Craig Clayton

iOS 10 Programming for Beginners

Begin your iOS mobile application development journey with this accessible, practical guide



Packt

iOS 10 Programming for Beginners

Begin your iOS mobile application development journey with this accessible, practical guide

Craig Clayton



BIRMINGHAM - MUMBAI

iOS 10 Programming for Beginners

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2016

Production reference: 1191216

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78646-450-7

www.packtpub.com

Credits

Author
Craig Clayton

Reviewer
Fernando Rodríguez

Commissioning Editor
Ashwin Nair

Acquisition Editor
Reshma Raman

Content Development Editor
Divij Kotian

Technical Editor
Gebin George

Copy Editor
Charlotte Carneiro

Project Coordinator
Sheejal Shah

Proofreader
Safis Editing

Indexer
Tejal Daruwale Soni

Graphics
Jason Monteiro

Production Coordinator
Melwyn Dsa

About the Author

Craig Clayton is a self-taught, Senior iOS Engineer at Adept Mobile, which specializes in building mobile experiences primarily for NBA and NFL teams. He also volunteers as the organizer of the Suncoast iOS meetup group in the Tampa/St. Petersburg area, and prepares presentations and hands-on talks for the group as well as for other groups in the community.

He has worked with both adults and kids who wish to start learning how to program, or those who aspire to become iOS developers. On top of all that, starting in 2017 Craig has plans to launch Cocoa Academy online, which will specialize in bringing a diverse list of iOS courses. The courses will range from building apps to games for all programming levels.

I would like to thank my friends and family for their support, especially my mom, Corliss Smith. I hope that one day I will get to be there for you when you write your book. I would also like to thank Kim Smallman; you were amazing with your help and guidance in every step of the way. Since you are not a programmer, I really appreciated getting your feedback on things that beginners might not fully understand.

About the Reviewer

Fernando Rodríguez has more than 20 years of experience in developing and teaching other developers. Although currently specialized in the Apple stack of tools, he's a nerd of all trades with a strong interest in Big Data and automated trading.

Fernando has taught iOS development at the Big Nerd Ranch, Udacity, and Keep coding, ranging from Facebook developers to indie devs.

He was awarded for being an outstanding instructor at Udemy for his intro course to iOS development. This course was mentioned in the Financial Times, Venture beat, and Information Week.

I'd like to thank Sheejal Shah from Packt for her patience and Craig Clayton for his hard work.

---

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thank you for purchasing this Packt book. We take our commitment to improving our content and products to meet your needs seriously – that's why your feedback is so valuable. Whatever your feelings about your purchase, please consider leaving a review on this book's Amazon page. Not only will this help us, more importantly it will also help others in the community to make an informed decision about the resources that they invest in to learn.

You can also review for us on a regular basis by joining our reviewers club. **If you're interested in joining, or would like to learn more about the benefits we offer, please contact us:** `customerreviews@packtpub.com`.

Table of Contents

Preface **ix**

Chapter 1: Getting Familiar with Xcode **1**

Getting started **2**

The Xcode interface **5**

Navigator panel 6

Standard editor 6

Utilities panel 7

Debug panel 7

Toolbar 7

Generic iOS Device 10

iOS device 10

Window Pane Controls 14

Summary **16**

Chapter 2: Building a Foundation with Swift **17**

Playgrounds – an interactive coding environment **18**

Data types – where it all starts **20**

String 20

Integer data type 21

Floating-point numbers 21

Booleans 21

Variables and constants – where data is held 22

Creating a variable with a String 22

Creating a variable with an Integer (Int) 22

Debug and print() – detecting your bugs 23

Adding floating-point numbers 24

Creating a Boolean 25

Hungarian notation 27

Why constants versus variables? 28

Comments – leaving yourself notes or reminders **28**

Type safety and type inference **29**
Concatenating strings 29
String interpolation 30
Operations with our Integers **31**
Increment and decrement 33
Comparison operators 34
If-Statements – having fun with logic statements 35
Optionals and Optional Bindings **42**
Why optionals? 45
Functions **46**
Let's Work **52**
Summary **52**
Chapter 3: Digging Deeper **53**
Creating a Playground project **54**
Ranges **55**
Closed Range 55
Half closed Range 56
Control flow **56**
The for…in loop 57
The while loop 61
The repeat…while loop 63
Summary **65**
Chapter 4: Digging into Collections **67**
Arrays **67**
Creating an empty array 68
Creating an array with initial values 69
Creating a mutable array 69
Adding items to an array 70
Checking the number of elements in an array 73
Checking for an empty array 73
Retrieving a value from an array 75
Iterating over an array 78
Removing items from an array 79
Dictionaries **81**
Creating a dictionary 82
Adding and updating dictionary elements 83
Accessing an item in a dictionary 86
Iterating over dictionary values 86
Iterating over dictionary keys 88
Iterating over dictionary keys and values 89

Checking the number of items in a dictionary 90
Removing Items from a dictionary 92
Sets 93
Creating an empty set 94
Creating a set with an array literal 94
Creating a mutable set 95
Adding items into a set 95
Checking if a set contains an item 96
Iterating over a set 97
Intersecting two sets 99
Joining two sets 100
Removing items from a Set 100
Summary 101
Chapter 5: Starting the UI Setup 103
Useful terms 103
View Controller 104
Table View Controller 104
Collection View Controller 105
Navigation Controller 105
Tab Bar Controller 106
Storyboard 106
Segue 107
Auto layout 108
Model View Controller (MVC) 108
App Tour 108
Explore tab 109
Locations 110
Restaurant listings 111
Restaurant detail 112
Map tab 112
Project setup 113
Creating a new project 114
Creating our files 116
Storyboard setup 118
Adding our App assets 122
Storyboards 131
Creating our launch screen 132
Adding a Navigation Controller 140
Creating a custom title view 142
Adding a container 142
Using Stack Views 145

Adding a Custom Label and an Arrow to Our Custom Title View 147
Summary 151
Chapter 6: Setting Up UI 153
Design clean up 154
Adding a Clear Background to the Custom Title View 154
Updating the UIStackView 157
Updating our Arrow 158
Updating our Label 159
Adding a Button 162
Collection View 167
Adding a Modal 169
Map Kit View 171
Fixing the Crash 174
Refactoring the Storyboard 177
Creating a New Storyboard for the Explore Tab 178
Creating a New Storyboard for the Map Tab 181
Folder Setup 181
Organizing folders 182
Setting up Global Settings 184
Breaking Down our App Delegate 185
Summary 190
Chapter 7: Getting Started with the Grid 191
Understanding the Model View Controller architecture 192
Model 192
View 192
Controller 193
Getting familiar with the setup 193
Classes and Structures 193
Controllers and Classes 199
Creating our Controller 200
Understanding Collection View Controllers and Collection View Cells 205
Getting Data into Collection View 208
Understanding the Data Source 209
Updating the Grid 211
Model 216
ExploreData.plist 217
ExploreItem.swift 217
ExploreDataManager.Swift 220
Getting Data 223
CollectionView Cell 225
Connecting to Our Cell 227
Hooking up Our UI with IBOutlets 228

Cell Selection 232
Restaurant Listing 233
Summary 239
Chapter 8: Getting Started with the List 241
Updating UI in Storyboard 242
Updating Bar Button Items 245
Unwinding our cancel button 246
Adding Our First Table View 248
Updating Our Edges 249
Creating Our View Controller Class 250
Connecting our TableView with our Location View Controller 252
Adding the Data Source and Delegate 253
Creating a Prototype Cell 254
Digging into Our Table View code 255
Adding Locations to Our Table View 258
Creating Our First Property List (plist) 260
Adding Data to Our Property List 262
Creating Our Location Data Manager 264
Working with Our Data Manager 265
Summary 266
Chapter 9: Working More with Lists 269
Creating our Restaurant detail 269
Setting up our static Table View 272
Exploring Restaurant details 274
Creating our section headers 276
Adding our labels 277
Address section 286
Creating Reviews 291
Reservations 301
Adding reservation times 301
Reservation information 303
Reservation header 306
Summary 312
Chapter 10: Where Are We? 313
Setting up map annotations 313
What is an MKAnnotation? 314
Creating a restaurant annotation 314
Creating our Map Data Manager 318
Creating a base class 321
Refactoring ExploreDataManager 323

Creating and adding annotations **324**
Creating our Map View Controller 324
Creating custom annotations 328
Refactoring restaurant detail **332**
Creating a Storyboard reference 332
Map to restaurant detail 336
Passing data to Restaurant detail 337
Organizing your code **344**
Summary **352**
Chapter 11: Where's My Data? **353**
Creating an API Manager **353**
What is an API? 354
Understanding a JSON file 354
Exploring the API Manager file 355
Location list **357**
Selecting a location 358
Passing a selected location back to Explore View 359
Getting the last selected location **361**
Passing location and cuisine to the restaurant list 365
Building our restaurant list **371**
Updating our background 371
Updating our restaurant list cell 372
Positioning elements in our restaurant list cell 375
Adding auto layout to our restaurant list cell 377
Creating our restaurant cell class 379
Setting up restaurant list cell outlets **380**
Creating RestaurantDataManager 381
Displaying data in Restaurant list cell 385
Restaurant details **386**
Displaying data in the Restaurant Detail view 386
Passing data to our Restaurant List View Controller 395
Map update **397**
Challenge yourself **398**
Summary **399**
Chapter 12: Foodie Reviews **401**
Getting started with reviews **401**
Setting up our table view controllers 401
Creating reviews **406**
Setting up the Review Storyboard 406
Updating the Review Cells 408

Positioning UI elements 409
Adding Auto Layout for creating reviews 411
Adding Ratings View 414
Adding Auto Layout for Ratings View 416
Adding our Photo Filter View 417
Adding Auto Layout for the Photo Filter View 419
Presenting our Views as Modals 420
Setting up our unwind segues 422
Hooking up our unwind segues 423
Working with filters 426
Creating our Filter Scroller 431
Creating our apply Filter View Controller 435
Creating review images 441
Summary 451
Chapter 13: Saving Reviews 453
What is Core Data? 453
NSManagedObjectModel 454
NSManagedObjectContext 454
NSPersistentStoreCoordinator 455
Creating a data model 455
Entity auto-generation 463
Review item 464
Core Data Manager 465
Creating star ratings 471
Setting up the cell UI 495
Adding Auto Layout 498
Adding Review List extension 501
Summary 504
Chapter 14: Universal 505
Explore 505
Restaurant listing 511
Updating restaurant details 517
Updating the header layout 518
Updating the table details section layout 521
Updating the No Reviews Layout 522
Updating the Reviews layout 526
Updating the Map section layout 530
Summary 534

Chapter 15: iMessages **535**
Understanding iMessages **536**
Creating our extension 536
Updating our assets 539
Implementing our Messages UI 540
Adding Auto Layout to our cell 543
Creating a framework **544**
Connecting our message cell 552
Showing restaurants 554
Sending reservations 557
Summary **560**
Chapter 16: Notifications **561**
Starting with the basics **562**
Getting permission 562
Setting up notifications 563
Showing notifications 566
Customizing our notifications **569**
Embedding images 569
Adding buttons 572
Custom UI in notifications **575**
Summary **579**
Chapter 17: Just a Peek **581**
Adding 3D Touch quick actions **581**
Adding favorites **593**
Creating a new model object 594
Updating our Core Data manager 599
Summary **605**
Chapter 18: Beta and Store Submission **607**
Creating a bundle identifier **608**
Creating a Certificate Signing Request **614**
Creating production and development certificates **618**
Creating a Production Provisioning Profile **624**
Creating a Development Provisioning Profile **629**
Creating the App Store Listing **632**
Creating an archive build **634**
Internal and External Testing **639**
Internal testing 640
External testing 643
Summary **646**
Index **649**

Preface

In this book, we will build a Restaurant Reservation app called Let's Eat. We will start the book off by exploring Xcode, our programming environment, which is also known as **Interface Development Environment** (**IDE**). Next, you will start learning the foundations of Swift, the programming language used in iOS apps. Once we are comfortable with the basics of Swift, we will dig deeper to build a more solid foundation.

After we have a solid foundation of using Swift, we will start creating the visual aspects of our Let's Eat app. During this process, we will work with storyboards and connect our apps structure together using segues. With our UI complete, we will go over the different ways that we can display data. To display our data in a grid, we will use Collection Views, and to display our data in a list, we will use Table Views.

We will also look at how to add basic and custom annotations on to a map. Finally, it's time to get real data; we will look at what an **Application Programming Interface** (**API**) is and how we can get real restaurant data into our Collection Views, Table Views, and Map.

We now have a complete app, but what about adding some bells and whistles? The first place we can add a feature will be on the restaurant detail page where we can add restaurant reviews. Here, users will be able to take or choose a picture and apply a filter on to their picture. They will also be able to give the restaurant a rating as well as a review. When they are done, we will save this data using Core Data.

Since we built our app to work on both iPhone and iPad, we should add the ability to make our app support iPad Multitasking. Doing this will allow our app to be open alongside another app at the same time.

If we want to be able to send our reservation to a friend, we can create a custom UI for iMessages, that will send them the details for the reservation along with the app it came from. The one thing missing from our app is the ability to notify the user with a custom notification to alert when they have an upcoming reservation.

Finally, let's create some quick access for our app using 3D touch, where by tapping our app icon, the user can quickly jump to their reservations. Now that we have added some bells and whistles, let's get this app to our friends using TestFlight, and finally get it into the App Store.

What this book covers

Chapter 1, *Getting Familiar with Xcode*, will take us through a tour of Xcode and talk about all of the different panels we will use throughout the book.

Chapter 2, *Building a Foundation with Swift*, deals with the basics of Swift.

Chapter 3, *Digging Deeper*, teaches us to build on our Swift foundation and learn some more basics of Swift.

Chapter 4, *Digging into Collections*, will talk about the different types of Collections.

Chapter 5, *Starting the UI Setup*, is about building the Let's Eat app. We will focus on getting our structure setup using storyboards.

Chapter 6, *Setting Up UI*, deals with working on our Let's Eat app in a storyboard.

Chapter 7, *Getting Started with the Grid*, is about working with Collection Views and how we can use them to display a grid of items.

Chapter 8, *Getting Started with the List*, teaches us to work with Table View and takes a deep look at dynamic Table Views.

Chapter 9, *Working More with Lists*, will talk about working with Table Views, but we will look at static Table Views.

Chapter 10, *Where Are We?*, deals with working with MapKit and learning how to add annotations to a map. We will also create custom annotations for our map.

Chapter 11, *Where's My Data?*, is about learning how to use a JSON API within our app.

Chapter 12, *Foodie Reviews*, talks about working with the phone's camera and library. We will then look at how to apply filters to our photos.

Chapter 13, *Saving Reviews*, wraps up Reviews by saving them using Core Data.

Chapter 14, *Universal*, deals with multitasking on the iPad, and how we can get an update to be supported on all devices.

Chapter 15, *iMessages*, is about building a custom message app UI. We will also create a framework to share data between both apps.

Chapter 16, *Notifications*, provides learning on how to build basic notifications. Then, we will look at embedding images into our notifications as well as building a custom UI.

Chapter 17, *Just a Peek*, looks at 3D touch and how to add quick actions to our app. We will also look at how we can add peek and pop to our restaurant list.

Chapter 18, *Beta and Store Submission*, is about how to submit apps for testing as well as submitting apps to the App Store.

What you need for this book

You will need a computer that runs Xcode 8 or greater.

Who this book is for

This book is for beginners who want to be able to create iOS applications. If you have some programming experience, this book is a great way to get a full understanding of how to create an iOS application from scratch and submit it to the App Store. You do not need any knowledge of Swift or any prior programming experience.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You can add comments to your code, such as a `TODO` item or just a brief explanation of what something is doing."

A block of code is set as follows:

```
let arrOfInts:[Int] = []
let arrStrings = [String]()
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Once installed, launch Xcode, and you should see the following **Welcome to Xcode** screen."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading theexample code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enterthe name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the**Code Files**button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged into your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/iOS-10-Programming-for-Beginners`.We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/iOS10ProgrammingforBeginners_ColoredImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Familiar with Xcode

So, you want to get into iOS development? I was in your shoes on January 27th, 2010, when Apple first announced the iPad. Literally as soon as the conference was over, I knew that I wanted to learn how to create apps for the iPad. I signed up on the Apple Developer website and paid my $99 annual fee. But then, I realized that I did not know where to begin. A large variety of instructional books or videos did not exist, especially since the iPad had not yet been released. I had previous programming experience—however, I had no idea how to write Objective-C (the original programming language for iOS). Therefore, I had to teach myself the basics. In this book, we will learn together what it takes to become an iOS developer.

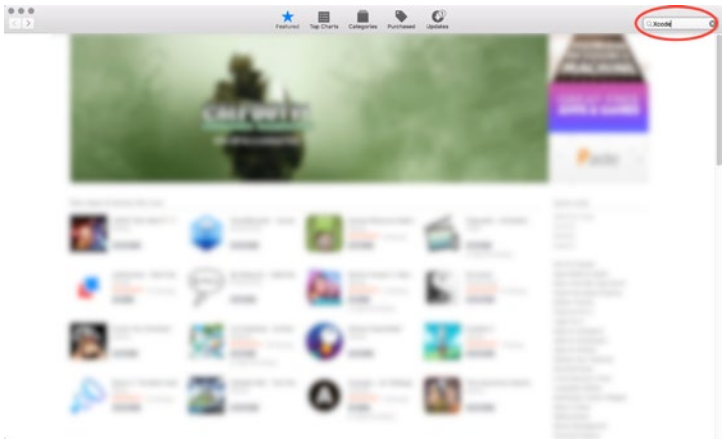
If you are new to programming, take your time. You should understand the lessons in one chapter before moving on to the next chapter. These important skills will set you up with a solid foundation in iOS development. If you have previous programming experience, you should still review the earlier chapters, as they will be a refresher for you.

Throughout this book, we will work in Xcode, specifically Xcode 8 (and Swift 3, which we will tackle later in the book). Xcode is known as an **Integrated Development Environment** (**IDE**). Using Xcode gives us everything we will need in order to build apps for iOS, tvOS, macOS (formerly, OS X), and watchOS. In this chapter, we will explore Xcode in order to help you get more comfortable using it. If you are not on Xcode 8, make sure to update Xcode, as the code in this book will not run properly otherwise.

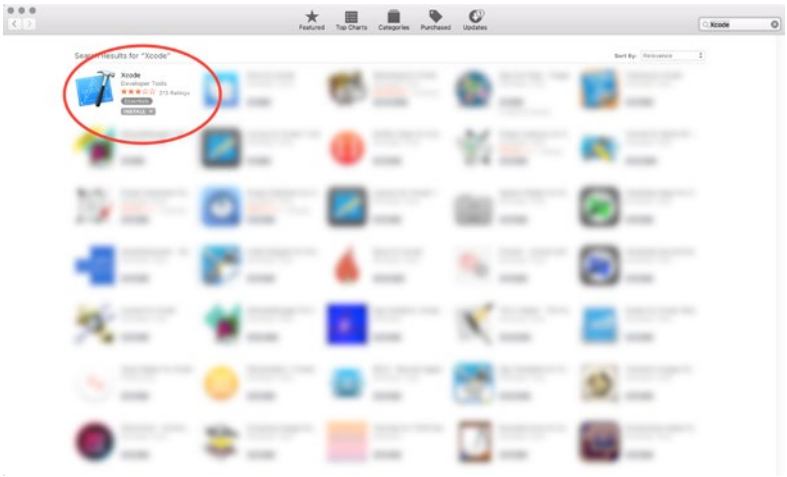
Our focus in this book will be to create a universal iOS app (an app for both the iPhone and iPad). The best way to do this is to create a project to familiarize yourself with where everything is and how to find what you need. So, let's first download and install Xcode.

Getting started

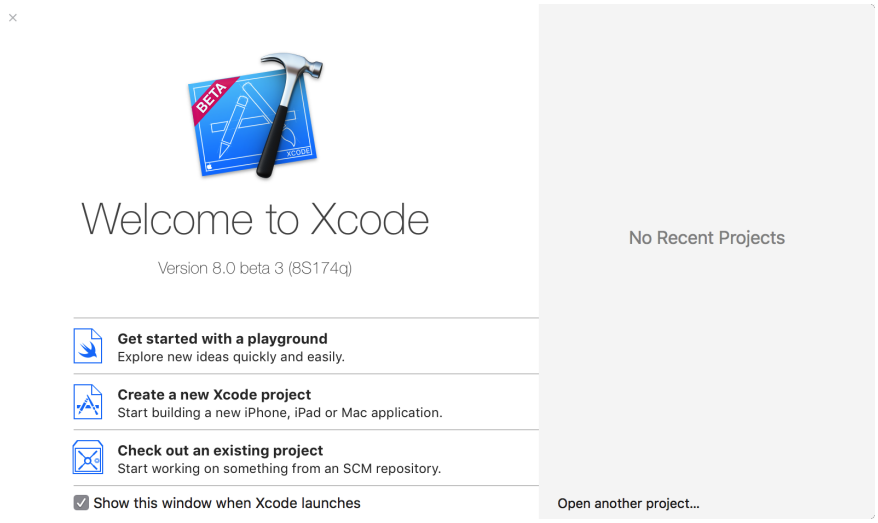
In order to download Xcode, launch the App Store on your Mac and then type `Xcode` into the search bar in the upper-right corner:



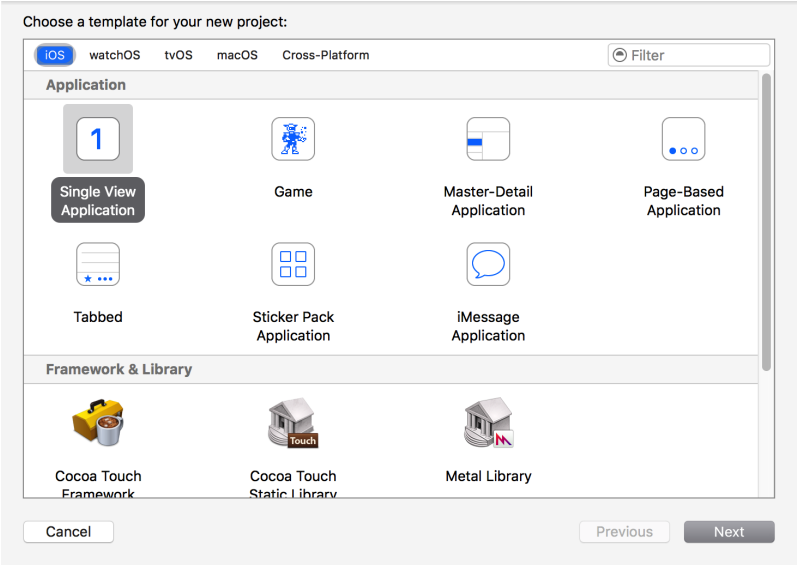
Next, click on **INSTALL**:



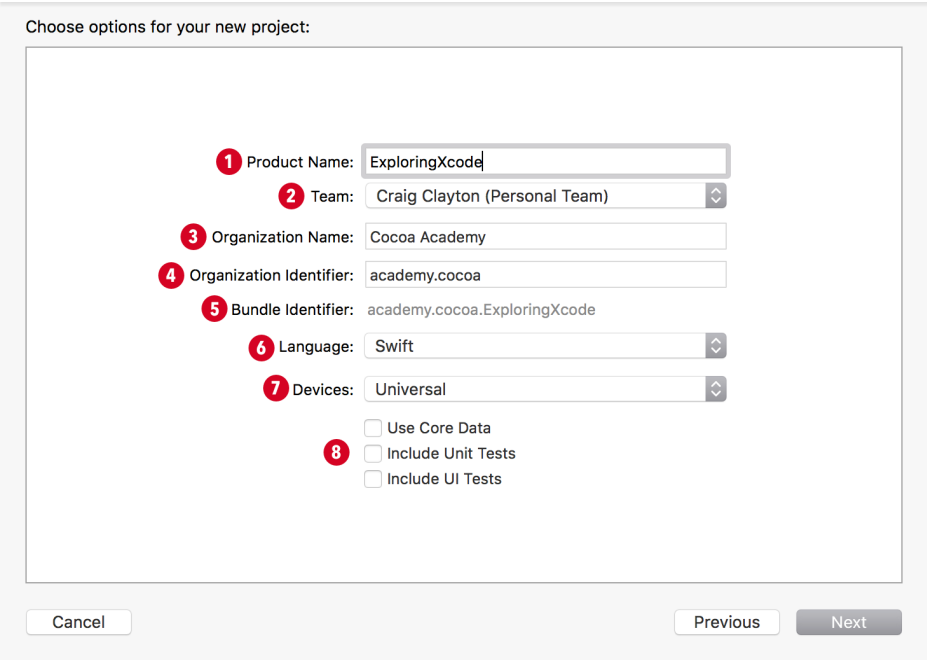
Once installed, launch Xcode, and you should see the following **Welcome to Xcode** screen:



If this is the first time you have launched Xcode, then you will see **No Recent Projects** in the right panel. If you have previously created projects, then you will see those listed to the right. To get started, we are going to click on **Create a new Xcode project** in the left panel of the welcome screen. This will take us to the new project screen:



Across the top of this screen, you can select one of the following items—**iOS**, **watchOS**, **tvOS**, **macOS**, and **Cross-Platform**. Since we are creating apps for iOS, make sure that you have **iOS** selected. Then, select **Single View Application** and click on **Next**. Now, you will see an options screen for a new project:



This option screen has the following eight items to complete or choose:

1. **Product Name**: The product name is your app. We are going to set ours as `ExploringXcode`.

2. **Team**: The team is connected to your Apple account. We are going to ignore this for now, because it is not needed for this chapter. If you have a team set up, just leave it as is. We will cover this in greater detail later in the book.

3. **Organization Name**: You can set the organization name to your company name or just your name.
4. **Organizer Identifier**: You will set the organizer identifier to be your domain name in reverse. For example, my website URL is `cocoa.academy`, and therefore, my identifier is `academy.cocoa`. Since URLs are unique, it will ensure that no one else will have your identifier. If you do not have a domain, then just use your first and last name for now. You will eventually have to purchase a domain if you would like to submit your app to the Apple store.
5. **Bundle Identifier**: When you create a new project, Apple will combine your **Product Name** with your **Organizer Identifier** to create your unique bundle identifier. So even if 10,000 people create this project, each person will have a different bundle identifier.
6. **Language**: Make sure your language is set to `Swift`.
7. **Devices:** We are going to set this as `Universal` (iPhone and iPad).
8. **Checkboxes**: You can uncheck **Use Core Data**, **Include Unit Tests,** and **Include UI Tests**, as these are things we will not be using in this chapter.

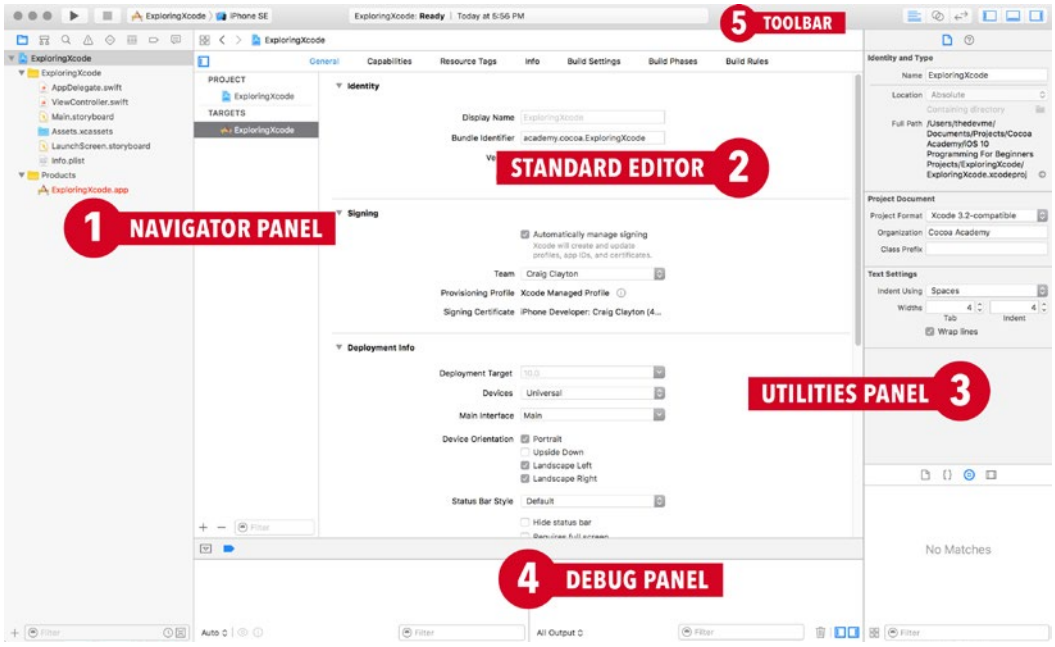
Now, select **Next**, and Xcode will prompt us to save our project. I have a dedicated folder for all my projects, but you can save it on your desktop for easy access.

The Xcode interface

Your project is now open, and it is time for us to get familiar with all of the panels. If this is your first time in Xcode, then it probably will be a bit overwhelming for you. Therefore, we will break it down into five parts:

- **NAVIGATOR PANEL**
- **STANDARD EDITOR**
- **UTILITIES PANEL**

- **DEBUG PANEL**
- **TOOLBAR**
- **WINDOW PANE CONTROLS**



Navigator panel

The primary use for the Navigator panel is to add new files and/or select existing files. The other icons are used from time to time, which we will cover as we need them.

Standard editor

The Standard editor is a single panel view used to edit files. The Standard editor area is the primary area in which you will work. In this area, we can view Storyboard files, see our Swift files, or view our project settings.

Utilities panel

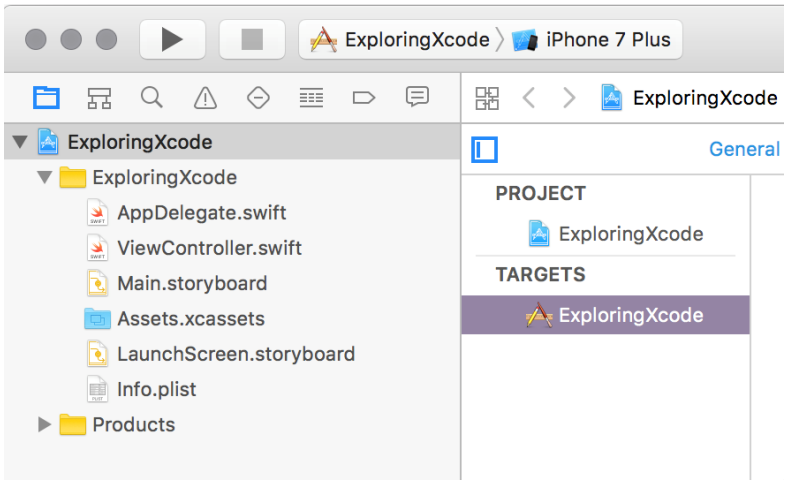
The Utilities panel can be a bit confusing when you first use Xcode, because this menu changes based on what you have selected in the Standard editor. When we start building an app, we will dig deeper into this. For now, just know that the Utilities panel is made up of the inspector pane at the top and the library pane at the bottom. The inspector pane allows you to change attributes or properties of things you put in your Storyboard—the library pane allows you to insert objects, image assets, and code snippets into your app.

Debug panel

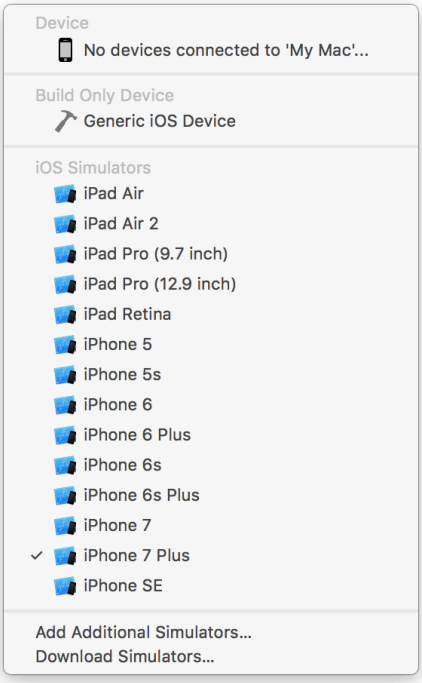
The Debug panel will allow us to see log messages from our app. You will become very familiar with this panel by the time you finish this book. The Debug panel is one of the greatest tools to get feedback on what your app is doing or not doing.

Toolbar

Next, we look at the Toolbar:

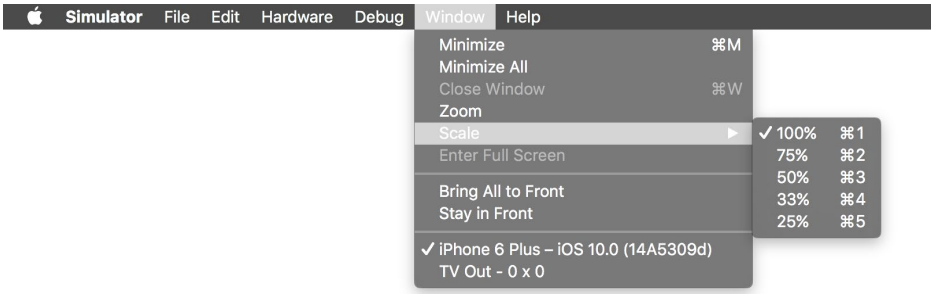


First, we have a **Play** button, which is how we launch our app (or use *Cmd* + *R*). Next, you will see a **Stop** button, which will not be active until you run your app. This **Stop** button (or *Cmd* + .) is used to stop your app from running. To the right of the **Stop** button, you will see your target (your project name) along with the current simulator selected. If you click on your project name, you will see a screen similar to this:



This drop-down menu, which we will call the *device and Simulator drop-down menu,* allows you to change your simulator type. For the purposes of our project, select **iPhone 7 Plus** as your simulator and then click on the play icon (or use *Cmd* + *R*) to run your app. Your app will be blank and most likely will not fit the entire screen (unless you are on a large screen).

In order to see the entire screen, you can scroll right and left—however, doing this gets harder once you have elements in your app. So, an alternative to scrolling is to resize your app to fit the screen. We can do this by going to your **Simulator** menu and navigating to **Window** | **Scale**:



Here, you will be able to scale from **100**% to **75**%, **50**%, **33**%, or **25**% (select whichever percentage allows the app to best fit your screen size).

Now, let's return to Xcode and select the **Stop** button (or use *Cmd* + .).

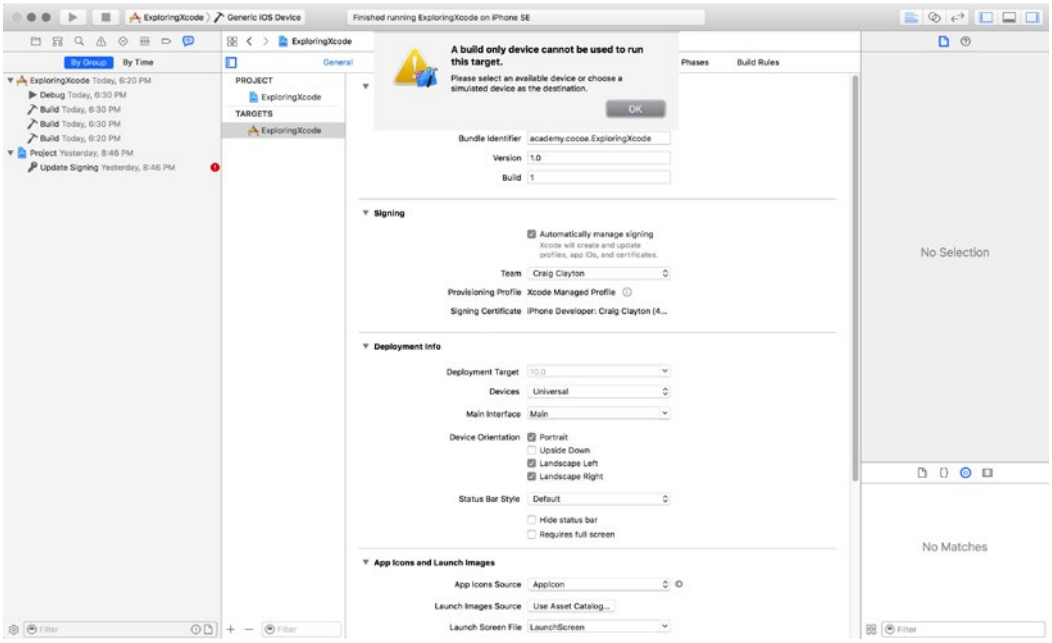


If you use the keyboard shortcut, make sure Xcode is in focus; otherwise, this shortcut will not work. I work on a 15-inch Macbook Pro Retina—therefore, when I am working on an app, I will use the iPhone 6 or iPad Air 2 simulator in landscape. They both fit nicely on my screen without having to resize either.

In addition to the simulator, there is a Build Only Device as well as a Device section, both at the top of the *device and Simulator drop-down* menu that was shown earlier in this chapter. Note that, for our purposes, you will only need a Simulator while we are building the app—however, you can add an iOS device if you would like (see under **iOS Device**).

Generic iOS Device

The Generic iOS Device, under the Build Only Device section of the *device and simulator drop-down* menu, is used for when you need to archive your app, which means that you are preparing your app for submission to Apple (either to the App Store or to Test Flight). If you try to select **Generic iOS Device** now and run the app, you will get the following message:



Therefore, change **Generic iOS Device** to an actual Simulator, and then you will be able to continue.

iOS device

If you do not have a device connected to the computer, you will see **No devices connected . . .** under the Device section of the *device and simulator drop-down* menu.

As noted earlier, when we start building the *Let's Eat* app, you will have the option of using the Simulator or connecting a device to Xcode. Using a device is slower—however, the simulator will not perform the same as a device will.

In the past you, you needed to have a paid account to build your app on a device. Now, you do not need a developer account in order to run the app on your device. Note that if you decide to connect your device instead of using a simulator, you will need iOS 10 installed on it. The following steps are only for those who do not want to pay for the Apple Developer Program at this time:

1. Connect your iOS device via USB.
2. In the drop-down menu, select your device (here, **Craig's iPhone**):



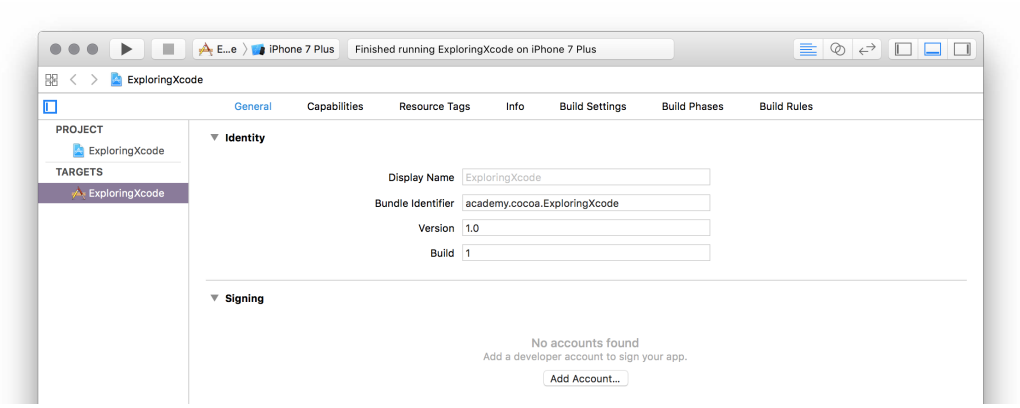
3. Wait for Xcode 8 to finish indexing and processing. This may take a bit of time. Once complete, the status will say **Ready**.
4. Run the project by hitting the **Play** button (or use *Cmd* + *R*).

   You will get two errors that state the following:

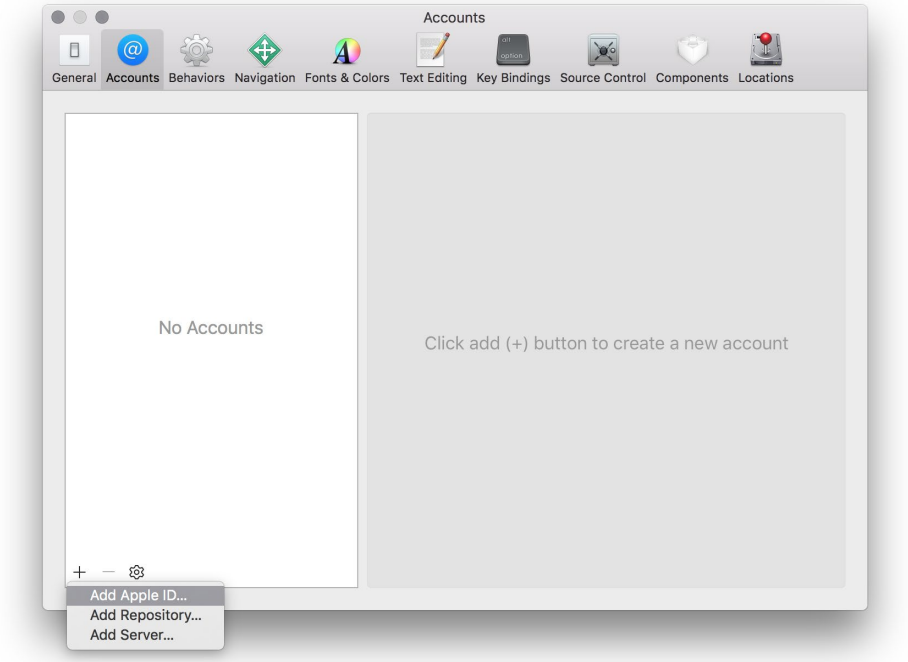
   - **Signing for ExploringXcode requires a development team. Select a development team in the project editor**.
   - **Code signing is required for product type Application in SDK iOS 10.0**.

   Ignore the specifics of these errors as they basically indicate that we need to create an account and add our device to that account.

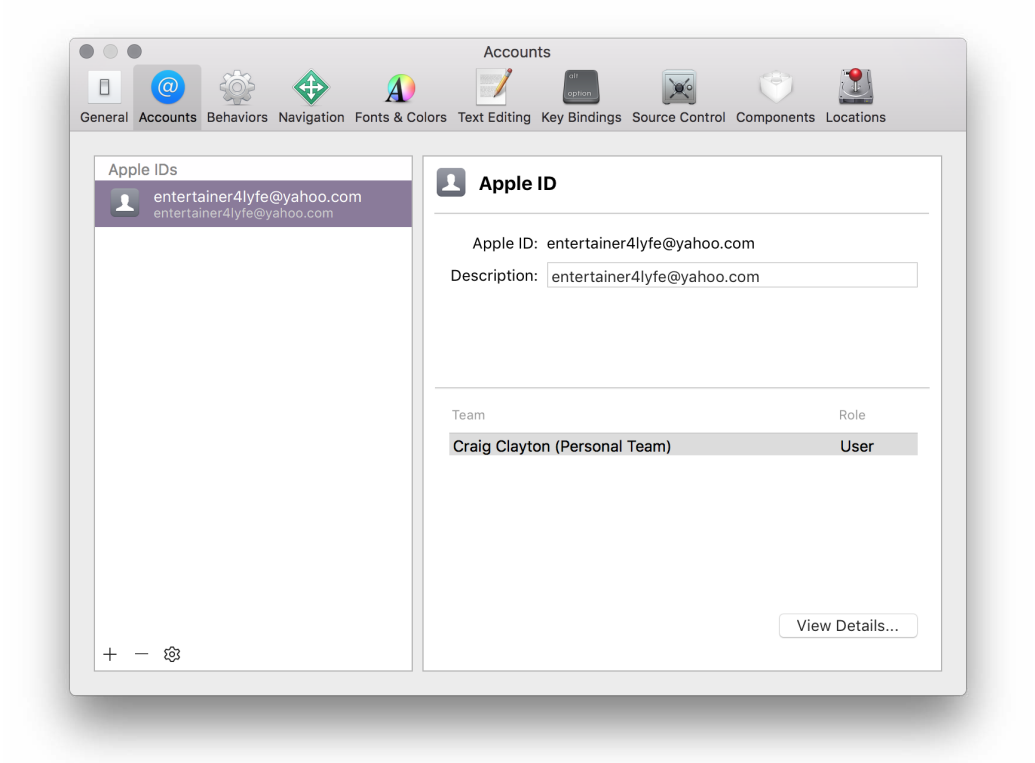
5. Now, in the Standard editor, you will see under **Signing** that you need to add an account:



6. Click on **Add Account**. If a Sign in to Xcode with your Apple ID dialog box does not pop up, inside of the **Accounts** screen on the bottom left, click on the **+** and select **Add Apple ID**:



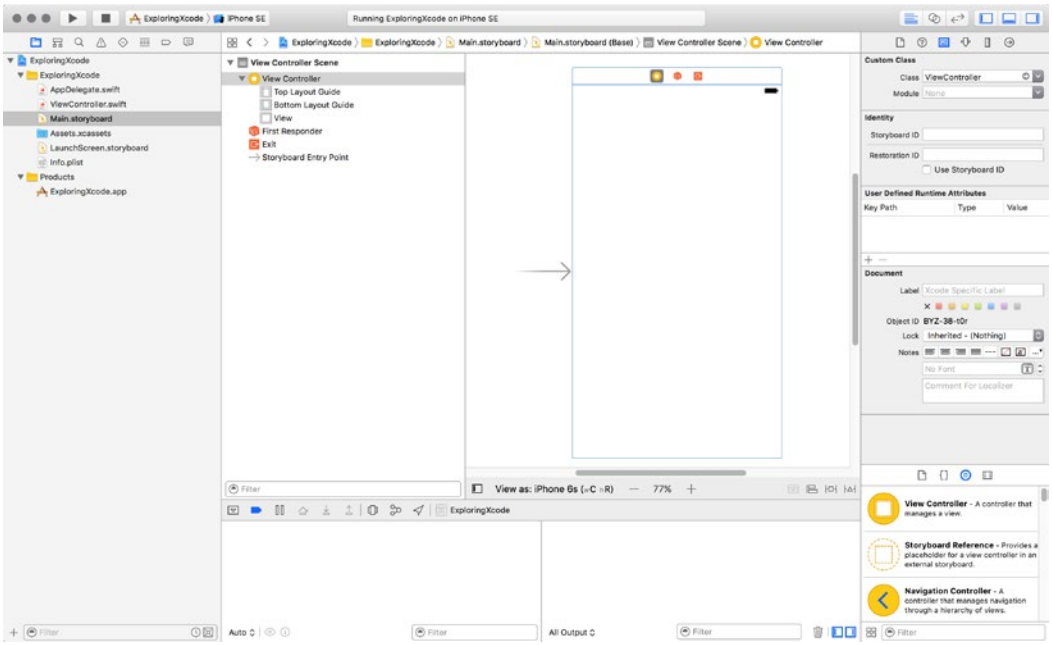
7. Then, you will click on **Create Apple ID**. You will be asked to enter your birth date, name, e-mail, and password, along with security questions. Make sure that you verify your e-mail before you answer the security questions; otherwise, you will have to come back to this screen and add Apple ID again.
8. Once you have finished all the steps, you will see your account:



If you already have an account, then, instead of seeing **Add Account**, you will see a drop-down menu with your account listed. If your device is not connected to this account, you might see a message asking if you would like to add your device to your account.

You will not need to use a device for the majority of this book—however, depending on the type of Macbook you have, you might need to use a device.

Before we get to the right side of the Toolbar, select the `Main.storyboard` file in your Navigator panel. This file is used to do all of your visual setup for your entire app. We will cover this in detail later in the book. After you select the file, you should see the following:



Window Pane Controls

The following screenshot shows Window Pane Controls:





For better quality of images, download the Graphics bundle from : `https://www.packtpub.com/sites/default/files/downloads/iOS10ProgrammingforBeginners_ColoredImages.pdf`

Moving onto the Window Pane Controls, you will see two groups of icons. The first group is called the Editor Mode, and the second group is called the View. Let's look at the functions of the Editor Mode icons:

Editor Mode icons	Function
	This icon controls the Standard editor (which is the center panel in the earlier screenshot of the `Main.storyboard` file in the Navigator panel).
	This icon splits the Standard editor into two panels, where you will see the `ViewController.swift` file on the right. We will use this split screen throughout the book.
	This icon is the Version editor. We will not address the Version editor in this book, since it is a more advanced feature.

At this point, you might be thinking that there are way too many panels open, and I would agree with you. This is where the last group of View icons in the Toolbar comes in handy.

Let's look at these icons and their functions in the following table:

View Mode icons	Functions
	This icon will toggle (hide or show) the Navigator panel (or use *Cmd* + *0*).
	This icon will toggle (hide or show) the Debug panel (or use *Cmd* + *Shift* + *Y*).
	This icon will toggle (hide or show) the Utilities panel (or use *Cmd* + *Alt* + *0*).

Summary

Congratulations! You have finished exploring the basics of Xcode. When we start building our app, we will cover the more important parts of Xcode in depth. It is now time to start learning Swift 3.

2
Building a Foundation with Swift

Now that we have had a short tour of Xcode, it is time to start learning about Swift. Remember, if you are new to programming, things will be very different for you, so take your time. The important skills that you will learn here will set you up with a solid foundation in iOS development. If you have previous programming experience, you should still review this chapter, as it can only enhance your programming skills and act as a refresher for you.

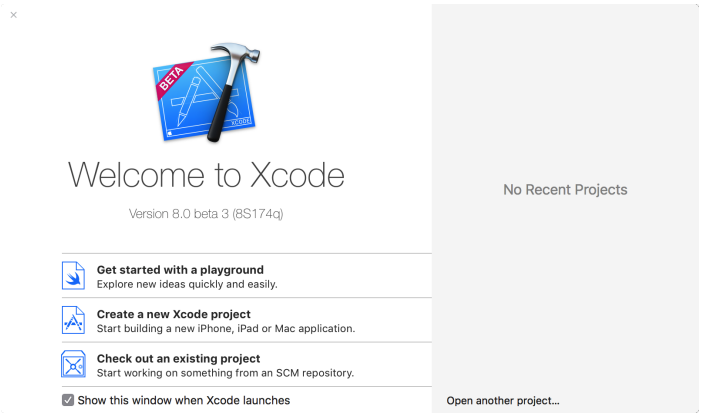
On June 2, 2014, Apple changed the game for iOS development, because this was the day Swift was announced to the world. With this announcement, everybody was put on an even playing field, because they had to learn a new programming language. Swift has brought a more modern approach to developing apps and has seen a huge influx of new developers of all ages wanting to build iOS apps. But, enough about history! Let's dig in and see what you are going to learn.

The following will be covered in this chapter:

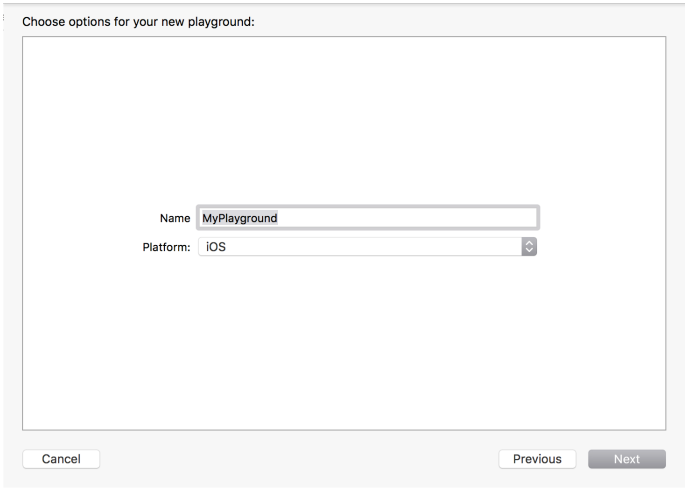
- Playgrounds
- Data types
- Variables and constants
- Debug and `print()`
- Comments
- Type safety and type inference
- `if` statements
- Optionals and optional bindings
- Functions

Playgrounds – an interactive coding environment

Before we jump into building the app that we will be creating in later chapters, called *Let's Eat*, we need to understand the basics of Swift. An easy way to experiment with Swift is to use **Playgrounds**. It is an interactive coding environment, which evaluates your code and displays the results. Using Playgrounds gives us the ability to work with Swift without needing to create a project. It is great for prototyping a particular part of your app. So, whether you are learning or experimenting, Playgrounds is an invaluable tool. In order to create a Playground, we need to launch Xcode and click on **Get started with a playground**:

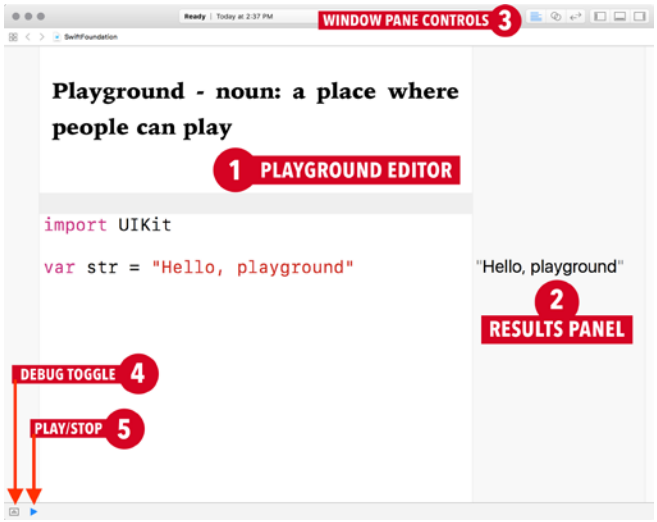


The options screen for creating a new Playground screen will appear:



First, name your new Playground `SwiftFoundation`, and then make sure your **Platform** is set to **iOS**. Now, we can explore Playgrounds a bit.

When you launch the app, you will see five distinct areas:



Let's breakdown each area in Playgrounds:

- **PLAYGROUND EDITOR**: This area is where you will write all of your code.
- **RESULTS PANEL**: The **RESULT PANEL** is a feature only found in Playgrounds and provides immediate feedback.
- **WINDOW PANE CONTROLS**: The **WINDOW PANE CONTROLS** has two groups of icons:



  As we discussed earlier, the first group is called the Editor Mode, and the second group is called the View. Refer to the detailed description of these icons in the previous chapter for information about what each of these icons does.

- **DEBUG TOGGLE:** This button allows you to hide and show the Debug panel and toggle on the Debug panel.
- **Play/Stop:** This button is used in order to make Playgrounds execute code or in order to stop Playgrounds from running. Typically, Playgrounds runs on its own, but sometimes you need to manually toggle it on when Playgrounds does not execute your code for you.

Now that we have our setup finished, delete everything in this file. Your Playground should have three open panels—your Playground editor, Results panel, and Debug panel. Let's start digging into some code.

Data types – where it all starts

Swift offers a collection of built-in data types. Data types are known as string, integer, floating-point numbers, and Booleans. These data types can be found in most programming languages. Therefore, if you are not new to programming, you can skip this section and start at the variables and constants section here.

Let's walk through each data type for those of you who are new to programming or would like a refresher.

String

The first data type we will discuss is a string. A string is represented by a series of characters. strings are used to display text in an app. When a string is wrapped in quotes, it is known as a string literal. In programming, we cannot just add text into Playgrounds. So, in order to write a string, we must wrap our string inside of quotes. Let's now add our name into Playgrounds wrapped in quotes:



In Playgrounds, your values also will appear inside of your Results panel. So, we now know that, in order to create a String, we need to use quotes.

Integer data type

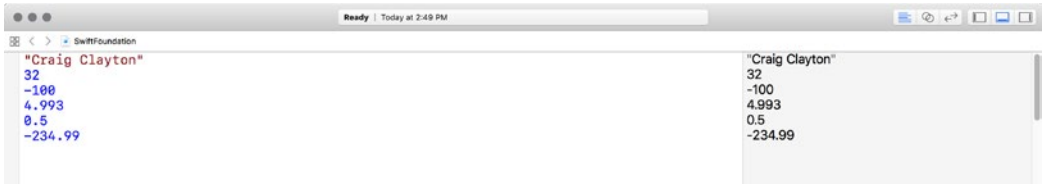
Integers (**Ints**) are whole numbers, such as `32` and `-100`. Integers are useful for when you need to do calculations (that is, adding, subtracting, multiplication, and so on). Let's add some numbers into Playgrounds. On the next line, under your name, type `32` and then, on the following line, `-100`:



Again, you will see both `32` and `-100` in the Results panel under your name.

Floating-point numbers

Floating-point numbers are numbers with a fractional component, such as `4.993`, `0.5`, and `-234.99`. Let's add these values into Playgrounds as well:



Booleans

Booleans (**bools**) for short, are referred to as logical, because they either can be `true` or `false`. Booleans are used when you need to determine whether some logic is `true` or `false`. For example, `Did the user log in?` this statement would either be `true`, `Yes, they did` or `false`, `No, they did not`. So, in Playgrounds, add `true` and `false`:



So, now we covered all of the basic data types in Swift. But, right now, we have no way to use these data types. This is where variables and constants come into play.

Variables and constants – where data is held

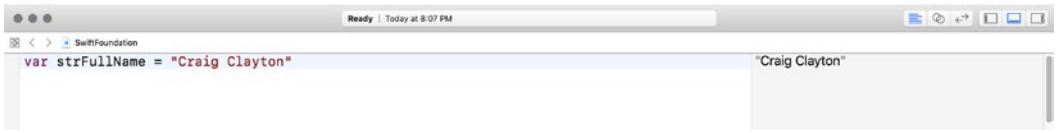
Variables and constants are like a container that holds some kind of data. When you want to declare a variable, you have to use the `var` keyword. Let's declare each of the data types we did earlier, but, this time, using variables and constants instead.

Creating a variable with a String

First, delete what you have entered into Playgrounds already, and now, let's declare our first variable, named `strFullName`, and set it to your name:

```
var strFullName = "Craig Clayton"
```

The preceding code says that we have a variable named `strFullName` and that it is holding a string value of `Craig Clayton`. Your results panel will have your actual name as its data:

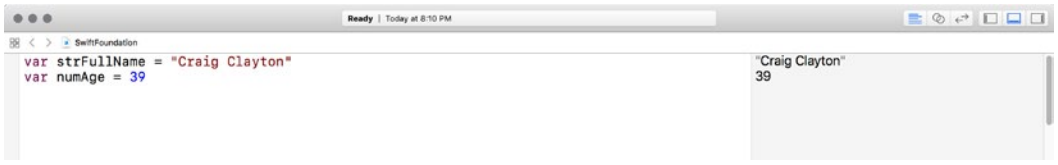


Creating a variable with an Integer (Int)

Now, let's create a variable with an Int called `numAge` and set it to our age (or whatever you want your age to be) by adding the following:

```
var numAge = 39
```

Our program now knows that age is an Int. You should see both your name and age in the Results panel, just like you did previously:

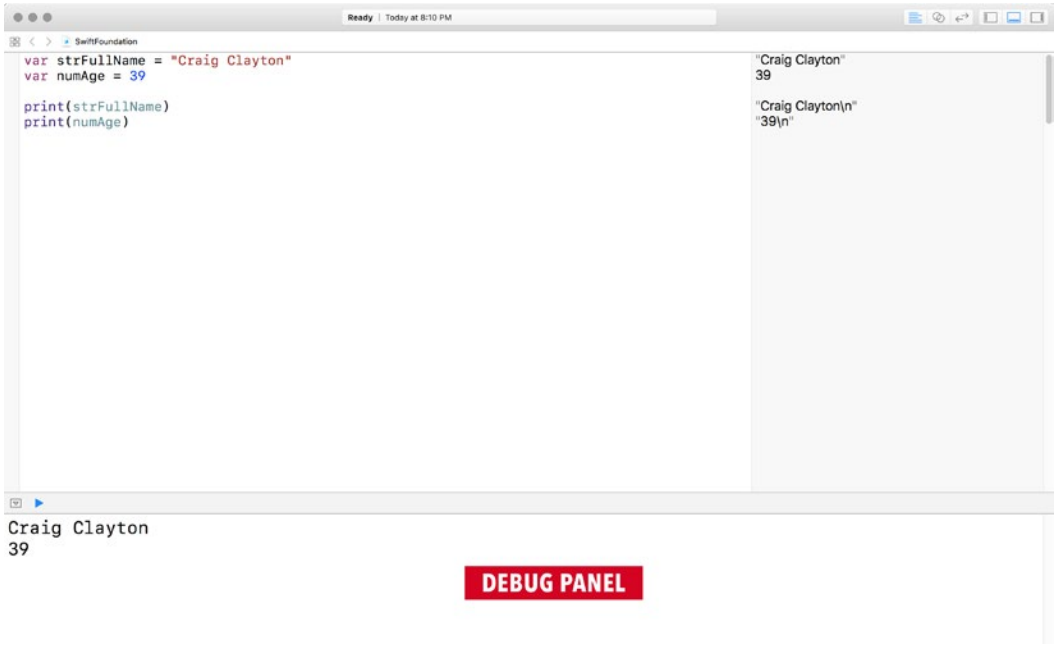


Debug and print() – detecting your bugs

We can use the Debug panel (at the bottom of the following screenshot) using what is called `print()`. So, let's see how `print()` works by printing both our name and age. We can do this by adding the following:

```
print(strFullName)
print(numAge)
```

It should appear on your screen as follows:

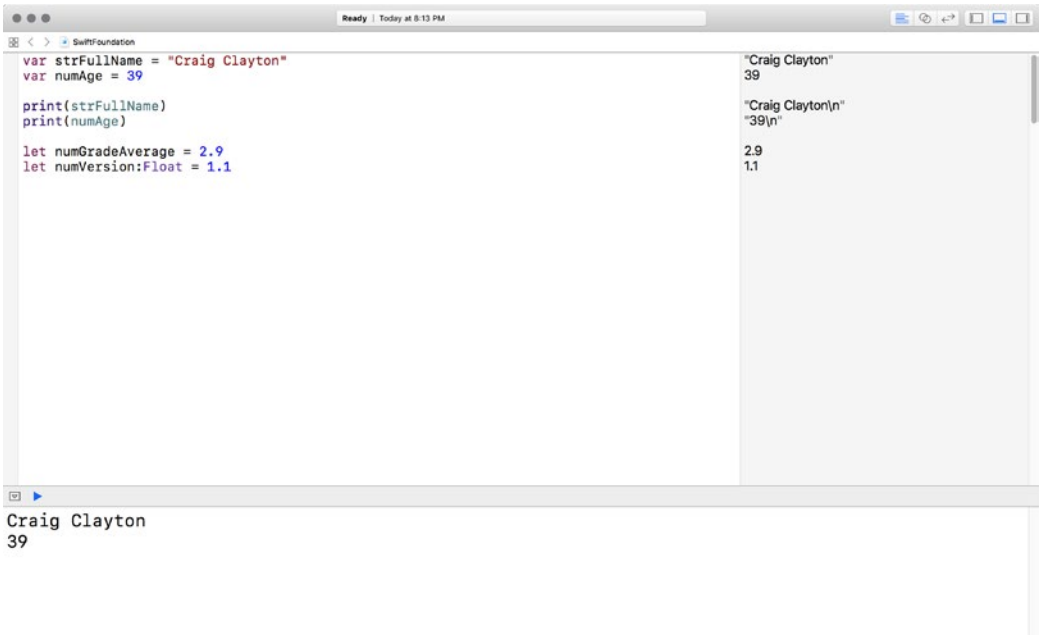


You should now see the output in both the Results and Debug panels. Using `print()` allows us to see things in our Debug panel and therefore verify expected results. This is a very useful debugging tool.

Adding floating-point numbers

Let's now add floating-point numbers, using the constant `let` constant in the Playground:

```
let numGradeAverage = 2.9
let numVersion:Float = 1.1
```



You will notice that a couple of things are different. First, we are using the `let` keyword. Using let tells our program that this is a constant. Constants are variables that cannot be changed once they are set (as opposed to a non-constant variable, which can be changed after being set).

The other thing you might have noticed is that we explicitly set our `numVersion` to Float. When dealing with a floating-point number, it can be a Double or a Float. Without getting too technical, a Double is much more precise than a Float. The best way to explain this is to use pi as an example. Pi is a number in which its digits goes on forever. Well, we cannot use a number that goes on forever; however, a Double and Float will handle how precise that number will be. Let's look at the following image to see what I mean by precise:

Double vs Float

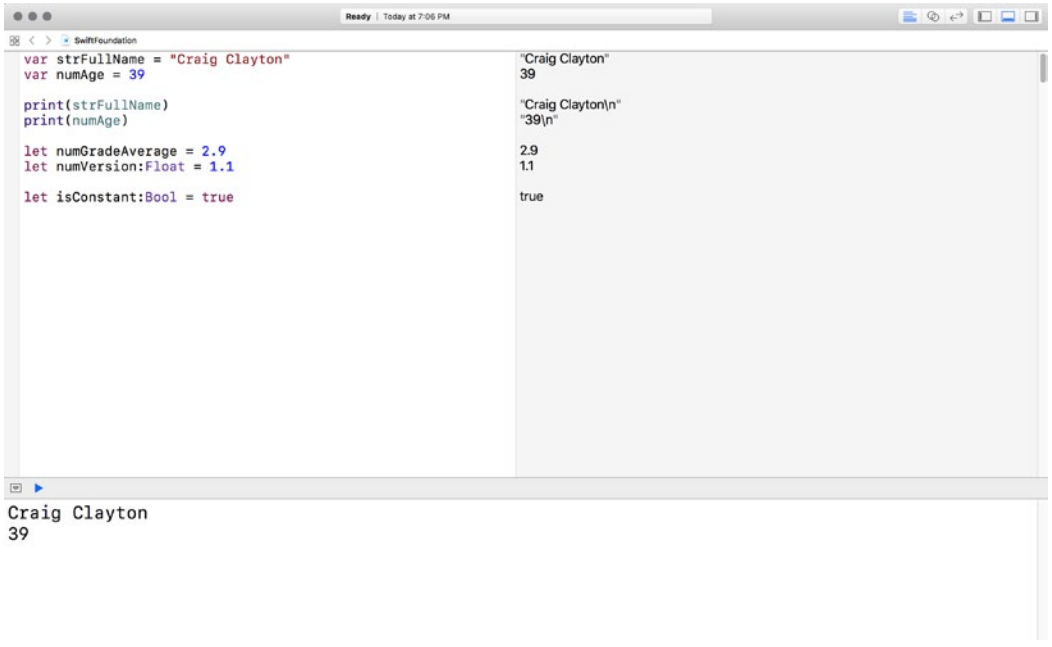
```
let lessPrecisePI   = Float("3.14")
let morePrecisePI = Double("3.1415926536")
```

So, in the preceding example, you can see that Float will only display 3.14, whereas Double will give you a much more accurate number. In Swift, a Double is preferred. Therefore, if you do not explicitly set the floating-point number to a Float, Swift will default it to a Double. To set `numVersion` to a Float, you must purposely set it that way.

Creating a Boolean

Now, it is time to create a `Bool`, and we will make it a constant. Enter the following code:

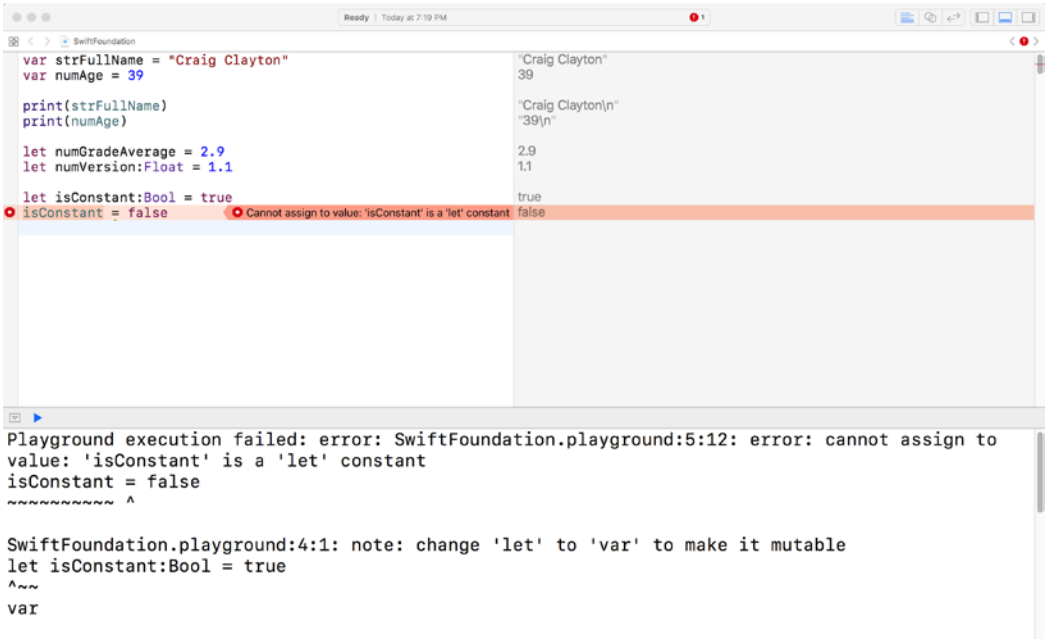
```
let isConstant:Bool = true
```



Since `isConstant` is set, let's make it false by adding this:

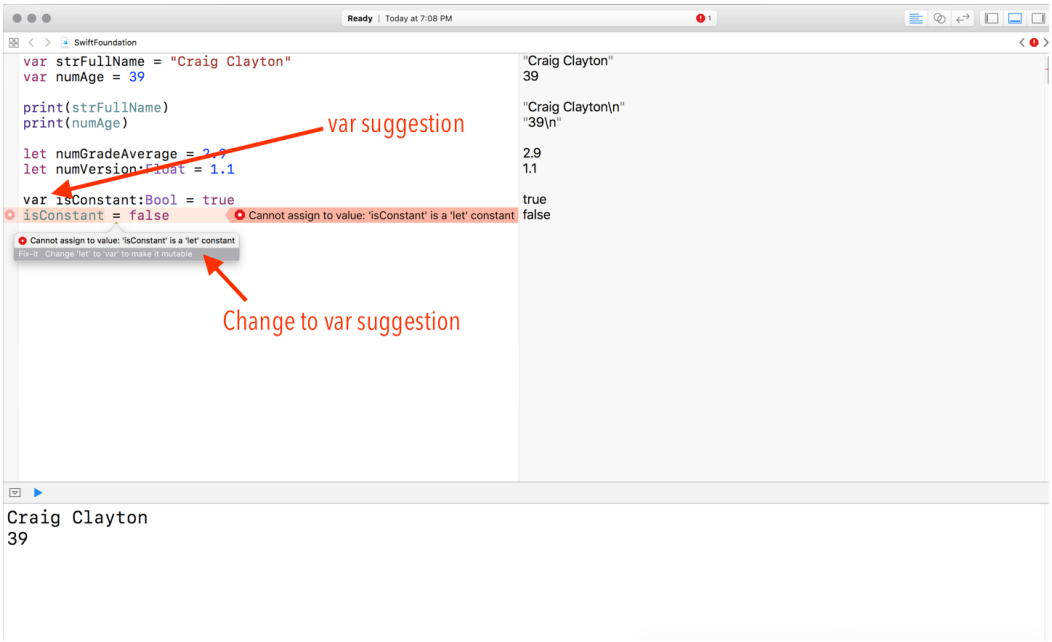
```
isConstant = false
```

On the same line as what you just entered, you now will see a red circle with a white dot. The red circle means that there is an error. The white circle inside of it means that Xcode can fix the error for you (most of the time):



You also will notice an error in your Debug panel, which is just a more detailed error. This error is telling us that we are trying to change the value of a constant when we cannot do so.

If you tap on the circle, you will see that Playgrounds suggests that you change the `var` to a `let`, since you cannot assign a value to a constant:



Since we want it to remain a constant, let's delete the line `isConstant = false`. We have covered basic data types, but there are some other programming basics we should discuss as well.

Hungarian notation

Typically, I like to use Hungarian notation when writing variables, because it acts as an identifier for the data type of the variable. For example, earlier we wrote `strName` and `numGradeAverage`. Using `str` helps me know that this variable is a String and `num` helps identify this variable as an Int or Float. Throughout this book, you will see that I use Hungarian notation.

Why constants versus variables?

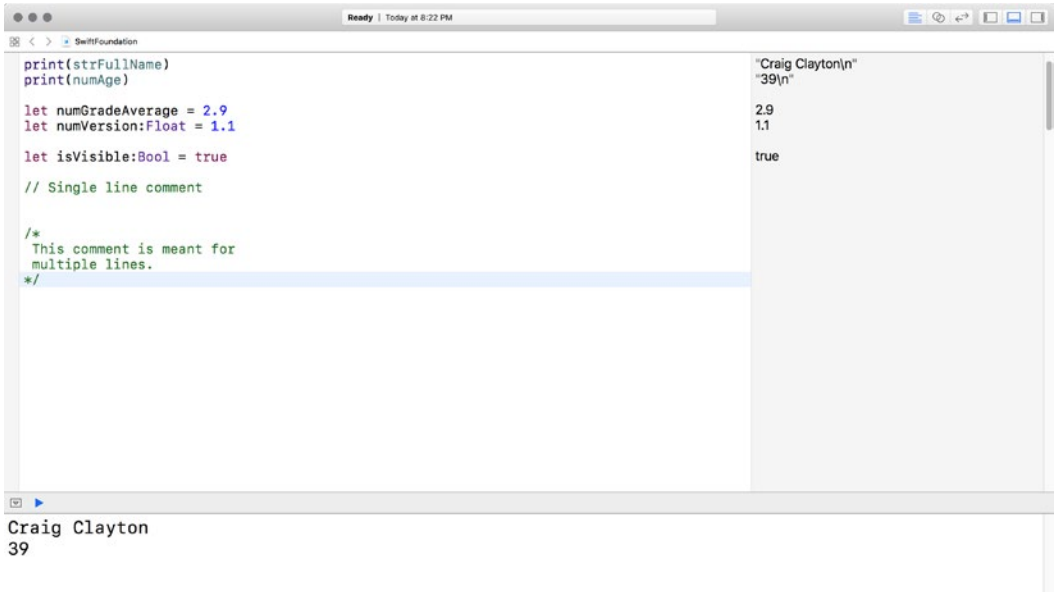
You might be asking yourself, *Why would you ever want to make something constant?* Since constants cannot change after you run your app, they keep you from accidentally changing a value that should not be changed. Another really good use for constants is for base URLs, as you would not want these to change. When you are getting data, you do not want to accidentally change the value midway through your app. Apple recommends that you use `let` whenever possible. Typically, I will use a let until Xcode warns me that a `var` is preferable. If I change the value from `let` to `var`, then I am verifying that this is the behavior I want.

Comments – leaving yourself notes or reminders

Comments are a great way to create notes or reminders to yourself. When you comment code, it means that it will not be executed when your code runs. There are two types of comments used `//` or `/* */`. `//` is typically used for a one-line comment and `/**/` is used for a block of text.

You can add comments to your code, such as a `TODO` item or just a brief explanation of what something is doing.

Let's see what both of these look like:



Type safety and type inference

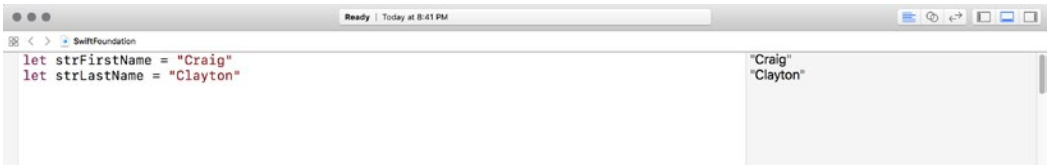
Swift is a type-safe language, which means that you are encouraged to be clear about the types of values with which your code will work. Type inference means that before your code runs, it will be able to quickly check to ensure that you did not set anything to a different type. If you do, Xcode will give you an error. Why is this good? Let's say that you have an app in the store, and you set one of your variables as a String in one part of your code, but then accidentally set the same variable as an Int in another part of your code. This error may cause some bad behavior in your app that could cause it to crash. Finding these kind of errors is like finding a needle in a haystack. Therefore, type checking helps you write safer code by helping you avoid errors when working with different types.

We have now looked at data types and know that strings are for textual data, Int for Integer, Bool for Boolean, and a Double and a Float for floating-point numbers. Let's look a bit deeper into data types and see how we can do more than just assign them to variables.

Concatenating strings

String concatenation is the result of combining multiple string literals together to form an expression. So, let's create one by first entering two String literals:

```
let strFirstName = "Craig"
let strLastName = "Clayton"
```



Combining these two gives us a String concatenation. We can combine Strings by using the + operator. Add the following:

```
let strName = strFirstName + strLastName
```



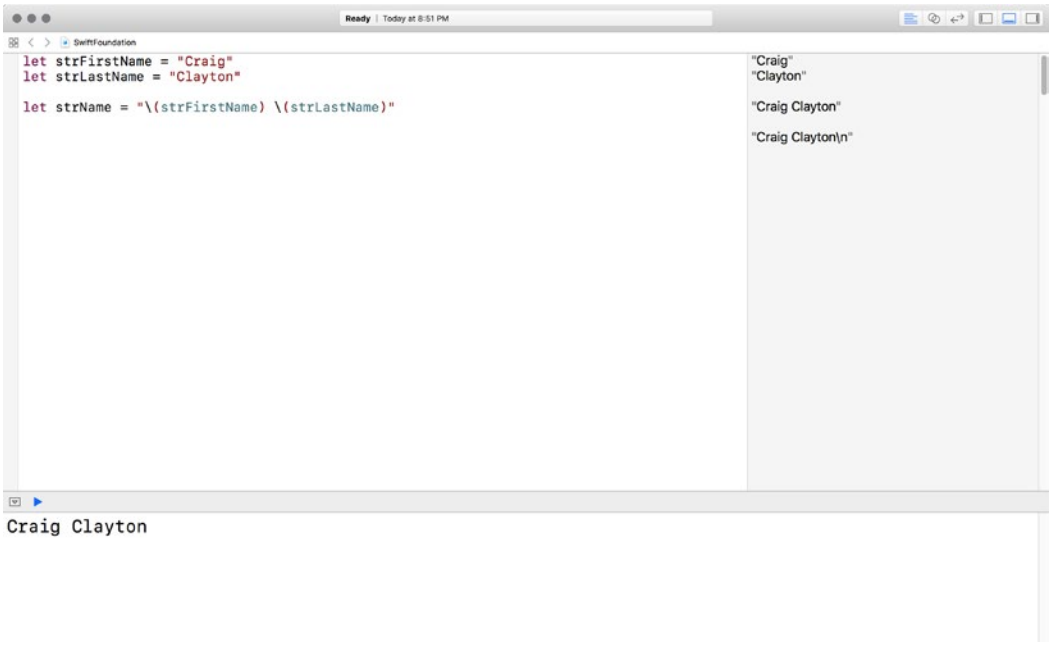
When you look in the Results panel, you will notice that there is no space between our first and last names.

In addition, if we just put the variables in quotes, they will revert to simple string literals and will no longer be variables.

String interpolation

In order to correct that, we can put these variables inside of quotes, which is known as string interpolation, using a backslash and parentheses around each of our variables inside of the string interpolation. Let's update our `strName` variables to be the following, and you will see the space in the name in the Results panel:

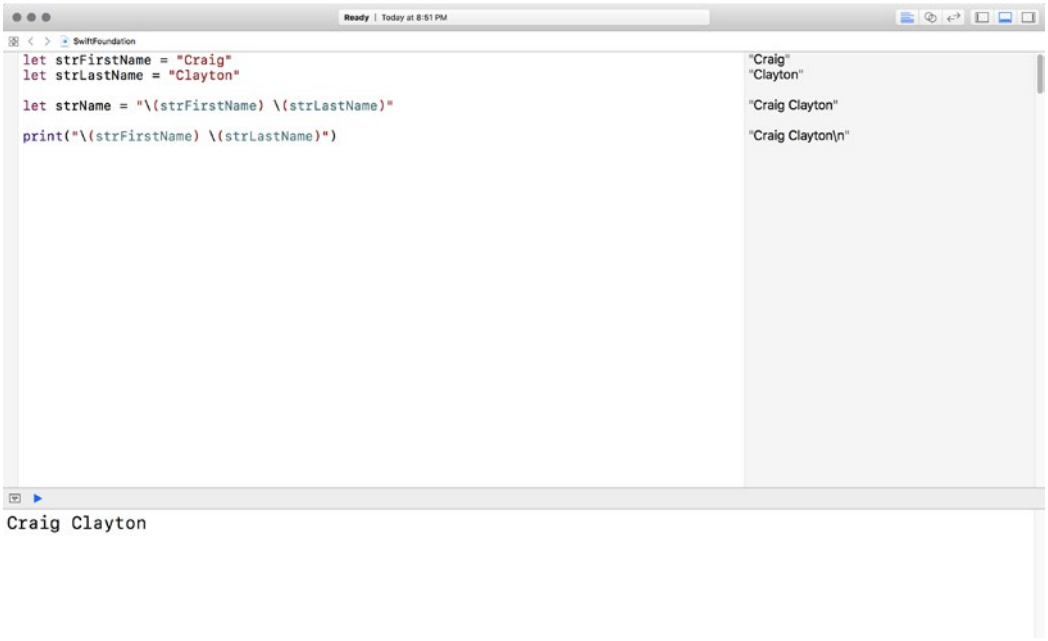
```
let strName = "\(strFirstName) \(strLastName)"
```



Now that we know about using variables inside of quotes, we can do the same inside of `print()`. Let's put the `strFirstName` and `strLastName` inside of `print()`, as follows:

```
print("\(strFirstName) \(strLastName)")
```

Print statements are great for checking to see you are getting the value you want:



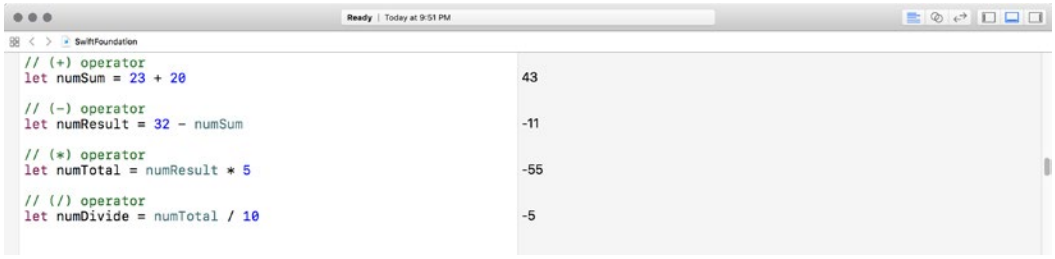
Bam! Now, we have a way to see multiple variables inside of `print()` as well as how to create string interpolation by combining multiple strings together. We can do much more with Strings, and we will cover them later in the book.

Operations with our Integers

In our Playground, we know that `numAge` is an Int, but with Ints, we also can write arithmetic expressions using numbers, variables/constants, operators, and parentheses. Let's start with addition, subtraction, multiplication, and division. Add the following into Xcode:

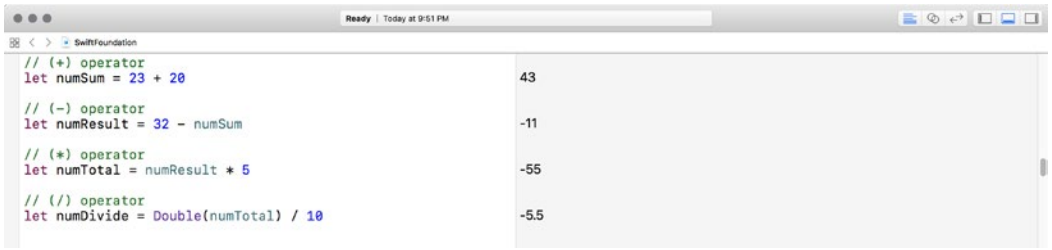
```
// (+) operator
let numSum = 23 + 20
// (-) operator
let numResult = 32 - numSum
// (*) operator
```

```
let numTotal = numResult * 5
// (/) operator
let numDivide = numTotal / 10
```



So, `numSum` added two integers (`+` operator) together, totaling `43` in our preceding example. Then, we subtracted (`-` operator) `numSum` from `32` to create `numResult` (`-11` in our example). After that, we took `numResult` and multiplied (`*` operator) it by `5` (see `-55` in the Results panel). All of this is pretty basic math—however, you may have noticed something different with our division equation (`/` operator). When you divide two integers, the result will be a third integer. So, instead of `-55` divided by `10` equaling `-5.5`, our result was `-5`. In order to get the correct floating-point value of `-5.5`, we need to make our division value a Double. Therefore, let's change our `numDivide` to the following:

```
let numDivide = Double(numTotal) / 10
```



All of these operations will look familiar to you, but there is one with which you might not be familiar and that is the remainder operator. The remainder operator returns the remainder when one number is divided by another.

So, for example, `7` divided by `3` equals `2.33`. When we apply the remainder operator, we get back `1`. Add the following to Playgrounds:

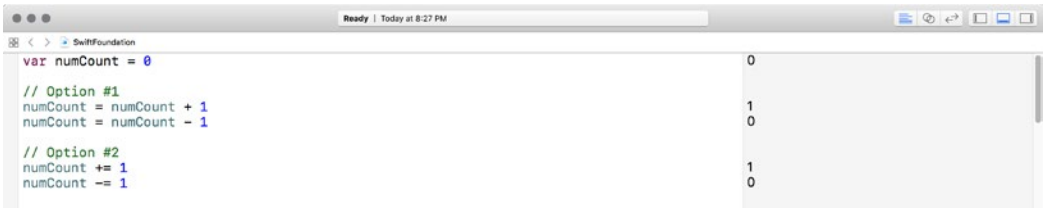
```
// (%) modulo operator
let numMod = 7 % 3
```



Increment and decrement

There will be times when you need to increment (increase) or decrement (decrease) a value. There are two ways you can accomplish this. Add the following into Playgrounds:

```
var numCount = 0
// Option #1
numCount = numCount + 1
numCount = numCount - 1
// Option #2
numCount += 1
numCount -= 1
```



Both of these options do the same thing, but option `#2` is just written in shorthand. The preferred way is to use option `#2` (the `+=` (addition assignment operator) and `-=` (subtraction assignment operator)), but it really is your preference.

Comparison operators

We also can compare different numerical variables. These might be familiar to you from math class. Let's enter these into Playgrounds:

```
let numFirstValue = 1
let numSecondValue = 2

// Checking for greater than
numFirstValue > numSecondValue
// Checking for less than
numFirstValue < numSecondValue
// Checking for greater than or equal
numFirstValue >= numSecondValue
// Checking for less than or equal
numFirstValue <= numSecondValue
// Checking for equal
numFirstValue == numSecondValue
// Checking for not equal
numFirstValue != numSecondValue
```

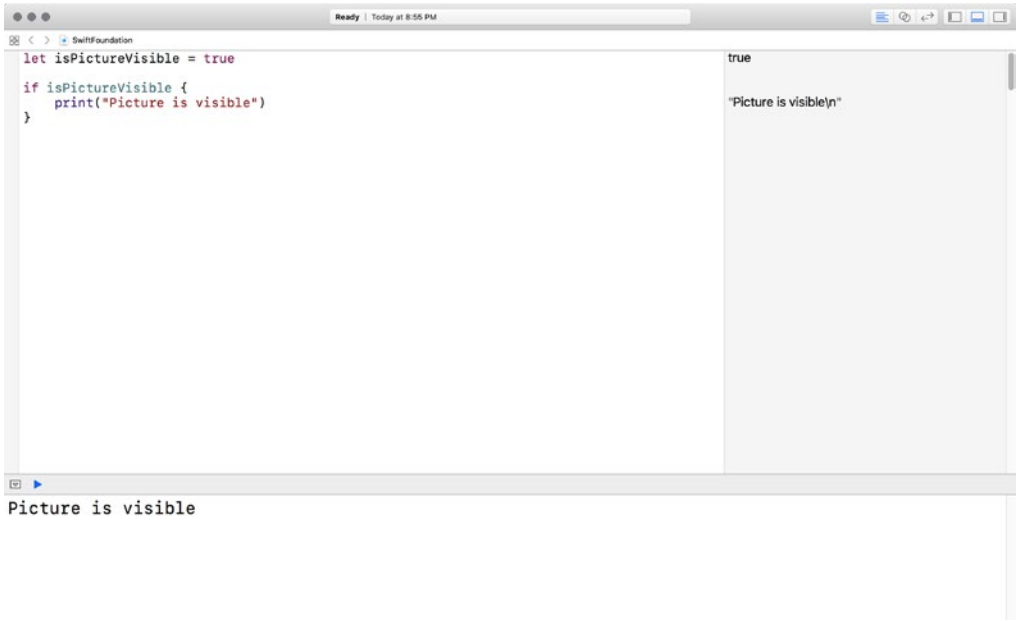


As you can see in the Results panel, these comparison entries result in `true` or `false` based on the values that you enter (here, `1` and `2`).

If-Statements – having fun with logic statements

Let's add our first piece of logic using an if-statement. An `if` statement is a simple statement to determine whether or not the statement is true. Input the following into Xcode:

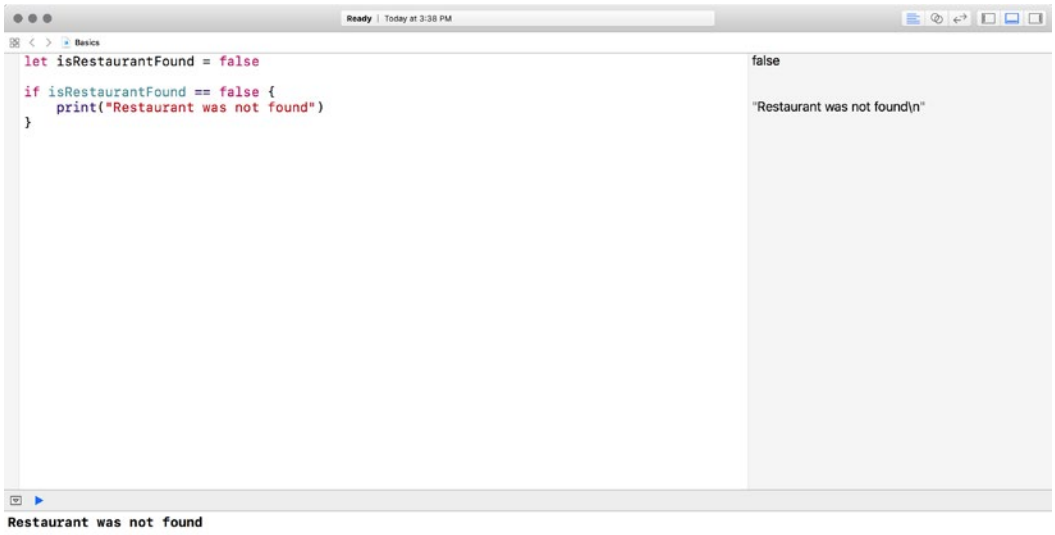
```
let isPictureVisible = true
if isPictureVisible {
  print("Picture is visible")
}
```



In the first line of the preceding code, we created a constant named, `isPictureVisible`, and we set it to `true`. The next line starts our `if` statement and is read as follows—if `isPictureVisible` is `true`, then print `Picture is visible`. When we write `if` statements, we must use the curly braces to enclose our logic. It is good practice to put the opening curly brace ({) on the same line as the `if` and the closing curly brace (}) on the line after you write your logic.

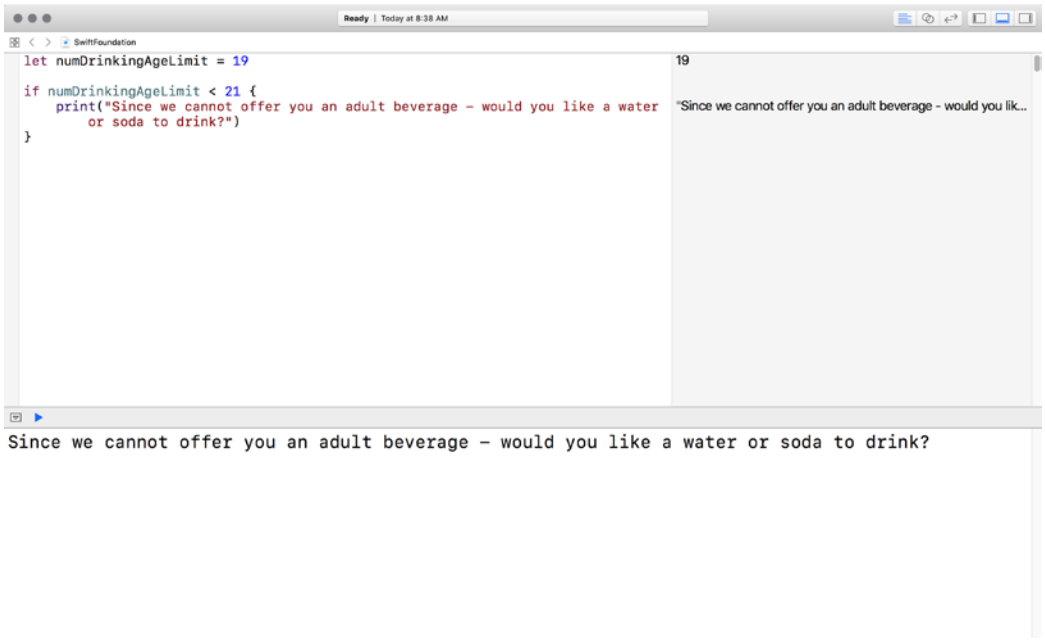
When writing `if` statements using a `bool`, you are always checking for `true`; however, if you wanted to check for `false`, you would do the following:

```
let isRestaurantFound = false
if isRestaurantFound == false {
  print("Restaurant was not found")
}
```



Bools work great with `if` statements, but we also can use them with other data types. Let's try an `if` statement with an Int next. Write the following into Playgrounds:

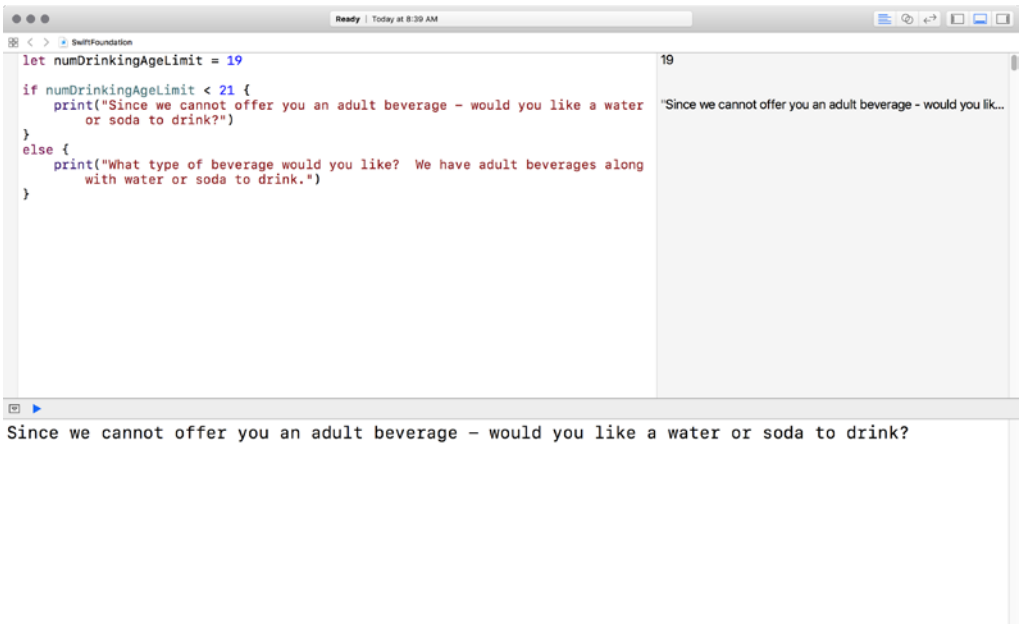
```
let numDrinkingAgeLimit = 19
if numDrinkingAgeLimit < 21 {
  print("Since we cannot offer you an adult beverage - would you like
a water or soda to drink?")
}
```



In the preceding example, we first created another constant with our Int set to `19`. The next line says—if `numDrinkingAgeLimit` is less than `21`, then print `Since we cannot offer you an adult beverage—would you like a water or soda to drink?` When you are using Ints within `if` statements, you will use the comparison operators (`<`, `>`, `<=`, `>=`, `==`, or `!=`). However, our last `if` statement feels incomplete, because we are not doing anything for someone over `21`. This is where you will utilize an `if...else` statement. You enter an `if...else` statement exactly as you did an `if` statement, but, at the end, you add the word `else`.

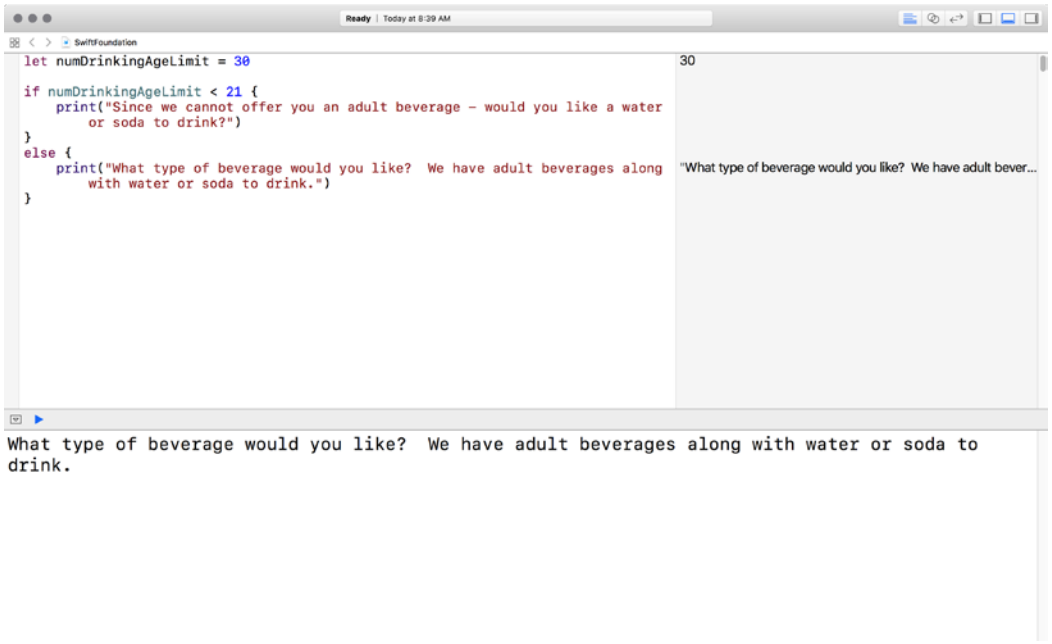
You can add else to both of the `if` statements we have inputted so far, but, for now, just add it to the end of our last `if` statement:

```
if numDrinkingAgeLimit < 21 {
  print("Since we cannot offer you an adult beverage - would you like
a water or soda to drink?")
}
else {
  print("What type of beverage would you like?  We have adult
beverages along with water or soda to drink.")
}
```



With `else` added onto the end of our `if` statement, it turns into an `if…else` statement, which now reads—if `numDrinkingAgeLimit` is less than `21`, then print `Since we cannot offer you an adult beverage—would you like a water or soda to drink?` Otherwise (or else), print `What type of beverage would you like? We have adult beverages along with water or soda to drink.`

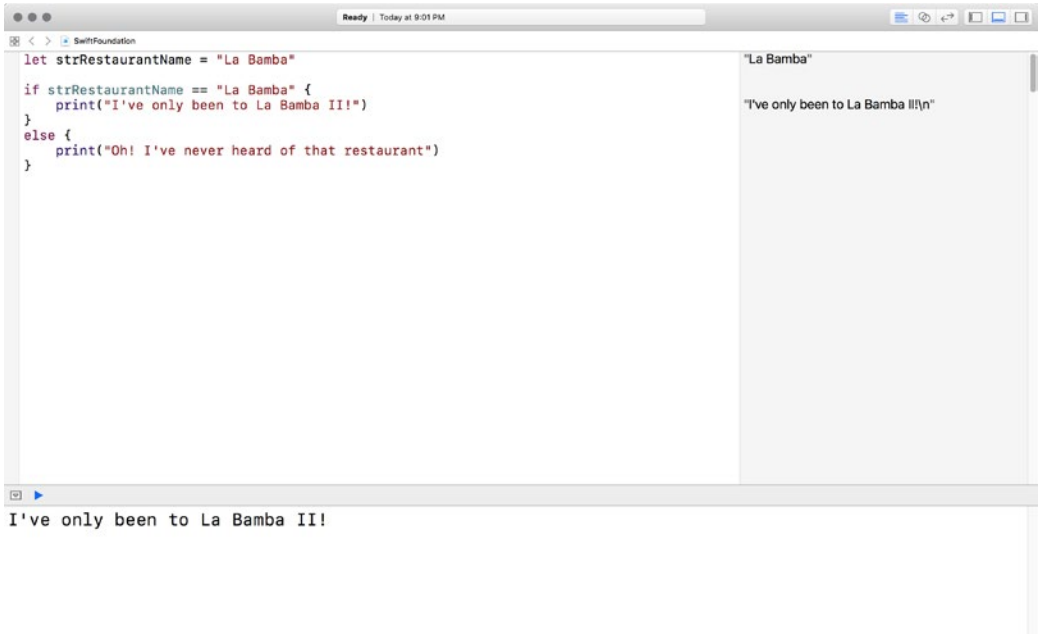
Now, our if…else statement can handle both conditions. Based on the value `19` for our `numDrinkingAgeLimit`, we can see in the Debug panel, `Since we cannot offer you an adult beverage—would you like a water or soda to drink?` If we change `numDrinkingAgeLimit` to `30`, our Debug panel says, `What type of beverage would you like? We have adult beverages along with water or soda to drink`. Go ahead and change `19` to `30` in Playgrounds:



Note that we got the behavior we wanted in the Debug panel.

So far, we have covered using an `if` statement with a `bool` and an Int. Let's take a look at one more example using a string. Add this next bit of code into Playgrounds:

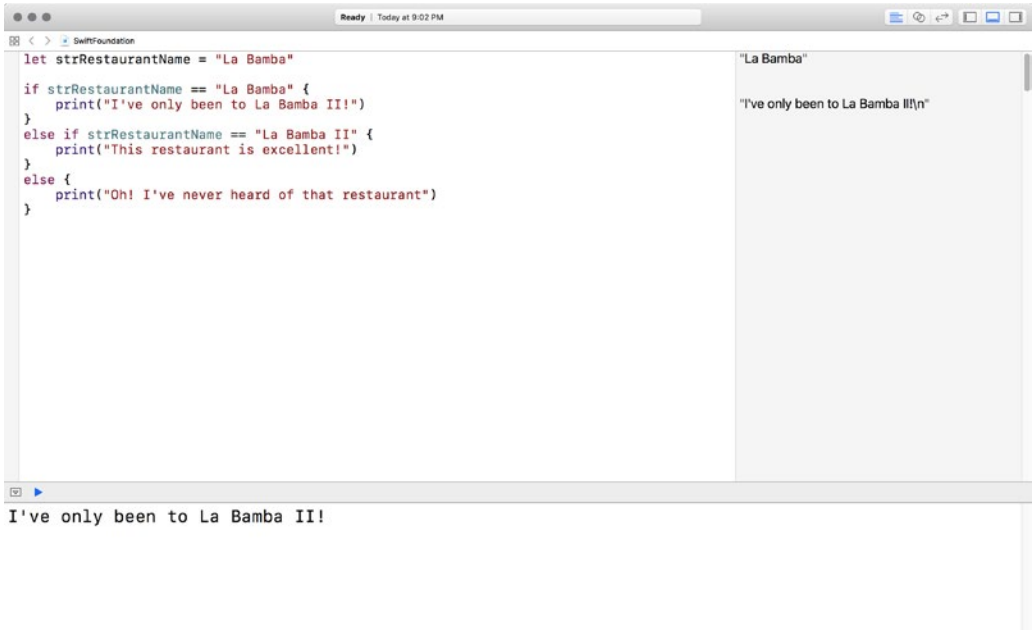
```
let strRestaurantName = "La Bamba"
if strRestaurantName == "La Bamba" {
  print("I've only been to La Bamba II!")
}
else {
  print("Oh! I've never heard of that restaurant")
}
```



In programming, we use equals (=) when setting data to variables. But in order to compare two data types, we must use the double equals (==). Therefore, when we write an if-statement that compares two strings we must use double equals (==) instead of just equals (=) to determine equality.

An `if…else` statement only lets us check two conditions, whether it is `true` or not. If we wanted to add more conditions, we would not be able to use simply an `if…else` statement. In order to accomplish this, we would use what is called an `if...else if...else`. This statement gives us the ability to add any number of else-ifs inside of our `if…else` statement. We will not go overboard, so let's just add one. Update your last `if…else` statement to the following:

```
if strRestaurantName == "La Bamba" {
  print("I've only been to La Bamba II!")
else if strRestaurantName == "La Bamba II" {
  print("This restaurant is excellent!")
}
else {
  print("Oh! I've never heard of that restaurant")
}
```



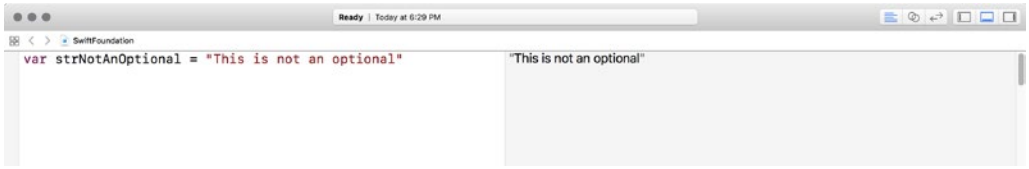
In this example of an `if…else if…else` statement, we are checking if `strRestaurantName` equals `La Bamba`, print `I've only been to La Bamba II!` else, if `strRestaurantName` equals `This restaurant is excellent`! else print `Oh! I've never heard of that restaurant`.

Using if, `if…else`, and `if…else if…else` statements really helps you create simple or complex logic for your app. Being able to use them with Strings, bools, Ints, and floating-point numbers gives you more flexibility.

Optionals and Optional Bindings

Optionals are used when a value cannot be set. Think of optionals as a container, which can take either a value or nil. This gives us the ability to check whether the value is nil or not. In order to create an optional value, you will have to give it a data type followed by a question mark (`?`). Before we do that, let's create a string that is not an optional. Add the following to Playgrounds:

```
var strNotAnOptional = "This is not an optional"
```



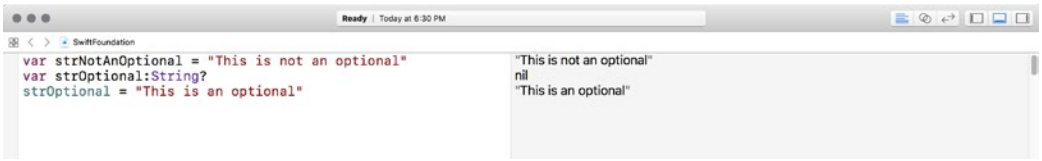
Now, let's add an optional to Playgrounds:

```
var strOptional:String?
```



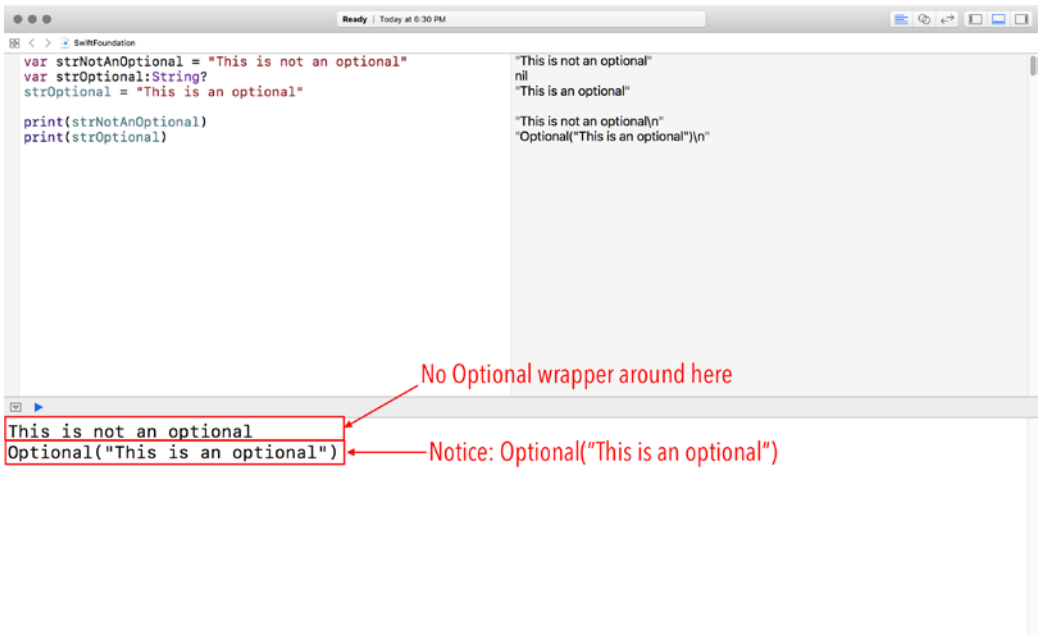
In this example, we created a string optional, and, if you notice in the Results panel, it is nil. But for our `strNotAnOptional`, we see `This is not an optional`. Now, on the next line, let's set `strOptional` equal to `This is an optional`:

```
strOptional = "This is an optional"
```



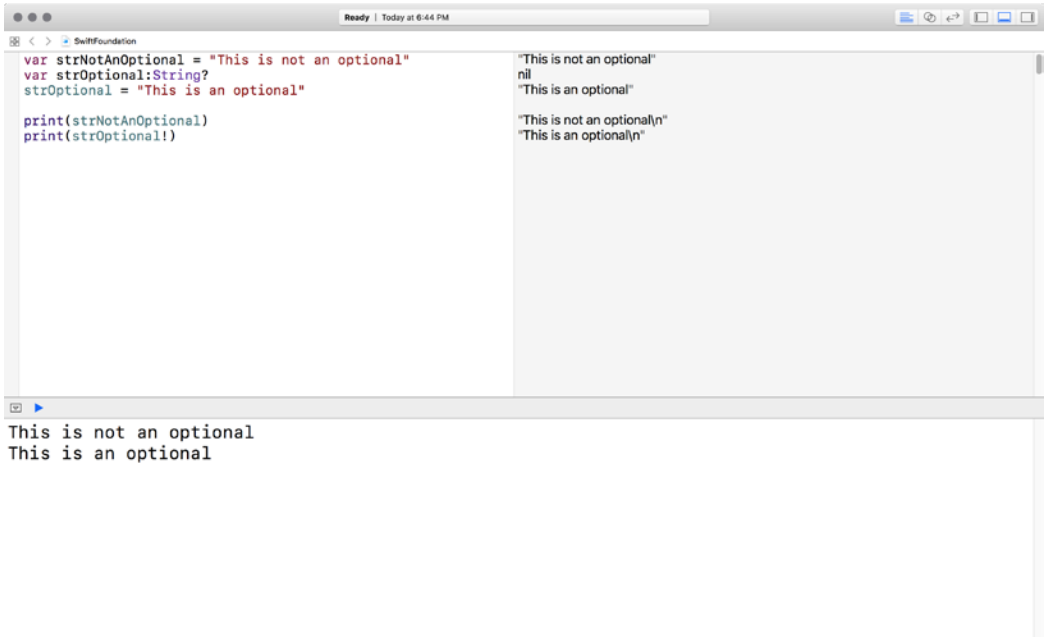
In our Results panel, we see `This is an optional`. Let's now print both `strNotAnOptional` and `strOptional`, as you will see a difference between the two:

```
print(strNotAnOptional)
print(strOptional)
```



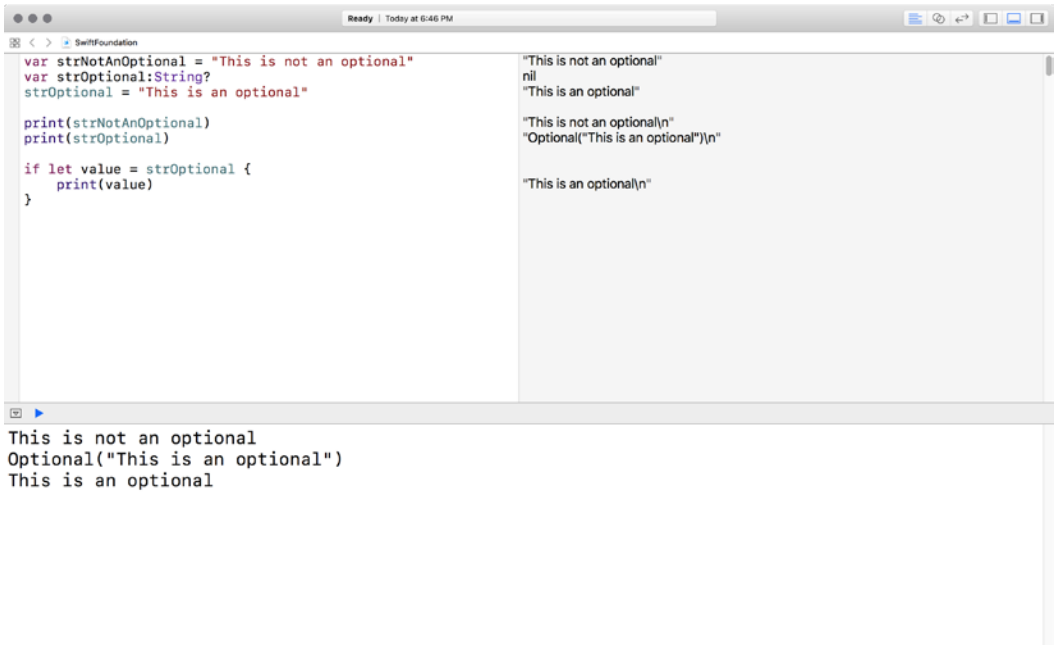
Note that our `strNotAnOptional` variable looks fine, but `strOptional` has optional wrapped (`""`) around the String, `This is an optional`. This means that in order for us to access the value, we must unwrap the optional. One way we could do this is by force unwrapping the optional using an (`!`). Let's update our `print` statement and change it to the following:

```
print(strOptional!)
```



We just forced unwrapped our optional, but this method is not recommended. We should use what is called optional binding, which is the safe way to access the value using an `if...let` statement. Remove the (`!`) from the `print` statement and instead write the following optional binding:

```
if let strValue = strOptional {
  print(strValue)
}
```



This `if…let` statement is saying that if the optional is not nil, set it to `strValue`—but, if this optional is nil, ignore it and do nothing. We now do not have to worry about anything setting our value and causing our app to crash.

Why optionals?

So now you are probably asking, `Why do you have to do this?` Trust me, when I first learned about optionals, I felt the same way. Optionals were actually made for your protection. For now, just understand that, when you see a data type followed by a question mark, this variable is an optional. As we work with optionals more and more throughout the book, it will become clearer to you.

Functions

Now, it is time to get into a really fun part of programming and learn how to write functions. Functions are self-contained pieces of code that you want to run on something. In Swift 3, Apple has made a change to how you should write functions. All of the functions we will write in this chapter will perform an action (think of verbs). Let's create a simple function called `greet()`:

```
func greet() {
  print("Hello")
}
```

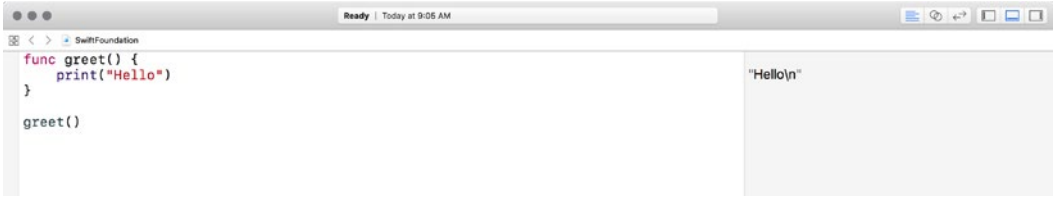


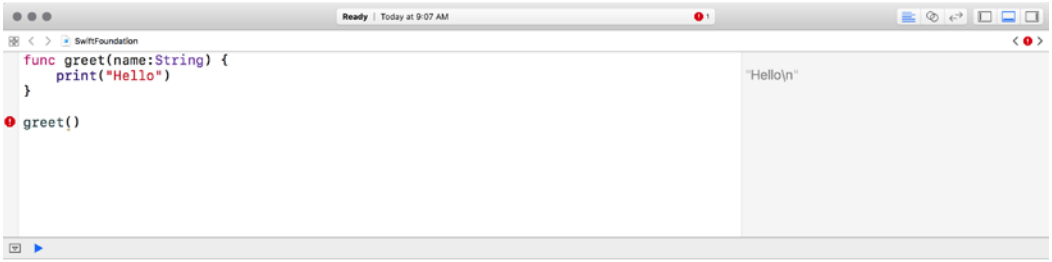
This example is a basic function with a `print` statement in it. In programming, functions do not actually run until you call them. We call a function simply by calling its name. So, let's call `greet`:

```
greet()
```

That's it! We just created our first function and called it, but functions can do so much more. We can add what is called a parameter to a function. A parameter allows us to accept data types inside of our parentheses. By doing this, it allows us to build more reusable chunks of code. So, let's update our `greet()` function to accept a parameter called `name`:

```
func greet(name:String) {
  print("Hello")
}
```





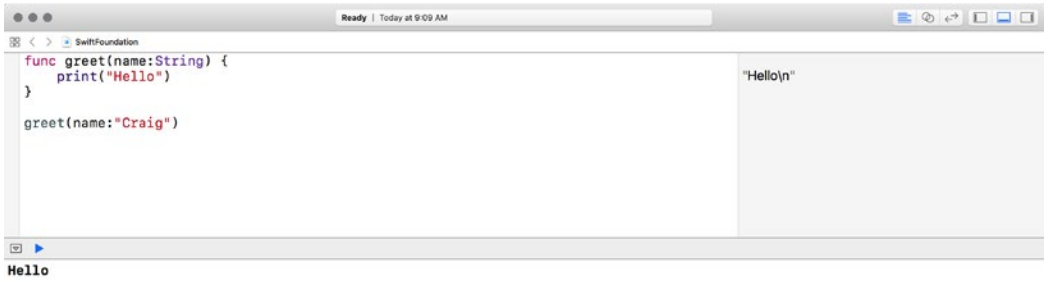

```
Playground execution failed: error: SwiftFoundation.playground:5:6: error: missing argument for parameter 'name' in call
greet()
     ^

SwiftFoundation.playground:1:6: note: 'greet(name:)' declared here
func greet(name:String) {
     ^


* thread #1: tid = 0x78a4a, 0x00000001022213c0 SwiftFoundation`executePlayground, queue = 'com.apple.main-thread', stop reason
= breakpoint 1.2
  * frame #0: 0x00000001022213c0 SwiftFoundation`executePlayground
    frame #1: 0x00000001022209c0 SwiftFoundation`__37-[XCPAppDelegate enqueueRunLoopBlock]_block_invoke + 32
    frame #2: 0x0000000102d3b25c CoreFoundation`__CFRUNLOOP_IS_CALLING_OUT_TO_A_BLOCK__ + 12
    frame #3: 0x0000000102d20304 CoreFoundation`__CFRunLoopDoBlocks + 356
    frame #4: 0x0000000102d1fa75 CoreFoundation`__CFRunLoopRun + 901
    frame #5: 0x0000000102d1f494 CoreFoundation`CFRunLoopRunSpecific + 420
    frame #6: 0x000000010812aa6f GraphicsServices`GSEventRunModal + 161
    frame #7: 0x00000001038c9f34 UIKit`UIApplicationMain + 159
    frame #8: 0x00000001022206e9 SwiftFoundation`main + 201
    frame #9: 0x000000010627768d libdyld.dylib`start + 1
```

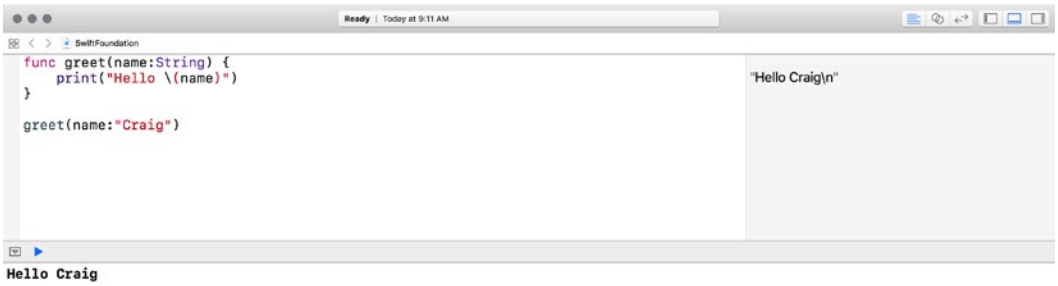
Notice the preceding error. We received this error because we updated our function, but we did not update the line where we called it. Let's update where we call `greet()` to the following:

```
greet(name: "Craig")
```



This looks good; however, the Debug panel shows us that we are not using the name in our greeting. Earlier, you learned how to create a string interpolation. So, we just need to append our variable name inside of our `print` statement as follows:

```
print("Hello \(name)")
```

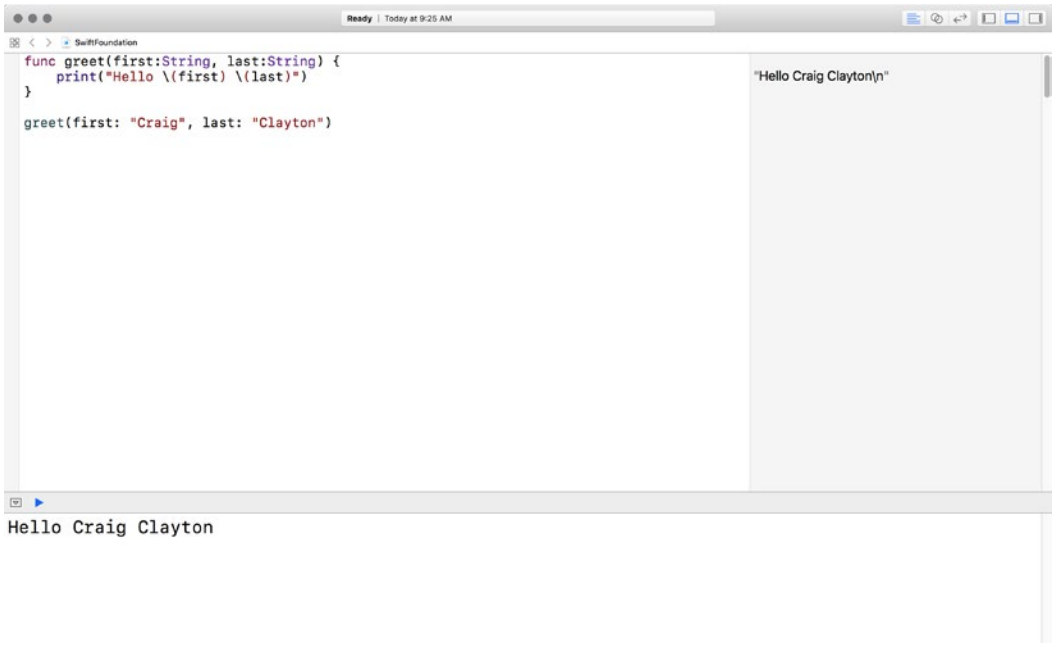


Functions can take multiple parameters, so let's update our `greet()` function to take a first and last name:

```
func greet(first:String, last:String) {
  print("Hello \(first) \(last)")
}
```

We also need to update where we called `greet` to accept multiple parameters as well:

```
greet(first: "Craig", last: "Clayton")
```

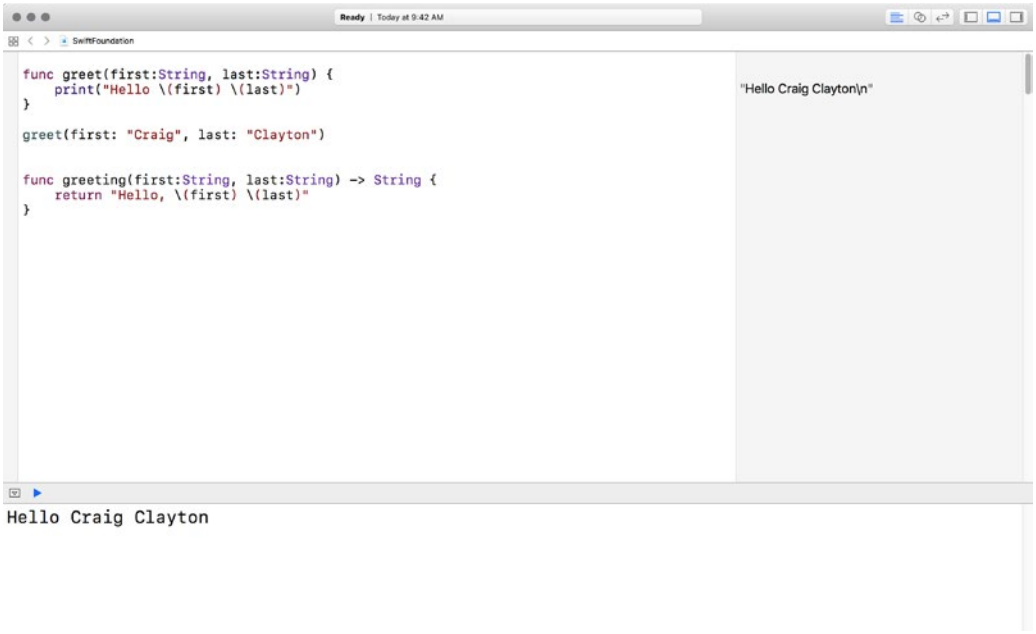


We now have a function that accepts multiple parameters.

What would be great is if we could make this function return the greeting to us. Well, we can! Whenever we want our function to return something, we need to use a noun as a way to describe what our function will do. We just created a function called `greet()` that takes a first and last name and creates a full name.

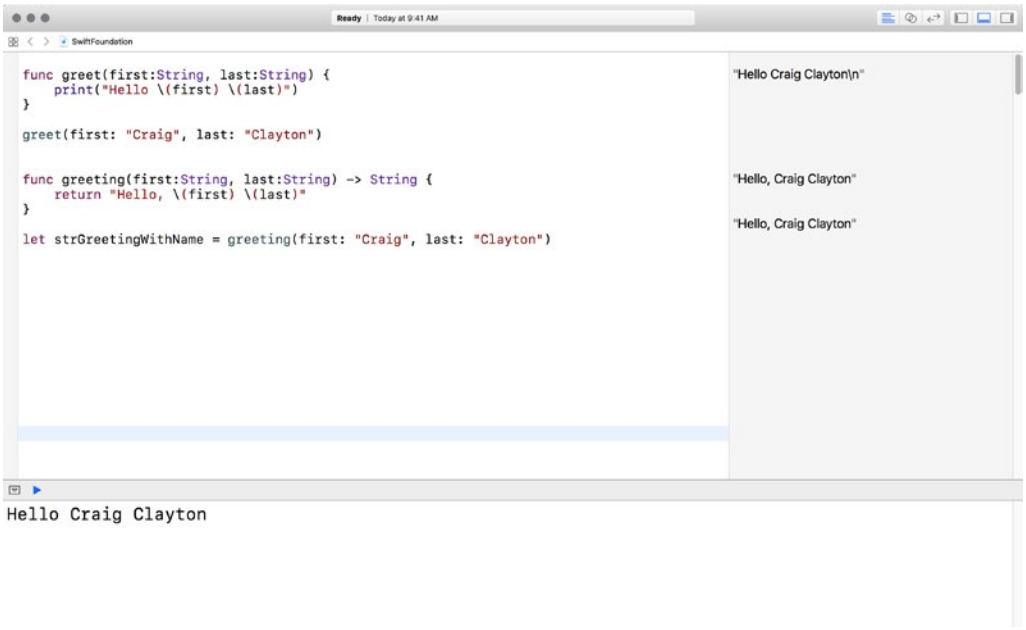
Now, let's create another function called `greeting()`, which will return a full name back with a greeting. Let's see what this looks like:

```
func greeting(first:String, last:String) -> String {
  return "Hello \(first) \(last)"
}
```



This function is almost the same as the previous one, but with a couple of new things. `First -> String` tells the function that we want to return a string. Inside of our function, we return `Hello (first) (last)`. Since we said that we want to return something after our parentheses, then we have to do just that. Now, let's see how we do this. Enter the following:

```
let strGreetingWithName = greeting(first:"Craig", last:"String")
```



If you notice in the Results panel, we now have our full name with `Hello` added to the beginning. As you start to build on functions, you really start to see the power.

These are just the basics of functions. We will cover more advanced functions throughout our *Let's Eat* app. The biggest thing beginning programmers forget is that functions should be small. Your function should do one thing and one thing only. If your function is too long, then you need to break it up into smaller chunks. Sometimes longer functions are unavoidable, but you should always be mindful of keeping them as small as possible. Nice work!

Let's Work

We covered a lot in this chapter, and now it is time to put everything we covered into practice. Here are two challenges. If you are comfortable with them, then work on them on your own. Otherwise, go back into this chapter, and you can follow along with me and see how to do each one:

- **Challenge 1**: Write a function that accepts and returns a custom greeting (other than `Hello`, which we addressed earlier in this chapter) along with your first and last name.
- **Challenge 2**: Write a function that will take two numbers and either add, subtract, multiply, or divide those two numbers.

Summary

In this chapter, we learned about how Playgrounds is an interactive coding environment. In Playgrounds, we worked our way through the basics starting with data types and variables. We moved on to type safety and type inference along with operations and if-statements. Finally, we discussed the power of optionals and learned about what functions are and how to use them. In the next chapter, we will move on to some more Swift basics before we start building out our Let's Eat app.

3
Digging Deeper

When I first started programming, I was in my mid-twenties. I started a lot older than most, but I will say that grasping the basics took me a bit longer than most. I remember when I bought my first programming book, and I read and reread chapters over and over again until the concepts made sense to me. I found that a lot of books talked to me like I had majored in computer science. As you progress through this book, take your time—and, if you need to go back, it is okay to do so. No one is going to care that it took you an extra day to understand a concept. It is more important that you fully understand that concept.

One tip I would give you is to not copy and paste code. No matter where you find the code and no matter how long it takes, it benefits you to type it out. Doing this really helped me as I eventually started to remember the code and it became second nature to me.

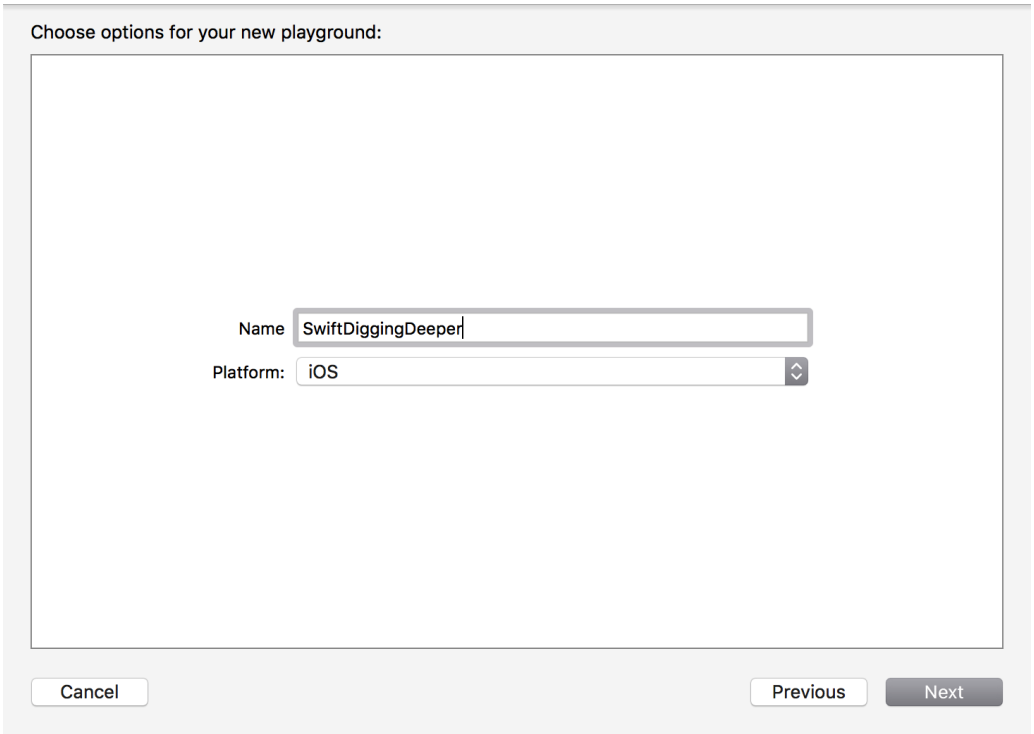
In the last chapter, we went over the basics of Swift to get you warmed up. Now, we will dig deeper and learn some more programming concepts. These concepts will build on what you have already learned. In this chapter, we will cover:

- Ranges
- Control flow

Let's begin by creating a new Playground project.

Creating a Playground project

As you learned earlier, launch Xcode and click on **Get started with a playground**. The options screen for creating a new playground screen will appear:



Name your new Playground `SwiftDiggingDeeper`, and make sure that your **Platform** is set to `iOS`. Next, delete everything inside of your file and toggle on the Debug panel using the toggle button (*Cmd* + *Shift* + *Y*). You should now have a blank screen with the Results panel on the right and the Debug panel on the bottom opened.

We focused on the basics earlier—and now we will build upon those skills. Ranges are one such data type that we should learn and are very useful and can come in handy for a variety of reasons. Let's take a look at what Ranges are and then start to understand the difference between a *closed Range* and a *half closed Range*.

Ranges

Ranges are generic data types that represent a sequence of numbers. Let's look at the following image to understand:

10 11 12 13 14 15 16 17 18 19 20

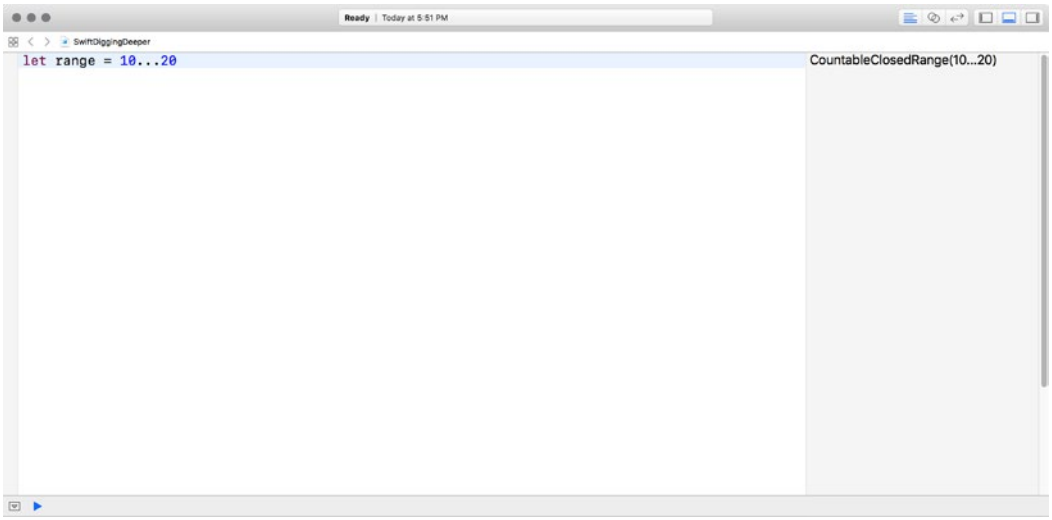
Closed Range

Notice that, in the preceding image, we have numbers ranging from **10** to **20**. Rather than having to write each value, we can use Ranges to represent all of these numbers in shorthand form. In order to do this, let's remove all of the numbers in the image except **10** and **20**:

10 11 12 13 14 15 16 17 18 19 20

Now that we have removed those numbers, we need a way to tell Swift that we want to include all of the numbers that we just deleted. This is where the range operator (...) comes into play. Therefore, in Playgrounds, let's create a constant called range and set it equal to `10...20`:

```
let range = 10...20
```

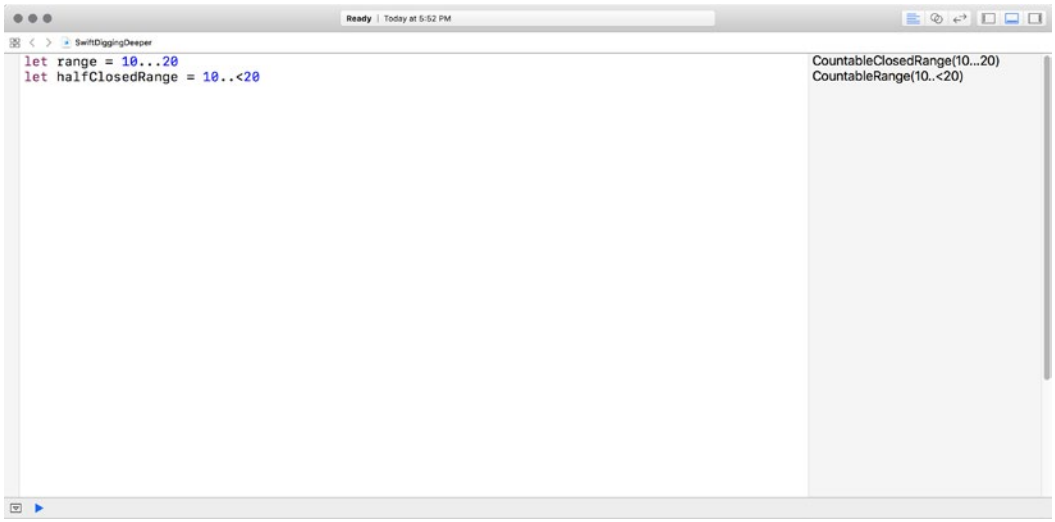


The range that we just entered says that we want the numbers between `10` and `20` as well as both `10` and `20` themselves. This type of Range is known as a closed Range. We also have what is called a half closed Range.

Half closed Range

Let's make another constant that is known as a half-closed Range and set it equal to `10<20`. Add the following into Playgrounds:

```
let halfClosedRange = 10..<20
```



A half-closed Range is the same as a closed Range, except that the end value will not be included. In this example, that means that `10` through `19` will be included, and `20` will be excluded.

At this point, you will notice that your Results panel shows you `CountableClosedRange(10...20)` and `CountableRange(10..<20)`. We cannot see all the numbers within the Range. In order to see all the numbers, we need to use a loop.

Control flow

In programming, control flow is the order in which your code is executed. When working with Swift, we can use a variety of control statements. Loops, in particular, are useful for when you want to repeat a task multiple times. Let's take a look at a few different types of loop.

The for…in loop

One of the most common control statements is a `for…in` loop. It allows you to iterate over each element in a sequence. Let's see what a `for…in` loop looks like:

```
for <value> in <sequence>  {
  // Code here
}
```

We start the `for…in` loop with `for`, which is proceeded by `<value>`. This is actually a local constant (only the `for…in` loop can access it) and can be any name you like. Typically, you will want to give this value an expressive name. Next, we have `in`, which is followed by `<sequence>`. This is where we want to give it our sequence of numbers. Let's write the following into Playgrounds:

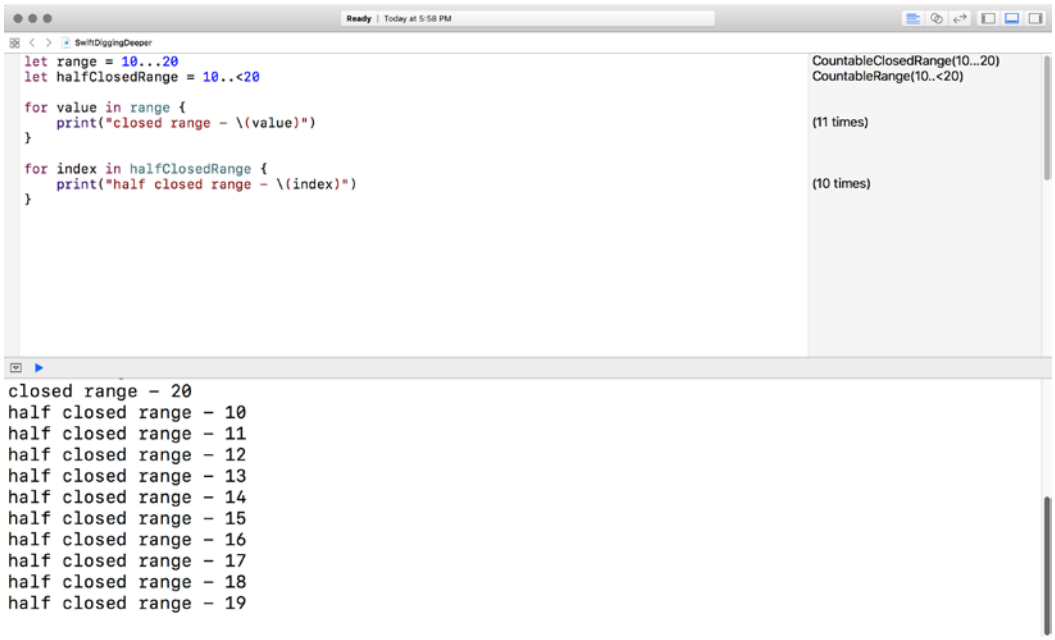
```
for value in range  {
  print("closed range - \(value)")
}
```



Notice that, in our Debug panel, we see all of the numbers we wanted in our range.

Let's do the same for our `halfClosedRange` variable by adding the following:

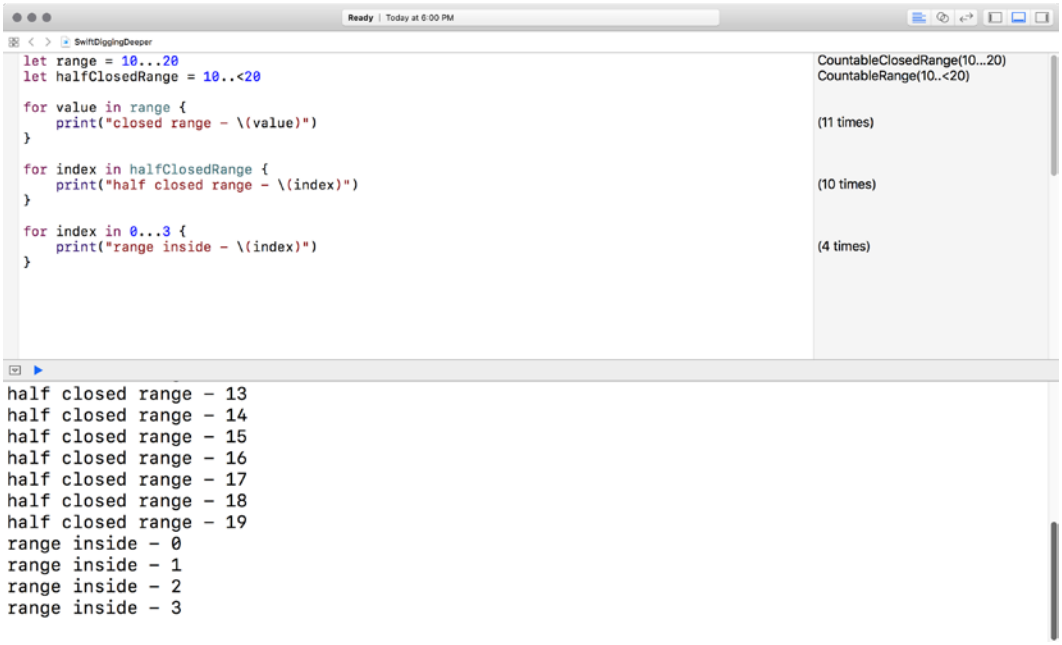
```
for index in halfClosedRange  {
  print("half closed range - \(index)")
}
```



In our Debug panel, we see that we get the numbers `10` through `19`. One thing to note is that these two `for...in` loops have different variables. In the first loop, we used `value`, and in the second one, we used `index`. You can make these whatever you choose them to be.

In addition, in the two preceding examples, we used constants, but we could actually just use the Ranges within the loop. Add the following:

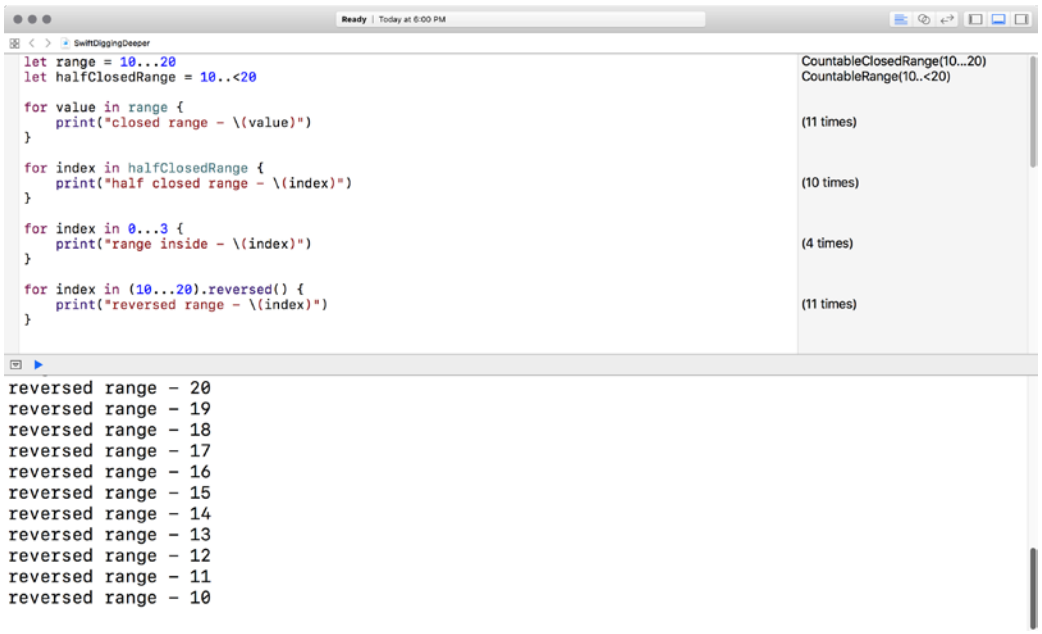
```
for index in 0...3  {
  print("range inside - \(index)")
}
```



Now, you see `0` to `3` print inside of the Debug panel.

What if you wanted numbers to go in reverse order? Let's input the following `for...in` loop:

```
for index in (10...20).reversed()  {
  print("reversed range - \(index)")
}
```



We now have the numbers in descending order in our Debug panel. When we add Ranges into a `for...in` loop, we have to wrap our range inside parentheses so that Swift recognizes that our period before `reversed()` is not a decimal.

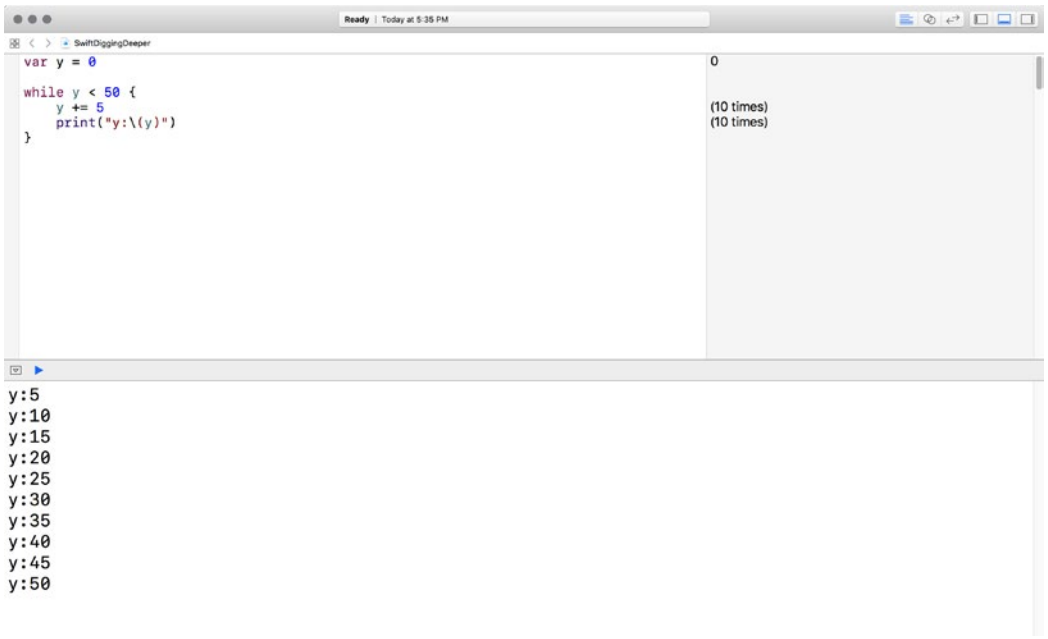
The while loop

A `while` loop executes a bool expression at the start of the loop, and the set of statements run until a condition becomes false. It is important to note that `while` loops can be executed zero or more times. Here is the basic syntax of a `while` loop:

```
while <condition>  {
  // statement
}
```

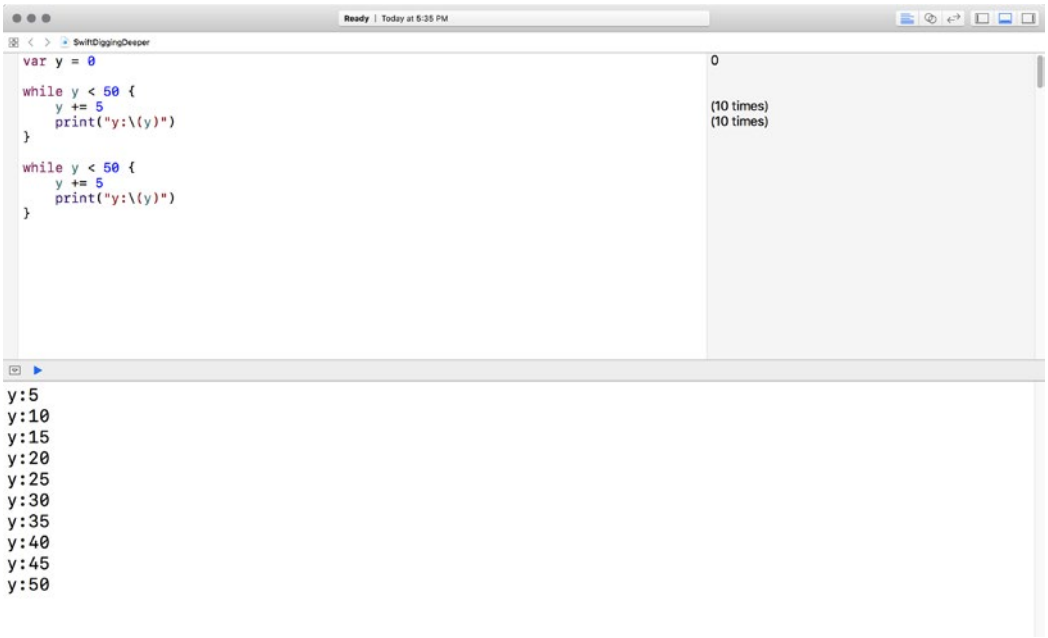
Let's write a `while` loop in Playgrounds and see how it works. Add the following:

```
var y = 0
while y < 50 {
  y += 5
  print("y:\(y)")
}
```



So, this `while` loop starts with a variable that begins at zero. Before the `while` loop executes, it checks to see if `y` is less than `50`—and, if so, it continues into the loop. Using the `+=` operator, which we covered earlier, we increment `y` by `5` each time. Our `while` loop will continue to do this until `y` is no longer less than `50`. Now, let's add the same `while` loop after the one we created and see what happens:

```
while y < 50 {
  y += 5
  print("y:\(y)")
}
```



You will notice that the second `while` loop never runs. This may not seem like it is important until we look at our next type of loop.

The repeat…while loop

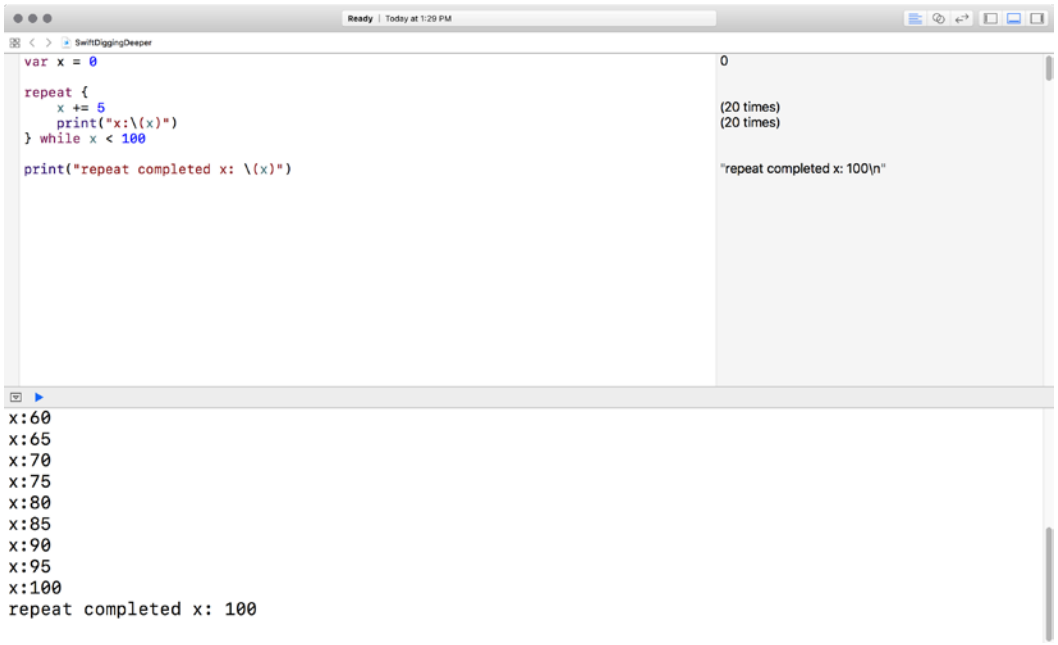
The `repeat...while` loop is pretty similar to a `while` loop in that it continues to execute the set of statements until a condition becomes false. The main difference is that the `repeat...while` loop does not evaluate its `bool` condition until the end of the loop. Here is the basic syntax of a `repeat...while` loop:

```
repeat {
  // statement
} <condition>
```

Let's write a `repeat...while` loop in Playgrounds and see how it works. Add the following into Playgrounds:

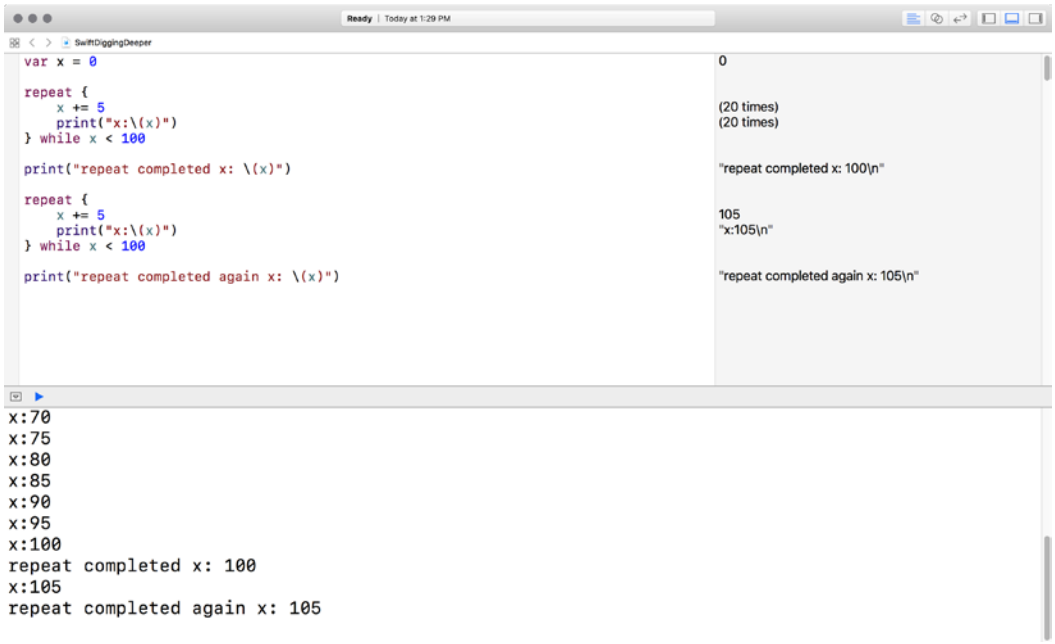
```
var x = 0

repeat {
  x += 5
  print("x:\(x)")
} while x < 100
print("repeat completed x: \(x)")
```



You will notice that our `repeat...while` loop executes first and increments `x` by `5`; and after (as opposed to checking the condition before, as with a `while` loop), it checks to see if `x` is less than `100`. This means that our `repeat...while` loop will continue until the condition hits `100`. But here is where it gets interesting. Let's add another `repeat...while` loop after the one we just created:

```
repeat {
  x += 5
  print("x:\(x)")
} while x < 100
print("repeat completed again x: \(x)")
```



Now, you can see that our `repeat...while` loop incremented to `105` instead of `100`, like the previous `repeat...while` loop. This happens because the bool expression does not get evaluated until after it is incremented by `5`. Knowing this behavior will help you pick the right loop for your situation.

Summary

So far, we have looked at three loops: the `for…in` loop, the `while` loop, and the repeat-while loop. We will use the `for…in` loop again, but first we need to talk about collections. In the next chapter, we will focus on what are collections and how to use them when working with data. Make sure you fully understand loops, because we will build on them in the next chapter and throughout the book. Therefore, review as much as you need in order to make sure you feel that you are proficient in the topics contained in this chapter.

4
Digging into Collections

In the last couple of chapters, we reviewed the basics of Swift to get you warmed up. Before we start building our app, we need to look at one more programming concept—collections. In Swift, we have three primary collection types, which we will cover in this chapter:

- Arrays
- Dictionaries
- Sets

We will dig deeper into each one, but we will start with the most common collection type—arrays.

Arrays

Arrays are an ordered collection of values and can hold any number of items. For example, a list of Strings, Ints, floating-point values, and so on. Arrays are stored in an ordered list, starting at 0. Let's look at a diagram:

0	Florida
1	California
2	Ohio
3	North Carolina
4	Colorado
5	Nevada
6	New York

0	45
1	66
2	23
3	10
4	88

0	Florida
1	California
2	32
3	New York
4	99
5	true
6	9.0

Starting from left to right in the preceding examples, we first have an array that holds a collection of Strings. In the second example, we have another array that holds a collection of Ints. In our third example, we have an array that holds a collection of floating-point values.

Now, let's review the following diagram that looks like an array, but actually is not one:

0	Florida
1	California
2	32
3	New York
4	99
5	true
6	9.0

Since this example contains mixed data types, such as Strings, Ints, and bools, it is not considered an array. If you enter this, you will receive an error message.

An array can hold any data type, but since an array is strongly typed, every element in it must be of the same type.

Creating an empty array

Let's now create a few arrays in Playgrounds.

Sometimes, you may want to remove your prior entries from your Playground, so that it makes it easier for you to see each new print statement. Do that now and input the following:

```
let arrOfInts:[Int] = []
let arrStrings = [String]()
```

We just created our first two arrays. The data types within each set of brackets tells Swift what type of an array we want to create. The first array (`arrOfInts`) we created has a data type of Ints, and our second array (`arrStrings`) has a data type of Strings.

Creating an array with initial values

Arrays can have initial values when they are created. Let's see how this would look by entering the following in Playgrounds:

```
let arrOfMoreInts = [54, 29]
```

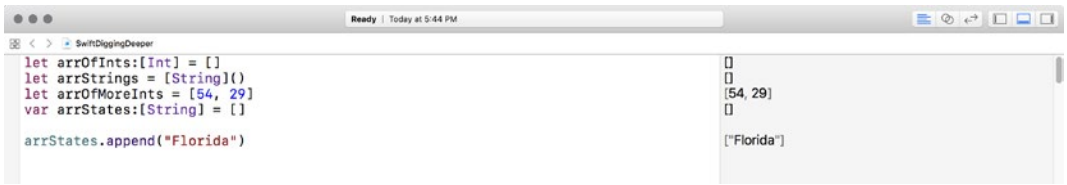


The array that we just entered uses type inference to declare the data type of the array using its initial values. In this case, Swift understands that it is an array of Ints, because the values we entered are integers. In addition, when we use a constant (`let`) on an array, we are telling Swift that the array is an immutable array, which means that the contents or size cannot change once it is instantiated.

Creating a mutable array

It is best practice to make all arrays (and for that matter, collections) immutable, but there are some cases where you will need to create an array that is mutable. Let's have some fun and create a mutable array:

```
var arrStates:[String] = []
```



As an aside, when creating a mutable array (or any variable), note that each variable must be unique.

One use for a mutable array is so that we can change an array. Let's look at some ways we can do this.

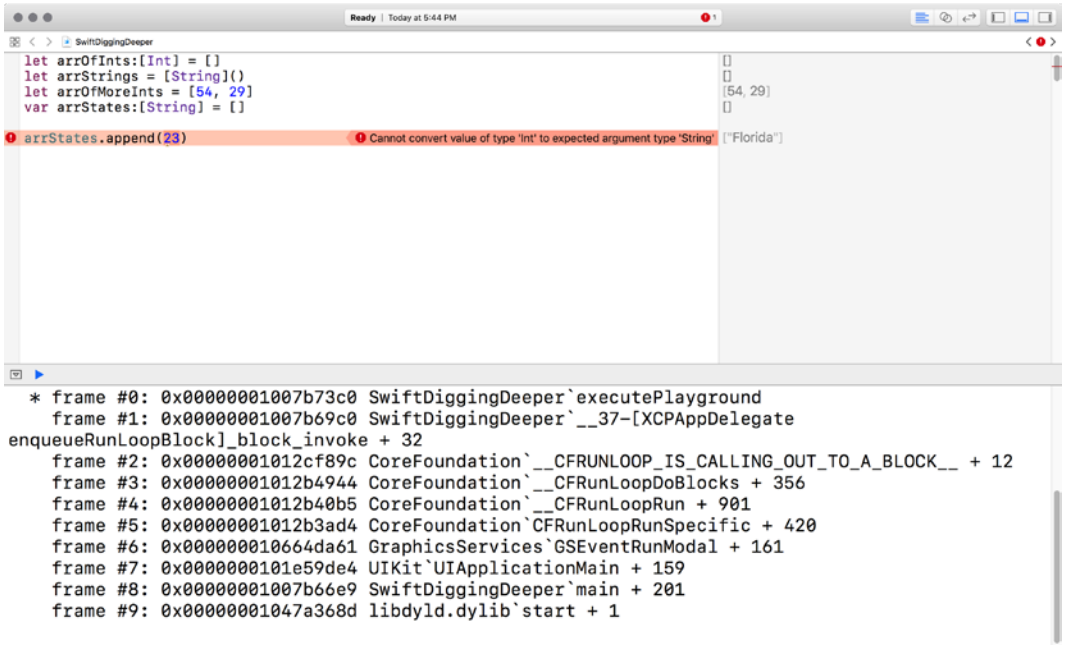
Adding items to an array

Let's add some data to our array. There are a few different convenience methods for adding data to an array.

A convenience method is just as it implies, a method that makes things convenient. A method is a function that lives inside of a class. We will discuss classes later in this book. If this is starting to get overwhelming, it is understandable. You do not need to worry about every single detail at this time. We will cover this again, and things will slowly start to click at some point. Everyone learns differently, so there is no reason to worry if someone else understands something more quickly. Just go at your own pace.

The first convenience method, which we will look at, is the `append()` method:

```
arrStates.append(23)
```



Houston, we have a problem! Actually, I did this for a couple of reasons. Getting errors is normal and common. Most people who start out coding are afraid to make a mistake or get scared about getting or seeing errors. Trust me, I have been coding for years, and I make mistakes all the time. The error is telling us that we tried to add an Int into an array that only holds Strings.



Every developer, from a beginner to an experienced one, will face a time when he or she will encounter an error that he or she cannot figure out. This error might get you frustrated to the point where you want to throw the computer across the room (I have been there a few times). The best advice my boss ever gave me was to take a walk for 10-15 minutes or do something to take your mind off of it. Sometimes this helps, and you will come up with an idea after you walk away. Even if you come back, and it still takes you hours to figure out what is wrong, this is still part of the process. The best errors are the ones that were the simplest thing you overlooked, and you had to spend hours trying to figure out. You might have lost time, but you will have learned a great lesson. Lessons like these will stay with you forever, and you will never forget the error the next time you encounter it. So, if your coding results in an error, even in this book, embrace the challenge, because there is no greater feeling than figuring out a challenging error.

So, let's correct what we just did above by revising the array to show the following:

```
arrStates.append("Florida")
```

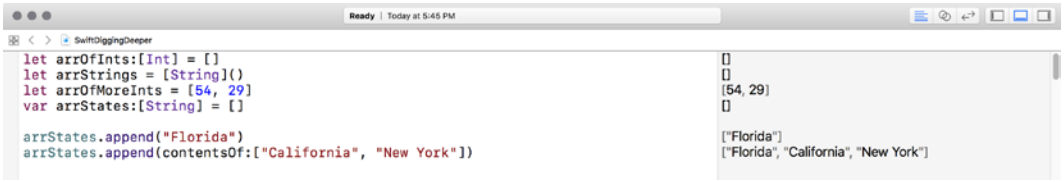


In the Results panel, you can actually see the contents of our corrected array.

Since an array can hold any number of items, let's add some more. Earlier, I mentioned that we have a variety of ways to add items to an array. The `append()` method allows us to add only one item at a time. In order to add multiple items, we can use the convenience called `append(contentsOf:)`.

Add the following to Playgrounds:

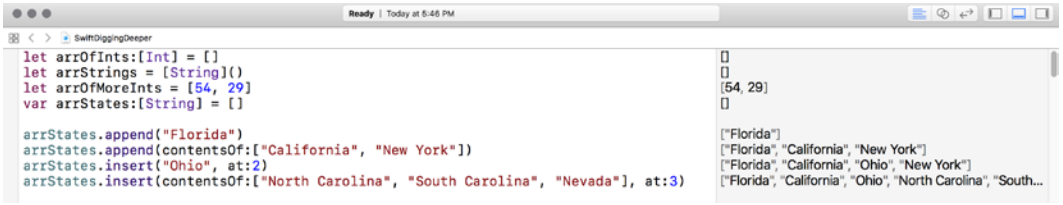
```
arrStates.append(contentsOf:["California", "New York"])
```



We added two more items into our array, but, so far, every example we have utilized has added items at the end of our array. We have two convenience methods that allow us to add items at any index position that is available in the array.

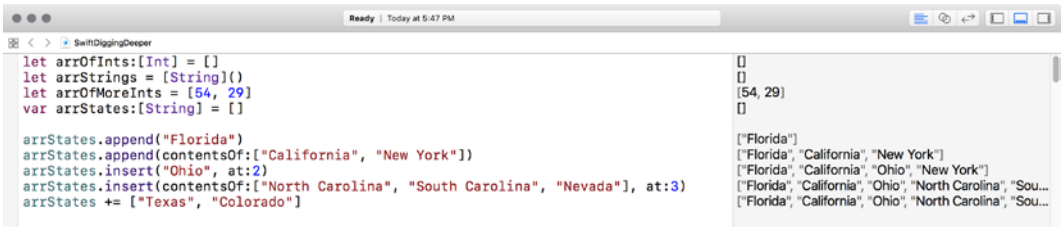
The first method we can use to do this is called `insert(at:)`, which allows us to add a single item at a certain index position. We also have `insert(contentsOf:at:)`, which allows us to add multiple items into an array at a certain index position. Let's use them both and add `Ohio` after `California` and then `North Carolina`, `South Carolina`, and `Nevada` after `Ohio`:

```
arrStates.insert("Ohio", at:1)
arrStates.insert(contentsOf:["North Carolina", "South Carolina",
"Nevada"],at:3)
```



We just added items to our array using append (`contentsOf:`), but there also is a shorthand version of this using the `+=` operator. Let's add the following:

```
arrStates += ["Texas", "Colorado"]
```

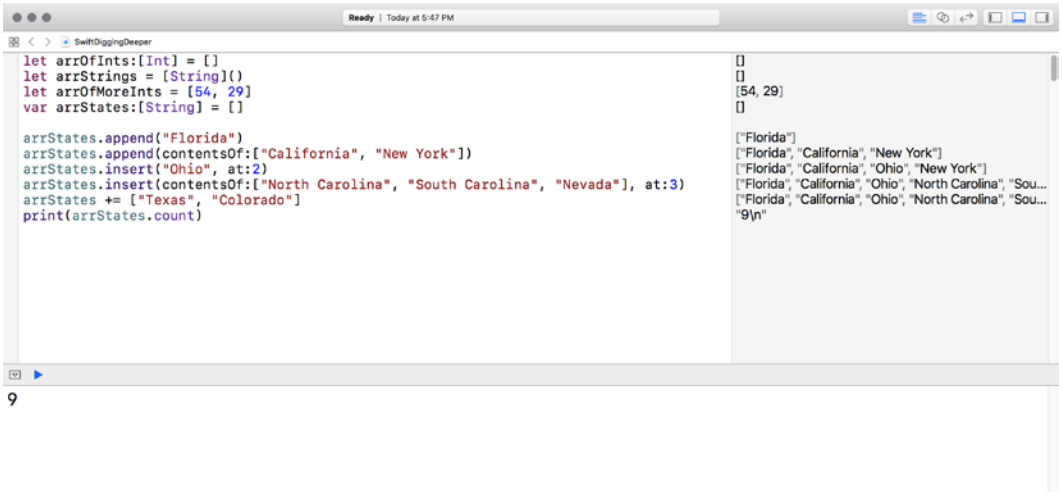


This technique for adding items is much more concise and is my preferred way of inserting items into an array. Writing less code is not always better; but, in this case, using the `+=` operator is my go to method.

Checking the number of elements in an array

If you are keeping track, we now have nine items in our array. But luckily, we do not have to keep track of how many items are in our array, because we have a property called count. This property will keep track of the current item count and give us the total count of our array when we want to check. Let's print our current count:

```
print(arrStates.count)
```

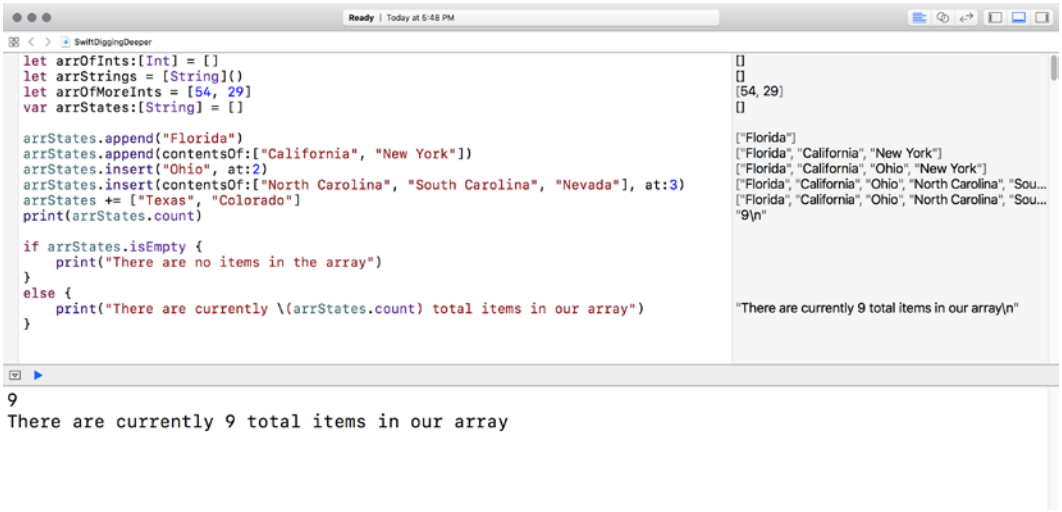


Checking for an empty array

The `count` property is not the only property we can use to calculate how many items are in an array. The most commonly used property for an array is called, `isEmpty`. This property uses the count property by checking to see if the count is greater than `0`. Using this method will return true or false, depending on whether there are any items within our array. Since you learned that `if...else` statements work well with bools, let's use this `isEmpty` property in an `if...else` statement.

Add the following into Playgrounds:

```
if arrStates.isEmpty {
  print("There are no items in the array")
}
else {
  print("There are currently (arrStates.count) total items in our
array")
}
```



Now, our Debug panel is printing out `There are currently 9 total items in our array`.

One thing to remember in programming is that sometimes there are multiple ways of writing a piece of code. It is not shocking to meet someone who will approach the same problem differently than you did. To me, this is why programming is so great. Ultimately, all that matters is that it works as expected, especially when you are new to programming.

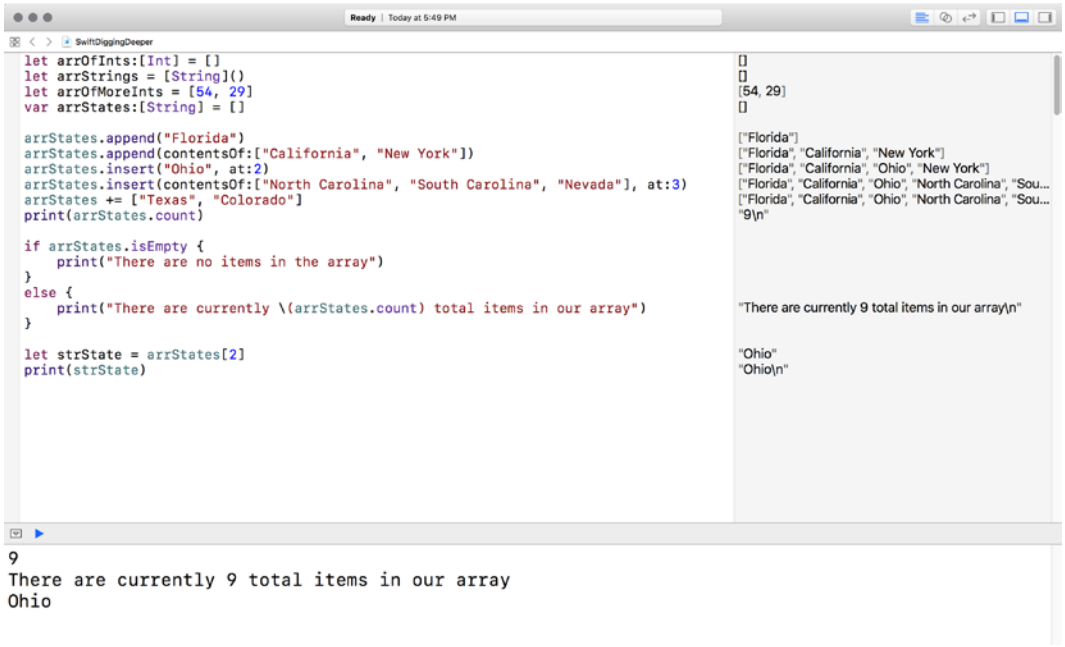
All programming languages have what is known as a style guide, which is a preferred way to write code; and it is no different in Swift. Preferred just means a suggested method, but, even then, you will notice that most preferred methods vary on certain things. For now, you do not need to worry about different style guides other than to know that they exist. In this book, we will follow a style that I have adopted into my code.

Once you get comfortable, I recommend that you start to look at style guides and adapt them into your code. Knowing different styles helps you to know your options as well as to understand what others are doing with their code, even if you do not agree with how they write something. If you write your code with a certain structure or style throughout a project, it will make it easier for you to come back to your code if you, for instance, had to take a break for some reason, such as starting another project or just taking some time off.

Retrieving a value from an array

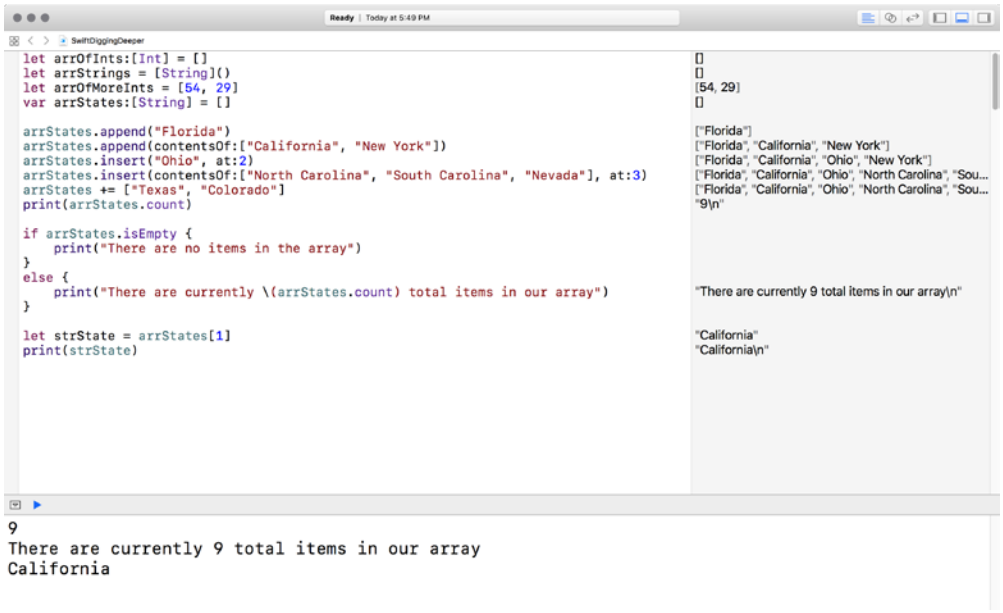
We discussed creating arrays as well as adding items into an array. Now, let's turn to retrieving a value from an array. Since arrays are stored by their index, we can use their index to retrieve values. If we wanted to retrieve `California` we would do the following:

```
let strState = arrStates[2]
print(strState)
```



The Results and Debug panels both show `Ohio` and not `California`. Remember, arrays start at `0` not `1`. Therefore, in order for us to get `California`, we would actually need use the index position of `1`. Let's make that update in Playgrounds:

```
let strState = arrStates[1]
print(strState)
```

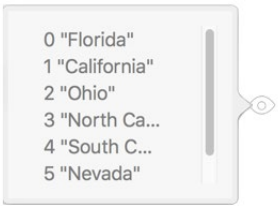


There we go!

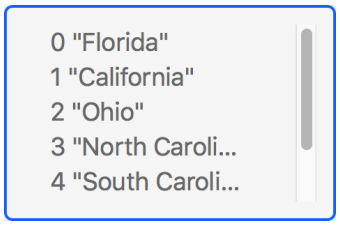
Now, if we would like to view the entire contents of our array, we can in Playgrounds. In the Results panel, hover over your array, and you will see two icons appear:



The left icon is called, `Quick Look`, and will do the following:

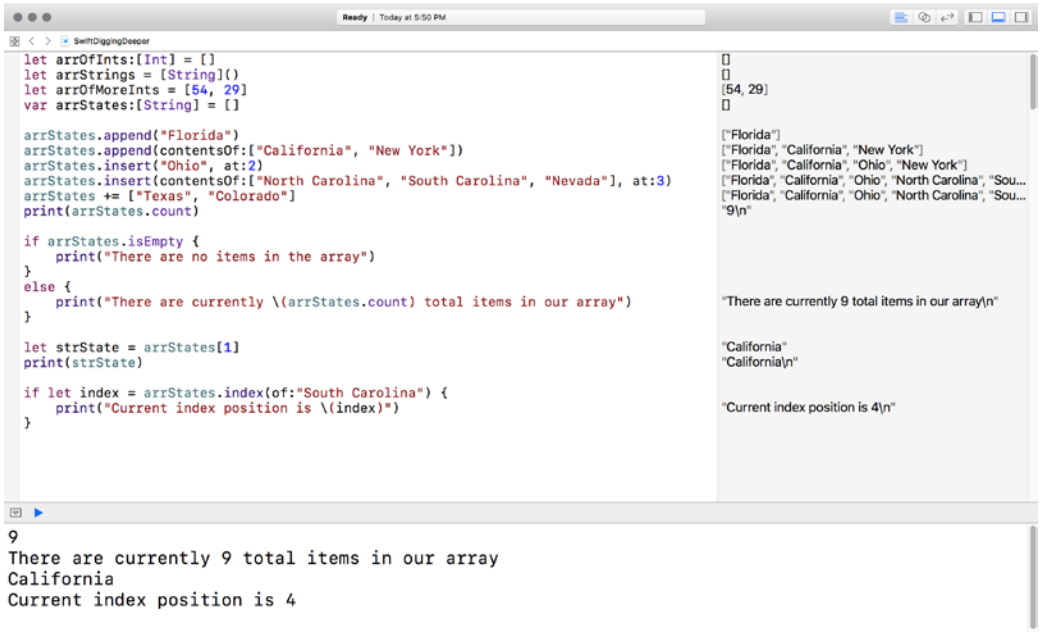


The right icon is called, `Show Results`, and will do the following:



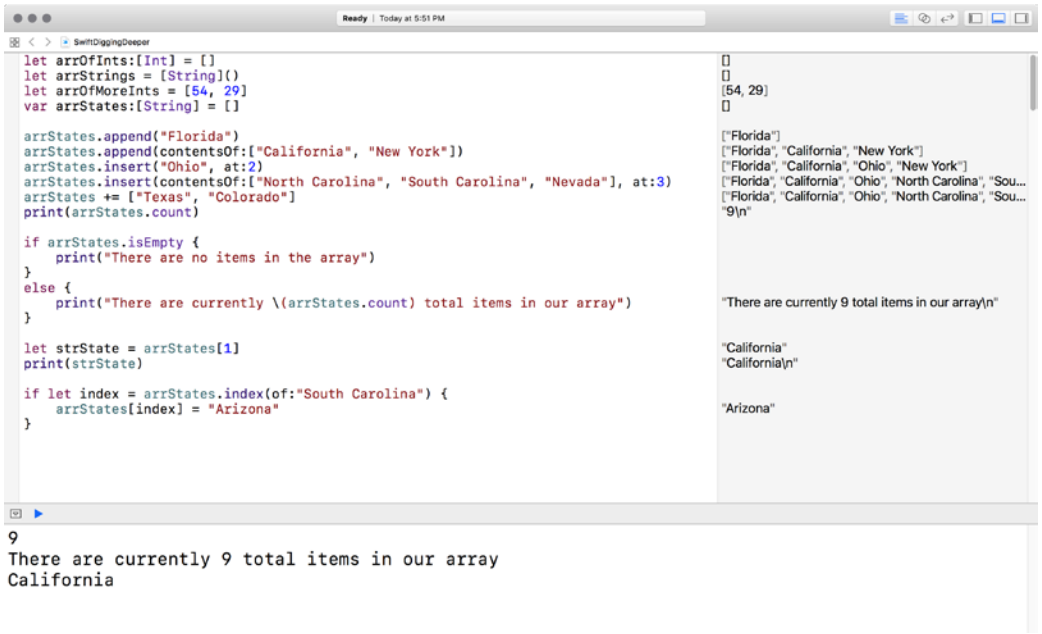
We now have this great list of states, but someone told you that `Arizona` also is amazing. Instead of just adding `Arizona` to our list, you decide that you actually prefer to replace `South Carolina` with `Arizona`. We could simply look at our array and see in which index `South Carolina` is. This would not be helpful, however, if it were to change or if the state for which you were searching did not exist. So, the safe way to code this is to check the array for an item, and, if that item is found, then Swift will give us its current index position. The method `index(of:)` is what we will use to get the index position of `South Carolina`:

```
if let index = arrStates.index(of:"South Carolina") {
  print("Current index position is \(index)")
}
```



Now that we have the position, we can update `South Carolina` to `Arizona` like so:

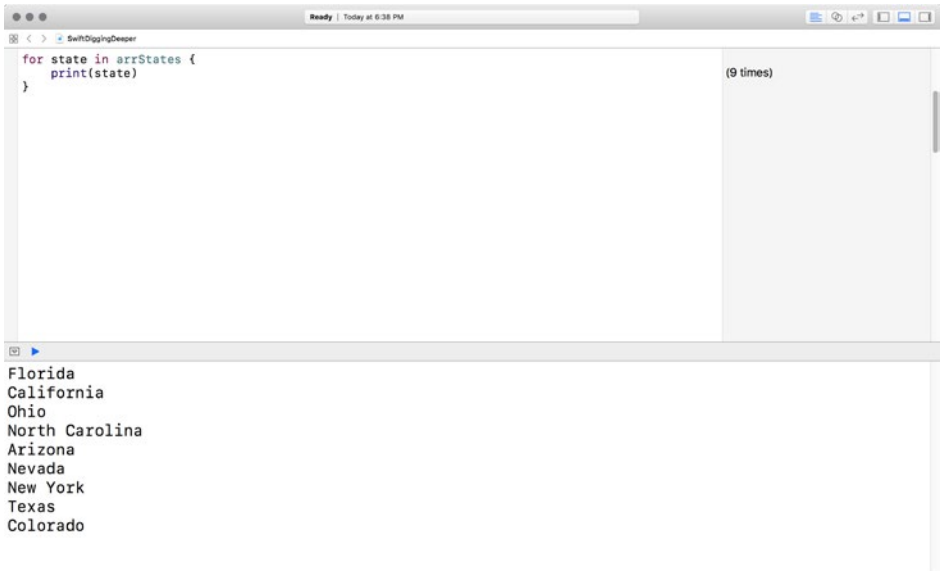
```
if let index = arrStates.index(of:"South Carolina") {
  arrStates[index] = "Arizona"
}
```



Iterating over an array

It would be nice if we could see a list of the states in our array. Earlier, you learned that `for...in` loops work with sequences. Since our array is a sequence, we can use `for...in` loops to loop through each element. When working on a project that has arrays, it is helpful to use a print statement inside of a `for...in` loop. This lets us print every item in our array to the Debug panel. So, let's use a `for...in` loop to look at the contents of our array:

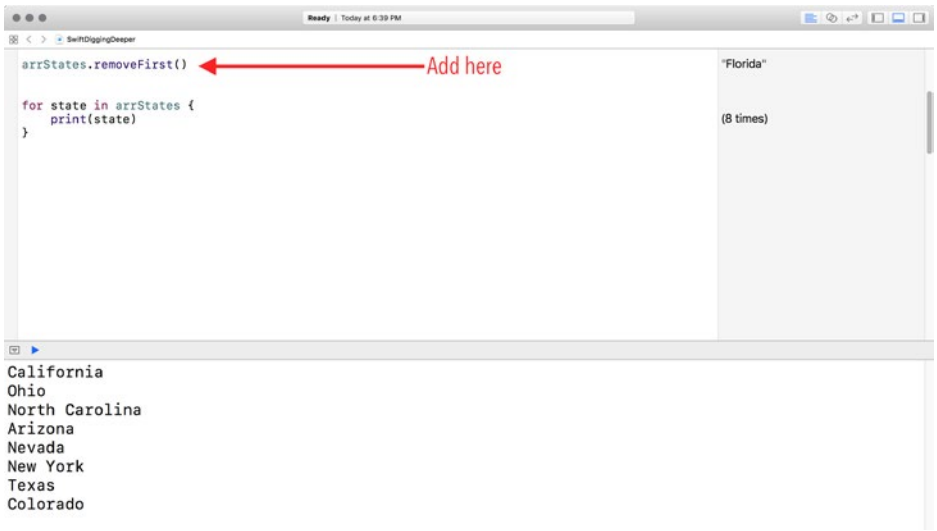
```
for state in arrStates {
  print(state)
}
```



Removing items from an array

Now, it is time to start deleting items from our array. Let's delete the first item from our list. We have a convenience method for removing items from an array called, `removeFirst()`. This method will remove the first item from our array, which in our case is `Florida`. Let's remove `Florida` and add this line above our `for...in` loop:

```
arrStates.removeFirst()
```



Since we removed `Florida`, all of our states' index positions will be updated to move one position closer to the top of the array. But, what if we wanted to remove an item that was not first? In order to do this, we can use the `remove(at:)` convenience. So, let's remove `North Carolina` and `New York`, which are sitting at positions `2` and `4`, respectively. We will add the following above our `for...in` loop:

```
arrStates.remove(at:2)
arrStates.remove(at:4)
```



Now, both `North Carolina` and `New York` are removed. You will see that `California` and `Ohio` did not move, but `Colorado` and `Nevada` moved up closer to the top of the list. To remove the remaining six items, we could use `remove(at:)` for each one, but instead we will use the simpler method of `removeAll()`. So, let's use `removeAll()` in Playgrounds:

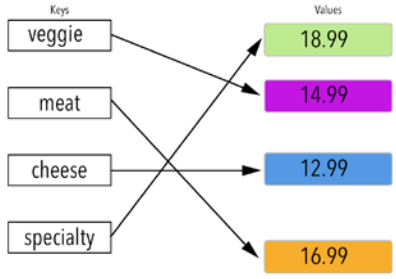
```
arrStates.removeAll()
```



Now, we are back to where we started with an empty array. We have only scratched the surface for arrays. We will do more with arrays later in this book, but we first need to look at the next collection type, Dictionaries.

Dictionaries

A dictionary is an unordered collection of values with each one accessed through a unique key. Let's look at the following diagram:



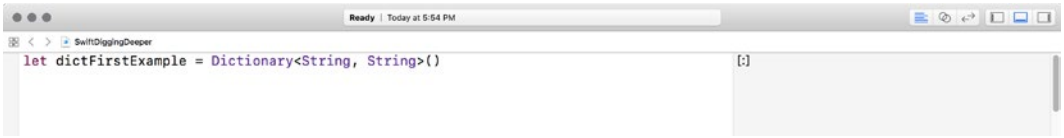
In our diagram, we have a dictionary of pizzas (**Keys**) with their prices (**Values**). In order to find something inside of a dictionary, we must look it up by its key. Let's look at a dictionary syntax:

```
Dictionary<Key, Value>
```

Creating a dictionary

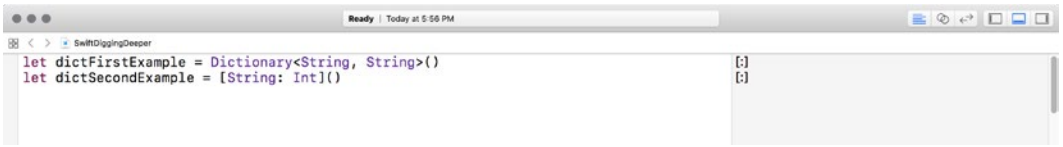
The traditional way of creating a dictionary is to first declare it as a dictionary, and then, inside angle brackets, declare a type for the key and value. Let's create our first dictionary inside Playgrounds:

```
let dictFirstExample = Dictionary<String, String>()
```



The immutable dictionary we just created above has a data type of String for both its key and value. We have multiple ways to create a dictionary. Let's look at another by adding the following into Playgrounds:

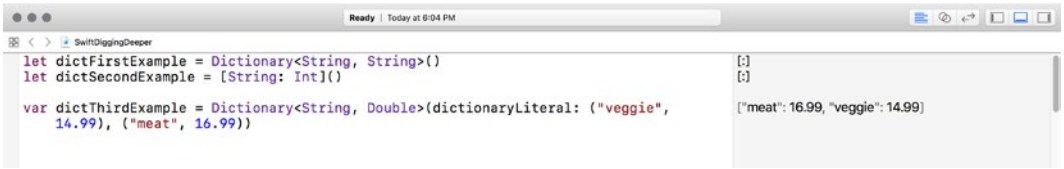
```
let dictSecondExample = [String: Int]()
```



In this latest example, we created another immutable dictionary with its key having a data type of String and its value having a data type of Int.

If we wanted to use our pizza diagram, the key would have a data type of a String and the value would have a data type of a Double. Let's create this dictionary in Playgrounds, but, this time, we will make it a mutable dictionary and give it an initial value:

```
var dictThirdExample = Dictionary<String, Double>(dictionaryLiteral:
("veggie", 14.99), ("meat", 16.99))
```



The above is just one way of creating a dictionary for our pizza diagram example. Let's look at a much more common way using type inference:

```
var dictPizzas = ["veggie": 14.99]
```



This is a much simpler way to create a dictionary with an initial value. When initializing a dictionary, it can have any number of items. In our case, we are starting off with just one.

Now, let's look at how we can add more pizzas into our dictionary.

Adding and updating dictionary elements

Let's add another item to our dictPizzas dictionary:

```
dictPizzas["meat"] = 17.99
```



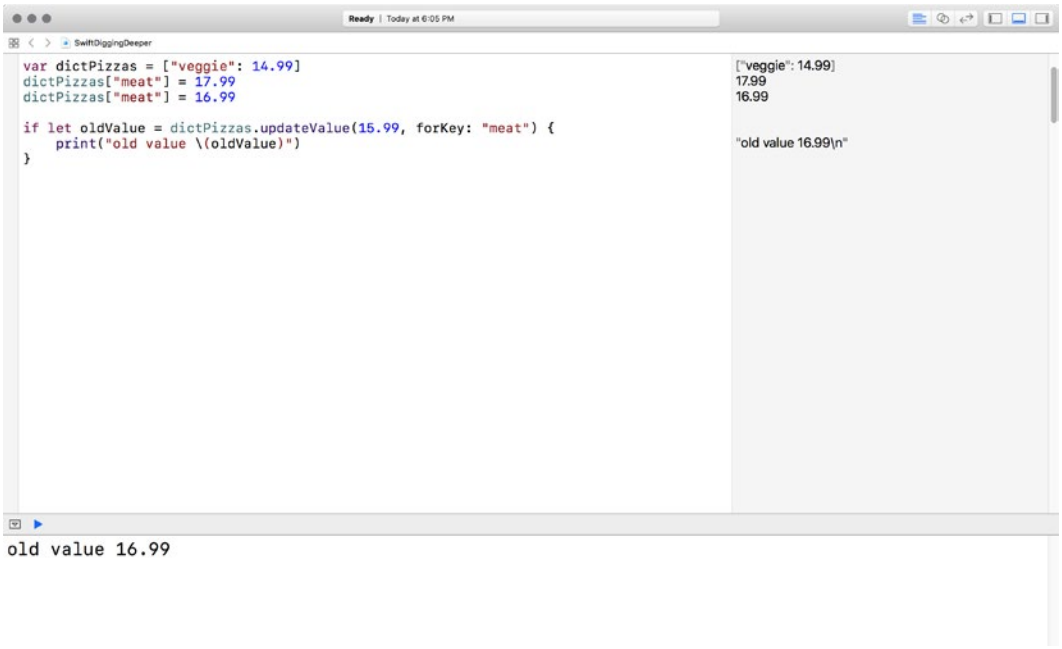
This is the shorthand method for adding an item to a dictionary. After the dictionary variable, we add the key inside the brackets. Since the key for this dictionary is Strings, we must put this key in quotes. Next, we assign a Double to our value. Now, our dictionary has two items. This syntax is also used to update a dictionary item. Let's change the price of meat pizza to `16.99`:

```
dictPizzas["meat"] = 16.99
```



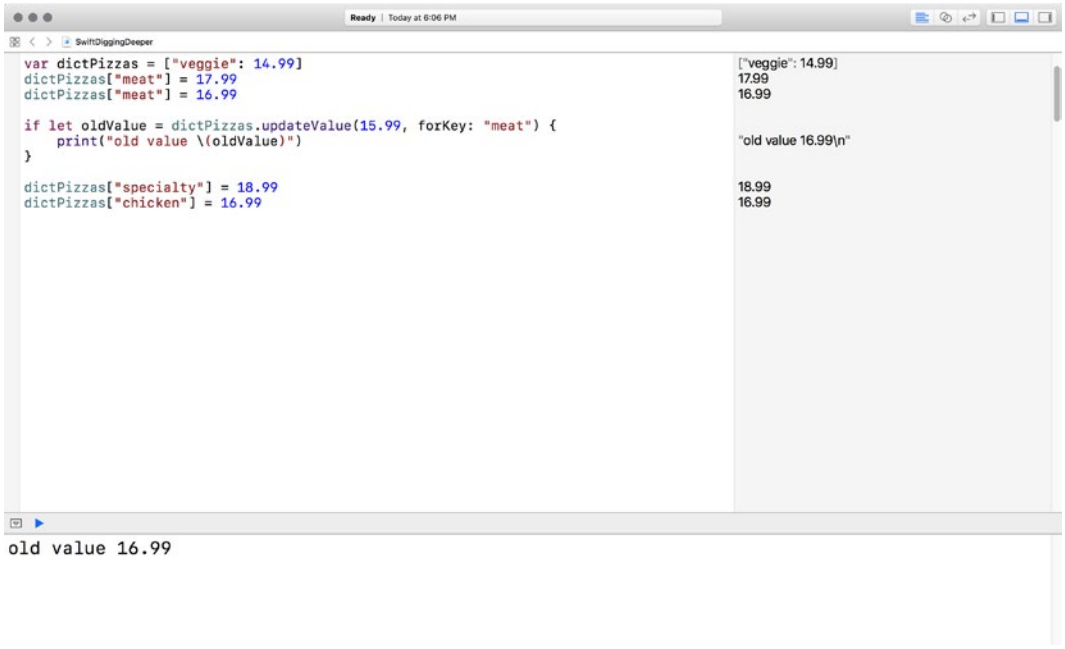
Instead of using the shorthand syntax, you can use the `updateValue(_:forKey:)` method instead. This method does almost the same thing as the shorthand syntax. If the value does not exist, it creates the item; and, if it does exist, it will update the item. The only difference is that when using the `updateValue(_:forKey:)`, it actually returns the old value after performing the update. Using this method, you will get an optional value, because it is possible that no value exists in the dictionary. Let's change the value now from `16.99` to `15.99`:

```
if let oldValue = dictPizzas.updateValue(15.99, forKey: "meat") {
  print("old value \(oldValue)")
}
```



Since we do not need the old value, we will just use the shorthand syntax to add a couple more pizzas:

```
dictPizzas["specialty"] = 18.99
dictPizzas["chicken"] = 16.99
```

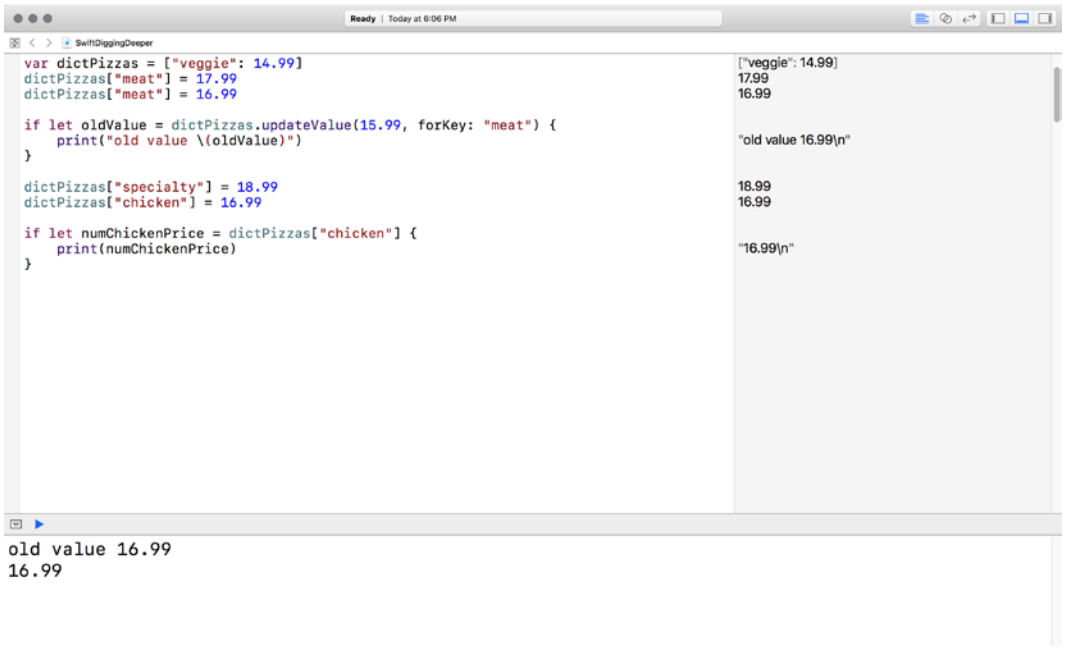


Now that we have some data inside our dictionary, let's see how we can access that data.

Accessing an item in a dictionary

When trying to access an item inside a dictionary, you will always receive an optional value. The reason for this is that you could potentially receive a nil value if the value does not exist. So, you should always use an if-let statement in order to safeguard your code:

```
if let numChickenPrice = dictPizzas["chicken"] {
  print(numChickenPrice)
}
```

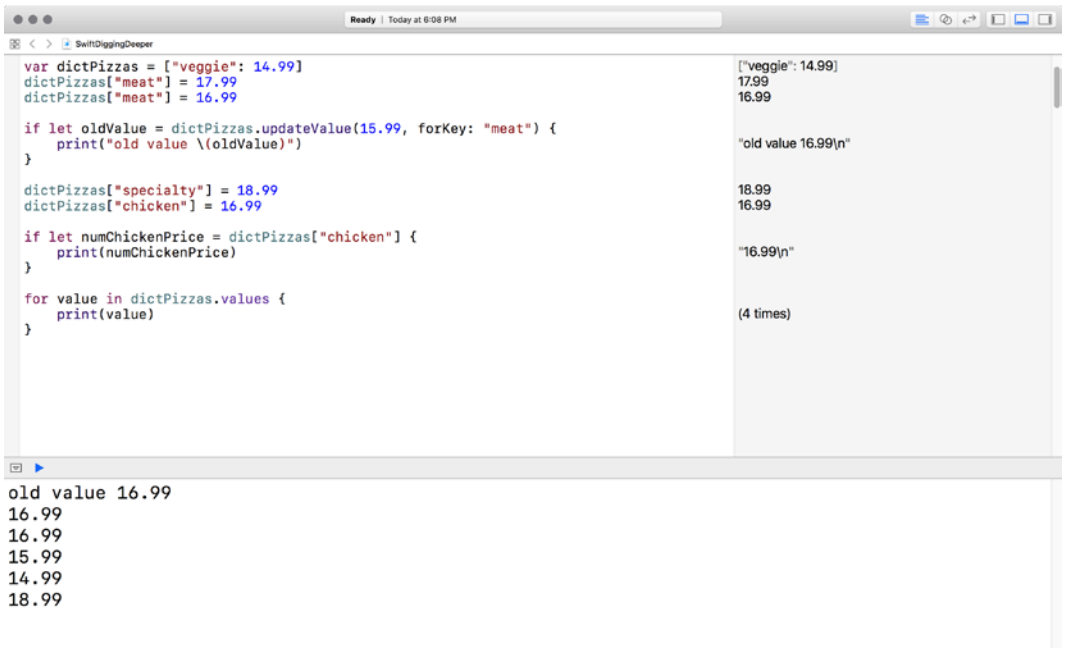


Iterating over dictionary values

Just like an array, we can iterate through our dictionary; however, there are a few differences. Since a dictionary is unordered, each time you loop through, the values will never be in the same order. With dictionaries, you can loop through both the values and keys.

Let's iterate over a dictionary's values using a for-in loop. Add the following into Playgrounds:

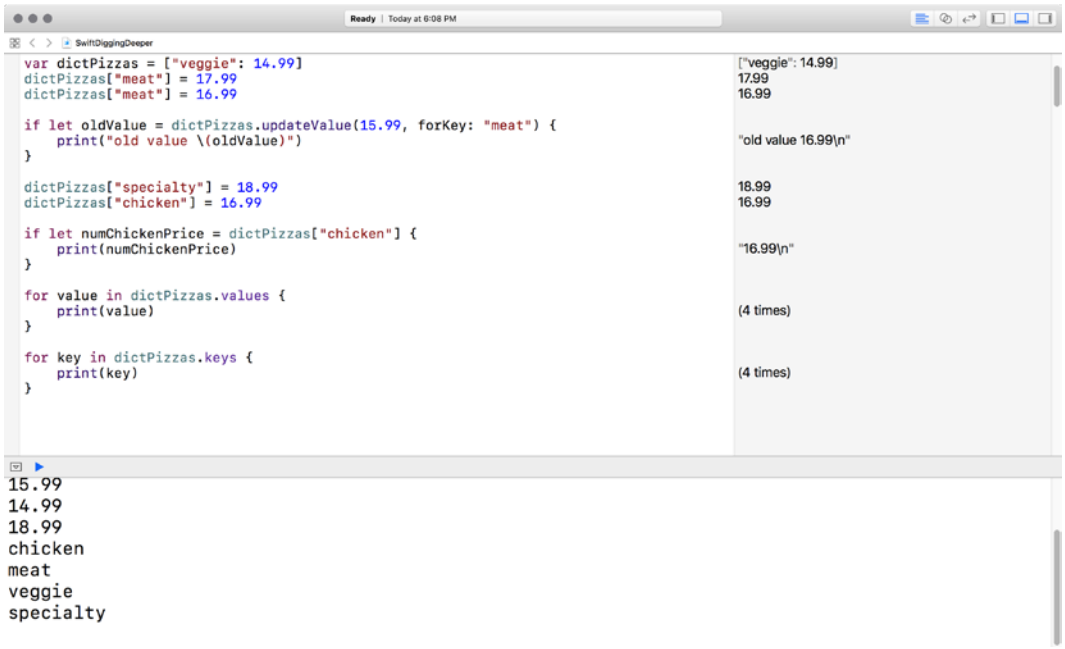
```
for value in dictPizzas.values {
  print(value)
}
```



Iterating over dictionary keys

To iterate over a dictionary's keys using a `for...in` loop, add the following into Playgrounds:

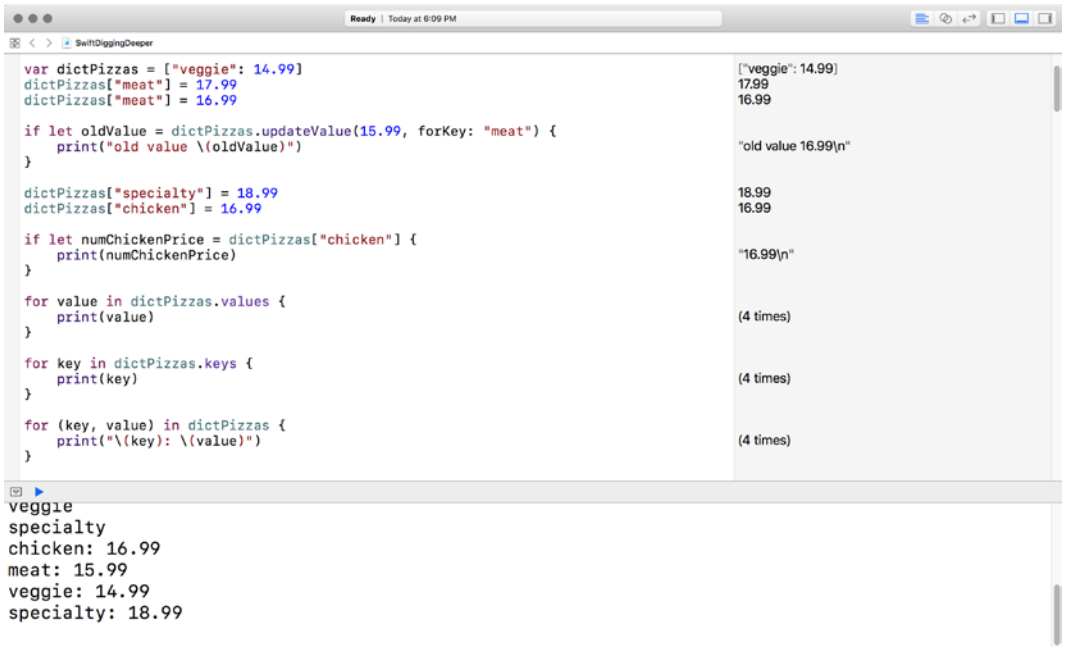
```
for key in dictPizzas.keys {
  print(key)
}
```



Iterating over dictionary keys and values

When you need to iterate over both dictionary keys and values using a for-in loop, you use the following:

```
for (key, value) in dictPizzas {
  print("\(key): \(value)")
}
```



So, we now looked at how to loop through a dictionary.

Checking the number of items in a dictionary

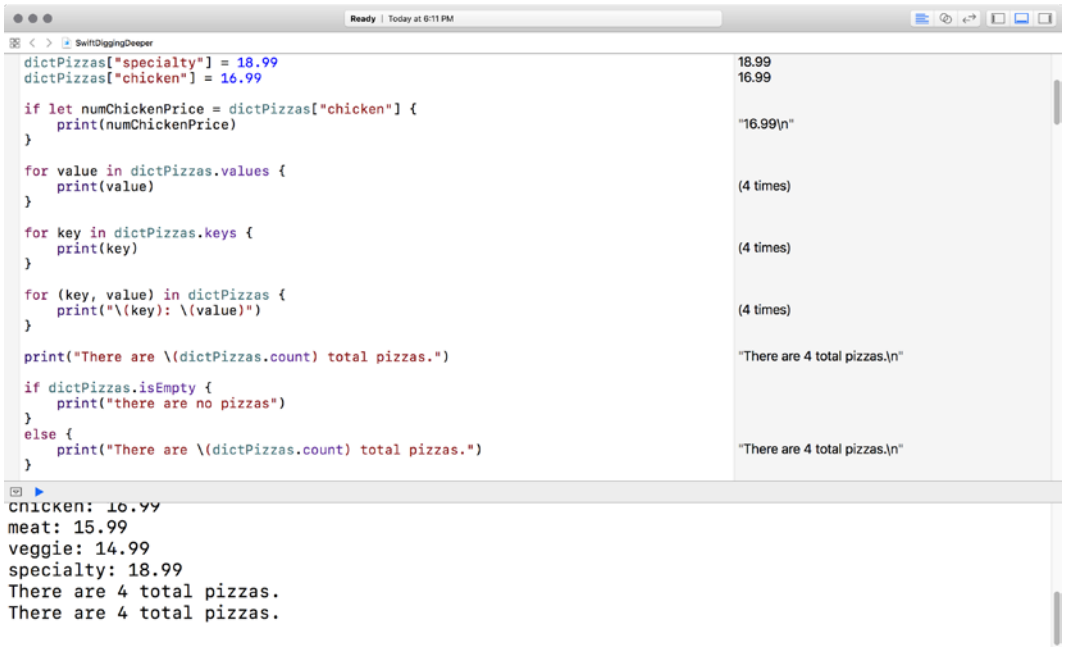
In addition to keys and values, we have other useful properties. We can see the number of items in a dictionary using the count property. Let's try that by adding the following:

```
print("There are \(dictPizzas.count) total pizzas.")
```



Along with count, we can check whether a dictionary `isEmpty` or not. Let's use this in an `if...else` statement by adding the following:

```
if dictPizzas.isEmpty {
  print("there are no pizzas")
}
else {
  print("There are \(dictPizzas.count) total pizzas.")
}
```

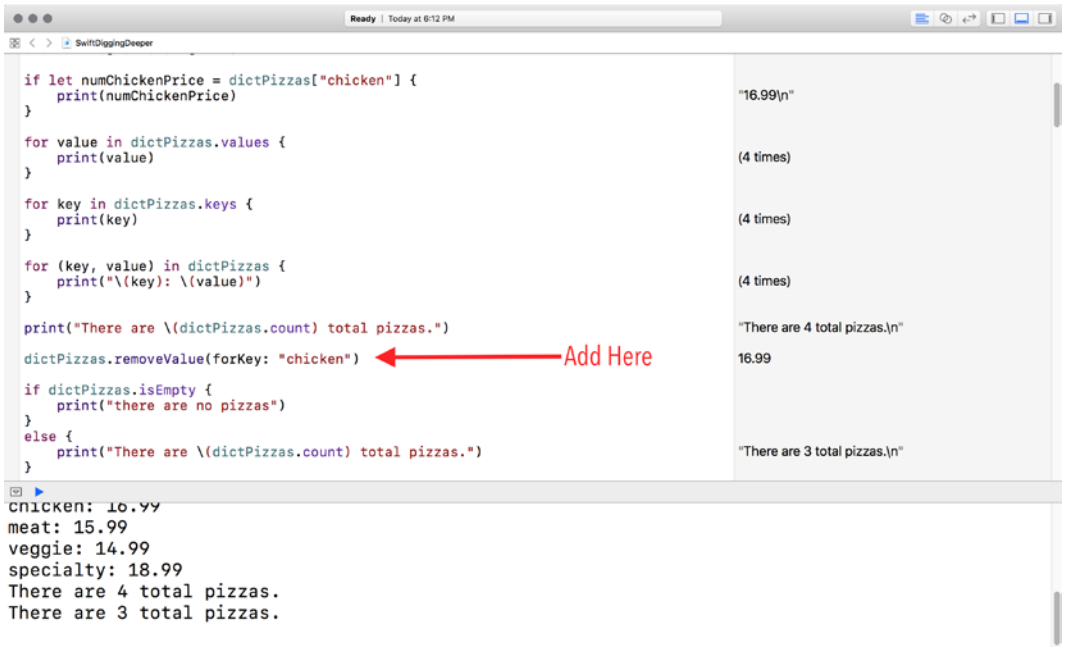


This kind of logic is helpful when you want to display something back to the user or hide some UI.

Removing Items from a dictionary

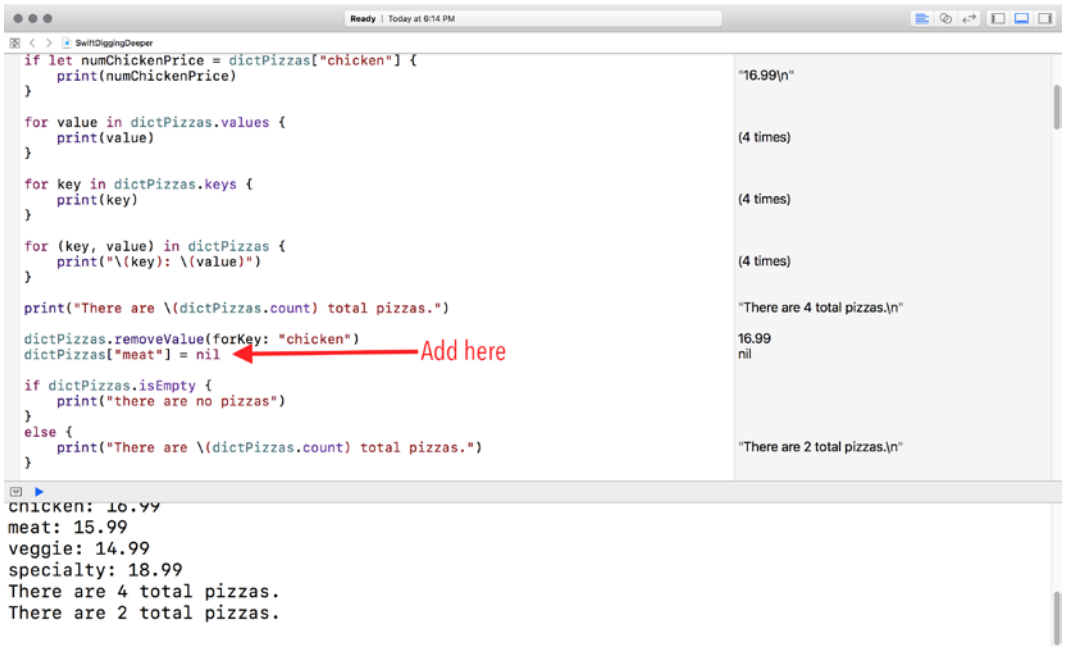
Next, let's learn how to remove an item from a dictionary. When deleting items from a dictionary, we have two primary ways of doing this. The first uses `removeValue(forKey:)`. Let's add this right above our `if...else` statement that checks if the dictionary `isEmpty`:

```
dictPizzas.removeValue(forKey: "chicken")
```



Let's look at the second way of removing dictionary items, the shorthand syntax. Add the following to Playgrounds following the `removeValue(forKey:)`:

```
dictPizzas["meat"] = nil
```

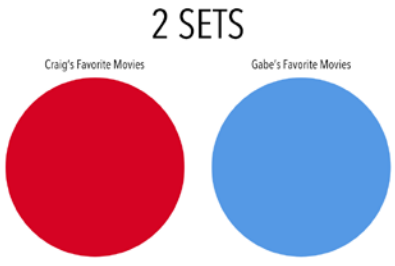


Notice that, just like with `updateValue(_:forKey:)`, `removeValue(forKey:)` will return you the value before it is removed. If you do not need the value, the shorthand syntax is the preferred method.

So far, we covered arrays and dictionaries, and now we will review one last collection, sets.

Sets

A set stores unique values of the same type in a collection without a defined order. Let's look at a diagram:

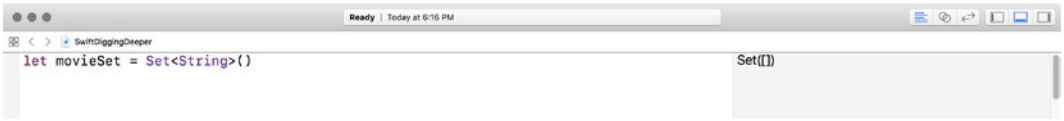


In the above diagram, we have two circles, both of which represent a set. On the left, we have Craig's favorite movies; and, on the right, we have Gabe's favorite movies.

Creating an empty set

Before we create these sets, let's just create an empty set and see what that looks like:

```
let movieSet = Set<String>()
```

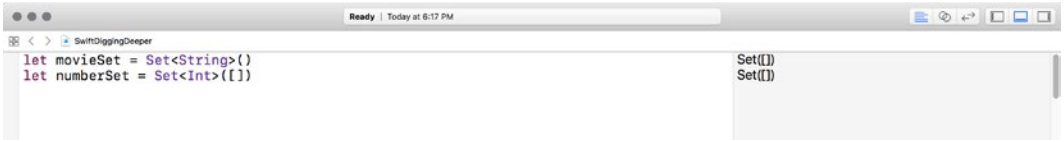


In this first set, after the equals sign, we create the set and give it a data type of String. Then, we use the parentheses to initialize the set.

Creating a set with an array literal

Our first set was an empty String set, but we can create a set using an array literal. Let's add the following into Playgrounds:

```
let numberSet = Set<Int>([])
```

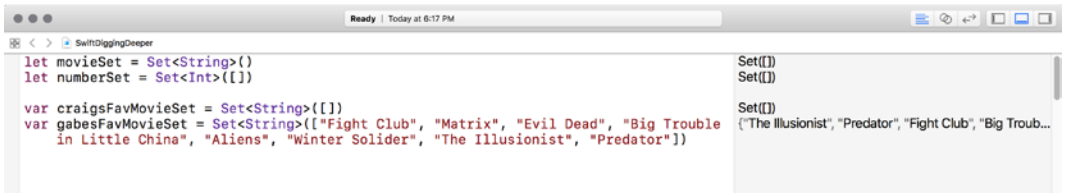


This above immutable set has a data type of Int; but, in the parentheses, we pass an empty array literal when we used the brackets.

Creating a mutable set

Now that we are familiar with the way sets are created, let's create a mutable set for Craig's favorite movies and one for Gabe's favorite movies. Add the following into Playgrounds:

```
var craigsFavMovieSet = Set<String>([])
var gabesFavMovieSet = Set<String>(["Fight Club", "Matrix", "Evil
Dead", "Big Trouble in Little China", "Aliens", "Winter Solider", "The
Illusionist", "Predator"])
```

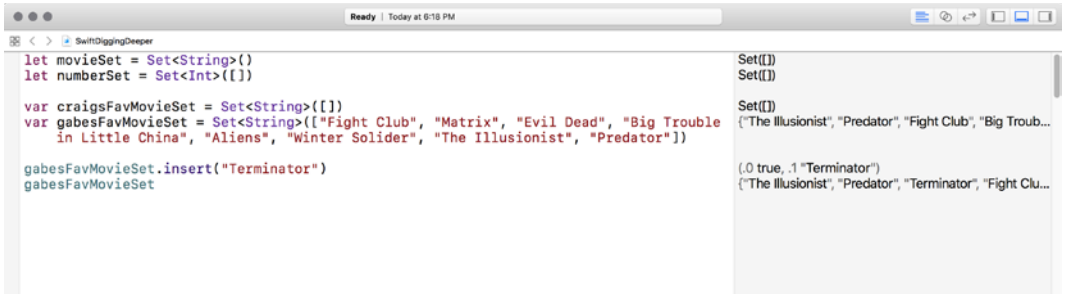


We now have two mutable sets. The first set is created with an empty array literal, and the second set is created with some initial values. Let's add some more items to both sets.

Adding items into a set

In order to add an item into a set, we have to use the `insert()` method. Let's use that to add another movie into Gabe's favorite movies:

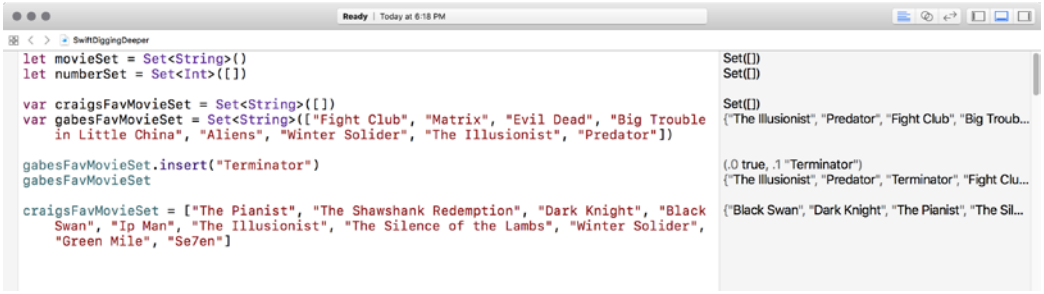
```
gabesFavMovieSet.insert("Terminator")
gabesFavMovieSet
```



Now, Gabe has nine movies, and Craig still has none. We added the `gabeFaveMovieSet` variable again so that we can see the contents update in the Results panel. In order to add multiple items into a set, we can use an array literal.

Let's add `10` movies into Craig's list as follows:

```
craigsFavMovieSet = ["The Pianist", "The Shawshank Redemption", "Dark
Knight", "Black Swan", "Ip Man", "The Illusionist", "The Silence of
the Lambs", "Winter Solider", "Green Mile", "Se7en"]
```

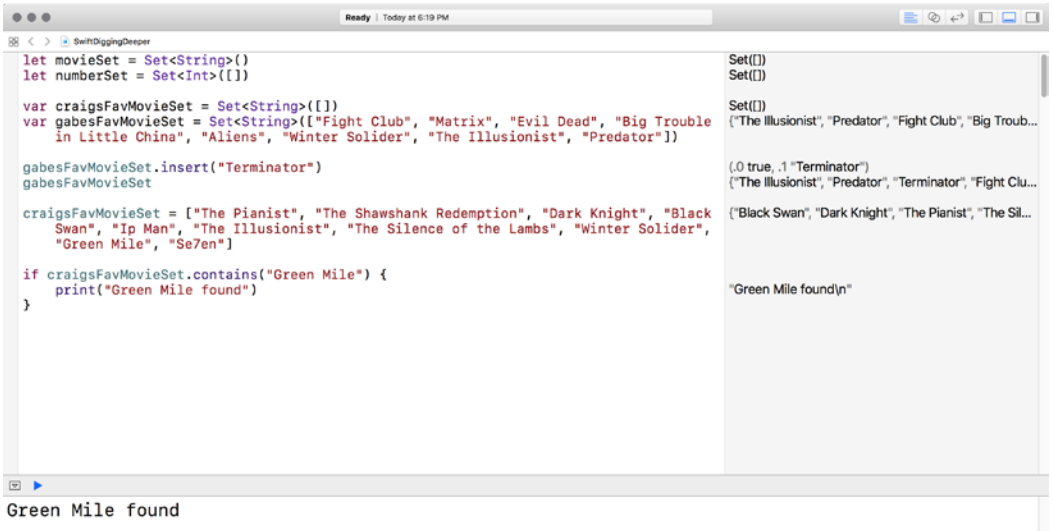


Craig's set now has `10` movies. Next, let's see how we can work with sets.

Checking if a set contains an item

The first thing we can do with sets is to check if a set contains an item. Let's see if Craig's movie list has the movie Green Mile:

```
if craigsFavMovieSet.contains("Green Mile") {
  print("Green Mile found")
}
```

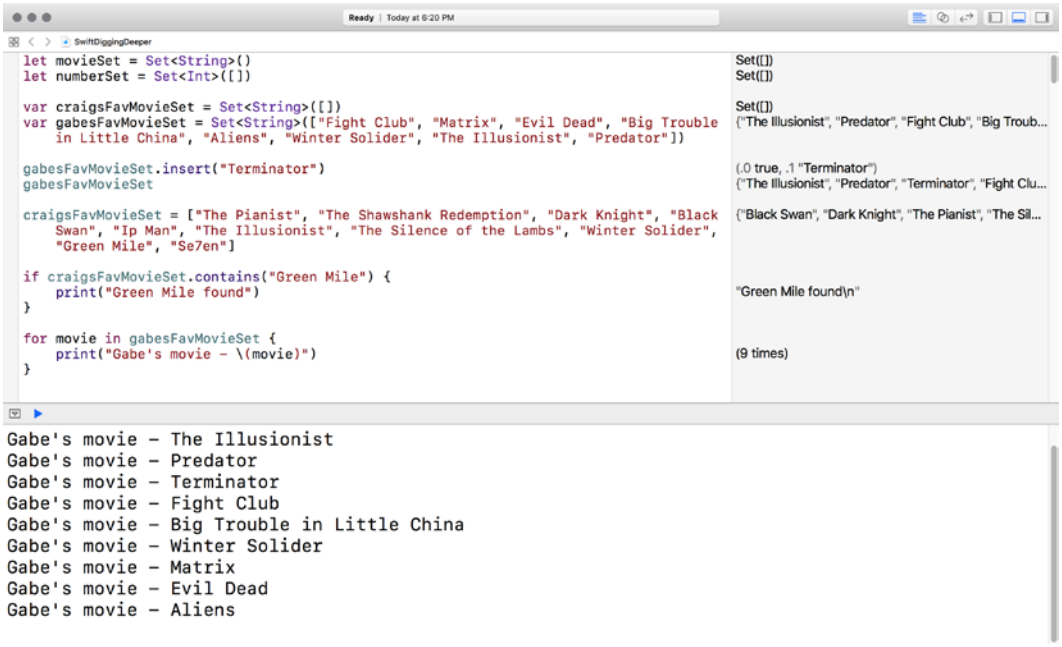


In this preceding example, we used the `contains()` method in order to discover whether an item is in the set.

Iterating over a set

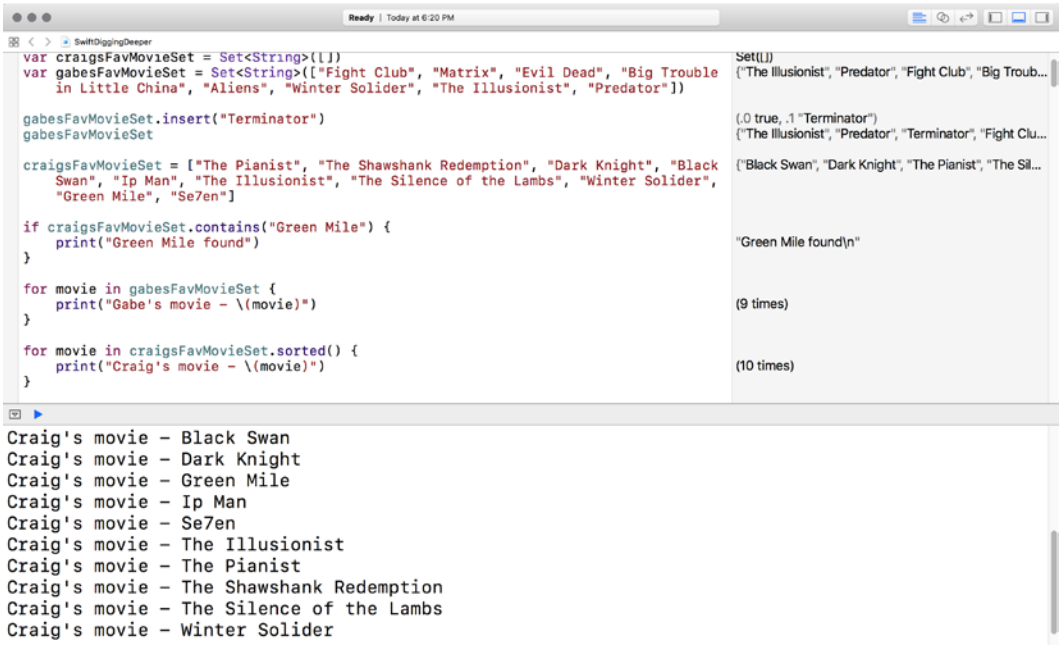
If we wanted a list of all the movies in Gabe's list, we can use a `for...loop`. Let's see how that works:

```
for movie in gabesFavMovieSet {
  print("Gabe's movie - \(movie)")
}
```



Now that we have seen a for-in loop for all three collections, arrays, dictionaries, and sets, you can see that there are a lot of similarities. Remember, since sets come unordered, every time we run our `for...in` loop we will get a list in a different order. The way around this is to use the `sorted()` method. This will ensure that, every time we loop through our list, it will always be in the same order. Let's do that on Craig's movie list:

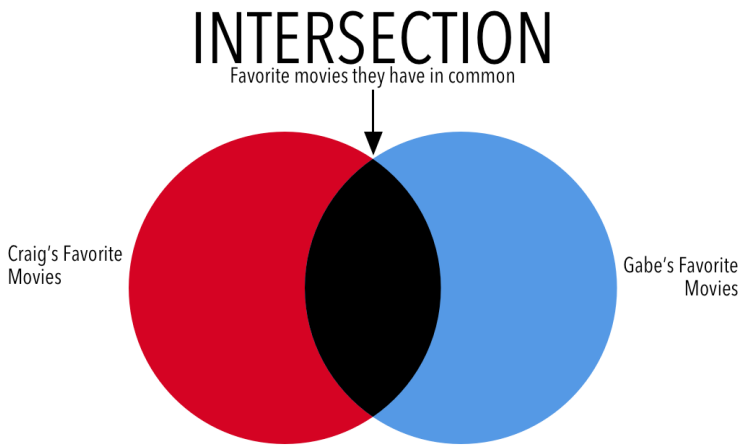
```
for movie in craigsFavMovieSet.sorted() {
  print("Craig's movie - \(movie)")
}
```



Now that we have our set sorted, let's look at the real power of using sets.

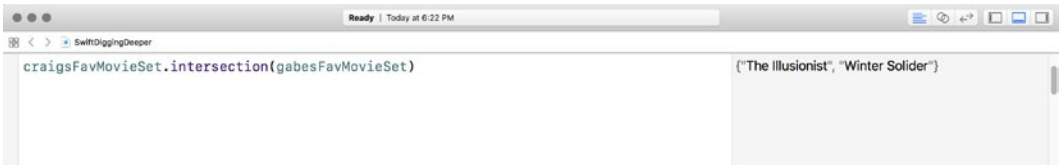
Intersecting two sets

In the following diagram, we see that, if we intersect both sets together, we should get a list of any movies they have in common:



We can do the same using the `intersection()` method in our code. Let's intersect both movie lists and see what happens:

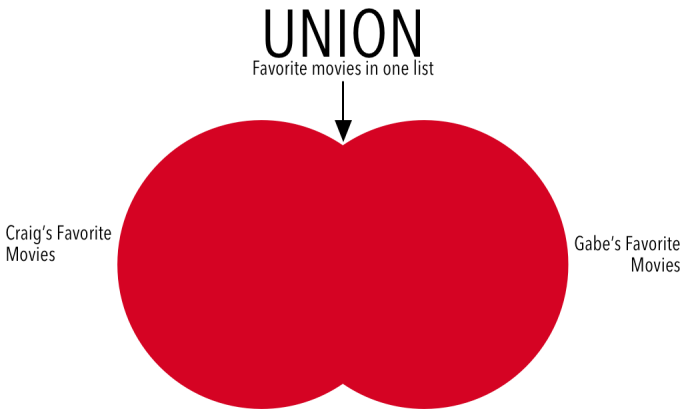
```
craigsFavMovieSet.intersection(gabesFavMovieSet)
```



We can see that the only two movies these sets have in common are `Winter Solider` and `The Illusionist`. In addition to seeing which movies the two sets have in common, we also can join the lists to get one consolidated list of the movies from both sets.

Joining two sets

If you look at the following diagram, you can see the two sets joined together:



Using the `union()` method, we get a consolidated list of items with no duplicates. Let's try this in Playgrounds:

```
craigsFavMovieSet.union(gabesFavMovieSet)
```

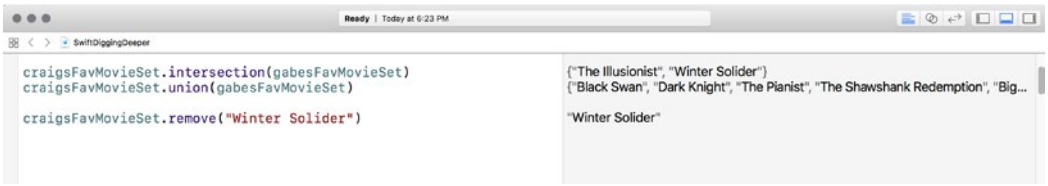


We have a combined list of movies that includes all the movies that the two sets did not have in common and the two movies that were in common, but only listed once. As you can see, sets are really powerful, and you can use them to manipulate data. Finally, we need to look at how you can remove items from a set.

Removing items from a Set

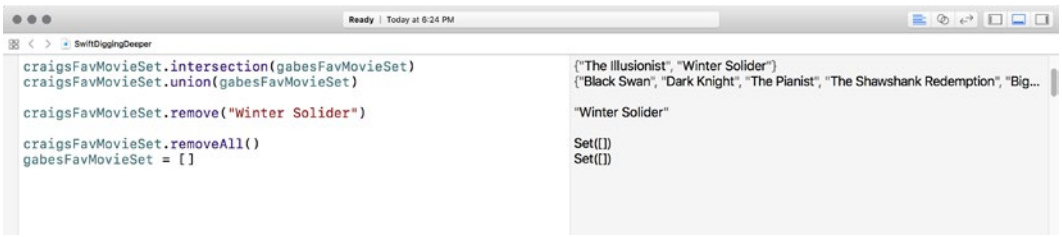
In order to remove an item from a set, we can use the `remove()` method. When we use this method, we just input the item we want to remove into the parentheses. Let's remove `Winter Solider` from Craig's movie list:

```
craigsFavMovieSet.remove("Winter Solider")
```



If you wanted to remove more than a single item from a set (for instance, all of the items), then you can use the `removeAll()` method or give it an empty array literal:

```
craigsFavMovieSet.removeAll()
gabesFavMovieSet = []
```



Now, both sets are empty.

Summary

We covered a lot in this chapter. Even though we will touch on these things throughout the creation of the *Let's Eat* app, it is best to make sure you are comfortable with what we covered here. So, please review as much as you need in order to make sure you feel that you are proficient in the topics contained in this chapter. In the next chapter, we will start building our *Let's Eat* app. Over the next two chapters, we will work on getting our project set up, and then we will start working on the visual aspects of our app.

5
Starting the UI Setup

Now that you have learned Swift, which will help you understand a lot of the boilerplate code you will see later, it is time to start building our *Let's Eat* app. Let's begin by getting an overview of what we are going to build. We will review the finished product and then get into how to build this app. Before we start, there will be a lot of new terms and things with which you may or may not be familiar. Learn as much as you can and do not let the finer details stop you from progressing.

We will cover the following in this chapter:

- Useful terms
- App tour
- Project setup
- Storyboards
- Creating a custom title view

Useful terms

Before we dig in and start getting our UI set up, we need to take a few minutes to introduce (or re-introduce) you to some terms that you should understand while we build our app:

- View Controller
- Table View Controller
- Collection View Controller
- Navigation Controller
- Tab Bar Controller
- Storyboard

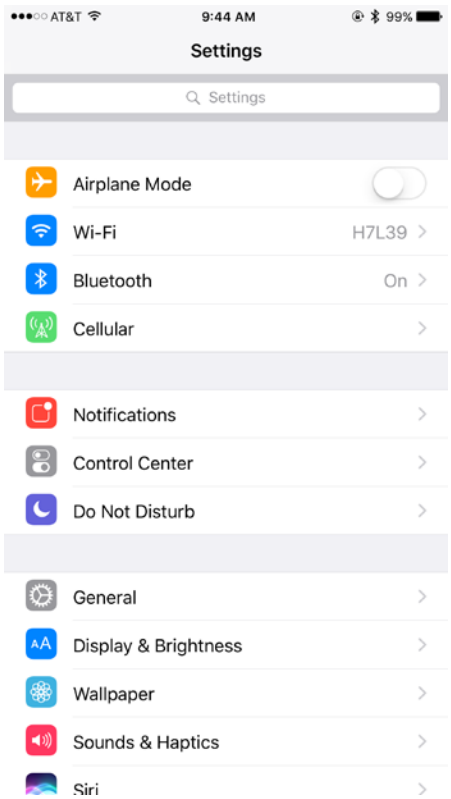
- Segue
- Auto layout
- **Model View Controller** (**MVC**)

View Controller

View Controllers (UIViewControllers) are blank scenes that you can use to hold other UI elements. They give you the ability to create a custom interface.

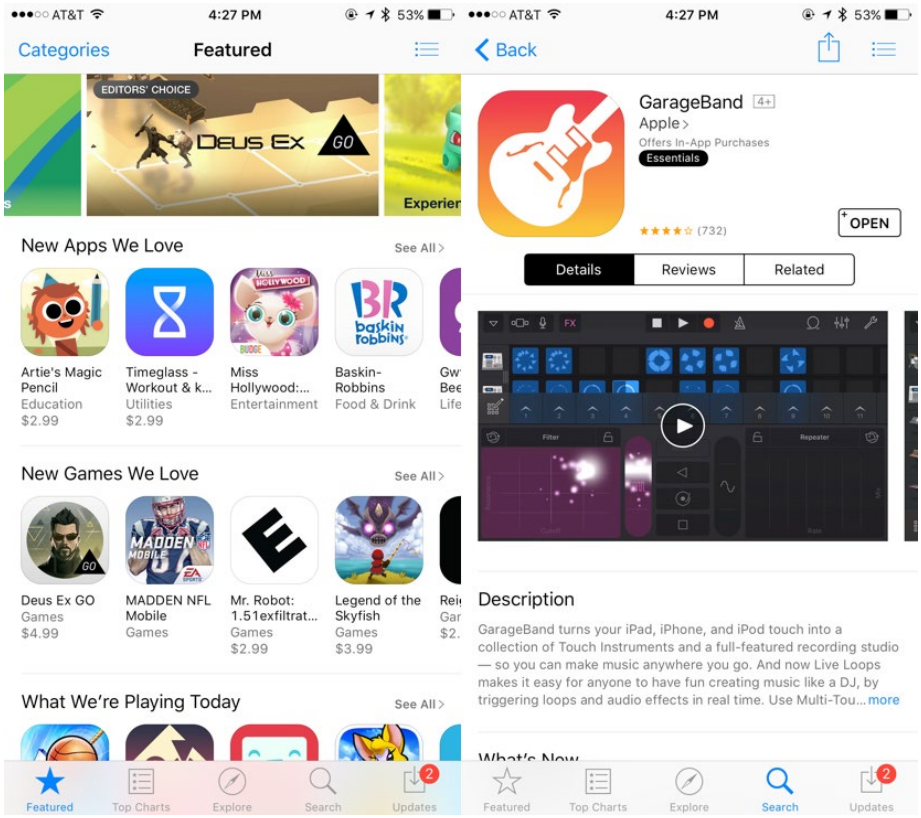
Table View Controller

A Table View Controller (UITableViewController) is one of the most common UI elements and is used to display a list of items. For example, Apple's Settings screen uses Table View Controller to display the list of settings a user can access and change:



Collection View Controller

Collection View Controllers (UICollectionViewControllers) are typically used when you want to display elements within a grid. They are highly customizable and, because of that, are becoming more popular in non-grid based layouts. The App Store, for example, currently uses CollectionViewControllers for both its featured page and app detail page:



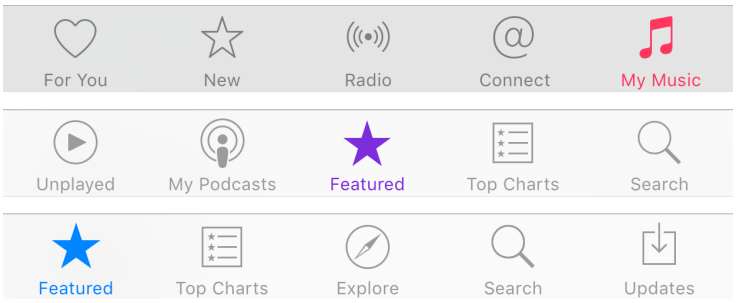
Navigation Controller

A Navigation Controller (UINavigationController) is a UI element that allows you to build a drill-down interface for hierarchical content. When you embed a Navigation Controller into a View Controller, Table View Controller or Collection View Controller, it will manage navigating from one controller to another controller.

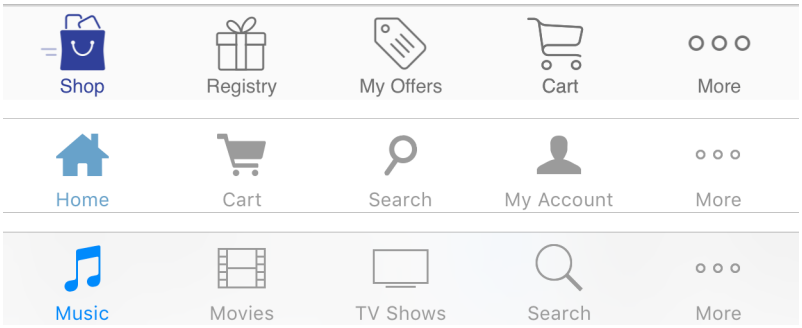
Tab Bar Controller

The Tab Bar Controller (UITabBarController) manages an array of View Controllers. Our *Let's Eat* app will use a Tab Bar Controller. This controller will give us the ability to have navigation for our app with very little setup.

Apple has a few apps with which you might be familiar that use the Tab Bar Controller:

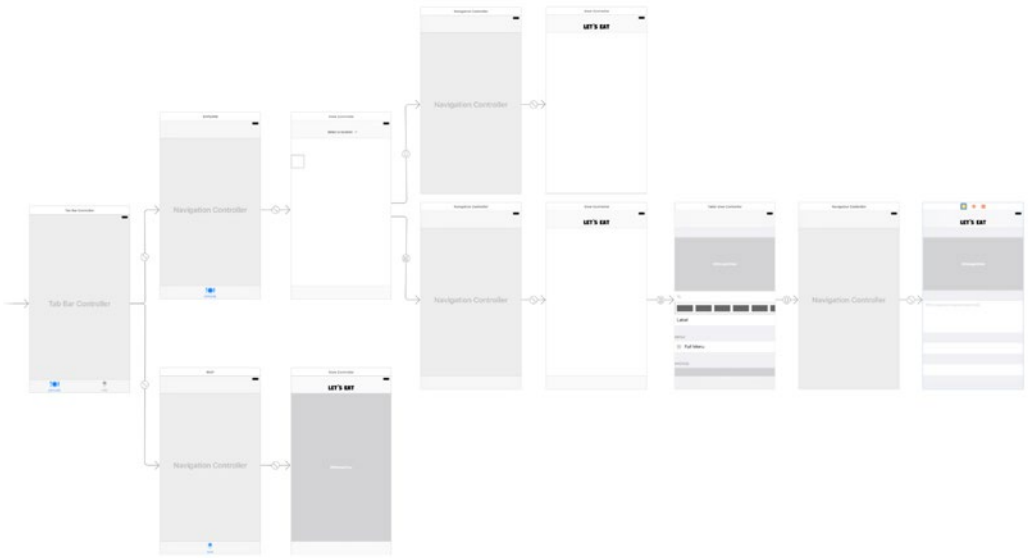


A TabBarController can only have five tabs. If your TabBarController has more than five tabs, the fifth tab and any thereafter will be reduced underneath a **More** button:



Storyboard

A `storyboard` is a file that is used as a visual representation of your app's UI. This is what a storyboard looks like for an app:



Storyboards let you create your entire app visually using View Controllers, Table View Controllers, and Collection View Controllers as scenes. Along with creating your app visually, you will be able to connect scenes together and set up transitions between scenes using segues.

Segue

Segues are used to connect one controller to another controller. In storyboard, segues are represented by an arrow with an icon:



Segues also give you the ability to specify a transition from one scene to another with very little to no programming.

Auto layout

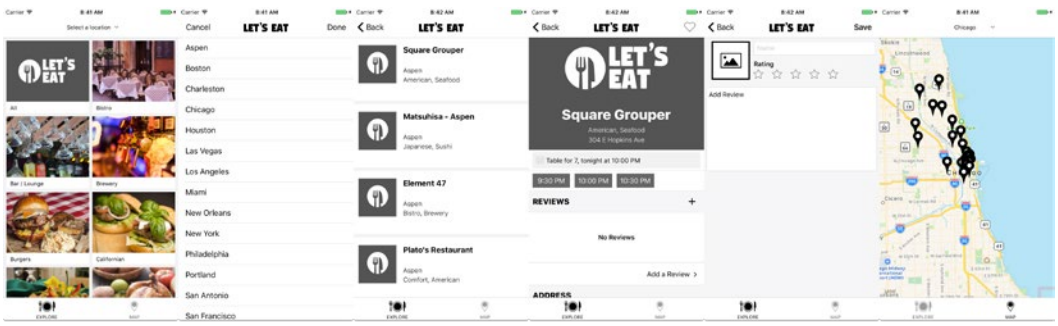
Auto layout is a wonderful tool that allows you to support different screen sizes and device rotation. With auto layout, you can set different constraints on UI elements in order for it to adjust to changes in size and/or rotation. Using auto layout, in your app, allows you to use one storyboard for all devices.

Model View Controller (MVC)

MVC is a common software design pattern, which is a solution for commonly occurring problems within software design. Apple has built iOS apps on the MVC design pattern. This pattern divides our app into three camps known as the Model, View, and Controller. We will cover this in detail later in this book.

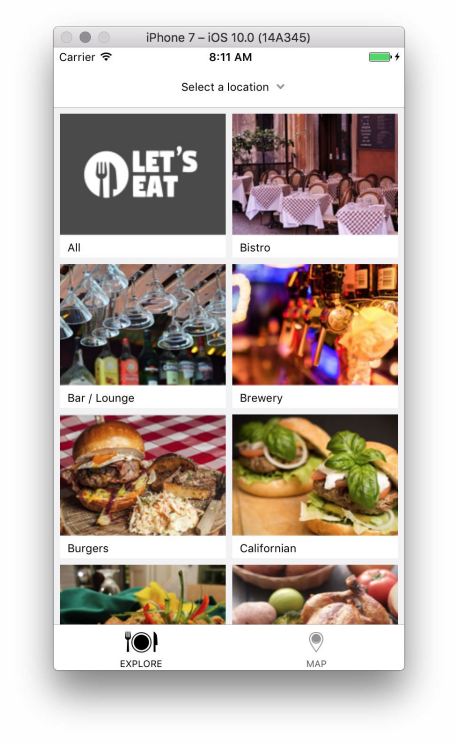
App Tour

The *Let's Eat* app that we will be building is a restaurant reservation app that allows users to find restaurants in a specific area and create reservations from within the app (although our app will not actually book those reservations). I chose a restaurant reservation app for purposes of the lessons in this book, because most of the new iOS 10 features will work really well together in such an app. The app will cover a lot of different aspects from maps to iMessage extensions. Let's take a look at the overall flow of the app so that as we build, you will have a good idea of the direction we will be heading:



Explore tab

When the app launches, you will see the **Explore** tab. This tab will allow users to pick a particular cuisine that they would like as well as select a specific predefined location. Let's breakdown each component in this view:



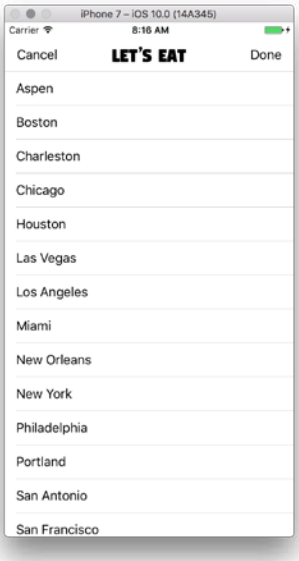
For this screen, we will be working with an empty View Controller, which is where all of our UI components live. As you can see, this view in our app is designed to be a grid, so we will be using a Collection View Controller. We will be setting up this Collection View Controller ourselves.



When I build apps, I typically start with a blank Collection View or Table View, because it gives more flexibility in my code as well as with my user interface.

Locations

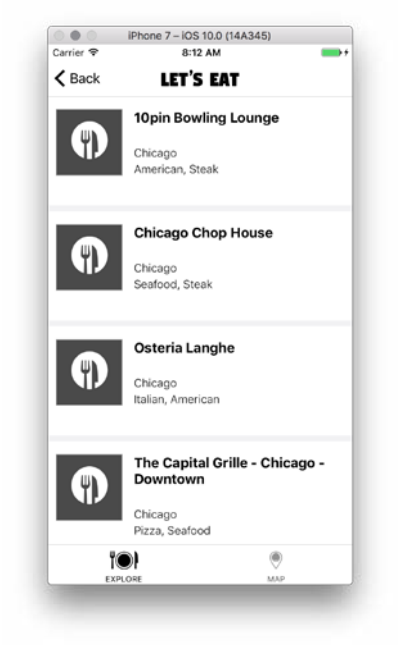
The Locations view will be a list of cities that can be accessed from the Explore tab. We will load this list of cities from a local file—and, if the user selects a city, the app will load all of the restaurants from that area:



For this Locations view, we will be working with a View Controller that uses a Table View Controller.

Restaurant listings

In Restaurant listings, we will see restaurants in the area by the selected cuisine:



This view will display like a list, but we will actually be using a Collection View instead of a Table View.

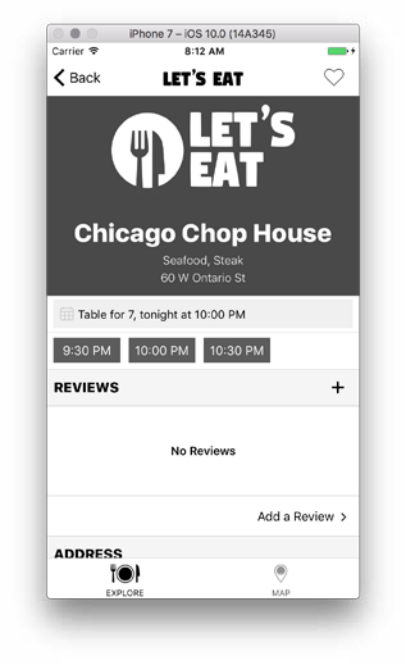


We will be covering both UICollectionViews and UITableViews in this book, but, as an introduction, you should know that UICollectionViews are very powerful. The reason they are powerful is because you can really customize them to look how you want. For example, the App Store detail is a custom UICollectionView.

One great feature when using UICollectionView is that, when you are building a universal app like this one, you can make your view look like a list for the iPhone, but appear as a grid on the iPad with very little effort.

Restaurant detail

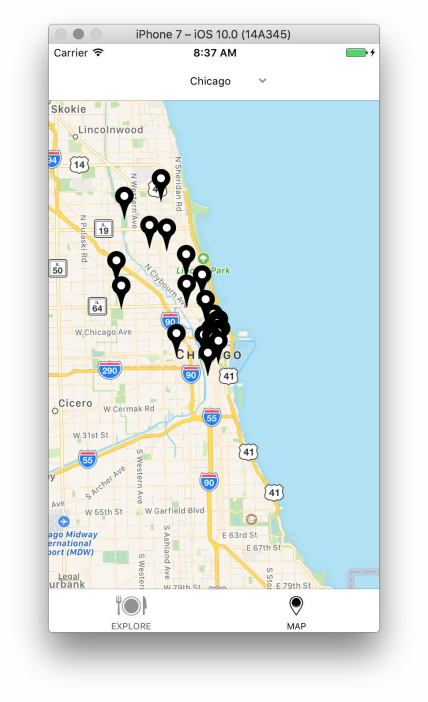
Our Restaurant detail will have more information about the restaurant. This view will be built using a static UITableView:



Using a static UITableView allows us to create a UI using very little code. This screen will use some other UI elements, such as a UISwitch and a UITextField. We will cover more about each of them later in the book.

Map tab

Our Map tab will be a View Controller with a map. Here, you will be able to select from a list of locations. When you select a location, the map will have pins dropped on it, denoting all of the restaurants in the area:



Project setup

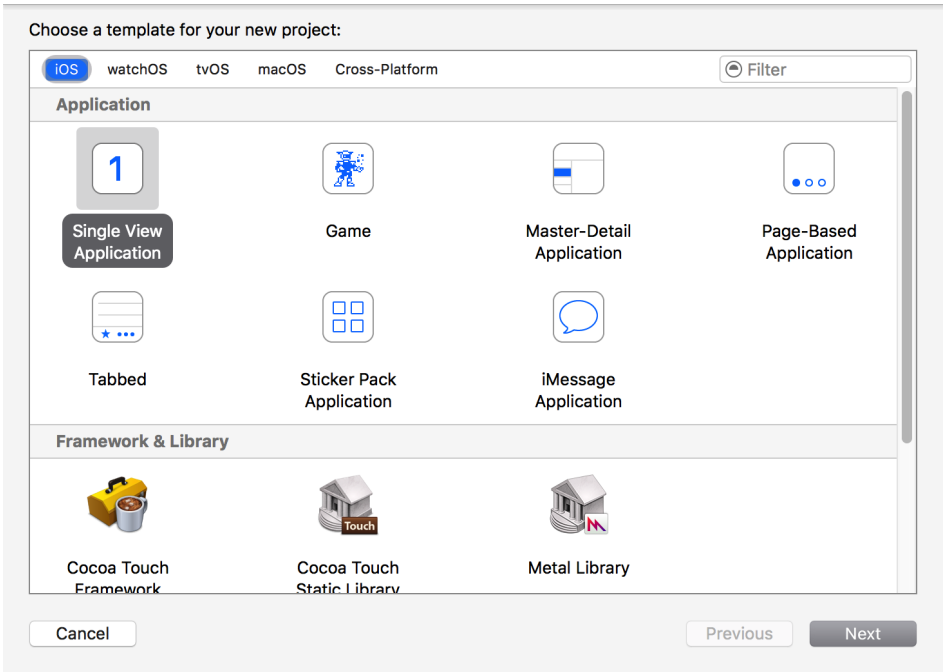
Now that we have gotten a tour of the app, we are going to build the *Let's Eat* app. First, we will set up the app, and then we will create the UI. Lastly, we will design our app in a storyboard.

For the initial setup of the app, we will look at some basics of iOS, starting with creating a new project.

Creating a new project

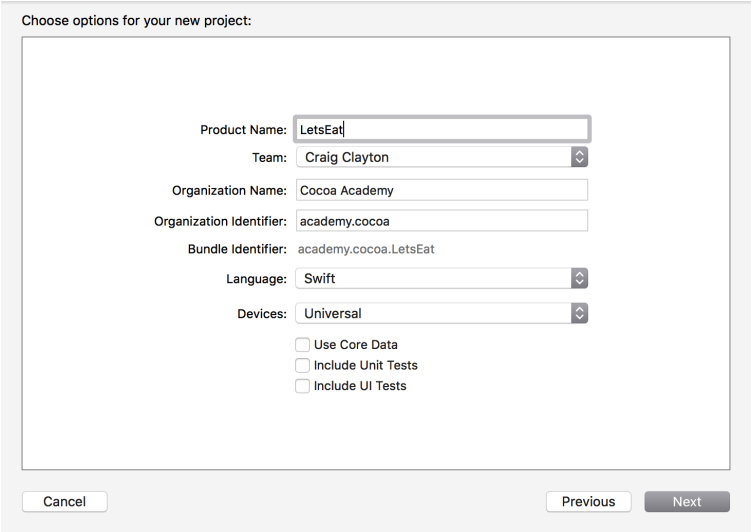
To create a new project:

1. Open Xcode, and the Xcode welcome screen will appear. Click on **Create a new Xcode project** in the left panel of the welcome screen.
2. Select **Single View Application** and click on **Next**:

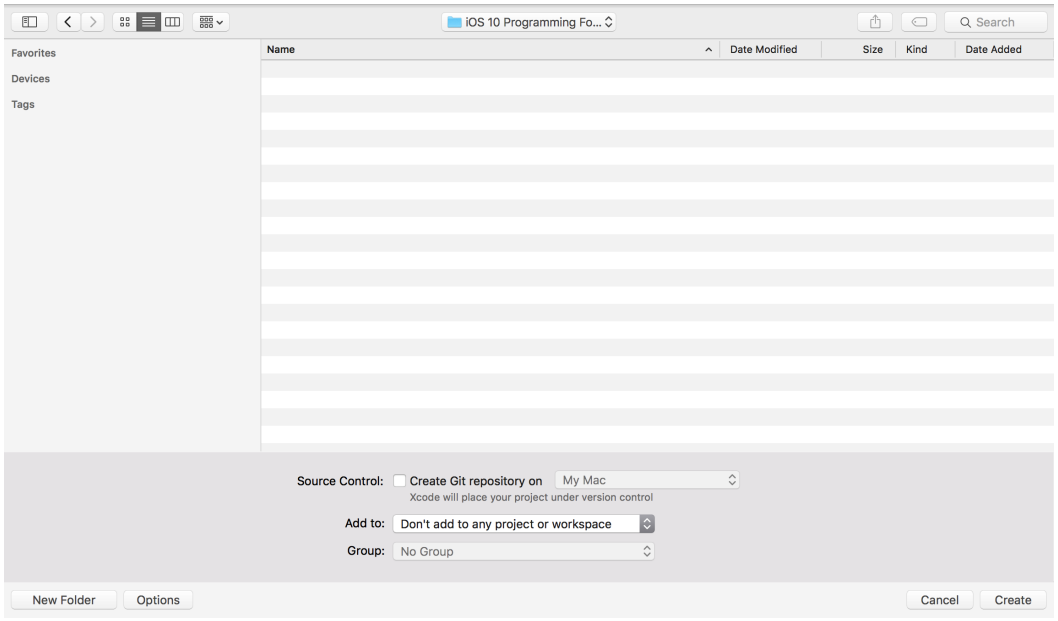


3. In the options screen that will appear, there will be a number of items to complete or choose. Add the following into that options screen and then hit **Next**:
    - **Product Name**: `LetsEat`
    - **Team**: Your account or leave blank
    - **Organization Name**: Your name/company name
    - **Organization Identifier**: Your domain name in reverse order
    - **Language**: `Swift`
    - **Device**: `Universal`
    - **Use Core Data**: Unchecked

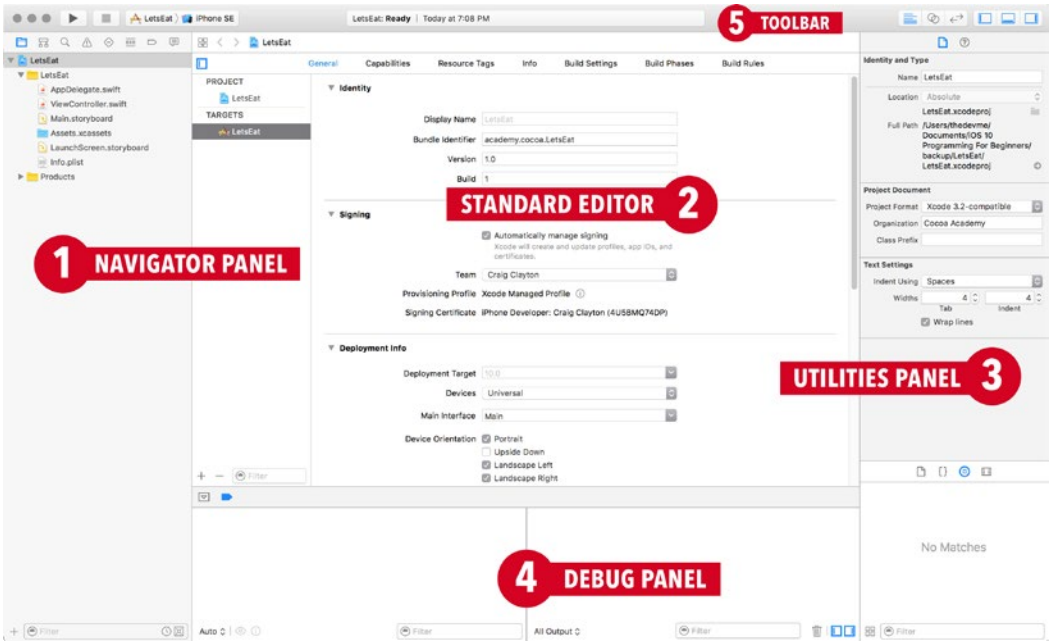
- **Include Unit Tests**: Unchecked
- **Include UI Tests**: Unchecked



4. Choose your desktop or a folder in which to save your project, and then hit **Create**:



5. After you save, you will be presented with the following screen:

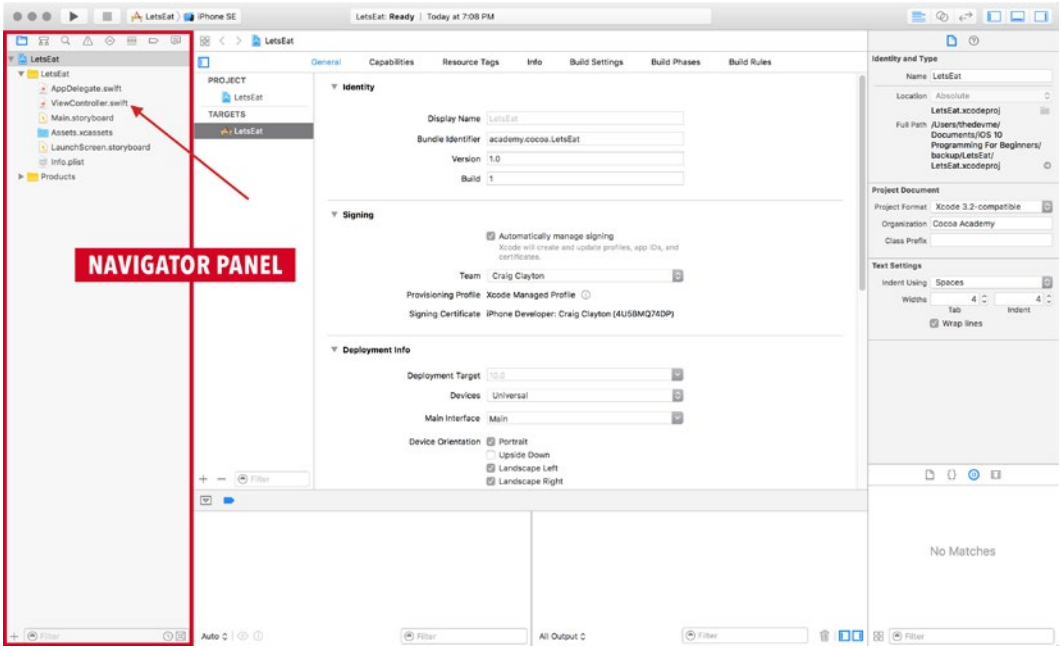


Creating our files

We will be creating all of our files from scratch, so, rather than keeping any existing files in our project, we will delete them and recreate them in proceeding chapters.

In order to delete the `ViewController.swift` file, you will do the following:

1. Select the `ViewController.swift` file in the Navigator panel:



2. With the file selected, hit the **delete** key. You will get the following message:



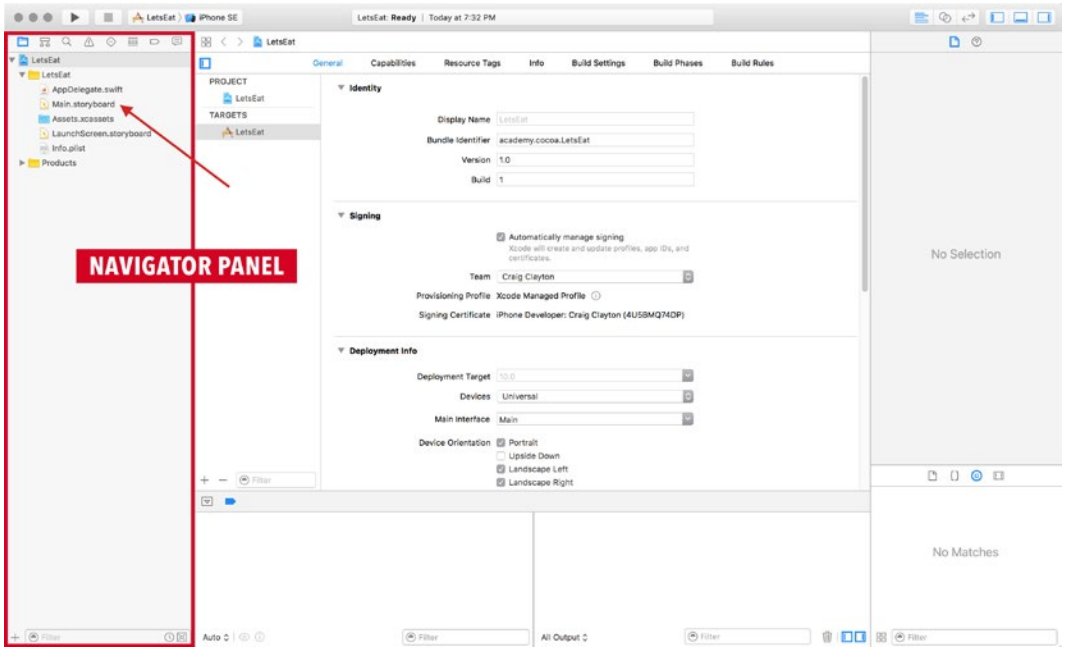
3. Select **Move to Trash**.

Now, we can continue to the setup of the storyboard.

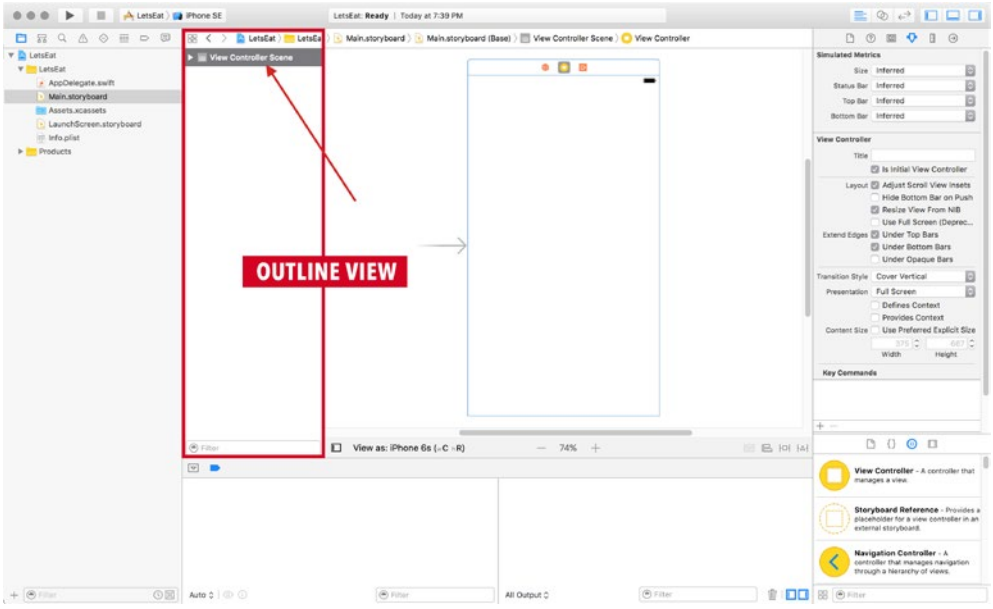
Storyboard setup

Let's get familiar with UI setup. In order to update your `Main.storyboard`, do the following:

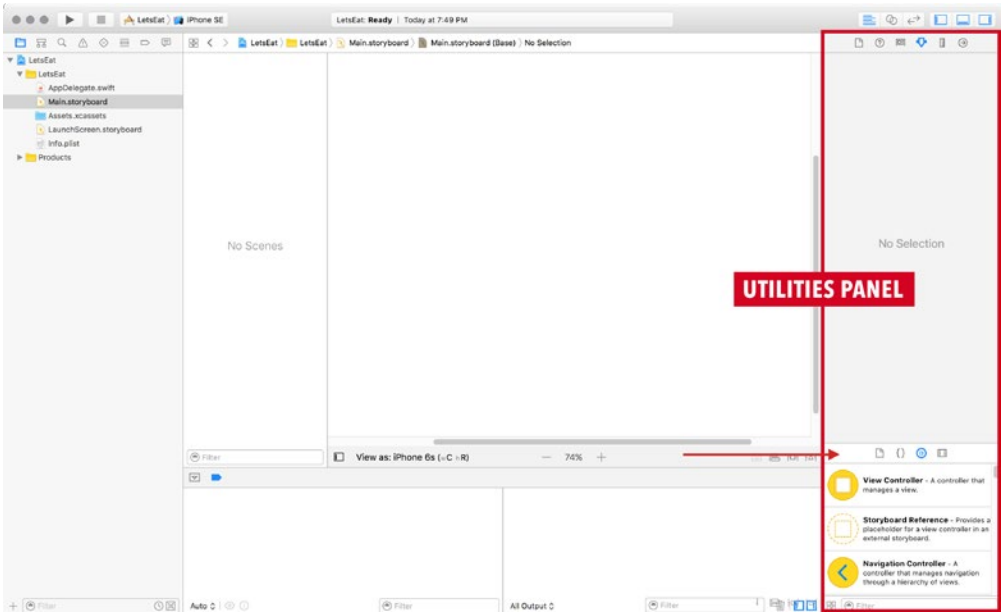
1. Select `Main.storyboard` file in the Navigator panel:



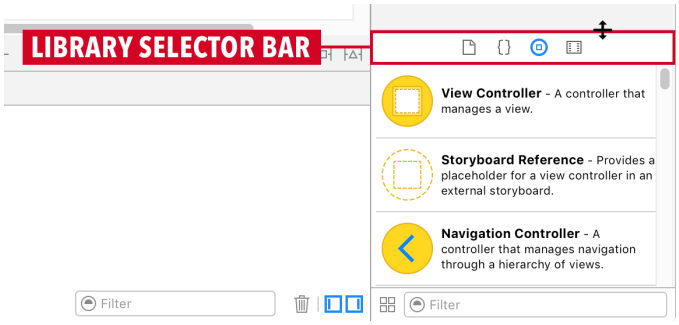
2. In this `storyboard` file, select **View Controller scene** in the Outline view:



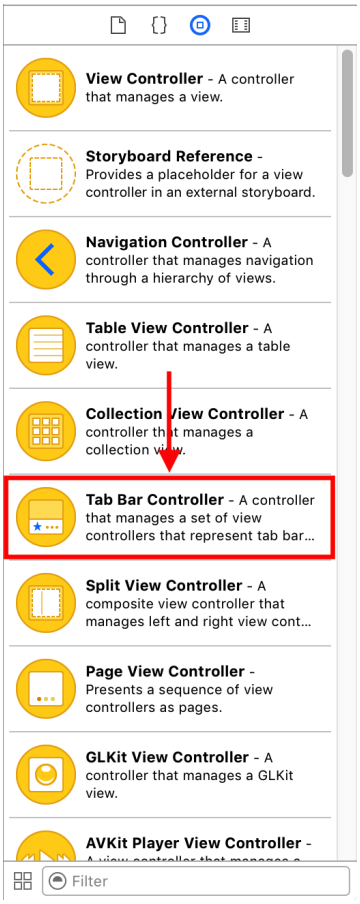
3. With the scene selected, you are going to press the **delete** key, and now your `Main.storyboard` file will be empty.
4. Then, in your Utilities panel, in the bottom pane, you will see the Library selector bar. In the bar, select the object library:



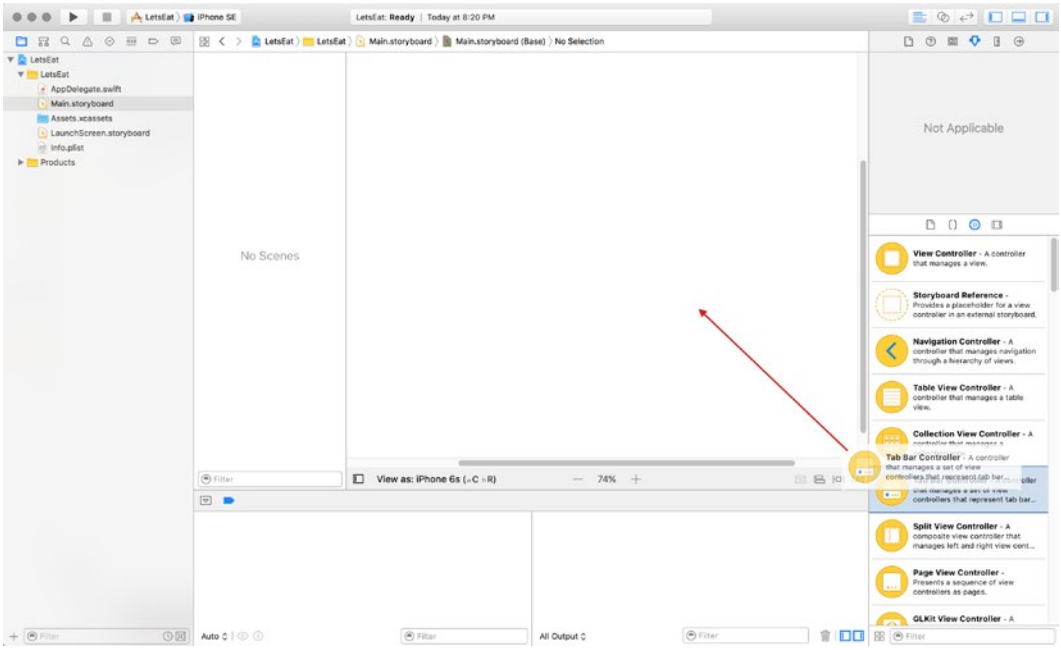
5. Next, you should pull up on the Library selector bar in order to view more of the object library:



6. Find the **Tab Bar Controller**:



7. Now, drag the **Tab Bar Controller** out onto the canvas:



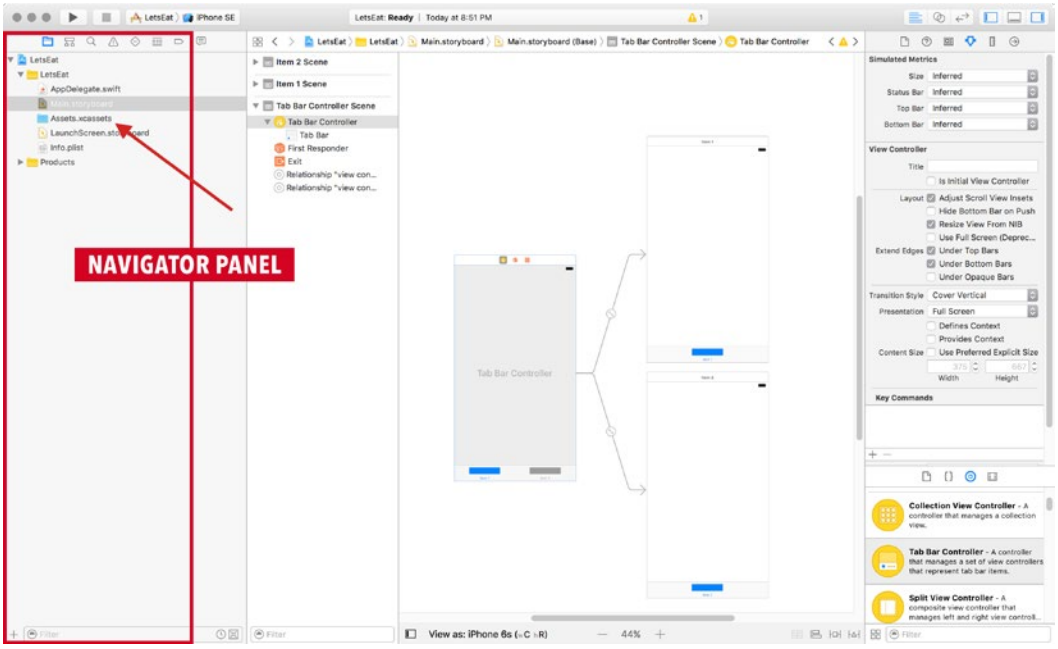
We now have our **Tab Bar Controller**, which will only have two tabs.

Next, we will get our app assets set up so that we can give our tabs image icons.

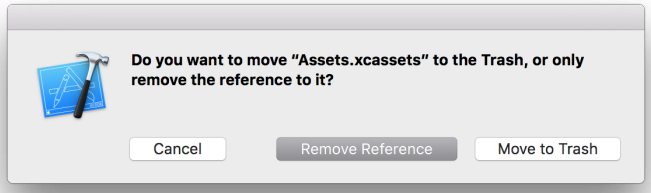
Adding our App assets

Let's add images into our project:

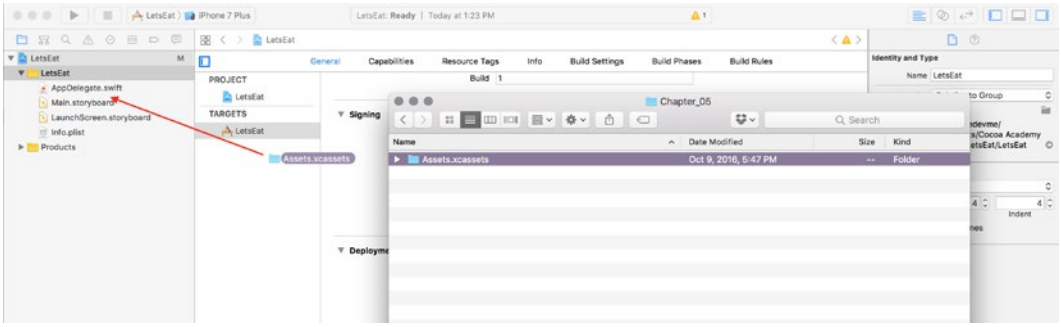
1. Select the `Assets.xcassets` folder in the Navigator panel:



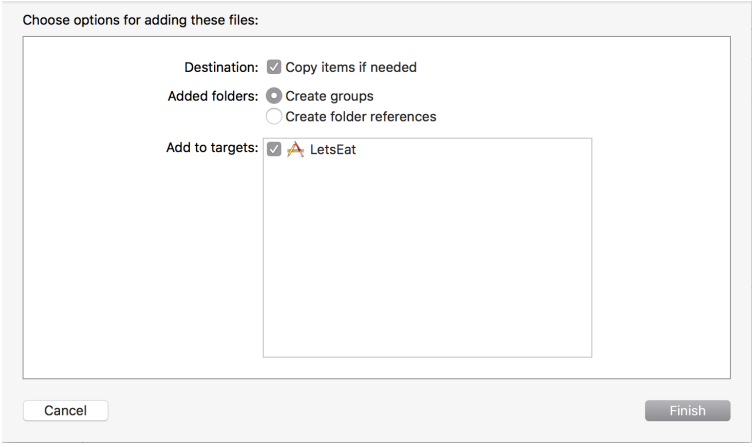
2. Hit the **delete** button, and you will get the following message:



3. Select **Move to Trash**.
4. Open the project assets folder that you downloaded from PacktPub or GitHub. Open `Chapter_05`. Drag the `Assets.xcassets` folder into your project in the Navigator panel:

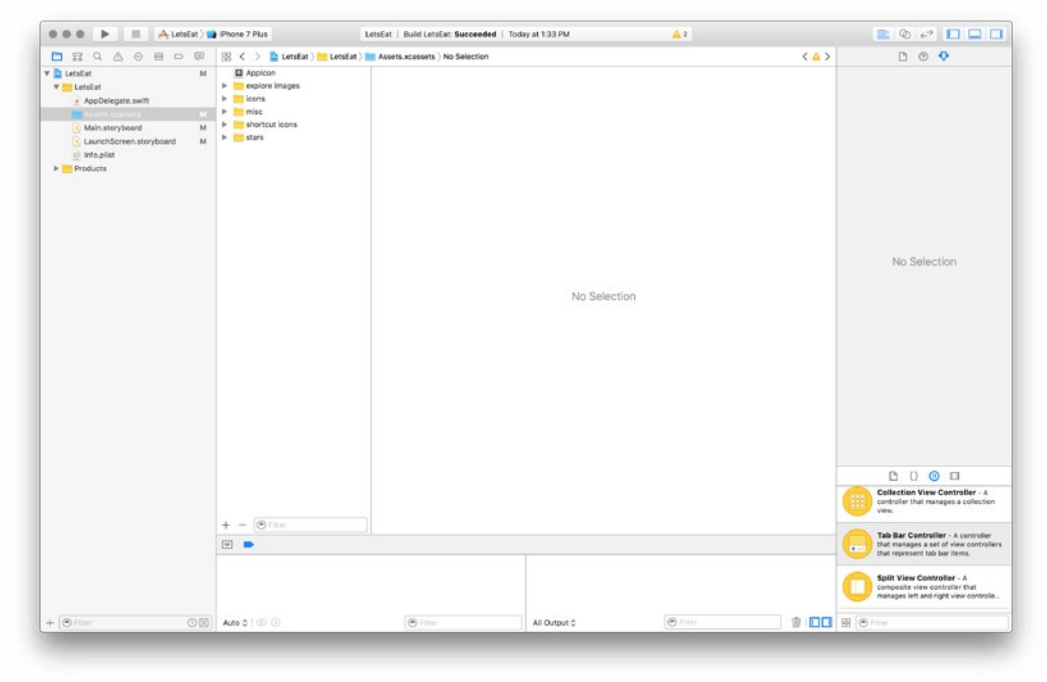


5. When you drop the folder, you will get the following message:



6. Make sure you have both **Copy items if needed** and **Create groups** selected. Then, hit **Finish**.

If you open the `Assets.xcassets` folder, you will now see all the assets for your entire project:

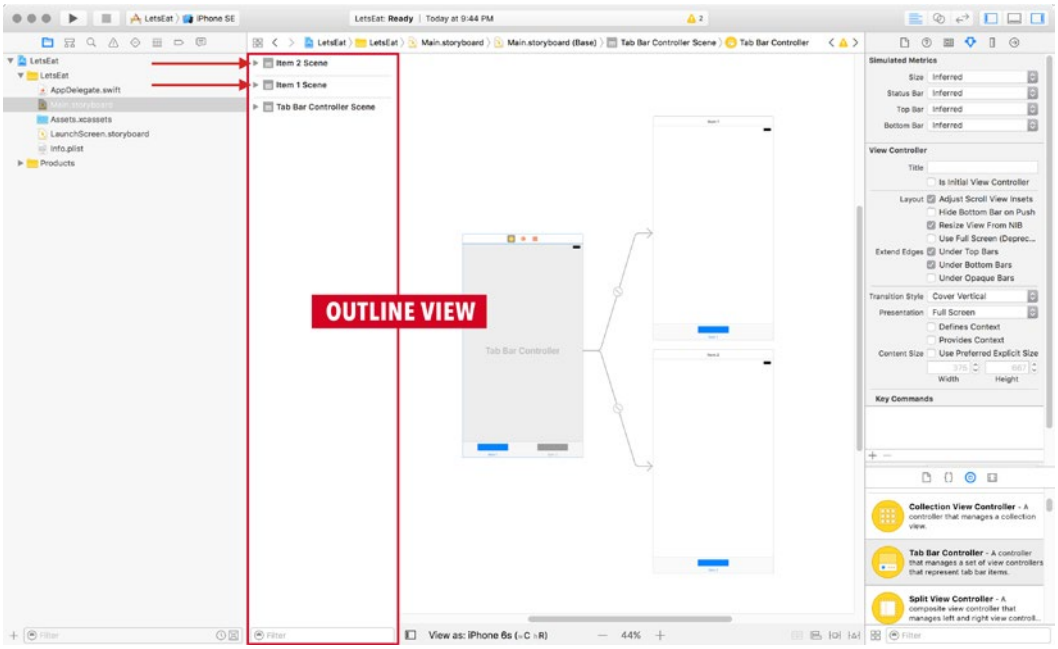


When you explore the assets, you will notice that we will be using both PNGs and PDFs.

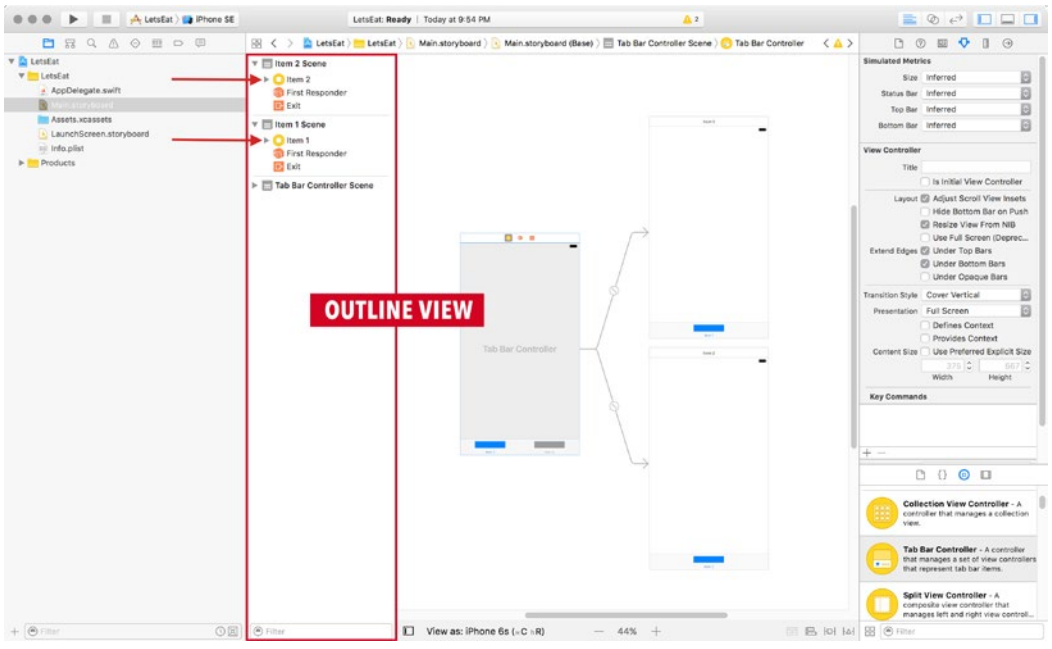


Using PDFs allows us to support multiple device resolutions with only one image. Therefore, Xcode can handle supplying assets for all resolutions.

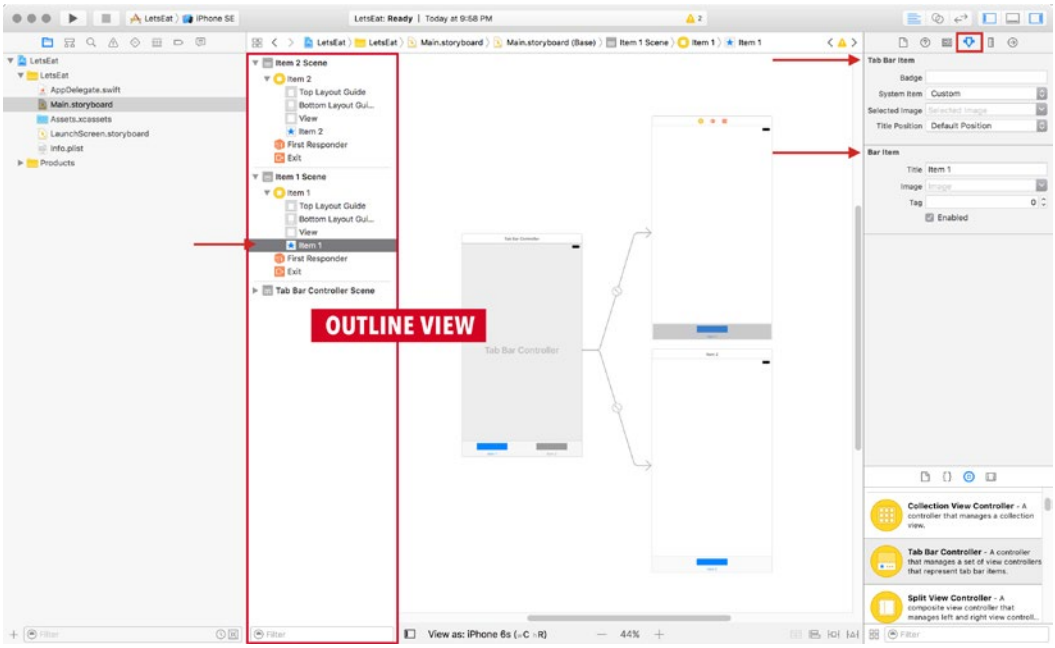
7. Select `Main.storyboard` again, and in the Outline view, you will select both disclosure arrows for **Item 1 Scene** and **Item 2 Scene** in order to have them face downwards:



8. Now, select both disclosure arrows for Item 1 under **Item 1 Scene** and Item 2 under **Item 2 Scene**. Both should be downward facing:



9. Select `Item 1` with the blue star to the left of it, and then select the Attributes Inspector in the Utilities panel:



10. In the panel, use the following values to update your first tab icons:
    - **Tab Bar Item**

      Badge: Leave blank

      System Item: Custom

      Selected Image: `icon-explore-on`

      Title Position: `Default Position`

    - **Bar Item**

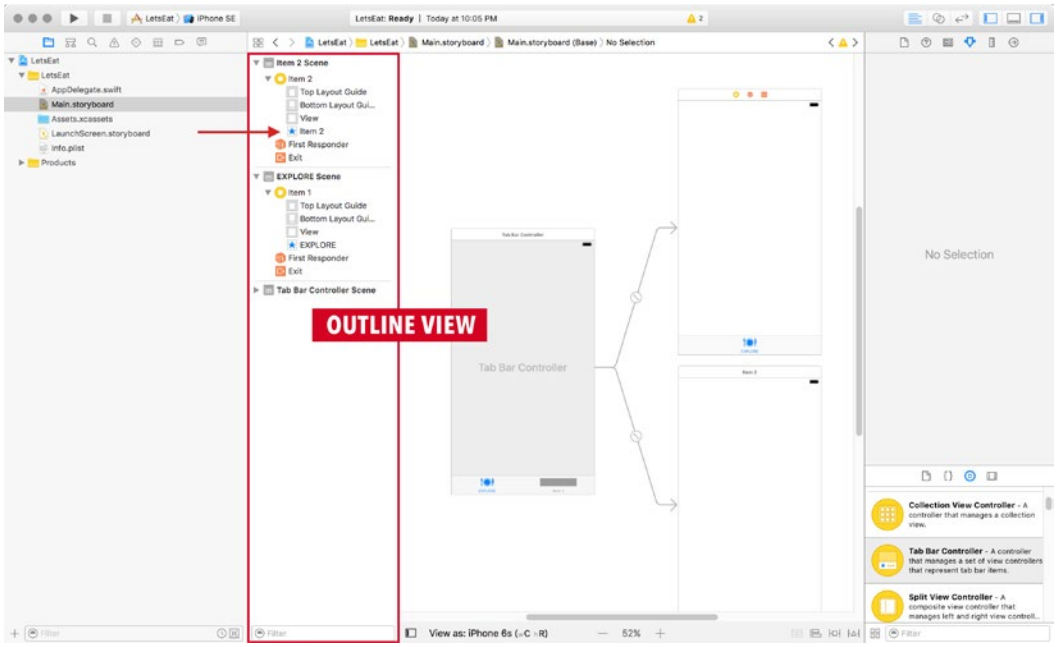
      Title: EXPLORE (ALL CAPS)

      Image: `icon-explore-off`

      Tag: `0`

      Enabled: Selected

11. Now, select Item 2 with the blue star to the left of it in the Outline view, and the Attributes Inspector should already be open:



12. Add the following into the panel:
    - **Tab Bar Item**

      Badge: Leave blank

      System Item: Custom

      Selected Image: icon-map-on

      Title Position: Default Position
    - **Bar Item**

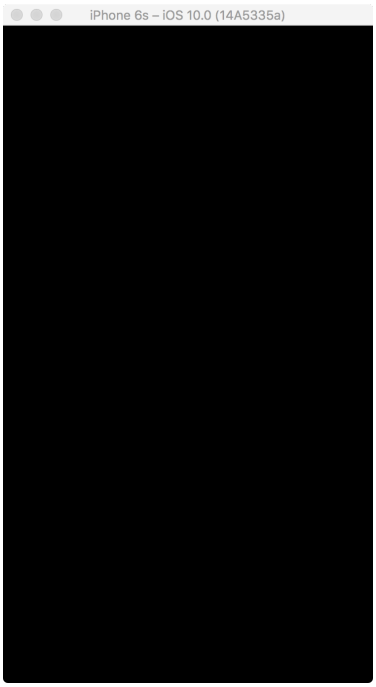
      Title: MAP (ALL CAPS)

      Image: `icon-map-off`

      Tag: `0`

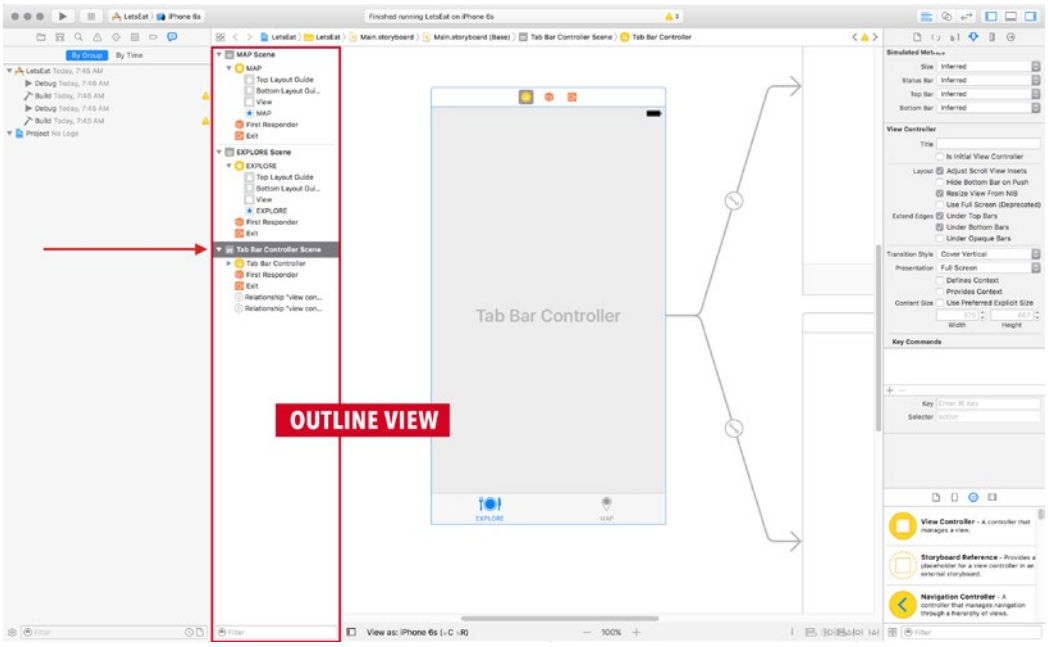
      Enabled: Selected

13. Now, let's run the project by hitting the play button (or use *cmd* + *R*) in order to see where we are:

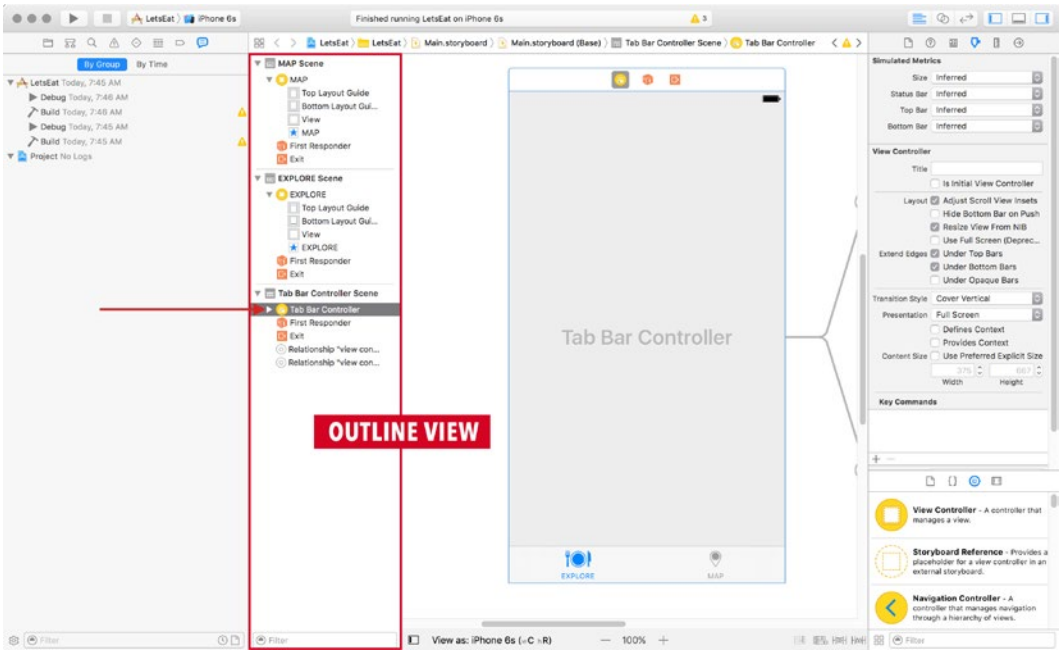


As you may have noticed, this screen does not look like an app. Since we are building a Tab Bar Controller from scratch, we need to add an entry point. Close the simulator and continue with the following steps:

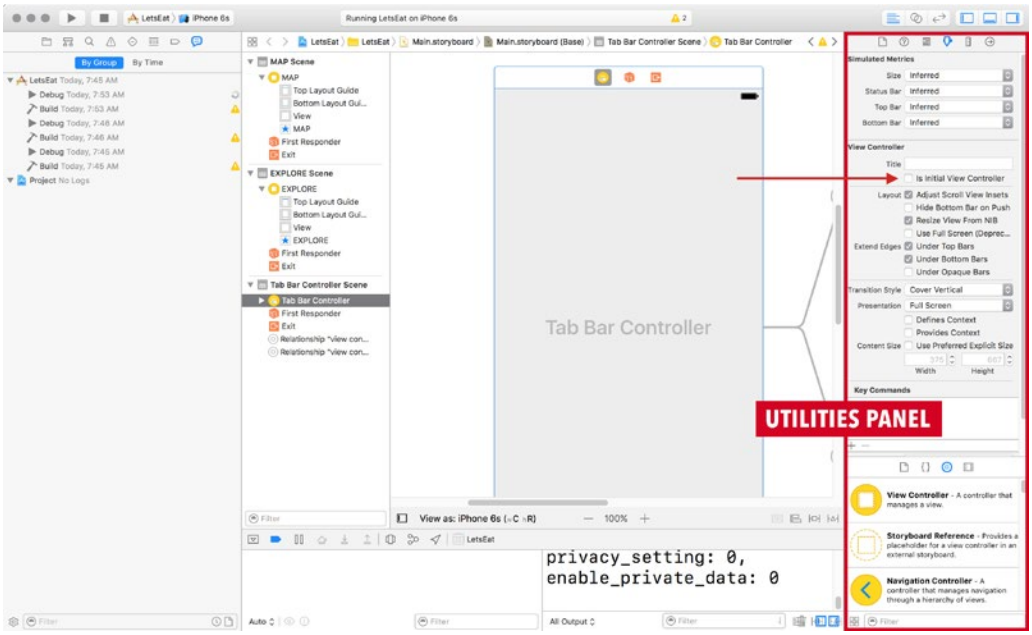
1. Select `Main.storyboard` again in the Outline view, and make sure the disclosure arrow is down for **Tab Bar Controller Scene**:



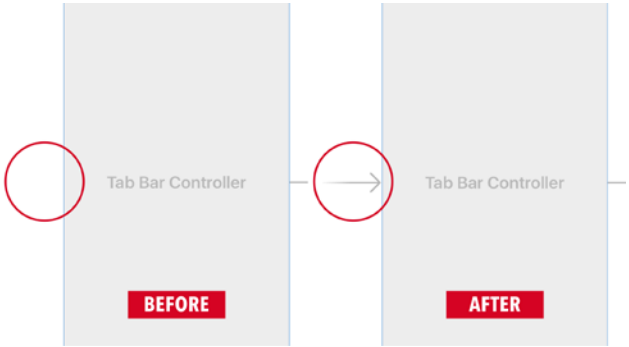
2. Select the **Tab Bar Controller** under the **Tab Bar Controller Scene**, and, in the Utilities panel, make sure the Attributes Inspector is selected:



3. Now, under the **View Controller** section, you will need to check the box for **Is Initial View Controller**:



After you set the initial **View Controller**, there will be an arrow now pointing to the **Tab Bar Controller**. This arrow signifies the entry point of our app:



4. Let's run the project again by hitting the **play** button (or use *CMD+R*):



Perfect! Now, with our basic structure established, we can start adding more specific elements to our views.

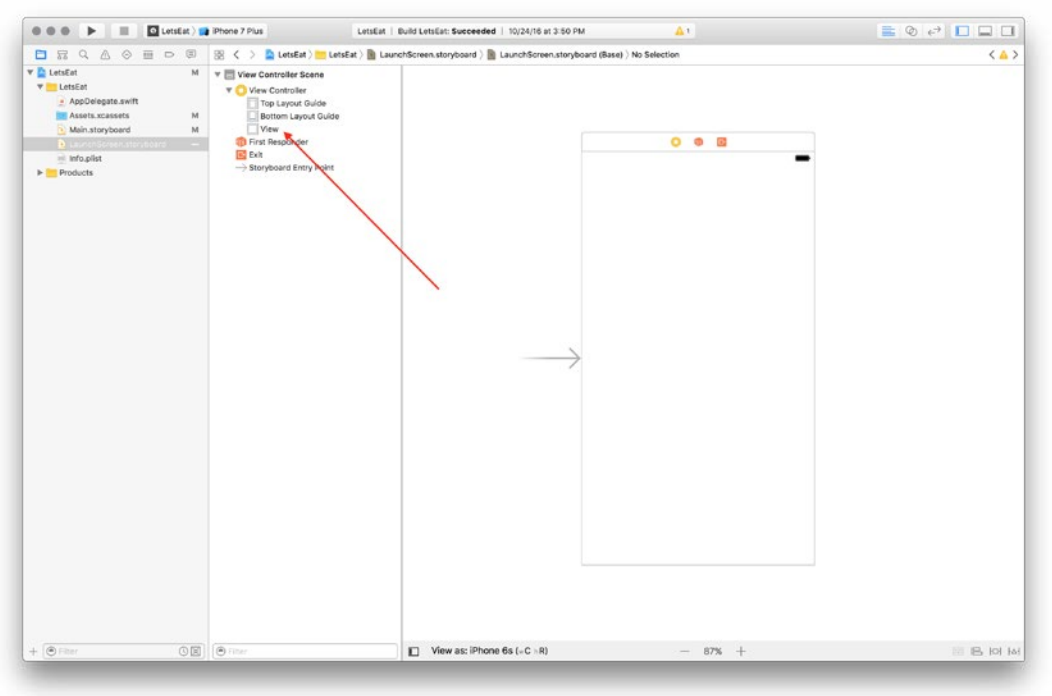
Storyboards

Before we do that, let's update `LaunchScreen.storyboard`. This storyboard is used when our app first launches.

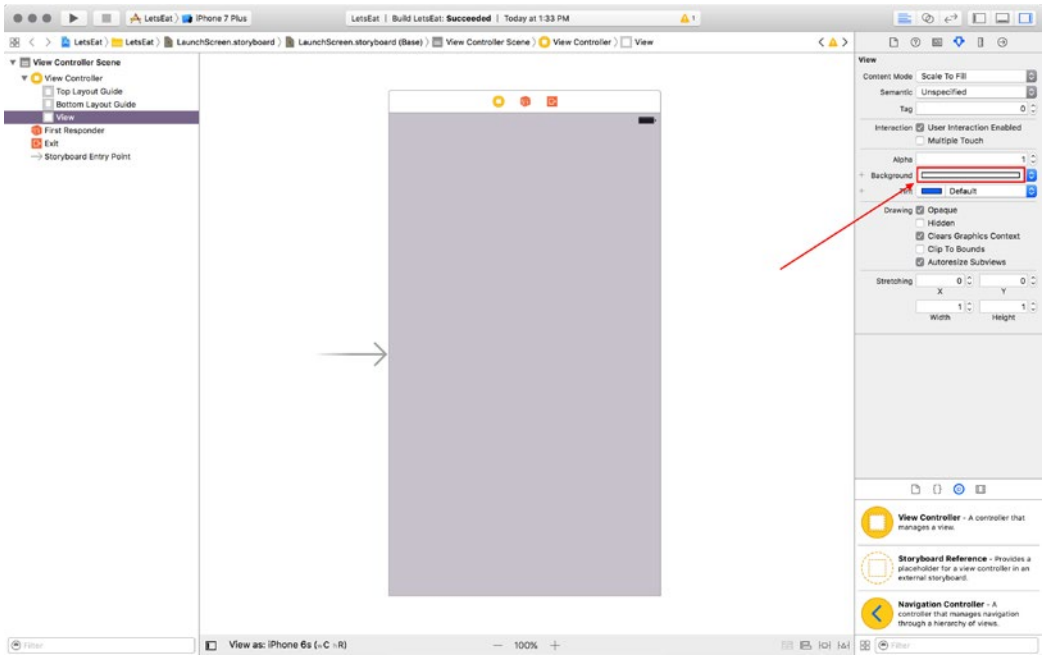
Creating our launch screen

Launch screens can be created using images, but that would mean that you would have to create images for every device and device orientation. Using the `LaunchScreen.storyboard` gives us the ability to create just one asset for all devices and orientations.

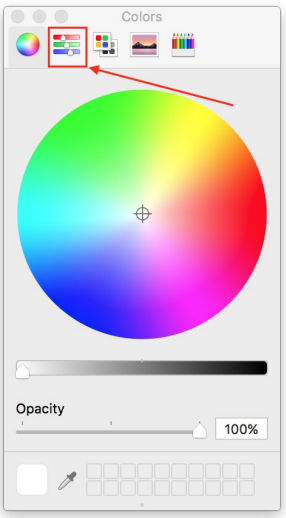
1. Select the `LaunchScreen.storyboard` file, and in the Outline view, make sure that the disclosure arrows for **View Controller Scene** and then **View Controller** are facing downwards. Then, select **View** under **View Controller**:



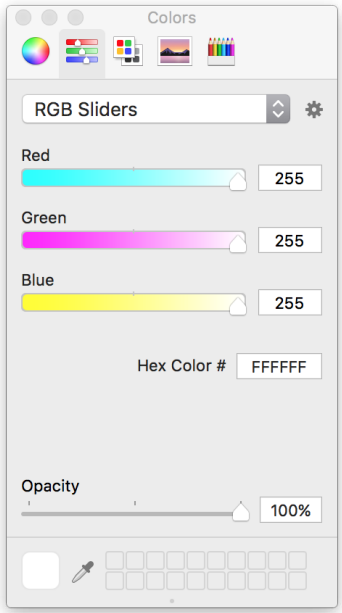
2. In the Utilities panel, select the Attributes Inspector and click on the white Background bar:



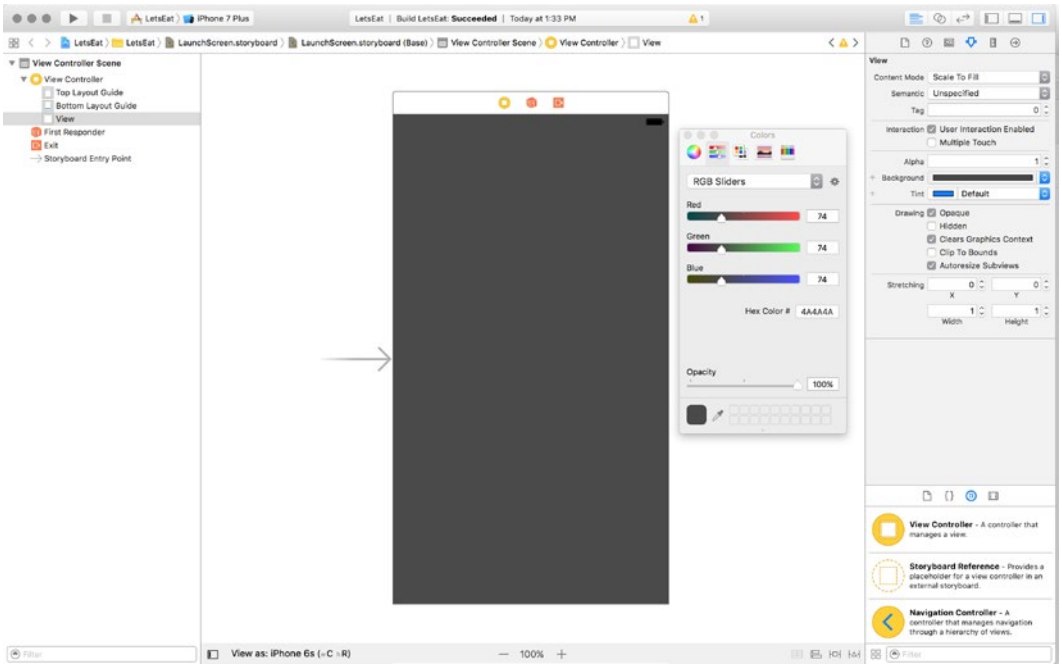
3. Now, you will see a **Colors** panel appear. Select the second tab, which is called the **Color Sliders**:



4. Under **RGB Sliders**, **Hex color #**, update the value from **FFFFFF** to **4A4A4A**. This should change your background color from white to a dark grey:

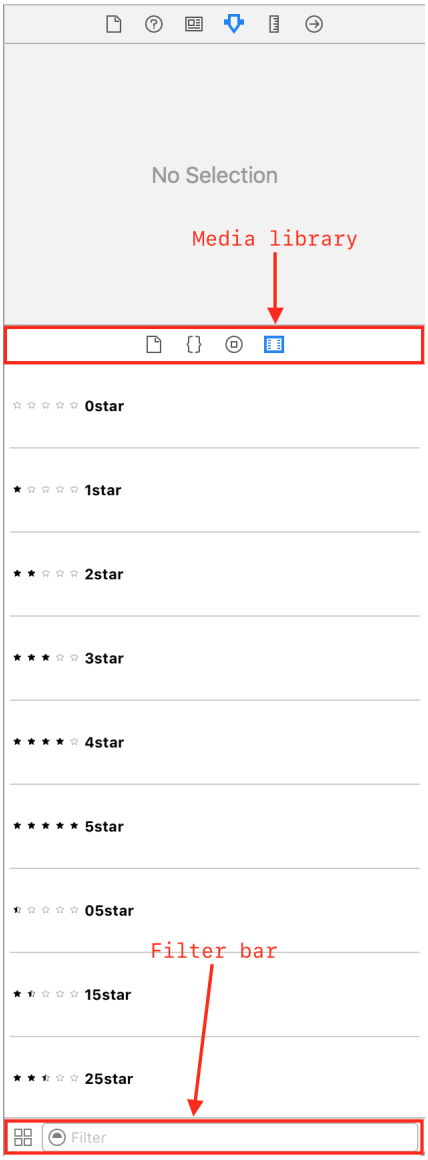


5. You might have to select the background color a second time. If so, just select the Background bar in the Attributes Inspector again, which should change the **Hex Color #** back to **FFFFFF**. Then, just change it again to **4A4A4A**. You now can close the **Color** panel, and you should see the background color update in your Standard Editor panel:



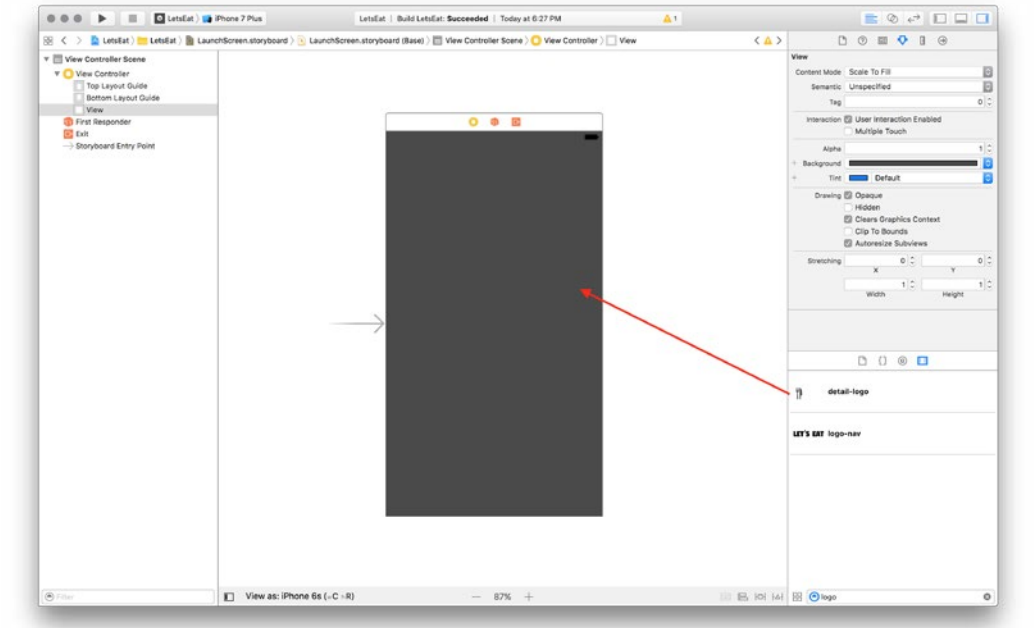
Next, we need to bring the app logo onto the screen:

1. While still in `LaunchScreen.storyboard`, select **Media Library** under the Library selector bar in the Utilities panel:

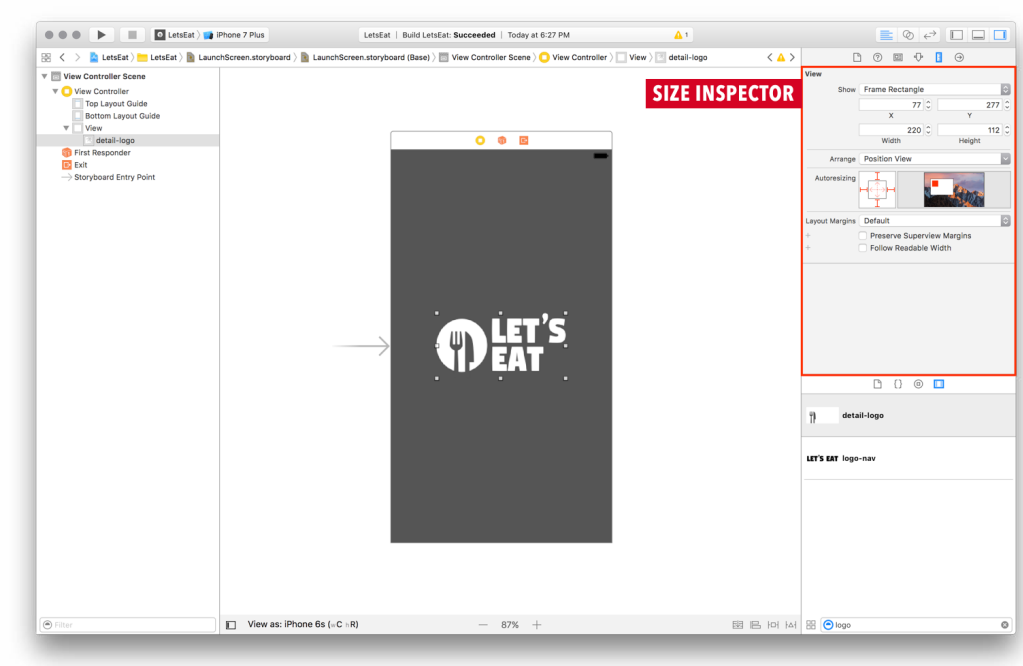


The Media library allows us to access our image assets, and it will place them inside of a UIImageView for us.

2. In the filter, at the bottom of the Library pane, type detail-logo. Once that appears, drag and drop the logo onto the `LaunchScreen.storyboard`:



3. We want our *Let's Eat* logo to appear in the center of the screen, so with the logo selected, open the Size Inspector in the Utilities panel. Set your **X** to `77` and your **Y** to `277`. We will place the logo in the center of our current view:



4. In order for our logo to appear in the center for all devices, we need to apply Auto layout. Select the **detail-logo**, and then select the **Pin** icon near the filter bar:



5. Ensure that the following values are already entered into the **Add New Constraints** panel that appears. If the values need to be entered, do so, and then click on **Add 2 Constraints**:
    - **Width**: `220`
    - **Height**: `112`
6. Next, select the Align icon (shown above) that is to the left of the Pin icon and check the following boxes that appear:
    - Horizontally in container
    - Vertically in container
7. Then, click on **Add 2 Constraints**.

    When you are done, you will see the following:



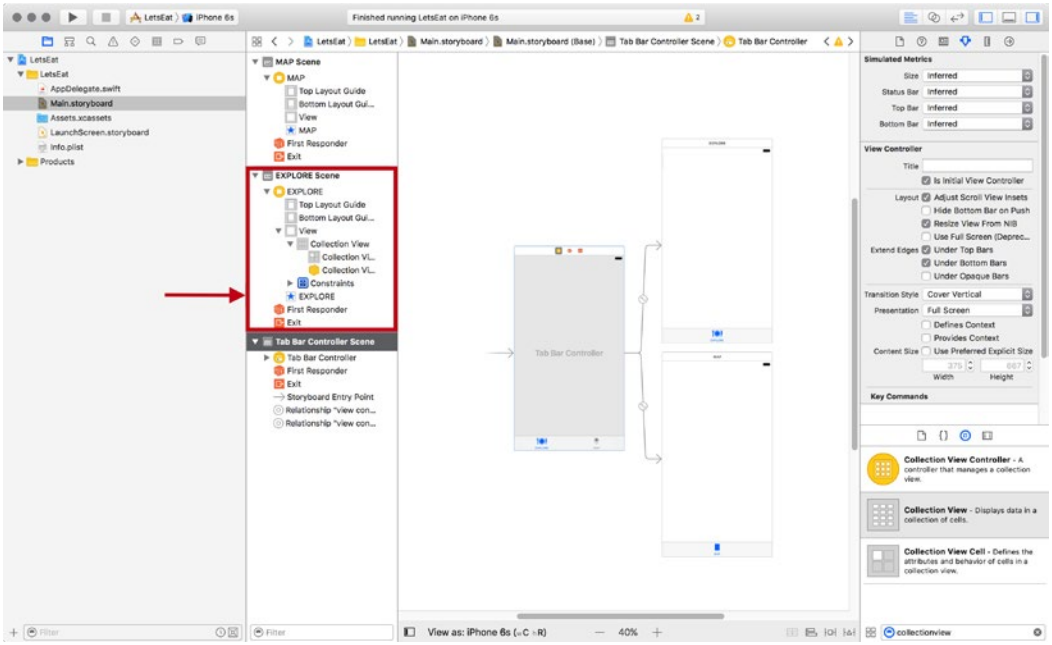
Our launch screen is now set up for all devices. If you run the project again, you will now see your launch screen with the *Let's Eat* logo and new background color.

Let's move onto adding detail to our Explore tab, since this is the first thing a user will see after the app launches.

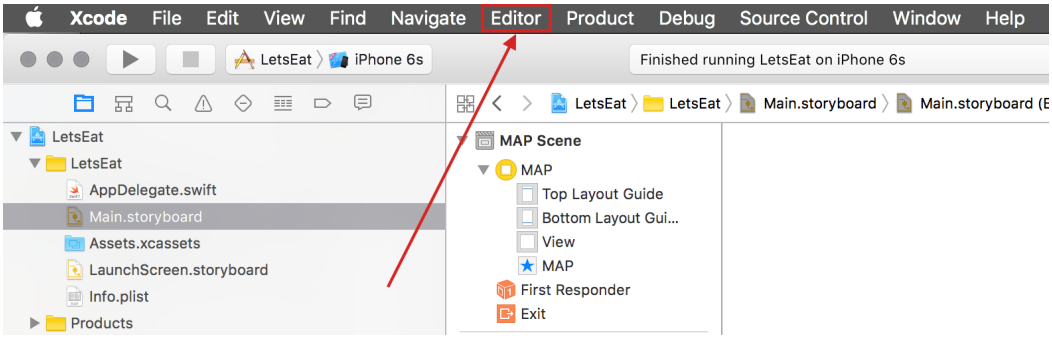
Adding a Navigation Controller

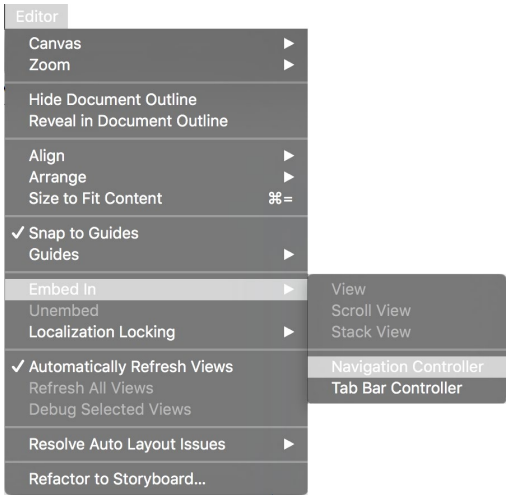
We first need to add a Navigation Controller to our Explore tab. This will allow us to do a few things, such as adding a button into the title bar of the navigation in order to present our cities list.

1. Select `Main.storyboard`, and in the Outline view, select **EXPLORE** with the blue star to the left of it, under **EXPLORE** in the **Explore Scene**:

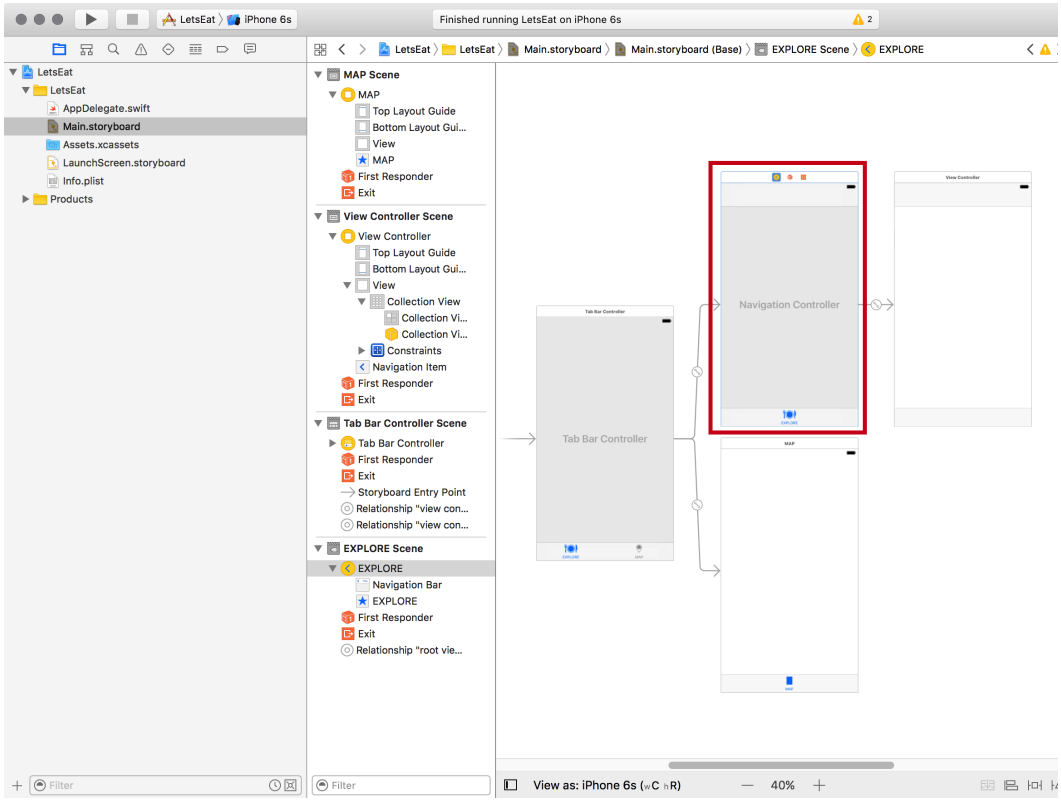


2. Then, navigate to **Editor | Embed In | Navigation Controller**:





3. Now, our View Controller has a Navigation Controller:



4. Let's run the project by hitting the play button (or use *cmd* + *R*):



Repeat steps 1 to 4 under **Adding a Navigation Controller** above for the **Map** tab. After you add both Navigation Controllers, we need to create a custom title bar for each of the Explore and Map tabs. We will start with the Explore tab.

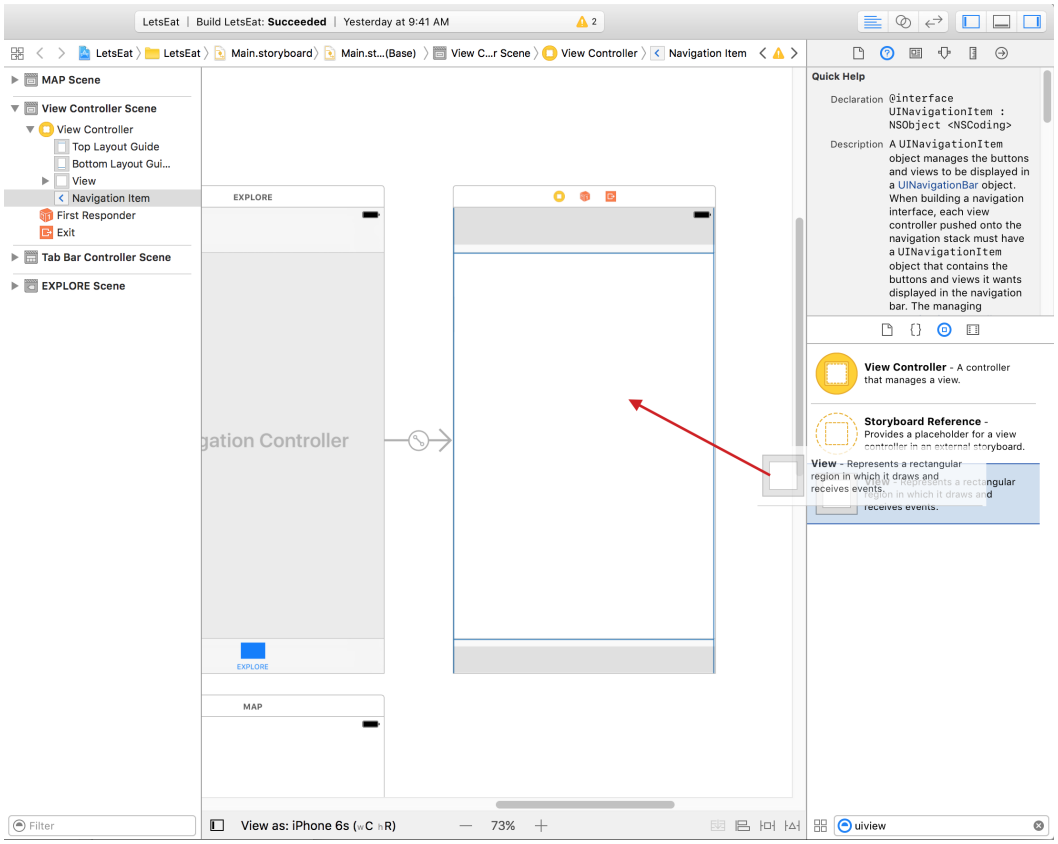
Creating a custom title view

In order to add anything into our title view, we must first add a UIView. They are containers into which you can put other UI elements.

Adding a container

Perform the following steps for adding a container:

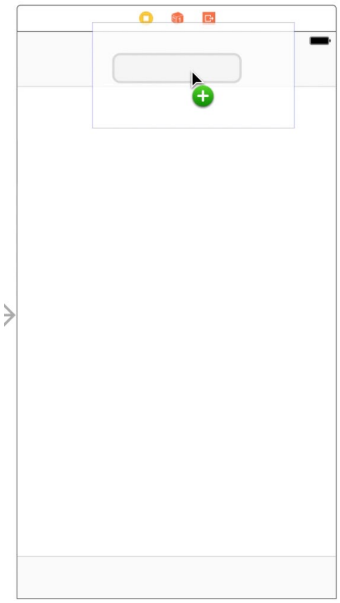
1. In the filter field of your Utilities panel, type—`uiview`.
2. You will see **View** at the bottom. Drag and drop this into your Explore navigation bar:



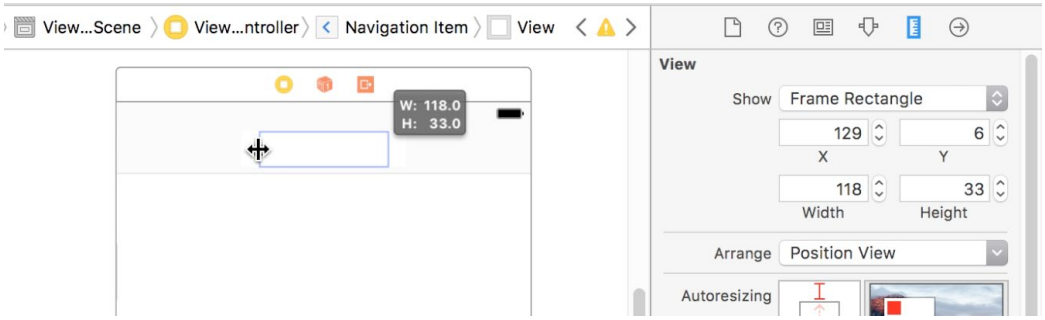


A way to confirm that you have dragged and dropped items into the proper place is to go to Outline view and see that the new item has been added under the correct label.

3. You will see it highlighted when you are inside of it:



4. With the **View** selected, go to the Utilities panel and select the Size Inspector.
5. Now, drag the View box until its width is `118` and height is `33` in the Size Inspector:



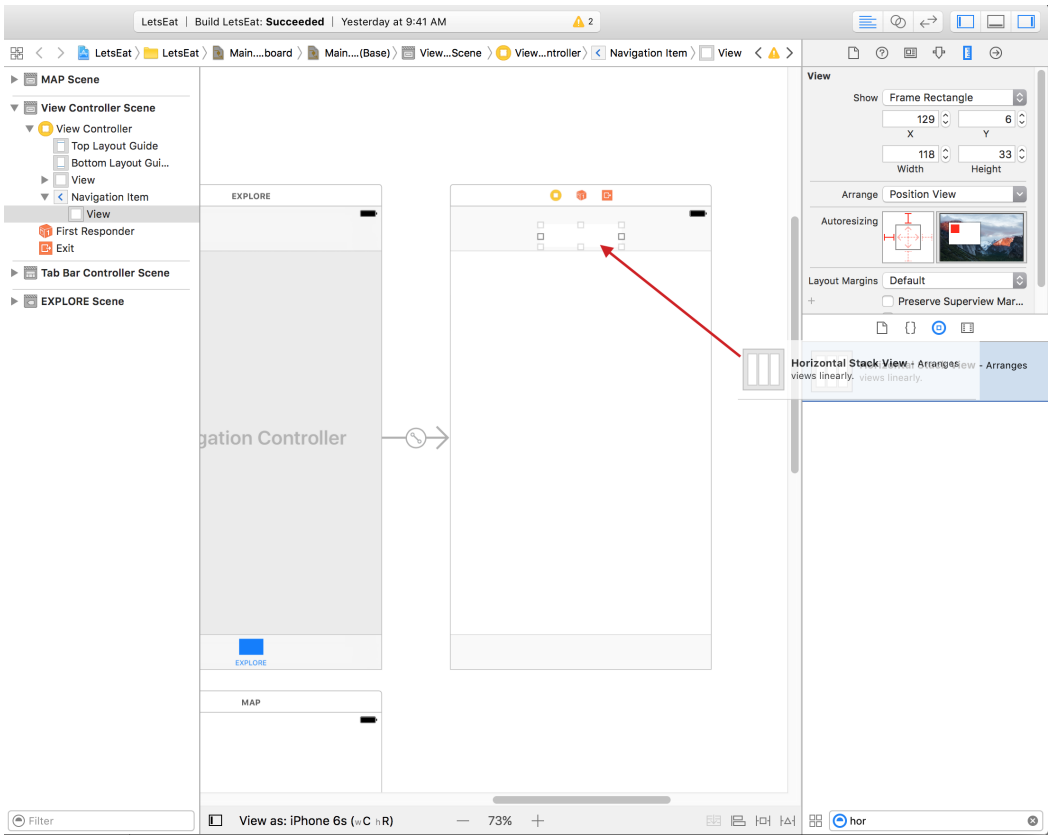
6. We have now created our container. Let's add our Stack View into this container.

Using Stack Views

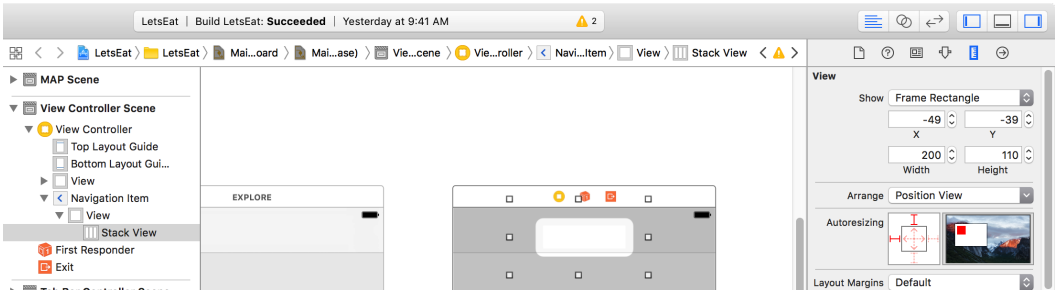
Stack Views are really helpful to ensure that things are stacked either horizontally or vertically. We will use a Stack View inside of our custom title view to keep things in place better, such as a label and an arrow that we will be adding. Stack View makes it easier to have the label and arrow line up next to each other. This also allows us to limit the amount of Auto layout we need to use.

Now that we have a better idea of what a Stack View is, let's see how it works.

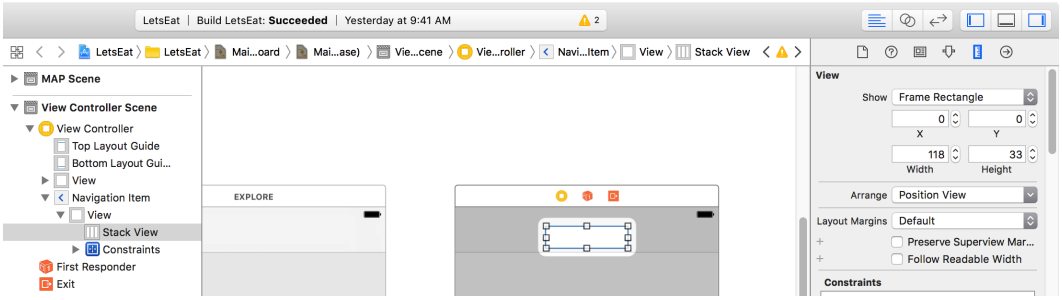
1. In the Utilities panel, go to the filter area and type: `hor`. You will notice only one item, **Horizontal Stack View**.
2. Drag and drop the **Horizontal Stack View** into the View that we just added:



3. Your **Horizontal Stack View** should now be inside of the View:



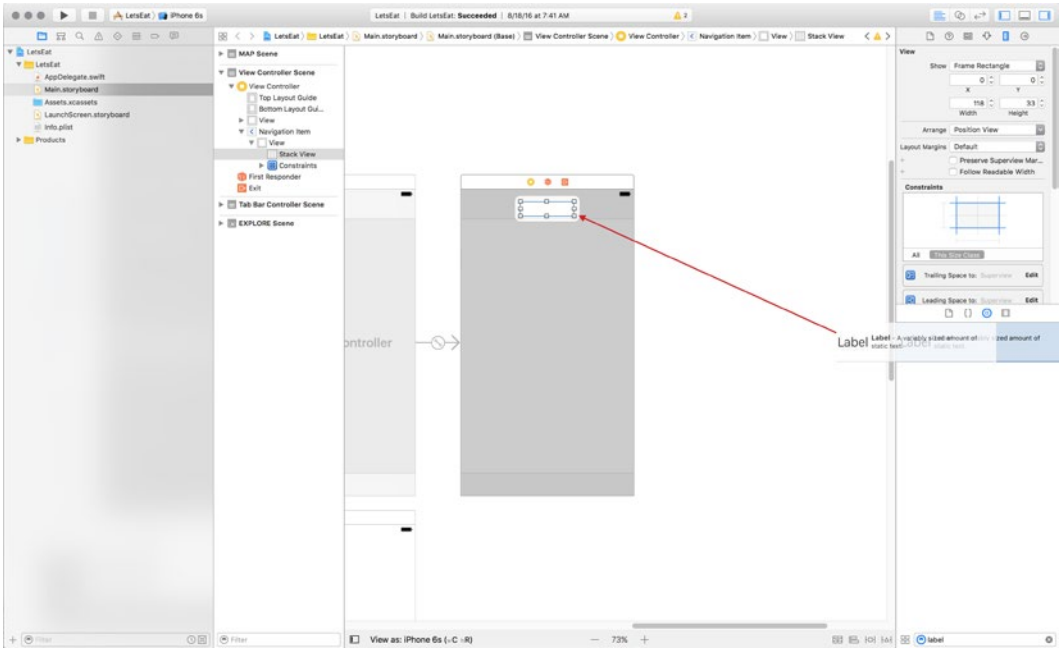
4. With the **Stack View** selected, in the Utilities panel, select the Size **Inspector** and add the following values:
    - **X**: `0`
    - **Y**: `0`
    - **Width**: `118`
    - **Height**: `33`
5. With the **Stack View** still selected, select the **Pin** icon and add the following values, and then click on **Add 4 Constraints**:
    - All values under **Add New Constraints** should be set to `0`
    - **Update Frames** section should be changed to **Items of New Constraints**
6. Your Stack View will now look like the following:



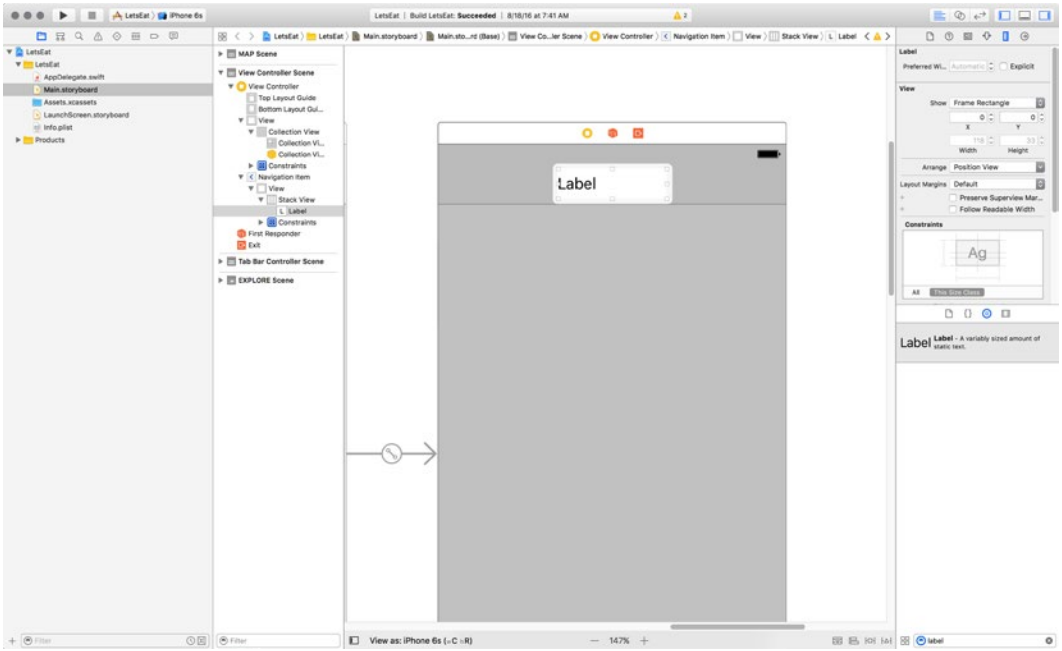
Now that we have our Stack View set up, we will add our UI elements into the Stack View; a label and an arrow.

Adding a Custom Label and an Arrow to Our Custom Title View

1. Type in the filter area of the Utilities panel—label. You will notice only one item, **Label**.
2. Drag style into the Stack View:



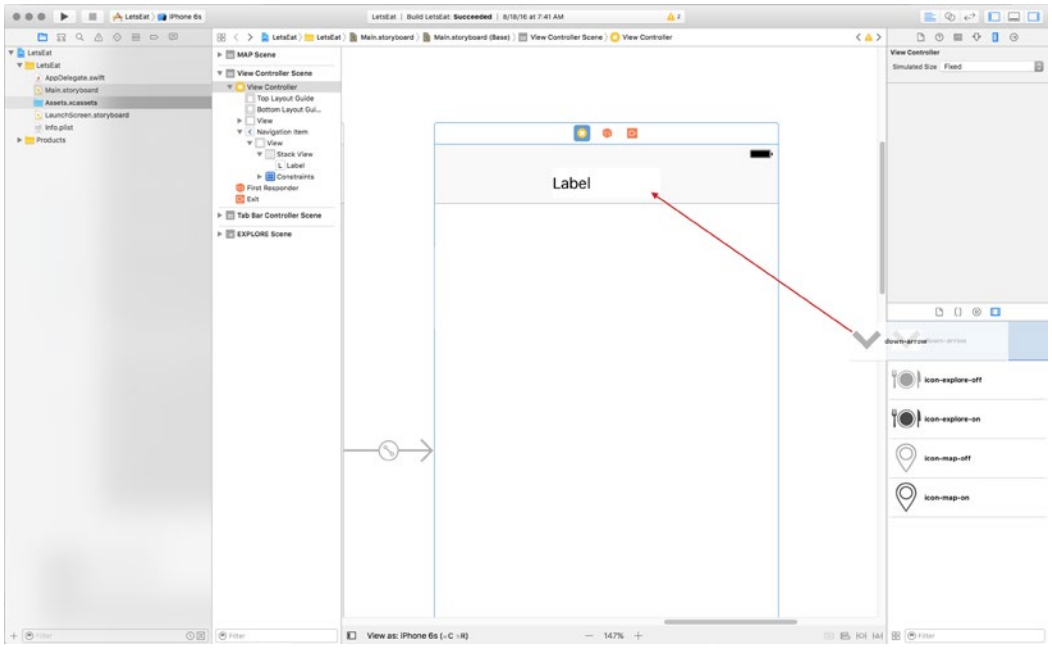
3. When we drag the Label inside of the Stack View, it will fill the entire area:



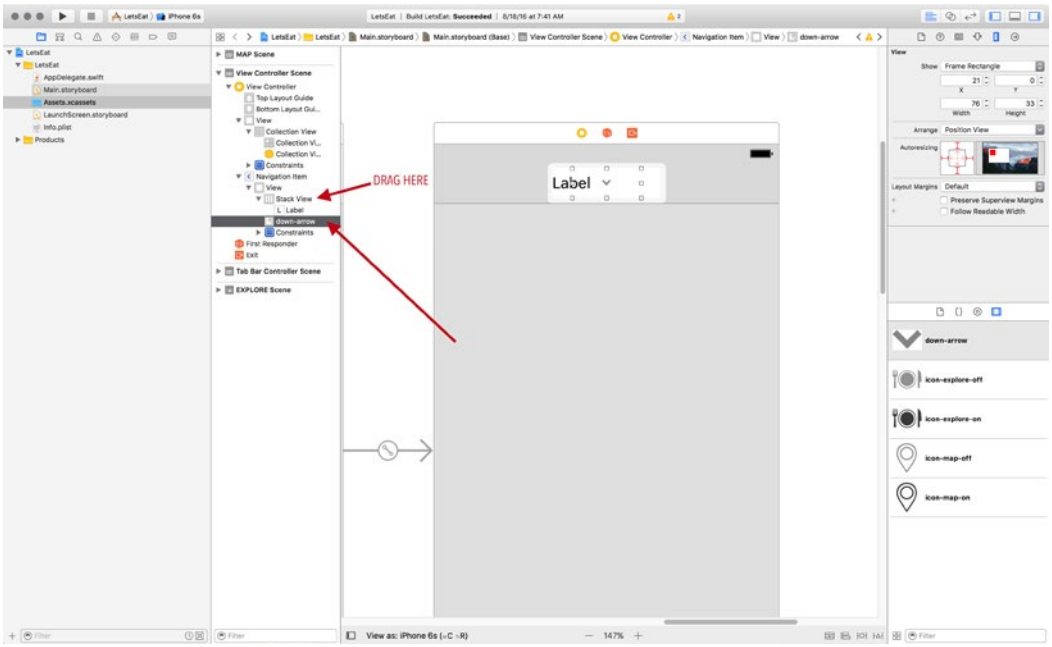
Now, let's add an arrow into our custom title view.

4. In the Utilities panel, select the Media library. You might see "No Matches" because we still have a search term in our filter field. Delete the search term, and then you will see our assets that we added earlier.

5. In the filter area, type `down-arrow`. Grab it and drag it into the Stack View next to the Label:



6. If you accidentally dragged the down arrow into the wrong area, you can go to your Outline view and drag it into the Stack View under the Label:



If you run the project again, you should end up with the following:



This is a good place to stop for now. We still have elements left to add, but this gives us a great starting point. Now that we have our app structure build out, we need to start making the app come alive using Swift 3.

Summary

In this chapter, we took an app tour of our *Let's Eat* app design. We also covered the basic UI setup of our app with some really simple Auto layout. We were able turn our Single View application into a Tab Bar Controller. You also learned how to add images for our tab bar icons.

Storyboarding is one of the things I really enjoy doing. It is quick and easy to set up your UI with storyboards. Being able to drag and drop what you need onto the canvas is such an efficient method of developing app storyboards. There are times when you will need to code, but, being able to work on things without having to write any code, is a wonderful capability. My preference is to use storyboards as much as possible, but there are many developers who prefer to do it in code. If you come from another programming language, try to keep an open mind and really learn storyboarding. When you work on a project that uses storyboards, you can get a high-level overview of the project. When everything is written in code, it takes more time to get a basic idea of how the app is structured and its overall flow. Again, there are people who love to code their UI; and we will do some of that in this book. My main point is that you have to find what works for you. This book leans more toward the storyboard side versus coding side of setting up your UI.

In the next chapter, we will continue setting up our UI and get familiar with more of the UI elements that you have seen in many iOS apps.

6
Setting Up UI

When you start using Storyboard, it can be a bit overwhelming as with most things that you do for the first time. The hardest part for most is getting used to all of the panels. As you get more comfortable, you will see that it will become easier, and jumping from panel to panel will become second nature to you.

In the last chapter, we created our project and set up our Tab Bar Controller. We also started working on creating our custom title view. In addition, we have some basic understanding of UI terms, such as storyboards, segues, and all the different controllers.

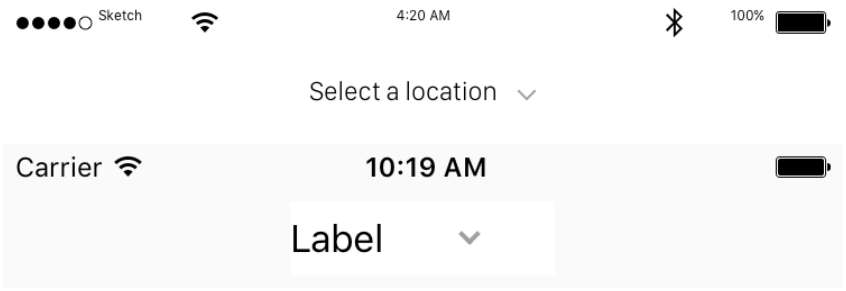
In this chapter, after cleaning up the custom title view, we will turn our attention to Collection View UI setup as well as Map UI setup. We will focus on the visual side of our app in this chapter, and before we reach the end of this chapter, we will do a little bit of Swift.

We will cover the following in this chapter:

- Design Clean Up
- Collection View
- Map Kit View
- Refactoring the Storyboard
- Folder Setup
- Setting up Global Settings

Design clean up

In the last chapter, we left off with both our Label and our down arrow inside of our Stack View. Let's run the project and see what we have so far by hitting the **Play** button (or use *cmd* + *R*). If we compare what we have with what the final design should look like (shown in the App Tour in *Chapter 5*, *Starting the UI Setup*), we can see it is not the same:

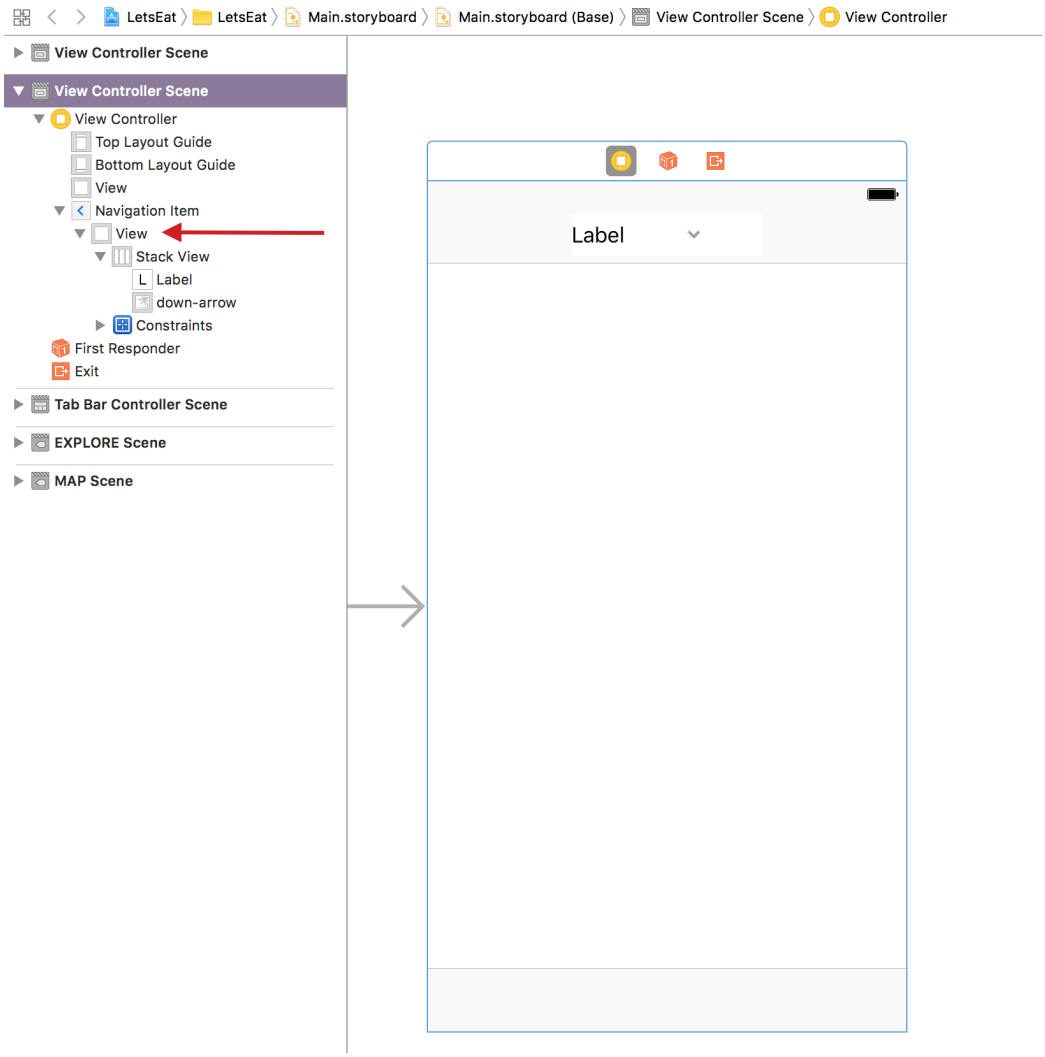


Therefore, we need to fix our custom title view.

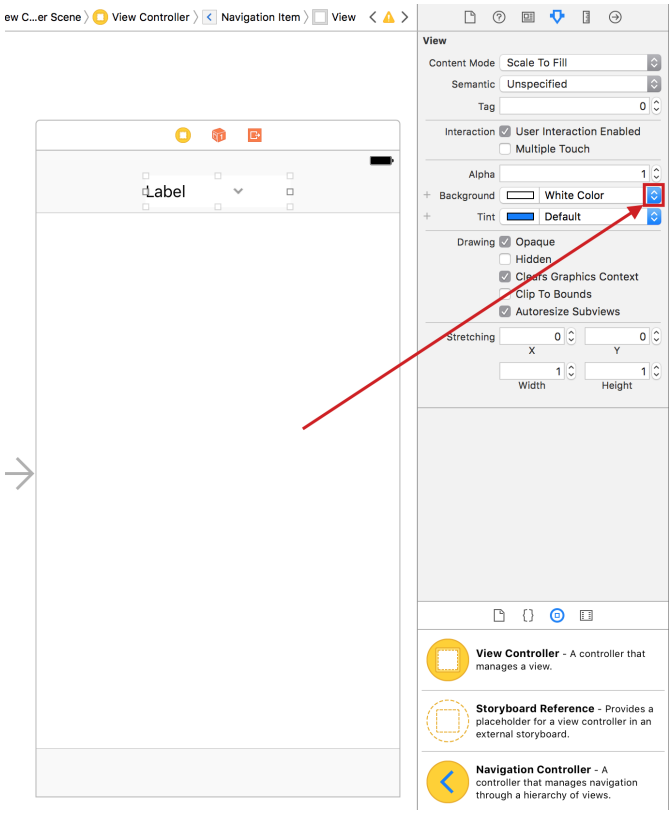
Adding a Clear Background to the Custom Title View

Our first update will be to change the custom title view's background from white to clear. Then, we need to correct the spacing between the **Label** and the arrow. Finally, we will need to rename the **Label**. Let's update our background first:

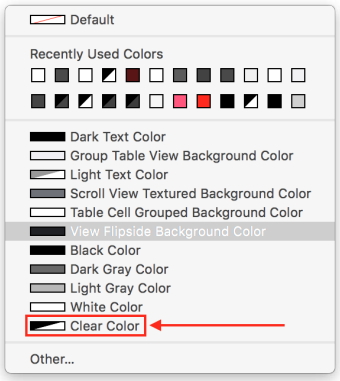
1. In the Outline view, select **View**:



2. In your Utilities panel, make sure you are in the Attributes Inspector and click on the color drop-down menu under **Background**:



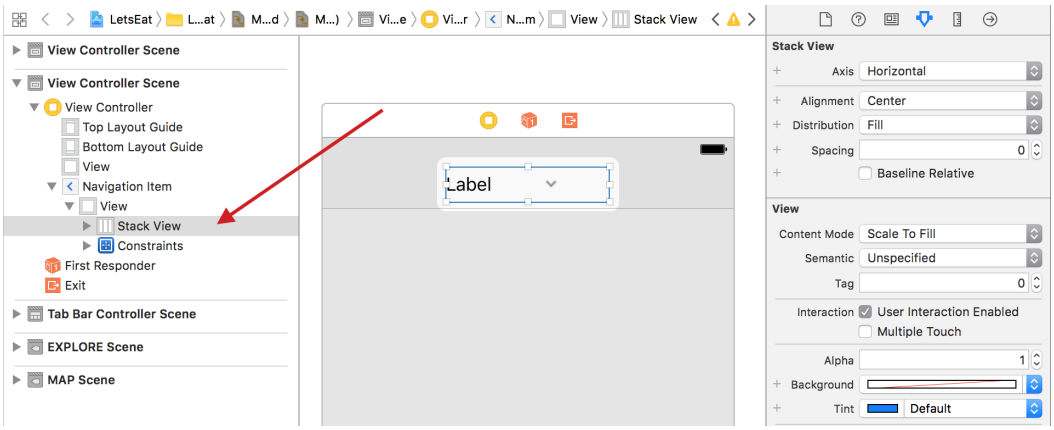
3. Now, select **Clear Color**, which will give us a clear background for our view:



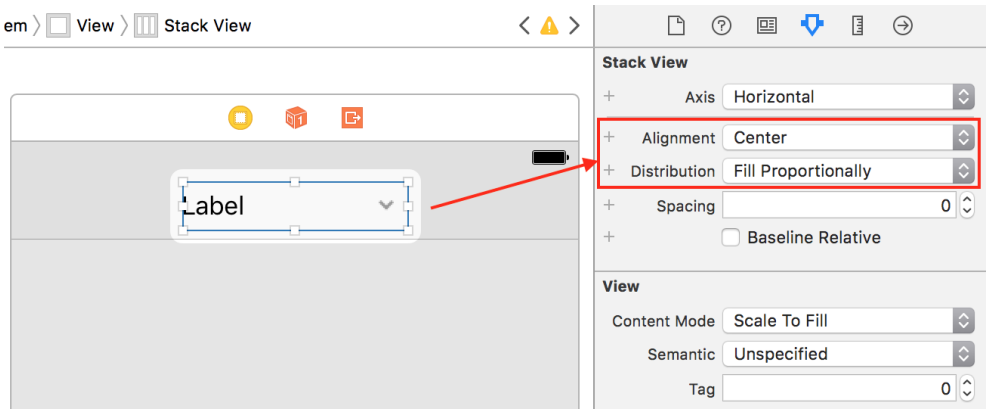
Updating the UIStackView

Now that the background color is clear, let's update our **Stack View**. We want to center both the **Label** and the arrow and to fill the **Stack View** space proportionally by following these steps:

1. Select the **Stack View**, and make sure that you are on the Attributes Inspector in the Utilities panel:



2. Update the following items:
    - **Alignment**: **Center**
    - **Distribution**: **Fill Proportionally**

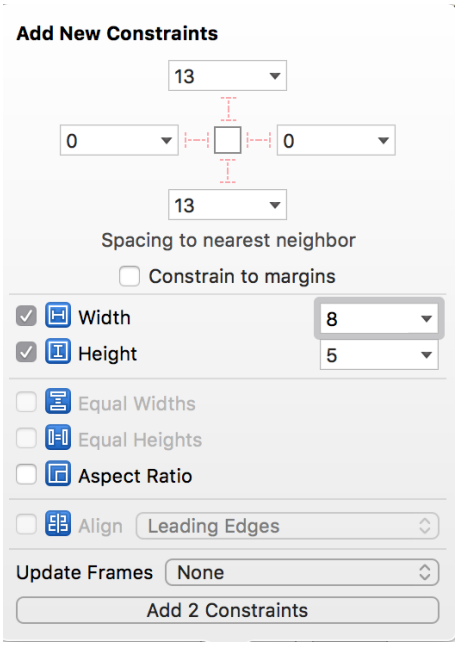


Now, we have both the **Label** and arrow centered and filling the space proportionally.

Updating our Arrow

Next, we need give our arrow a width and height:

1. Select the drop-down arrow under **Stack View** in the Outline view.
2. Now, select the Pin icon and enter the following values into the panel shown:
    - **Width**: **8**
    - **Height**: **5**

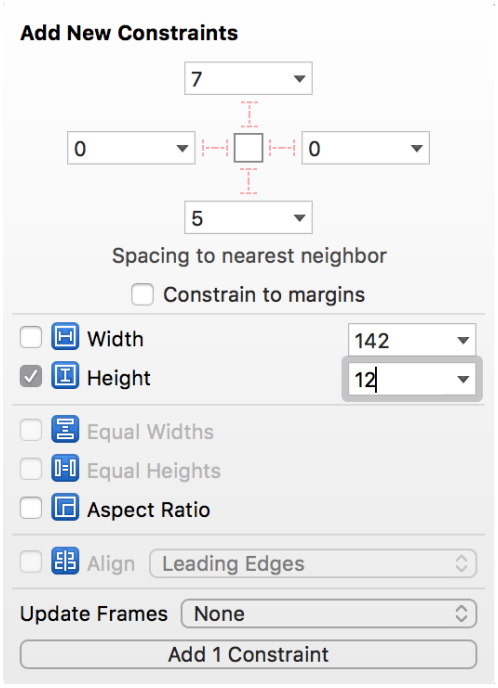


3. Click on **Add 2 Constraints**.

Updating our Label

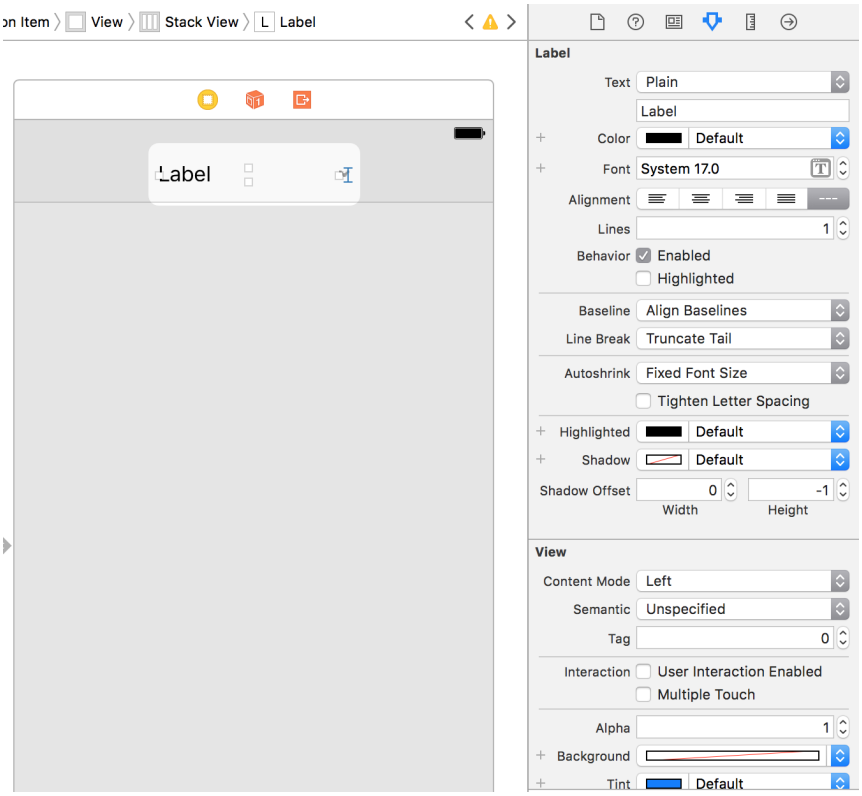
We also need to give our Label an updated height, and then rename it:

1. Select Label under Stack View in the Outline view.
2. Now, select the Pin and enter the following values into the panel shown in the following screenshot:
    - **Height**: **12**
    - **Update Frames**: Items of New Constraint



3. Click on **Add 1 Constraint**.

4. Now, with the **Label** still selected in Outline view; make sure that you are on the Attributes Inspector in the Utilities panel:



5. Update the following items:
    - **Text**: Change **Label** to **Select a location**
    - **Alignment**: Center
    - **Font Size**: 12

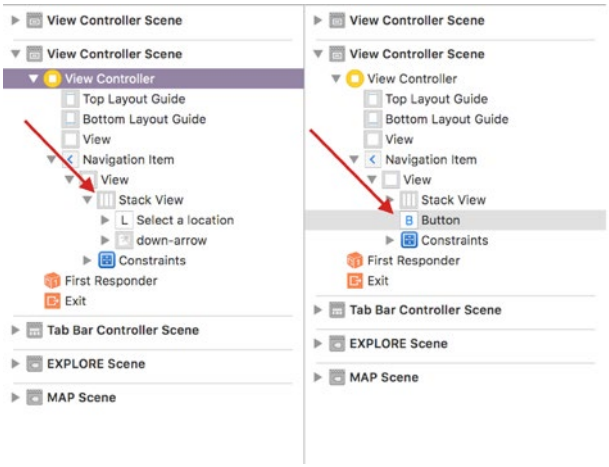
6. Let's run the project by hitting the **Play** button (or use *cmd* + *R*):



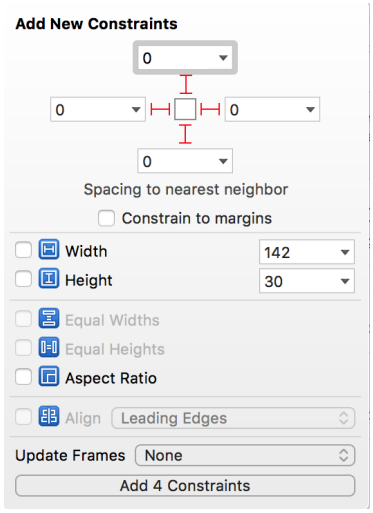
The custom title view is now correctly updated; however, if you tap it, nothing happens. Therefore, we need to make the view act like a button. In order to do this, we will be placing a button above the Stack View, which will allow us to accept a button tap.

Adding a Button

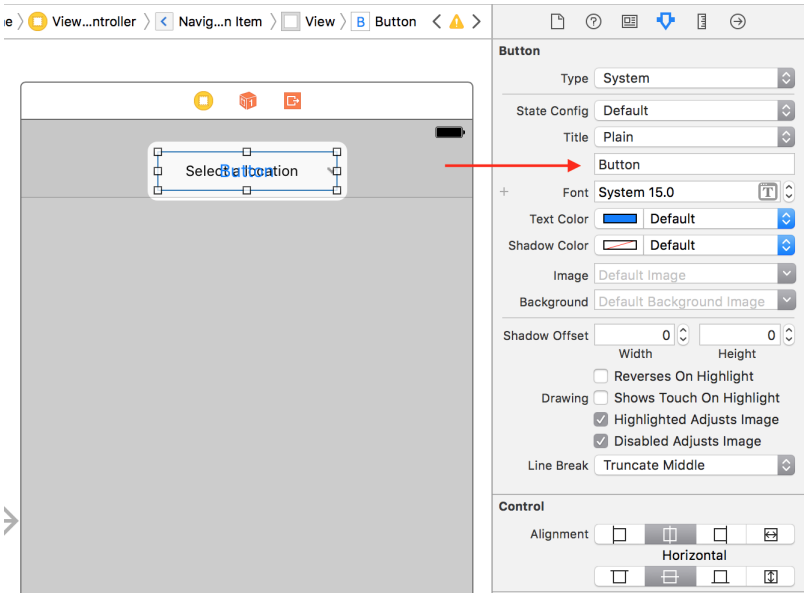
1. Type in the filter area of the Utilities panel—`uibutton`. You will notice only one item, **Button**.
2. Instead of dragging our button into our view, we are going to drag it into the Outline view instead, right below the **Stack View**:



3. With your button selected in the Outline view, select the Pin icon and enter the following values:
    - All values under **Add New Constraints** are set to 0
    - **Update Frames**: Items of New Constraints



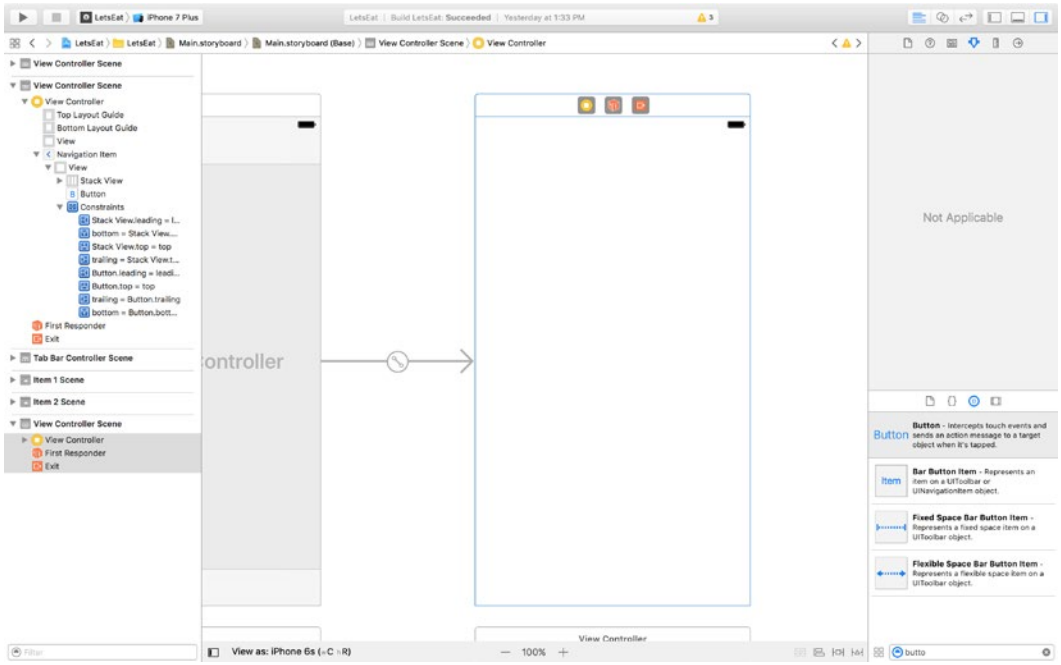
4. Click on **Add 4 Constraints**.
5. Next, in the Utilities panel, under the Attributes Inspector, delete the word **Button** under **Title**:



We are almost finished with this setup.

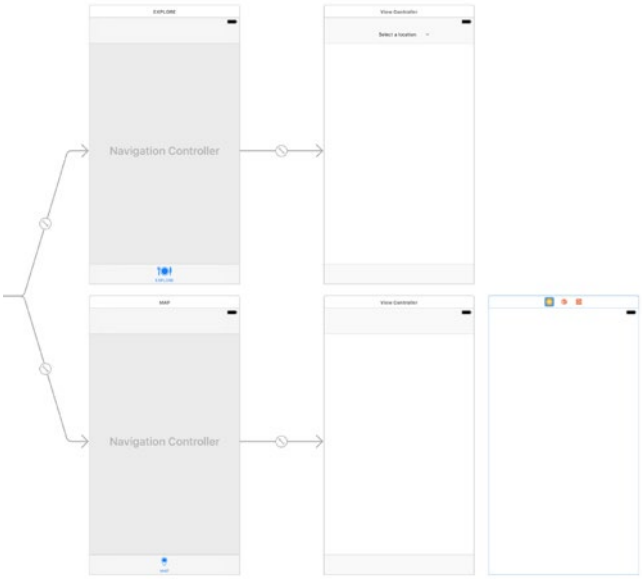
This custom title view will be used on both the Map and Explore tabs. Instead of doing all of the steps again, we are going to take a shortcut.

1. In the scene (not in the Outline view), select the **View Controller** with the custom title view in it.
2. Hit *cmd* + *D* to duplicate the View Controller:

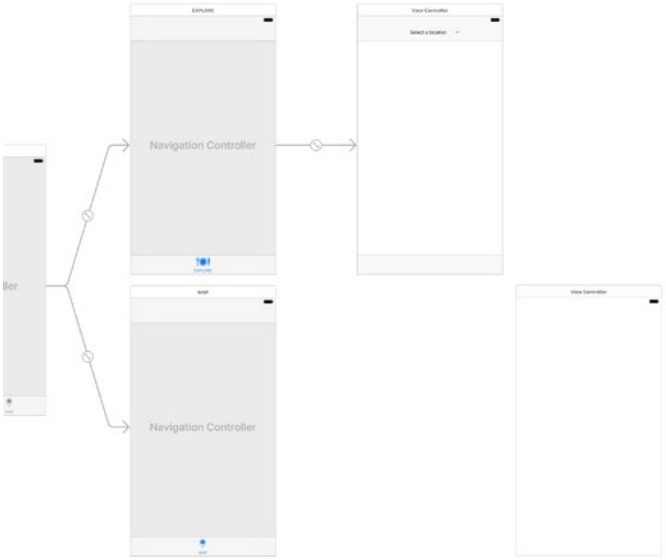


When you duplicate in storyboard, it will copy and paste it in the same spot (right on top of what you duplicated). You will see all three icons at the top of the View Controller highlighted. It appears that our custom title view is no longer there, but it actually still is there.

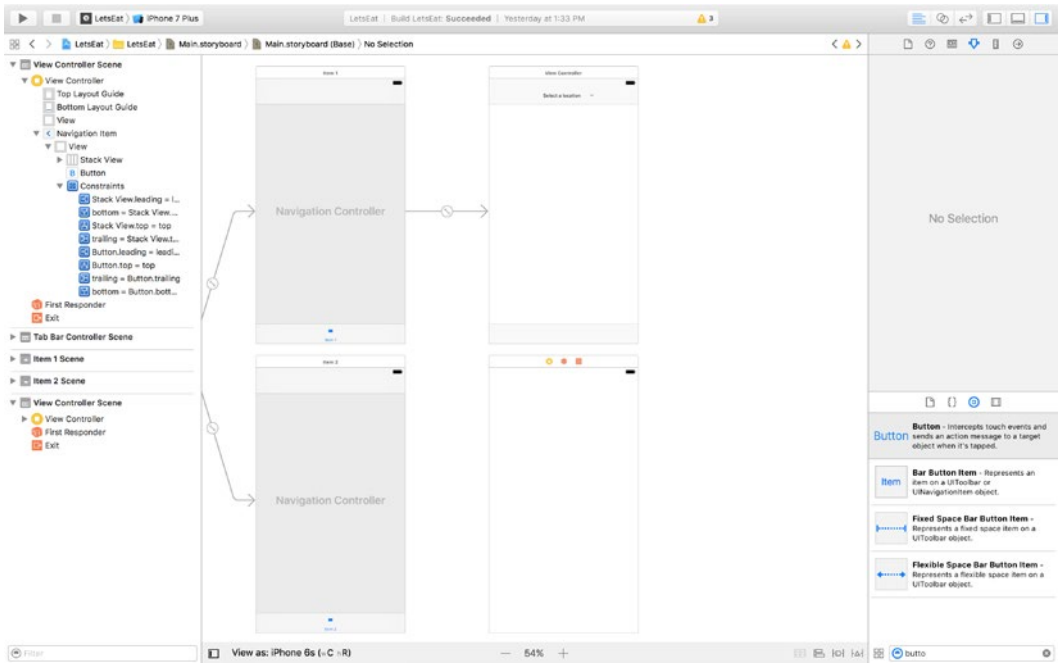
3. Now, drag the new **View Controller** next to where the empty Map View Controller is located:



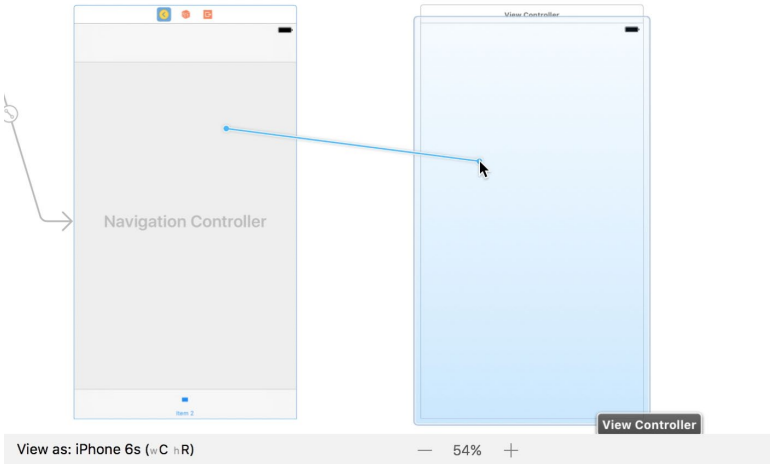
4. Select the empty Map View Controller and hit **delete**:



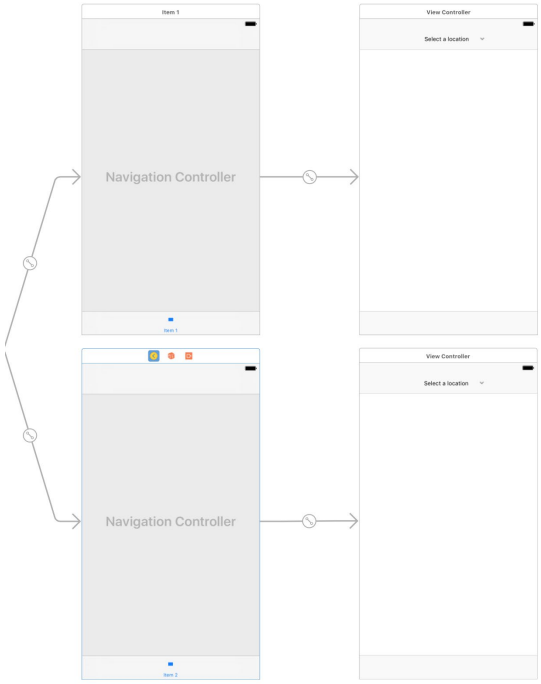
5. Now, move the new duplicated View Controller into the spot next to the Navigation Controller for Map and zoom out to about **54%**, in order to see the MapNavigation Controller and the newly duplicated View Controller next to each other:



6. CTL drag from the **Navigation Controller** to the **View Controller** and release:



7. In the pop-up menu, select Root View Controller.
8. You should now have the following:

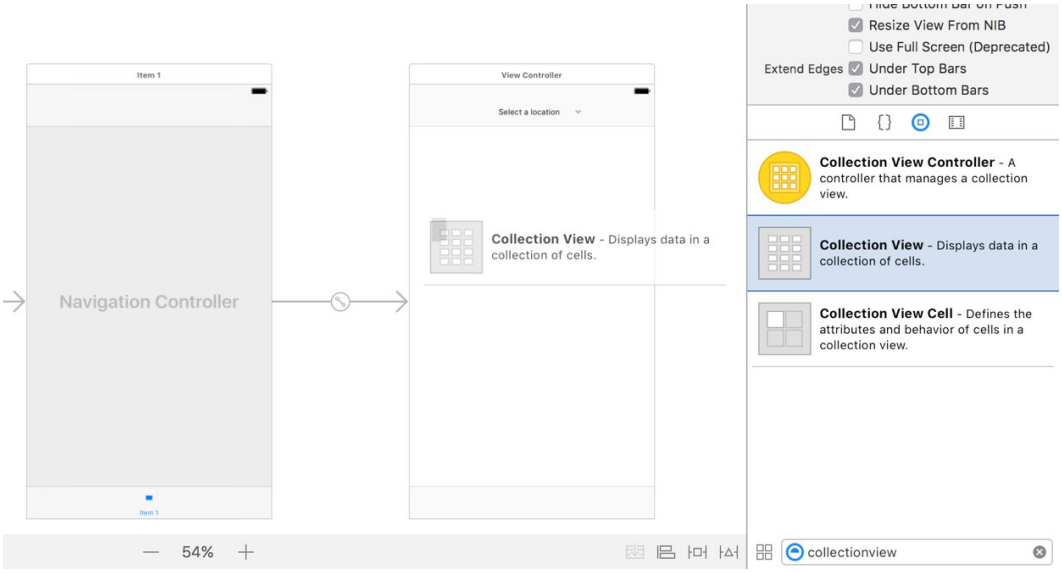


Collection View

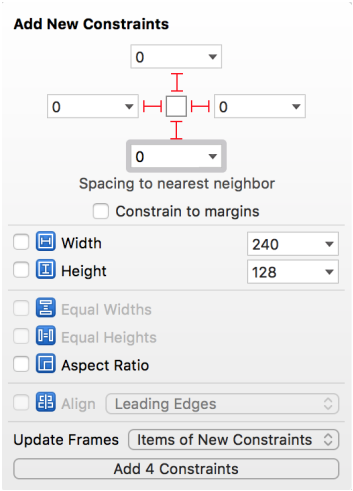
Earlier we discussed Collection View Controllers, and now we are going to use one in our Explore listing. Let's get started:

1. Select the `Main.storyboard` file, making sure that you are zoomed out and can see all of your scenes (around 54% should be good, depending on your screen resolution). In the Utilities panel, ensure that you have the **Object library** tab selected.
2. Next, in the filter field, you are going to type: `collectionview`.

3. Click on and drag **Collection View** and drop it onto the Explore View Controller.



4. After you drop it onto the scene, you will see small boxes around the entire **Collection View** component.
5. Select the Pin icon and enter the following values:
    - All values under **Add New Constraints** are set to `0`
    - **Update Frames**: **Items of New Constraints**



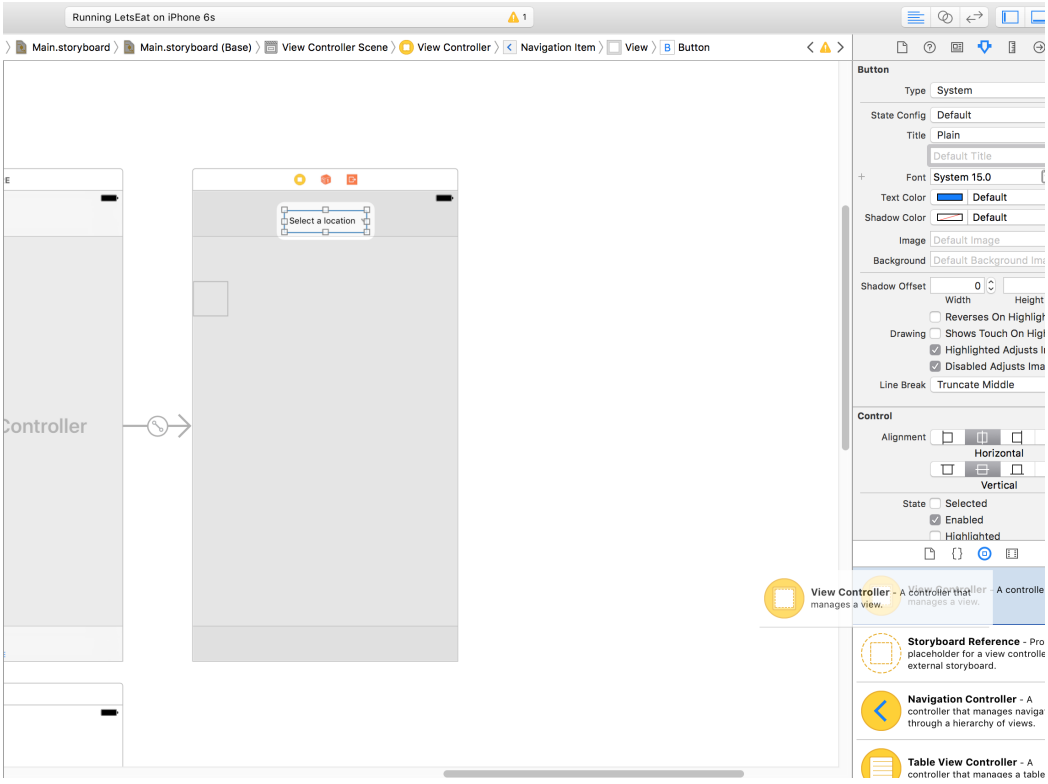
6. Click on **Add 4 Constraints**.

We now have our Collection View component set up for our Explore tab. Let's now go back to our custom title.

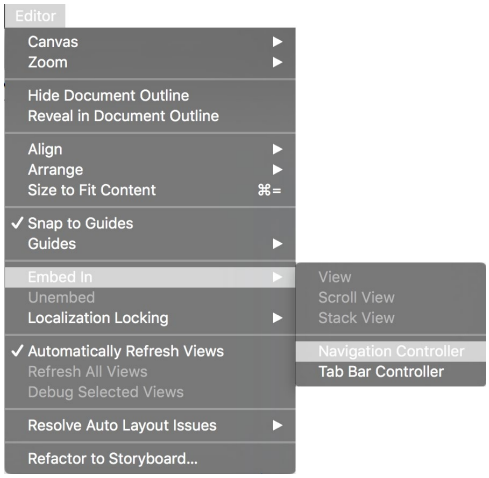
Adding a Modal

We previously added a button to our custom title, but we now need to tailor it so that when a user taps on it, it will display a list of locations. We do that by adding a modal, which, when presented, means that the list of locations will present over the content. Here's how we do it:

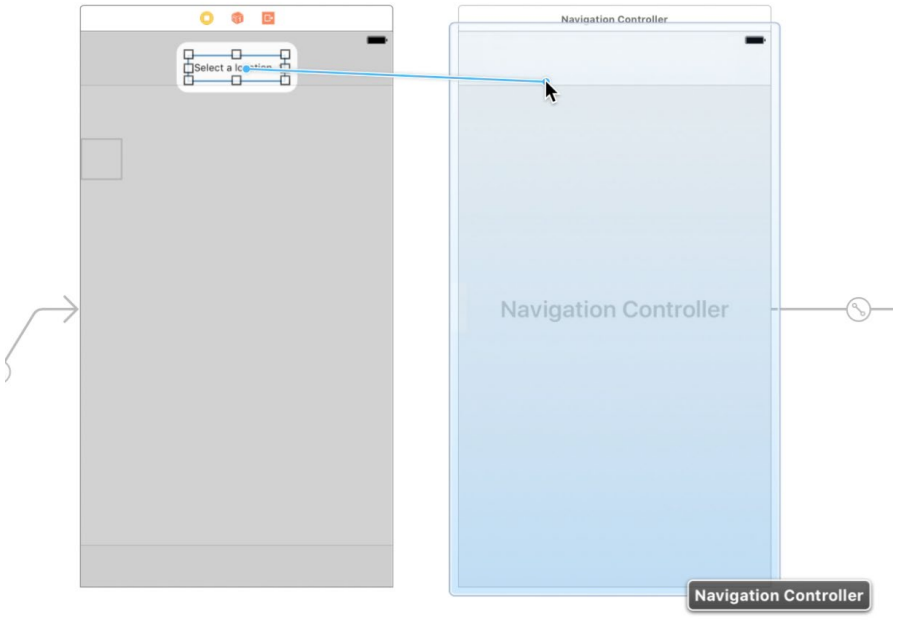
1. In the Utilities panel, under the Object library, after entering **view** into the filter field, select and drag out another View Controller to the scene:



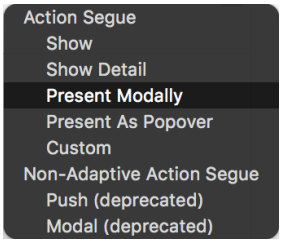
2. With the View Controller selected, navigate to **Editor** | **Embed In** | **Navigation Controller**:



3. Now, CTL drag from where it says **Select a location** in the View Controller under the Explore tab to the Navigation Controller that was just created (you can also do this within Outline view by CTL dragging from the Button to the new Navigation Controller you just created):



4. When you let go, you will be presented with the following menu, and you should select **Present Modally**:

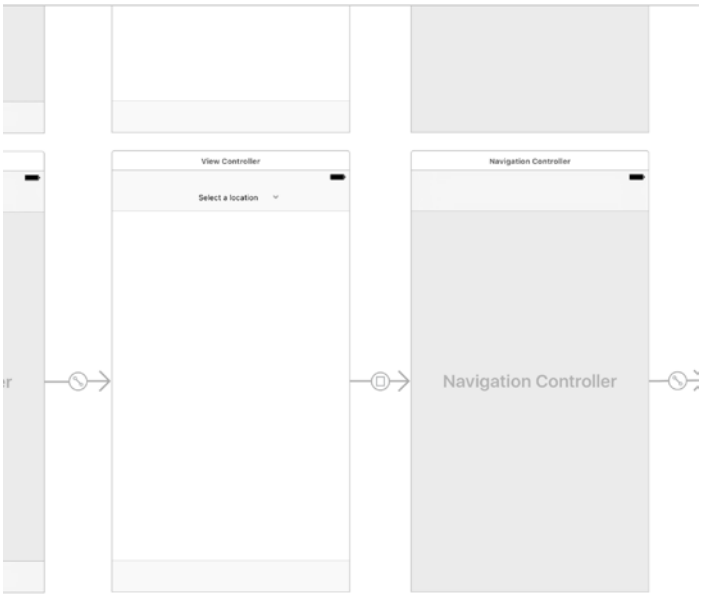


5. Now, let's run the project by hitting the **play** button (or use *cmd* + *R*).You will see that our custom title launches a modal.
6. Next, repeat steps 1 to 5 for the Map View Controller and custom title.

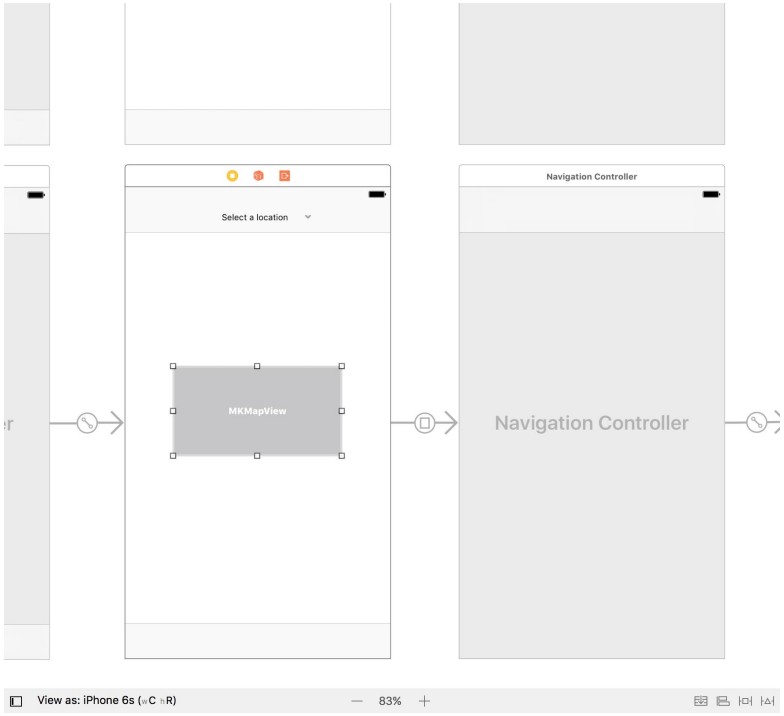
Currently, we cannot dismiss or close the modal at this time, but we will add this functionality later in this book. Let's get our Map tab set up.

Map Kit View

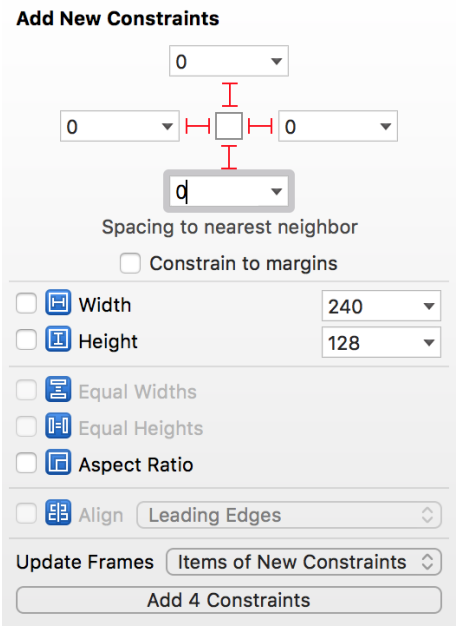
1. Select the `Main.storyboard` file, making sure that you are zoomed out and can see both our View Controller with the custom title view and our Navigation Controller for our Map tab (around 80% should be good):



2. In the Utilities panel, with the Object library tab selected type map in the filter field.
3. Drag and drop **Map Kit View** onto the Map View Controller. You will see small boxes around the entire Map Kit View component:



4. Next, select the Pin icon and enter the following values:
    - All values under **Add New Constraints** are set to 0
    - Constrain to Margins: Uncheck
    - **Update Frames**: **Items of New Constraints**



5. Click on **Add 4 Constraints**.

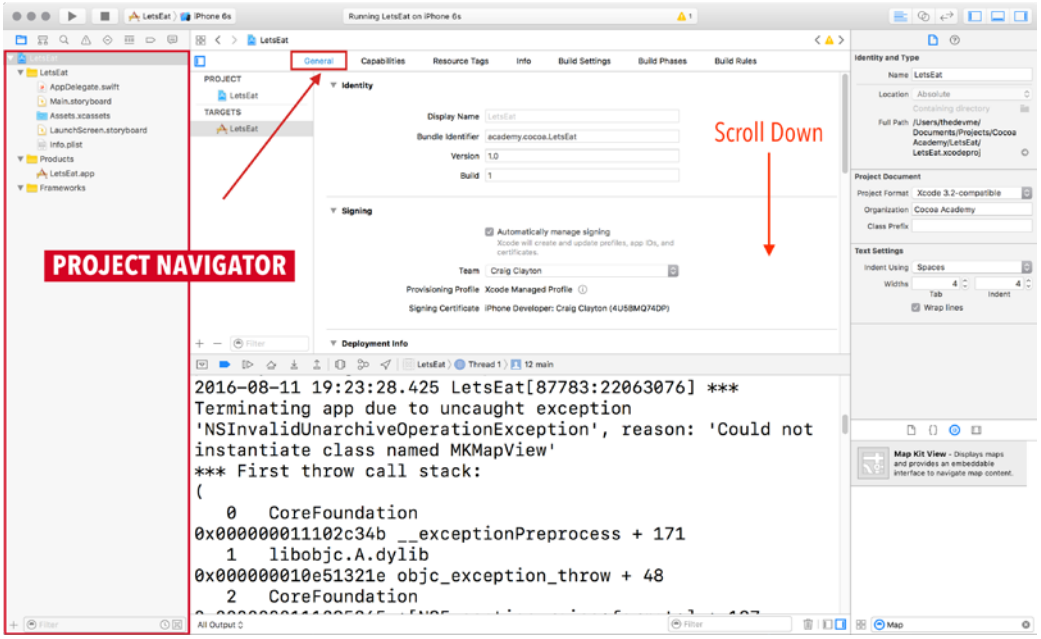
Let's run the project by hitting the **Play** button (or use *cmd* + *R*) and select the Map tab.

Crash!!! We have a problem, and, although it is a minor error, when you are not familiar with errors, they can be a bit overwhelming. Basically, our app does not know about the Map Kit framework (Map Kit framework gives us what we need to use Maps in our app). We need to go into our project settings and import the framework.

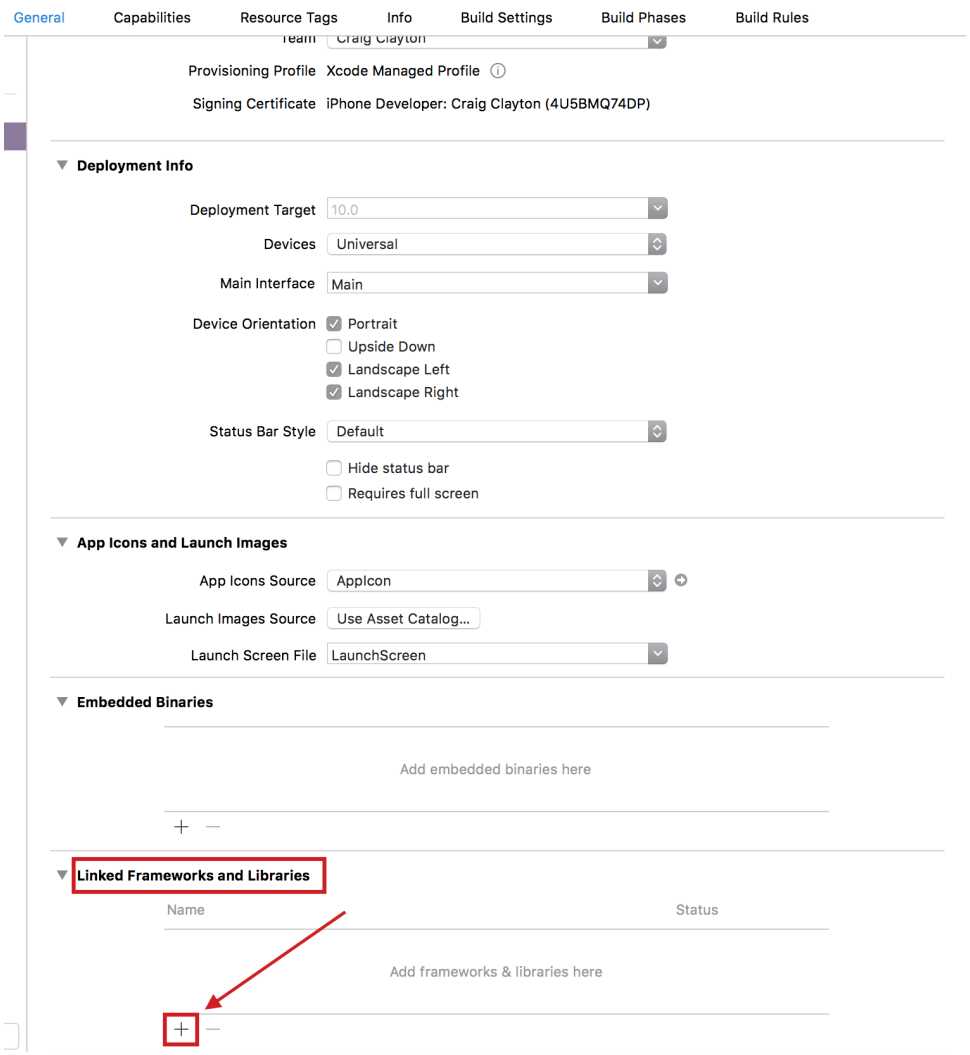
Fixing the Crash

Let's fix our crash:

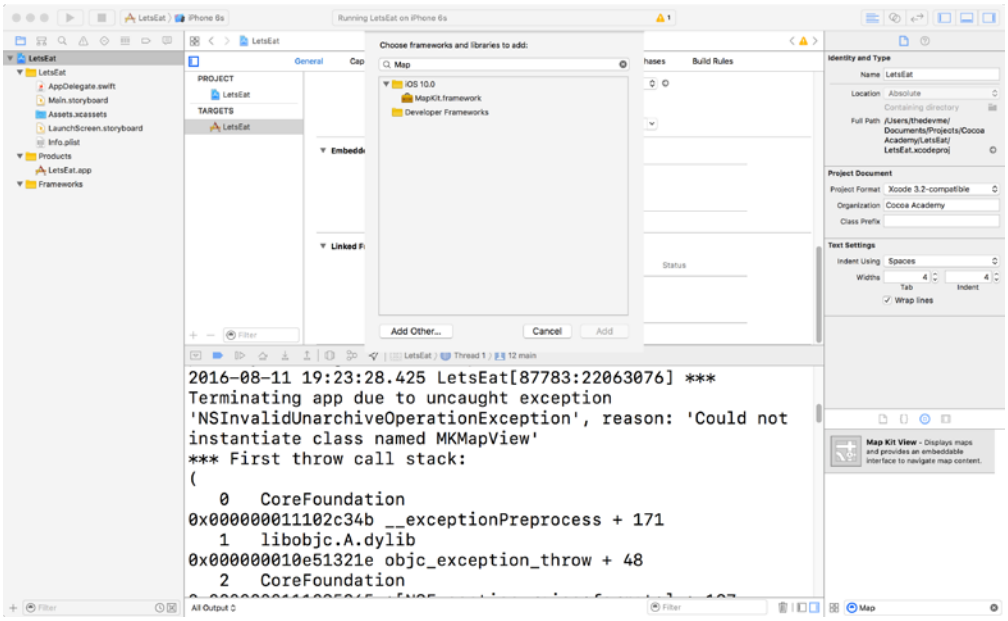
1. Go to your Navigator panel and select the Project navigator icon all the way to the left and then select your project and the **General** tab. Scroll down to the bottom until you see the **Linked Frameworks and Libraries** section:



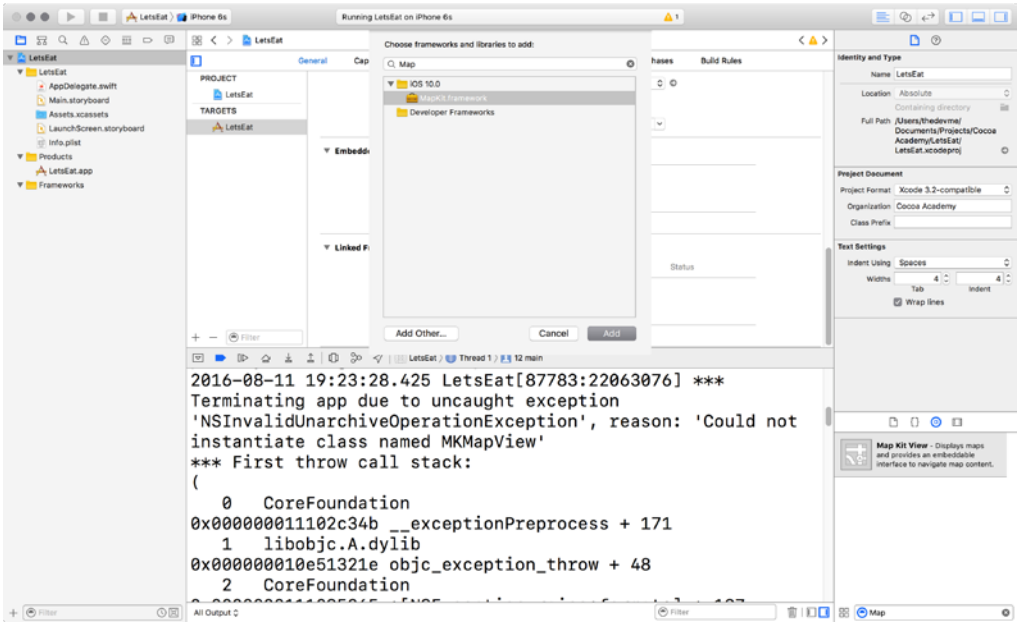
2. Once there, click on the **+** button:



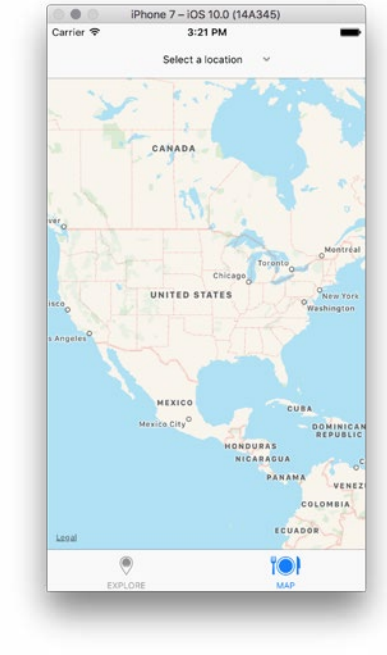
3. In the **Choose frameworks and libraries to add** pop-up screen, type `Map` into the search box:



4. Now select **MapKit.framework** and click on **Add**:



5. Let's run the project again by hitting the **Play** button (or *cmd* + *R*) and select the **Map** tab. You should now see a map:



We now have both tabs set up, but, as we progress through the book, we will add more scenes to the storyboard. Next, we are going to learn how to refactor our storyboard.

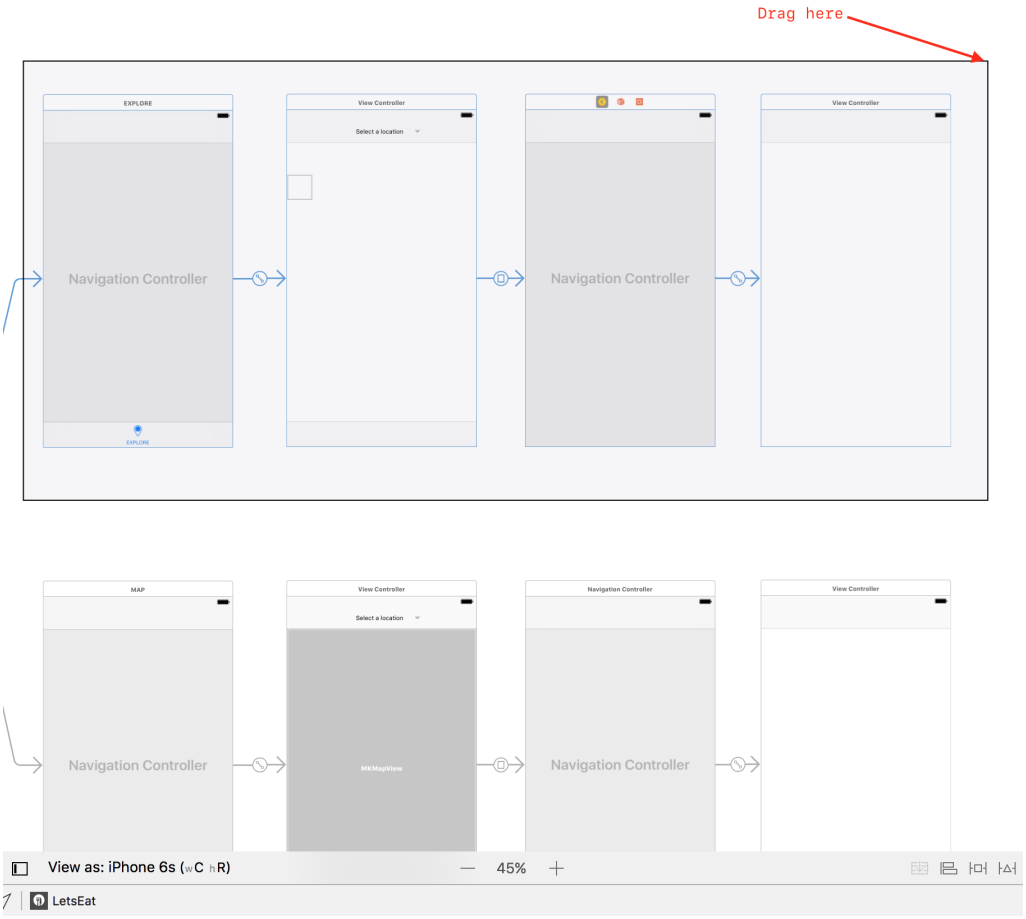
Refactoring the Storyboard

In programming, the term **refactor** means to take your existing code and improve on it without changing its behavior. We can apply refactoring to storyboards. We are going to refactor our storyboard so that each tab in our app will have its own storyboard file.

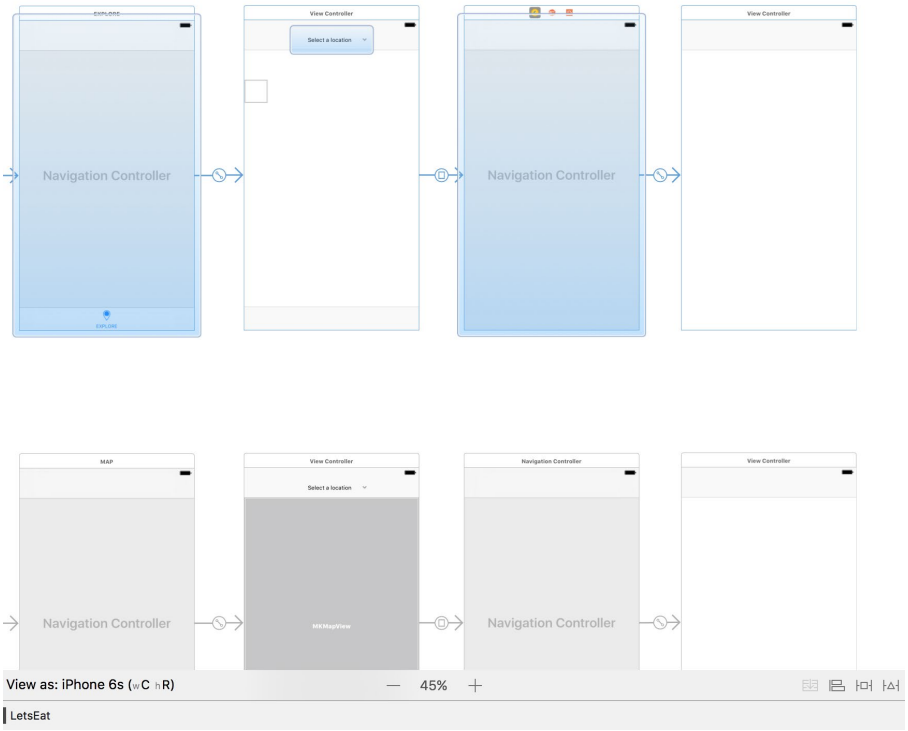
We will be using what is called **storyboard reference**, which is to add references between storyboards. A storyboard reference in one storyboard can point to an area in a different storyboard. This gives us a way to better organize our storyboards, rather than having one massive storyboard with which to work. Open your `Main.storyboard` file.

Creating a New Storyboard for the Explore Tab

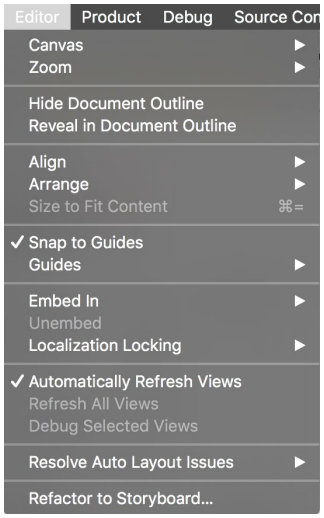
1. In `Main.storyboard`, you are going to click on and drag over all of the scenes that are in the **Explore** tab. Note that when you click, make sure you are not clicking any scene or View Controller.



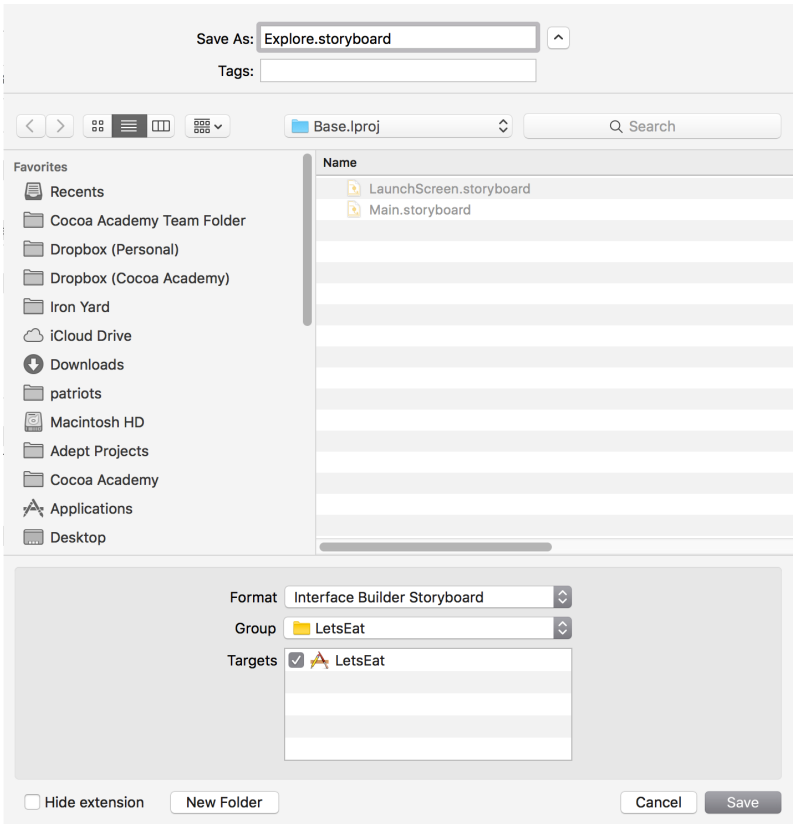
2. You should now see the following:



3. With the scenes selected, navigate to **Editor | Refactor to Storyboard**:



4. You will be prompted to name the storyboard. Name this `Explore.storyboard` and then hit **Save**:



5. Now, you will be in your new `Explore.storyboard` file; if you return to `Main.storyboard`, you will see this icon:



This icon is the storyboard reference, which will tell your app where to go to access the files.

Let's run the app, and you should see that the app is still working the same as before. Now, we will repeat these steps for the **Map** tab.

Creating a New Storyboard for the Map Tab

1. In `Main.storyboard`, click on and drag over all of the scenes that are in the **Map** tab.
2. With the scenes selected, navigate to **Editor | Refactor to Storyboard**.
3. You will be prompted to name the storyboard. Name this `Map.storyboard` and then hit **Save**.
4. You will automatically be in your `Map.storyboard` file; if you return to `Main.storyboard`, you will see this icon:



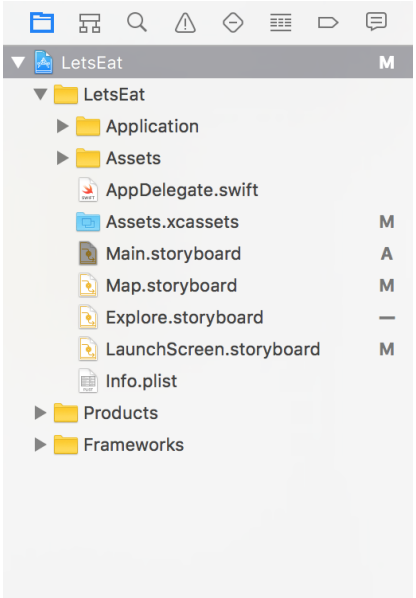
We are now done with refactoring our storyboard. From this point on, when we add scenes to our storyboards, we will go to either your `Explore.storyboard` or `Map.storyboard`. Now, let's set up our folder structure.

Folder Setup

Before we finish this chapter, we need to set up our project folders. It is good to have a structure for your app so that it is easier to find things, especially if someone else needs to update something in your app. It is really easy to create groups (folders) inside of Xcode by following these steps:

1. In the Navigator panel, ensure that you have the Project navigator icon selected.
2. Now, right-click on the yellow folder that says `LetsEat` and select **New Group**.
3. Name your new group, `Assets`.

4. Next, right-click the `LetsEat` folder again, and create another group called, `Application`. Your folder structure should now look like the following:



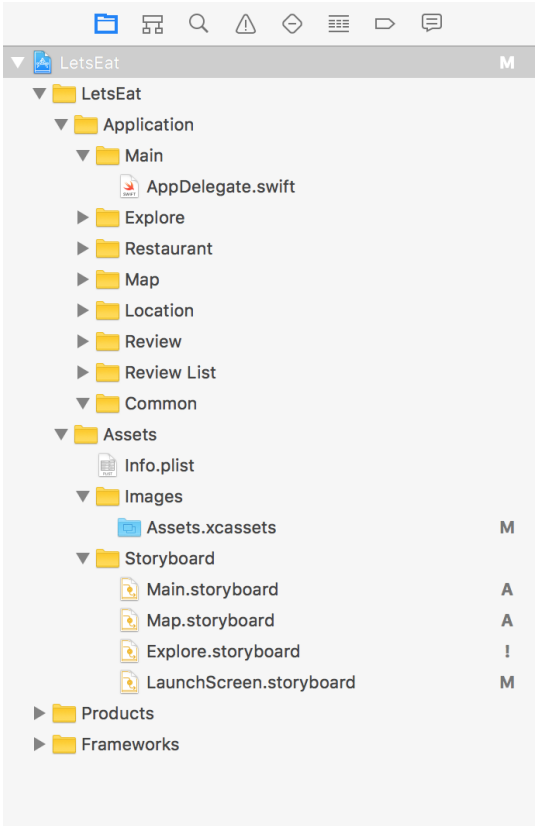
5. Now, right-click on the `Assets` group and create a group called, `Storyboard` and `Images`.
6. Do the same with the `Application` group by right-clicking on and then creating the following new groups called, `Common`, `Review List`, `Review`, `Location`, `Map`, `Restaurant`, `Explore`, and `Main`.

Organizing folders

Now, let's organize some of our folders:

1. First, we are going to drag and drop `Main.storyboard`, `LaunchScreen.storyboard`, `Explore.storyboard`, and `Map.storyboard` into the `Storyboard` folder under the `Assets` folder.
2. Next, select the `Info.plist` file and move it into the `Assets` folder.
3. Then, drag the `Assets.xcassets` folder and move it into the `Images` folder that is inside of the new `Assets` folder.

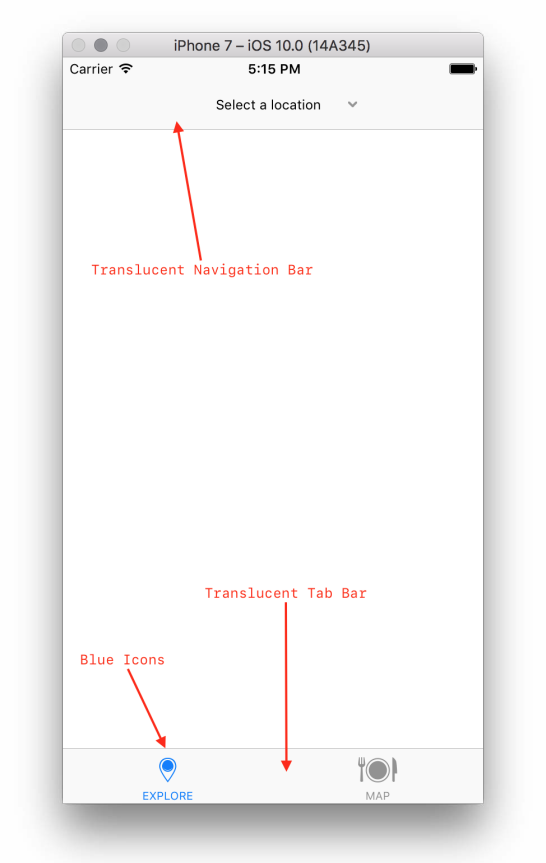
4. Finally, move the `AppDelegate.swift` file into the `Main` folder under the `Application` folder:



There is one more thing that we need to update before we move onto the next chapter, specifically establishing global settings throughout the app.

Setting up Global Settings

The global settings that we will set up will update our Navigation and Tab Bars as well as the icons in the Tab Bar. Currently, our app has translucent Navigation and Tab Bars. It would look much cleaner to make both of these white. Also, our Tab Bar icons are currently blue when selected and gray when they are not selected. We are going to change the blue to black when the icons are selected. Here is what we currently have:



In order to create global settings, one could go into each storyboard and update it, but that could be time consuming depending on how many storyboards an app has. The simplest way to create global settings across storyboards is to create the settings in one place and then make it so that everything updates at once. Well, we can do just that.

Breaking Down our App Delegate

Let's discuss the App Delegate's primary job and how it can help us. The **App Delegate** is the entry and exit point in your app. You can call it the nervous system. When you enter the app, you do so through the App Delegate. When you leave the app, the App Delegate is the last thing to know that you left.

Keeping your App Delegate clean is very important because it does so much. In our app, we will not use it much, but, as you progress into being an iOS developer, you will use it more, and so remembering to keep it clean will help you a lot. We will be working in the App Delegate here in order to add a method to globally update our app. A method is a function that lives inside a class or structure (also known as struct). In the following screenshot, notice the six methods next to **A** through **F** (comments were removed):

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

A   func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

        return true
    }

B   func applicationWillResignActive(_ application: UIApplication) {

    }

C   func applicationDidEnterBackground(_ application: UIApplication) {

    }

D   func applicationWillEnterForeground(_ application: UIApplication) {

    }

E   func applicationDidBecomeActive(_ application: UIApplication) {

    }

F   func applicationWillTerminate(_ application: UIApplication) {

    }
}
```

In order to establish global settings when the app is launched, we need to call it under the first method above, `application:didFinishLaunchingWithOptions` (hereinafter referred to as `Method A`. Let's begin by creating a new method called `setupDefaultColors`:

```
func setupDefaultColors() {
    UITabBar.appearance().tintColor = .black                                    a
    UITabBar.appearance().barTintColor = .white

b   UITabBarItem.appearance().setTitleTextAttributes([NSForegroundColorAttributeName: .darkGray], for: .normal)
    UITabBarItem.appearance().setTitleTextAttributes([NSForegroundColorAttributeName: .black], for: .selected)

    UINavigationBar.appearance().tintColor = .black                             c
    UINavigationBar.appearance().barTintColor = .white

    UITabBar.appearance().isTranslucent = false                                 d
    UINavigationBar.appearance().isTranslucent = false
}
```

Add all of this preceding code (the `setupDefaultColors` method) where indicated in the following screenshot:

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

        return true
    }

    func applicationWillResignActive(_ application: UIApplication) {

    }

    func applicationDidEnterBackground(_ application: UIApplication) {

    }

    func applicationWillEnterForeground(_ application: UIApplication) {

    }

    func applicationDidBecomeActive(_ application: UIApplication) {

    }

    func applicationWillTerminate(_ application: UIApplication) {

    }

    Add Code Here
}
```

Now, let's get a better understanding of each of the four parts of this code:

- **Part A**:

```
UITabBar.appearance().tintColor = .black
UITabBar.appearance().barTintColor = .white
```

  We added this code in order to update the Tab Bar so that the tint color is black, which means that anything inside of the Tab Bar (that is, buttons, labels, and so on) will be black, and the bar tint color is white.

- **Part B**:

```
UITabBarItem.appearance().setTitleTextAttributes([NSForegroundColo
rAttributeName:.darkGray], for:.normal)
UITabBarItem.appearance().setTitleTextAttributes([NSForegroundColo
rAttributeName: .black], for:.selected)
```

  This codes et the Tab Bar items to be gray when not selected and black when selected.

- **Part C**:

```
UINavigationBar.appearance().tintColor = .black
UINavigationBar.appearance().barTintColor = .white
```

  Here, we changed the Navigation Bar to have the tint color be black and the bar tint color be white.

- **Part D**:

```
UITabBar.appearance().isTranslucent = false
UINavigationBar.appearance().isTranslucent = false
```

  Finally, this code ensures that both the Tab Bar and Navigation Bar are not translucent.

Now that we have our method created, we need to call it in `Method A`, shown in a preceding screenshot. Add the following code after the opening curly brace and before `return true` in `Method A`:

```
setupDefaultColors()
```

Your App Delegate should now look like the following:

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

        setupDefaultColors()

        return true
    }

    func applicationWillResignActive(_ application: UIApplication) {

    }

    func applicationDidEnterBackground(_ application: UIApplication) {

    }

    func applicationWillEnterForeground(_ application: UIApplication) {

    }

    func applicationDidBecomeActive(_ application: UIApplication) {

    }

    func applicationWillTerminate(_ application: UIApplication) {

    }

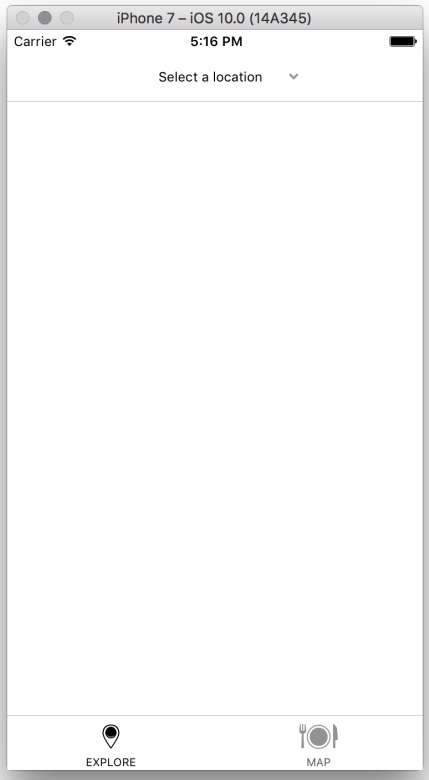

    func setupDefaultColors() {
        UITabBar.appearance().tintColor = .black
        UITabBar.appearance().barTintColor = .white

        UITabBarItem.appearance().setTitleTextAttributes([NSForegroundColorAttributeName: .darkGray], for: .normal)
        UITabBarItem.appearance().setTitleTextAttributes([NSForegroundColorAttributeName: .black], for: .selected)

        UINavigationBar.appearance().tintColor = .black
        UINavigationBar.appearance().barTintColor = .white

        UITabBar.appearance().isTranslucent = false
        UINavigationBar.appearance().isTranslucent = false
    }
}
```

Let's run the project by hitting the **Play** button (or use *cmd* + *R*). You should now see that the app has an overall appearance of white. All text and buttons in the Navigation Bar should be black; and the Tab Bar items should be black when selected and gray when not selected as shown in the following screenshot:



That will do it for this chapter, but we are in a good place for our Storyboard setup.

Summary

In this chapter, we covered a lot to get you more comfortable working in Storyboard and setting up our visual elements; we even did a little coding with Swift at the end of the chapter.

If you understand everything we did, then you will be good to proceed to the next chapter. If you feel somewhat overwhelmed, you may want to get the starter file for this chapter and go over the steps again. This is a really good idea, especially if you feel that you need a bit more understanding. When I first started, I found it helpful to repeat tutorials and try to recreate the steps without looking. This helped me see if I really understood what I was doing and where my weaknesses were. If you move onto the next chapter and are not fully comfortable, it may be harder for you to proceed.

In the next chapter, we will learn what Model View Controller is and how to work with it. We will also start working with collections and, from the next chapter on, we will be digging more into Swift 3.

7
Getting Started with the Grid

Typically, when I am working on a project, I like to set up my UI in Storyboard. Since I am a visual person, I prefer to start with the visuals and make sure that the app looks like the design. This helps me to identify the data structure and allows me to get familiar with the app. Therefore, I can focus my attention on the code.

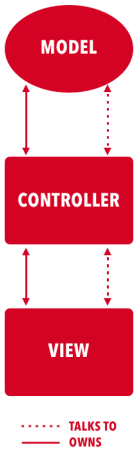
In the earlier chapters, we set up our app structure and developed a good understanding of the basics involved. As we progress through the book, we will add more UI elements. In this chapter, you will learn about app architecture and how to create it for our *Let's Eat* app.

We will cover the following in this chapter:

- Understanding the Model View Controller architecture
- Classes and Structures
- Controllers and Classes
- Model
- Collection View Cell
- Restaurant Listing

Understanding the Model View Controller architecture

Apple built iOS apps to use what is known as **Model View Controller** (**MVC**), which is an architectural pattern that describes a way to structure the code in your app. In layman's terms, this just means breaking up our app into three distinct camps, Model, View, and Controller. Here is a diagram of MVC to better understand:



Let's discuss each camp:

Model

The **Model** camp is responsible for an app's data and business logic. The Model's only job is to handle representations of data, storage of data, and the operations performed on the data.

View

The **View** camp is responsible for all the things that you see on the screen. The View handles presenting and formatting the data that results from the user's interactions.

Controller

The **Controller** camp is the liaison or coordinator between the other two camps. The Controller handles a lot of setup and connections to the View. The Controller also interprets user interactions. Since the Controller is between both the View and the Model, they should know nothing about each other.

In summary, the Controller will take user interactions and either respond back to the View or pass it onto the Model. When the Model is done with a task, it will pass it back to the Controller and then the Controller will talk with the View.

Getting familiar with the setup

For beginners, the MVC architecture can make you uncertain about where things should go. As we progress through the book, you will learn where to put things and why. So, you need not worry about where things should be placed, as we will work through this process together step by step.

As your project grows, the MVC architecture will place a lot of responsibility on the Controller. Therefore, in this book, we will tweak the MVC pattern in order to not put so much pressure on the Controller.

Before we continue with our coding, we need to discuss classes and structures.

Classes and Structures

Classes and structures (also known as structs) are files that contain properties and methods. You use these properties and methods to add functionality. You have been working with structs since *Chapter 1*, *Getting Familiar with Xcode*. Strings, Ints, Bools, Arrays, Dictionaries, and Sets are all structs.

Earlier in the book, we created functions. As noted in *Chapter 6*, *Setting Up UI*, a method is a function that lives inside a class or struct.

Classes and structs are very similar; however, Swift handles each of them a bit differently. To truly get a better understanding of how classes and structs work, we will create a new Playground project. This gives us the ability to really learn how to create custom classes and structs as well as gain an understanding of each of their positives and negatives.

You can keep your project open, but let's jump back into Playgrounds. Since we have Xcode open, go to **File** | **New** |**Playground**.

In the options screen that appears, name your new Playground, `FunctionsStructs`, and make sure that your Platform is set to iOS. Hit **Next** and then **Create**. Now, let's delete everything inside of your new Playground and toggle on the Debug panel, using either the toggle button or *CMD + Shift + Y*.

In your empty Playground, add the following:

```
class Cat {
}

struct Dog {
}
```

We just created our first class and struct and defined two new custom data types (known as **Swift types**), `Cat` and `Dog`. Since we have not yet given the class or struct a property (such as a name) or created an instance of either `Cat` or `Dog`, you will see nothing in the Results or Debug panels.



When you create classes and structs, you must always make sure that you start with a capital letter. In addition, you must have different names for your class and for your struct. Otherwise, you will get an error. Even though one is a class and the other is a struct, each of them needs a distinct name.

Now, we need to give each of our `Cat` class and our `Dog` struct names. Therefore, let's give them both a property, called `name`:

```
class Cat {
var name:String?
}

struct Dog {
var name:String?
}
```



If you cannot set a property when it is created, then you must set that property to an Optional using the question mark (?). This protects against your code trying to access the name if you never set it.

With both `Cat` and `Dog` now having a property called `name`, let's create an instance of each of them:

```
let yellowCat = Cat()
yellowCat.name = "Whiskers"
print(yellowCat.name)

var yellowDog = Dog()
yellowDog.name = "Bruno"
print(yellowDog.name)
```

So far, everything on the surface looks the same. We created both a `Cat` and a `Dog` and gave them each names. However, let's say `Whiskers` runs away and, a few weeks later, finds a home with a new family, who decides to change his name to `Smokey`. After `Whiskers` runs away, `Bruno` becomes lonely and decides to find him, but also gets lost. `Bruno` finds a new home as well, and this new family decides to name him, `Max`.

In Playgrounds, we will create a new constant called `yellowStrayCat` and set it equal to `yellowCat`, since it is still `Whiskers`. But, we will change the name of `yellowStrayCat` to `Smokey`. We also will also create a new constant called `yellowStrayDog`, setting it equal to `yellowDog` and naming it `Max`.

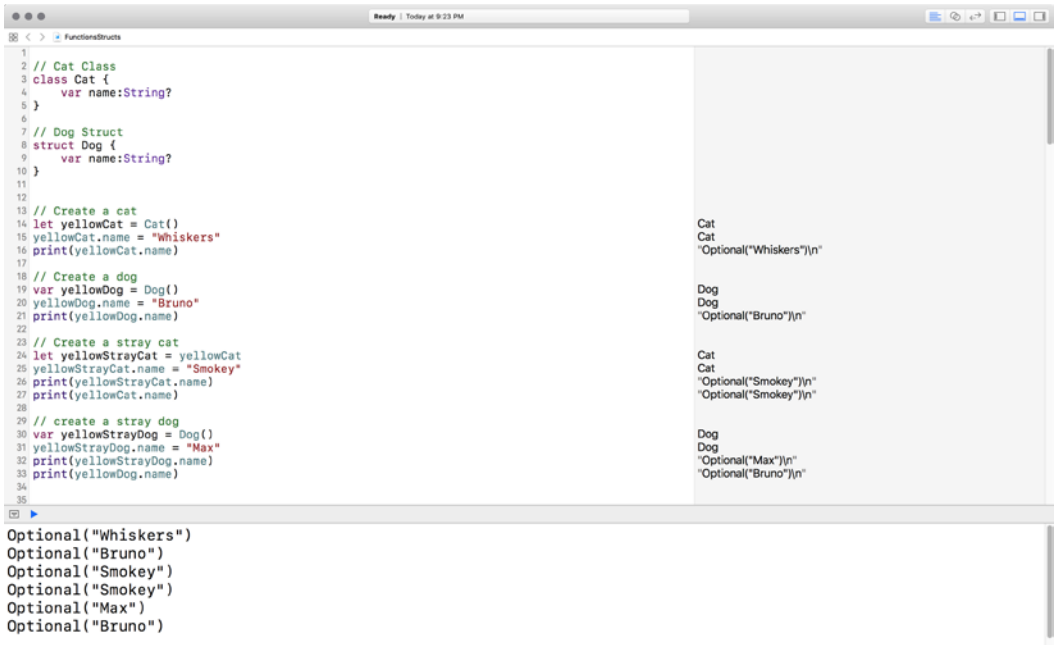
```
let yellowStrayCat = yellowCat
yellowStrayCat.name = "Smokey"
print(yellowStrayCat.name)

var yellowStrayDog = yellowDog
yellowStrayDog.name = "Max"
print(yellowStrayDog.name)
```

Our Results panel shows that the names of `yellowStrayCat` and `yellowStrayDog`, respectively, are now `Smokey` and `Max`. So, everything seems to be the same between our class and our struct, right? No, they are not the same. Let's print the name of `yellowCat` underneath the line where we have `print(yellowStrayCat.name)`. In addition, let's do the same for the name of `yellowDog` underneath where we have `print(yellowStrayDog.name)`. Your code should now look as follows:

```
let yellowStrayCat = yellowCat
yellowStrayCat.name = "Smokey"
print(yellowStrayCat.name)
print(yellowCat.name)
```

```
var yellowStrayDog = yellowDog
yellowStrayDog.name = "Max"
print(yellowStrayDog.name)
print(yellowDog.name)
```



In our Results panel, as shown in the preceding screenshot, you will notice an unexpected result. The `yellowCat`, `Whiskers`, now has the name `Smokey`, but the `yellowDog` is still `Bruno`. Without getting too technical, when you use a class and copy it like we did, it refers back to the original instance created. This is known as a reference type. Whereas, when structs get copied, they create a new instance and the original is not affected. This is known as a value type.

Before we move on, let's look at one more difference between the two. In programming, we have what is called inheritance, which means that we can create another object with default values, and objects can inherit from those default values. Let's create an `Animal` class that will be known as a base class immediately below our `Cat` class:

```
class Animal {
var age:Int?
}
```

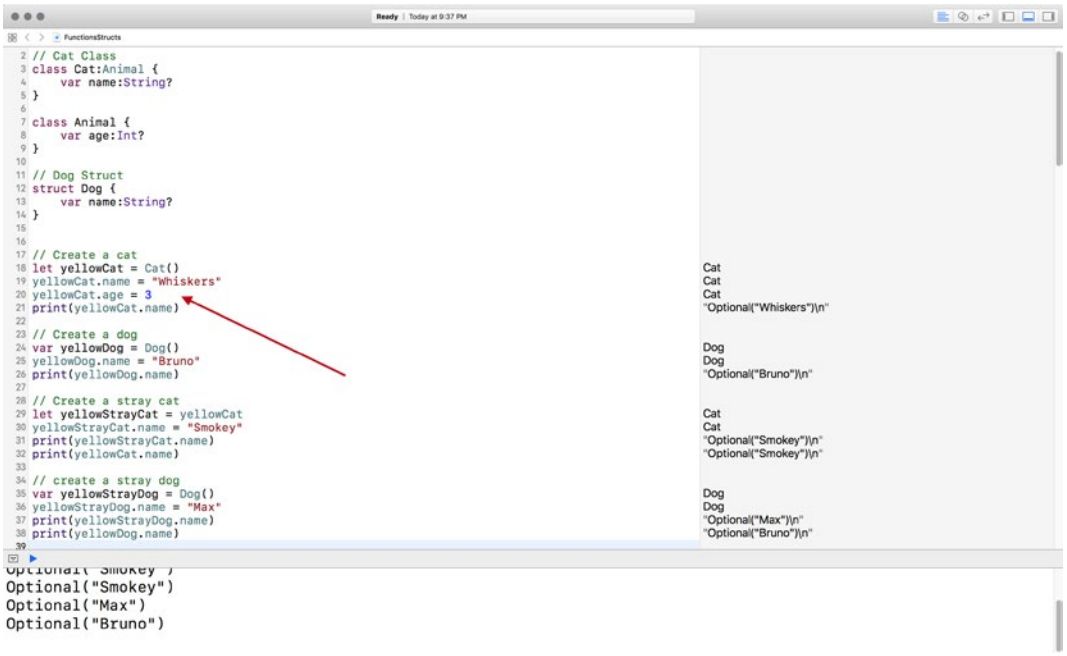
Now, let's update our `Cat` class to inherit from it as shown in the following code:

```
class Cat:Animal {
   ...
}
```

Note that we are only updating what goes directly after `Cat`. The rest of the class in the curly braces will stay the same.

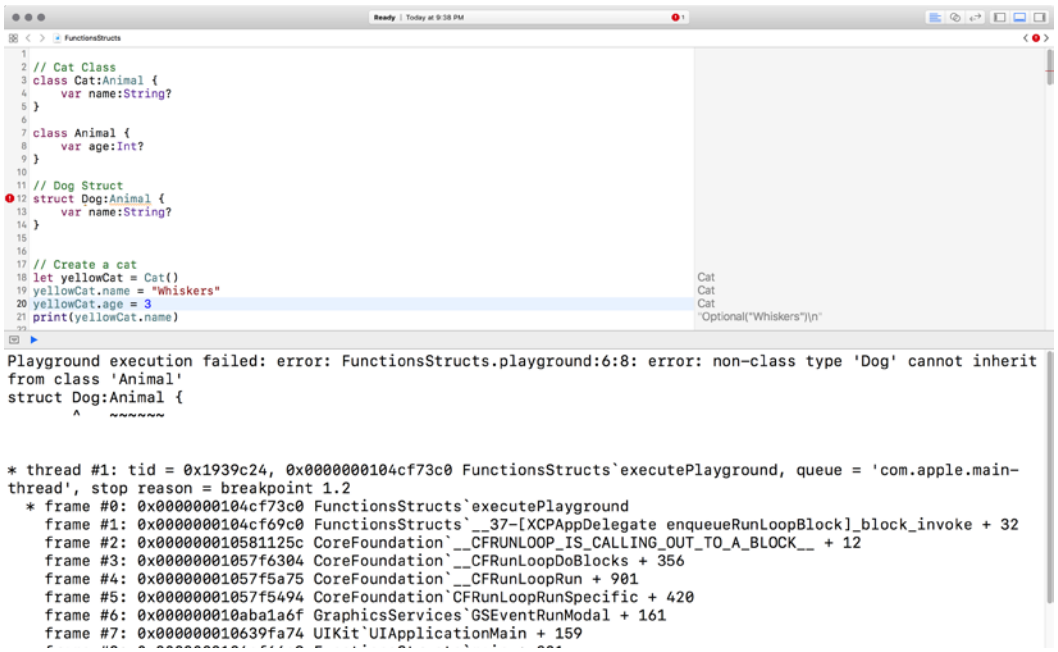
Since our class now inherits from `Animal`, we should have a new property called `age`. Underneath where we name `yellowCat`, `Whiskers` and above our print statement, add:

```
yellowCat.age = 3
```



So, as expected, we were able to give `Whiskers` an age. Let's do the same for our `Dog` struct by adding `Animal` directly after `Dog`:

```
struct Dog:Animal {
var name:String?
}
```



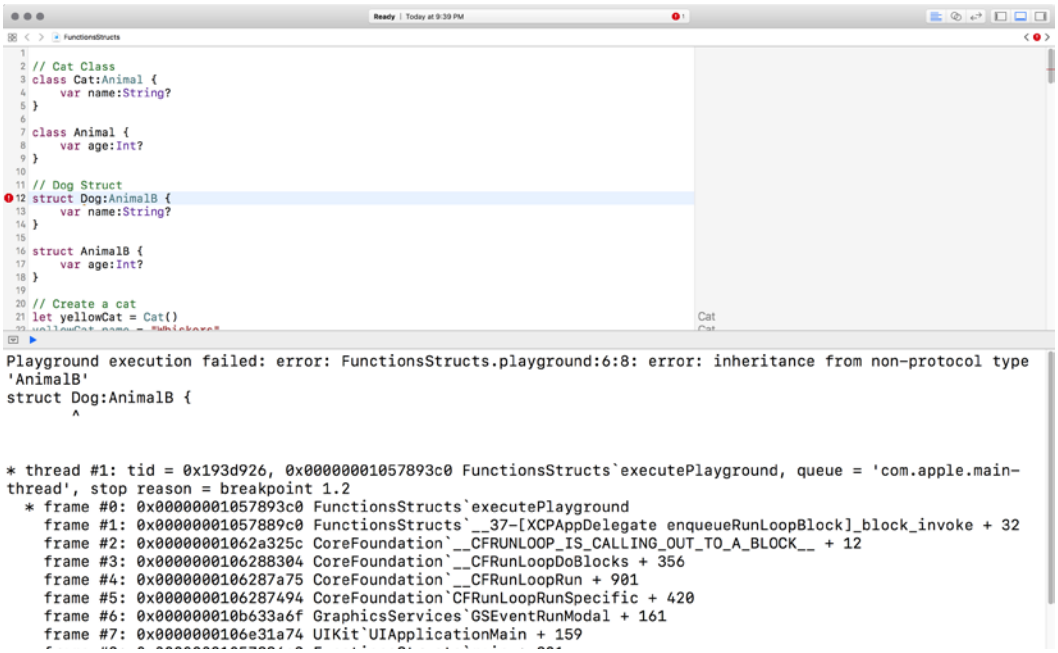
A red error will display that informs you that `Non-class type 'Dog' cannot inherit from class Animal`. Therefore, we need to create a struct called, `AnimalB`, since we cannot have the same name:

```
struct AnimalB {
var age:Int?
}
```

Update your `Dog` struct from `Animal` to `AnimalB`:

```
struct Dog:AnimalB {
var name:String?
}
```

Now, you will see an error called, `Inheritance from non-protocol type 'AnimalB`, which means that our struct cannot inherit from another struct.



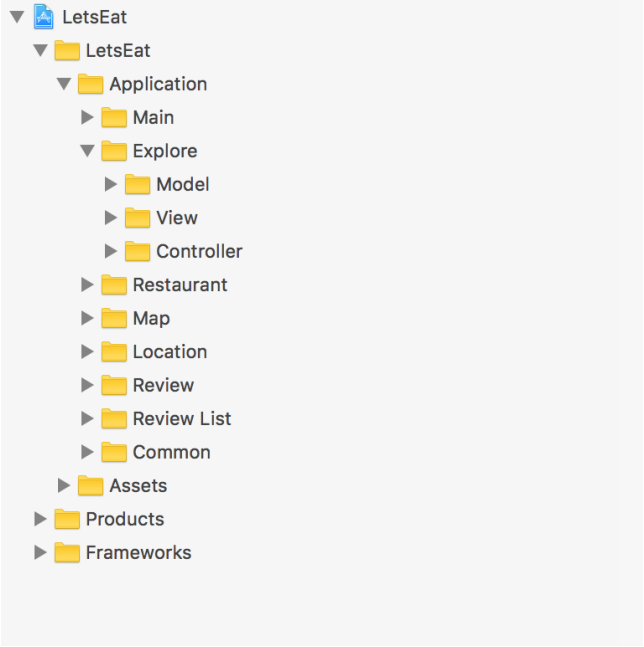
Inheritance is something that you can do with classes, but not with structs. Therefore, this is another difference between classes and structs. There are a couple of other advanced technical differences, but, for our purposes, the two described here are sufficient.

Controllers and Classes

In order to work with `UIViewController`, `UICollectionViewController`, and `UITableViewController`, you will need to create a class file for each of these elements. Each file will handle all of the logic and interactions that the controller sends and receives. Along with interactions, the class file is responsible for receiving data. You will understand this more when we delve deeper into creating each of these class files.

The first thing we should do is to create three folders inside of the `Explore` folder we created earlier by following these steps:

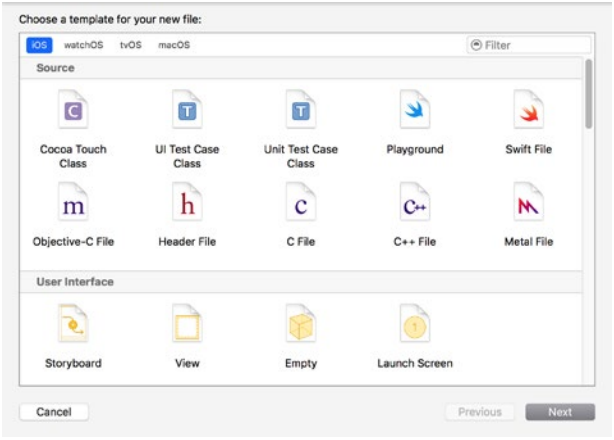
1. In the Navigator panel, with the Project navigator icon selected, right-click on the `Explore` folder and select **New Group** and name it `Controller`.
2. Repeat these steps two more times, adding `View` and `Model` into the `Explore` folder, which should now look as follows:



Creating our Controller

Let's create our first file inside of the `Controller` folder:

1. Right-click on the `Controller` folder and select **New File**.
2. Inside of the **Choose a template for your new file** screen, select **iOS** at the top and then **Cocoa Touch Class**. Then, hit **Next**:



3. You will now see an options screen. Add the following:

   New File:

   - **Class**: `ExploreViewController`
   - **Subclass**: `UIViewController`
   - **Also create XIB**: `Unchecked`
   - **Language**: `Swift`

4. After hitting **Next**, you will be asked to create this file. Select **Create**, and then your file should look like mine:

```
//
//  ExploreViewController.swift
//  LetsEat
//
//  Created by Craig Clayton on 8/28/16.
//  Copyright © 2016 Cocoa Academy. All rights reserved.
//

import UIKit

class ExploreViewController: 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.
    }

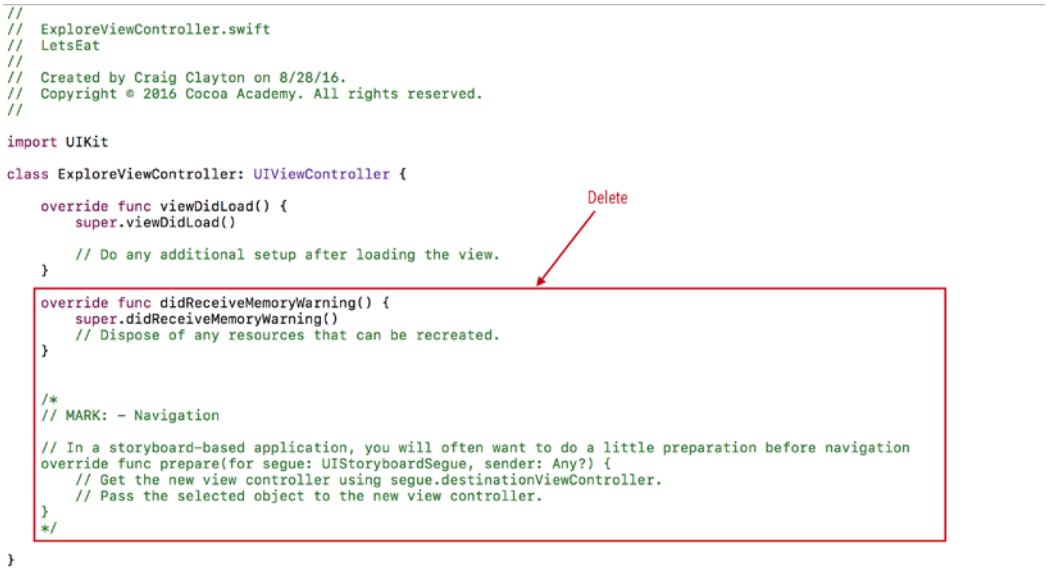

    /*
    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destinationViewController.
        // Pass the selected object to the new view controller.
    }
    */

}
```

Let's review this `ExploreViewController` class file and also do some maintenance inside of the file. We created this file to use with our `UIViewController` that we created when we initially set up our UI.

Note that there are three methods in this file—`viewDidLoad()`, `didReceiveMemoryWarning()`, and `prepare()` (which is commented out). Let's delete both `didReceiveMemoryWarning()` and `prepare()` as we do not need them at this time:




```
//
//  ExploreViewController.swift
//  LetsEat
//
//  Created by Craig Clayton on 8/28/16.
//  Copyright © 2016 Cocoa Academy. All rights reserved.
//

import UIKit

class ExploreViewController: 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.
    }


    /*
    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destinationViewController.
        // Pass the selected object to the new view controller.
    }
    */

}
```

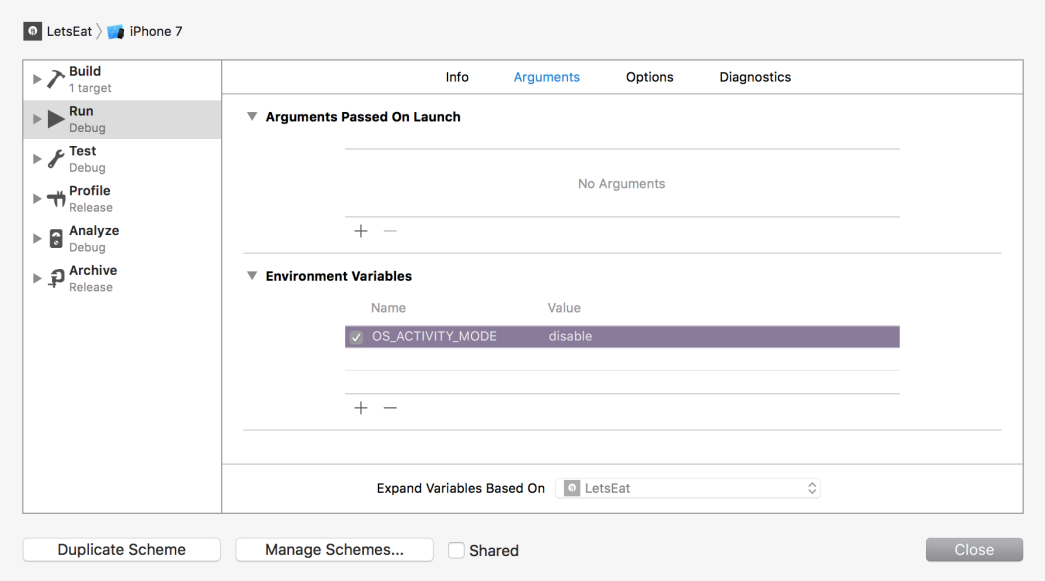
What remains is `viewDidLoad()`, which method is called only once during the life of the View Controller. Let's see what this means by updating `viewDidLoad()` to be the following:

```
func viewDidLoad() {
super.viewDidLoad()
  print("Hello Explore View Controller")
}
```

Now, run the project by hitting the **play** button (or use *CMD* + *R*). Although we expect to see the above print statement in our Debug panel, nothing appears in that panel. This is a result of our not having linked our file, `ExploreViewController`, to our `UIViewController` in Storyboard.

If you are currently running Xcode 8.0, you will see certain log statements in the Debug panel. These log statements are actually coming from Apple. In order to remove these statements, you need to do the following:

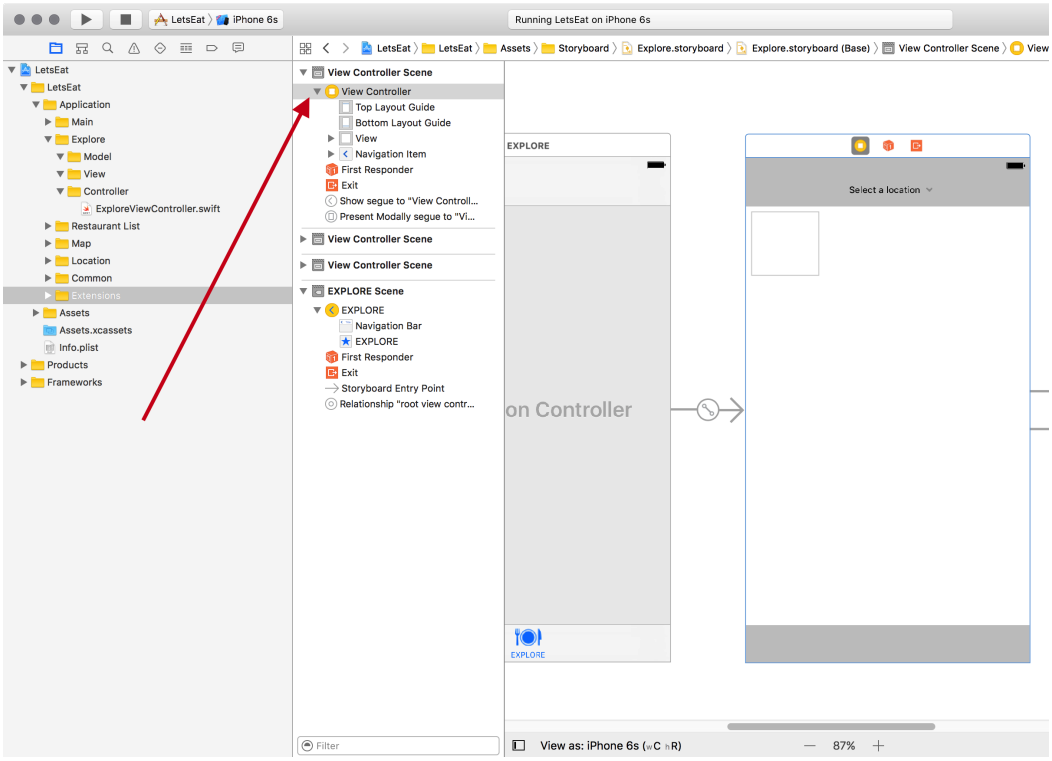
1. Select your active scheme in the Toolbar.
2. In the drop-down menu, select **Edit Scheme**.
3. In this window, under the **Arguments** tab, find the **Environment Variables** section and click on the **+** button.
4. Set Name to **OS_ACTIVITY_MODE** and set **Value** to **disable**:



5. Build and run the project again, and now you will have an empty Debug panel.

Let's now link our file, `ExploreViewController`, to our `UIViewController` in Storyboard:

1. Select `Explore.storyboard`.
2. Then, select the `UIViewController` with the custom title view that we created earlier:



3. Now, in the Utility Panel, select the Identity Inspector, which is the third icon from the left.
4. Under Custom Class, in the Class drop-down menu, select `ExploreViewController`and hit *Enter*.

Let's run the project again by hitting the **Play** button (or use *cmd* + *R*), and you should now only see `Hello Explore View Controller` inside of the Debug panel.

Now that we have our `ExploreViewController` hooked up with our `UIViewController` in Storyboard, let's start working with our `UICollectionView`.

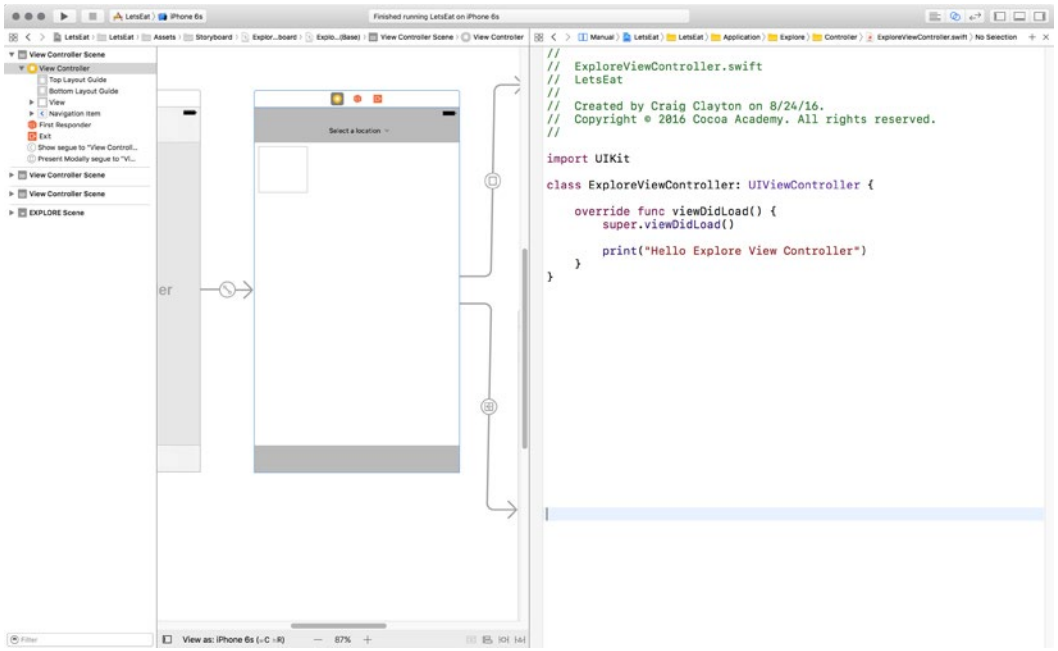
Understanding Collection View Controllers and Collection View Cells

As noted earlier in the book, Collection View Controllers allow us to display our data in a grid. The individual items inside of Collection Views are called cells, and these cells are what actually show the data. This data can be anything from an image to text or both an image and text. You have complete control over what your Collection View cell can display. Our Collection View Controller is responsible for making sure the correct number of cells is displayed.

Let's now connect our file, `ExploreViewController`, with our `UICollectionView` in Storyboard. In order to do this, we will use the Assistant editor (or split screen), which we access by doing the following:

1. Open `Explore.storyboard`
2. Close the Navigator panel using the hide Navigator toggle or *cmd + 0*.
3. Close the Utilities panel by hitting the Utilities toggle or use *cmd + ALT + 0*.
4. Next, select the Assistant editor or use *cmd + ALT + ENTER*.

You should now see `Explore.storyboard` on the left side and `ExploreViewController.swift` on the right:



1. Add the following inside of your `ExploreViewController.swift` file on the line after:

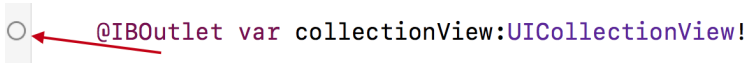
```
class ExploreViewController: UIViewController {":

@IBOutlet var collectionView:UICollectionView!
```

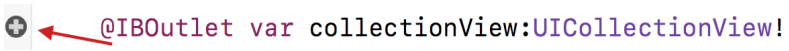


`IBOutlet` is a way to a connect to UI element. We have a Collection View on our `UIViewController`; and, now, we are creating a variable that will allow us to hook into it.

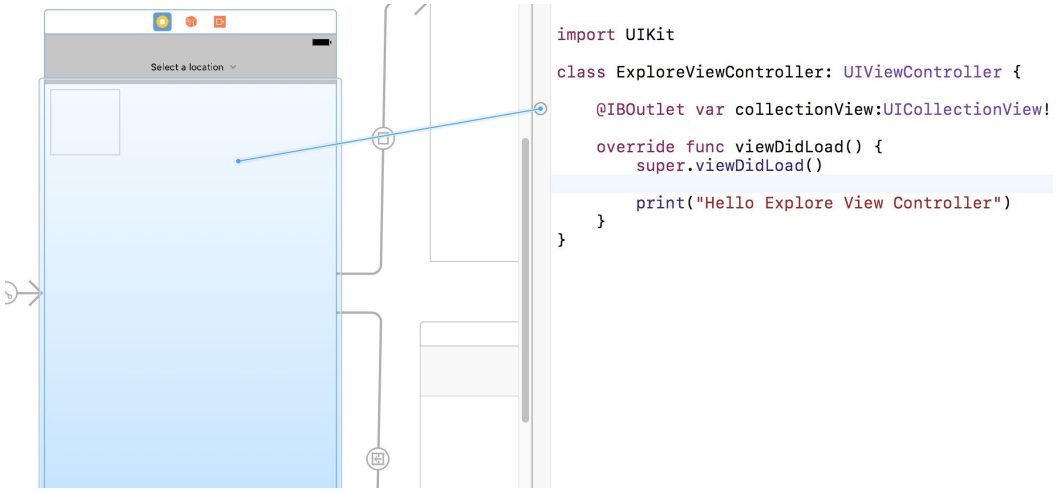
2. After you create the variable, you will see a small circle to the left of the variable:



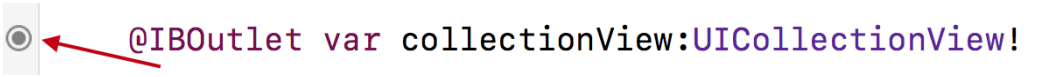
3. When you hover over it, you will see a plus button appear inside of the circle:



4. Click on it and drag this to your Collection View inside of your `UIViewController`:

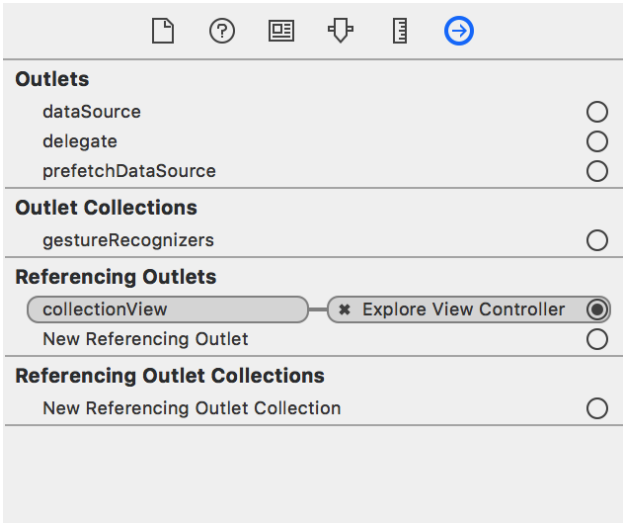


5. Once you release, you will see the circle become filled:



6. Now, select the Standard editor or use *cmd* + *ENTER*. This will bring you to just the `Explore.storyboard`. You will see a small box inside of your Collection view. This is your Collection View cell.
7. Open the Utilities panel by hitting the Utilities toggle (or use *cmd* + *ALT*/*ALT* + *0*), select the Size Inspector, and update the following:
    - **Size**: `Custom`
    - **Width**: `75`
    - **Height**: `75`
8. Then, select the Attributes Inspector in the Utilities panel, and update the following:
    - **Identifier**: `exploreCell`
    - **Background**: `Light Gray Color`
9. In your scene, select your Collection View. Then, in your Utilities panel select the Connections Inspector, which is the last icon on the right.

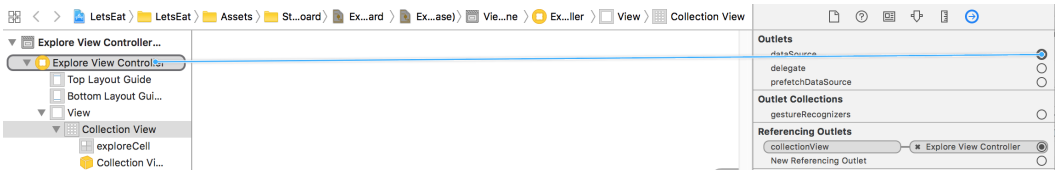
Under the **Outlets** section, you will see two empty circles, `dataSource` and `delegate`:



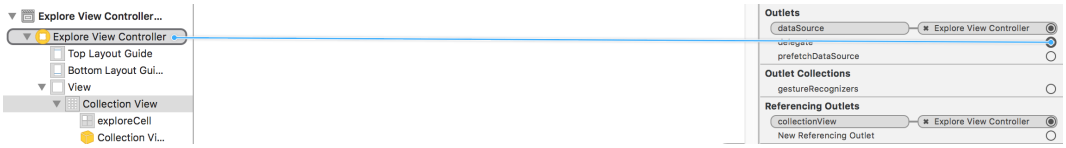


The `dataSource` property is what is used to supply the data for our Collection View, so we need to pass whatever data we have to this property. On the other hand, the `delegate` property, which supplies the behavior, does not require us to supply anything as it receives interactions that happen within our Collection View.

1. Click on and drag the `dataSource` property to the Explore View Controller in your Outline view.



2. Click on and drag the `delegate` property to the Explore View Controller in your Outline view:



3. Now, select the Size Inspector and update the following values under **CellSize**:
    - **Width**: `75`
    - **Height**: `75`

Getting Data into Collection View

It is time to display something inside of our Collection View:

1. Use *cmd* + *SHIFT* + *O*, which will open a small window called "**Open Quickly**." Inside of the window, type `ExploreView` and hit **Enter** to select the `ExploreViewController.swift` file.
2. Update our class definition from class `ExploreViewController:UIViewController` to the following:

```
class ExploreViewController:UIViewController,
UICollectionViewDataSource
```

Understanding the Data Source

Whenever we use Collection View in order to get data, we must conform to a protocol. A protocol is a set of methods to which we have access and can either be required or optional. For Collection Views, we are required to implement three methods to get data into a Collection View. So let's add the following three required methods (each beginning with `func`) after the closing curly brace of `viewDidLoad()`:

```
import UIKit

class ExploreViewController: UIViewController, UICollectionViewDataSource {

    @IBOutlet var collectionView:UICollectionView!

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return 1
    }

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section:
        Int) -> Int {
        return 20
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath:
        IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "exploreCell",
            for: indexPath)

        return cell
    }

}
```


Let's breakdown the code to better understand what we are doing:

- **Part A**:

  `numberOfSections(in collectionView: UICollectionView) ->Int`

  Our first method tells our Collection View how many different sections we want to display.

- **Part B**: `return 1`

  Here, we are telling our Collection View that we only want 1 section.

- **Part B**:

  ```
  func collectionView(_ collectionView: UICollectionView,
  numberOfItemsInSection section: Int) ->Int
  ```

  Our second method tells our Collection View how many different items we are going to display inside of the section we set up.

- **Part C**: `return 20`

  We are telling our Collection View that we want to display 20 items.

- **Part D**:

  ```
  func collectionView(_ collectionView: UICollectionView,
  cellForItemAtindexPath: IndexPath) ->UICollectionViewCell
  ```

  Our third and final method gets called for every item we need. Therefore, in our case, it will get called 20 times.

- **Part E**:

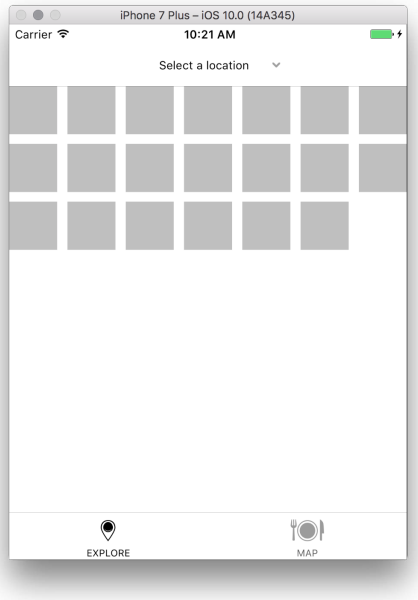
  ```
  let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
  "exploreCell", for: indexPath)
  ```

  Here, we are creating a cell every time this method E is called. The identifier, `exploreCell`, is the name we gave it in Storyboard, so this is the cell that is grabbed and used inside of our Collection View.

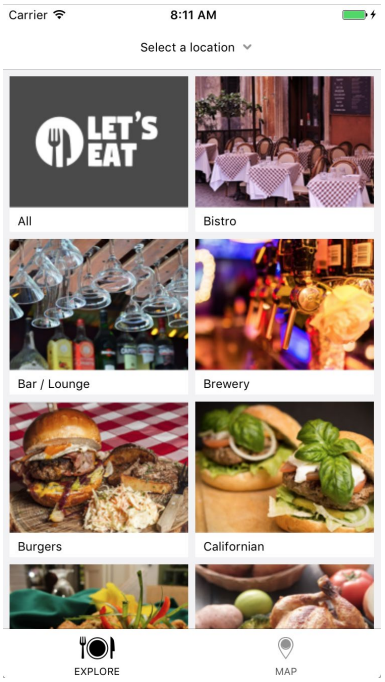
- **Part F**: `return cell`

  Finally, after each time we create a new cell, we give the cell back to the Collection View in order to display that cell.

Let's build and run the project by hitting the **Play** button (or use *cmd* + *R*) in order to see what happens:



We now have a grid displaying inside of our Collection View, but we need to clean it up in order to match our design of two items per row with certain size cells:



Updating the Grid

In order to update our grid, we need to take the following steps:

1. Use *cmd* + *SHIFT* + *O*, and in the **Open Quickly** window, type `Explore.storyboard` and then hit **Enter**.
2. Select the Collection View, and then, in the Utilities panel, select the Size Inspector. Update the following values, based on the simulator that you are currently using. These values may need to be changed so that your grid has two columns of cells, so feel free to alter the values:

   iPhone 7:

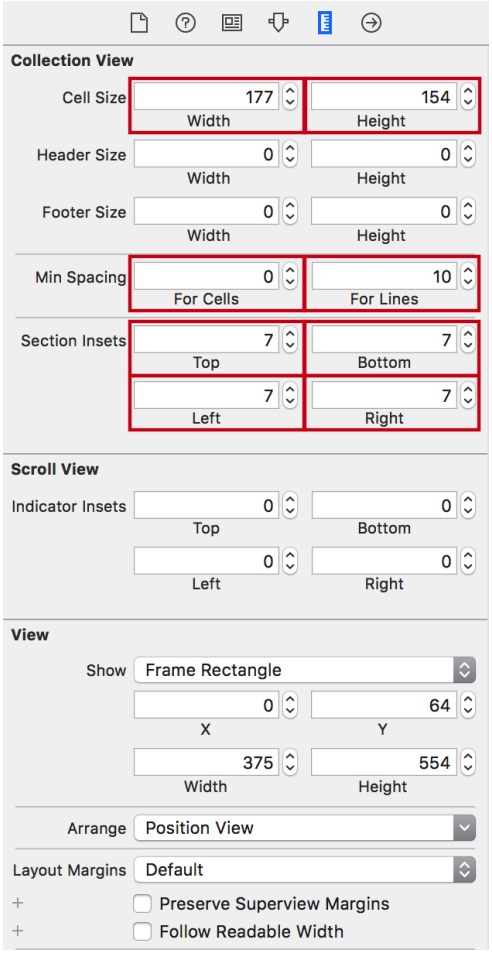
Cell Size	Width: 177	Height: 154		
Min Spacing	For Cells: 0	For Lines: 10		
Section Insets	Top: 7	Bottom: 7	Left: 7	Right: 7

iPhone 7 Plus:

Cell Size	Width: 196	Height: 154		
Min Spacing	For Cells: 0	For Lines: 10		
Section Insets	Top: 7	Bottom: 7	Left: 7	Right: 7

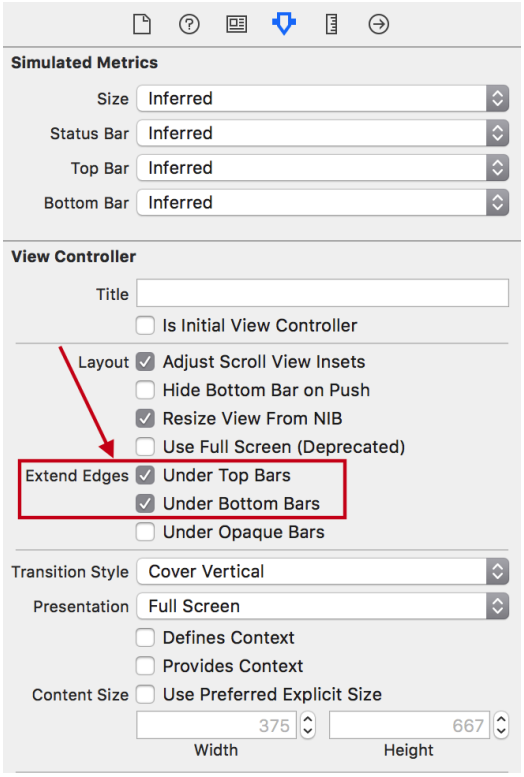
iPhone 4/iPhone SE/iPhone 5/iPhone 5s:

Cell Size	Width: 150	Height: 154		
Min Spacing	For Cells: 0	For Lines: 10		
Section Insets	Top: 7	Bottom: 7	Left: 7	Right: 7



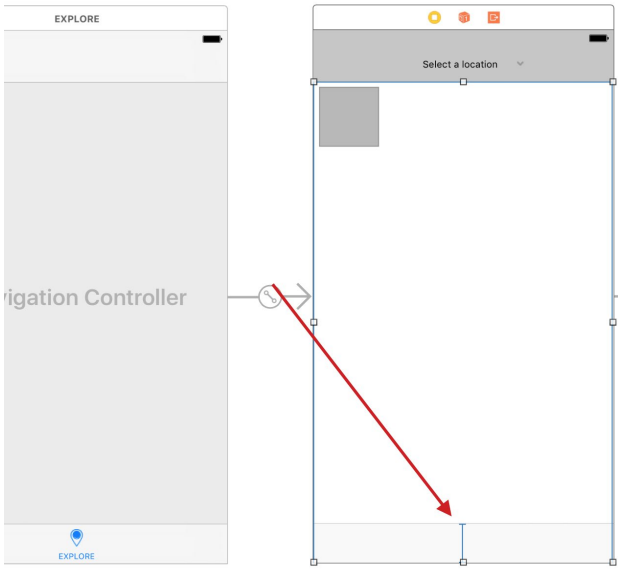
For now, as we just did, we will use Storyboard settings to get our cells set up. Later in the book, we will make this dynamic so that our widths and heights adjust with code.

3. Next, select the Explore View Controller in the Outline view.
4. In the Utilities panel, select the Attributes Inspector; and you will see, under View Controller, **Extend Edges**:



5. Uncheck both **Under Top Bars** and **Under Bottom Bars**.

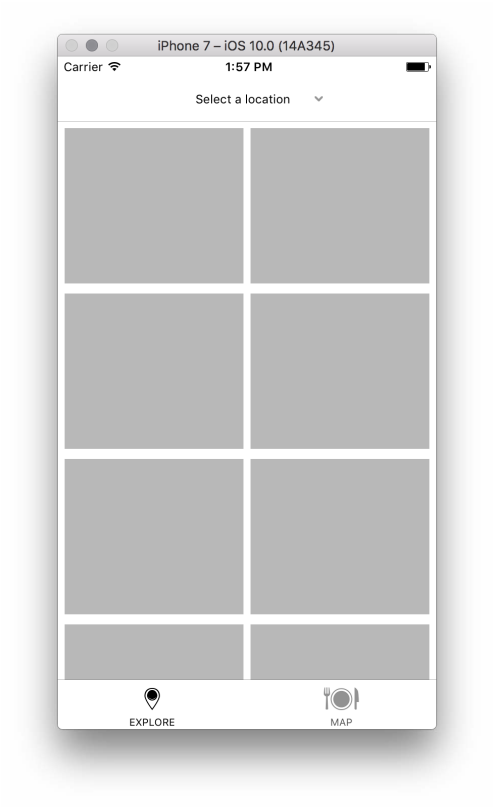
You will notice that our Collection View's bottom constraint is now incorrect:



Therefore, we need to update our Auto Layout:

1. Inside of the Storyboard, select the Collection View and then select the Pin icon.
2. Update the bottom constraint from -49 to 0 by entering the following:
    - **Bottom**: `0`
    - **Update Frames**: `Items of New Constraints`
3. Click on **Add 1 Constraint**.

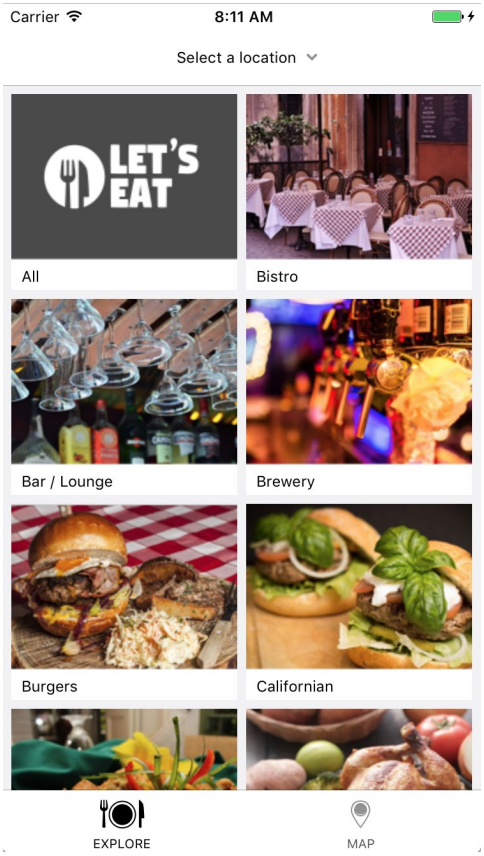
Let's build and run the project by hitting the **Play** button (or use *cmd*+ *R*) in order to see our changes:



Now that we have our Collection View set up as desired, it is time to get data into our cells. Using the MVC pattern, we need to work on our Model next.

Model

Typically, when developing your Model, you want to look at the data associated with your View. Let's look at our app design again:



The items (`UICollectionViewCell`) that are being displayed in the grid are each supported by some data. Looking at the design, we see that each item needs an image and a name (cuisine). Therefore, we need to create a Model called `ExploreItem` with two properties, specifically `image` and `name`.

In the `Model` camp, we have three files that we will create—`ExploreData. plist`, `ExploreItem.swift`, and `ExploreDataManager.swift`.

ExploreData.plist

The first file, `ExploreData.plist`, has already been created for you and can be found in your project files for this chapter. This file contains all the data we need in order to have a list of cuisines. Drag this file into your `Model` folder in the Navigator panel.

In the file, there is an array of dictionary items. Each item will have a cuisine name and image for that particular cuisine. Let's take a look at the first few elements of this file:

▼ Root	Array	(31 items)
▼ Item 0	Dictionary	(2 items)
name	String	All
image	String	all.png
▼ Item 1	Dictionary	(2 items)
name	String	Bistro
image	String	bistro.png
▼ Item 2	Dictionary	(2 items)
name	String	Bar / Lounge / Bottle Service
image	String	bar.png
▼ Item 3	Dictionary	(2 items)
name	String	Brewery
image	String	brewery.png

We will load this file into our `Explore` list, and this will be what we use to filter restaurants by a specific cuisine.

ExploreItem.swift

Next, we need to create a file to represent our data. Our `Explore` list will display an image and a name which match the corresponding image and name that we see in our `Explore.plist` file. Let's create this `ExploreItem` file now:

1. Right-click on the `Model` folder and select **New File**.
2. Inside of the template screen, select **iOS** at the top and then **Swift File**. Then, hit **Next**.
3. Name the file, `ExploreItem`, and then hit **Create**.

The only thing in this file is an import statement.



The import statement allows us to import other libraries into our file, giving us the ability to see inside of these libraries and use properties from them. Foundation is one of Apple's core frameworks, and it has a bunch of tools that we can use while we program.

Since we will not need to use inheritance, we are going to make this file a struct. Add the following into your file:

```
struct ExploreItem {
}
```

Now that we have declared it a struct, let's add the two properties we need for this file, an image and a name. For both of these properties, we are going to make them String data types. For the title, this makes sense, because it is text that we are displaying in our Collection View. However, for the image, using a String datatype might not seem as obvious. The reason we are making the image a data type of String is because, in order to get it, we have to access it by name. For example, `american.png` is the file name for the American cuisine image. Add the following to the inside of your curly braces ({ }):

```
var name:String?
var image:String?
```

We now have added two properties, one for image and one for name, both of which are Optional. Since we cannot give either of them an initial value, we have to make them Optional.

Your file should look like the following:

```
struct ExploreItem {
var name:String?
var image:String?
}
```

We next need to add one more thing to this file.

We will take the dictionary data we get from the `plist` and create an `ExploreItem` for each item. Our dictionary will look like the following:

```
["name": "All", "image": "all.png"]
```

We need to pass this dictionary object to our `ExploreItem`. Doing this requires that we add a custom initializer that will take a dictionary object into it; then we can set each item from the dictionary to the data of both of our properties, `image` and `name`.



When you create a struct, by default, you will get an `init()` method that will have all the properties you created in the parameters. For example, our `ExploreItem` will have a default initializer that looks like the following:

```
init(name:String, image:String)
```

Instead of using this initializer, we will create our own in order to pass a dictionary object into it.

In order to create a custom initializer, we are going to use what is called an extension, which gives us the ability to extend our code and add more functionality to it. Inside of your `ExploreItem` file, after the ending curly brace, add the following:

```
extension ExploreItem {
}
```

Next, let's create our custom initializer that will take a dictionary object into the parameters. Add the following between the curly braces of the extension we just added:

```
init(dict:[String:AnyObject]) {
}
```

We have now created an `init()` method; in the parameters, we will accept a dictionary object. As stated in the preceding section, we know that our data will look like the following:

```
["name": "All", "image": "all.png"]
```

In order to pass each value, we need to use the dictionary syntax, such as:

```
dict["name"]
dict["image"]
```

Let's proceed by mapping the dictionary data to our two properties. Add the following inside of the `init()` method curly braces:

```
self.name  = dict["name"] as? String
self.image = dict["image"] as? String
}
```



Since our dictionary value is `AnyObject`, we have to specify that our data is a String by using the `as? String` at the end.

We now have our data item set up for our Explore View (cuisine list), and your file should look like the following:

```
extension ExploreItem {
init(dict:[String:AnyObject]) {
self.name  = dict["name"] as? String
self.image = dict["image"] as? String
  }
}
```

Let's now focus on our Data Manager. We want our Data Manager to handle parsing the `plist` and giving us the data. Since our data will be coming from a `plist`, we need to have a method that will get the data from the `plist` first.

ExploreDataManager.Swift

In our app, the data manager will be responsible for communicating with a service (for example, Yelp API, which we will discuss later in the book) as well as manipulating the data from the service. Once the data from the service has been received, the data manager will create Model objects that we can use for our app.

In some apps, these two jobs are handled by the Controller. However, rather than putting that responsibility on our Controller, we will limit the Controller to talking to the manager so that it will never know anything about the service.



As you get comfortable with programming, you will find that there are a few different types of architectures. We are sticking as closely as we can to MVC, because it is what Apple uses to build iOS apps.

Let's create the `ExploreDataManager` file now:

1. Right-click on the `Model` folder and select **New File**.
2. Inside of the template screen, select **iOS** at the top and then **Swift File**. Then, hit **Next**.
3. Name this file, `ExploreDataManager`, and hit **Create**.

Since we need to define our class first, add the following under the import statement:

```
class ExploreDataManager {

}
```

Here, we used a class instead of a struct, because this is a file that we will inherit from later. You do not always necessarily know if you are going to inherit from another class or not; therefore, you can simply default to a struct and then change to a class if you realize that you need to inherit from another class.

Now, we need to load data from the `ExploreData.plist` file. Add the following method into our `ExploreDataManager`:

```
import Foundation

class ExploreDataManager {

    fileprivate func loadData() -> [[String: AnyObject]] {
        guard let path = Bundle.main.path(forResource: "ExploreData", ofType: "plist"),
            let items = NSArray(contentsOfFile: path) else {
              return [[:]]
        }

        return items as! [[String : AnyObject]]
    }
}
```

- **Part A**: `fileprivate`

  This function starts with the keyword `fileprivate`. Think of `fileprivate` as a way to give your methods an access level. If you do not use `fileprivate`, it defaults to internal, which means anyone can access or use the method outside of the class.

- **Part B**: `[[String: AnyObject]]`

  Our function `loadData()` is returning something back. The `->` states that our function has a return value. The return value for this method is an array with dictionary objects. Our dictionary will have a key of a String, and the value will be `AnyObject`.



`AnyObject` lets us take any data type that comes back. Therefore, we can have one item give us an Int, while another gives us back a String.

- **Part C**:

  ```
  guard let path = Bundle.main.path(forResource: "ExploreData",
  ofType: "plist")
  ```

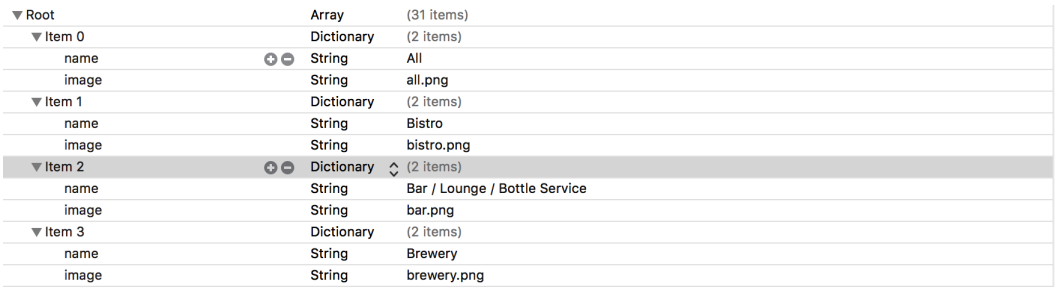
  Inside of the function, we are using what is known as a guard statement. A guard statement was designed for exiting a method or function early if a given statement returns false. Our guard actually checks for two statements that both need to return true.

The first statement checks to see if the file `ExploreData.plist` exists in our app bundle. If the file is found, the statement will return true, and the file path is set to the constant `path`. Our next statement, which is separated by a comma is discussed in **D** below.

- **Part D**:

```
let items = NSArray(contentsOfFile: path)
```

In this statement, if the first statement returns true, we take the constant "path"; and then we check the contents inside of the file. Let's take a look at the data in our file again:



▼ Root	Array	(31 items)
▼ Item 0	Dictionary	(2 items)
name	String	All
image	String	all.png
▼ Item 1	Dictionary	(2 items)
name	String	Bistro
image	String	bistro.png
▼ Item 2	Dictionary	(2 items)
name	String	Bar / Lounge / Bottle Service
image	String	bar.png
▼ Item 3	Dictionary	(2 items)
name	String	Brewery
image	String	brewery.png

If you look at the `Root` of this plist, you will see that its type is an array. NSArray has a method that we can use to get the data out of our file and put it into an array with which we can work.



Typically, plist come in two types, an array or a dictionary. Currently, neither the standard array nor dictionary gives us a method that allows us to get data out of a file, so we need to utilize NSArray (as we are here) or NSDictionary, respectively, to do that.

This statement will now check to verify that we are, indeed, working with an array and, then, will return true if so. If both conditions return true, our array inside of our plist will be given to us. The array will be set to our constant `items`.



NSArray and NSDictionary come from Objective C (Apple's main programming language for building iOS apps); they have some extra features. Just know that they are similar to their Swift counterparts without the "NS."

- **Part E**: `else { return [[:]] }`

  Here, if any of the conditions are false, we will return an array with an empty dictionary. Otherwise, we will run the following return:

- **Part F**: `return items as! [[String : AnyObject]]`

  This return will give back an array of dictionary items. Once we have our data being loaded out of the `plist`, we can create our `ExploreItem`. Therefore, we will need a method so that we can access all of our Explore Items and return an array of items.

Getting Data

To get our data out of the `plist`, add the following method above `loadData()` inside of our `ExploreDataManager`:

```
func fetch() {
  for data in loadData() {
    print(data)
  }
}
```

Our method `fetch()` is going to loop through our dictionary data from the `plist`. Here is what your file should look like now:

```
import Foundation

class ExploreDataManager {

    func fetch() {
        for data in loadData() {
            print(data)
        }
    }

    fileprivate func loadData() -> [[String: AnyObject]] {
      guard let path = Bundle.main.path(forResource: "ExploreData", ofType: "plist"),
          let items = NSArray(contentsOfFile: path) else {
            return [[:]]
        }

      return items as! [[String : AnyObject]]
    }
}
```

Inside of your `ExploreViewController.swift` file, delete the previous print statement that was inside of your `viewDidLoad()` and replace it with the following:

```
let manager = ExploreDataManager()
manager.fetch()
```

Let's build and run the project by hitting the **Play** button (or use *cmd* + *R*). You will notice that, in the Debug panel, every time our loop runs, it gives a dictionary object, such as the following:

```
["image": all.png, "name": All]
["image": bistro.png, "name": Bistro]
["image": bar.png, "name": Bar / Lounge]
["image": brewery.png, "name": Brewery]
["image": burgers.png, "name": Burgers]
["image": californian.png, "name": Californian]
["image": caribbean.png, "name": Caribbean]
["image": comfort.png, "name": Comfort Food]
["image": cuban.png, "name": Cuban]
["image": continental.png, "name": Continental]
["image": french.png, "name": French]
["image": international.png, "name": International]
["image": italian.png, "name": Italian]
["image": japanese.png, "name": Japanese]
["image": latin.png, "name": Latin American]
["image": mediterranean.png, "name": Mediterranean]
["image": mexican.png, "name": Mexican]
["image": organic.png, "name": Organic]
["image": panasian.png, "name": Pan-Asian]
["image": peruvian.png, "name": Peruvian]
["image": pizza.png, "name": Pizzeria]
["image": primerib.png, "name": Prime Rib]
["image": seafood.png, "name": Seafood]
["image": southamerican.png, "name": South American]
["image": southern.png, "name": Southern]
["image": spanish.png, "name": Spanish]
["image": steak.png, "name": Steakhouse]
["image": sushi.png, "name": Sushi]
["image": tapas.png, "name": Tapas / Small Plates]
["image": vietnamese.png, "name": Vietnamese]
["image": wine.png, "name": Wine Bar]
```

This is exactly what we want.

Now, inside of `ExploreDataManager`, add the following directly above our `fetch` method:

```
fileprivate var items:[ExploreItem] = []
```

Next, inside of our `fetch()`, we will update our for-in loop by replacing print(data) with the following:

```
items.append(ExploreItem(dict: data))
```

When you are done, your file should look like mine:

```
import Foundation

class ExploreDataManager {

    fileprivate var items:[ExploreItem] = []

    func fetch() {
        for data in loadData() {
            items.append(ExploreItem(dict: data))
        }
    }

    fileprivate func loadData() -> [[String: AnyObject]] {
        guard let path = Bundle.main.path(forResource: "ExploreData", ofType: "plist"),
            let items = NSArray(contentsOfFile: path) else {
            return [[:]]
        }

        return items as! [[String : AnyObject]]
    }
}
```

Let's build and run the project by hitting the **Play** button (or use *cmd* + *R*). In the Debug panel, you should see an array of Explore Items.

Next, let's open `Explore.storyboard` and update our Collection View to have a gray background:

1. Select the Collection View in the Outline view.
2. Next, select the Attributes Inspector in the Utilities panel.
3. Scroll down to **View** and click **Background**.
4. In the Colors panel that appears, select the second tab, called **Color Sliders** and update the Hex Color #value to `F2F2F4` and hit **Enter**. You may have to repeat this a second time to change the Hex Color #. You can now close the Color panel.

Now, we need to focus on the individual cells that are being displayed in our Collection View.

CollectionView Cell

We currently have our data, and we have cells. However, we need to get our data to our cells so that we can see the image and name. Let's open up `Explore.storyboard` and update our `exploreCell`.

Currently, our `exploreCell` has a random background color, but we need our cell to have a white background. We also need to update the cell size:

1. With `exploreCell` selected in the Outline view, select the Size Inspector in the Utilities panel and update the following:
    - **Size**: `Custom`
    - **Width**: `177`
    - **Height**: `154`
2. Next, open up the Attributes Inspector in the Utilities panel and change the `Background` to `White Color`.
3. Then, in the Utilities panel, select the Object library; and, in the filter, type `image`.
4. Drag the Image View out into your cell.
5. Open the Size Inspector panel again and update the following under **View**:
    - **X**: `0`
    - **Y**: `0`
    - **Width**: `177`
    - **Height**: `130`
6. Now, we will apply Auto Layout to this image by selecting the Pin icon and entering the following values:
    - Top, Left and Right under **Add New Constraints** are set to `0` (if already set, re-enter)
    - **Height**: `130` (should be checked)
    - **Update Frames**: `Items of New Constraints`



These constraints will force our image to always be the same size and be constrained to the upper-left corner of the cell.

7. Click **Add 4 Constraints**.
8. Next, in the Utilities panel, select the Object library, and, in the filter, type `label`.
9. Drag the Label out into your cell under the Image View.
10. With Label still selected, go to the Attributes Inspector in the Utilities panel and update the Font to be System 12.0, using the down arrow.

11. Then select the Size Inspector in the Utilities panel, and, under **View**, update the following:
    - **X**: `8`
    - **Y**: `135`
    - **Width**: `124`
    - **Height**: `16`
12. Finally, we need to apply Auto Layout to this Label by selecting the Pin icon and entering the following values under **Add New Constraints**:
    - **Top**: `5`
    - **Left**: `8`
    - **Right**: `8`
    - **Height**: `16` (should be checked)
    - **Update Frames**: `Items of New Constraints`
13. Click on **Add 4 Constraints**.

Connecting to Our Cell

Now that we have our cell set up, we need to create a file so that we can connect to our cells:

1. Right-click on the `View` folder in the Navigator panel and select **New File**.
2. Inside of the template screen, select **iOS** at the top, and then **Cocoa Touch Class**. Then, hit **Next**.
3. You will now see an options screen. Add the following:

   New File:

   - **Class**: `ExploreCell`
   - **Subclass…**: `UICollectionViewCell`
   - **Also create XIB**: Unchecked
   - **Language**: `Swift`

4. After hitting **Next**, you will be asked to create this file. Select **Create**, and your file should look like mine:

```
import UIKit

class ExploreCell: UICollectionViewCell {

}
```

5. Next, open `Explore.storyboard` and select the `exploreCell` in the Outline view.
6. In the Utilities panel, select the Identity I0nspector and, under **Custom Class**, type `ExploreCell`. Then, hit **Enter**.

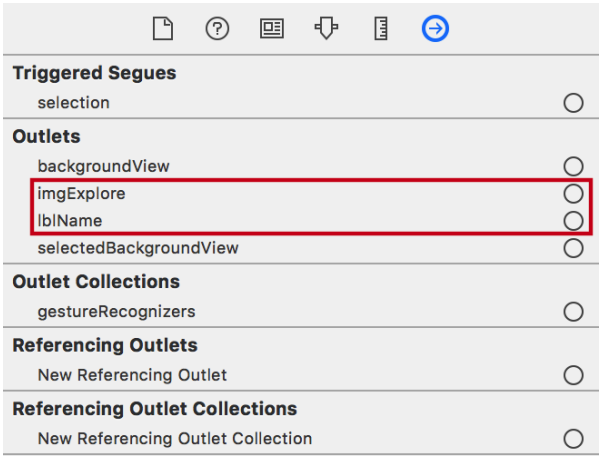
Hooking up Our UI with IBOutlets

In order to access our UI elements, we need to connect them with `IBOutlets`.

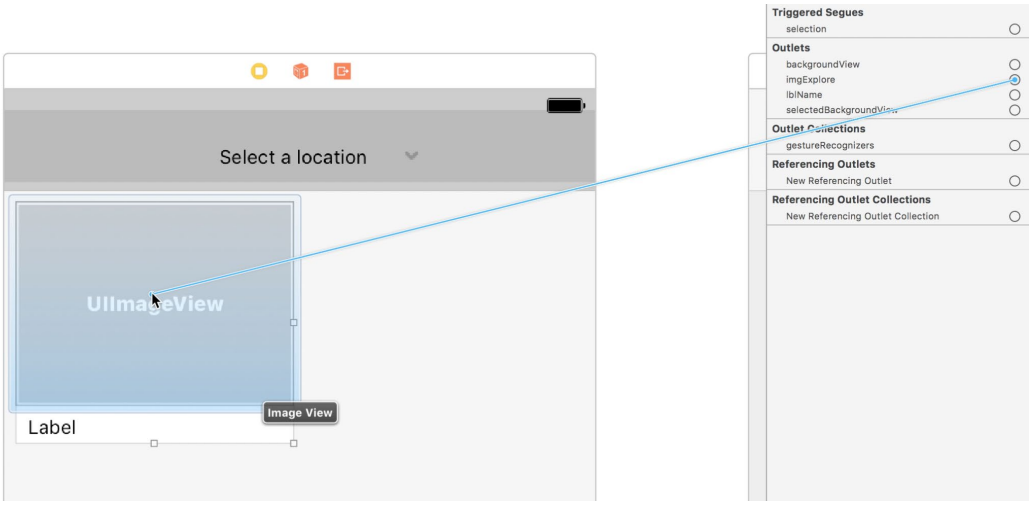
1. Open the `ExploreCell.swift` file in the Navigator panel (or use *cmd* + *SHIFT* + *O*, and type `ExploreCell` and then hit **Enter**).
2. Inside of the class declaration add the following:

```
@IBOutletvarlblName:UILabel!
@IBOutletvarimgExplore:UIImageView!
```

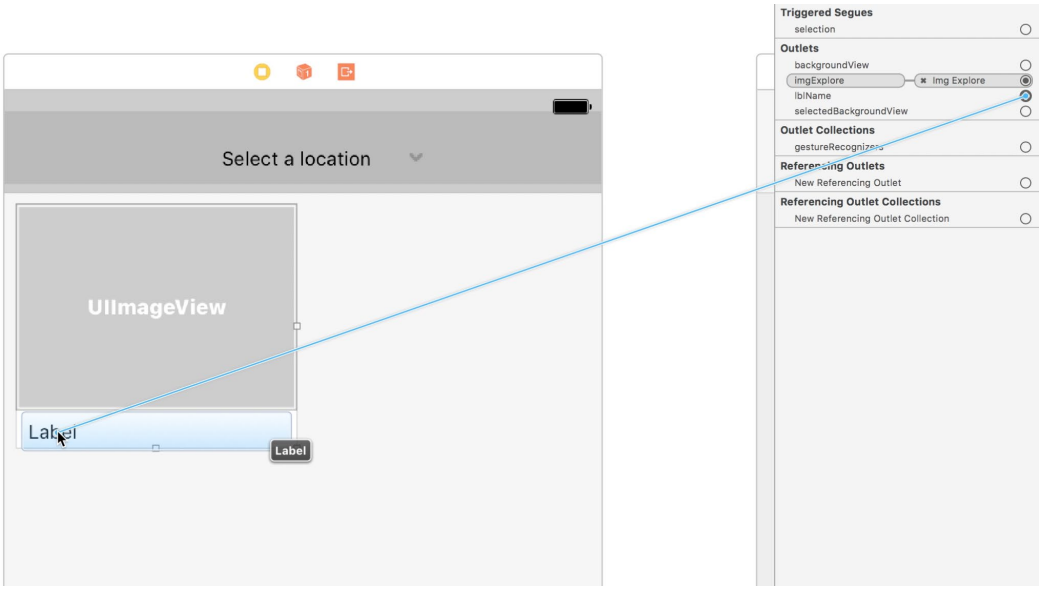
3. Open `Explore.storyboard` and select your `exploreCell` again.
4. In the Utilities panel, select the Connection Inspector. You will now see both variables we just created, `lblName` and `imgExplore` under `Outlets`:



5. Now, CTL drag from `imgExplore` to the `UIImageView` we put in our cell.



6. Repeat this step for `lblName` by CTL dragging from `lblName` to the `UILabel` in our cell.



Great! Now that we have our cell set up, let's pull data into it. In our `ExploreDataManager`, add these two methods above the `loadData()` method:

```
func numberOfItems() ->Int {
   return items.count
}

func explore(at index:IndexPath) ->ExploreItem {
   return items[index.item]
}
```

We use the first method, `numberOfItems`(), to update the total number of items in our Collection View. The second method, explore (at `index:IndexPath`), will be called for each item we create in our Collection View. Then, we will use this to pass the data to our cell to display the name and the image.

Now that we have these two methods added, let's open up our `ExploreViewController` file. We currently have the following inside of our `viewDidLoad()`:

```
 let manager = ExploreDataManager()
manager.fetch()
```

Let's move `let manager` underneath our Collection View, so that it is outside `viewDidLoad()`; and, therefore, we can access it anywhere within the class as opposed to only within the function. You should now have this before the `viewDidLoad`():

```
@IBOutlet var collectionView: UICollectionView!
let manager = ExploreDataManager()
```

Inside of `viewDidLoad()`, only `manager.fetch()` remains.

Next, we need to update our `numberOfItemsInSection()` to say:

```
func collectionView(_ collectionView: UICollectionView,
numberOfItemsInSection section: Int) ->Int {
  return manager.numberOfItems()
}
```

Therefore, instead of returning 20, we are going to get the number of items from our `plist`.

Finally, inside of our `cellForItemAt`(), revise the let statement in the third required method before `return cell` by adding `as! ExploreCell` as follows:

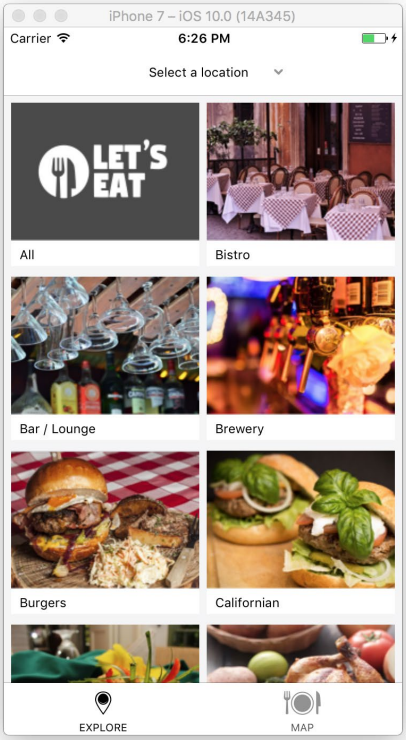
```
let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
"exploreCell", for: indexPath) as! ExploreCell
```

Then, add the following after the code snippet you just added and before `return cell`:

```
let item = manager.explore(at: indexPath)
if let name = item.name { cell.lblName.text = name }
if let image = item.image { cell.imgExplore.image = UIImage(named:
image) }
```

This will get an `ExploreItem` for each cell in our Collection View and pass the data to the cell.

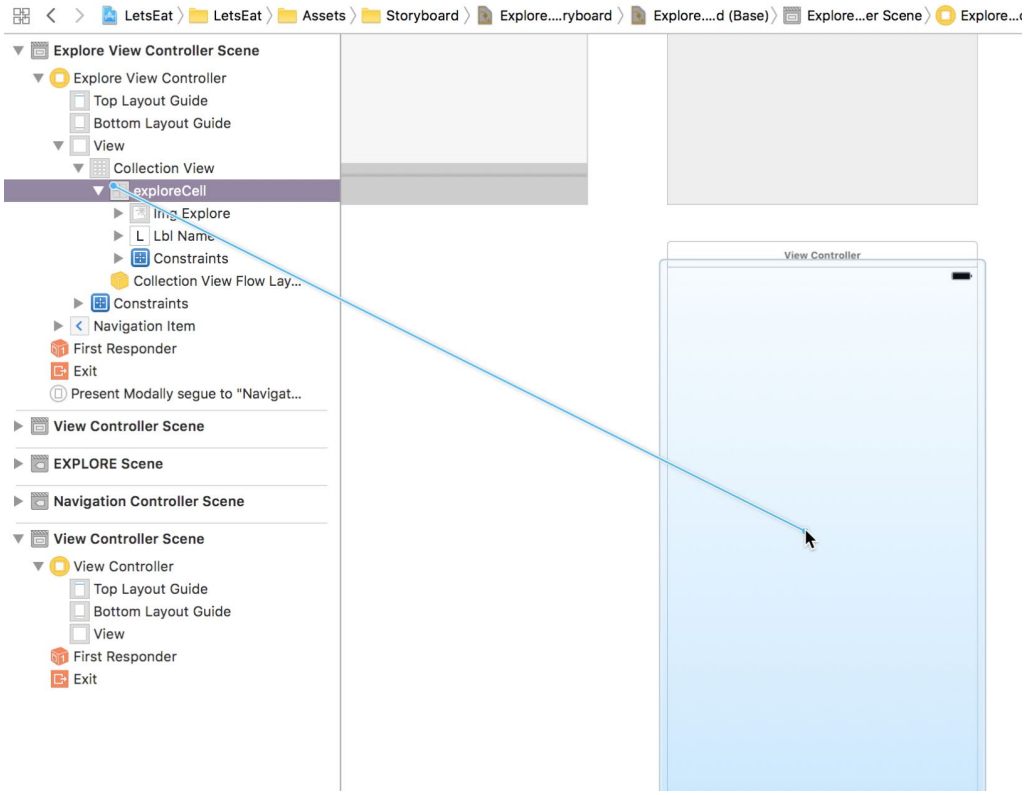
Let's build and run the project by hitting the **Play** button (or use *cmd* + *R*). You should now see your Collection View come to life with images and text:



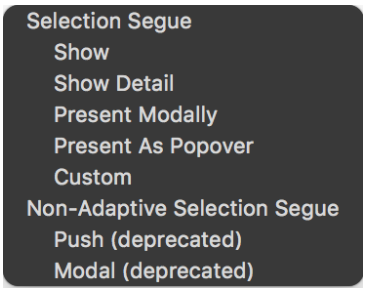
Now that we have our cells displaying content, we need to make it so that when you select a cell, it will go to our restaurant listing.

Cell Selection

1. Go to our `Explore.storyboard` and open up the Utilities panel.
2. In the Object library, drag out a View Controller.
3. Now, CTL drag from our `exploreCell` in the Outline view to the View Controller we just added:



You will be prompted with a screen:



4. Select **Show**.

Let's build and run the project by hitting the **Play** button (or use *cmd* + *R*). You should now be able to select your cell, and it will go to what will be your Restaurant listing page. This page will be empty for now, so let's work on this next.

Restaurant Listing

Now that we have our Explore listing going to our Restaurant listing, we need to get our Collection View connected to our `RestaurantListViewController`. The first thing we should do is create a folder inside of the `Restaurant` folder:

1. Right-click on the `Restaurant` folder and create a group called `Restaurant List`.
2. Then, right-click on the `Restaurant List` folder and create two new groups—`Controller` and `View`.
3. Right-click on the `Controller` and select **New File**.
4. Inside of the template screen, select **iOS** at the top and then **Cocoa Touch Class**. Then, hit **Next**.
5. You will now see an options screen. Add the following:

   New File:

   - **Class**: `RestaurantListViewController`
   - **Subclass…**: `UIViewController`
   - **Also create XIB**: `Unchecked`
   - **Language**: `Swift`

6. After hitting **Next**, you will be asked to create this file. Select **Create**, and your file should look like mine:

```
//
//  RestaurantListViewController.swift
//  LetsEat
//
//  Created by Craig Clayton on 11/1/16.
//  Copyright © 2016 Craig Clayton. All rights reserved.
//

import UIKit

class RestaurantListViewController: 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.
    }


    /*
    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before
        navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destinationViewController.
        // Pass the selected object to the new view controller.
    }
    */
}
```

Let's delete both `didReceiveMemoryWarning()` and `prepare()` (which has been commented out) as we do not need them at this time.

Next, we need to drag out a `UICollectionView` and connect it to our View Controller class:

1. Select `Explore.storyboard`.
2. Then, select the `UIViewController` we just created.
3. Now, in the Utility Panel, select the Identity Inspector, which is the third icon from the left.
4. Under Custom Class, in the Class drop-down menu, select `RestaurantListViewController` and hit **Enter**.
5. Next, in the Utility Panel under the **Object library**, type `collection` into the filter.
6. Then, drag the Collection View out into your `View`.

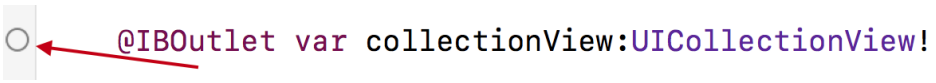
7. With your Collection View still selected, select the Pin icon to apply Auto Layout and enter the following values under **Add New Constraints**:
    - **Top**: `0`
    - **Left**: `0`
    - **Right**: `0`
    - **Bottom**: `0`
    - **Constrain to Margins**: `Uncheck`
    - **Update Frames**: Items of New Constraints
8. Click on **Add 4 Constraints**.

Next, we are going to use the Assistant editor:

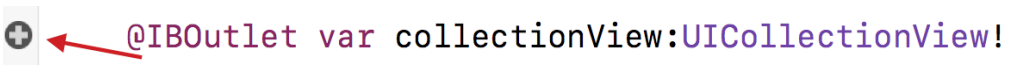
1. Open `Explore.storyboard`.
2. If your Navigator panel is currently open, close it by clicking on the hide navigator toggle or *cmd + 0*.
3. If your Utilities panel is currently open, close it by clicking on the Utilities toggle or use *cmd+ ALT/ALT + 0*.
4. Next, select the Assistant editor or use *cmd+ ALT/ALT + ENTER*.

   You should now see `Explore.storyboard` on the left side and `RestaurantListViewController.swift` on the right.
5. Now, add the following after `class RestaurantListViewController: UIViewController {`:

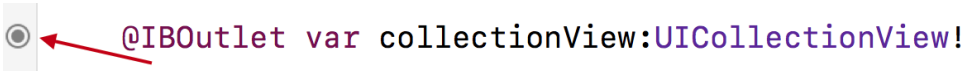
   ```
   @IBOutletvarcollectionView:UICollectionView!
   ```
6. After you create the variable, you will see a small circle to the left of the variable:



7. When you hover over it you, will see a plus button appear inside of the circle. Click on and drag this to your Collection View inside of your `UIViewController`:



8. Once you release, you will see the circle become filled:



9. Now, select the Standard editor or use *cmd* + *ENTER*. This will bring you to just the `Explore.storyboard`, where you will see a small box inside of your Collection View. This is your Collection View cell.
10. Select the cell and then the Attributes Inspector in the Utilities panel (toggle this panel on if it is hidden). Update the following:
    - **Identifier**: `restaurantListCell`
    - **Background**: Any color
11. In your scene, select your Collection View. Then, in your Utilities panel select the Connections Inspector, where you will see, under the **Outlets** section, two empty circles, `dataSource` and `delegate`.
12. Click and drag each one to the Restaurant List View Controller in your Outline view (as we did earlier with our Explore View Controller).
13. Now, select the Size Inspector and update the following values under **Cell Size**:
    - **Width**: `375`
    - **Height**: `135`
14. Now, select the Restaurant List View Controller in the Outline view.
15. In the Utilities panel, select the Attributes Inspector and scroll down to **View Controller**, where you will see **Extend Edges**.
16. Uncheck both **Under Top Bars** and **Under Bottom Bars**.
17. Inside of the Storyboard, select the Collection View and then the Pin icon, and update the bottom constraint (value should currently be -49) by entering the following values:
    - **Bottom**: `0030`
    - **Update Frames**: **Items of New Constraints**
18. Click on **Add 1 Constraint**.

Now, it is time to display something inside of our Collection View:

19. Use *cmd* + *SHIFT* + *O*, which will open the **Open Quickly** search box, and type `Restaurant`. Then, hit **Enter** to select the `RestaurantListViewController.swift` file.
20. Update our class definition from class `RestaurantListViewController:UIViewController` to the following:

    ```
    class RestaurantListViewController:UIViewController,
    UICollectionViewDataSource
    ```

As you learned earlier with our Explore grid, we are required to implement `numberOfSections()`, `numberOfItemsInSection()`, and `cellForItemAt()` in order to use a Collection View. Therefore, add those three methods inside of our `RestaurantListViewController`:

```
import UIKit

class RestaurantListViewController: UIViewController, UICollectionViewDataSource {

    @IBOutlet var collectionView: UICollectionView!

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return 1
    }

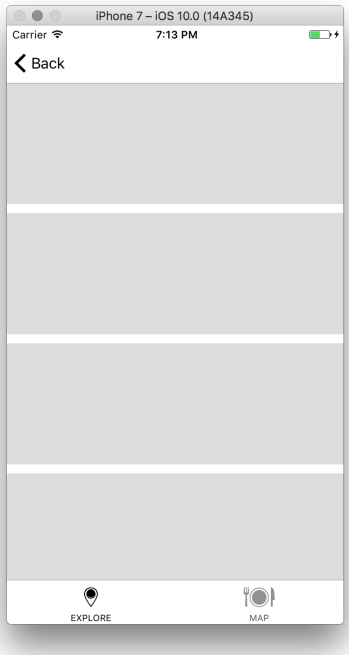
    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return 20
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) ->
        UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "restaurantCell", for: indexPath)

        return cell
    }

}
```

Let's build and run the project by hitting the play button (or use *cmd*+ *R*) in order to see what happens:



Instead of having a grid, like we did for Explore, our Restaurant list displays a column of cells. However, when the Restaurant list displays on the iPad, it will show a grid instead. This is the one of the flexibilities from which we benefit by using a Collection View. We will further setup our Restaurant list cells along with displaying the data later in the book.

Summary

In this chapter, we covered quite a few new topics as well as a lot of code. As long as you have a basic understanding of what we covered in this chapter, you will be fine to continue. A lot of these concepts and ideas will be covered again, as these are common design patterns in iOS.

You now have a better understanding of what MVC is and how it is used in our app. We also covered getting data from a `plist` and how to represent that data as a Model object. In addition, we worked with two Collection Views to display custom cells, each displaying differently (one with a grid and one with a single column).

In the next chapter, we will look at the differences between static and prototype Table Views. You will see a lot of similarities between Table Views and Collection Views, especially with respect to how we get data into them.

8
Getting Started with the List

When I started doing iOS development, I first worked with Table Views. At the time, Collection Views had not yet been introduced. As you progress in iOS development, you will work with a lot of Table and Collection Views. You will begin with just the basics to get them going; then, you will slowly progress into more advanced Table and Collection Views.

The reason that I bring this up is because, by the end of this chapter, you may feel as though things are not clicking. This is perfectly normal, but the more you do the steps in these chapters, the more they will become second nature to you.

For those of you that have not done iOS development, Table Views are great for presenting a list of data. The iPhone's mail app is an example of what a Table View typically looks like.

In this chapter, we are going to work with our first Table View. In our *Let's Eat* app, users will select a specific location to look for restaurants.

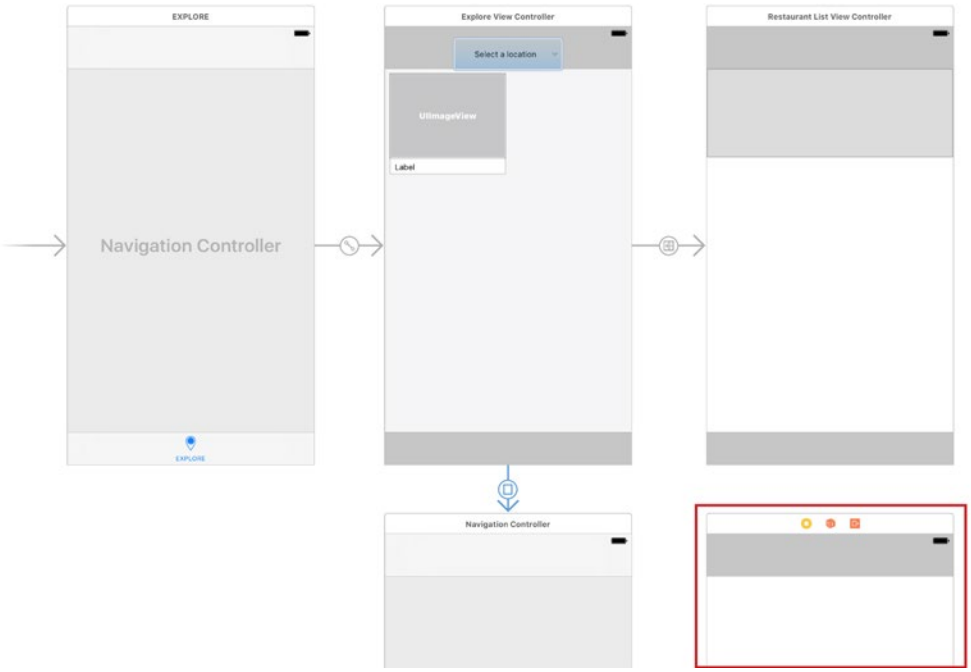
We will cover the following in this chapter:

- Updating UI in Storyboard
- Adding Our First Table View
- Creating Our First Property List (plist)
- Creating Our Location Data Manager

Updating UI in Storyboard

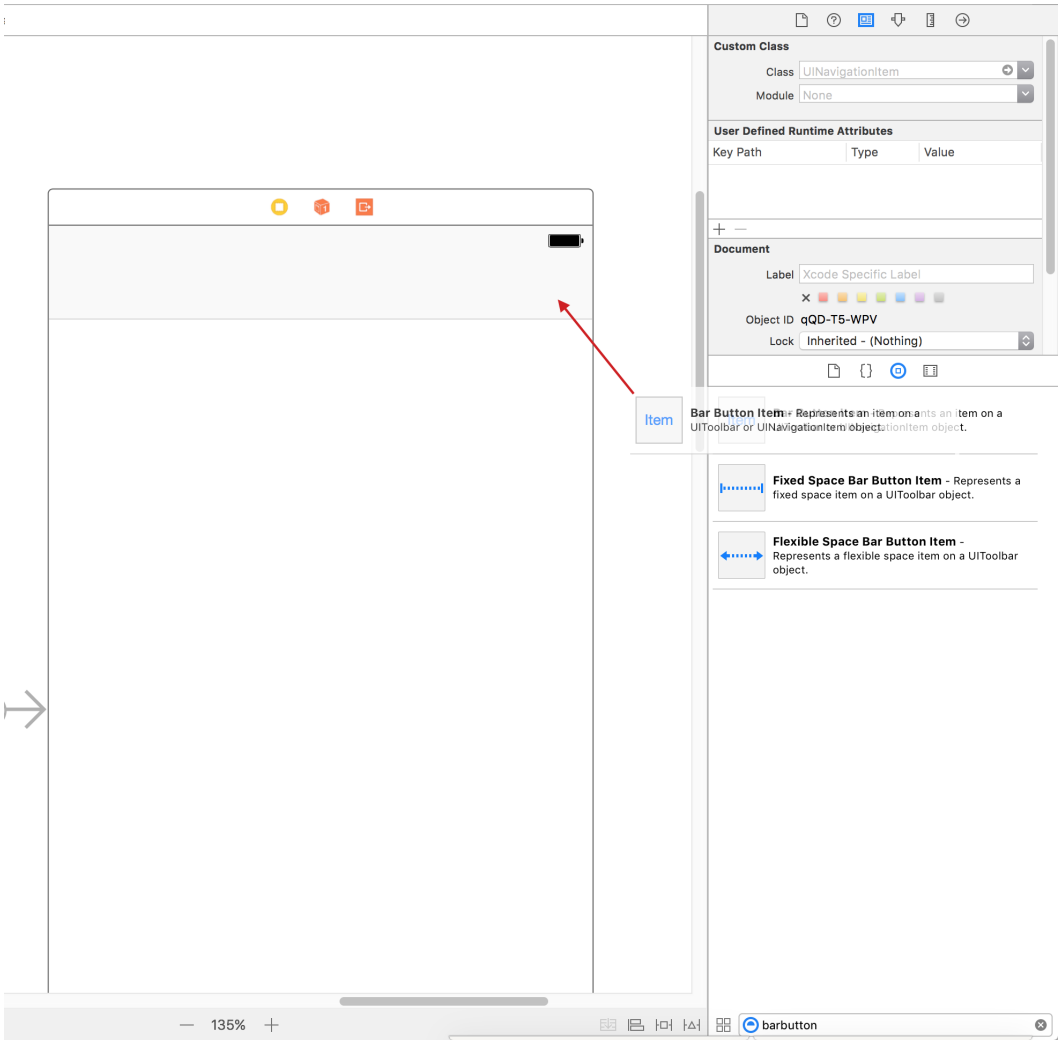
Currently, if you hit **Select a location** in your simulator, a modal pops up. However, you cannot dismiss the modal without restarting the entire app. Therefore, we need a **Cancel** button and a **Done** button to dismiss the view. Let's work on this first:

1. Open `Explore.storyboard` and select the segue that is connected to your **Select a Location**. It should now be highlighted.
2. Then, go to your View Controller (not the Navigation Controller) of your modal:

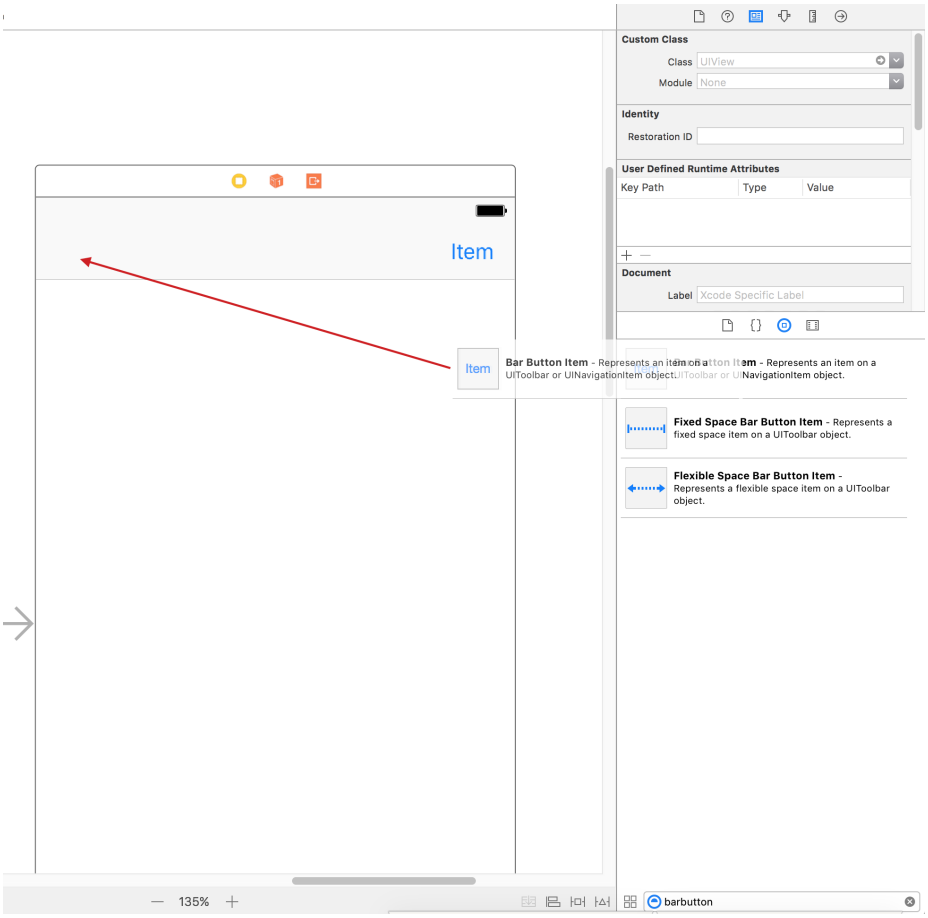


3. As you have done in previous chapters, type in `bar button` into the filter area of the `Objects` library in the Utilities panel.

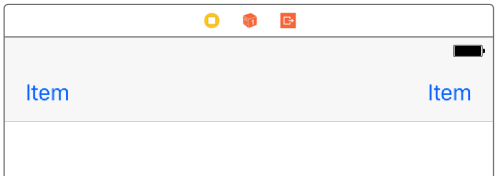
4. Drag and drop a **Bar Button Item** into the right area of the Navigation Bar of your View Controller Scene:



5. Drag another **Bar Button Item** into the left area of the Navigation Bar:



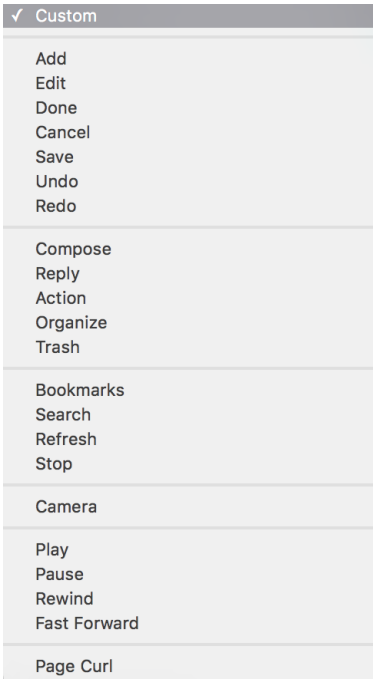
6. You should now have two Bar Button Items that both say Item:



Updating Bar Button Items

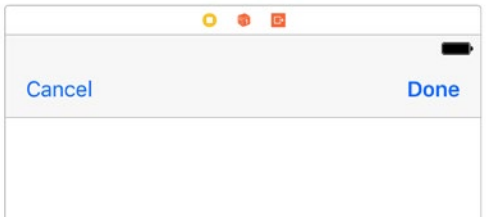
Next, we need to update both of the **Bar Button Items** to say `Cancel` and `Done`:

1. Select the left **Bar Button Item**, and in the Utilities panel, select the Attributes Inspector.
2. Click on **System Item**, and select **Cancel** in the drop-down menu:



3. Select the right **Bar Button Item**, and while still in the Attributes Inspector in the Utilities Panel, update **System Item** to `Done`.

Now, you should see **Cancel** on the left and **Done** on the right:



Unwinding our cancel button

Now that we have our buttons, we want to dismiss the modal when a user hits "Cancel":

1. Open up your `ExploreViewController.swift` file in the Navigation panel and add the following code after the `cellForItemAt` method:

   ```
   @IBAction func unwindLocationCancel(segue:UIStoryboardSegue) {}
   ```

2. Your file should now look like the following:

   ```
       func collectionView(_ collectionView: UICollectionView,
           numberOfItemsInSection section: Int) -> Int {
           return manager.numberOfItems()
       }

       func collectionView(_ collectionView: UICollectionView, cellForItemAt
           indexPath: IndexPath) -> UICollectionViewCell {
           let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
               "exploreCell", for: indexPath) as! ExploreCell

           let item = manager.explore(at: indexPath)
           if let name = item.name { cell.lblName.text = name }
           if let image = item.image { cell.imgExplore.image = UIImage(named:
               image) }

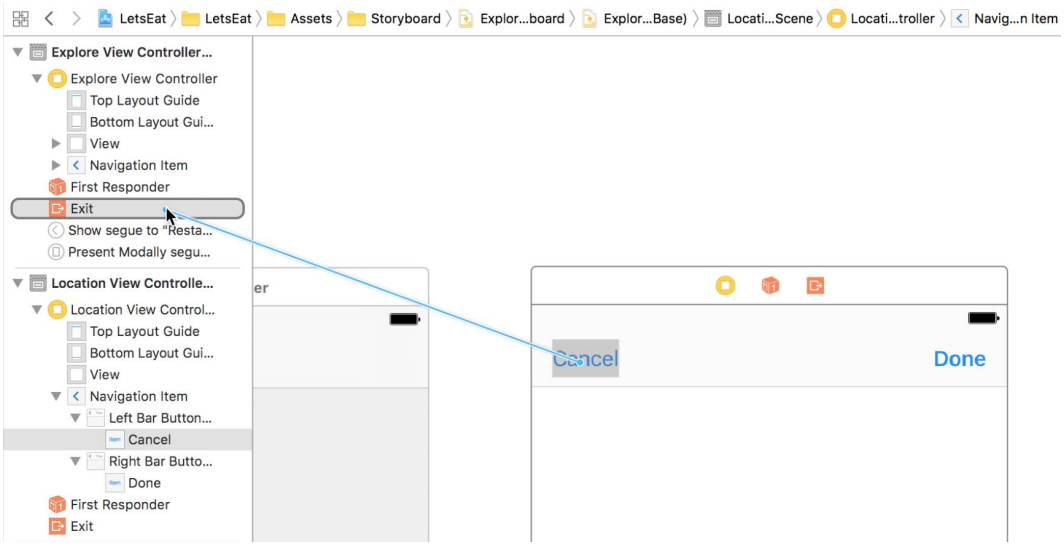
           return cell
       }

       @IBAction func unwindLocationCancel(segue:UIStoryboardSegue) {}
   }
   ```

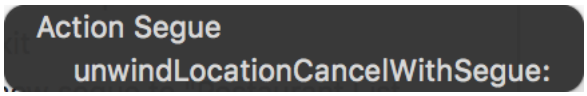
3. Save the file by hitting *cmd*+ *S*.

   This function is dismissing (unwinding) our modal. We actually do not need to put anything inside of this function, but we do need to call it.

4. Therefore, open `Explore.storyboard` again, and in the Outline view, make sure you click on the disclosure arrow for Explore View Controller Scene.
5. Then, CTL drag from the **Cancel** button to **Exit** in your Outline view:



6. You will see a window popup that says `Action Segue - unwindLocationCancelWithSegue`. Select `unwindLocationCancelWithSegue`:

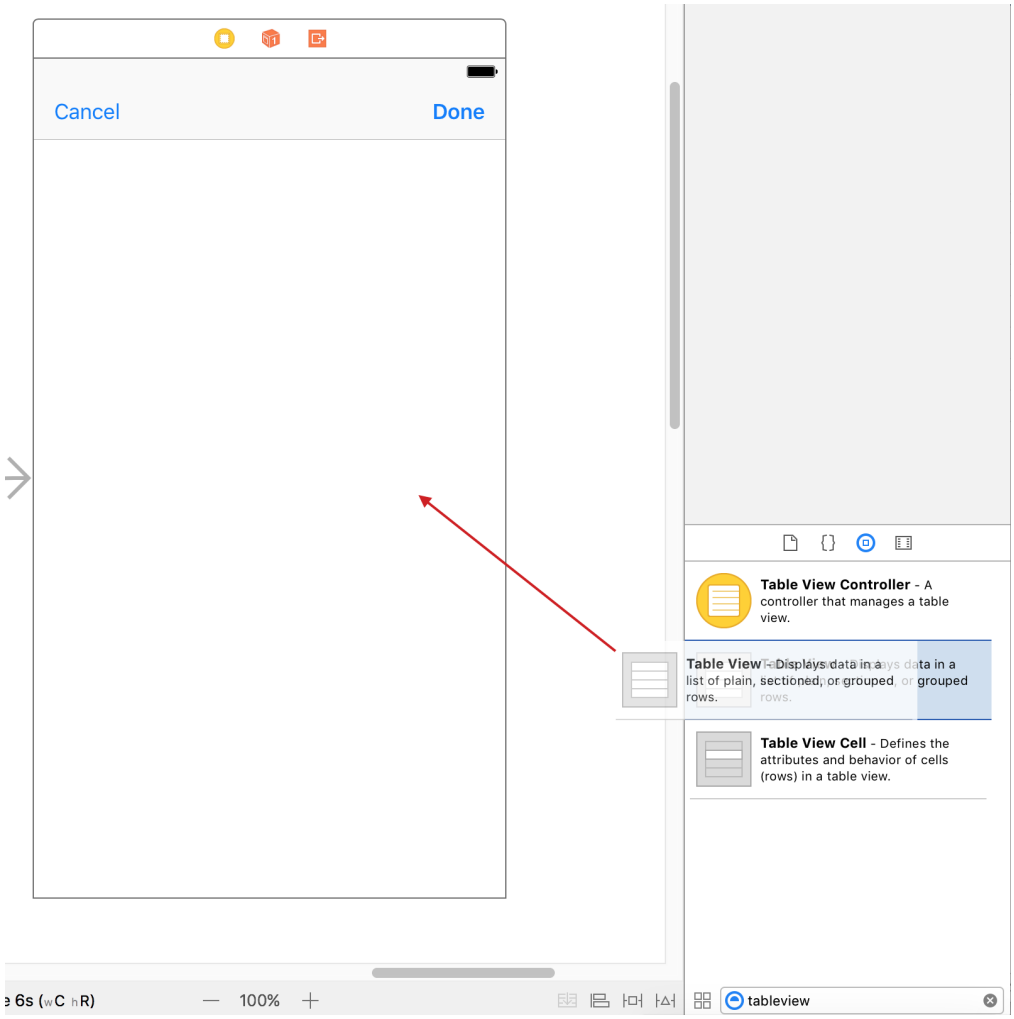


Let's build and run the project by hitting the **play** button (or use *cmd* +*R*) and test our **Cancel** button. It should now dismiss the View. We will update the **Done** button later.

Adding Our First Table View

Now, let's add a `UITableView` into our `UIViewController`:

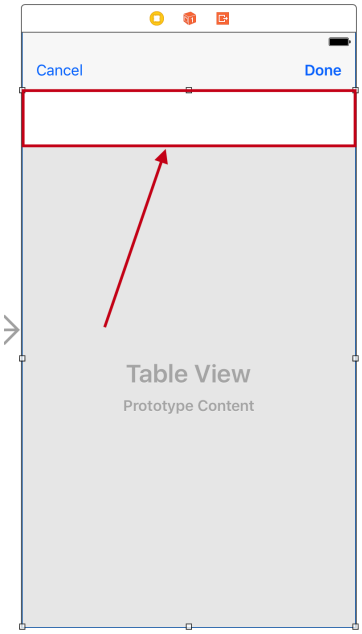
1. Open `Explore.storyboard`.
2. In the Utilities panel, in the filter field, type `tableview`; then, drag the Table View onto the scene:



3. Select the Pin icon and enter the following values:
    - Set all values under **Add New Constraints** to `0`
    - **Constrain to margins**: `unchecked`
    - **Update Frames**: `Items of New Constraints`
4. Now click on **Add 4 Constraints**.

Updating Our Edges

We now need to correct the gap at the top of our Table View:



1. With the View Controller selected (as opposed to the Table View), select the Attributes Inspector in the Utilities panel.
2. Scroll down to **Extend Edges** and uncheck both **Under Top Bars** and **Under Bottom Bars**.

Creating Our View Controller Class

Now that we have our UI set up, we need to get our data to display inside of our Table View. Before we start, create three new folders inside of the `Location` folder—`Controller`, `View`, and `Model`. As we have previously done, right-click on the `Location` folder and hit **New Group** in order to create a new folder.

Next, we need to create a Location View Controller class that we can use with our `UIViewController`:

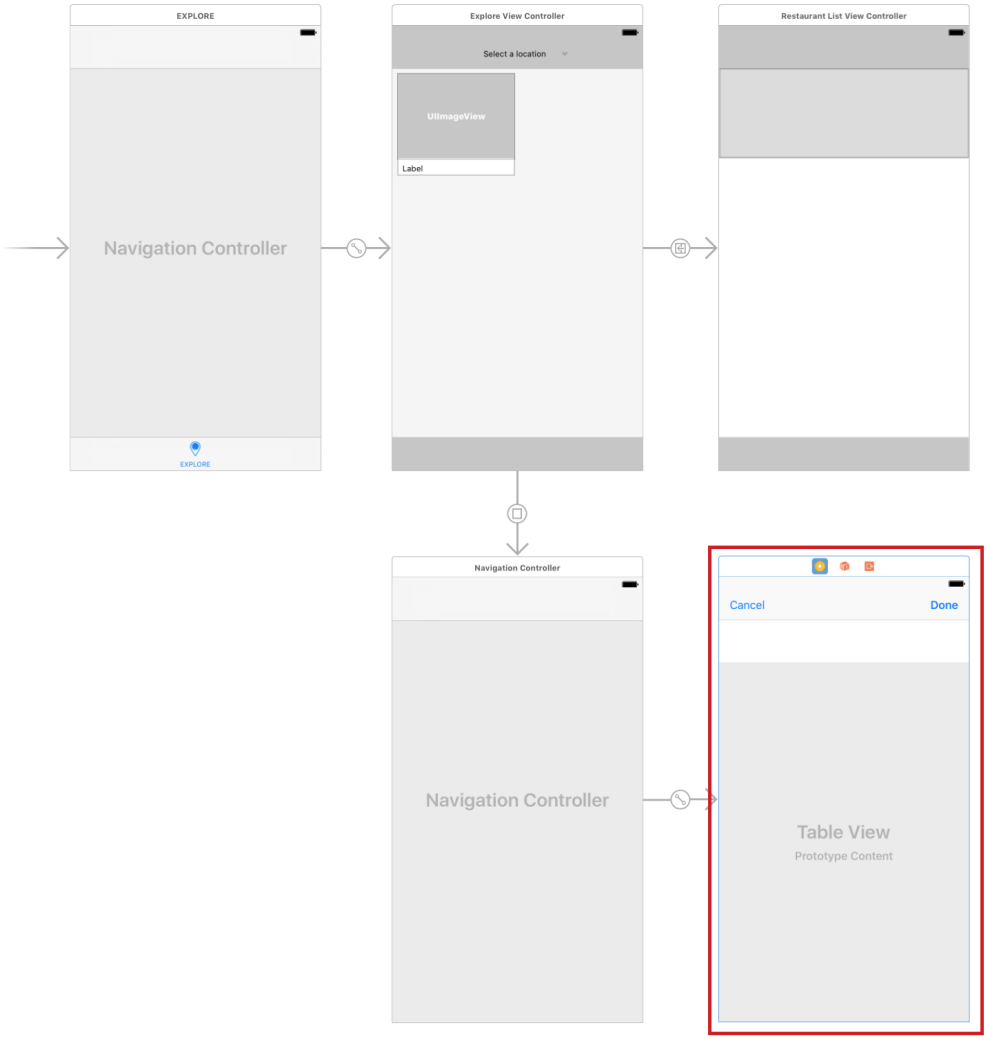
1. Right–click on the `Controller` folder you just created and select **New File**.
2. Inside of the **Choose a template for your new file** screen, select **iOS** at the top and **Cocoa Touch Class**. Then, hit **Next**.
3. In the options screen that appears, add the following:

   New File:

   - **Class**: `LocationViewController`
   - **Subclass…**: `UIViewController`
   - **Also create XIB**: `Unchecked`
   - **Language**: `Swift`
4. Click on **Next** and then **Create**.

Next, we need to connect our View Controller with our class:

1. Select `Explore.storyboard`.
2. Then, select the `UIViewController` that contains the `UITableView`:



3. Now, in the Utilities panel, select the Identity Inspector.
4. Under Custom Class, in the Class drop-down menu, select **LocationViewController** and hit **Enter**.

Connecting our TableView with our Location View Controller

Currently, we have no way to communicate between our TableView and our Location View Controller. Let's see how we can connect these two together:

1. Open the `LocationViewController.swift` file and add the following code after the class declaration:

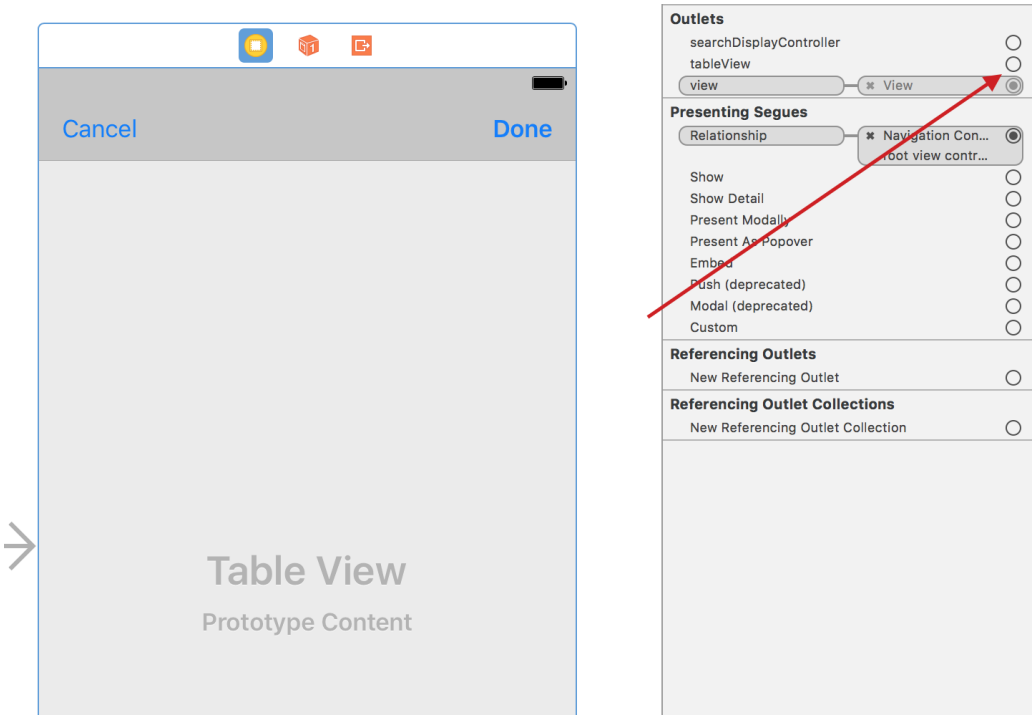
```
@IBOutlet var tableView:UITableView!
```

2. Save the file by hitting *cmd+ S*. Your file should look like the following with an empty circle next to the variable:

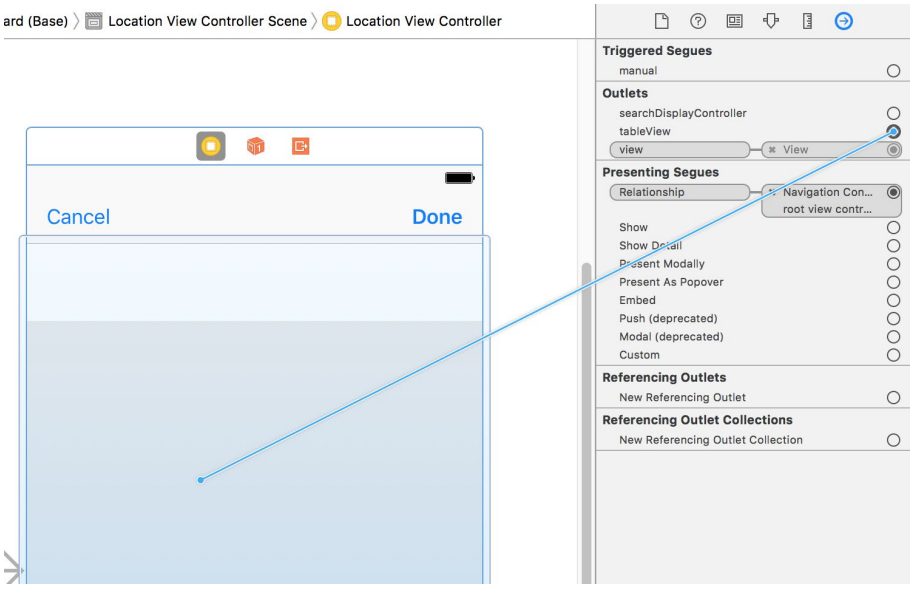
```
class LocationViewController: UIViewController {

    @IBOutlet var tableView:UITableView!
```

3. Open `Explore.storyboard` again and make sure that you have the Location View Controller selected in the Outline view.
4. Then, in the Utilities panel, select the Connections Inspector. Under the **Outlets** section, you will see an empty circle, TableView:



5. Click on and drag from the empty circle to the Table View in the Storyboard:

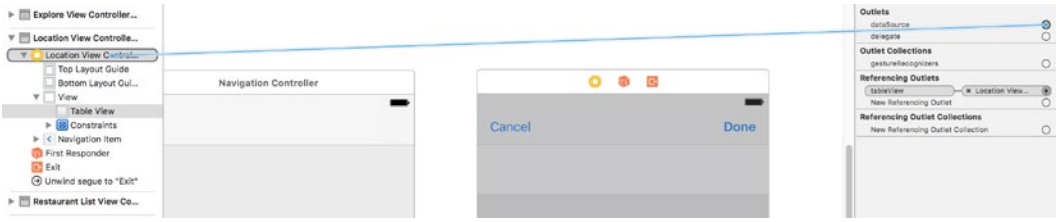


We have now connected our Table View and our Location View Controller.

Adding the Data Source and Delegate

As discussed in the previous chapter, we need to add a data source and delegate to our Table View. The Table View uses what is called **dynamic cells**, which require us to add these.

1. Select **Table View** in the Outline view, and then, Connections Inspector in the Utilities panel.
2. Click on and drag from `dataSource` to the **LocationViewController** in the Outline view:

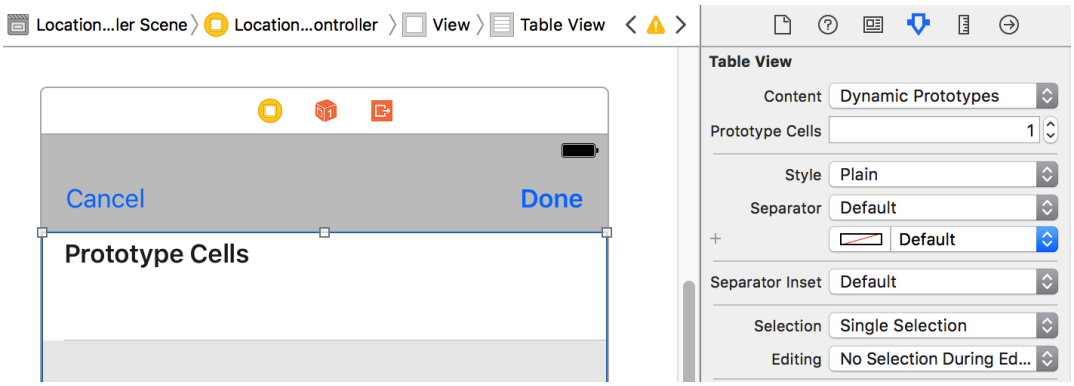


3. Repeat with the delegate property.

Creating a Prototype Cell

In the last chapter, we had a cell inside of our Collection View. In order for us to create a cell inside of the Table View, we need to do the following:

1. In the Outline view, select **Table View** under the **Location View Controller**.
2. In the Utilities panel, select the Attributes Inspector, and update the following value:
    - **Prototype Cells**: `1`

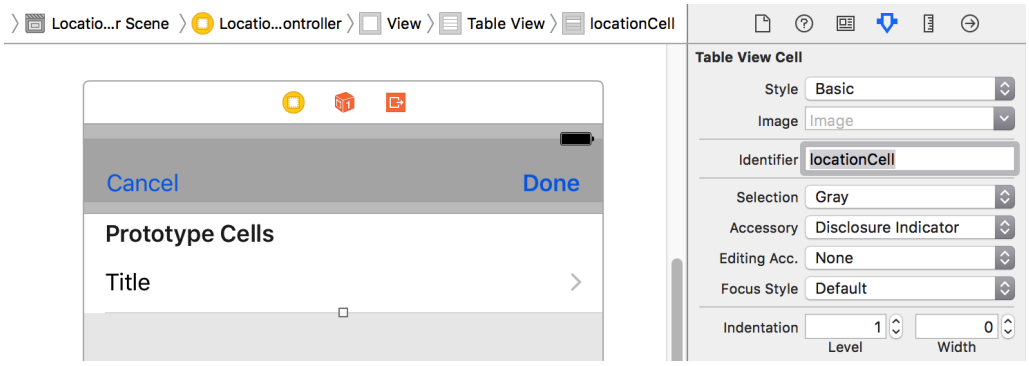


Next, we need to give our cell an identifier so that we can reference it to inform our Table View that this is the cell we want used inside of it:

3. In the Outline view, select **Table View Cell** under **Table View** in the **Location View Controller**.
4. In the Utilities panel, select the Attributes Inspector; if not already selected, and update the following values:
    - **Style**: `Basic`
    - **Identifier**: `locationCell`
    - **Selection**: `Gray`
    - **Accessory**: `Disclosure Indicator`



Basic cell gives us a label we can use without needing to add one in like we did with Collection View.



Digging into Our Table View code

Now, let's display something inside of our Table View.

Before we get started, we are going to clean up our `LocationViewController.swift` file. Delete everything after `viewDidLoad()`:

```
//
//  LocationViewController.swift
//  LetsEat
//
//  Created by Craig Clayton on 11/19/16.
//  Copyright © 2016 Craig Clayton. All rights reserved.
//

import UIKit

class LocationViewController: UIViewController {

    @IBOutlet var tableView:UITableView!

    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.
    }


    /*
    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destinationViewController.
        // Pass the selected object to the new view controller.
    }
    */

}
```

Delete

In order to get data into our Table View, we must conform to a protocol like we did with Collection View. In this case, we must implement `UITableViewDataSource`:

1. First, we need to update our class declaration. We currently have the following:

   ```
   class LocationViewController: UIViewController
   ```

2. We now need to add `UITableViewDataSource` as follows:

   ```
   class LocationViewController: UIViewController,
   UITableViewDataSource
   ```

Next, the `UITableViewDataSource` protocol requires we implement the following three methods after the closing curly brace of `viewDidLoad()`:

```
                                                    A
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return 15
}       C       B

func numberOfSections(in tableView: UITableView) -> Int {
    return 1
}           D                                       E

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

    cell.textLabel?.text = "A cell"          G

    return cell          H
}
```

Let's break down the code to better understand what we are doing:

- **Part A**: `func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int`

  This method tells our Table View how many rows we want to display.

- **Part B**: `return 15`

  Here, we are telling our Table View that we want to display 15 rows.

- **Part C**: `func numberOfSections(in tableView: UITableView) -> Int`

  This method tells our Table View how many sections we want to display. Sections in Table Views are traditionally used as headers, but they can be used however you choose.

- **Part D**: `return 1`

  We are telling our Table View that we only want 1 section.

- **Part E**: `func tableView(_ tableView:UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell`

  Our third and final method gets called for every item we need. Therefore, in our case, it will get called 15 times.

- **Part F**: `let cell = tableView.dequeueReusableCell(withIdentifier: "locationCell", for: indexPath) as UITableViewCell`

  Here, we create a cell every time this method **E** is called, either by taking one from the queue, if available, or by creating a new cell. The identifier, `"locationCell"`, is the name we gave it in Storyboard. Therefore, we are telling our Table View that we want to use this cell. If we had multiple Table Views, we would reference the identifier for the row and section in which we want the cell to display.

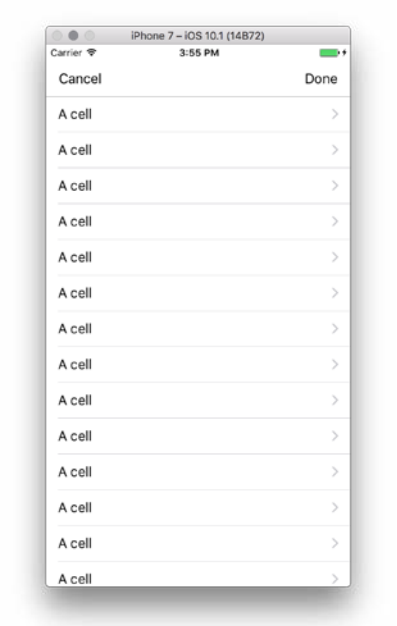
- **Part G**: `cell.textLabel?.text = "A cell"`

  Since we do not have any data yet, we will set our label to `"A cell"`. The variable `textLabel` variable is the default label we get when we selected Basic cell.

- **Part H**: `return cell`

  Finally, after each time we create a new cell, we give the cell back to the Table View in order to display that cell.

Let's build and run the project by hitting the **Play** button (or use *cmd*+ *R*) in order to see what happens. You should now see `"A cell"` repeating 15 times:



Adding Locations to Our Table View

We now have our Table View displaying data, but we need it to display a list of actual locations. Let's update our Table View to show our list of locations:

1. Directly under the `tableView` variable, add the following:

```
let arrLocations = ["Aspen", "Boston", "Charleston", "Chicago",
"Houston", "Las Vegas", "Los Angeles", "Miami", "New Orleans",
"New York", "Philadelphia", "Portland", "San Antonio", "San
Francisco", "Washington District of Columbia"]
```

2. Your file should now look like mine:

```
//  LocationViewController.swift
//  LetsEat
//
//  Created by Craig Clayton on 8/28/16.
//  Copyright © 2016 Cocoa Academy. All rights reserved.
//

import UIKit

class LocationViewController: UIViewController, UITableViewDataSource {

    @IBOutlet var tableView:UITableView!

    let locations = ["Aspen, CO", "Boston, MA", "Charleston, SC", "Chicago, IL", "Houston, TX", "Las
        Vegas, NV", "Los Angeles, CA", "Miami, FL", "New Orleans, LA", "New York, NY", "Philadelphia,
        PA", "Portland, OR", "San Antonio, TX", "San Francisco, CA", "Washington, DC"]

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return 15
    }

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

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

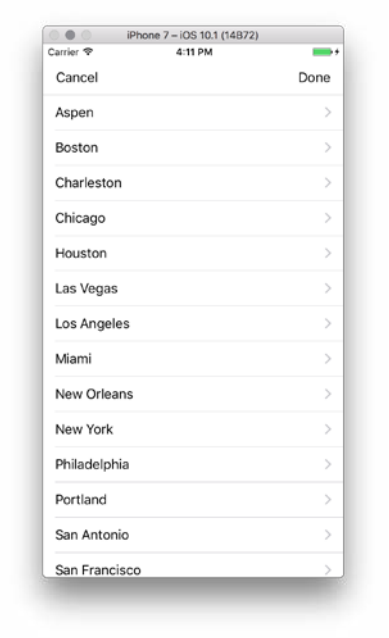
        cell.textLabel?.text = "A cell"

        return cell
    }
}
```

3. Next, in order to update our cell to display the locations, we need to replace the line of code—`cell.textLabel?.text = "A cell"` with the following:

```
cell.textLabel?.text = arrLocations[indexPath.item]
```

Let's build and run the project by hitting the **Play** button (or use *cmd*+ *R*). You should see the following after hitting Select a location in your simulator:



However, there are a couple of problems. If we add another location to the array, it will crash, because we are manually setting the value. In addition, we are just loading this list from an array we built in the app. If we decide to add more locations, we would have to update our cell number count as well as our list of locations. Therefore, we should instead pull our locations from a `plist` as we did in the last chapter. This provides a place where we can quickly add or remove a location from our list.

Creating Our First Property List (plist)

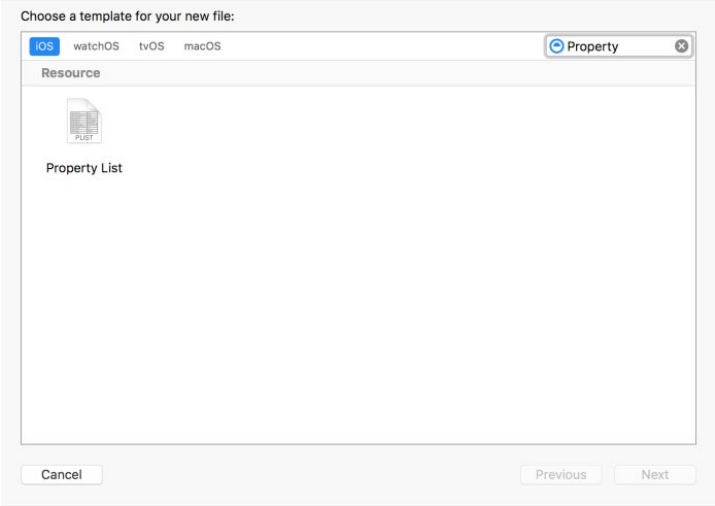
In the last chapter, we used a provided `plist` in order to load our cuisine list. We will do the same in this chapter, but now that you are familiar with what a `plist` is, we will create one from scratch together.



I use `plists` all the time, from creating menus to having a file that holds app settings, like colors or social media URL. I find them very useful, especially if I need to come back later and update or change things.

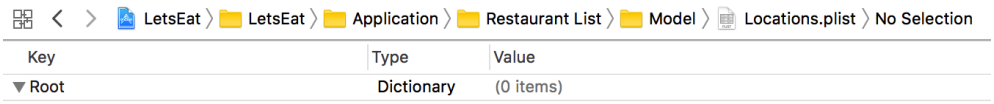
Let's learn how to create a `plist` from scratch. In order to create a `plist` in Xcode, do the following:

1. Right-click on the `Model` folder inside of `Location` and select **New File**.
2. Inside of the **Choose a template for your new file**, select **iOS** at the top, and then, type **Property** in the filter field:



3. Select **Property List** and, then, hit **Next**.
4. Name the file, **Locations**, and hit **Create**.

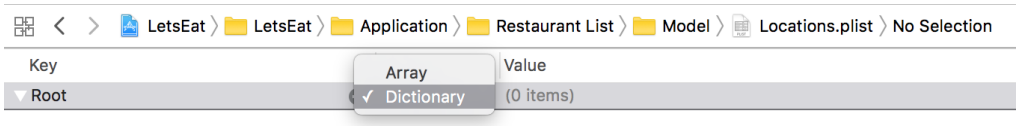
You should now have a file that looks like mine:



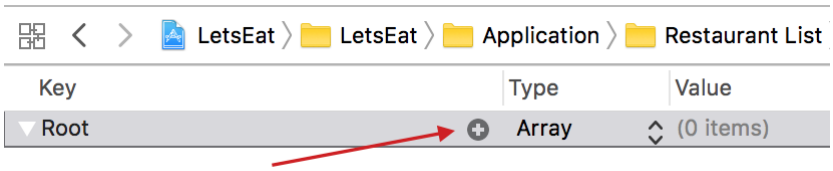
Adding Data to Our Property List

As you learned in the previous chapter, our `plist` has a **Root**; for this new file, we created a **Dictionary** as our Root type. Since we are going to display a list of locations, we will need our Root to be an **Array**.

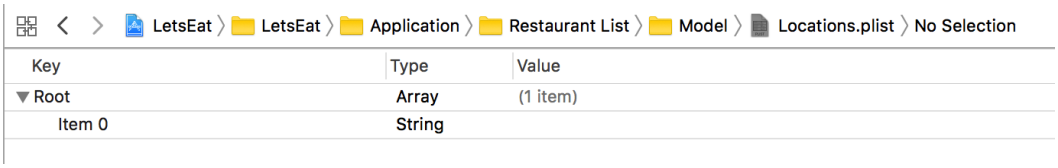
1. Click on **Dictionary** in the plist and change it to **Array**:



2. You will see a plus next to **Array** (if the plus button is not displaying, simply hover your mouse over that line item, and it will appear):



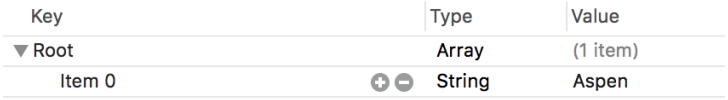
3. Click on the **plus** button, and it will add a new item with a String type:



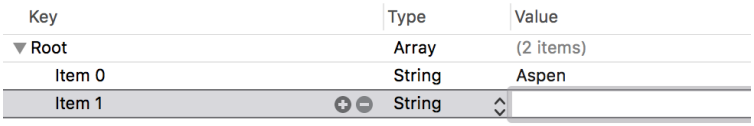
4. Update the **Value** property of the new item by entering `Aspen`:

Key	Type	Value
▼ Root	Array	(1 item)
Item 0	String	Aspen

5. To add another item, click on the plus icon of `Item 0` next to `String` (if the plus icon is not displaying, hover your mouse over that line item to get the plus and minus icons to show):



6. You will see another new item created with a data type of String:



7. Update `Item 1` by entering `Boston` in the **Value** section.
8. Add twelve more items, updating them with the following data for each of their `Value` property:

Key	**Type**	**Value**
Item	String	Charleston
Item	String	Chicago
Item	String	Houston
Item	String	Las Vegas
Item	String	Los Angeles
Item	String	Miami
Item	String	New Orleans
Item	String	New York
Item	String	Philadelphia
Item	String	Portland
Item	String	San Antonio
Item	String	San Francisco

When you are done, your file should look like mine:

Key	Type	Value
▼ Root	Array	(14 items)
Item 0	String	Aspen
Item 1	String	Boston
Item 2	String	Charleston
Item 3	String	Chicago
Item 4	String	Houston
Item 5	String	Las Vegas
Item 6	String	Los Angeles
Item 7	String	Miami
Item 8	String	New Orleans
Item 9	String	New York
Item 10	String	Philadelphia
Item 11	String	Portland
Item 12	String	San Antonio
Item 13	String	San Francisco

We just set up our data source. We now need to create a data manager similar to what we did in the previous chapter.

Creating Our Location Data Manager

Let's create the `LocationDataManager` file:

1. Right-click on the `Model` folder in the `Location` folder and select **New File**.
2. Inside of the **Choose a template for your new file**, select **iOS** at the top and then **Swift File**. Then, hit **Next**:
3. Name this file, `LocationDataManager`, and, then, hit **Create**.
4. We need to define our class definition now, so add the following under the import statement:

```
class LocationDataManager {

}
```

5. Inside of the class declaration, add the following variable in order to keep our array private, as there is no reason to have to access this outside of the class:

```
private var arrLocations:[String] = []
```

6. Now, let's add the following methods after our variable:

```
func fetch() {
    for location in loadData(){
        arrLocations.append(location)
    }
```

```
}

func numberOfItems() -> Int {
    return arrLocations.count
}

func locationItem(at index:IndexPath) -> String {
    return arrLocations[index.item]
}

private func loadData() -> [String] {
    guard let path = Bundle.main.path(forResource: "Locations",
ofType: "plist"),
    let items = NSArray(contentsOfFile: path) else { return [] }

    return items as! [String]
}
```

These methods are basically the same as we had in `ExploreDataManager`, except that we are getting back an array of Strings from our `plist`.

Working with Our Data Manager

We now need to update our `LocationViewController`.

First, because we do not need it anymore, delete the following array we created in the class:

```
let arrLocations = ["Aspen", "Boston", "Charleston", "Chicago",
"Houston", "Las Vegas", "Los Angeles", "Miami", "New Orleans", "New
York", "Philadelphia", "Portland", "San Antonio", "San Francisco",
"Washington District of Columbia"]
```

Next, since we need to create an instance of our data manager in this class, add the following above `viewDidLoad()`:

```
let manager = LocationDataManager()
```

Inside of `viewDidLoad()`, we want to fetch the data for the Table View, so add the following under `super.viewDidLoad()`:

```
manager.fetch()
```

Now, your `viewDidLoad()` should look like the following:

```
override func viewDidLoad() {
    super.viewDidLoad()
manager.fetch()
}
```

For the method `numberOfRowsInSection()`, instead of 15, we will use the following:

```
manager.numberOfItems()
```

Lastly, we need to update our `cellForRowAt`. Replace cell.`textLabel?.text = arrLocations[indexPath.item]` with the following:

```
cell.textLabel?.text = manager.locationItem(at:indexPath)
```

Your `cellForRowAt` should now look like this:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier:
"locationCell", for: indexPath) as UITableViewCell
    cell.textLabel?.text = manager.locationItem(at:indexPath)

return cell
}
```

Let's build and run the project by hitting the **Play** button (or use *cmd* + *R*). We should still see our locations, but now they are coming from our `plist`.

Summary

In this chapter, we worked with a Table View that has dynamic cells, which allow the Table View to change based off the data. We also worked with unwinding segues. Later, we will actually pass data that we need through our segues. Along with segues, we looked at plists, learning how to create them as well as how to add data to them. Finally, we created our Locations Data Manager, which is responsible for giving data to the View Controller.

In the next chapter, we will work with a Table View that has static cells in order to build out our restaurant detail. Static cells are great for forms or detail views. We could actually do the restaurant detail using a Collection View; however, a static Table View will work well and be less complicated.

At this point, before moving onto the next chapter, you may want to get the starter project for this chapter and try to do it again without using the book as your guide. This will really help solidify your understanding of what you learned.

9
Working More with Lists

When Storyboards were first introduced in late 2011, I was really excited to use them. However, some developers did not (and still do not) like Storyboards for many different reasons. I am a visual person; so for me, Storyboards give me the ability to see my app UI and be able to design it without having to run the app every time.

With the introduction of Storyboards, came static Table Views. For me, static Table Views changed the game, because I could create a complex layout all within Storyboard without any, or with very little code. Static Table Views can be used in many different ways, including for login screens, project settings, detail views, and so much more.

In this chapter, we are going to work with static Table Views; as you will see, you can create complex looks with very little code.

We will cover the following in this chapter:

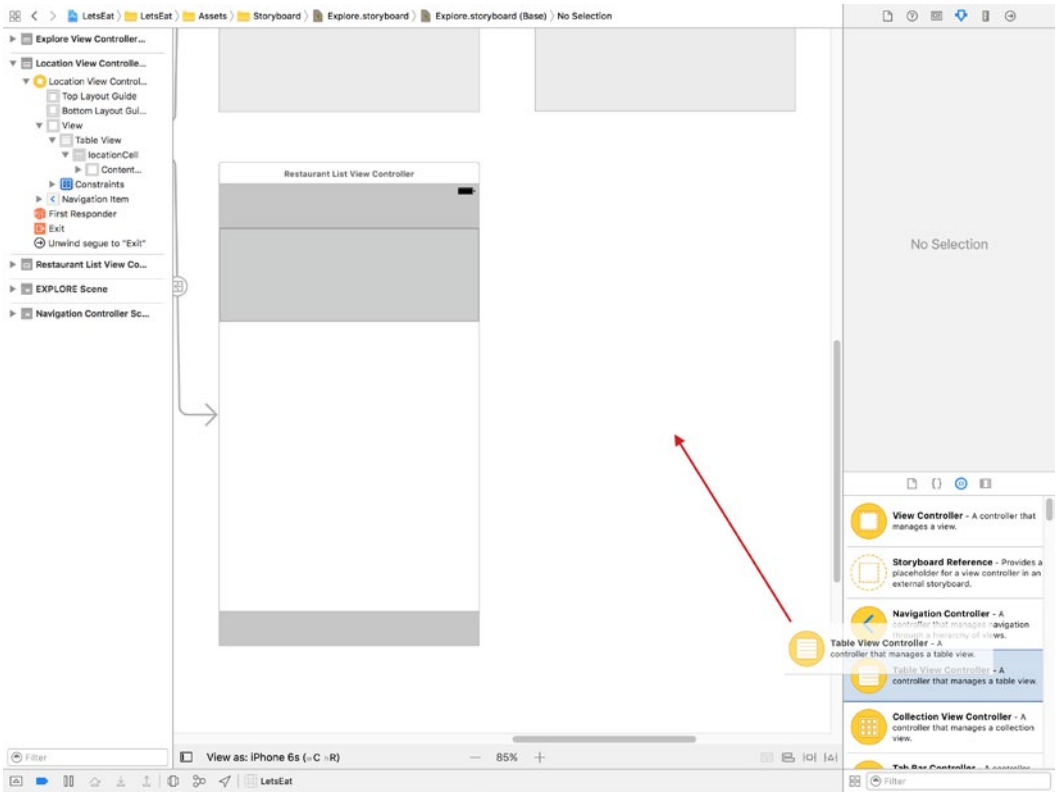
- Creating our Restaurant detail
- Setting up our static Table View

Creating our Restaurant detail

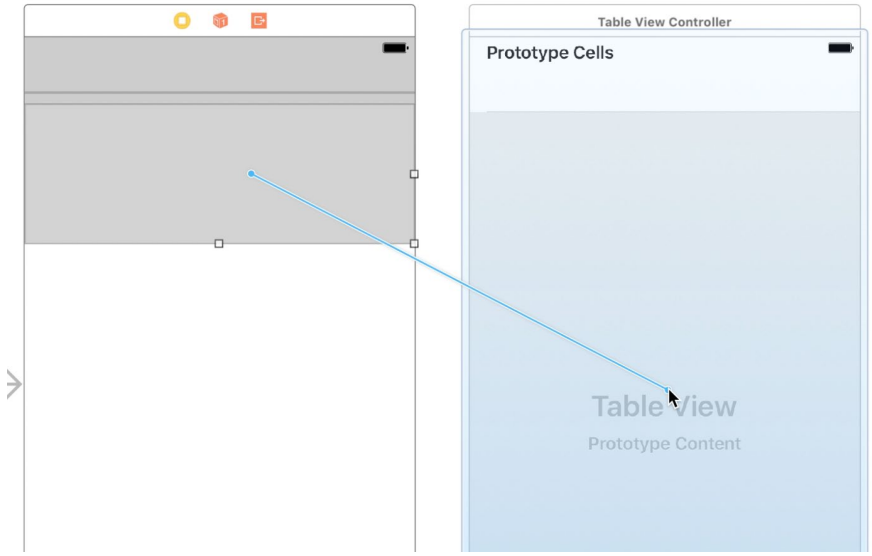
Before we can get started creating a static Table View, we need to set up our Restaurant detail page. Currently, if you select a restaurant item, nothing happens. We need the user to be able to click on a restaurant item and then be directed to a restaurant detail page.

1. Open `Explore.storyboard`, and in the **Utilities** panel, in the filter field of the **Object** library, type `tableview`.

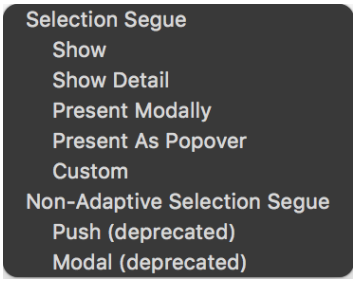
2. Drag out a Table View Controller (not a Table View) next to our Restaurant List View Controller:



3. Now, *CTL* drag from our `RestaurantListCell` to the Table View Controller we just added to the Storyboard:



4. In the screen that appears, select **Show**:



Let's build and run the project by hitting the play button (or use *CMD* + *R*). You should now be able to select a restaurant (one of the gray cells you see after picking a type of cuisine); it will open up what will be your restaurant detail page, which, for now, is an empty Table View.

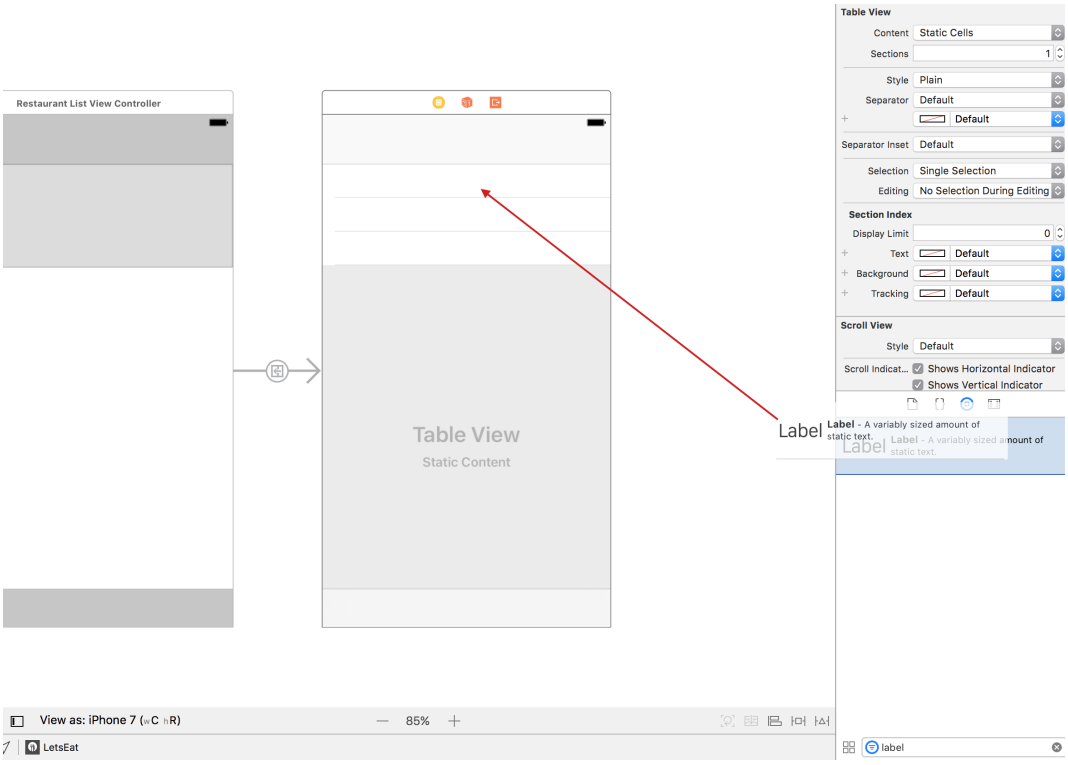
Setting up our static Table View

Currently, our restaurant detail is going to a dynamic Table View; we want it to go to a static Table View:

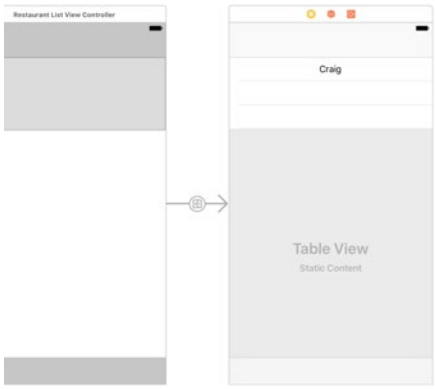
1. In `Explore.storyboard` Outline view, select the **Table View** in the `UITableViewController` we just dragged out.
2. In the **Utilities** panel, select the **Attributes Inspector** and change **Content** from **Dynamic Prototypes** to **Static Cells**. Our Table View has now changed to static cells, which means we can drag out items right into the cell.

Now that our Table View is using static cells, we can see how it works before we actually start working with it.

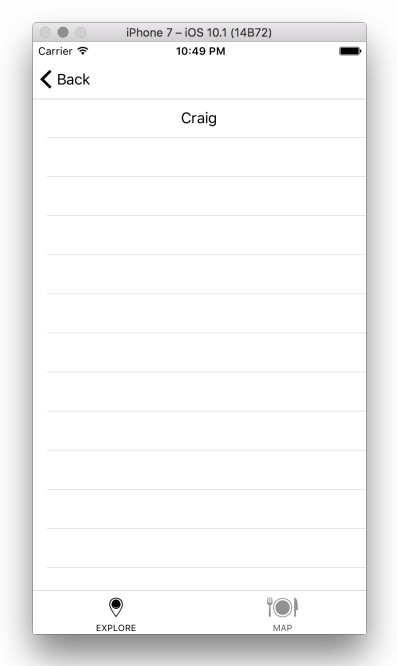
1. In the **Utilities** panel, select the **Object library**, and in the filter field, type `label`.
2. Drag and drop the Label directly into any of the three Table View cells:



3. Double click on the **Label** and put your name inside it:



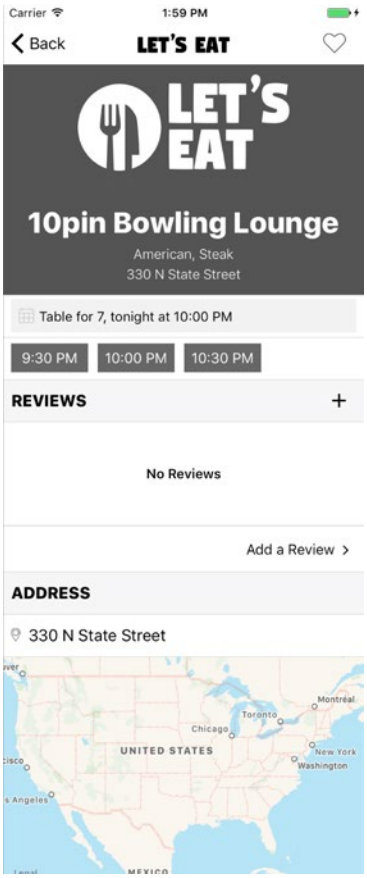
4. Build and run the project by hitting the play button (or use *CMD + R*). When you go to restaurant detail, you will see your name in the Table View, and you did not even have to add one line of code. This is a static Table View:



Let's delete the label we just added and start looking at how to create our restaurant details.

Exploring Restaurant details

Before we start working on the restaurant details, let's take a look at what our restaurant details look like in the finished app:



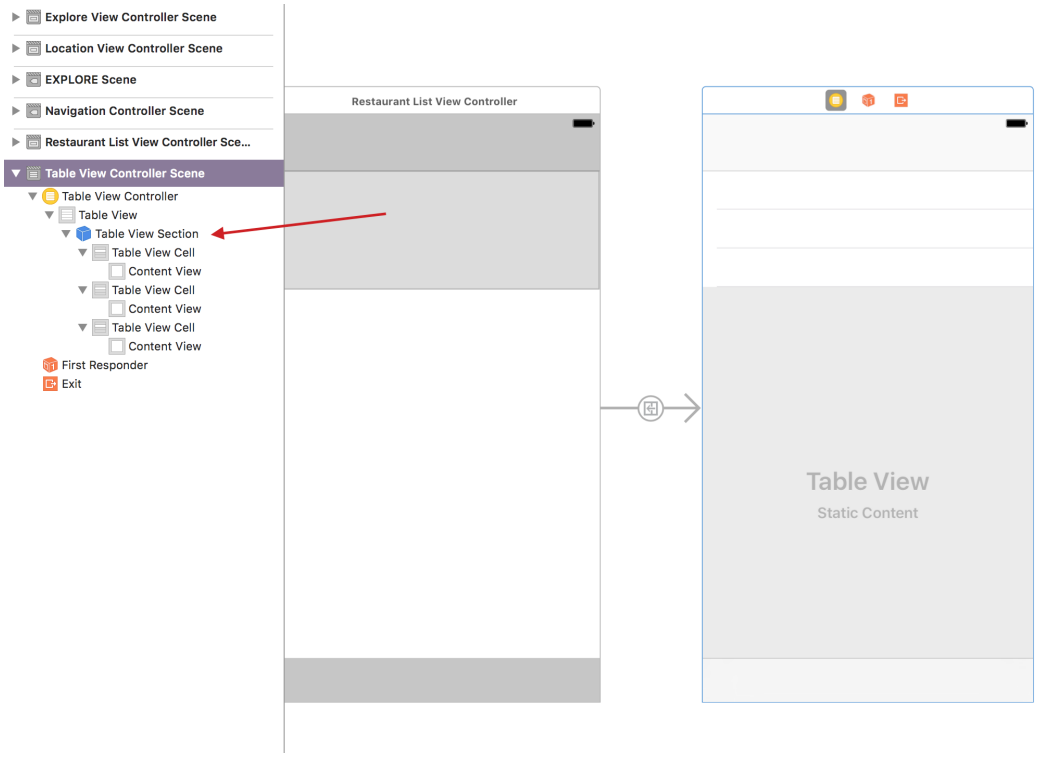
It would be complicated to create this layout in code; however, using static Table Views will make it really easy for us to build this out.



We will use the Outline view exclusively for static Table Views because it makes it easier to access elements.

The first thing we should do is set up our cells. We will have a total of eight cells in our Table View.

1. Select `Explore.storyboard`.
2. In the Outline View, make sure that the disclosure arrows are down for **Table View Controller Scene**, **Table View Controller**, and **Table View**. Select **Table View Section** inside the **Table View**.

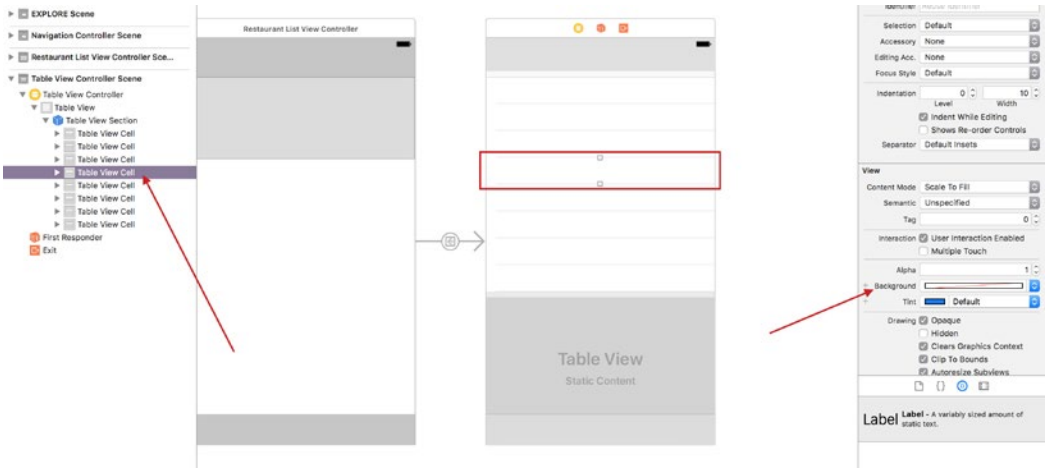


3. In the **Attributes Inspector** of the **Utilities** panel, update **Rows** under **Table View Section** to `8`. Then, hit *Enter*.
4. These eight cells are all we need to create our restaurant details. Let's create our section headers next.

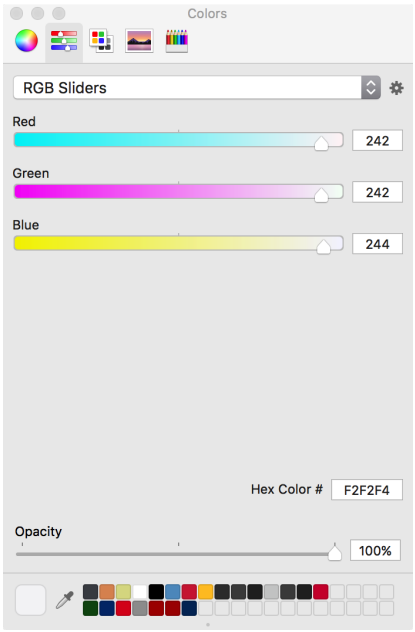
Creating our section headers

We are going to make headers for rows 4 and 6. Let's see how we do it:

1. In the **Outline** view, select the 4th row; then, in the **Attributes Inspector**, select **Background**:



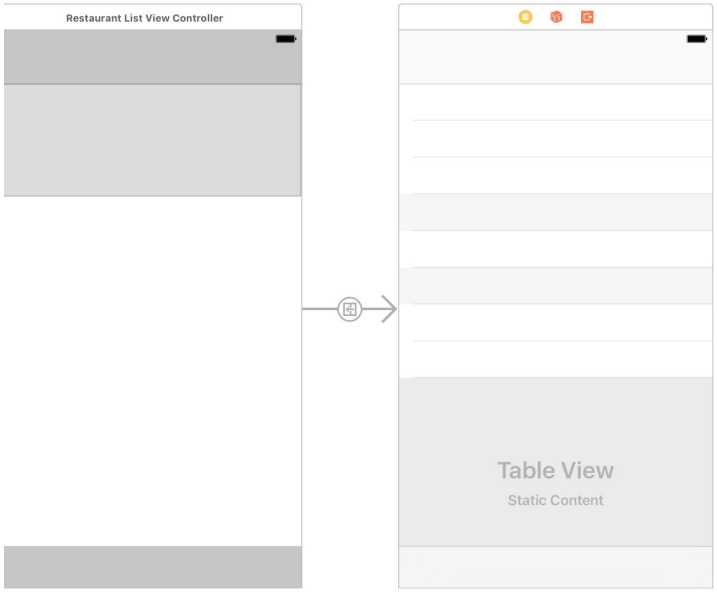
2. Under the **Color Sliders** tab, make sure that RGB Sliders in the drop-down menu is selected. Then, set the **Hex Color #** to `F2F2F4`:





You sometimes have to set the Hex Color two times. I find it easier to set the cell to any color first, and then set it to the color I want as a second step. If you have trouble, try this method.

3. Now, select the 6th row and change its **Background color** to `F2F2F4`:

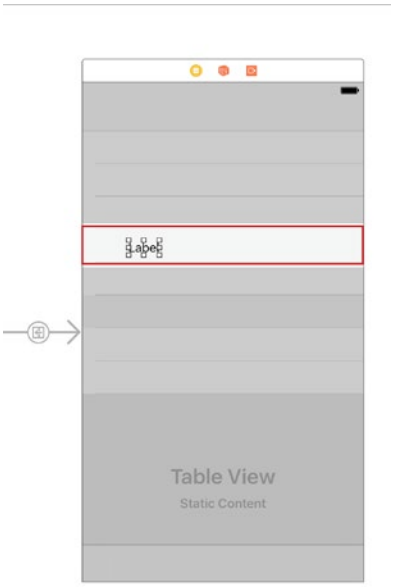


Adding our labels

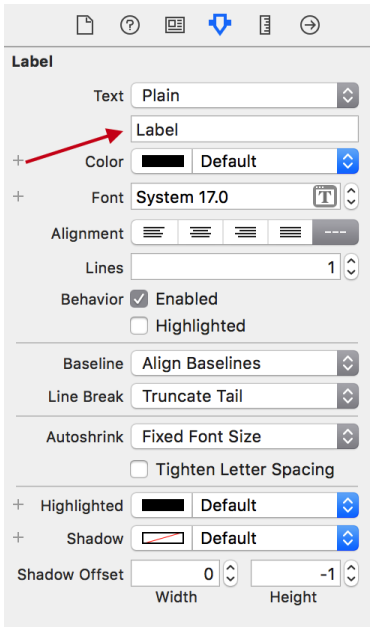
We now have the header background colors set. Let's add labels in, as well:

1. In the `Object library` filter field in the **Utilities** panel, type `label`.

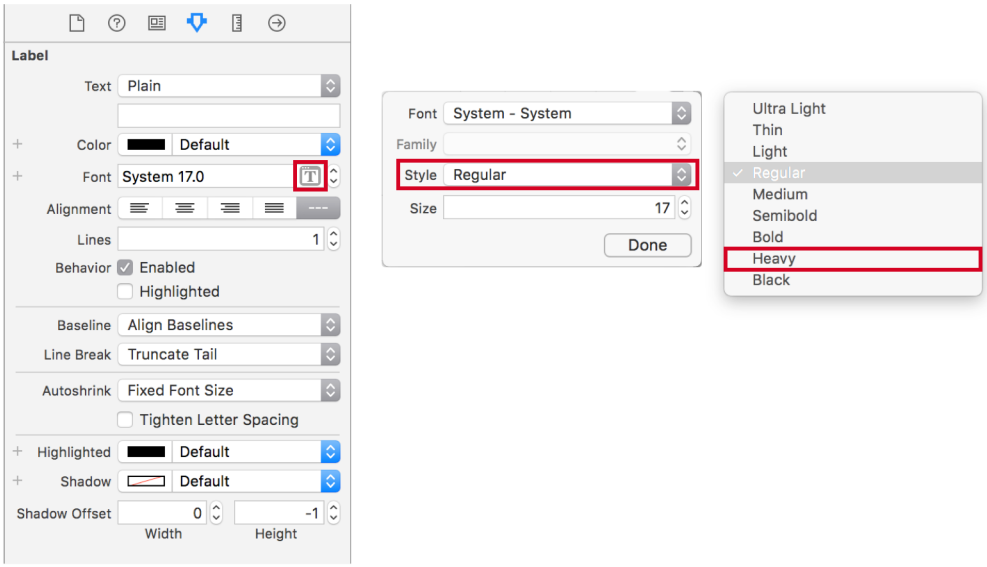
2. Drag a Label to the 4th row:



3. Under the **Label** section at the top of the **Utilities Panel Attributes Inspector**, change the word `Label` under **Text** to `REVIEWS`:

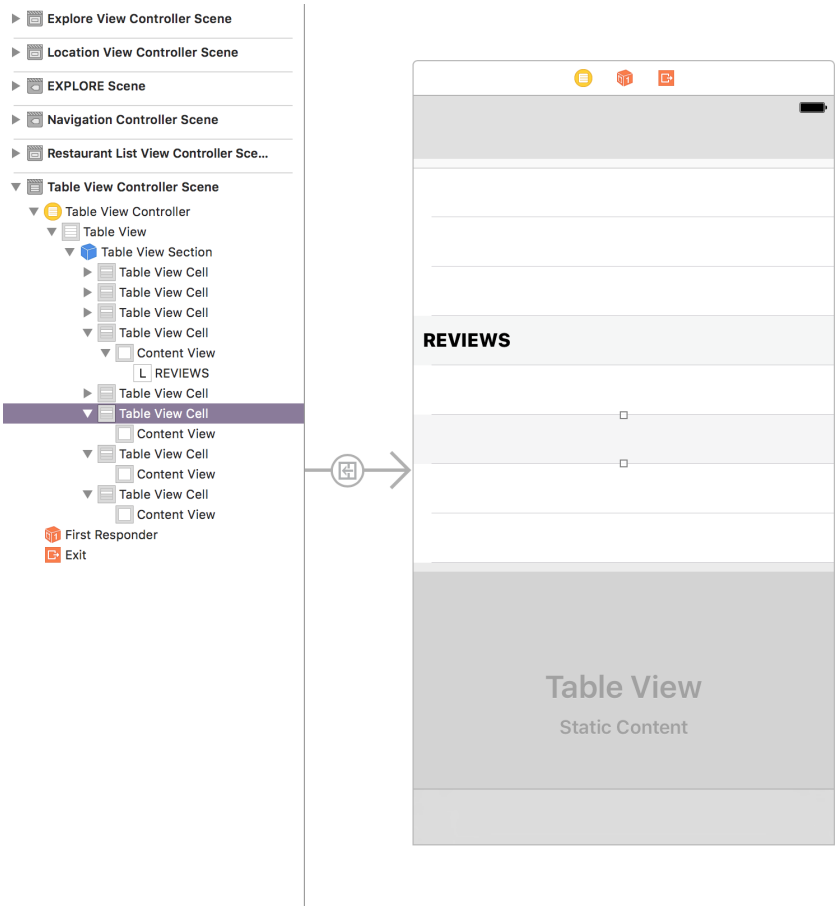


4. Now, let's update the **Font** in the **Attributes Inspector** by selecting the **T** icon and changing the **Style** to **Heavy**:

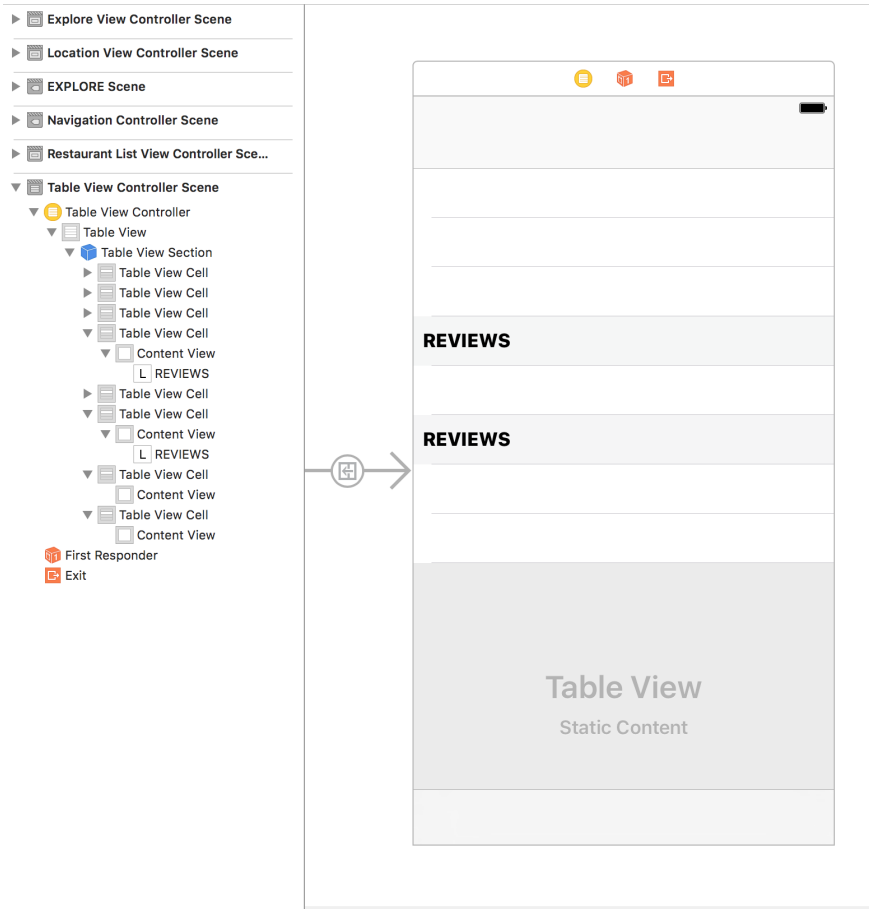


5. Select the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: 8
    - **Y**: 11
    - **Width**: 320
    - **Height**: 21
6. Now, we have to do the same for the next title.
7. To make it easier, select the current label and hit CTRL + *C* to copy.

8. In the **Outline** view, select the 6th row and make sure that the disclosure arrow is down for that **Table View Cell**:



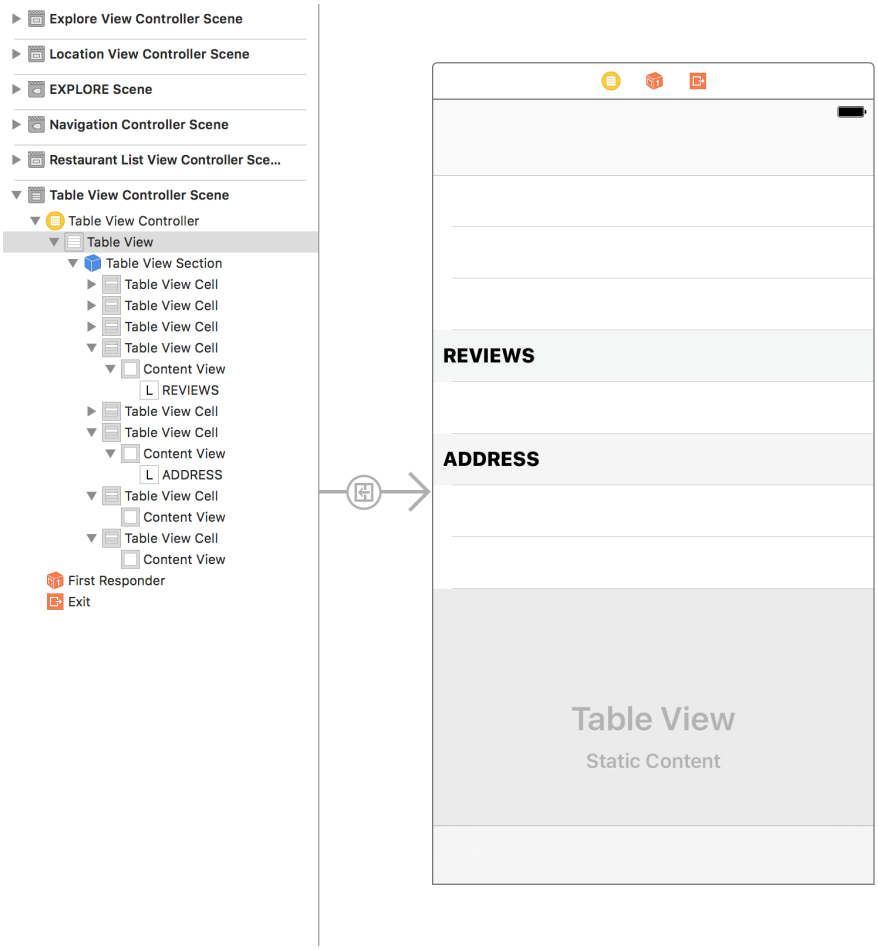
9. Then, select the **Content View** of that **Table View Cell** and hit CTRL + *V* to paste:



10. Note that, when you paste the **Label**, it might not be put in the correct position. Make sure that your values match the following in the **Size Inspector**:

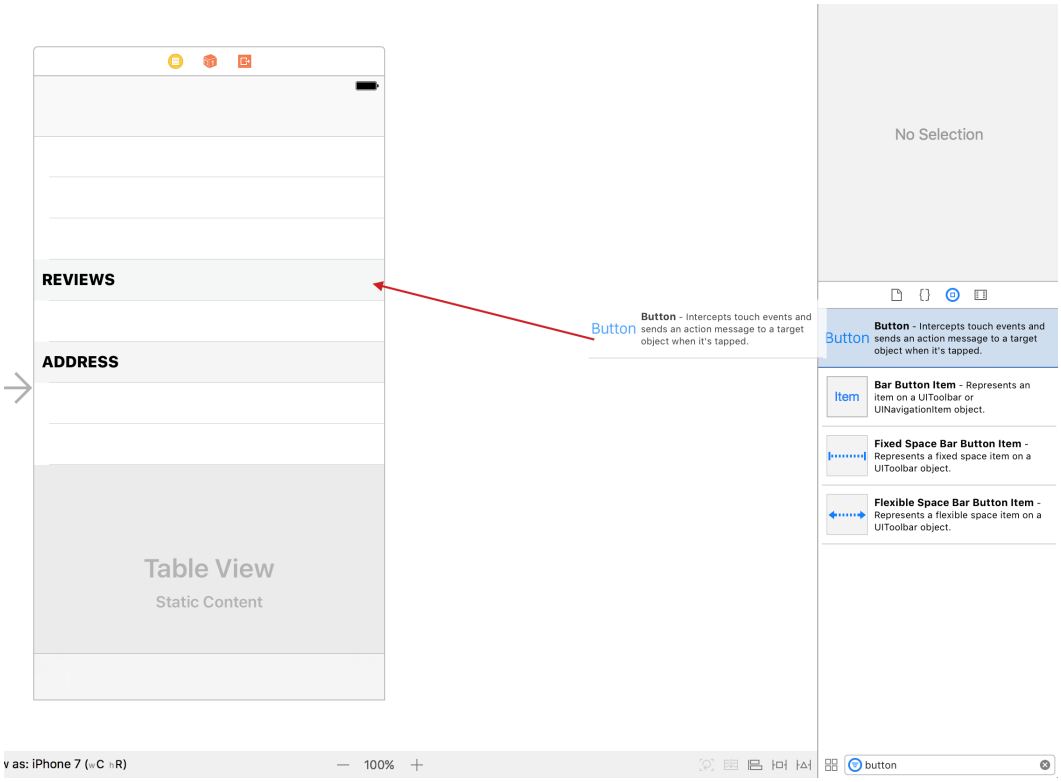
    - **X**: 8
    - **Y**: 11
    - **Width**: 320
    - **Height**: 21

11. Double-click on the **UILabel** and change the **Label** text from `REVIEWS` to `ADDRESS`. Your headers should now look like mine:

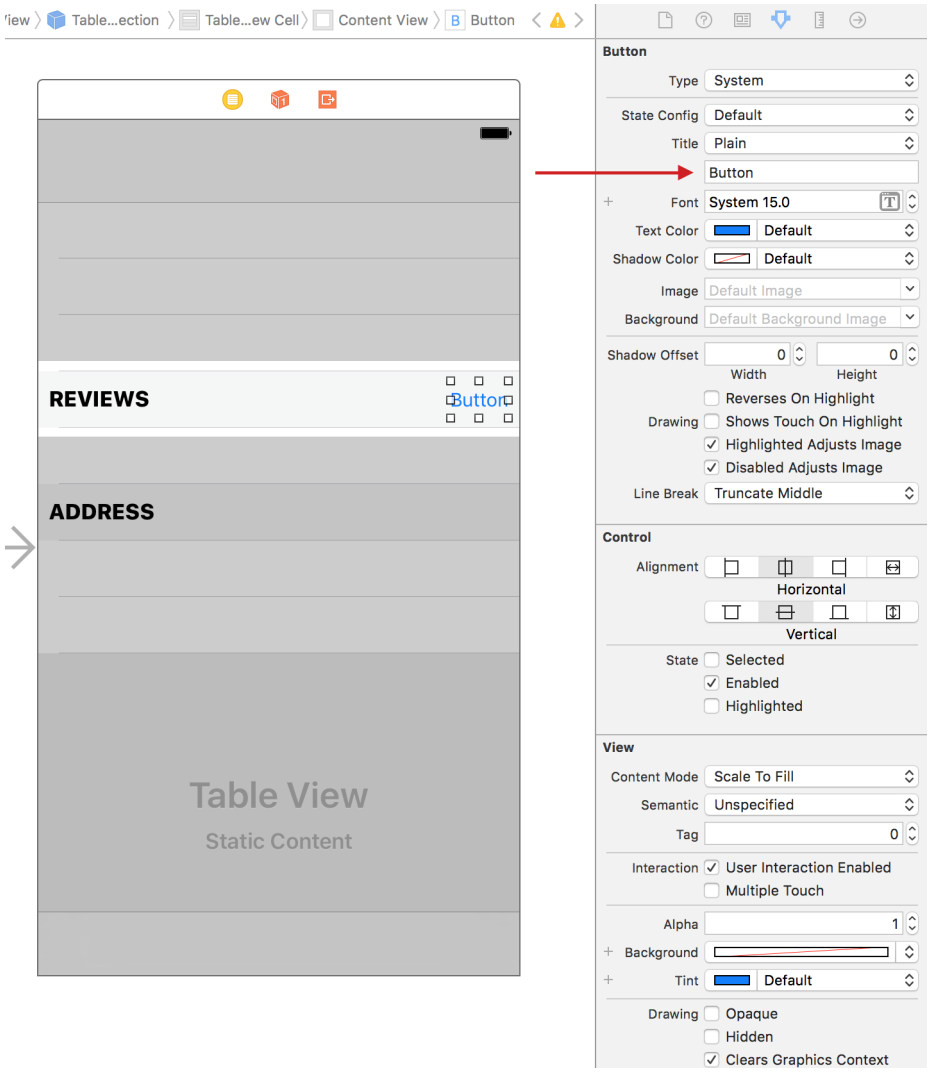


We next need to add a plus button for the **Reviews** header:

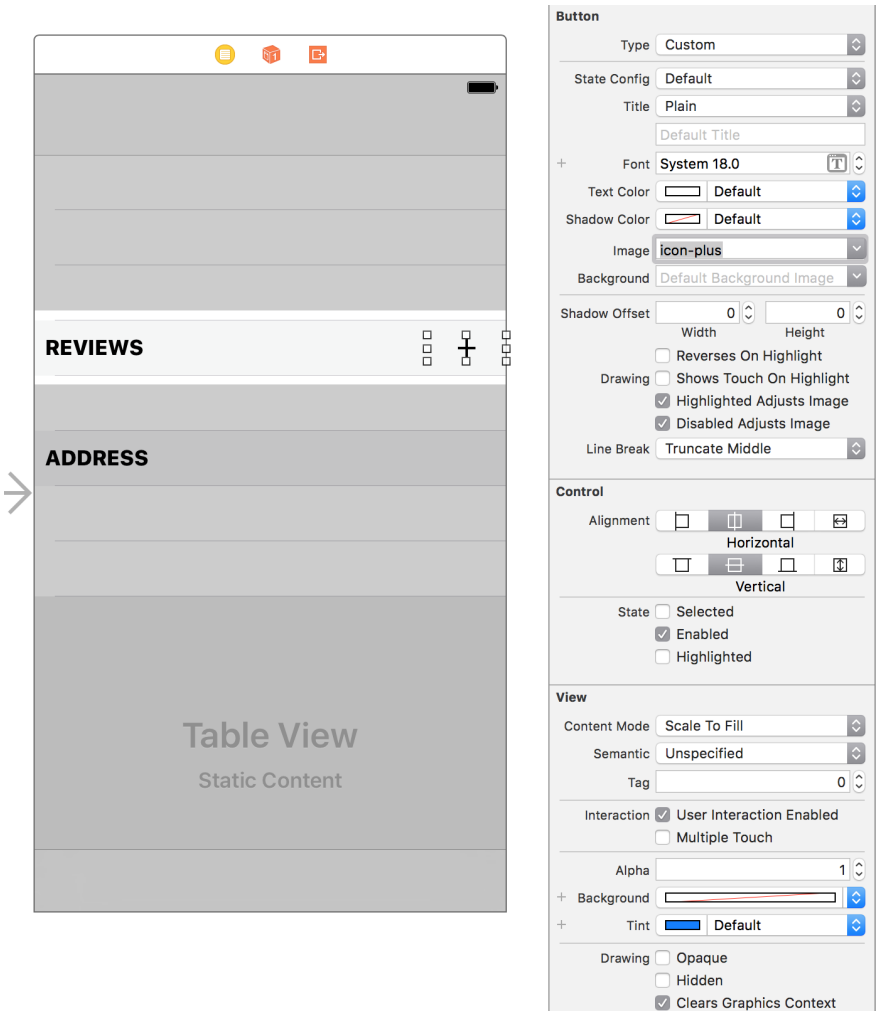
1. In the **Utilities** panel, select the **Object library**, and in the filter area, type `button`.
2. Drag and drop a **Button** into the **Reviews** header:



3. With the **Button** selected, in the **Attributes Inspector** of the **Utilities** panel, delete the `Button` text under **Title**:

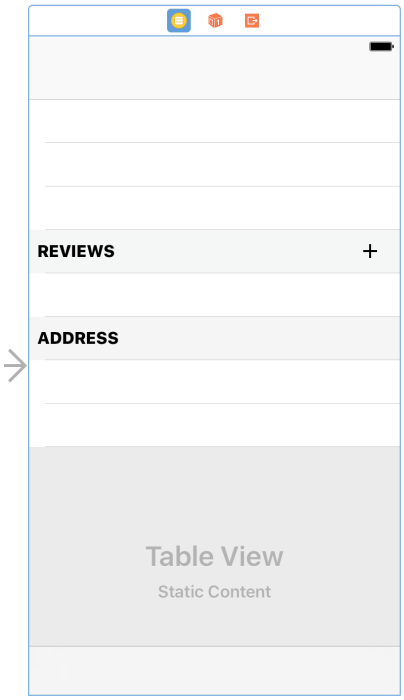


4. Under the **Image** field, type `icon-plus`:



5. In the **Size Inspector** of the **Utilities** panel, update the following values:
    - **X**: 323
    - **Y**: 11
    - **Width**: 44
    - **Height**: 21

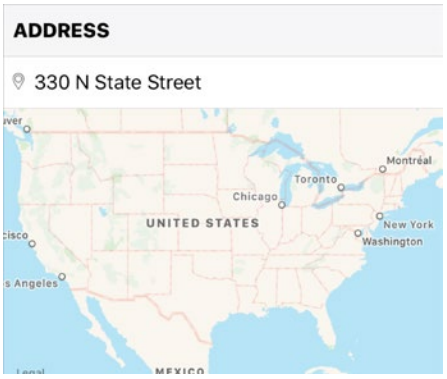
6. When you are done, your header should look as follows:



We now have our headers setup. Let's move onto the rest of our Table View setup.

Address section

Let's start with the Address section:



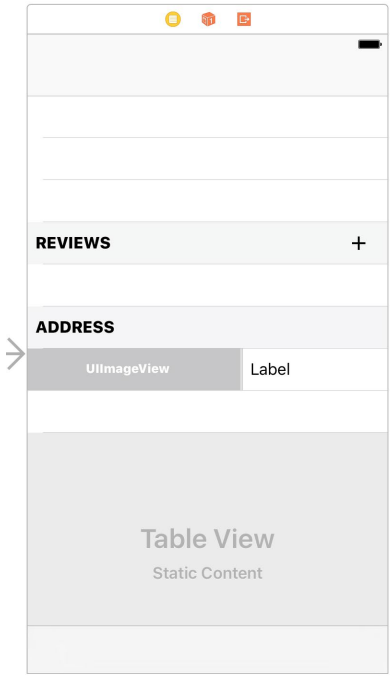
In this section, we are going to add three UI elements: image, label, and map.

1. In the filter field of the **Object** library in the **Utilities** panel, type `image`.
2. Drag the **UIImage** into the **Content View** of the 7th row.



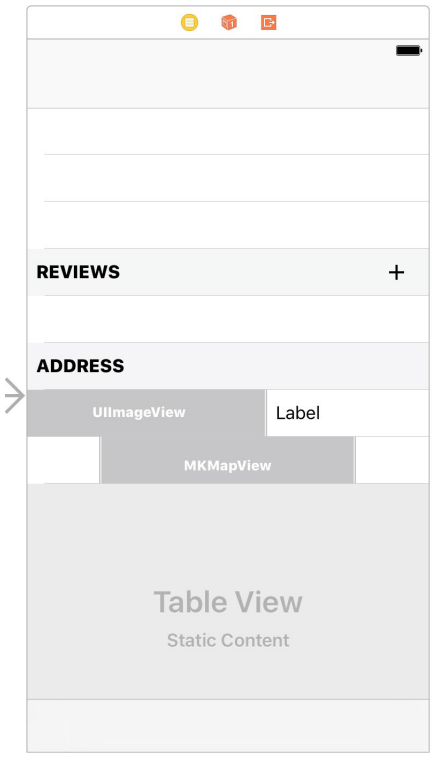
To ensure you are dragging the UI element into the correct area, it is easier to drag it into the **Outline** view.

3. Next, in the **filter** field, type `label`.
4. Drag the **UILabel** into the **Content View** of the 7th row:



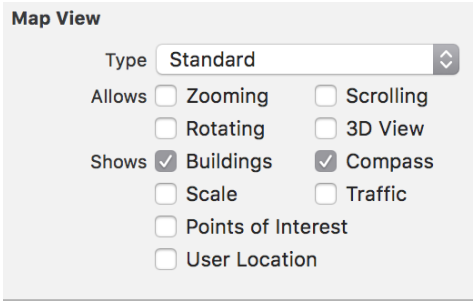
5. Now, in the **filter** field, type `map`.

6. Drag the MapKit into the **Content View** of the 8th row:



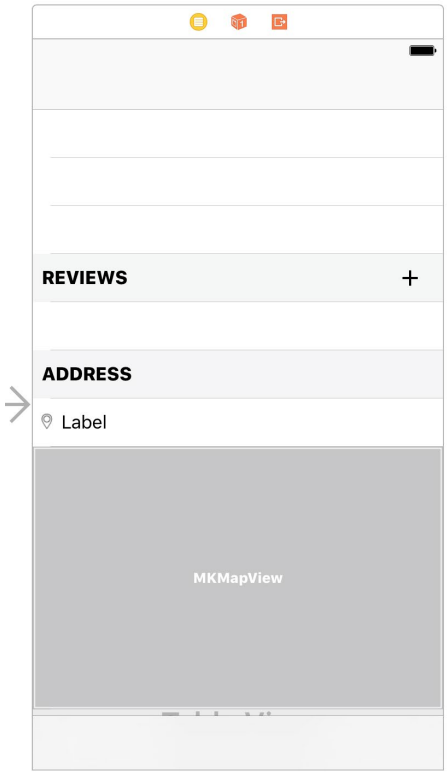
We now have all of our UI elements for the address section. Let's properly size them and place them in the correct spots:

1. Select **Map View** in the **Outline** view, if it is not already selected, and in the **Utilities** panel, select the **Attributes Inspector**.
2. Uncheck everything in **Map View**, except **Buildings** and **Points of Interest**:



3. In the **Utilities** panel, select the **Size Inspector** and update the following values:
    - **X**: 0
    - **Y**: 0
    - **Width**: 375
    - **Height**: 240
4. In the **Outline** view, select the **Table View Cell** that contains the M**ap Kit View**, and in the **Size Inspector**, under **Table View Cell**, update **Row Height** to `240`, then hit *Enter*.
5. Next, select the 7th row in the **Outline** view. Select the **Image View** and open the **Size Inspector** in the **Utilities** panel. Update the following values:
    - **X**: 8
    - **Y**: 14
    - **Width**: 10
    - **Height**: 14
6. With **Size Inspector** open, select the **Label** and update the following values:
    - **X**: 26
    - **Y**: 11
    - **Width**: 340
    - **Height**: 21
7. Finally, reselect the **Image View** in the **Outline** view. Then, select the **Attributes Inspector** in the **Utilities** panel. Update **Image** to `icon-map-on`.

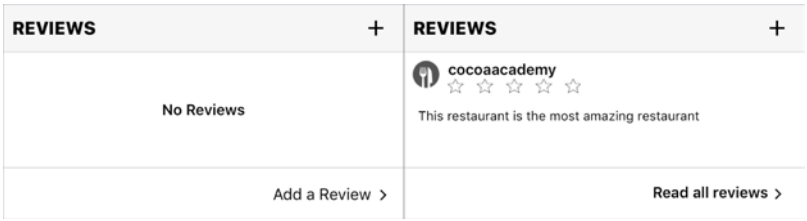
We now completed the **Address** section. We will add text to our **Label** in this section later through coding. Your address section should now look like mine:



With our address section completed, let's work on the **Reviews** section next.

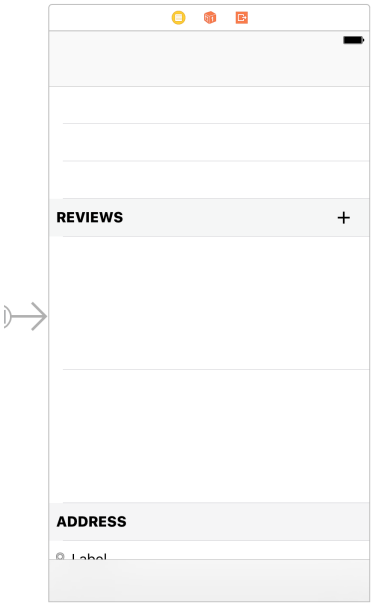
Creating Reviews

In this section, we will add the ability for users to add restaurant reviews. Let's look at the design and see all the functionality:



Here, we have two views, one for when there are no reviews, and one in order to display a review. The simplest way to do this is to add two UIViews; then, we can hide or show based on the number of reviews we have. To make designing easier, we will put each UIView in its own row. Then, we will combine them into one when we are finished setting up each one.

1. In the **Outline** view, select the 5th **Table View Cell**; and, in the **Size Inspector** of the **Utilities** panel, update **Row Height** to `156` and press *Enter*.
2. Again, in the **Outline** view, select the 5th **Table View Cell**. Now, hit CTRL + *C* and then CTRL + *V*. This will give us another cell:

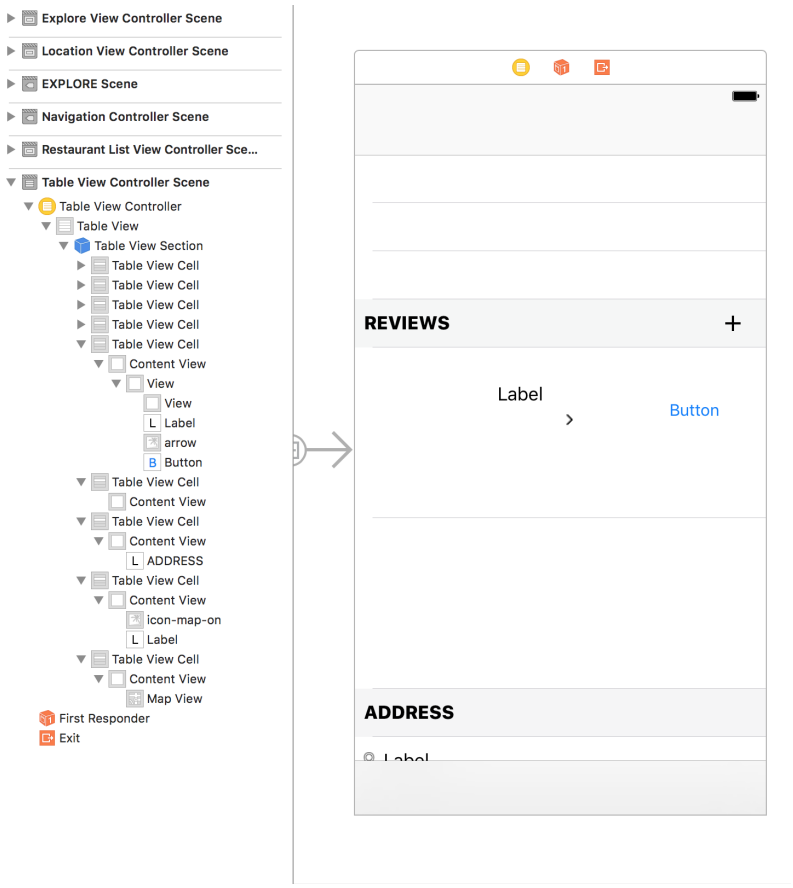


3. Next, in the **Utilities** panel, type `uiview` in the **filter** field.
4. Drag a **View** into the first cell under the **Review** header. This **View** will be our container into which we drag everything.
5. With the **Size Inspector** open, select the **View** and update the following values:
    - **X**: 0
    - **Y**: 0
    - **Width**: 375
    - **Height**: 155
6. Drag another **View** so that it is inside the one just added.
7. Next, in the **filter** area of the **Object** library, type `label`.
8. Drag a **Label** so that it is inside our initial container.
9. Select the Media library icon in the **Utilities** panel. Then, in the filter area, type `arrow`.
10. Drag an arrow into the same container.
11. Finally, select the Object library and, in the filter area, type `button`.
12. Drag a Button into the same container.



Because we have a second View inside our initial View container, you might have accidentally dragged some of your components into this second View. Check your Outline view to confirm that you did not do this.

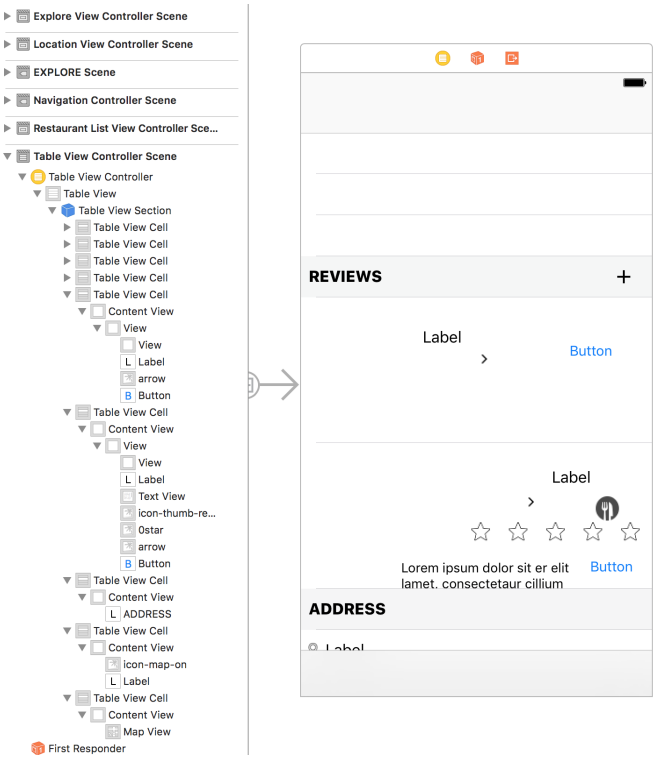
Make sure that your Outline view and cell look like the following:



Let's work on the second cell we created under the **Review** header; then, we will go back and position everything correctly.

1. In the **Utilities** panel, type `uiview` in the **filter** field.
2. Drag a **View** into the second cell under the **Review** header. This **View** will again be our container into which we drag everything.
3. With the **Size Inspector** open, select the **View** and update the following values:
    - **X**: 0
    - **Y**: 0
    - **Width**: 375
    - **Height**: 155

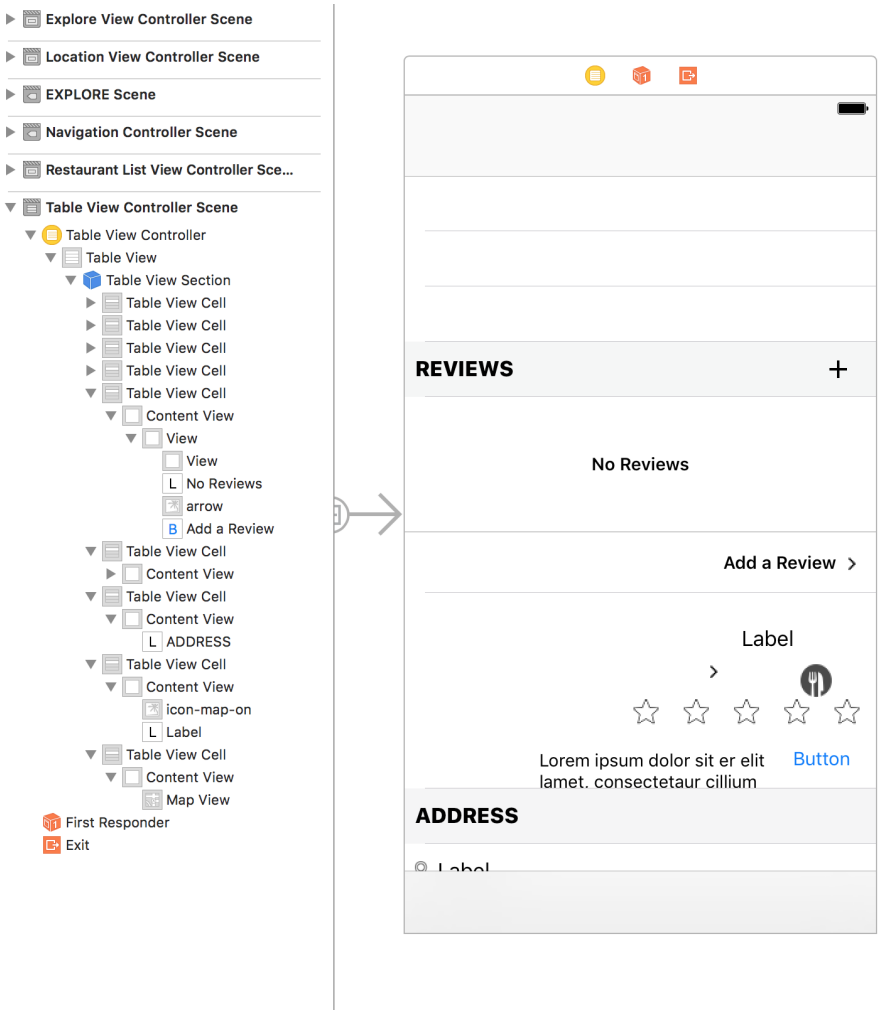
4. Drag another **View** so that it is inside the one just added.
5. Next, in the filter area of the Object library, type `label`.
6. Drag a **Label** into our initial container.
7. Now, in the filter area, type `text view`.
8. Drag a Text View into the same container.
9. Select the Media library icon in the **Utilities** panel. Then, in the filter area, type `icon-thumb-review`.
10. Drag an icon-thumb-review into the same container.
11. Now, in the filter area, type `0star`.
12. Drag 0star into our container.
13. Then, in the filter area, type `arrow`.
14. Drag an arrow into our container.
15. Finally, select the Object library, and in the filter area, type `button`.
16. Drag a Button into our container.
17. Your Outline view and cell should now look as follows:



We now have all of our elements in our containers. Let's properly size and place these elements. We will start working with the **No Review** container:

1. Inside the first cell under the **Review** header, in **Outline** view, select the second **View**.
2. With the **Size Inspector** open in the **Utilities** panel, update the following values:
    - **X**: 0
    - **Y**: 107
    - **Width**: 375
    - **Height**: 1
3. Select the **Attributes Inspector**, click on the **Background** and set the **Hex Color #** to `C4C4C4` under **RGB** Sliders in the drop-down menu.
4. Next, select the **Label**, open the Size Inspector in the **Utilities** panel and update the following values:
    - **X**: 24.5
    - **Y**: 45
    - **Width**: 326
    - **Height**: 17
5. Select the **Attributes Inspector** and update the **Font** by selecting the **T** icon and changing the **Style** to **Semibold** with a font size of `14`.
6. Change the **Alignment** to **Center**.
7. Update the Label **Text** from `Label` to `No Reviews`.
8. Next, select the arrow, open the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: 353
    - **Y**: 128
9. Finally, select the Button, and with the **Size Inspector** open, update the following values:
    - **X**: 234
    - **Y**: 117
    - **Width**: 129
    - **Height**: 30

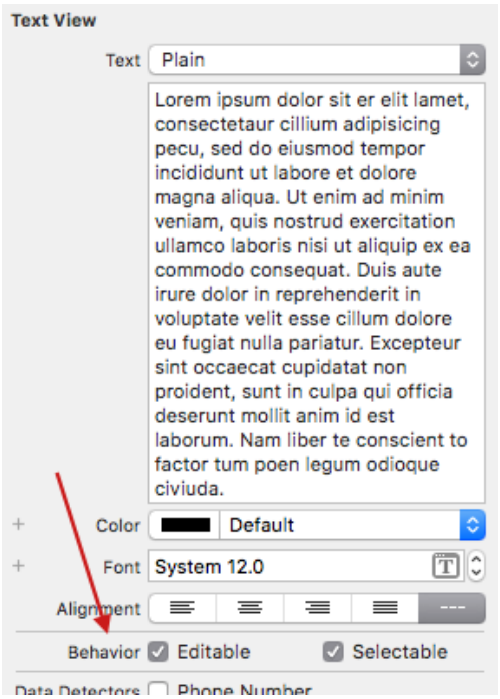
10. Select the **Attributes Inspector**, and update the Button **Title** from `Button` to `Add a Review`.
11. Update the **Font** to **Semibold** size `14`.
12. Click on **Text Color** and set the **Hex Color #** to `000000` under **RGB** Sliders in the drop-down menu.
13. Your first cell is complete and should look like the following:



Now, we will work on our next cell—the **Review** container—and position everything correctly:

1. Inside the second cell under the **Review** header, in **Outline** view, select the second **View**.
2. Open the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: 0
    - **Y**: 107
    - **Width**: 375
    - **Height**: 1
3. Select the **Attributes Inspector**, click on the **Background**, and set the **Hex Color #** to `C4C4C4` under **RGB** Sliders in the drop-down menu.
4. Next, select the **Label** and, with the **Attributes Inspector** still selected, update the **Font** to **Semibold** size `14`.
5. Open the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: 41
    - **Y**: 8
    - **Width**: 326
    - **Height**: 16
6. Now, select your **Text View**, open the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: 8
    - **Y**: 44
    - **Width**: 359
    - **Height**: 60
7. Select the **Attributes Inspector** and ensure that the font is **System** size `14`.

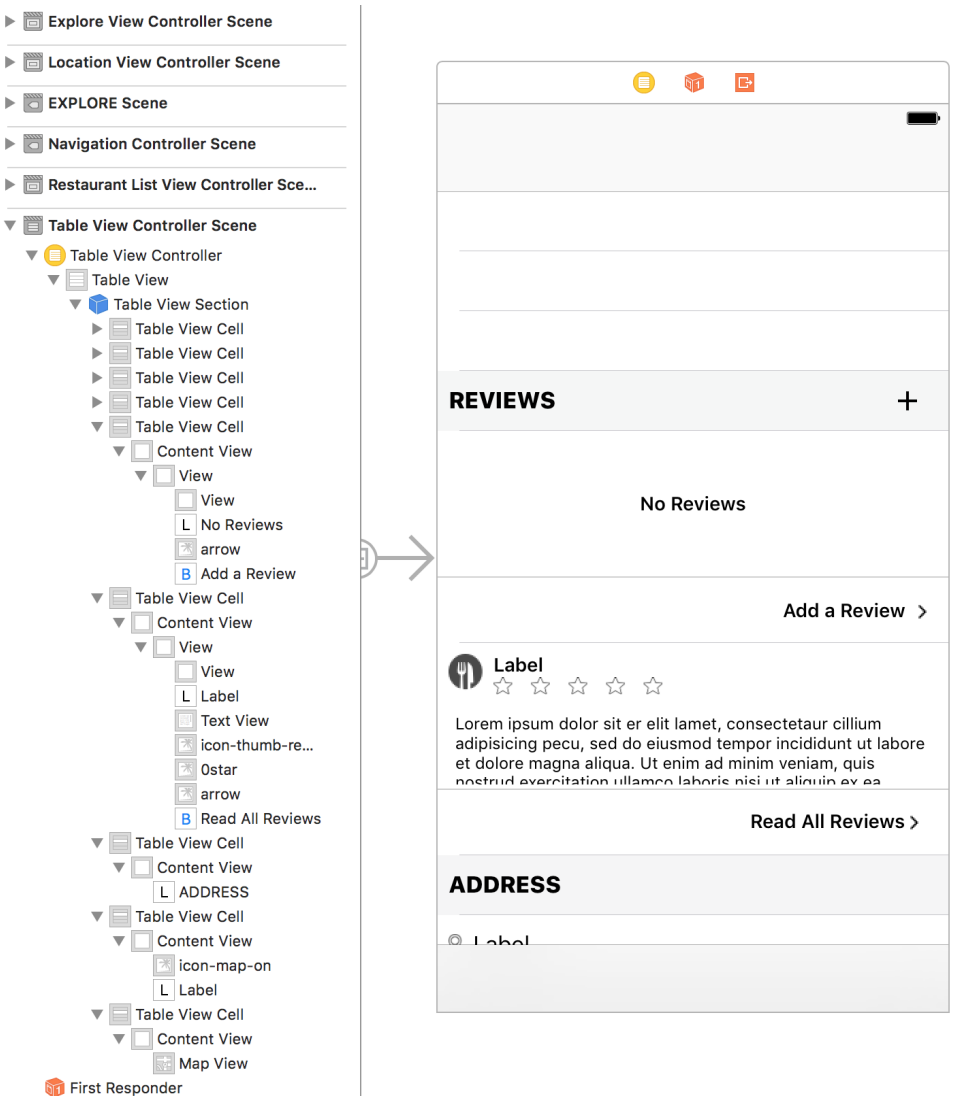
8. Under **Behavior** in the **Attributes Inspector**, uncheck **Editable** and **Selectable**:



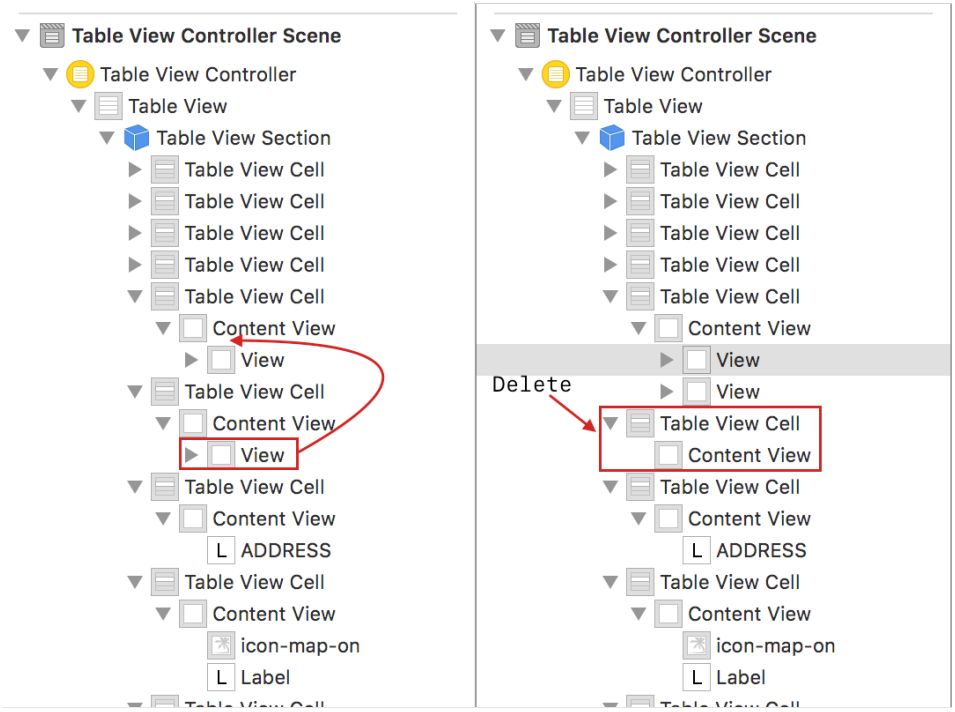
9. Next, select **icon-thumb-review** in the **Outline** view, open the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: 8
    - **Y**: 8

10. Now, select **0star** in the **Outline** view, and with the **Size Inspector** open, update the following values:
    - **X**: 41
    - **Y**: 25
    - **Width**: 124
    - **Height**: 13
11. Select the **Attributes Inspector** and update **Content Mode** to **Scale** to **Fill**.
12. Next, select the arrow, open the **Size Inspector** in the **Utilities** panel, and update the following values:
    - X: 347
    - Y: 127
13. Finally, select the Button, and with the **Size Inspector** still open, update the following values:
    - X: 216
    - Y: 116
    - Width: 141
    - Height: 30
14. Select the **Attributes Inspector** and update the Button **Title** from `Button` to `Read All Reviews`.
15. Update the **Font** to **Semibold** size `14`.
16. Click on **Text Color** and set the **Hex Color #** to `000000` under **RGB** Sliders in the drop-down menu.

17. Your second cell is complete and should look like the following:



Now that the design is finished for these two containers, we need to combine them into one cell. Drag the View from the second cell into the first cell above the **View** container in the first cell, and delete the second cell:



Only the **No Review** cell should remain.

Reservations

Let's get started working on the reservation section of our restaurant details. In this section, we need to add reservation times, seating, and basic restaurant information. Let's get started.

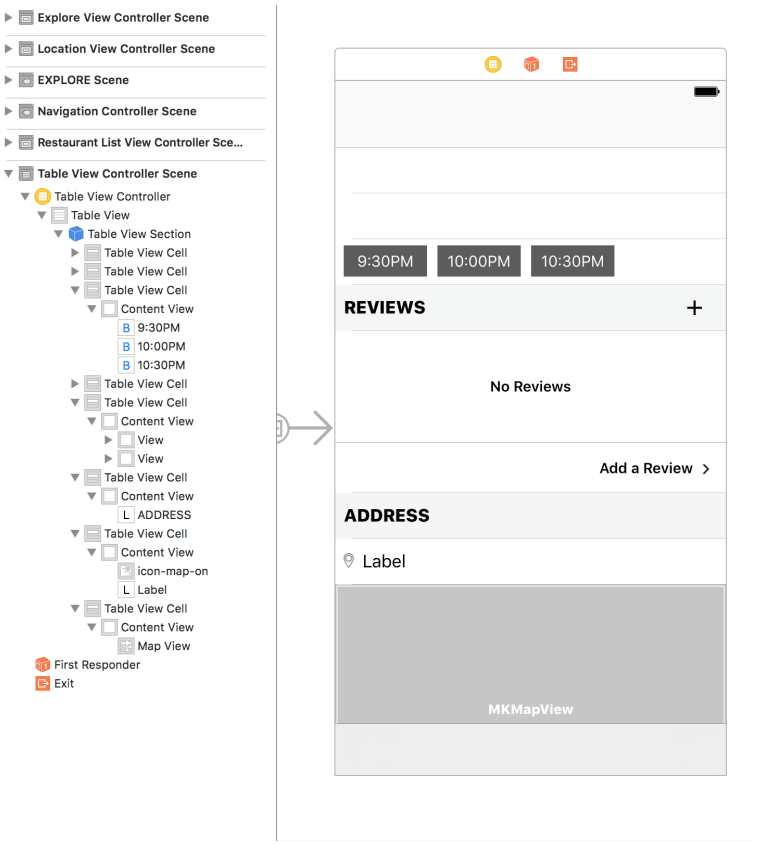
Adding reservation times

Next, we are going to add reservation times:

1. In the Object library filter field in the **Utilities** panel, type `button`.
2. Drag out a Button into the 3rd row.

3. Select the Button, open the **Size Inspector** in the **Utilities** panel, and update the following values:
    - X: 8
    - Y: 6
    - Width: 80
    - Height: 30
4. Select the **Attributes Inspector**, click on the **Background** and set the **Hex Color #** to `4A4A4A` under **RGB** Sliders in the drop-down menu.
5. Then, set the **Text Color** to white using the up and down arrows and ensure that the font size is `15`.
6. Now, select the Button in the Outline view and hit CTRL + *C* to copy. Then, hit CTRL + *V* two times to paste. You should now have three Buttons.
7. Select the first of the Buttons you just pasted, open the **Size Inspector** in the **Utilities** panel, and update the following values:
    - X: 98
    - Y: 6
8. Select the other Button you pasted, and in the **Size Inspector**, update the following values:
    - X: 188
    - Y: 6
9. Update each Button to say **9:30PM**, **10:00PM**, and **10:30PM**, respectively, from left to right, by changing the **Title** of each Button.

10. Your reservation times section should now look as follows:

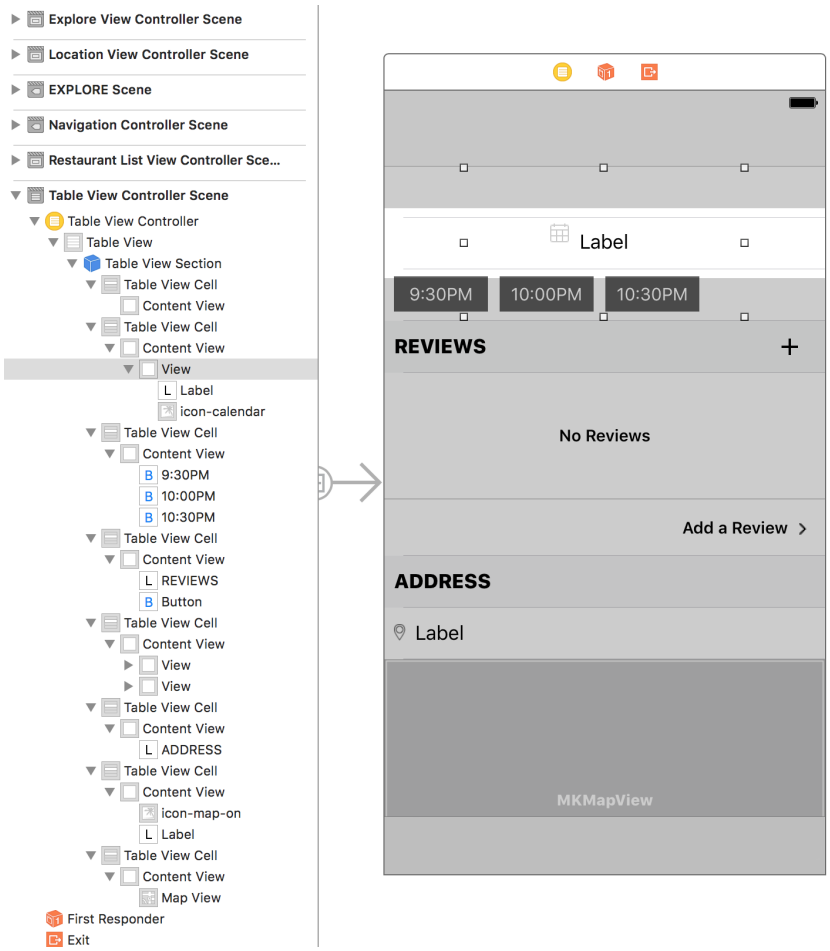


Reservation information

Now, let's move onto the reservation information section:

1. In the filter field of the Object library, type `uiview`.
2. Using the Outline view, drag a UIView into the 2nd row's **Content** View.
3. Next, in the filter field, type `label`.
4. In the Outline view, drag a UILabel into the UIView we just added.
5. Select the Media library icon in the **Utilities** panel, and in the filter area, type `icon-calendar`.
6. Drag and drop the icon-calendar into our **View** container.

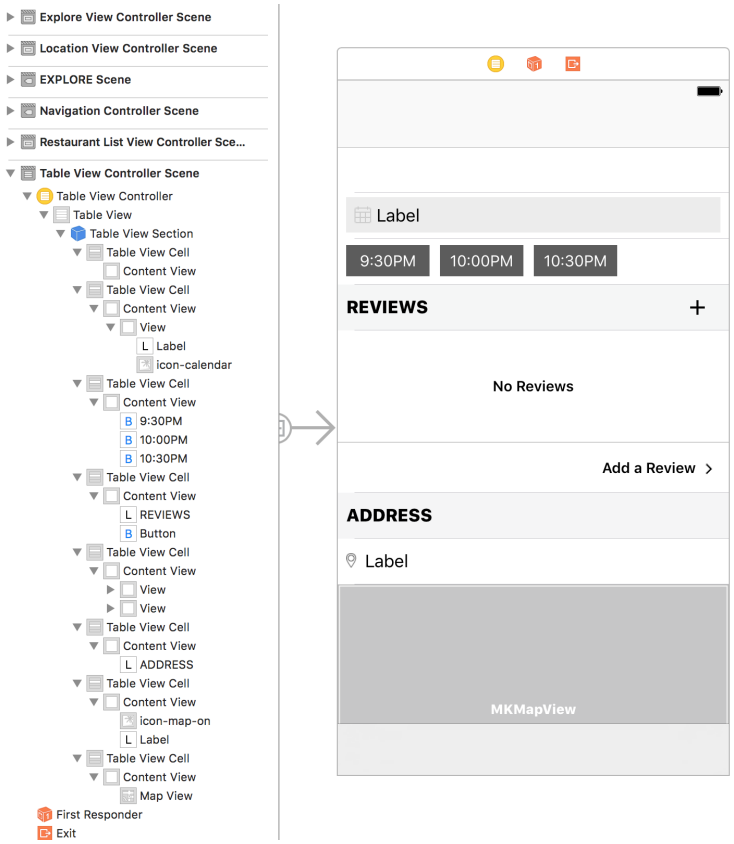
7. You should now see the following structure in your **Outline** view:



Now, that we have all of the UI elements for this row, let's properly size and place these elements:

1. First, select the View in the **Outline** view, open the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: 8
    - **Y**: 4
    - **Width**: 359
    - **Height**: 34

2. Select the **Attributes Inspector**, click on the **Background** and set the **Hex Color #** to `ECECEC` under **RGB** Sliders in the drop-down menu.
3. Now, select the Label in the Outline view, open the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: 29
    - **Y**: 7
    - **Width**: 322
    - **Height**: 21
4. Finally, select the **icon-calendar** in the **Outline** view, and in the **Size Inspector**, update the following values:
    - X: 8
    - Y: 9
5. You should now have the following:



We finished the reservation information cleanup. Lastly, we need to update the reservation header.

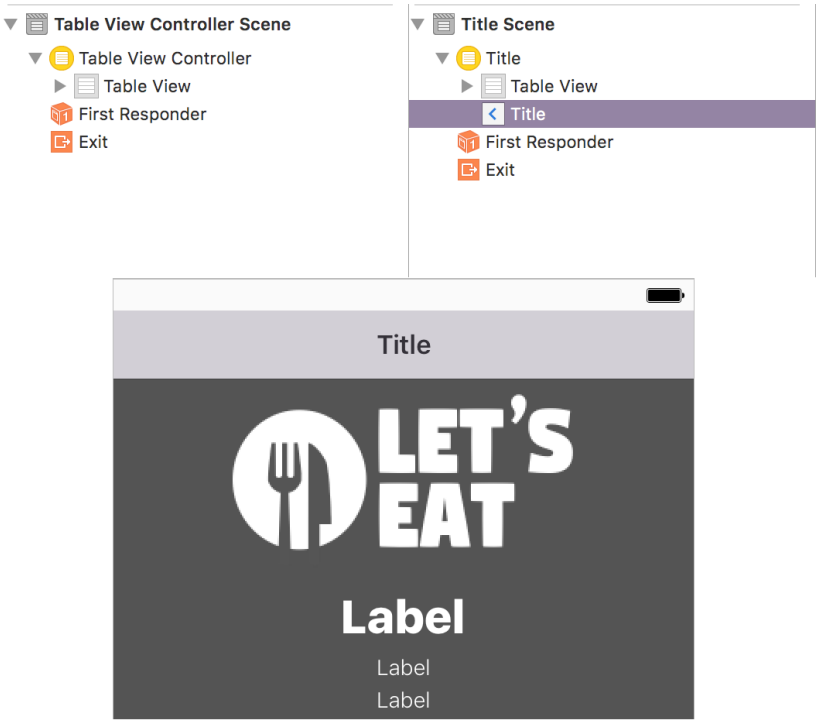
Reservation header

Our first row is our reservation header; it will contain four elements, an image and three labels.

1. Select the 1st row, and, in the **Size Inspector**, update **Row Height** to `240`.
2. In the Object library of the **Utilities** panel, type `uiview` into the filter area.
3. Using the **Outline** view, drag a UIView into the 1st **Table View Cell Content View**.
4. Select the **View**, and in the **Size Inspector**, update the following values:
    - **X**: 0
    - **Y**: 0
    - **Width**: 375
    - **Height**: 240
5. Select the **Attributes Inspector**, click on the **Background**, and set the **Hex Color #** to `393939` under **RGB** Sliders in the drop-down menu.
6. Next, in the **Utilities** panel, select the Object library and type label in the filter field.
7. Drag three Labels into the View we just created.
8. Next, select the Media library icon in the Utilities panel, and in the filter, type detail-logo.
9. Drag a detail-logo into the same View.
10. Select the detail-logo in the Outline view, select the **Size Inspector**, and update the following values:
    - **X**: 77
    - **Y**: 10

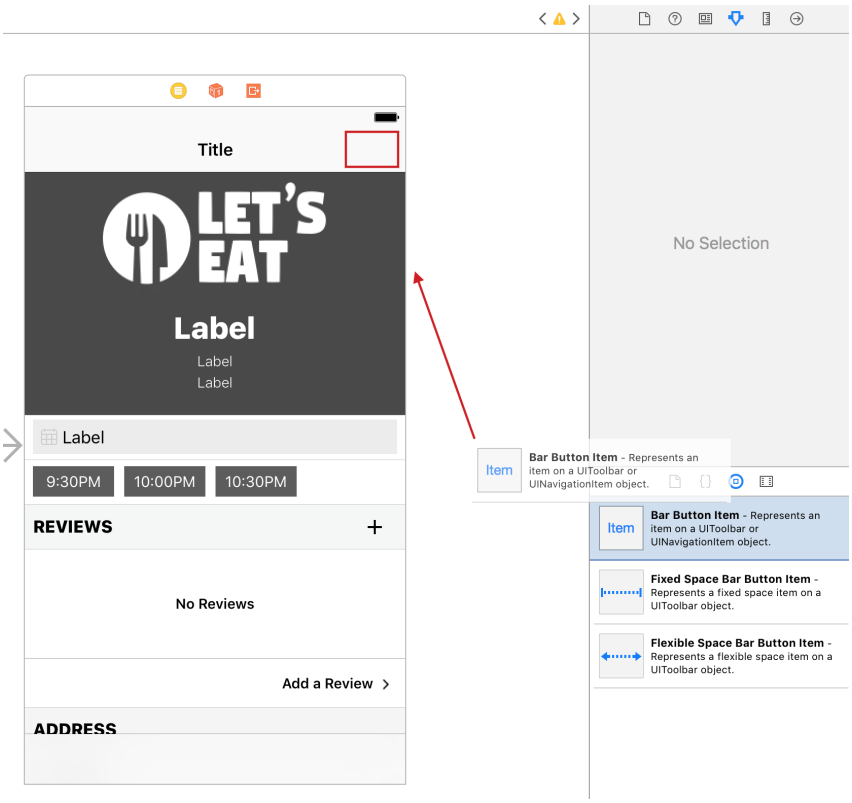
11. Select the first Label in the Outline view, and in the **Size Inspector**, update the following values:
    - **X**: 8
    - **Y**: 132
    - **Width**: 359
    - **Height**: 44
12. Select the **Attributes Inspector** and change the **Alignment** to **Center**, the **Color** to **White Color**, and the **Font** to **Heavy** size `30`.
13. Next, select the second Label in the Outline view and update the following values in the **Size Inspector**:
    - **X**: 8
    - **Y**: 176
    - **Width**: 359
    - **Height**: 21
14. Open the **Attributes Inspector** and change the **Alignment** to **Center**, the **Color** to **White Color**, and the **Font** to **Thin** size `14`.
15. Now, select the third Label in the Outline view, and in the **Size Inspector**, update the following values:
    - **X**: 8
    - **Y**: 197
    - **Width**: 359
    - **Height**: 21
16. In the **Attributes Inspector**, change the **Alignment** to **Center**, the **Color** to **White Color**, and the **Font** to **Thin** size `14`.
17. Finally, we are going to add a heart in the **Navigation** Bar. This will allow users to add restaurants as favorites:
18. In the Object library of the **Utilities** panel, type `navigationitem` in the filter.
19. Make sure to close the disclosure arrow for the Table View, and drag a **Navigation** Item into the Outline view under **Table View**.

20. When you are done, your Outline view will be updated from looking like the left side of the following screenshot to looking like the right side; the scene will look like the image in the screenshot:



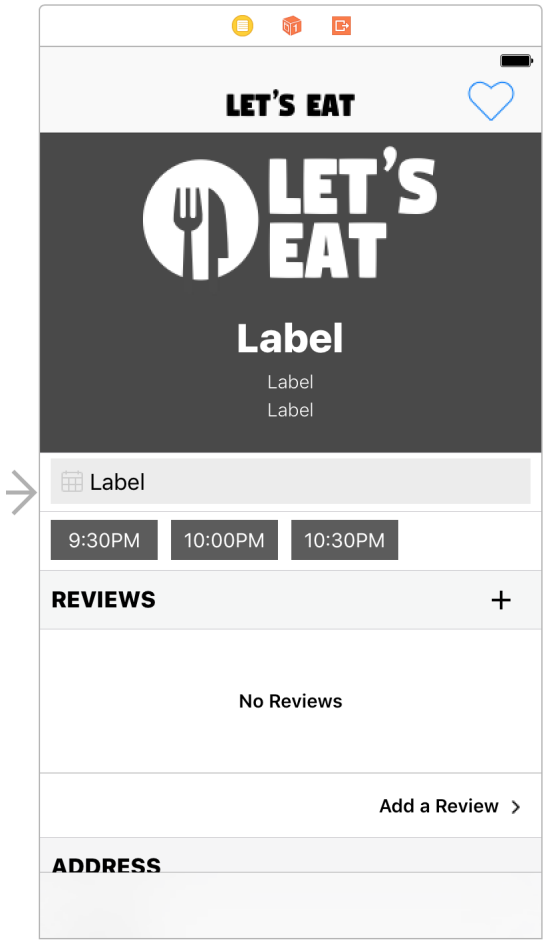
21. Next, in the **Object** library, type `barbuttonitem` in the filter.

22. Drag a **Bar Button Item** into the **Navigation** Bar:



23. Select the Bar Button Item, and the **Attributes Inspector** in the **Utilities** panel and update the following values:
    - **Title**: Delete "Item" text
    - **Image**: heart-unselected
24. In the **Utilities** panel, select the Object library and type `uiview` in the filter.
25. Drag a View into the title bar where it currently says `Title`.
26. With the **View** selected, in the **Attributes Inspector** of the **Utilities** panel, update the **Background** to **Clear Color**.
27. Next, in the **Utilities** panel, select the **Media** library. In the filter, type `logo-nav`.
28. Drag **logo-nav** into the **View** we just added in the title bar.

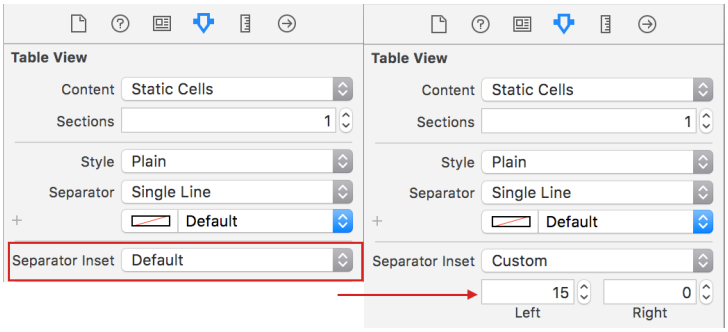
29. In the **Outline** view, select the **logo-nav**, and in the **Size Inspector**, update the following values:
    - **X**: 27
    - **Y**: 6
30. By adding these updates, you should now have the following:



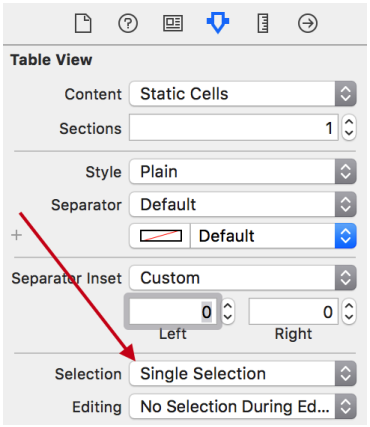
Let's build and run the project by hitting the play button (or use CTRL + *R*) and see what our restaurant detail looks like.

Our restaurant detail is very close to what we want, but there are a couple of remaining issues. Currently, our **Table View** separators have an inset, which we do not want. In addition, if you select a cell in the restaurant detail view, the cell becomes highlighted, which we also do not want. To correct these items, follow the below steps:

1. In `Explore.storyboard`, select the **Table View** in the Outline view of the restaurant detail.
2. In the **Attributes Inspector** of the **Utilities** panel, change **Separator Inset** from **Default** to **Custom**. Then, change the **Left** inset to `0`:



3. Then, change **Selection** from **Single Selection** to **No Selection**:



Let's build and run the project again by hitting the play button (or use CTRL + *R*), and you will see those items corrected.

We completed our layout for our restaurant detail; later in the book, we load data into this View.

Summary

In this chapter, we did a lot of Storyboard setup. We took a complex layout and simplified it by making it all in Storyboard. You should now be getting more and more comfortable with being inside Xcode, especially Storyboard.

In the next chapter, we will focus our attention on the Map tab. We will cover annotations (pins) and what they are. Then, we will create and add our own custom annotations to our Map. When we are done with annotations, we will add callouts to our annotations (bubbles that display the restaurant name and cuisine if you tap the annotation) so that we can click on the callout to go to our restaurant detail.

10
Where Are We?

We have all used a map at some point in our lives, either an actual map or a map on our phone or another device. Apple Maps has come a long way from when it was first announced in 2012. Apple has made steady improvements to Apple Maps every year.

During this chapter, we will display our restaurant list using a map and custom pins. When users tap a pin on the map, they will be taken directly to the restaurant detail page that we created in the last chapter.

In this chapter, we will cover:

- What are annotations and how to add them to a map?
- How to create custom annotations?
- Learn how to create a Storyboard reference

Before we start working with our Map, we need to create two folders inside of our `Map` folder. Right-click on the `Map` folder in the **Navigator** panel and create two new groups: **Controller** and **Model**. Then, add the assets for this chapter, which can be found in the project folder for this chapter, by dragging and dropping them into this new `Model` folder.

Setting up map annotations

In our Map, we are going to drop pins down for each restaurant location. These pins are called annotations, more specifically, **MKAnnotations**. Since we are going to create multiple MKAnnotations, we are going to create a class that subclasses MKAnnotations.

What is an MKAnnotation?

MKAnnotation is a protocol that provides us with information related to a map view. Protocols provide a blueprint for methods, properties, and other required functionality. The MKAnnotation will provide information, such as the coordinates (latitude and longitude), title, and subtitle of the annotation. In order to drop a pin onto a map, we must subclass the MKAnnotation. When we first looked at classes versus structs, we discussed that classes can subclass or inherit from other classes, which means that we can get properties, methods, and other requirements from the class that we are subclassing. Let's create an annotation that subclasses MKAnnotation and see how this works.

Creating a restaurant annotation

Before we jump into creating our file, we should first look at the data that we will be using. The data for the Map view will actually be the same data that we use for our restaurant listing page. Let's take a look at what the restaurant data will look like:

Key	Type	Value
▼ Root	Array	(5 items)
▼ Item 0	Dictionary	(16 items)
address	String	108 West 2nd Street #104
area	String	Los Angeles / Orange County
city	String	Los Angeles
cuisine	String	American
country	String	US
id	Number	104,173
image_url	String	https://www.opentable.com/img/restimages/104173.jpg
lat	Number	34.051061
lng	Number	-118.244705
mobile_reserve_url	String	http://mobile.opentable.com/opentable/?restId=104173
name	String	Badmaash
phone	String	2132217466x
postal_code	String	90012
price	Number	2
reserve_url	String	http://www.opentable.com/single.aspx?rid=104173
state	String	CA

We need to create a file to represent this data for the Map View, which will differ from the restaurant listing page, because we need to subclass MKAnnotation. Let's get started by creating this file now:

1. Right-click on the `Model` folder and select **New File**.
2. Inside of the **Choose a template for your new file** screen, select iOS at the top and then **Cocoa Touch Class**. Then, hit **Next**.

3. In the options screen that appears, add the following:
4. **New File**:
    - **Class**: `RestaurantAnnotation`
    - **Subclass**…: `NSObject`
    - **Also create XIB**: Unchecked
    - **Language**: `Swift`
5. Click **Next** and then **Create**.

In this new `RestaurantAnnotation.swift` file, under `import UIKit`, add `import MapKit`. We need this import statement so that Xcode knows where the files are that we are going to use.

Next, we need to update our class declaration in order to make our annotation. Since this is subclassing MKAnnotation, we need to change what we currently have (`class RestaurantAnnotation: NSObject`) to the following:

```
class RestaurantAnnotation: NSObject, MKAnnotation
```

You will see an error when you add the MKAnnotation. Just ignore it for now, as we will fix this error shortly.

Inside of the class declaration, add the following:

```
var name: String?
var cuisines:[String] = []
var latitude: Double?
var longitude:Double?
var address:String?
var postalCode:String?
var state:String?
var imageURL:String?
```

When the user taps on the annotation, the name of the restaurant and types of cuisine will appear along with a detail icon. This detail icon will take the user to the restaurant detail page. Then, we will pass along all of this data and will use it to populate the restaurant detail page we just created in the last chapter.

We need to initialize all of the data being passed into the object. Therefore, let's create an `init()` method to which we can pass a dictionary object through its parameters.

```
import UIKit
import MapKit
                                    Ignore Error
class RestaurantAnnotation: NSObject, MKAnnotation {

    var name: String?
    var cuisines:[String] = []
    var latitude: Double?
    var longitude:Double?
    var address:String?
    var postalCode:String?
    var state:String?
    var imageURL:String?

    init(dict:[String:AnyObject]) {
        if let lat = dict["lat"] as? Double { self.latitude = lat }
        if let long = dict["lng"] as? Double { self.longitude = long }
        if let name = dict["name"] as? String { self.name = name }
        if let cuisines = dict["cuisines"] as? [String] { self.cuisines = cuisines }
        if let address = dict["address"] as? String { self.address = address }
        if let postalCode = dict["postal_code"] as? String { self.postalCode = postalCode }
        if let state = dict["state"] as? String { self.state = state }
        if let image = dict["image_url"] as? String { self.imageURL = image }
    }


}
```

This method is large, but it is nothing you have not seen before. We are using the `if-let` statement to check for data in each element. If something is missing, it will not be sent.

Let's address the error now. The reason we are getting an error is because we are subclassing MKAnnotation and have not yet declared `coordinate`, which is a required property. We also have two other optional properties, `title` and `subtitle`, that we are using for our Map and will also need to declare. What we want to be able to do is pass the data that we have over to these three properties so that we can use them in our Map.

In order to get rid of the error, we need to add the coordinate first. We need to set up the latitude and longitude, so add the following after the `init()` method:

```
var coordinate: CLLocationCoordinate2D {
    guard let lat = latitude, let long = longitude else { return
CLLocationCoordinate2D() }
    return CLLocationCoordinate2D(latitude: lat, longitude: long )
}
```

`CLLocationCoordinate2D` is a class that is used by `MapKit` to set the exact location of a pin.

Note that we are using curly braces for this property. It is defined in MKAnnotation, and we are using the computed property to set the value. For the `coordinate` property, we will pass a latitude and longitude to it, using a `CLLocationCoordinate2D`. In our `init()` method, we created the data that sets the latitude and longitude, and now, we are passing those coordinates over to the `coordinate` property.

Let's do the same with `subtitle` by adding the following above the variable coordinate:

```
var subtitle: String? {
    if cuisines.isEmpty { return "" }
    else if cuisines.count == 1 { return cuisines.first }
    else { return cuisines.joined(separator: ", ") }
}
```

This is also a computed property, but this time we are using an `else-if` statement. We first check to see if the array is empty; if so, nothing will be displayed. If we only have one item in the array, we just return that item. Finally, if we have multiple items in our array, we take each item and put them in one String, separating each item with a comma. For example, if your array had the following items: *["American", "Bistro", "Burgers"]*, then we would create a string that looks like: *American, Bistro, Burgers.*

Finally, we need to add the `title`. Therefore, enter the following above the `subtitle` variable:

```
var title: String? {
    return name
}
```

Your file should no longer have an error and should now look as follows:

```
import UIKit
import MapKit

class RestaurantAnnotation: NSObject, MKAnnotation {

    var name: String?
    var cuisines:[String] = []
    var latitude: Double?
    var longitude:Double?
    var address:String?
    var postalCode:String?
    var state:String?
    var imageURL:String?

    init(dict:[String:AnyObject]) {
        if let lat = dict["lat"] as? Double { self.latitude = lat }
        if let long = dict["lng"] as? Double { self.longitude = long }
        if let name = dict["name"] as? String { self.name = name }
        if let cuisines = dict["cuisines"] as? [String] { self.cuisines = cuisines }
        if let address = dict["address"] as? String { self.address = address }
        if let postalCode = dict["postal_code"] as? String { self.postalCode = postalCode
        if let state = dict["state"] as? String { self.state = state }
        if let image = dict["image_url"] as? String { self.imageURL = image }
    }

    var title: String? {
        return name
    }

    var subtitle: String? {
        if cuisines.isEmpty { return "" }
        else if cuisines.count == 1 { return cuisines.first }
        else { return cuisines.joined(separator: ", ") }
    }

    var coordinate: CLLocationCoordinate2D {
        guard let lat = latitude, let long = longitude else { return CLLocationCoordinate2D() }
        return CLLocationCoordinate2D(latitude: lat, longitude: long )
    }
}
```

Next, we want to create a manager that will take our data and create annotations for our Map.

Creating our Map Data Manager

In the next chapter, we will deal with data, but for now, we can mock up some data in order to set up our structure. We will use a `plist` to load our data, just like we did in the last chapter.

Let's create the `MapDataManager` file now:

1. Right-click on the `Model` folder in the `Location` folder and select **New File**.
2. Inside of the **Choose a template for your new file** screen, select **iOS** at the top and then **Swift File**. Then, hit **Next**.

3. Name this file, `MapDataManager` and then hit **Create**.

4. Next, we need to define our class definition, so add the following under the import statement:

```
class MapDataManager {}
```

5. Inside of the class declaration, add the following variables:

```
fileprivate var items:[RestaurantAnnotation] = []

  var annotations:[RestaurantAnnotation] {
  return items
}
```

6. Note that we are keeping our array private, since there is no reason to have to access this outside of the class.

7. Now, let's add the following methods inside of our class declaration, after our variables:

```
func fetch(completion:(_ annotations:[RestaurantAnnotation]) ->
()) {
 if items.count > 0 { items.removeAll() }
    for data in loadData() {
        items.append(RestaurantAnnotation(dict: data))
    }

    completion(items)
}

fileprivate func loadData() -> [[String:AnyObject]] {
     guard let path = Bundle.main.path(forResource:
"MapLocations", ofType: "plist"),
       let items = NSArray(contentsOfFile: path) else { return
[[:]] }

    return items as! [[String : AnyObject]]
}
```

8. Your file should now look as follows:

```
import Foundation

class MapDataManager:DataManager {

    fileprivate var items:[RestaurantAnnotation] = []

    var annotations:[RestaurantAnnotation] {
        return items
    }

    func fetch(completion:(_ annotations:[RestaurantAnnotation]) -> ()) {
        if items.count > 0 { items.removeAll() }
        for data in loadData() {
            items.append(RestaurantAnnotation(dict: data))
        }

        completion(items)
    }

    fileprivate func loadData() -> [[String:AnyObject]] {
        guard let path = Bundle.main.path(forResource: "MapLocations", ofType: "plist"),
            let items = NSArray(contentsOfFile: path) else { return [[:]] }

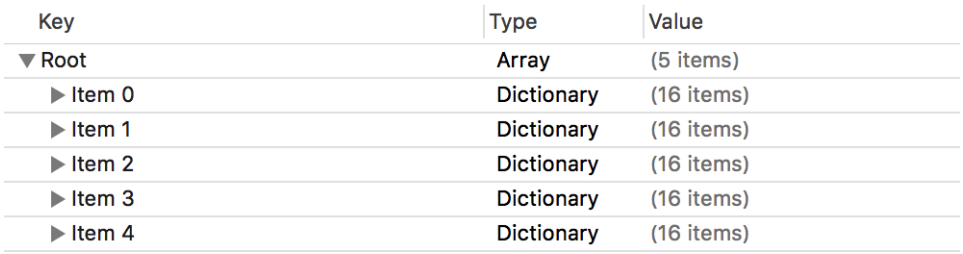
        return items as! [[String : AnyObject]]
    }
}
```

9. The `fetch()` and `loadData()` methods are basically the same as those which we had in the `ExploreDataManager` file. However, the `fetch()` method here has something new inside of the parameters, specifically:

   ```
   completion:(_ annotations:[RestaurantAnnotation]) -> ())
   ```

10. This is called a closure block, which allows us to signify when we completed the method, and it then dictates an action to occur (here, returning an array of annotations). We will use these annotations to load pins on our Map. We are looping through the `for-in` loop; when we are done, we call `completion()`. When we get to our `MapViewController`, you will see how we write this.

Now, let's take a look at our `MapLocations.plist` file:



Key	Type	Value
▼ Root	Array	(5 items)
▶ Item 0	Dictionary	(16 items)
▶ Item 1	Dictionary	(16 items)
▶ Item 2	Dictionary	(16 items)
▶ Item 3	Dictionary	(16 items)
▶ Item 4	Dictionary	(16 items)

This file is the same structure as our `ExploreData.plist` file. Our **Root** is an array, and each item inside of our **Root** is a dictionary item. There is an acronym that many programmers call **DRY** (**don't repeat yourself**). Since both `plist` files have an array of dictionary objects, we can update our code so that we can use the same method in multiple places.

Creating a base class

In order to keep from repeating ourselves, we are going to create a base class. This base class will have a new method called `load(file name:)`, but we will add a parameter in order to pass the file name. Let's create a `DataManager` file now under our `Common` folder:

1. Right-click on the `Common` folder in the **Navigator** panel and create a new group called, **Misc**. Then, right-click on this folder and select **New File**.
2. Inside of the **Choose a template for your new file** screen, select **iOS** at the top and then **Swift File**. Then, hit **Next**:
3. Name this file `DataManager`, and then hit **Create**.
4. In this new file, we need to define our class definition; therefore, add the following under the `import` statement:

```
class DataManager {}
```

5. Inside of the class declaration, add the following method:

```
func load(file name:String) -> [[String:AnyObject]] {
    guard let path = Bundle.main.path(forResource: name, ofType:
"plist"),  let items = NSArray(contentsOfFile: path) else { return
[[:]] }

   return items as! [[String : AnyObject]]
}
```

6. Other than changing the function name to include parameters, we created the same function as we had in our `Explore` and `Map Data Manager` files. However, this function here is no longer a private method, because we want it to be accessible to any class that wants to use it.
7. Your file should now look like the following:

```
import Foundation

class DataManager {
    func load(file name:String) -> [[String:AnyObject]] {
        guard let path = Bundle.main.path(forResource: name, ofType: "plist"),
            let items = NSArray(contentsOfFile: path) else { return [[:]] }

        return items as! [[String : AnyObject]]
    }
}
```

8. This is all we need to do in this file, so let's return to the `MapDataManager`:
9. Delete the file private function `loadData()`, because we will not need it anymore.

You will see an error after you delete the `loadData()` method. This error is happening because we need to give the `fetch()` method a filename to load whenever we call the `loadData()` method. We will fix this shortly.

1. Next, we need to update our class declaration to say:

   ```
   class MapDataManager: DataManager
   ```

2. We now have our `MapDataManager` class subclassing our `DataManager`, which means that we will use the `loadData()` method from our `DataManager` inside of our `MapDataManager`.
3. Now, let's fix the error by updating our `fetch()` method from `for data in loadData()` to the following:

   ```
   for data in load(file: "MapLocations")
   ```

Your updated file should now look like the following:

```
import Foundation

class ExploreDataManager:DataManager {

    fileprivate var items:[ExploreItem] = []

    func fetch() {
        for data in load(file:"ExploreData") {
            items.append(ExploreItem(dict: data))
        }
    }

    func numberOfItems() -> Int {
        return items.count
    }

    func explore(at index:IndexPath) -> ExploreItem {
        return items[index.item]
    }
}
```

We now removed the error in our `MapDataManager`, but we need to do some refactoring of our `ExploreDataManager` file to do the same.

Refactoring ExploreDataManager

Because our `loadData()` was written exactly the same in both the `ExploreDataManager` and `MapDataManager` files, we need to update our `ExploreDataManager` in the same way we just did for the `MapDataManager`. Open `ExploreDataManager` and do as follows:

1. Delete the private `loadData()` function, because we will not need it anymore.

   Again, ignore the error as we are going to fix this shortly.

2. Next, update our class declaration to now say:

   ```
   class ExploreDataManager: DataManager
   ```

3. Now, let's fix the error by updating our `fetch()` method from `for data in loadData()` to the following:

   ```
   for data in load(file: "ExploreData")
   ```

4. Your updated function should now look like the following:

   ```
   func fetch(completion: (_ annotations:[RestaurantAnnotation]) ->
   ()) {
       if items.count > 0 { items.removeAll() }
       for data in load(file: "MapLocations") {
           items.append(RestaurantAnnotation(dict: data))
       }

       completion(items)
   }
   ```

We completed refactoring our files, and we can now use the same method anytime we need to load a `plist` that has an array of dictionary items.



Refactoring is something with which you will become more comfortable the more you write code. Understanding when to refactor is a bit harder when you first start out, because you are still learning. The biggest indicator that you need to refactor is when you have written something more than once. However, refactoring does not always work for everything; at times, writing the same code more than once can be unavoidable. Just being aware of when refactoring may be useful is a good sign and half the battle to a greater understanding of this method. I have been coding for years; there will be times when I copy and paste something I wrote to see if it works and then never refactor. Then, months later, I will wonder why I did not write a method to handle it in both places.

Creating and adding annotations

Now, we need to get our Map hooked up and start getting the annotations displaying on the Map. Then, we will customize our annotations to look like those in our design.

Creating our Map View Controller

We need to create our Map View Controller file and then connect it with our UIViewController and Map View in Storyboard. First, let's create this file:

1. In the **Navigator** panel, right-click on the `Controller` folder in the `Map` folder and select **New File**.
2. Inside of the **Choose a template for your new file** screen, select **iOS** at the top and then **Cocoa Touch Class**. Then, hit **Next**.
3. Add the following into the options screen that appears:
4. **New File**:
    - **Class**: `MapViewController`
    - **Subclass**…: `UIViewController`
    - **Also create XIB**: Unchecked
    - **Language**: `Swift`
5. Click on **Next** and then **Create**.
6. Under the import `UIKit` statement, add `import MapKit`.
7. Update your class declaration to include the following subclass:

```
class MapViewController: UIViewController, MKMapViewDelegate
```

Let's now connect this file with our `UIViewController` and our **Map View** in Storyboard:

1. Add the following after the class declaration:

   ```
   @IBOutlet var mapView: MKMapView!
   ```

2. Open your `Map.storyboard` file.
3. In the **Outline** view, select the **View Controller** that contains the **Map View**.
4. Now, in the **Utilities** panel, select the **Identity Inspector**.
5. Under **Custom Class**, in the **Class** drop-down menu, select **MapViewController** and hit *Enter* in order connect the **View Controller** to the class.
6. Now, select the **Connections Inspector**.
7. Under the **Outlets** section, you will see an empty circle next to **mapView**. Click on and drag the outlet to the Map View in the View Controller in the **Outline** view.

We are going to start working with our Map, but first we need to add some things to our MapDataManager:

1. Open the `MapDataManager.swift` file in the **Navigator Panel**; underneath the `import Foundation` statement, add `import MapKit`.
2. Next, add the following method to our `MapDataManager`:

```
func currentRegion(latDelta:CLLocationDegrees, longDelta:CLLocationDegrees) -> MKCoordinateRegion {
    guard let item = items.first else { return MKCoordinateRegion() }
    let span = MKCoordinateSpanMake(latDelta, longDelta)

    return MKCoordinateRegion(center: item.coordinate, span: span)
}
```

A B C D

Before we delve into the particular sections of this function, we need to understand what this function does. When you use a map and drop pins down onto it, you want the map to zoom into a certain area. In order to zoom in on a map, you need a latitude and longitude. What this method is doing is grabbing the first pin (or annotation) in the array and zooming in on the area. Let's break down the code to better understand each part:

- **Part A:**

  ```
  func currentRegion(latDelta:CLLocationDegrees,
  longDelta:CLLocationDegrees) -> MKCoordinateRegion {
  ```

Our method has two parameters, both of which are `CLLocationDegrees`. It is just a class that represents a latitude or longitude coordinate in degrees.

- **Part B:**

  ```
  guard let item = items.first else { return MKCoordinateRegion() }
  ```

  This guard statement is obtaining the first item in the array. If there are no items in the array, it will just return an empty coordinate region. If there are items in the array, it will return the coordinate region.

- **Part C:**

  ```
  let span = MKCoordinateSpanMake(latDelta, longDelta)
  ```

  Here, we are creating an `MKCoordinate` with the latitude and longitude that we passed into the function. `MKCoordinateSpan` defines a span, in the latitude and longitude directions, to show on the Map.

- **Part D:**

  ```
  return MKCoordinateRegion(center: item.coordinate, span: span)
  ```

  Lastly, we are setting the center and the span of our region, and returning them back so that when the pins are dropped, the Map can zoom in on the area.

Now, let's set up our `MapViewController` to display annotations:

1. Open the `MapViewController.swift` file in the **Navigator** panel and delete both `didReceiveMemoryWarning()` and `prepare()` (which has been commented out), as we do not need them for our purposes.
2. Directly under our `IBOutlet` statement, add the following:

   ```
   let manager = MapDataManager()
   ```

3. Then, inside of the class definition, add the following method after `viewDidLoad()`:

   ```
   func addMap(_ annotations:[RestaurantAnnotation]) {
       mapView.setRegion(manager.currentRegion(latDelta: 0.5,
   longDelta: 0.5), animated: true)
       mapView.addAnnotations(manager.annotations)
   }
   ```

In this method we are doing a couple things. We first pass annotations through the parameter. When we call `fetch()` and it is completed, it will return the array of annotations. We will pass that array over to our `addMap(_ annotations:)` to use. Next, we set the region by obtaining it from our `MapDataManager`, thus setting the latitude and longitude delta. This will set our zoom and region for our Map. Once we have that, we then pass all of our annotations for the Map to display.

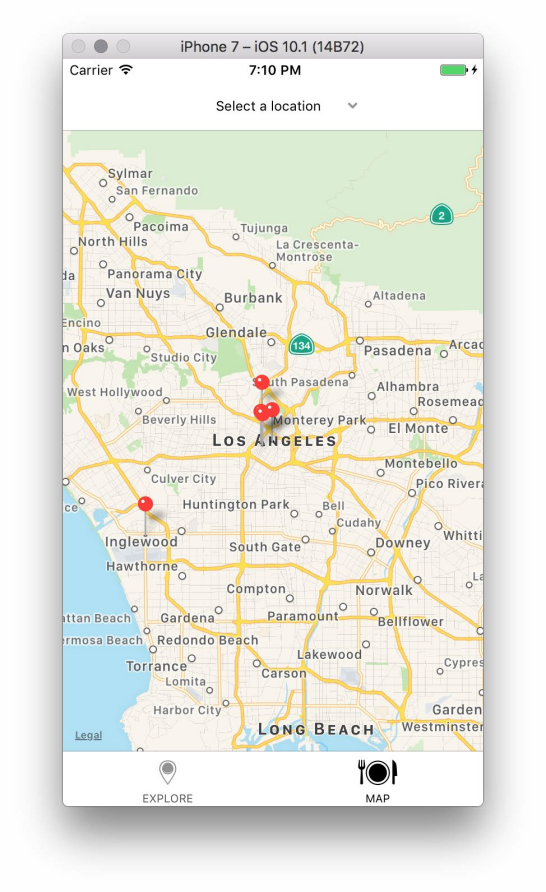
Therefore, we need to have our manager fetch the annotations:

1. Add the following method above `addMap(_ annotations:)`:

```
func initialize() {
    mapView.delegate = self
    manager.fetch { (annotations) in
        addMap(annotations)
    }
}
```

2. Inside of the `initialize()` method, we are setting the Map delegate to the class. In previous chapters, we did this using **Storyboard**; however, you can also do this with code. This line allows us to be notified when the user taps on an annotation or taps the disclosure indicator in the annotation.
3. Earlier in this chapter, we created a `fetch()` method in the `MapDataManager`, wherein we used a closure block. This closure block requires that we wrap it in curly braces. Once, the `completion()` block is called in the manager, everything inside of the curly braces will run. For our purposes in building this app, we are going to have a small number of pins or annotation; therefore, we do not need a completion block. However, if you have 100 or 500 annotations, for instance, a closure block would be more efficient. We will do more with this later so that you can get more practice with closure blocks.
4. Add `initialize()` inside of `viewDidLoad()` so that everything will run when the view loads.

5. Let's build and run the project by hitting the play button (or use CMD + *R*):



We now have pins on our Map, but we need to update them so that they look more like the ones in our design. Let's learn how to customize the annotations in our Map.

Creating custom annotations

If you ever owned an iPhone and used Apple Maps, you are familiar with the pins. When you have a map inside of your own app, having custom pins (annotations) gives your app a bit more polish. Let's create our own custom annotations:

Open up `MapViewController` in the **Navigator** panel, and add the following directly under the `addMap(_ annotations:)` method:

```
func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {
    let identifier = "custompin"

    guard !annotation.isKind(of: MKUserLocation.self) else {
        return nil
    }

    var annotationView:MKAnnotationView?

    if let customAnnotationView = mapView.dequeueReusableAnnotationView(withIdentifier:
        identifier) {
        annotationView = customAnnotationView
        annotationView?.annotation = annotation
    }
    else {
        let av = MKAnnotationView(annotation: annotation, reuseIdentifier: identifier)
        av.rightCalloutAccessoryView = UIButton(type: .detailDisclosure)
        annotationView = av
    }

    if let annotationView = annotationView {
        annotationView.canShowCallout = true
        annotationView.image = UIImage(named: "custom-annotation")
    }

    return annotationView
}
```

Let's discuss what we just added:

- **Part A**

  `func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView?`

  This method will call the `mapView.delegate` we set up earlier when annotations need to be placed. We will use this method to grab the annotations before they are placed and replace the default pins with custom pins.

- **Part B**

  `let identifier = "custompin"`

  Here, we set an identifier, similar to those which we set when using Collection Views and Table Views.

- **Part C**

```
guard !annotation.isKind(of: MKUserLocation.self) else {
   return nil
}
```

This guard will ensure that our annotation is not the user location. If the annotation is the user location, the guard will return nil, otherwise, it will move on through the method.

- **Part D**

```
var annotationView:MKAnnotationView?
```

`MKAnnotationView` is the class name for the pin; here, we create a variable that we can use to set our custom image.

- **Part E**

```
if let customAnnotationView = mapView.dequeueReusableAnnotationVie
w(withIdentifier: identifier) {
   annotationView = customAnnotationView
   annotationView?.annotation = annotation
}
```

In this statement, we are checking to see if there are any annotations already created that we can reuse. If so, we point it to the variable we just added above. Otherwise, we create the annotation in the next `else` statement.

- **Part F**

```
else {
    let av = MKAnnotationView(annotation: annotation,
reuseIdentifier: identifier)
    av.rightCalloutAccessoryView = UIButton(type:
.detailDisclosure)
            annotationView = av
}
```

If there are no annotations to reuse; we create a new `MKAnnotationView` and give it a callout with a button. A callout is a bubble that appears above the annotation when you tap it in order to display the title (restaurant name) and subtitle (cuisines) associated with that annotation. If the user selects this callout button, he or she will be taken to the restaurant detail view.

- **Part G**

```
if let annotationView = annotationView {
    annotationView.canShowCallout = true
    annotationView.image = UIImage(named: "custom-annotation")
}
```

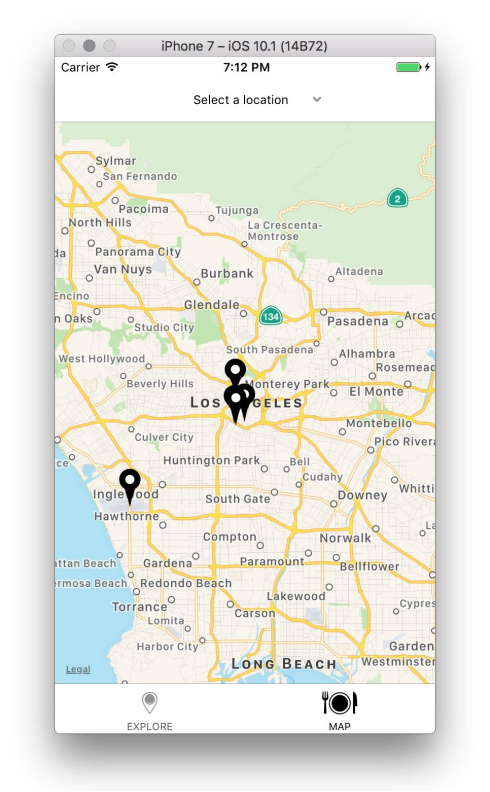
  Here is where we make sure that our custom annotation will show a callout. We also set our custom image for our annotation.

- **Part G**

```
return annotationView
```

  Once we are finished going through the method, we return our custom annotation to the Map. This method will be called for every annotation that appears on the Map.

Let's build and run the project by hitting the play button (or use *CMD* + *R*):



We now have custom annotations displaying on our Map. Each pin's callout shows the restaurant name as well as the cuisines for the restaurant associated with that particular pin. If you tap on the callout, the restaurant detail disclosure does not yet work. Let's now set that up.

Refactoring restaurant detail

In order for us to go to the restaurant detail from the callout, we need to update our app so that we can share the restaurant detail with the Map.

We currently have our restaurant detail inside of `Explore.storyboard`. To better organize our Storyboards, we need to create a Storyboard reference and put our restaurant detail into its own Storyboard:

1. In `Explore.storyboard`, click on the **Table View Controller Scene** that contains the restaurant detail.
2. With the scene selected, navigate to **Editor | Refactor to Storyboard**.
3. In the screen that appears, name this `RestaurantDetail.storyboard`, and then hit **Save**.

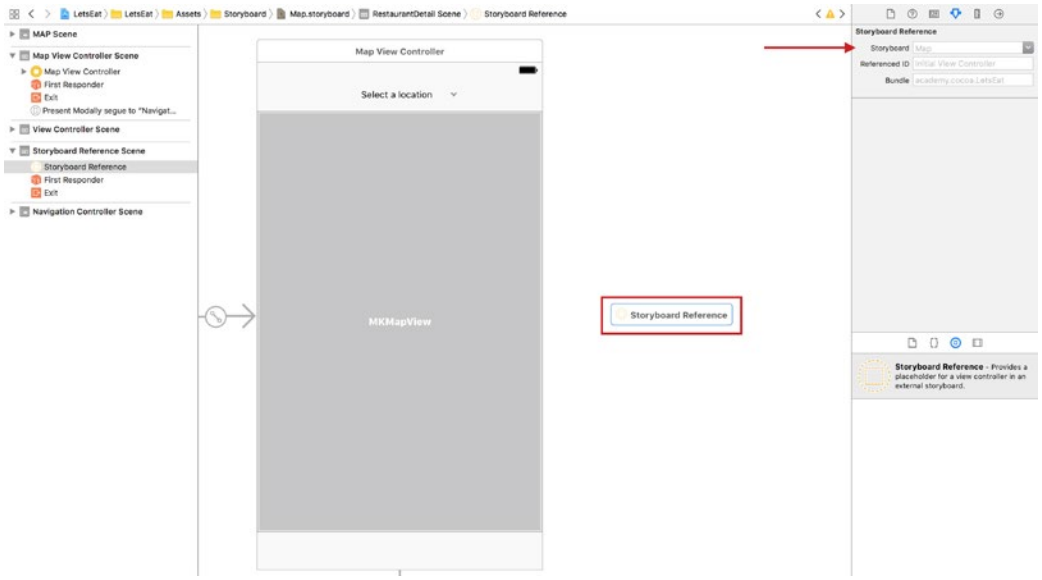
You now have a new `RestaurantDetail.storyboard` file in your `Storyboard` folder in the **Navigator** panel.

Let's build and run the project again by hitting the play button (or use CMD + *R*). You should still be able to go to restaurant details from the restaurant listings. Now, we need to link to the restaurant detail from the Map.

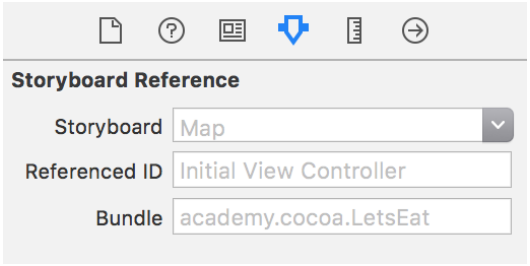
Creating a Storyboard reference

In order to link to restaurant detail from the Map, we need to create a Storyboard reference:

1. Open the `Map.storyboard`, and in the **Object** library of the **Utilities** panel, drag a **Storyboard reference** into the `Map.storyboard` scene:



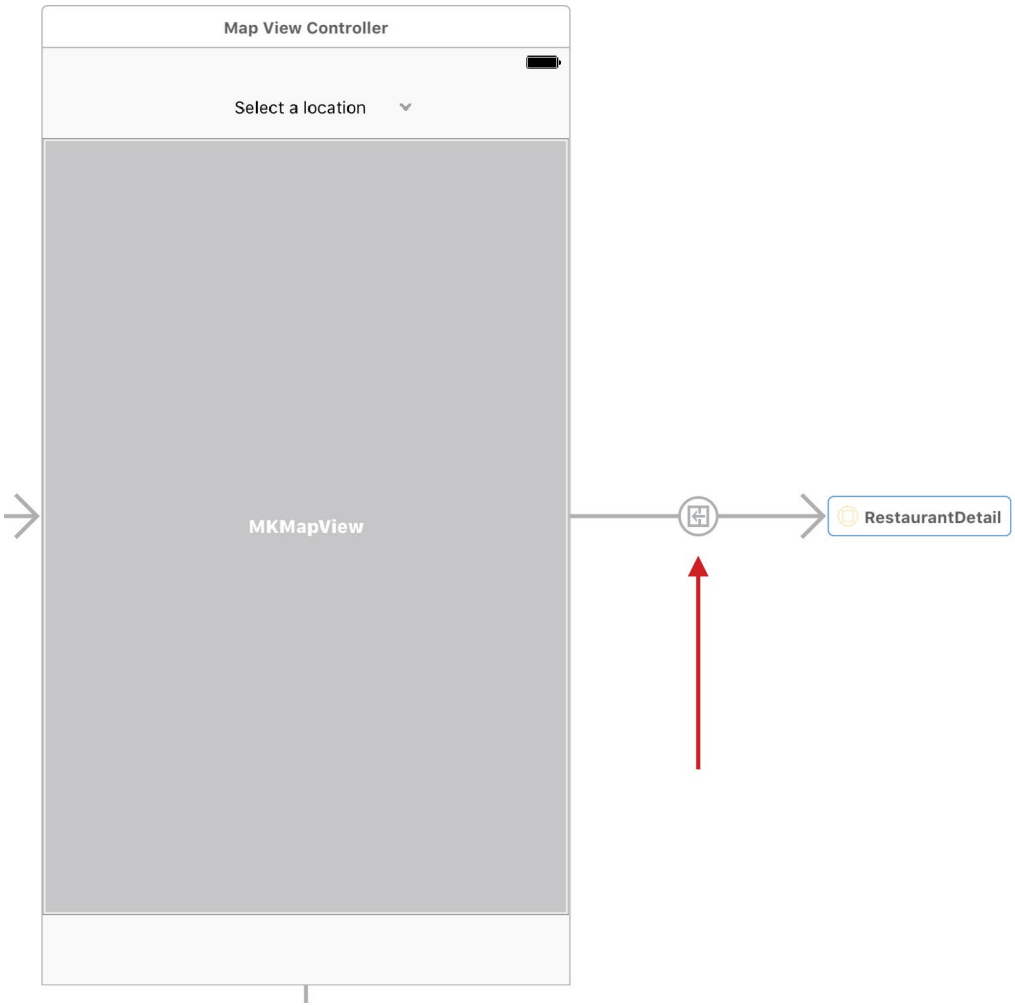
2. Next, select the **Attributes Inspector** in the **Utilities** panel, and update Storyboard under **Storyboard Reference** to say `RestaurantDetail`. Then, hit *Enter*:



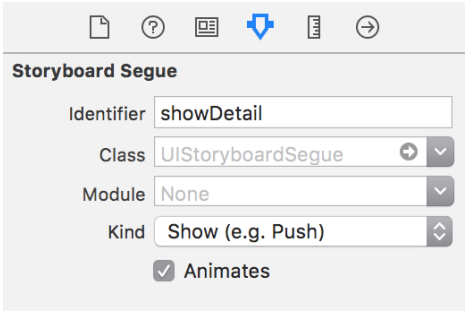
3. *CTL* drag from **Map View Controller** to the Storyboard reference we just created, and select **Show** in the screen that appears. Note that you can *CTL* drag from either the **Map View Controller** in the **Outline** view or the Map View Controller icon in the scene as shown in the following screenshot:



4. Select the segue connecting the **Map View Controller** to the Storyboard reference:



5. In the **Attributes Inspector**, update **Identifier** under **Storyboard Segue** to say `showDetail`. Then, hit *Enter*:



This identifier is what we are going to call whenever the restaurant detail disclosure is tapped. Therefore, let's connect our segue next.

Map to restaurant detail

Before we connect our segue, we should create an enumeration (an enum for short) to keep track of our segues. An enum is a user-defined data type, which consists of a set of related values.

1. Right-click on the `Misc` folder inside the `Common` folder and select **New File**.
2. Inside of the **Choose a template for your new file** screen, select **iOS** at the top and then **Swift File**. Then, hit **Next**.
3. Name this file `Segue` and hit `Create`.
4. Under `import Foundation` in the new file, add the following:

```
enum Segue:String {
    case showDetail
}
```

That is all we need for this file.

The next thing we need is to know when the user taps the detail disclosure of the callout.

In the `MapViewController.swift` file, add the following under the `addMap(_ annotations:)` method:

```
func mapView(_ mapView: MKMapView, annotationView view:
MKAnnotationView, calloutAccessoryControlTapped control: UIControl) {
    self.performSegue(withIdentifier: Segue.showDetail.rawValue,
sender: self)
}
```

We are using `performSegue()` in order to call our custom segue. Now, when you tap the annotation and, then, the callout, you will go to the restaurant detail view.

```
        mapView.delegate = self

        manager.fetch { (annotations) in
            addMap(annotations)
        }
    }

    func addMap(_ annotations:[RestaurantAnnotation]) {
        mapView.setRegion(manager.currentRegion(latDelta: 0.5, longDelta: 0.5), animated: true)
        mapView.addAnnotations(annotations)
    }

    func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView,
        calloutAccessoryControlTapped control: UIControl) {
        self.performSegue(withIdentifier: Segue.showDetail.rawValue, sender: self)
    }

    func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {
        let identifier = "custompin"

        guard !annotation.isKind(of: MKUserLocation.self) else {
            return nil
        }

        var annotationView:MKAnnotationView?
```

Let's build and run the project by hitting the play button (or use *CMD* + *R*). We can now get to the restaurant detail view from the Map.

Passing data to Restaurant detail

In the next chapter, we are going to display the data in our restaurant detail. For now, we want to pass the data over to the detail view. We really cannot use the `RestaurantAnnotation.swift` file, because that is for the Map. Therefore, we need to create a restaurant item that we can use in both the Map and detail view:

1. Right-click on the `Model` folder in the `Map` folder and select **New File**.

2. Inside of the **Choose a template for your new file** screen, select **iOS** at the top and then **Swift File**. Then, hit **Next**.
3. Name this file, `RestaurantItem` and hit **Create**.
4. Under `import Foundation` in the new file, add the following:

```
struct RestaurantItem {
    var name:String?
    var city:String?
    var address:String?
    var price:Int?
    var state:String?
    var longitude:Float?
    var latitude:Float?
    var cuisines:[String] = []

    var cuisine: String? {
        if cuisines.isEmpty { return "" }
        else if cuisines.count == 1 { return cuisines.first }
        else { return cuisines.joined(separator: ", ") }
    }
}

extension RestaurantItem {
    init(dict:[String:AnyObject]) {
        name  = dict["name"] as? String
        city = dict["city"] as? String
        address  = dict["address"] as? String
        price = dict["price"] as? Int
        state = dict["state"] as? String
        longitude = dict["lng"] as? Float
        latitude = dict["lat"] as? Float
        if let cuisines = dict["cuisines"] as? [AnyObject] {
            for data in cuisines {
                if let cuisine = data["cuisine"] as? String {
                    self.cuisines.append(cuisine)
                }
            }
        }
    }
}
```

5. This file will give us all of the data that we need for both our restaurant listing as well as our restaurant detail. One thing that is different about this file is that we are using what is known as an extension, which we will address in detail later in this chapter. For now, just add the extension to our file.

Next, we need to update our `RestaurantAnnotation.swift` file so that it creates a `RestaurantItem` every time it creates an annotation. Let's make those changes now by opening your `RestaurantAnnotation.swift` file and adding the code highlighted as A-C in the following screenshot:

```
import UIKit
import MapKit

class RestaurantAnnotation: NSObject, MKAnnotation {

    var name: String?
    var cuisines:[String] = []
    var latitude: Double?
    var longitude:Double?
    var address:String?
    var postalCode:String?
    var state:String?
    var imageURL:String?
    var data:[String:AnyObject]? A

    init(dict:[String:AnyObject]) {
        if let lat = dict["lat"] as? Double { self.latitude = lat }
        if let long = dict["lng"] as? Double { self.longitude = long }
        if let name = dict["name"] as? String { self.name = name }
        if let cuisines = dict["cuisines"] as? [String] { self.cuisines = cuisines }
        if let address = dict["address"] as? String { self.address = address }
        if let postalCode = dict["postal_code"] as? String { self.postalCode = postalCode }
        if let state = dict["state"] as? String { self.state = state }
        if let image = dict["image_url"] as? String { self.imageURL = image }

        data = dict B
    }

    var title: String? {
        return name
    }

    var subtitle: String? {
        if cuisines.isEmpty { return "" }
        else if cuisines.count == 1 { return cuisines.first }
        else { return cuisines.joined(separator: ", ") }
    }

    var coordinate: CLLocationCoordinate2D {
        guard let lat = latitude, let long = longitude else { return CLLocationCoordinate2D() }
        return CLLocationCoordinate2D(latitude: lat, longitude: long )
    }

    var restaurantItem:RestaurantItem {
C       guard let restaurantData = data else { return RestaurantItem() }
        return RestaurantItem(dict: restaurantData)
    }
}
```

Let's understand the code we just added:

- **Part A**

  ```
  var data:[String:AnyObject]?
  ```

  Here, we added a new variable called `data`, where we will store the dictionary of restaurant data.

- **Part B**

  ```
  data = dict
  ```

  Next, we set the data to our new variable data.

- **Part C**

  ```
  var restaurantItem:RestaurantItem {
      guard let restaurantData = data else { return RestaurantItem()
  }
      return RestaurantItem(dict: restaurantData)
  }
  ```

  Finally, we created a new variable that will have all the restaurant information in it; when the user taps the detail disclosure, we can pass this to the restaurant detail.

In order to make this work, we need to update both our `RestaurantDetailViewController` (which we have not created yet) and the `MapViewController`. Let's create the `RestaurantDetailViewController`:

1. Right-click on the `Restaurant` folder and create a new group named **Restaurant Detail**.
2. Then, right-click on the new **Restaurant Detail** folder and create a new group named **Controller**.
3. Next, right-click on the new `Controller` folder and select **New File**.
4. Inside of the **Choose a template for your new file** screen, select **iOS** at the top and then **Cocoa Touch Class**. Then, hit **Next**.
5. In the options screen, add the following:
6. New File:
   - **Class**: `RestaurantDetailViewController`
   - **Subclass…**: `UITableViewController`
   - **Also create XIB**: Unchecked
   - **Language**: `Swift`

7. Click on **Next** and then **Create**.
8. Delete everything after the `viewDidLoad()` method, as we do not need all of the other code.
9. Your file should now look as follows:

```
//
//  RestaurantDetailViewController.swift
//  LetsEat
//
//  Created by Craig Clayton on 11/15/16.
//  Copyright © 2016 Craig Clayton. All rights reserved.
//

import UIKit

class RestaurantDetailViewController: UITableViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

    }
}
```

10. Next, inside of the class declaration, add the following:

```
var selectedRestaurant:RestaurantItem?
```

11. Then, add the following code inside of `viewDidLoad()`:

```
print(selectedRestaurant as Any)
```

12. Your file should now look like the following:

```
import UIKit

class RestaurantDetailViewController: UITableViewController {

    var selectedRestaurant:RestaurantItem?

    override func viewDidLoad() {
        super.viewDidLoad()

        print(selectedRestaurant as Any)
    }
}
```

13. Open your `RestaurantDetail.storyboard` file.
14. In the **Outline** view, select the **Table View Controller**.
15. In the **Utilities** panel, select the **Identity Inspector**.
16. Under **Custom Class**, in the **Class** drop-down menu, select **RestaurantDetailViewController** and hit *Enter* in order to connect the View Controller to the class.

This is all we need to do in `RestaurantDetailViewController`. Next, we need to update our **MapViewController:**

1. Open the `MapViewController.swift` file.
2. Directly under where we declare our manager, add the following code:

```
var selectedRestaurant:RestaurantItem?
```

3. Then, add the following code into the `calloutAccessoryControlTapped()` method above `performSegue`:

```
guard let annotation = mapView.selectedAnnotations.first else {
return }
let data = annotation as! RestaurantAnnotation
selectedRestaurant = data.restaurantItem
```

4. Your file should now look as follows:

```
func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView, calloutAccessoryControlTapped control:
    UIControl) {

    guard let annotation = mapView.selectedAnnotations.first else { return }
    let data = annotation as! RestaurantAnnotation
    selectedRestaurant = data.restaurantItem

    self.performSegue(withIdentifier: Segue.showDetail.rawValue, sender: self)
}
```

5. Next, add the following code after `viewDidLoad()`:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?)
{
switch segue.identifier! {
      case Segue.showDetail.rawValue:
          showRestaurantDetail(segue: segue)
      default:
          print("Segue not added")
   }
}
```



You will see an error, but ignore it as we are going to fix this in the next step.

Whenever we transition with a segue, this method gets called. First, we check for the `showDetail` identifier; if this identifier is called, we want to do something (in this case, get the selected restaurant and pass it to the detail view) before we transition.

1. Add the following code after the `addMap(_ annotations:)` method:

```
func showRestaurantDetail(segue:UIStoryboardSegue) {
    if let viewController = segue.destination as?
RestaurantDetailViewController, let restaurant =
selectedRestaurant  {
        viewController.selectedRestaurant = restaurant
    }
}
```

2. Here, we are checking to make sure that the segue destination is going to the `RestaurantDetailViewController`; if so, we make sure that we have a selected restaurant. When it is confirmed that the segue destination is going to the `RestaurantDetailViewController` and we have a selected restaurant, we use the `selectedRestaurant` variable that we created in `RestaurantDetailViewController` and set it to the selected restaurant in `MapViewController`.
3. Your file should now look like the following with the two new methods we just added:

```
override func viewDidLoad() {
    super.viewDidLoad()

    initialize()
}

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    switch segue.identifier! {
    case Segue.showDetail.rawValue:
        showRestaurantDetail(segue: segue)
    default:
        print("Segue not added")
    }
}

func initialize() {
    mapView.delegate = self

    manager.fetch { (annotations) in
        addMap(annotations)
    }
}

func addMap(_ annotations:[RestaurantAnnotation]) {
    mapView.setRegion(manager.currentRegion(latDelta: 0.5, longDelta: 0.5), animated: true)
    mapView.addAnnotations(annotations)
}

func showRestaurantDetail(segue:UIStoryboardSegue) {
    if let viewController = segue.destination as? RestaurantDetailViewController, let restaurant = selectedRestaurant  {
        viewController.selectedRestaurant = restaurant
    }
}
```

Let's build and run the project by hitting the play button (or use *CMD* + *R*) and test out that we can pass data to our `RestaurantDetailViewController`. You should see the following in your **Debug** panel, if everything worked:

```
Optional(LetsEat.RestaurantItem(name: Optional("Red Hill Restaurant"), city:
Optional("Los Angeles"), address: Optional("1325 Echo Park Avenue"), price:
Optional(2), state: Optional("CA"), longitude: Optional(-118.256968), latitude:
Optional(34.077721000000011), cuisines: [], image: nil, restaurantID:
Optional(151228)))
```

We now have our `RestaurantDetailViewController` capable of receiving data; in the next chapter, we will display that data. However, before we write any more code, we should organize our code a bit better.

Organizing your code

Earlier, we wrote an extension for our `RestaurantItem`, in which we created a custom `init()` method that takes a dictionary object. Extensions are useful for adding your own functionality onto standard libraries, structs, or classes, such as arrays, ints, and strings, or onto your own data types, such as RestaurantItem.

Here is an example. Let's say that you wanted to know the length of a String:

```
let name = "Craig"
name.characters.count
```

For us to access the count of the String, we would need to access the characters and then get a count.

Let's simplify this by creating an extension:

```
extension String {
    var length: Int {
        return self.characters.count
    }
}
```

With this newly created String extension, we can now access count by writing the following:

```
let name = "Craig"
name.length
```

As you can see, extensions are very powerful by adding extra functionality without having to change the main class or struct.

Up until now, we paid very little attention to file structure and more attention to understanding what we are writing. Organizing your code is also very important, which is why we are going to refactor our code. The refactoring will mostly consist of copying and pasting code that you have already written. Extensions can help us organize our code better and stay away from cluttering our View Controllers. In addition, we can extend the functionality of View Controllers through extensions. We are going to update four classes: `ExploreViewController`, `RestaurantListViewController`, `LocationViewController`, and `MapViewController`.

Let's start with our `ExploreViewController`:

1. In the `ExploreViewController` file, after the last curly brace, hit *Enter* a couple of times and add the following code (remember this should be outside of the class, not inside):

   ```
   extension ExploreViewController {}
   ```

2. Next, remove the `UICollectionViewDataSource` subclass from the main class and paste it into our extension. Make sure that you delete this from the main class; otherwise, you will get errors. You should now have the following:

   ```
   extension ExploreViewController: UICollectionViewDataSource {}
   ```

3. Now, let's move all of our `CollectionViewDataSource` methods into our extension. You should be moving the following:



```
class ExploreViewController: UIViewController {

    @IBOutlet var collectionView:UICollectionView!
    let manager = ExploreDataManager()

    override func viewDidLoad() {
        super.viewDidLoad()
        manager.fetch()
    }

    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return 1
    }

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return manager.numberOfItems()
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "exploreCell", for: indexPath) as! ExploreCell

        let item = manager.explore(at: indexPath)
        if let name = item.name { cell.lblName.text = name }
        if let image = item.image { cell.imgExplore.image = UIImage(named: image) }

        return cell
    }

    @IBAction func unwindLocationCancel(segue:UIStoryboardSegue) {}
}

extension ExploreViewController: UICollectionViewDataSource {

}
```

4. Your file, including the extension, should now look as follows:

```
import UIKit

class ExploreViewController: UIViewController {

    @IBOutlet var collectionView:UICollectionView!
    let manager = ExploreDataManager()

    override func viewDidLoad() {
        super.viewDidLoad()
        manager.fetch()
    }

    @IBAction func unwindLocationCancel(segue:UIStoryboardSegue) {}
}

extension ExploreViewController: UICollectionViewDataSource {
    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return 1
    }

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return manager.numberOfItems()
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "exploreCell", for: indexPath) as! ExploreCell

        let item = manager.explore(at: indexPath)
        if let name = item.name { cell.lblName.text = name }
        if let image = item.image { cell.imgExplore.image = UIImage(named: image) }

        return cell
    }
}
```

We completed cleaning up our `ExploreViewController`. Now, let's update our `RestaurantListViewController`:

1. Inside our `RestaurantListViewController`, after the last curly brace, hit *Enter* a couple of times and add the following code (remember this should be outside of the class, not inside):

   ```
   extension RestaurantListViewController {}
   ```

2. Next, remove the `UICollectionViewDataSource` subclass from the main class and move it into our extension. Make sure that you delete this from the main class; otherwise, you will get errors. You should now have the following:

   ```
   extension RestaurantListViewController: UICollectionViewDataSource
   {}
   ```

3. Now, let's move all of our `CollectionViewDataSource` methods into our extension. You should be moving the following:

```
import UIKit

class RestaurantListViewController: UIViewController {

    @IBOutlet var collectionView:UICollectionView!


    override func viewDidLoad() {
        super.viewDidLoad()

    }

    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return 1
    }

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return 20
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "restaurantListCell", for: indexPath)

        return cell
    }
}

extension RestaurantListViewController: UICollectionViewDataSource {

}
```

4. Your file, including the extension, should now look as follows:

```
import UIKit

class RestaurantListViewController: UIViewController {

    @IBOutlet var collectionView:UICollectionView!


    override func viewDidLoad() {
        super.viewDidLoad()

    }
}

extension RestaurantListViewController: UICollectionViewDataSource {
    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return 1
    }

    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return 20
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "restaurantListCell", for: indexPath)

        return cell
    }
}
```

We successfully updated our `RestaurantListViewController`. Next, let's take a look at our `LocationViewController`:

1. Inside of our `LocationViewController`, after the last curly brace, hit *Enter* a couple of times and add the following code (remember this should be outside of the class, not inside):

   ```
   extension LocationViewController {}
   ```

2. Next, remove the `UITableViewDataSource` subclass from the main class and move it into our extension. Make sure that you delete this from the main class; otherwise, you will get errors. You should now have the following:

   ```
   extension LocationViewController: UITableViewDataSource {
   }
   ```

3. Now, let's move all of our `TableViewDataSource` methods into our extension. You should be moving the following:

```
import UIKit

class LocationViewController: UIViewController {

    @IBOutlet var tableView:UITableView!
    let manager = LocationDataManager()

    override func viewDidLoad() {
        super.viewDidLoad()
        manager.fetch()
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return manager.numberOfItems()
    }

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

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "locationCell", for: indexPath) as UITableViewCell
        cell.textLabel?.text = manager.locationItem(at: indexPath)

        return cell
    }
}

extension LocationViewController: UITableViewDataSource {

}
```

4. Your file, including the extension, should now look as follows:

```
import UIKit

class LocationViewController: UIViewController {

    @IBOutlet var tableView:UITableView!
    let manager = LocationDataManager()

    override func viewDidLoad() {
        super.viewDidLoad()
        manager.fetch()
    }
}

extension LocationViewController: UITableViewDataSource {

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return manager.numberOfItems()
    }

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

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "locationCell", for: indexPath) as UITableViewCell
        cell.textLabel?.text = manager.locationItem(at: indexPath)

        return cell
    }
}
```

5. We completed cleaning up our `LocationViewController`. Finally, let's take a look at our `MapViewController`:
6. Inside of our `MapViewController`, after the last curly brace, hit Enter a couple of times and add the following code (remember this should be outside of the class, not inside):

   ```
   extension MapViewController {}
   ```

7. Next, remove the `MKMapViewDelegate` subclass from the main class and move it into our extension. Make sure that you delete this from the main class; otherwise, you will get errors. You should now have the following:

   ```
   extension MapViewController: MKMapViewDelegate {}
   ```

8. Now, let's move all of our `MKMapViewDelegate` methods into the extension. You should be moving the following:

```
import UIKit
import MapKit

class MapViewController: UIViewController, MKMapViewDelegate {

    @IBOutlet var mapView: MKMapView!
    let manager = MapDataManager()
    var selectedRestaurant:RestaurantItem?

    override func viewDidLoad() {
        super.viewDidLoad()

        initialize()
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        switch segue.identifier! {
        case Segue.showDetail.rawValue:
            showRestaurantDetail(segue: segue)
        default:
            print("Segue not added")
        }
    }

    func initialize() {
        mapView.delegate = self
        manager.fetch { (annotations) in
            addMap(annotations)
        }
    }

    func addMap(_ annotations:[RestaurantAnnotation]) {
        mapView.setRegion(manager.currentRegion(latDelta: 0.5, longDelta: 0.5), animated: true)
        mapView.addAnnotations(annotations)
    }

    func showRestaurantDetail(segue:UIStoryboardSegue) {
        if let viewController = segue.destination as? RestaurantDetailViewController, let restaurant = sel
            viewController.selectedRestaurant = restaurant
        }
    }
    func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView, calloutAccessoryControlTapped control:
        UIControl) {

        guard let annotation = mapView.selectedAnnotations.first else { return }
        let data = annotation as! RestaurantAnnotation
        selectedRestaurant = data.restaurantItem

        self.performSegue(withIdentifier: Segue.showDetail.rawValue, sender: self)
    }

    func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {
        let identifier = "custompin"

        guard !annotation.isKind(of: MKUserLocation.self) else {
            return nil
        }

        var annotationView:MKAnnotationView?

        if let customAnnotationView = mapView.dequeueReusableAnnotationView(withIdentifier: identifier) {
            annotationView = customAnnotationView
            annotationView?.annotation = annotation
        }
        else {
            let av = MKAnnotationView(annotation: annotation, reuseIdentifier: identifier)
            av.rightCalloutAccessoryView = UIButton(type: .detailDisclosure)
            annotationView = av
        }

        if let annotationView = annotationView {
            annotationView.canShowCallout = true
            annotationView.image = UIImage(named: "custom-annotation")
        }

        return annotationView
    }
}
```

9. Your file, including your extension, should now look as follows:

```
import UIKit
import MapKit

class MapViewController: UIViewController {

    @IBOutlet var mapView: MKMapView!
    let manager = MapDataManager()
    var selectedRestaurant:RestaurantItem?

    override func viewDidLoad() {
        super.viewDidLoad()

        initialize()
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        switch segue.identifier! {
        case Segue.showDetail.rawValue:
            showRestaurantDetail(segue: segue)
        default:
            print("Segue not added")
        }
    }

    func initialize() {
        mapView.delegate = self
        manager.fetch { (annotations) in
            addMap(annotations)
        }
    }

    func addMap(_ annotations:[RestaurantAnnotation]) {
        mapView.setRegion(manager.currentRegion(latDelta: 0.5, longDelta: 0.5), animated: true)
        mapView.addAnnotations(annotations)
    }

    func showRestaurantDetail(segue:UIStoryboardSegue) {
        if let viewController = segue.destination as? RestaurantDetailViewController, let restaurant = selectedRestaurant  {
            viewController.selectedRestaurant = restaurant
        }
    }
}

extension MapViewController:MKMapViewDelegate {
    func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView, calloutAccessoryControlTapped control:
        UIControl) {

        guard let annotation = mapView.selectedAnnotations.first else { return }
        let data = annotation as! RestaurantAnnotation
        selectedRestaurant = data.restaurantItem

        self.performSegue(withIdentifier: Segue.showDetail.rawValue, sender: self)
    }

    func mapView(_ mapView: MKMapView, viewFor annotation: MKAnnotation) -> MKAnnotationView? {
        let identifier = "custompin"

        guard !annotation.isKind(of: MKUserLocation.self) else {
            return nil
        }

        var annotationView:MKAnnotationView?

        if let customAnnotationView = mapView.dequeueReusableAnnotationView(withIdentifier: identifier) {
            annotationView = customAnnotationView
            annotationView?.annotation = annotation
        }
        else {
            let av = MKAnnotationView(annotation: annotation, reuseIdentifier: identifier)
            av.rightCalloutAccessoryView = UIButton(type: .detailDisclosure)
            annotationView = av
        }

        if let annotationView = annotationView {
            annotationView.canShowCallout = true
            annotationView.image = UIImage(named: "custom-annotation")
        }

        return annotationView
    }
}
```

We finished cleaning up the four View Controllers.

You might be wondering what are the benefits of this. In this project, it may not seem like these updates are very important, because we are not doing a lot in our View Controllers. However, as a project grows, there will be some cases, where multiple protocols and/or delegates are adopted; thus, these updates will be extremely helpful.

Here is an example:

```
class NewsListingView: UIViewController, NewsListingViewProtocol,
UICollectionViewDelegate, UICollectionViewDataSource,
LiveGameNewsViewDelegate, UIGestureRecognizerDelegate
```

This class is subclassing a View Controller and adopting one protocol, three delegates and one data source. If you have two methods for each one that you need you, you would have 12 functions in your class that would need to have code. You can see that; without extensions, your View Controller would become massive in no time.



The naming of your extensions and how many extensions is completely up to you. However, if we were to create extensions for the preceding example, I would do the following:

One file, `NewsListingDelegate`, would include `NewsListingViewProtocol`, `LiveGameNewsViewDelegate`, and `UIGestureRecongizerDelegate`. Another file, `NewsListingDataSource`, would just include `UICollectionViewDataSource`. Alternatively, you could have a file for each delegate or have one file for all of your delegates.

Summary

In this chapter, we discussed what MKAnnotations are and how to add and subclass them in order to use them in our Map. In addition, you learned how to customize our annotations. Our app now takes us from tapping on an annotation to a restaurant detail page. You also learned that extensions help to organize code as well as add functionality without having to change the main class or struct with which we are working.

In the next chapter, we are going to display data in our restaurant list. We also will set up our restaurant detail page to display data.

11
Where's My Data?

When building iOS apps, data can be the most important part. Typically, the apps you build require getting data from an online data source, known as an **Application Programming Interface** (**API**). In the previous chapters, we have only worked with a `plist` to supply our data. The `plist` bridges the gap to understanding how to work with an API, as you will see shortly. In this chapter, we will work with an API that is in **JavaScript Object Notation** (**JSON**) format. This format is common no matter which backend service was used to create the JSON. In this chapter, we will cover:

- What a JSON file is and the different components of this data feed
- Passing data using segues
- What extensions are and how to use them to clean up your code

For our app, we need one class to handle our data, but we need to share it between both our **Explore View Controller** and **Map View Controller**. Let's first understand the responsibility of our **API Manager**.

Creating an API Manager

In this chapter, we will be building an **API Manager**. This manager will be responsible for anything that has to do with getting data from online. When dealing with data online, you will typically get it in a particular format, which you then need to convert into something that your app can read.

What is an API?

An API is a web service from which an app can receive data. Typically, when you are dealing with APIs, such as **YELP**, they tend to change often. For our purposes, we want to use static files so that we can work through this project without having to be concerned about changes to the API. Therefore, most of the data we are going to use comes from the site, `opentable.herokuapp.com`, which is not managed full time and does not change often. The site's API, however, is missing some data that we need; therefore, I have updated these files (which can be found in the project files for this chapter) to include that missing data.

APIs are typically in JSON format and working with them is similar to working with `plists`. The transition from one to the other should be pretty seamless. Let's get familiar with the JSON format.

Understanding a JSON file

Before we write any code, we need to take a look at the structure of a simple JSON file. Let's create a new group inside the `Common` folder in the Navigator panel called **API Manager**. Then, create a new group inside this new API Manager folder called `json`. Next, drag and drop all of the JSON files found in the project files for this chapter into the new `json` folder, by clicking on **Finish** in the screen that appears. Lastly, open up the `Charleston.json` file and let's review the first part of it, including the first restaurant listing:


```
{
    "total_entries": 67,
    "per_page": 25,
    "current_page": 1,
    "restaurants": [
        {
            "id": 147475,
            "name": "Union Provisions",
            "address": "513 King Street",
            "city": "Charleston",
            "state": "SC",
            "area": "South Carolina",
            "postal_code": "29403",
            "country": "US",
            "phone": "8436410821x",
            "lat": 32.790291,
            "lng": -79.93936,
            "price": 2,
            "reserve_url": "http://www.opentable.com/single.aspx?rid=147475",
            "mobile_reserve_url": "http://mobile.opentable.com/opentable/?restId=147475",
            "image_url": "https://www.opentable.com/img/restimages/147475.jpg",
            "cuisines": [
                {
                    "cuisine": "American"
                },
                {
                    "cuisine": "Bar"
                }
            ]
        },
```

This file has four nodes inside it, `total_entries`, `per_page`, `current_page`, and `restaurants`. When you work with a feed, it will split items up into pages so that you are not trying to load all the data at once. This feed tells us that there are 67 total pages with 25 restaurants per page and that we are currently on page 1. We do not need the first three nodes in this book, since we are just going to load 25 restaurants.

The `restaurants` node, on the other hand, is important for purposes of this book. The `restaurants` node is an array of data, recognizable as such by the brackets (`[ ]`) used in the node. If you review the individual items in the `restaurants` node, you will notice that everything needed for our app's name, address, city, and so on is covered. This structure is the same as that which we saw in the `plists` earlier in this book. If you look at `cuisines`, you will notice that it is also wrapped inside brackets (`[ ]`). Again, this is exactly what we had in our `plist` data previously. Now that we have an idea of what a JSON file looks like, let's see how we can work with it.

Exploring the API Manager file

We just created our `API Manager` folder; now, let's create the `API Manager` file:

1. Right-click on the `Misc` folder in the `Common` folder of the **Navigator** panel and select **New File**.
2. Inside the **Choose a template for your new file** screen, select **iOS** at the top. Then, select **Swift File**. Hit **Next** after.
3. Name this file, `RestaurantAPIManager`, and hit **Create**.

We need to define our class definition first; therefore, add the following under the `import` statement:

```
import Foundation

struct RestaurantAPIManager {
    static func loadJSON(file name:String) -> [[String:AnyObject]] {
        var items = [[String : AnyObject]]()

        guard let path = Bundle.main.path(forResource: name, ofType: "json"),
            let data = NSData(contentsOfFile: path) else {
            return [[:]]
        }

        do {
            let json = try JSONSerialization.jsonObject(with: data as Data, options: .allowFragments) as AnyObject
            if let restaurants = json["restaurants"] as? [[String: AnyObject]] {
                items = restaurants as [[String : AnyObject]]
            }
        }
        catch {
            print("error serializing JSON: \(error)")
            items = [[:]]
        }

        return items
    }
}
```

- **Part A:**

```
struct RestaurantAPI Manager {
static func loadJSON(file name:String) -> [[String:AnyObject]] {
```

Here, we defined the class.

- **Part B:**

```
RestaurantAPIManager.loadJSON(file: "File name goes here")
```

The `loadJSON()` method is known as a **type** method, because it has the keyword `static` in front of it. Type methods are called using the dot syntax. Static functions cannot be overridden. When we want to call the `loadJSON` method inside the `RestaurantAPIManager` file, we would write the following:

- **Part C:**

```
var items = [[String : AnyObject]]()
```

Calling this method will return an array of dictionary objects. If this sounds familiar, it is because our `plist` data returns the same thing.

- **Part D:**

```
guard let path = Bundle.main.path(forResource: name, ofType:
"json"), let data = NSData(contentsOfFile: path) else {
   return [[:]]
}
```

On this line, we are declaring an array of dictionary objects.

- **Part E:**

```
do {
let json = try JSONSerialization.jsonObject(with: data as Data,
options: .allowFragments) as AnyObject
if let restaurants = json["restaurants"] as? [[String: AnyObject]]
{
       items = restaurants as [[String : AnyObject]]
  }
}
```

Since we are not loading from the Internet, we need to make sure to call the right file name. If the path is found and there is nothing wrong with the data, we will use the data. Otherwise, we will return an empty array with no dictionary objects.

- **Part F:**

  Here, we are using a `do-catch`; in order to employ it, we must utilize it with what is known as a `try`. We first try to serialize or convert the data from the JSON file; and, if that is successful, we can then access the information inside that file. In order to access the restaurant items in the JSON file (all of which are located inside the `restaurants` node), we used `json["restaurants"]`.

- **Part G:**

```
catch {
    print("error serializing JSON: \(error)")
    items = [[:]]
}
```

  Next, we cast this using the `as?` as an array of dictionary objects. In addition, since our data types are mixed, we used `AnyObject` in order to accept the dictionary of mixed data types. Finally, we set our data to the array of items. We now have the exact same structure, an array of dictionary objects that we had in the `Map` section.

- **Part H:**

```
return items
```

  This `catch` will run only if there is a problem serializing the data from the file. If there is a problem, we will return an empty array with no dictionary objects. This allows for our app to keep running without crashing.

- **Part I:**

  Finally, if all goes well, we return the array of dictionary items back.

This entire class was built so that we can pass any name we want; it will return data if it finds the file.

Location list

Let's review how our app will work. A user will select a cuisine and location. Then, the location will be passed to the **Explore** View. The user will get restaurants from the selected location filtered by the selected cuisine.

If this were online, we would pass the location to the API, and the API would return the JSON data. As you can see, we are doing the same. When you eventually deal with an API, the transition of working with online data will be seamless.

Selecting a location

Therefore, as stated earlier, in order to get data, we need a location. In order to get the location, we need to get it from the `LocationViewController`. When a location is selected, we will show a checkmark. We will need this checkmark to update each time a new item is set. Finally, when the `Done` button is tapped, we need to pass this location to `ExploreViewController`.

Let's update our `LocationViewController` first. We need a variable to keep track of the selected location. Add the following inside the `LocationViewController.swift` file, under the constant manager:

```
var selectedCity:String?
```

Then, update the current extension declaration by subclassing `UITableViewDelegate` as follows:

```
extension LocationViewController: UITableViewDataSource,
UITableViewDelegate
```

As we discussed earlier in the book, delegates supply the behavior. Here, we want a behavior for when the user selects a **Table View** row and another behavior for when the user deselects the row. First, let's add the selection behavior. Before the last curly brace inside your extension, add the following code:

```
func tableView(_ tableView: UITableView, didSelectRowAt
indexPath:IndexPath) {
    if let cell = tableView.cellForRow(at: indexPath) {
        cell.accessoryType = .checkmark
        selectedCity = manager.locationItem(at:indexPath)
        tableView.reloadData()
    }
}
```

Here, we will get the cell of the selected row and set its `accessoryType` to a checkmark. Then, we will get the location and set it to the `selectedCity` variable. In order to only see the checkmark in our **Table View** cell, we need to remove the disclosure arrow and gray cell selection. Let's update this by doing the following:

1. Open `Explore.storyboard`.
2. Select the Table View `locationCell` the **Location View Controller**.
3. Select the Attributes Inspector in the **Utilities** panel, and update the **Selection** field from **Gray** to **None**.
4. Next, update the **Accessory** field from **Disclosure Indicator** to **None**.

Passing a selected location back to Explore View

Now, we need to be able to hit **Done** and send the selected city back to our `ExploreViewController`. Therefore, we need a selected city as well as an unwind for the **Done** button inside `ExploreViewController`. First, let's get our selected city to display in our **Explore View**:

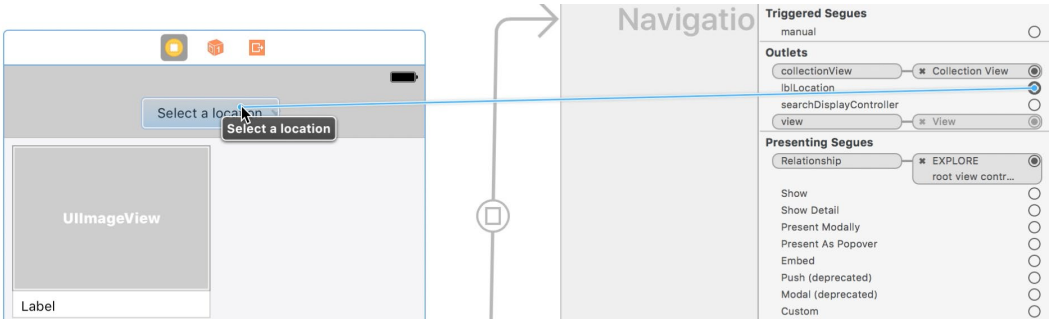
1. Add the following variable under the constant manager in our `ExploreViewController.swift` file:

   ```
   var selectedCity:String?
   ```

2. Then, in order to display the selected location in our custom **Title View**, add the following under `IBOutlet` for `CollectionView`:

   ```
   @IBOutlet var lblLocation:UILabel!
   ```

3. Next, open `Explore.storyboard` and select **Explore View Controller Scene**.
4. Then, in order to connect our selected location to our **Explore View Controller** label in the **Title View**, select the **Connections Inspector** in the **Utilities** panel, and click and drag from the empty circle `lblLocation` under **Outlets** to the label in the **Explore View Controller** Scene:



5. Next, let's unwind our **Done** button in our **Explore View Controller**:
6. Open the `ExploreViewController.swift` file again and, under the `unwindLocationCancel()` function, add the following code:

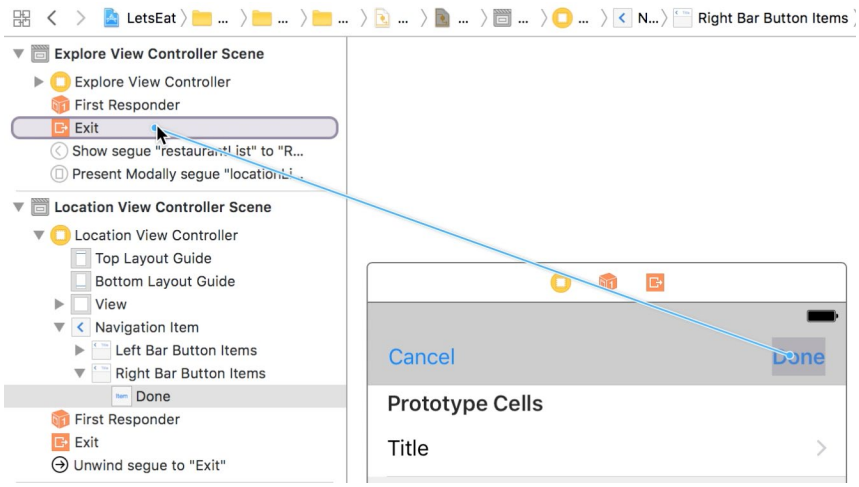
```
@IBAction func unwindLocationDone(segue:UIStoryboardSegue) {
    if let viewController = segue.source as?
LocationViewController {
        selectedCity = viewController.selectedCity
        if let location = selectedCity {
            lblLocation.text = location
```

```
            }
        }
    }
```

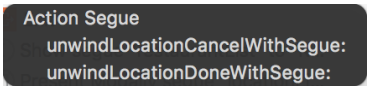
The code we just added is checking the source of the segue. If its source is a class of `LocationViewController`, then we want to grab the selected city and set the `selectedCity` variable inside `ExploreViewController` to that city. We then use an `if-let` statement to make sure that `selectedCity` is not nil; if it is not, then we set the label in the **Title View** to the current selected city.

Finally, we need to hook up this `IBAction`:

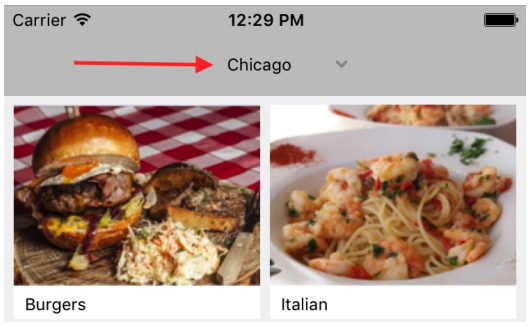
1. Return to `Explore.storyboard`, and in the **Outline** view, make sure that the disclosure arrow in the **Explore View Controller Scene** is opened.
2. Now, Ctrl drag from the **Done** button in the **Location View Controller** to **Exit** in the **Explore View Controller** Scene:



3. When you let go, select `unwindLocationDoneWithSegue:` in the menu that appears:



4. Let's build and run the project by hitting the play button (or use CMD + R).
5. You should now be able to select a location; when you hit **Done**, the **Explore Title** View should show you the selected location:

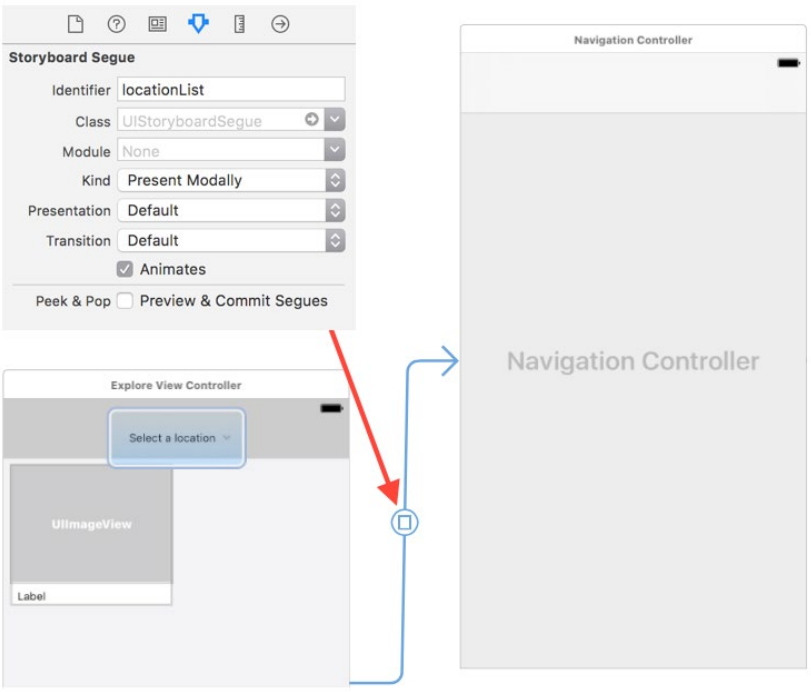


Getting the last selected location

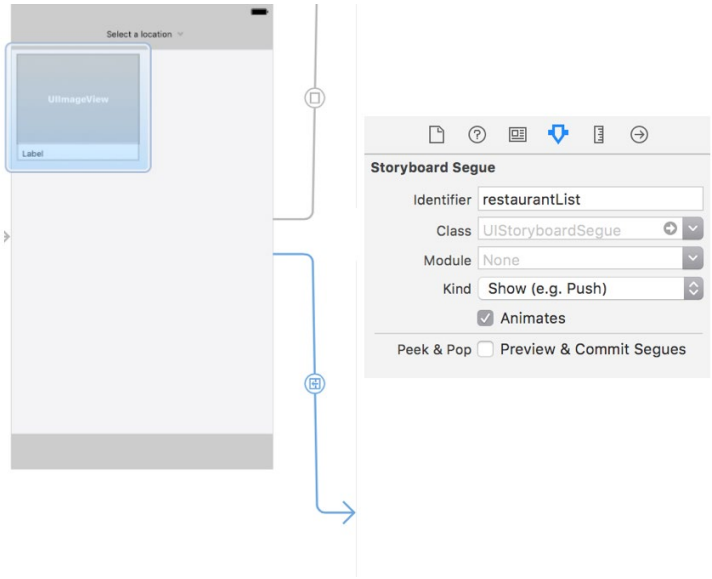
We have a couple of issues that we need to correct under **Select a location**. You will notice that when you click on **Select a location**, you can check multiple locations. We only want the user to be able to select one location. In addition, the checkmark next to your selected location disappears if you click on **Done** in **Location View** and then click to choose a location again. We need to set the last selected location so that it is saved when you go back to your location list. We can address these issues at the same time:

1. While in `Explore.storyboard`, select the segue that is connected to the `LocationViewController`.

2. Select the **Attributes Inspector** in the **Utilities** panel and set "**Identifier**" under **Storyboard Segue** to **locationList**. Then, hit **Enter**:



3. Now, select the segue that is connected to the `RestaurantListViewController` and set **Identifier** to **restaurantList**. Then, hit **Enter**:



4. Next, we need to set up these identifiers. We are going to update the segue `enum`, which we created in the last chapter. Add the following code inside the `Segue.swift` file under the `Misc` folder in the `Common` folder:

```
case restaurantList
case locationList
```

5. Then, open up the `LocationDataManager.swift` file and add the following code before the last curly brace:

```
func findLocation(by name:String) -> (isFound:Bool, position:Int)
{
    guard let index = arrLocations.index(of: name) else { return
(isFound:false, position:0) }

    return (isFound:true, position:index)
}
```

6. This method will allow us to find the location, and then obtain its index position within the array. We will return a tuple, which is a compound type in Swift, meaning that it can hold multiple values. Tuples allow you to combine different data types into one. The method will check the tuple to see whether or not we found the data. If we found the data, then we will use the index position; if not, we will not do anything.

Next, we need to check whether or not a previous location was set. Open up the `LocationViewController.swift` file and create the following method after the `viewDidLoad()` method:

```
func set(selected cell:UITableViewCell, at indexPath:IndexPath) { A
 B if let city = selectedCity {
        let data = manager.findLocation(by: city) C
       if data.isFound {
  D        if indexPath.row == data.position {
               cell.accessoryType = .checkmark
           }
           else { cell.accessoryType = .none }
       }
   }
   else { cell.accessoryType = .none } E
}
```

Let's breakdown this method:

- **Part A:**

  `func set(selected cell:UITableViewCell, at indexPath:IndexPath)`

  In the parameters of this method, we are taking in a cell and an `index` path.

  `if let city = selectedCity`

- **Part B:**

  `let data = manager.findLocation(by: city)`

  Here, we are checking to make sure that selected city is set.

- **Part C:**

```
if data.isFound {
    if indexPath.row == data.position {
        cell.accessoryType = .checkmark
    }
    else { cell.accessoryType = .none }
}
```

  Then, we are calling the method we created in `LocationDataManager`, passing the selected city into the manager and getting back a tuple of data.

- **Part D:**

```
else { cell.accessoryType = .none }
```

  Next, we are checking to see if data was found in the tuple; if so, we are checking to see if the selected row is the same as the position in the array. If the row and position are the same, we are directing the cell to set its `accessoryType` to a checkmark; otherwise, the `accessoryType` will be set to none.

- **Part E:**

  Finally, if no data is found, we are set `accessoryType` to none.

Add the following inside `cellForRowAt()` after we set the text for the cell:

```
set(selected: cell, at: indexPath)
```

Build and run the project by hitting the play button (or use CMD + *R*). You should see that you can only select one location now. However, after you select the location, if you click on "**Done**" in the **Location** view and then click to show locations again, your last selected location is not saved. We still need to address that issue, which we will do next.

Passing location and cuisine to the restaurant list

Open the `ExploreViewController.swift` file and add the following method above the `unwindLocationCancel()` method:

```
func showLocationList(segue:UIStoryboardSegue) {
    guard let navController = segue.destination as?
UINavigationController,
        let viewController = navController.topViewController as?
LocationViewController else {
```

```
                return
        }

        guard let city = selectedCity else { return }
        viewController.selectedCity = city
    }
```

Our `showLocationList()` method will be called whenever our destination view has **Navigation Controller**. Then, it checks to see if the `topViewController` is of the class, `LocationViewController`. If either of these two statements are `false`, we do nothing. If both are `true`, we check the `selectedCity;`. If it is nil, then we also do nothing. If the `selectedCity` has a location, we set the `selectedCity` variable inside the `LocationViewController` to the `selectedCity` in the `ExploreViewController`. This will save the last selected location if we return to the locations list after having selected a location earlier.

We also need to pass the selected city over to the `RestaurantListViewController`. Therefore, add the following variables inside the `RestaurantListViewController.swift` file after your `@IBOutlet` variable:

```
var selectedRestaurant:RestaurantItem?
var selectedCity:String?
var selectedType:String?
```

While still in the `RestaurantListViewController.swift` file, add the following code under the `viewDidLoad()` method:

```
override func viewDidAppear(_ animated: Bool) {
 super.viewDidAppear(animated)
    print("selected city \(selectedCity as Any)")
 print("selected type \(selectedType as Any)")
}
```

The `viewDidAppear()` method will get called every time we load the **View Controller**; whereas, the `viewDidLoad()` method only gets called once. We can print the `selectedCity` variable to verify that we are in fact passing the location over correctly.

Next, open the `ExploreViewController.swift` file again and add the following under the `showLocationList()` method:

```
func showRestaurantListing(segue:UIStoryboardSegue) {
    if let viewController = segue.destination as?
RestaurantListViewController, let city = selectedCity,
        let index = collectionView.indexPathsForSelectedItems?.first,
let type = manager.explore(at: index).name {
```

```
            viewController.selectedType = type
            viewController.selectedCity = city
        }
    }
```

We now check to see if the segue destination is `RestaurantListViewController`; and we make sure that `selectedCity` is set in `ExploreViewController`. Next, we get the selected `indexPath` of the `Collection` view. Once we have that, we then get the item from the `ExploreDataManager` at the index position. Finally, we get the name from the item. If we get all those items back, then we pass the `selectedCity` and `selectedType` variables to the `RestaurantListViewController`. If we do not, then we will display an alert, letting the user know that they need to select a location first.

Let's create the three methods that will display such an alert.

First, we will create the actual alert. While still in the `ExploreViewController`, add the following code before `unwindLocationCancel()`:

```
func showAlert() {
    let alertController = UIAlertController(title: "Location Needed",
message:"Please select a location.", preferredStyle: .alert)
    let okAction = UIAlertAction(title: "OK", style: .default,
handler: nil)
    alertController.addAction(okAction)

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

Then, we need to check that we have a location; if not, we want to make sure that the user cannot go to the restaurant list. Inside the `ExploreViewController`, add the following method after the `viewDidLoad()` method:

```
override func shouldPerformSegue(withIdentifier identifier: String,
sender: Any?) -> Bool {
    if identifier == Segue.restaurantList.rawValue {
        guard selectedCity != nil else {
            showAlert()
            return false
        }
        return true
    }

    return true
}
```

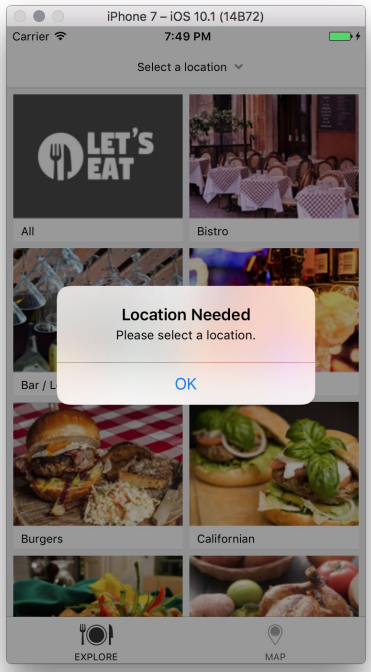
Here, we check whether the segue equals `restaurantList;`. If it does, we check to see if the `selectedCity` variable is set. If we return `true`, then the segue will be performed, and we will go the restaurant list. If we return `false`, then we display our alert, letting the users know that they need to select a location first.

Lastly, we need to show either the location list or restaurant list, depending on whether or not the user chose a location before trying to see the restaurant list. Add the following method after `viewDidLoad()`, and before the `shouldPerformSegue` method we just added:

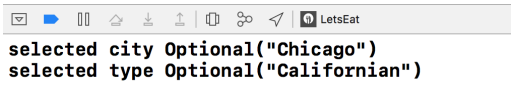
```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    switch segue.identifier! {
    case Segue.locationList.rawValue:
        showLocationList(segue: segue)
    case Segue.restaurantList.rawValue:
        showRestaurantListing(segue: segue)
    default:
        print("Segue not added")
    }
}
```

The `prepare()` method is checks which identifier is being called. If it is the location list, then we call the `showLocationList()` method; if it is the restaurant list, then we call the `showRestaurantListing()` method.

Now, build and run the project by hitting the play button (or use CMD + *R*). If you try to select a cuisine first, you should not be able to go to the restaurant list. Instead, you should receive an alert, stating that you need to select a location:



If you select a location, hit **Done** and then tap the locations list again; you should see your location still selected. Now, if you select a cuisine, you should be directed to the restaurant listing and see the selected location printing in the Debug panel. If you do not see that panel, you can open it using the toggle or CMD + *SHIFT* + *Y*.



Now that we have the location, we need to check our `RestaurantAPIManager` for data. Therefore, let's update our print statement inside the `RestaurantListViewController` by revising the `viewDidAppear()` method to the following:

```
override func viewDidAppear(_ animated: Bool) {

    guard let location = selectedCity, let type = selectedType else {
        return
    }

    print("type \(type)")
    print(RestaurantAPIManager.loadJSON(file: location))
}
```

You should now see the type selected along with an array of dictionary objects in the **Debug** panel.



Now that we have our data, let's get that data to display in our `RestaurantListViewController`. In order to do this, we need to set up our cell as well as a **Restaurant Data Manager**. The **Restaurant Data Manager** rather than the `RestaurantListViewController`, will actually be the class that uses our `RestaurantAPIManager`.

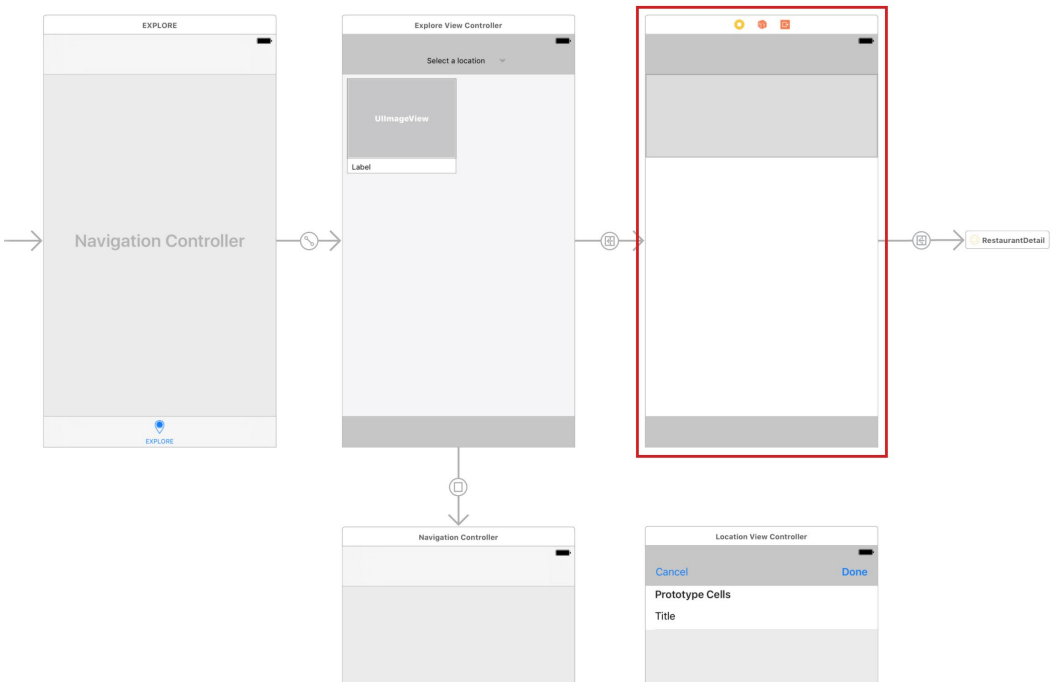
Building our restaurant list

Currently, our restaurant list is not displaying anything other than gray boxes. We need to update this `RestaurantListViewController` in order to display actual restaurants. Let's get started.

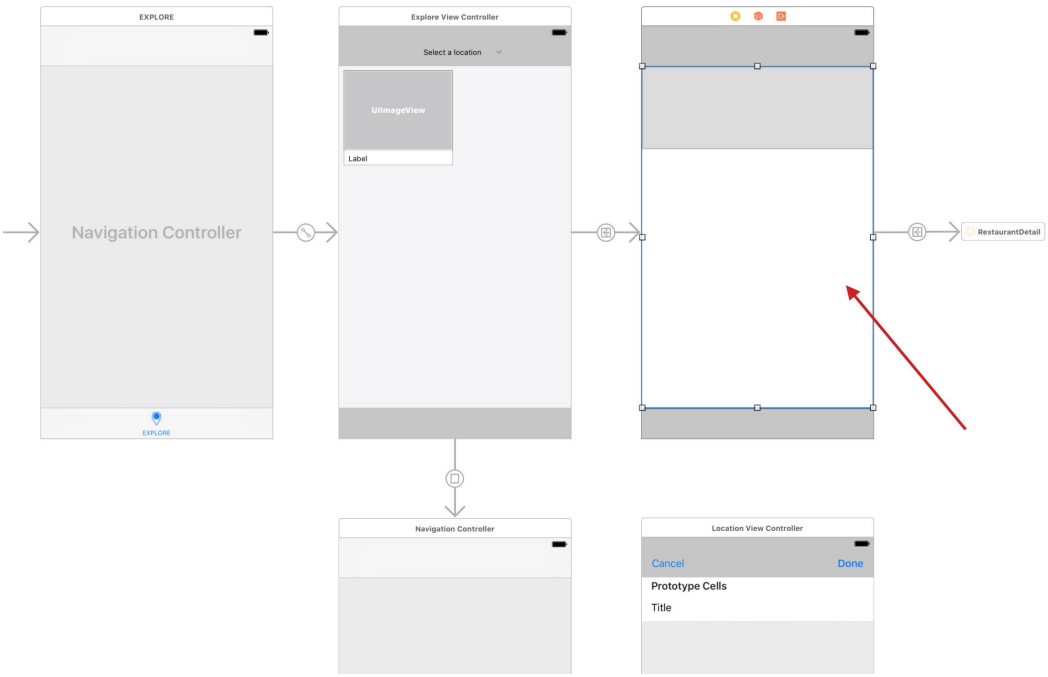
Updating our background

First, we are going to change the background color of our **Collection View**:

1. Open `Explore.storyboard` and locate the `RestaurantListViewController`:



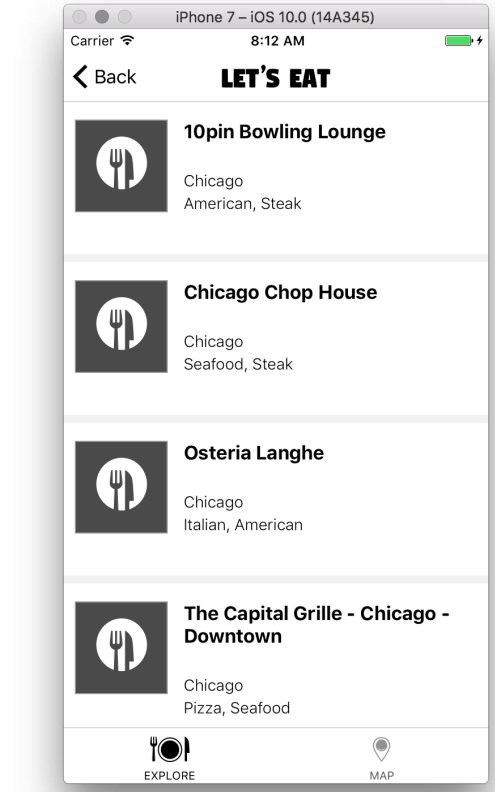
2. Then, select the **Collection View**:



3. Next, open the **Attributes Inspector** in the **Utilities** panel and select **Background**.
4. Under the **Color Sliders** tab, set the **Hex Color #** to `F2F2F4` under RGB **Sliders** in the drop-down menu.

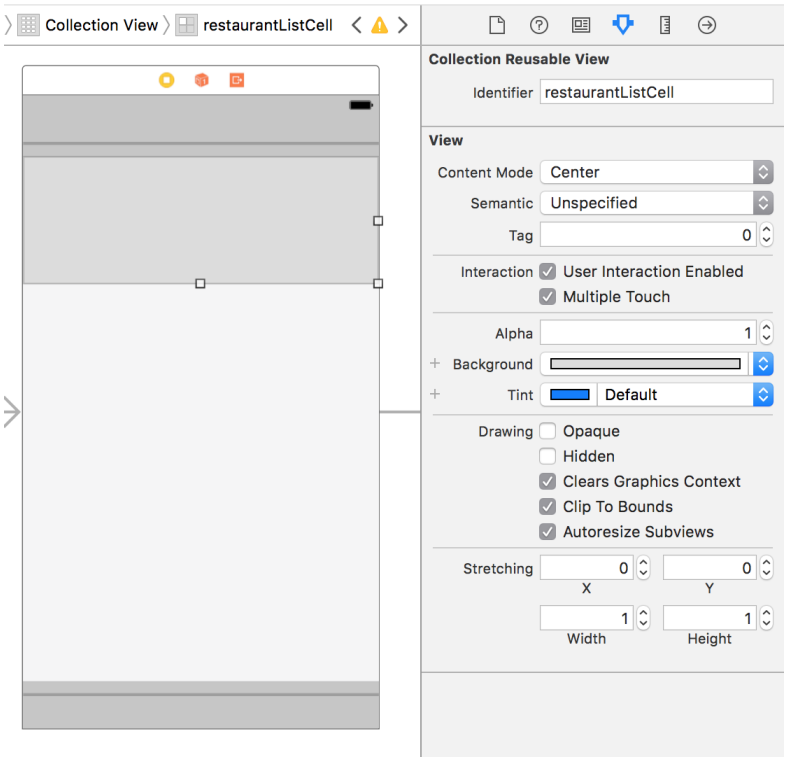
Updating our restaurant list cell

Now, we can start updating our `restaurantListCell`. Let's take a look at the design again, just as a reminder:



1. First, thing we need to do is update the cell background:

2. Select the `restaurantListCell` either in the **Outline view** or the scene. Then, in the **Attributes Inspector**, select **Background**:

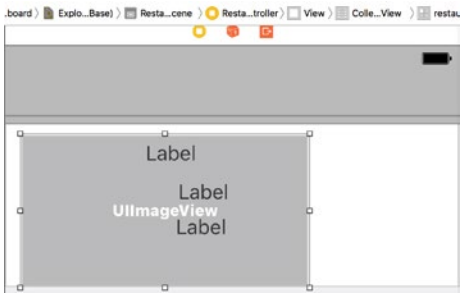


3. Under the **Color Sliders** tab, set the **Hex Color #** to **FFFFFF** under **RGB Sliders** in the drop-down menu.
4. We now need to add three labels and an image.
5. In the **Object library** of the **Utilities** panel, type `label` in the filter field.
6. Drag out three **Labels** into the cell:



7. Next, type `image` into the filter area of the **Object library**.

8. Drag out an **Image View** into the cell:



We now have everything that we need inside our cell and can focus on positioning and sizing these elements.

Positioning elements in our restaurant list cell

Next, let's position our elements inside our cell:

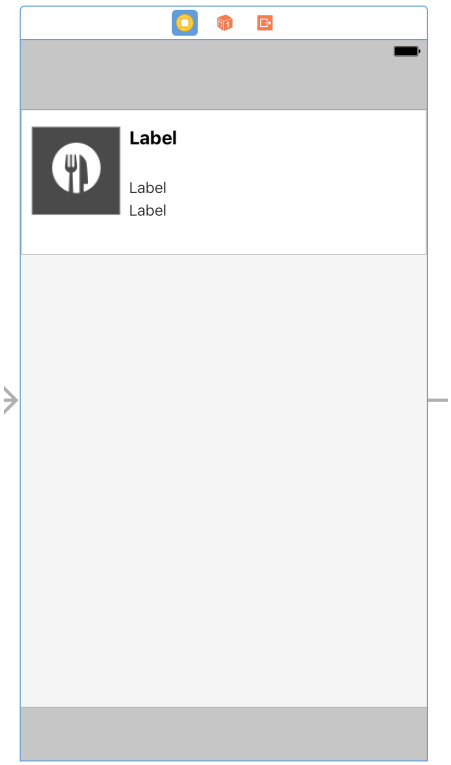
1. Select the **Image View** in the **Outline view**, open the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: `10`
    - **Y**: `16`
    - **Width**: `82`
    - **Height**: `82`
2. Next, select one of the **Labels** in the **Outline** view, and in the **Size Inspector**, update the following values:
    - **X**: `100`
    - **Y**: `16`
    - **Width**: `267`
    - **Height**: `21`
3. Select another **Label**, and update the following values in the **Size Inspector**:
    - **X**: `100`
    - **Y**: `62`
    - **Width**: `267`
    - **Height**: `16`

4. Finally, select the last **Label**, and update the following values in the **Size Inspector**:
    - **X**: `100`
    - **Y**: `83`
    - **Width**: `267`
    - **Height**: `16`
5. Your cell should now look like the following:



Now, that we have our elements where we want them, let's update their fonts:

1. Select the topmost **Label**, open the **Attributes Inspector** in the **Utilities** panel, and change the font style to **Bold**.
2. Select the middle **Label** and change the font to **Light size 14**.
3. Select the last **Label** and change the font to **Light size 14**.
4. Select the **Image View**, and in the **Attributes Inspector**, set **Image** under **ImageView** to `restaurant-list-img` by starting to type it in the space provided and then choosing it from the drop-down menu.
5. You should now see the following in your cell:



Adding auto layout to our restaurant list cell

Now that we completed the design side of the elements in our `restaurantListCell`, we just need to apply **Auto Layout** to these elements.

1. Select the `restaurant-list-img` in the **Outline view** and, then, the **Pin** icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 16

      Left: 10
    - Constrain to margins: unchecked
    - **Height**: 82 (checked)
    - **Width**: 82 (checked)
2. Now, click on **Add 4 Constraints**.

3. Next, select the topmost label and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 16

      Left: 8

      Right: 8
    - **Constrain to margins**: unchecked
    - **Height**: 21 (checked)
4. Click on **Add 4 Constraints**.
5. Next, select the height constraint that we just set for the topmost label.
6. In the **Size Inspector** of the **Utilities** panel, change **Relation** under **Height Constraint** from **Equal** to **Greater Than or Equal**.
7. Then, select the topmost label again and, in the **Attributes Inspector**, change **Lines** under **Label** from `1` to `2`. This will ensure that a longer title will be on two lines.
8. Next, select the middle label and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 25

      Left: 8

      Right: 8
    - Constrain to margins: **unchecked**
    - **Height**: 21 (checked)
9. Click on **Add 4 Constraints**.
10. Finally, select the last **Label** and the **Pin** icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 0

      Left: 8

      Right: 8
    - **Constrain to margins**: unchecked
    - **Height**: 21 (checked)
11. Click on **Add 4 Constraints**.

Creating our restaurant cell class

Now that we have our cell setup, we need to create a file so that we can connect to the cell:

1. Inside the `Restaurant List` folder in the **Navigator** panel, right-click on the **View** folder and select **New File**.
2. Inside the **Choose a template for your new file** screen, select **iOS** at the top and then **Cocoa Touch Class**. Then, hit **Next**.
3. In the options screen that appears, add the following:
    - New File:

      Class: RestaurantCell

      Subclass: UICollectionViewCell

      Also create XIB file: Unchecked

      Language: Swift
4. Click on **Next** and then **Create**.
5. Your file should look like the following:

```
import UIKit

class RestaurantCell: UICollectionViewCell {

}
```

6. Next, open `Explore.storyboard` and select the `restaurantListCell` in the Rnow have everything that we need inside our cell and can focus on positioning **List View Controller** Scene.
7. Now, in the **Utilities** panel, select the **Identity Inspector**.
8. Under **Custom Class**, in the **Class** drop-down menu, select **RestaurantCell** and hit *Enter*.

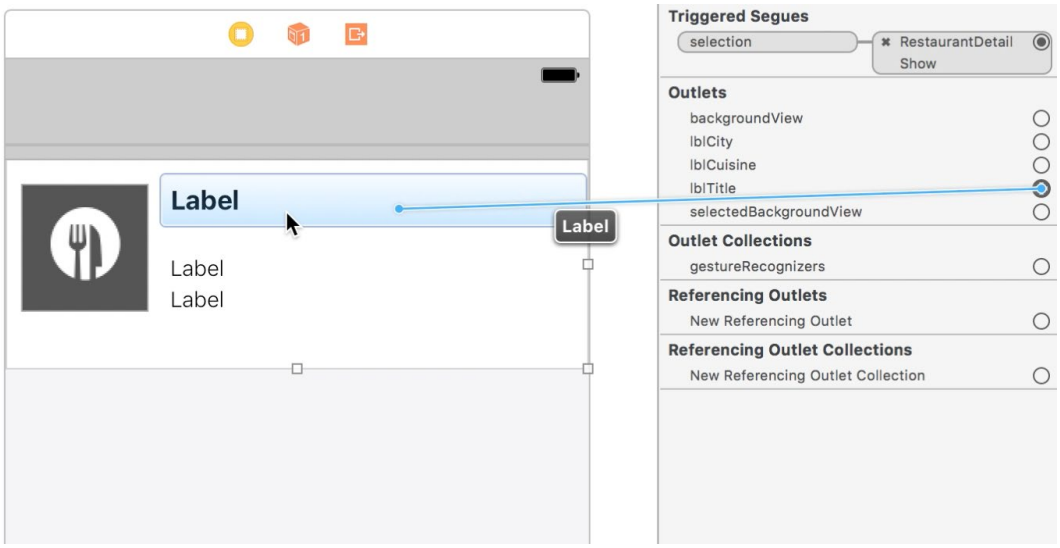
Setting up restaurant list cell outlets

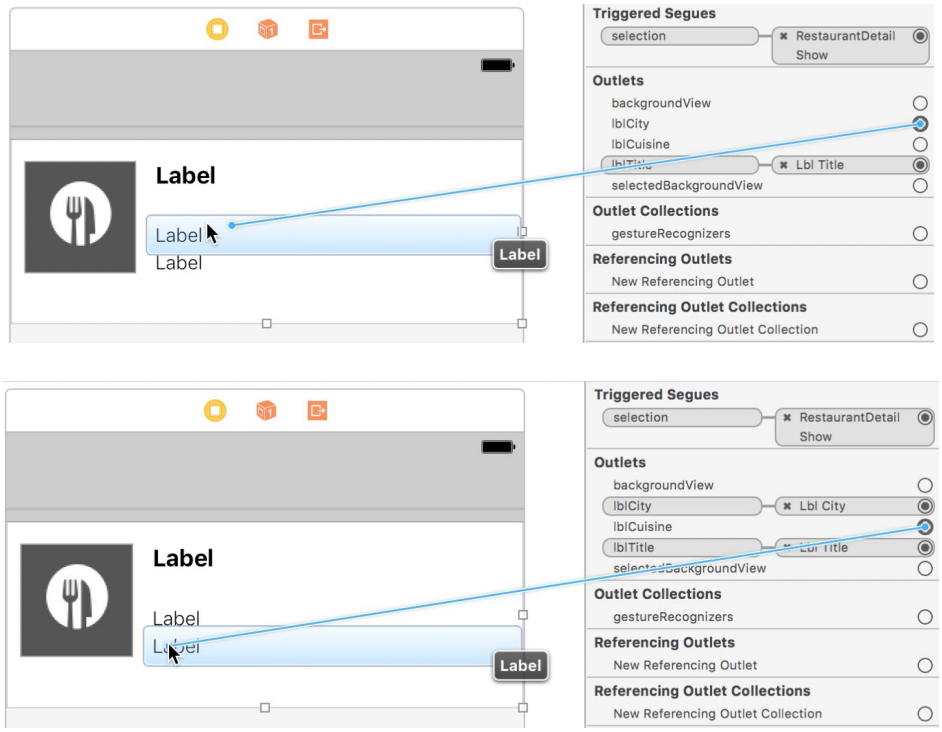
Now, we need to set up our `restaurantListCell` outlets:

1. Open the `RestaurantCell.swift` file in the **Navigator** panel (or click on CMD + *SHIFT* + *O*, type `RestaurantCell`, and hit *Enter*).
2. Inside the class declaration, add the following:

```
@IBOutlet var lblTitle:UILabel!
@IBOutlet var lblCity:UILabel!
@IBOutlet var lblCuisine:UILabel!
```

3. Open `Explore.storyboard` and select our `restaurantListCell` again in the **Outline** view.
4. Then, in the **Utilities** panel, select the **Connections Inspector**, where you will see a list of **Outlets**.
5. Click on and drag from the empty circle `lblTitle` under **Outlets** to the topmost label in our `restaurantListCell`, from the empty circle `lblCity` to the middle label in our cell and from the empty circle `lblCuisine` to the last label in our cell:





Now that we have our `restaurantListCell` outlets set up, let's get some data into our cell. We previously created our `RestaurantItem.swift` file; we will use this in our restaurant list.

Creating RestaurantDataManager

Let's create the `RestaurantDataManager` file now:

1. Inside the `Restaurant List` folder in the **Navigator** panel, create a new group called **Model** if you have not done so previously.
2. Then, right-click on the **Model** folder and select **New File**.
3. Inside the **Choose a template for your new file** screen, select **iOS** at the top and then **Swift File**. Then, hit **Next**:
4. Name this file, **RestaurantDataManager**, and hit **Create**.

We need to define our class definition first, so add the following under the import statement in this new file:

```
class RestaurantDataManager {}
```

Inside the class declaration, add the following variable:

```
    private var items:[RestaurantItem] = []
```

Here, we keep our array private, since there is no reason to have to access this outside of the class.

Now, let's add the following three methods:

```
func fetch(by location:String, withFilter:String="All",
completionHandler:() -> Swift.Void) {
    var restaurants:[RestaurantItem] = []

    for restaurant in RestaurantAPIManager.loadJSON(file: location) {
        restaurants.append(RestaurantItem(dict: restaurant))
    }

    if withFilter != "All" {
        items = restaurants.filter({ $0.cuisines.contains(withFilter)
})
    }
    else { items = restaurants }

    completionHandler()
}

func numberOfItems() -> Int {
    return items.count
}

func restaurantItem(at index:IndexPath) -> RestaurantItem {
    return items[in dex.item]
}
```

The first method here differs from the one we looked at in `ExploreDataManager`; whereas, the last two methods here are basically the same as those in `ExploreDataManager`. Let's breakdown these methods to better understand what we are doing:

```
import Foundation

class RestaurantDataManager {

    private var items:[RestaurantItem] = []  A

    func fetch(by location:String, withFilter:String="All",  completionHandler:() -> Swift.Void) {  C  B
        var restaurants:[RestaurantItem] = []

        for restaurant in RestaurantAPIManager.loadJSON(file: location) {  D
            restaurants.append(RestaurantItem(dict: restaurant))
        }

        if withFilter != "All" {
            items = restaurants.filter({ $0.cuisines.contains(withFilter) })  E
        }
        else { items = restaurants }

        completionHandler()  F
    }

G   func numberOfItems() -> Int {
        return items.count
    }

H   func restaurantItem(at index:IndexPath) -> RestaurantItem {
        return items[index.item]
    }
}
```

- **Part A**:

  ```
  private var items:[RestaurantItem] = []
  ```

  In the parameters of this method, we are passing in a cell and an index path.

- **Part B**:

  ```
  func fetch(by location:String, withFilter:String="All",
  completionHandler:() -> Swift.Void)
  ```

  This function is pretty long; however, we are simply fetching for restaurants with location as a filter. We have a closure block, which will allow us to let the function run until it is complete.

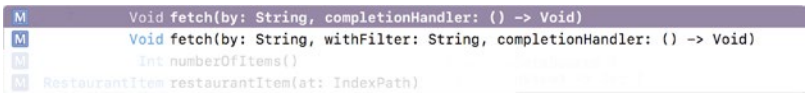
- **Part C**:

  ```
  withFilter:String="All"
  ```

  In this parameter, we are setting a default. If we do not pass anything into this parameter, it will use **All**; otherwise, it will use whatever we give it.



As you type your code, Xcode will provide code hints (choices) that it believes that you might want. When you type this method, Xcode gives you two hints, one that includes the parameter `withFilter` and one that does not:



- **Part D**:

```
for restaurant in RestaurantAPIManager.loadJSON(file: location) {
    restaurants.append(RestaurantItem(dict: restaurant))
}
```

  Here, we get restaurants from the JSON file.

- **Part E**:

```
if withFilter != "All" {
    items = restaurants.filter({ $0.cuisines.contains(withFilter)
})
}
else { items = restaurants }
```

  Here, we are filtering the restaurants by cuisine. Since our restaurants have multiple cuisines, we must check each cuisine, which is why we use `contains`.

- **Part F**:

```
completionHandler()
```

  This is used to tell our method that we are finished.

- **Part G**:

```
func numberOfItems() -> Int
```

  This method tells us how many restaurant items we have.

- **Part H**:

```
func restaurantItem(at index:IndexPath) -> RestaurantItem
```

  This method allows us to get the restaurant at the index position at which it is located.

Now, we have a greater understanding of our Restaurant Data Manager. Although, it is a relatively small file, it does a lot.

Displaying data in Restaurant list cell

Let's update our `RestaurantListViewController` so that it can use our newly created `RestaurantDataManager`. Inside the `RestaurantListViewController` file's class declaration, we need to create an instance of our `RestaurantDataManager`. Therefore, add the following right above our `selectedCity` variable:

```
let manager = RestaurantDataManager()
```

Then, in order to fetch the data for the **Collection View**, inside the `viewDidAppear()` method, delete the two print statements and add the following under `super.viewDidAppear(animated)`:

```
manager.fetch(by: location, withFilter: type, completionHandler: {
    collectionView.reloadData()
})
```

Now, your `viewDidAppear()` method should look like the following:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    guard let location = selectedCity, let type = selectedType else {
        return
    }

    manager.fetch(by: location, withFilter: type, completionHandler: {
        collectionView.reloadData()
    })
}
```

Next, for the `numberOfItemsInSection()` method inside the extension, replace `20` with the following:

```
manager.numberOfItems()
```

Lastly, we need to update our `cellForItemAt` method to say the following:

```
func collectionView(_ collectionView: UICollectionView, cellForItemAt
indexPath: IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentif
ier: "restaurantListCell", for: indexPath) as! RestaurantCell
        let item = manager.restaurantItem(at: indexPath)

        if let name = item.name { cell.lblTitle.text = name }
        if let city = item.city { cell.lblCity.text = city }
```

```
            if let cuisine = item.cuisine { cell.lblCuisine.text = cuisine
    }

            return cell
        }
```

If you build and run the project by hitting the play button (or use *CMD* + *R*), you should see that the Restaurant List is now displaying data filtered based on cuisine. If you select **All** instead of a particular cuisine, then you will see all the restaurants in the area. We now need to get this data passed to our Restaurant Detail View and start displaying the data.

Restaurant details

In order to show data in Restaurant Details, we must first pass the data over to the Restaurant Detail View, just like we did with our Map. We will start with displaying data in our Restaurant Detail View and, then, in our Restaurant List View.

Displaying data in the Restaurant Detail view

First, let's set up our `RestaurantDetailViewController` and add the following:

1. Under import UIKit, add:

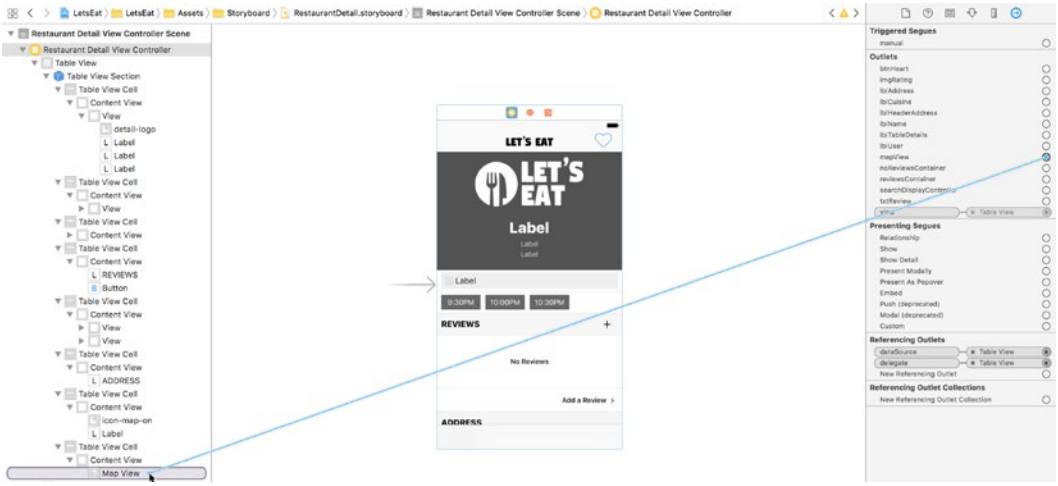
   ```
   import MapKit
   ```

2. Add the following variables after the class declaration and before the `selectedRestaurant` variable:

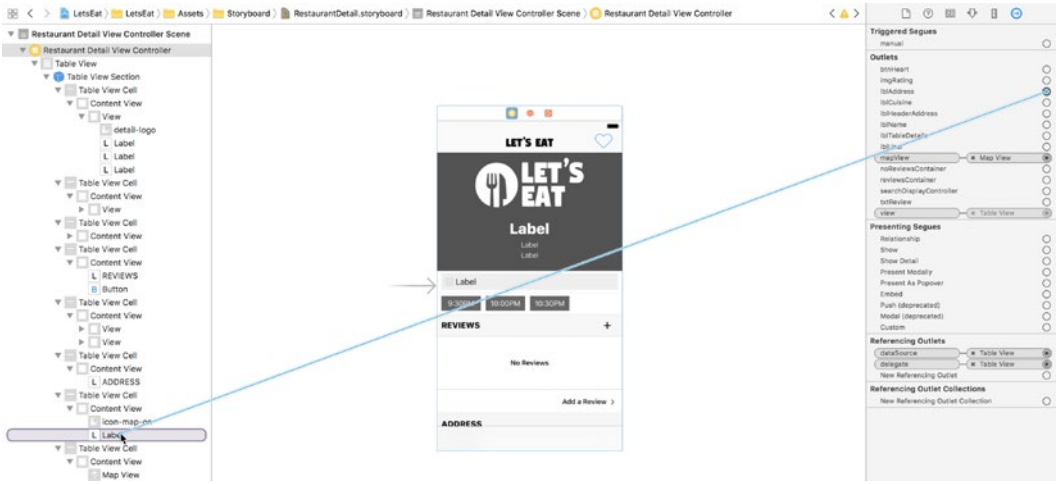
   ```
   @IBOutlet var lblName: UILabel!
   @IBOutlet var lblCuisine: UILabel!
   @IBOutlet var lblHeaderAddress: UILabel!
   @IBOutlet var lblTableDetails: UILabel!
   @IBOutlet var lblAddress: UILabel!
   @IBOutlet var mapView: MKMapView!
   @IBOutlet var reviewsContainer: UIView!
   @IBOutlet var lblUser: UILabel!
   @IBOutlet var txtReview: UITextView!
   @IBOutlet var imgRating: UIImageView!
   @IBOutlet var btnHeart: UIBarButtonItem!
   @IBOutlet var noReviewsContainer: UIView!
   ```

3. Make sure you save the file.
4. Now that we created our `IBOutlets`, we need to connect them (except for the `btnHeart` variable, which we will connect later):

5. Open the `RestaurantDetail.storyboard`, select the Restaurant Detail View Controller in the **Outline** view and, then, open the **Connections Inspector** in the **Utilities** panel.
6. Now, click on and drag from the empty circle of each of the following variables we just added under **Outlets** to their respective elements listed below in either the scene or **Outline** view (ensuring that the disclosure arrows for the **Map**, **Address Label**, and **Review** sections are open).
7. An empty circle for `mapView` to the **Map View** in the **Outline** view:



8. An empty circle for `lblAddress` to the address **Label** in the **Outline** view:



9. An empty circle for `noReviewsContainer` to the **View** in the **Outline** view:

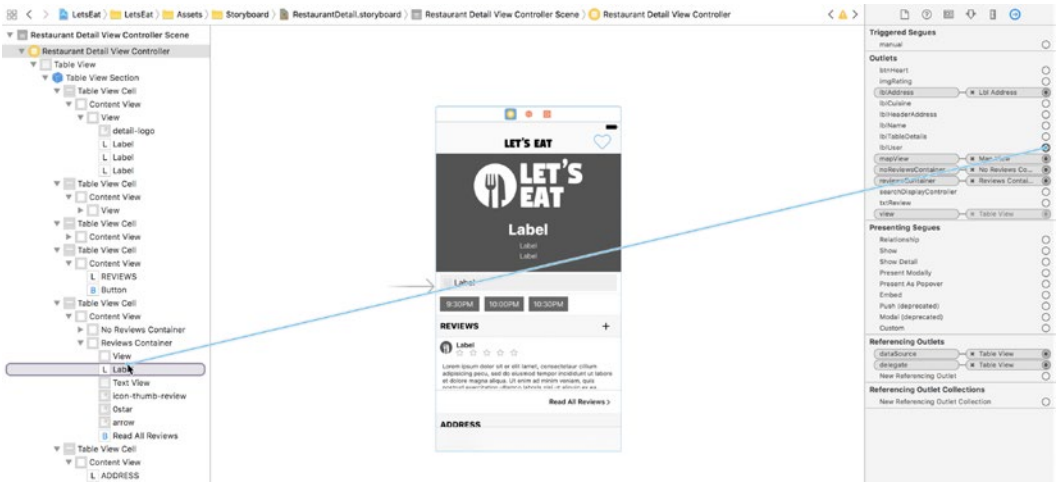


10. An empty circle for `reviewsContainer` to the **View** in the **Outline** view:

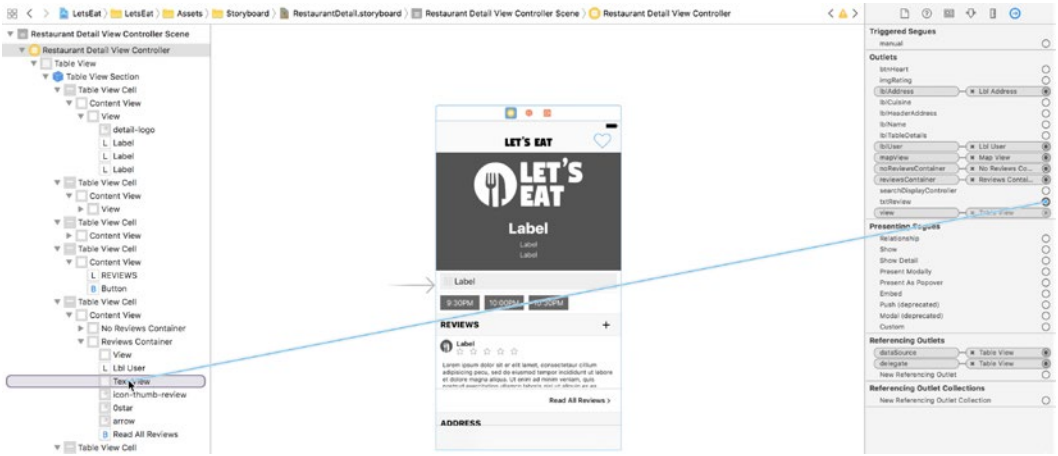


11. Now, swap the **No Reviews** Container with the **Reviews** Container. This way, the **No Reviews** Container is on top, and you can see each element.

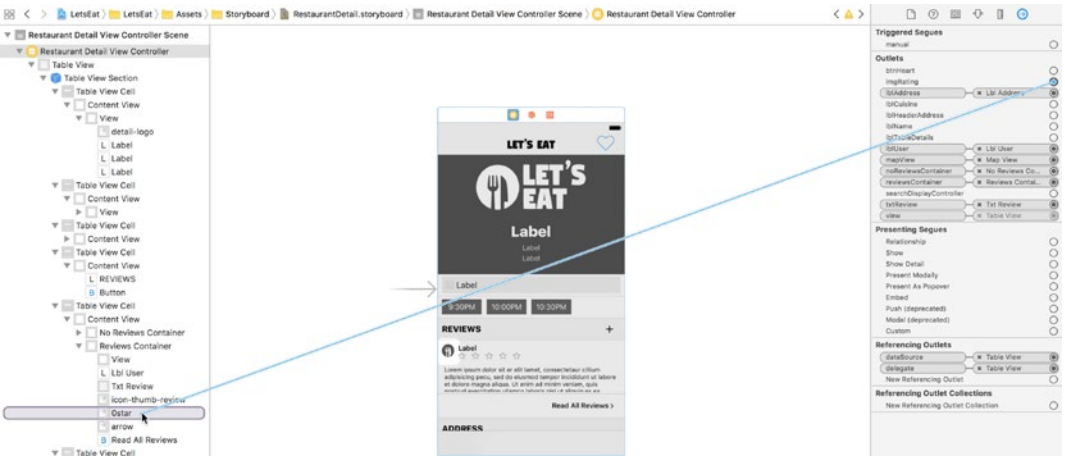
12. An empty circle for `lblUser` to the **Label** inside the **Reviews Container** in the **Outline** view:



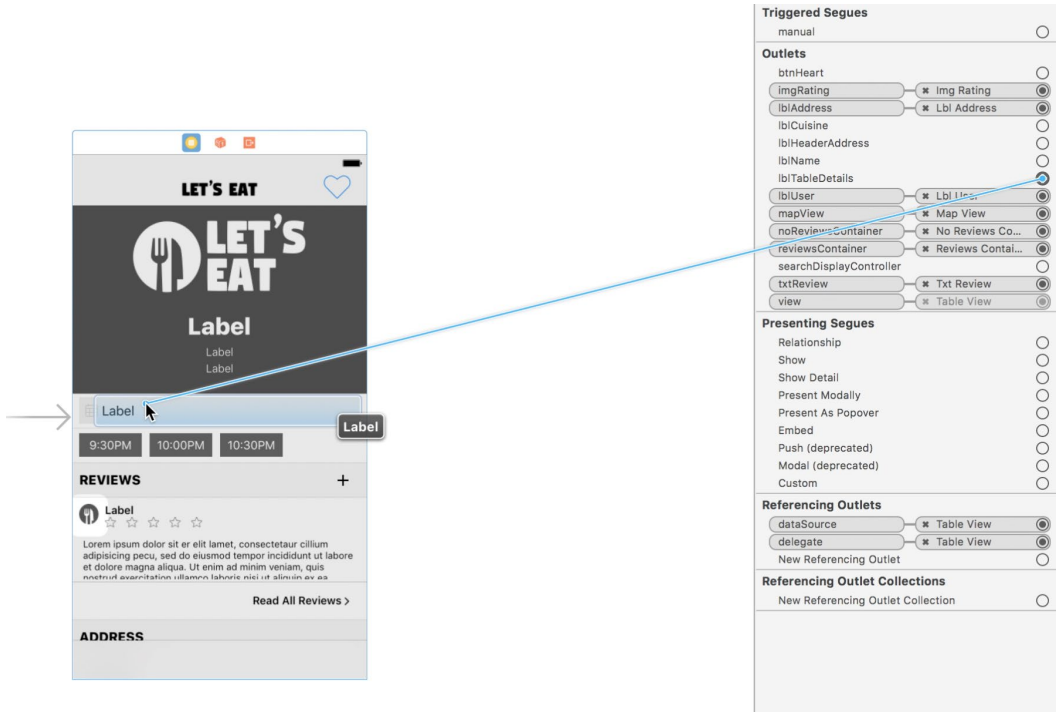
13. An empty circle for `txtReview` to the **Text View** inside the **Reviews Container** in the **Outline** view:



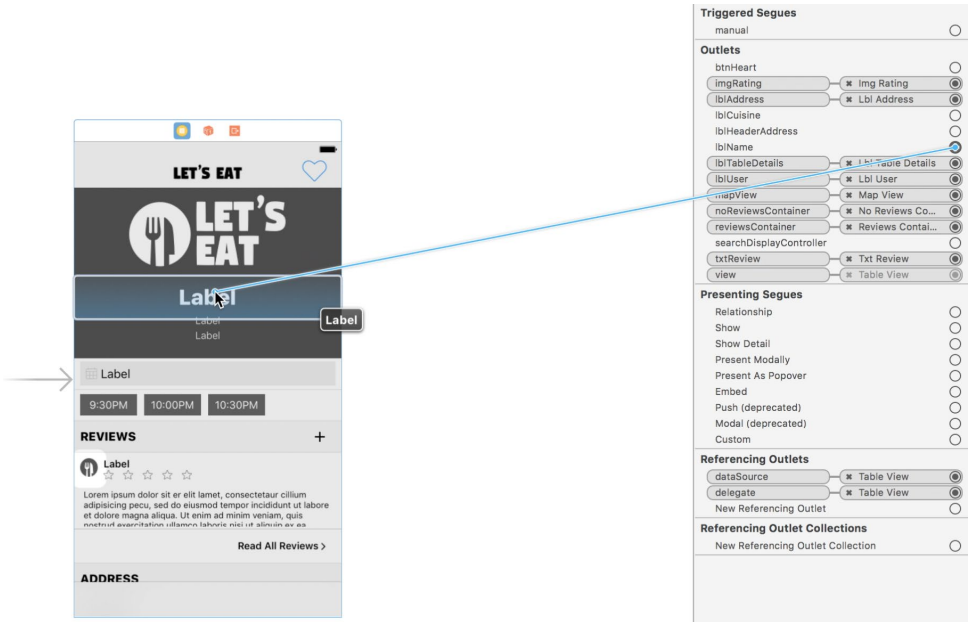
14. An empty circle for `imgRating` to the `0star` image inside the **Reviews** Container in the **Outline** view:



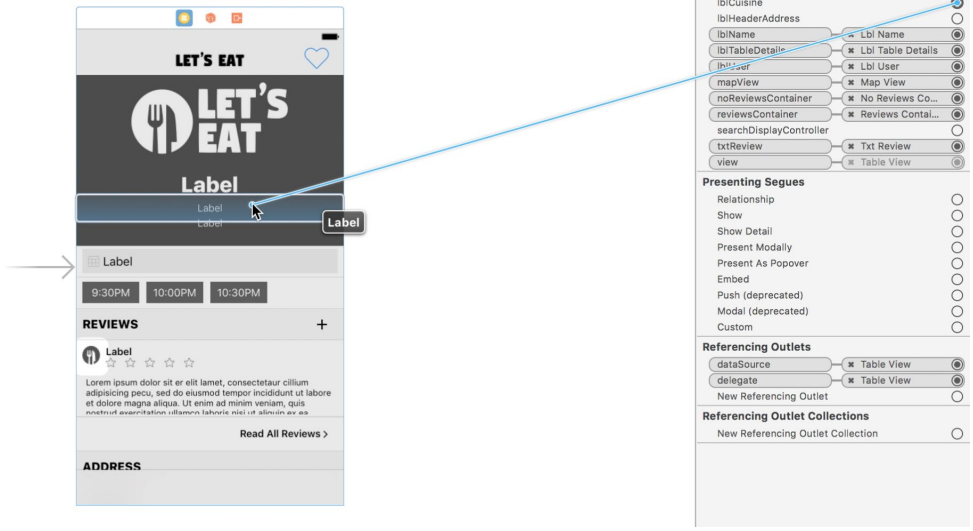
15. An empty circle for `lblTableDetails` to the **Label** under the header in the scene:



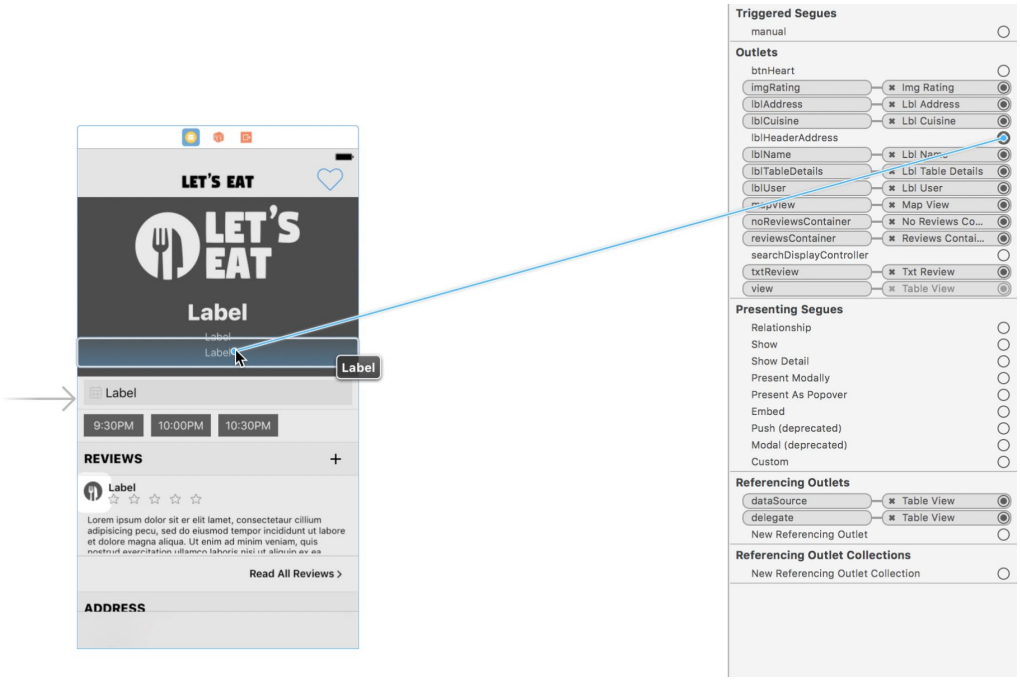
16. An empty circle for `lblName` to the **Label** under the logo in the scene:



17. An empty circle for `lblCuisine` to the label under `lblName` in the scene:



18. An empty circle for `lblHeaderAddress` to the **Label** under `lblCuisine` in the scene:



19. When you are done, swap **No Reviews** Container and **Reviews** Container again so that the **Reviews** Container is back on top.

Next, we need to have a method that will display all of our data in our Labels:

1. Open our `RestaurantDetailViewController.swift` file and add the following before the last curly brace:

```
func setupLabels() {
    guard let restaurant = selectedRestaurant else {
        return
    }

    if let name = restaurant.name {  lblName.text = name }
    if let cuisine = restaurant.cuisine {  lblCuisine.text =
cuisine }
    if let address = restaurant.address {
        lblAddress.text = address
        lblHeaderAddress.text = address
```

```
    }

    lblTableDetails.text = "Table for 7, tonight at 10:00 PM"
}
```

2. This method will now get the data and display it in our Labels.
3. Next, we need to make our Map work in our `RestaurantDetailViewController`. First, we need to update our `RestaurantItem.swift` file. As you learned in the previous chapter, our `RestaurantItem` cannot be used because it does not subclass `MKAnnotation`. Since our data is the same for both `RestaurantItem` and `RestaurantAnnotation`, we will create a variable that will give us an annotation. We did something similar in our `RestaurantAnnotation`.swift file where we used the data to create a `RestaurantItem`.
4. Open the `RestaurantItem.swift` file and add the following variables under the cuisines variable:

```
Var image:String?
Var restaurantID:Int?
var data:[String:AnyObject]?
```

5. Next, add the following variable under the cuisine variable:

```
var annotation:RestaurantAnnotation {
    guard let restaurantData = data else { return
RestaurantAnnotation(dict:[:]) }
    return RestaurantAnnotation(dict: restaurantData)
}
```

6. Then, in our extension, add the following after `latitude = dict["lat"] as? Float`:

```
restaurantID = dict["id"] as? Int
```

7. Next, set up the data by adding the following before the last curly brace of the `init()` method in our extension:

```
image = dict["image"] as? String
data = dict
```

8. Now, in order to make our Map work, and have a pin for our Map, open the `RestaurantDetailViewController.swift` file and add the following under the `setupLabels()` method and before the last curly brace:

```
func setupMap() {
    guard let annotation = selectedRestaurant?.annotation, let
long = annotation.longitude, let lat = annotation.latitude else {
return }

    let location = CLLocationCoordinate2D(
        latitude: lat,
        longitude: long
    )

    let span = MKCoordinateSpanMake(0.5, 0.5)
    let region = MKCoordinateRegion(center: location, span: span)
    mapView.setRegion(region, animated: true)
    mapView.addAnnotations([annotation])
}
```

9. This method takes the restaurant annotation from our selected restaurant and adds the annotation to the screen.

Now that we have our functions created, we just need to call them:

Add the following after the `viewDidLoad()` method in the `RestaurantDetailViewController.swift` file:

```
func initialize() {
    setupLabels()
    setupMap()
}
```

This method now needs to be called inside your `viewDidLoad()` method. Replace the `print` statement in the `viewDidLoad()` method with the following:

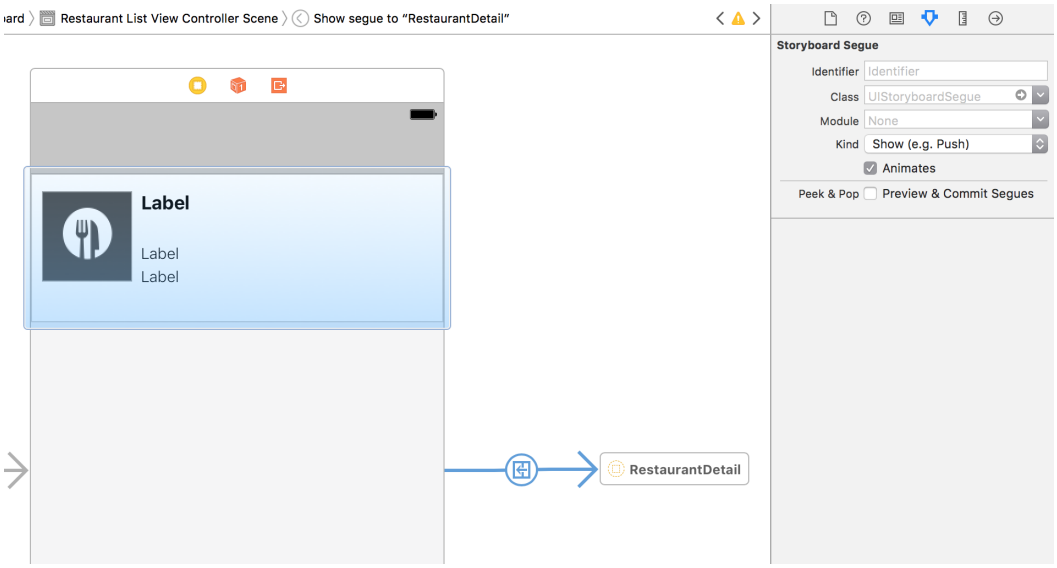
```
initialize()
```

We finished with our Restaurant Detail View Controller. Next, we need to pass the data to our Restaurant List View Controller.

Passing data to our Restaurant List View Controller

We need to make an update to our Storyboard so that we can pass data to our Restaurant List View Controller:

1. Open `Explore.storyboard` and locate the segue that connects our `restaurantListCell` to our `RestaurantDetail.storyboard` reference:



2. In the Attributes Inspector of the **Utilities** panel, update **Identifier** under **Storyboard Segue** to say `showDetail`. Then, hit *Enter*.
3. Finally, open the `Segue.swift` file in the `Misc` folder under the `Common` folder and verify that the following case statement is included, if not, add it:

```
case showDetail
```

Now, we need to set up our RestaurantListViewController so that we can pass data over to our Restaurant Details:

1. Open `RestaurantListViewController.swift` file and add this method before the last curly brace of the class and before the extension:

```
func showRestaurantDetail(segue:UIStoryboardSegue) {
    if let viewController = segue.destination as?
RestaurantDetailViewController, let index = collectionView.
indexPathsForSelectedItems?.first {
        selectedRestaurant = manager.restaurantItem(at: index)
        viewController.selectedRestaurant = selectedRestaurant
    }
}
```

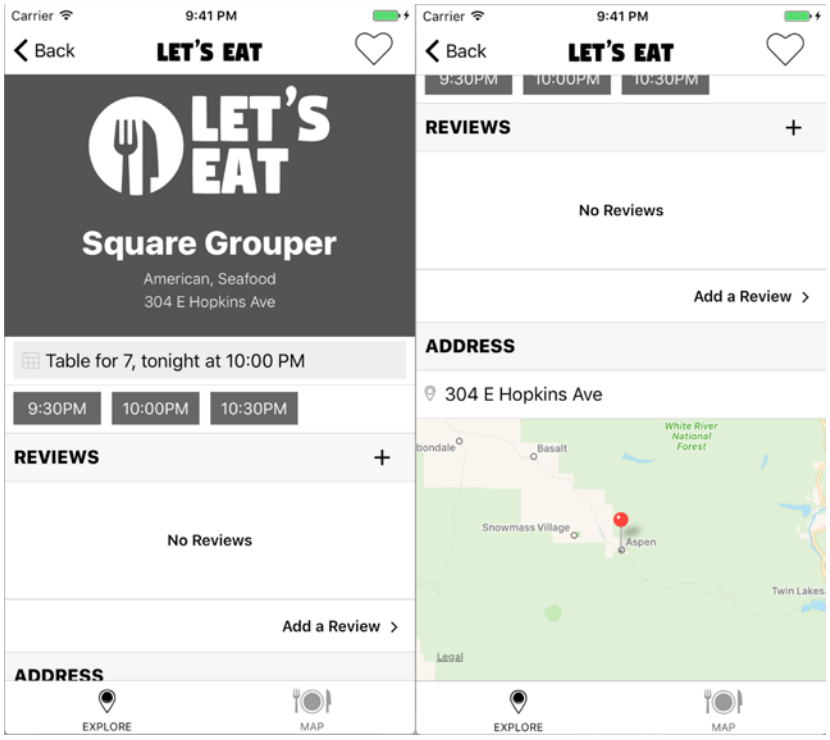
2. This method is called when the `showDetail` segue is used. It checks for the `showDetail` segue identifier and makes sure that the destination is the `RestaurantDetailViewController`. We then get the selected index and fetch the restaurant at that index. With this value, we can set our `selectedRestaurant` and also set the `selectedRestaurant` for the `RestaurantDetailViewController`.
3. Finally, in order to call the `prepare()` method, add the following above the `showRestaurantDetail()` method we just added:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?)
{
    if let identifier = segue.identifier {
        switch identifier {
        case Segue.showDetail.rawValue:
            showRestaurantDetail(segue: segue)
        default:
            print("Segue not added")
        }
    }
}
```

4. This method gets called every time a segue is triggered. We check for `showDetail`; if it is called, we run our `showRestaurantDetail()` method.

This is all we need to do to get our data passed from the **Restaurant** List to the **Restaurant** Detail.

Let's build and run the project by hitting the play button (or use CMD + *R*). When you select a restaurant, you should see all of the restaurant's information on the details page. In addition, you should see a pin dropped on the Map at the restaurant's location:



If you go to the **Map** tab and click on the annotation and then the callout, you will see the **Restaurant Details** page; however, you will not see types of cuisines. We will fix this next.

Map update

We have a couple of minor updates to make in order to have the **Restaurant Details** page show cuisines when you access that page through the **Map** tab. First, we need to change an identifier to ensure our app does not crash (however, we will not actually have our **Map** select from this updated identifier):

1. Open up `Map.storyboard` and select the segue that connects to **Select a location** (the segue that has a square inside the circle) and update the Identifier under **Storyboard Segue** to `locationList`. Then, hit *Enter*.
2. Next, we need our Map to load data from the `RestaurantAPIManager` rather than from the `plist` from which it is currently loading data. In order for us to do this, we must make a few adjustments inside our `RestaurantAnnotation` and our `MapDataManager`:

3. Open the `RestaurantAnnotation.swift` file and remove the following line:

```
if let cuisines = dict["cuisines"] as? [String] { self.cuisines = cuisines }
```

4. Now, let's replace that line with the following:

```
if let cuisines = dict["cuisines"] as? [AnyObject] {
   for data in cuisines {
       if let cuisine = data["cuisine"] as? String {
           self.cuisines.append(cuisine)
       }
   }
}
```

5. Next, open the `MapDataManager.swift` file and, inside the `fetch()` method, replace the line, `for data in loadData(file: "MapLocations")` with the following:

```
for data in RestaurantAPIManager.loadJSON(file: "Chicago")
```

Let's build and run the project by hitting the play button (or use CMD + *R*). You should see your Map now displaying data in Chicago by default. You can always change the default to another location if you want.

Challenge yourself

Go back to the starter files for this chapter and try the following challenges:

1. Add the Title View that we use in `RestaurantDetailViewController` and add it the `RestaurantListViewController`.
2. Update the Map annotation for Restaurant Details to use the custom annotation.
3. Add a **Cancel** and **Done** Button to the **Select a Location** modal. Make each button dismiss the modal.
4. All challenge solutions will be in the starter files for the next chapter.

Summary

We now have JSON data loading into our app. As you can see, going from a plist to a JSON file was not a huge step. Our app is now looking more like an app in the App Store. Over the next few chapters, we will turn our attention to adding features that you might want to use in your app. These features enhance the users' experience, therefore, learning them will be invaluable. Even if the features may not seem like something you want or need, you should become familiar with them because knowing about them and how they work will be beneficial in the long run.

In the next chapter, you will work with the camera and learn how to apply filters and save it to the camera roll.

12
Foodie Reviews

In this chapter, we will focus on creating reviews of restaurants and how to use the camera and camera roll in such reviews. The user will be able to take a picture and apply a filter to that picture. In the next chapter, we will tie it all together by completing the work on the review form and enabling users to save their reviews.

In this chapter, you will learn how to:

- Create a form that users can use to write a review
- Use the camera to take pictures
- Use UIScrollView
- Apply filters to the pictures taken with the camera

Getting started with reviews

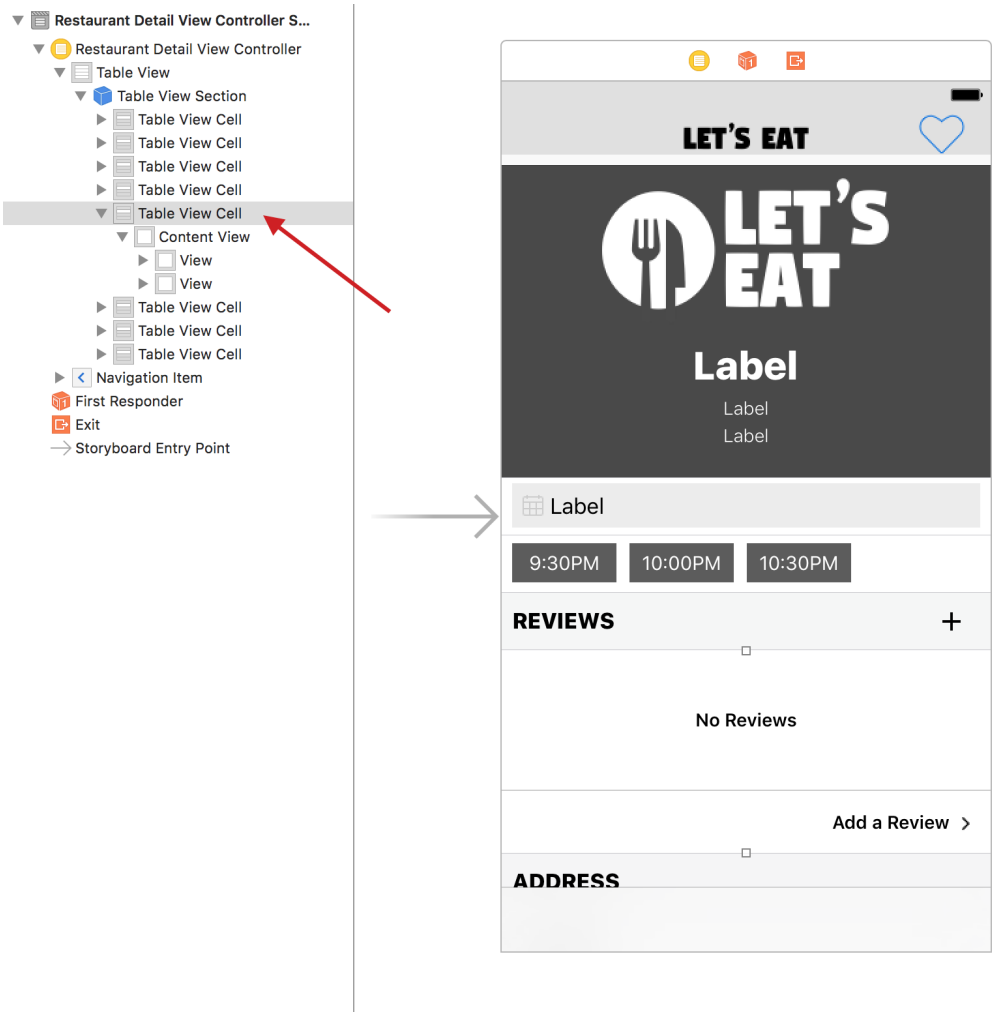
First, we are going to set up the UI for our Reviews section. By doing this, we can then focus strictly on coding. Earlier in this book, we set up the screens for when we have reviews and for when we do not. Now, we need to have two more screens: one for creating reviews and the other for reading reviews.

Setting up our table view controllers

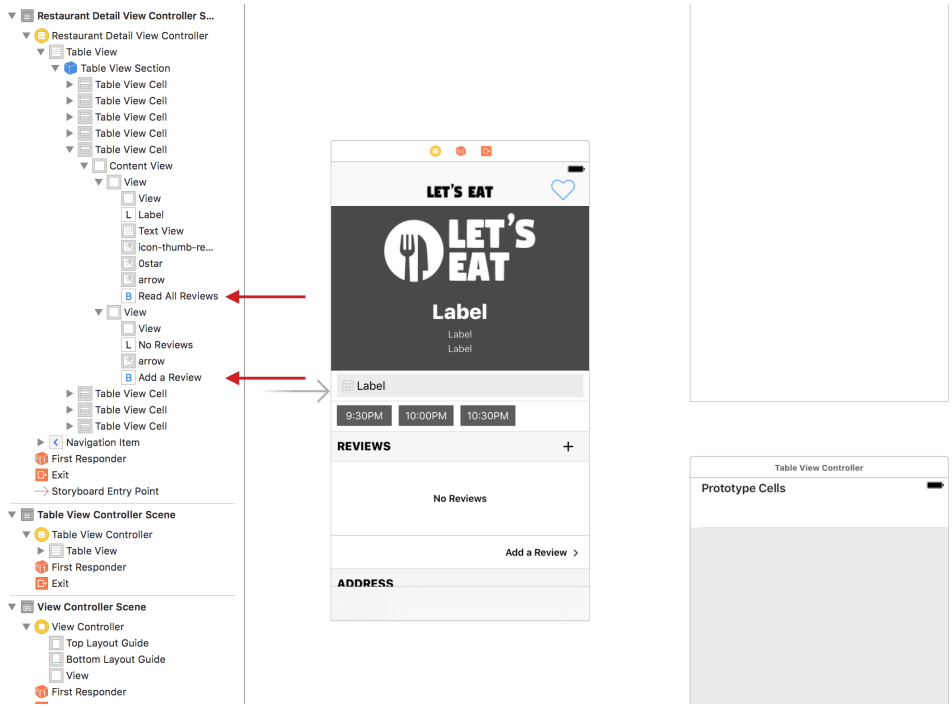
Each of these new screens will be inside of a Table View Controller. Therefore, we need to set up the Table View Controllers:

1. Open the `RestaurantDetail.storyboard` in the **Navigator** panel; and then, in the **Object** library of the **Utilities** panel, type `view` into the filter field.
2. Drag out one Table View Controller and one View Controller into the scene.

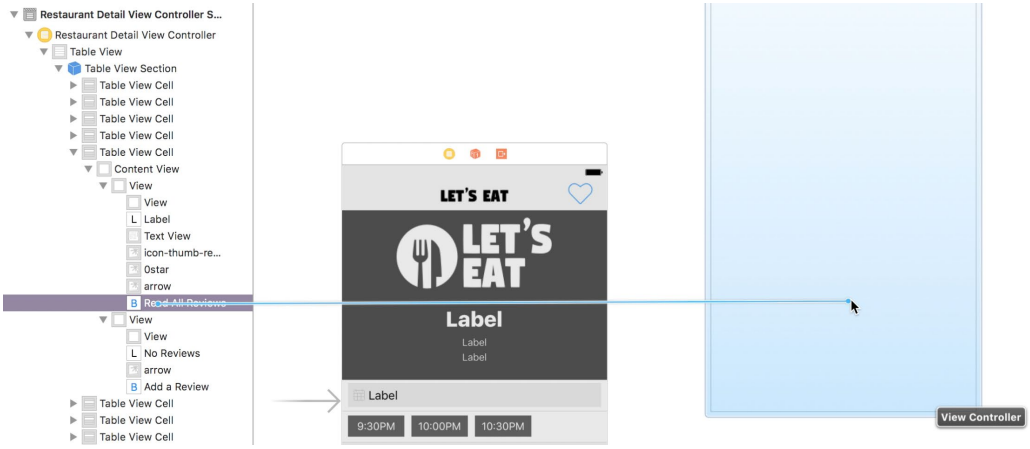
3. In the **Outline** view, find the Table View Cell that contains the **No Reviews** container and the **Reviews Container** we created earlier:



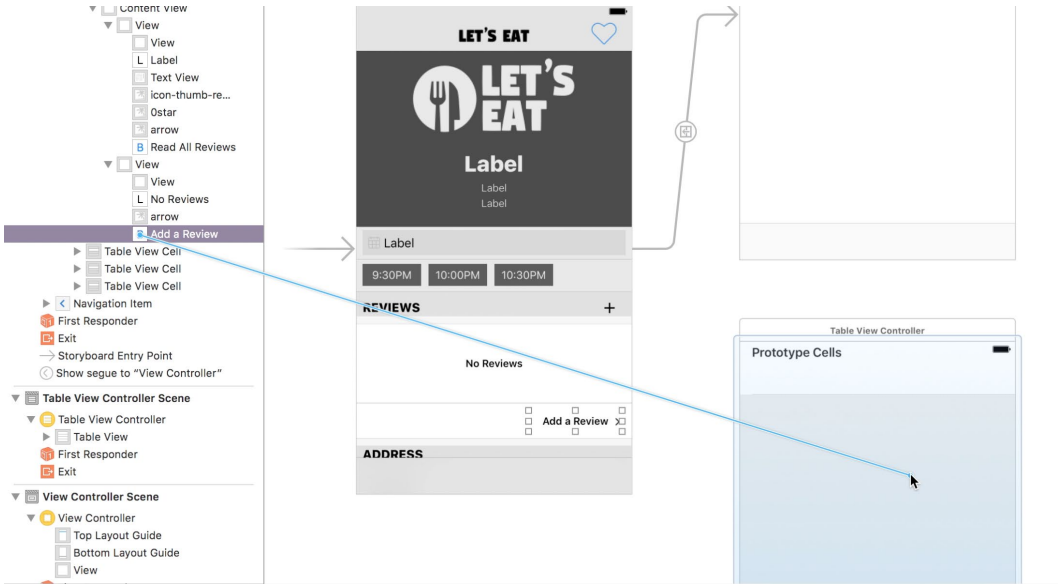
4. Open up the disclosure arrows for both Views. You should see the **Read All Reviews** button in the top **View** with the **Reviews** container and the **Add a Review** in the bottom **View** with the **No Reviews** container:



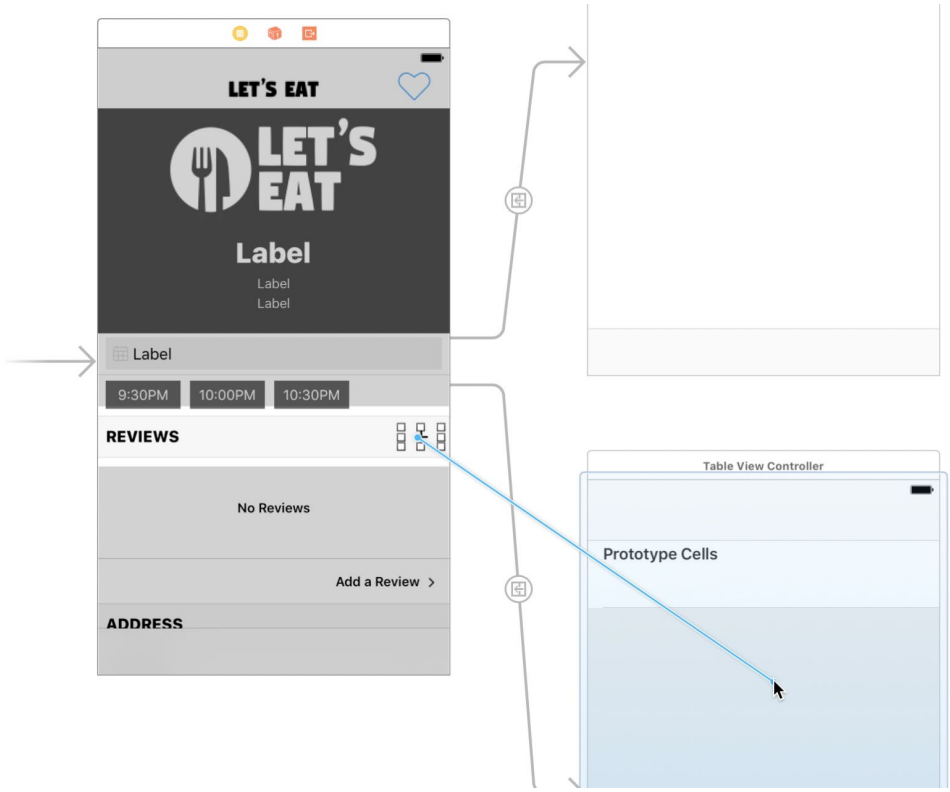
5. CTL drag from **Read All Reviews** in the **Reviews** container to the empty **View Controller**; and under **Action Segue**, select **Show**:



6. CTL drag from **Add a Review** in the **No Reviews** container to the Table View Controller; and under **Action Segue**, select **Show**:



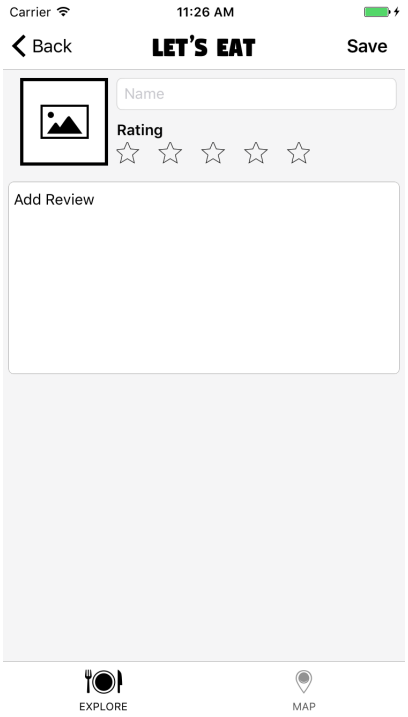
7. Next, in the **Reviews** header, CTL drag from the plus button (either from the **Outline** view or from the scene) to the Table View Controller; and then, under **Action Segue**, select **Show**:



Now that we have the initial View Controllers set up, let's look at what we need to do to create reviews.

Creating reviews

We will have a form that users can use to write reviews. This **Create Review** form takes a name, a rating, a review, and an image, and will look like the following:



To create this form, we will use a static Table View as we did with restaurant details. This makes it easier for us, because the static Table View comes with a lot of built-in functionality that we do not need to code. For example, keyboard dismissal and scrolling the View when a text area is tapped are all built-in to the static Table View.

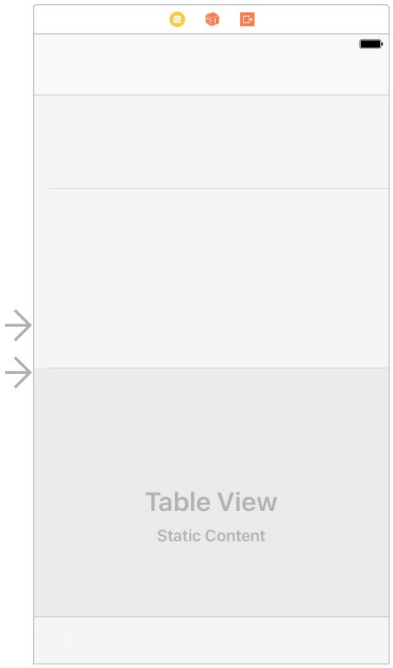
Setting up the Review Storyboard

Let's now set up this static Table View:

1. In the **Outline** view, select the **Table View** that we just added, and then open the **Attributes Inspector** in the **Utilities** panel.
2. Under **Table View**, update **Content** from **Dynamic Prototypes** to **Static Cells**.
3. Then, select **Background** in the **Attributes Inspector**; and under the **Color Sliders** tab, set the **Hex Color #** to `F2F2F4` under **RGB Sliders** in the drop-down menu.

4. Next, select the **Table View Section** in the **Outline** view.
5. Under **Table View Section** in the **Attributes Inspector**, update **Rows** to `2`, and then hit *Enter*.
6. Next, in the **Outline** view, select the first **Table View** Cell under the **Table View Section**; and in the **Attributes Inspector**, change **Background** to **Clear Color** using the arrows to show the drop-down menu.
7. Now, select the **Size Inspector** in the **Utilities** panel; and under **Table View Cell**, update **Row Height** to `100`. Then, hit *Enter*.
8. Lastly, after selecting the second **Table View** Cell under **Table View** Section in the **Outline** view, select **Clear Color** for the **Background** in the **Attributes Inspector** and update the **Row Height** in the **Size Inspector** to `190` (making sure to hit *Enter* after setting the **Row Height**).

You should now have two cells that have a gray background and different heights:



Next, we need to update the inside of each Table View Cell.

Updating the Review Cells

To know what we need in each cell, we need to look at the design, and then match our storyboard to the design. Looking at the design we discussed earlier in this chapter, in our first cell, we need an image, a text field, and two buttons. In our second cell, we need a text view.



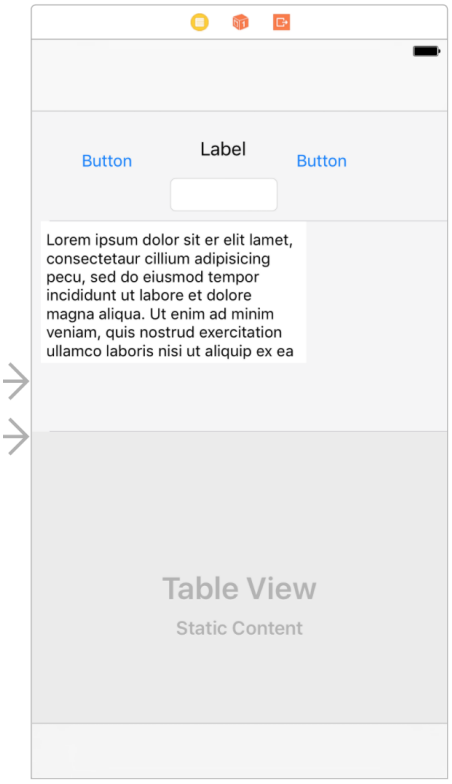
When working with static cells, you may notice that, sometimes, you cannot drag out UI elements directly into the cell in the scene. If you encounter this issue, just drag them into the cell in the Outline view.

1. In the **Object** library of the **Utilities** panel, type `label` in the filter field, and then drag a label into the first cell.
2. Then, type `button` into the filter and drag two buttons into the first cell.
3. Next, type `text` into the filter and drag a Text Field into the first cell.
4. Using the **Outline** view, drag a **Text View** into the second cell.



Do not concern yourself with the positioning of your elements at this time, as we are going to fix their positions shortly.

Your cells should now look like the following:



We now have all the elements we need for this form and just need to position them correctly. You should be getting comfortable with positioning elements in a cell (as well as applying Auto Layout), as we have done this a few times previously in this book.

Positioning UI elements

Let's now place all of our elements into the correct spots:

1. In the **Outline** view, select the **Text View** inside the second cell.
2. Select the **Size Inspector** in the **Utilities** panel and add the following values:
    - **X**: 5
    - **Y**: 5

- **Width**: 365
- **Height**: 179

3. Then, select one of the buttons in the first cell in the **Outline** view, and in the **Size Inspector**, add the following values:
    - **X**: 16
    - **Y**: 8
    - **Width**: 82
    - **Height**: 82
4. Open the **Attributes Inspector** in the **Utilities** panel and delete the `Button` text under **Title**, and then hit *Enter*.
5. In addition, update the **Image** field to say `photo-thumb`, and then hit *Enter*.
6. Next, select the other button in the first cell in the **Outline** view, and in the **Size Inspector**, add the following values:
    - **X**: 106
    - **Y**: 68
    - **Width**: 180
    - **Height**: 20
7. Open the **Attributes Inspector** and again delete the text, `Button`, under **Title**, and then hit *Enter*.
8. Update the **Image** field to `0star`, and then hit *Enter*.
9. Next, in the **Outline** view, select the **Text Field** in the first cell; and then, in the **Size Inspector**, add the following values:
    - **X**: 106
    - **Y**: 8
    - **Width**: 261
10. Open the **Attributes Inspector**; and under **Text Field**, update **Placeholder** to say `Name`. Then, hit *Enter*.
11. Now, select the Label in the first cell in the **Outline** view, and in the **Size Inspector**, add the following values:
    - **X**: 106
    - **Y**: 46
    - **Width**: 261
    - **Height**: 21

12. Lastly, select the **Attributes Inspector**; and then, under **Text**, change `Label` to **Rating** and change **Font** to **Semibold** size `14`.

Your cells should now look as follows:



Your cells' UI elements are now positioned correctly; and we just need to set up Auto Layout.

Adding Auto Layout for creating reviews

Let's now add Auto Layout to our form:

1. Select the **Text View** in the second cell of the **Outline** view, and then the **Pin** icon. Enter the following values:
    - All values under **Add New Constraints** are set to `5`
    - **Constrain to margins**: unchecked

2. Click **Add 4 Constraints**.
3. Next, select the `photo-thumb` Button in the **Outline** view, and then the **Pin** icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 8

      Left: 16
    - **Constrain to margins**: unchecked
    - **Height**: 82 (checked)
    - **Width**: 82 (checked)
4. Click **Add 4 Constraints**.
5. Next, select the Text Field in the first cell in the **Outline** view (called **Name**), and then the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 8

      Left: 8

      Right 8
    - **Constrain to margins**: unchecked
    - **Height**: 30 (checked)
    - Click **Add 4 Constraints**
6. Then, select the **Rating Label** in the first cell in the **Outline** view, and then the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 8

      Left: 8

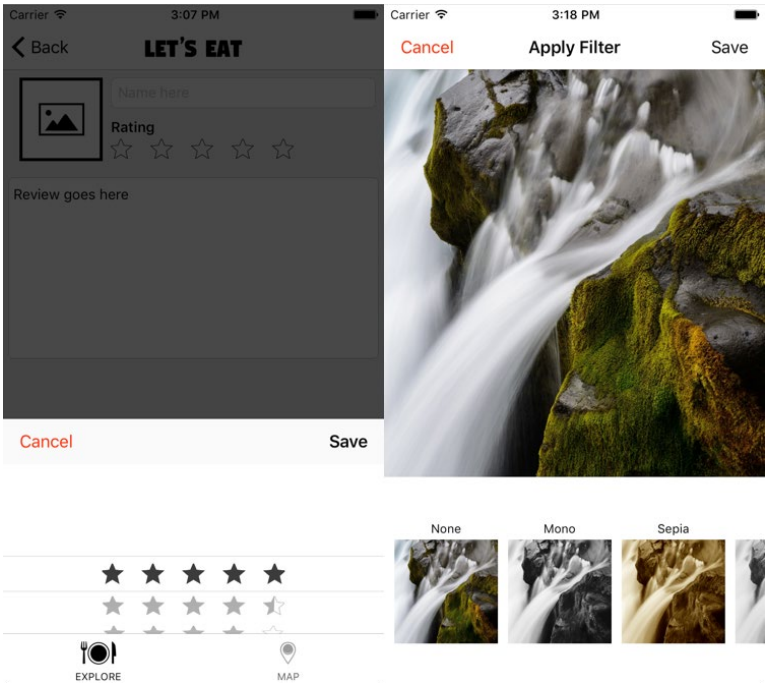
      Right 8
    - **Constrain to margins**: unchecked
    - **Height**: 21 (checked)
7. Click **Add 4 Constraints**.

8. Lastly, select the **0star** Button in the first cell in the **Outline** view, and then the Pin icon. Enter the following values:
    - Under Add New Constraints:

      Top: 0

      Left: 8
    - **Constrain to margins**: unchecked
    - **Width**: 180 (checked)
    - **Height**: 22 (checked)
9. Click **Add 4 Constraints**.

Now that we have set up our Create Review form, we need to add two more scenes. Let's take a look at what these scenes will look like:



Our first scene will be for the star rating. We will create a semi-modal for this. A semi-modal is a modal that will appear over the content, but which allows you to still see the content underneath. Our second scene will be for adding photos.

Now that we have a better idea of what we need to create, let's start with the semi-modal screen.

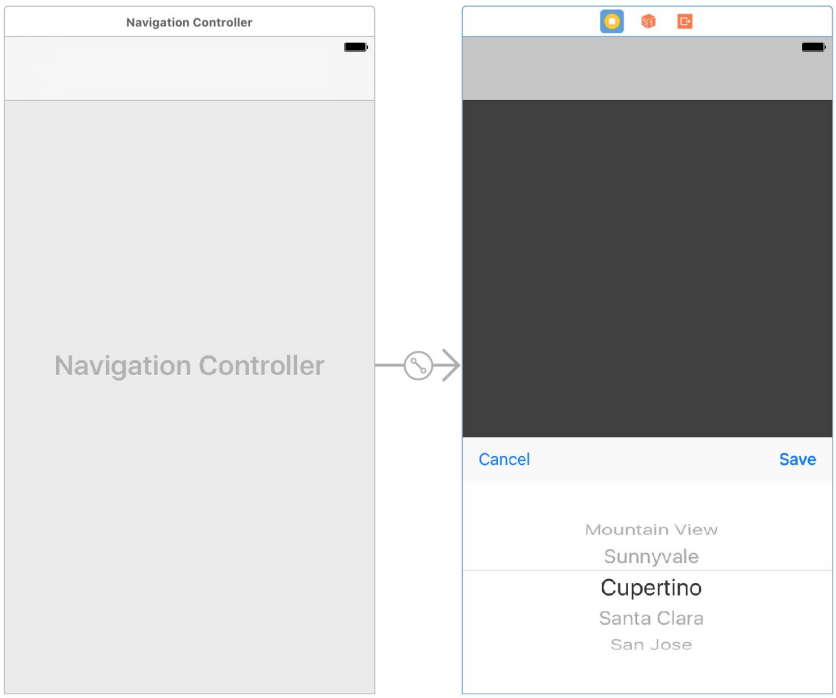
Adding Ratings View

Let's start working with the Ratings View screen (the semi-modal screen) first:

1. Open `RestaurantDetail.storyboard` in the **Navigator** panel, and in the **Object** library of the **Utilities** panel, type `uiview` into the filter field.
2. Drag out a **View Controller** into the scene, and then with this **View Controller** selected, select **Editor** | **Embed In** | **Navigation Controller**.
3. In the **Outline** view, select the **View Controller**; and then, open the **Attributes Inspector** in the **Utilities** panel.
4. Under **Simulated Metrics**, change **Top Bar** from **Inferred** to **None** and **Bottom Bar** from **Translucent Toolbar** to **None**.
5. Then, scroll down to **View Controller** and uncheck **Under Top Bars** in **Extend Edges**.
6. Next, in the **Outline** view, select the **View** in this same **View Controller**; and in the **Attributes Inspector**, update **Background** to **Clear Color**.
7. Then, drag out a **View** into the **View Controller**; and with this new View selected, open the **Size Inspector** in the **Utilities** panel and add the following values:
    - **X**: 0
    - **Y**: 0
    - **Width**: 375
    - **Height**: 667
8. Now, open the **Attributes Inspector** and select **Background**.
9. Under the **Color Sliders** tab, set the **Hex Color #** to `000000` under the **RGB** Sliders in the drop-down menu and the opacity (at the bottom of that tab) to `75%`.
10. Drag out another View into the **Outline** view, confirming that this **View** in the scene is on top (meaning, in front) of the **View** we just created and not inside our black background. This will be our container.
11. Select the container **View**; and, in the **Size Inspector** of the **Utilities** panel, add the following values:
    - **X**: 0
    - **Y**: 407
    - **Width**: 375
    - **Height**: 260

12. Next, in the filter field of the **Object** library, type `tool`.
13. Drag out a **Toolbar** into the **Outline View**, ensuring that this **Toolbar** is inside of our container **View**.
14. Select the **Toolbar**; and in the **Size Inspector**, add the following values:
    - **X**: 0
    - **Y**: 0
    - **Width**: 375
15. Then, in the filter field of the **Object** library, type `picker`.
16. Drag out a **Picker View** into the **Outline View**, making sure that the **Picker View** is inside our container **View**.
17. Select the **Picker View**; and in the **Size Inspector**, add the following values:
    - **X**: 0
    - **Y**: 44
    - **Width**: 375
    - **Height**: 216
18. Inside the **Toolbar**, select the **Toolbar Item**, and then the **Attributes Inspector** in the **Utilities** panel.
19. Under **Bar Button Item**, change **System Item** from **Custom** to **Cancel**.
20. Then, in the **Object** library, type `bar button` into the filter field.
21. Drag a **Bar Button Item** next to **Cancel**.
22. Select this newly added **Bar Button Item**; and then in the **Attributes Inspector**, under **Bar Button Item**, change **System Item** from `Custom` to `Save`.
23. Now, drag a **Flexible Space Bar Button Item** in between **Cancel** and **Save**. This will push **Save** to the right side of the Toolbar.

Your scene should now look like the following:



Adding Auto Layout for Ratings View

Next, we need to add Auto Layout for the Ratings View section:

1. Select the View with the black background in the **Outline** view, and then the Pin icon. Enter the following values:
    - All values under **Add New Constraints** are set to `0`
    - **Constrain to margins**: unchecked
2. Click **Add 4 Constraints**.
3. Next, in the **Outline** view, select the container **View**, and then the **Pin** icon. Enter the following values:
    - Under Add New Constraints:

      Left: 0

      Right: 0

      Bottom: 0

- **Constrain to margins**: unchecked
- **Height**: 260 (checked)

4. Click **Add 4 Constraints**.
5. Now, select the **Toolbar** in the **Outline** view, and then the Pin icon. Enter the following values:
   - Under **Add New Constraints**:

     Top: 0

     Left: 0

     Right: 0
   - **Constrain to margins**: unchecked
   - **Height**: 44 (checked)
6. Click **Add 4 Constraints**.
7. In the **Outline** view, select the **Picker View**, and then the Pin Icon. Enter the following values:
   - All values under **Add New Constraints** are set to `0`
   - **Constrain to margins**: unchecked
8. Now, click **Add 4 Constraints**.

We have completed the design of the **Ratings View** and now we will address our last UI View, which is the Photo Filter View.

Adding our Photo Filter View

The Photo Filter View is where we will show the selected image and a Scroll View from which the user can choose different filters to apply to the image. Let's create the Photo Filter View:

1. While still in the `RestaurantDetail.storyboard`, type `view` into the filter field of the **Object** library in the **Utilities** panel.
2. Drag out a **View Controller** into the scene, and with this **View Controller** selected, select **Editor** | **Embed In** | **Navigation Controller**.
3. Next, with the **View Controller** selected in the **Outline** view, open the **Attributes Inspector** in the **Utilities** panel.
4. Under **View Controller**, uncheck **Under Top Bars** in **Extend Edges**.

5. Then, in the filter field of the Object library, type `image` and drag an **Image View** inside this **View Controller**.
6. Select the **Image View** in the **Outline** view, open the **Size Inspector** in the **Utilities** panel, and then add the following values:
    - **X**: 0
    - **Y**: 0
    - **Width**: 375
    - **Height**: 400
7. Next, in the **Object** library, type `uiview` into the filter field and drag out a **View** into the **View Controller**. This **View** will be our container.
8. Select the container **View** in the **Outline** view, and in the **Size Inspector**, add the following values:
    - **X**: 0
    - **Y**: 400
    - **Width**: 375
    - **Height**: 203
9. Now, type `scroll` into the filter field and drag out a **Scroll View** into the Outline view, ensuring that it is inside the container **View**.
10. With the **Scroll View** selected in the **Outline** view, add the following values in the **Size Inspector**:
    - **X**: 0
    - Y: 39.5
    - **Width**: 375
    - **Height**: 124
11. Next, in the filter field of the **Object** library, type `bar button` and drag a **Bar** Button Item to the left side of the **Navigation** Bar in the **View Controller**.
12. With the Bar Button Item selected in the **Outline** view, open the **Attributes Inspector** in the **Utilities** panel, and under **Bar Button Item**, change **System Item** from **Custom** to **Cancel**.
13. Drag another Bar Button Item to the right side of the **Navigation** Bar in the **View Controller**.
14. With this Bar Button Item selected in the Outline view, in the **Attributes Inspector** under **Bar Button Item**, change **System Item** from **Custom** to **Save**.

Adding Auto Layout for the Photo Filter View

Next, let's apply Auto Layout to the Photo Filter View:

1. In the **Outline** view, select the **Image View**, and then the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 0

      Left: 0

      Right: 0
    - **Constrain to margins**: unchecked
    - **Height**: 400 (checked)
2. Click **Add 4 Constraints**.
3. Next, in the **Outline** view, select the container **View** that is holding our **Scroll View** and enter the following values:
    - Under **Add New Constraints**:

      Left: 0

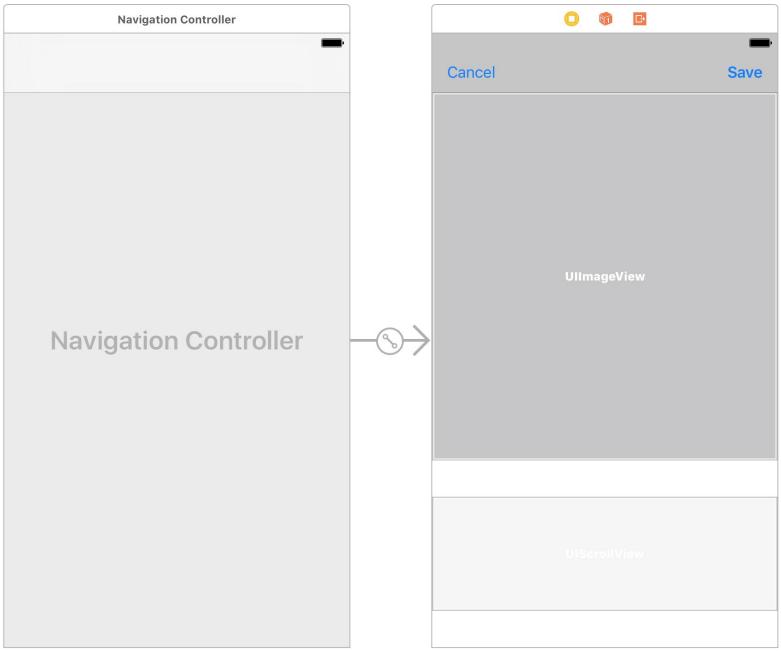
      Bottom: 0

      Right: 0
    - **Constrain to margins**: unchecked
    - **Height**: 203 (checked)
4. Click **Add 4 Constraints**.
5. Then, in the **Outline** view, select the **Scroll View** inside of our container, and then the Pin icon. Enter the following values:
    - Under Add New Constraints:

      Left: 0

      Right: 0
    - **Constrain to margins**: unchecked
    - **Height**: 124 (checked)
6. Click **Add 3 Constraints**.
7. Now, click the **Align** icon, which is located to the left of the Pin icon.
8. Check **Vertically In Container**, and then click **Add 1 Constraint**.

Your scene should now look as follows:



Next, we need to set up these Views so that they will be presented as modals (presented over our content).

Presenting our Views as Modals

Let's get started with presenting our modals:

1. Using the **Outline** view, CTL drag from the **Table View Controller** that contains the Create Review form to the **Navigation Controller** that has the **View Controller** with the **Scroll View** inside of it.
2. In the screen that appears, under **Manual Segue**, select **Present Modally**.
3. Now, select the segue that we just created, and then open the **Attributes Inspector** in the **Utilities** panel.
4. Update **Identifier** under **Storyboard Segue** to say `applyFilter`, and then hit *Enter*.
5. Next, in the **Outline** view, CTL drag from the **Ratings Button** to the **Navigation Controller** that has the **View Controller** with the Picker View inside of it.

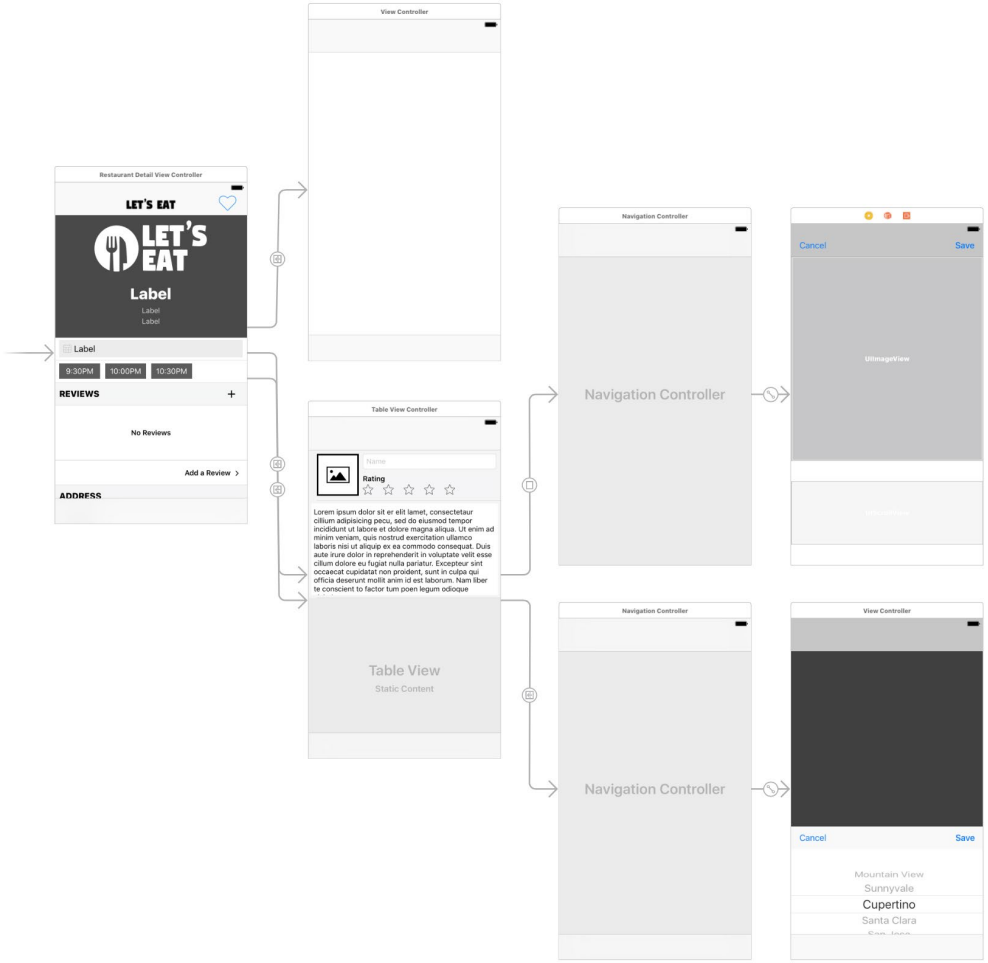
6. In the screen that appears, under **Action Segue**, select **Present Modally**.



The difference between a Manual Segue and an Action Segue is that the Manual Segue gets called by its identifier and the Action Segue gets called through the button's action.

7. Now, select the segue that we just created; and then, in the **Attributes Inspector**, change **Presentation** to **Over Current Context** and **Transition** to **Cross Dissolve**.

Your storyboard should now look like the following:



If you run the project now, when you tap the photo, no modal appears. We still have to write code to present this View. Secondly, if you tap on ratings, you will not be able to dismiss this window. We will work on dismissing the Ratings View next.

Setting up our unwind segues

As we have done before, we need to add code in order for us to unwind (dismiss) a View Controller:

1. Right-click on the `Review` folder in the **Navigator** panel and create three groups, **Controller**, **Model**, and **View**.
2. Then, right click on the `Controller` folder and create two new groups, **Star Rating** and **Apply Filter**.
3. Right click the `Controller` folder again and select **New File**.
4. Inside of the **Choose a template for your new file** screen, select **iOS** at the top, and then **Cocoa Touch Class**. Then, hit **Next**.
5. In the options screen that appears, add the following:
6. New File:
    - **Class**: CreateReviewViewController
    - **Subclass…**: UITableViewController
    - **Also create XIB**: Unchecked
    - **Language**: Swift
7. Click **Next**, and then **Create**.
8. When the file opens, delete everything except the `viewDidLoad()` method.
9. Add the following code after the `viewDidLoad()` method, but before the last curly brace:

```
@IBAction func unwindReviewCancel(segue:UIStoryboardSegue) {}
@IBAction func unwindRatingSave(segue:UIStoryboardSegue) {}
@IBAction func unwindFilterSave(segue:UIStoryboardSegue) {}
```

10. Save the file and return to the `RestaurantDetail.storyboard`.
11. Select the Table View Controller that contains our **Create Review** form, and then open the **Identity Inspector** in the **Utilities** panel.

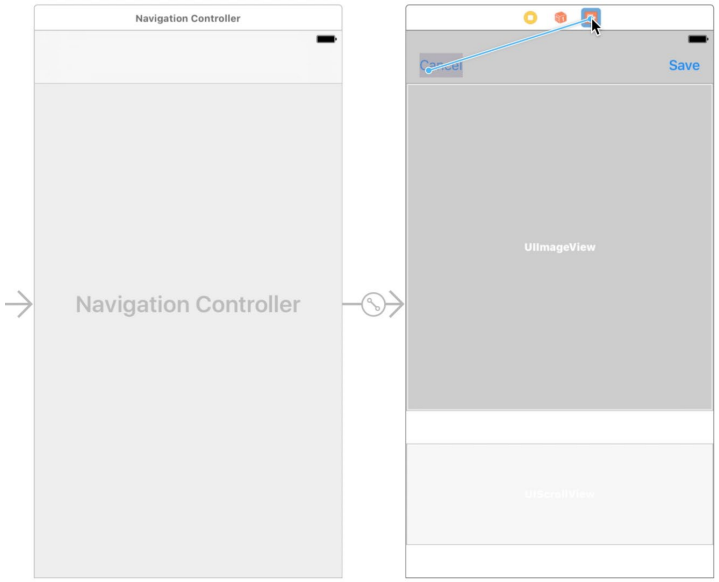
12. Under **Custom Class**, in the **Class** drop-down menu, select **CreateReviewViewController** and hit *Enter* in order to connect the View Controller to the class.

Now, let's hook up our cancel segues, which will allow us to dismiss the Ratings View and Filter View.

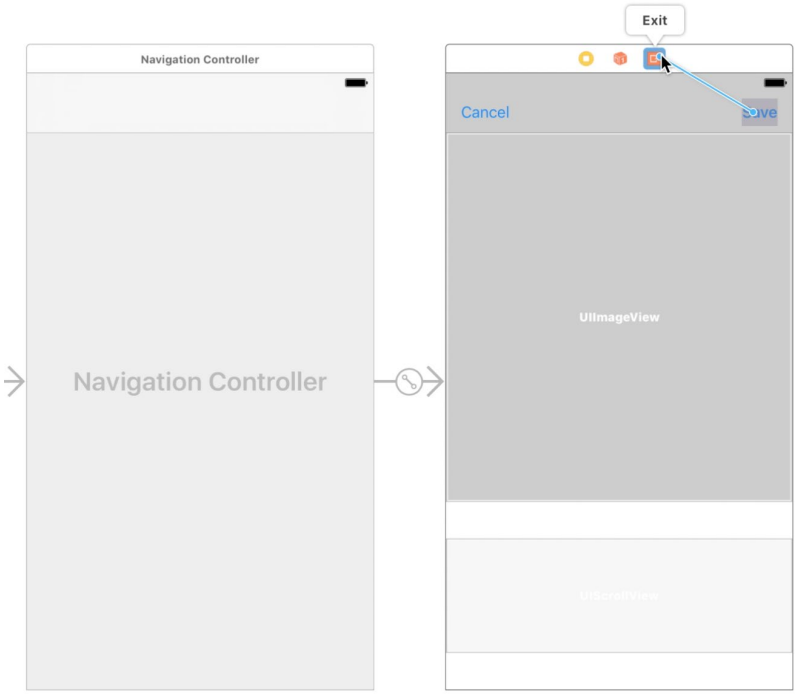
Hooking up our unwind segues

We need to connect the unwind cancel segue for both **Cancel** buttons in order to dismiss our modals. Then, we need to hook the unwind ratings segue to the **Save** Button in the **Ratings View** and the filter unwind to the **Save** button in the **Filter View**. Let's begin:

1. Locate the **View Controller** that contains the **Scroll View**, and CTL drag from the **Cancel** button to the **Exit** icon inside of that **View Controller**:

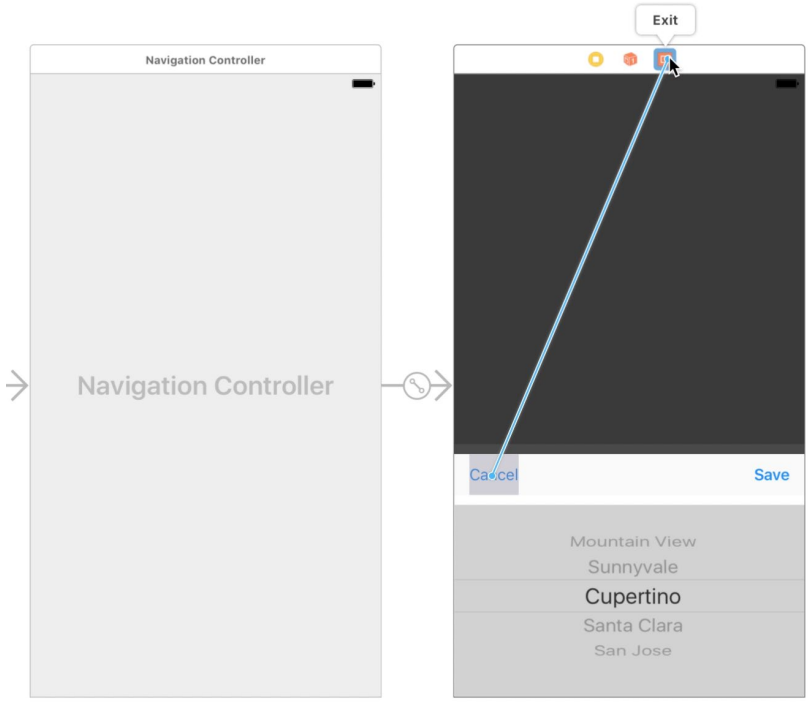


2. In the screen that appears, under **Action Segue**, select **unwindReviewCancelWithSegue:**.
3. Next, CTL drag the **Save** button to the **Exit** icon inside of the same **View Controller**:



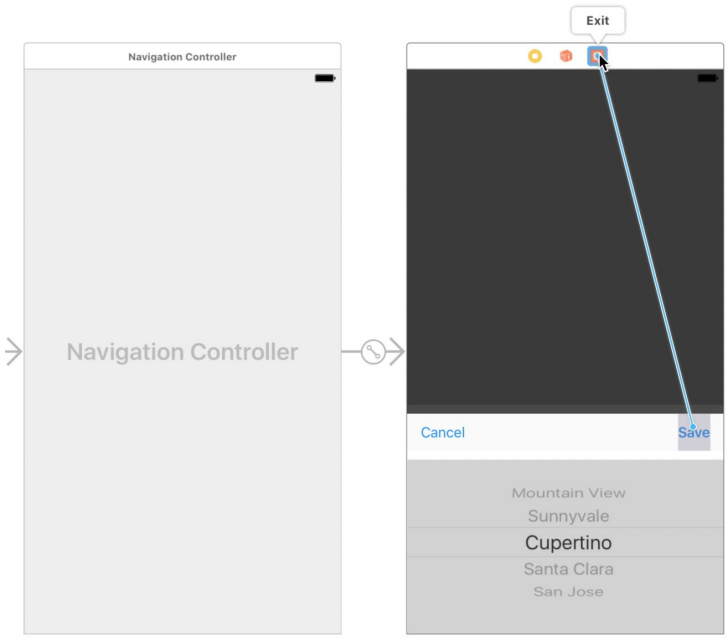
4. In the screen that appears, under **Action Segue**, select **unwindFilterSaveWithSegue**.
5. Now, go to the **View Controller** with the **Picker View**.

6. Then, CTL drag from the **Cancel** button to the **Exit** icon inside of that **View Controller**:



7. In the screen that appears, under **Action Segue**, select **unwindReviewCancelWithSegue:**.

8. Next, CTL drag the **Save** button to the **Exit** icon inside the same View Controller:



9. In the screen that appears, under **Action Segue**, select **unwindRatingSaveWithSegue:**.

If you build and run the project by hitting the play button (or use *CMD* + *R*), you should now be able to dismiss the **Ratings View**. However, you will not be able to test the filter View yet. We will do that once we set up the remainder of that View. Now that we have our setup in place, it is time to get everything coded.

Working with filters

Based on our design, we know that we are going to need to apply filters to a photo. Instead of just creating an array with filters, we are going to use a `plist` to load in a set of filters that we want. You can find the `FilterData.plist` file inside this chapter's asset folder. Drag and drop this file into the Model folder that is inside the `Review` folder. Make sure that **Copy items if needed** is checked, and then hit **Finish**.

Let's take a look at the `plist` and see what it contains:

Key	Type	Value
▼ Root	Array	(10 items)
▼ Item 0	Dictionary	(2 items)
filter	String	None
name	String	None
▼ Item 1	Dictionary	(2 items)
filter	String	CIPhotoEffectMono
name	String	Mono
▼ Item 2	Dictionary	(2 items)
filter	String	CISepiaTone
name	String	Sepia
▼ Item 3	Dictionary	(2 items)
filter	String	CIPhotoEffectTonal
name	String	Tonal
▼ Item 4	Dictionary	(2 items)
filter	String	CIPhotoEffectNoir
name	String	Noir
▼ Item 5	Dictionary	(2 items)
filter	String	CIPhotoEffectFade
name	String	Fade
▼ Item 6	Dictionary	(2 items)
filter	String	CIPhotoEffectChrome
name	String	Chrome
▼ Item 7	Dictionary	(2 items)
filter	String	CIPhotoEffectProcess
name	String	Process
▼ Item 8	Dictionary	(2 items)
filter	String	CIPhotoEffectTransfer
name	String	Transfer
▼ Item 9	Dictionary	(2 items)
filter	String	CIPhotoEffectInstant
name	String	Instant

This list only has 10 of over 170 filters and effects that you can use. If you would like to see a full list of filters, you can find the list at `http://tinyurl.com/coreimage-ios`. Feel free to add, remove, or update any filters.

Now that we have seen what our `plist` looks like, we need to create a Model that represents this data. We also need to create a **Manager** class to manage our items. Let's create the **Model** first:

1. Right-click the `Model` folder in the `Review` folder and select **New File**.
2. Inside the **Choose a template for your new file** screen, select **iOS** at the top, and then **Swift File**. Then, hit **Next**.

3. Name this file, **FilterItem**, and hit **Create**.
4. Next, we need to define our `struct`; therefore, add the following under the import statement:

```
struct FilterItem {
    var filter:String?
    var name:String?
}

extension FilterItem {
    init(dict:[String:AnyObject]) {
        name  = dict[name] as? String
        filter = dict[filter] as? String
    }
}
```

The filter property will be the class passed to apply the filter; and the name property will be used as a display.

Let's create our `FilterManager` file next:

1. Right-click the `Model` folder in the `Review` folder and select `New File`.
2. Inside of the **Choose a template for your new file** screen, select `iOS` at the top, and then **Swift File**. Then, hit **Next**.
3. Name this file **FilterManager,** and hit **Create**.
4. Next, we need to define our class definition; therefore, add the following under the `import` statement:

```
class FilterManager: DataManager {

  private var items:[FilterItem] = []

  func fetch() {
      for data in load(file: FilterData) {
          items.append(FilterItem(dict: data))
      }
  }

  func numberOfItems() -> Int {
      return items.count
  }

  func filterItemAtIndexPath(index:IndexPath) -> FilterItem {
```

```
            return items[index.item]
        }
    }
```

This file is using our DataManager base class, which is converting our plist data into an array of dictionary objects. Once that is complete, we are creating FilterItems from that.

Next, we need to create a file that will take a `FilterItem` and apply a filter to an image. Since we are going to do this in numerous places, it is best to have all of this code in one place. Therefore, we are going to create a file that will handle all of this processing for us. Let's create our `ImageFiltering` file:

1. Right-click the `Model` folder in the `Review` folder and select **New File**.
2. Inside the **Choose a template for your new file** screen, select **iOS** at the top, and then **Swift File**. Then, hit **Next**.
3. Name this file, **ImageFiltering**, and hit **Create**.
4. Update your file to the following:

```
import UIKit
import CoreImage

protocol ImageFiltering {
    func apply(filter:String, originalImage:UIImage) -> UIImage
}

protocol ImageFilteringDelegate:class {
    func filterSelected(item:FilterItem)
}

extension ImageFiltering {
    func apply(filter:String, originalImage:UIImage) -> UIImage {
        let initialCIImage = CIImage(image: originalImage, options: nil)
        let originalOrientation = originalImage.imageOrientation


        guard let ciFilter = CIFilter(name: filter) else {
            print("filter not found")
            return UIImage()
        }

        ciFilter.setValue(initialCIImage, forKey: kCIInputImageKey)

        let context = CIContext()
        let filteredCIImage = (ciFilter.outputImage)!
        let filteredCGImage = context.createCGImage(filteredCIImage, from: filteredCIImage.extent)

        return UIImage(cgImage: filteredCGImage!, scale: 1.0, orientation: originalOrientation)
    }
}
```

Let's break down each section so that we can understand what we are doing with this code:

- **Part A**:

```
import UIKit
import CoreImage
```

`CoreImage` will give us access to the image processing we need for filtering.

- **Part B:**

```
protocol ImageFiltering {
  func apply(filter:String, originalImage:UIImage) -> UIImage
}
```

Creating this protocol allows us to have other classes conform to it, therefore giving us access to the method and allowing us to use it wherever we want.

- **Part C**

```
protocol ImageFilteringDelegate:class {
    func filterSelected(item:FilterItem)
}
```

This protocol will be used when a filter is selected and when we need the selected filter to be passed from one View or View Controller to another.

- **Part D**

```
extension ImageFiltering {
  func apply(filter:String, originalImage:UIImage) -> UIImage {
      let initialCIImage = CIImage(image: originalImage, options:
nil)
      let originalOrientation = originalImage.imageOrientation
guard let ciFilter = CIFilter(name: filter) else {
          print(filter not found)
          return UIImage()
      }

      ciFilter.setValue(initialCIImage, forKey: kCIInputImageKey)

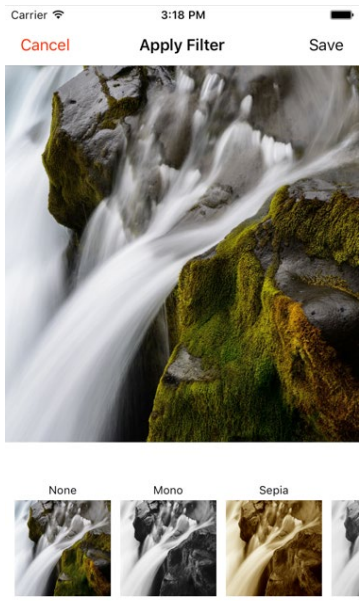
      let context = CIContext()
      let filteredCIImage = (ciFilter.outputImage)!
      let filteredCGImage = context.createCGImage(filteredCIImage,
from: filteredCIImage.extent)

      return UIImage(cgImage: filteredCGImage!, scale: 1.0,
orientation: originalOrientation)
    }
}
```

Here, we are creating an extension and adding all of the code that we are going to use for applying filters to images. This code is called **Protocol Oriented Programming** (**POP**). Although we will not address POP much in this book, if you are interested in learning more, there is a great book from Packt Publishing that goes into much more detail.

Creating our Filter Scroller

After a user selects a photo to use, we will present the user with a screen, which contains that image. Below that image, we will have a scroller, better known as a `UIScrollView`, which allows us to create content that scrolls either horizontally or vertically. The `UIScrollView` will display an image (thumbnail) with the filter applied to it as well as the name of the filter. This image and name will represent our filters visually to our users. When the user taps on the image, the user will see the selected filter change the primary image. Let's look at an example:



We are now going to create the elements inside the `UIScrollView`. Since we have created a lot inside Storyboard, let's create the `PhotoItem` all in code:

1. Right-click the `Model` folder in the `Review` folder and select **New File**.
2. Inside the **Choose a template for your new file** screen, select **iOS** at the top, and then **Swift File**. Then, hit **Next**.

3. Name this file, **PhotoItem**, and hit **Create**.
4. Update your file to the following:

```
import UIKit

class PhotoItem: UIView, ImageFiltering {

    var imgThumb:UIImageView?
    var lblTitle:UILabel?
    var data:FilterItem?

    weak var delegate: ImageFilteringDelegate?

    required init?(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    init(frame:CGRect, image:UIImage, item:FilterItem) {

        super.init(frame: frame)

        setDefaults(item: item)
        createThumbnail(image: image, item: item)
        createLabel(item: item)
    }

    func setDefaults(item:FilterItem) {
        data = item
        let tap = UITapGestureRecognizer(target: self, action:#selector(thumbTapped))
        self.addGestureRecognizer(tap)
        self.backgroundColor = .clear
    }

    func createThumbnail(image:UIImage, item:FilterItem) {

        guard let filterName = item.filter else {
            return
        }

        if filterName != "None" {
            let filteredImg = apply(filter: filterName, originalImage: image)
            imgThumb = UIImageView(image: filteredImg)
        else { imgThumb = UIImageView(image: image) }

        guard let thumb = imgThumb else {
            return
        }

        thumb.contentMode = .scaleAspectFill
        thumb.frame = CGRect(x: 0, y: 22, width: 102, height: 102)
        thumb.clipsToBounds = true

        addSubview(thumb)
    }

    func createLabel(item:FilterItem) {
        guard let displayName = item.name else {
            return
        }

        lblTitle = UILabel(frame: CGRect(x: 0, y: 0, width: 102, height: 22))

        guard let label = lblTitle else {
            return
        }

        label.text = displayName
        label.font = UIFont.systemFont(ofSize: 12.0)
        label.textAlignment = .center
        label.backgroundColor = .clear


        addSubview(label)
    }

    func thumbTapped() {
        if let data = self.data {
            filterSelected(item: data)
        }
    }

    func filterSelected(item:FilterItem) {
        delegate?.filterSelected(item: item)
    }
}
```

Let's break down each section of this code:

- **Part A:**

  ```
  class PhotoItem: UIView, ImageFiltering {
  ```

  Here, we are telling our `PhotoItem` that we want to conform to the `ImageFiltering` protocol, which we created earlier.

- **Part B:**

  ```
  weak var delegate: ImageFilteringDelegate?
  ```

  Here, we are creating a delegate, which will be used to let any class know when something happens. We will use this delegate when someone taps on the object itself. This will allow us to pass the `FilterItem` data to a parent class.



You have used this pattern already plenty of times. Table Views and Collection Views both have delegates to which you conform.

- **Part C:**

  ```
  required init?(coder aDecoder: NSCoder) {
      fatalError(init(coder:) has not been implemented)
  }
  ```

  Whenever you create a `UIView`, you are required to add this method. If you do not, it will give you an error, and then you will have to add it.

- **Part D:**

  ```
  init(frame:CGRect, image:UIImage, item:FilterItem) {
      super.init(frame: frame)

  setDefaults(item: item)
  createThumbnail(image: image, item: item)
      createLabel(item: item)
  }
  ```

  This is a custom `init()` method, which allows us to pass data (here, the frame, image, and filter items) when the item gets created.

- **Part E:**

  ```
  func setDefaults(item:FilterItem) {
      data = item
      let tap = UITapGestureRecognizer(target: self,
  action:#selector(thumbTapped))
      self.addGestureRecognizer(tap)
      self.backgroundColor = .clear
  }
  ```

Our `setDefaults()` method is used to create a tap gesture. When the item gets tapped, we will call the `thumbTapped` method. We also set the data and the background color inside this method.



Typically, I like to have a `setDefaults()` method whenever I need to set some basic values.

- **Part F:**

```
func createThumbnail(image:UIImage, item:FilterItem) {

    guard let filterName = item.filter else {
        return
    }

    if filterName != None {
        let filteredImg = apply(filter: filterName, originalImage:
image)
        imgThumb = UIImageView(image: filteredImg)
    }
    else { imgThumb = UIImageView(image: image) }

    guard let thumb = imgThumb else {
        return
    }

    thumb.contentMode = .scaleAspectFill
    thumb.frame = CGRect(x: 0, y: 22, width: 102, height: 102)
    thumb.clipsToBounds = true

    addSubview(thumb)
}
```

Here, we are creating an image and applying a filter. Then, we are setting its frame and adding the image to the View.

- **Part G:**

```
func createLabel(item:FilterItem) {
    guard let displayName = item.name else {
        return
    }

    lblTitle = UILabel(frame: CGRect(x: 0, y: 0, width: 102,
height: 22))

    guard let label = lblTitle else {
```

```
            return
        }

        label.text = displayName
        label.font = UIFont.systemFont(ofSize: 12.0)
        label.textAlignment = .center
        label.backgroundColor = .clear

        addSubview(label)
    }
```

Here, we are creating a label and passing in the name of the filter. Then, we are setting its frame and adding the label to the View.

- **Part H:**

```
func thumbTapped() {
    if let data = self.data {
        filterSelected(item: data)
    }
}
```

This method is used to detect taps. When the item is tapped, it will call `filterSelected`.

- **Part I:**

```
func filterSelected(item:FilterItem) {
    delegate?.filterSelected(item: item)
}
```

This method is from the protocol we created earlier; and all we are doing is calling the delegate method, `filterSelected`. We will see what happens next when the selected filter gets called.

Creating our apply Filter View Controller

We next need to create a class called, `ApplyFilterViewController`:

1. Right-click the **Apply Filter** in the `Controller` folder in the `Review` folder and select `New File`.
2. Inside the **Choose a template for your new file** screen, select **iOS** at the top, and then **Cocoa Touch Class**. Then, hit **Next**.

3. In the options screen that appears, add the following:
4. New File:
    - **Class**: ApplyFilterViewController
    - **Subclass…**: UIViewController
    - **Also create XIB**: Unchecked
    - **Language**: Swift
5. Click **Next**, and then **Create**.

When the file opens, delete everything after the `viewDidLoad()` method. Then, add the following extension declaration after the last curly brace:

```
extension ApplyFilterViewController: ImageFiltering,
ImageFilteringDelegate {
}
```



You will get an error after you create this extension. Ignore the error for now as we will fix it shortly.

In this extension, we will use the protocol we created called, `ImageFiltering`. We will also use the `ImageFilteringDelegate` here.

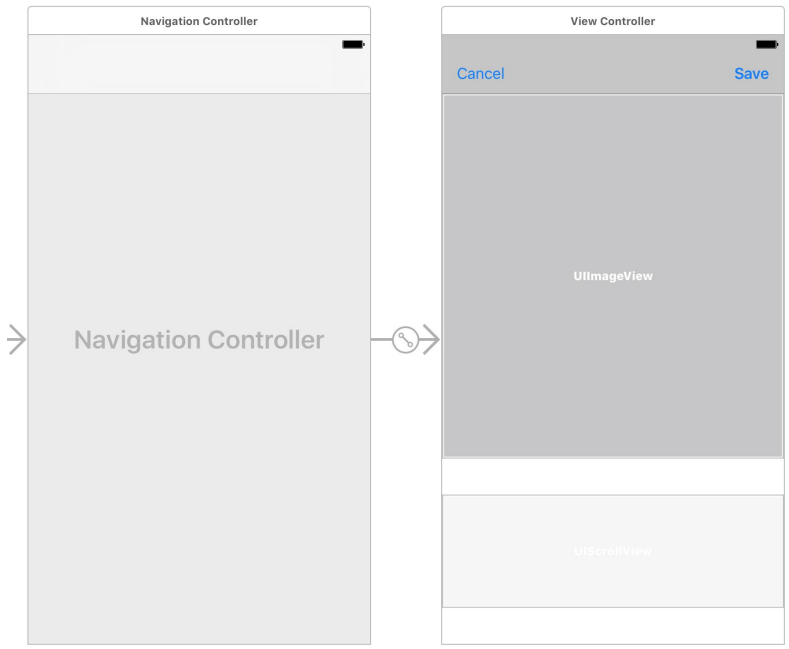
Back inside the class declaration, add the following properties:

```
var image: UIImage?
var thumbnail: UIImage?
let manager = FilterManager()

@IBOutlet var scrollView: UIScrollView!
@IBOutlet weak var imgExample: UIImageView!
```

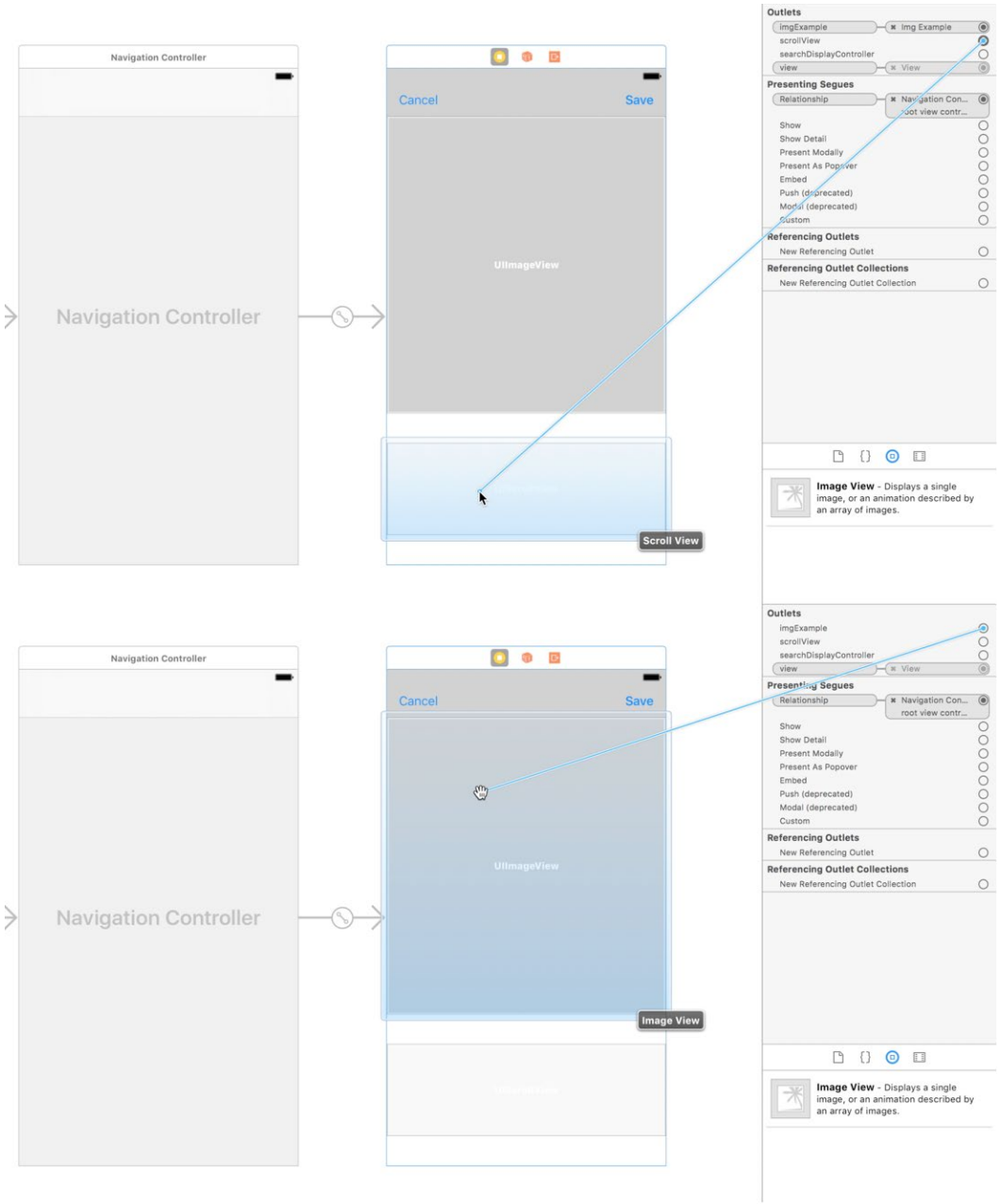
Save the file, and now let's hook up our two `IBOutlets`:

1. Open the `RestaurantDetail.storyboard` and go to the scene that contains our Scroll View that we added earlier when we set up the UI:



2. Select the **View Controller** in the **Outline** view, and then the **Identity Inspector** in the **Utilities** panel.
3. Under **Custom Class**, in the **Class** drop-down menu, select **ApplyFilterViewController** and hit *Enter*.
4. Then, select the **Connections Inspector** in the **Utilities** panel.

5. Under **Outlets**, click and drag from the empty circle of each of the components, **imgExample** and **scrollView**, to the **Image View** and **Scroll View**, respectively, in the scene:



Now, please open the `ApplyFilterViewController.swift` file again, and let's add some more code. Update your `viewDidLoad()` method by adding `initialize()` so that it now looks as follows:

```
override func viewDidLoad() {
    super.viewDidLoad()

    initialize()
}
```

You will get an error that says `Use of unresolved identifier 'initialize'`, which we will resolve by adding the following after the updated `viewDidLoad()` method:

```
func initialize() {
    manager.fetch()

    if let image = self.image, let thumb = self.thumbnail {
        createScrollContent(img: thumb)
        imgExample.image = image
    }
}
```

Here, we are calling the manager that we created earlier to get us the filter data. Next, we are grabbing the thumbnail and image properties and passing the thumbnail to our `createScrollContent()` method and the image to our primary image. These two images will contain the images that are selected when the user takes a picture or chooses an image from the photo library.

You will get another error that says `Use of unresolved identifier 'createScrollContent'`, which we will resolve by adding the following code in the screenshot below the `initialize()` method:

```
func createScrollContent(img:UIImage) {
    DispatchQueue.main.async {                                          A
        let size = CGFloat(102)                                         B
        var currentViewOffset = CGFloat(10)

        for index in 0..<self.manager.numberOfItems() {
            let item = self.manager.filterItemAtIndexPath(index: IndexPath(item: index, section: 0))
            let frame = CGRect(x: currentViewOffset, y: 0, width: size, height: 124)
            let subview = PhotoItem(frame: frame, image: img, item: item)   C
            subview.delegate = self

            self.scrollView.addSubview(subview)
            currentViewOffset += (size + 10)
        }

        self.scrollView.showsHorizontalScrollIndicator = false          D
        self.scrollView.contentSize = CGSize(width: CGFloat(self.manager.numberOfItems()) * 113, height: size)
    }
}
```

Now, let's break down this code:

- **Part A:**

  ```
  DispatchQueue.main.async {
  ```

  This line allows us to run the code on the main thread. This entire method is doing a lot of processing. Without this, due to the large size of many images, the screen would lock, making the entire UI unresponsive. By adding this code, we let the processing continue. There will still be a slight delay before the scroller is created.

- **Part B:**

  ```
  let size = CGFloat(102)
  var currentViewOffset = CGFloat(10)
  ```

  Here, we are setting the size of the items in the scroller as well as the space in between each item.

- **Part C:**

  ```
  for index in 0..<self.manager.numberOfItems() {
      let item = self.manager.filterItemAtIndexPath(index:
  IndexPath(item: index, section: 0))
      let frame = CGRect(x: currentViewOffset, y: 0, width: size,
  height: 124)
      let subview = PhotoItem(frame: frame, image: img, item: item)
      subview.delegate = self

      self.scrollView.addSubview(subview)
      currentViewOffset += (size + 10)
  }
  ```

  This for-in loop is used to add a new `PhotoItem` into the scroller every time the loop runs. It also passes the image to which we are applying the filter and the name of the filter it receives to the `PhotoItem`. In addition, `subview.delegate = self` is where we are telling our class that we want to know every time the `filterSelected()` method is called.

- **Part D:**

  ```
  self.scrollView.showsHorizontalScrollIndicator = false
  self.scrollView.contentSize = CGSize(width: CGFloat(self.manager.
  numberOfItems()) * 113, height: size)
  }
  ```

  Finally, we hide the scroll indicator and then calculate the size of the scroller based on the number of items we created.

We have one more method to create. We need to be able to change our main image every time a new filter is tapped. Add the following method inside of the `ApplyFilterViewController` extension:

```
func filterSelected(item: FilterItem) {
    let filteredImg = image

    if let filterName = item.filter, let img = filteredImg {
        if filterName != None {
            imgExample.image = self.apply(filter: filterName,
originalImage: img)
        }
        else {
            imgExample.image = img
        }
    }
}
```

The error you were seeing when we first created the extension in this file will no longer appear now that we've added this method in our extension. Since we used `subview.delegate = self` when we created the scroller, we have to implement this delegate method. Now, with this method implemented, we will get the filter that is selected from the scroller and apply it to the primary image for the user.

Creating review images

Now, when a user clicks **Save**, we need to pass this image back over to our `CreateReviewViewController`. Therefore, open up your `CreateReviewViewController` and add the following extension after the last curly brace:

```
extension CreateReviewViewController: UIImagePickerControllerDelegate,
UINavigationControllerDelegate {

}
```

At the top of the file, add the following import statements under `import UIKit`:

```
import AVFoundation
import MobileCoreServices
```

`AVFoundation` is a framework that gives us access to the camera; and `MobileCoreServices` gives us access to the filters.

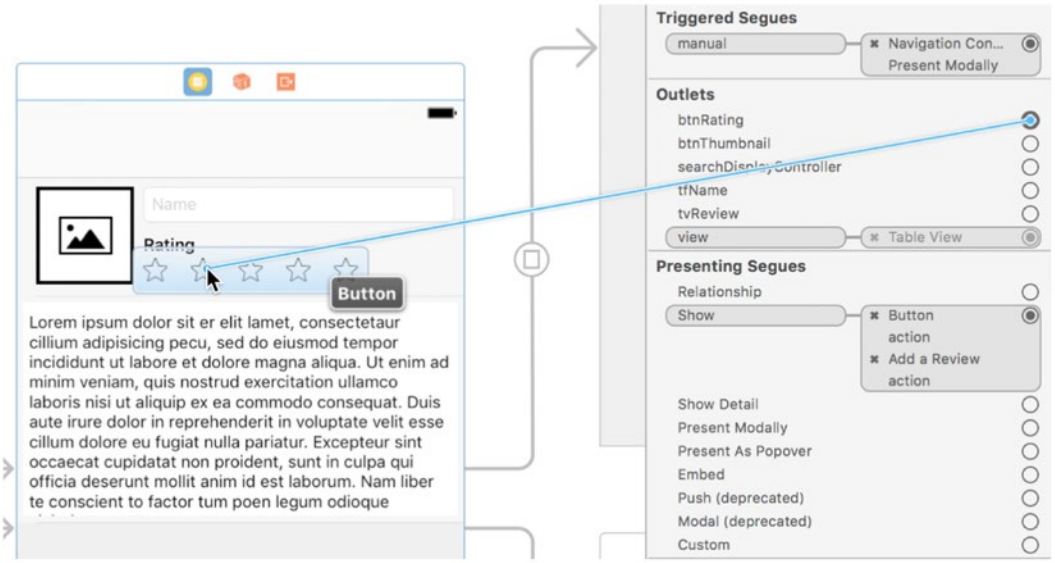
Next, we need to add a few variables to our **Create Review** form. Add the following above the `viewDidLoad()` method:

```
@IBOutlet var tvReview: UITextView!
@IBOutlet var tfName: UITextField!
@IBOutlet var btnThumbnail: UIButton!
@IBOutlet var btnRating: UIButton!

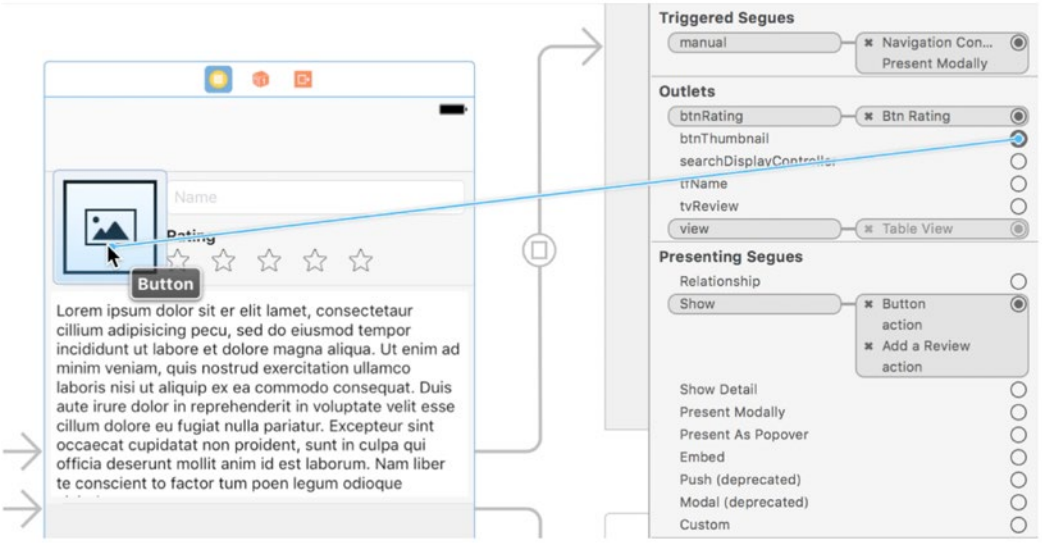
var image: UIImage?
var thumbnail: UIImage?
```

Save the file. Now that we have our variables, let's hook them up to our UI elements:

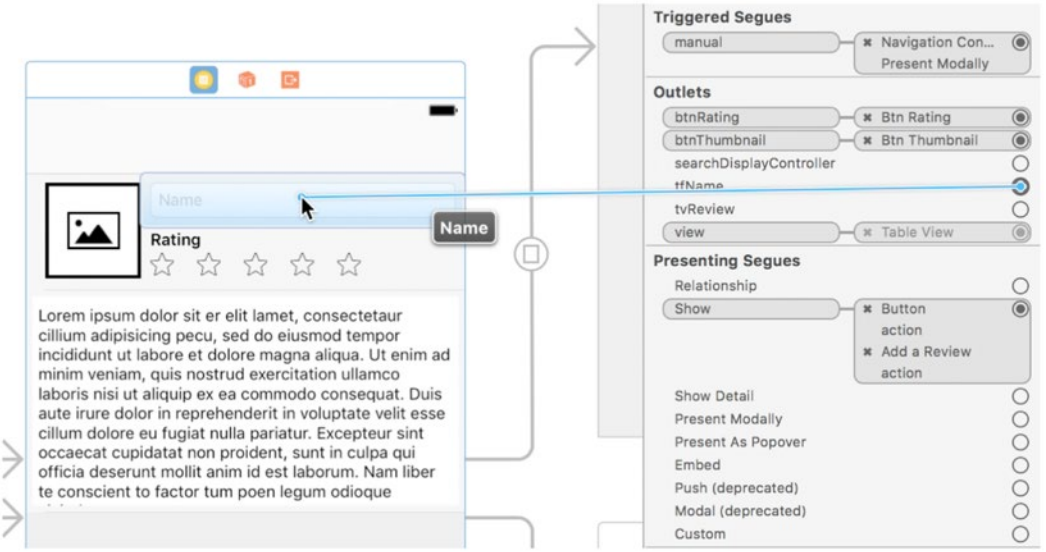
1. Open the `RestaurantDetail.storyboard` and select the **Create Review View** Controller.
2. In the **Connections Inspector** in the **Utilities** panel, click and drag from the empty circle next to **btnRating** to the stars button in the scene:



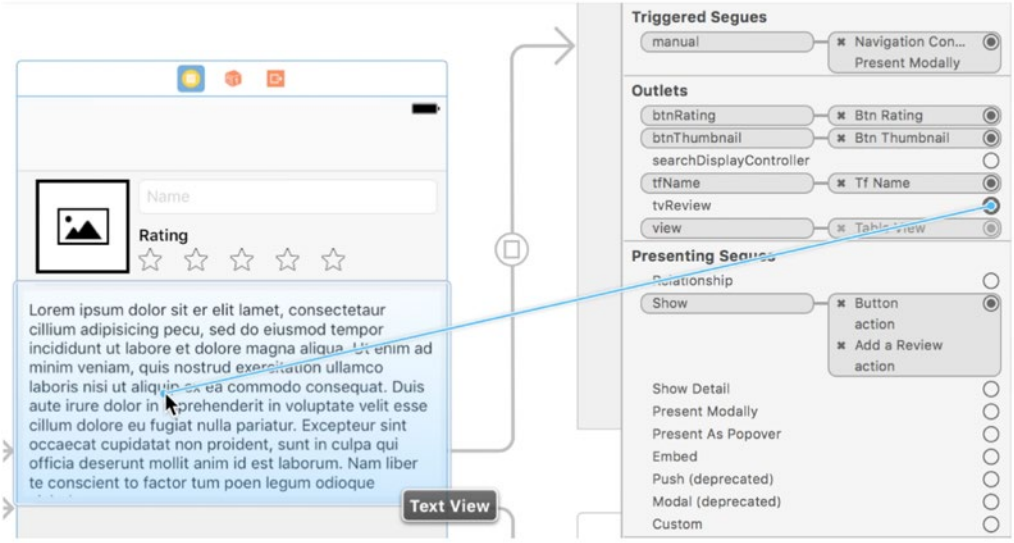
3. Next, click and drag from the empty circle next to **btnThumbnail** to the **photo-thumb** button in the scene:



4. Now, click and drag from the empty circle next to **tfName** to the Text Field in the scene:



5. Finally, click and drag from the empty circle next to **tvReview** to the Text View in the scene:



Open the `CreateReviewViewController.swift` file, and let's add the following inside our extension:

```
extension CreateReviewViewController: UIImagePickerControllerDelegate, UINavigationControllerDelegate {
A   func showCameraUserInterface() {
        let imagePicker = UIImagePickerController()
        imagePicker.delegate = self

        #if (arch(i386) || arch(x86_64)) && os(iOS)
            imagePicker.sourceType = UIImagePickerControllerSourceType.photoLibrary
        #else
            imagePicker.sourceType = UIImagePickerControllerSourceType.camera
            imagePicker.showsCameraControls = true

        #endif

        imagePicker.mediaTypes = [kUTTypeImage as String]
        imagePicker.allowsEditing = true
        self.present(imagePicker, animated: true, completion: nil)
    }

B   func generate(image:UIImage, ratio:CGFloat) -> UIImage {
        let size = image.size
        var croppedSize:CGSize?
        var offsetX:CGFloat = 0.0
        var offsetY:CGFloat = 0.0

        if size.width > size.height {
            offsetX = (size.height - size.width) / 2
            croppedSize = CGSize(width: size.height, height: size.height)
        }
        else {
            offsetY = (size.width - size.height) / 2
            croppedSize = CGSize(width: size.width, height: size.width)
        }
        guard let cropped = croppedSize, let cgImage = image.cgImage else {
            return UIImage()
        }

        let clippedRect = CGRect(x: offsetX * -1, y: offsetY * -1, width: cropped.width, height: cropped.height)
        let imgRef = cgImage.cropping(to: clippedRect)

        let rect = CGRect(x: 0.0, y: 0.0, width: ratio, height: ratio)
        UIGraphicsBeginImageContext(rect.size)

        if let ref = imgRef {
            UIImage(cgImage: ref).draw(in: rect)
        }

        let thumbnail = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()

        guard let thumb = thumbnail else { return UIImage() }

        return thumb
    }

C   func imagePickerControllerDidCancel(_ picker: UIImagePickerController) {
        picker.dismiss(animated: true, completion: nil)
    }

D   func imagePickerController(_ picker: UIImagePickerController, didFinishPickingMediaWithInfo info: [String : Any]) {
        let image = info[UIImagePickerControllerEditedImage] as? UIImage

        if let img = image {
            self.btnThumbnail.imageView?.image = generate(image: img, ratio: CGFloat(102))
            self.thumbnail = generate(image: img, ratio: CGFloat(102))
            self.image = generate(image: img, ratio: CGFloat(752))
        }

        picker.dismiss(animated: true, completion: {
            self.performSegue(withIdentifier: Segue.applyFilter.rawValue, sender: self)
        })
    }
}
```

This is a lot of code; therefore, let's break it all down so that you understand everything that you are doing:

- **Part A:**

  ```
  func showCameraUserInterface()
  ```

  This method is used to show the camera on the phone; however, if you are using a simulator, it will open the Photo Library instead.

- **Part B:**

  ```
  func generate(image:UIImage, ratio:CGFloat) -> UIImage
  ```

  This method is what we will use to take the images and crop them to the size we need and return an image in a smaller size.

  The Photo Library and Camera images are quite large. Therefore, if we did not use this method, it would take a long time for UI to go through and do everything we need.

- **Part C:**

  ```
  func imagePickerControllerDidCancel(_ picker:
  UIImagePickerController)
  ```

  If this method is called, it means that the user hit the Cancel button; and, therefore, we just dismiss the Controller and do nothing.

- **Part D:**

  ```
  func imagePickerController(_ picker: UIImagePickerController,
  didFinishPickingMediaWithInfo info: [String : Any])
  ```

  Here is where we will get the image from the Picker once it has been dismissed. We set our thumbnail and image values here; and then, we apply the `generate()` method in order to get them in a smaller size. Finally, we dismiss the Controller and use a segue to now display our `ApplyFilterViewController`.

Now, let's go back up to our `CreateReviewViewController` class and add the following:

```
import UIKit
import AVFoundation
import MobileCoreServices

class CreateReviewViewController: UITableViewController {

    @IBOutlet var tvReview: UITextView!
    @IBOutlet var tfName: UITextField!
    @IBOutlet var btnThumbnail: UIButton!
    @IBOutlet var btnRating: UIButton!

    var image: UIImage?
    var thumbnail: UIImage?

A   override func viewDidLoad() {
        super.viewDidLoad()

        initialize()
    }

B   override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

        switch segue.identifier! {
        case Segue.applyFilter.rawValue:
            showApplyFilter(with: segue)
        default:
            print("Segue not added")
        }
    }

C   func initialize() {
        requestAccess()
        updateTextView()
    }

D   func updateTextView() {
        tvReview.layer.borderColor = UIColor.lightGray.cgColor
        tvReview.layer.borderWidth = 0.5
        tvReview.layer.cornerRadius = 5.0
        tvReview.text = ""
    }

E   func showApplyFilter(with segue:UIStoryboardSegue) {
        guard let navController = segue.destination as? UINavigationController,
            let viewController = navController.topViewController as? ApplyFilterViewController else {
                return
        }

        viewController.image = image
        viewController.thumbnail = thumbnail
    }

F   func requestAccess() {
        AVCaptureDevice.requestAccess(forMediaType: AVMediaTypeVideo) { granted in
            if granted {}
        }
    }

G   func checkSource() {
        let cameraMediaType = AVMediaTypeVideo
        let cameraAuthorizationStatus = AVCaptureDevice.authorizationStatus(forMediaType: cameraMediaType)

        switch cameraAuthorizationStatus {

        case .authorized:
            showCameraUserInterface()

        case .restricted:
            break

        case .denied:
            break

        case .notDetermined:
            AVCaptureDevice.requestAccess(forMediaType: cameraMediaType) { granted in
                if granted {
                    self.showCameraUserInterface()
                } else {
                }
            }
        }
    }

    @IBAction func unwindReviewCancel(segue:UIStoryboardSegue) {}
    @IBAction func unwindRatingSave(segue:UIStoryboardSegue) {}
    @IBAction func unwindFilterSave(segue:UIStoryboardSegue) {
        if segue.source is ApplyFilterViewController {
H           if let thumbnail = thumbnail {
                btnThumbnail.setImage(thumbnail, for: .normal)
            }
        }
    }

I   @IBAction func onPhotoTapped(_ sender: AnyObject) {
        checkSource()
    }
}
```

Let's understand this code:

- **Part A:**

```
override func viewDidLoad() {
        super.viewDidLoad()

        initialize()
    }
```

  Here, we updated our `viewDidLoad()` method to add `initialize()`.

- **Part B:**

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?)
```

  This method will check the identifier for our `applyFilter` segue. If we have that segue, when it is called, we will run the `showApplyFilter()` method. Please note that if you tap on the Stars the app will crash. This is fixed in the next chapter.

- **Part C:**

```
func initialize()
```

  We are using this method to help keep our `viewDidLoad()` method clean and organized. We call two methods in this function.

- **Part D:**

```
func updateTextView()
```

  This method is used to add a border and a corner radius to our Text View.

- **Part E:**

```
func showApplyFilter(with segue:UIStoryboardSegue)
```

  As the method implies, we are going to show the `ApplyFilterViewController` when this method is called. We first check to make sure that our segue destination has a Navigation Controller; and, if so, we make sure that the top View Controller is the `ApplyViewController`. If that is true, we then pass our image and thumbnail that we obtained from our camera or camera roll.

- **Part F:**

```
func requestAccess()
```

  We use this method to ask the user for permission to use their camera and/or camera roll. This will only be called once; and, if the user accepts the request, the camera will be available. If the user denies the request, then the camera will not be available.



Using this code will result in intermittent crashes if you run your project in a simulator. This is a known bug in Xcode. The best way to test this code is to run the project on a device, where you should not have any crashes.

- **Part G:**

  ```
  func checkSource()
  ```

  This method is used to determine the device being used as well as what to do if the user does not give permission. For example, if the user does not give permission and still taps the photo thumb, you could present a modal that informs the user that they will need to go to the settings to enable the use of the camera or camera roll.

- **Part H:**

  ```
  @IBAction func unwindFilterSave(segue:UIStoryboardSegue)
  ```

  We created this earlier, but here we are checking for the instance when the user hits **Save**. When the user hits **Save**, we grab the image from the **Apply Filter** and set it to our thumbnail image.

- **Part I:**

  ```
  @IBAction func onPhotoTapped(_ sender: AnyObject)
  ```

  This method is called whenever the user hits the thumbnail image in our form. This will call the `checkSource()` method and display either the camera or the camera roll.

Now, in order to get rid of the errors that you see in the `CreateReviewViewController.swift` file, open the `Segue.swift` file and add the following:

```
case applyFilter
```

Next, we have a couple more housekeeping items to do. Firstly, we need to connect our `onPhotoTapped()` method to the thumbnail:

1. Open the `RestaurantDetail.storyboard` and select the **Create Review View Controller**.
2. Select the **Connections Inspector** in the **Utilities** panel.
3. Under **Received Actions**, click and drag from the empty circle next to **onPhotoTapped()** to the `btnThumbnail`, and then click **Touch Up Inside** in the pop-up screen that appears.

Lastly, Apple requires that if we use the camera or access the camera roll, we must let the user know that we are doing so and why. If you fail to do this, your code regarding the camera will not work and your app would be rejected when you submitted it. Let's take care of this now:

1. Open the `Info.plist` file and add the following two keys by hovering over any key and hitting the plus icon for the first key and then repeating again for the second key:
    - `NSPhotoLibraryUsageDescription`
    - `NSCameraUsageDescription`
2. For each key's value, enter anything you want as an alert that the user will see. In the following example, the value is set as **The app uses your camera to take pictures**:

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
Localization native development re...	String	en
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
Privacy - Camera Usage Description	String	The app uses your camera to take pictures
Privacy - Photo Library Usage Des...	String	The app uses your camera to take pictures
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(3 items)
▶ Supported interface orientations (i...	Array	(4 items)

Let's build and run the project by hitting the play button (or use *CMD* + *R*). You should now be able to get a photo from the Photo Library or use the camera. Once you have a photo, you will be able to apply a filter and save it. After you save, you should now able to see it in the **Create Review** form.

Summary

In this chapter, we covered a lot of new things. You learned how to use the camera and how to integrate the camera roll when a camera is not available. We used our first `UIScrollView` in order to put in a row of images. We also added some interaction for our Scroll View so that we could apply filters to our image. This chapter had a lot of code, and there may be some parts that were confusing. Review these parts and make sure that you fully understand them. There are numerous things in this chapter that you can reuse in a number of other apps.

In the next chapter, we will wrap up the Create Reviews section.

13
Saving Reviews

Our app is coming along nicely, and we are close to wrapping it up. In the last chapter, we created a restaurant review form, the Create Review form, which allows us to take pictures or use photos from our library. We can apply filters to the photos and even add more filters quickly by updating our `plist` file.

In this chapter, we will finish up working on the Create Review form. We will get the form fully working and be able to save the data entered into the form to what is known as Core Data. Core Data is a framework that handles persistent data. We will go much deeper into what Core Data is and how to use it in this chapter.

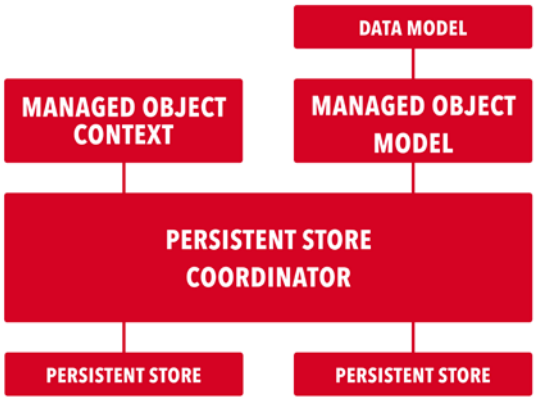
This is what we will cover in this chapter:

- What is Core Data?
- Creating our first Core Data model
- Saving items to Core Data
- Fetching items from Core Data
- Display items from Core Data into a table view

What is Core Data?

Let's start by taking a quote directly from Apple: *Core Data is a framework for managing and persisting an object graph.* Apple does not call Core Data a database, even though, behind the scenes, it saves data to a SQLite file. Core Data is very hard to explain to someone new to programming or to someone who has come from a different programming language. However, in iOS 10, Core Data has been greatly simplified. Having a general understanding of what Core Data does and how it works is sufficient for our purposes in this book.

When using the Core Data framework, you should be familiar with the **Managed Object Model**, the **Managed Object Context**, and the **Persistent Store Coordinator**. Let's look at a diagram to get a better understanding of how they interact with each other:



NSManagedObjectModel

The **Managed Object Model** represents the data model of your Core Data application. The Managed Object Model will interact with all of your data models (also known as entities) that you create within your app. This model will know of any relationships that your data may have in your app. The Managed Object Model will interact with your data model as well as with the Persistent Store Coordinator.

Entities are just objects that represent your data. In our app, since we are going to be saving customer reviews for restaurants, we will need to create a review entity.

NSManagedObjectContext

The **Managed Object Context** manages a collection of model objects, which it receives from the Persistent Store Coordinator. The Managed Object Context is responsible for creating, reading, updating, and deleting models. The Context is what you will be interacting with the most.

NSPersistentStoreCoordinator

The **Persistent Store Coordinator** has a reference to the Managed Object Model as well as the Managed Object Context. The Persistent Store Coordinator communicates with the Persistent Object Store. The Persistent Store Coordinator will interact with an object graph. This graph is where you will create your Entities and set up relationships within your app.

Core Data is not an easy topic, so you do not need to worry about the finer details. The more you work with Core Data, the easier it becomes to understand it. In this chapter, focus on obtaining a high level understanding, and the rest will come.

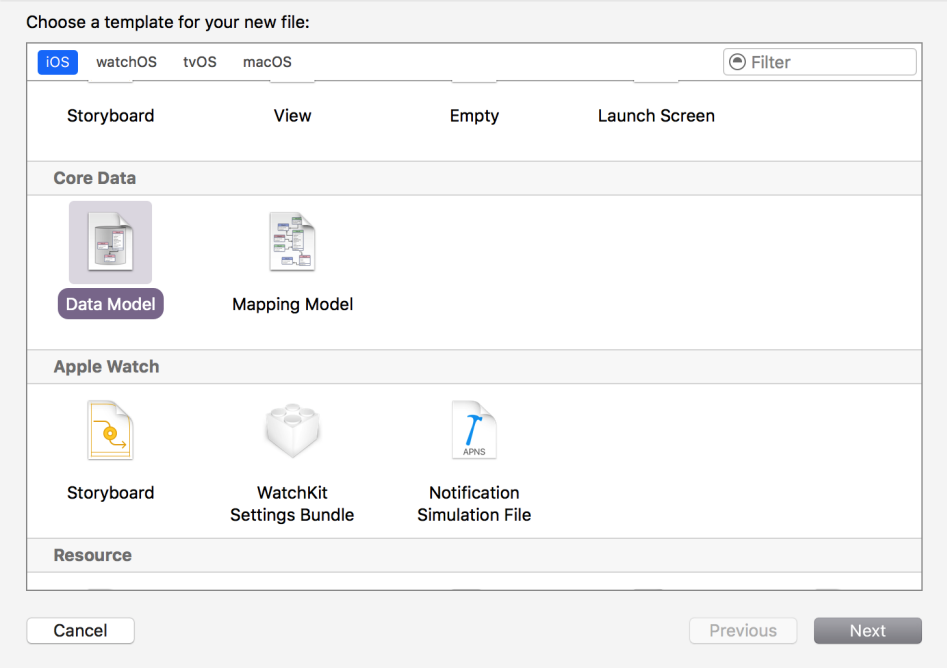
Before iOS 10, you had to create an instance of each of the Managed Object Model, the Managed Object Context, and the Persistent Store Coordinator. Now, in iOS 10, these have been consolidated into what is called NSPersistentContainer. We will cover this shortly, but first we need to create our data model.

Creating a data model

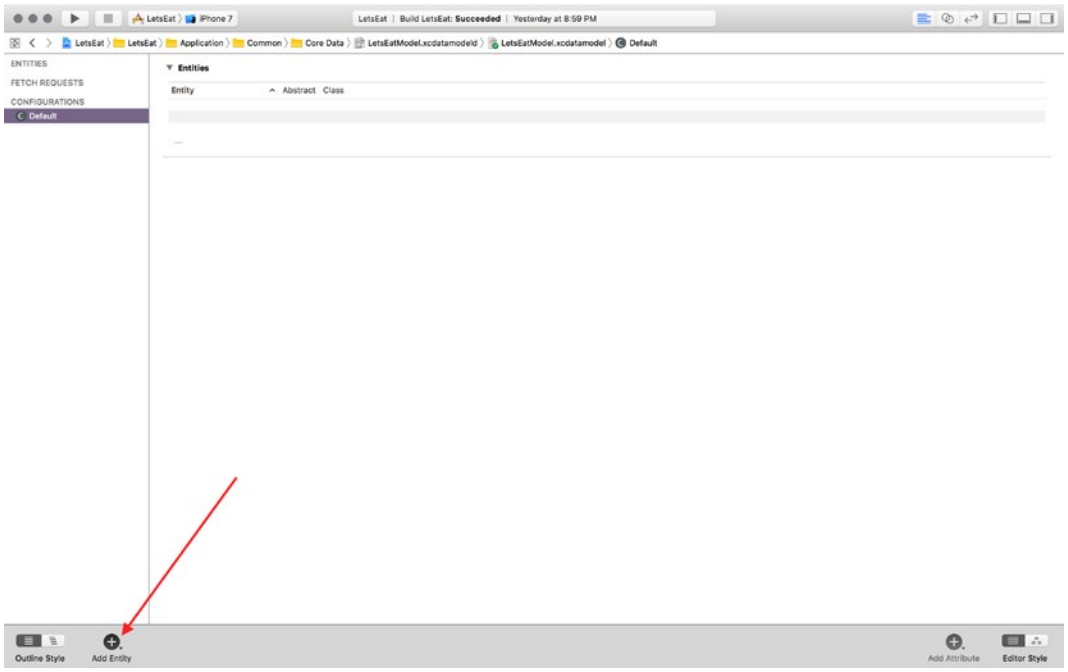
The data model is where you create your app's model objects and their properties. For our project, we only need to create one model object, called **Review**. Let's create a Managed Object Model now:

1. In the **Navigator** panel, right-click on the `Common` folder and create a new group, called `Core Data`.
2. Next, right-click this new `Core Data` folder and click **New File**.

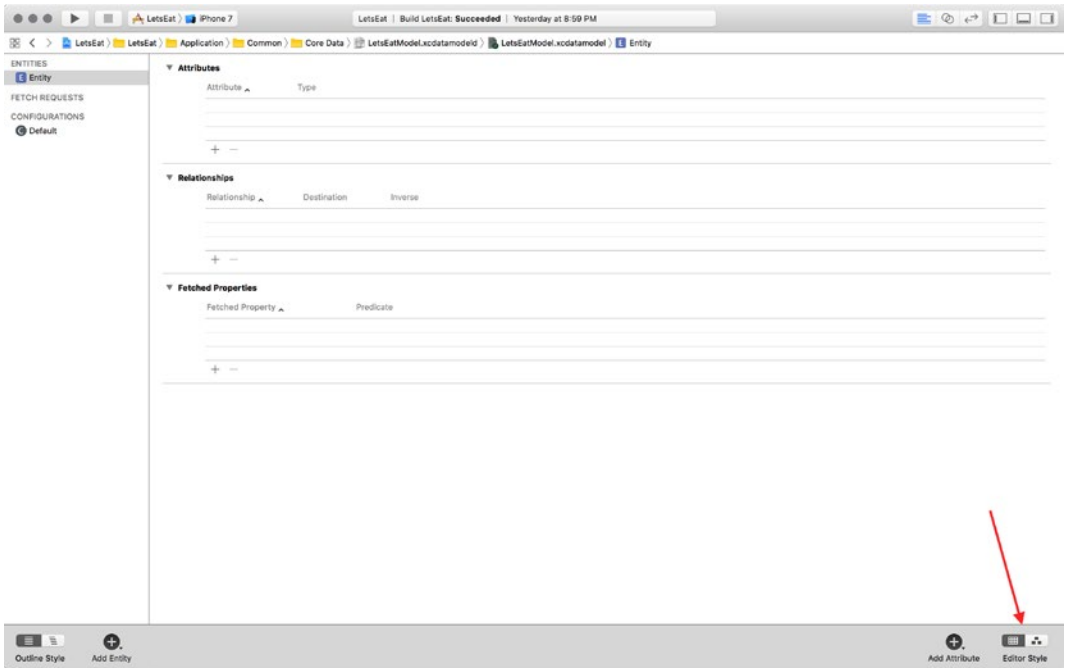
3. Inside the **Choose a template for your new file screen**, select **iOS** at the top, and then scroll down to the **Core Data** section and select **Data Model**. Then, hit **Next**:



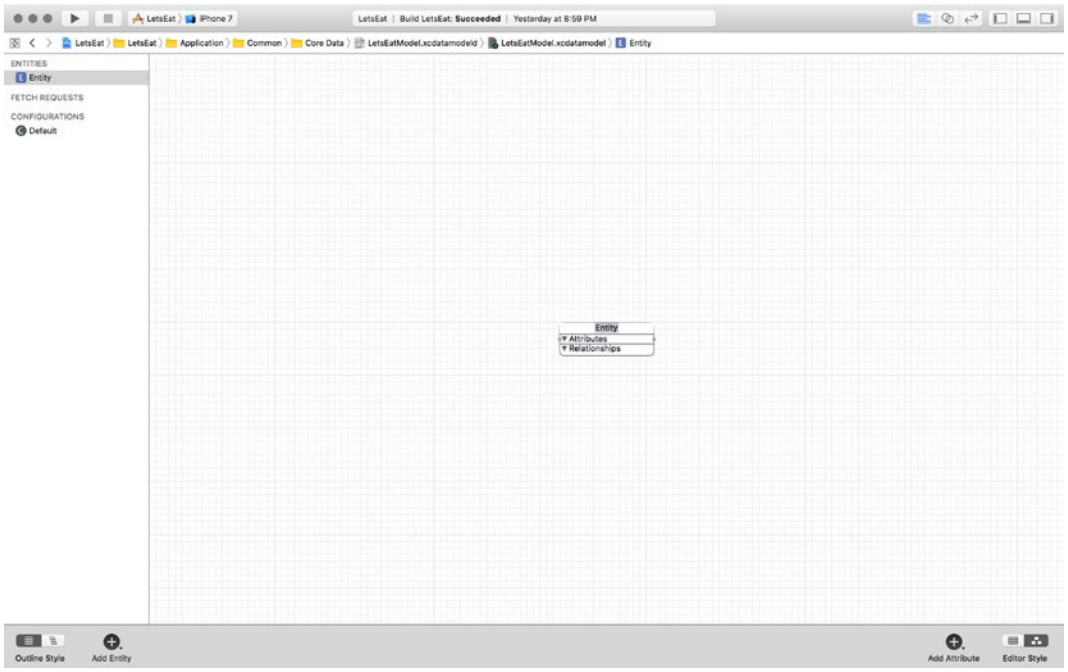
4. Name the file `LetsEatModel` and click **Create**.
5. Click **Add Entity** in the screen that appears:



6. Then, in the new screen, in the bottom-right corner, change the **Editor Style** to see the **Graph Style**:



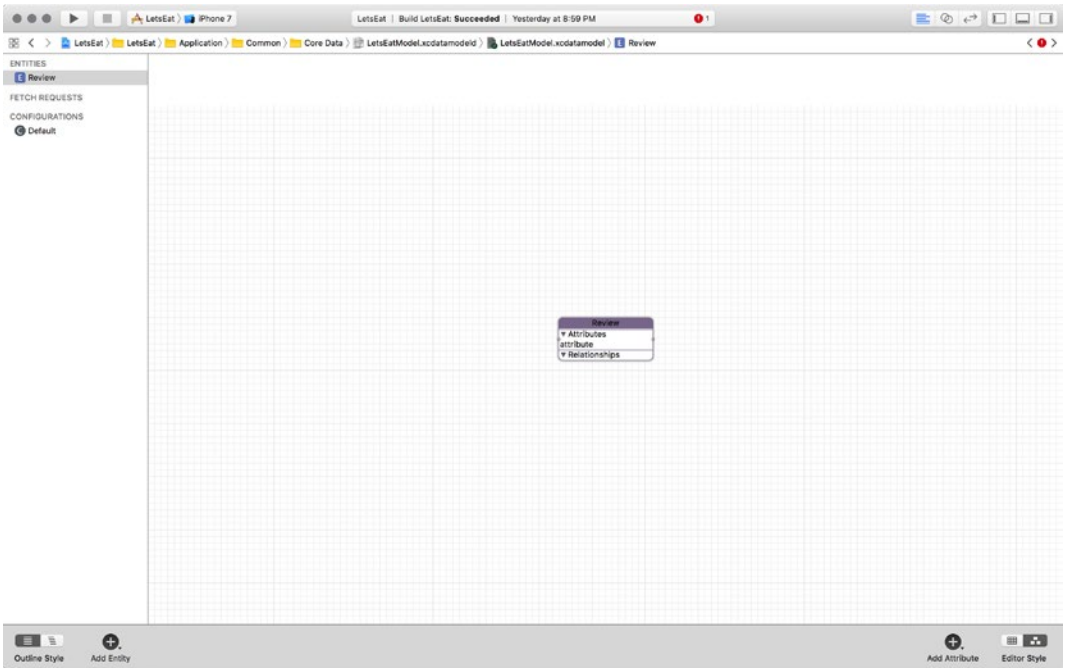
7. In the **Graph Style**, double click on **Entity** in the box in the middle of the graph to change our Entity's name:



8. Update the text to say **Review** and then hit *Enter*.

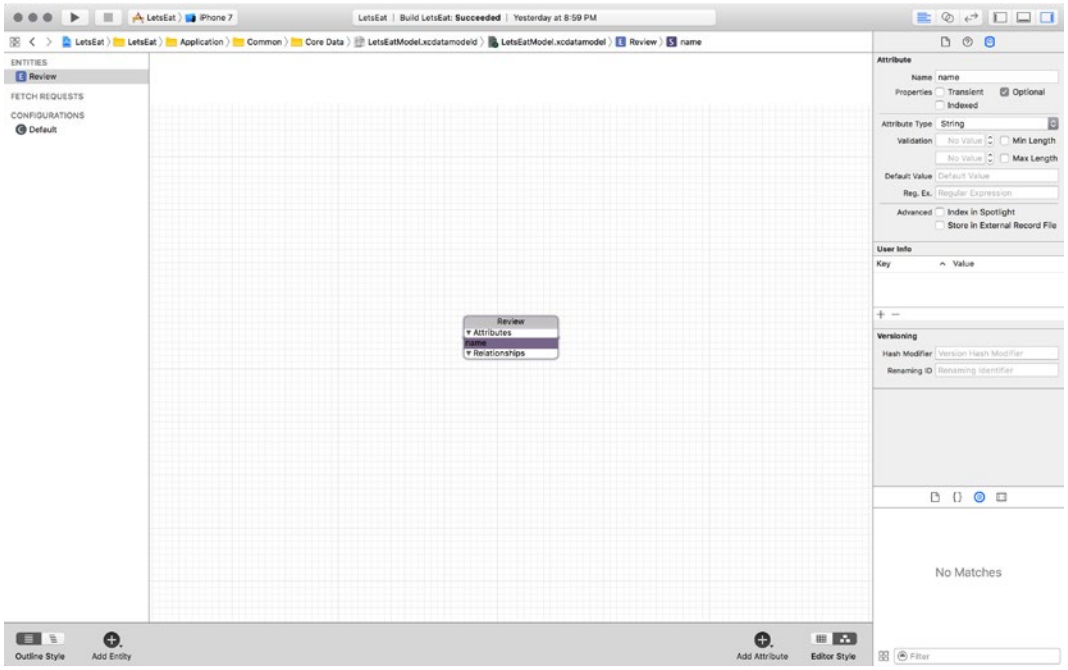
Now that we have our first Entity created, let's add our first attribute:

1. Select our **Review Entity** and click the **Add Attribute** button in the bottom right corner of the screen. The word attribute will now be under **Attributes** in the box in the middle of the screen:



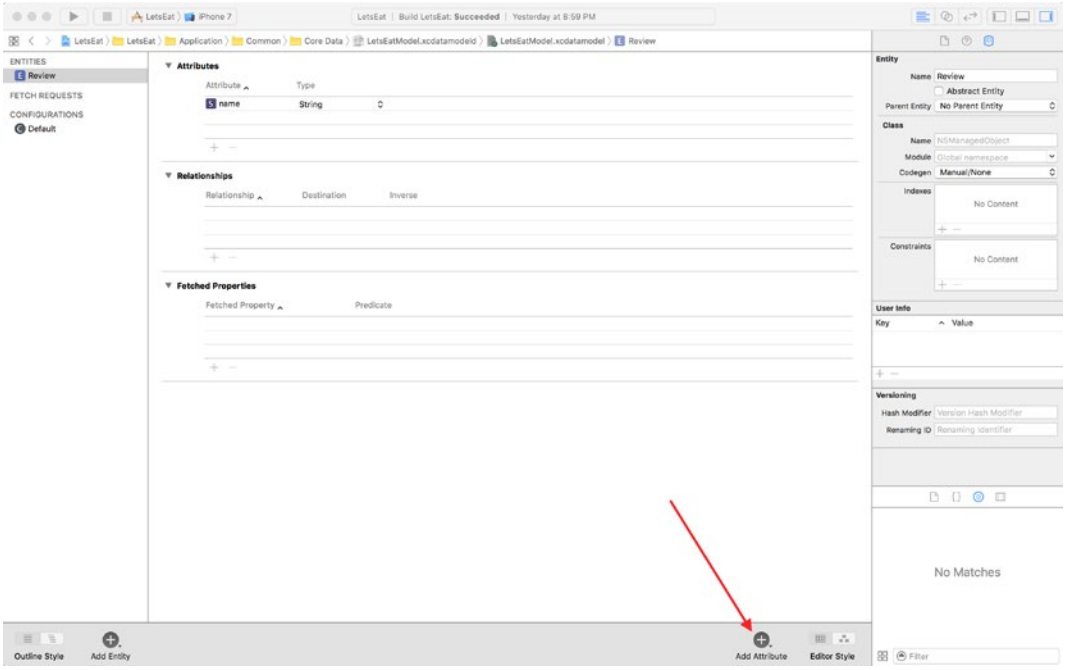
2. You will see that Xcode has given you an error. The reason for the error is that we created an attribute without giving it a type yet. Let's do this now.
3. Select the word **attribute** and open your **Utilities** panel. You will only see three icons, the **File Inspector**, the **Quick Help Inspector**, and the **Data Model Inspector**.
4. Select the last icon, the **Data Model Inspector**, and under **Attribute**, click on the dropdown for **Attribute Type** and change it from **Undefined** to **String**. The error should now disappear.
5. In addition, under **Attribute** in the **Data Model Inspector**, change **Name** from attribute to name, and hit **Enter**.

Your first attribute should now look as follows:



We created our first attribute in the **Graph Style** and now need to set up the rest of our attributes, which we will do in the **Table Style**:

1. Switch the **Editor Style** to the **Table Style** and then click **Add Attribute**:



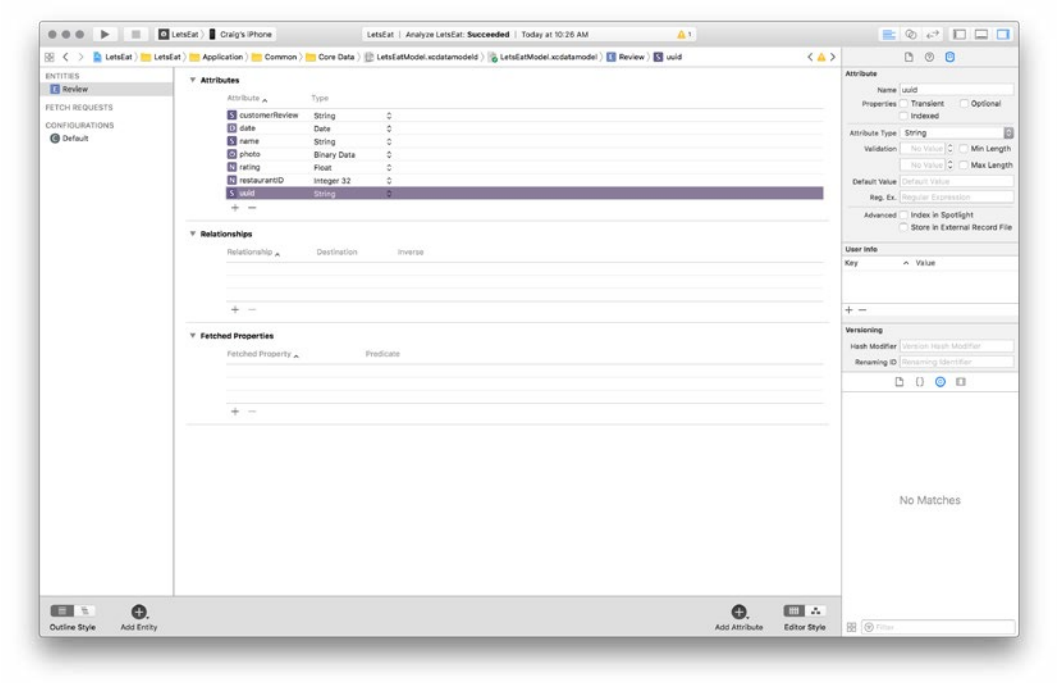
2. Update the attribute to date and set its data type to **Date**. You will not have to do anything in the **Data Model Inspector** for this attribute.
3. Next, select the **+** button in the **Attributes** section of the **Table Style** screen under the two attributes we just added.
4. Update this third attribute to `customerReview` and set its data type to **String**.
5. Now, add a fourth attribute, named `photo`, with a data type of **Binary Data**.
6. We cannot just store images in **Core Data** as they have to be converted to data first. Therefore, we will take the image used in the review and will convert it to binary data for **Core Data** to save. Then, when we pull the review out of the **Core Data**, we will convert it back to an image in order to display it.



For the purposes of learning, we will store images in Core Data. I would stay away from doing this as much as possible, because images can be large and you can quickly fill up the user's storage. If you are using a feed, you can save the URL path to the image instead of the actual image. If the user is not online, then you can simply display a placeholder in its place.

7. Next, add a fifth attribute, named `rating` with a data type of **Float**.
8. Now, add a sixth attribute, named `restaurantID` with a data type of **Integer 32**.
9. When we save reviews, we will save them with their `restaurantID`. Whenever we go to a restaurant detail page, we will get all of the reviews just for that specific restaurant and then display them. If we do not have any reviews, then we will display a default message.
10. Lastly, add a seventh attribute, named `uuid` with a data type of **String**, and under **Attribute** in the **Data Model Inspector**, uncheck the **Optional** checkbox. This attribute will be our unique ID for each review.

Your Attributes table should now look like the following:

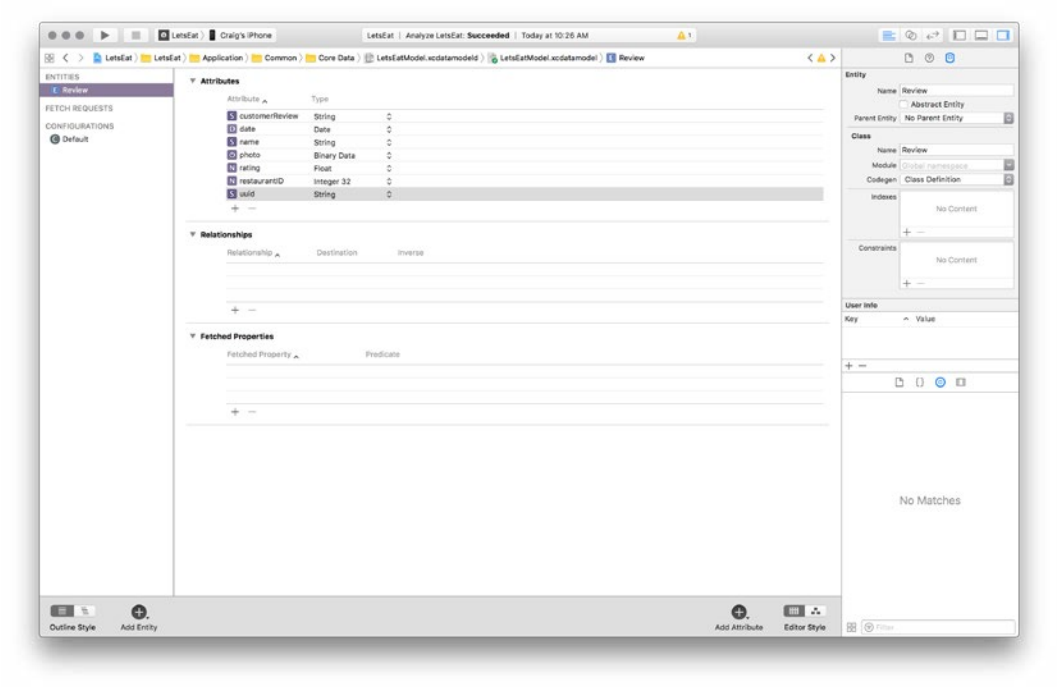


Now that we have our attributes set, we need to do a few more things before we start working on some code.

Entity auto-generation

We could have Xcode create a file for our **Review Entity**; however, if we wanted to add more attributes, we would have to generate more code. Core Data offers an ability to auto-generate our code for us. We will take advantage of this feature:

1. In the left panel, in the list of **Entities**, select our only **Entity**, **Review**.
2. After you select the **Entity**, select the **Data Model Inspector** in the **Utilities** panel. You will notice that your **Data Model Inspector** panel changed from when we were working on our **Attributes**.
3. Under **Class**, change **Codegen** to **Class Definition**:



4. Now, hit *CMD + B* to build the project. This will create the **Review** class that we created in **Core Data**. You will not see the file anywhere, but it has been created.

Review item

This new **Review** class will be what we get back from **Core Data** when we need to fetch items from it. Instead of passing the **Review** class around, we will create a generic data object that we can use instead.



When I work with stored data, I typically like to have two model objects: one that is used when storing the data and the other that is generic. In the past, passing around Core Data objects caused a lot of technical issues. These issues have been addressed in iOS 10; however, in an overabundance of caution, I typically will get the items from Core Data and then convert those objects into a `struct`.

Let's create this file now:

1. Right-click the `Model` folder in the `Review` folder and select **New File**.
2. Inside the **Choose a template for your new file screen**, select **iOS** at the top and then **Swift File**. Then hit **Next**.
3. Name this file `ReviewItem` and hit **Create**.
4. Update your file to the following:

```
import UIKit

struct ReviewItem {
    var rating:Float?
    var photo:UIImage?
    var name:String?
    var customerReview:String?
    var date:NSDate?
    var restaurantID:Int?
    var uuid = UUID().uuidString

    var photoData:NSData {
        guard let image = photo else {
            return NSData()
        }

        return NSData(data: UIImagePNGRepresentation(image)!)
    }

  var displayDate:String {
         let formatter = DateFormatter()
         formatter.dateFormat = "MMMM dd, yyyy"

         return formatter.string(from: self.date as! Date)
```

```
            }
    }

    extension ReviewItem {
        init(data:Review) {
            self.date = data.date
            self.customerReview = data.customerReview
            self.name = data.name
            self.restaurantID = Int(data.restaurantID)

            if let photo = data.photo {
                self.photo = UIImage(data:photo as Data, scale:1.0)
            }
            else {
                self.photo = UIImage(named: "restaurant-list-img")
            }

            self.rating = data.rating
            if let uuid = data.uuid { self.uuid = uuid }
        }
    }
```

The first part of this file should be familiar to you, except for the `photoData` variable. Since we cannot store an image directly into Core Data, we need to convert it to binary data. This variable handles that for us and makes it easier for when we save an item to just pass `photoData` to Core Data.

The extension in this file allows us take the **Review** from **Core Data** and map it to a `ReviewItem`. Our custom `init()` method allows us to just pass the Review object into the init parameters. Now that we have our `ReviewItem`, we need to set up our manager next.

Core Data Manager

As we have done throughout the book, we are going to create a Manager class. This class will be responsible for getting data in and out of Core Data. Let's get started:

1. Right-click the `Core Data` folder in the `Common` folder and select **New File**
2. Inside of the **Choose a template for your new file screen**, select **iOS** at the top and then **Cocoa Touch Class**. Then hit **Next**.
3. In the options screen that appears, add the following:
4. New File:

    Class: CoreDataManager

    Subclass…: NSObject

Also create XIB: Unchecked

Language: Swift

5. Click **Next** and then **Create**.

When the file opens, under your `import UIKit`, add the following:

```
import CoreData
```

This allows us to have access to the Core Data library.

Next, inside of the class definition, add the following:

```
let container:NSPersistentContainer
```

This constant, which is a `NSPersistentContainer`, gives us everything we need within a Core Data stack. As we discussed earlier, the `NSPersistentContainer` is composed of three things, a Persistent Store Coordinator, a Managed Object Context, and a Managed Object Model.

You may have noticed an error after adding this variable. The reason for the error is that we do not have an `init()` method created.

Let's add this `init()` method after the constant we just added:

```
override init() {
    container = NSPersistentContainer(name: "LetsEatModel")
    container.loadPersistentStores { (storeDesc, error) in
        guard error == nil else {
            print(error?.localizedDescription as Any)
            return
        }
    }

    super.init()
}
```

This code is initializing the container and grabbing the Managed Object Model we created earlier. The model will now be able to see all of our Entities and attributes therein.

Our `CoreDataManager` needs to do two things for us. We need to be able to add new a `ReviewItem` and fetch it. When we save a restaurant review, we want to be able to save the review with the restaurant. We do not need to save all of the restaurant information, since we can simply use the `restaurantID`. When we go to restaurant details, we can check Core Data for any reviews for a particular restaurant by its `restaurantID`. Let's add the following method after our `init()` method to accomplish this task for us:

```
                          A
func fetchReviews(by identifier:Int) -> [ReviewItem] {
    let moc = container.viewContext
B   let request:NSFetchRequest<Review> = Review.fetchRequest()
    let predicate = NSPredicate(format: "restaurantID = %i", Int32(identifier))

    var items:[ReviewItem] = []

C   request.sortDescriptors = [NSSortDescriptor(key: "date", ascending: false)]
    request.predicate = predicate

    do {
        for data in try moc.fetch(request) {
            items.append(ReviewItem(data: data))
        }
D
        return items

    } catch {
        fatalError("Failed to fetch reviews: \(error)")
    }
}
```

Let's review this code:

- **Part A:**

  `func fetchReviews(by identifier:Int) -> [ReviewItem]`

  We are passing an ID over and using it to find reviews for a particular restaurant.

- **Part B:**

  `let moc = container.viewContext`

  Here, we are creating an instance of the **Managed Object Context** (**moc**). This variable allows us to interact with Core Data.

  `let request:NSFetchRequest<Review> = Review.fetchRequest()`

  Then, we are creating a `fetch` request. This request is passed to the Managed Object Context and tells it what we need.

  ```
  let predicate = NSPredicate(format: "restaurantID = %i",
  Int32(identifier))
  ```

  Next, we are creating a `predicate`, which allows us to apply some search parameters. Specifically, we are saying that we want every `ReviewItem` that has the id that we pass it.

Part C:

```
request.sortDescriptors = [NSSortDescriptor(key: "date",
ascending: false)]
request.predicate = predicate
```

Here, we are applying a sort descriptor to our request. Instead of getting reviews back in a random order, we sort all of the reviews by date.

Part D:

```
do {
    for data in try moc.fetch(request) {
        items.append(ReviewItem(data: data))
    }

    return items
} catch {
    fatalError("Failed to fetch reviews: \(error)")
}
```

Finally, we are wrapping everything into a `do-catch` block. When the search occurs, it will return an array of `ReviewItems` or, if there were no `ReviewItems`, an empty array. If there was a problem with your setup, then you'll get a fatal error. When the fetch is complete, we then loop through the items and create our `ReviewItems`.

Here, we are applying a sort descriptor to our request. Instead of getting reviews back in a random order, we sort all of the reviews by date.

We've added our method to get reviews, and now we need to add a method to save them into Core Data. Let's add this method by adding the following after our `init()` method:

```
func addReview(_ item:ReviewItem) {
    let review = Review(context: container.viewContext)
    review.name = item.name
    review.date = NSDate()
    if let rating = item.rating { review.rating = rating }
    review.customerReview = item.customerReview
    review.photo = item.photoData
    review.uuid = item.uuid

    if let id = item.restaurantID {
        review.restaurantID = Int32(id)

        print("restaurant id \(id)")
     save()
        }
}
```



You will get an error because we have not created the `save()` method yet. Ignore it for now as it will be created next.

This `addReview()` method takes a `ReviewItem` in the parameters. We convert the `ReviewItem` into a Review and then call the `save()` method.

Now, let's add the `save()` method after the `addReview()` method we just created:

```
fileprivate func save() {
    do {
        if container.viewContext.hasChanges {
            try container.viewContext.save()
        }
    }
    catch let error {
        print(error.localizedDescription)
    }
}
```

Once again, we are wrapping everything into a `do-catch` block. Inside of the `do`, we check to see if the Managed Object Context has changed. If it has changed, then we call the `save()` method. We have now completed our Core Data Manager.

Next, we need to create another manager class. This manager will be responsible for making the calls to the Core Data Manager, similar to how the corresponding manager in the Explore Manager is responsible for getting the data from the `plist`. Let's create this manager file now:

1. Right-click the `Model` folder in the `Review` folder and select **New File**.
2. Inside of the **Choose a template for your new file screen**, select **iOS** at the top and then **Cocoa Touch Class**. Then hit **Next**.
3. In the options screen that appears, add the following:
4. New File

   Class: ReviewDataManager

   Subclass…: NSObject

   Also create XIB: Unchecked

   Language: Swift

5. Hit **Next** and then **Create**.

Update your file to the following:

```
import UIKit

class ReviewDataManager: NSObject {

    private var items:[ReviewItem] = []
    let manager = CoreDataManager()

    func fetchReview(by id:Int) {

        if items.count > 0 { items.removeAll() }

        for data in manager.fetchReviews(by: id) {
            items.append(data)
        }
    }

    func numberOfItems() -> Int {
        return items.count
    }

    func reviewItem(at index:IndexPath) -> ReviewItem {
        return items[index.item]
    }
}
```

This manager class is similar to the other managers that we have created so far. In this manager, our fetch method takes an ID in the parameter. This ID will represent the `restaurantID` that we will use in order to search for `ReviewItems` in Core Data. If we find any `ReviewItems`, we add them to our array.

Although we now have most of what we will need here, we are missing a method to handle the two **Views** that are in the **Restaurant Detail View**, one that displays no reviews and another that displays the most recent review. After the `reviewItem(at index:)` method and before the last curly brace, add this method:

```
func getLatestReview() -> ReviewItem {
    guard let item = items.first else {
        return ReviewItem()
    }

    return item
}
```

Since the array is already sorted by date, we can grab the first item in the array and use that to get the latest review.

Creating star ratings

Before we can save the reviews, we need to get all of the data. Getting text from a text field and text view is pretty simple, so we will do that. However, we need a way to let users rate a restaurant by giving it a star rating. The ratings range from 0 to 5 with increments of 0.5. We will use a **Picker View** to display the stars. Let's get started with this:

1. Right-click the `Model` folder in the `Review` folder and select **New File**.
2. Inside of the **Choose a template for your new file screen**, select **iOS** at the top and then select **Swift File**. Then hit **Next**.
3. Name this file `Rating` and hit **Create**.

Under the `import` statement, add the following:

```
enum Rating {
    case zero
    case half
    case one
    case oneHalf
    case two
    case twoHalf
    case three
    case threeHalf
    case four
    case fourHalf
    case five
}
```

Now that we have a case for each type of rating, we want a way to be able to have a value that we can use for a rating. Add the following variable inside of the `enum` after `case five`, but before the last curly brace:

```
var value: Float {
    switch self {
    case .zero: return Float(0)
    case .half:return Float(0.5)
    case .one: return Float(1)
    case .oneHalf: return Float(1.5)
    case .two: return Float(2)
    case .twoHalf: return Float(2.5)
    case .three: return Float(3)
    case .threeHalf: return Float(3.5)
    case .four: return Float(4)
    case .fourHalf: return Float(4.5)
    case .five: return Float(5)
    }
}
```

This `value` variable will return us a float value. For example, if we wanted to get a value for `3.5`, we would write the following:

```
let value = Rating.value
```

Finally, we need another variable that will let us display an image for that value. Add this inside the enum, but under the `value` variable you just added and before the last curly brace:

```
static func image(rating:Float) -> String {
    var images = ["0star", "05star", "1star", "15star", "2star",
"25star", "3star",  "35star", "4star", "45star", "5star"]
    var values:[Float] = []
    for i in stride(from: 0, through: 5.0, by: 0.5) {
        values.append(Float(i))
    }

    guard let index = values.index(of: rating) else {
        return ""
    }
    return images[index]
}
```

Here, we loop through an array using the `stride` method, which will create numbers from 0.0 to 5.0, but increment each time by 0.5. Then, when we pass in a value, for example, 1, the method will find the index of the rating and give us back an image.

For example, if we wanted an image for **1star**, we would write the following:

```
let image = Rating.image(1.0)
```

Now, in order to display stars inside of a Picker View, we need to create a custom **UIView**:

1. Right-click the `View` folder in the `Review` folder and select **New File**.
2. Inside of the **Choose a template for your new file screen**, select **iOS** at the top and then **Cocoa Touch Class**. Then, hit **Next**.
3. In the options screen that appears, add the following:
4. New File:

   Class: RatingView
   Subclass…: UIView
   Language: Swift

5. Click **Next** and then **Create**.

In this new file, delete the commented code and add the following code inside of the class declaration:

```
@IBOutlet private var contentView:UIView?
@IBOutlet var imgRating: UIImageView!

override init(frame: CGRect) {
    super.init(frame: frame)
}

init(frame:CGRect, value:Rating) {
    super.init(frame: frame)

    setupDefaults()
    setup(rating: value)
}

required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    setupDefaults()
}

private func setupDefaults() {
    Bundle.main.loadNibNamed("RatingView", owner: self, options: nil)
    guard let content = contentView else { return }
    content.frame = self.bounds
    content.autoresizingMask = [.flexibleHeight, .flexibleWidth]
    self.addSubview(content)
}
```

First, let's get rid of the error you see by adding the following under the `setupDefaults()` method:

```
func setup(rating:Rating) {
    imgRating.image = UIImage(named: Rating.image(rating: rating.
value))
}
```

This method will allow us to pass a rating in and display an image.

Now, let's discuss the `RatingView` class, which is going to be handled a bit differently. We are actually going to attach this class to another file, which we will create next. The way this will work is similar to how we made our cells work. For the cells, we have used the story board file; this will be similar, but on a much smaller scale.

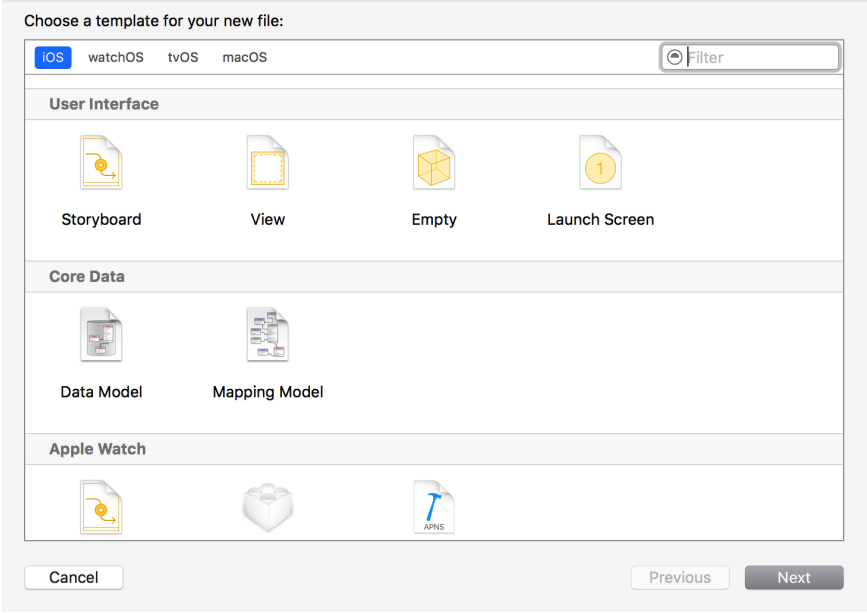
First, we have created two `IBOutlets` that we will need to hook up. In addition, we have three `init()` methods. All of them are required, because this is boilerplate code for attaching a UIView class to an **XML Interface Builder** (`xib`) file (also called a `nib`). A `XIB` file is a resource that allows you to create a visual element. You can then attach this element to a class file and use it to display functionality.



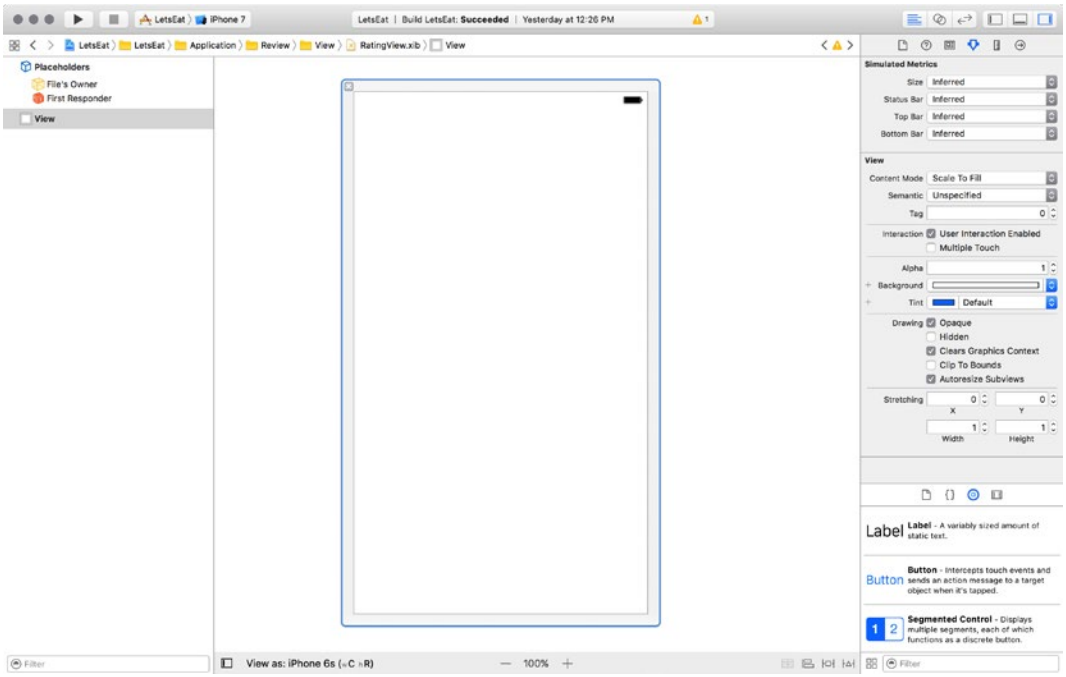
My preference is to use `xibs` for Table View and Collection View cells. I also like using them whenever I need to do some small view. I prefer to have my design be in a `xib` and my logic be in code. This is just a preference, and you may prefer to do everything in code. This book is here to show you all of the options so that you can pick and choose what you like. Having options helps you make a decision for when you need to decide how to approach something. As I have said, there are different ways to code things, and some are more efficient than others. Do not be afraid to experiment, and when you find a better or more effective way, you can just refactor.

Next, let's create our `xib` file. You might have noticed that when we created the `RatingView` file, it would not allow us to create a `xib` file. We will need to create this file ourselves:

1. Right-click the `View` folder in the `Review` folder and select **New File**.
2. Inside of the **Choose a template for your new file screen**, select **iOS** at the top and then scroll down to the **User Interface** section and select **View**. Then, hit **Next**:

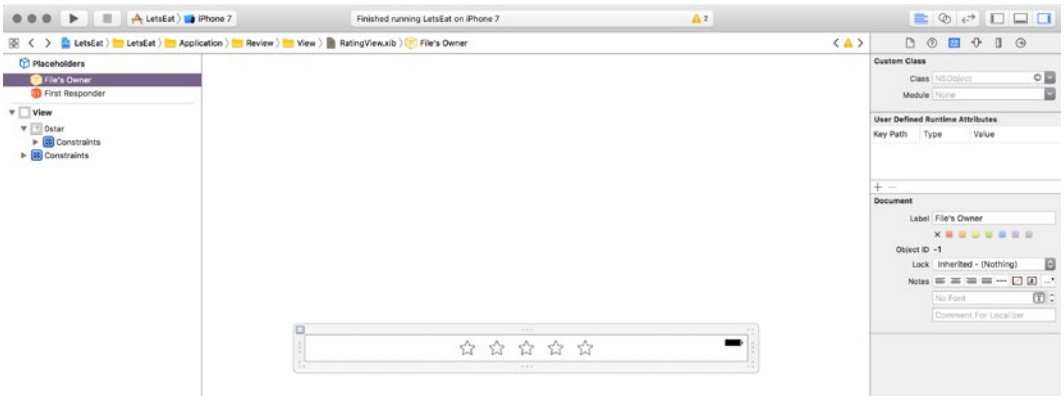


3. Name the file `RatingView`, and hit **Create**:

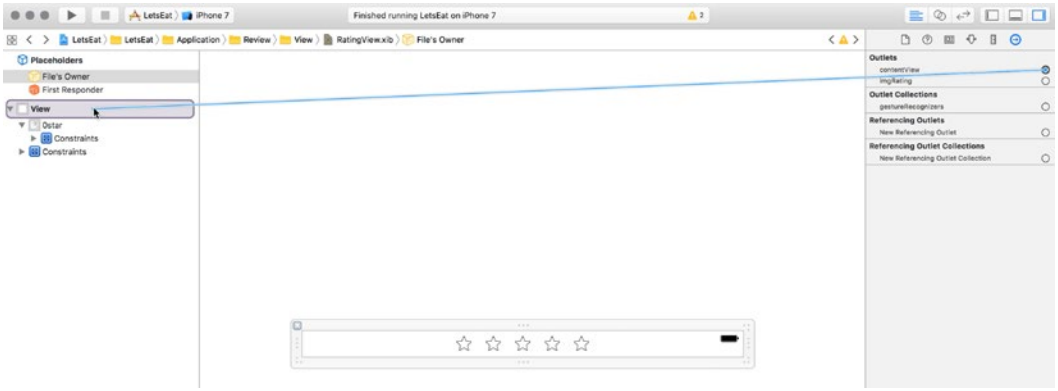


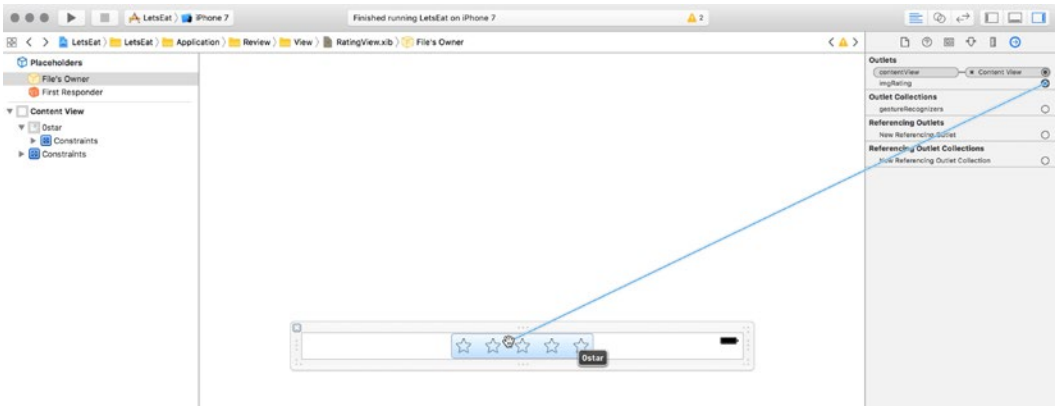
As you can see, inside of the file, there is a blank screen. This view is too big for what we need. Therefore, let's update this file to fit our needs:

1. With `RatingView.xib` selected, open the **Attributes Inspector** in the **Utilities** panel, and in **Size** under **Simulated Metrics**, select **Freeform** from the drop down menu.
2. Next, select the **Size Inspector** in the **Utilities** panel and update the width to **600** and the **height** to **35**.
3. Select the **Media library** at the bottom of the **Utilities** panel and drag out the **0star** into the **View**.
4. With this **0star** selected, in the **Size Inspector**, update the following values:
    - **X**: 210
    - **Y**: 7.5
5. Next, select the Pin icon and update the following values:
    - **Width**: checked
    - **Height**: checked
6. Then, click **Add 2 Constraints**.
7. Now, select the **Align** icon (to the left of the Pin icon) and update the following values, and then click **Add 2 Constraints**:
    - **Horizontally in Container**: checked
    - **Vertically in Container**: checked
8. This will ensure that no matter how big our Picker is, the images will be centered. Our `setupDefaults()` method is loading the `xib` file in setting up the frame.
9. Hit **Save**, and in the **Outline view**, select **File's Owner** and then the **Identity Inspector** in the **Utilities** panel:



10. Under **Custom Class**, in the **Class** drop-down menu, type in `RatingView`, select it, and then hit *Enter*.
11. Then, select the **Connections Inspector** in the **Utilities** panel. Then, under **Outlets**, click and drag from the empty circle next to `contentView` to the View in the Outline view and from the empty circle next to `imgRating` to the stars image in the scene:





Now that we made our connections, we need to turn to the rating part of our **Create Review** form, since we have addressed the photo part of the form in the last chapter. In Storyboard, we already have the setup complete, but now we need to attach a class to it:

1. Right-click the `Star Rating` folder inside the `Controller` folder that is in the `Review` folder and select **New File**.
2. Inside of the **Choose a template for your new file screen**, select **iOS** at the top, and then **Cocoa Touch Class**. Then, hit **Next**.
3. In the options screen that appears, add the following:
4. New File:

   Class: StarRatingViewController

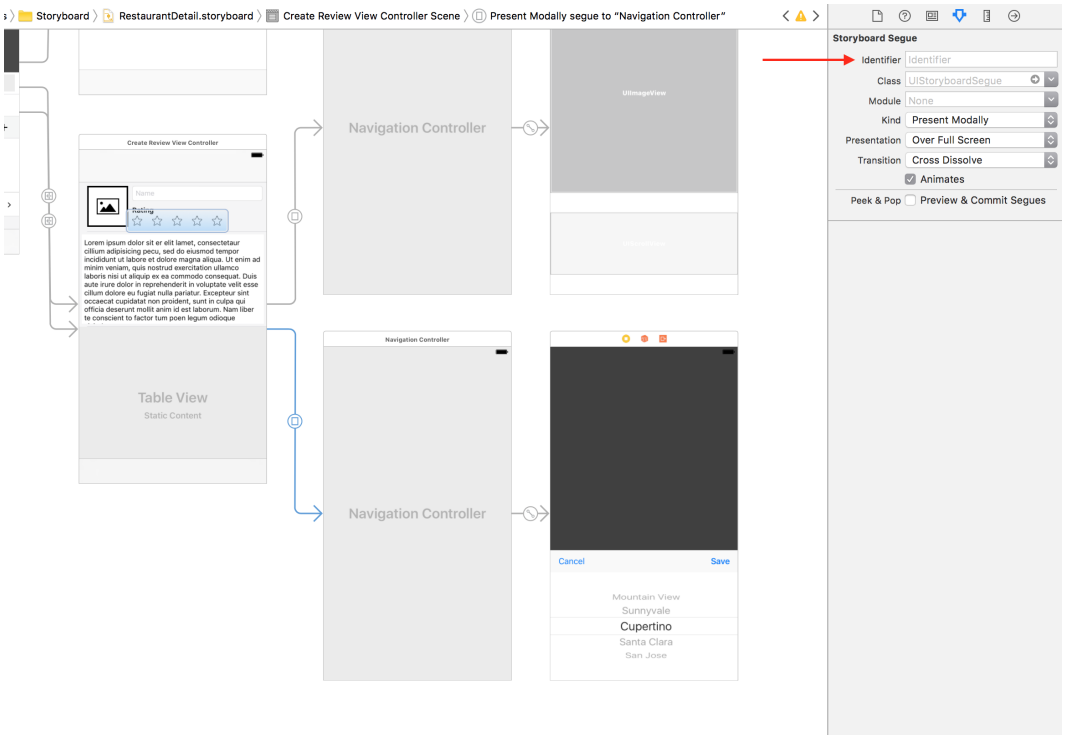
   Subclass…: UIViewController

   Also create XIB: Unchecked

   Language: Swift

5. Click **Next** and then **Create**.

6. Now, open the `RestaurantDetail.storyboard` and, in the **Outline** view, select the **View Controller** for the scene that has the **UIPickerView**, and then the **Identity Inspector** in the **Utilities** panel.
7. Under **Custom Class**, in the **Class** drop-down menu, type in `StarRatingViewController`, select it, and then hit *Enter*.
8. Next, select the segue that is connected to the **Navigation Controller** and **Star Rating View Controller**.
9. In the **Utilities** area, select the **Attributes Inspector** and update **Identifier** to `showRating`. Then, hit *Enter*:



10. Next, one at a time, select each of the two segues that are connected to the **Restaurant Detail** and **Create Review View Controllers** (one from the **plus** button and the other from the **Add a Review** button) and update each of the segue's Identifiers to `showReview` in the **Attributes Inspector**. Hit *Enter* after each update:



11. Now, open the `Segue.swift` file and add the following case statements before the last curly brace:

```
case showRating
case showReview
```

Let's build and run the project by hitting the **play** button (or use *CMD* + *R*). When you tap on the stars in the **Create Review** form, you should now see the modal appear. Since we set it up in the previous chapter, you should also be able to dismiss the modal by using either the **Save** or **Cancel** Buttons.

Next, we need to complete the Picker View by filling it with the different star ratings from which to choose.

Open the `StarRatingViewController.swift` file and then delete the `didReceiveMemoryWarning()` method.

Next, inside of the class declaration, add the following variables:

```
@IBOutlet var pickerView: UIPickerView!

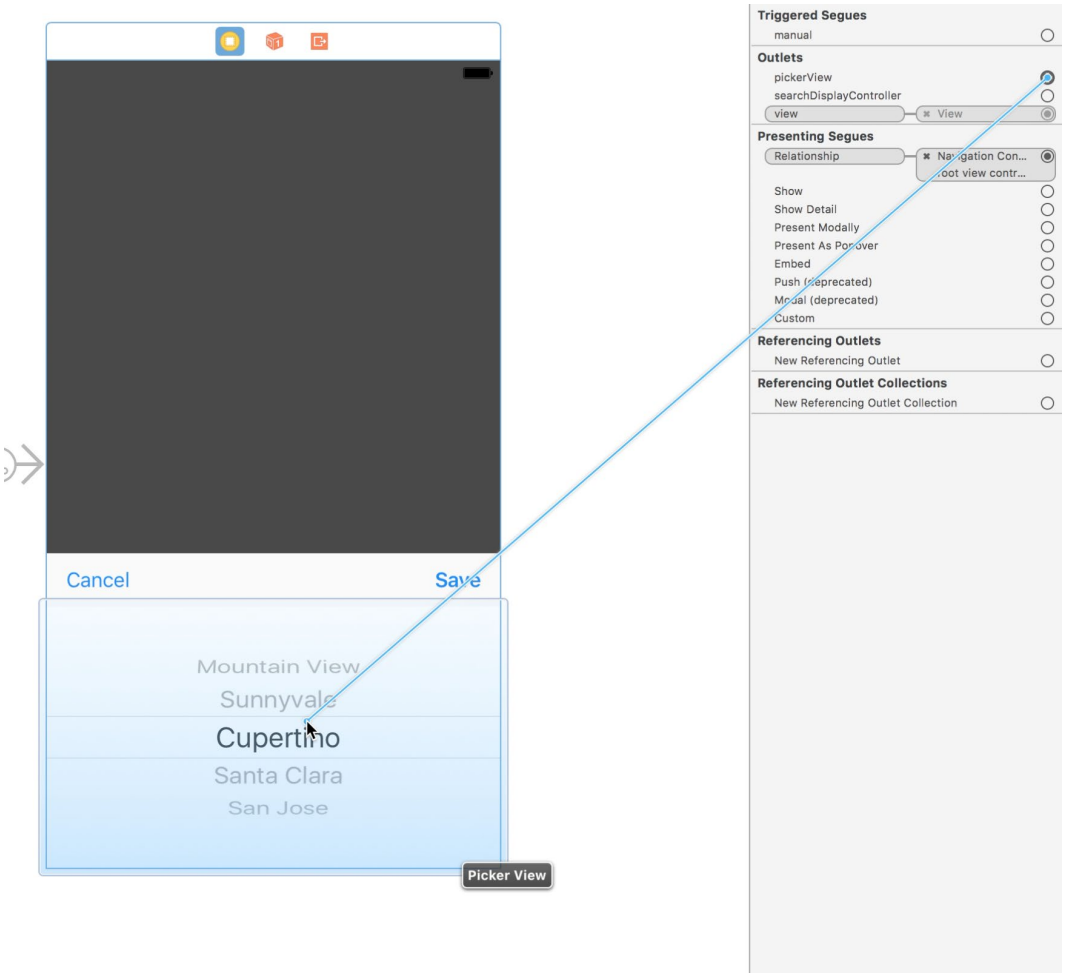
var selectedRating:Rating = Rating.zero
var pickerDataSource = [Rating.five, Rating.fourHalf, Rating.four,
Rating.threeHalf, Rating.three, Rating.twoHalf, Rating.two, Rating.
oneHalf, Rating.one, Rating.half, Rating.zero]
```

Here, we have two variables, one for our `UIPickerView` and one for the `selectedRating` that we will use to pass to the **Create Review View** Controller. In addition, we are setting up our data source for our Picker and filling it with stars, ranging from 5.0 to 0.0.

Next, we need to connect our `pickerView` variable to our `UIPickerView`:

1. Open the `RestaurantDetail.storyboard` and select the **Star Rating View** Controller.

2. Select the **Connections Inspector** in the **Utilities** panel, and click and drag from the **pickerView** variable to the **Picker View** in the scene:



Return to the `StarRatingViewController.swift` file and add the following after the `viewDidLoad()` method, but before the last curly brace:

```
func initialize() {
    setDefaults()
}

func setDefaults() {
    pickerView.dataSource = self
    pickerView.delegate = self
```

```
    if let index = pickerDataSource.index(of: selectedRating) {
pickerView.selectRow(index, inComponent: 0, animated: false)
}
}
```

Here, we are setting up some defaults for our Picker View. You also might notice that we are setting the data source and delegate methods in code. Therefore, you have the option to set the data source and delegate methods in Storyboard or code.



After entering the preceding code, you will see errors. Ignore them for now as we will fix them when we add our extension shortly.

Then, add `initialize()` to the `viewDidLoad()` method.

After the last curly brace, add the following extension (which will get rid of the errors noted previously):

```
extension StarRatingViewController:UIPickerViewDataSource, UIPickerViewDelegate {
A   func numberOfComponents(in pickerView: UIPickerView) -> Int {
        return 1
    }

B   func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
        return pickerDataSource.count
    }

    func pickerView(_ pickerView: UIPickerView, viewForRow row: Int, forComponent component: Int, reusing view: UIView?) ->
        UIView {
C       let frame = CGRect(x: 0, y: 0, width: pickerView.rowSize(forComponent: component).width-10, height:
            pickerView.rowSize(forComponent: component).height)
        let ratingView = RatingView(frame: frame, value: pickerDataSource[row])

        return ratingView
    }

D   func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int) {
        selectedRating = pickerDataSource[row]
    }
}
```

Let's discuss the various methods in this code:

- Part A

  ```
  func numberOfComponents(in pickerView: UIPickerView) -> Int
  ```

  Here, we are telling our Picker View how many components (or columns) there are.

- Part B

  ```
  func pickerView(_ pickerView: UIPickerView,
  numberOfRowsInComponent component: Int) -> Int
  ```

  This method is where we set the number of rows we will need.

- Part C

  ```
  func pickerView(_ pickerView: UIPickerView, viewForRow row: Int,
  forComponent component: Int, reusing view: UIView?) -> UIView
  ```

  This method is where we add our custom View into the Picker View. We create an instance of the View and then set it into the Picker View row.

- Part D

  ```
  func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int,
  inComponent component: Int)
  ```

  Lastly, we check for which row was selected, and we save it to our `selectedRating` variable.

Now that we have our Star Rating View set up, we need to get the `selectedRating` from our Star Rating View.

Open the `CreateReviewViewController.swift` file, and declare the following variable under `var thumbnail`:

```
var selectedRating:Rating = Rating.zero
var selectedRestaurantID:Int?
var imageFiltered: UIImage?
```

The first variable will represent the current `selectedRating` from the **Star Rating View**. We will use the second variable, the `restaurantID`, to save with a review, and we will use the final variable to save as the user's edited and saved review image.

Next, add a new method, called `showRating()`, after the `showApplyFilter()` method:

```
func showRating(with segue:UIStoryboardSegue) {
    guard let navController = segue.destination as?
UINavigationController,
        let viewController = navController.topViewController as?
StarRatingViewController else {
            return
    }

    viewController.selectedRating = selectedRating
}
```

This method will ensure that, if you select a rating and then change your mind, your last rating will appear in the `StarRatingViewController`.

Next, we want to update our `prepare()` method in order to check for when the `showRating` segue is called. Please add the following case statement inside of the `prepare()` method under the `applyFilter` case and before `default`:

```
case Segue.showRating.rawValue:
    showRating(with: segue)
```

Here is what the `prepare()` method should look like after this update:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

        switch segue.identifier! {
        case Segue.applyFilter.rawValue:
            showApplyFilter(with: segue)
        case Segue.showRating.rawValue:
            showRating(with: segue)
        default:
            print("Segue not added")
        }
    }
```

The last update we need is for when the user taps **Save**. We already created this `IBAction`; therefore, please add the following inside of the curly braces of `unwindRatingSave()`:

```
guard let viewController = segue.source as? StarRatingViewController
else { return }

selectedRating = viewController.selectedRating
btnRating.setImage(UIImage(named:Rating.image(rating: selectedRating.
value)), for: .normal)
```

We also need to update our `unwindFilterSave()`. Please replace the code that is in the curly braces with the following:

```
if let viewController = segue.source as? ApplyFilterViewController {
    if let thumbnail = viewController.imgExample.image {
        btnThumbnail.setImage(thumbnail, for: .normal)
        imageFiltered = generate(image: thumbnail, ratio:
CGFloat(102))
    }
}
```

Here, we are replacing the default thumbnail in the **Create Review View Controller** with the newly created thumbnail. In addition, we are setting the `imageFiltered` variable to the user's newly created image. This image will be what we save to Core Data.

Now that we have a rating and an image, we need to get the text name and the text from the review. Then, we will have everything we need to save reviews to Core Data.

After the `onPhotoTapped()` method in the `CreateReviewViewController.swift` file, add the following code:

```
@IBAction func onSaveTapped(_ sender: AnyObject) {

    var item = ReviewItem()
    item.name = tfName.text
    item.customerReview = tvReview.text
    item.restaurantID = selectedRestaurantID
    item.photo = imageFiltered
  item.rating = selectedRating.value

    let manager = CoreDataManager()
    manager.addReview(item)

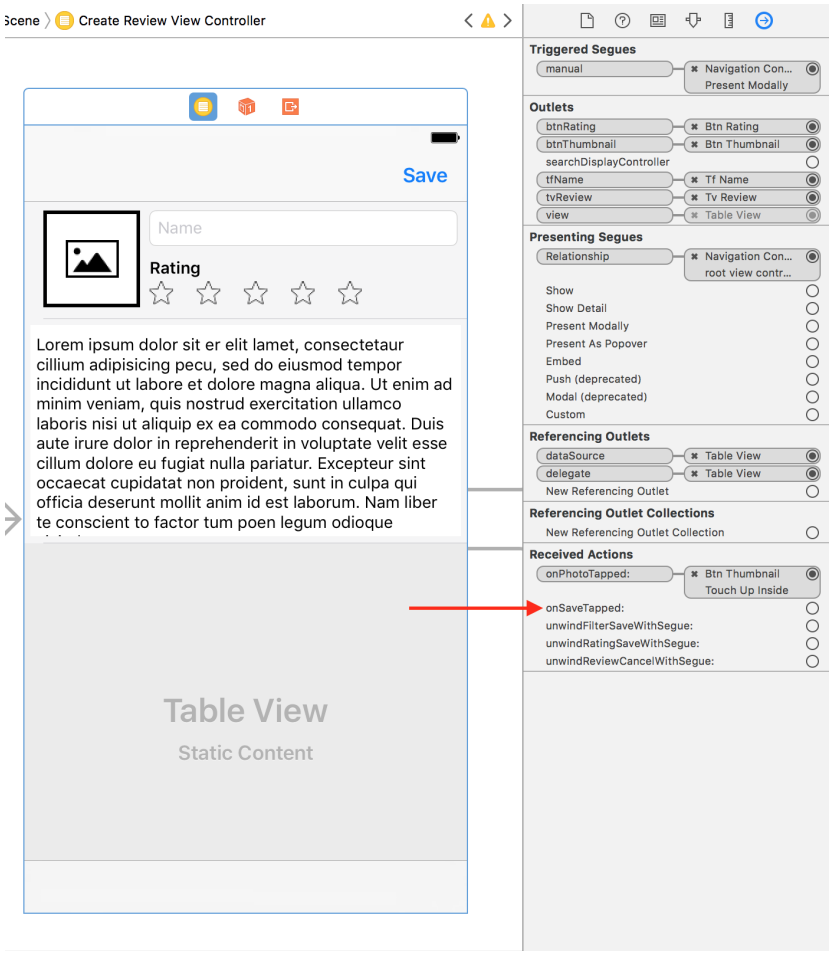
    _ = navigationController?.popViewController(animated: true)
}
```

When **Save** is tapped, we create an instance of the `ReviewItem` and then we assign all of our properties. Next, we create an instance of our Core Data Manager and use the `addReview()` method. This will run the code we wrote earlier and save the data into Core Data. Finally, we send the user back to the Restaurant Details.

We now need to add a **Save** button and connect it to our IBAction:

1. Open the `RestaurantDetail.storyboard` and select the **Create Review View** Controller. Then, in the **Object** library of the **Utilities** panel, type `navigationitem` into the filter field.
2. Drag a **Navigation** Item into the **Create Review View** Controller. Remove the word `Title` from the title bar by double-clicking it and deleting it.
3. Next, in **Object** library's filter area, type `bar button`.
4. Drag a **Bar** button item to the right side of the **Navigation Item**.
5. Now, select the **Bar** button item, open the **Attributes Inspector** in the **Utilities** panel, and update **System Item** to **Save**.
6. Next, in **Storyboard**, select the **View Controller** for the **Create Review** form again.

7. Then, in the **Utilities** panel, select the **Connections Inspector** and scroll down to **Received Actions**, where you will see the `onSaveTapped` method that we added earlier:



8. From the empty circle `onSaveTapped`, click and drag to the **Save** button in the **Navigation** item.

We have now finished connecting our **Save** button. The last thing we need to do before we can run the project is to set up our **Restaurant** section to update with the most recent restaurant review.

Let's add the following method after the `setupMap()` method inside the `RestaurantDetailViewController`:

```
func checkReviews() {
    if let id = selectedRestaurant?.restaurantID {
        manager.fetchReview(by: id)
    }

    let count = manager.numberOfItems()

    if count > 0 {
        noReviewsContainer.isHidden = true
    }

    let item = manager.getLatestReview()
    lblUser.text = item.name
    txtReview.text = item.customerReview
    if let rating = item.rating { imgRating.image = UIImage(named:
Rating.image(rating: rating)) }
}
```

In this method, we are checking first for the `selectedRestaurantID` and passing it to the Core Data Manager. If there are any restaurants found, then we will hide the **No Reviews** container. Since the Reviews container is behind the **No Reviews** container, we do not need to do anything else other than to assign the data.



You will see errors after having added this last method. Ignore these errors for now as we will fix them shortly.

Next, we need to call this method. If we put this method in the `viewDidLoad()` method, it would only be called once. Therefore, we need to add this method into a newly created `viewDidAppear()` method, since this gets called every time the view is shown. Add the following code after our `viewDidLoad()` method:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    checkReviews()
}
```

Then, we need to make sure that, when a user creates a review, we pass `restaurantID` to the **Create Review View Controller**. After the `selectedRestaurant` variable, please add the following:

```
let manager = ReviewDataManager()
```

Now, add this method after the `viewDidAppear()` method (which will remove the errors we noted earlier):

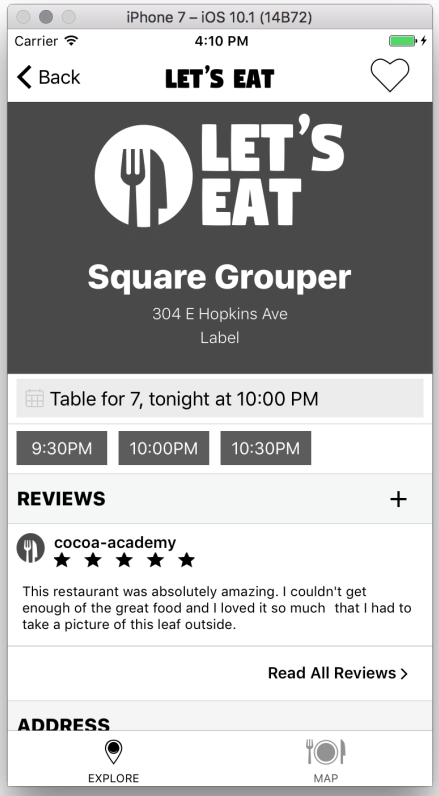
```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier {
        switch identifier {
        case Segue.showReview.rawValue:
            showReview(segue: segue)
        default:
            print("Segue not added")
        }
    }
}
```

Finally, add the following method after the `checkReviews` method, but before the last curly brace:

```
func showReview(segue:UIStoryboardSegue) {
    if let viewController = segue.destination as?
CreateReviewViewController {
        viewController.selectedRestaurantID = selectedRestaurant?.
restaurantID
    }
}
```

The `prepare()` method will check for the `showReview` identifier. If successful, it will call the `showReview()` method, which will take you to the Reviews list.

Let's build and run the project by hitting the **Play** button (or use *CMD* + *R*). You should now be able to create a review. After you save the review, you should be brought back to the **Restaurant Detail** view and see the review in the **Reviews** section:



We are missing one last thing. The review section only shows one review, and we want to show all of them. We need to create a class for this and set up a Table View. Let's start by adding this class now:

1. Create two new folders inside of the `Review List` folder, called `Controller` and `View`.
2. Now, right-click the `Controller` folder and select **New File**.
3. Inside of the **Choose a template for your new file screen**, select **iOS** at the top and then **Cocoa Touch Class**. Then, hit **Next**.

4. In the options screen that appears, add the following:
5. New File:

   Class: ReviewListViewController

   Subclass…: UIViewController

   Also create XIB: Unchecked

   Language: Swift

6. Hit **Next** and then **Create**.

When the file opens, replace everything with the following code:

```
import UIKit
import CoreData

class ReviewListViewController: UIViewController {

    @IBOutlet var tableView: UITableView!
    var selectedRestaurantID:Int?
    let manager = ReviewDataManager()

    override func viewDidLoad() {
        super.viewDidLoad()
        initialize()
    }

    func initialize() {
        if let id = selectedRestaurantID {
            manager.fetchReview(by: id)
            setupTableView()
        }
    }

    func setupTableView() {
        tableView.estimatedRowHeight = 128
        tableView.rowHeight = UITableViewAutomaticDimension
        tableView.tableFooterView = UIView()
    }

}
```

Next, in order for our Table View to work, we need to create our cell and an extension for this class:

1. Right-click the `View` folder in the `Review List` folder and select `New File`.
2. Inside of the **Choose a template for your new file screen**, select **iOS** at the top and then **Cocoa Touch Class**. Then, hit **Next**.
3. In the options screen that appears, add the following:
4. New File:

   Class: ReviewCell

   Subclass…: UITableViewCell

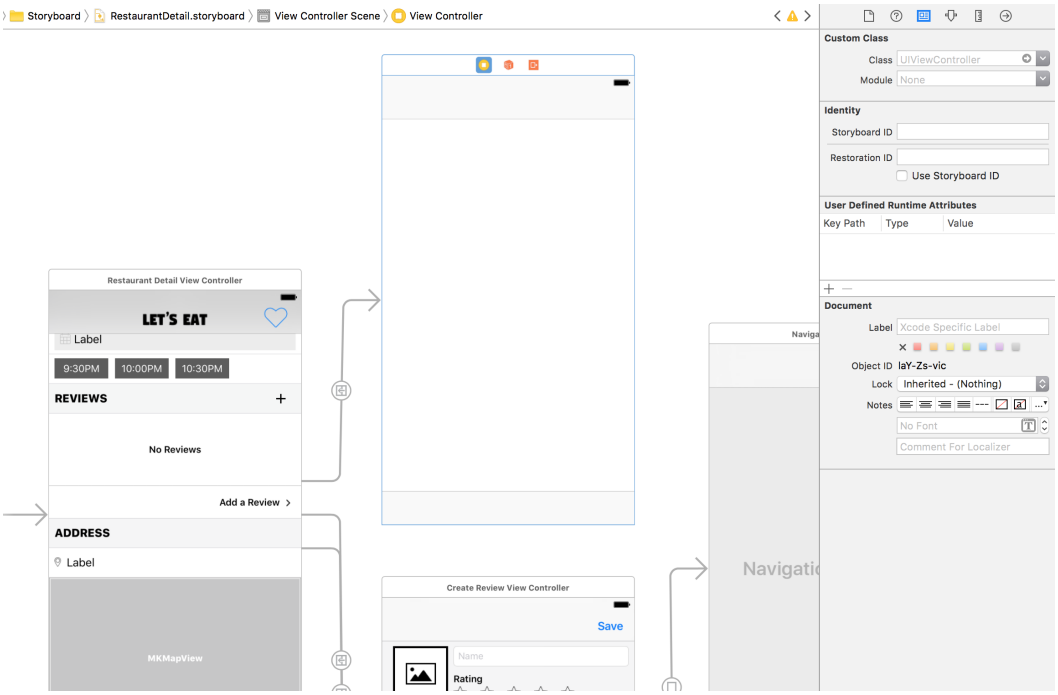
   Also create XIB: Unchecked

   Language: Swift

5. Click **Next** and then **Create**.
6. In this new file, above the `awakeFromNib()` method, add the following code:

```
@IBOutlet var imgReview: UIImageView!
@IBOutlet var imgRating: UIImageView!
@IBOutlet var lblUser: UILabel!
@IBOutlet var lblDate: UILabel!
@IBOutlet var lblReview: UILabel!
```

7. Save the file and open up the `RestaurantDetail.storyboard`.
8. Locate the **View Controller** that we have not yet set up and that is connected to the **Read all Reviews** button we previously created. This **Controller** is what we will use to display the reviews:



9. Next, select the segue that is connected from **Read all Reviews** to this empty **View Controller**, then select the **Attributes Inspector** in the **Utilities** panel and update **Identifier** to `showAllReviews`. Then, hit *Enter*.
10. Now, in the **Outline** view, select this empty **View Controller**.
11. Then, select the **Identity Inspector** in the **Utilities** panel, and under **Custom Class**, in the **Class** drop-down menu, type in `ReviewListViewController`, select it, and hit *Enter*.
12. Now, select the **Attributes Inspector** and uncheck both **Under Top** bars and **Under Bottom** bars in **Extend Edges**.
13. Next, in the **Object** library of the **Utilities** panel, type `tableview` into the filter field and drag it out into the **View** of this **Review List View Controller**.
14. With the **Table** View selected in the **Outline** View, select the Pin icon and enter the following values:
    - Under Add New Constraints:

      Top: 0

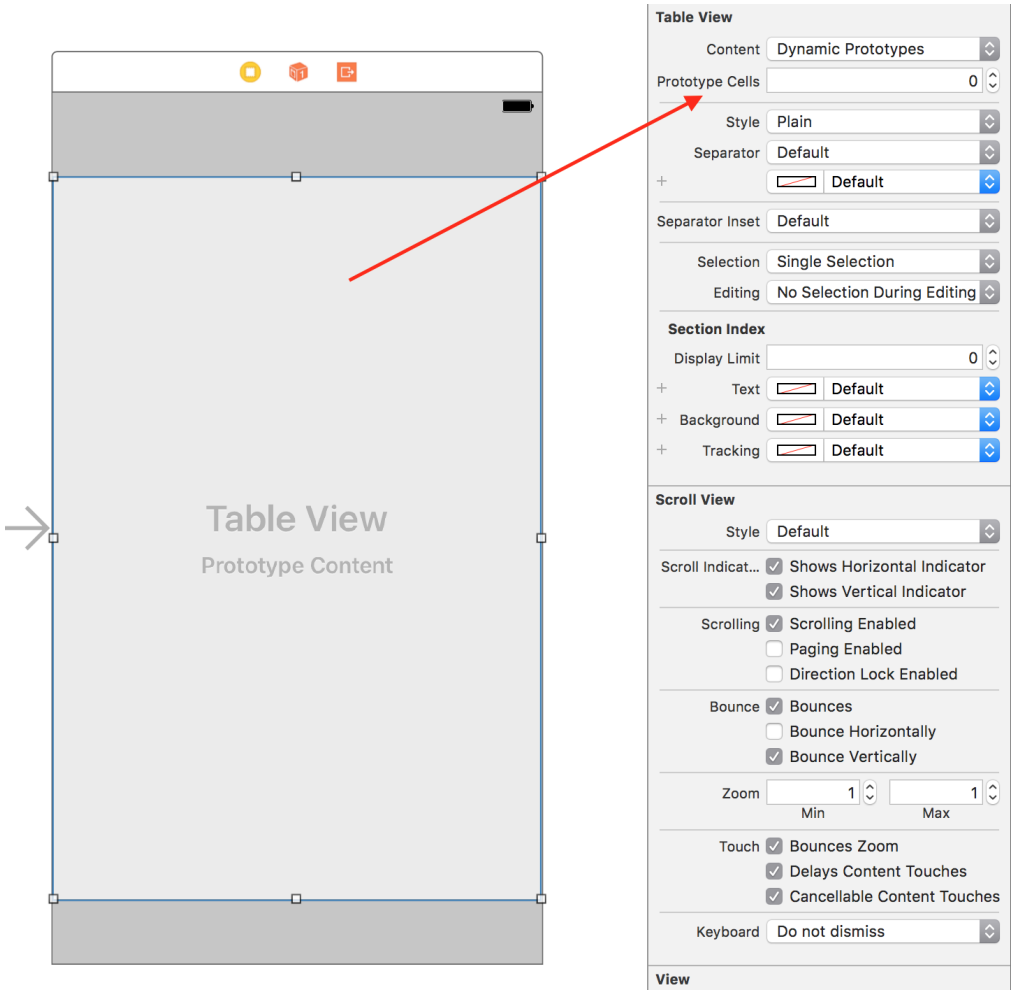
      Left: 0

Right: 0

Bottom: 0

- **Constrain to margins**: unchecked
- **Update Frames**: Items of New Constraints

15. Now, click **Add 4 Constraints**.
16. Select the **Attributes Inspector** again and change **Prototype Cells** under Table View from 0 to 1 and then hit *Enter*. You will now see a **Prototype Cell** inside of the **Table View**:



17. Then, select **Table View** in the **Outline** view and then the **Connections Inspector** in the **Utilities** panel.
18. Click and drag from the empty circle next to `tableView` to the **Table View** in the scene.
19. Next, select the **Table View** cell and, in the **Attributes Inspector**, set **Identifier** to `reviewCell` and hit *Enter*.
20. Now, select the **Identity Inspector** in the **Utilities** panel and, under **Custom Class**, update the **Class** to `ReviewCell` and hit *Enter*.
21. Finally, in the **Utilities** panel, select the **Size Inspector** and set the `reviewCell` **Row Height** to `128`. Then, hit *Enter*.

Setting up the cell UI

Now, let's get all of our UI elements into the `reviewCell`:

1. In the **Object** library of the **Utilities** panel, type image in the filter field.
2. Then, drag out two **Image Views** into the **View**.
3. Select one of the **Image Views** and, in the **Size Inspector**, update the following values:
    - **X**: 8
    - **Y**: 8
    - **Width**: 82
    - **Height**: 82
4. Select the other **Image View** and update the following values in the **Size Inspector**:
    - **X**: 98
    - **Y**: 53
    - **Width**: 124
    - **Height**: 13
5. Next, in the **Object** library, type `label` into the filter.
6. Drag out three labels into the **View**.
7. Select the first label and update the following values in the **Size Inspector**:
    - **X**: 98
    - **Y**: 8

- **Width**: 269
- **Height**: 21

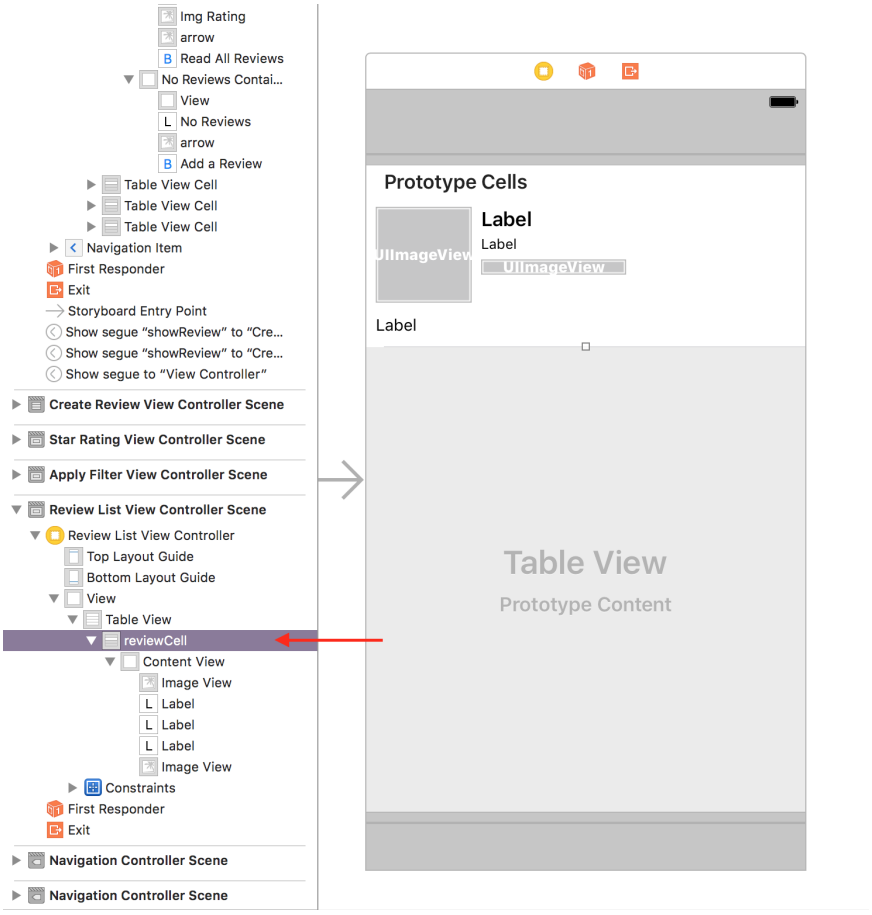
8. Then, select the **Attributes Inspector** in the **Utilities** panel and update the **Font Style** to **Semibold**.
9. Select the next label and update the following values in the **Size Inspector**:
   - **X**: 98
   - **Y**: 29
   - **Width**: 269
   - **Height**: 21
10. Then, in the **Attributes Inspector**, update the **Font** to size `12`.
11. Select the last label and update the following values in the Size Inspector:
    - **X**: 8
    - **Y**: 97
    - **Width**: 359
    - **Height**: 22
12. Now, in the **Attributes Inspector**, update the **Font** to size `14` and set **Lines** to 0.



Since a review can be long or short, we want our layout to adjust based on the length of the review. Setting the Lines value to 0 means that it will just adjust based on content. We still need to do some Auto Layout to get this to fully work. We will add Auto Layout shortly.

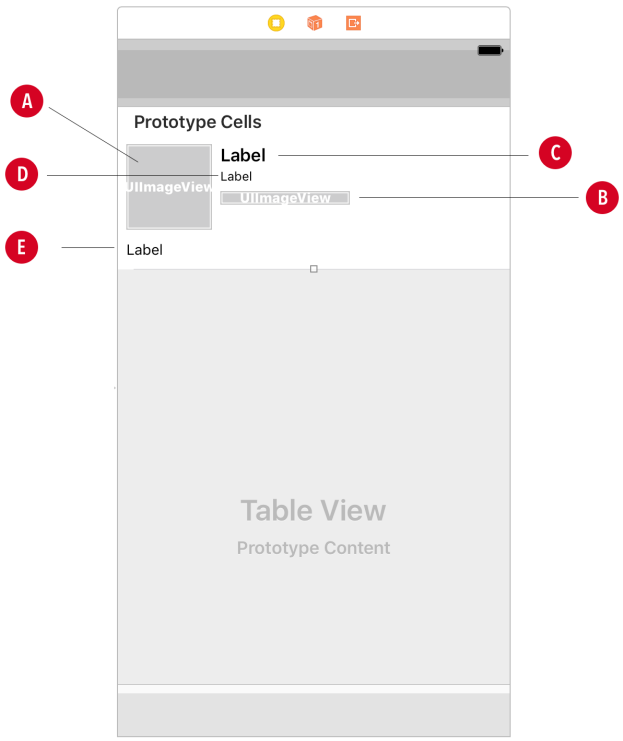
Now, we need to connect all of our UI elements:

1. In the **Outline** view, select **reviewCell**:



2. Select the **Connections Inspector** in the **Utilities** panel and click and drag the empty circles next to the following variables to their respective UI elements:
    - `imgReview` to the leftmost Image View (**A** in the following diagram)
    - `imgRating` to the rightmost Image View (**B** in the following diagram)
    - `lblUser` to the top Label (**C** in the following diagram)
    - `lblDate` to the middle Label (**D** in the following diagram)

- `lblReview` to the bottom Label (**E** in the following diagram)



Adding Auto Layout

Now that we have our elements in place, it is time to add Auto Layout:

1. In the **Outline** view, select the `imgReview` and then the Pin icon. Enter the following values:
    - Under Add New Constraints:

      Top: 8

      Left: 8
    - Constrain to margins: unchecked

      Height: 82

      Width: 82

2. Now, click **Add 4 Constraints**.
3. In the **Outline** view, select the `lblUser` and then the Pin icon. Enter the following values:
    - Under Add New Constraints:

      Top: 8

      Left: 8

      Right: 8
    - **Constrain to margins**: unchecked
    - **Height**: 21
4. Now, click **Add 4 Constraints**.
5. In the **Outline** view, select the `lblDate` and then the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 0

      Left: 8

      Right: 8
    - **Constrain to margins**: unchecked
    - **Height**: 21
6. Now, click **Add 4 Constraints**.
7. In the **Outline** view, select the **imgRating** and then the Pin icon. Enter the following values:
    - Under Add New Constraints:

      Top: 3

      Left: 8
    - **Constrain to margins**: unchecked
    - **Height**: 13
    - **Width**: 124
8. Now, click **Add 4 Constraints**.

9. Finally, in the **Outline** view, select the **lblReview** and then the Pin icon. Enter the following values:
    - Under Add New Constraints:

      Top: 7

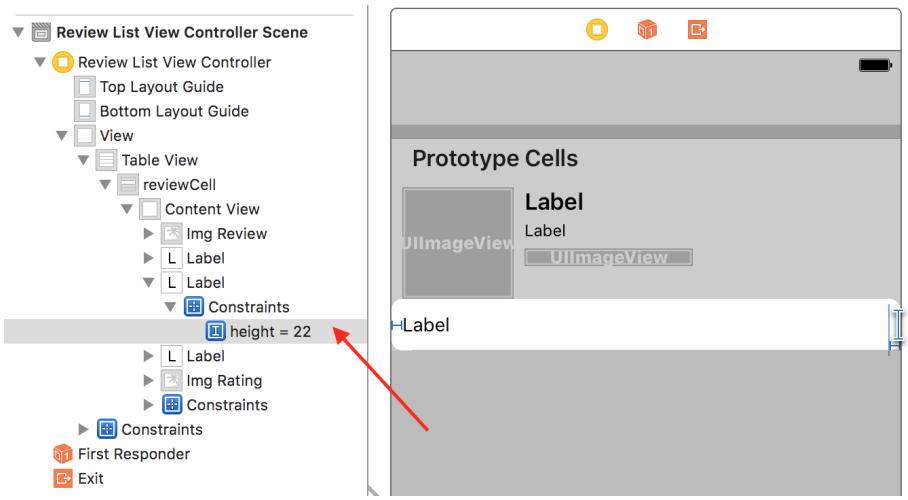
      Left: 8

      Right: 8

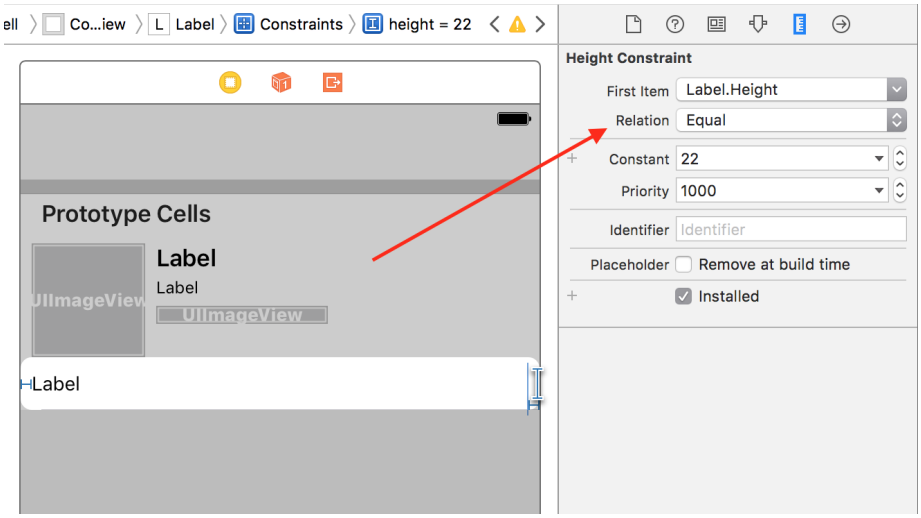
      Bottom: 8
    - **Constrain to margins**: unchecked
    - **Height**: 22
10. Now, click Add 5 Constraints.

We now have everything set up, but if we add a long review, nothing would happen to the label. We need to tell the label that we need the height to adjust:

1. In the Outline view, select **lblReview** and then the height constraint:



2. Select the **Size Inspector** in the **Utilities** panel and change **Relation** from **Equal** to **Greater Than** or **Equal**:



Now, with the height change and our bottom pin, all of our cells will have different heights based on the length of the review.

Adding Review List extension

Finally, we need to add our extension in order to display reviews in our Table View. Open the `ReviewListViewController` and go to the `setupTableView()` method and, before `tableView.estimatedRowHeight`, add the following:

```
tableView.dataSource = self
```

Now, after the last curly brace of the class, add the following:

```
extension ReviewListViewController: UITableViewDataSource {

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

    func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
        return manager.numberOfItems()
```

```
    }

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

        guard let cell = tableView.dequeueReusableCell(withIdent
ifier: "reviewCell", for: indexPath) as? ReviewCell else {return
UITableViewCell()}

        let review = manager.reviewItem(at:indexPath)
        cell.lblUser.text = review.name
        cell.lblReview.text = review.customerReview
  cell.lblDate.text = review.displayDate

        if let rating = review.rating { cell.imgRating.image =
UIImage(named: Rating.image(rating: rating)) }
        if let photo = review.photo {
            cell.imgReview.image = photo
        }

        return cell
    }
}
```

In this extension, we are setting up the Table View data source methods as we have done before. The `numberOfSections()` is set to 1 and the `numberOfRows()` is set to the number of reviews that are saved in Core Data. Finally, `cellForRowAt()` is used to pass data to our cell.

We added a segue identifier earlier, so we need to add this new `showAllReviews` to our Segue class. Open the `Segue.swift` file and add the following case:

```
case showAllReviews
```

Next, in our `RestaurantDetailViewController`, we need to pass our `selectedRestaurantID` over to the `ReviewListViewController`, just like we did with `CreateReviewViewController`. Please open the `RestaurantDetailViewController.swift` file add the following method after the `showReview()` method:

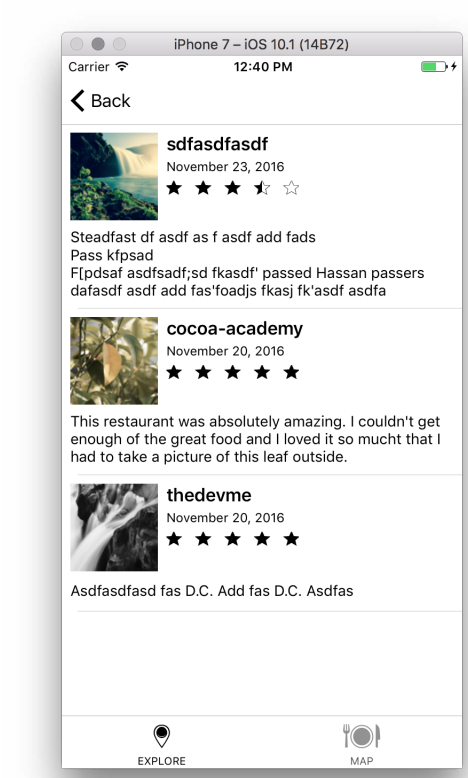
```
func showAllReviews(segue:UIStoryboardSegue) {
    if let viewController = segue.destination as?
ReviewListViewController {

        viewController.selectedRestaurantID = selectedRestaurant?.
restaurantID
    }
}
```

Finally, inside of the `prepare()` method in the `RestaurantDetailViewController.swift` file, please add the following case statement:

```
case Segue.showAllReviews.rawValue:
    showAllReviews(segue: segue)
```

Build and run your project, and you will see that you can now see all of the reviews in one list:



We covered a lot in this chapter, and we've now finished building our main app's major functionality.

Summary

Our app is really starting to take shape. We were able to create a Core Data Model and can now save reviews to the Core Data. We also are able to display all the reviews for a restaurant or pull out the last review and display it.

In the next chapter, we will work on putting the final touches to our app in order to make it more of a universal app. Once we do that, our main app will be finished and then we can focus on adding some cool features, such as an **iMessages** app, notifications, and 3D Touch.

14
Universal

We spent most of this entire book focusing on logic for our app and getting the app to work on the iPhone. We did not pay much attention to the app working on iPad or other devices. During this chapter, we will look at the app on an iPad as well as update the app on all iPhone devices. You will be surprised at how much is already working and that only minor changes will need to be made to get our app to look the way we want.

We will cover the following topics in this chapter:

- Updating our app to be supported on all devices
- Learning about multitasking and how to code for it

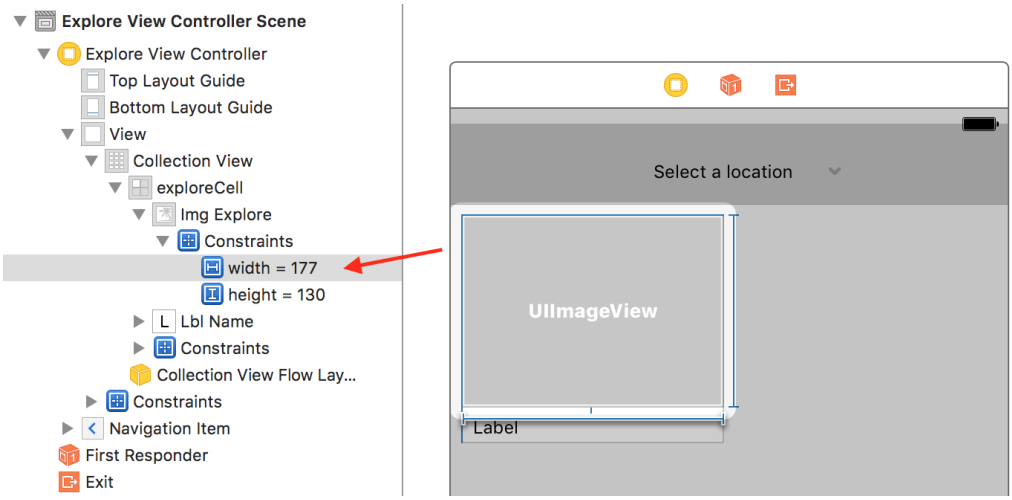
Explore

Let's switch our device to any iPad (other than iPad Air as it does not support all of the iOS 10 features) and build and run the project by hitting the **play** button (or use *CMD + R*). You will notice a lot of layout issues. Currently, we have set up values that really only work for one device, but we need this to work on all devices.

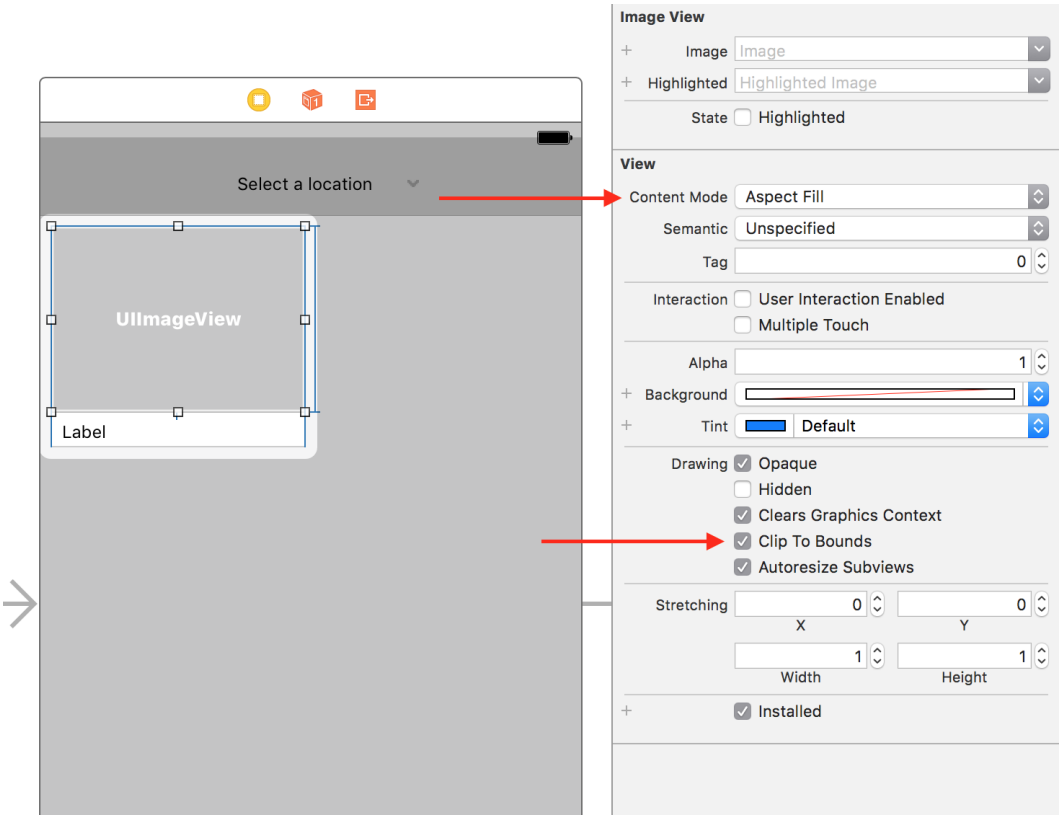
Let's start with our `Explore.storyboard`. First, we need to update some Auto Layout for our Explore cells. Right now, we have a width set up for our Image that needs to be more dynamic:

1. Open up the `Explore.storyboard` and go to your `exploreCell`.

2. If you have a width constraint for your **Img Explore**, select that constraint and delete it by hitting the *Delete* key:



3. Next, with **Img Explore** selected, select the Pin icon and set the right constraint to `0`.
4. Click on **Add 1 Constraint**.
5. Then, in the **Utilities** panel, select the **Attributes Inspector** and change the **Content Mode** under the **View** section to **Aspect Fill**. This will keep the images from looking stretched, while still filling the entire area.
6. Then, in **Drawing** in the **Attributes Inspector**, check **Clip To Bounds**.



These are the only updates we need to make to our Explore cell.

Next, we are going to create a file that will let us know which device is being used. We can then use this to set up different looks depending on the device. Let's create this file:

1. Right-click on the `Misc` folder inside the `Common` folder and select **New File**.
2. Inside the **Choose a template for your new file screen**, select **iOS** at the top and **Swift File**. Then, hit **Next**.
3. Name this file `Device` and, then, hit **Create**.

First, we need to update our import statement from `import Foundation` to `import UIKit`.

Next, add the following under the import statement:

```
struct Device {

    static var currentDevice: UIDevice {
        struct Singleton {
            static let device = UIDevice.current
        }
        return Singleton.device
    }

    static var isPhone: Bool {
        return currentDevice.userInterfaceIdiom == .phone
    }

    static var isPad: Bool {
        return currentDevice.userInterfaceIdiom == .pad
    }
}
```

Our new struct will now tell us whether we are on an iPad or an iPhone. Having a file like this is good, because it allows you to avoid having to write the same code. To implement this code, all we would need to do is add a snippet of code like the following:

```
if Device.isPhone { }
```

This statement will make our code more readable; if we need to add any more checks for particular devices, we can do it all in the same file. One more great use of putting code like this into its own file is that, when you build the next app, you can just add this file to your project and continue.

Next, let's open the `ExploreViewController.swift` file and make some more updates to our code. We need to create a variable that we will use for the spacing we want between items. Add the following before our `viewDidLoad()` method:

```
fileprivate let minItemSpacing: CGFloat = 7
```

Now, we need to create a function to set up some default Collection View values. We also need to create an `initialize()` method to call our setup function. Add the following two methods after the `prepare()` method:

```
func initialize() {
    manager.fetch()
    if Device.isPad { setupCollectionView() }
```

```
}

func setupCollectionView() {
    let flow = UICollectionViewFlowLayout()

    flow.sectionInset = UIEdgeInsets(top: 7, left: 7, bottom: 7,
right: 7)
    flow.minimumInteritemSpacing = 0
    flow.minimumLineSpacing = 7
    collectionView?.collectionViewLayout = flow
}
```

Now, we need to add some updates to our `ExploreViewController` extension. First, update our extension declaration to the following:

```
extension ExploreViewController: UICollectionViewDataSource,
UICollectionViewDelegateFlowLayout
```

Adding the `UICollectionViewDelegateFlowLayout` allows us to update our cell item size in code.

Next, add the following method in the extension before the last curly brace:

```
func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout,
    sizeForItemAt indexPath: IndexPath) -> CGSize {  A
B   if Device.isPad {
        let factor = traitCollection.horizontalSizeClass == .compact ? 2 : 3

        let screenRect = collectionView.frame.size.width
        let screenWidth = screenRect - (CGFloat(minItemSpacing) * CGFloat(factor + 1))
        let cellWidth = screenWidth / CGFloat(factor)

        return CGSize(width: cellWidth, height: 154)
    }
    else {
        let screenRect = collectionView.frame.size.width
        let screenWidth = screenRect - 21
        let cellWidth = screenWidth / 2.0

        return CGSize(width: cellWidth, height: 154)
    }
}
```

Let's discuss each part of the method we just added:

- **Part A:**

  ```
  func collectionView(_ collectionView: UICollectionView, layout
  collectionViewLayout: UICollectionViewLayout, sizeForItemAt
  indexPath: IndexPath) -> CGSize
  ```

  This method is used for setting the size of the cell.

- **Part B:**

  ```
  if Device.isPad
  ```

  Here, we use the struct we created. We check to see if we are using an iPad or an iPhone.

- **Part C:**

  ```
  let factor = traitCollection.horizontalSizeClass == .compact ? 2:3
  let screenRect = collectionView.frame.size.width
  let screenWidth = screenRect - (CGFloat(minItemSpacing) *
  CGFloat(factor + 1))
  let cellWidth = screenWidth / CGFloat(factor)
  return CGSize(width: cellWidth, height: 154)
  ```

  In the first part of the code, we check whether the screen is compact or not. If the screen is compact, then we need a two-column grid, otherwise, we need a three-column grid. In addition, we are distributing our items evenly across the width of the screen.

- **Part D:**

  ```
  let screenRect = collectionView.frame.size.width
  let screenWidth = screenRect - 21
  let cellWidth = screenWidth / 2.0

  return CGSize(width: cellWidth, height: 154)
  ```

  Here, we just set up a two-column grid on all phones. We get the screen width, subtract 21, and then divide the result by 2 in order to distribute the cells evenly.

If you run the project, everything will look fine, except when you attempt to rotate the device using CMD + right arrow or CMD + left arrow. Then, you will see that our layout spacing does not update.

In order to fix this, we need to make one more update. At the bottom of your `ExploreViewController` extension before the last curly brace, add the following:

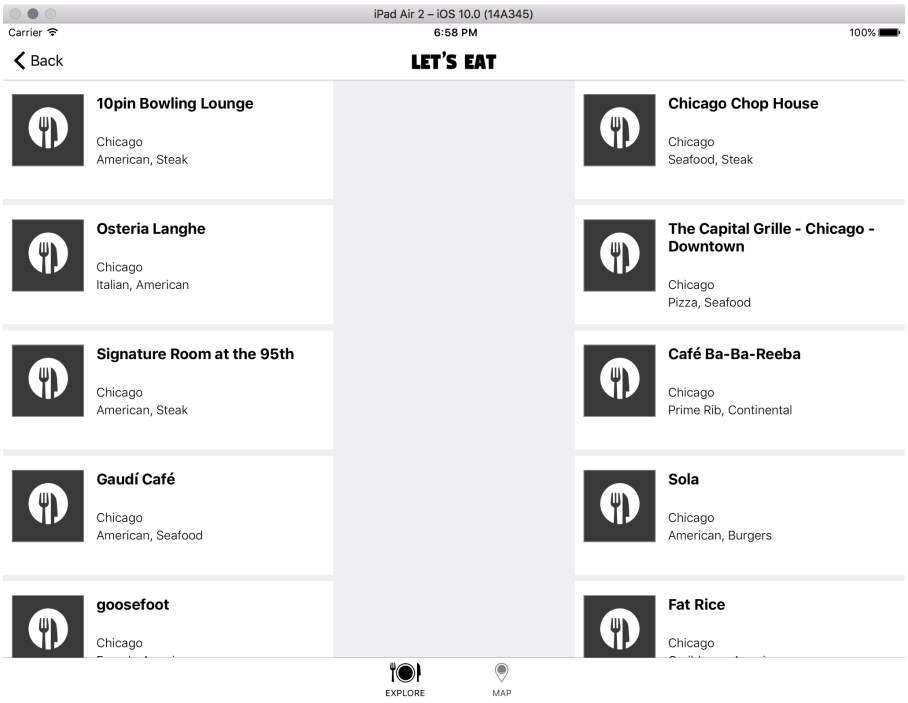
```
override func viewWillTransition(to size: CGSize, with coordinator:
UIViewControllerTransitionCoordinator) {
    collectionView.reloadData()
}
```

Now, build and run your project again by hitting the play button (or use *CMD + R*) and rotate the device. You will see that our layout spacing now updates.

Next, we will direct our attention to the restaurant listing page and get into more detail on iPad and multitasking.

Restaurant listing

For our restaurant listing page, we want a one-column grid on all phones and a two-column grid on all iPads. If you build and run the project by hitting the play button (or use *CMD + R*) and go to a restaurant listing page, you will see that we need to fix the spacing on the iPad to show two-columns correctly:



Let's see how we can fix this.

Open the `RestaurantListViewController.swift` file and add the following under the `prepare()` method:

```
func initialize() {
    if Device.isPad { setupCollectionView() }
}
```



You will get an error for the `setupCollectionView()` method. Ignore it for now as we will fix it shortly.

This method checks if the device is an iPad; if it is, it calls the `setupCollectionView()` method.

Next, add the following under the `initialize()` method we just added:

```
func setupCollectionView() {
    let flow = UICollectionViewFlowLayout()

    flow.sectionInset = UIEdgeInsets(top: 7, left: 7, bottom: 7,
right: 7)
    flow.minimumInteritemSpacing = 0
    flow.minimumLineSpacing = 7

    collectionView?.collectionViewLayout = flow
}
```

The preceding method is actually the same thing as we previously added in Storyboard regarding spacing between items, but, here, we are implementing it programmatically.

We have a couple of more things that we need to address. First, we are going to programmatically have the size of the screen calculated for us. Inside the `RestaurantListViewController` extension, add the following code after `cellForItemAt()`, but before the last curly brace:

```
func collectionView(_ collectionView: UICollectionView, layout
collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath:
IndexPath) -> CGSize {
    if Device.isPhone {
     let cellWidth = collectionView.frame.size.width
     return CGSize(width: cellWidth, height: 135)
}
else {
     let screenRect = collectionView.frame.size.width
     let screenWidth = screenRect - 21
     let cellWidth = screenWidth / 2.0

     return CGSize(width: cellWidth, height: 135)
    }
}
```

This code states that if the device is an iPhone, a one-column grid will be shown, else for an iPad, a two-column grid.

Next, we need to update our extension declaration to the following:

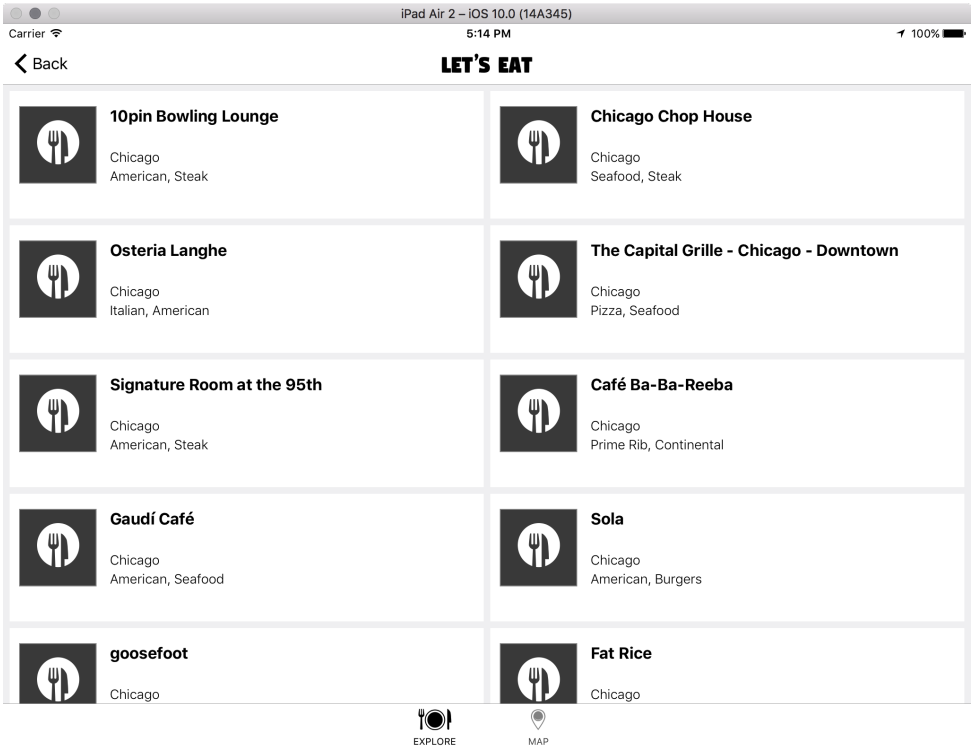
```
extension RestaurantListViewController: UICollectionViewDataSource,
UICollectionViewDelegateFlowLayout
```

Then, we need to add `initialize()` into our `viewDidLoad()` method. Therefore, our `viewDidLoad()` method should now look as follows:

```
override func viewDidLoad() {
    super.viewDidLoad()

    initialize()
}
```

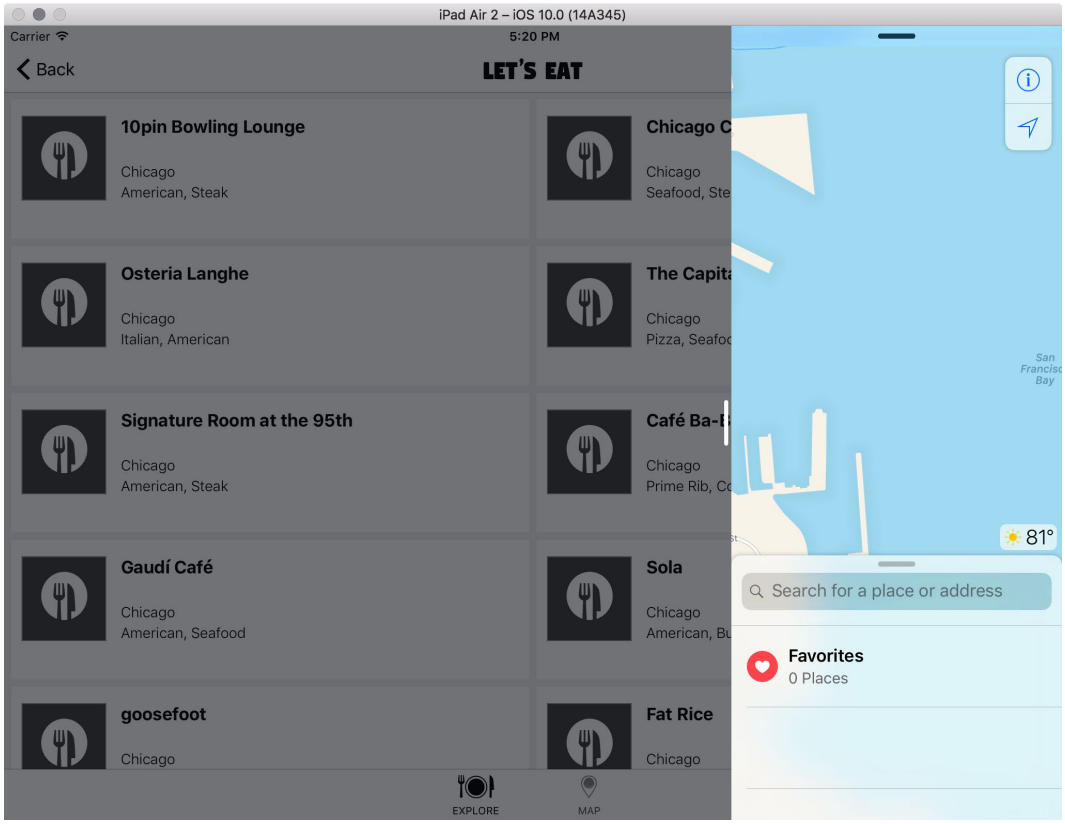
Let's build and run the project for the iPad by hitting the play button (or use *CMD* + *R*):

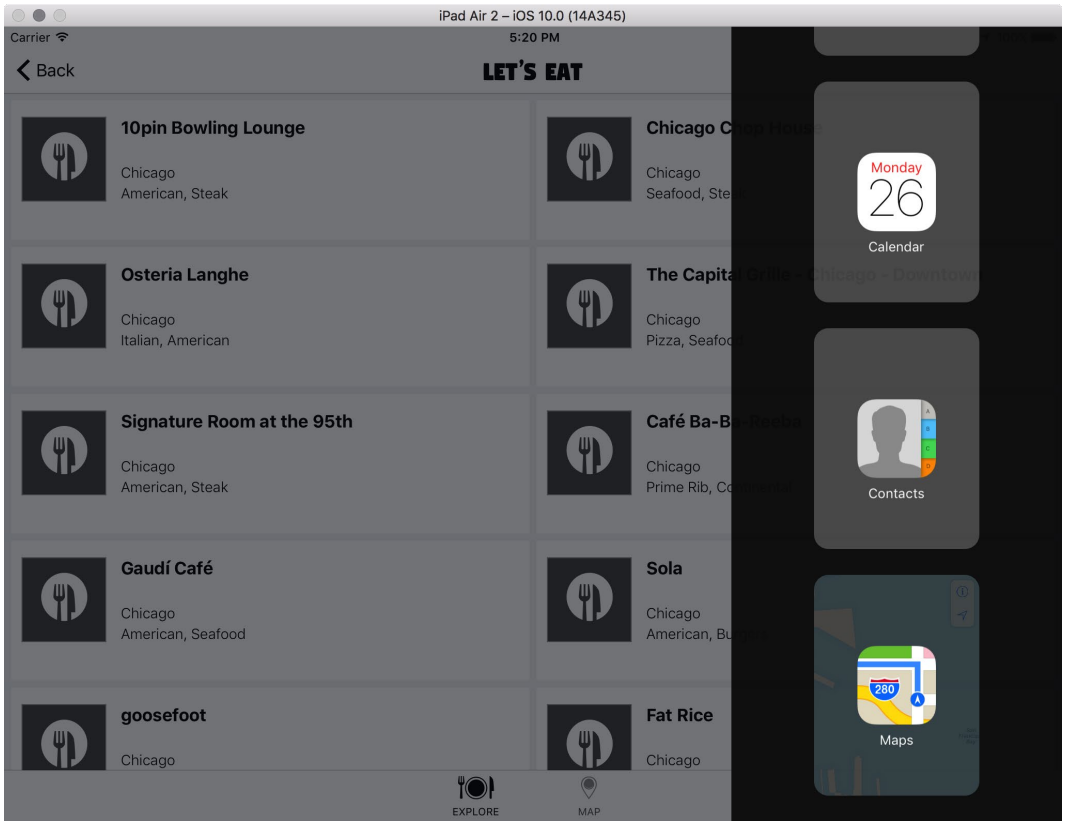


The two-column grid is what we want for the iPad for our restaurant listing page, but we need to verify that we did not change the one-column grid on the iPhone. Switch the device back to any iPhone and, after building and running the project again, you should still see a one-column grid on the iPhone.

There are still issues with the iPad setup. Switch back to the iPad and build and run the project again by hitting the play button (or use *CMD + R*). When the project launches, hit CMD + right arrow to rotate the device. Then, return to your restaurant listing page. The first issue is that if you rotate the device, the cell spacing does not update. Another issue is that, if you multitask, your app may be resized and you need to make sure that your layout adjusts accordingly.

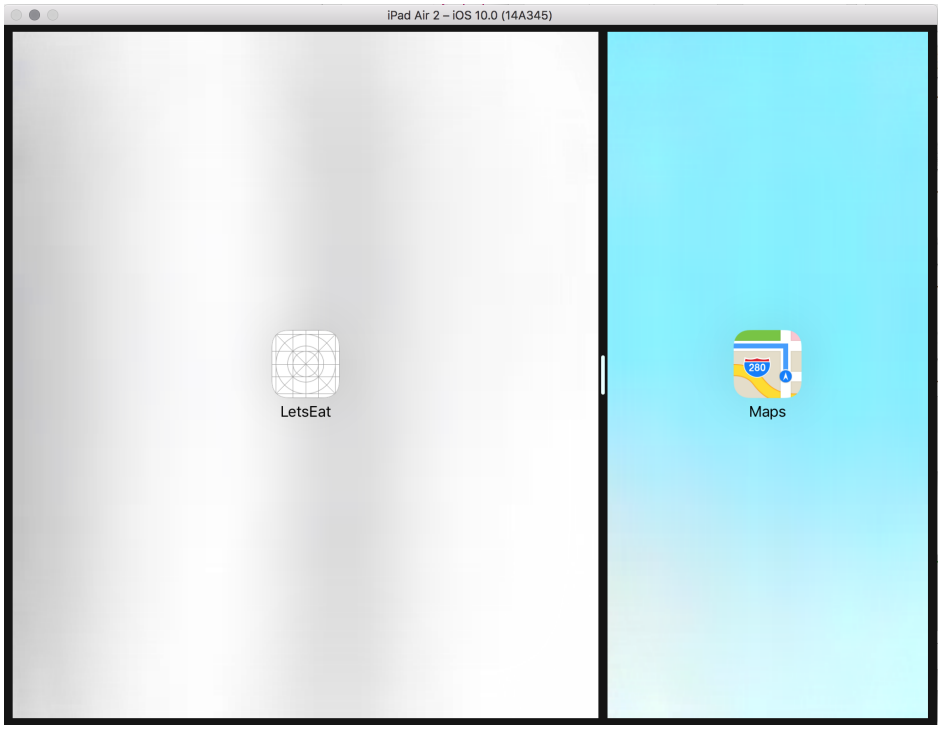
To see multitasking, swipe from right to left on the right side of the device or in the simulator, and you will see one of the two following screens:





If you are seeing the second picture, then just select any app (I chose the Maps app) to have your screen look similar to the first picture above.

Grab the small tab that will be in the middle of this new screen and pull it to the left. This will split the screen with our app and allow you to take over more (or less if you move it back to the right) of the screen:



When you move the tab to the left, you will note that the *Let's Eat* app does not resize the cells and instead shows a column. Our app needs to adjust no matter the size it has available when splitting the screen between our app and another app.

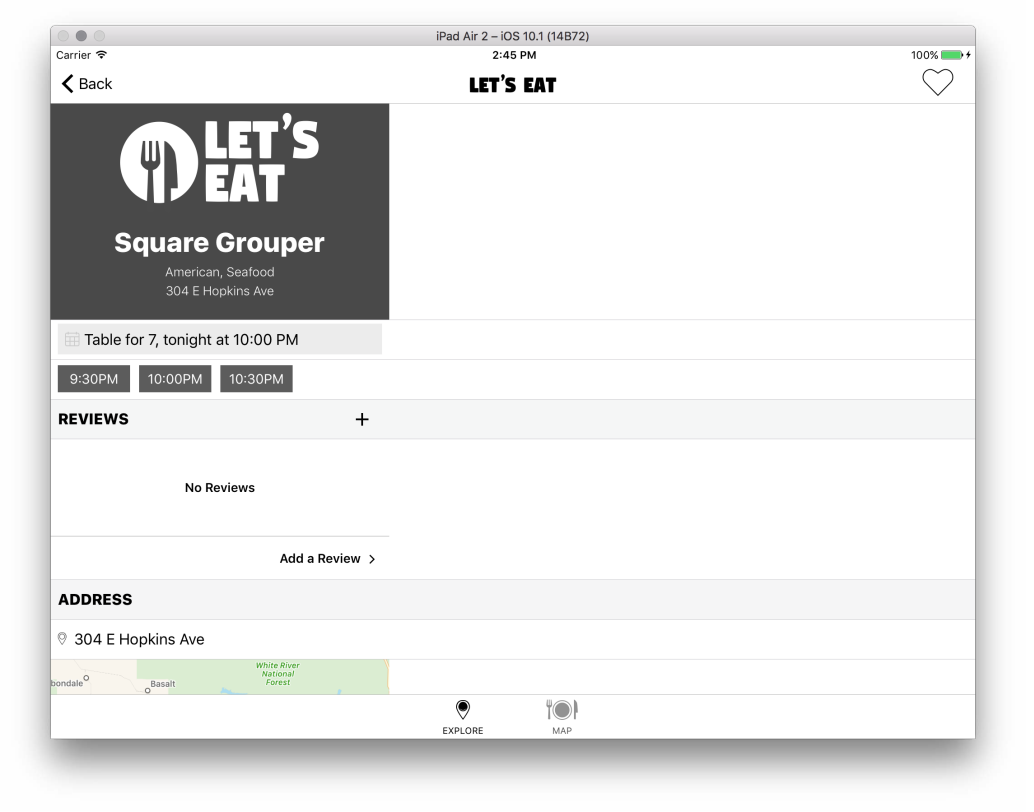
Fixing these problems is actually a really simple fix. In your `RestaurantListViewController` extension, before the last curly brace, add the following code:

```
override func viewWillTransition(to size: CGSize, with coordinator:
UIViewControllerTransitionCoordinator) {
    collectionView.reloadData()
}
```

Build and run the project again by hitting the play button (or use *CMD* + *R*) and rotate the device using *CMD* + right arrow. You will now see that, every time you update the size of the restaurant listing page, the grid updates as well to fit the new size. Let's now move onto the restaurant detail page.

Updating restaurant details

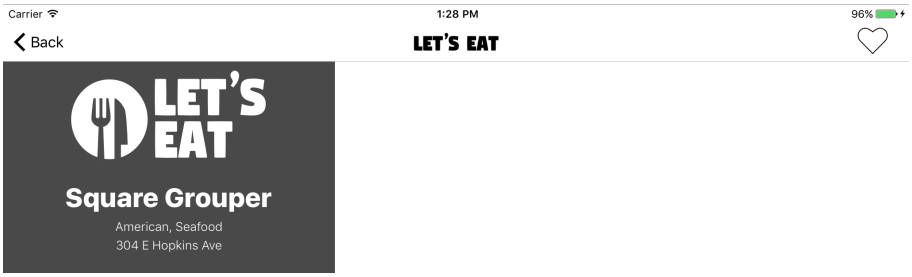
If you click on a restaurant and go to a restaurant detail page, you should see something similar to the following screenshot:



We have some Auto Layout elements that are missing. Let's fix these elements so that they appear correctly on an iPad.

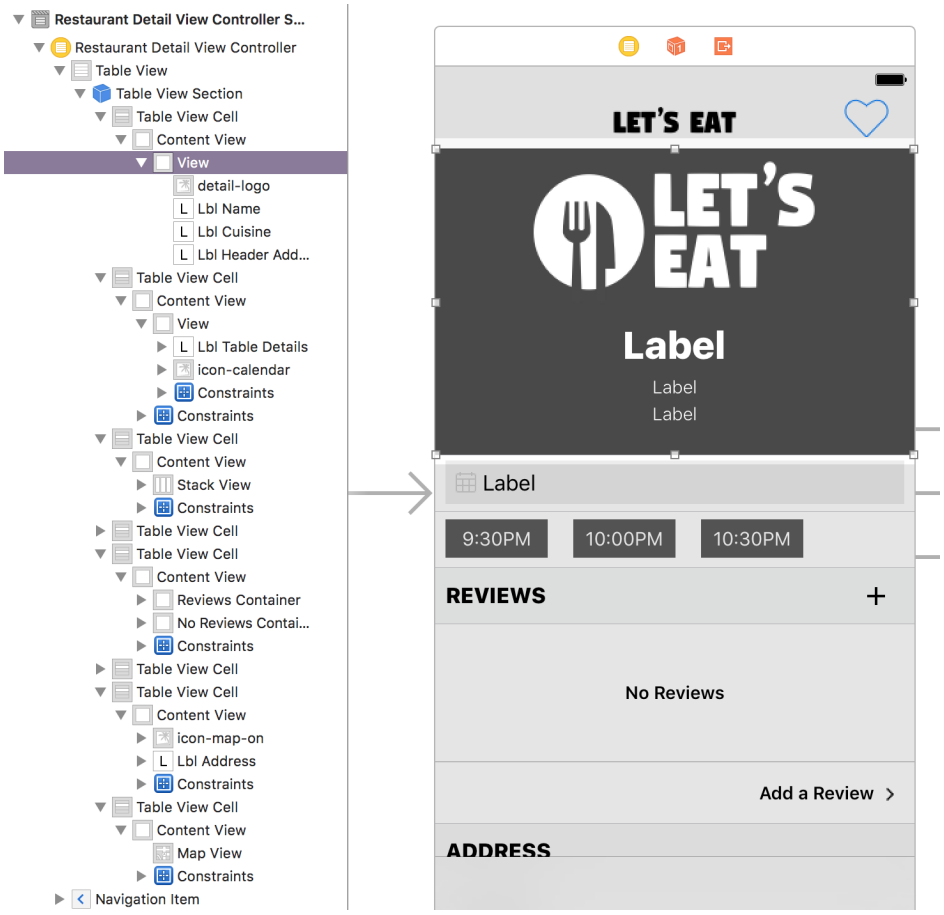
Updating the header layout

First, we need to fix the header information section:



We want a gray background and the Labels to expand across the screen and the logo to be in the middle:

1. Open `RestaurantDetail.storyboard` and, in the `RestaurantDetailViewController`, select the View container that contains the header information:



2. Select the Pin icon and enter the following values:
    - All values under **Add New Constraints** are set to **0**

      Constrain to margins: unchecked

      Update Frames: Items of New Constraints

3. Click on **Add 4 Constraints**.
4. Next, select the logo in the **View** container and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: `10`

      Constrain to margins: unchecked

Width: `220` (checked)

Height: `112` (checked)

5. Click on **Add 3 Constraints**.
6. Then, click on the **Align** icon to the left of the Pin icon and check Horizontally in **Container**.
7. Next, select the first Label (`lblName`) in the **View** container and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: `10`

      Left: `8`

      Right: `8`

      Constrain to margins: `unchecked`

      Height: `44 (checked)`
8. Click on **Add 4 Constraints**.
9. Next, select the middle Label (`lblCuisine`) in the **View** container and, then, the Pin icon. Enter the following values:
    - Under "Add New Constraints":

      Top: `0`

      Left: `8`

      Right: `8`

      Constrain to margins: `unchecked`

      Height: `21 (checked)`
10. Click on **Add 4 Constraints**.
11. Next, select the third and last Label (`lblHeaderAddress`) in the **View** container and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: `0`

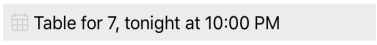
      Left: `8`

      Right: `8`
    - **Constrain to margins**: `unchecked`
    - **Height**: `21 (checked)`

12. Click on **Add 4 Constraints**.
13. We have now completed the header section and need to address the table details section.

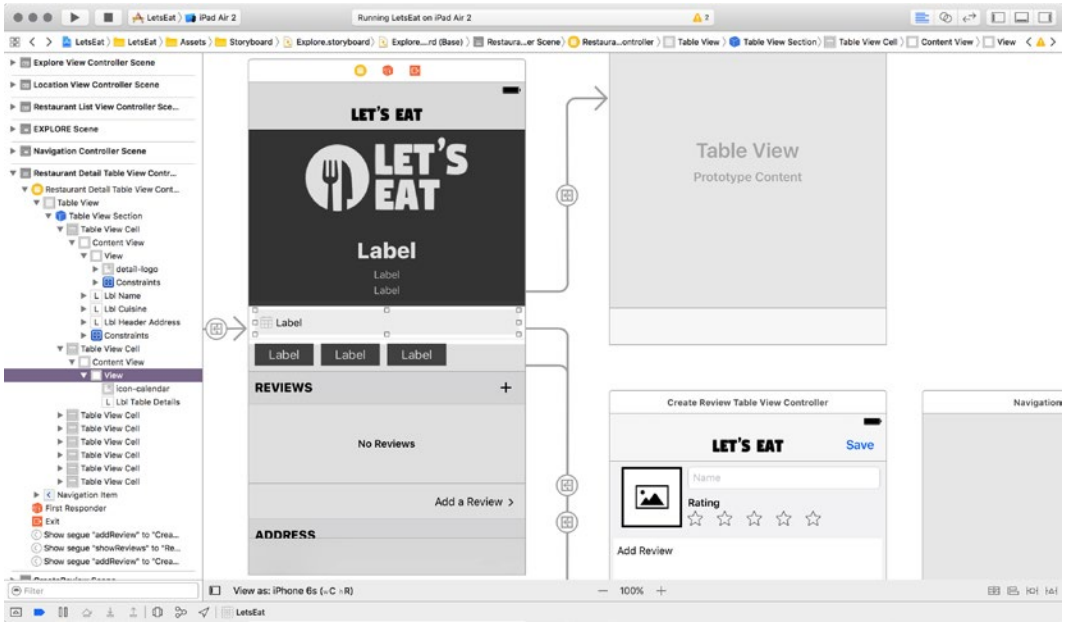
Updating the table details section layout

Our table details section looks as follows:



We just need to update this section so that it expands all the way across the screen.

1. Select the **View** container that holds all of the elements in this section, `lblTableDetails` and `icon-calendar`:



2. Select the Pin icon and enter the following values:
    - Under **Add New Constraints**:

      Top: `4`

      Left: `8`

      Right: `8`

- Constrain to margins: `unchecked`
- **Height**: `34 (checked)`

3. Click on **Add 4 Constraints**.
4. Next, select the icon-calendar inside the **View** container and, then, the Pin icon. Enter the following values:
   - Under **Add New Constraints**:

     Top: `9`

     Left: `8`
   - **Constrain to margins**: `unchecked`
   - **Width**: `16 (checked)`
   - **Height**: `16 (checked)`
5. Click on **Add 4 Constraints**.
6. Finally, select the Label (`lblTableDetails`) inside the View container and, then, the Pin icon. Enter the following values:
   - Under **Add New Constraints**:

     Top: `7`

     Left: `5`

     Right: `8`
   - **Constrain to margins**: `unchecked`
   - **Height**: `21 (checked)`
7. Now, click on **Add 4 Constraints**.

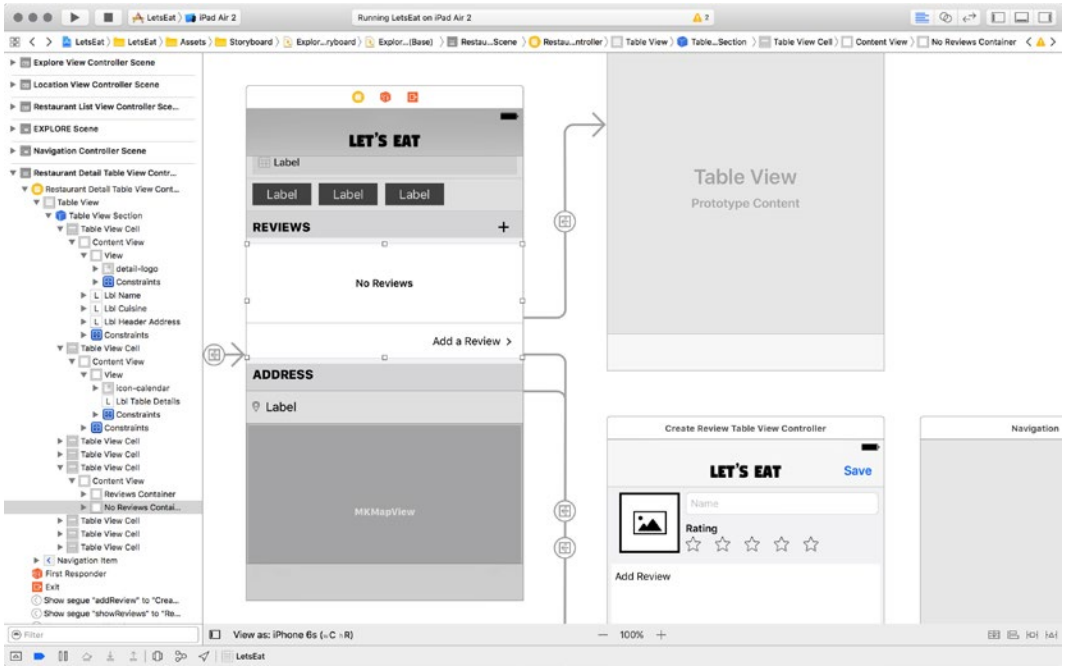
Let's build and run the project by hitting the play button (or use *CMD* + *R*). You should now see the header section and table details section filling across the entire screen.

Next, let's update the **Review** section, including both the **No Reviews** View and **Reviews** View.

Updating the No Reviews Layout

First, we need to fix the No Reviews View. In order to get to the Reviews, you might have to scroll inside the Table View. This might be difficult at first, but you will get more comfortable quickly, especially if you treat it like you are scrolling a web page. Let's start updating our No Reviews View:

1. Select the **No Reviews** container in the `RestaurantDetailViewController`:



2. Select the Pin icon and enter the following values:
    - All values under "**Add New Constraints**" are set to 0

      Constrain to margins: unchecked

      Click on **Add 4 Constraints**

    - Next, open the disclosure arrow of the **No Reviews** container, and, then, select the **Label** that says **No Reviews**.
    - Select the Pin icon and enter the following values:
    - Under "Add New Constraints":

      Top: 45

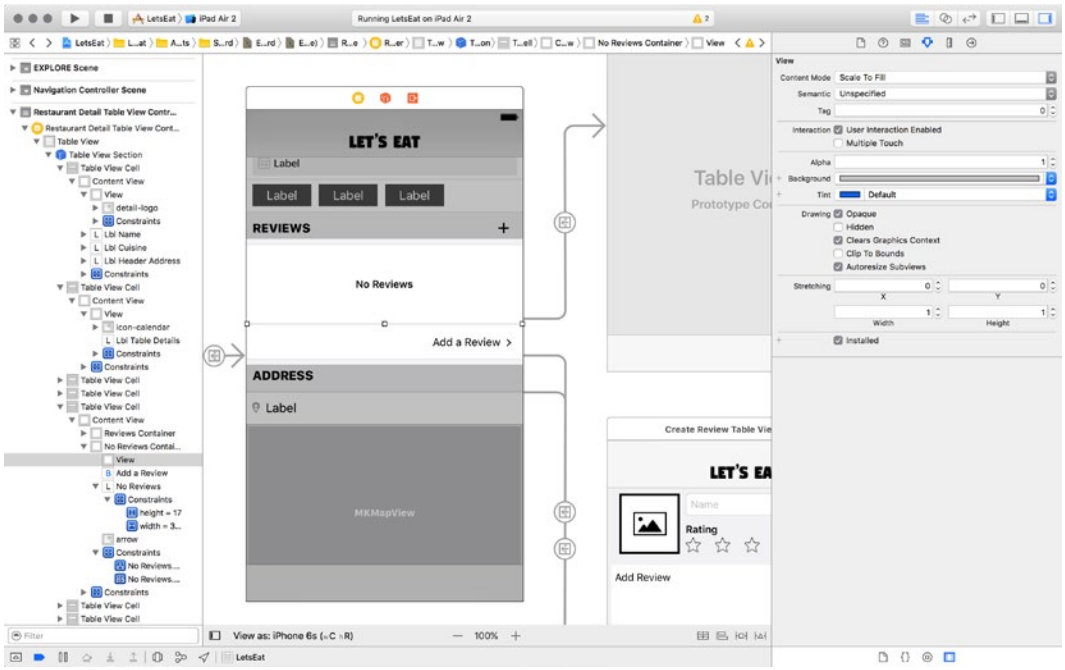
      Constrain to margins: unchecked

      Width: 326 (checked)

      Height: 17 (checked)

3. Click on **Add 3 Constraints**.

4. Next, select the Align icon to the left of the Pin icon and check **Horizontally** in **Container**.
5. Click on **Add 1 Constraint**.
6. Select the gray line separator between the **No Reviews** Label and the **Add a Review** Button in the scene (or you can select the **View** underneath the **No Reviews** container in the **Outline** view):



7. Select the Pin icon and enter the following values:
    - Under **Add New Constraints**:

      Top: 45

      Left: 0

      Right: 0
    - **Constrain to margins**: unchecked
    - **Height**: 1 (checked)

8. Click on **Add 4 Constraints**.
9. Next, select the **Add a Review** Button in the **No Reviews** container and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 9

      Right: 12
    - **Constrain to margins**: unchecked
    - **Width**: 129 (checked)
    - **Height**: 30 (checked)
10. Click on **Add 4 Constraints**.
11. Finally, select the arrow in the **No Reviews** container and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 20

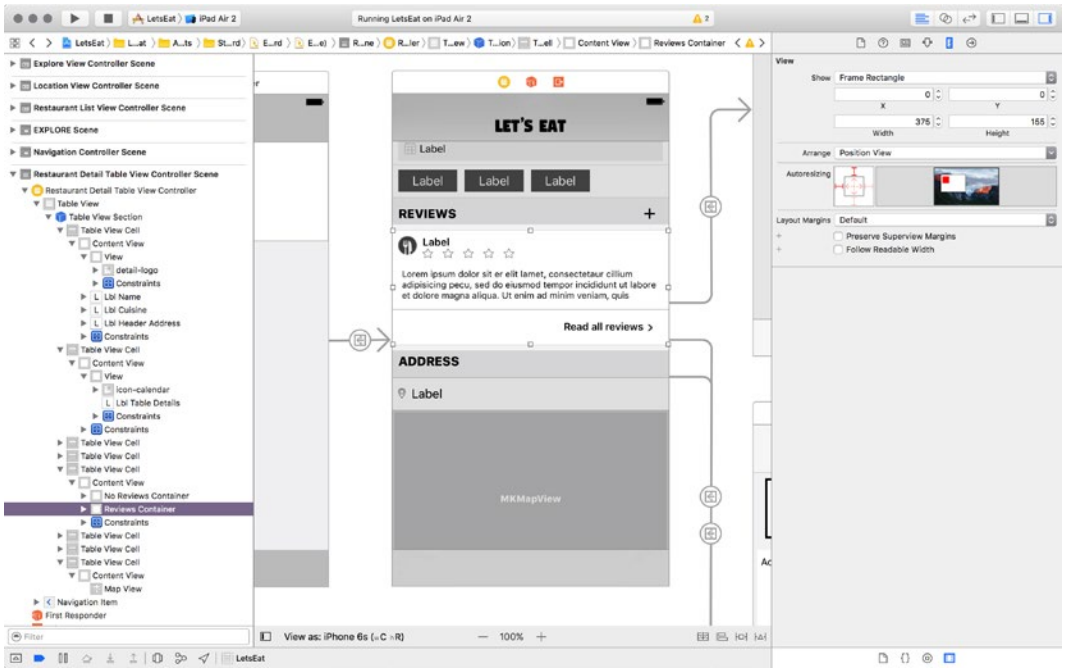
      Right: 16
    - **Constrain to margins**: unchecked
    - **Width**: 6 (checked)
    - **Height**: 11 (checked)
12. Click **Add 4 Constraints**.

Let's build and run the project by hitting the play button (or use *CMD* + *R*). You should see that the No Reviews section now covers the entire area for that section on the iPad.

Updating the Reviews layout

Next, we need to update the layout for the Reviews View:

1. Select the **No Reviews** container and move it above the **Reviews** container in the **Outline** view. This will allow you to see the elements in the **Reviews** container in the scene. Then, select the **Reviews** container:



2. Select the Pin icon and enter the following values:
    - All values under **Add New Constraints** are set to 0
    - **Constrain to margins**: unchecked
3. Click on **Add 4 Constraints**.
4. Next, open the disclosure arrow of the **Reviews** container and, then, select the **Image**, `icon-thumb-review`.
5. Select the **Pin** icon and enter the following values:
    - Under "Add New Constraints":

      Top: 8

      Left: 8

- **Constrain to margins**: unchecked
- **Width**: 25 (checked)
- **Height**: 30 (checked)

6. Click on **Add 4 Constraints**.
7. Next, select the **Label** that says **User in the Reviews** container and, then, the Pin icon. Enter the following values:
   - Under "Add New Constraints":

     Top: 8

     Left: 8

     Right: 8

   - **Constrain to margins**: unchecked
   - **Height**: 16 (checked)
8. Click on **Add 4 Constraints**.
9. Next, select the Image that says **Img Rating** (our **0star** Image) in the **Reviews** container and, then, the Pin icon. Enter the following values:
   - Under **Add New Constraints**:

     Top: 1

     Left: 8

   - **Constrain to margins**: unchecked
   - **Width**: 124 (checked)
   - **Height**: 13 (checked)
10. Click on **Add 4 Constraints**.
11. Next, select **Txt Review** (the **Text View**) in the **Reviews** container and, then, the Pin icon. Enter the following values:
    - Under "Add New Constraints":

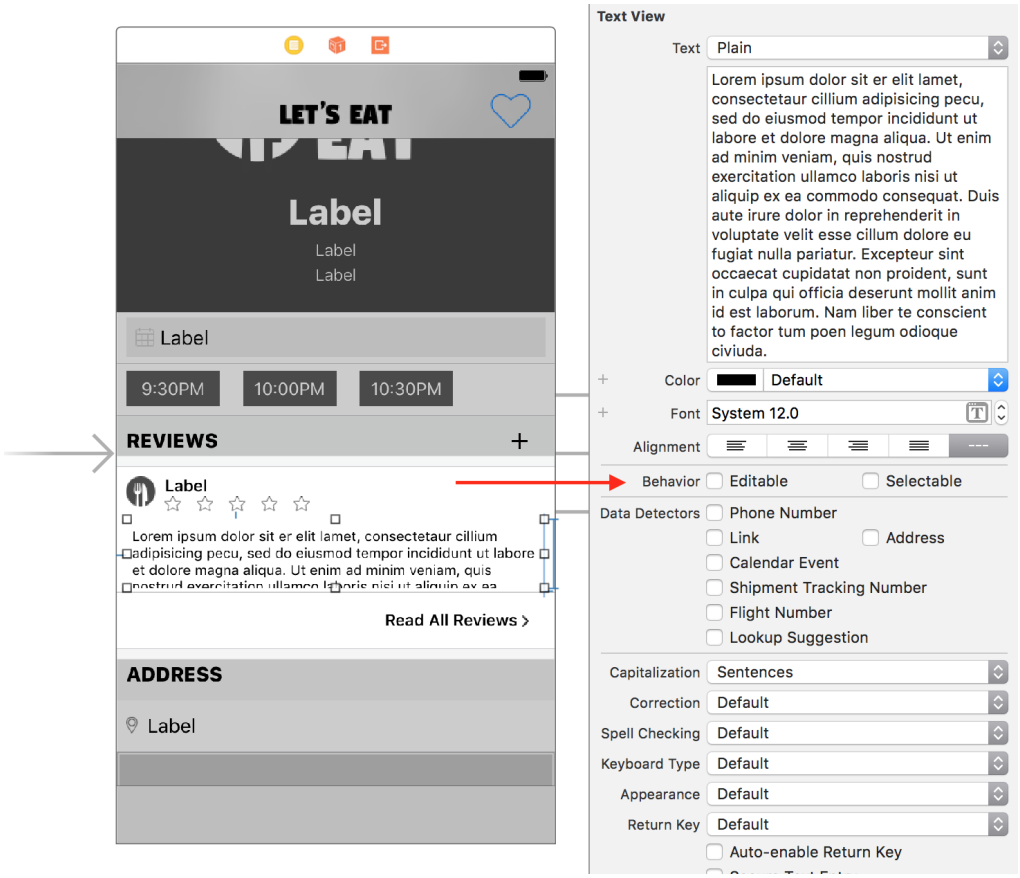
      Top: 6

      Left: 8

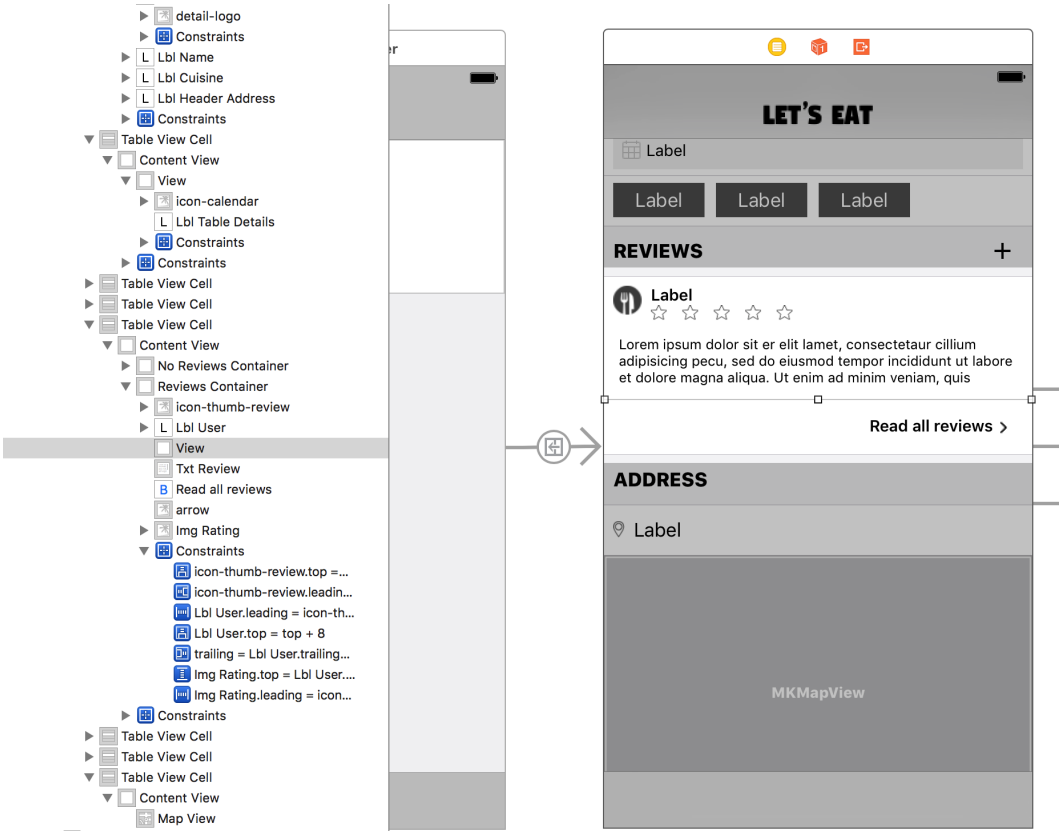
      Right: 8

    - **Constrain to margins**: unchecked
    - **Height**: 60 (checked)
12. Click on **Add 4 Constraints**.

13. With **Txt Review** still selected, open the **Attributes Inspector** in the **Utilities** panel and verify that **Editable** and **Selectable** under **Behavior** are unchecked; if checked, uncheck them:



14. Select the gray line separator between the **Txt Review** and the **Read All Reviews** button in the scene (or you can select the **View** underneath the **Reviews** container in the **Outline** view):



15. Select the Pin icon and enter the following values:
    - Under "Add New Constraints":

      Top: 3

      Left: 0

      Right: 0
    - **Constrain to margins**: unchecked
    - **Height**: 1 (checked)
16. Click on **Add 4 Constraints**.

17. Next, select the **Read All Reviews** Button in the **Reviews** container and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 8

      Right: 18
    - **Constrain to margins**: unchecked
    - **Width**: 141 (checked)
    - **Height**: 30 (checked)
18. Click on **Add 4 Constraints**.
19. Finally, select the arrow in the **Reviews** container and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: 19

      Right: 22
    - **Constrain to margins**: unchecked
    - **Width**: 6 (checked)
    - **Height**: 11 (checked)
20. Click on **Add 4 Constraints**.

Remember to drag your **No Reviews** container in the Outline view back below your Reviews container. It is easiest to do this when both of the containers' disclosure arrows are closed.

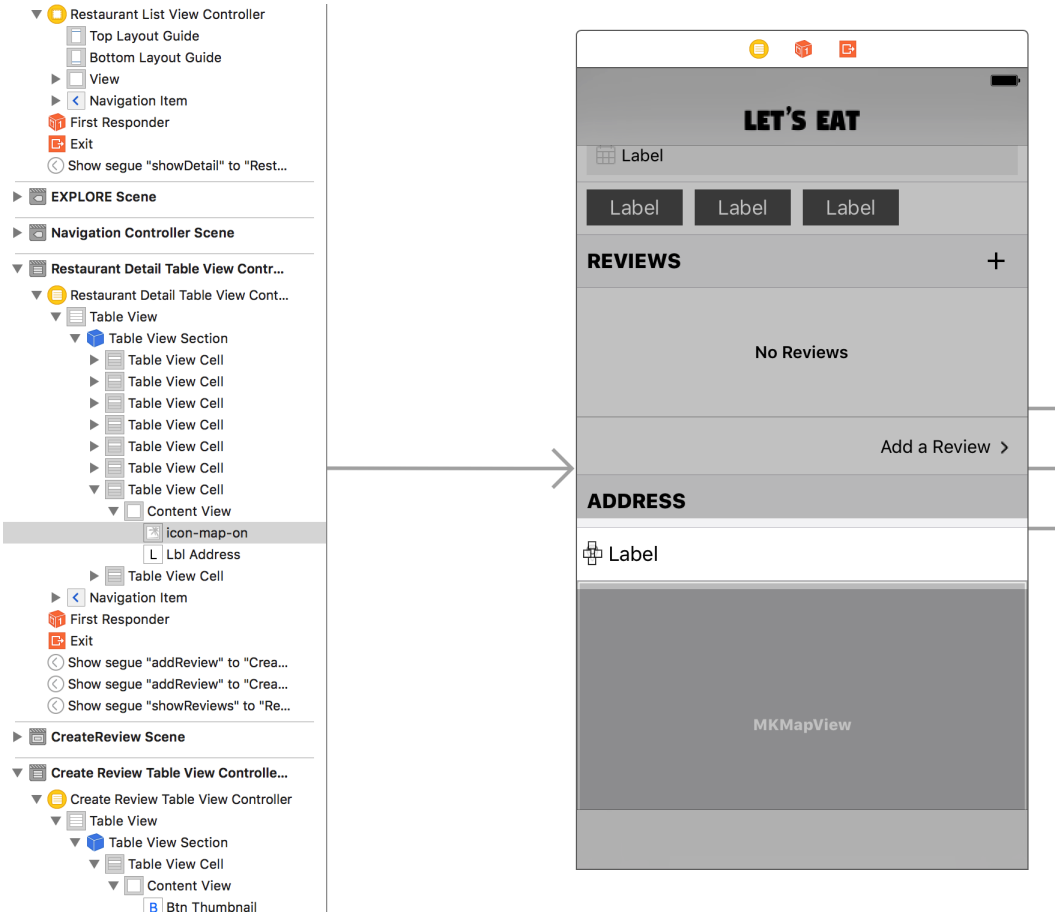
Let's build and run our project again by hitting the play button (or use *CMD* + *R*) to verify that our Reviews section fills the entire screen of the iPad.

We have now completed updating the No Reviews and Reviews sections. Lastly, we need to update the Map section in our `RestaurantDetail.storyboard`. Although you cannot tell visually when you run the project, we need to add some Auto Layout to our Map Label and icon.

Updating the Map section layout

Let's make some updates to our Map section:

1. Select the **Image**, `icon-map-on`:

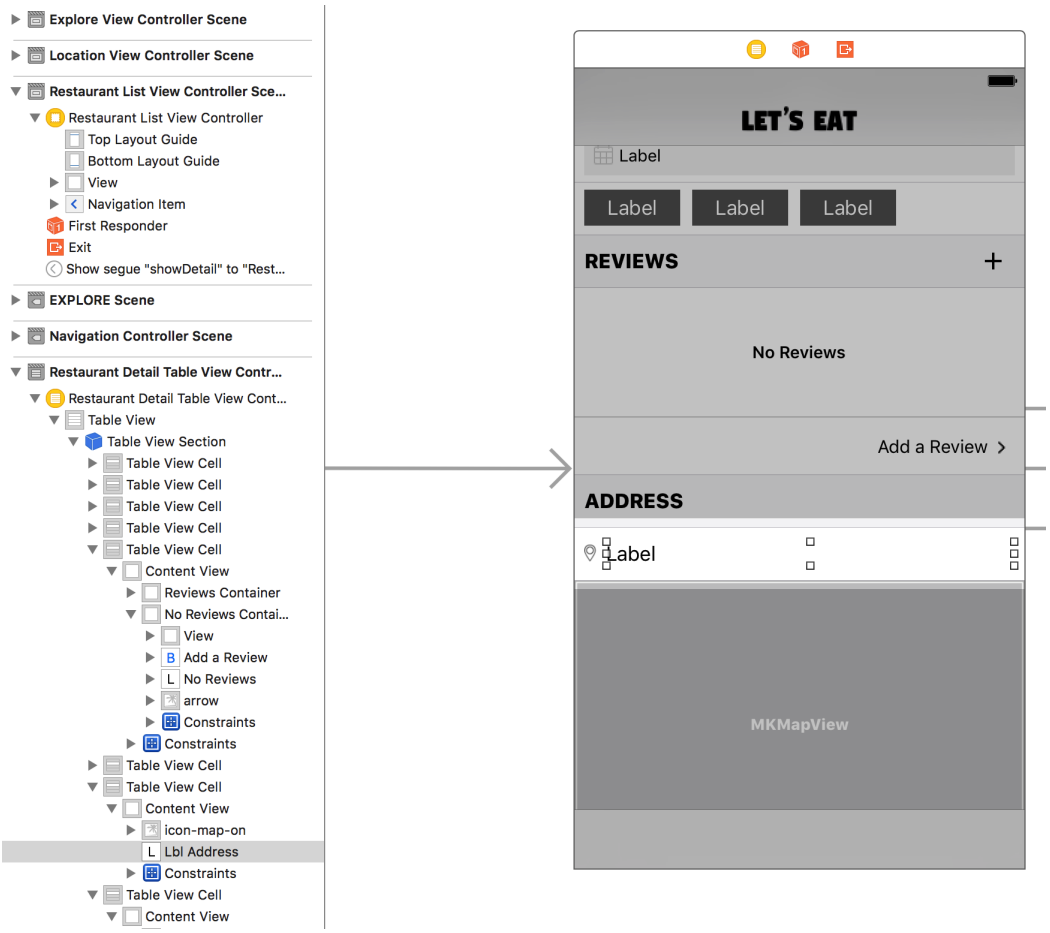


2. Select the Pin icon and enter the following values:
    - Under "Add New Constraints":

      Top: 14

      Left: 8
    - **Constrain to margins**: unchecked
    - **Width**: 10 (checked)
    - **Height**: 14 (checked)
3. Click on **Add 4 Constraints**.

4. Next, select the Label (`lblAddress`):



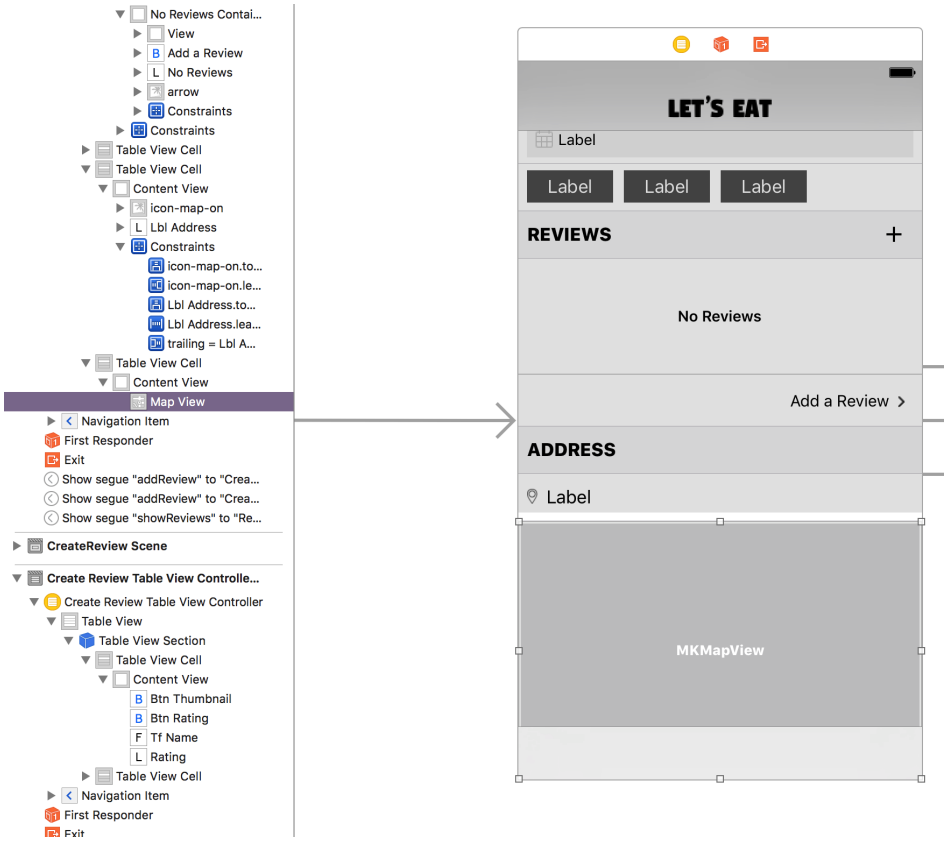
5. Select the Pin icon and enter the following values:
    - Under **Add New Constraints**:

      Top: 11

      Left: 8

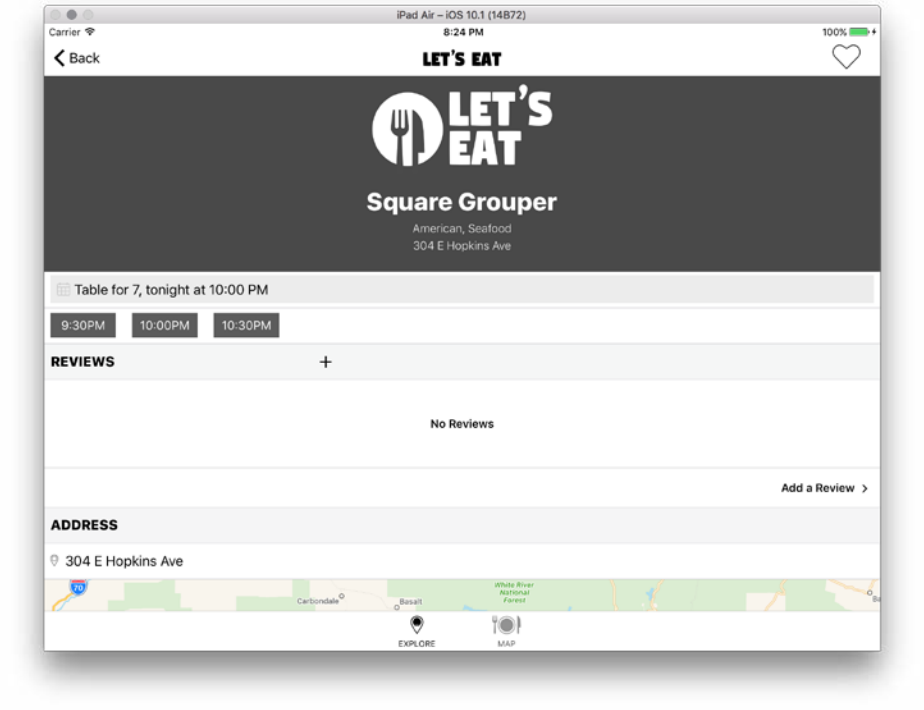
      Right: 9
    - **Constrain to margins**: unchecked
    - **Height**: 21 (checked)

6. Click on **Add 4 Constraints**.
7. Finally, select the **Map View**:



8. Select the Pin icon and enter the following values:
    - All values under **Add New Constraints** are set to **0**
    - **Constrain to margins**: unchecked
    - **Update Frames**: Items of New Constraints
9. Click on **Add 4 Constraints**.

We finished updating our app to correct any layout issues we might encounter on an iPad. Our restaurant detail now fills the screen for the iPad:



You can try other size devices, and you should see the same on all of the screens.

Summary

You now have an app that fully functions on all devices. You can see how using the Collection View gives your app some variety on different devices with very little code. As you get more and more comfortable, you will find other ways to make your app look unique on different devices.

We could submit the app as it is right now and it would be perfectly fine, but why not take advantage of some of the features that you can implement. In the next chapter, we will do just that by creating an iMessages app for our app.

15
iMessages

Text messaging started with just simple text and the creation of faces using special characters. As smartphones started to become more and more commonplace, so did text messaging. Messages are now a major form of communication for a large majority of people. People find it easier to respond to a text message versus answering a phone call.

When Apple announced the iMessages App and Stickers, it took messaging to another level. We had stickers before this announcement, but now we had a fully integrated system. iMessages not only allow you to send a sticker to express a feeling or an emotion more effectively than words; you can now use messages to send the score of a game or even play games through text message.

In this chapter, we are going to create an iMessages app. This app will allow the user to look for restaurants and send reservations to others. We will build our UI to look similar to what our phone looks like. In order to create the iMessages app, we need to add a message extension to our app.

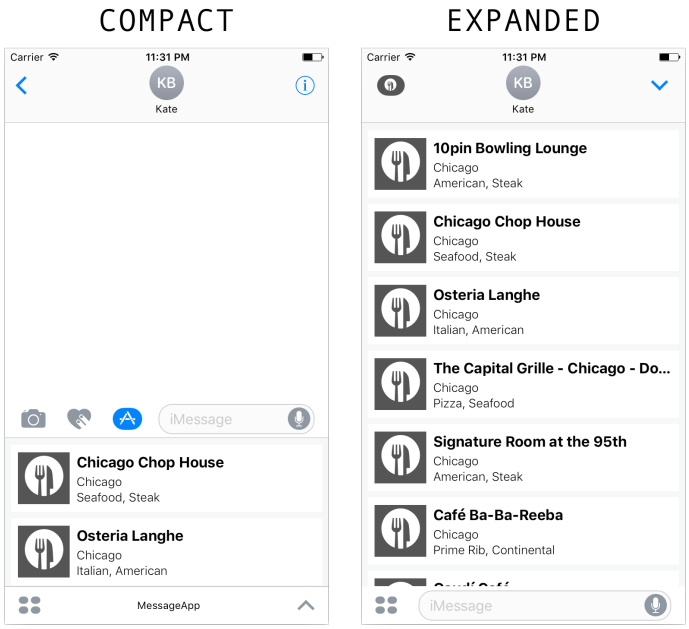
We will cover the following in this chapter:

- Building a custom message app UI
- Creating a framework
- Sharing code between multiple targets
- Learning how to send a reservation to others

Understanding iMessages

Starting with the UI is always my preferred way to begin building an app, because you can really get a feel for what you need to code. We are going to implement a single screen that will be a list of restaurants (accessible by hitting the Sticker icon next to where a user writes his or her message). The user can choose a restaurant for which he or she has a reservation and send it via messages to another person. Once that other person receives the message, that user will be able to tap on the reservation and see all of the details.

In a messages View Controller, there are two types of presentation styles: compact and expanded.

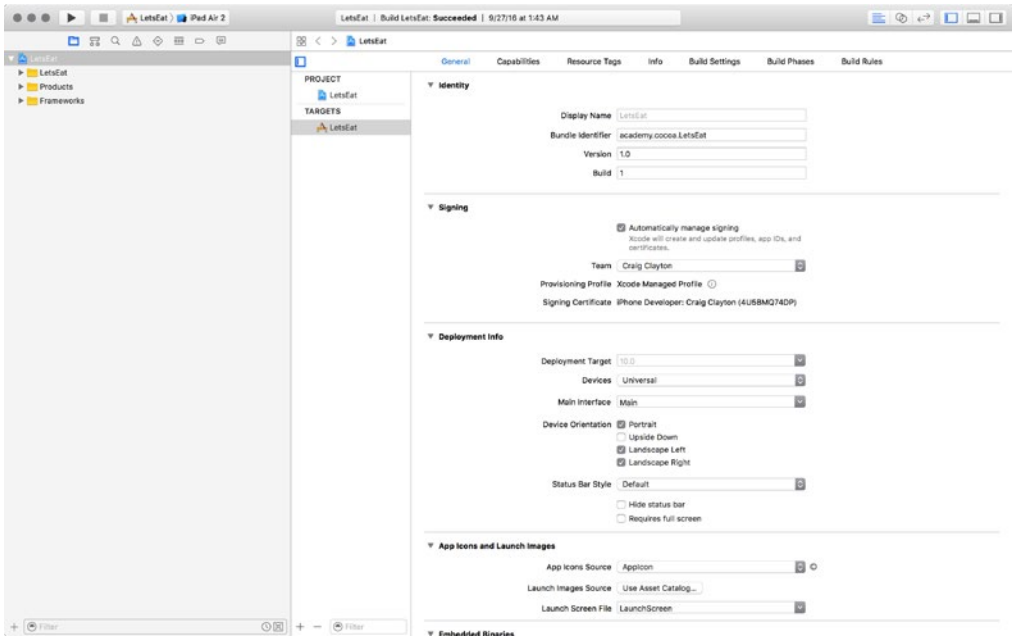


Apple recommends that you have two different View Controllers for each style. However, since our screen is simple, we will use just one. Keep in mind, however, that, if you want to do a more complex layout, you should use two Controllers.

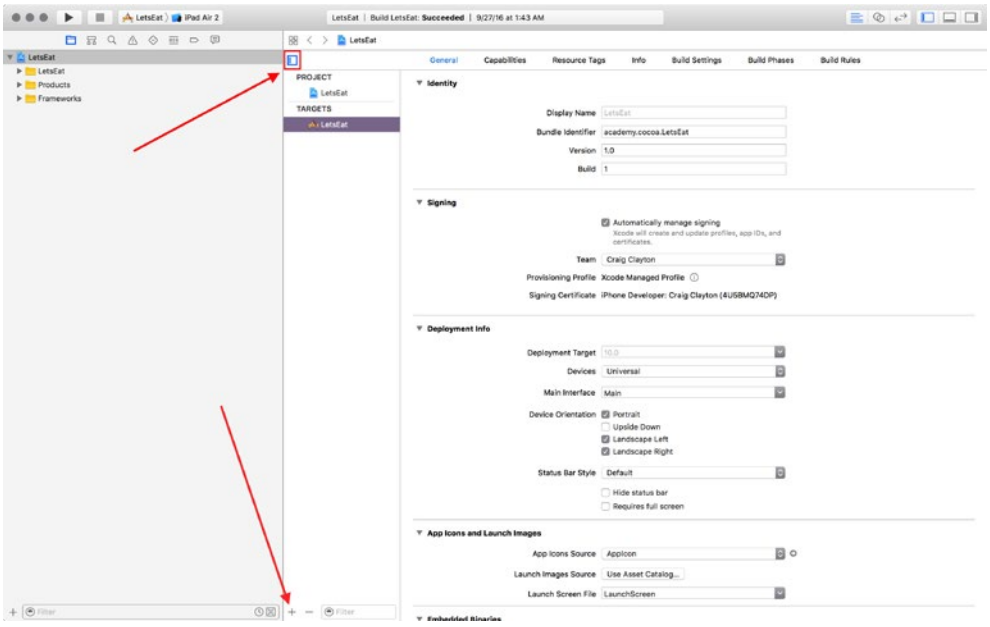
Creating our extension

Let's get started by working on the UI now:

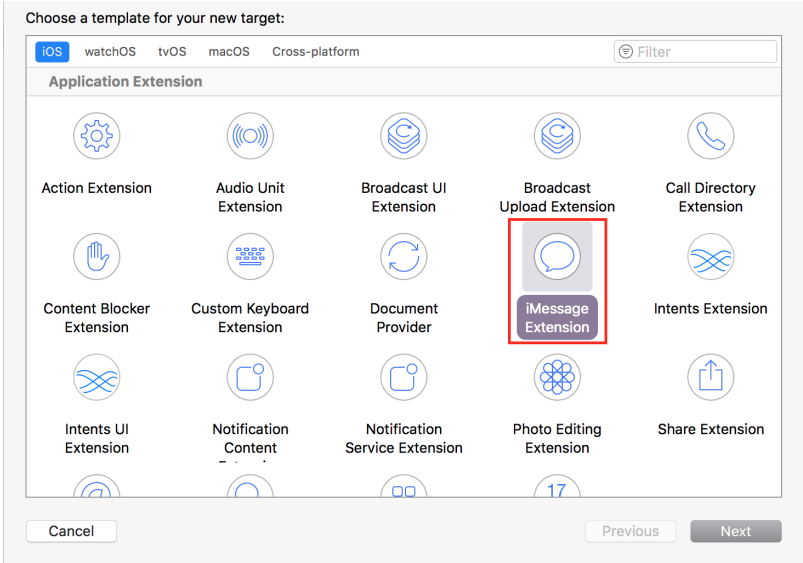
1. In the **Navigator** panel, select the **Project navigator** and, then, your project:



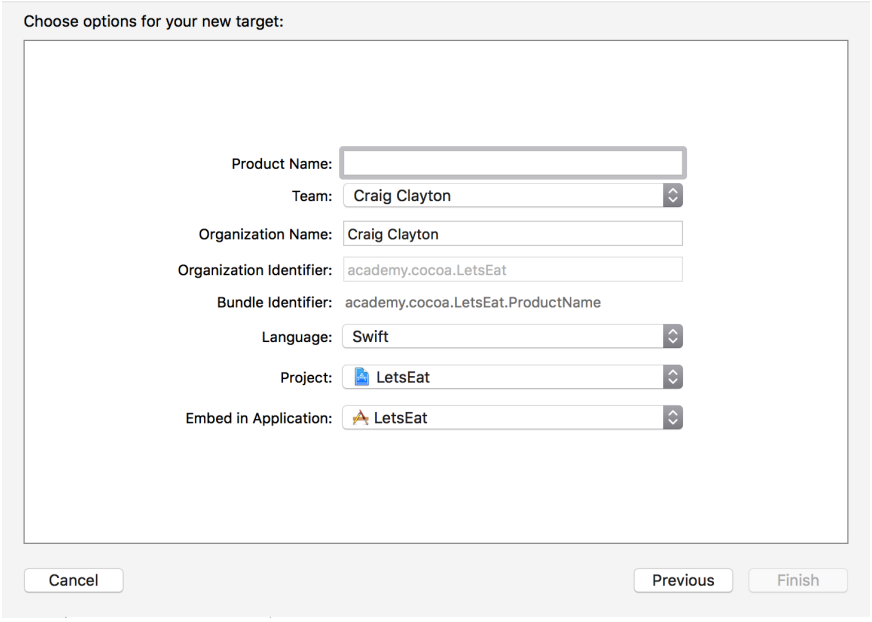
2. In the **Standard Editor**, locate the **Targets** area and the **+** (plus button) at the bottom of the **Targets** area (if your **Targets** area is not showing, hit the icon highlighted in blue to the left of **General** in the following screenshot):



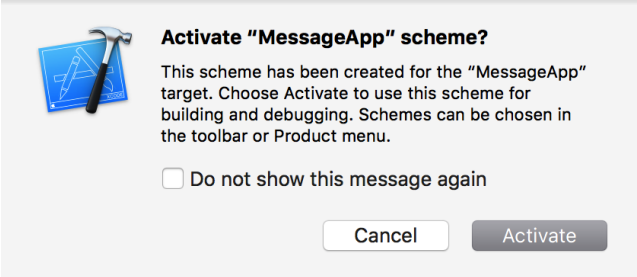
3. Click on the + (plus button) and, then, select **iMessage Extension**:



4. Click on **Next** and, then, you will see the following screen:



5. Set the **Product Name** to **MessageApp** and click on **Finish**.
6. You will receive the following message after you click on **Finish**. Select **Activate**:



By activating the **MessageApp** scheme, we will be able to build and run iMessages from the simulator. Now, you will have the choice of running either our *Let's Eat* app or our iMessages app.

Updating our assets

Next, we need to add assets that are necessary for our iMessages app:

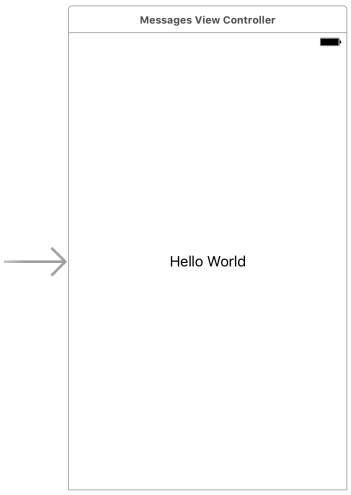
1. In the `MessageApp` folder in the **Navigator** panel, select the `Assets.xcassets` folder.
2. Hit the **Delete** button and, then, select **Move to Trash** in the screen that appears.
3. Then, open the project assets folder that you downloaded from PacktPub.
4. Open `Chapter_15` and drag the `Assets.xcassets` folder into your `MessageApp` folder in the **Navigator** panel.
5. In the options screen that appears, ensure that **copy items if needed** and **Create groups** are both selected; then, select **Finish**.

If you open the `Assets.xcassets` folder, you will see that you now have an icon and two other image assets that we will need for our iMessages app.

Implementing our Messages UI

Next, we need to set up our UI. In our iMessages app, we will have a single screen; in this screen, we will show a list of restaurants using a Collection View. When you tap on the restaurant, you will be able to send a reservation message to someone else. Let's get started:

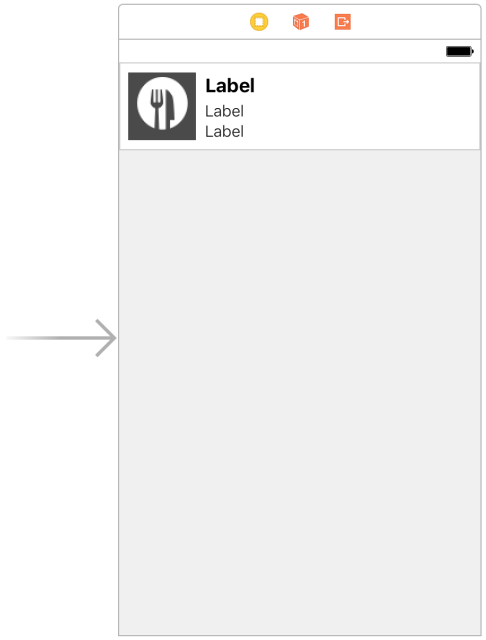
1. In your `MessageApp` project, select your `MainInterface.storyboard`.
2. You will see a single storyboard with a Label that says **Hello World**:



3. Delete the Label that says **Hello World** by selecting it in the **Outline** view and, then, hitting *Delete*.
4. Select the **Messages View Controller** and, then, the **Attributes Inspector** in the **Utilities** panel; change the **Status Bar** under **Simulated Metrics** from **Inferred** to **None**.
5. Next, in the **Object** library of the **Utilities** panel, type `collectionview` in the filter; then, drag out a **Collection View** into the **View Controller** in the scene.
6. With the **Collection View** selected, select the **Pin** icon and enter the following values:
    - All values under **Add New Constraints** are set to `0`
    - **Constrain to margins**: unchecked
    - **Update Frames**: Items of New Constraints

7. Click on **Add 4 Constraints**.
8. Next, with the **Collection View** still selected, open the **Attributes Inspector** in the **Utilities** panel.
9. Select **Background** in the **Attributes Inspector**; under the **Color Sliders** tab, set the **Hex Color #** to `ECECEC` under **RGB Sliders** in the drop-down menu.
10. Next, select the **Collection View** cell and, then, the **Size Inspector** in the **Utilities** panel.
11. Change the **Size** from **Default** to **Custom**. Then, set the **Width** to `320` and the **Height** to `78`.
12. Then, select **Background** in the **Attributes Inspector**; under the **Color Sliders** tab, set the **Hex Color #** to `FFFFF` under **RGB Sliders** in the drop-down menu.
13. In the **Utilities** panel, select the **Media** library and type `restaurant-list` into the filter field.
14. Drag a `restaurant-list-img` into your **Collection View** Cell.
15. With the image still selected, go to the **Size Inspector** in the **Utilities** panel and update the following values:
    - **X**: `8`
    - **Y**: `9`
    - **Width**: `60`
    - **Height**: `60`
16. Next, select the **Object** library in the **Utilities** panel and type `label` into the filter.
17. Drag three Labels into the Cell.
18. Select the first Label and, in the **Size Inspector**, update the following values:
    - **X**: `76`
    - Y: `10`
    - **Width**: `236`
    - **Height**: `21`
19. Next, select the **Attributes Inspector** and update the **Font** to **Bold** and verify that the **Font** size is `17`.

20. Select the second **Label** and, in the **Size Inspector**, update the following values:
    - **X**: `76`
    - **Y**: `35`
    - **Width**: `236`
    - **Height**: `16`
21. Then, in the **Attributes Inspector**, update the **Font** to **Light**, size `14`.
22. Select the last **Label** and, in the **Size Inspector**, update the following values:
    - **X**: `76`
    - **Y**: `53`
    - **Width**: `236`
    - **Height**: `16`
23. Lastly, in the **Attributes Inspector**, update the **Font** to **Light**, size `14`.
24. When you are done, your cell should look like the following:



Now that we have our items in place, we need to add some Auto Layout to our elements.

Adding Auto Layout to our cell

Auto Layout will allow our layout to adjust to all devices. Let's get started:

1. Select the image in our Cell and, then, select the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: `9`

      Left: `8`

      Constrain to margins: unchecked
    - **Width**: `60` (checked)
    - **Height**: `60` (checked)
2. Click on **Add 4 Constraints**.
3. Next, select the first Label in our Cell and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: `10`

      Left: `8`

      Right: `8`
    - **Constrain to margins**: unchecked
    - **Height**: `21` (checked)
4. Click on **Add 4 Constraints**.
5. Next, select the second Label in our Cell and, then, the Pin icon. Enter the following values:
    - Under Add New Constraints:

      Top: `4`

      Left: `8`

      Right: `8`
    - **Constrain to margins**: unchecked
    - **Height**: `16` (checked)
6. Click on **Add 4 Constraints**.

7. Finally, select the last Label in our Cell and, then, the Pin icon. Enter the following values:
    - Under **Add New Constraints**:

      Top: `2`

      Left: `8`

      Right: `8`
    - **Constrain to margins**: unchecked
    - **Height**: `16` (checked)
8. Click on **Add 4 Constraints**.

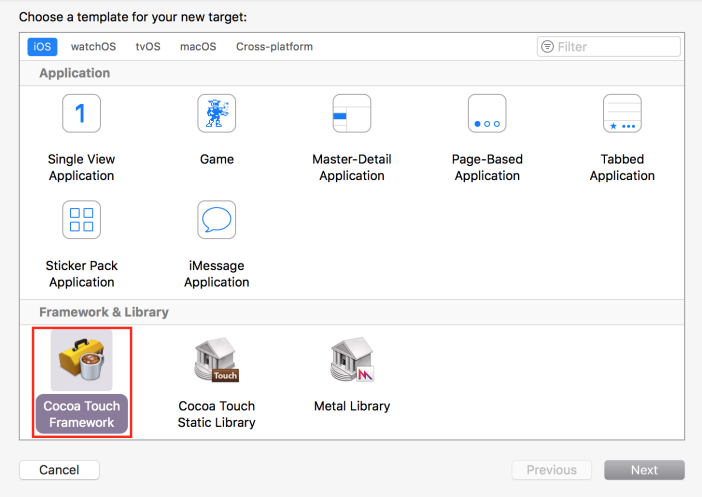
We completed setting up our UI and can now proceed to getting data into our app and displaying it.

Creating a framework

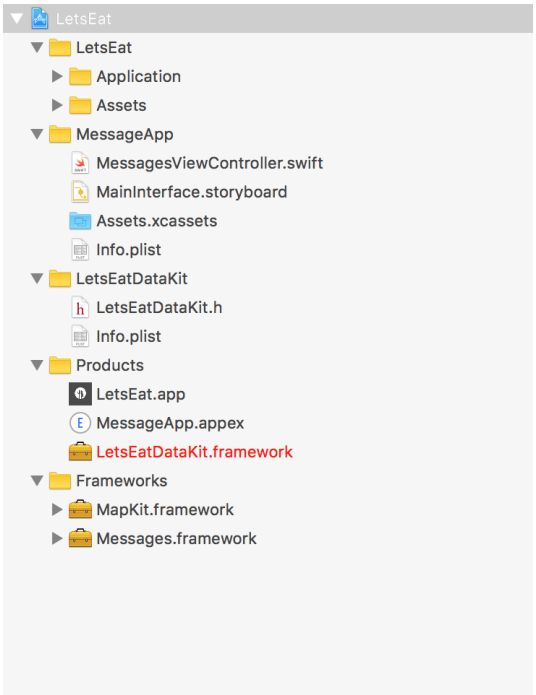
Since all of our code for data has already been created in our iOS app, it does not make sense to rewrite our code for our iMessages app. We can create what is known as a framework in order to share our data between our iOS and iMessages apps.

Using frameworks along with app extensions allows us to put shared code in one place. That means less code and more efficiency, because you will not need to update code in multiple places when you have to make a change. Let's get started creating our framework:

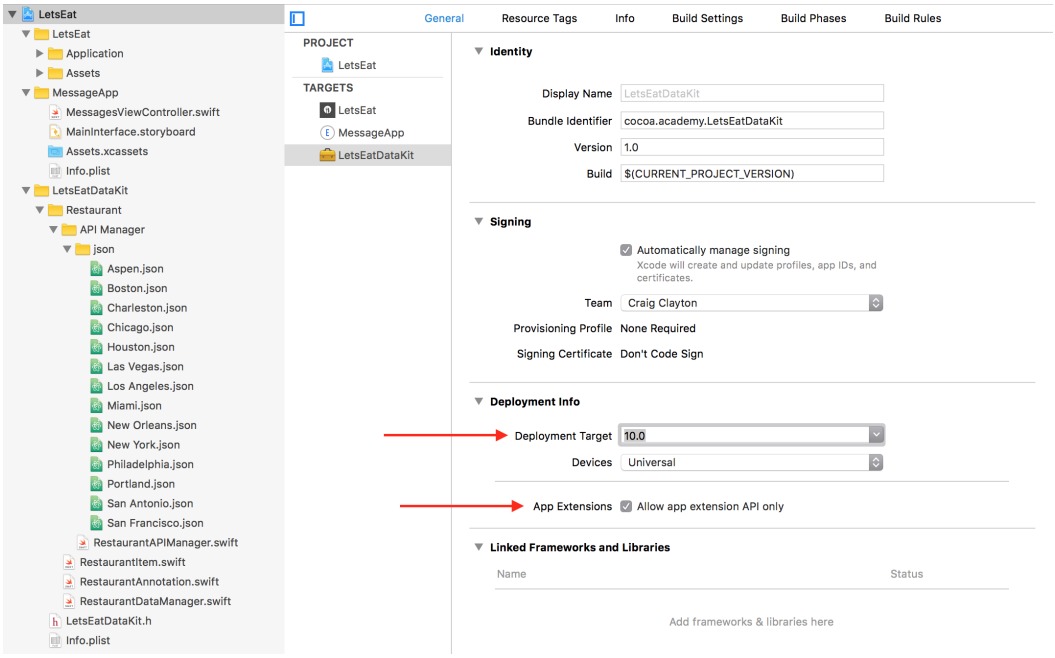
1. In the **Navigator** panel, select the **Project** navigator and, then, your project again as we did earlier.
2. Find the **Targets** area and click on the **+** button at the bottom of that area.
3. Under the **iOS** tab, scroll to the bottom to **Framework & Library** and select **Cocoa Touch Framework** and, then, hit **Next**:



4. Under **Product Name**, type `LetsEatDataKit` and, then, hit **Finish**.
5. You should now see the following folder and files in the `Products` folder in your **Navigator** panel:

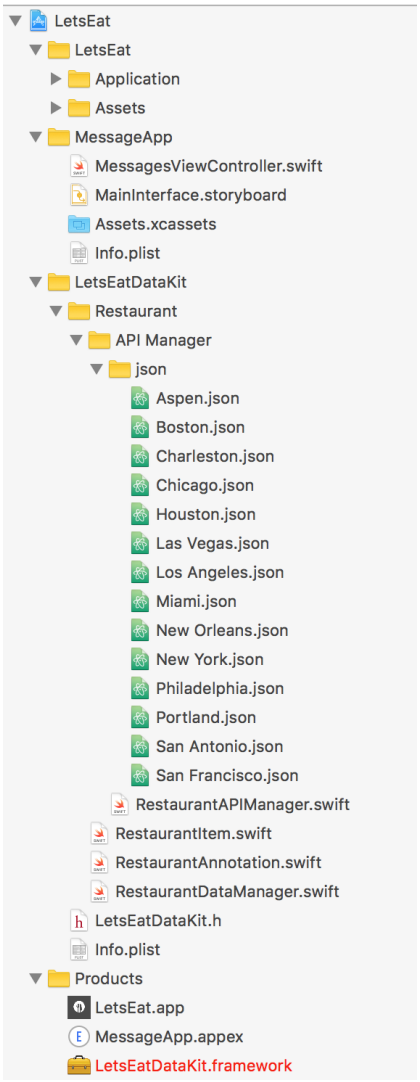


6. Select the **LetsEatDataKit** Target and ensure that, under **Deployment Info**, your **Deployment Target** is set to `10.0` and the **App Extensions** (allow app extension API only) is checked:



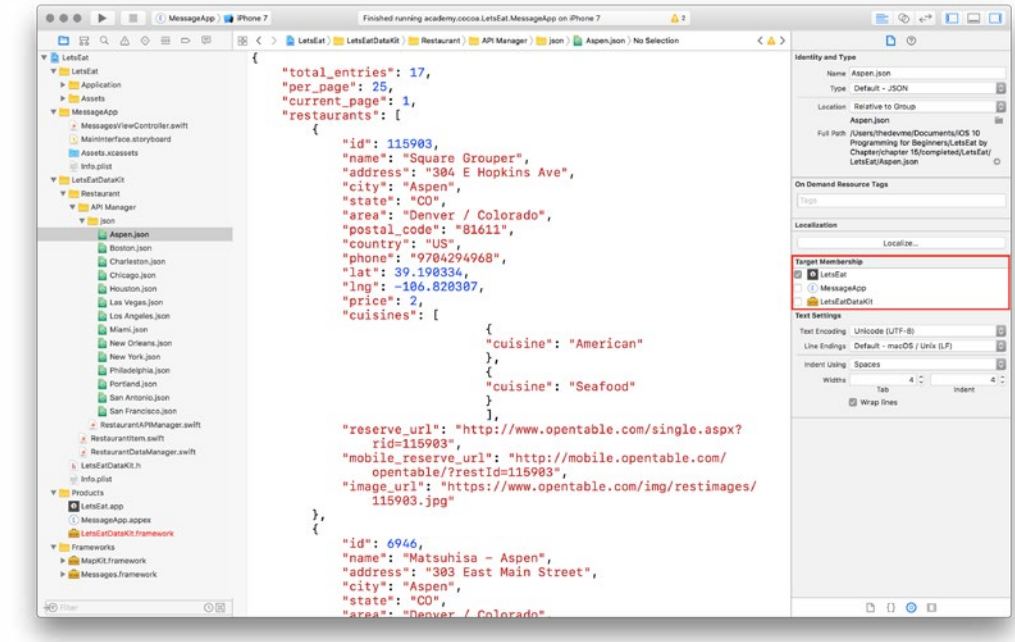
7. Right-click on the `LetsEatDataKit` folder in the **Navigator** panel and create a new group, named `Restaurant`.
8. Now, from your **Let's Eat** app, drag the `RestaurantDataManager.swift` file from the `Model` folder inside the `Restaurant List` folder into the newly created `LetsEatDataKit Restaurant` folder.
9. Next, drag the `RestaurantAnnotation.swift` and `RestaurantItem.swift` files from the `Model` folder inside the `Map` folder into the `LetsEatDataKit Restaurant` folder.
10. Then, drag the `RestaurantAPIManager.swift` file from the `Misc` folder inside the `Common` folder into the `LetsEatDataKit Restaurant` folder.
11. Finally, drag the entire `API Manager` folder, including the `JSON` folder, from inside the `Common` folder into the `LetsEatDataKit Restaurant` folder.

12. When you completed these steps, you should have the following files in your `LetsEatDataKit` folder:

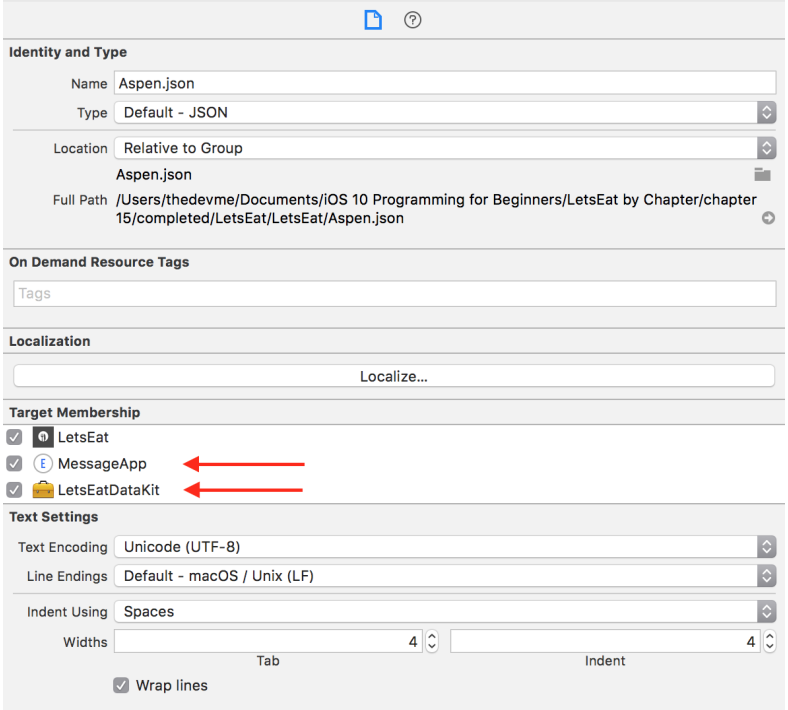


13. Next, open the `API Manager` folder we just moved and, in the `json` subfolder, select the `Aspen.json` file.

14. In the **Utilities** panel, select the **File Inspector** and locate the **Target Membership** section:

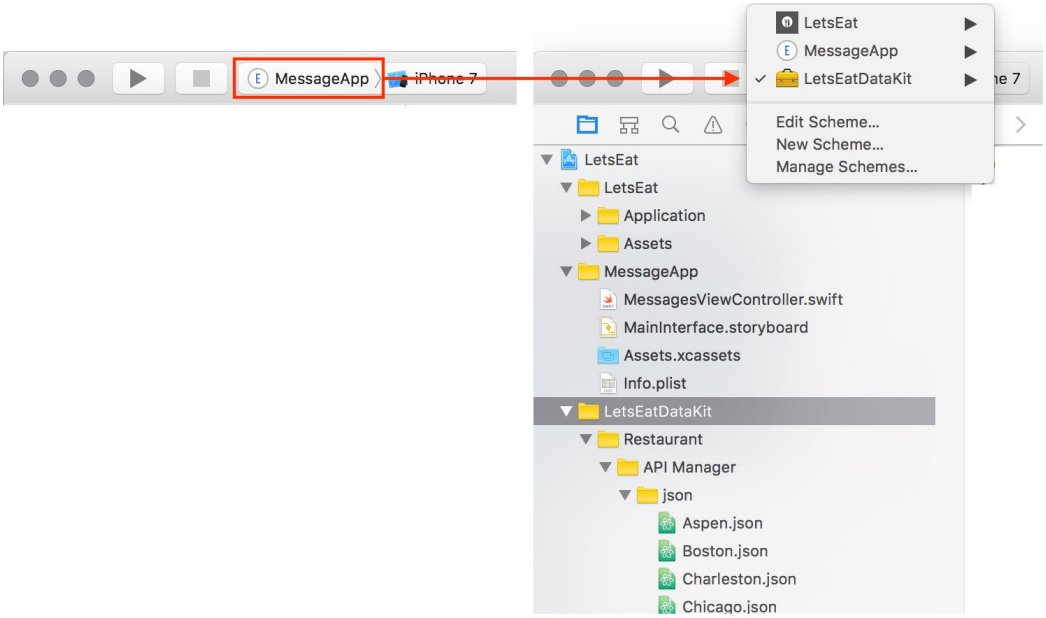


15. In order to set the target of this file to not only our app, but to our **MessageApp** and **LetsEatDataKit** as well, check **MessageApp** and **LetsEatDataKit** under **Target Membership**. Therefore, **LetsEat**, **MessageApp**, and **LetsEatDataKit** should all be checked:



16. Next, select each `json` file inside the **API Manager** `json` folder and update all of the files so that they are all targeted to **LetsEat**, **MessageApp**, and **LetsEatDataKit**.
17. Then, select each of the remaining four files inside the `LetsEatDataKit Restaurant` folder and update them so that each one is targeted to **LetsEatDataKit** only.

18. Now, change your target from **MessageApp** to **LetsEatDataKit**:



19. Hit *CMD* + *B* to build the app but not run it, and your build should be successful as long as you updated the target of all of your files.
20. Now, switch back to the **Let's Eat** app and hit *CMD* + *B*. You will notice some errors. These errors are expected and are easy to fix.
21. Inside the `MapViewController.swift` file, add the following import at the top of the file:

    ```
    import LetsEatDataKit
    ```

22. Then, continue by updating your `RestaurantItem`. We need to make this file public so that it can be seen by other files. Therefore, inside the `RestaurantItem.swift` file, update our `struct` declaration to add `public` before the `struct` so that it looks like the following:

    ```
    public struct RestaurantItem
    ```

Similarly, we need to make each of our variables public, since we are using them all as data. Update all variables by adding the keyword `public` in front of them and, then, save the file. When you add `public` to the annotation variable, you will see another error occur. This error is complaining because we are trying to make the variable public while the class is not public. Therefore, open your `RestaurantAnnotation` class and update; the class, each of the following variables, and the `init()` method with public access:

```
public class RestaurantAnnotation: NSObject, MKAnnotation
public var name: String?
public var cuisines:[String] = []
public var latitude: Double?
public var longitude:Double?
public var address:String?
public var postalCode:String?
public var state:String?
public var imageURL:String?
public var data:[String:AnyObject]?

public init(dict:[String:AnyObject])
public var title: String?
public var subtitle: String?
public var coordinate: CLLocationCoordinate2D
public var restaurantItem:RestaurantItem
```

Save the file and, now, your `RestaurantItem` errors will disappear.

We still have more minor updates to make. We need to make both our `RestaurantAPIDataManager` and `RestaurantDataManager` public as well. Let's start with the `RestaurantAPIDataManager` and update the following `struct` and method with public access:

```
public struct RestaurantAPIManager
public static func loadJSON(file name:String) -> [[String:AnyObject]]
```

Next, update the class and each of the following methods inside `RestaurantDataManager` with public access:

```
public class RestaurantDataManager
public func fetch(by location:String, withFilter:String="All",
completionHandler:() -> Swift.Void)
public func numberOfItems() -> Int
public func restaurantItem(at index:IndexPath) -> RestaurantItem
```

We also need to make our `init()` method for our `RestaurantDataManager` class public, so after the class declaration, add the following:

```
public init() {}
```

Having this `init()` method allows us to write the following:

```
let manager = RestaurantDataManager()
```

When we make it public, we are calling the `init()` method when we have `RestaurantDataMananger()`.

Now, change the target to the **LetsEatDataKit** and build it again by hitting *CMD* + *B*. The build should be successful again at this point. If you, open the `MapViewController` file, you should see that all of the errors have been fixed in this file.

However, we still have more errors to address inside the **MapDataManager**, **RestaurantListViewController**, and **RestaurantDetailViewController**. Therefore, inside each of these three files, add the following at the top of each file in the import statement section:

```
import LetsEatDataKit
```

Next, hit *command* + *B* again, and there should be no errors inside any of these three files or in your entire project.

If you switch the target back to our **Let's Eat** app and build and run it by hitting the play button (or use *command* + *R*), you should see that everything is working as expected. We can now start using this data in our iMessages app.

Connecting our message cell

Now that we have our files in order, we can start connecting everything. Earlier we created our Cell, and now we need to create a Cell class with which to connect it:

1. Right-click on the `MessageApp` folder in the **Navigator** panel and select **New File**.
2. Inside the **Choose a template for your new file screen**, select **iOS** at the top and, then, **Cocoa Touch Class**. Then, hit **Next**.
3. You will now see an options screen. Add the following:
    - New File:

      Class: RestaurantMessageCell

      Subclass…: UICollectionViewCell

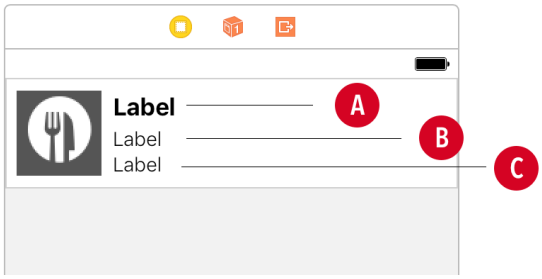
Also create XIB: Unchecked

Language: Swift

4. Click on **Next** and, then, **Create**.
5. In the new file, add the following inside the class declaration:

```
@IBOutlet var lblTitle:UILabel!
@IBOutlet var lblCity:UILabel!
@IBOutlet var lblCuisine:UILabel!
```

6. Save the file and, then, open `MainInterface.storyboard` in the `MessageApp` folder in the **Navigator** panel.
7. In the **Outline** view, select the **Collection View** Cell.
8. Then, select the **Identity Inspector** in the **Utilities** panel, and under **Custom Class**, in the **Class** drop-down menu, select **RestaurantMessageCell** and hit *Enter*.
9. Next, switch to the **Attributes Inspector** in the **Utilities** panel and update the identifier to `restaurantCell` and, then, hit **Enter**.
10. Now, switch to the **Connections Inspector** in the **Utilities** panel, and click on and drag from the empty circle next to each Outlet listed here to the corresponding `UILabel` in the scene shown in the following screenshot:
    - `lblTitle`
    - `lblCity`
    - `lblCuisine`



We now have our Cell setup. Let's continue getting our iMessages app working.

Showing restaurants

We will be showing a list of restaurants just like in our app, but we will not be doing the entire interface. Most of this code will be familiar to you as we have done it before.

Open up the `MessagesViewController.swift` file in the **Navigator** panel and under `import Messages`, add the following:

```
import LetsEatDataKit
```

Then, inside the class declaration, add the following code:

```
@IBOutlet var collectionView: UICollectionView!

let manager = RestaurantDataManager()
var selectedRestaurant:RestaurantItem?
```

Next, we need to set up our **Collection View** defaults. Add the following method after the `didReceiveMemoryWarning()` method:

```
func setupCollectionView() {
    let flow = UICollectionViewFlowLayout()

    flow.sectionInset = UIEdgeInsets(top: 7, left: 7, bottom: 7,
right: 7)
    flow.minimumInteritemSpacing = 0
    flow.minimumLineSpacing = 7

    collectionView.collectionViewLayout = flow
    collectionView.delegate = self
    collectionView.dataSource = self
}
```



You will see errors once you add the preceding code. Ignore them for now as we will fix them shortly.

Now, we will create an `initialize()` method that will set up the **Collection** View and fetch our data. Add the following method after the `didReceiveMemoryWarning()` method:

```
func initialize() {
    setupCollectionView()

    manager.fetch(by: "Chicago", completionHandler: {
```

```
            self.collectionView.reloadData()
        })
    }
```

Since we are not doing a location list, we will just pass a city in manually. Here, we use Chicago, but you can change it to any city of your choice.

Next, call the `initialize()` method inside the `viewDidLoad()` method so that your `viewDidLoad()` method now looks as follows:

```
override func viewDidLoad() {
        super.viewDidLoad()

        initialize()
    }
```

Then, let's create an extension with our **Collection View** delegates and data source. After the last curly brace in the `MessagesViewController.swift` file, add the following extension declaration:

```
extension MessagesViewController:UICollectionViewDelegate,
UICollectionViewDataSource, UICollectionViewDelegateFlowLayout {
}
```

Now that we have our extension set up, let's add all of the methods we need to get our **Collection View** showing data. Add the following inside our extension (which will get rid of our earlier errors):

```
func numberOfSections(in collectionView: UICollectionView) -> Int {
    return 1
}

func collectionView(_ collectionView: UICollectionView,
numberOfItemsInSection section: Int) -> Int {
    return manager.numberOfItems()
}

func collectionView(_ collectionView: UICollectionView, cellForItemAt
indexPath: IndexPath) -> UICollectionViewCell {
    let cell = collectionView.dequeueReusableCell(withReuseIdentifier:
"restaurantCell", for: indexPath) as! RestaurantMessageCell
    let item = manager.restaurantItem(at: indexPath)

    if let name = item.name { cell.lblTitle.text = name }
    if let city = item.city { cell.lblCity.text = city }
```

```
        if let cuisine = item.cuisine { cell.lblCuisine.text = cuisine }

        return cell
    }

    func collectionView(_ collectionView: UICollectionView, layout
    collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath:
    IndexPath) -> CGSize {
        let cellWidth = self.collectionView.frame.size.width - 14

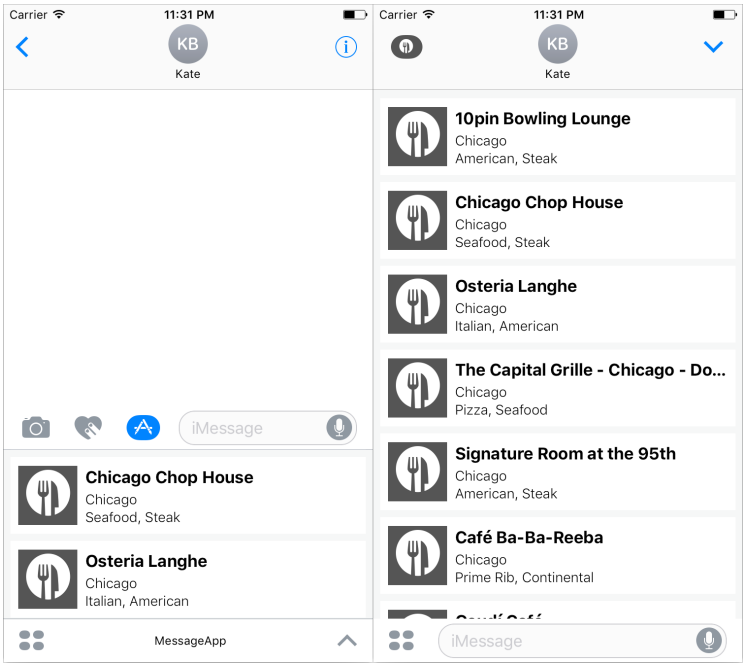
        return CGSize(width: cellWidth, height: 78)
    }
```

You should be very familiar with what we just added. We are setting up our **Collection View** data source as well as making sure our cells have spacing of 14 pixels (seven on each side).

Lastly, before we build our app, we need to connect our **Collection View** in Storyboard:

1. Open up `MainInterface.storyboard` in the `MessageApp` folder in the **Navigator** panel.
2. Select the **Message View Controller** and, then, the **Connections Inspector** in the **Utilities** panel.
3. Then, under **Outlets**, click on and drag from the empty circle next to **collectionView** to the **Collection View** in our scene.

Let's change the target to **Message App** and build and run our iMessages app by hitting the play button (or use *command* + *R*). Your app should look similar to the following after clicking on the stickers button:



Hitting the arrow (highlighted by the red boxes) will expand the screen to expanded mode from compact mode and back again. Now that we have our restaurants being displayed, we need to be able to send the restaurant reservation to other people. Let's add that next.

Sending reservations

We need to set up our **Collection View** so that when the user taps on a cell, it will add the reservation to the conversation in iMessages. When creating a message to send, we have the following things to set:



We will use everything but the **Trailing Caption** and **Trailing Subcaption**.

Open up `MessagesViewController` in the `MessageApp` folder in the **Navigator** panel.

Then, in our main class declaration, add the following method after the `setupCollectionView()` method:

```
func createMessage(with restaurant:RestaurantItem) {
    if let conversation = activeConversation {
        let layout = MSMessageTemplateLayout()
        layout.image = UIImage(named: "restaurant-detail")
        layout.caption = "Table for 7, tonight at 10:00 PM"
        layout.imageTitle = restaurant.name
        layout.imageSubtitle = restaurant.cuisine

        let message = MSMessage()
        message.layout = layout
        message.url = URL(string: "emptyURL")

        conversation.insert(message, completionHandler: { (error:
Error?)  in
            if error != nil {
                print("there was an error \(error)")
            }
            else {
                self.requestPresentationStyle(.compact)
            }
        })
    }
}
```

In this method, we set up an **MSMessage**. We check for an active conversation first. If we have an active conversation, we then set up our layout. Here, we are just using an image from our assets to create an image background (we could have also used a video, for example). In addition, we set the caption to Table for 7, tonight at 10:00 PM. This allows the receiver to see all of the relevant information for the reservation. Next, we set the restaurant name as the image title and the restaurant's cuisine as the image subtitle. Then, we create an instance of MSMessage, pass it the layout we created, and give it a URL (which, in our case, is just an empty String, since we do not have a URL). Finally, we insert the message into the conversation. We need to make sure that, when we want to send a message, we are in compact mode; otherwise, the user will think that the app does not work.

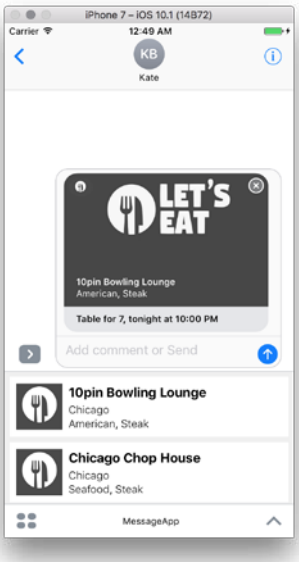
Lastly, we just need to add the code that calls our `createMessage()` method. Add the following method in our extension, but before the last curly brace:

```
func collectionView(_ collectionView: UICollectionView,
didSelectItemAt indexPath: IndexPath) {
    selectedRestaurant = manager.restaurantItem(at: indexPath)
    guard let restaurant = selectedRestaurant else { return }

    createMessage(with: restaurant)
}
```

Here, we check for when the user taps a cell. Then, we get the `selectedRestaurant` and pass it to our `createMessage()` method.

Let's build and run the project by hitting the play button (or use *command* + *R*). Select a restaurant, and you will now see a message with the selected restaurant in the message area:



You can see that, with a little bit of work, you can add a nice iMessages app to your app.

Summary

In this chapter, we looked at how to add an iMessages app to our app. We also created a framework that allowed us to use data in both our apps without having to duplicate code. We looked at what is involved with creating an MSMessage and how we can pass an `MSMessageTemplateLayout` to an MSMessage. We now know that we can also send embedded videos as well as images when we send messages. In addition, we can now send reservations through the iMessages app with relevant data for a reservation.

In the next chapter, we will go back to our *Let's Eat* app, and you will learn how to work with notifications in our app.

16
Notifications

Notifications were first launched in 2009 and have been a staple of the iOS system. Whether you receive a notification from your favorite app or a text message, you will encounter a notification at some point while using a smartphone. Pre-iOS 10, if you had to work with notifications in iOS, you had two types of notifications: remote (from a server) and local.

iOS 10 made changes to notifications that simplified them, but also made them more robust. In iOS 10, there is now one notification that covers both remote and local notifications, which is great for those who worked with them in the past. In terms of breadth of functionality, notifications now allow you to embed rich media (such as images, videos, and audios) as well as having custom UI content.

In this chapter, you are going to learn how to create basic notifications as well as notifications with embedded images. After we look at both of these examples, we will also look at how to create a custom UI for our notifications.

We will cover the following topics in this chapter:

- Learning how to build basic notifications
- Learning how to embed images into notifications
- Learning how to build a custom notification UI

Starting with the basics

Let's begin by getting our app to send us basic notifications. Inside our restaurant details page, we have three buttons (9:30PM, 10:00PM, and 10:30PM) that currently do not do anything. We are going to update these buttons so that when you tap on one of them, it will create a restaurant reservation notification. If this were a real reservations app, we would want to store these reservations. When the reservation date and time neared, we would then post a notification to the user as a reminder. Doing all of that is out of the scope of this book, so we will just address creating a restaurant reservation notification.

Getting permission

Before we can send any notifications, we must get the user's permission. Therefore, open the `AppDelegate.swift` file and add the following method after the `didFinishLaunchingWithOptions()` method:

```
func checkNotifications() {
    UNUserNotificationCenter.current().requestAuthorization(options:
[.alert, .sound, .badge]) { (isGranted, error) in
    }
}
```

When you add this method, you will get an error. The reason for this error is because we need to import `UserNotifications`. At the top of the file, under `import UIKit`, add the following:

```
import UserNotifications
```

Next, the method we just added will need to be run inside the `applicationDidFinishLaunching(_ application: UIApplication)` method. Add the following after `setupDefaultColors()`:

```
checkNotifications()
```

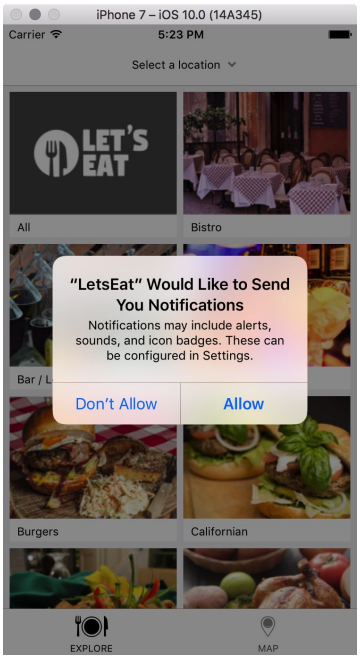
Your `applicationDidFinishLaunching(_ application: UIApplication)` method should now look like the following:

```
func application(_ application: UIApplication,
didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {

    setupDefaultColors()
    checkNotifications()

    return true
}
```

Build and run the project by hitting the play button (or use *CMD* + *R*), and you should see the following message:



Setting up notifications

Now that we have permission, we need to set up notifications. We will start with setting up our buttons:

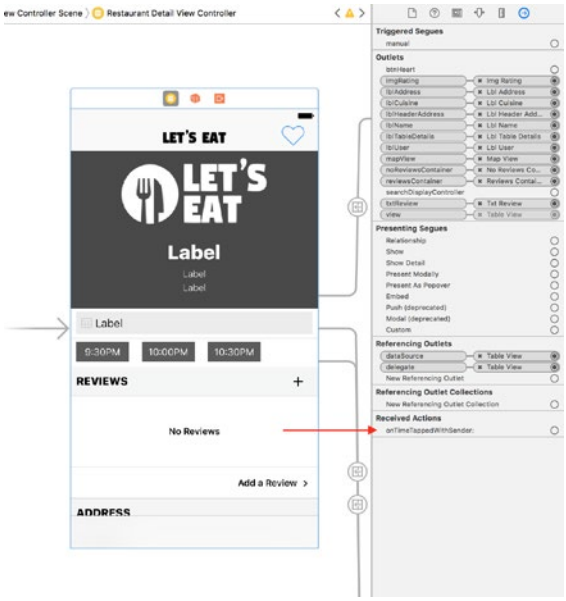
1. Open the `RestaurantDetailViewController.swift` file and add the following method after our `showAllReviews()` method and before the last curly brace of our class file:

```
@IBAction func onTimeTapped(sender: UIButton) {

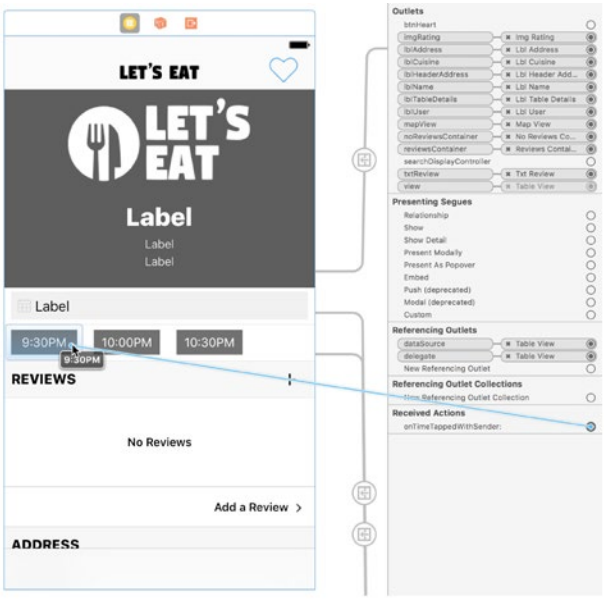
}
```

2. Save the file, and you will see an empty circle appear next to this new `@IBAction`.
3. Now, open the `RestaurantDetail.storyboard` and select the `RestaurantDetailViewController`.

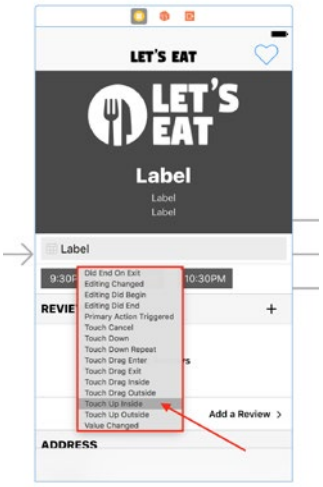
4. Select the **Connections Inspector** in the **Utilities** panel; under **Received Actions**, you will see `onTimeTappedWithSender`:



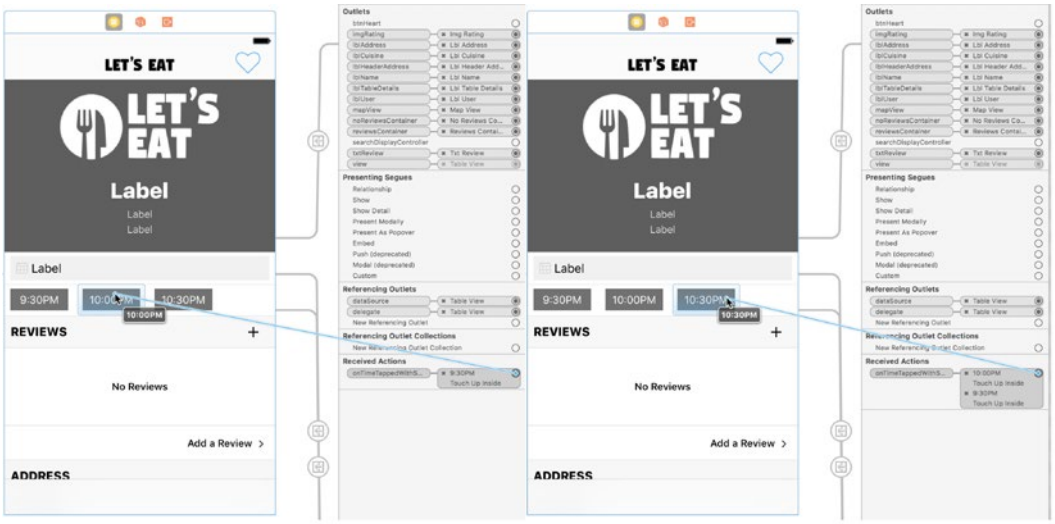
5. Click on and drag from the empty circle next to `onTimeTappedWithSender` to the first button (marked 9:30 PM) in the restaurant detail scene.



6. In the prompt, select **Touch Up Inside**.



7. Repeat these steps for the remaining two buttons (10:00 PM and 10:30 PM), clicking and dragging the same circle (now filled) to each of the remaining buttons in the scene and then choosing **Touch Up Inside** for each prompt that follows.



Now that we have our buttons set up, return to the `RestaurantDetailViewController.swift` file where we need to get the time inside the buttons to be passed to our notifications. Add the following method after the `showAllReviews()` method and above the `onTimeTapped()` method:

```
func showNotification(sender:String?) {
print(sender as Any)
}
```

Inside the `onTimeTapped()` method, add the following:

```
showNotification(sender: sender.titleLabel?.text)
```

We are now passing the time value to our `showNotification()` method. Build and run the project by hitting the play button (or use *command* + *R*). You should now see in the console the time of each selected button.

Showing notifications

Now that we have a time, let's show our notification along with the time selected.

In the `RestaurantDetailViewController.swift` file, after `import LetsEatDataKit`, add the following:

```
import UserNotifications
```

Next, inside the `showNotification()` method, add the following:

```
let content = UNMutableNotificationContent()

if let name = selectedRestaurant?.name { content.title = name }
if let time = sender { content.body = "Table for 7, tonight at \(time)
" }
content.subtitle = "Restaurant Reservation"
content.badge = 1
content.sound = UNNotificationSound.default()

let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 5,
repeats: false)
let identifier = "letsEatReservation"
let request = UNNotificationRequest(identifier: identifier, content:
content, trigger: trigger)

UNUserNotificationCenter.current().add(request, withCompletionHandler:
{ error in
    // handle error
})
```

In the preceding code, we are creating a notification content object. In this object, we are going to set the title, body, subtitle, badge, and sound.

After the `initialize()` method, add the following method:

```
func setupNotificationDefaults() {
    UNUserNotificationCenter.current().delegate = self
}
```

This is our delegate method for notifications. We get an error for our delegate, because we have not yet implemented the required functions. Let's do that now by creating an extension at the end of this file after the last curly brace. You may already have an extension in this file for our Map if you tackled the challenges at the end of *Chapter 11*, *Where's My Data?*; if so, add this new extension after the last curly brace of that Map extension. In either case, add the following code:

```
extension RestaurantDetailViewController:
UNUserNotificationCenterDelegate {

    func userNotificationCenter(_ center: UNUserNotificationCenter,
willPresent notification: UNNotification, withCompletionHandler
completionHandler: @escaping (UNNotificationPresentationOptions) ->
Void) {
        completionHandler([.alert, .sound])
    }
}
```

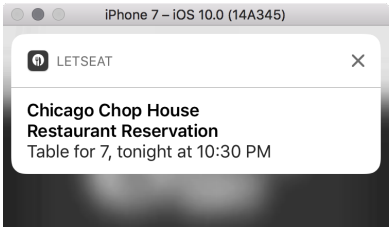
Finally, we just need to call the `setupNotificationDefaults()` method inside our `initialize()` method. Your updated `initialize()` method should now look like the following:

```
func initialize() {
    setupLabels()
    setupMap()
    setupNotificationDefaults()
}
```

Build and run the project by hitting the play button (or use *command* + *R*). Open a restaurant detail page, tap the time button and wait for 5 seconds. You should see the following:



Then, if you click on and pull down on the notification, you will see the following:



We've just implemented a basic notification; however, we can do so much more. Next, let's get an image inside our notification.

Customizing our notifications

Before we can embed an image, we will need a test image. In the `Assets` folder of the **Navigator** panel, create a new group, called **Images**. Then, in the project folder for this book, open the `asset` folder for this chapter and, then, drag the image assets into the Images folder that we've just created.

Embedding images

Next, let's embed our images. First, return to the `RestaurantDetailViewController.swift` file, and in the `showNotification()` method we created, remove the following code:

```
let trigger = UNTimeIntervalNotificationTrigger(timeInterval: 5,
repeats: false)
let identifier = "letsEatReservation"
let request = UNNotificationRequest(identifier: identifier, content:
content, trigger: trigger)

UNUserNotificationCenter.current().add(request, withCompletionHandler:
{ error in
    // handle error
})
```

Replace the deleted section of code with the following code:

```
do {
    let url = Bundle.main.url(forResource: "sample-restaurant-img@3x",
withExtension: "png")
    if let imgURL = url {
        let attachment = try UNNotificationAttachment(identifier:
"letsEatReservation", url: imgURL, options: nil)
        content.attachments = [attachment]

        let trigger = UNTimeIntervalNotificationTrigger(timeInterval:
5, repeats: false)
        let identifier = "letsEatReservation"
        let request = UNNotificationRequest(identifier: identifier,
content: content, trigger: trigger)

        UNUserNotificationCenter.current().add(request,
withCompletionHandler: { error in
            // handle error
        })
    }
```

```
    }
    catch {
        print("there was an error with the notification")

    }
```

In this `do-catch`, we get the image URL from our project and create an attachment. We attach the rich media (here, an image) to the notification. The rest of the code is what we removed and just added back inside the `do-catch`.

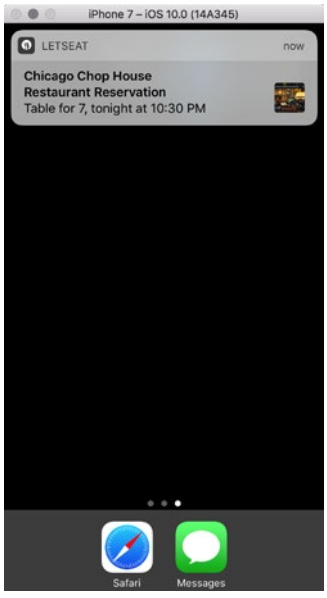
Build and run the project again by hitting the play button (or use *CMD* + *R*). When you get to a restaurant detail page, tap the time button and wait 5 seconds. You will now see a thumbnail image in the notification:



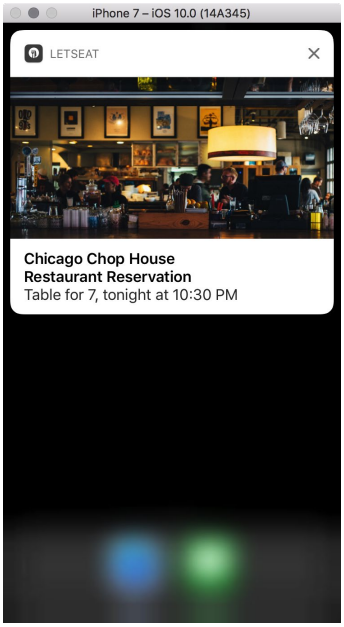
In addition, if you click on and pull down on the notification, you will see the following:



Thus far, we have been receiving notifications while inside the app. If you want to test notifications outside the app, take the following steps. Build and run the project by hitting the play button (or use *CMD* + *R*). When you get to a restaurant detail page, tap the time button and, then, immediately hit *CMD* + *SHIFT* + *H*. This will take you out of the app, and you will then see the following:



If you click on and pull down the notification, you will see the following:



Our notifications are looking good, but you really cannot do anything with them. It would be nice to confirm your reservation with a *yes* or *no*, for example. We need to add some buttons for the notifications in order to do this.

Adding buttons

We only need to add a few things in order to add buttons to our notifications. First, we need to update our restaurant detail:

1. Inside the `RestaurantDetailViewController.swift` file, add the following into the `showNotification()` method after `content.sound = UNNotificationSound.default()`:

   ```
   content.categoryIdentifier = "reservationCategory"
   ```

2. Next, we need to create a file for our button identifiers:
3. Right-click on the `Misc` folder inside the `Common` folder and select `New File`.
4. Inside the **Choose a template for your new file screen**, select **iOS** at the top and, then, **Swift File**. Then, hit **Next**.
5. Name this file, **Identifier**, and hit **Create**.

It is a good practice to eliminate as many Strings from your app as you can. Adding this file, will not only eliminate Strings, but also keep you from accidentally typing the wrong value. For example, we could easily misspell identifier. Therefore, it is a protective measure to have it in an enum. Add the following under the import statement in this new file:

```
enum Identifier:String {
    case reservationCategory
    case reservationIdentifier = "letsEatReservation"
}

enum Option:String {
    case one = "optionOne"
    case two = "optionTwo"
}
```

We will use this to create our button options for our notification. Open the `AppDelegate.swift` file. In the `checkNotifications()` method, add the following code:

```
let optionOne = UNNotificationAction(identifier: Option.one.rawValue,
title: "Yes", options: [.foreground])
let optionTwo = UNNotificationAction(identifier: Option.two.rawValue,
title: "No", options: [.foreground])

let category = UNNotificationCategory(identifier: Identifier.
reservationCategory.rawValue, actions: [optionOne, optionTwo],
intentIdentifiers: [], options: [])
UNUserNotificationCenter.current().setNotificationCategories([catego
ry])
```

Add this code inside the `requestAuthorization` block:

```
func checkNotifications() {
    UNUserNotificationCenter.current().requestAuthorization(options: [.alert, .sound, .badge]) { (isGranted, error) in

        // Add code here

    }
}
```

In this code, we set up two actions: one for yes and one for no. We create a category and, then, set it to our notification category, which defines the type of notification that we want to use.

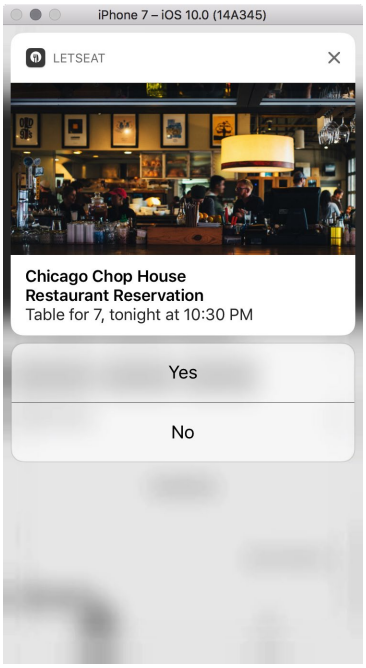
Lastly, we need to write code to handle when we receive a notification. Return to the `RestaurantDetailViewController.swift` file and add the following inside your new extension for notifications after the `willPresent()` method:

```
func userNotificationCenter(_ center: UNUserNotificationCenter,
didReceive response: UNNotificationResponse, withCompletionHandler
completionHandler: @escaping () -> Void) {

    if let identifier = Option(rawValue: response.actionIdentifier) {
        switch identifier {
        case .one:
            print("User selected yes")
        case .two:
            print("User selected no")
        }
    }

    completionHandler()
}
```

Build and run the project by hitting the play button (or use *CMD + R*). When you get the notification and pull down on it, you will see that you now have button options:



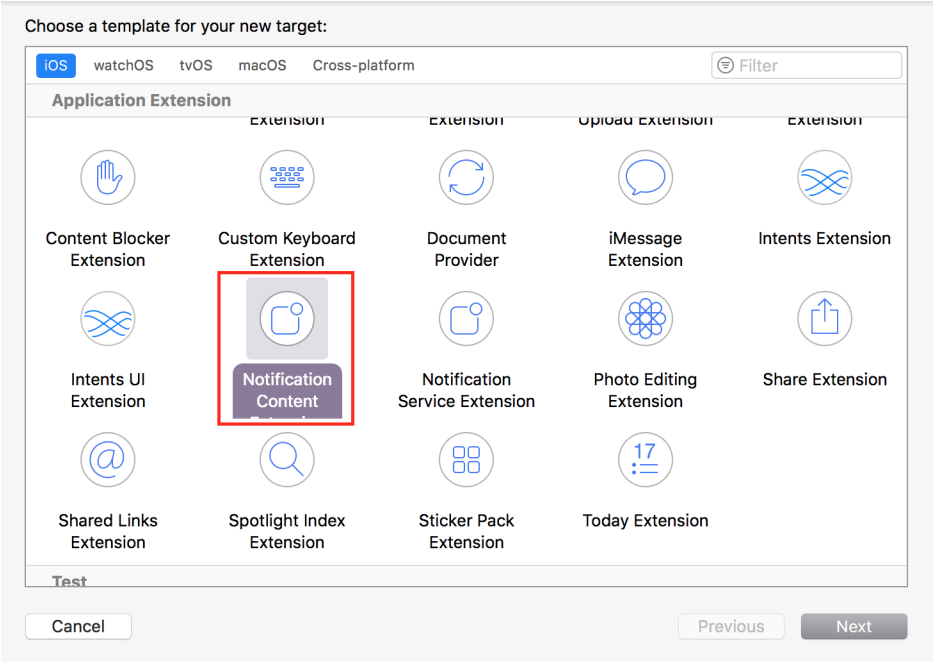
Inside our `didReceive()` method, we print out what the user selected, but you can choose whatever print statement you would like.

Up until this point, we looked at how to create basic notifications as well as notifications with images embedded in them. Next, we can take our app a step further by adding our own custom UI into our notifications.

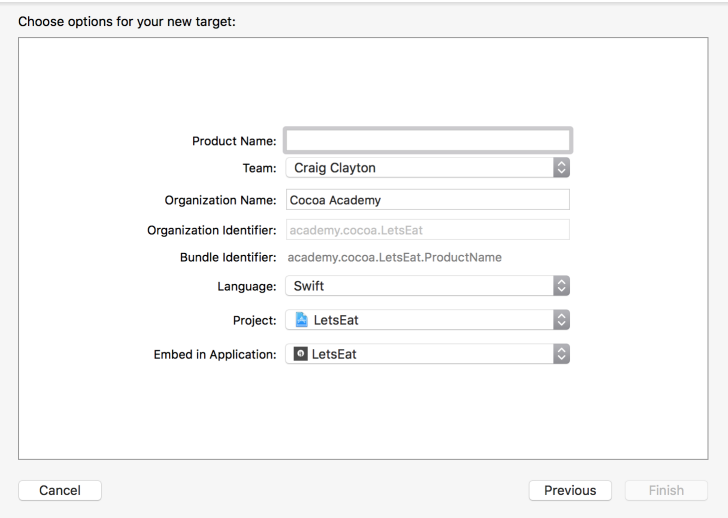
Custom UI in notifications

In order to add custom UI to our notifications, we need to add an extension. Let's get started by doing the following:

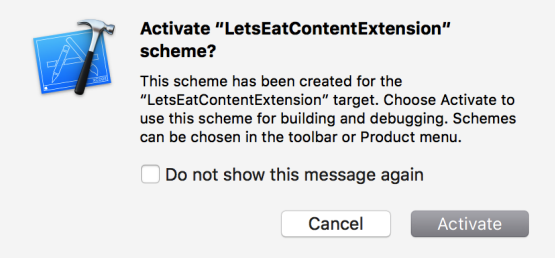
1. In the **Navigator** panel, select the **Project** navigator and, then, your project.
2. At the bottom of the **Targets** area, click on the + button.
3. Select **Notification Content Extension** under **Application Extension** and, then, click on **Next**:



4. In the options screen that appears, set **Product Name** to `LetsEatContentExtension` and click on **Finish**:



5. Next, select **Activate** in the screen that appears:



6. This activation will allow us to build and run our custom UI extension in the simulator. Now, you have a choice of running our app, the iMessages app, our kit, or our custom UI extension.

Now that our extension is created, we need to be able to use it.

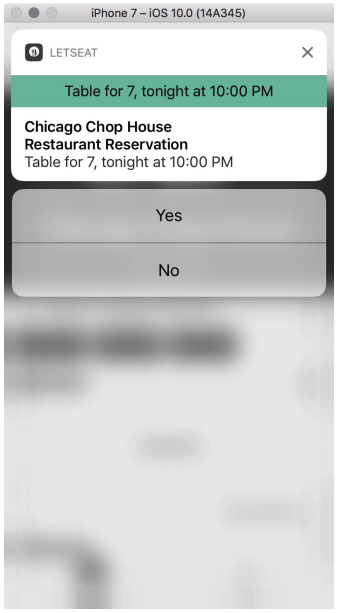
1. Open the `info.plist` file in our `LetsEatContentExtension` folder.
2. Tap the **NSExtension** disclosure arrow to open up that key.

3. Then, tap the disclosure arrow to open **NSExtensionAttributes**, under which you will see **UNNotificationExtensionCategory**:

Key	Type	Value
▼ Information Property List	Dictionary	(10 items)
Localization native development re...	String	en
Bundle display name	String	LetsEatContentExtension
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	XPC!
Bundle versions string, short	String	1.0
Bundle version	String	1
▼ NSExtension	Dictionary	(3 items)
▼ NSExtensionAttributes	Dictionary	(2 items)
UNNotificationExtensionCategory	String	myNotificationCategory
UNNotificationExtensionInitialC...	Number	1
NSExtensionMainStoryboard	String	MainInterface
NSExtensionPointIdentifier	String	com.apple.usernotifications.content-extension

4. This category is the name of the category of our notification we set earlier.
5. Update **myNotificationCategory** to **reservationCategory**.

Save the file and switch your target back to the **Let's Eat** app. Build and run the project by hitting the play button (or use *CMD* + *R*). This time, instead of seeing our custom image, we now have the following:



You might notice something slightly off when you pull down on the notification. The notification starts out large and, then, it shrinks down. Inside your `Info.plist` file, there is a property, `UNNotificationExtensionInitialContentSizeRatio`, which is currently set to 1. Changing it to about 0.25 should make this less obvious.

Currently, this custom notification is showing us the custom and default content together. We can fix this by returning to our `Info.plist` inside the `LetsEatContentExtension`:

Inside `NSExtensionAttributes`, add a new item called, `UNNotificationExtensionDefaultContentHidden` and set the type as `Boolean` and the value to `YES`:

Key	Type	Value
▼ Information Property List	Dictionary	(10 items)
Localization native development region	String	en
Bundle display name	String	LetsEatContentExtension
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	XPC!
Bundle versions string, short	String	1.0
Bundle version	String	1
▼ NSExtension	Dictionary	(3 items)
▼ NSExtensionAttributes	Dictionary	(3 items)
UNNotificationExtensionCategory	String	reservationCategory
UNNotificationExtensionInitialContentSizeRatio	Number	0.25
UNNotificationExtensionDefaultContentHidden	Boolean	YES ←
NSExtensionMainStoryboard	String	MainInterface
NSExtensionPointIdentifier	String	com.apple.usernotifications.content-extension

Save the file and build and run the project by hitting the play button (or use *CMD + R*). Once you pull down on the notification, you will see that the default content is now hidden:



You can now update the `MainInterface.storyboard` inside your `LetsEatContentExtension` folder. In this book, we are not going to do this as we have done plenty of storyboarding, and this will give you an opportunity to practice what you learned in this book to create a custom interface for this.

Summary

Notifications in iOS 10 are now even more powerful and really give you the flexibility to create rich custom content with very little work. You learned how to build basic notifications and then we stepped it up a bit more by adding embedded images into our notifications. Lastly, we briefly looked at how to add a custom notification using an extension.

In the next chapter, we will address how we can integrate 3D Touch into our app. It will allow users to quickly jump into our app using shortcuts.

17
Just a Peek

In 2015, when the iPhone 6s and iPhone 6s Plus were announced, Apple also introduced 3D Touch. It uses a taptic engine with haptic feedback, which allows the device to sense the pressure of a touch, thus triggering certain actions. For example, pressing hard on an icon allows us to have a quick action menu.

We will cover the following topics in this chapter:

- Adding 3D Touch quick actions
- Understanding the difference between static and dynamic quick actions
- Adding 3D Touch support inside a Collection View

The first thing we will do for our app is to add quick actions for our app icon.

Adding 3D Touch quick actions

For our app, we are going to add four quick actions (which is the maximum amount that you can have). These actions will do the following:

1. Launch the Map.
2. Launch the locations.
3. Select Los Angeles as a location.
4. Select Las Vegas as a location.

There are two types of quick actions: static and dynamic. Static means that they cannot be changed and dynamic means that they can be. For example, Apple has 3D Touch on their Messages app. If you press hard on the Messages app, you will see one static quick action, New Message, and three dynamic quick actions, the three most texted contacts.

In our app, we will have two static quick actions, launching the Map tab and the locations list, and two dynamic quick actions, launching Los Angeles and Las Vegas as locations. Let's start setting up our quick actions:

1. Right-click on the `Misc` folder inside the `Common` folder and select **New File**.
2. Inside the **Choose a template for your new file screen**, select **iOS** at the top and, then, **Swift File**. Then, hit **Next**.
3. Name this file, `Shortcut`, and hit **Create**.
4. Inside this file, add the following enum after the `import` statement:

```
enum Shortcut: String {
    case openMap
    case openLocations
    case openLosAngeles
    case openLasVegas

    init?(with identifier: String) {
        guard let shortIdentifier = identifier.
components(separatedBy: ".").last else { return nil }
        self.init(rawValue: shortIdentifier)
    }

    var type: String {
        guard let identifier = Bundle.main.bundleIdentifier else {
return "" }
        return identifier + ".\(self.rawValue)"
    }
}
```

5. This enum will be used for our quick actions. As we discussed, we will have four quick actions for our app.

Now, open your `AppDelegate.swift` file. After the `window` variable, add the following:

```
var launchedShortcutItem: UIApplicationShortcutItem?
    static let applicationShortcutUserInfoIconKey =
"applicationShortcutUserInfoIconKey"
```

Here, we have a variable for our shortcut item and a constant for our user info key. When the application launches, we need to check to see if the app was launched using a quick action.

Next, we need to create a method to handle our shortcuts. Add the following after the `checkNotifications()` method:

```
func checkShortCut(_ application: UIApplication, launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    var isPerformingAdditionalDelegateHandling = true
                                                  A
    if let shortcutItem = launchOptions?[UIApplicationLaunchOptionsKey.shortcutItem] as? UIApplicationShortcutItem {

        launchedShortcutItem = shortcutItem
        isPerformingAdditionalDelegateHandling = false
    }

    if let shortcutItems = application.shortcutItems, shortcutItems.isEmpty {
        let laShortcut = UIMutableApplicationShortcutItem(type: Shortcut.openLosAngeles.type, localizedTitle: "Los
B           Angeles", localizedSubtitle: "", icon: UIApplicationShortcutIcon(templateImageName: "shortcut-city"),
            userInfo:nil
        )

C       let lvShortcut = UIMutableApplicationShortcutItem(type: Shortcut.openLasVegas.type, localizedTitle: "Las Vegas",
            localizedSubtitle: "", icon: UIApplicationShortcutIcon(templateImageName: "shortcut-city"), userInfo: nil
        )

        application.shortcutItems = [laShortcut, lvShortcut]
    }

    return isPerformingAdditionalDelegateHandling
                                                  D
}
```

Now, let's break down our code and take a look at each part:

- **Part A:**

  ```
  if let shortcutItem = launchOptions?[UIApplicationLaunchOptionsKey.shortcutItem] as? UIApplicationShortcutItem
  ```

  Here is where we check for a shortcut item. If we have a shortcut item, then the rest of the code inside this `if-let` statement will run.

- **Part B:**

  ```
  let laShortcut = UIMutableApplicationShortcutItem(type: Shortcut.openLosAngeles.type, localizedTitle: "Los Angeles", localizedSubtitle: "", icon: UIApplicationShortcutIcon(templateImageName: "shortcut-city"), userInfo:nil
  )
  ```

  This is our first dynamic shortcut. This will set our current selected location to `Los Angeles`. We also set the icon image here to a custom image that is in our image assets.

- **Part C:**

  ```
  let lvShortcut = UIMutableApplicationShortcutItem(type: Shortcut.openLasVegas.type, localizedTitle: "Las Vegas", localizedSubtitle: "", icon: UIApplicationShortcutIcon(templateImageName: "shortcut-city"), userInfo: nil
  )
  ```

This is our second dynamic shortcut. This will set our current selected location to `Las Vegas`. We also set the icon image here to a custom image that is in our image assets.

- **Part D:**

  `return isPerformingAdditionalDelegateHandling`

  Finally, when we are done, we will return `true` or `false`. `True` will be sent if a shortcut icon was not selected, and `false` will be sent if one was selected.

Now that we have our method created, let's update our return value inside `didFinishLaunchingWithOptions`. Update the return from the current value of `true` to the following:

```
return checkShortCut(application, launchOptions: launchOptions)
```

Next, we need to add a few more methods in order for our application to handle shortcuts. Let's add a method that will handle any shortcut links that are selected. Add the following method after the `checkShortCut()` method:

```
func handleShortCut(_ item: UIApplicationShortcutItem) -> Bool {
    var isHandled = false

A   guard Shortcut(with: item.type) != nil, let shortCutType = item.type as String?, let tabBarController = self.window?.
        rootViewController as? UITabBarController else { return false }

    switch (shortCutType) {

    case Shortcut.openLocations.type:

        tabBarController.selectedIndex = 0
B       let navController = self.window?.rootViewController?.childViewControllers.first as! UINavigationController
        let viewController = navController.childViewControllers.first as! ExploreViewController
        viewController.performSegue(withIdentifier: "locationList", sender: self)

        isHandled = true

        break

    case Shortcut.openMap.type:

C       tabBarController.selectedIndex = 1
        isHandled = true

        break

    case Shortcut.openLosAngeles.type:

        let navController = self.window?.rootViewController?.childViewControllers.first as! UINavigationController
        let viewController = navController.childViewControllers.first as! ExploreViewController
        viewController.selectedCity = "Los Angeles"
D       tabBarController.selectedIndex = 1
        tabBarController.selectedIndex = 0
        isHandled = true

        break

    case Shortcut.openLasVegas.type:

        let navController = self.window?.rootViewController?.childViewControllers.first as! UINavigationController
        let viewController = navController.childViewControllers.first as! ExploreViewController
        viewController.selectedCity = "Las Vegas"
E       tabBarController.selectedIndex = 1
        tabBarController.selectedIndex = 0
        isHandled = true

        break

    default:
        break
    }

    return isHandled
}
```

Let's break down this code so you better understand the different parts of the code:

- **Part A:**

```
guard Shortcut(with: item.type) != nil, let shortCutType =
item.type as String?, let tabBarController = self.window?.
rootViewController as? UITabBarController else { return false }
```

  Here, we are guarding to make sure that we have a shortcut item, shortcut type, and a **Tab Bar** Controller. If we do not have any of these things, we will return `false` and not go any further.

- **Part B:**

```
tabBarController.selectedIndex = 0
let navController = self.window?.rootViewController?.
childViewControllers.first as! UINavigationController
let viewController = navController.childViewControllers.first as!
ExploreViewController
viewController.performSegue(withIdentifier: "locationList",
sender: self)
```

  This shortcut will allow us to launch the Location List View. We are setting the selected index of the **Tab Bar** Controller and then using the `performSegue` to enable the modal to appear.

- **Part C:**

```
tabBarController.selectedIndex = 1
isHandled = true
```

  This shortcut will allow us to go directly to the **Map** tab.

- **Part D:**

```
let navController = self.window?.rootViewController?.
childViewControllers.first as! UINavigationController
let viewController = navController.childViewControllers.first as!
ExploreViewController
viewController.selectedCity = "Los Angeles"

tabBarController.selectedIndex = 1
tabBarController.selectedIndex = 0
isHandled = true
```

  This shortcut will allow us to launch the Explore View with Los Angeles already selected.

- **Part E:**

```
let navController = self.window?.rootViewController?.
childViewControllers.first as! UINavigationController
let viewController = navController.childViewControllers.first as!
ExploreViewController
viewController.selectedCity = "Las Vegas"

tabBarController.selectedIndex = 1
tabBarController.selectedIndex = 0
isHandled = true
```

  This shortcut will allow us to launch the **Explore View** with `Las Vegas` already selected.

Now that we can have a method for handling any shortcut links, add the following method after the `handleShortCut()` method:

```
func application(_ application: UIApplication, performActionFor
shortcutItem: UIApplicationShortcutItem, completionHandler: @escaping
(Bool) -> Void) {
    let handledShortCutItem = handleShortCut(shortcutItem)

    completionHandler(handledShortCutItem)
}
```

This method gets called every time a shortcut action is performed.

Finally, we need to add code to check for when the app becomes active. Find the `applicationDidBecomeActive()` method and add the following inside the curly braces:

```
guard let item = launchedShortcutItem else { return }
_ = handleShortCut(item)

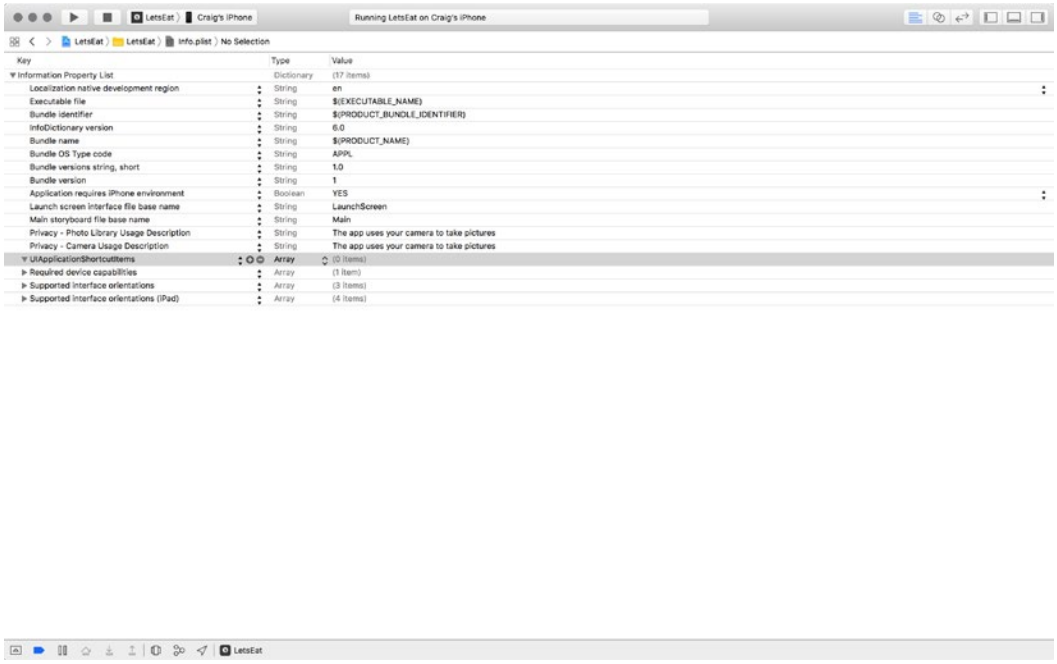
launchedShortcutItem = nil

if (application.applicationIconBadgeNumber != 0) {
application.applicationIconBadgeNumber = 0
}
```

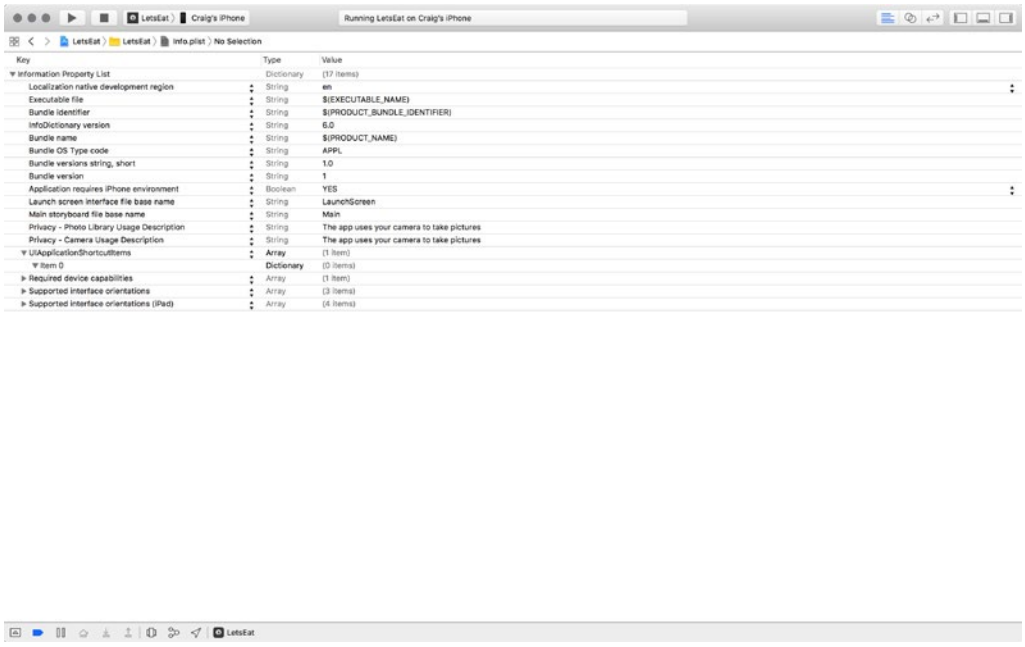
Here, is where we handle any shortcut actions. In addition, we are checking whether the badge icon is set to a number other than 0; if so, we reset it to 0.

If you are keeping track, you will see that we have only added two items so far—our dynamic items. We still need to add our static items; however, these will actually be added to your `Info.plist`.

Open the `Info.plist` in the `Assets` folder of the **Navigator** panel. Hover over **Privacy - Camera Usage Description** and, then, click on the + button to add the **UIApplicationShortcutItems** key. Make the **type** an **Array**:

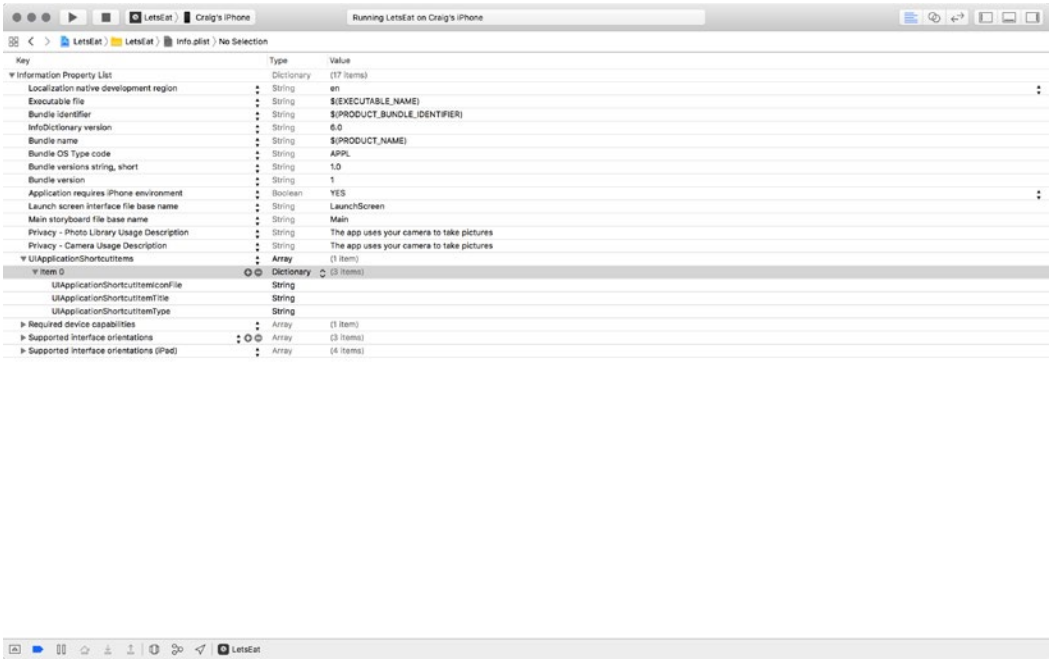


Next, hover over **UIApplicationShortcutItems** and click on the + button in order to add an item to this **Array**. Change the **type** to **Dictionary**:

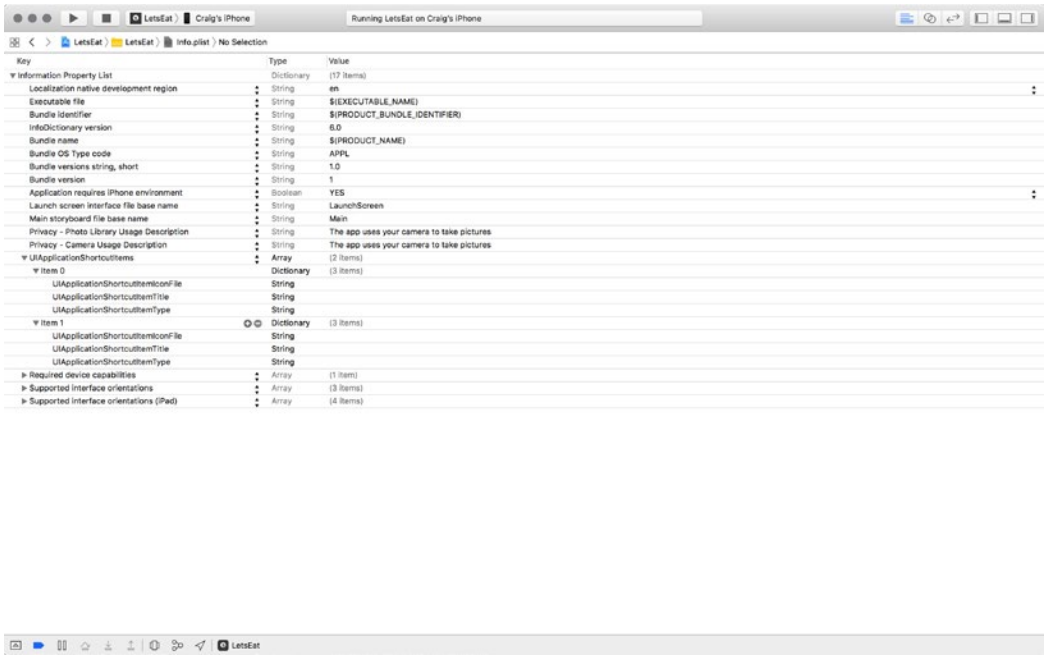


Hover over **Item 0** and click on the + button three times to add three Strings. Name the three keys the following:

```
UIApplicationShortcutItemIconFile
UIApplicationShortcutItemTitle
UIApplicationShortcutItemType
```



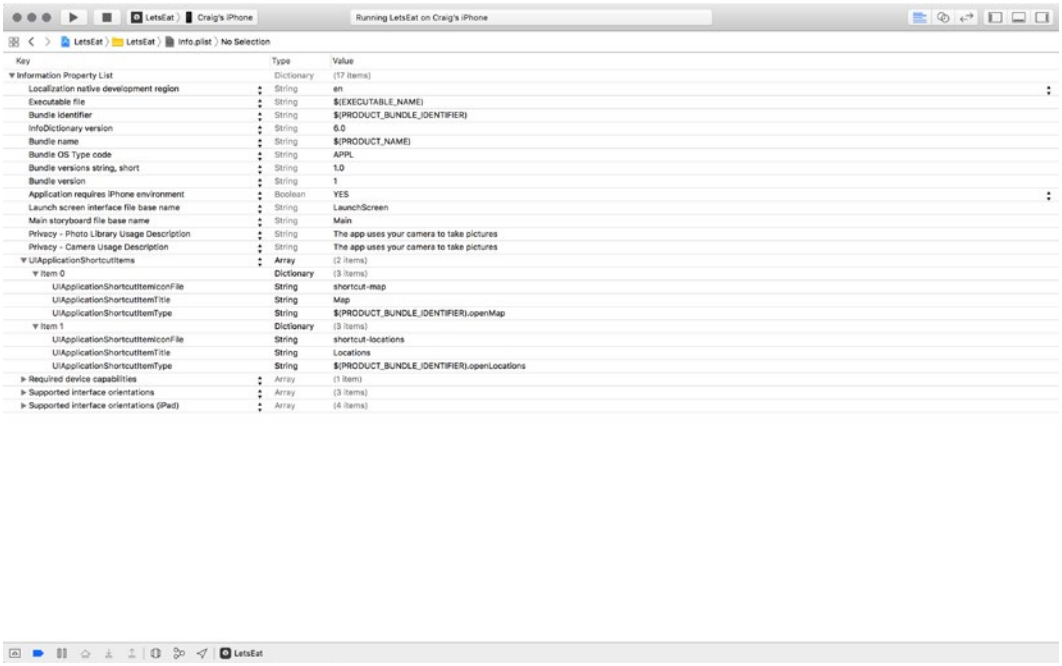
Now, copy **Item 0** (*CMD + C*) and paste it into **UIApplicationShortcutItems**. You should now have **Item 0** and **Item 1**, each with the same three Strings:



Next, set their values to the following:

- **Item 0**
    - **UIApplicationShortcutItemIconFile**: shortcut-map
    - **UIApplicationShortcutItemTitle**: Map
    - **UIApplicationShortcutItemType**: $(PRODUCT_BUNDLE_IDENTIFIER).openMap
- **Item 1**
    - **UIApplicationShortcutItemIconFile**: shortcut-location
    - **UIApplicationShortcutItemTitle**: Locations
    - **UIApplicationShortcutItemType**: $(PRODUCT_BUNDLE_IDENTIFIER).openLocations

When you are finished, your file should look like the following:

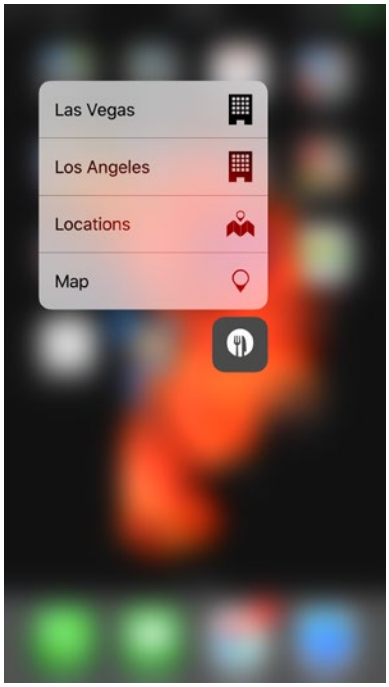


Save your file and build and run the project on your device. If you have a MacBook or a MacBook Pro with a force touch mouse, you can run this in the simulator.



If you are building for your phone, you might encounter errors. These errors occur because you have not built the framework for your phone. Simply switch to the framework, then hit *CMD + B*, switch to your iMessages app and do the same. Then, build and run the project on your device.

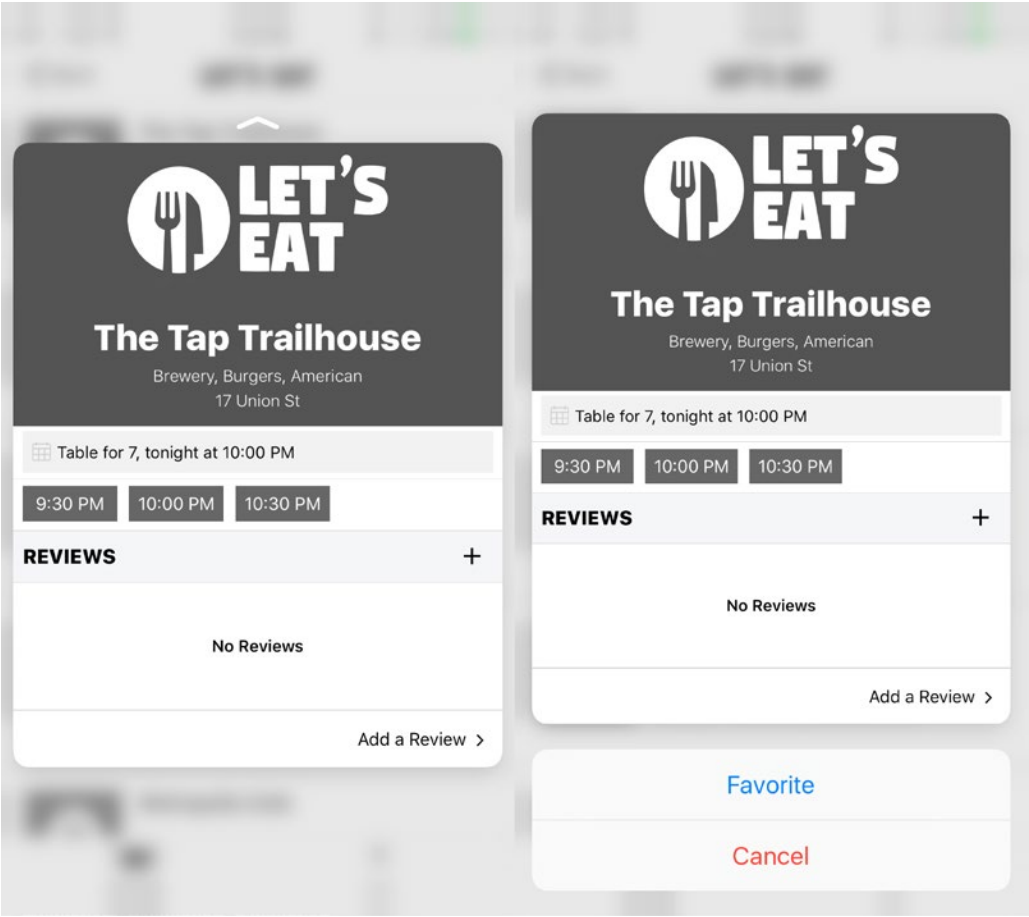
Once your app launches, hit *command* + *H* (if you are in a simulator) or the home button on your device. 3D Touch the **Let's Eat** app icon, and you should now see the following:



If you select Los Angeles or Las Vegas, you will see that, at the top, the location is now set for you. If you select the Map shortcut, you will be taken to the Map tab. If you select the Locations shortcut, you will be in the locations list. We have now added 3D Touch quick actions to our app. Let's add 3D Touch to one more place.

Adding favorites

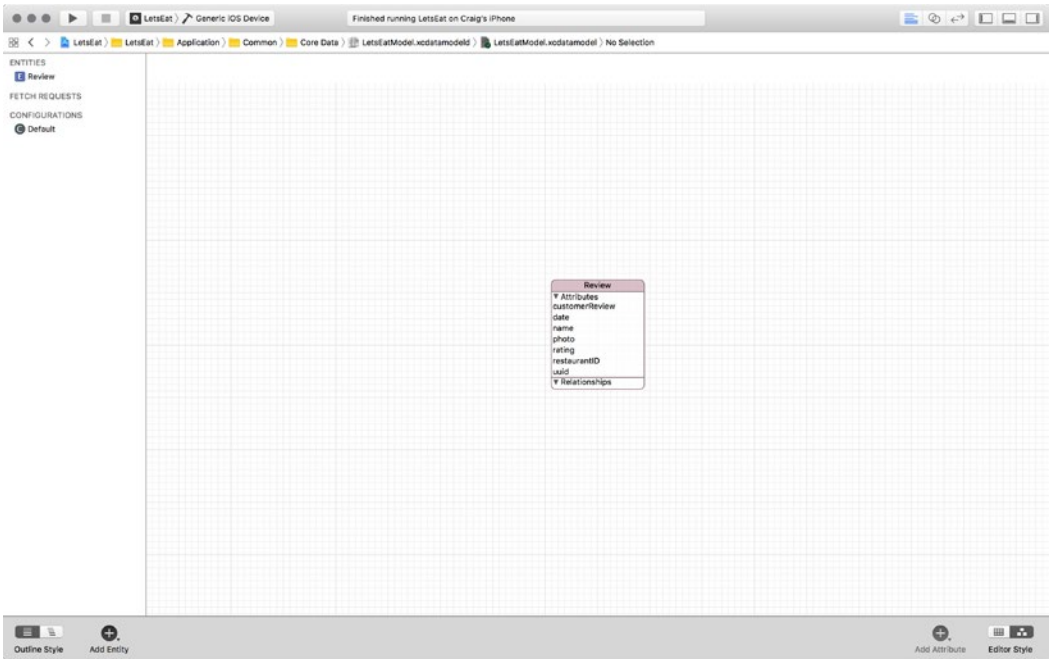
A nice feature for our app would be to use 3D Touch to allow users to add favorites within our Restaurant List View. We already have the heart in our Restaurant Detail page. Therefore, let's add 3D Touch to the heart to add favorites. This is how we want it to look when we are finished:



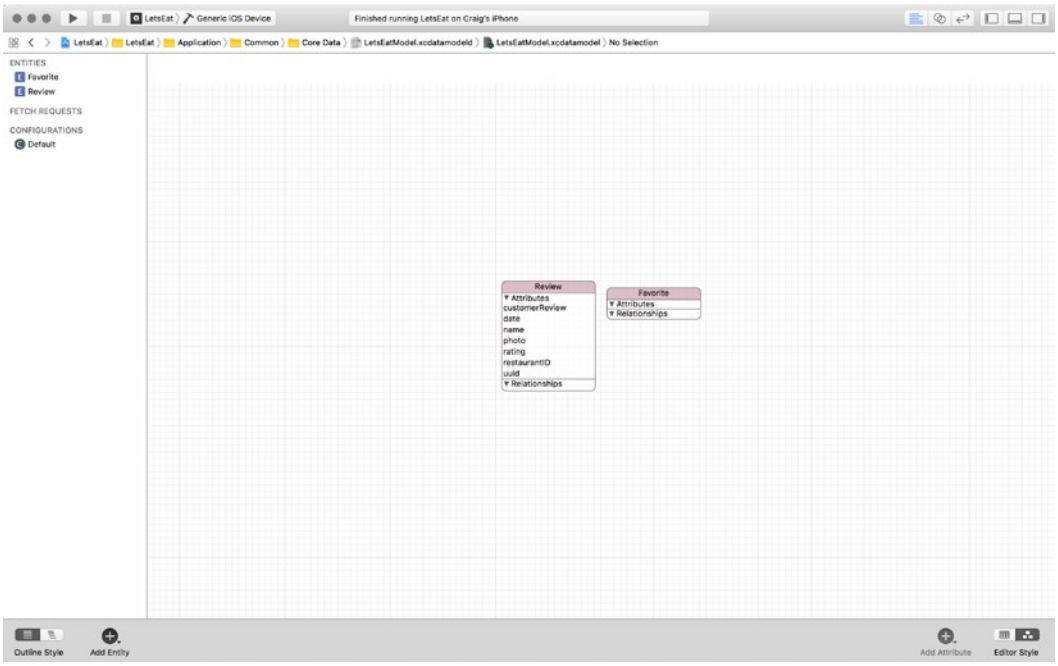
First thing we need to do is add a new model object so that we can save our restaurant favorites to Core Data.

Creating a new model object

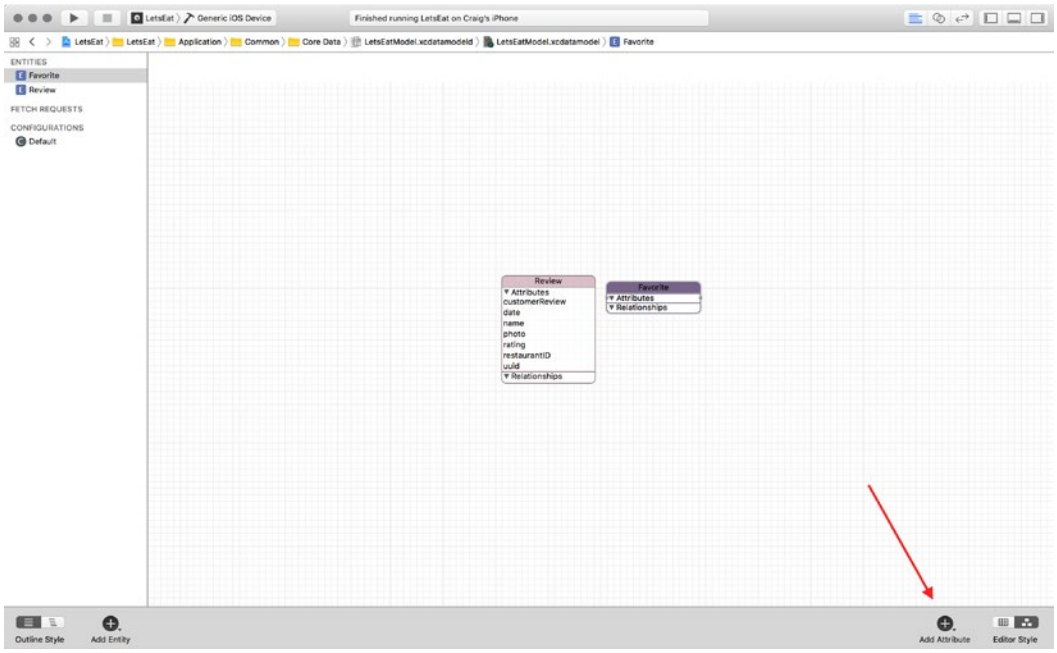
1. In the **Navigator** panel, open the `LetsEatModel.xcdatamodeld` file that can be found in the `Core Data` folder in the `Common` folder.
2. Make sure that you have the Graph Style selected:



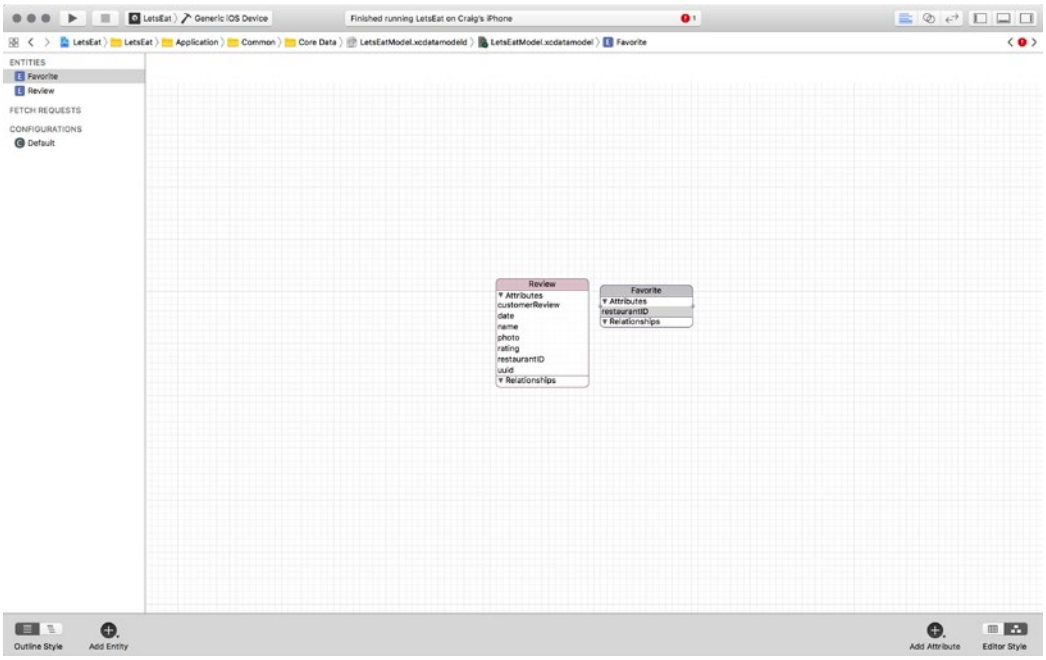
3. Click on the **+** button for **Add Entity** and, then, double-click on the text, **Entity**, and update it to **Favorite**:



4. Next, ensure that the **Favorite Entity** is selected and click on the **+** button for **Add Attribute**:

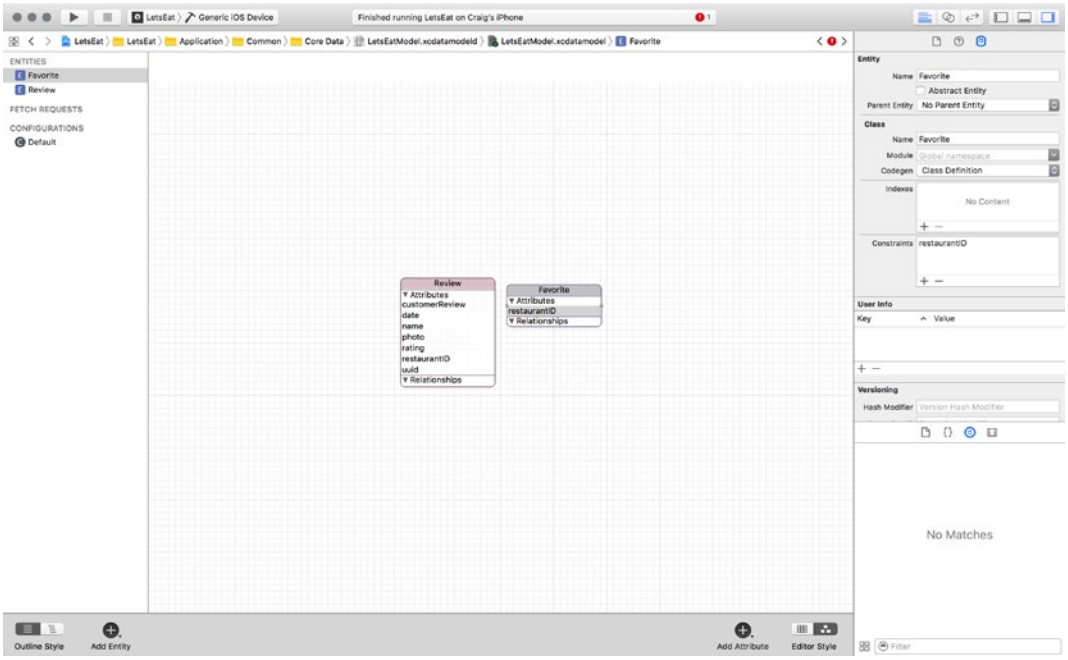


5. In the box, in the center of the screen, under **Attributes**, you should now see the word attribute. Double-click on attribute and change it to `restaurantID`:

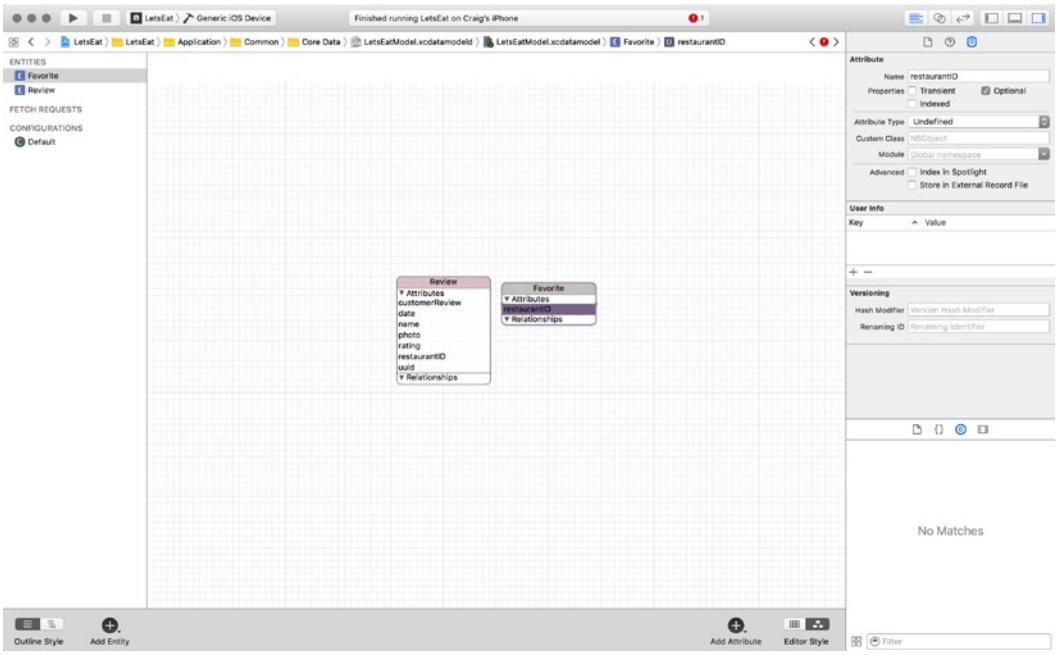


6. Next, select the **Favorite Entity** again and, then, the **Data Model Inspector** in the **Utilities** panel.
7. Under **Class**, update the following:
    - **Name**: Favorite
    - **Codegen**: Class Definition
    - **Constraints**: `restaurantID` (you add this constraint by hitting the + button in Constraints and then replacing the default constraint that auto-populates with this new constraint)

8. Your **Entity** settings should now look like the following:



Finally, select the `restaurantID` attribute; under the **Data Model Inspector** in the **Utilities** panel, select **Integer 32** under **Attribute Type**. The error regarding this attribute should now disappear.



Now, build the project using *CMD* + *B*. This will create our **Favorite** class that we created in **Core Data**. You will not see the file anywhere, but it has been created.

Updating our Core Data manager

Now that we have our **Favorite Entity** created, we need to update our **Core Data** Manager in order to actually save restaurants as favorites. Inside the `CoreDataManager.swift` file, add the following before the last curly brace:

```
func addFavorite(by restaurantID:Int) {
    let item = Favorite(context: container.viewContext)
    item.restaurantID = Int32(restaurantID)

    save()
}
```

This method creates a **Favorite** object and, then, calls the `save()` method.

Now, let's add one more method to this file by adding the following before the last curly brace:

```
func isFavorite(with identifier:Int) -> Bool {
    let moc = container.viewContext
    let request:NSFetchRequest<Favorite> = Favorite.fetchRequest()
    let predicate = NSPredicate(format: "restaurantID = %i",
Int32(identifier))
    request.predicate = predicate

    do {
        let count = try moc.count(for: request)

        if count == 0 { return false }
        else { return true }

    } catch {
        fatalError("Failed to fetch reviews: \(error)")
    }
}
```

In this method, we are going to `fetch` for a favorite restaurant by passing a `restaurantID`. The method will check Core Data; if we get back data, that restaurant will be set as a favorite. Otherwise, if we get back no data, we return `false`. We can now save favorite restaurants.

Next, open `RestaurantListViewController.swift` file and define a new extension after the extension we created for our **Collection View** by adding the following code:

```
extension RestaurantListViewController: UIViewControllerPreviewingDelegate {

    func previewingContext(_ previewingContext: UIViewControllerPreviewing, viewControllerForLocation location: CGPoint) ->
        UIViewController? {
        let restaurantDetail : UIStoryboard = UIStoryboard(name: "RestaurantDetail", bundle: nil)
    A   guard let indexPath = collectionView?.indexPathForItem(at: location), let cell = collectionView?.cellForItem(at:
            indexPath), let detailVC = restaurantDetail.instantiateViewController(withIdentifier: "RestaurantDetail") as?
            RestaurantDetailViewController else { return nil }

        selectedRestaurant = manager.restaurantItem(at: indexPath)
        detailVC.selectedRestaurant = selectedRestaurant
    B   detailVC.preferredContentSize = CGSize(width: 0.0, height: 528)
        previewingContext.sourceRect = cell.frame

        return detailVC
    }

C   func previewingContext(_ previewingContext: UIViewControllerPreviewing, commit viewControllerToCommit: UIViewController)
        {

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

Let's discuss what we just added:

- **Part A:**

```
let restaurantDetail : UIStoryboard = UIStoryboard(name:
"RestaurantDetail", bundle: nil)
guard let indexPath = collectionView?.indexPathForItem(at:
location), let cell = collectionView?.cellForItem(at: indexPath),
let detailVC = restaurantDetail.instantiateViewController(withId
entifier: "RestaurantDetail") as? RestaurantDetailViewController
else { return nil }
```

  First, we get an instance of our `RestaurantDetail.storyboard`. Then, we obtain a current `index` path. Once we have an `index` path, we check that we have a cell. Finally, we create an instance to our `RestaurantDetailViewController`.

- **Part B:**

```
selectedRestaurant = manager.restaurantItem(at: indexPath)
detailVC.selectedRestaurant = selectedRestaurant
detailVC.preferredContentSize = CGSize(width: 0.0, height: 450)
previewingContext.sourceRect = cell.frame
```

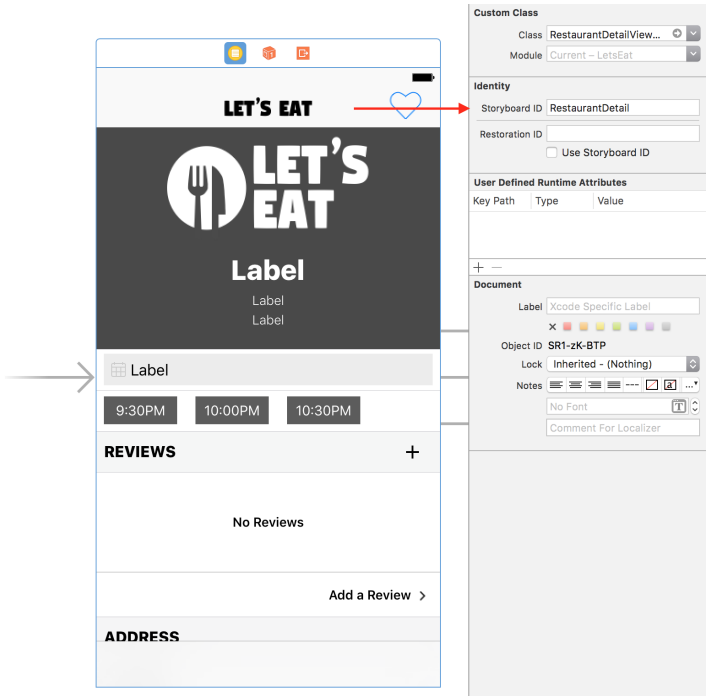
  Here, we set our `selectedRestaurant`, and, then, we pass the `selectedRestaurant` over to the detail View. We then set the height that we want and pass the cell frame to the previewing context.

- **Part C:**

```
func previewingContext(_ previewingContext:
UIViewControllerPreviewing, commit viewControllerToCommit:
UIViewController) {
   show(viewControllerToCommit, sender: self)
}
```

  This method is called so that we can prepare the presentation of the **View Controller**, which here is the **Commit View** Controller. In our case, we prepare the `RestaurantDetailViewController` to be shown (or popped).

Next, open the `RestaurantDetail.storyboard` and select the `RestaurantDetailViewController`. Open the **Identity Inspector** in the **Utilities** panel and, in **Storyboard ID** under **Identity**, add **RestaurantDetail**. Then, hit *Enter*:



Now, return to the `RestaurantListViewController.swift` file and add the following code after the `showRestaurantDetail()` method:

```
func setup3DTouch() {
    if( traitCollection.forceTouchCapability == .available){
        registerForPreviewing(with: self, sourceView: view)
    }
}
```

Here, we need to ensure that our `RestaurantListViewController` can accept 3D Touch. We need to call this inside the `initialize()` method after our `if`-statement. We've finished our setup inside the `RestaurantListViewController`.

Finally, we need to update our `RestaurantDetailViewController.swift` file. Open this file and add the following variable above the `viewDidLoad()` method:

```
override var previewActionItems: [UIPreviewActionItem] {

    let favorite = UIPreviewAction(title: "Favorite", style: .default)
{ (action, viewController) -> Void in
        let manager = CoreDataManager()
        if let id = self.selectedRestaurant?.restaurantID { manager.
addFavorite(by: id) }
    }

    let cancel = UIPreviewAction(title: "Cancel", style: .destructive)
{ (action, viewController) -> Void in
        print("You hit cancel")
    }

    return [favorite, cancel]
}
```

This variable will allow us to have two actions, **Favorite** and **Cancel**, when we peek at a restaurant. If the user taps **Favorite**, we will get the `restaurantID` and save it to **Core Data**. If the user taps **Cancel**, we will dismiss the **View**.

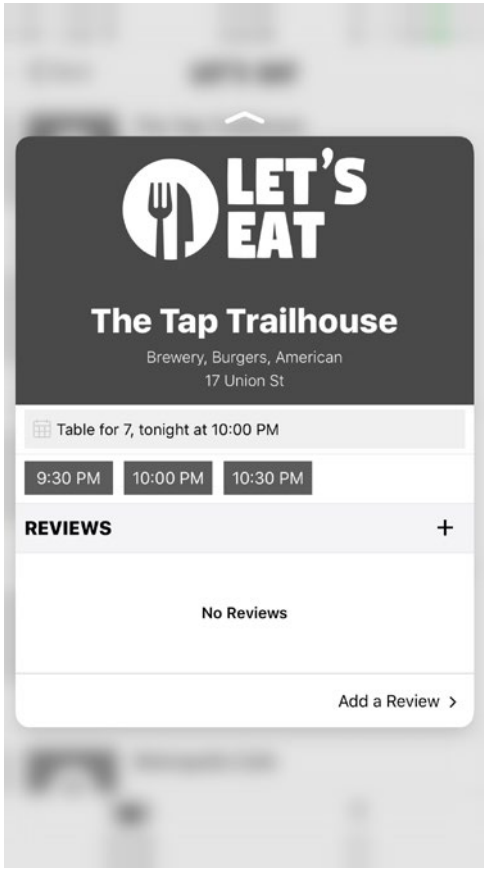
Next, add the following code above the `checkReviews()` method:

```
func checkFavorites() {
    let manager = CoreDataManager()
    if let id = selectedRestaurant?.restaurantID {
        let isFavorite = manager.isFavorite(with: id)
        let btnImage = UIButton()
        btnImage.frame = CGRect(x: 0, y: 0, width: 30, height: 30)
        btnImage.addTarget(self, action: #selector(getter:
UIDynamicBehavior.action), for: .touchUpInside)

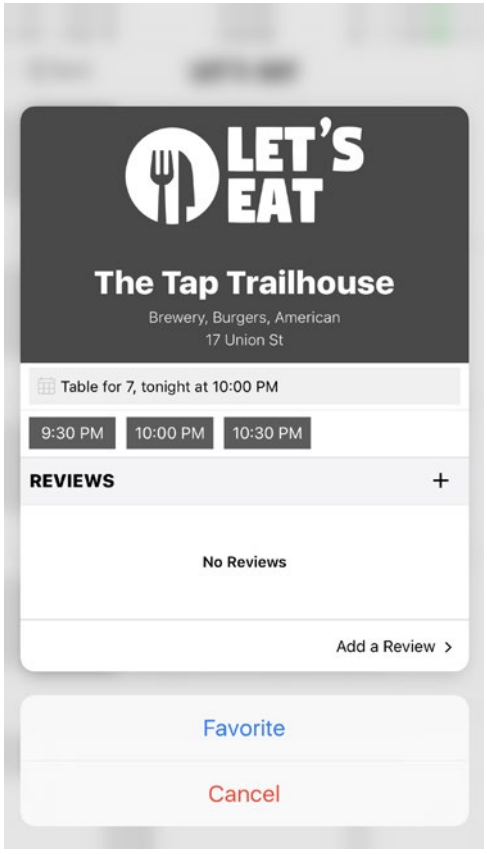
        if isFavorite {
            btnImage.setImage(UIImage(named: "heart-selected"), for:
.normal)
            btnHeart.customView = btnImage
        }
        else {
            btnImage.setImage(UIImage(named: "heart-unselected"), for:
.normal)
            btnHeart.customView = btnImage
        }
    }
}
```

This method will now be run whenever we go to a detail **View**. First, we check **Core Data** to see if the current restaurant is a favorite. If it is a favorite, we will show a filled-in heart; if it is not, we will show a heart with just an outline. This method will be called after the `checkReviews()` method inside the `viewDidAppear()` method.

Build and run the project by hitting the play button (or use *CMD* + *R*). When you get to the restaurant list, force touch one of the restaurant items and you will see the following:



If you swipe up while force touching, you will see that we now have two buttons:



If you tap **Cancel**, it will dismiss the **View**. If you tap **Favorite** and select the same restaurant, you will now see that the heart will change to a filled-in one.

Summary

We are officially done with the building of our app. In this chapter, you learned about the two different types of 3D Touch quick actions that we can add to our app. We also added 3D Touch support to our Collection View. Our restaurant list now has 3D Touch support so that we can add favorite restaurants from the restaurants list.

It is now time to move onto the most exciting part of this app and that is getting our app into the App Store. In the next chapter, we will discuss everything you will need to know regarding how to submit your app to the App Store.

18
Beta and Store Submission

You have come a long way; from learning about Xcode to building an entire app. But this process would not be complete without actually learning how to submit the app to the App Store. This process may seem like a lot when doing this for the first time, but it becomes easier and second nature after a while.

When I submitted my first app, I was extremely nervous. I remember the relief I felt after having submitted the app, but then, I was constantly checking the site and my e-mails for that approval e-mail. I heard many stories of people who spent so much time working on an app only to have it rejected. These fears are understandable, but know that Apple wants you to succeed. Even if your app gets rejected (and my first one did), it is usually not so bad.

My first app was a sports app, and it was rejected for two reasons. First, in Apple's eyes, the logo for my app was too close to an NFL logo. To address this, I simply made a generic logo with the initials of the app. Second, the image quality of my images were not up to par. Therefore, I obtained better images. Then, I resubmitted my app; within a couple of days, my app was approved. It is almost a certainty that you will encounter rejections, even when you have been doing this for a while. Take comfort in the fact that you can address any issues and resubmit in order to have your app approved for the App Store.

In this chapter, we will cover everything you need to know about getting your app into the Store. In Xcode 8, a lot of things are done for you behind the scenes; however, the goal of this chapter is to show you how you can set up things on your own. You will need a developer account in order to follow along with these steps. Go to `www.developer.apple.com/programs/` if you would like to purchase a developer account.

We cover the following topics in this chapter:

- Creating a bundle identifier
- Creating a Certificate Signing Request

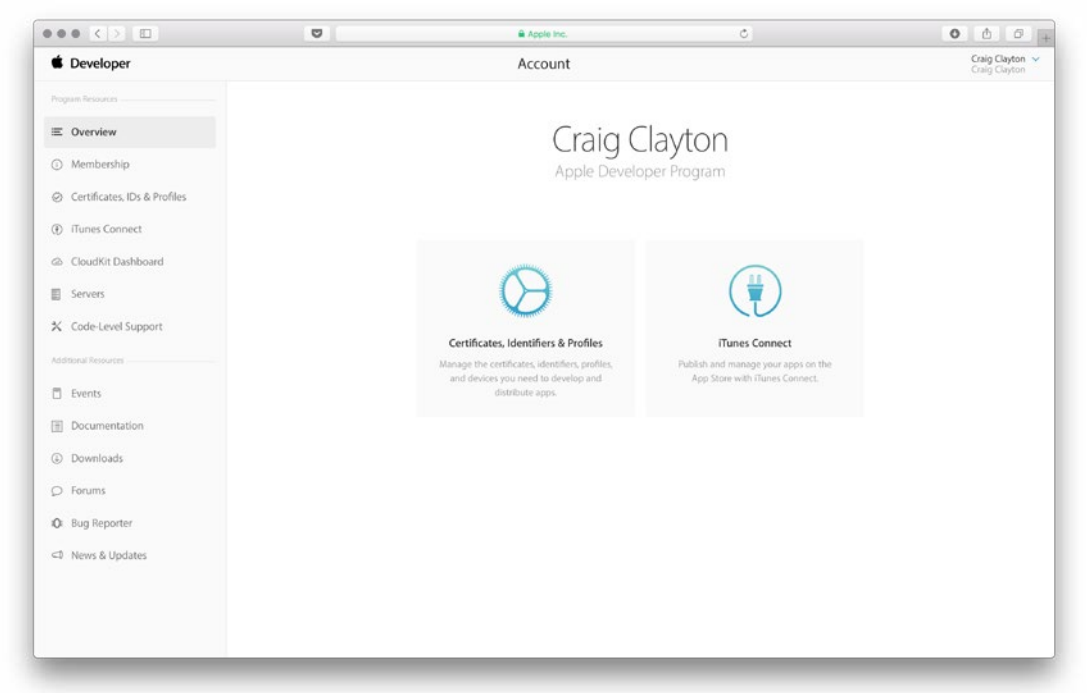
- Creating production and development certificates
- Creating production and development provisioning profiles
- Creating an App Store listing
- Making the release build and submitting to the App Store
- Conducting internal and external testing

This chapter is set up for you to use as you need it. It is not meant to be done in order as with the other chapters in this book. For example, you may need to create a bundle identifier and then need to know how to add external testers. Use this chapter as a resource for when you need to do one of these tasks.

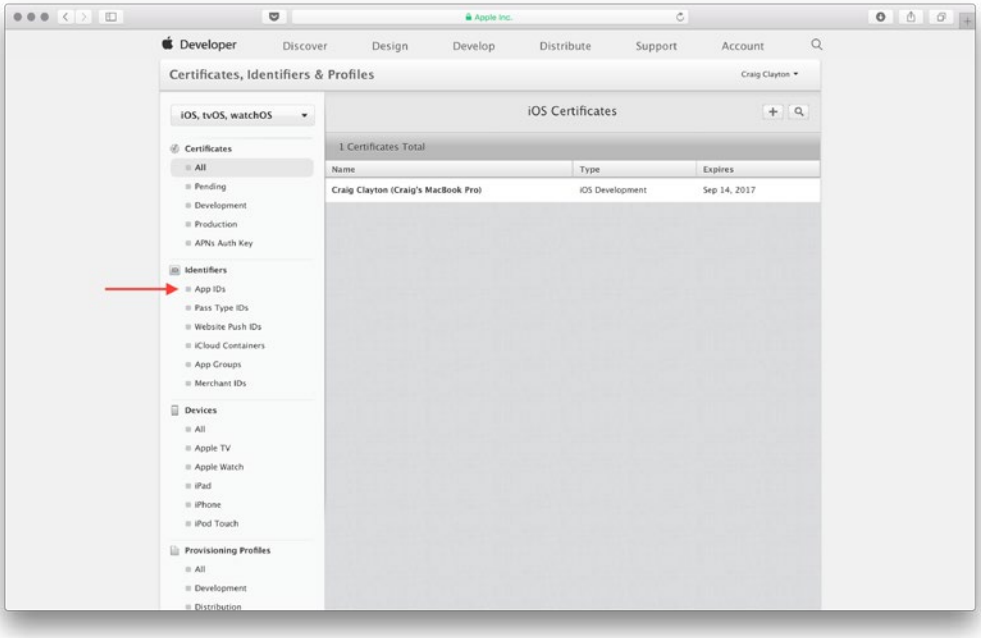
Creating a bundle identifier

When we created our project, we talked about the bundle identifier (also known as your App ID). This bundle identifier is used to identify your app; therefore, it must be unique. Let's proceed with the following steps:

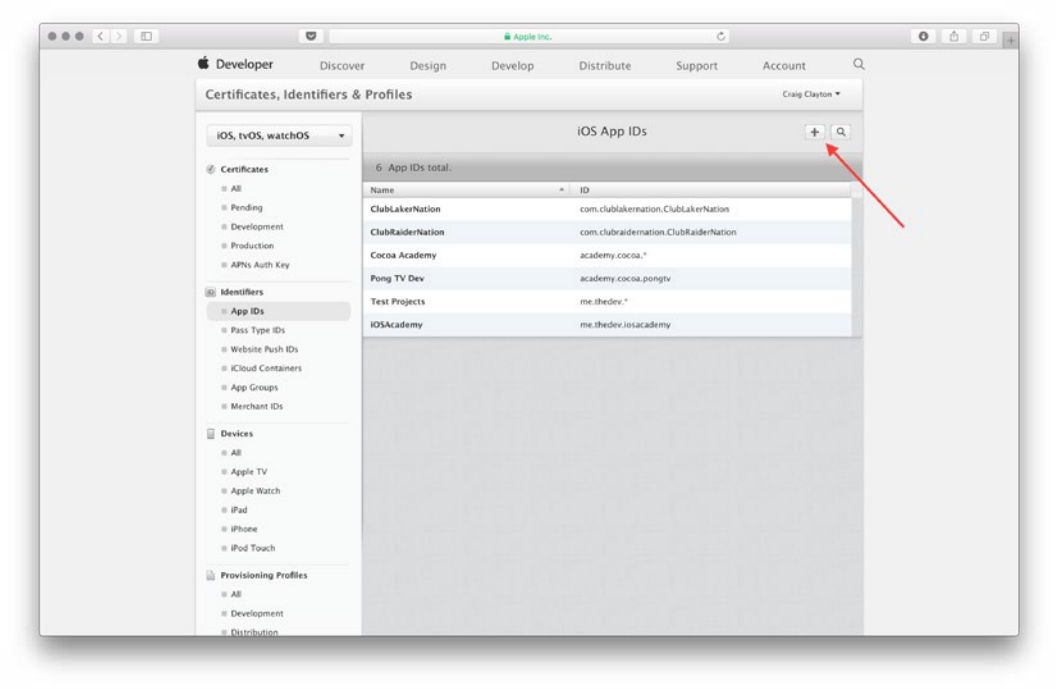
1. Login to the Apple developer account, and you will see the following screen:



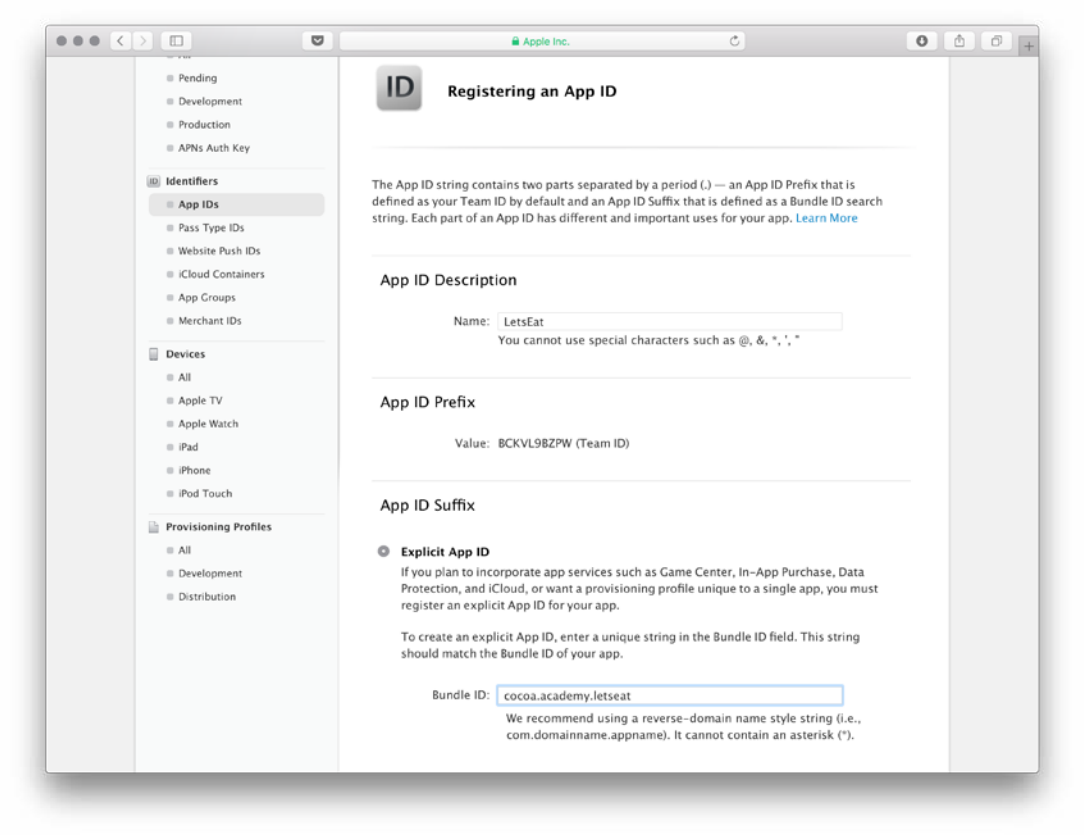
2. Click on **Certificates, Identifiers & Profiles**.
3. Then, under **Identifiers**, click on **App IDs**:



4. Click on the **+** at the top right of the screen:

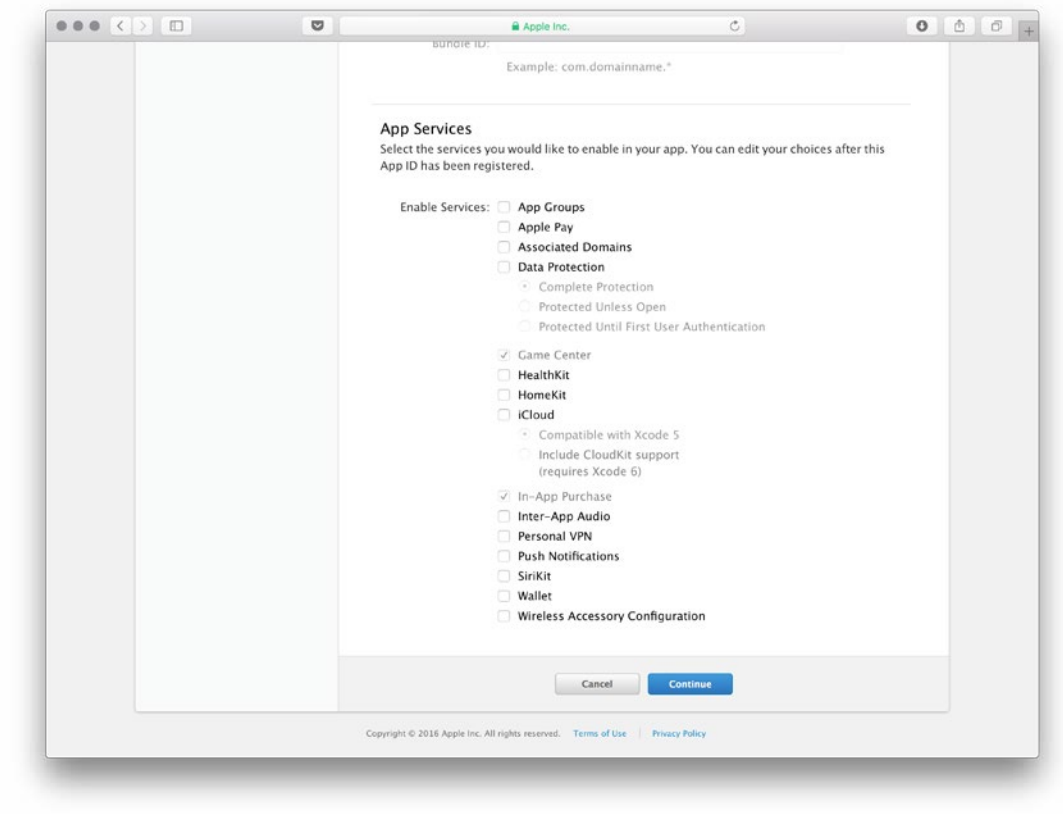


5. The **Registering an App ID** screen will appear:

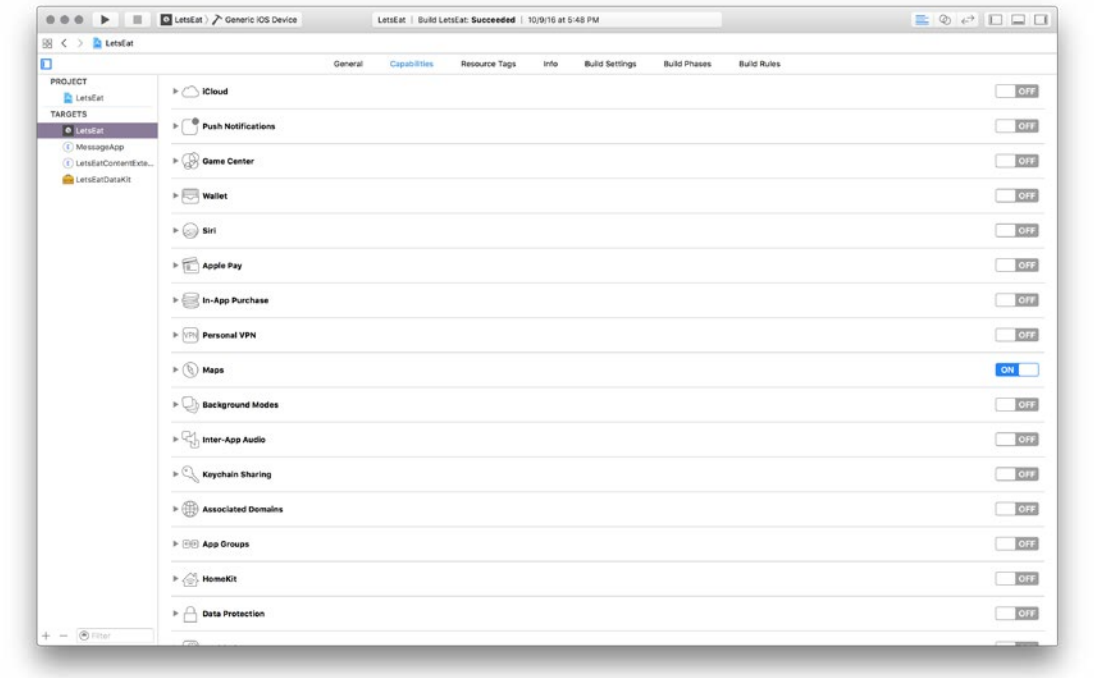


6. In the top part of the **Registering an App ID** preceding screen, add the following:
    - **Name under App ID Description**: `LetsEat`
    - **Explicit App ID under App ID Suffix**: selected
    - **Bundle ID under App ID Suffix**: enter yours
7. Make sure that the **Bundle ID** follows the standard naming convention: `com.yourcompanyname.letseat`. Your **Bundle ID** should be the same ID that you set up when we created the project. For example, mine is `cocoa.academy.letseat`.
8. Next, in the bottom part of the **Registering an App ID** screen shown here, select the **App Services** that the app requires and, then, click on **Continue**.

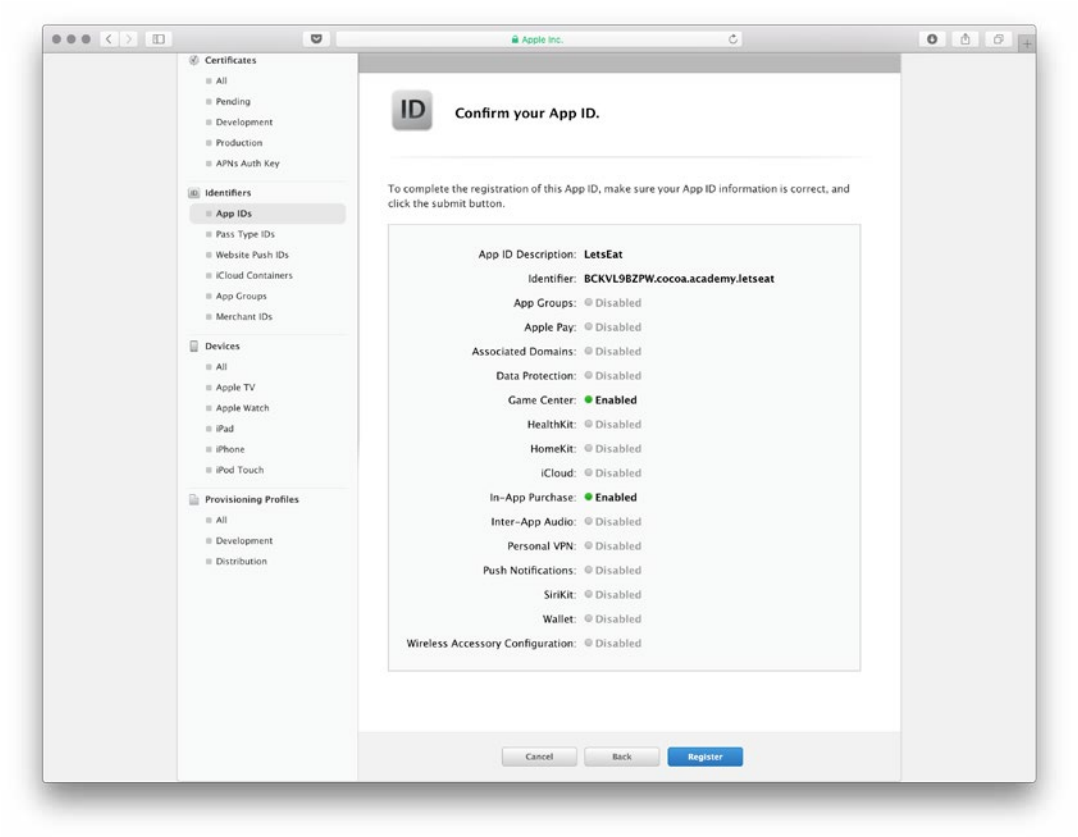
9. Our project does not have any **App Services**, but this is where you would set them if a future app requires them:



10. If you decide later to add **App Services**, you can do so inside **Xcode**. You would select the project under **Targets** and, then, select the **Capabilities** tab and modify as necessary:



11. After verifying your **App ID** information, click on **Register**:

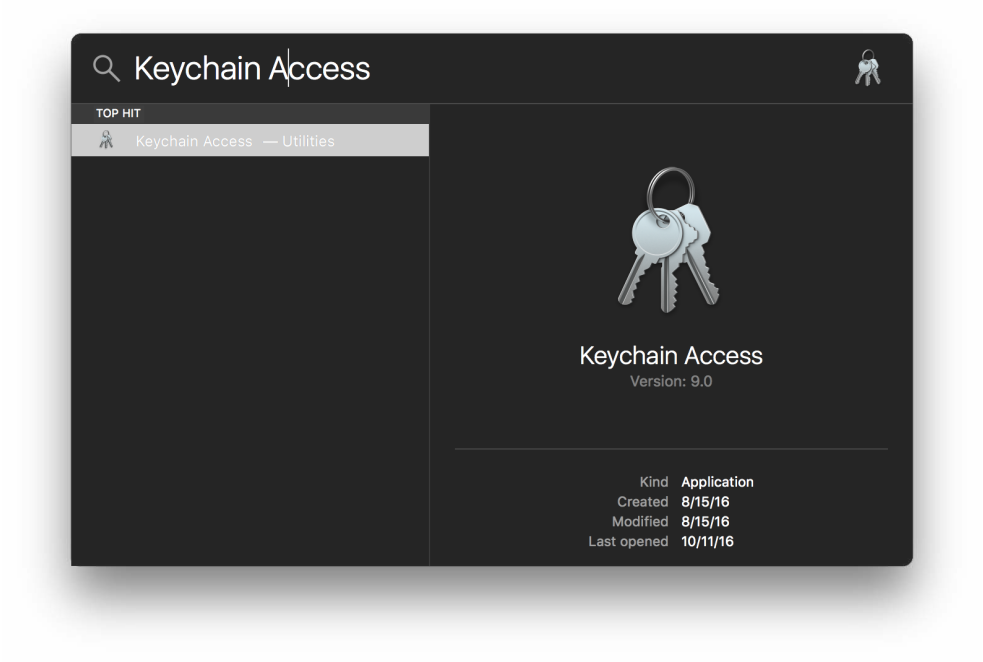


Your App ID is now created. Let's look at what certificates are and how to use them.

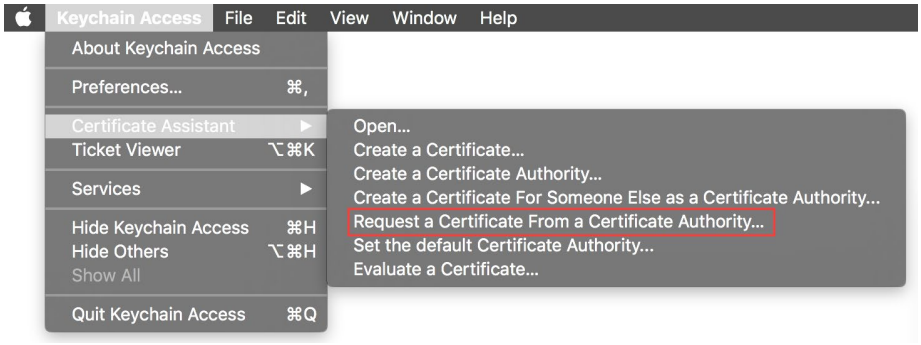
Creating a Certificate Signing Request

Whenever you work on a project, you will need to create a Certificate Signing Request (CSR). You create this certificate on your computer and then upload it into the Apple developer account. Then, you download this file and open it into Keychain Access when you are done. Let's create one certificate for production (for the App Store) and one certificate for development (for building locally):

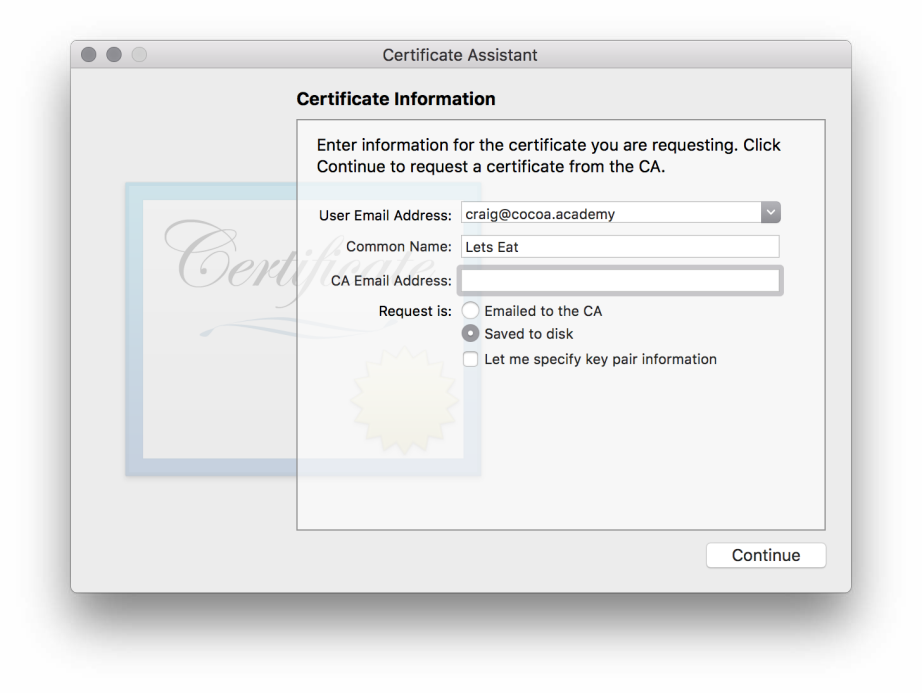
1. Open **Keychain Access** (which you can find by clicking on the search icon in the upper-right corner of your menu bar and typing `Keychain Access` into the search bar):



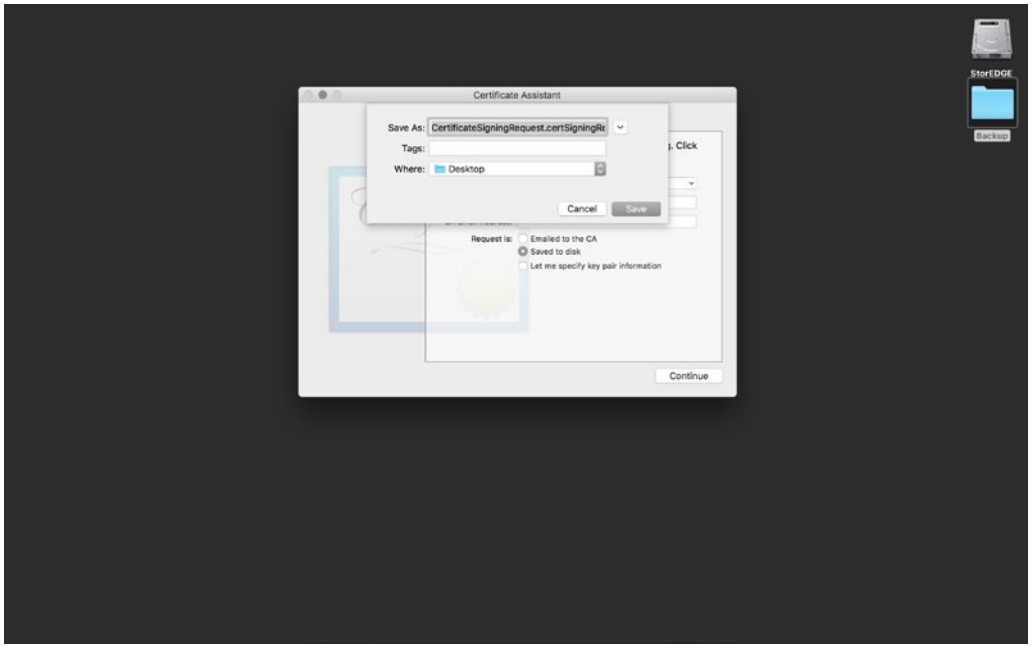
2. In the menu bar, while in **Keychain Access**, navigate to **Keychain Access** | **Certificate Assistant** | **Request a Certificate From a Certificate Authority**:



3. Enter your e-mail address for **User Email Address** and the App name for **Common Name**. Then, select **Saved to disk** under **Request is**:



4. Then, click on **Continue**.
5. In the screen that appears, enter the certificate name, select a save location and click on **Save**:



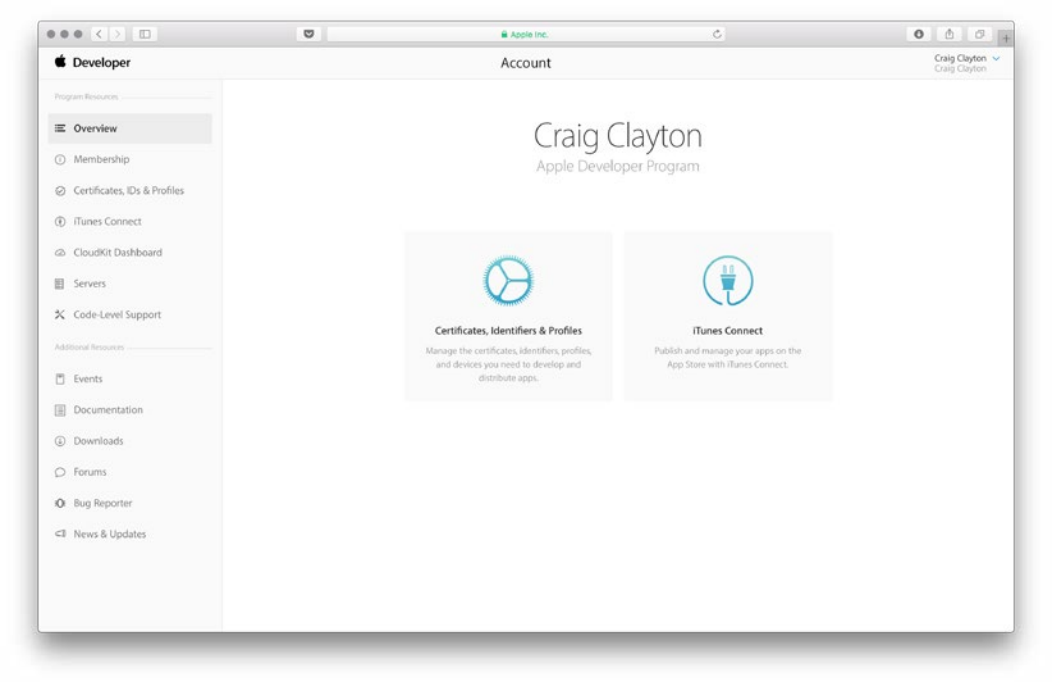
6. Click on **Done**, export the certificate, and save it to your computer:



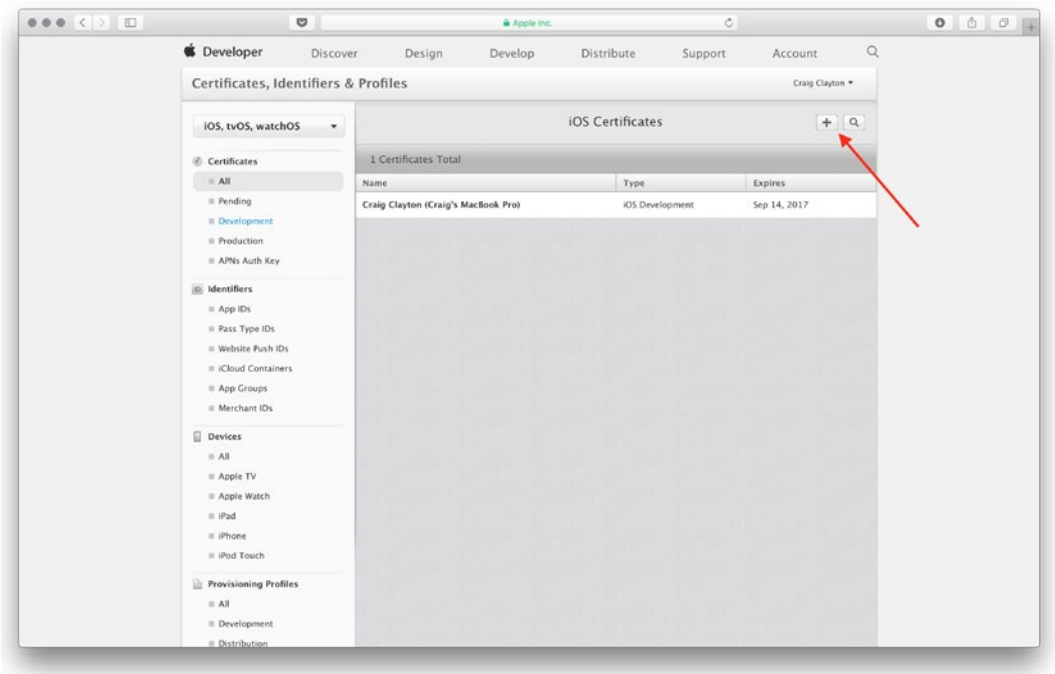
Creating production and development certificates

We need to create production and development certificates. Production certificates are used for the App Store. Development certificates are used to verify that you are a team member allowing apps signed by you to launch on a device. Remember that Xcode 8 can now handle this for you, but knowing the process is still useful. Let's start by creating a production certificate first:

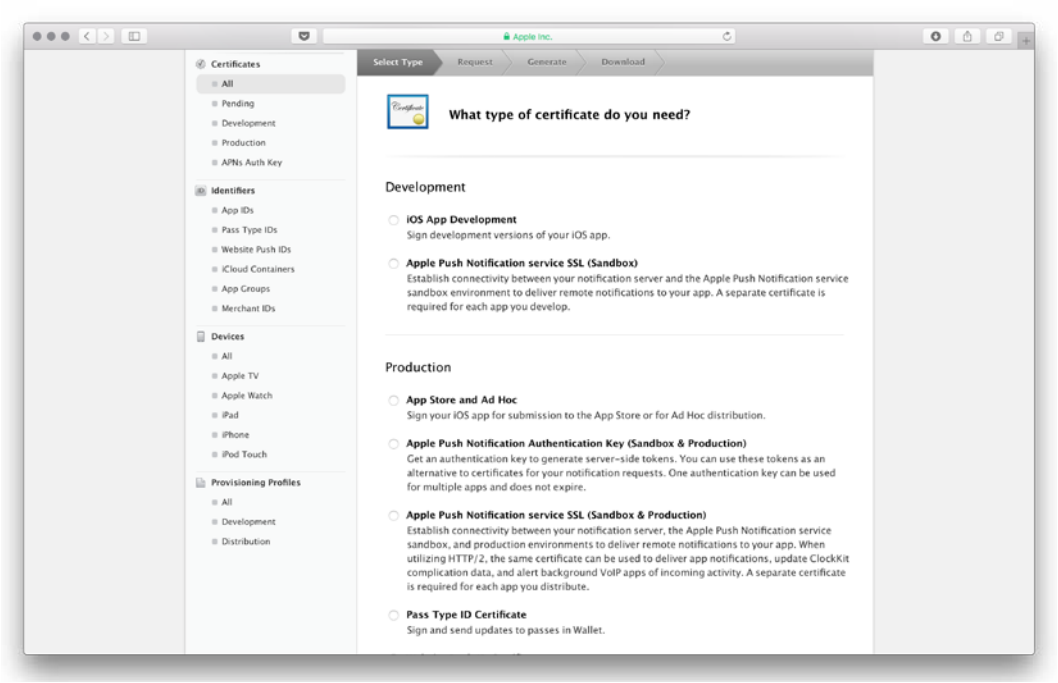
1. Log in to the Apple developer account, and you will see the following screen:



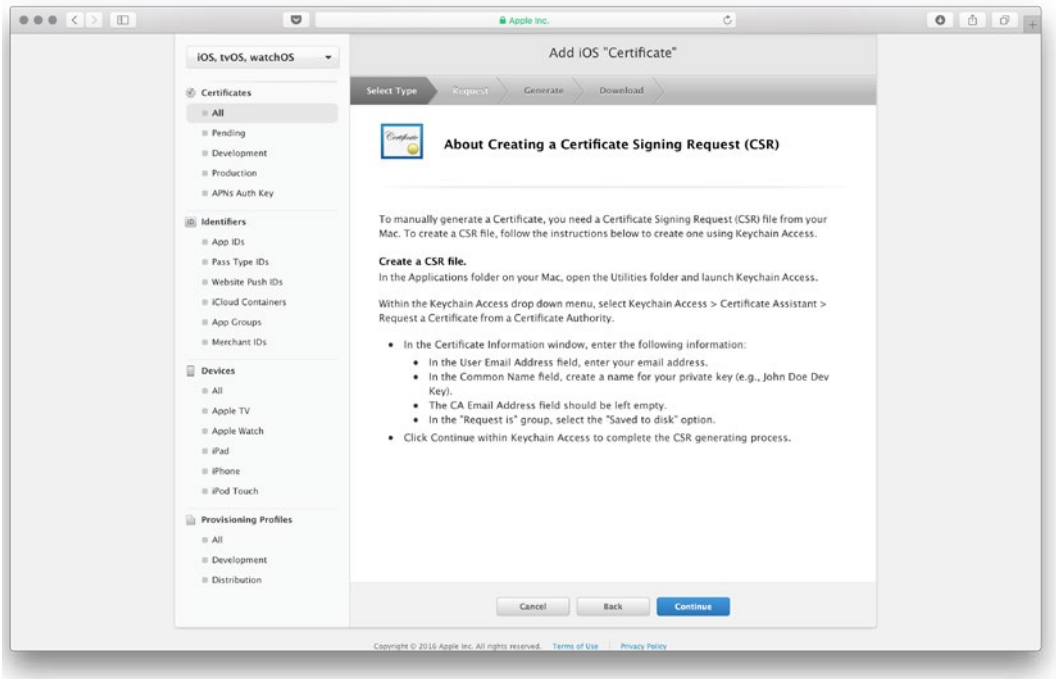
2. Click on **Certificates, Identifiers & Profiles** and, then, under **Certificates**, select **All**.
3. Click on the **+** at the top right of the screen:



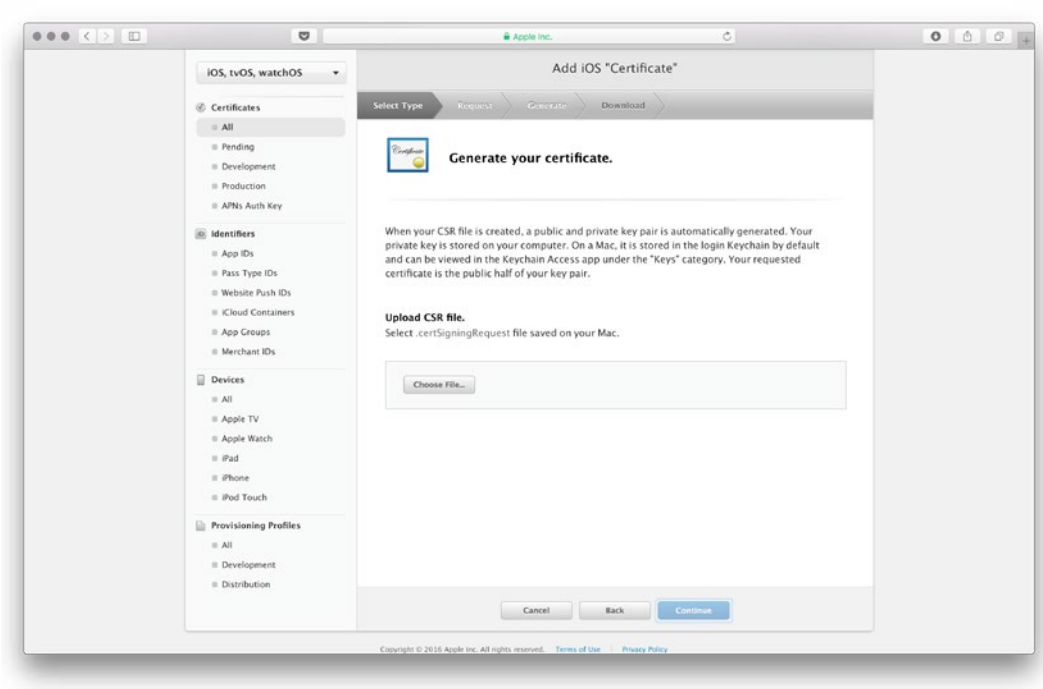
4. In the screen that appears, select **App Store** and **Ad Hoc** under **Production** and, then, click on **Continue**:



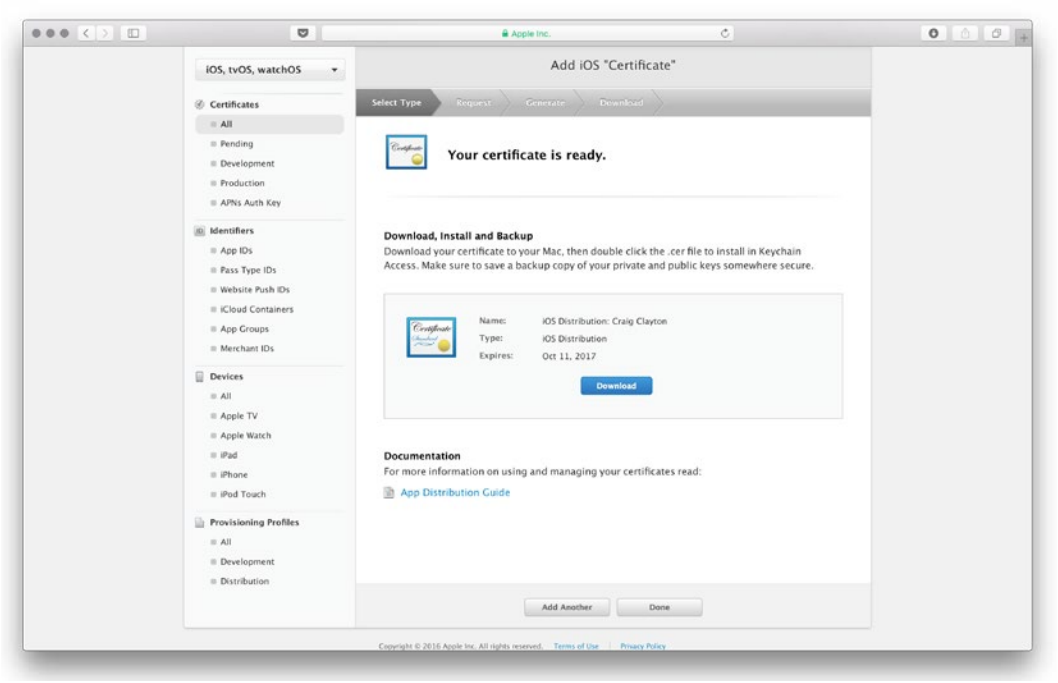
5. Then, the following screen lists the steps on how to create a **CSR** file (which we already created). Click on **Continue**:



6. Upload the CSR created earlier by selecting **Choose File** under **Upload CSR file** and, then, selecting the certificate file you saved and clicking on **Open**. Then, click on **Continue**:



7. Next, download the certificate:



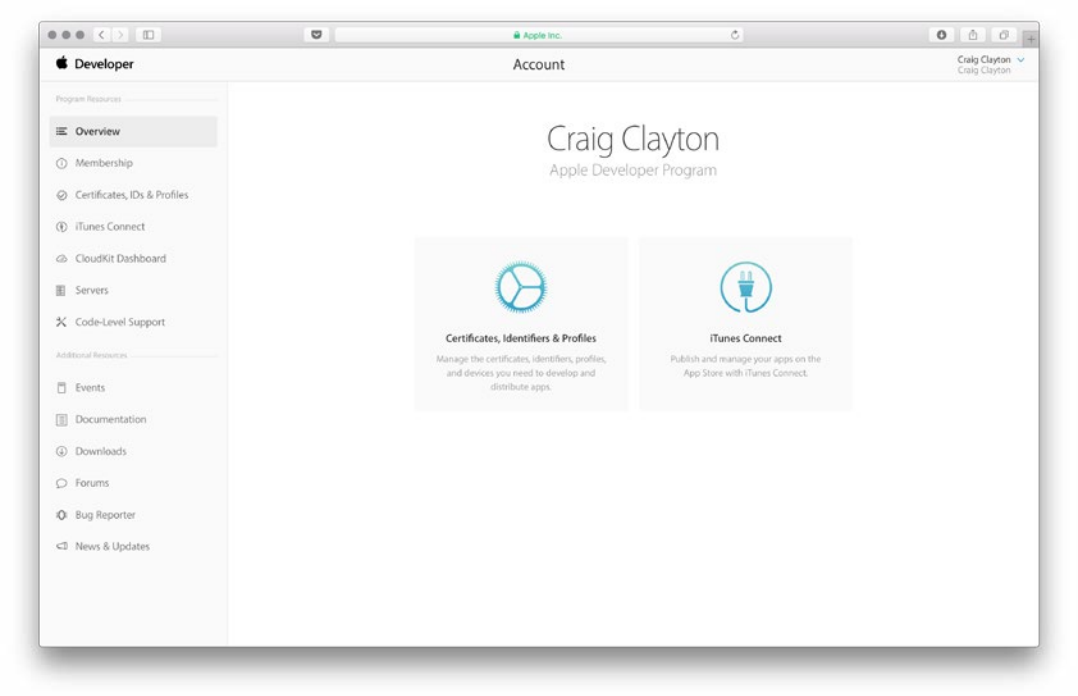
8. Then, install the downloaded certificate by double-clicking on it.

For the development certificate, you will repeat these steps again, except in the step where you choose the type of certificate you need, instead of selecting App Store and Ad Hoc under Production, you will select iOS App Development under Development. All the other steps will be the same.

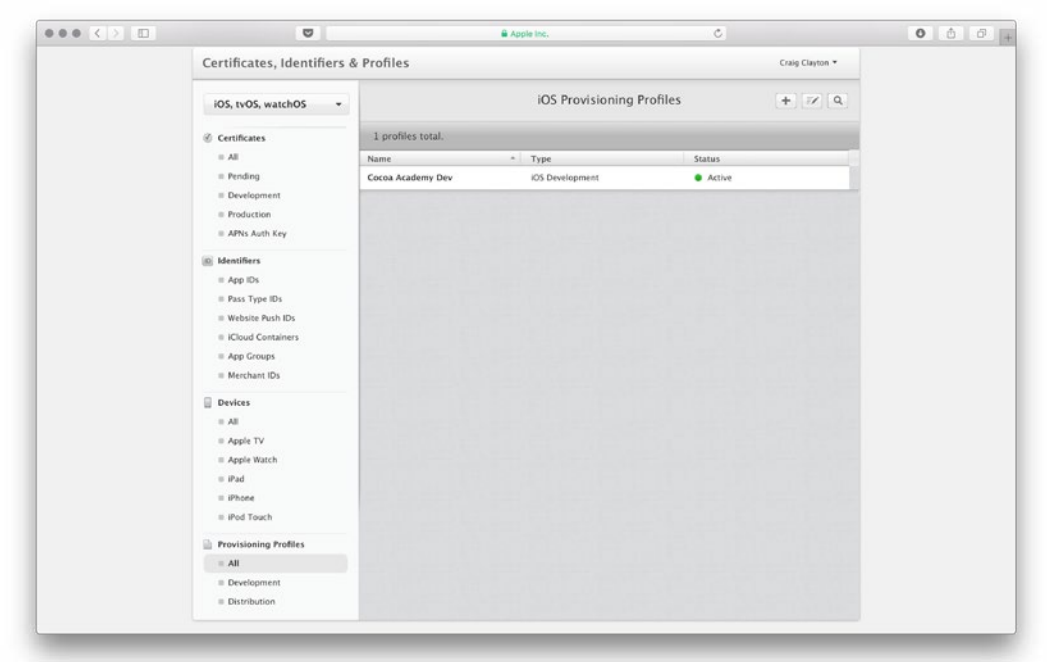
Creating a Production Provisioning Profile

Next, let's create a production provisioning profile, which is used for distributing your application. Xcode 8 creates these for you, but, again, it is still good to know how to do it:

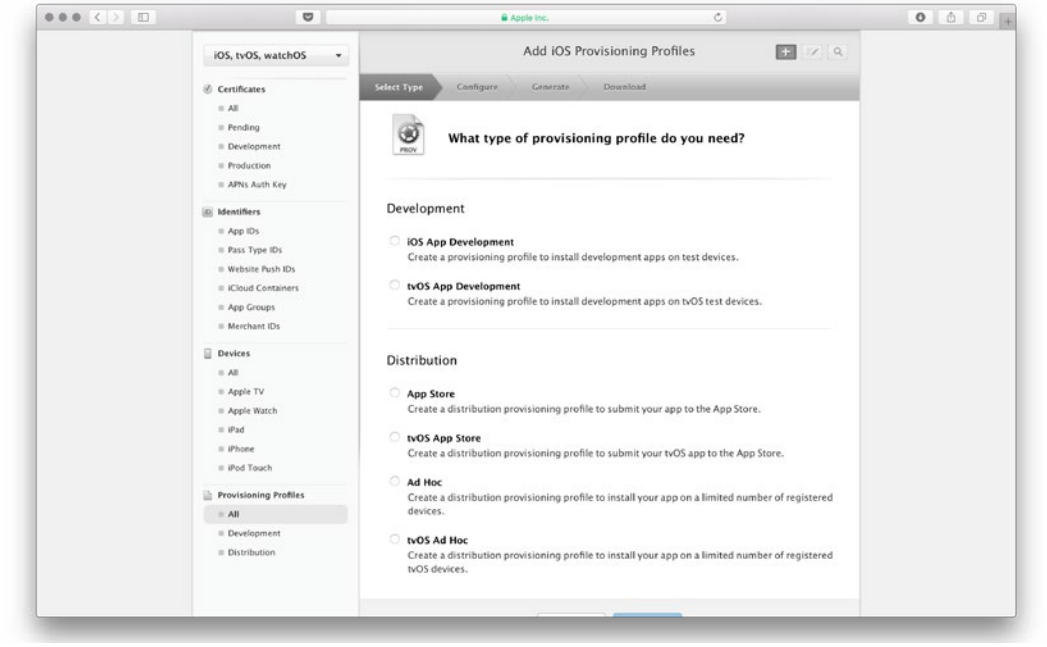
1. Log in to the Apple developer account, and you will see the following screen:



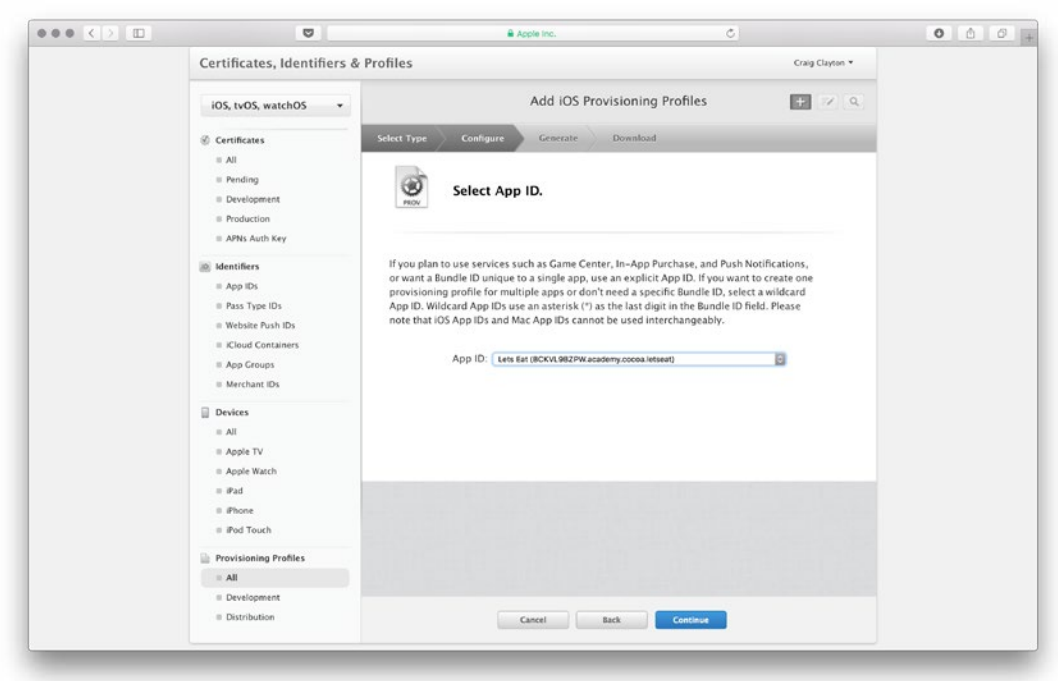
2. Click on **Certificates, Identifiers & Profiles** and, under **Provisioning Profiles**, select **All**.
3. Click on the **+** at the top right of the screen:



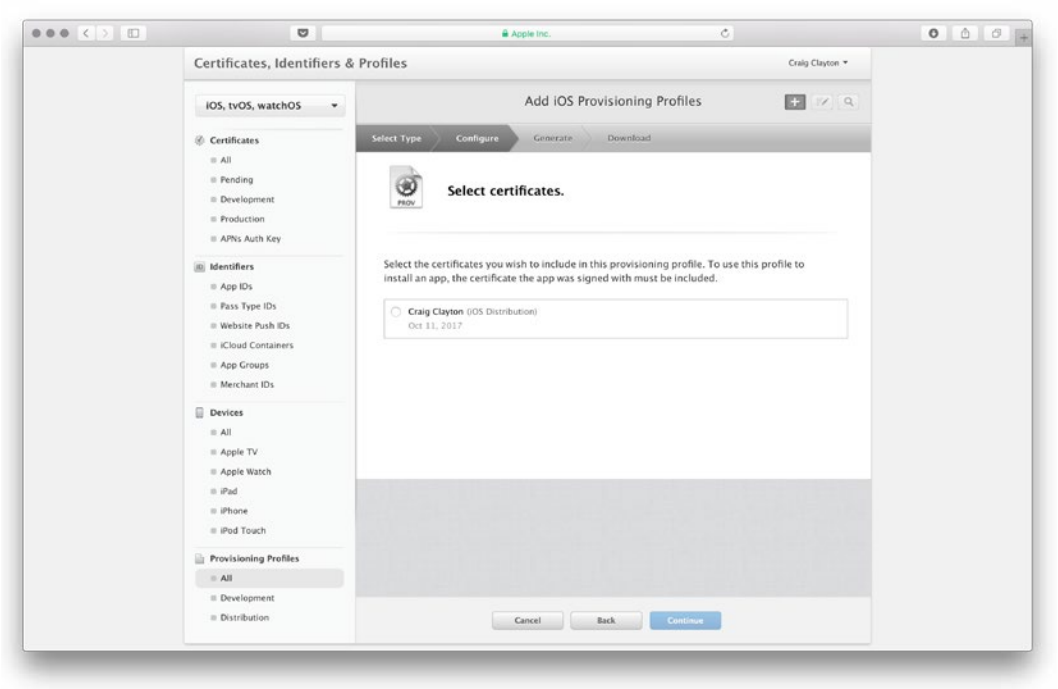
4. Select **App Store** under **Distribution** and, then, click on **Continue**:



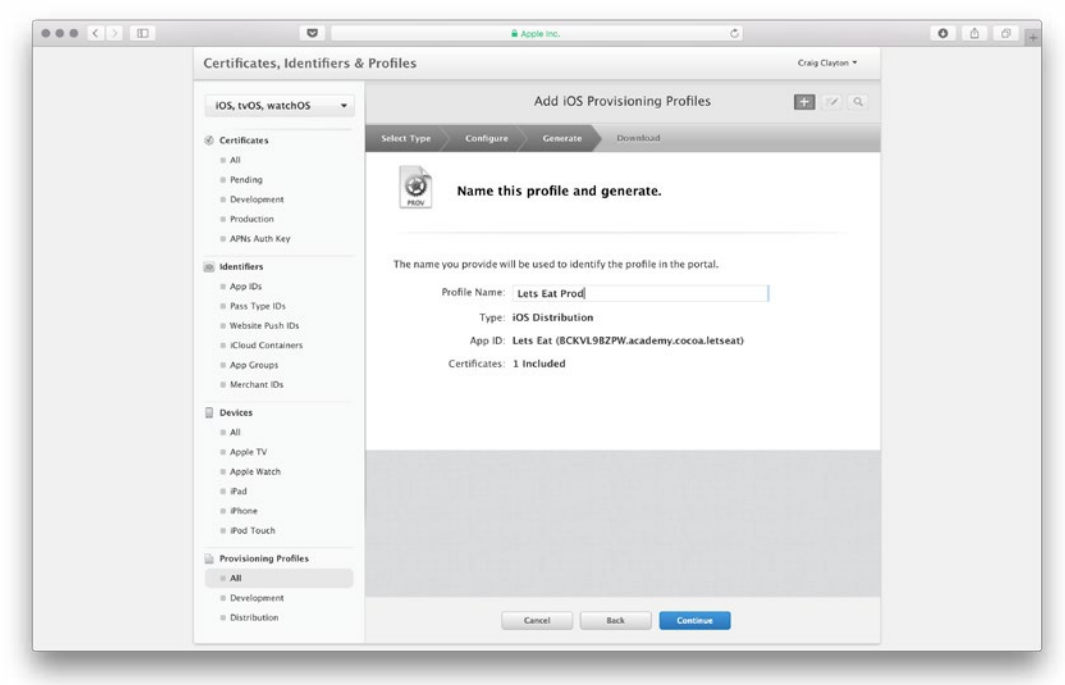
5. Select the **Bundle ID** created earlier and, then, click on **Continue**:



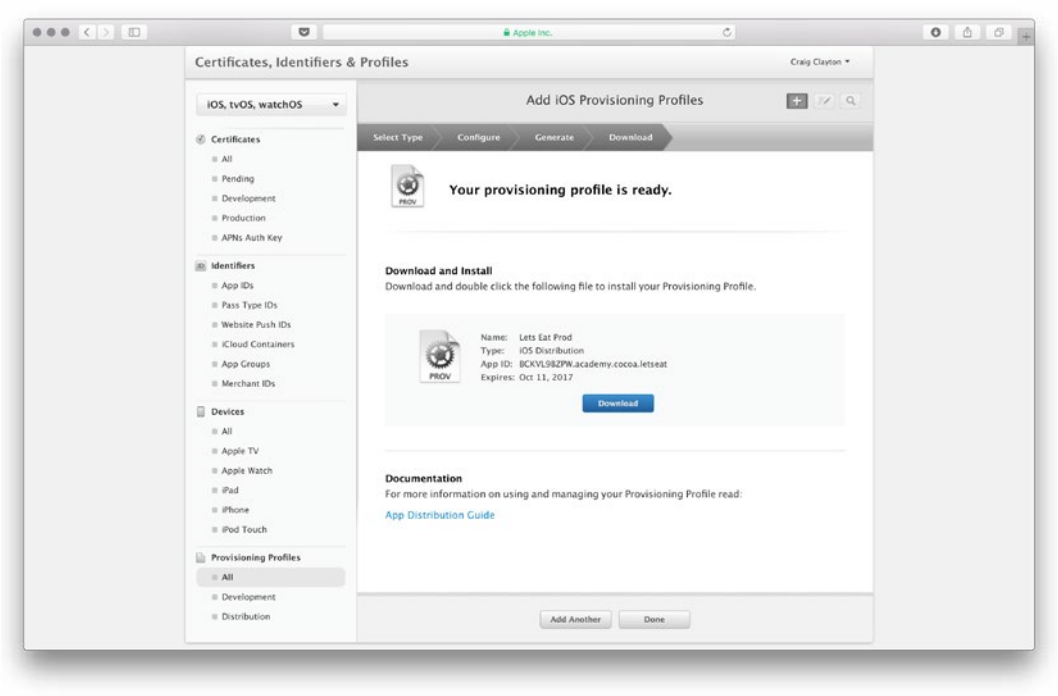
6. Next, select the certificate created earlier and, then, click on **Continue**:



7. Next, enter the **Profile Name**, `Lets Eat Prod` and, then, click on **Continue**:



8. Download the certificate:



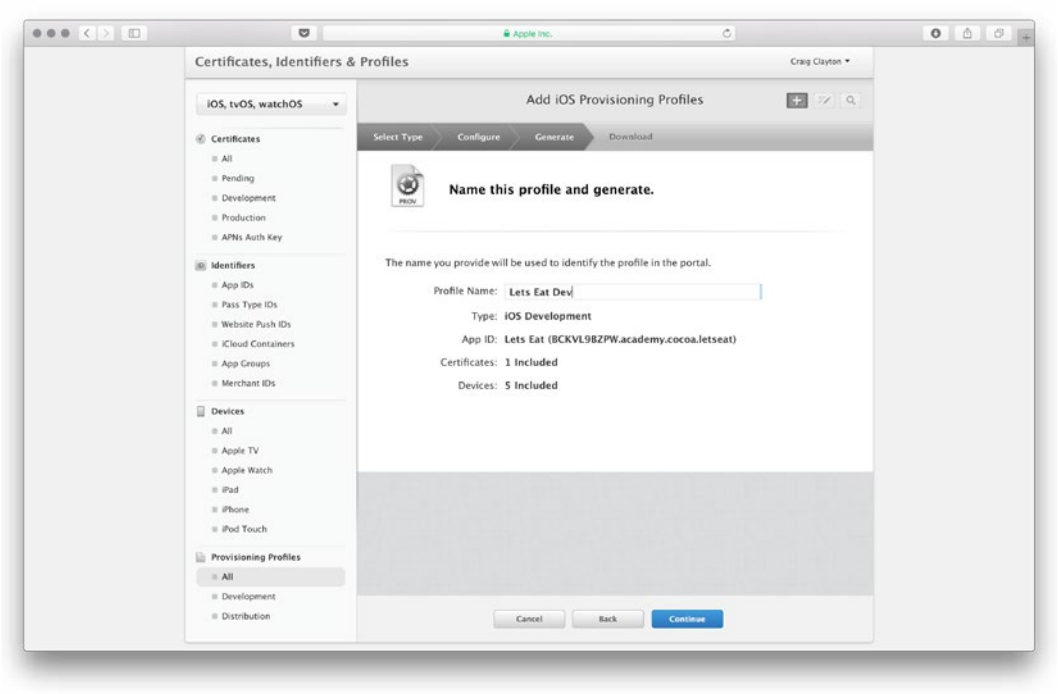
9. Install the downloaded certificate by double-clicking on it.

Creating a Development Provisioning Profile

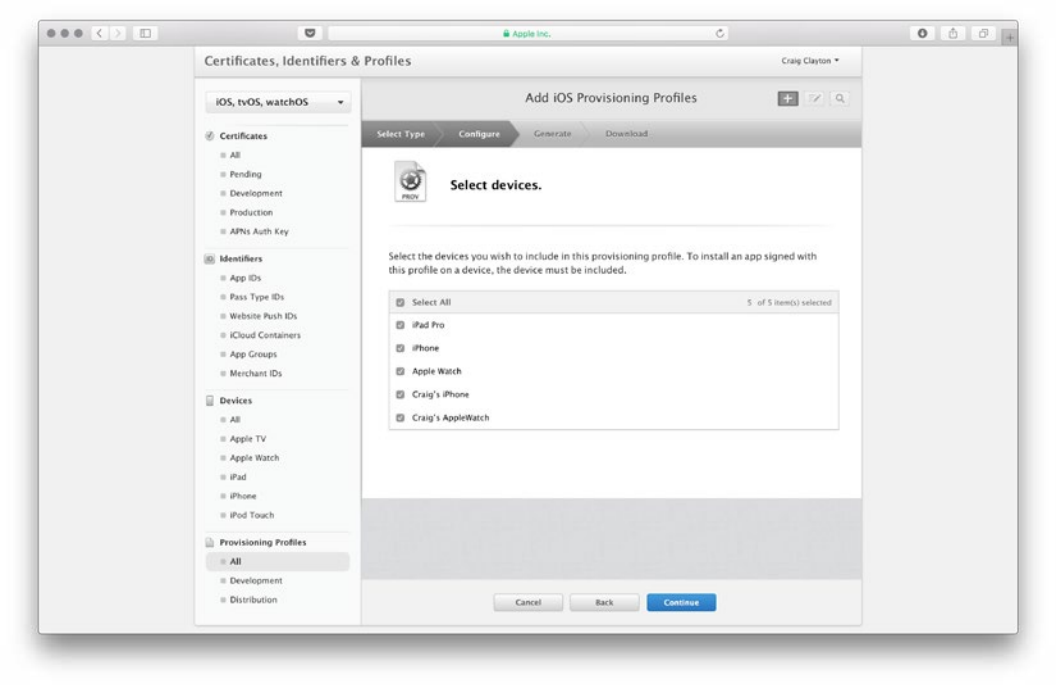
Next, let's create a development provisioning profile, which is used for building apps on your device using Xcode:

1. Log in to the Apple developer account.
2. Click on **Certificates, Identifiers & Profiles**.
3. Next, under **Provisioning Profiles**, click on **All**.
4. Then, click on the **+** at the top right of the screen.
5. Next, select **App Store** under **Distribution** and, then, click on **Continue**.
6. Select the **Bundle ID** created earlier and, then, click on **Continue**.
7. Next, select the certificate created earlier and, then, click on **Continue**.

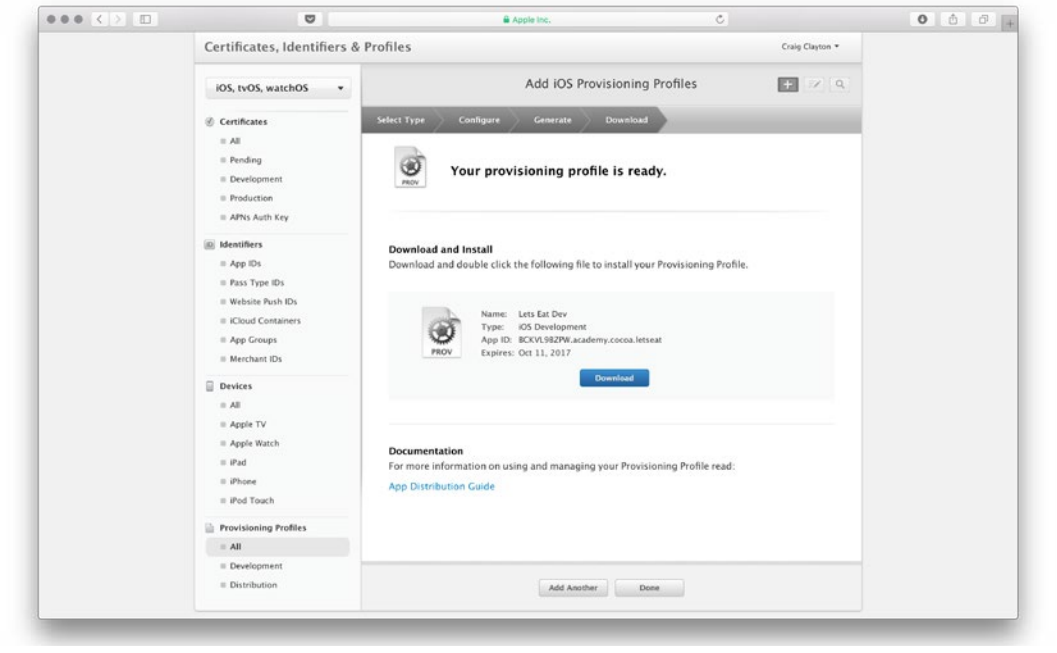
8. Enter the **Profile Name**, `Lets Eat Dev` and, then, click on **Continue**:



9. Select the devices you wish to use or choose **Select All**:



10. Download the certificate:

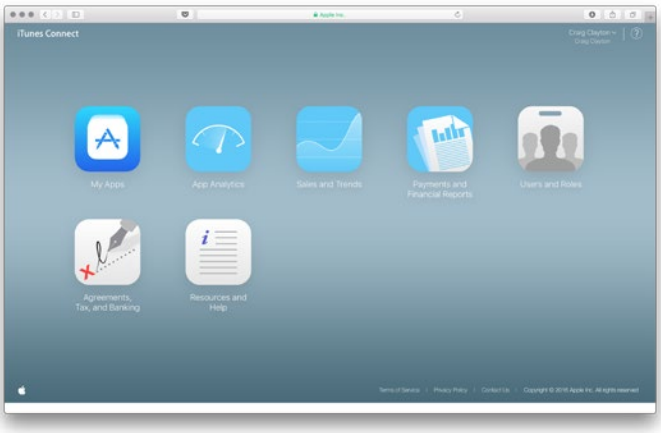


11. Install the downloaded certificate by double-clicking on it.

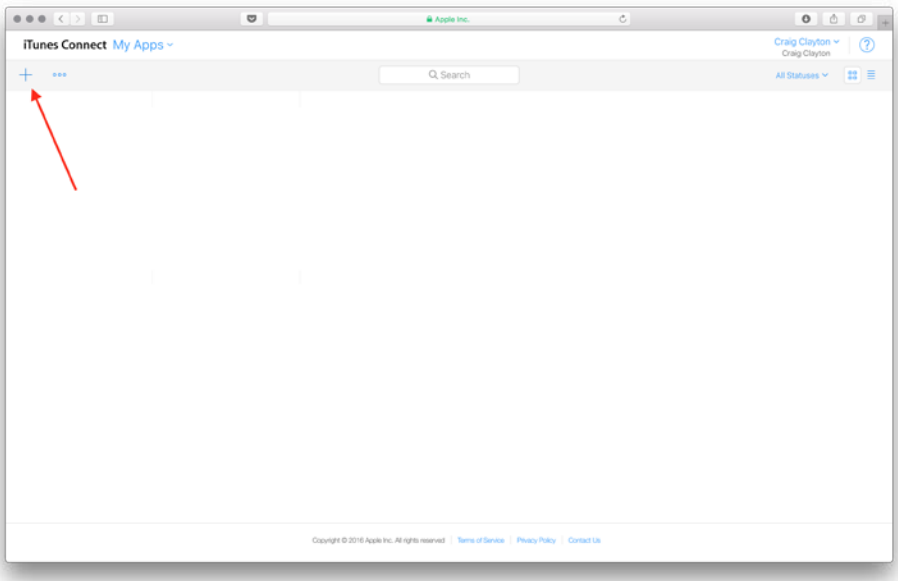
Creating the App Store Listing

Next, we are going to create the App Store listing:

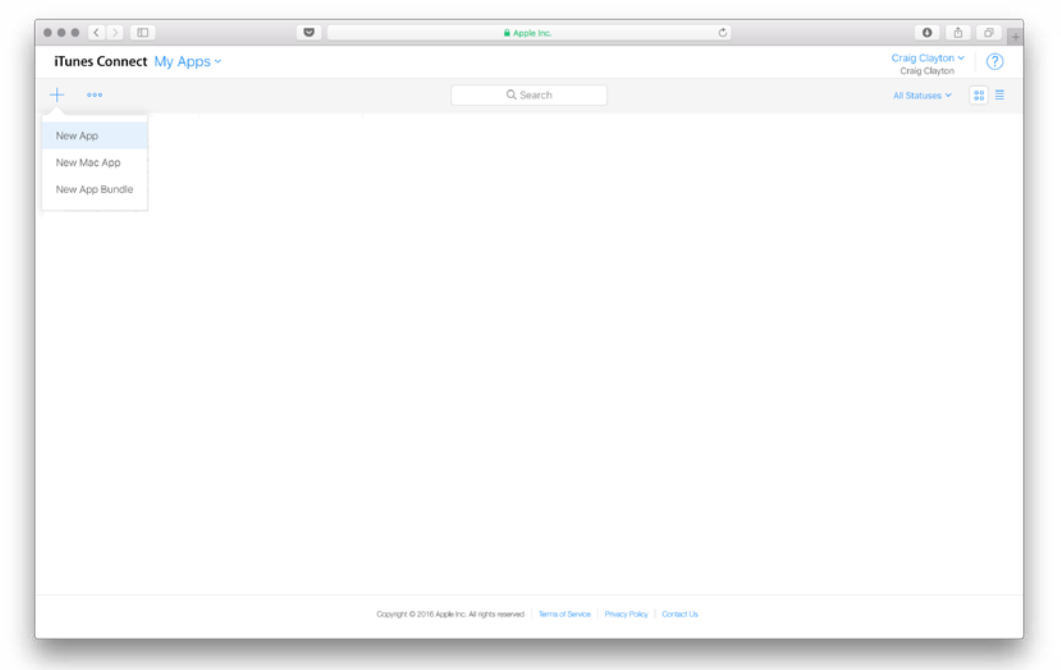
1. Login to your iTunes Account and select **My Apps**:



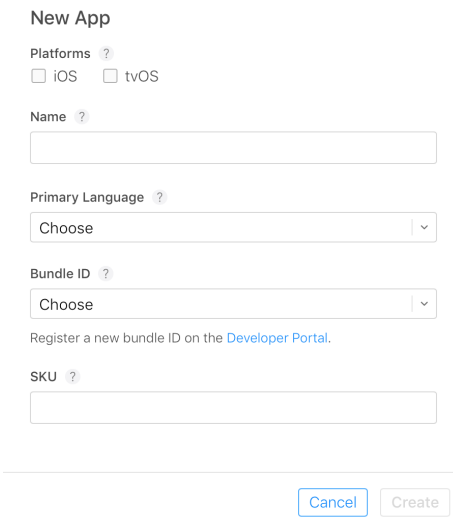
2. Click on the **+** at the top left of the screen:



3. Select **New App**:



4. Enter your app details and, then, hit **Create**:

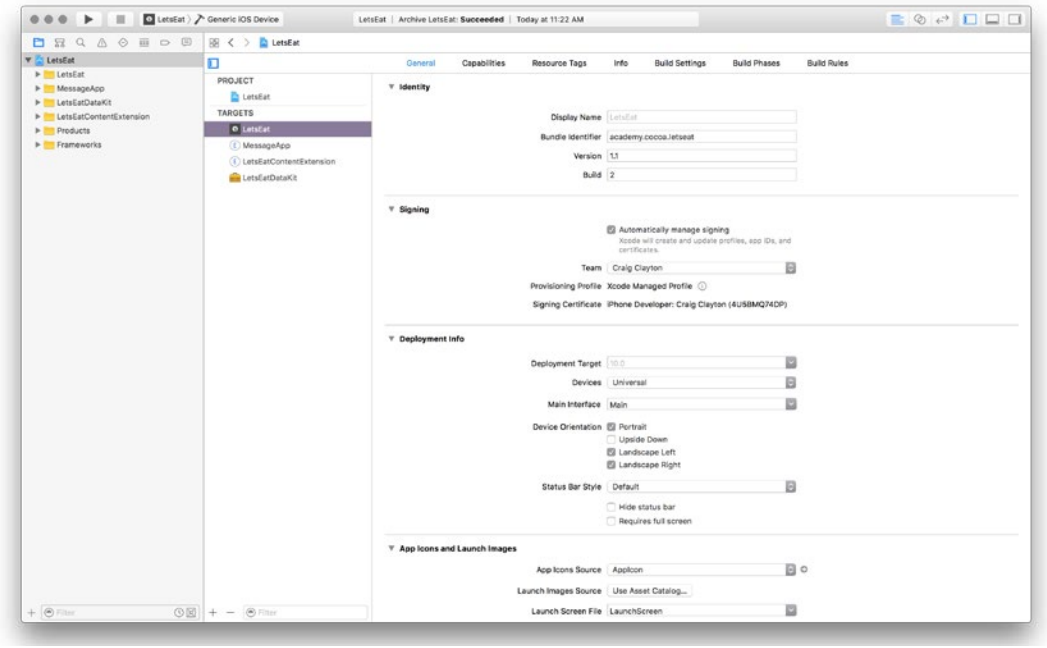


The app will now be listed in your iTunes account.

Creating an archive build

When you submit your app to the App Store, you need to create an archive. This archive will also be used for internal and external testing, which we will address shortly. When your archive is complete, you will upload this to the App Store. Let's create an archive now:

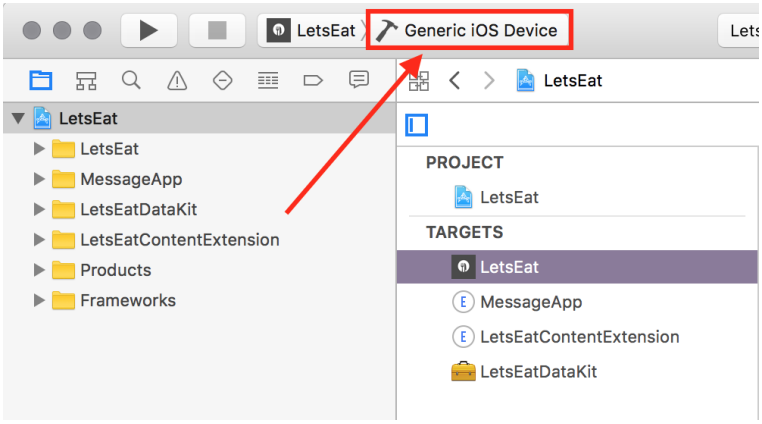
1. Open Xcode and select the project and enter the following information:
    - Under **Identity**: update the **Version** and **Build** numbers to `1.1` and `2`, respectively
    - Under **Signing**: ensure **Automatically manage signing** is checked
    - Under **Signing**: select **Team**



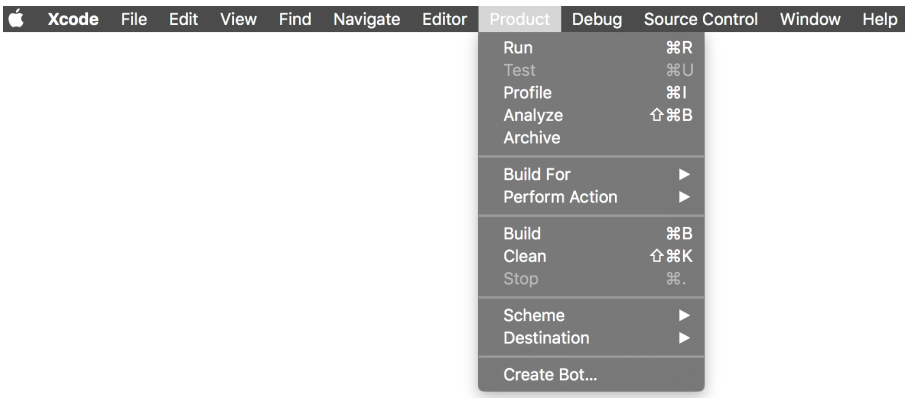


For minor builds, you want to increment your version number by .1 and your Build number by 1. In some instances, developers use 3 digits (for example, 1.1.2). This is all based on your business and how you want to handle version numbers. If you are performing a major update, then you typically increment your version number by 1.

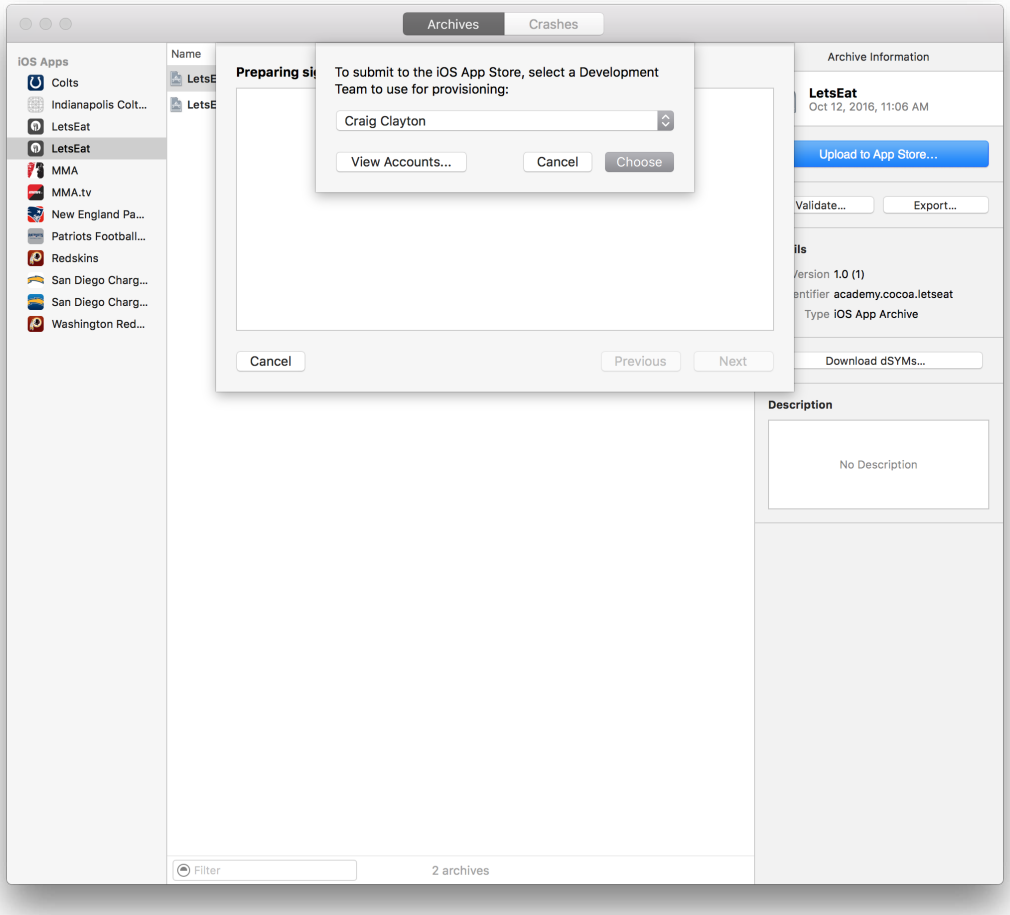
2. Select **Generic iOS Device** as the build destination:



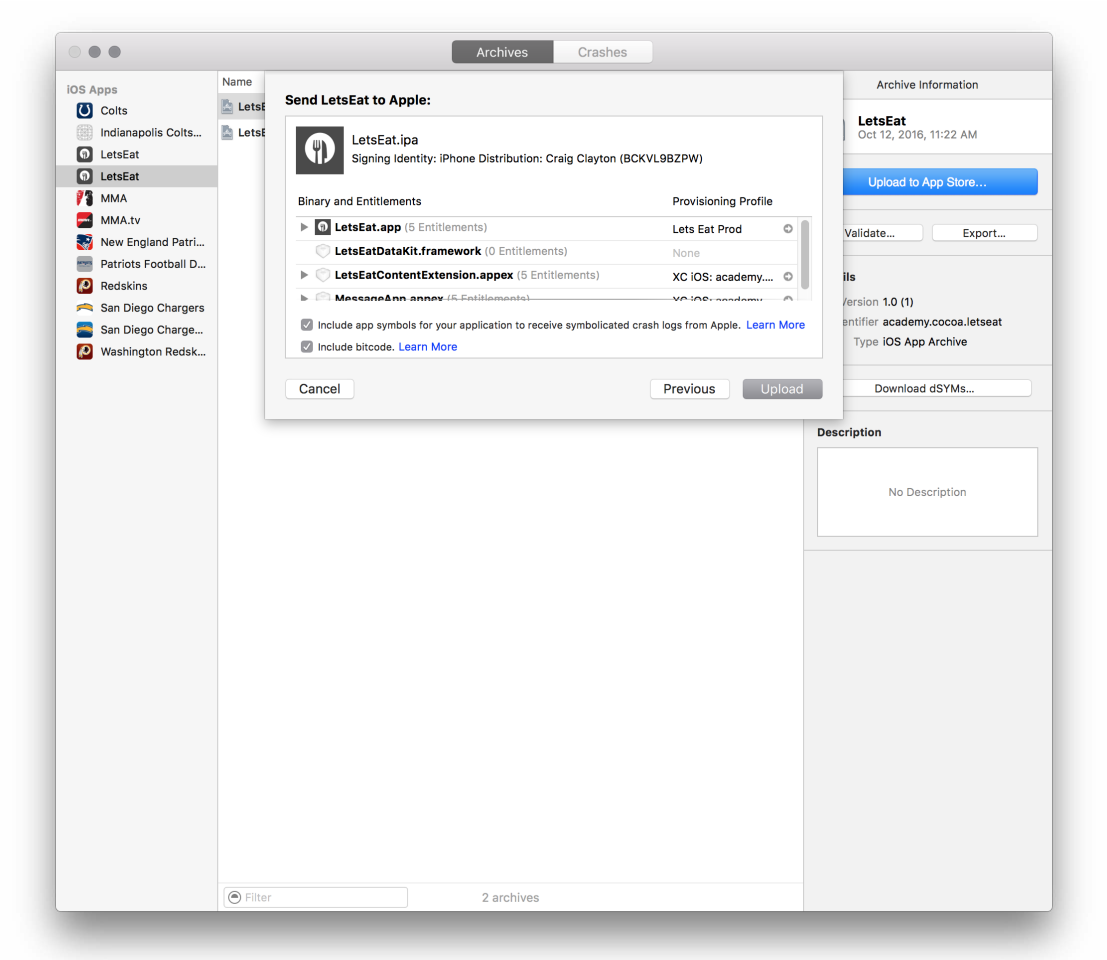
3. Update your `Info.plist` by adding `ITSAppUsesNonExemptEncryption`, making its type `Boolean` and setting its value to `NO`. The value should be `NO`, unless you are using some special encryption. Since our app does not have special encryption, we will set our value to `NO`.
4. Select **Product** | **Archive**:



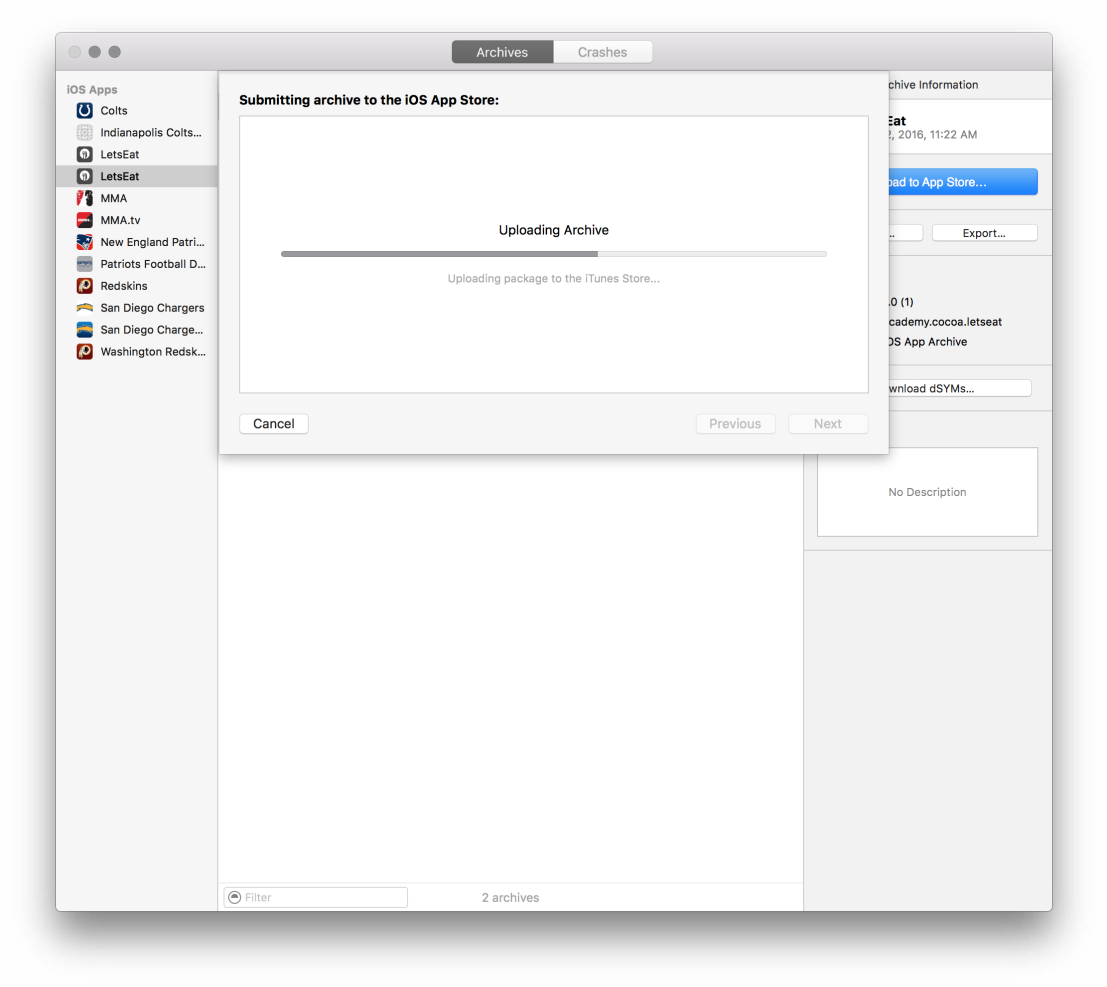
5. Under the **Archives** tab in the screen that appears, select your **Development Team** and, then, hit **Choose**:



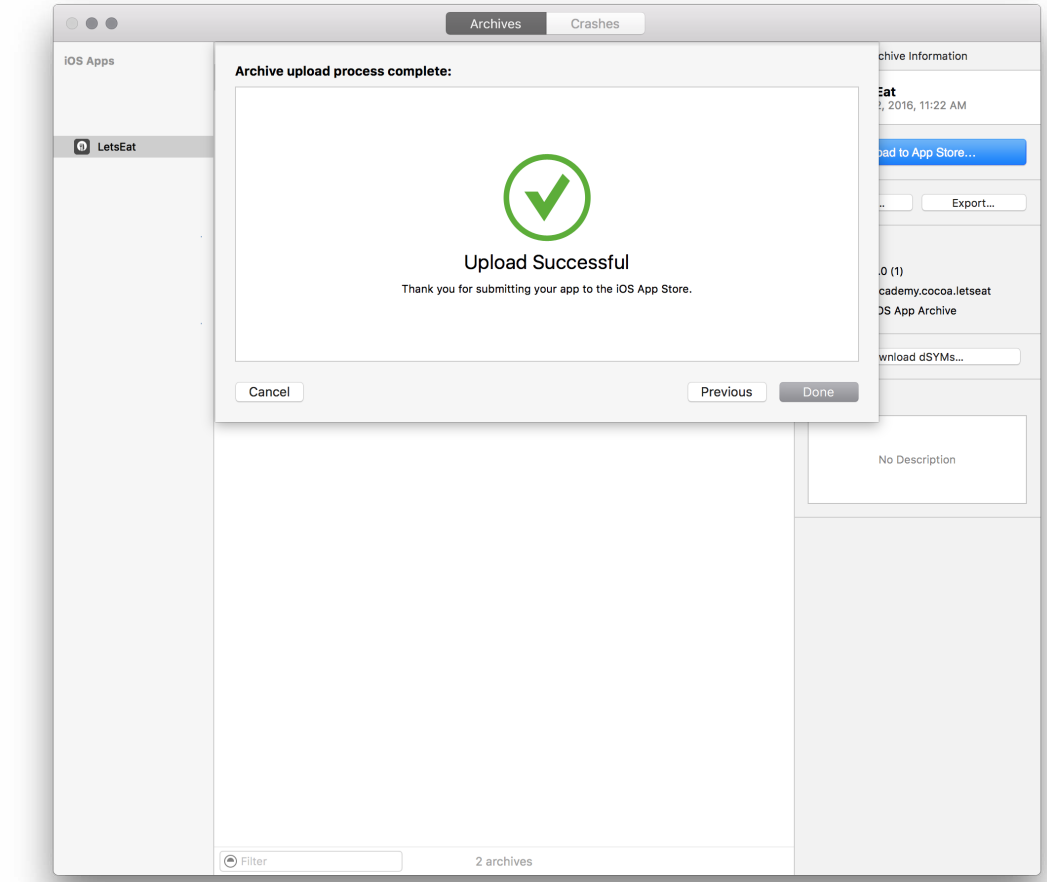
6. Your **IPA** file will now be created and, then, click on **Upload**:



7. You will see the upload begin:



8. Then, when your upload is successful, you will see the following:



9. You will receive an e-mail when your app is approved or rejected. If rejected, once you fix the issues, resubmit it in the same manner by updating the archive and following the steps laid out above.

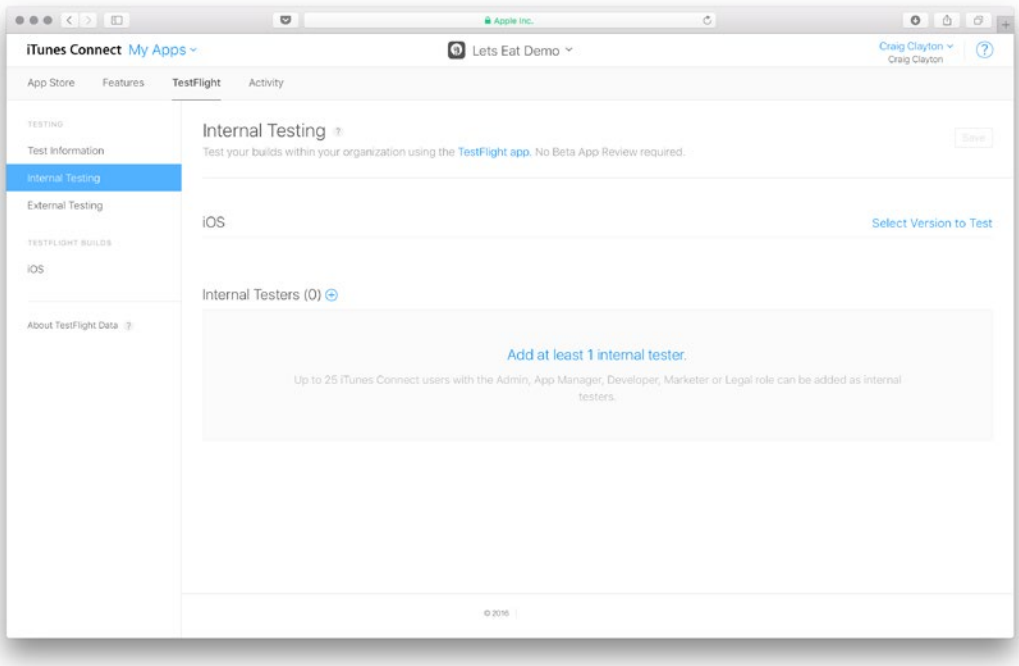
Internal and External Testing

Internal and external testing use what is known as **TestFlight**. This TestFlight app can be downloaded from the App Store, and you use e-mails to sign people up to TestFlight. Let's look at how to create each type of testing.

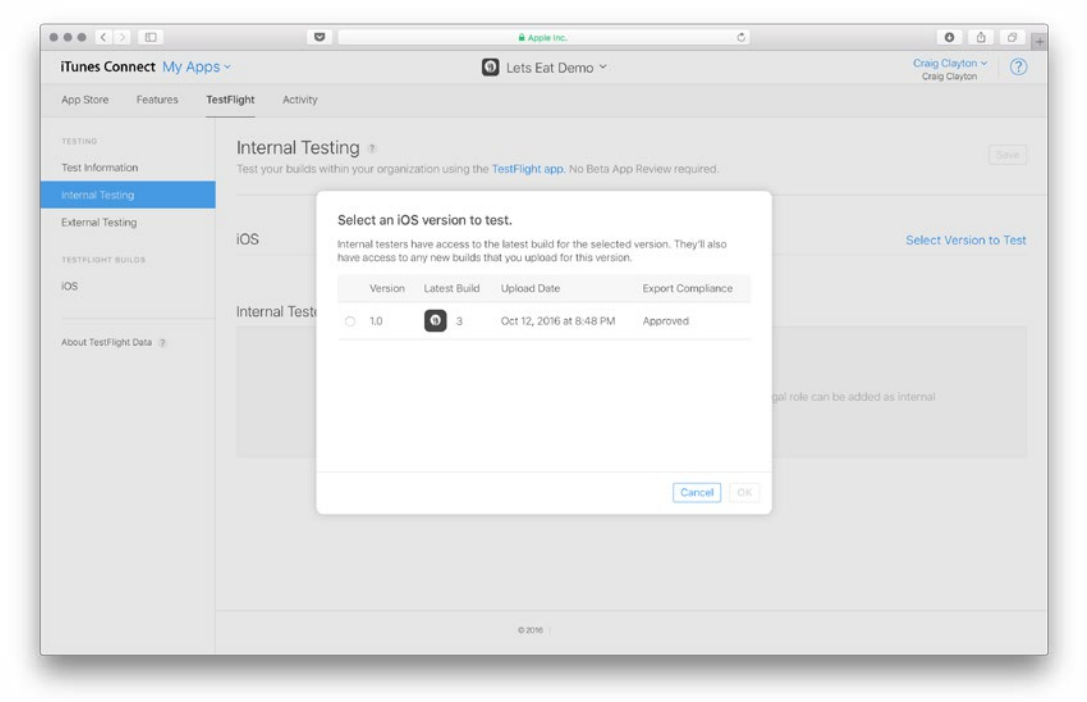
Internal testing

Internal testing does not go through a review process. You can only send builds to up to 25 testers for internal testing. Let's begin:

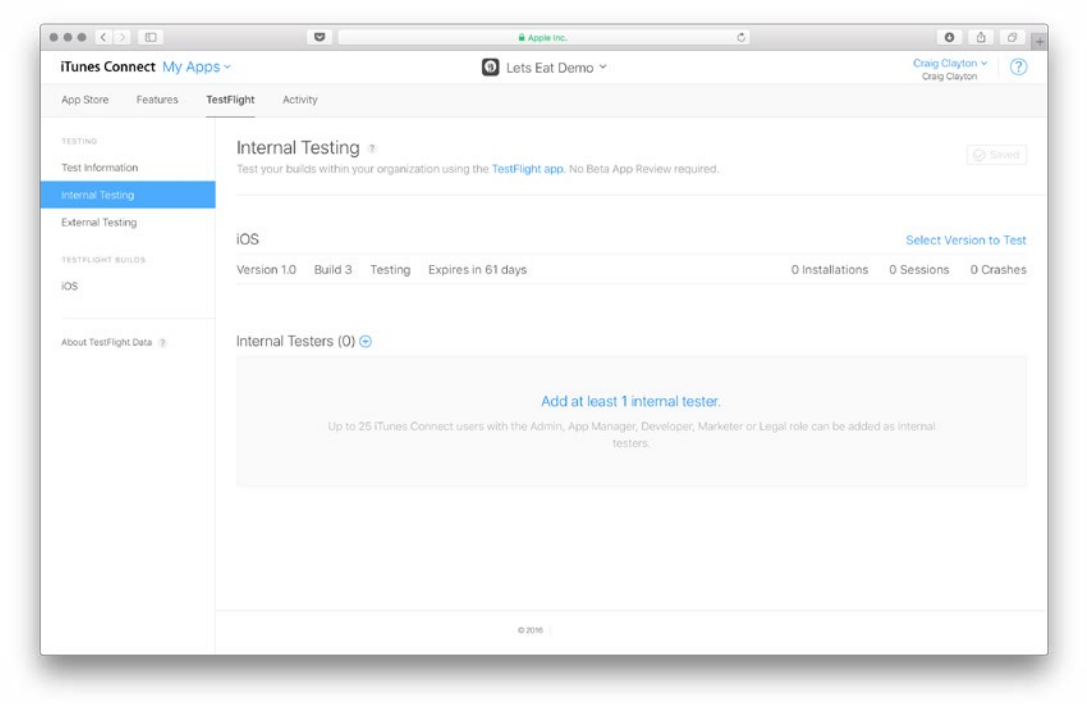
1. Log in to your iTunes Account and select **My Apps**.
2. Select your **Let's Eat** app and then **Test Flight**.
3. On the left side of the page, select **Internal Testing** and, then, on the right side of the page, click on **Select Version** to **Test**:



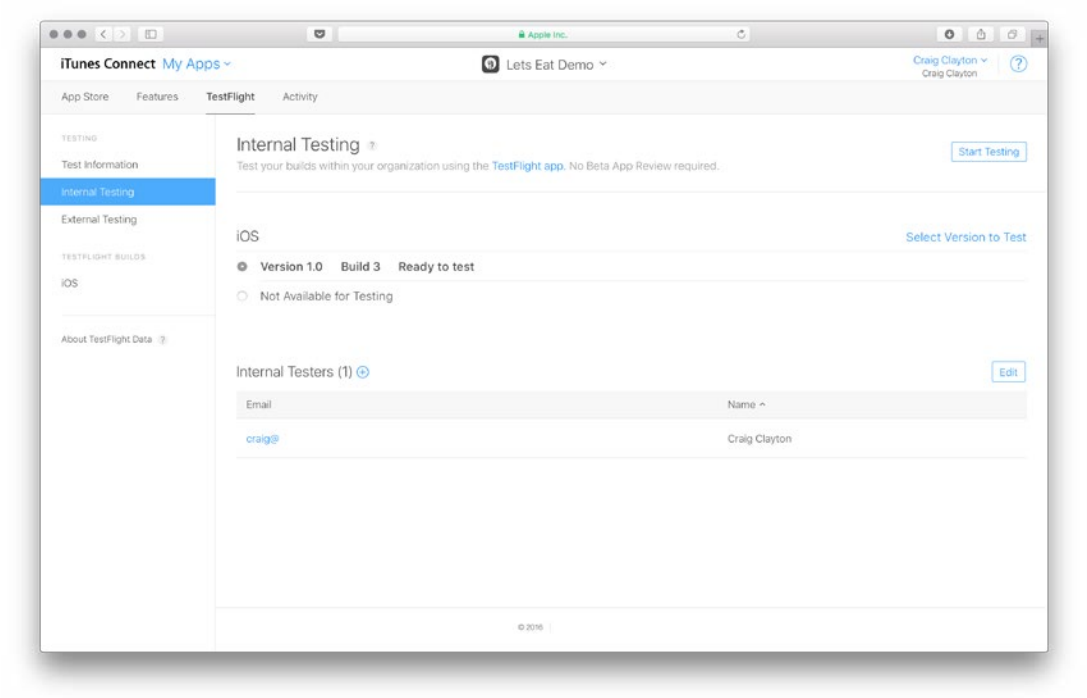
4. Then, select the version you want to test and click on **OK**:



5. You will now see the following screen:



6. Finally, click on the **+** button next to **Internal Testers** and add your **Internal Testers**:

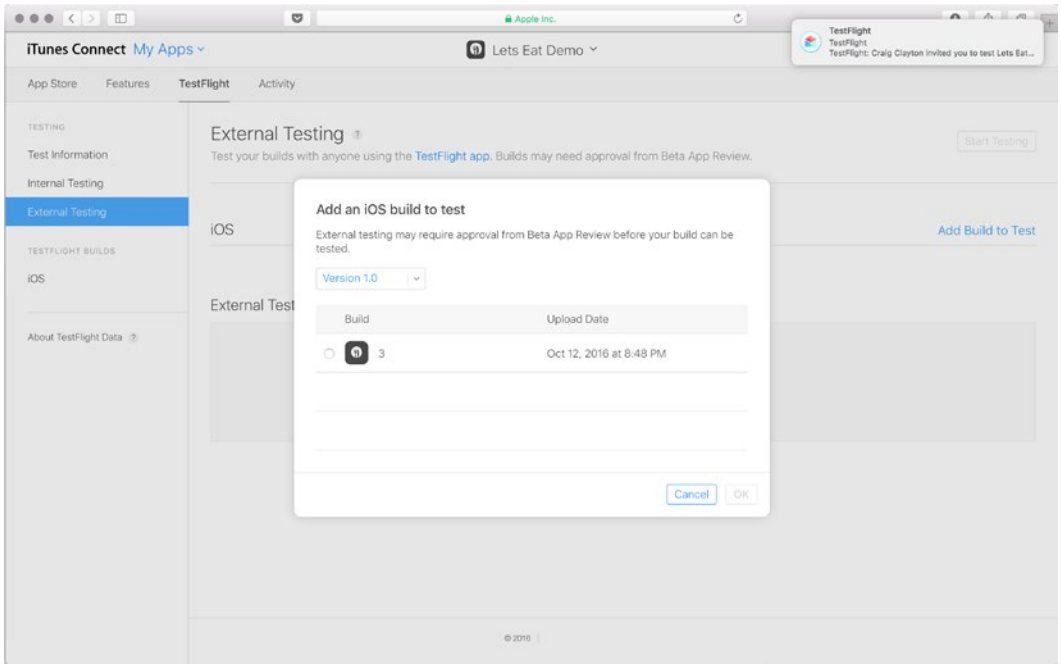


External testing

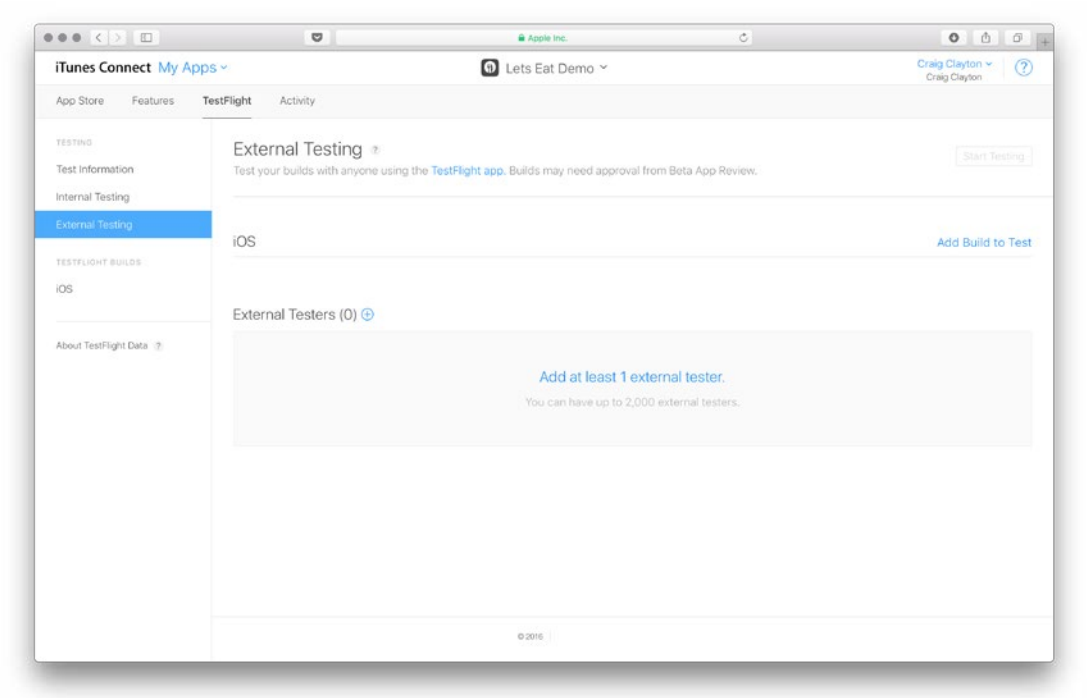
External testing may or may not go through a review process, but with external testing you can have up to 2,000 testers. For external testing, follow these steps:

1. Log in to your iTunes Account and select **My Apps**.
2. Select your **Let's Eat** app and, then, **TestFlight**.
3. On the left side of the page select **External Testing**.

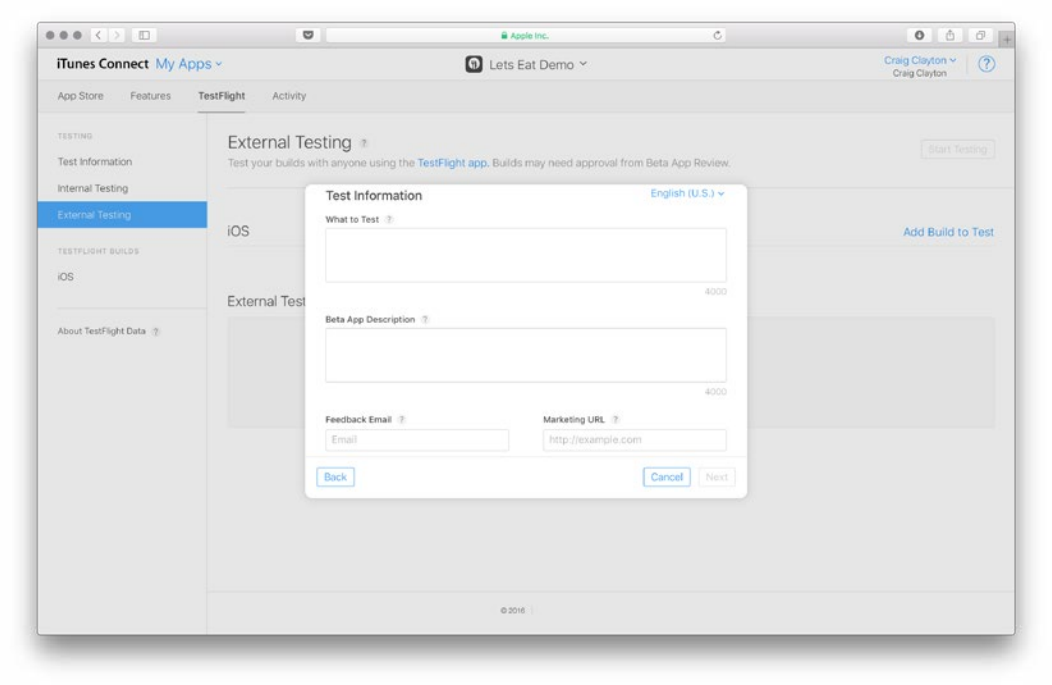
4. Next, on the right side of the page, click on **Add Build to Test** and, then, select your build and hit **OK**:



5. Finally, click on the **+** next to **External Testers** and add your **External Testers**:



6. When you are done adding testers, click on the **Start Testing** button. You will see the following screen and will need to complete the information requested:



7. Next, we submit to Apple for review; you will receive an e-mail when it is approved or rejected. If rejected, once you fix the issues, resubmit it in the same manner by updating the archive and following the steps laid out above.

Summary

You have now completed the entire process of building an app and submitting it to the App Store. If you have gone from the beginning to the end, congratulate yourself, because it is truly a big feat.

At this point, all you can do is wait for Apple to review your project. The next week or so will be the most nerve wracking (at least it was for me), because you will be hoping that nothing was missed. However, do not worry if your app gets rejected, because it happens to the most experienced of developers, and it is fixable. Apps are often rejected for minor reasons that are easy to fix; however, you do not want to work for months on a project and miss something big that Apple will never approve. So, do your research as to what is and is not acceptable to Apple. As you submit your apps to the App Store, reach out to me and let me know, because I would love to see what you have built.

Index

Symbols

3D touch quick actions
adding 581-592

A

Address section, Restaurant detail 286-290
annotations
adding 324
creating 324
custom annotations, creating 328-331
Map View Controller, creating 324-328
API 354
API Manager
about 354
creating 353
API Manager file
exploring 355-357
App Delegate
breaking down 185-189
Application Programming Interface (API) 353
App Store listing
creating 632-634
App Tour 108
archive build
creating 634-639
area, in Playgrounds
DEBUG TOGGLE 20
PLAYGROUND EDITOR 19
Play/Stop 20
RESULTS PANEL 19
WINDOW PANE CONTROLS 19
array literal
set, creating with 94
arrays
about 67, 68
creating, with initial values 69
empty array, creating 68
items, adding to 70-73
items, removing from 79, 80
iterating over 78
mutable array, creating 69
number of elements, checking 73
value, retrieving from 75-78
arrow
adding, to custom title view 147-151
auto layout
about 108
adding, to restaurant list cell 377, 378

B

Bar Button Items
updating 245
basic notifications
sending 562
Booleans
about 21
creating 25-27
bugs
detecting 23
bundle identifier
creating 608-614
buttons
adding, to notifications 572-575

C

cancel button
unwinding 246, 247
Certificate Signing Request
creating 614-617
clear background
adding, to custom title view 154-156
closed range 55, 56
code
organizing 344-352
Collection View Controller 105
comments
about 28
adding, to code 28
constants
about 22
versus variables 28
control flow
about 56
for…in loop 57-60
repeat...while loop 63, 64
while loop 61, 62
Core Data
about 453
NSManagedObjectContext 454
NSManagedObjectModel 454
NSPersistentStoreCoordinator 455
Core Data Manager 465-470
cuisine
passing, to restaurant list 365-370
custom title view
arrow, adding to 147-151
clear background, adding to 154-156
container, adding 142-144
creating 142
Custom Label, adding to 147-151
custom UI
adding, to notifications 575-579

D

data
displaying, in Restaurant Detail view 386-394
displaying, in restaurant list cell 385, 386
passing, to Restaurant List View Controller 395-397
data model
creating 455-462
entity auto-generation 463
items, reviewing 464, 465
data types
about 20
Booleans (bools) 21
floating-point numbers 21
integer data type 21
string 20
Debug panel, Xcode interface 7
Design Clean Up
about 154
arrow, updating 158
button, adding 162-166
clear background, adding to custom title view 154-156
Collection View Controllers 167-169
crash, fixing 174-177
label, updating 159-161
map kit view 171-173
modal, adding 169-171
UIStackView, updating 157
development provisioning profile
creating 629-632
dictionary
about 81
creating 82
item, accessing in 86
items, removing from 92, 93
number of items, checking 90, 91
dictionary elements
adding 83-85
updating 83-85
dictionary keys
iterating over 88, 89
dictionary values
iterating over 86-89
dynamic cells 253

E

elements
positioning, in restaurant list cell 375, 376
empty array
checking for 73, 74
creating 68

empty set
creating 94
entity auto-generation 463
Explore tab
about 109
storyboard, creating for 178-180
Explore View Controller 353
external testing 639, 643-646

F

favorites, adding within Restaurant List View
about 593
Core Data manager, updating 599-605
new Model Object, adding 594-599
files
creating 116, 117
filters
apply Filter View Controller, creating 435-441
working with 426-431
Filter Scroller
creating 431-435
floating-point numbers
about 21
adding 24, 25
for...in loop 57-60
functions
about 46
creating 46-51

G

Generic iOS Device, Xcode interface 10

H

half-closed range 56
Hungarian notation 27

I

images
embedding, in notifications 569-572
iMessages
about 536
assets, updating 539
auto layout, adding to cell 543, 544
extension, creating 536-539
framework, creating 544-552
Message Cell, connecting 552, 553
reservations, sending 557-559
restaurants, displaying 554-557
UI, implementing 540-542
initial values
array, creating with 69
Integers (Ints)
about 21
variables, creating with 22
Integrated Development Environment (IDE) 1
internal testing 639-642
iOS device, Xcode interface 10-13
items
accessing, in dictionary 86
adding, into set 95
adding, to arrays 70-73
removing, from arrays 79, 80
removing, from dictionary 92, 93
removing, from set 100

J

JavaScript Object Notation (JSON) 353
JSON file 354, 355

L

location
last selected location, obtaining 361-365
passing, to Restaurant list 365-370
selected location, passing to Explore View 359-361
selecting 358
location list 357
Locations view 110
Location View Controller
used, for connecting Table View 252, 253

M

Managed Object Context 454
Managed Object Model 454

map annotations
MKAnnotation 314
setting up 313
Map Data Manager
base class, creating 321, 322
creating 318-321
ExploreDataManager, refactoring 323
Map tab
about 112
storyboard, creating for 181
Map View Controller 353
MKAnnotation
about 314
Restaurant annotation, creating 314-318
Model
data, obtaining from plist 223-225
developing 216
ExploreDataManager.Swift file 220-222
ExploreData.plist file 217
ExploreItem.swift file 217-219
Model View Controller (MVC) architecture
about 108, 192, 193
cell, connecting to 227
classes and controllers 199
classes and structures 193-199
CollectionView Cells 205-208, 225-227
CollectionView Controllers 205-208
Controller camp 193
data, displaying into CollectionView 208
data source 209-211
file, creating in Controller folder 200-204
grid, updating 211-215
Model camp 192
setup 193
UI elements, hooking up with IBOutlets 228-231
View camp 192
mutable array
creating 69
mutable set
creating 95

N

Navigation Controller 105
Navigator panel, Xcode interface 6
notifications
about 561
basic notifications 562
buttons, adding to 572-575
customizing 569
custom UI, adding to 575-579
displaying 566-568
images, embedding 569-572
setting up 563-566
NSManagedObjectContext 454
NSManagedObjectModel 454
NSPersistentStoreCoordinator 455

O

operations, with Integers
about 31, 32
comparison operators 34
decrement 33
if statement 35-42
increment 33
optional bindings 42-44
optionals
about 42-44
need for 45
option screen, Xcode 4, 5

P

permission
obtaining 562
Persistent Store Coordinator 454, 455
Playground project
creating 54
Playgrounds
about 18
area 19
creating 18, 19
print()
using 23
production and development certificates
creating 618-623
production provisioning profile
creating 624-629
project
creating 114, 115
project setup 113

Property List (plist)
creating 260, 261
data, adding 262-264
Data Manager, working with 265
Location Data Manager file, creating 264
Protocol Oriented Programming (POP) 431

R

ranges
about 55
closed range 55, 56
half-closed range 56
refactor 177
repeat...while loop 63, 64
reservations
about 301
header 306-311
information section 303-305
times, adding 301, 302
restaurant cell class
creating 379
RestaurantDataManager file
creating 381-384
restaurant detail
about 112
creating 269, 271
data, parsing 337-344
exploring 274, 275
mapping 336, 337
refactoring 332
Storyboard reference, creating 332-336
Restaurant Detail view
about 386
data, displaying in 386-394
header layout, updating 518-521
Map Section Layout, updating 530-534
map update, performing 397, 398
No Reviews Layout, updating 522-525
Reviews Layout, updating 526-530
Table Details Section layout, updating 521, 522
updating 517
restaurant list
background, modifying 371, 372
building 371
cuisine, passing to 365-370
location, passing to 365-370
restaurant list cell
auto layout, adding to 377, 378
data, displaying in 385, 386
elements, positioning in 375, 376
updating 372-375
restaurant list cell outlets
setting up 380, 381
Restaurant listing 233-238
restaurant listing page 511-516
Restaurant List View Controller
data, passing to 395-397
reviews
creating 291-299, 406
Reviews section
about 401
auto layout, adding 411-413
auto layout, adding for Photo Filter View 419
auto layout, adding for Ratings View section 416, 417
Photo Filter View, adding 417, 418
Ratings View screen, adding 414, 415
Review Cells, updating 408, 409
review images, creating 441-450
Review Storyboard, setting up 406, 407
UI elements, positioning 409-411
Unwind Segues, hooking up 423-426
Unwind Segues, setting up 422
views, presenting as modals 420-422

S

segue 107
set
about 93, 94
creating, with array literal 94
empty set, creating 94
items, adding into 95
items, checking in 96
items, removing from 100
iterating over 97
mutable set, creating 95
two sets, intersecting 99
two sets, joining 100

Stack Views
using 145, 146
Standard editor, Xcode interface 6
star ratings
auto layout, adding 498-501
cell UI, setting up 495-497
creating 471-495
review list extension, adding 501-503
static Table View
labels, adding 277-286
restaurant details, exploring 274, 275
section headers, creating 276
setting up 272, 273
storyboard
about 106, 107, 131, 177
App assets, adding 122-131
Bar Button Items, updating 245
Cancel Button, unwinding 246, 247
creating, for Explore tab 178-180
creating, for Map tab 181
folders, organizing 182, 183
global settings, setting up 184
launch screen, creating 132-139
Navigation Controller, adding 140-142
project folders, setting up 181, 182
refactoring 177
setting up 118-121
UI, updating 242-244
storyboard reference 177
string interpolation 30, 31
strings
about 20
concatenating 29, 30
variables, creating with 22
Swift types 194

T

Tab Bar Controller 106
Table View
adding 248, 249
code, exploring 255-257
connecting, with Location View Controller 252, 253
data source, adding 253
delegate, adding 253
Edges, updating 249
locations, adding 258-260
Prototype Cell, creating 254
View Controller Class, creating 250
table view controllers
setting up 401-405
TestFlight 639
Toolbar, Xcode interface 8, 9
type inference 29
type safety 29

U

UI elements, hooking up with IBOutlets
about 228-231
cell, selecting 232, 233
UIStackView
updating 157
Utilities panel, Xcode interface 7

V

value
retrieving, from array 75-78
variables
about 22
creating, with Integer (Int) 22
creating, with strings 22
versus constants 28
View Controller 104

W

while loop 61, 62
Window Pane Controls, Xcode interface 15

X

Xcode
downloading 2
installing 3
launching 3, 4
option screen 4, 5
Xcode interface
about 5
Debug panel 7

Generic iOS Device 10
iOS device 10-13
Navigator panel 6
Standard editor 6
Toolbar 7-9
Utilities panel 7
Window Pane Controls 15
XML Interface Builder (xib) file 474

Y

YELP 354