNotification` method (before the instantiation of the trigger):

```
let tempDirURL = URL(fileURLWithPath: NSTemporaryDirectory(), isDirectory:
true)
let tempFileURL = tempDirURL.appendingPathComponent("suggested-restaurant.jpg")

if let image = UIImage(data: suggestedRestaurant.image! as Data) {

    try? UIImageJPEGRepresentation(image, 1.0)?.write(to: tempFileURL)
    if let restaurantImage = try? UNNotificationAttachment(identifier:
"restaurantImage", url: tempFileURL, options: nil) {
        content.attachments = [restaurantImage]
    }
}
```

The iOS SDK provides a built-in function called `UIImageJPEGRepresentation` to convert image data into a JPEG image file. What's going on in the above code is we first find the temporary directory for saving the image. The `NSTemporaryDirectory()` function returns you the directory for temporary files. And we set the temporary file name as `suggested-restaurant.jpg`. If you print the file path to console, it will be something like this:

```
file:///Users/simon/Library/Developer/CoreSimulator/Devices/1234882-3638-3333-
A273-AFA633992D8A/data/Containers/Data/Application/8A91238F-64EE-4F33-223C-
5BB12D40BB23/tmp/suggested-restaurant.jpg
```

The `UIImageJPEGRepresentation` function takes in an `UIImage` object, so we first create an `UIImage` object and then pass it to the function for writing the image data to a JPEG file. With the image file created in the temporary directory, we can create the `UNNotificationAttachment` object and assign it to the `attachments` property of the notification.

Note: Both `UIImageJPEGRepresentation` and `UNNotificationAttachment` throw an exception when it encounters an error. You can refer to the appendix to learn about error handling in Swift and how to use do-try-catch.

It's time to test the app again. Run the project to open the app in simulator. Remember to go back to home screen and wait for the notification to appear. This time you should see a small

thumbnail in the notification. Swipe it down to view a large version of the image.

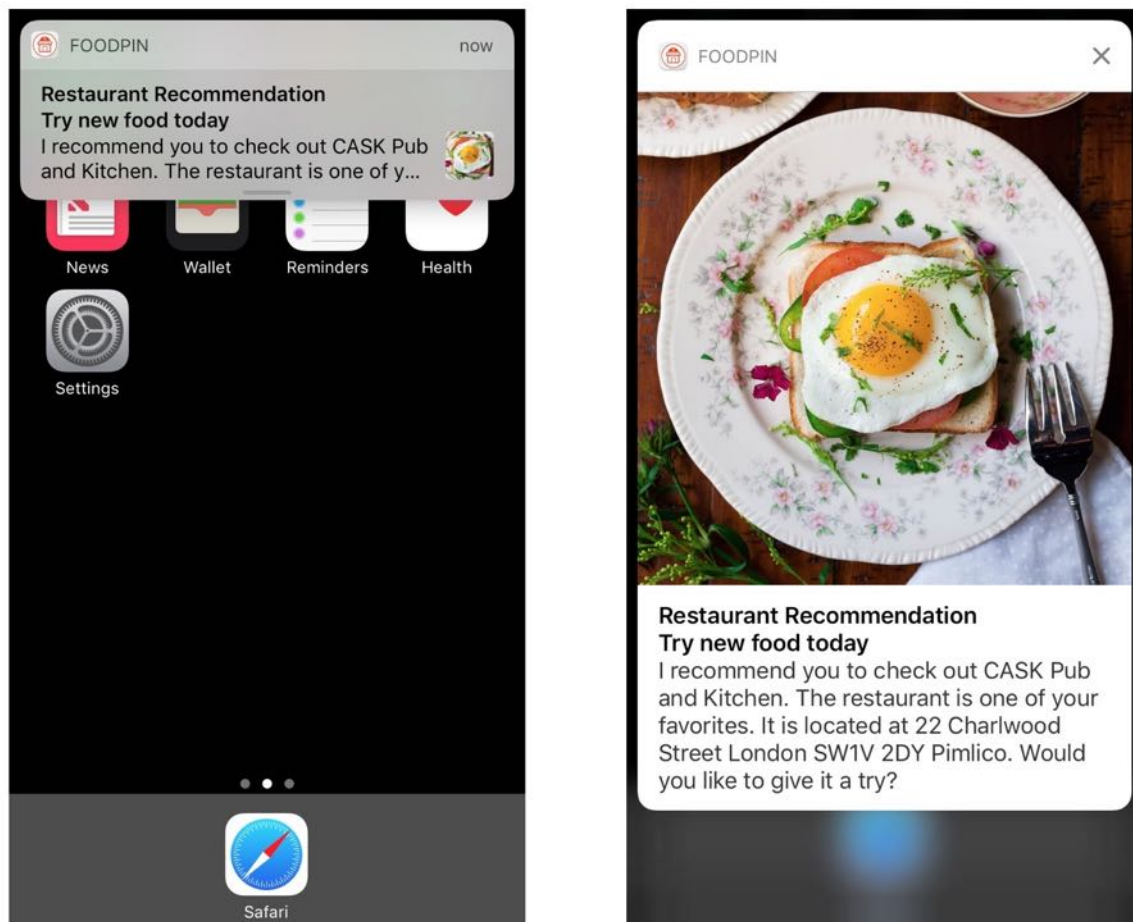


Figure 30-7. Adding an image to notifications

Interacting with User Notifications

Now users have only one way to interact with the notification: tap to launch the app. This is the default action if you do not provide any custom implementation.

Actionable notifications let users respond to the notification without having to switch to the app. One great example of actionable notifications is reminders. When you receive a notification from the Reminders app, you will have the option to manage the reminder directly from the notification. You can either mark it as completed or reschedule the reminder. All these can be done without the need of launching the app.

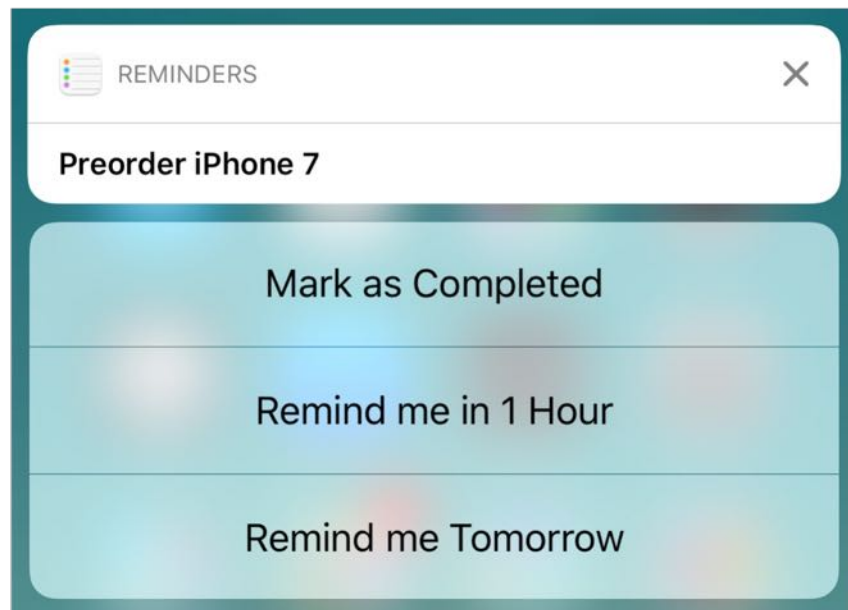


Figure 30-8. A sample notification from Reminders

Creating and Registering Custom Actions

With the User Notifications framework, we can implement custom actions for notifications from the FoodPin app. When a notification appears on screen, it provides two options for users to choose:

1. Reserve a table - if the user selects this option, the app will make a call to the suggested restaurant so the user can reserve a table.
2. Later - for this option, we just dismiss the notification.

To implement custom actions, you need to create a `UNNotificationAction` object and associate it with a notification category. The action object has a unique identifier and a title (e.g. Reserve a table), which appears on the action button. Optionally, you can specify how the action should be performed. By default, action will be a background action, which means it dismisses the notification and perform your custom task in background. For example, if we write the `Later` action in code, it will be like this:

```
let laterAction = UNNotificationAction(identifier: "foodpin.cancel", title: "Later", options: [])
```

If the `Later` is selected, the action is to dismiss the notification. Thus, we do not provide additional options when creating the action object.

On the other hand, the "Reserve a table" action will be a foreground action because we have to bring the app to foreground so that it can make a call. Thus, the action will be implemented like this:

```
let makeReservationAction = UNNotificationAction(identifier:
"foodpin.makeReservation", title: "Reserve a table", options: [.foreground])
```

Once you set up the action objects, you associate it with a category:

```
let category = UNNotificationCategory(identifier: "foodpin.restaurantaction",
actions: [makeReservationAction, cancelAction], intentIdentifiers: [], options:
[])
```

You give the category a unique identifier and pass the action objects to be associated with the category. Once you have the category ready, you can register it to the

`UNUserNotificationCenter` object like this:

```
UNUserNotificationCenter.current().setNotificationCategories(["foodpin.restaurar
```

So far we have created the actions and registered them to the notification center, but these actions are not yet associated with the notification. To do that, all you need to do is set the category identifier to the `categoryIdentifier` property of the notification content.

```
content.categoryIdentifier = "foodpin.restaurantaction"
```

That's the code you need to implement custom actions for user notifications. So insert the following code snippet to the `prepareNotification` method (before the trigger variable):

```
let categoryIdentifier = "foodpin.restaurantaction"
let makeReservationAction = UNNotificationAction(identifier:
"foodpin.makeReservation", title: "Reserve a table", options: [.foreground])
let cancelAction = UNNotificationAction(identifier: "foodpin.cancel", title:
"Later", options: [])
let category = UNNotificationCategory(identifier: categoryIdentifier, actions:
[makeReservationAction, cancelAction], intentIdentifiers: [], options: [])
UNUserNotificationCenter.current().setNotificationCategories([category])
```

```
content.categoryIdentifier = categoryIdentifier
```

The above code is exactly the same what we have just discussed. Now run the app to test the notification actions. When the notification banner appears, swipe it down and you should see the custom actions just implemented.



Figure 30-9. Custom actions for User Notifications

Handling the Actions

If you tap the *Reserve a table* button, it brings up the FoodPin app. However, it will not call the restaurant for you. As explained in the earlier section, the action object has an option that indicates how the action should be performed. For the *Later* button, we did not provide any additional options, so it is default to dismiss the notification. And for the *Reserve a table* button, we set the option to `.foreground`, that will bring the app to foreground.

But how can we handle the action when the app returns to foreground?

The `UNUserNotificationCenterDelegate` protocol in the User Notifications framework is designed for this purpose. The protocol defines a method for responding to the actionable notifications:

```
optional func userNotificationCenter(_ center: UNUserNotificationCenter,
didReceive response: UNNotificationResponse, withCompletionHandler
completionHandler: () -> Void)
```

To handle the action and execute custom code, you need to implement the protocol in a delegate object and assign it to the notification center object. When the app is returned to foreground, the methods will be called accordingly.

We will implement this protocol in `AppDelegate`. But before we do that, you probably have another question in mind. We're going to make a call to the suggested restaurant. How can we pass the restaurant number from `RestaurantTableViewController` to `AppDelegate`?

The notification content has a property named `userInfo` for you to store custom information in the form of a dictionary. For example, you can store the phone number in the notification like this:

```
content.userInfo = ["phone": suggestedRestaurant.phone!]
```

Put the above line of code in the `prepareNotification` method right below `content.sound`.

With the phone number associated with the notification, we now implement the `UNUserNotificationCenterDelegate` protocol. As we use `AppDelegate` as the delegate object, update the class declaration to implement the protocol:

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

When an action of a notification is selected by a user, the `userNotificationCenter(_:didReceive:withCompletionHandler:)` is called. So we provide our own implementation for making a call to the suggested restaurant. Insert this method in the `AppDelegate.swift` file:

```

func userNotificationCenter(_ center: UNUserNotificationCenter, didReceive
response: UNNotificationResponse, withCompletionHandler completionHandler:
@escaping () -> Void) {

    if response.actionIdentifier == "foodpin.makeReservation" {
        print("Make reservation...")
        if let phone = response.notification.request.content.userInfo["phone"]
{
            let telURL = "tel://\(\(phone))"
            if let url = URL(string: telURL) {
                if UIApplication.shared.canOpenURL(url) {
                    print("calling \(\(telURL))")
                    UIApplication.shared.open(url)
                }
            }
        }
    }

    completionHandler()
}

```

As we only need to handle the *Reserve a table* action, we first verify the identifier of the action. And then we retrieve the phone number of the restaurant from the `userInfo` property of the notification content.

In iOS, you can use launch some of the system apps using an appropriate URL. In this case, we want to open the Phone app to call up the number. You can use the `tel` scheme to initiate a call. Here is a sample URL:

```
tel://<phone-number>
```

So in the above code, we construct the `telURL` and then call the `open` method of `UIApplication` to launch the Phone app.

At the end of the method, it is required to call the `completionHandler` block to let the system know that you are done processing the notification.

The last thing you have to do is set the delegate of the notification center. In the `application(_:didFinishLaunchingWithOptions:)` of `AppDelegate`, add the following line of code:

```
UNUserNotificationCenter.current().delegate = self
```

That's it. Run the project and deploy the app on a real device for testing. You have to use a real device because you can't make a call using the simulator. If you select the *Reserve a table* button, the app will launch the FoodPin app and switch over to the Phone app to make a call.

Summary

The User Notifications framework in iOS 10 is a brand new framework for developers to manage and schedule user notifications. In this chapter, I have given you an overview of the framework and demonstrate how to schedule a local notification.

With interactive and rich content notifications, you can improve the user experience and engagement of your app. This is a good way to increase app retention rate. When you're going to build your next app, try to make use of user notifications and see how it can add value to your app.

Lastly, thanks for reading this book. This has been a long journey for both of us. I wish you the best of luck and hope you will release your app very soon. If you've submitted an app that's been approved, I'd love to hear about your success story. Feel free to email me at simonng@appcoda.com and remember to join our developer group on Facebook (<https://www.facebook.com/groups/appcoda>).

For reference, you can download the complete Xcode project from <http://www.appcoda.com/resources/swift3/FoodPinUserNotifications.zip>.

Appendix - Swift Basics

Swift is a new programming language for developing iOS, macOS, watchOS and tvOS apps. As compared to Objective-C, Swift is a neat language and will definitely make developing iOS apps easier. In this appendix, I will give you a brief introduction of Swift including a couple of the changes in Swift 3.

Variables, Constants and Type Inference

In Swift, you declare variables with the `var` keyword and constants using the `let` keyword. Here is an example:

```
var numberOfRows = 30
let maxNumberOfRows = 100
```

These are the two keywords you need to know for variable and constant declaration. You simply use the `let` keyword for storing value that is unchanged. Otherwise, use `var` keyword for storing values that can be changed.

Isn't it easier than Objective-C?

What's interesting is that Swift allows you to use nearly any character for both variable and constant names. You can even use an emoji character for the naming.

Quick Tip: You may wonder how you can type an emoji character in Mac OS. It's easy. Just press control-command-spacebar and an emoji picker will be displayed.

You may notice a huge difference in variable declaration between Objective-C and Swift. In Objective-C, developers have to specify explicitly the type information when declaring a variable. Be it an `int` or `double` or `NSString`, etc.

```
const int count = 10;
double price = 23.55;
NSString *myMessage = @"Objective-C is not dead yet!";
```


It's your responsibility to specify the type. In Swift, you no longer need to annotate variables with type information. It provides a huge feature known as *Type inference*. This feature enables the compiler to deduce the type automatically by examining the values you provide in the variable.

```
let count = 10
// count is inferred to be of type Int
var price = 23.55
// price is inferred to be of type Double
var myMessage = "Swift is the future!"
// myMessage is inferred to be of type String
```

It makes variable and constant declaration much simpler, as compared to Objective-C. Swift provides an option to explicitly specify the type information if you wish. The below example shows how to specify type information when declaring a variable in Swift:

```
var myMessage : String = "Swift is the future!"
```

No Semicolons

In Objective-C, you need to end each statement in your code with a semicolon. If you forget to do so, you will end up with a compilation error. As you can see from the above examples, Swift doesn't require you to write a semicolon (;) after each statement, though you can still do so if you like.

```
var myMessage = "No semicolon is needed"
```

Basic String Manipulation

In Swift, strings are represented by the `String` type, which is fully Unicode-compliant. You can declare strings as variables or constants:

```
let dontModifyMe = "You cannot modify this string"
var modifyMe = "You can modify this string"
```

In Objective-C, you have to choose between `NSString` and `NSMutableString` classes to indicate whether the string can be modified. You do not need to make such choice in Swift. Whenever you assign a string as variable (i.e. `var`), the string can be modified in your code.

Swift simplifies string manipulating and allows you to create a new string from a mix of constants, variables, literals, as well as, expressions. Concatenating strings is super easy. Simply add two strings together using the `+` operator:

```
let firstMessage = "Swift is awesome. "  
let secondMessage= "What do you think?"  
var message = firstMessage + secondMessage  
print(message)
```

Swift automatically combines both messages and you should see the following message in console. Note that `print` is a global function in Swift to print the message in console.

Swift is awesome. What do you think? You can do that in Objective C by using the `stringWithFormat:` method. But isn't the Swift version more readable?

```
NSString *firstMessage = @"Swift is awesome. ";  
NSString *secondMessage = @"What do you think?";  
NSString *message = [NSString stringWithFormat:@"%@@", firstMessage,  
secondMessage];  
NSLog(@"%@", message);
```

String comparison is more straightforward. You can use the `==` operator to compare two strings like this:

```
var string1 = "Hello"  
var string2 = "Hello"  
if string1 == string2 {  
    print("Both are the same")  
}
```

Arrays

The syntax of declaring an array in Swift is similar to that in Objective C. Here is an example:

Objective C:

```
NSArray *recipes = @[@"Egg Benedict", @"Mushroom Risotto", @"Full Breakfast",  
@"Hamburger", @"Ham and Egg Sandwich"];
```

Swift:

```
var recipes = ["Egg Benedict", "Mushroom Risotto", "Full Breakfast",  
"Hamburger", "Ham and Egg Sandwich"]
```

While you can put any objects in `NSArray` or `NSMutableArray` in Objective C, arrays in Swift can only store items of the same type. In the above example, you can only store strings in the string array. With type inference, Swift automatically detects the array type. But if you like, you can also specify the type in the following form:

```
var recipes : String[] = ["Egg Benedict", "Mushroom Risotto", "Full Breakfast",  
"Hamburger", "Ham and Egg Sandwich"]
```

Swift provides various methods for you to query and manipulate an array. Simply use the `count` method to find the number of items in the array:

```
var numberOfItems = recipes.count  
// recipes.count will return 5
```

Swift makes operations on array much simpler. You can add an item by using the `+=` operator:

```
recipes += ["Thai Shrimp Cake"]
```

This also applies when you need to add multiple items:

```
recipes += ["Creme Brelee", "White Chocolate Donut", "Ham and Cheese Panini"]
```

To access or change a particular item in an array, pass the index of the item by using subscript syntax just like that in Objective C and other programming language:

```
var recipeItem = recipes[0]  
recipes[1] = "Cupcake"
```

One interesting feature of Swift is that you can use `...` to change a range of values. Here is an example:

```
recipes[1...3] = ["Cheese Cake", "Greek Salad", "Braised Beef Cheeks"]
```

This changes the item 2 to 4 of the `recipes` array to "Cheese Cake", "Greek Salad" and "Braised Beef Cheeks". (Remember the first item in an array starts with the index 0. This is why index 1 refers to item 2.)

If you print the array to console, here is the result:

- Egg Benedict
- Cheese Cake
- Greek Salad
- Braised Beef Cheeks
- Ham and Egg Sandwich

Dictionaries

Swift provides three primary collection types: *arrays*, *dictionaries* and *sets*. Now let's talk about dictionaries. Each value in a dictionary is associated with a unique key. To declare a dictionary in Swift, you write the code like this:

```
var companies = ["AAPL" : "Apple Inc", "GOOG" : "Google Inc", "AMZN" :  
"Amazon.com, Inc", "FB" : "Facebook Inc"]
```

The key and value in the key-value pairs are separated by a colon. The key-value pairs are surrounded by a pair of square brackets, each of which is separated by commas.

Like array and other variables, Swift automatically detects the type of the key and value. But, if you like, you can specify the type information by using the following syntax:

```
var companies: [String: String] = ["AAPL" : "Apple Inc", "GOOG" : "Google Inc",  
"AMZN" : "Amazon.com, Inc", "FB" : "Facebook Inc"]
```

To iterate through a dictionary, use the for-in loop.

```
for (stockCode, name) in companies {  
    print("\(stockCode) = \(name)")  
}  
  
// You can also use the keys and values properties to  
// retrieve the keys and values of the dictionary.  
for stockCode in companies.keys {  
    print("Stock code = \(stockCode)")  
}  
for name in companies.values {  
    print("Company name = \(name)")  
}
```

To access the value of a particular key, specify the key using the subscript syntax. If you want to add a new key-value pair to the dictionary, simply use the key as the subscript and assign it with a value like below:

```
companies["TWTR"] = "Twitter Inc"
```

Now the `companies` dictionary contains a total of 5 items. The "TWTR":"Twitter Inc" pair is automatically added to the `companies` dictionary.

Set

A set is very similar to an array. While an array is an ordered collection, a set is an unordered collection. Items in an array can be duplicated. A set stores no repeated values.

To declare a set, you can write like this:

```
var favoriteCuisines: Set = ["Greek", "Italian", "Thai", "Japanese"]
```

The syntax is very similar to creating an array, but you have to explicitly specify the type `Set`.

As mentioned, a set is unordered collection of distinct items. If you declare a set with duplicated values, the set will not store the duplicates. Here is an example:

```
let favoriteCuisines: Set = ["Greek", "Italian", "Thai", "Japanese", "Thai", "Italian"]
```

```
{"Greek", "Italian", "Thai", "Japanese"}
```

Operations on sets are quite similar to array. You can use `for-in` loop to iterate over a set. But to add a new item to a set, you can't use the `+=` operator. You have to call the `insert` method:

```
favoriteCuisines.insert("Indian")
```

With sets, you can easily determine the values two sets have in common, or vice versa. For example, you use two sets to represent the favorite cuisines of two persons:

```
var tomsFavoriteCuisines: Set = ["Greek", "Italian", "Thai", "Japanese"]
var petersFavoriteCuisines: Set = ["Greek", "Indian", "French", "Japanese"]
```

You want to find out the common cuisines which they both love. You can call the `intersection`

method like this:

```
tomsFavoriteCuisines.intersection(petersFavoriteCuisines)
```

This will return you the result: `{"Greek", "Japanese"}`.

Or if you want to determine which cuisines they don't have in common, you can use the `symmetricDifference` method:

```
tomsFavoriteCuisines.symmetricDifference(petersFavoriteCuisines)
// Result: {"French", "Italian", "Thai", "Indian"}
```

Classes

In Objective C, you create separate interface (.h) and implementation (.m) files for classes. Swift no longer requires developers to do that. You can define classes in a single file (.swift) without separating the external interface and implementation.

To define a class, you use the `class` keyword. Here is a sample class in Swift:

```
class Recipe {
    var name: String = ""
    var duration: Int = 10
    var ingredients: [String] = ["egg"]
}
```

In the above example, we define a `Recipe` class with three properties including name duration and ingredients. Swift requires you to provide the default values of the properties. You'll end up with a compilation error if the initial values are missing.

What if you don't want to assign a default value? Swift allows you to write a question mark (?) after the type of a value to mark the value as optional.

```
class Recipe {
    var name: String?
    var duration: Int = 10
    var ingredients: [String]?
}
```

In the above code, the name and ingredients properties are automatically assigned with a

default value of `nil`. We will discuss optionals in details in later section. To create an instance of a class, just use the below syntax:

```
var recipeItem = Recipe()
// You use the dot notation to access or change the property of an instance.
recipeItem.name = "Mushroom Risotto"
recipeItem.duration = 30
recipeItem.ingredients = ["1 tbsp dried porcini mushrooms", "2 tbsp olive oil",
"1 onion, chopped", "2 garlic cloves", "350g/12oz arborio rice", "1.2 litres/2
pints hot vegetable stock", "salt and pepper", "25g/1oz butter"]
```

Swift allows you to subclass classes and adopt protocols. For example, if you have a `SimpleTableViewController` class that extends from `UIViewController` and adopts both `UITableViewDelegate` and `UITableViewDataSource` protocols, you can write the class declaration like this:

```
class SimpleTableViewController : UIViewController, UITableViewDelegate,
UITableViewDataSource
```

Methods

Swift allows you to define methods in class, structure or enumeration. I'll focus on instance methods of a class here. You can use the `func` keyword to declare a method. Here is a sample method without return value and parameters:

```
class TodoManager {
    func printWelcomeMessage() {
        print("Welcome to My TODO List")
    }
}
```

In Swift, you call a method by using dot syntax:

```
todoManager.printWelcomeMessage()
```

If you need to declare a method with parameters and return values, the method will look this:

```
class TodoManager {
    func printWelcomeMessage(name:String) -> Int {
        print("Welcome to \(name)'s TODO List")
    }
}
```



```
        return 10
    }
}
```

The syntax looks a bit awkward especially for the `->` operator. The above method takes a name parameter in String type as the input. The `->` operator is used as an indicator for method with a return value. In the above code, you specify a return type of `Int` that returns the total number of todo items. Below demonstrates how you call the method:

```
var todoManager = TodoManager()
let numberOfTodoItem = todoManager.printWelcomeMessage(name: "Simon")
print(numberOfTodoItem)
```

Control Flow

Control flow and loops employ a very C-like syntax. As you can see in the previous section, Swift provides for-in loop to iterate through arrays and dictionaries.

for loops In case you just want to iterate over a range of values, you can use the `...` or `..<` operators. These are the new operators introduced in Swift for expressing a range of values. Here is an example:

```
for i in 0..<5 {
    print("index = \(i)")
}
```

This would print out the following result in console:

```
index = 0
index = 1
index = 2
index = 3
index = 4
```

So what's the difference between `..<` and `...` ? If we replace the `..<` operator with `...` in the above example, this defines a range that runs from 0 to 5 and 5 is included in the range. Here is the console result:

```
index = 0
index = 1
```

```
index = 2
index = 3
index = 4
index = 5
```

if-else statement Just like Objective-C, you can use `if` statement to execute code based on a certain condition. The syntax of the `if-else` statement is pretty similar to that in Objective-C. Swift just makes the syntax even simpler in that you no longer need to enclose the condition within a pair of round brackets.

```
var bookPrice = 1000;
if bookPrice >= 999 {
    print("Hey, the book is expensive")
} else {
    print("Okay, I can afford it")
}
```

switch statement I'd just like to highlight the switch statement in Swift that is a dramatic improvement compared to its Objective C counterpart. Take a look at the following sample switch statement. Do you notice anything special?

```
switch recipeName {
    case "Egg Benedict":
        print("Let's cook!")
    case "Mushroom Risotto":
        print("Hmm... let me think about it")
    case "Hamburger":
        print("Love it!")
    default:
        print("Anything else")
}
```

First up, the `switch` statement can now handle strings. You can't do switching on `NSString` in Objective-C. You had to use several `if` statements to implement the above code. At last Swift brings us this most sought after utilization of switch statement.

Another interesting feature that you may notice is that there is no `break`. Recalled that in Objective-C, you need to add a `break` in every switch case. Otherwise, it will fall through to the next case. In Swift, you do not need to add the `break` statement explicitly. Switch statements in Swift do not fall through the bottom of each case and into the next one. Instead, as soon as the first matching case completes, the entire switch statement completes its execution.

On top of that, the switch statement now supports range matching. Take a look at the below code:

```
var speed = 50
switch speed {
case 0:
    print("stop")
case 0...40:
    print("slow")
case 41...70:
    print("normal")
case 71..<101:
    print("fast")
default:
    print("not classified yet")
}

// as the speed falls within the range of 41 and 70, it'll print normal to
console
```

The switch case lets you check values within a certain range by using two new operators: `...` and `..<`. Both operators serve as a shortcut for expressing a range of values.

Consider the sample range of "41...70", the `...` operator defines a range that runs from 41 to 70, including both 41 and 70. If we use the `..<` operator instead of `...` in the example, this defines a range that runs from 41 to 69. In other words, 70 is excluded from the range.

Tuples

Swift introduces an advanced type that is not available in Objective-C known as tuples. Tuples allow developers to create a group of values and pass it around. Consider that you're developing a method call to return multiple values, you can now use tuples as a return value instead of returning a custom object.

Tuples treat multiple values as a single compound value. Here is an example:

```
let company = ("AAPL", "Apple Inc", 93.5)
```

The above code creates a tuple that includes stock code, company name and stock price. As you may be aware, you're allowed to put any value of any type within a tuple. You can decompose

the values of tuples and use it like this:

```
let (stockCode, companyName, stockPrice) = company
print("stock code = \(stockCode)")
print("company name = \(companyName)")
print("stock price = \(stockPrice)")
```

A better way to use tuple is to give each element in tuple a name and you can access the element value by using dot notation. Here is another example:

```
let product = (id: "AP234", name: "iPhone 6", price: 599)
print("id = \(product.id)")
print("name = \(product.name)")
print("price = USD\(product.price)")
```

A common use of tuples is to serve as a return value. In some cases, you want to return multiple values in a method without using a custom class. You can use tuples as the return value like the following example:

```
class Store {
  func getProduct(number: Int) -> (id: String, name: String, price: Int) {
    var id = "IP435", name = "iMac", price = 1399
    switch number {
      case 1:
        id = "AP234"
        name = "iPhone 6"
        price = 599
      case 2:
        id = "PE645"
        name = "iPad Air"
        price = 499
      default:
        break
    }

    return (id, name, price)
  }
}
```

In the above code, we create a method call named `getProduct` that takes in a number and returns a product as tuple. You can call the method and store the return value like below:

```
let store = Store()
let product = store.getProduct(number: 2)
```

```
print("id = \(product.id)")
print("name = \(product.name)")
print("price = USD\(product.price)")
```

Optionals Overview

What is optional? When declaring variables in Swift, they are designated as non-optional by default. In other words, you have to assign a non-nil value to the variable. If you try to set a `nil` value to a non-optional, the compiler will say, "Nil cannot be assigned to type String!"

```
var message: String = "Swift is awesome!" // OK
message = nil // compile-time error
```

The same applies when declaring properties in a class. The properties are designated as non-optional by default.

```
class Messenger {
    var message1: String = "Swift is awesome!" // OK
    var message2: String // compile-time error
}
```

You will get a compile-time error for `message2` because it's not assigned with an initial value. If you have written in Objective-C before, you may be a bit surprised. In Objective-C, you won't get any compile-time error when assigning `nil` to a variable or declaring a property without initial value:

```
NSString *message = @"Objective-C will never die!";
message = nil;

class Messenger {
    NSString *message1 = @"Objective will never die!";
    NSString *message2;
}
```

However, it doesn't mean you can't declare a property without assigning an initial value in Swift. Swift introduces optional type to indicate the absence of a value. It is defined by adding a question mark `?` operator after the type declaration. Here is an example:

```
class Messenger {
    var message1: String = "Swift is awesome!" // OK
    var message2: String? // OK
```

```
}
```

You can still assign a value when the variable is defined as optional. However if the variable is not assigned with any value like the below code, its value automatically defaults to nil.

Why Optionals?

Swift is designed for safety. As Apple mentioned, optionals are an example of the fact that Swift is a type safe language. As you can see from the above examples, Swift's optionals provide compile-time check that would prevent some common programming errors happening at runtime. Let's go through the below example and you will have a better understanding of the power of optionals.

Consider the following method in Objective-C:

```
- (NSString *)findStockCode:(NSString *)company {
    if ([company isEqualToString:@"Apple"]) {
        return @"AAPL";
    } else if ([company isEqualToString:@"Google"]) {
        return @"GOOG";
    }

    return nil;
}
```

You can use the `findStockCode:` method to get the stock code for a particular listed company. For demo purposes, the method only returns you the stock code of Apple and Google. For other inputs, it returns `nil`.

Assuming the method is defined within the same class and we use it like this:

```
NSString *stockCode = [self findStockCode:@"Facebook"]; // nil is returned
NSString *text = @"Stock Code - ";
NSString *message = [text stringByAppendingString:stockCode]; // runtime error
NSLog(@"%@", message);
```

The code can compile correctly but as the method returns `nil` for Facebook, a runtime exception is thrown when running the app. With Swift's optionals, instead of discovering the error at runtime, it reveals the error at compile time. If we rewrite the above example in Swift, it will look like this:

```

func findStockCode(company: String) -> String? {
    if (company == "Apple") {
        return "AAPL"
    } else if (company == "Google") {
        return "GOOG"
    }

    return nil
}

var stockCode:String? = findStockCode(company: "Facebook")
let text = "Stock Code - "
let message = text + stockCode // compile-time error
print(message)

```

`stockCode` is defined as an optional. That means it can either contain a string or `nil`. You can't execute the above code because the compiler detects a potential error ("value of optional type `String?` is not unwrapped") and tells you to correct it.

As you can see from the example, Swift's optionals reinforce the nil-check and offer compile-time cues to developers. Thus, the use of optionals contributes to better code quality.

Unwrapping Optionals

So how can we make the code work? Apparently, we need to test if `stockCode` contains a nil value or not. We modify the code as follows:

```

var stockCode:String? = findStockCode(company: "Facebook")
let text = "Stock Code - "
if stockCode != nil {
    let message = text + stockCode!
    print(message)
}

```

Just like the Objective-C counterpart, we used `if` to perform the nil-check. Once we know the optional must contain a value, we unwrap it by placing an exclamation mark (`!`) to the end of the optional's name. In Swift this is known as *forced unwrapping*. You use the `!` operator to unwrap an optional and reveal the underlying value.

Referring to the above example, we only unwrap the "stockCode" optional after the nil-check. We know the optional must contain a non-nil value before unwrapping it using the `!`

operator. It is always recommended to ensure that an optional contains a value before unwrapping it.

But what if we forget the verification like below?

```
var stockCode:String? = findStockCode(company: "Facebook")
let text = "Stock Code - "
let message = text + stockCode! // runtime error
```

There will be no compile-time error. The compiler assumes that the optional contains a value as forced unwrapping is used. When you run the app, a runtime error is shown with the following message:

```
fatal error: Can't unwrap Optional.None
```

Optional Binding

Other than forced unwrapping, optional binding is a simpler and recommended way to unwrap an optional. You use optional binding to check if the optional contains a value or not. If it does contain a value, unwrap it and put it into a temporary constant or variable.

There is no better way to explain optional binding than using an example. We convert the sample code in the previous example into optional binding:

```
var stockCode:String? = findStockCode(company: "Facebook")
let text = "Stock Code - "
if let tempStockCode = stockCode {
    let message = text + tempStockCode
    print(message)
}
```

The `if let` (or `if var`) are the two keywords of optional binding. In plain English, the code says, "If `stockCode` contains a value, unwrap it, set its value to `tempStockCode` and execute the conditional block. Otherwise, just skip it the block". As `tempStockCode` is a new constant, you no longer need to use the `!` suffix to access its value.

You can further simplify the code by evaluating the function in the `if` statement:

```
let text = "Stock Code - "
```

```
if var stockCode = findStockCode(company: "Apple") {
    let message = text + stockCode
    print(message)
}
```

Here `stockCode` is not an optional. There is no need to use the `!` suffix to access its value in the conditional block. If a `nil` value is returned from the function, the block will not be executed.

Optional Chaining

Before explaining optional chaining, let's tweak the original example. We create a new class named `Stock` with the `code` and `price` properties, which are optionals. The `findStockCode` function is modified to return a `Stock` object instead of `String`.

```
class Stock {
    var code: String?
    var price: Double?
}

func findStockCode(company: String) -> Stock? {
    if (company == "Apple") {
        let aapl: Stock = Stock()
        aapl.code = "AAPL"
        aapl.price = 90.32

        return aapl
    } else if (company == "Google") {
        let goog: Stock = Stock()
        goog.code = "GOOG"
        goog.price = 556.36

        return goog
    }

    return nil
}
```

We rewrite the original example as below. We first find the stock code/symbol by calling the `findStockCode` function. And then we calculate the total cost needed when buying 100 shares of the stock.

```

if let stock = findStockCode(company: "Apple") {
    if let sharePrice = stock.price {
        let totalCost = sharePrice * 100
        print(totalCost)
    }
}

```

As the return value of `findStockCode()` is an optional, we use optional binding to check if it contains an actual value. Apparently, the price property of the `stock` class is an optional. Again we use the `if let` statement to test if `stock.price` contains a non-nil value.

The above code works without any error. Instead of writing nested `if let`, you can simplify the code by using *Optional Chaining*. The feature allows us to chain multiple optionals together with the `?.` operator. Here is the simplified version of the code:

```

if let sharePrice = findStockCode(company: "Apple")?.price {
    let totalCost = sharePrice * 100
    print(totalCost)
}

```

Optional chaining provides an alternative way to access the value of price. The code now looks a lot cleaner and simpler. Here I just cover the basics of optional chaining. You can find further information about optional chaining in Apple's Swift guide.

Failable Initializers

Swift introduced a new feature called *Failable Initializers*. Initialization is the process of providing initial values to each of the stored properties of a class. In some cases, the initialization of an instance may fail. Now such failure can be reported using a failable initializer. The resulting value of a failable initializer either contains the object or `nil`. You will need to use `if let` to check if the initialization is successful or not. Let me give you an example:

```

let myFont = UIFont(name : "AvenirNextCondensed-DemiBold", size: 22.0)

```

The initialization of the `UIFont` object will fail if the font file doesn't exist or is unreadable. This initialization failure will report using a failable initializer. The returned object is an optional that can either be the object itself or nil. Thus, we need to use `if let` to handle the

optional:

```
if let myFont = UIFont(name : "AvenirNextCondensed-DemiBold", size: 22.0) {  
    // Further processing  
}
```

Generics

The concept of Generics is not new and has been around for a long time in other programming languages like Java. But for iOS developers, you may be new to Generics.

Generic Functions Generics are one of the most powerful features of Swift and allow you to write flexible functions. So what are Generics exactly? Well, let's take a look at an example. Suppose you're developing a process function:

```
func process(a: Int, b: Int) {  
    // do something  
}
```

The function accepts two integer values for further processing. What if you need to take in other type of values like `Double` ? You probably write another function like this:

```
func process(a: Double, b: Double) {  
    // do something  
}
```

Both functions look very similar. Assuming the bodies of the functions are identical, the main difference is the types of inputs they take in. With Generics, you can simplify them into one generic function that handles multiple input types:

```
func process<T>(a: T, b: T) {  
    // do something  
}
```

Now it defines a placeholder type instead of an actual type name. The `<T>` after the function name indicates that this is a generic function. For the function arguments, the actual type name is replaced with a generic type `T`.

You can call the process function in the same way. The actual type to use in place of T will be determined each time the function is called.

```
process(a: 689, b: 167)
```

Generic Functions with Type Constraints

Let's take a look at another example. Suppose you're writing another function to compare if two integer values are equal.

```
func isEqual(a: Int, b: Int) -> Bool {  
    return a == b  
}
```

If you need to compare other types of value such as String, you'll write another function like this:

```
func isEqual(a: String, b: String) -> Bool {  
    return a == b  
}
```

With Generics, you'll combine the two functions into one:

```
func isEqual<T>(a: T, b: T) -> Bool {  
    return a == b  
}
```

Again, we use T as a placeholder of the value types. But if you test out the above code in Xcode, the function will not compile. The problem lies with the `a==b` equality check. Though the function accepts values with any types, not every type can support the equal to operator (`==`). This is why Xcode indicates an error. In this case, you need to apply a type constraint for the generic function.

```
func isEqual<T: Equatable>(a: T, b: T) -> Bool {  
    return a == b  
}
```

You write type constraint by placing a protocol constraint after a type parameter's name, separated by a colon. Here the Equatable is the protocol constraint. In other words, the

function will only accept values that support the Equatable protocol.

In Swift, it comes with a standard protocol called *Equatable*. For any types conforming to the *Equatable* protocol, they support the equal to (==) operator. All standard types like String, Int, Double support the Equatable protocol.

So you can use the `isEqual` function like this:

```
isEqual(a: 3, b: 3)           // true
isEqual(a: "test", b: "test") // true
isEqual(a: 20.3, b: 20.5)    // false
```

Generic Types

You are not limited to use Generics in functions. Swift allows you to define your own generic types. This can be custom classes or structure. The built-in Array and Dictionary are examples of generic types.

Let's take a look at the below example:

```
class IntStore {
    var items = [Int]()

    func addItem(item: Int) {
        items.append(item)
    }

    func findItemAtIndex(index: Int) -> Int {
        return items[index]
    }
}
```

The `IntStore` class is a simple class to store an array of `Int` items. It provides two methods for:

- Adding a new item to the store
- Returning a specific item from the store

Apparently, the `IntStore` class supports items in `Int` type. Wouldn't it be great if you can define a generic `ValueStore` class that manages any types of values. Here is the generic version

of the class:

```
class ValueStore<T> {
    var items = [T]()

    func addItem(item: T) {
        items.append(item)
    }

    func findItemAtIndex(index: Int) -> T {
        return items[index]
    }
}
```

Like what you have learned in the Generic functions section, you use a placeholder type parameter (T) to indicate a generic type. The type parameter () after the class name indicates the class is a generic type.

To instantiate the class, you write the type to be stored in the `ValueStore` within angle brackets.

```
var store = ValueStore<String>()
store.addItem(item: "This")
store.addItem(item: "is")
store.addItem(item: "generic")
store.addItem(item: "type")
let value = store.findItemAtIndex(index: 1)
```

You can call the method the same way as before.

Computed Properties

A computed property does not actually store a value. Instead, it provides its own getter and setter to compute the value. Here is an example:

```
class Hotel {
    var roomCount: Int
    var roomPrice: Int
    var totalPrice: Int {
        get {
            return roomCount * roomPrice
        }
    }
}
```



```

    init(roomCount: Int = 10, roomPrice: Int = 100) {
        self.roomCount = roomCount
        self.roomPrice = roomPrice
    }
}

```

The `Hotel` class has two stored properties: `roomPrice` and `roomCount`. To calculate the total price of a hotel, we can simply multiply `roomPrice` by `roomCount`. In the past, you might create a method that performs the calculation and returns the total price. With Swift, you can use computed properties instead. In the example, `totalPrice` is a computed property. Rather than storing a fixed value, it defines a custom getter that actually performs the calculation and returns the total price of the rooms. Just like stored properties, you can access the computed property through dotted syntax:

```

let hotel = Hotel(roomCount: 30, roomPrice: 100)
print("Total price: \(hotel.totalPrice)")
// Total price: 3000

```

Optionally, you can define a custom setter for the computed property. Consider the same example again:

```

class Hotel {
    var roomCount: Int
    var roomPrice: Int
    var totalPrice: Int {
        get {
            return roomCount * roomPrice
        }

        set {
            let newRoomPrice = Int(newValue / roomCount)
            roomPrice = newRoomPrice
        }
    }

    init(roomCount: Int = 10, roomPrice: Int = 100) {
        self.roomCount = roomCount
        self.roomPrice = roomPrice
    }
}

```

Here we define a custom setter to calculate the new room price when the value of total price is

updated. When a new value of `totalPrice` is set, a default name of `newValue` can be used in the setter. Base on `newValue`, you can then perform the calculation and update the `roomPrice` accordingly.

Could you use methods instead of computed properties? Sure. To me, it is a matter of coding style. Computed properties are especially useful for performing simple conversions and calculations. As you can see from the above example, the implementation is much cleaner.

Property Observers

Property observers are one of my favorite features of Swift. Property observers observe and respond to changes in a property's value. The observers are called every time a property's value is set. You have the option to define two kinds of observers on a property:

- `willSet` is called just before the value is stored.
- `didSet` is called immediately after the new value is stored.

Consider the `Hotel` class again. For instance, we want to limit the room price to a thousand dollars. Whenever a caller sets the room price to a value larger than 1000, we will set it to 1000. You can then use a property observer to monitor the value change like this:

```
class Hotel {
    var roomCount: Int
    var roomPrice: Int {
        didSet {
            if roomPrice > 1000 {
                roomPrice = 1000
            }
        }
    }

    var totalPrice: Int {
        get {
            return roomCount * roomPrice
        }

        set {
            let newRoomPrice = Int(newValue / roomCount)
            roomPrice = newRoomPrice
        }
    }
}
```

```

init(roomCount: Int = 10, roomPrice: Int = 100) {
    self.roomCount = roomCount
    self.roomPrice = roomPrice
}
}

```

Say, you set the `roomPrice` property to `2000`. The `didSet` observer will be called and perform the validation. Because the value is larger than `1000`, the room price is then set it to `1000`. As you can see, property observers are particularly useful for value change notifications.

Failable Casts

`as!` (or `as?`) is known as a failable cast operator. You have to either use `as!` or `as?` to downcast an object to a subclass type. If you're quite sure that the downcasting will succeed, you can use `as!` to force the casting. Here is an example:

```

let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for:
indexPath) as! RestaurantTableViewCell

```

If you're not sure if the casting will succeed, just use the `as?` operator. By using `as?`, it returns an optional value, but in case the downcasting fails, the value will be `nil`.

repeat-while

Starting from Swift 2, Apple introduces a new control flow operator called `repeat-while`, which is to replace the classic `do-while` loop. Here is an example:

```

var i = 0
repeat {
    i += 1
    print(i)
} while i < 10

```

`repeat-while` evaluates its condition at the end of each pass through the loop. If the condition is `true`, it repeats the block of code again. It exits the loop when the condition evaluates to `false`.

for-in where Clauses

Not only can you iterate over all items in an array using `for-in` loop, you can define a condition to filter the items using the `where` clause. When you loop through an array, for example, only those items that meet the criteria will be processed.

```
let numbers = [20, 18, 39, 49, 68, 230, 499, 238, 239, 723, 332]
for number in numbers where number > 100 {
    print(number)
}
```

In the above example, it only prints out those numbers that are greater than 100.

Guard

The `guard` keyword was first introduced in Swift version 2. According to Apple's documentation, guard is described like this:

A guard statement, like an if statement, executes statements depending on the Boolean value of an expression. You use a guard statement to require that a condition must be true in order for the code after the guard statement to be executed.

Before I further explain the guard statement, let's go straight to the following example:

```
struct Article {
    var title:String?
    var description:String?
    var author:String?
    var totalWords:Int?
}

func printInfo(article: Article) {
    if let totalWords = article.totalWords where totalWords > 1000 {
        if let title = article.title {
            print("Title: \(title)")
        } else {
            print("Error: Couldn't print the title of the article!")
        }
    } else {
        print("Error: It only works for article with more than 1000
words.")
    }
}

let sampleArticle = Article(title: "Swift Guide", description: "A beginner's
```

```
guide to Swift 2", author: "Simon Ng", totalWords: 1500)
printInfo(sampleArticle)
```

In the above code, we create a `printInfo` function to display the title of an article. However, we will only print the information for an article with more than a thousand words. As the variables are optionals, we use `if let` to verify if the optional contains a value or not. If the optional is `nil`, we display an error message. If you execute the code in Playgrounds, it should display the title of the article.

In general, the `if-else` statements follow this pattern:

```
if some conditions are met {
    // do something
    if some conditions are met {
        // do something
    } else {
        // show errors or performs other operations
    }
} else {
    // show errors or performs other operations
}
```

As you may notice, if you have to test more conditions, it will be nested with more conditions. There is nothing wrong with that programmatically. But in terms of readability, your code will get messy if there are a lot of nested conditions.

This is where the `guard` statement comes in. The syntax of `guard` looks like this:

```
guard else {
    // what to do if the condition is not met
}
// continue to perform normal actions
```

If the condition, defined in the `guard` statement is not met, the code inside the `else` branch is executed. On the other hand, if the condition is met, it skips the `else` clause and continues the code execution.

If you rewrite the sample code using `guard`, it is a lot cleaner:

```
func printInfo(article: Article) {
    guard let totalWords = article.totalWords, totalWords > 1000 else {
```

```

        print("Error: It only works for article with more than 1000 words.")
        return
    }

    guard let title = article.title else {
        print("Error: Couldn't print the title of the article!")
        return
    }

    print("Title: \(title)")
}

```

With `guard`, you focus on handling the condition you don't want. Furthermore, it forces you to handle one case at a time, avoiding nested conditions. Thus, the code is cleaner and easier to read.

Error Handling

When developing an app or any programs, you'll need to handle every possible scenarios, whether it's good or bad. Obviously, things may go wrong. Say, if you're developing an app that connects to a cloud server, your app has to deal with situations where the Internet connection is unavailable or the cloud server is failed to connect.

In the older version of Swift, it lacks a proper error handling model. As an example, you handle error conditions like this:

```

let request = URLRequest(URL: NSURL(string: "http://www.apple.com")!)
var response: NSURLResponse?
var error: NSError?
let data = NSURLConnection.sendSynchronousRequest(request, returningResponse:
&response, error: &error)

if error == nil {
    print(response)
    // Parse the data
} else {
    // Handle error
}

```

When calling a method that may cause a failure, you normally pass it with an `NSError` object (as a pointer). If there is an error, the object will be set with the corresponding error. You then

check if the error object is nil or not and respond to the error accordingly.

That's how you handle errors in Swift 1.2.

Note: `NSURLConnection.sendSynchronousRequest()` has been deprecated in iOS 9. As most readers are familiar with the usage, it is used in the example.

try / throw / catch

Beginning from Swift 2, it comes with an exception-like model using `try-throw-catch` keywords. The same code snippet will become this:

```
let request = URLRequest(url: URL(string: "https://www.apple.com")!)
var response: URLResponse?
do {
    let data = try NSURLConnection.sendSynchronousRequest(request, returning:
&response)
    print(response)

    // Parse the data
} catch {
    // handle error
    print(error)
}
```

Now you use `do-catch` statement to catch errors and handle them accordingly. As you may notice, we put a `try` keyword in front of the method call. With the introduction of the error handling model, some methods can throw errors to indicate failures. When we invoke a throwing method, you will need to put a `try` keyword in front of it.

How do you know if a method throws an error? As you type the method in the built-in editor, the throwing methods are indicated by the `throws` keyword.

```
let data = NSString(contentsOfFile: String, encoding: UInt)
```

M NSString (contentsOfFile: String, encoding: UInt) throws

M NSString (contentsOfFile: String, usedEncoding: UnsafeMutablePointer<UInt>) throws

M NSString (contentsOfURL: NSURL, encoding: UInt) throws

M NSString (contentsOfURL: NSURL, usedEncoding: UnsafeMutablePointer<UInt>) throws

Returns an NSString object initialized by reading data from the file at a given path using a given encoding.

[More...](#)

Now that you should understand how to call a throwing method and catch the errors, how do you indicate a method or function that can throw an error?

Imagine you're modelling a lite shopping cart. Customers can use the cart to temporarily store and checkout their items, but this cart will throw an error for the following conditions:

- The shopping cart can only store a maximum of 5 items. Otherwise, it throws a `cartIsFull` error.
- There must be at least one item in the shopping cart during checkout. Otherwise, it throws a `cartIsEmpty` error.

In Swift, errors are represented by values of types conforming to the `Error` protocol.

Note: The protocol `ErrorType` in Swift 2 has been changed to `Error` in Swift 3.

Usually you use an enumeration to model the error conditions. In this case, you can create an enumeration that adopts `Error` like this for the shopping cart errors:

```
enum ShoppingCartError: Error {
    case cartIsFull
    case emptyCart
}
```

For the shopping cart, we create a `LiteShoppingCart` class to model its functions. Here is a sample code snippet:

```
struct Item {
    var price:Double
    var name:String
}

class LiteShoppingCart {
    var items:[Item] = []

    func addItem(item:Item) throws {
        guard items.count < 5 else {
            throw ShoppingCartError.cartIsFull
        }

        items.append(item)
    }
}
```

```

func checkout() throws {
    guard items.count > 0 else {
        throw ShoppingCartError.emptyCart
    }
    // Continue with the checkout
}
}

```

If you take a closer look at the `addItem` method, you probably notice the `throws` keyword. We append the `throws` keyword in the method declaration to indicate that the method can throw an error. In the implementation, we use `guard` to ensure the total number of items is less than 5. Otherwise, we throw the `ShoppingCartError.cartIsFull` error.

To throw an error, you just write the `throw` keyword, followed by the actual error. For the `checkout` method, we have a similar implementation. If the cart does not contain any items, we throw the `ShoppingCartError.emptyCart` error.

Now, let's see what happens when performing a checkout on an empty cart. I recommend you to fire up Xcode and use Playgrounds to test out the code.

```

let shoppingCart = LiteShoppingCart()
do {
    try shoppingCart.checkout()
    print("Successfully checked out the items!")
} catch ShoppingCartError.cartIsFull {
    print("Couldn't add new items because the cart is full")
} catch ShoppingCartError.emptyCart {
    print("The shopping cart is empty!")
} catch {
    print(error)
}

```

As the `checkout` method can throw an error, we use the `do-catch` statement to catch the errors. If you execute the above code in Playgrounds, it will catch the `ShoppingCartError.emptyCart` error and print the error message accordingly, because we haven't added any items.

Now insert the following code in the `do` clause, right before calling the `checkout` method:

```

try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #1"))
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #2"))
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #3"))

```

```
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #4"))
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #5"))
try shoppingCart.addItem(item: Item(price: 100.0, name: "Product #6"))
```

Here we try to add a total of 6 items to the `shoppingCart` object. Again, it will throw an error because the shopping cart cannot hold more than 5 items.

When catching errors, you can identify the exact error (e.g. `ShoppingCartError.cartIsFull`) to match, so you can provide very specific error handling.

If you do not specify a pattern in the `catch` clause, Swift will match any error and automatically bind the error to the error constant. As a best practice, you should try to catch the specific errors that are thrown by the throwing method. At the same time, you should write a `catch` clause that matches any errors. This ensures all possible errors are handled.

Availability Checking

If all users are forced to upgrade to the latest version of iOS, this would make our developers' life much easier. In reality, however, your apps have to cater for different versions of iOS (e.g. iOS 8, 9 and iOS 10). If you just use the latest version of APIs in your app, this may cause errors when the app runs on older versions of iOS. When using an API that is only available on the latest version of iOS, you will need to do some kinds of verification before using the class or calling the method.

Prior to Swift 2, there is no standard way to do availability check. As an example, the `UNMutableNotificationContent` class is only available on iOS 10. If you use the class on older versions of iOS, you'll end up with an error and that probably causes an app crash. To prevent the error, you may perform the availability check like this:

```
if (NSClassFromString("UNMutableNotificationContent") != nil) {
} else {
}
```

This is one way to check if the class exists. Starting from Swift 2, it has built-in support for checking API availability. You can easily define an availability condition so that the block of

code will only be executed on certain iOS versions. Here is an example:

```
if #available(iOS 10.0, *) {
    // iOS 10 or up
    let content = UNMutableNotificationContent()
} else {
    // Fallback on earlier versions
}
```

You use the `#available` keyword in a `if` statement. In the availability condition, you specify the OS versions (e.g. iOS 10) you want to verify. The `*` is required and indicates that the `if` clause is executed on the minimum deployment target and any other versions of OS. For the above example, the body of the `if` will be executed on iOS 10 or up, and other platforms such as watchOS.

Similarly, you can use `guard` instead of `if` for checking API availability. Here is another example:

```
guard #available(iOS 10.0, *) else {
    // what to do if it doesn't meet the minimum OS requirement
    return
}

let content = UNMutableNotificationContent()
```

What if you want to develop a class or method, which is available to certain versions of OS? Swift lets you apply the `@available` attribute on classes/methods/functions to specify the platform and OS version you want to target. Say, for example, you're developing a class named `SuperFancy` and it is only available for iOS 10 or later. You can apply `@available` like this:

```
@available(iOS 10.0, *)
class SuperFancy {
    // implementation
}
```

If you try to use the class on Xcode projects that supports multiple versions of iOS, Xcode will show you the errors below:

```
if #available(iOS 10.0, *) {  
    = SuperFancy()  
}
```

'SuperFancy' is only available on iOS 10.0 or newer

- Fix-it Add 'if #available' version check
- Fix-it Add @available attribute to enclosing instance method
- Fix-it Add @available attribute to enclosing class

```
ing() {
```

Note: You cannot test availability check in Playgrounds. If you want to give it a try, create a new Xcode project to test out the feature.