

INTERMEDIATE IOS 9 PROGRAMMING WITH SWIFT

SIMON NG

APPCODA

Fully updated for Xcode 7 and Swift 2

Copyright © 2015 AppCoda Limited

All rights reserved. Please do not distribute or share without permission. No part of this book or corresponding materials (such as images or source code) may be distributed by any means without prior written permission of the author.

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

Table of Contents

1. Preface
2. Building Adaptive User Interfaces
3. Adding Sections and Index list in UITableView
4. Animating Table View Cells
5. Working with JSON
6. How to Integrate Facebook and Twitter Sharing
7. Working with Email and Attachments
8. Sending SMS and MMS Using MessageUI Framework
9. How to Get Direction and Draw Route on Maps
10. Search Nearby Points of Interest Using Local Search
11. Audio Recording and Playback
12. Scan QR Code Using AVFoundation Framework
13. Working with URL Schemes
14. Building a Full Screen Camera
15. Video Capturing and Playback Using AVKit
16. Displaying Banner Ads using iAd
17. Working with Custom Fonts
18. Working with AirDrop and UIActivityViewController
19. Building Grid Layouts with Collection Views
20. Interacting with Collection Views
21. Adaptive Collection Views Using Size Classes and UITraitCollection
22. Building a Today Widget Using App Extensions
23. Building Slide Out Sidebar Menus
24. View Controller Transitions and Animations
25. Building a Slide Down Menu
26. Self Sizing Cells and Dynamic Type
27. XML Parsing and RSS
28. Applying a Blurred Background Using UIVisualEffect
29. Using Touch ID For Authentication
30. Building a Carousel-Like User Interface

31. Working with Parse
32. How to Preload a SQLite Database Using Core Data

Preface

At the time of this writing, the Swift programming language has been around for more than a year. The new programming language has gained a lot of traction and continues to evolve, and is clearly the future programming language of iOS. If you are planning to learn a programming language this year, Swift should be on the top of your list.

I love to read cookbooks. Most of them are visually appealing, with pretty and delicious photos involved. That's what gets me hooked and makes me want to try out the recipes. When I started off writing this book, the very first question that popped into my mind was "Why are most programming books poorly designed?" iOS and its apps are all beautifully crafted - so why do the majority of technical books just look like ordinary textbooks?

I believe that a visually stunning book will make learning programming much more effective and easy. With that in mind, I set out to make one that looks really great and is enjoyable to read. But that isn't to say that I only focus on the visual elements. The tips and solutions covered in this book will help you learn more about iOS 9 programming and empower you to build fully functional apps more quickly.

The book uses a problem-solution approach to discuss the APIs and frameworks of iOS SDK, and each chapter walks you through a feature (or two) with in-depth code samples. You will learn how to build a universal app with adaptive UI, use Touch ID to authenticate your users, create a widget in notification center and implement view controller animations, just to name a few.

I recommend you to start reading from chapter 1 of the book - but you don't have to follow my suggestion. Each chapter stands on its own, so you can also treat this book as a reference. Simply pick the chapter that interests you and dive into it.

Who Is This Book for?

This book is intended for developers with some experience in the Swift programming language and with an interest in developing iOS apps. It is not a book for beginners. If you have some

experience in Swift, you will definitely benefit from this book.

If you are a beginner and want to learn more about Swift, you can check out our beginner book at <http://www.appcoda.com/swift>.

What version of Xcode do you need?

This book is fully updated for iOS 9, Xcode 7 and Swift 2. Therefore, make sure you use Xcode 7.0 (or up) to go through the projects in this book.

Where to Download the Source Code?

I will build a demo app with you in each chapter of the book, and in this way walk you through the APIs and frameworks. At the end of the chapters, you will find the download links of the final projects for your reference. You are free to use the source code and incorporate it into your own projects. Both personal and commercial projects are allowed. The only exception is that they may not be reused in any way in tutorials or textbooks, whether in print or digital format. If you want to use it for educational purpose, attribution is required.

Do You Need to Join the Paid Apple Developer Program?

You can go through most of the projects using the built-in simulator. However, some chapters such as Touch ID and QR code scanning require you to run the app on a real device. The good news is that everyone can run and test their own app on a device for free, starting from Xcode 7. Even if you do not join the paid Apple Developer Program, you can deploy and run the app on your iPhone. All you need to do is sign in Xcode with your Apple ID, and you're ready to test your app on a real iOS device.

Swift is still evolving. Will you update the source code when Xcode 7.x releases?

Swift is ready for production. But you're right; Apple still keeps making changes to the language. Whenever a new version of Xcode 7 is released (e.g. Xcode 7.x), I will test all of the source code involved in this book again. You can always download the latest version of source

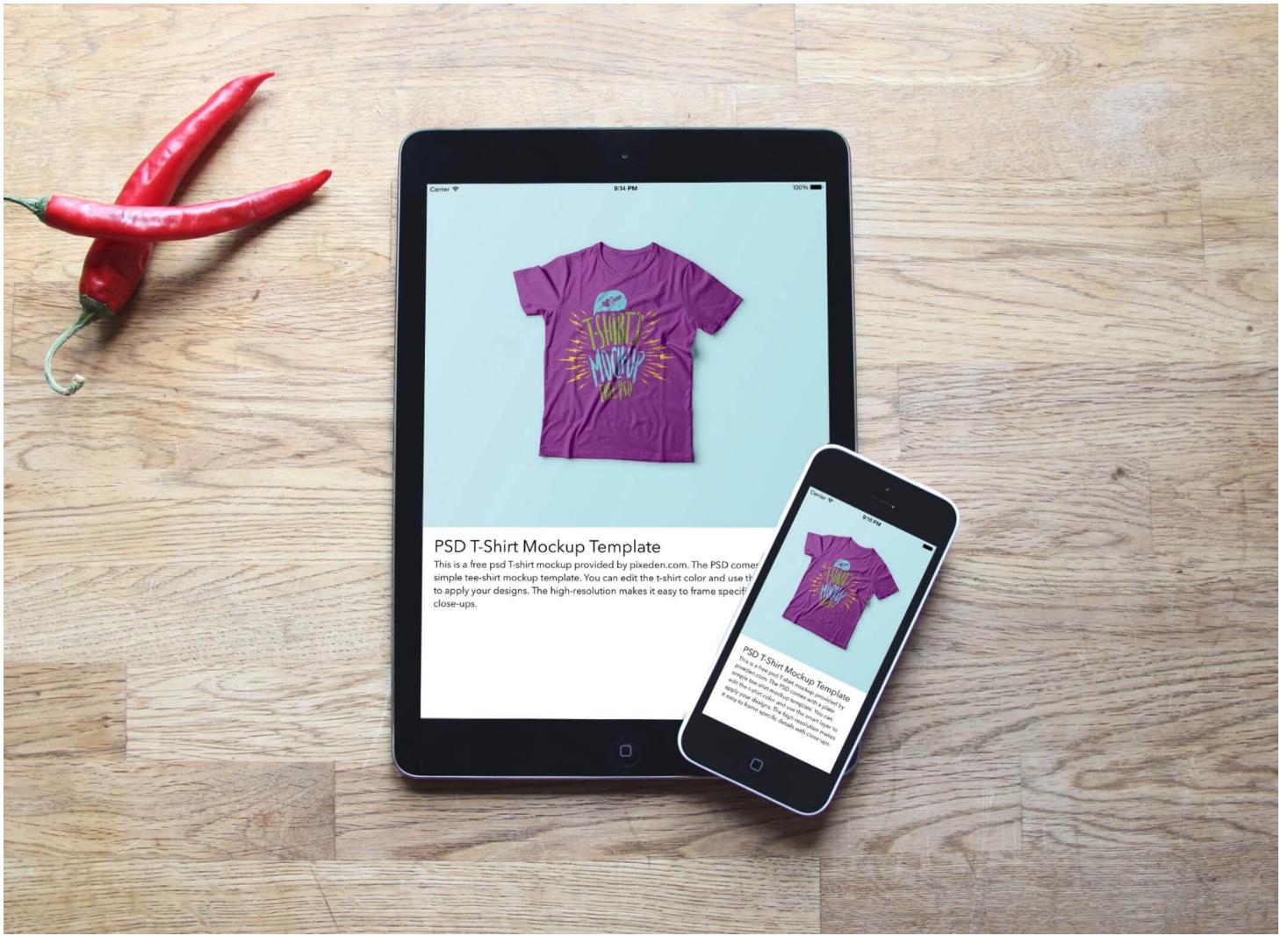
code using the download link included in each chapter. You can also join us on Facebook (<https://facebook.com/groups/appcoda>) or Twitter (<https://twitter.com/appcodamobile>) for any update announcement.

Got Questions?

If you have any questions about the book or find any error with the source code, post it on our private community (<https://facebook.com/groups/appcoda>) or reach me at simonng@appcoda.com.

Chapter 1

Building Adaptive User Interfaces



In the beginning, there was only one iPhone with a fixed 3.5-inch display. It was very easy to design your apps.; you just needed to account for two different orientations (portrait and orientation). Later on, Apple released the iPad with a 9.7-inch display. If you were an iOS developer at that time, you had to create two different screen designs (i.e. storyboards / XIBs) in Xcode for an app - one for the iPhone and the other for the iPad.

Gone are the good old days. Fast-forward to 2015: Apple's iPhone and iPad lineup has changed a lot. With the recent launch of the iPhone 6s and iPhone 6s Plus, your apps are required to

support an array of devices with various screen sizes and resolutions including:

- iPhone 4/4s (3.5-inch)
- iPhone 5/5c/5s (4-inch)
- iPhone 6/6s (4.7-inch)
- iPhone 6/6s Plus (5.5-inch)
- iPad / iPad 2 / iPad Air / iPad Air 2 (9.7-inch)
- iPad Mini / iPad Mini 2 / iPad Mini 3 / iPad Mini 4 (7.9-inch)
- iPad Pro (12.9-inch) (to be launched in November 2015)

It is a great challenge for iOS developers to create a universal app that adapts its UI for all of the listed screen sizes and orientations. So what can you do to design pixel-perfect apps?

Starting from iOS 8, the mobile OS comes with a new concept called *Adaptive User Interfaces*, which is Apple's answer to support any size display or orientation of an iOS device. Now apps can adapt their UI to a particular device and device orientation.

This leads to a new UI design concept known as *Adaptive Layout*. Xcode 7 allows developers to build an app UI that adapts to all different devices, screen sizes and orientation using a universal storyboard. You can now lay out user interface components in a single storyboard.

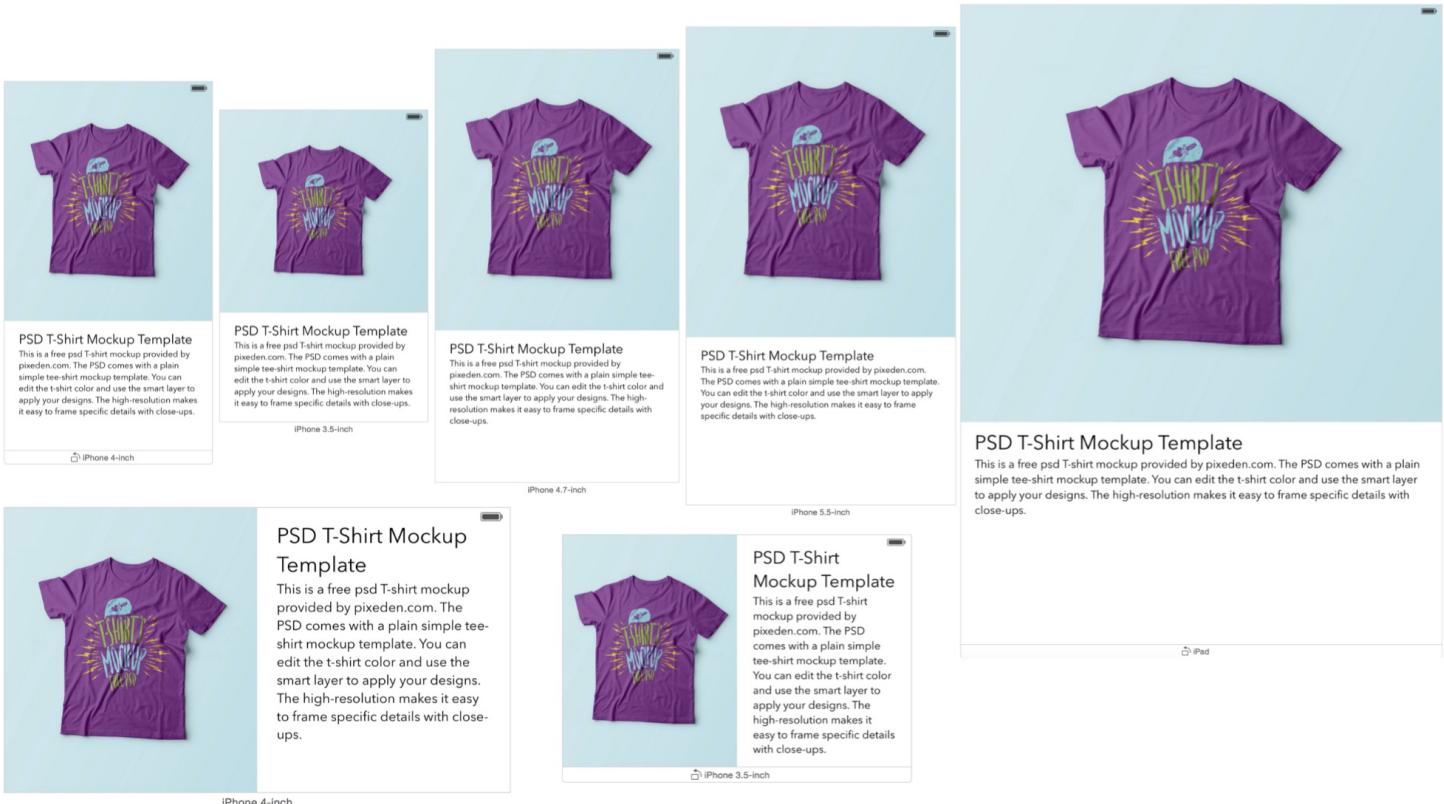
To achieve adaptive layout, you will need to make use of a new concept called *Size Classes*, which is available in iOS 8 and 9. This is probably the most important aspect which makes adaptive layout possible. Size classes are an abstraction of how a device is categorized depending on its screen size and orientation. You can use both size classes and auto layout together to design adaptive user interfaces. In iOS 8/9, the process for creating adaptive layouts is as follows:

- You start by designing a generic layout. The base layout is good enough to support most of the screen sizes and orientations.
- You choose a particular size class and provide your layout specializations. For example, you want to increase the spacing between two labels when a device is in landscape orientation.

In this chapter, I will walk you through all the adaptive concepts such as size classes, by building a universal app. The app supports all available screen sizes and orientations.

Adaptive UI demo

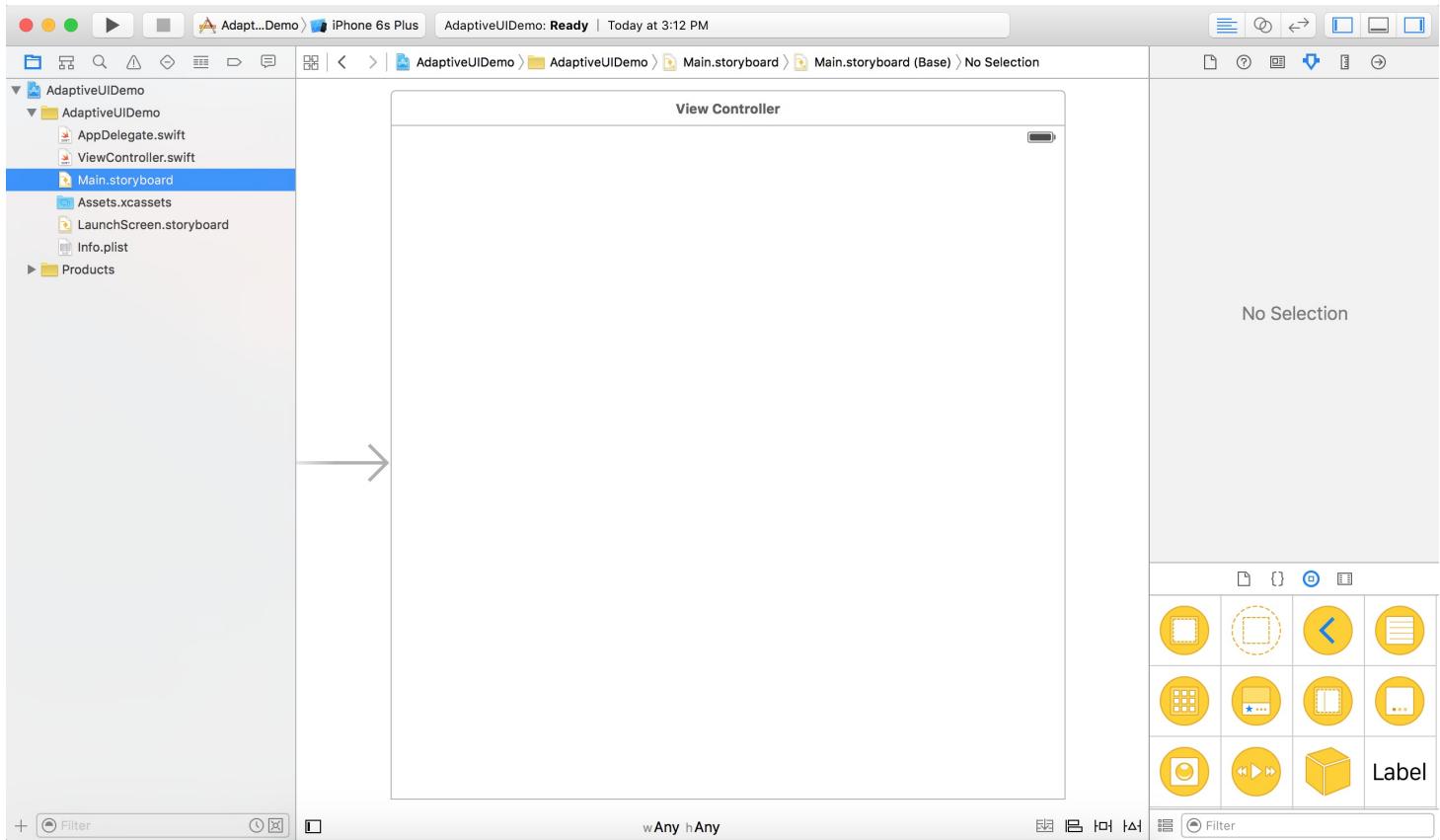
No coding is required for this project. You will primarily use storyboard to lay out the user interface components and learn how to use auto layout and size classes to make the UI adaptive. After going through the chapter, you will have an app with a single view controller that adapts to multiple screen sizes and orientations.



Creating the Xcode Project

First, fire up Xcode, and create a new project using the Single View Application template. In the project option, name the project *AdaptiveUIDemo* and make sure to select *Universal* for the device option.

Once the project is created, open `Main.storyboard`. In Interface Builder, you should find a square view controller with a default 600×600 canvas. With size classes enabled, the view controller represents a generic device instead of a concrete device (e.g. 4-inch iPhone). You can lay out the user interface components (e.g. label), much like in older versions of Xcode, or in a project with size classes disabled.

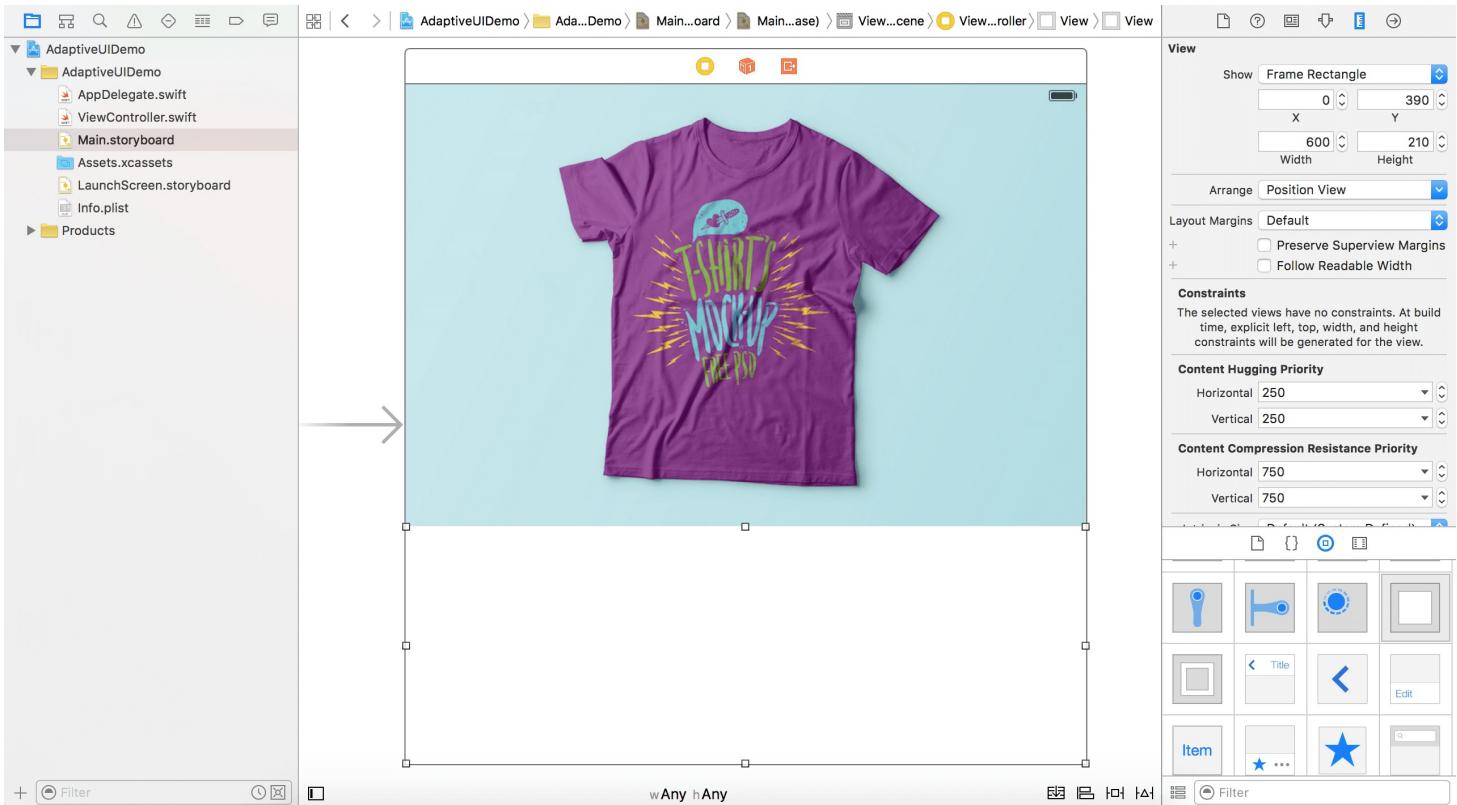


Now download the image pack from

<https://www.dropbox.com/s/1rnlpibodttom8/adaptiveui-images.zip?dl=0> and import the images to `Assets.xcassets`.

Next, go back to the storyboard. Drag an image view from the Object library to the view controller. Set its width to `600` and height to `390`. Choose the image view and go to the Attributes inspector. Set the image to `tshirt` and the mode to `Aspect Fill`.

Then, drag a view to the view controller and put it right below the image view. This view serves as a container for holding other UI components like labels. By grouping related UI components under the same view, it will be easier for you to work with auto layout in a later section. In Size inspector, make sure you set the width to `600` and height to `210`. Throughout the chapter, I will refer to this view as *Product Info View*.

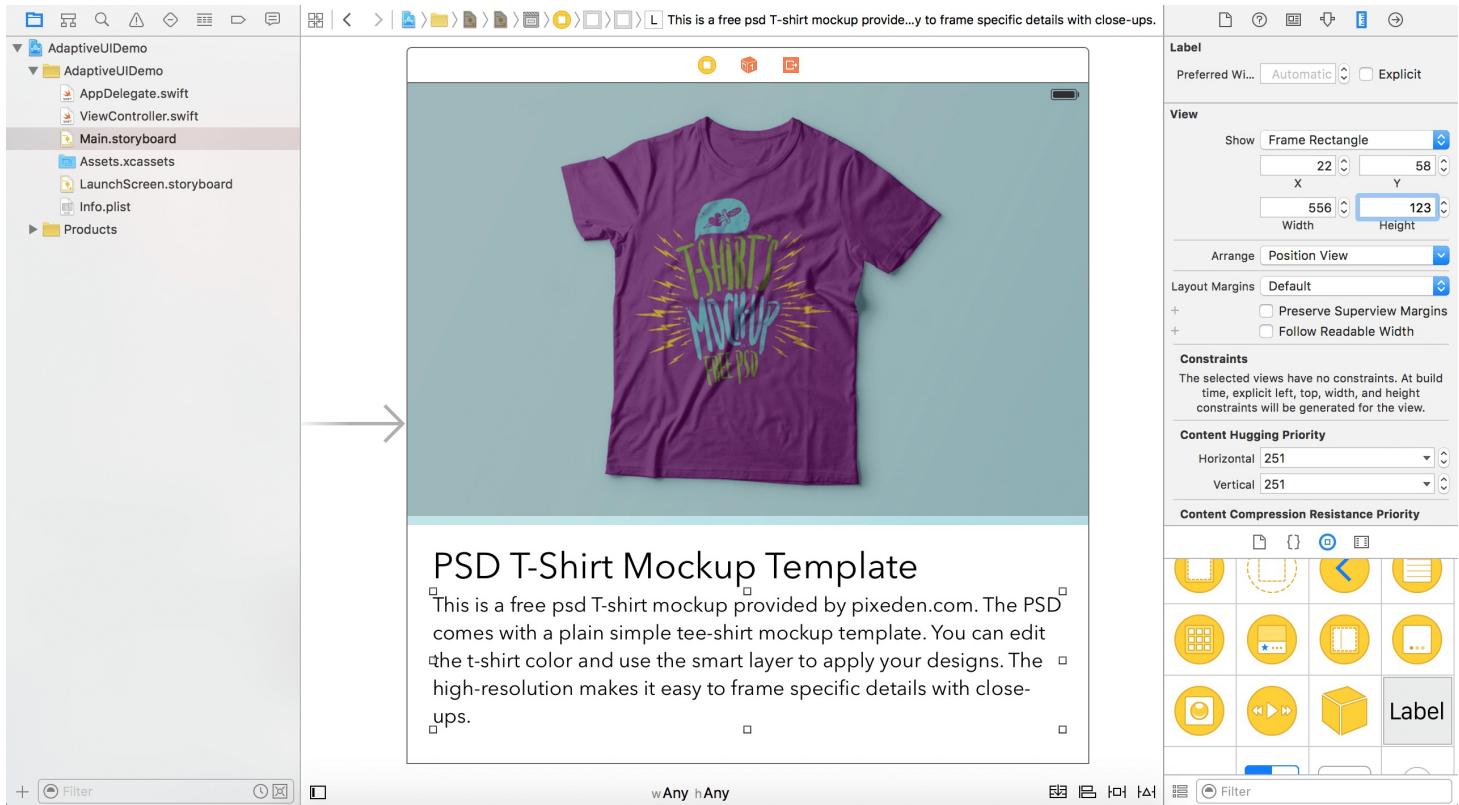


Next, drag a label to *Product Info View*. Change the label to `PSD T-Shirt Mockup Template`. Set the font to `Avenir Next`, and its size to `32` points. In Size inspector, change the value of X to `22` and Y to `15`. Set the width to `556` and height to `44`.

Drag another label and place it right below the previous label. In Attributes inspector, change the text to `This is a free psd T-shirt mockup provided by pixeden.com. The PSD comes with a plain simple tee-shirt mockup template. You can edit the t-shirt color and use the smart layer to apply your designs. The high-resolution makes it easy to frame specific details with close-ups.` and set the font to `Avenir Next`. Change the font size to `18` points and the number of lines to `0`.

Under Size inspector, change the value of X to `22` and Y to `58`. Set the width to `556` and height to `123`.

Note that the two labels should be placed inside *Product Info View*. You can double-check by opening Document Outline. The two labels are put under the view. If you've followed the procedures correctly, your screen should look similar to this:



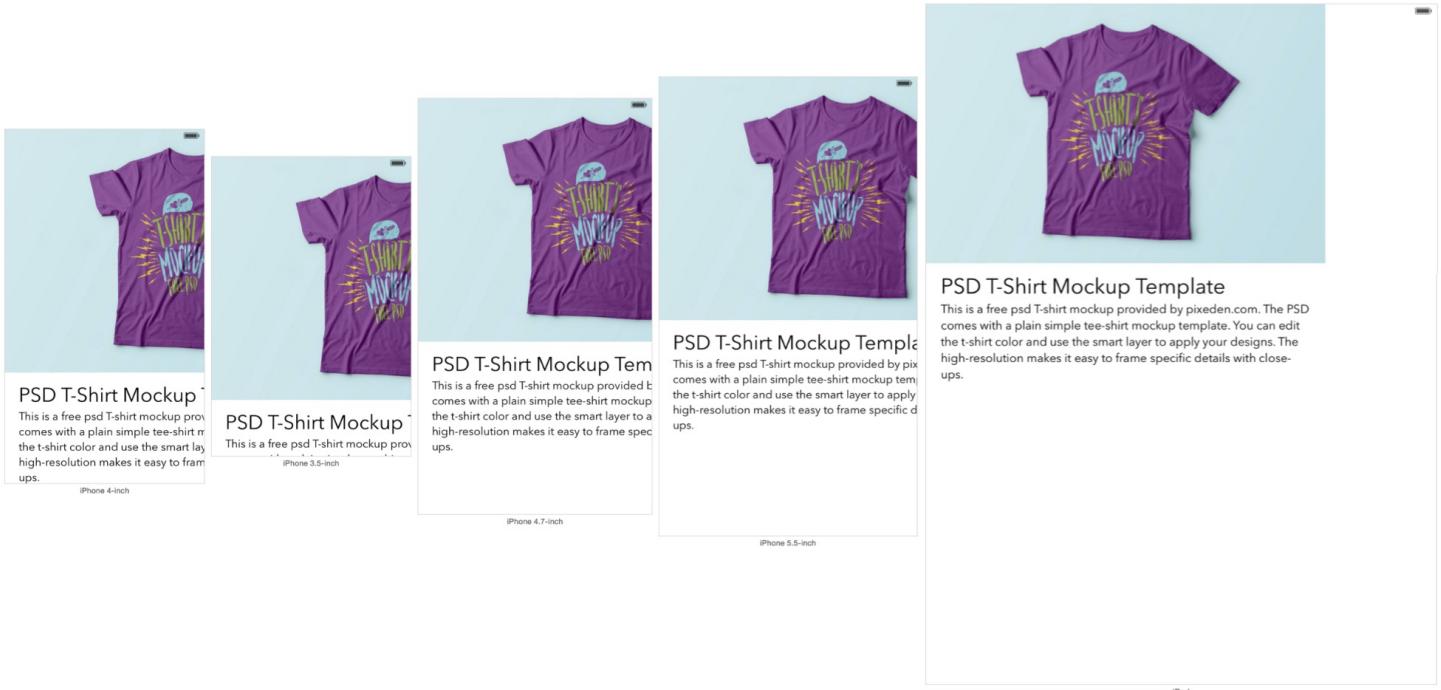
Even if your design does not match the reference design perfectly, it is absolutely fine, as we will use auto layout constraints to lay out the view later. Now, let's conduct a quick test to check out the look and feel of the design on different devices. In Xcode 7, you are not required to run the app in simulators to see how the view appears. The latest version of Xcode comes with a preview assistant which lets developers evaluate the resulting design on different size displays.

In Interface Builder, open the Assistant pop-up menu > Preview (1). Then press and hold option key, and click Main.storyboard (Preview).



Xcode will then display a preview of the app's UI in the assistant editor. By default, it shows you the preview on an iPhone 4-inch device. You can click the + button at the lower-left corner

of the assistant editor to get a preview of an iPhone 3.5-inch and other devices. If you add all the devices including the iPad in the assistant editor, your screen should look like the image pictured below. As you can see, the current design doesn't look good on any of the devices. So far we haven't defined any auto layout constraints. This is why the view doesn't fit properly on any of the devices.



Adding Auto Layout Constraints

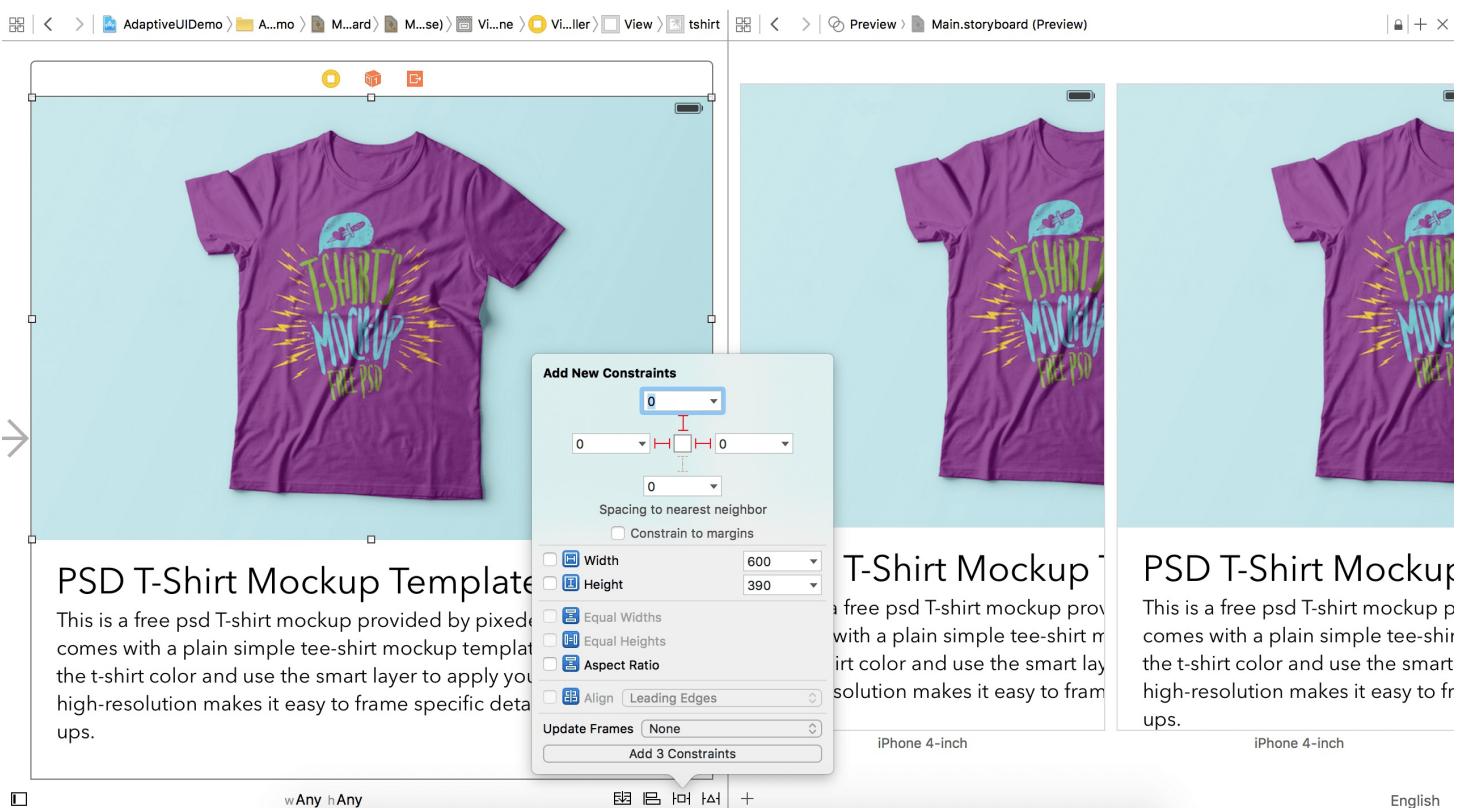
Okay, let's define the layout constraints for the UI components. First, let's start with the image view. Some developers are intimidated by auto layout constraints. I used to start by writing the layout constraints in a descriptive way. Taking the image view as an example, here are some of the constraints I can think of:

- There should be no spacing between the top, left and right side of the image view and the main view.
- The image view takes up 65-70% of the main view.
- There is no spacing between the image view, and the Product Info View should be zero.
- If you translate the above constraints into auto layout constraints, they will convert as such:
- Create spacing constraints for the top, leading (i.e. left) and trailing (i.e. right) edges of the

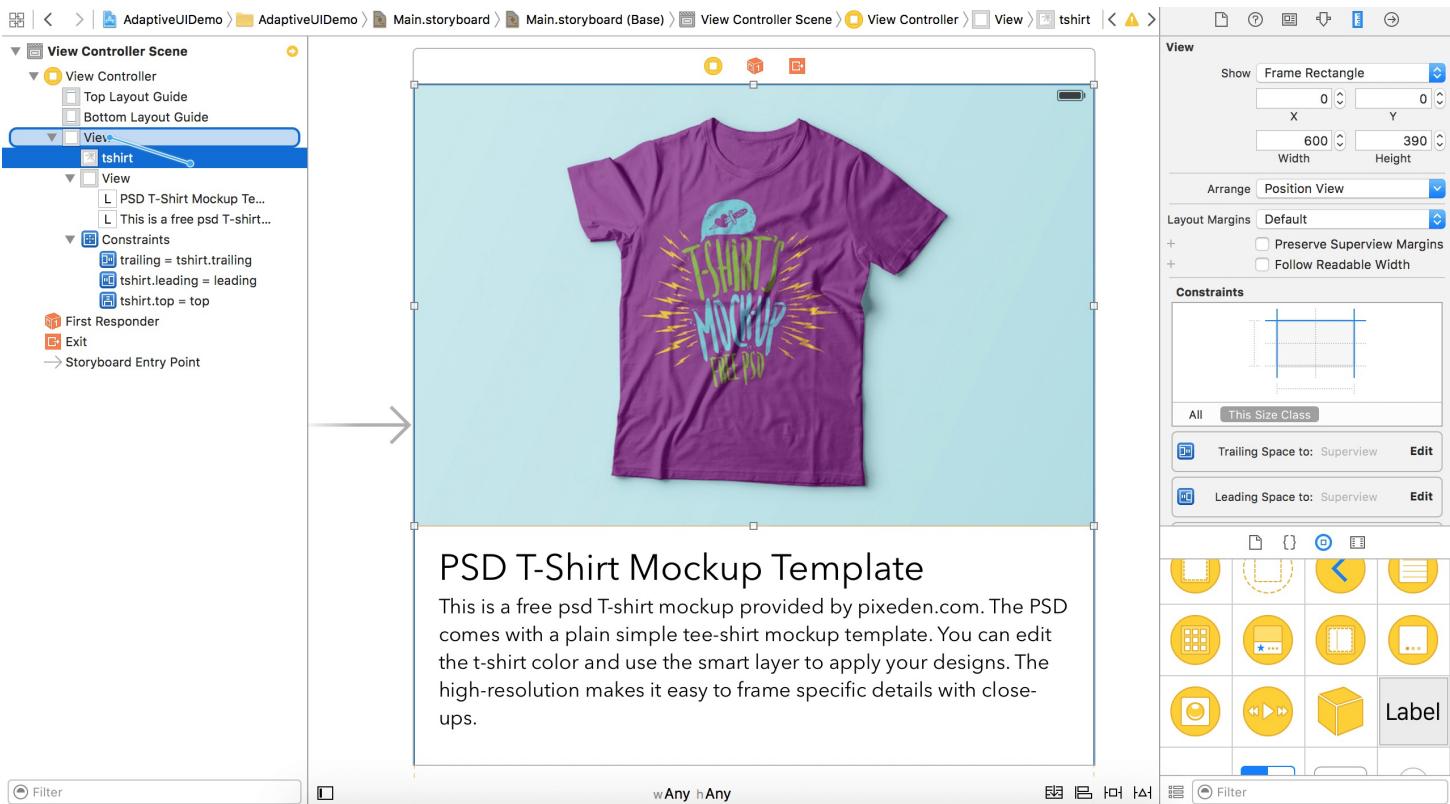
image view. Set the space to zero.

- Define a height constraint between the image view and the main view, and set the multiplier of the constraint to **0.65**.
- Create a spacing constraint between the image view and the Product Info View and set its value to zero.

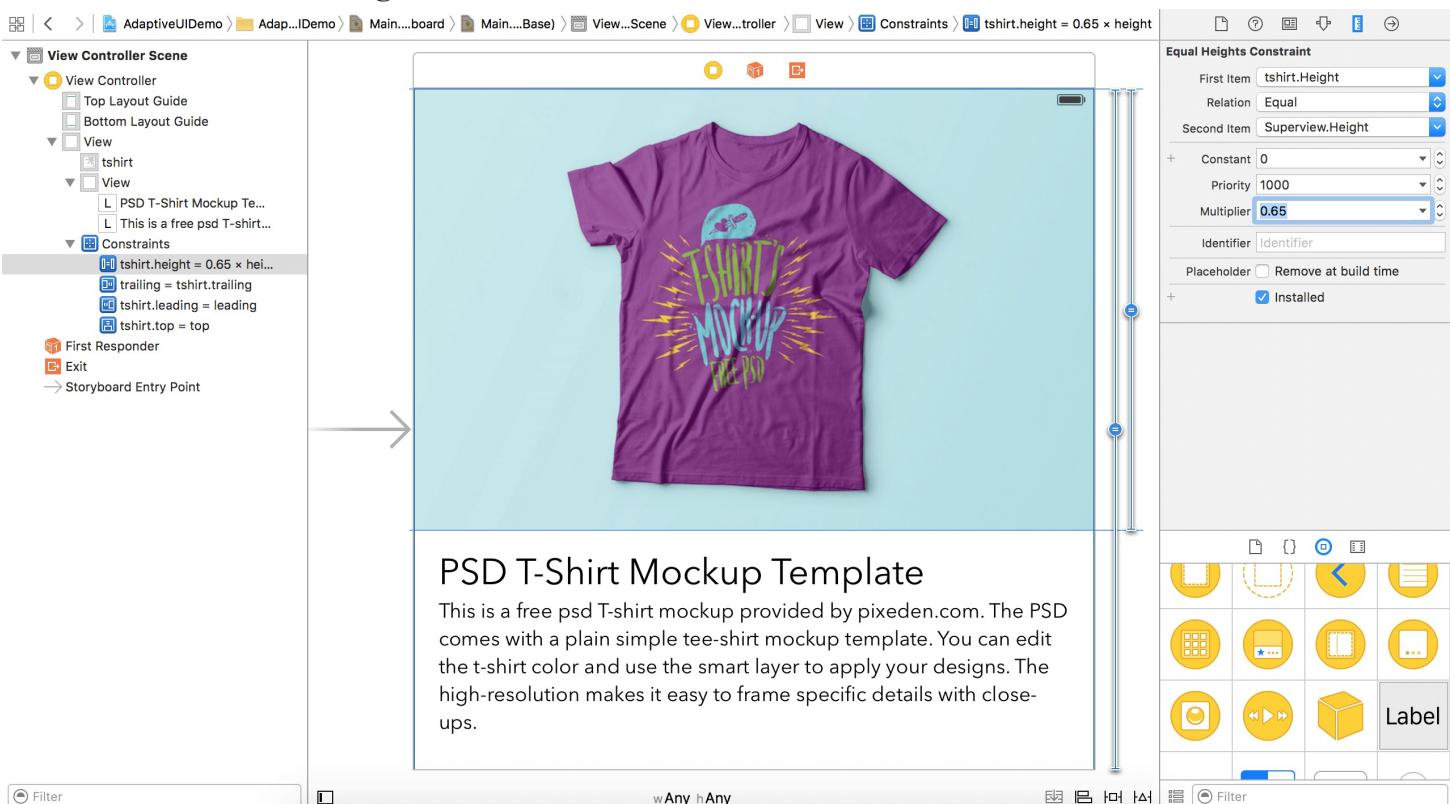
Now select the image view and click the Pin button on the auto layout menu to create spacing constraints. For the left, top and right sides, set the value to **0**. Make sure the **Constrain to margin** option is unchecked because we want to set the constraints relative to the super view's edge. Then click the **Add 3 constraints** button.



Next, open Document Outline. Control-drag from the image view (tshirt) to the main view. When prompted, select **Equal Heights** from the pop-up menu.



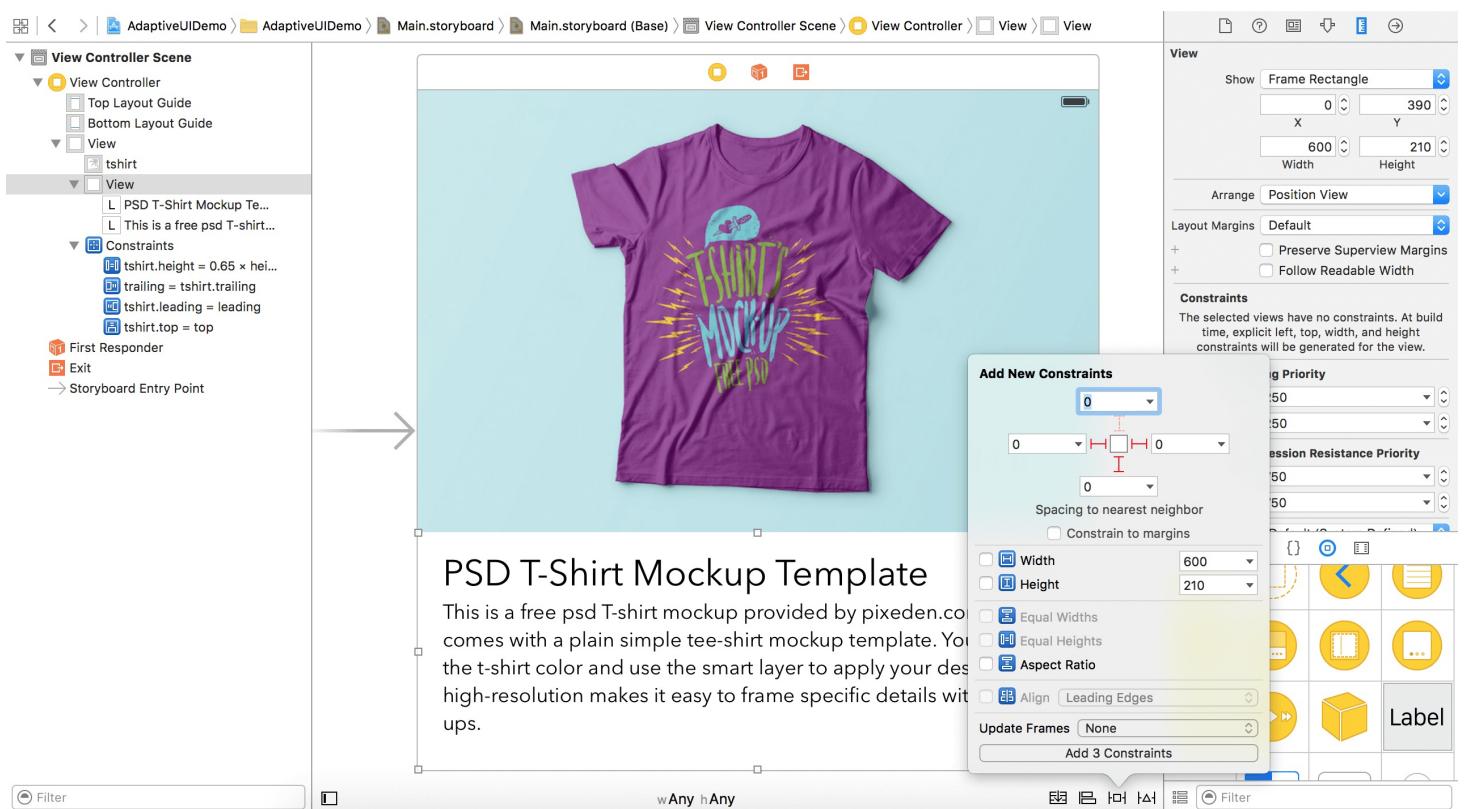
Once the Equal Heights constraint is added, it should appear in the Constraints section of Document Outline. Select the constraint and go to Size inspector. Here you can edit the value of the constraint to change its definition.



Before moving on, make sure the first item is set to `tshirt.Height` and the second item is set to `Superview.height`. If not, you can click the selection box of the first item and select `Reverse First and Second item`.

By default, the value of the multiplier is set to `1`, which means the tshirt image view takes up 100% of the main view (here, the main view is the superview). As mentioned earlier, the image view should only take up around 65% of the main view. So change the multiplier from `1` to `0.65`.

Next, select *Product Info View* and click the *Pin* button. Select the left, right, and bottom sides, and set the value to `0`. Make sure the `Constrain to margin` option is unchecked. Then click the `Add 3 constraints` button. This adds three spacing constraints for the Product Info View.



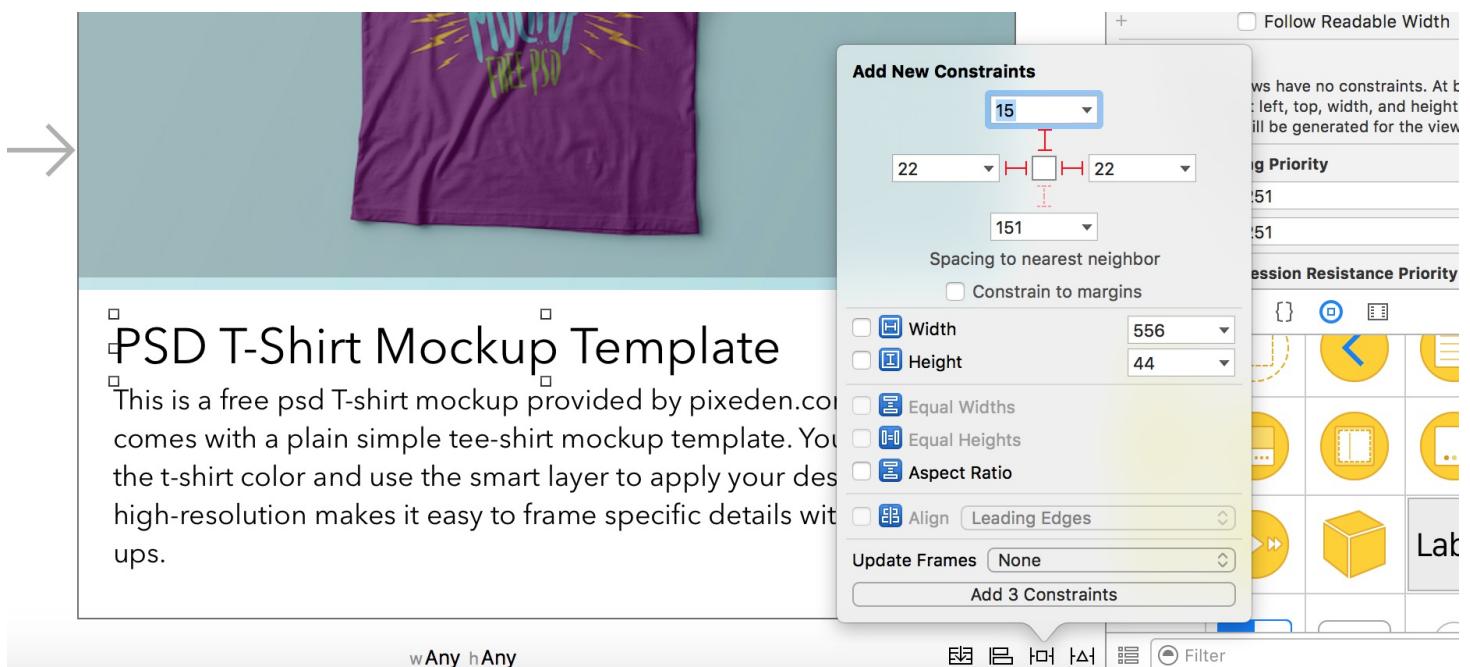
Furthermore, we have to define a spacing constraint between the image view and the Product Info View. In Document Outline, control-drag from the image view (tshirt) to *Product Info View*. When prompted, select `Vertical Spacing` from the menu. This creates a vertical spacing constraint such that there is no spacing between the bottom side of the image view and the top of the Product Info View.



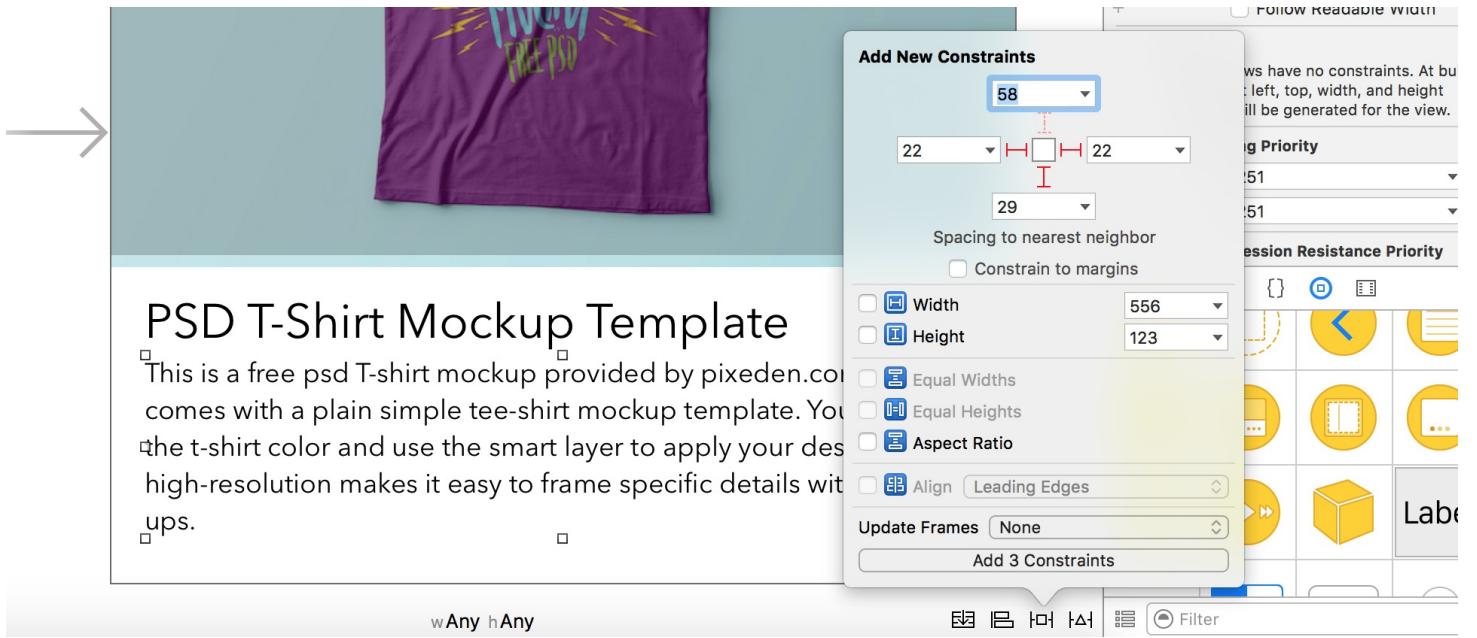
If you take a look at the views rendered in the preview assistant, the view should now look much better on all devices; however, there is still a lot of work to do to perfect the design.

Now let's define the constraints for the two labels.

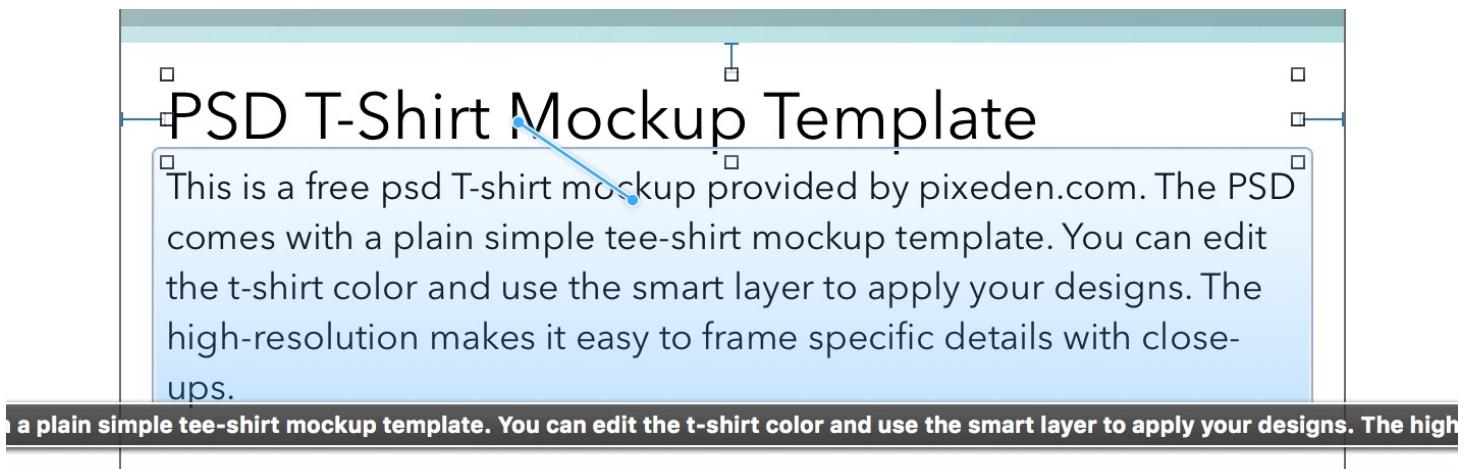
Select the title label, which is the one with the larger font size. Click the Pin button. Set the value of the top side to `15`, left side to `22` and right side to `22`. Again, make sure the `Constrain to margins` is deselected and click `Add 3 Constraints`.



Next, select the other label. Again, we'll add three spacing constraints for the left, right, and bottom sides. Click the Pin button and add the constraints accordingly.



Lastly, define a spacing constraint between these two labels. Control-drag from the title label to the description label. When prompted, select **Vertical Spacing** to create the constraint.

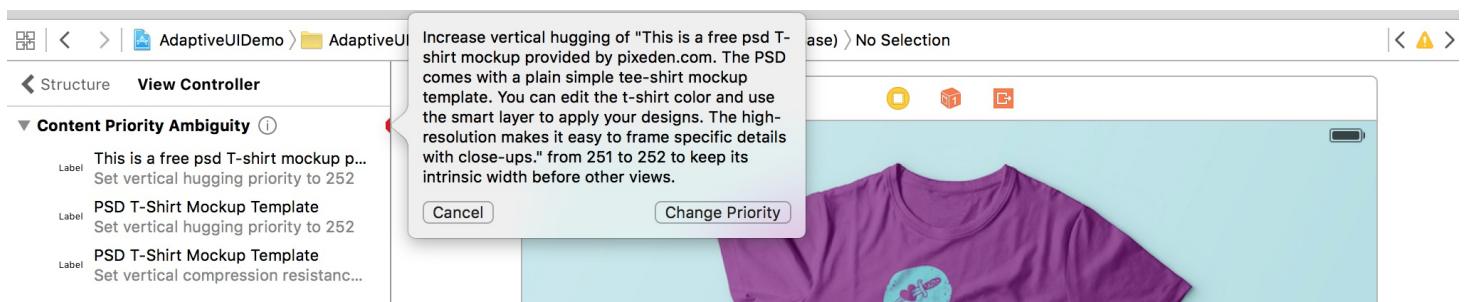


As soon as the constraint is added, you will see a few constraint lines in red, indicating some layout issues. Auto layout issues can occur when some of the constraints are ambiguous. To fix these issues, open Document Outline and click the red disclosure arrow to see a list of the issues.

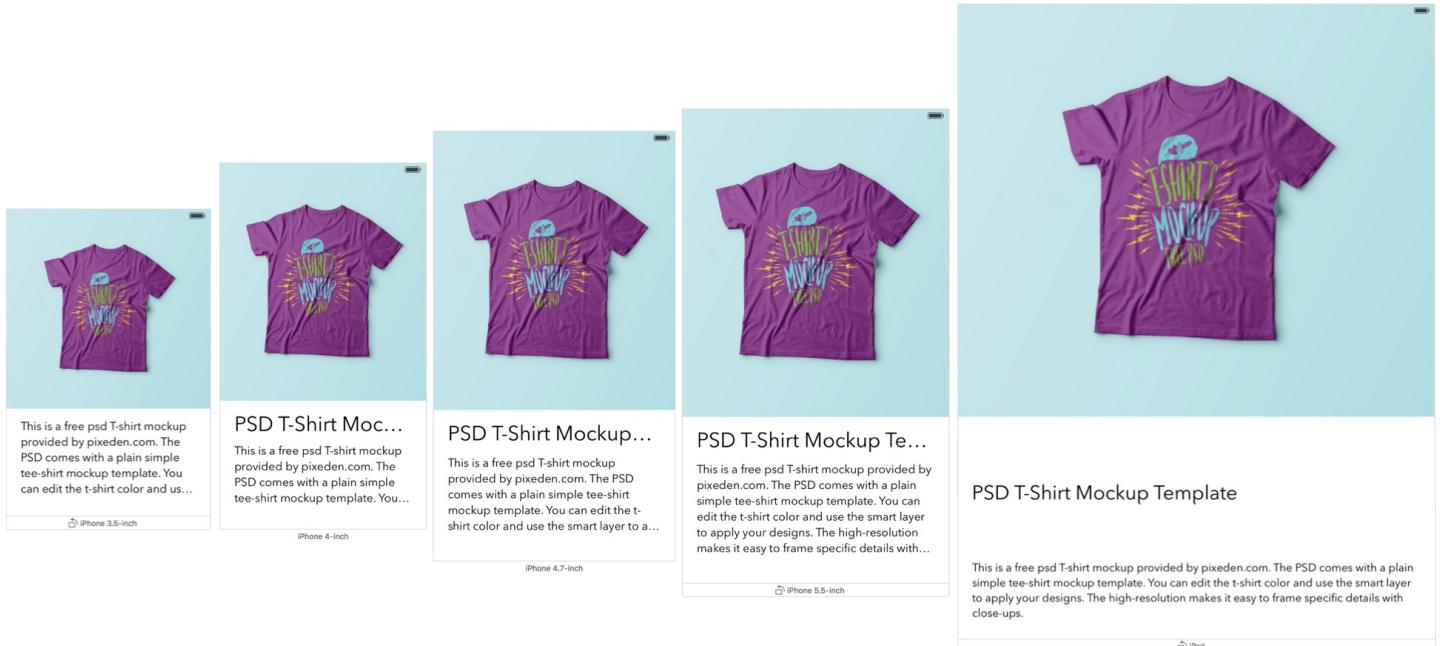
The red arrow indicates some layout issues



Xcode is smart enough to resolve these issues for us. Simply click the indicator icon and a popover shows you the possible solutions. When prompted, click `Change Priority` to resolve these issues.



Cool! You've created all the auto layout constraints. Let's check out the preview and see the result.



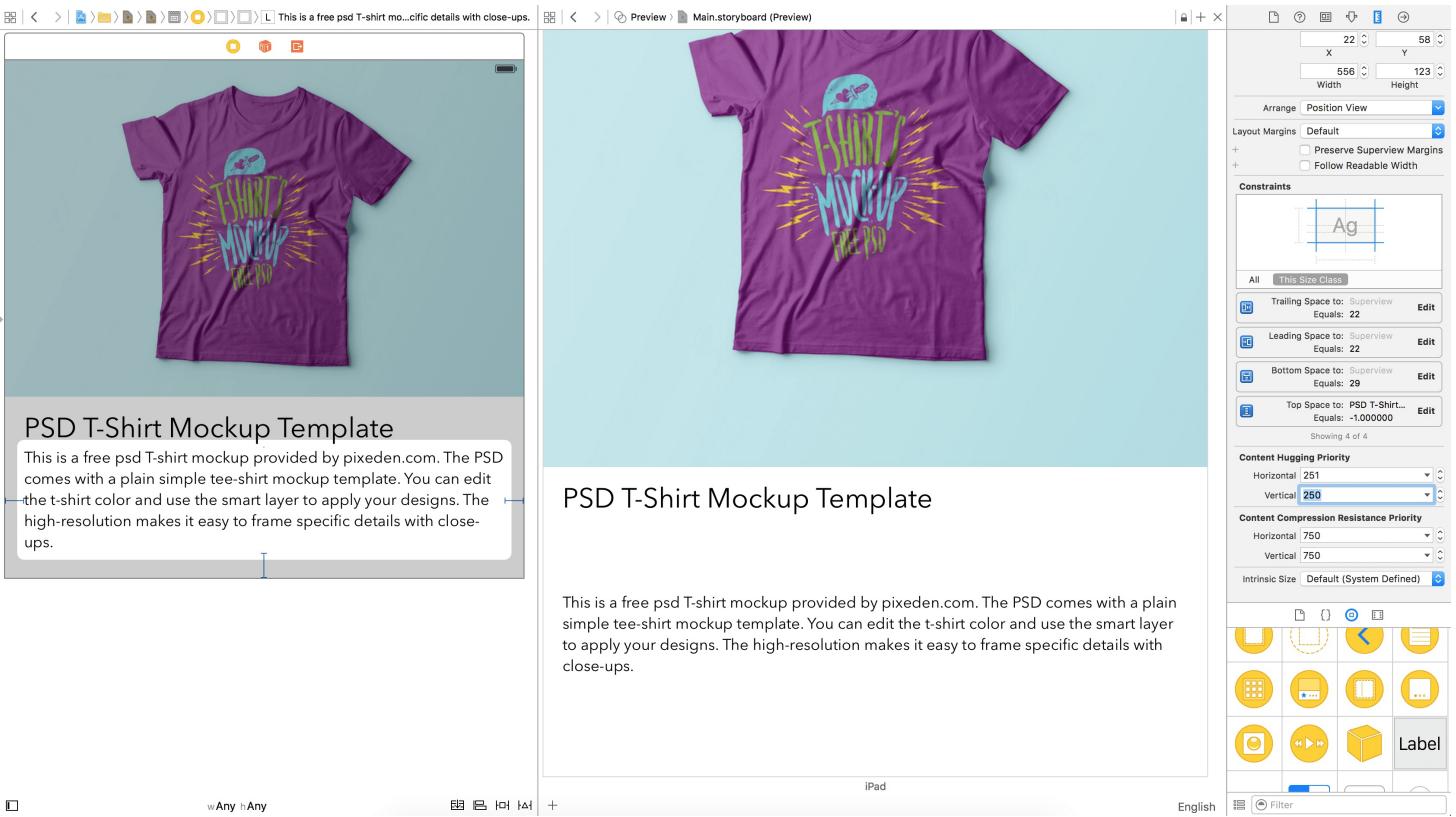
The view looks much better now, with the image view perfectly displayed and aligned.

However, there are still a few issues:

- The title label is vertically centered on iPad. Even worse, the title is hidden on iPhone 3.5-inch.
- The title label is partially displayed for the rest of the iPhone models.
- The description is only fully displayed on all iPhone models. Let's take a look at the first issue. Recalled that we have just asked Xcode to fix the layout issues for us, it has updated the content hugging priority of the title and description labels. If you select the description label and go to the Size inspector, you should notice that the content hugging priority (vertical) is set to `252`. For the title label, the same priority is set to `251`. In other words, the description label has a higher content hugging priority (vertical) than the title label. So what's content hugging priority?

Do you remember that we defined a vertical spacing constraint between these two labels? The constraint said that there is no space between the title and description. On iPad, in order to satisfy this constraint, iOS has to expand either the title or the description label. So which one should be made larger than its intrinsic size? iOS refers to the content hugging priority. The UI element with a lower hugging priority will be chosen. Here, the title label has a lower hugging priority. This is why it is enlarged, while the size of the description label is kept intact.

Now change the content hugging priority (vertical) of the description label from `252` to `250`, and see what you get.



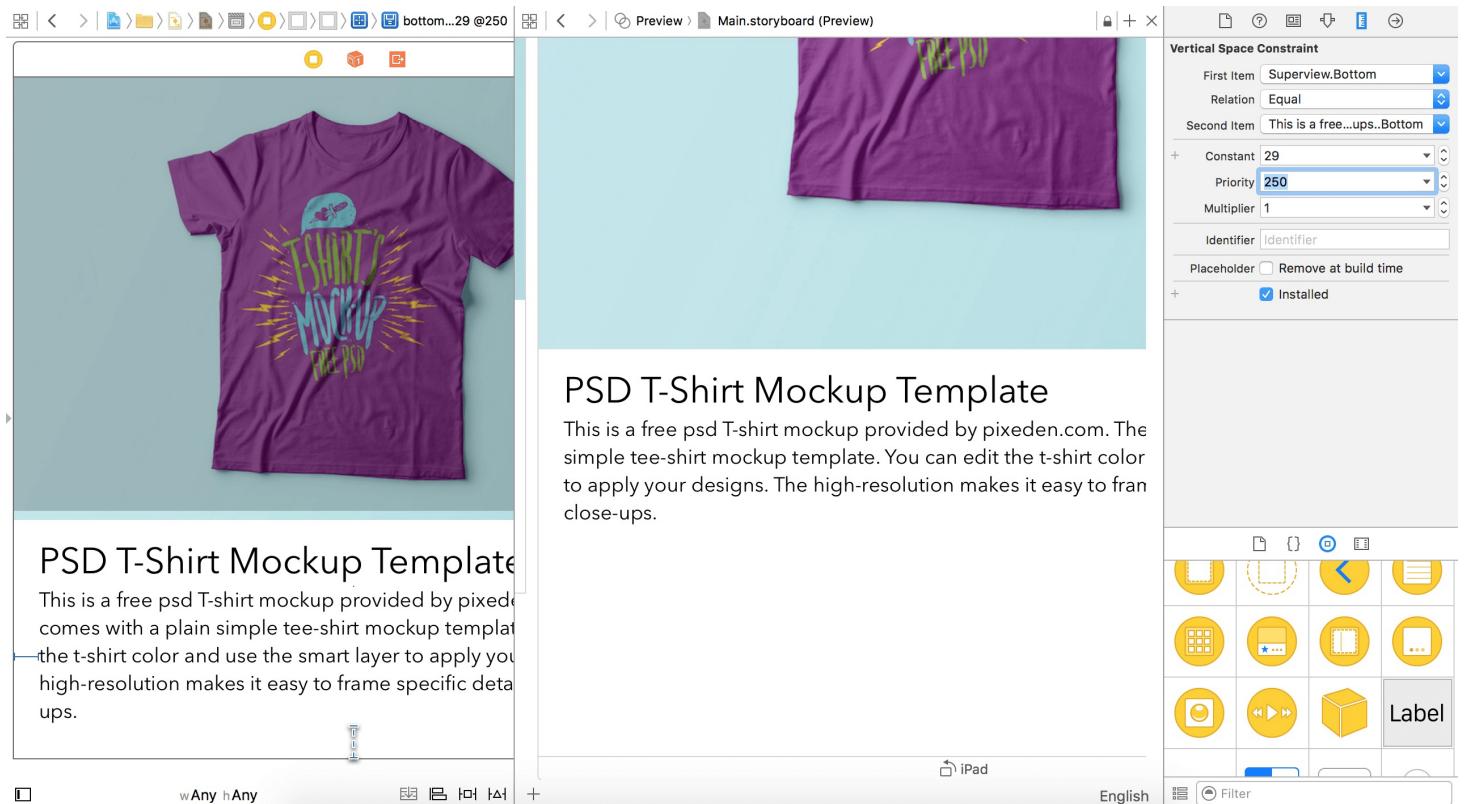
The hugging priority of the title label is now higher than that of the description label. iOS chooses to enlarge the description label. And, if you look at the preview of iPhone 3.5-inch, it now displays the title label.

Okay, it looks like the title label is displayed properly on the iPad. But the description label still doesn't look good. We want it to be placed right below the title label. Let's see how we can fix it.

The issue is related to the constraints defined in the description label. You can always view the constraints of a particular component under Size inspector. Select the description label and go to Size inspector. You will find a list of constraints under the Constraints section. We have defined four spacing constraints for the label. The issue is likely related to the top and bottom spacing constraints. In auto layout, constraints have a priority level. Constraints with higher priority levels are satisfied before constraints with lower priority levels. By default, all constraints are assigned with the same level (i.e. 1000). This means that they are important and must be satisfied. Here both the top and bottom spacing constraint have the same priority, but it appears the bottom spacing constraint wins, contributing to a large spacing between the title and description label.

To fix the issue, the bottom spacing constraint should be set to a lower priority level. Click the

`Edit` button of the Bottom Space constraint. Set the priority value to `250`. Once the change is made, the spacing issue should be fixed.



Size Classes

Remember, designing adaptive UI is a two-part process. So far we have just finished the generic layout. The base layout is good enough to support most screen sizes. The second part of the process is to use size classes to fine-tune the design.

A size class identifies a relative amount of display space for both vertical (height) and horizontal (width) dimensions. There are two types of size classes in iOS 8: *regular* and *compact*. A regular size class denotes a large amount of screen space, while a compact size class denotes a smaller amount of screen space.

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

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

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

To characterize a display environment, you must specify both a *horizontal* size class and *vertical* size class. For instance, an iPad has a regular horizontal (width) size class and a regular vertical (height) size class.

With the base layout in place, you can use size classes to provide layout specializations which override some of the design in the base layout. For example, you can change the font size of a label for devices that adopt compact height-regular width size. Or you can change a position of a button particularly for the regular-regular size.

Note that all iPhones in portrait orientation have a *compact* width and a *regular* height. In other words, your UI will appear almost identically on an iPhone 4s as it does on an iPhone 6.

The iPhone 6 Plus, in landscape orientation, has a *regular* width and *compact* height size. This allows you to create a UI design that is completely different from that of an iPhone 4/5/6.

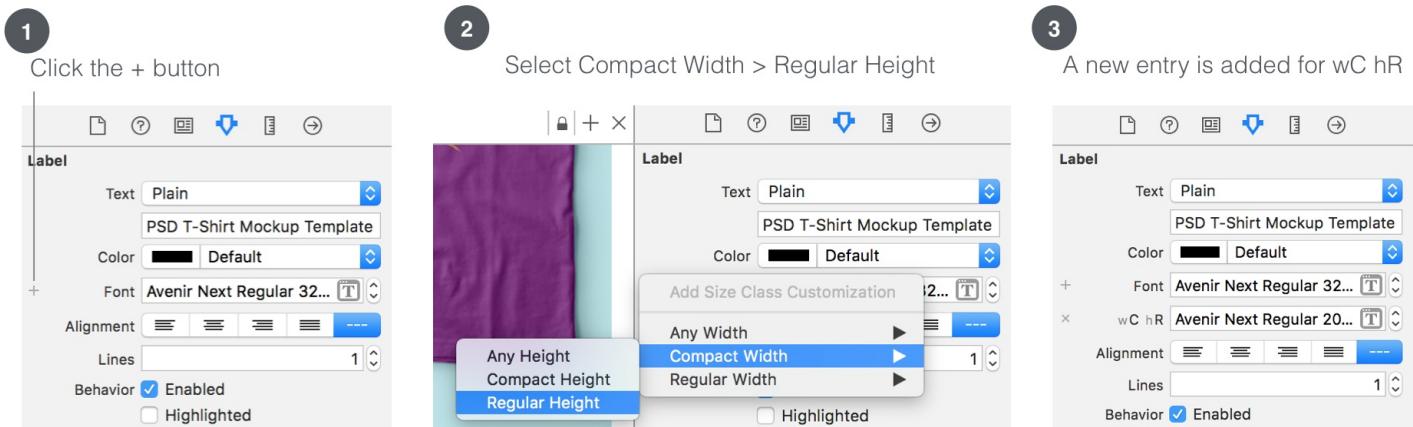
With some basic understanding of size classes, let's see how to fix the two design issues listed below:

- The title cannot be displayed perfectly on all iPhone models.
- The description is partially displayed on all iPhone models.

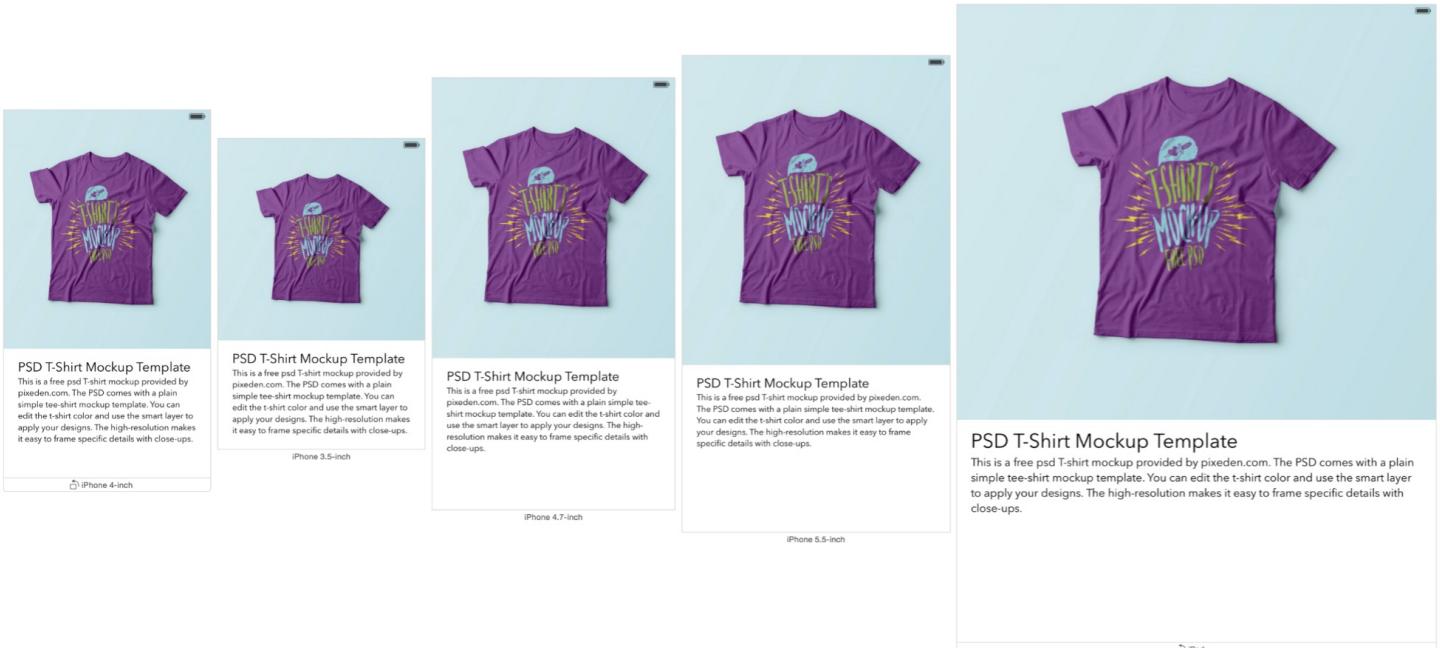
Using Size Classes for Font Customization

We want to make the title and description labels perfect for iPhones. The current font size is ideal for the iPad but too large when it is used on an iPhone. With size classes, you can now adjust the font for a particular screen size. Here, we want to change the font size for all iPhone models in portrait orientation. In terms of size classes, the iPhone in portrait defaults to *compact* size class for horizontal (width) and *regular* size class for vertical (height).

To set the font size for this particular size class, first select the title label. Under the Attributes inspector, you should see a plus (+) button next to the font field. Click the + button and select Compact Width > Regular Height. You will then see a new entry for the Font option, which is dedicated to that particular size class. Keep the size intact for the original Font option but change the size of `wC hR` font field to 20 points.



This will instruct iOS to use the second font with a smaller font size on iPhones (portrait). For the iPad (and iPhone landscape), the original font will still be used to display the text. Now select the description label. Again, under the Attributes inspector, click the + button and select Compact Width > Regular Height. Change the size of `wC hR` font field to 13 points. Look at the preview. The layout looks perfect on all screen sizes.



Using Size Classes to Customize a View

Now that the UI adapts perfectly for all devices in portrait orientation, how do they look in landscape orientation? In the preview assistant, click the rotate button on a device (e.g. iPhone 4-inch). Obviously, the view looks off for iPhones in landscape orientation.



PSD T-Shirt Mockup Template

This is a free psd T-shirt mockup provided by pixeden.com. The PSD comes with a plain simple tee-shirt mockup template. You

iPhone 4-inch

A simple way to fix the issue is to use a smaller font for both title and description labels. You should know how to do that. Instead of tweaking the existing design, I will show you how to create another design for the view to take advantage of the wider screen size. This is the true power of size classes.

With a wider but shorter screen size, it would be best to present the image view and Product Info View side by side; each takes up 50% of the main view. This screen shows the final view design for iPhone landscape.

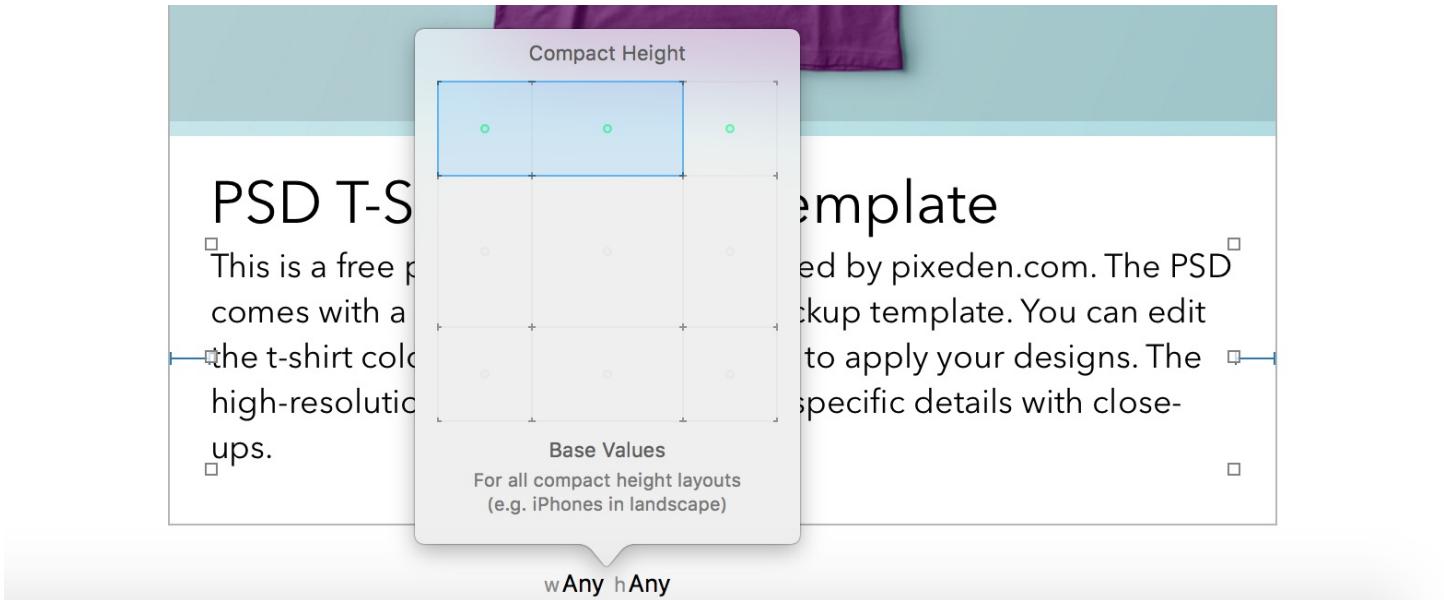


PSD T-Shirt Mockup Template

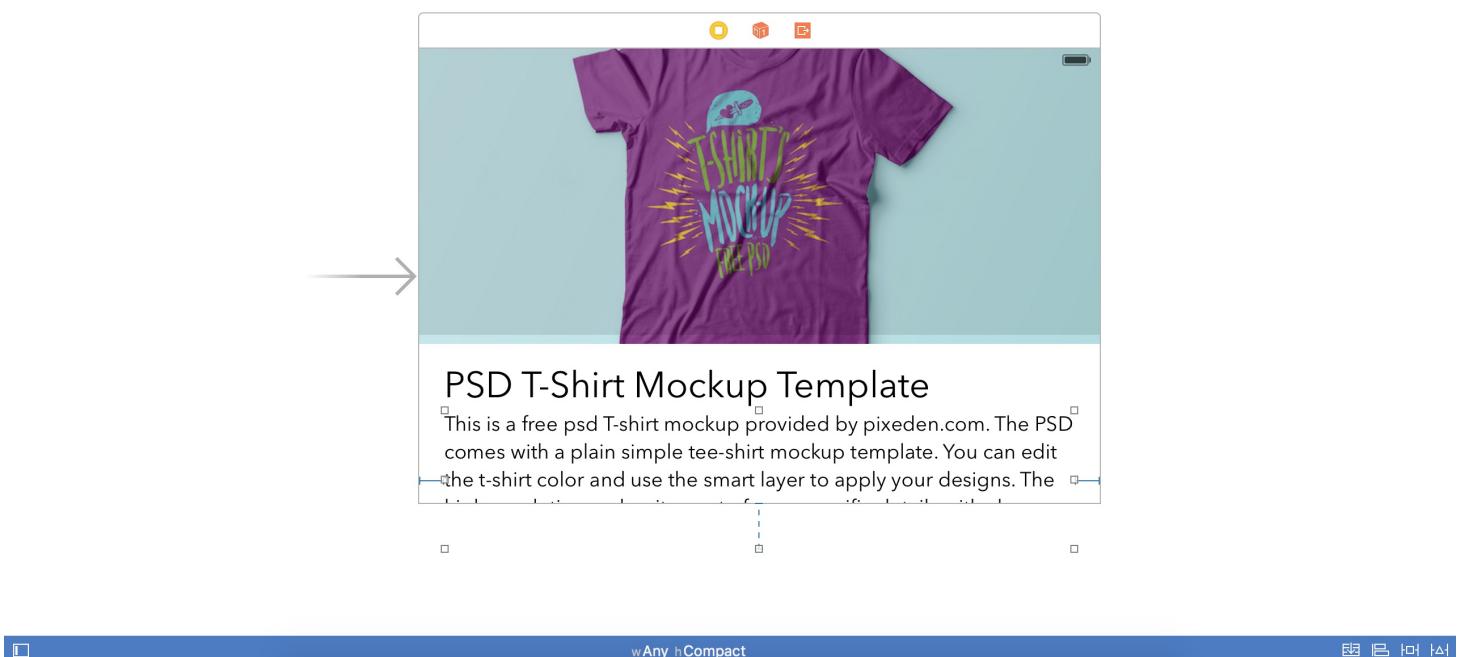
This is a free psd T-shirt mockup provided by pixeden.com. The PSD comes with a plain simple tee-shirt mockup template. You can edit the t-shirt color and use the smart layer to apply your designs. The high-resolution makes it easy to frame specific details with close-ups.

So far, we haven't discussed the size class control in Interface Builder. We just designed the view using the `wAny hAny` size classes.

Now, click the size class control (`wAny hAny`) at the bottom of the Interface Builder canvas. The control presents a grid for specifying width and height combinations. As we are going to customize the layout of the iPhone in landscape, we will provide layout specializations for Any-Compact size. This size class combination represents all iPhones in landscape orientation.



Move your pointer to set the width class to *Any*, and the height class to *Compact*.

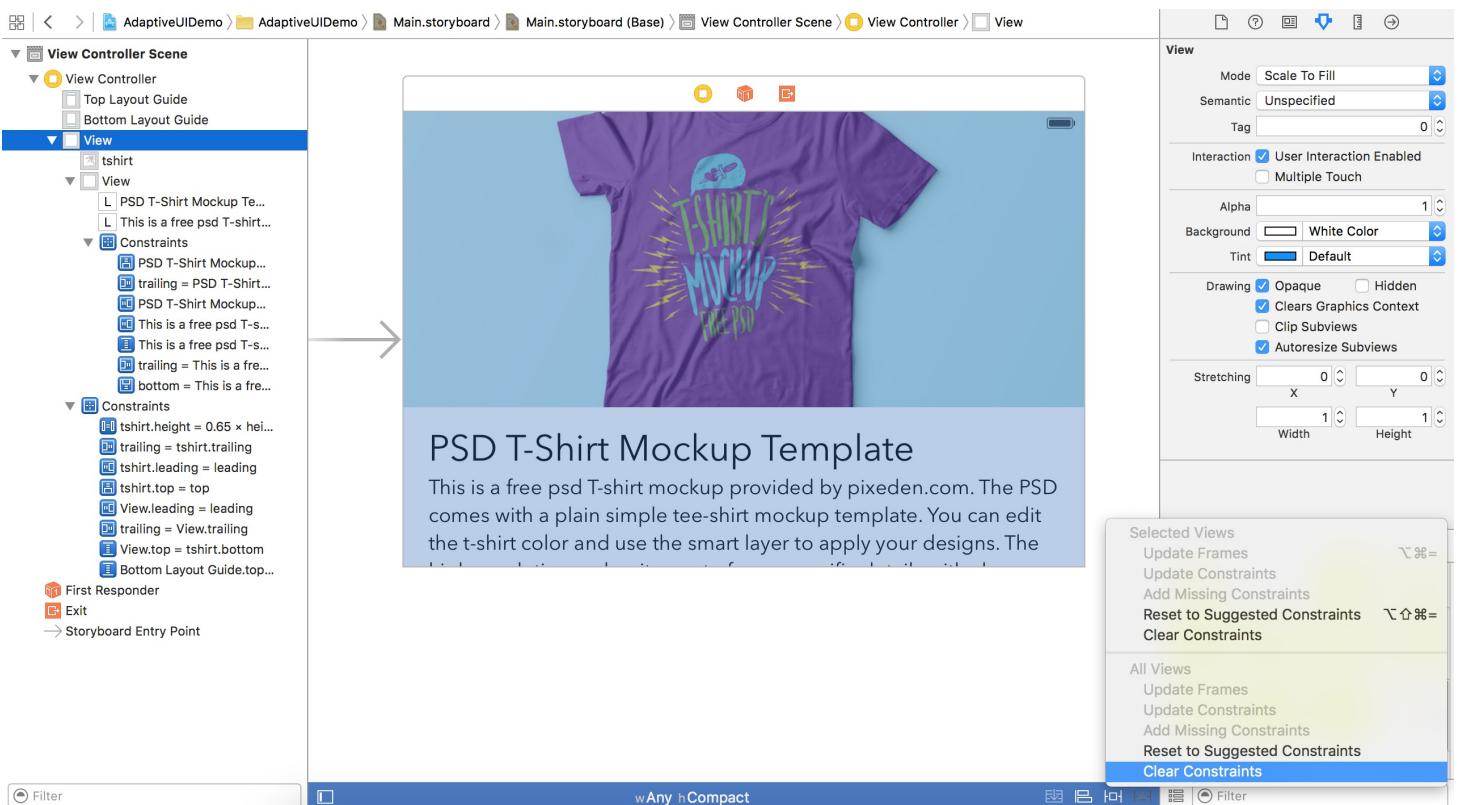


Once you've clicked to confirm the selected size class, the dimension of the view controller changes accordingly. Interface Builder indicates the current size class (`wAny hCompact`) in the layout toolbar. Any layout change you are going to make will only apply to this size class combination. This is how Xcode allows you to design different screen layouts in a single storyboard.

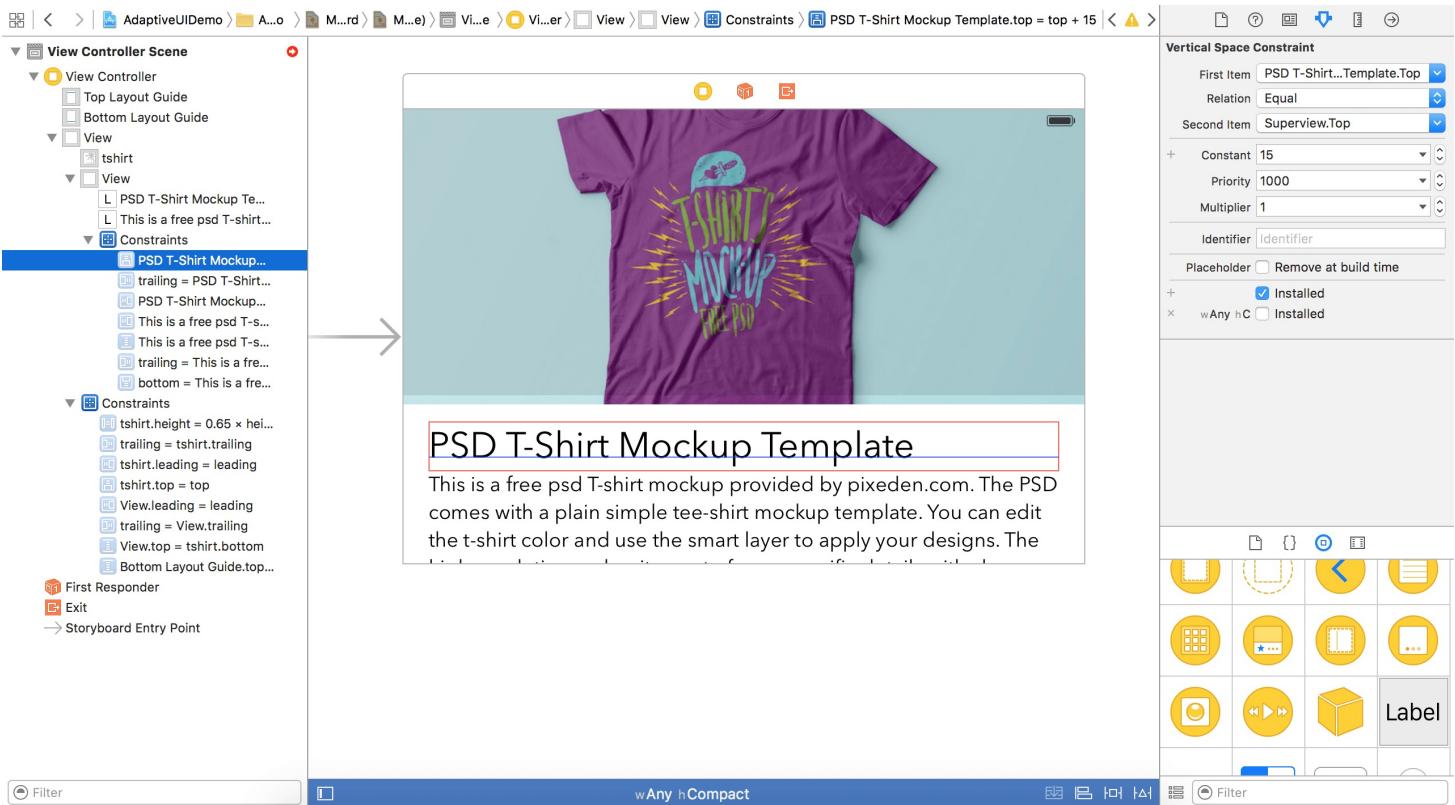
Okay, let's start to design the view.

The current view derives the design and layout constraints from the generic layout (wAny hAny). Since we are going to redesign the view, we will first remove the layout constraints, which are no longer suitable for the new design.

In Document Outline of Interface Builder, select the main view. Then click the *Issues* button of the layout menu and select *Clear Constraints* under the All Views section.

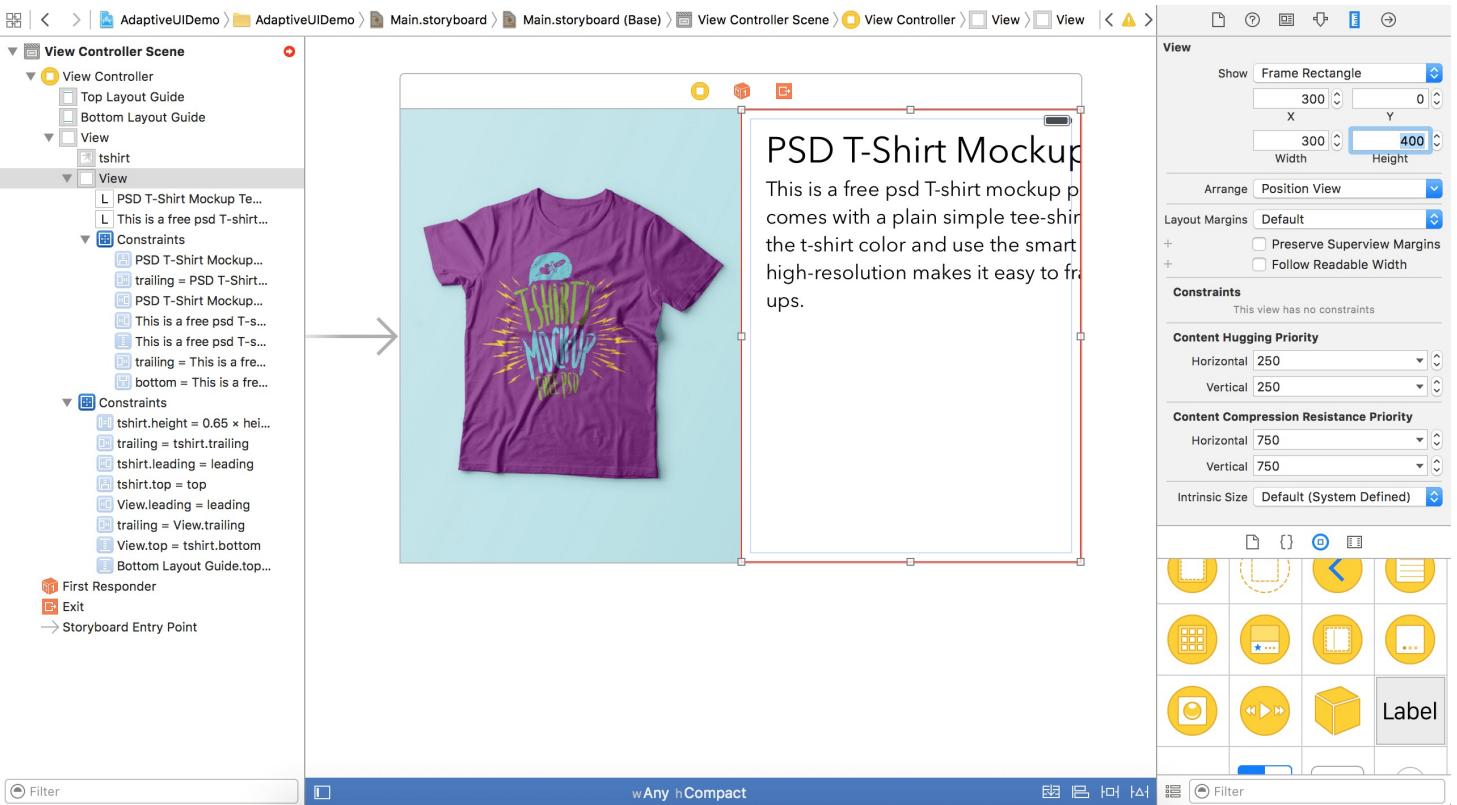


This would clear all layout constraints for this particular size class combination. Note that the layout constraints still apply to the generic layout; we just removed them from *Any-Compact*. You can open Document Outline and expand the Constraints section. All the constraints are still there but grayed out for this particular size class combination. If you select any of the constraints and go to Size inspector, you will find that the Installed option of wAny hC is deselected; however, the Installed option of the generic layout is still there. This indicates that you only removed the constraints for the current size class selection.



Next, we will design the view such that the image view and Product Info View are displayed side by side.

First, select the image view (tshirt). Under Size inspector, change the value of X to `0`, Y to `0`, width to `300` and height to `400`. Then select the Product Info View from Document Outline and change its size using the Size inspector. Set the value of X to `300`, Y to `0`, width to `300`, and height to `400`. The resulting screen should look like this:

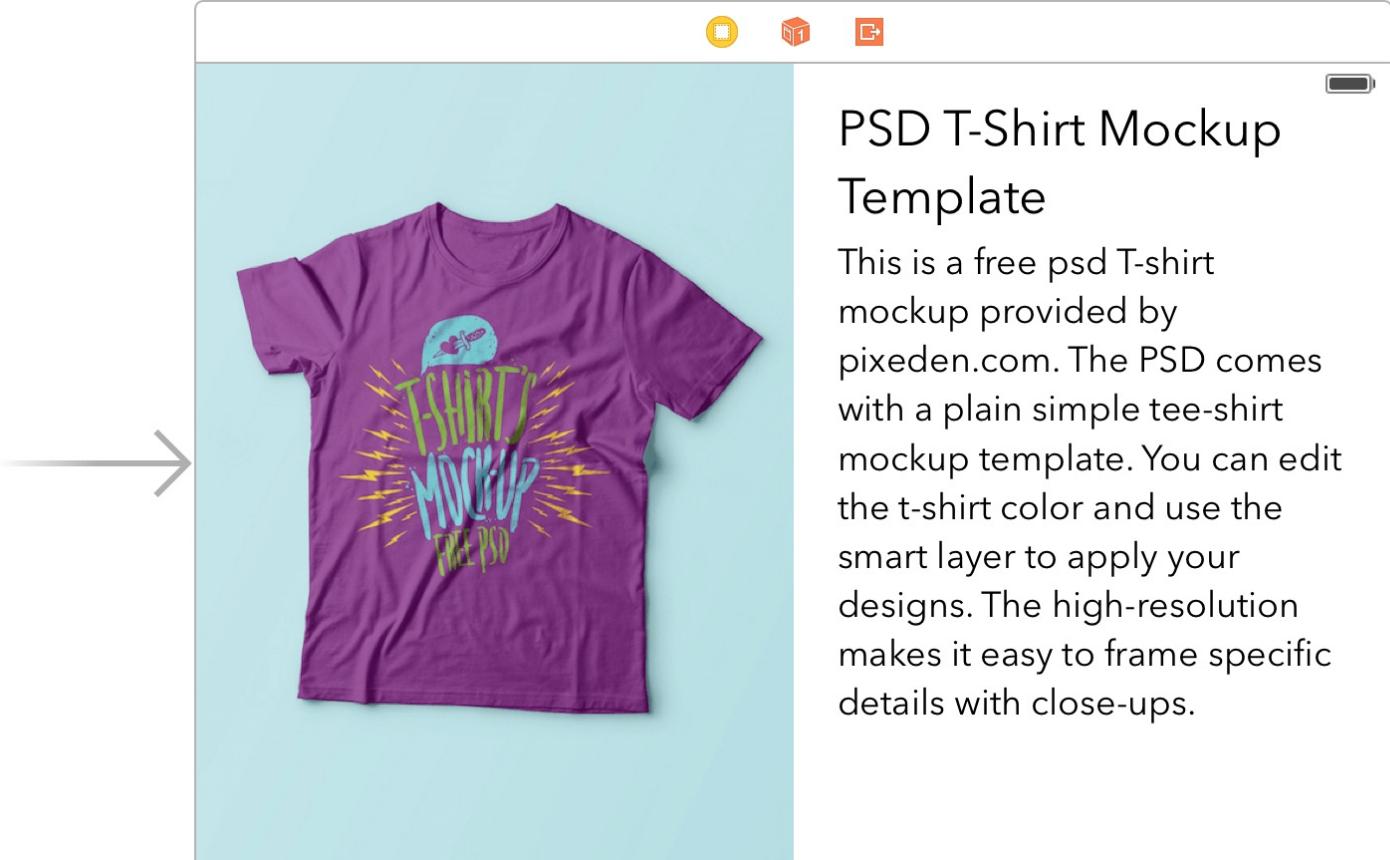


Now select the title label. Under Size inspector, set the value of X to 22, Y to 15, width to 256 and height to 70.

The font is too large, so we will add a size class dependent font for the label. In the Attributes inspector, click the + button next to the font option. Select Any Width | Compact Height (current) option. This adds another font option for the current size combination. Change the font size to 25 points. Presently, the title label can only display one line. Change the value of lines from 1 to 0; Interface Builder will determine the number of lines automatically.



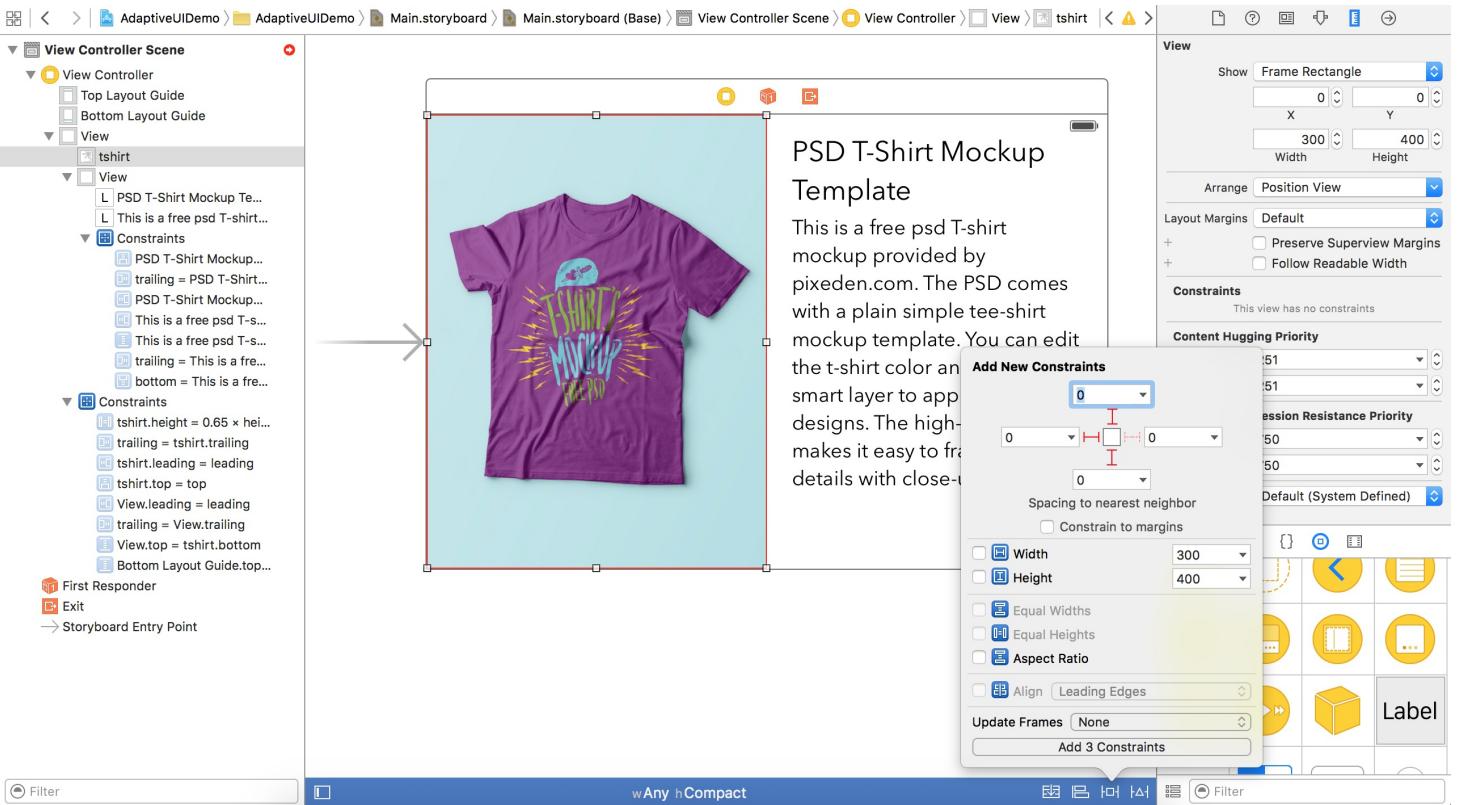
Select the description label. Under Size inspector, change the value of X to 22, Y to 85, width to 256, and height to 250. The new design will look like this:



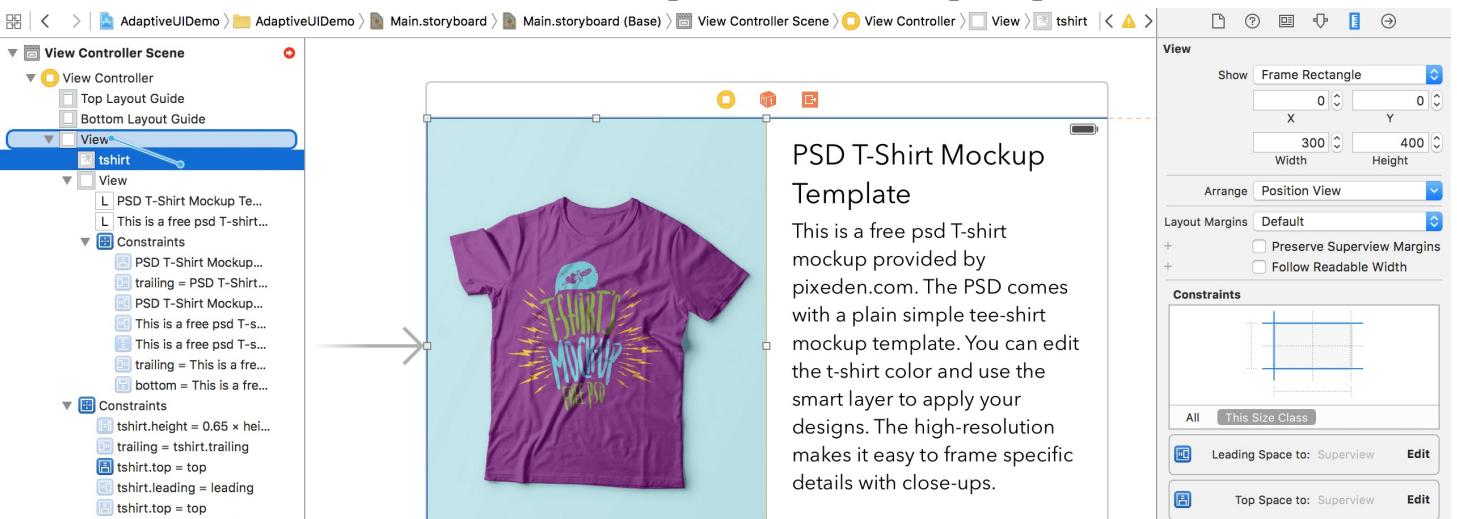
PSD T-Shirt Mockup Template

This is a free psd T-shirt mockup provided by pixeden.com. The PSD comes with a plain simple tee-shirt mockup template. You can edit the t-shirt color and use the smart layer to apply your designs. The high-resolution makes it easy to frame specific details with close-ups.

Because we have removed all the layout constraints, we must define a new set of constraints for elements in this particular size class combination. Select the image view and click the Pin button of the layout menu in order to define a few spacing constraints. Click the top, bottom, and left sides. Set each of the values to `0`. Remember to deselect the `Constrain to margins` option. Click the `Add 3 Constraints` button to add the constraints.

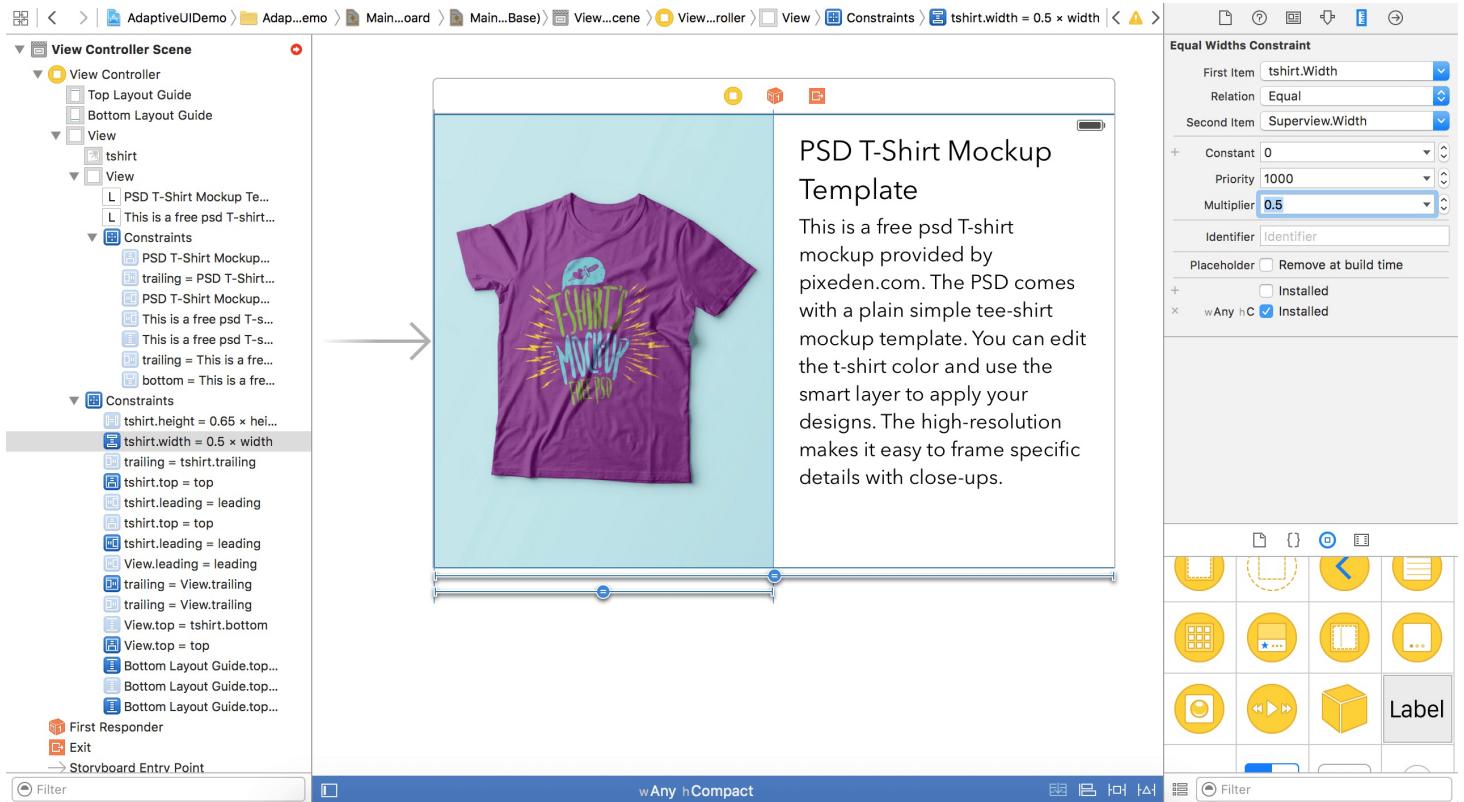


Next, select *Product Info View* and click the Pin button. Set the value of the top, bottom and right sides to `0`. Again, deselect the `constrain to margins` option. Click the `Add 3 Constraints` button to add the constraints. In Document Outline, control-drag from the image view (t-shirt) to the main view, and select *Equal Widths* when prompted.

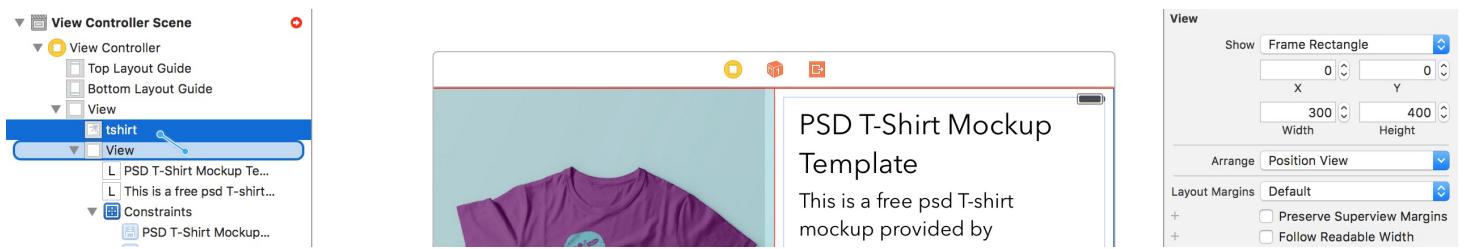


This constraint is used to ensure the image view takes up half of the main view. Like the Equal Heights constraint that we worked with earlier, we have to change its multiplier so that the image view will only take up 50% of the main view.

Expand the Constraints section in Document Outline, and select the Equal Widths constraint. Under Size inspector, change the multiplier from `1` to `0.5`. Note that your first item should be set to `tshirt.width` and the second item set to `Superview.Width`. Otherwise, select the first item and click the Reverse First and Second item option.



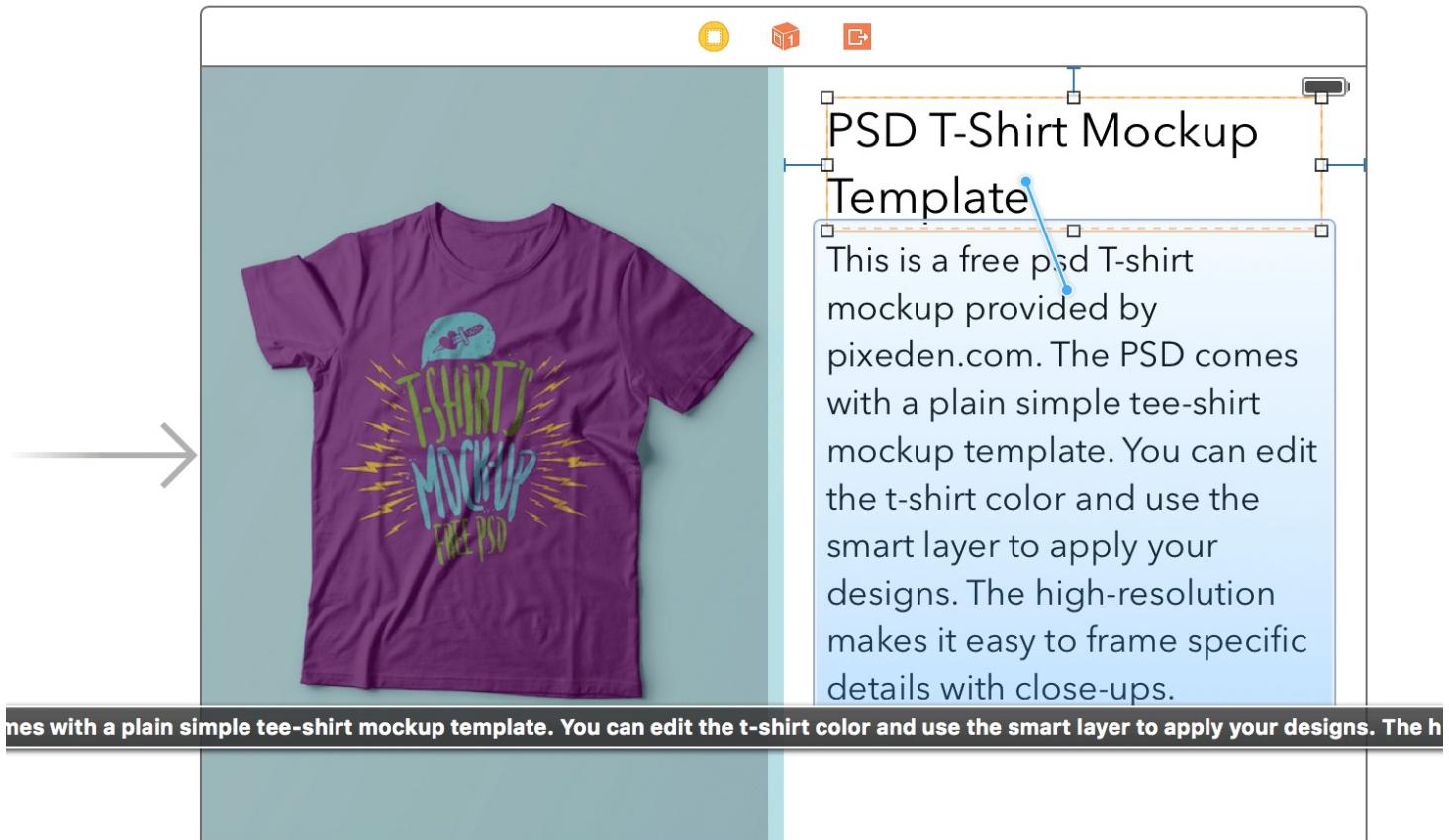
Next we will define a spacing constraint between the image view and *Product Info View*. In the Document Outline, control-drag from the image view (tshirt) to *Product Info View*. Select *Horizontal Spacing* when a pop-up appears.



Okay, now that we've defined the constraints for the image view and the Product Info View, what's left are the constraints for the labels.

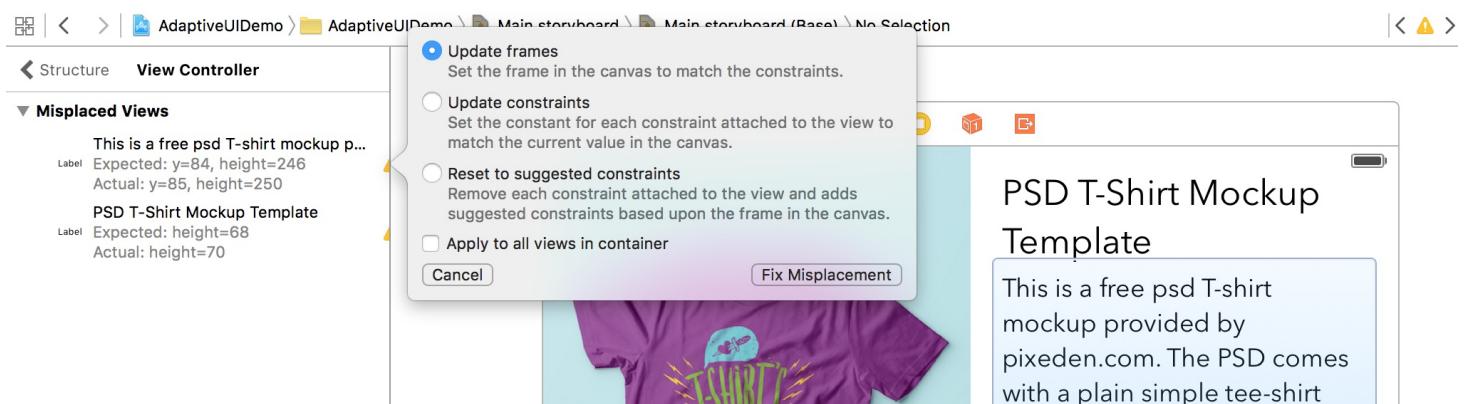
Select the title label and click the Pin button. Set the top constraint to `15`, left constraint to `22` and right constraint to `22`. Deselect the `Constrain to margins` option, and click the `Add 3`

`constraints` button. Control-drag from the title label to the description label, and select `Vertical Spacing` when prompted.

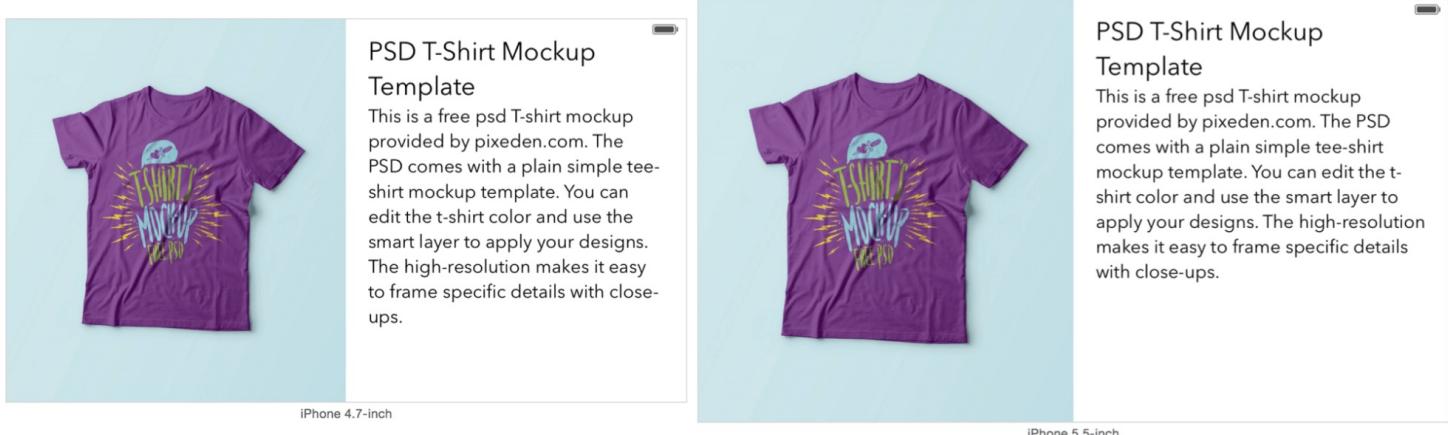


Select the description label and click the Pin button. For the left and right side, set the values to `22`. Again, make sure the `Constraint to margins` option is deselected, and click the `Add 2 Constraints` button to add the constraints.

Interface Builder should detect some layout issues. In Document Outline, click the yellow disclosure button to reveal the list of issues. Click each of the indicator icons and select `Update frame` to fix the issue.



Great! Now, go to the preview assistant to check out the design in landscape orientation. It looks pretty awesome, right?



However, there are still some minor issues for 3.5-inch and 4-inch displays, as the description is truncated. You will need to create a size class dependent font for the description label to fix the issue. I will leave it to you as an exercise.

Furthermore, try to run the app using the iPhone simulators. As you rotate the device, iOS renders a smooth transition when the view is changed from portrait to landscape.

Using Size Classes to Customize Constraints

Hopefully, you now understand how to use size classes to customize fonts and view design for a specific size class combination. On top of these customizations, you can use size classes to customize a particular constraint.

What if you want to change the view design of the iPhone 6 Plus (landscape) to this view but keep the design intact for other iPhones?



PSD T-Shirt Mockup Template

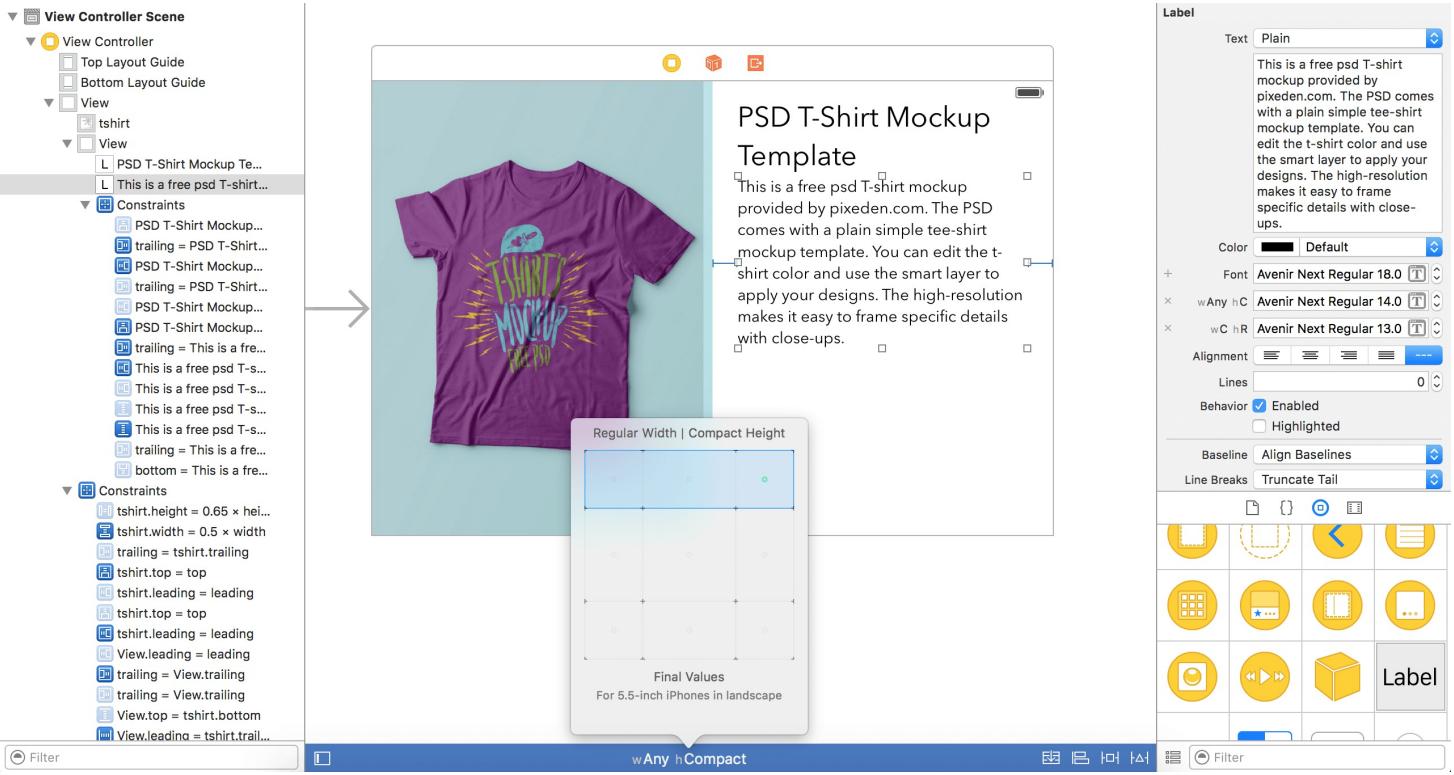
This is a free psd T-shirt mockup provided by pixeden.com. The PSD comes with a plain simple tee-shirt mockup template. You can edit the t-shirt color and use the smart layer to apply your designs. The high-resolution makes it easy to frame specific details with close-ups.

iPhone 5.5-inch

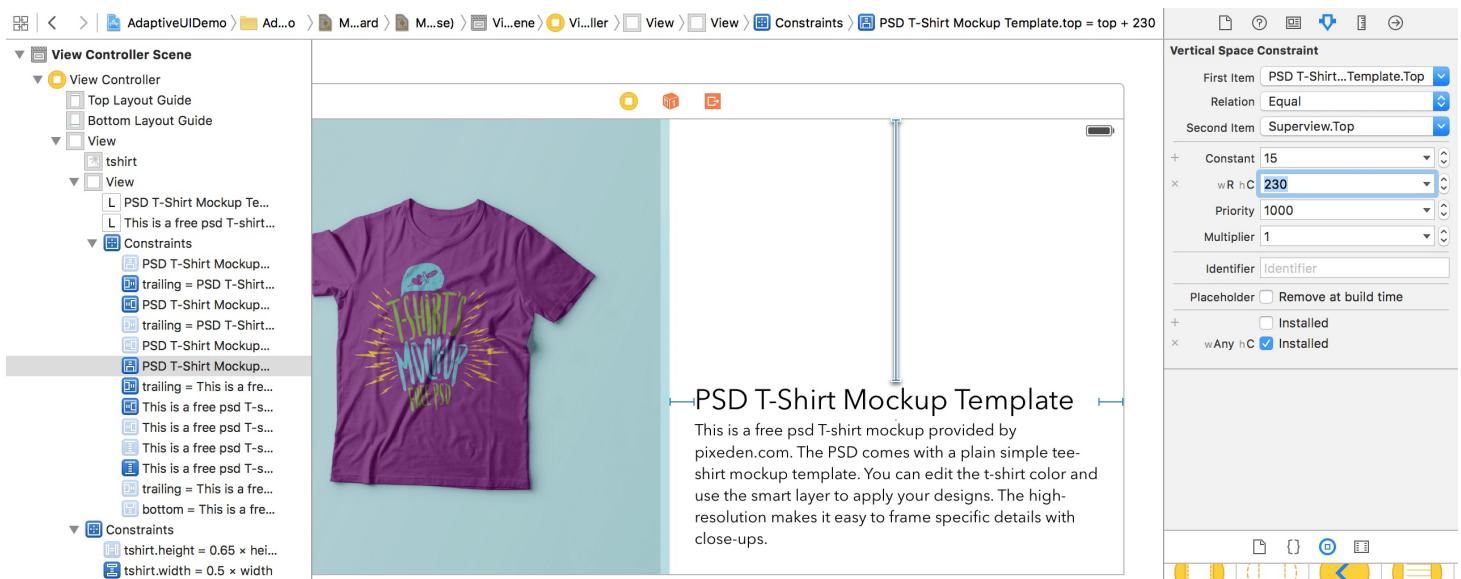
As you can see, the title and description have been moved to the lower-right part of the view. Obviously, we have to customize the top spacing constraints between the title label and its superview.

Let's see how it can be done.

The iPhone 6 Plus in landscape defaults to *regular* size class for horizontal and *compact* size class for vertical. Click the size class control in the Interface Builder. Select Regular width | Compact Height and confirm.



In Document Outline, select the Vertical Space constraint between the title label and its superview. Under Size Inspector, you should find a + button next to the Constant field. The value is currently set to 15 points. Since we want to increase the spacing for Regular | Compact size, click the + button and select “Regular Width | Compact Height (current)” to add a size class dependent constant. You should then see a new field named, `wR hc`. Change the field value to 230 to increase the spacing.



In case Interface Builder detects any layout issues, just click the issue indicator in Document

Outline and follow the suggestions to fix the issues. Look at the preview. The view of the 5.5 inch iPhone shows the new design, while the rest of the iPhone models use the original view design.

Summary

With iOS 9 and Xcode 7, Apple has provided developers with the tools to build adaptive layouts. In this chapter, I have covered the concept of size classes, showing you how to use them to create adaptive user interfaces.

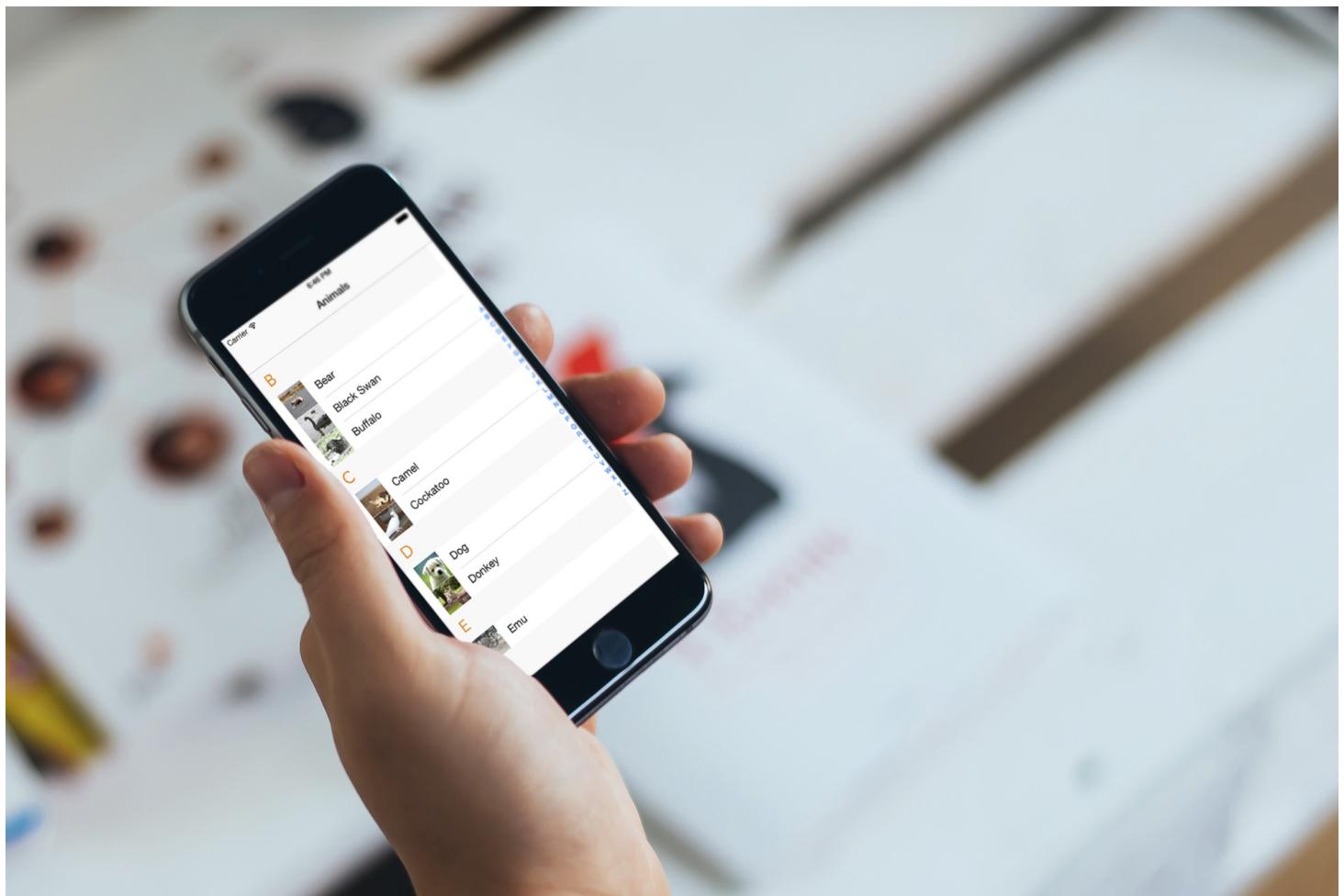
Adaptive layout is one of the most important concepts introduced since iOS 8. Gone are the days where developers had only a single device and screen size for which to design. If you are going to build your next app, make sure you grasp the concepts of size classes and auto layout, and make your app adapts to multiple screen sizes and orientations. The future of app design is more than likely going to be adaptive.

For reference, you can download the Xcode project from

<https://www.dropbox.com/s/j2dja7mck9rgfvd/AdaptiveUIDemo.zip?dl=0>.

Chapter 2

Adding Sections and Index list in UITableView



If you'd like to show a large number of records in UITableView, you'd best rethink the approach of how to display your data. As the number of rows grows, the table view becomes unwieldy. One way to improve the user experience is to organize the data into sections. By grouping related data together, you offer a better way for users to access it.

Furthermore, you can implement an index list in the table view. An indexed table view is more or less the same as the plain-styled table view. The only difference is that it includes an index on the right side of the table view. An indexed table is very common in iOS apps. The most

well-known example is the built-in Contacts app on the iPhone. By offering index scrolling, users have the ability to access a particular section of the table instantly without scrolling through each section.

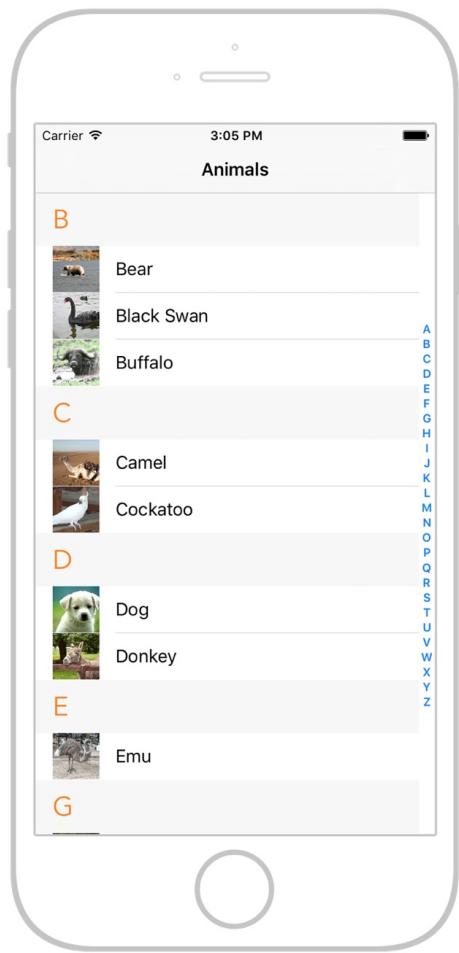
Let's see how we can add sections and an index list to a simple table app. If you have a basic understanding of the `UITableView` implementation, it's not too difficult to add sections and an index list. Basically you need to deal with these methods as defined in the `UITableViewDataSource` protocol:

- *numberOfSectionsInTableView* method – returns the total number of sections in the table view. Usually we set the number of sections to `1`. If you would like to have multiple sections, set this value to a number larger than `1`.
- *titleForHeaderInSection* method – returns the header titles for different sections. This method is optional if you do not prefer to assign titles to the section.
- *numberOfRowsInSection* method – returns the total number of rows in a specific section.
- *cellForRowAtIndexPath* method – this method shouldn't be new to you if you know how to display data in UITableView. It returns the table data for a particular section.
- *sectionIndexTitlesForTableView* method – returns the indexed titles that appear in the index list on the right side of the table view. For example, you can return an array of strings containing a value from `A` to `Z`.
- *sectionForSectionIndexTitle* method – returns the section index that the table view should jump to when a user taps a particular index.

There is no better way to explain the implementation than showing you an example. As usual, we will build a simple app, which should give you a better idea of an index list implementation.

A Brief Look at the Demo App

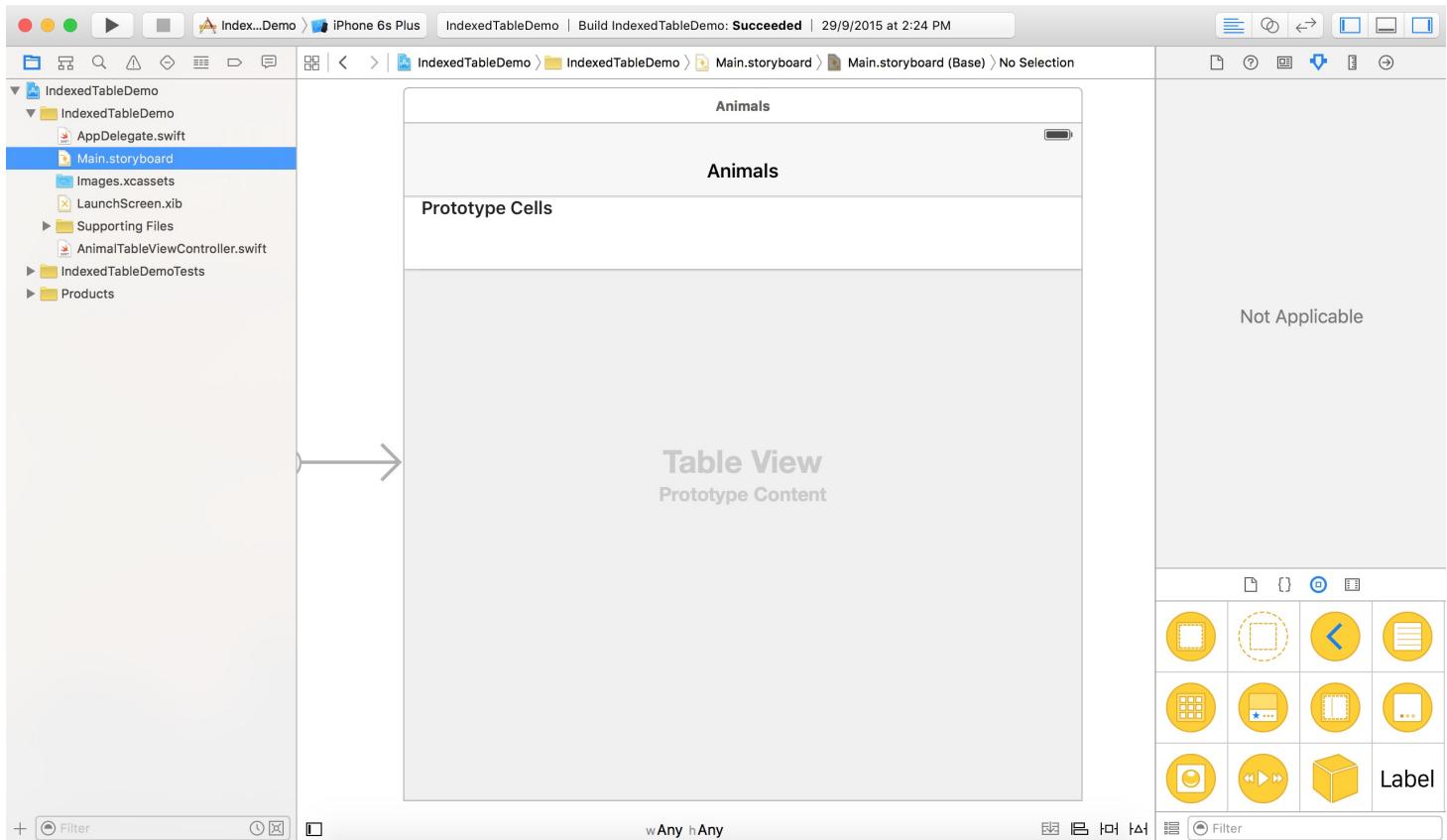
First, let's have a quick look at the demo app that we are going to build. It's a very simple app showing a list of animals in a standard table view. Instead of listing all the animals, the app groups the animals into different sections, and displays an index list for quick access. The screenshot below displays the final deliverable of the demo app.



Download the Xcode Project Template

The focus of this demo is on the implementation of sections and index list. Therefore, instead of building the Xcode project from scratch, you can download the project template from <https://www.dropbox.com/s/2whtwzodzw2xrq3/IndexedTableDemoTemplate.zip?dl=0> to start with.

The template already includes everything you need to start with. If you build the template, you'll have an app showing a list of animals in a table view (but without sections and index). Later, we will modify the app, group the data into sections, and add an index list to the table.



Displaying Sections in UITableView

Okay, let's get started. If you open the `IndexedTableDemo` project, the animal data is defined in an array:

```
let animals = ["Bear", "Black Swan", "Buffalo", "Camel", "Cockatoo", "Dog",
    "Donkey", "Emu", "Giraffe", "Greater Rhea", "Hippopotamus", "Horse", "Koala",
    "Lion", "Llama", "Manatus", "Meerkat", "Panda", "Peacock", "Pig", "Platypus",
    "Polar Bear", "Rhinoceros", "Seagull", "Tasmania Devil", "Whale", "Whale
    Shark", "Wombat"]
```

Well, we're going to organize the data into sections based on the first letter of the animal name. There are a lot of ways to do that. One way is to manually replace the `animals` array with a dictionary like I've shown below:

```
let animals: [String: [String]] = ["B" : ["Bear", "Black Swan", "Buffalo"],
    "C" : ["Camel", "Cockatoo"],
    "D" : ["Dog", "Donkey"],
    "E" : ["Emu"],
    "G" : ["Giraffe", "Greater Rhea"],
    "H" : ["Hippopotamus", "Horse"],
```

```
"K" : ["Koala"],  
"L" : ["Lion", "Llama"],  
"M" : ["Manatus", "Meerkat"],  
"P" : ["Panda", "Peacock", "Pig", "Platypus", "Polar Bear"],  
"R" : ["Rhinoceros"],  
"S" : ["Seagull"],  
"T" : ["Tasmania Devil"],  
"W" : ["Whale", "Whale Shark", "Wombat"]]
```

In the above code, we've turned the animals array into a dictionary. The first letter of the animal name is used as a key. The value that is associated with the corresponding key is an array of animal names.

We could manually create the dictionary, but wouldn't it be great if we could create the indexes from the `animals` array? Let's see how it can be done. First, declare two instance variables in the `AnimalTableViewController` class:

```
var animalsDict = [String: [String]]()  
var animalSectionTitles = [String]()
```

We initialize an empty dictionary for storing the animals and an empty array for storing the section titles of the table. The section title is the first letter of the animal name (e.g. B).

Because we want to generate a dictionary from the `animals` array, we need a helper method to handle the generation. Insert the following method in the `AnimalTableViewController` class:

```
func createAnimalDict() {  
    for animal in animals {  
        // Get the first letter of the animal name and build the dictionary  
        let animalKey =  
            animal.substringToIndex(animal.startIndex.advancedBy(1))  
        if var animalValues = animalsDict[animalKey] {  
            animalValues.append(animal)  
            animalsDict[animalKey] = animalValues  
        } else {  
            animalsDict[animalKey] = [animal]  
        }  
    }  
  
    // Get the section titles from the dictionary's keys and sort them in  
    // ascending order  
    animalSectionTitles = [String](animalsDict.keys)  
    animalSectionTitles = animalSectionTitles.sort({ $0 < $1 })  
}
```

In this method, we loop through all the items in the `animals` array. For each item, we initially extract the first letter of the animal's name. In Swift, the `substringToIndex` method of a string can return a new string containing the characters up to a given index. The index should be of the type `String.Index`. To obtain an index for a specific position, you have to ask the string itself for the `startIndex` and then use the global `advance()` function to iterate over all characters between the beginning of the string and the target position. In this case, the target position is `1`, as we are only interested in the first character.

As mentioned before, the first letter of the animal's name is used as a key of the dictionary. The value of the dictionary is an array of animals of that particular key. So once we got the key, we either create a new array of animals or append the item to an existing array. Here we show the values of `animalsDict` for the first four iterations:

- Iteration #1: `animalsDict["B"] = ["Bear"]`
- Iteration #2: `animalsDict["B"] = ["Bear", "Black Swan"]`
- Iteration #3: `animalsDict["B"] = ["Bear", "Black Swan", "Buffalo"]`
- Iteration #4: `animalsDict["C"] = ["Camel"]`

After `animalsDict` is completely generated, we can retrieve the section titles from the keys of the dictionary.

To retrieve the keys of a dictionary, you can simply call the `keys` method. However, the keys returned are unordered. Swift's standard library provides a function called `sort`, which returns a sorted array of values of a known type, based on the output of a sorting closure you provide.

The closure takes two arguments of the same type (in this example, it's the string) and returns a `Bool` value to state whether the first value should appear before or after the second value once the values are sorted. If the first value should appear before the second value, it should return true.

One way to write the sort closure is like this:

```
animalSectionTitles = animalSectionTitles.sort( { (s1:String, s2:String) ->
    Bool in
        return s1 < s2 }
```

```
)
```

You should be very familiar with the closure expression syntax. In the body of the closure, we compare the two string values. It returns `true` if the second value is greater than the first value. For instance, the value of `s1` is `B` and that of `s2` is `E`. Because B is smaller than E, the closure returns true, indicating that B should appear before E. In this case, we can sort the values in alphabetical order.

If you read the earlier code snippet carefully, you may wonder why I wrote the `sort` closure like this:

```
animalSectionTitles = animalSectionTitles.sort({ $0 < $1 })
```

It's a shorthand in Swift for writing inline closures. Here `$0` and `$1` refer to the first and second String arguments. If you use shorthand argument names, you can omit nearly everything of the closure including argument list and `in` keyword; you will just need to write the body of the closure.

In Swift 2, Apple introduces another sort function called `sortInPlace`. This new function is very similar to the `sort` function. Instead of returning you a sorted array, the `sortInPlace` function applies the sorting on the original array. You can replace the line of code with the one below:

```
animalSectionTitles.sortInPlace({ $0 < $1 })
```

With the helper method created, update the `viewDidLoad` method to call it up:

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

    // Generate the animal dictionary
    createAnimalDict()
}
```

Next, change the `numberOfSectionsInTableView` method and return the total number of sections:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    // Return the number of sections.
```

```
    return animalSectionTitles.count
}
```

To display a header title in each section, we need to implement the `titleForHeaderInSection` method. This method is called every time a new section is displayed. Based on the given section index, we simply return the corresponding section title.

```
override func tableView(tableView: UITableView, titleForHeaderInSection section: Int) -> String? {
    return animalSectionTitles[section]
}
```

It's very straightforward, right? Next, we have to tell the table view the number of rows in a particular section. Update the `numberOfRowsInSection` method in

`AnimalTableViewController.swift` like this:

```
override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    // Return the number of rows in the section.
    let animalKey = animalSectionTitles[section]
    if let animalValues = animalsDict[animalKey] {
        return animalValues.count
    }

    return 0
}
```

When the app starts to render the data in the table view, the `numberOfRowsInSection` method is called every time a new section is displayed. Based on the section index, we can get the section title and use it as a key to retrieve the animal names of that section, followed by returning the total number of animal names for that section.

Lastly, modify the `cellForRowIndexPath` method as follows:

```
override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
forIndexPath: indexPath)

    // Configure the cell...
    let animalKey = animalSectionTitles[indexPath.section]
    if let animalValues = animalsDict[animalKey] {
        cell.textLabel?.text = animalValues[indexPath.row]
```

```

    // Convert the animal name to lower case and
    // then replace all occurrences of a space with an underscore
    let imageFilename =
animalValues[indexPath.row].lowercaseString.stringByReplacingOccurrencesOfString
", withString: "_", options: [], range: nil)
    cell.imageView?.image = UIImage(named: imageFilename)
}

return cell
}

```

The `indexPath` argument contains the current row number, as well as, the current section index. So, based on the section index, we retrieve the section title (e.g. "B") and use it as the key to retrieve the animal names for that section. The rest of the code is very straightforward. We simply get the animal name and set it as the cell label. The `imageFilename` variable is computed by converting the animal name to lowercase letters, followed by replacing all occurrences of a space with an underscore.

Okay, you're ready to go! Hit the Run button and you should end up with an app with sections but without the index list.

Adding An Index List to UITableView

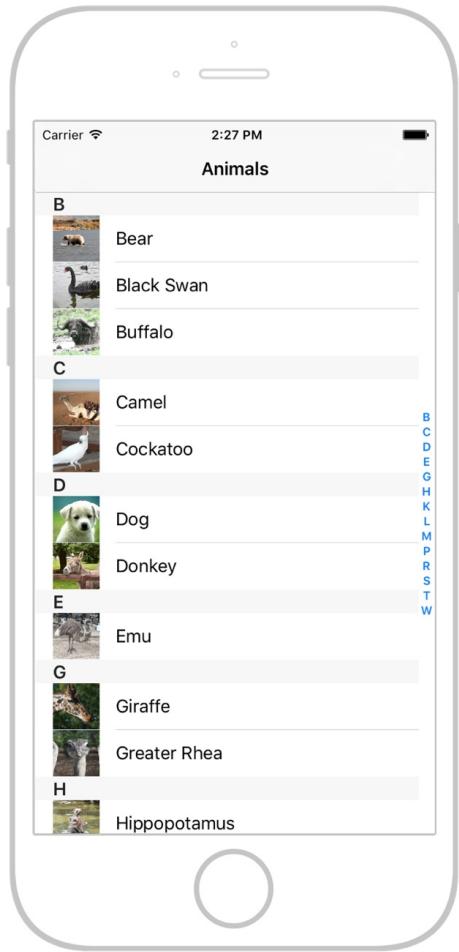
Cool, right? But how can you add an index list to the table view? Again it's easier than you thought and can be achieved with just a few lines of code. Simply add the `sectionIndexTitlesForTableView` method and return an array of section indexes. Here we will use the section titles as the indexes.

```

override func sectionIndexTitlesForTableView(tableView: UITableView) ->
[String]? {
    return animalSectionTitles
}

```

That's it! Compile and run the app again. You should find the index on the right side of the table. Interestingly, you do not need any implementation and the indexing already works! Try to tap any of the indexes and you'll be brought to a particular section of the table.



Adding An A-Z Index List

Looks like we've done everything. So why did we mention the `sectionForSectionIndexTitle` method at the very beginning?

Currently, the index list doesn't contain the entire alphabet. It just shows those letters that are defined as the keys of the `animals` dictionary. Sometimes, you may want to display A-Z in the index list. Let's declare a new variable named `animalIndexTitles` in

`AnimalViewController.swift` :

```
let animalIndexTitles = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K",
    "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
```

Next, change the `sectionIndexTitlesForTableView` method and return the `animalIndexTitles` array instead of the `animalSectionTitles` array.

```
override func sectionIndexTitlesForTableView(tableView: UITableView) ->
```

```
[AnyObject]! {
    return animalIndexTitles
}
```

Now, compile and run the app again. Cool! The app displays the index from A to Z.

But wait a minute... It doesn't work properly! If you try tapping the index "C," the app jumps to the "D" section. And if you tap the index "G," it directs you to the "K" section. Below shows the mapping between the old and new indexes.

Old Index	B	C	D	E	G	H	K	L	M	P	R	S	T	W
New Index	A	B	C	D	E	F	G	H	I	J	K	L	M	N

Well, as you may notice, the number of indexes is greater than the number of sections, and the `UITableView` object doesn't know how to handle the indexing. It's your responsibility to implement the `sectionForSectionIndexTitle` method and explicitly tell the table view the section number when a particular index is tapped. Add the following new method:

```
override func tableView(tableView: UITableView, sectionForSectionIndexTitle
title: String, atIndex index: Int) -> Int {

    guard let index = animalSectionTitles.indexOf(title) else {
        return -1
    }

    return index
}
```

Based on the selected index name (i.e. title), we locate the correct section index of `animalSectionTitles`. In Swift, you use the method called `indexOf` to find the index of a particular item in the array. If it's not found, we return `-1`. Otherwise, we return the index of the item. Compile and run the app again. The index list should now work!

Note: The old `find` function is not supported any more with Swift 2.0!

Customizing Section Headers

You can easily customize the section headers by overriding some of the methods defined in the

`UITableView` class and the `UITableViewDelegate` protocol. In this demo, we'll make two simple changes:

- Alter the height of the section header
- Change the font of the section header

To alter the height of the section header, you can simply override the `heightForHeaderInSection` method and return the preferred height:

```
override func tableView(tableView: UITableView, heightForHeaderInSection section: Int) -> CGFloat {
    return 50
}
```

Before the section header view is displayed, the `willDisplayHeaderView` method will be called. The method includes an argument named `view`. This view object can be a custom header view or a standard one. In our demo, we just use the standard header view, which is the `UITableViewHeaderFooterView` object. Once you have the header view, you can alter the text color and font accordingly.

```
override func tableView(tableView: UITableView, willDisplayHeaderView view: UIView, forSection section: Int) {
    let headerView = view as! UITableViewHeaderFooterView
    headerView.textLabel?.textColor = UIColor.orangeColor()
    headerView.textLabel?.font = UIFont(name: "Avenir", size: 25.0)
}
```

Run the app again. The header view should be updated with your preferred font and color.

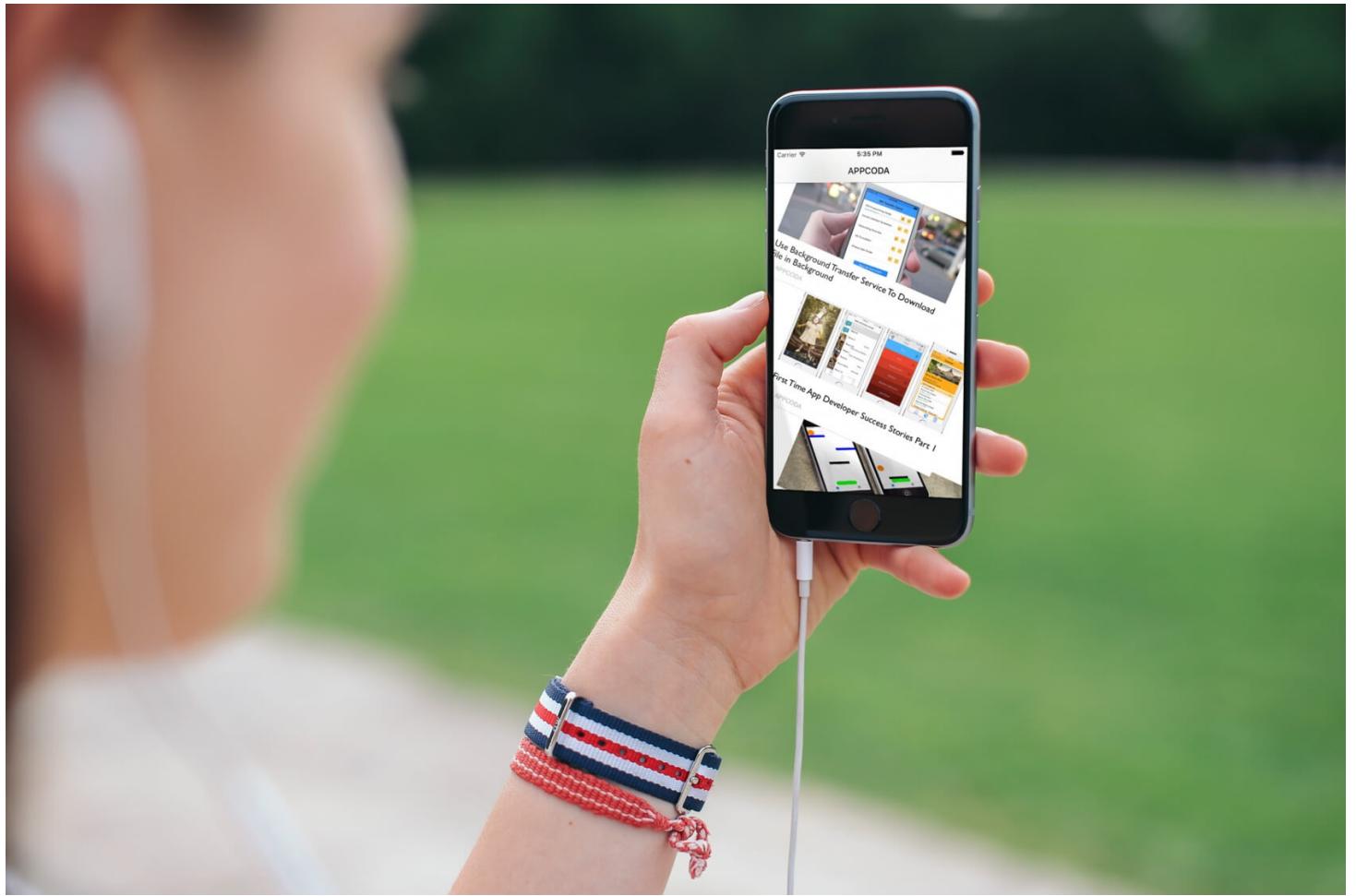
Summary

When you need to display a large number of records, it is simple and effective to organize the data into sections and provide an index list for easy access. In this chapter, we've walked you through the implementation of an indexed table. By now, I believe you should know how to add sections and an index list to your table view.

For your reference, you can download the complete Xcode project from
<https://www.dropbox.com/s/kpdoq1m5ccsaup7/IndexedTableDemo.zip?dl=0>.

Chapter 3

Animating Table View Cells



When you read this chapter, I assume you already knew how to use `UITableView` to present data. If not, go back and read the [Beginning iOS 9 Programming with Swift book](#).

The `UITableView` class provides a powerful way to present information in table form, and it is one of the most commonly used components in iOS apps. Whether you're building a productivity app, to-do app, or social app, you would make use of table views in one form or another. The default implementation of `UITableView` is preliminary and only suitable for basic apps. To differentiate one's app from the rest, you usually provide customizations for the table views and table cells in order to make the app stand out. In this chapter, we'll show you a powerful technique to liven up your app by adding subtle animation.

The [Google+ app](#) is a great example of table view animation. If you've used the app, every table row (or card) is animated as you scroll through the table. The card seems to slide in from the side when it first appears. This subtle animation would greatly enhance the user experience of your app.

It is very easy to animate a table view cell. Again, to demonstrate how the animation is done, we'll tweak an existing table-based app and add a subtle animation.

To start with, first download the project template from

<https://www.dropbox.com/s/qnxqfwppra62mg8/TableCellAnimationTemplate.zip?dl=0>.

After downloading, compile the app and make sure you can run it properly. It's just a very simple app displaying a list of articles.



Creating a Simple Fade-in Animation for Table View Cells

Let's start by tweaking the table-based app with a simple fade-in effect. So how can we add this

subtle animation when the table row appears? If you look into the documentation of the `UITableViewDelegate` protocol, you should find a method called `tableView(_:willDisplayCell:forRowAtIndexPath:)`:

```
optional func tableView(_ tableView: UITableView, willDisplayCell cell:  
UITableViewCell, forRowAtIndexPath indexPath: IndexPath)
```

The method will be called right before a row is drawn. By implementing the method, you can customize the cell object and add your own animation before the cell is displayed. Here is what you need to create the fade-in effect. Insert the code snippet in

`ArticleTableViewController.swift` :

```
override func tableView(tableView: UITableView, willDisplayCell cell:  
UITableViewCell, forRowAtIndexPath indexPath: IndexPath) {  
  
    // Define the initial state (Before the animation)  
    cell.alpha = 0  
  
    // Define the final state (After the animation)  
    UIView.animateWithDuration(1.0, animations: { cell.alpha = 1 })  
}
```

Core Animation provides iOS developers with an easy way to create animation. All you need to do is define the initial and final state of the visual element. Core Animation will then figure out the required animation between these two states.

In the above code, we first set the initial alpha value of the cell to `0`, which represents total transparency. Then we begin the animation; set the duration to `1` second and define the final state of the cell, which is completely opaque. This will automatically create a fade-in effect when the table cell appears.

You can now compile and run the app. Scroll through the table view and enjoy the fade-in animation.

Creating a Rotation Effect Using CATransform3D

Easy, right? With a few lines of code, your app looks a bit different than a standard table-based app. The `tableView(_:willDisplayCell:forRowAtIndexPath:)` method is the key to table view cell animation. You can implement whichever type of animation in the method. The fade-in

animation is very simple. Now let's try to implement another animation using `CATransform3D`. Don't worry, you just need a few lines of code.

To add a rotation effect to the table cell, update the method like this:

```
override func tableView(tableView: UITableView, willDisplayCell cell:  
UITableViewCell, forIndexPath indexPath: NSIndexPath) {  
  
    // Define the initial state (Before the animation)  
    let rotationAngleInRadians = 90.0 * CGFloat(M_PI/180.0)  
    let rotationTransform = CATransform3DMakeRotation(rotationAngleInRadians,  
0, 0, 1)  
    cell.layer.transform = rotationTransform  
  
    // Define the final state (After the animation)  
    UIView.animateWithDuration(1.0, animations: { cell.layer.transform =  
CATransform3DIdentity })  
}
```

Same as before, we define the initial and final state of the transformation. The general idea is that we first rotate the cell by 90 degrees clockwise and then bring it back to the normal orientation which is the final state.

Okay, but how can we rotate a table cell by 90 degrees clockwise?

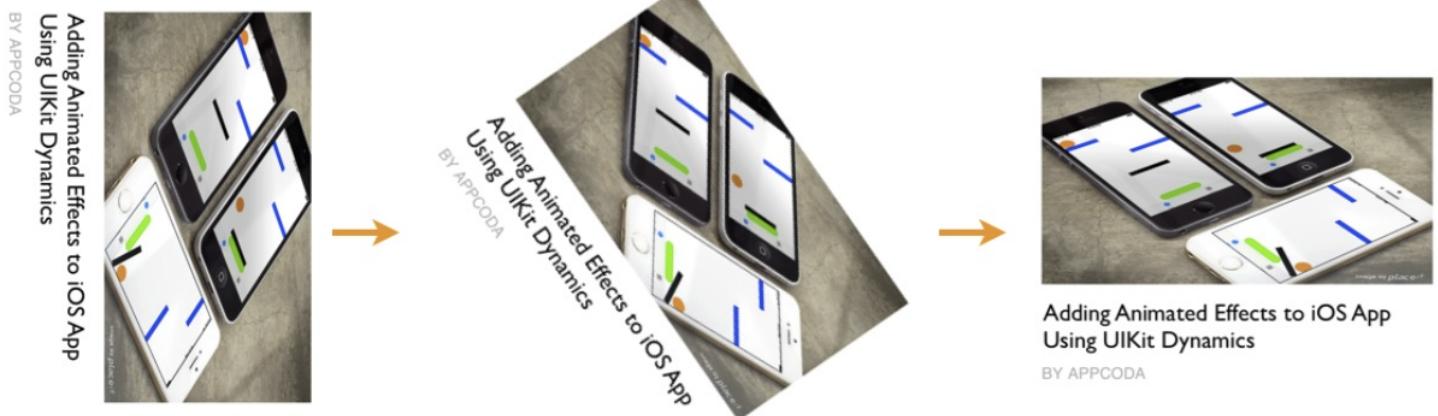
The key is to use the `CATransform3DMakeRotation` function to create the rotation transform. The function takes four parameters:

- Angle in radians - this is the angle of rotation. As the angle is in radian, we first need to convert the degrees into radians.
- X axis - this is the axis that goes from the left of the screen to the right of the screen.
- Y axis - this is the axis that goes from the top of the screen to the bottom of the screen.
- Z axis - this is the axis that points directly out of the screen.

Since the rotation is around the Z axis, we set the value of this parameter to `1`, while leaving the value of the X axis and Y axis at `0`. Once we create the transform, it is assigned to the cell's layer.

Next, we start the animation with the duration of `1` second. The final state of the cell is set to `CATransform3DIdentity`, which will reset the cell to the original position.

Okay, hit Run to test the app!



Quick Tip: You may wonder what `CATransform3D` is. It is actually a structure representing a matrix. Performing transformation in 3D space such as rotation, involves some matrices calculation. I'll not go into the details of matrices calculation. If you want to learn more, you can check out <http://www.matrix44.net/cms/notes/opengl-3d-graphics/basic-3d-math-matrices>.

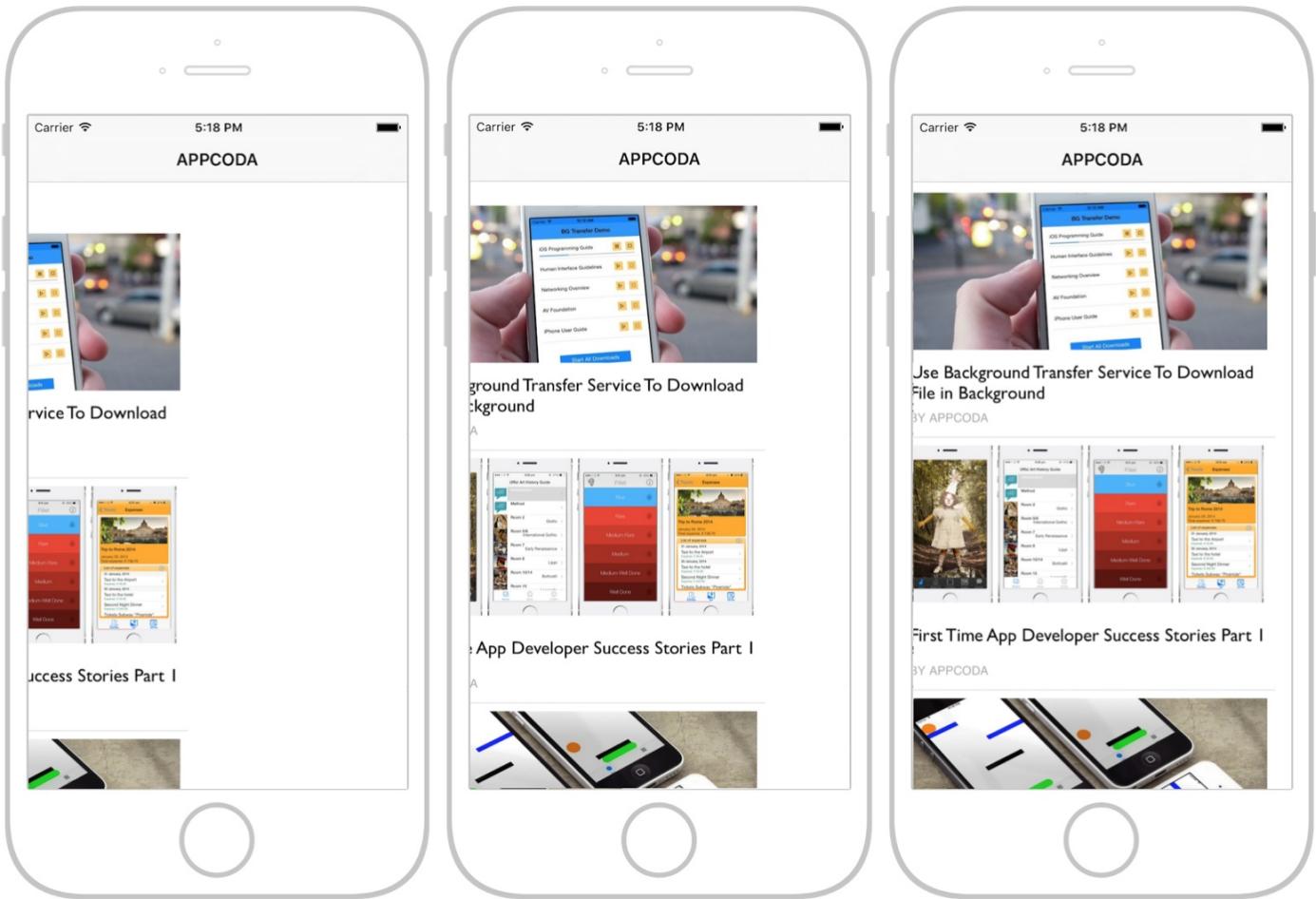
Creating a Fly-in Effect using `CATransform3DTranslate`

Does the rotation effect look cool? You can further tweak the animation to make it even better. Try to change the `tableView(_:willDisplayCell:forRowAtIndexPath:)` method and replace the initialization of `rotationTransform` with the following line of code:

```
let rotationTransform = CATransform3DTranslate(CATransform3DIdentity, -500,  
100, 0)
```

The line of code simply translates or shifts the position of the cell. It indicates the cell is shifted to the left (negative value) by `500` points and down (positive value) by `100` points. There is no change in the Z axis.

Now you're ready to test the app again. Hit the Run button and play around with the fly-in effect.



Your Exercise

For now, the cell animation is shown every time you scroll through the table, whether you're scrolling down or up the table view. Though the animation is nice, your user will find it annoying if the animation is displayed too frequently. You may want to display the animation only when the cell first appears. Try to modify the existing project and add that restriction.

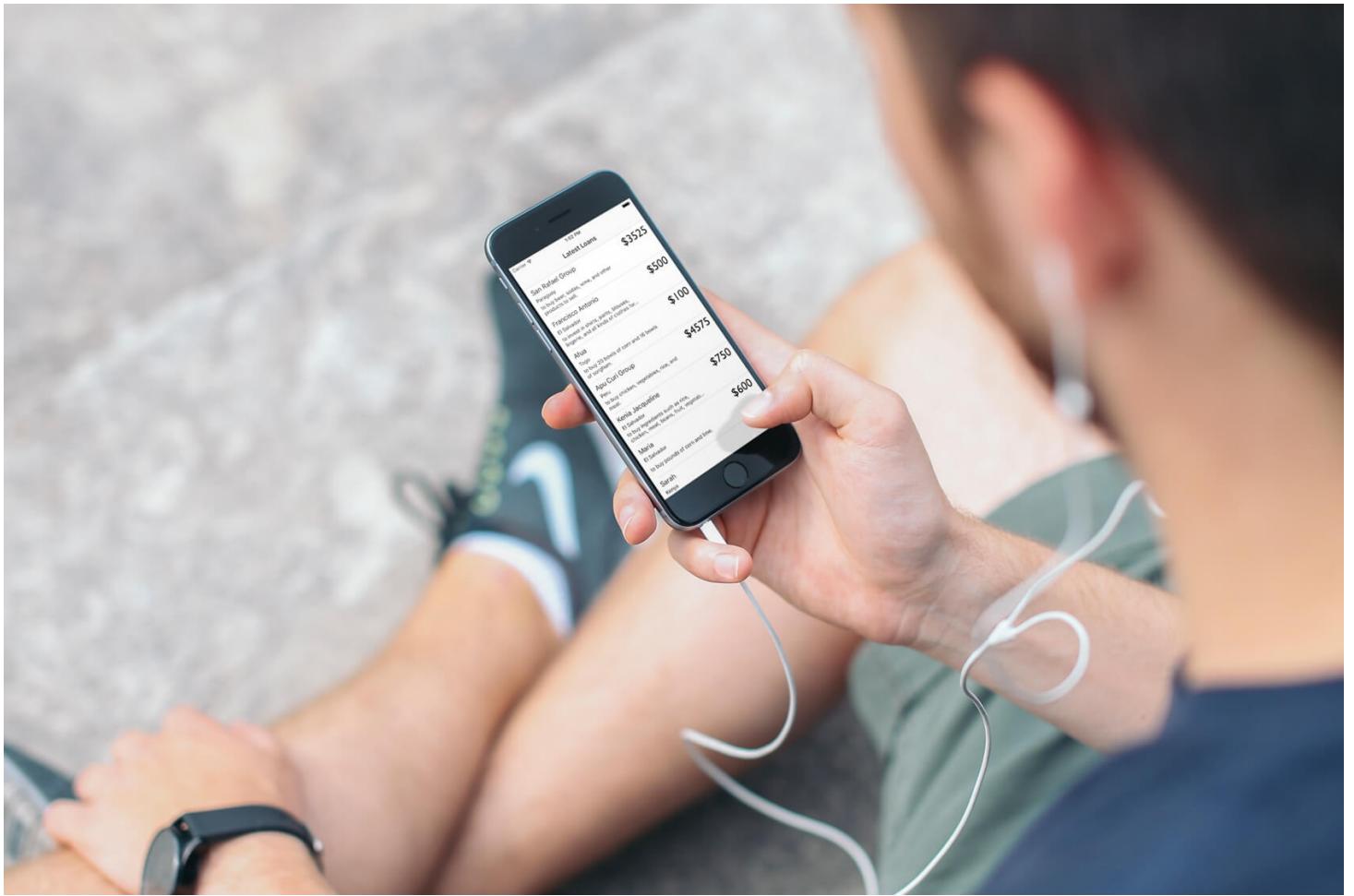
Summary

In this chapter, I just showed you the basics of table cell animation. Try to change the values of the transform and see what effects you get.

For reference, you can download the complete Xcode project from <https://www.dropbox.com/s/98s9wiwe38l81sc/TableCellAnimation.zip?dl=0>. The solution of the exercise is included in the project.

Chapter 4

Working with JSON



First, what's JSON? JSON (short for JavaScript Object Notation) is a text-based, lightweight, and easy way for storing and exchanging data. It's commonly used for representing structural data and data interchange in client-server applications, serving as an alternative to XML. A lot of the web services we use everyday have JSON-based APIs. Most of the iOS apps, including Twitter, Facebook, and Flickr send data to their backend web services in JSON format. As an example, here is a JSON representation of a sample Movie object:

```
{  
  "title": "The Amazing Spider-man",  
  "release_date": "03/07/2012",  
  "director": "Marc Webb",
```

```

"cast": [
  {
    "name": "Andrew Garfield",
    "character": "Peter Parker"
  },
  {
    "name": "Emma Stone",
    "character": "Gwen Stacy"
  },
  {
    "name": "Rhys Ifans",
    "character": "Dr. Curt Connors"
  }
]
}

```

As you can see, JSON formatted data is more human-readable and easier to parse than XML. I'll not go into the details of JSON. This is not the purpose of this chapter. If you want to learn more about the technology, I recommend you to check out the [JSON Guide at http://www.json.org/](http://www.json.org/).

Since the release of iOS 5, the iOS SDK has already made it easy for developers to fetch and parse JSON data. It comes with a handy class called `NSJSONSerialization`, which can automatically convert JSON formatted data to objects. Later in this chapter, I will show you how to use the API to parse some sample JSON formatted data, returned by a web service. Once you understand how it works, it is fairly easy to build an app by integrating with other free/paid web services.

Demo App

As usual, we'll create a demo app. Let's call it *KivaLoan*. The reason why we name the app *KivaLoan* is that we will utilize a JSON-based API provided by Kiva.org. If you haven't heard of Kiva, it is a non-profit organization with a mission to connect people through lending to alleviate poverty. It lets individuals lend as little as \$25 to help create opportunities around the world. Kiva provides free web-based APIs for developers to access their data. For our demo app, we'll call up the following Kiva API to retrieve the most recent fundraising loans and display them in a table view:

<https://api.kivaws.org/v1/loans/newest.json>

Quick note: Starting from iOS 9, Apple introduced a new feature called App Transport Security with the aim to improve the security of connections between an app and web services. By default, all outgoing connections should ride on HTTPS. Otherwise, your app will not be allowed to connect to the web service.

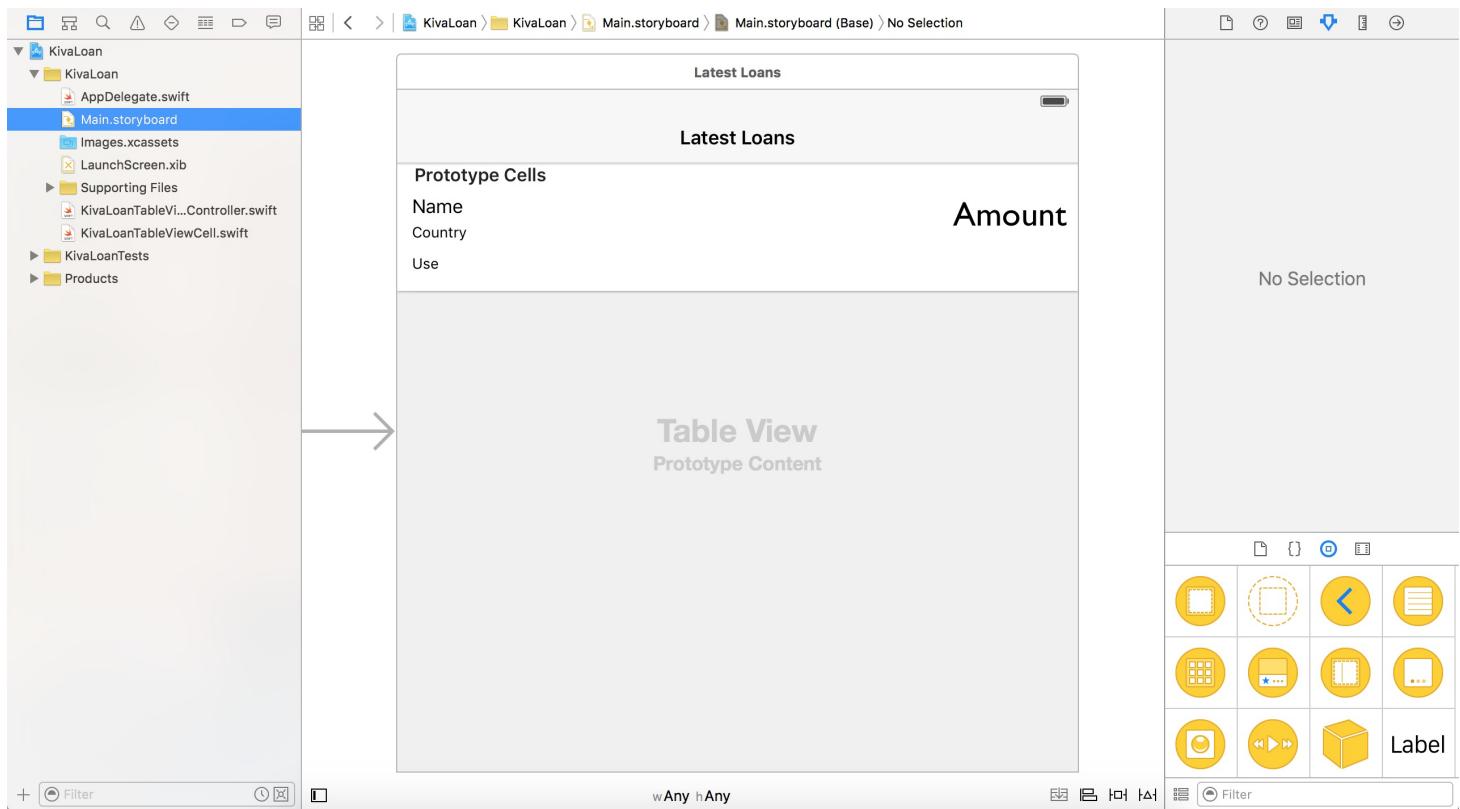
The returned data of the above API is in JSON format. Here is a sample result:

```
loans: (
    {
        activity = Retail;
        "basket_amount" = 0;
        "bonus_credit_eligibility" = 0;
        "borrower_count" = 1;
        description = {
            languages = (
                fr,
                en
            );
        };
        "funded_amount" = 0;
        id = 734117;
        image = {
            id = 1641389;
            "template_id" = 1;
        };
        "lender_count" = 0;
        "loan_amount" = 750;
        location = {
            country = Senegal;
            "country_code" = SN;
            geo = {
                level = country;
                pairs = "14 -14";
                type = point;
            };
        };
        name = "Mar\u00e8me";
        "partner_id" = 108;
        "planned_expiration_date" = "2014-08-05T09:20:02Z";
        "posted_date" = "2014-07-06T09:20:02Z";
        sector = Retail;
        status = fundraising;
        use = "to buy fabric to resell";
    },
    ...
)

```

You will learn how to use the `NSJSONSerialization` class to convert the JSON formatted data into objects. It's unbelievably simple. You'll see what I mean in a while.

To keep you focused on learning the JSON implementation, you can first download the project template from <https://www.dropbox.com/s/qud9sozfcya3ji2/KivaLoanTemplate.zip?dl=0>. I have already created the skeleton of the app for you. It is a simple table-based app that displays a list of loans provided by Kiva.org. The project template includes a pre-built storyboard and custom classes for the table view controller and prototype cell. If you run the template, it should result in an empty table app.



Creating JSON Data Model

We will first create a class to model a loan. It's not required for loading JSON but a best practice to create a separate class (or structure) for storing the data model. The `Loan` class represents the loan information in the KivaLoan app and is used to store the loan information returned by Kiva.org. To keep things simple, we won't use all the returned data of a loan. Instead, the app will just display the following fields of a loan:

- Name of the loan applicant

```
name = "Mar\u00e8me";
```

- Country of the loan applicant

```
location = {
    country = Senegal;
    "country_code" = SN;
    geo = {
        level = country;
        pairs = "14 -14";
        type = point;
    };
};
```

- How the loan will be used

```
use = "to buy fabric to resell";
```

- Amount

```
"loan_amount" = 750;
```

These fields are good enough for filling up the labels in the table view. Now create a new class file using the Swift File template. Name it `Loan.swift` and declare the `Loan` class like this:

```
class Loan {

    var name:String = ""
    var country:String = ""
    var use:String = ""
    var amount:Int = 0

}
```

JSON supports a few basic data types including number, String, Boolean, Array, and Objects (an associated array with key and value pairs).

For the loan fields, the loan amount is stored as a numeric value in the JSON-formatted data. This is why we declared the `amount` property with the type `Int`. For the rest of the fields, they are declared with the type `String`.

Fetching Loans with the Kiva API

As I mentioned earlier, the Kiva API is free to use. No registration is required. You may point your browser to the following URL and you'll get the latest fundraising loans in JSON format.

```
https://api.kivaws.org/v1/loans/newest.json
```

Okay, let's see how we can call up the Kiva API and parse the returned data. First, open `KivaLoanTableViewController.swift` and declare two variables at the very beginning:

```
let kivaLoadURL = "https://api.kivaws.org/v1/loans/newest.json"
var loans = [Loan]()
```

We just defined the URL of the Kiva API, and declare the `loans` variable for storing an array of `Loan` objects. Next, insert the following methods in the same file:

```
func getLatestLoans() {
    let request = NSURLRequest(URL: NSURL(string: kivaLoadURL)!)
    let urlSession = NSURLSession.sharedSession()
    let task = urlSession.dataTaskWithRequest(request, completionHandler: {
        (data, response, error) -> Void in

        if let error = error {
            print(error)
            return
        }

        // Parse JSON data
        if let data = data {
            self.loans = self.parseJsonData(data)

            // Reload table view
            NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
                self.tableView.reloadData()
            })
        }
    })

    task.resume()
}

func parseJsonData(data: NSData) -> [Loan] {
```

```

var loans = [Loan]()

do {
    let jsonResult = try NSJSONSerialization.JSONObjectWithData(data,
options: NSJSONReadingOptions.MutableContainers) as? NSDictionary

    // Parse JSON data
    let jsonLoans = jsonResult?["loans"] as! [AnyObject]
    for jsonLoan in jsonLoans {
        let loan = Loan()
        loan.name = jsonLoan["name"] as! String
        loan.amount = jsonLoan["loan_amount"] as! Int
        loan.use = jsonLoan["use"] as! String
        let location = jsonLoan["location"] as! [String:AnyObject]
        loan.country = location["country"] as! String
        loans.append(loan)
    }

} catch {
    print(error)
}

return loans
}

```

These two methods form the core part of the app. Both methods work collaboratively to call the Kiva API, retrieve the latest loans in JSON format and translate the JSON-formatted data into an array of `Loan` objects. Let's go through them in detail.

In the `getLatestLoans` method, we first create an instance of `NSURLSession` with the Kiva API. The `NSURLSession` class was introduced in iOS 7 as a successor to `NSURLConnection`, which has been around for several years. `NSURLSession` provides more features, flexibility and power when dealing with online content over HTTP. One improvement of `NSURLSession` is session tasks, which handle the loading of data, as well as uploading and downloading files and data fetching from servers (e.g. JSON data fetching). With sessions, you can schedule three types of tasks: data tasks (`NSURLSessionDataTask`) for retrieving data to memory, download tasks (`NSURLSessionDownloadTask`) for downloading a file to disk, and upload tasks (`NSURLSessionUploadTask`) for uploading a file from disk. Here we use the data task to retrieve contents from Kiva.org. To add a data task to the session, we call the `dataTaskWithURL` method with the specified URL of Kiva. To initiate the data task, you call the `resume` method (i.e. `task.resume()`). Like most networking APIs, the `NSURLSession` API is asynchronous. Once the

request completes, it returns the data by calling your completion handler closure.

In the completion handler, immediately after the data is returned, we check for an error and invoke the `parseJsonData` method. The data returned is in JSON format. We create a helper method called `parseJsonData` for converting the given JSON-formatted data into an array of `Loan` objects. The Foundation framework provides the `NSJSONSerialization` class, which is capable of converting JSON to Foundation objects and converting Foundation objects to JSON. In the code snippet, we call the `JSONObjectWithData` method with the given JSON data to perform the conversion.

When converting JSON formatted data to objects, the top-level item is usually converted to a Dictionary or an Array. In this case, the top level of the returned data of the Kiva API is converted to a dictionary. You can access the array of loans using the key `loans`. How do you know what key to use? You can either refer to the API documentation or test the JSON data using a JSON browser (e.g. <http://jsonviewer.stack.hu>). If you've loaded the Kiva API into the JSON browser, here is an excerpt of the result:

```
{
  "paging": {
    "page": 1,
    "total": 5297,
    "page_size": 20,
    "pages": 265
  },
  "loans": [
    {
      "id": 794429,
      "name": "Joel",
      "description": {
        "languages": [
          "es",
          "en"
        ]
      },
      "status": "fundraising",
      "funded_amount": 0,
      "basket_amount": 0,
      "image": {
        "id": 1729143,
        "template_id": 1
      },
      "activity": "Home Appliances",
      "sector": "Personal Use",
    }
  ]
}
```

```

"use": "To buy home appliances.",
"location": {
    "country_code": "PE",
    "country": "Peru",
    "town": "Ica",
    "geo": {
        "level": "country",
        "pairs": "-10 -76",
        "type": "point"
    }
},
"partner_id": 139,
"posted_date": "2014-11-20T08:50:02Z",
"planned_expiration_date": "2015-01-04T08:50:02Z",
"loan_amount": 400,
"borrower_count": 1,
"lender_count": 0,
"bonus_credit_eligibility": true,
"tags": [
]
},
{
    "id": 797222,
    "name": "Lucy",
    "description": {
        "languages": [
            "en"
        ]
    },
    "status": "fundraising",
    "funded_amount": 0,
    "basket_amount": 0,
    "image": {
        "id": 1732818,
        "template_id": 1
    },
    "activity": "Farm Supplies",
    "sector": "Agriculture",
    "use": "To purchase a biogas system for clean cooking",
    "location": {
        "country_code": "KE",
        "country": "Kenya",
        "town": "Gatitu",
        "geo": {
            "level": "country",
            "pairs": "1 38",
            "type": "point"
        }
    }
}

```

```

},
"partner_id": 436,
"posted_date": "2014-11-20T08:50:02Z",
"planned_expiration_date": "2015-01-04T08:50:02Z",
"loan_amount": 800,
"borrower_count": 1,
"lender_count": 0,
"bonus_credit_eligibility": false,
"tags": [
]
},
...

```

As you can see from the above code, `paging` and `loans` are two of the top-level items. Once the `NSJSONSerialization` class converts the JSON data, the result (i.e. `jsonResult`) is returned as a Dictionary with the top-level items as keys. This is why we can use the key `loans` to access the array of loans. Here is the line of code for your reference:

```
let jsonLoans = jsonResult?["loans"] as! [AnyObject]
```

With the array of loans (i.e. `jsonLoans`) returned, we loop through the array. Each of the array items (i.e. `jsonLoan`) is converted into a dictionary. In the loop, we extract the loan data from each of the dictionaries and save them in a `Loan` object. Again, you can find the keys (highlighted in yellow) by studying the JSON result. The value of a particular result is stored as `AnyObject`. `AnyObject` is used because a JSON value could be a String, Double, Boolean, Array, Dictionary or null. This is why you have to downcast the value to a specific type such as `String` and `Int`. Lastly, we put the `loan` object into the `loans` array, which is the return value of the method.

```

for jsonLoan in jsonLoans {
    let loan = Loan()
    loan.name = jsonLoan["name"] as! String
    loan.amount = jsonLoan["loan_amount"] as! Int
    loan.use = jsonLoan["use"] as! String
    let location = jsonLoan["location"] as! [String: AnyObject]
    loan.country = location["country"] as! String
    loans.append(loan)
}

```

After the JSON data is parsed and the array of loans is returned, we call the `reloadData`

method to reload the table. You may wonder why we need to call

`NSOperationQueue.mainQueue().addOperationWithBlock` and execute the data reload in the main thread. The block of code in the completion handler of the data task is executed in a background thread. If you just call the `reloadData` method in a background thread, the data reload will not happen immediately. To ensure a responsive GUI update, this operation should be performed in the main thread. This is why we call the

`NSOperationQueue.mainQueue().addOperationWithBlock` method and request to run the `reloadData` method in the main queue.

```
NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
    self.tableView.reloadData()
})
```

Quick note: You can also use `dispatch_async` function to execute a block of code in the main thread. But according to Apple, it is recommended to use `NSOperationQueue` over `dispatch_async`. As a general rule, Apple recommends using the highest-level APIs rather than dropping down to the low-level ones.

Displaying Loans in A Table View

With the loans array in place, the last thing we need to do is to display the data in the table view. Update the following methods in `KivaLoanTableViewController.swift`:

```
override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
    // Return the number of sections.
    return 1
}

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

override func tableView(tableView: UITableView, cellForRowAt indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath) as! KivaLoanTableViewCell

    // Configure the cell...
    cell.nameLabel.text = loans[indexPath.row].name
    cell.countryLabel.text = loans[indexPath.row].country
    cell.useLabel.text = loans[indexPath.row].use
```

```
cell.amountLabel.text = "$\$(loans[indexPath.row].amount)"  
  
    return cell  
}
```

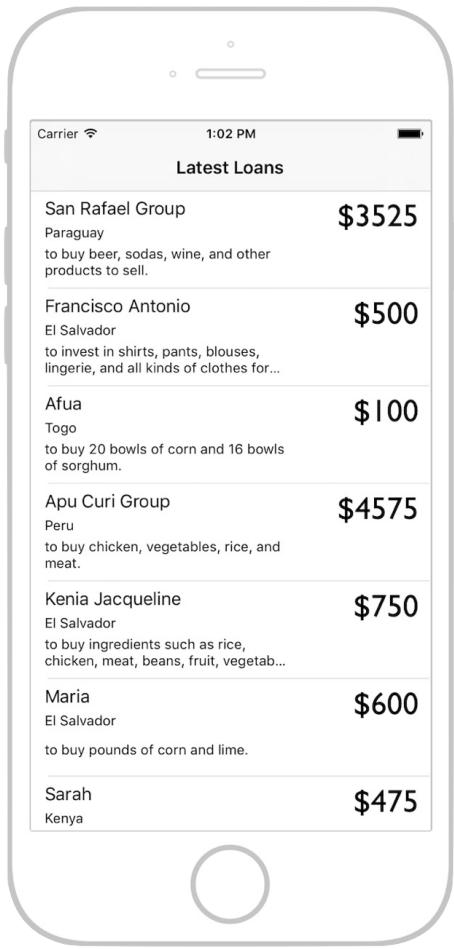
The above code is pretty straightforward if you are familiar with the implementation of `UITableView`. In the `cellForRowAtIndexPath:` method, we retrieve the loan information from the loans array and populate them in the custom table cell. One thing to take note of is the code below:

```
"$\$(loans[indexPath.row].amount)"
```

Sometimes you may want create a string by adding both string (e.g. \$) and integer (e.g. `loans[indexPath.row].amount`) together. Swift provides a powerful way to create these kinds of strings, known as string interpolation. You can make use of it by using the above syntax.

Lastly, insert the following line of code in the `viewDidLoad` method to start fetching the loan data:

```
getLatestLoans()
```



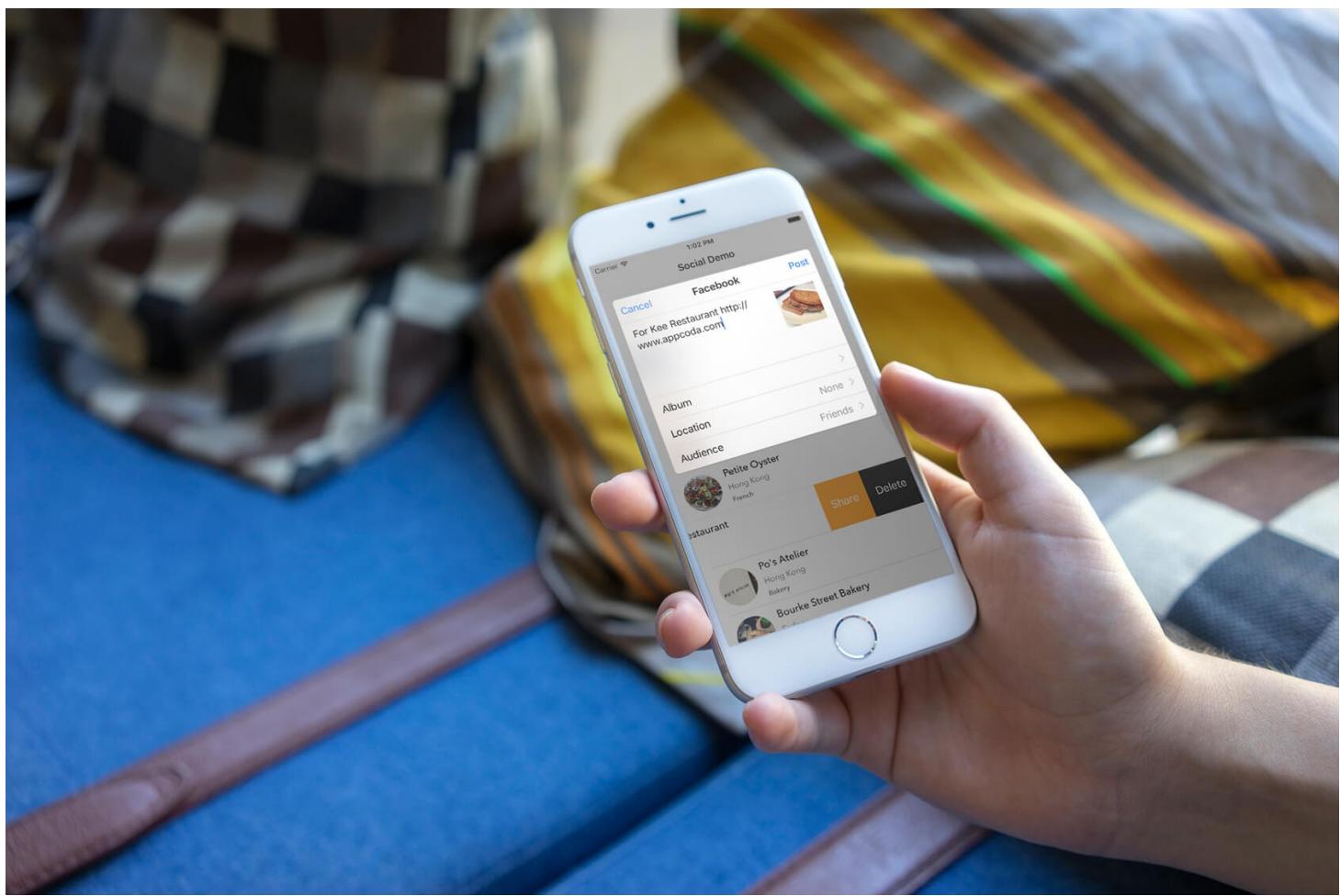
Compile and Run

Now it's time to test the app. Compile and run it in the simulator. Once launched, the app will pull the latest loans from Kiva.org and display them in the table view.

For your reference, you can download the complete Xcode project from <https://www.dropbox.com/s/ze9f3n2q3tzu341/KivaLoan.zip?dl=0>.

Chapter 5

How to Integrate Twitter and Facebook Sharing



With the advent of social networks, you may want to provide social network sharing in your apps. This is one of the many ways to increase user engagement. In the past, developers have had to make use of the Facebook and Twitter API (or other social networks) in order to implement the sharing feature.

Since iOS 6, Apple introduced a new framework known as *Social Framework*. The Social framework lets you integrate your app with any supported social networking services. Currently, it supports Facebook, Twitter, Sina Weibo, and Tencent Weibo. The framework gives you a standard composer to create posts for different social networks, and shields you

from learning the APIs of the social networks. You don't even need to know how to initiate a network request or handle single sign-on. The Social Framework simplifies everything. You just need to write a few lines of code to bring up the composer for users to tweet / publish Facebook posts within your app.

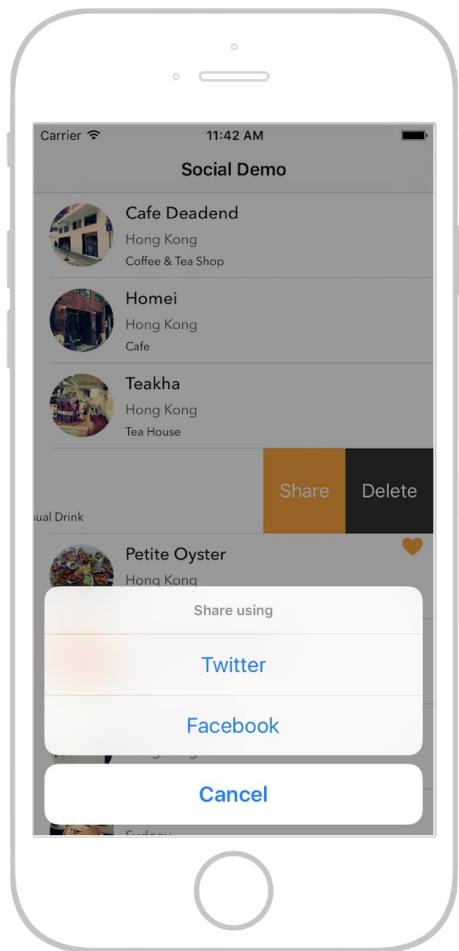
The framework comes with a very handy class called `SLComposeViewController`. Similar to `MFMailComposeViewController`, the `SLComposeViewController` class presents a standard view controller for users to compose tweets or Facebook posts. It also allows developers to preset the initial text, attach images, and add a URL to the post. If you just want to implement a simple sharing feature, this is the only class you need to know.

If you're not familiar with `SLComposeViewController`, the sample photo shown at the beginning of the chapter is what it looks like within your app.

Create the Demo Project and Design the Interface

Now that you should have a basic idea about the framework, let's get started and see how to add Twitter and Facebook sharing in a sample app.

To begin, you can download the project template from <https://www.dropbox.com/s/v2kosy25x4txsbq/SocialDemoTemplate.zip?dl=0>. The app, which is a modified version of a demo app in the beginner book, displays a list of restaurants in the main screen. When a user swipes a cell and taps the Share button, the app allows the user to share the selected restaurant on Facebook or Twitter.



The Facebook and Twitter sharing features are not yet implemented in the template. These are what we're going to work on.

Assumption: I assume that you understand how `UITableViewRowAction` works. If not, you can refer to [our beginner book](<http://www.appcoda.com/swift/>) or the [official documentation](<https://developer.apple.com/library/prerelease/ios/documentation/UIKit/Reference>)

Adding Twitter Support

Let's start with the implementation of a Twitter button. Open `RestaurantTableViewController.swift` and look into the `editActionsForRowAtIndexPath` method. You should find the code snippet shown below that instantiates the `UIAlertAction` instances of Twitter and Facebook actions. For both `UIAlertAction` instances, the handler is set to `nil`. Now we will implement the `twitterAction` for users to tweet:

```

let shareAction = UITableViewRowAction(style:
UITableViewCellStyle.Default, title: "Share", handler: {
(action:UITableViewRowAction, indexPath:NSIndexPath) -> Void in

    let shareMenu = UIAlertController(title: nil, message: "Share using",
preferredStyle: .ActionSheet)
    let twitterAction = UIAlertAction(title: "Twitter", style:
UIAlertActionStyle.Default, handler: nil)
    let facebookAction = UIAlertAction(title: "Facebook", style:
UIAlertActionStyle.Default, handler: nil)
    let cancelAction = UIAlertAction(title: "Cancel", style:
UIAlertActionStyle.Cancel, handler: nil)

    shareMenu.addAction(twitterAction)
    shareMenu.addAction(facebookAction)
    shareMenu.addAction(cancelAction)

    self presentViewController(shareMenu, animated: true, completion: nil)
}
)

```

Because the `SLComposeViewController` class is provided by the Social framework, the first thing to do is import the `Social` framework. Place the following statement at the very beginning of the `RestaurantTableViewController` class:

```
import Social
```

Next, update the `twitterAction` variable to the following:

```

let twitterAction = UIAlertAction(title: "Twitter", style:
UIAlertActionStyle.Default, handler: { (action) -> Void in

    // Check if Twitter is available. Otherwise, display an error message
    guard
        SLComposeViewController.isAvailableForServiceType(SLSERVICETypeTwitter) else {
            let alertMessage = UIAlertController(title: "Twitter Unavailable",
message: "You haven't registered your Twitter account. Please go to Settings >
Twitter to create one.", preferredStyle: .Alert)
            alertMessage.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))
            self presentViewController(alertMessage, animated: true, completion:
nil)

            return
}

```

```

// Display Tweet Composer
let tweetComposer = SLComposeViewController(forServiceType:
SLServiceTypeTwitter)
tweetComposer.setInitialText(self.restaurantNames[indexPath.row])
tweetComposer.addImage(UIImage(named:
self.restaurantImages[indexPath.row]))
self.presentViewController(tweetComposer, animated: true, completion: nil)

})

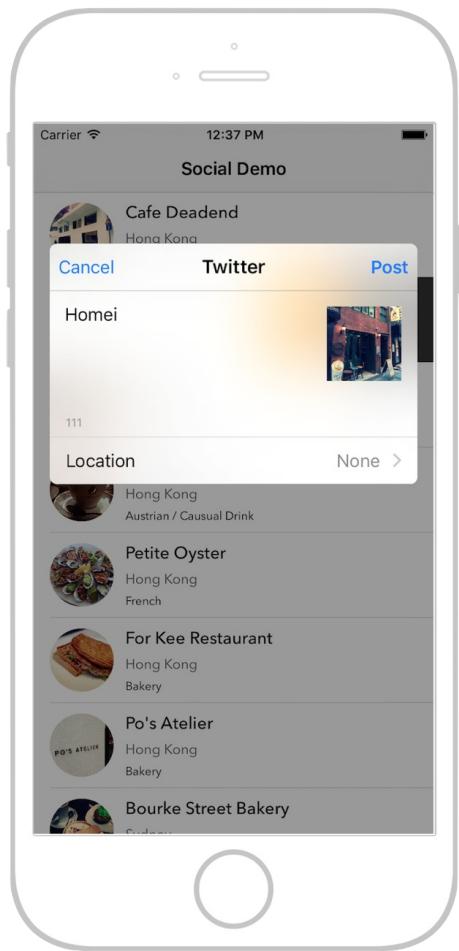
```

Before testing the app, let's go through the above code line by line. First, we have a guard statement to validate if the device is capable of sending tweets. We use the `isAvailableForServiceType` method to verify whether or not the Twitter service (`SLServiceTypeTwitter`) is available. One reason why users can't access the Twitter service is because they haven't signed into their Twitter accounts in Settings. If the Twitter service is unavailable, we simply prompt an error message and instruct the user to sign on the account in iOS.

If the service is accessible, we then create an instance of the `SLComposeViewController` of the Twitter service, followed by setting the initial text and image in the composer. Lastly, we invoke the `presentViewController` method to bring up the Twitter composer.

That's the code we need to let users tweet within your app. It's much easier than you thought, right? Cool! It's ready to go. Hit the Run button to compile and execute the app. Swipe a restaurant record and tap the Share button. Once you select Twitter, the app shows you a Tweet composer, populated with the restaurant name and image.

Quick tip: You must sign in with your Twitter account before you can test the sharing feature. Go to Settings > Twitter and sign in.



Adding Facebook Support

Next, we will implement the Facebook action for publishing a wall post on Facebook. In the `editActionsForRowAtIndexPath` method of `RestaurantTableViewController.swift`, replace `facebookAction` with the following code:

```
let facebookAction = UIAlertAction(title: "Facebook", style:  
UIAlertActionStyle.Default, handler: { (action) -> Void in  
  
    // Check if Twitter is available. Otherwise, display an error message  
    guard  
        SLComposeViewController.isAvailableForServiceType(SLSERVICETypeFacebook) else {  
            let alertMessage = UIAlertController(title: "Facebook Unavailable",  
            message: "You haven't registered your Facebook account. Please go to Settings >  
            Facebook to create one.", preferredStyle: .Alert)  
            alertMessage.addAction(UIAlertAction(title: "OK", style: .Default,  
            handler: nil))  
            self.presentViewController(alertMessage, animated: true, completion:  
            nil)  
        }  
    } )
```

```

        return
    }

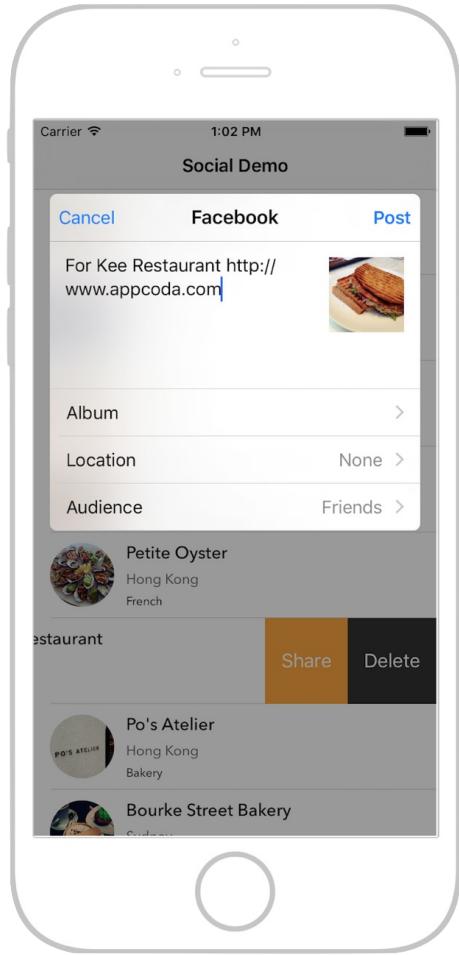
    // Display Tweet Composer
    let facebookComposer = SLComposeViewController(forServiceType:
SLServiceTypeFacebook)
    facebookComposer.setInitialText(self.restaurantNames[indexPath.row])
    facebookComposer.addImage(UIImage(named:
self.restaurantImages[indexPath.row]))
    facebookComposer.addURL(URL(string: "http://www.appcoda.com")!)
    self.presentViewController(facebookComposer, animated: true, completion:
nil)

})

```

That's it. The code is very similar to the code we've used in `twitterAction`. The only change is the service type. Instead of using `SLServiceTypeTwitter`, we tell `SLComposeViewController` to use `SLServiceTypeFacebook`. Further, we add a URL to the composer by calling the `addURL` method. Like the initial text and image, the URL is optional.

Let's run the app again and click the Facebook button. Your app should bring up the composer for publishing a Facebook post.



Summary

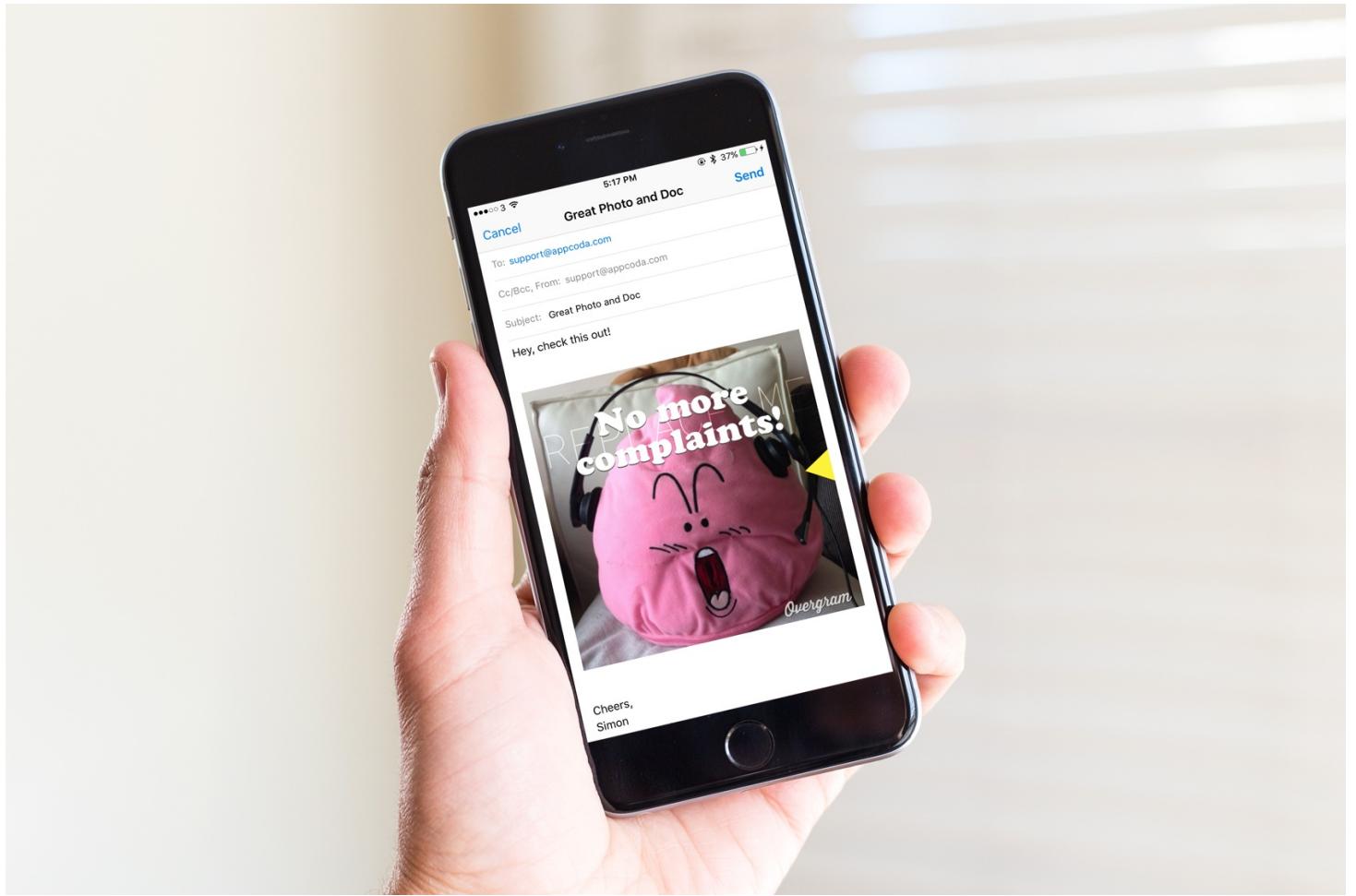
As you can see from this chapter, it's pretty easy to add Twitter and Facebook features using the Social Framework. If you're building your app, there is no reason why you shouldn't incorporate these social features.

In this chapter, I introduced the basics of Facebook and Twitter integration. You can try to tweak the sample app and upload multiple images to the social networks. However, if you want to access more advanced features such as displaying a user's Facebook friends, you'll need to make use of the Facebook API.

For reference, you can download the complete Xcode project from
<https://www.dropbox.com/s/sky3t2b6kuajwyu/SocialDemo.zip?dl=0>.

Chapter 6

Working with Email and Attachments



The `MessageUI` framework has made it really easy to send an email from your apps. You can easily use the built-in APIs to integrate an email composer in your app. In this short chapter, we'll show you how to send email and work with email attachments by creating a simple app.

Since the primary focus is to demonstrate the email feature of the `MessageUI` framework, we will keep the demo app very simple. The app simply displays a list of files in a plain table view. We'll populate the table with various types of files, including images in both PNG and JPEG formats, a Microsoft Word document, a Powerpoint file, a PDF document, and an HTML file. Whenever users tap on any of the files, the app automatically creates an email with the selected file as an attachment.

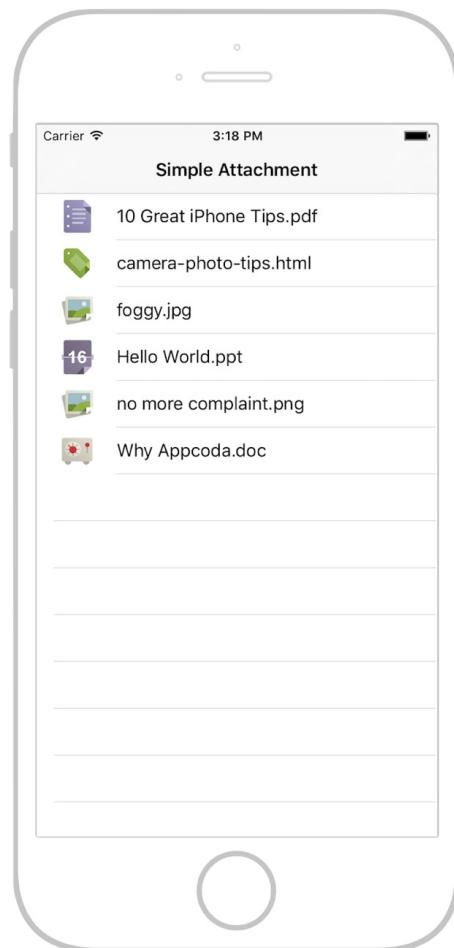
Starting with the Xcode Project Template

To save you from creating the Xcode project from scratch, you can download the project template from

<https://www.dropbox.com/s/ajq7wsogunnrcwd/EmailAttachmentTemplate.zip?dl=0> to begin the development. The project template comes with:

- a pre-built storyboard with a table view controller for displaying the list of files
- an AttachmentTableViewController class
- a set of files that are used as attachments
- a set of free icons from Pixeden (<http://www.pixeden.com/media-icons/flat-design-icons-set-vol1>)

After downloading and extracting the zipped file, you can compile and run the project. The demo app should display a list of files in the main screen. Now, we'll continue to implement the email feature.



Creating Email Using the MessageUI Framework

First, import the `MessageUI` framework and implement the `MFMailComposeViewControllerDelegate` in the `AttachmentTableViewController.swift` file. Your code should look like this:

```
import MessageUI

class AttachmentTableViewController: UITableViewController,
MFMailComposeViewControllerDelegate
```

Next, declare an enumeration for the MIME types in the class:

```
enum MIMETYPE: String {
    case jpg = "image/jpeg"
    case png = "image/png"
    case doc = "application/msword"
    case ppt = "application/vnd.ms-powerpoint"
    case html = "text/html"
    case pdf = "application/pdf"

    init?(type:String) {
        switch type.lowercaseString {
            case "jpg": self = .jpg
            case "png": self = .png
            case "doc": self = .doc
            case "ppt": self = .ppt
            case "html": self = .html
            case "pdf": self = .pdf
            default: return nil
        }
    }
}
```

MIME stands for Multipurpose Internet Mail Extensions. In short, MIME is an Internet standard that defines the way to send other kinds of information (e.g. graphic) through email. The MIME type indicates the type of data to attach. For instance, the MIME type of a PNG image is `image/png`. You can refer to the full list of MIME types at <http://www.iana.org/assignments/media-types/>.

Enumerations are particularly useful for storing a group of related values. So we use an enumeration to store the possible MIME types of the attachments. In Swift, you declare an

enumeration with the `enum` keyword and use the `case` keyword to introduce new enumeration cases. Optionally, you can assign a raw value for each case. In the above code, we define the possible types of the files and assign each case with the corresponding MIME type.

In Swift, you define initializers in enumerations to provide an initial case value. In the above initialization, we take in a file type/extension and look up for the corresponding case. Later we will use this enumeration when creating the `MFMailComposeViewController` object.

Next, create the methods for displaying the mail composer. Insert the following code in the same file:

```
func showEmail(attachmentFile: String) {

    // Check if the device is capable to send email
    guard MFMailComposeViewController.canSendMail() else {
        return
    }

    let emailTitle = "Great Photo and Doc"
    let messageBody = "Hey, check this out!"
    let toRecipients = ["support@appcoda.com"]

    // Initialize the mail composer and populate the mail content
    let mailComposer = MFMailComposeViewController()
    mailComposer.mailComposeDelegate = self
    mailComposer.setSubject(emailTitle)
    mailComposer.setMessageBody(messageBody, isHTML: false)
    mailComposer.setToRecipients(toRecipients)

    // Determine the file name and extension
    let fileparts = attachmentFile.componentsSeparatedByString(".")
    let filename = fileparts[0]
    let fileExtension = fileparts[1]

    // Get the resource path and read the file using NSData
    guard let filePath = NSBundle mainBundle().pathForResource(filename,
ofType: fileExtension) else {
        return
    }

    // Get the file data and MIME type
    if let fileData = NSData(contentsOfFile: filePath),
        mimeType = MIMEType(type: fileExtension) {

        // Add attachment
```

```

        mailComposer.addAttachmentData(fileData, mimeType: mimeType.rawValue,
fileName: filename)

        // Present mail view controller on screen
        presentViewController(mailComposer, animated: true, completion: nil)
    }
}

func mailComposeController(controller: MFMailComposeViewController,
didFinishWithResult result: MFMailComposeResult, error: NSError?) {

    switch result.rawValue {
    case MFMailComposeResultCancelled.rawValue:
        print("Mail cancelled")
    case MFMailComposeResultSaved.rawValue:
        print("Mail saved")
    case MFMailComposeResultSent.rawValue:
        print("Mail sent")
    case MFMailComposeResultFailed.rawValue:
        print("Failed to send: \(error)")
    default: break
    }

    dismissViewControllerAnimated(true, completion: nil)
}

```

The `showEmail` method takes an attachment. At the very beginning, we check to see if the device is capable of sending email using the `MFMailComposeViewController.canSendMail()` method. Once we get a positive result, we instantiate an `MFMailComposeViewController` object and populate it with some initial values including the email subject, message content, and the recipient email. The `MFMailComposeViewController` class provides the standard user interface for managing the editing and sending of an email message. Later when it is presented, you will see the predefined values in the mail message.

To add an attachment, all you need to do is call up the `addAttachmentData` method of the `MFMailComposeViewController` class.

```
mailComposer.addAttachmentData(fileData, mimeType: mimeType, fileName:
filename)
```

The method takes in three parameters:

- the data to attach – this is the content of a file that you want to attach in the form of

```
NSData .
```

- the MIME type – the MIME type of the attachment (e.g. image/png).
- the file name – that's the preferred file name to associate with the attachment.

The rest of the code in the `showEmail` method is used to determine the values of these parameters.

We first determine the path of the given file by using the `pathForResource` method of `NSBundle`. In iOS, an `NSBundle` object represents a location in the file system of a resource group. In general, the main bundle corresponds to the directory where the current application executable is located. Since our resource files are embedded in the app, we retrieve the main bundle object. We then call the `pathForResource` method to retrieve the path of the attachment.

The last block of the code is the core part of the method.

```
// Get the file data and MIME type
if let fileData = NSData(contentsOfFile: filePath),
    mimeType = MIMEType(type: fileExtension) {

    // Add attachment
    mailComposer.addAttachmentData(fileData, mimeType: mimeType.rawValue,
fileName: filename)

    // Present mail view controller on screen
    presentViewController(mailComposer, animated: true, completion: nil)
}
```

Base on the file path of the file, we create an `NSData` object for it. Furthermore, we determine the MIME type of the given file in reference to its file extension. Both initializations return an optional. In Swift 1.2 or later, you are allowed to combine multiple `if let` statements into one. Multiple optional bindings are separated by commas.

Before Swift 1.2:

Swift 1.2 or later:

```
if let fileData = NSData(contentsOfFile: filePath),  
    mimeType = MIMEType(type: fileExtension) {  
}  
}
```

Once we successfully initialized the file data and MIME type, we create the `addAttachmentData` method to attach the file and then present the mail composer.

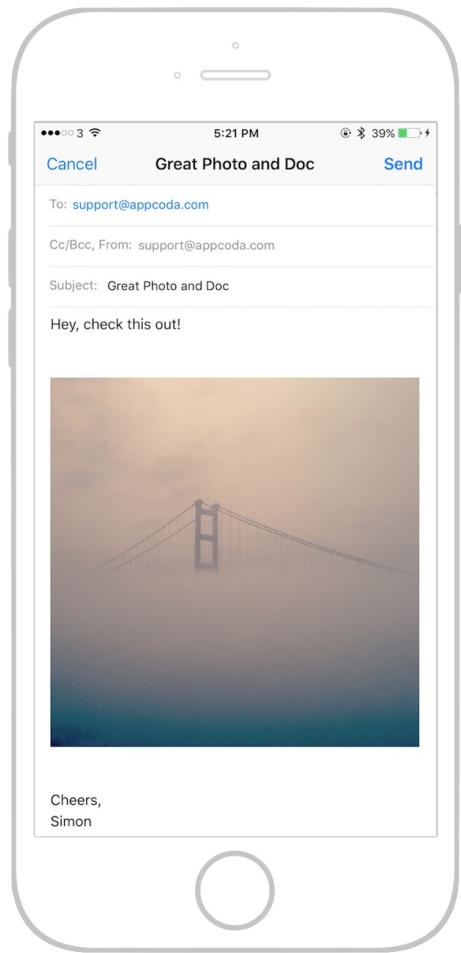
You may be familiar with the implementation of the

`mailComposeController(_:didFinishWithResult:_)` method. The method is defined in the `MFMailComposeViewControllerDelegate` protocol and is called when the mail interface is closed. For demo purposes, we just log the mail result and dismiss the mail controller. In real world apps, you can display an alert message if the mail fails to send.

We're almost ready to test the app. The app will bring up the mail interface when any of the files are selected. So the last thing is to add the `didSelectRowAtIndexPath:` method:

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {  
    let selectedFile = self.filenames[indexPath.row]  
    showEmail(selectedFile)  
}
```

You're good to go. Compile and run the app. Tap a file and the app should display the mail interface with your selected attachment. Note that there is a bug in the iOS simulator. You may not be able to bring up the mail composer. Try to run the app on a real iOS device, it should work.



For your complete reference, you can download the full source from
<https://www.dropbox.com/s/ymnpeqmk3wz73qu>EmailAttachment.zip?dl=0>.

Chapter 7

Sending SMS and MMS Using MessageUI Framework



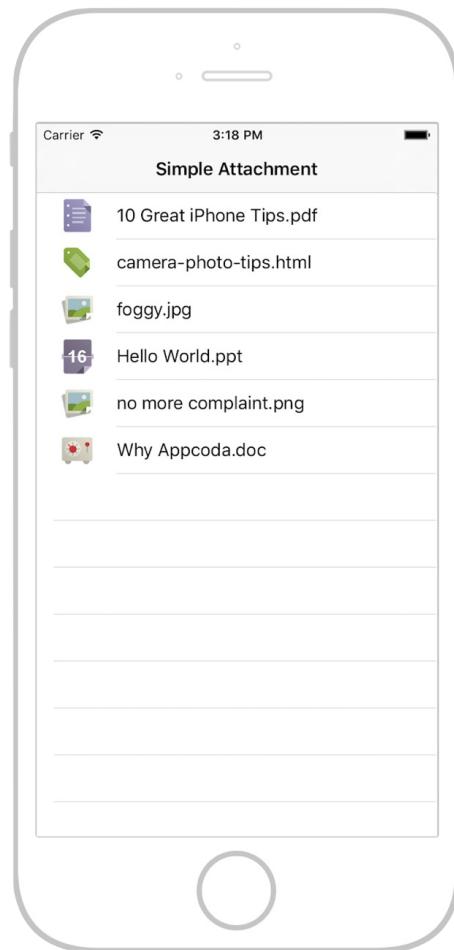
Not only designed for email, the `MessageUI` framework also provides a specialized view controller for developers to present a standard interface for composing SMS text messages within apps. While you can use the `MFMailComposeViewController` class for composing emails, the framework provides another class named `MFMessageComposeViewController` for handling text messages.

Basically the usage of `MFMessageComposeViewController` is very similar to the mail composer class. If you've read the previous chapter about creating emails with attachments, you will find it pretty easy to compose text messages. Anyway, I'll walk you through the usage of

`MFMessageComposeViewController` class. Again we will build a simple app to walk you through the class.

A Glance at the Demo App

We'll reuse the previous demo app but tweak it a bit. The app still displays a list of files in a table view. However, instead of showing the mail composer, the app will bring up the message interface with a pre-filled message when a user taps any of the files.



Getting Started

To save you time from creating the Xcode project from scratch, you can download the project template from <https://www.dropbox.com/s/k48pgf9vmzfox2o/SMSDemoTemplate.zip?dl=0> to begin with. I have pre-built the storyboard and already loaded the table data for you.

Implementing the Delegate

Go to the `AttachmentTableViewController.swift` file. Add the following code to import the `MessageUI` framework and implement the `MFMessageComposeViewControllerDelegate` protocol:

```
import MessageUI

class AttachmentTableViewController: UITableViewController,
MFMessageComposeViewControllerDelegate
```

The `MFMessageComposeViewControllerDelegate` protocol defines a single method which will be called when a user finishes composing the message. We have to provide the implementation of the method to handle various situations:

- A user cancels the editing of an SMS
- A user taps the send button and the SMS is sent successfully
- A user taps the send button, but the SMS has failed to send

Insert the following code in the `AttachmentTableViewController` class:

```
func messageComposeViewController(controller: MFMessageComposeViewController,
didFinishWithResult result: MessageComposeResult) {

    switch(result.rawValue) {
    case MessageComposeResultCancelled.rawValue:
        print("SMS cancelled")

    case MessageComposeResultFailed.rawValue:
        let alertMessage = UIAlertController(title: "Failure", message: "Failed
to send the message.", preferredStyle: .Alert)
        alertMessage.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))
        presentViewController(alertMessage, animated: true, completion: nil)

    case MessageComposeResultSent.rawValue:
        print("SMS sent")

    default: break
}

dismissViewControllerAnimated(true, completion: nil)
}
```

Here, we just display an alert message when the app fails to send a message. For other cases, we log the error to the console and dismiss the message composer.

Bring Up the Message Composer

When a user selects a file, we retrieve the selected file and call up a helper method to bring up the message composer. Insert the `didSelectRowAtIndexPath` method in the `AttachmentTableViewController` class:

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let selectedFile = filenames[indexPath.row]
    sendSMS(selectedFile)
}
```

The `sendsMS` method is the core method to initialize and populate the default content of the SMS text message. Create the method using the following code:

```
func sendSMS(attachment:String) {
    // Check if the device is capable of sending text message
    guard MFMessageComposeViewController.canSendText() else {
        let alertMessage = UIAlertController(title: "SMS Unavailable", message: "Your device is not capable of sending SMS.", preferredStyle: .Alert)
        alertMessage.addAction(UIAlertAction(title: "OK", style: .Default, handler: nil))
        presentViewController(alertMessage, animated: true, completion: nil)
        return
    }

    // Prefill the SMS
    let messageController = MFMessageComposeViewController()
    messageController.messageComposeDelegate = self
    messageController.recipients = ["12345678", "72345524"]
    messageController.body = "Just sent the \(attachment) to your email. Please check!"
}

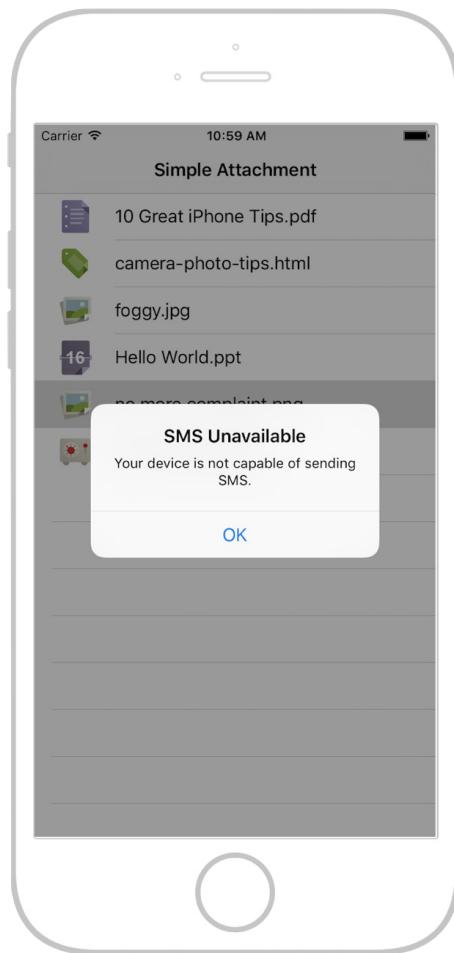
// Present message view controller on screen
presentViewController(messageController, animated: true, completion: nil)
}
```

Though most of the iOS devices should be capable of sending a text message, you should be prepared for the exception. What if your app is used on an iPod touch with iMessages

disabled? In this case, the device is not allowed to send a text message. So at the very beginning of the code, we verify whether or not the device is allowed to send text messages by using the `canSendText` method of `MFMessageComposeViewController`.

The rest of the code is very straightforward and similar to what we did in the previous chapter. We pre-populate multiple recipients (i.e. phone number) in the text message and set the message body.

With the content ready, simply invoke `presentViewController` to bring up the message composer. That's it! Simple and easy. You can now run the app and test it out. But please note you have to test the app on a real iOS device. If you use the simulator to run the app, it shows you an alert message.



Sending MMS

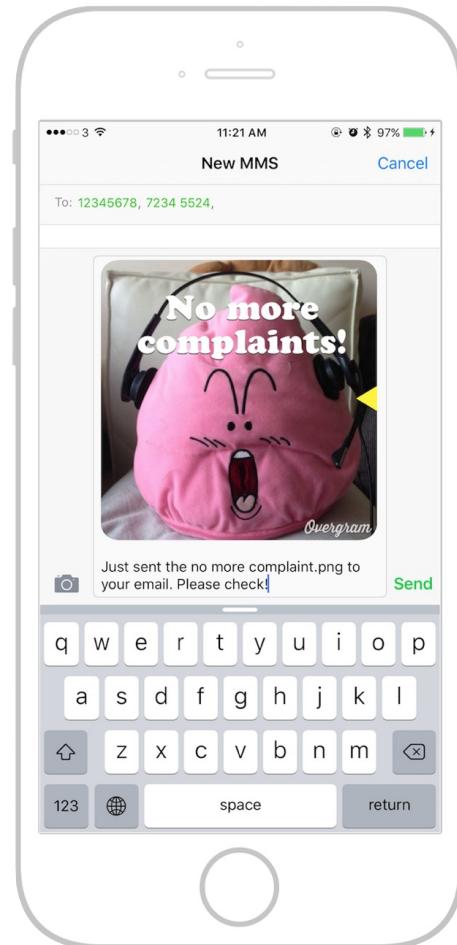
Wait! The app can only send a text message. How about a file attachment? The

`MFMessageComposeViewController` class also supports sending attachments via MMS. You can use the code below to attach a file.

```
// Adding file attachment
let fileparts = attachment.componentsSeparatedByString(".")
let filename = fileparts[0]
let fileExtension = fileparts[1]
let filePath = NSBundle mainBundle().pathForResource(filename, ofType:
fileExtension)
let fileUrl = NSURL.fileURLWithPath(filePath!)
messageController.addAttachmentURL(fileUrl, withAlternateFilename: nil)
```

Just add the above lines in the `sendsSMS` method. The code is self explanatory. We get the selected file and retrieve the actual file path using the `pathForResource` method of `NSBundle`. Lastly, we add the file using the `addAttachmentURL` method.

Quick note: I have explained the code snippet in details before. Please refer to the previous chapter for details.



What if You Don't Want In-App SMS

The above implementation provides a seamless integration of the SMS feature in your app. But what if you just want to redirect to the default Messages app and send a text message? It's even simpler. You can do that by using a single line of code:

```
UIApplication.sharedApplication().openURL(URL(string: "sms:123456789")!)
```

In iOS, you're allowed to communicate with other apps using a URL scheme. The mobile OS already comes with built-in support of the *http*, *mailto*, *tel*, and *sms* URL schemes. When you open an HTTP URL, iOS by default launches the URL via Safari. If you want to open the Messages app, you can use the SMS URL scheme and specify the recipient. However, that URL scheme doesn't allow you to insert default content.

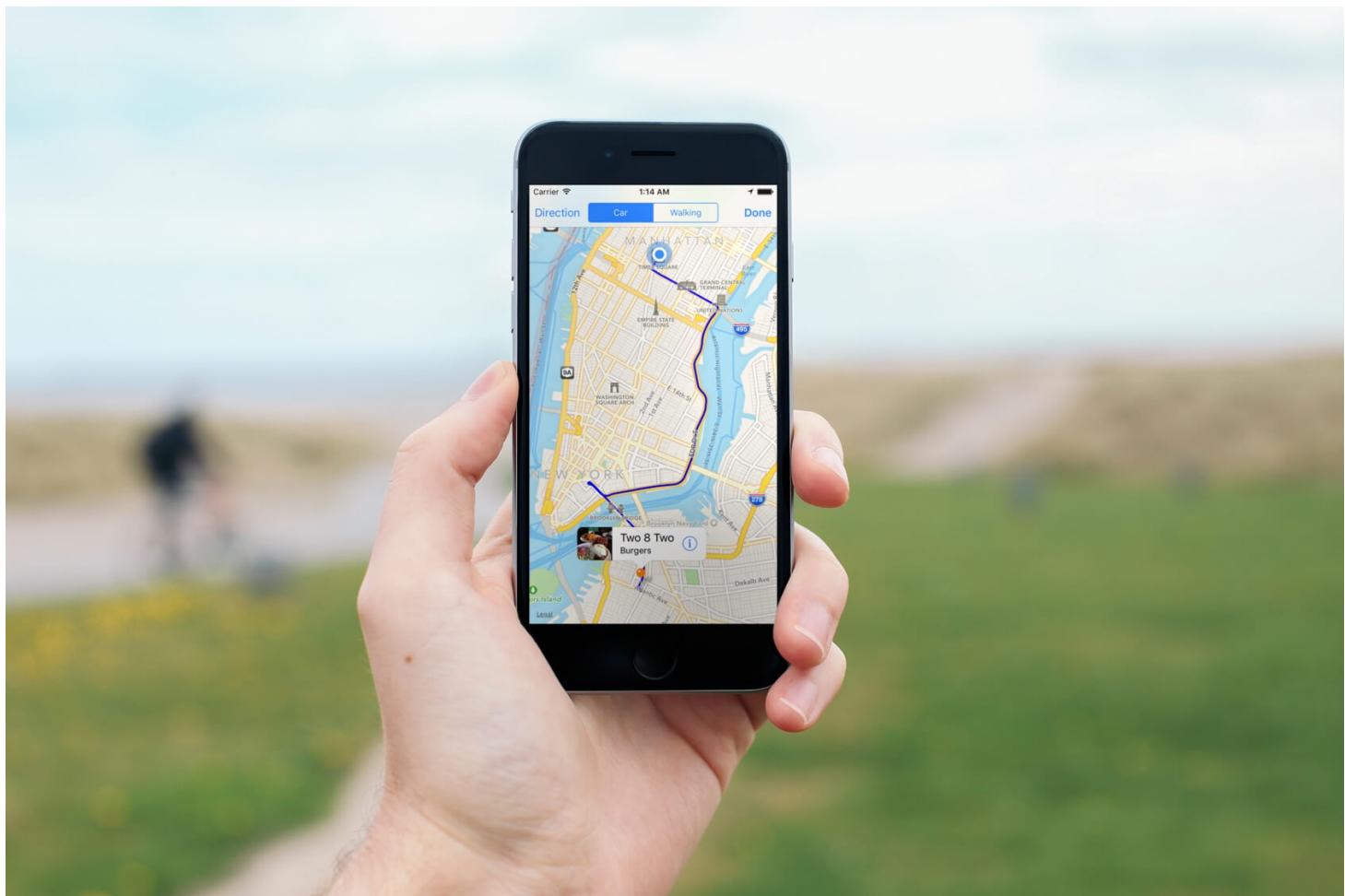
Wrap Up

In this chapter, I showed you a simple way to send a text message within an app. For reference, you can download the full source code from

<https://www.dropbox.com/s/eprt7qjkpxy4dti/SMSDemo.zip?dl=0>.

Chapter 8

How to Get Direction and Draw Route on Maps



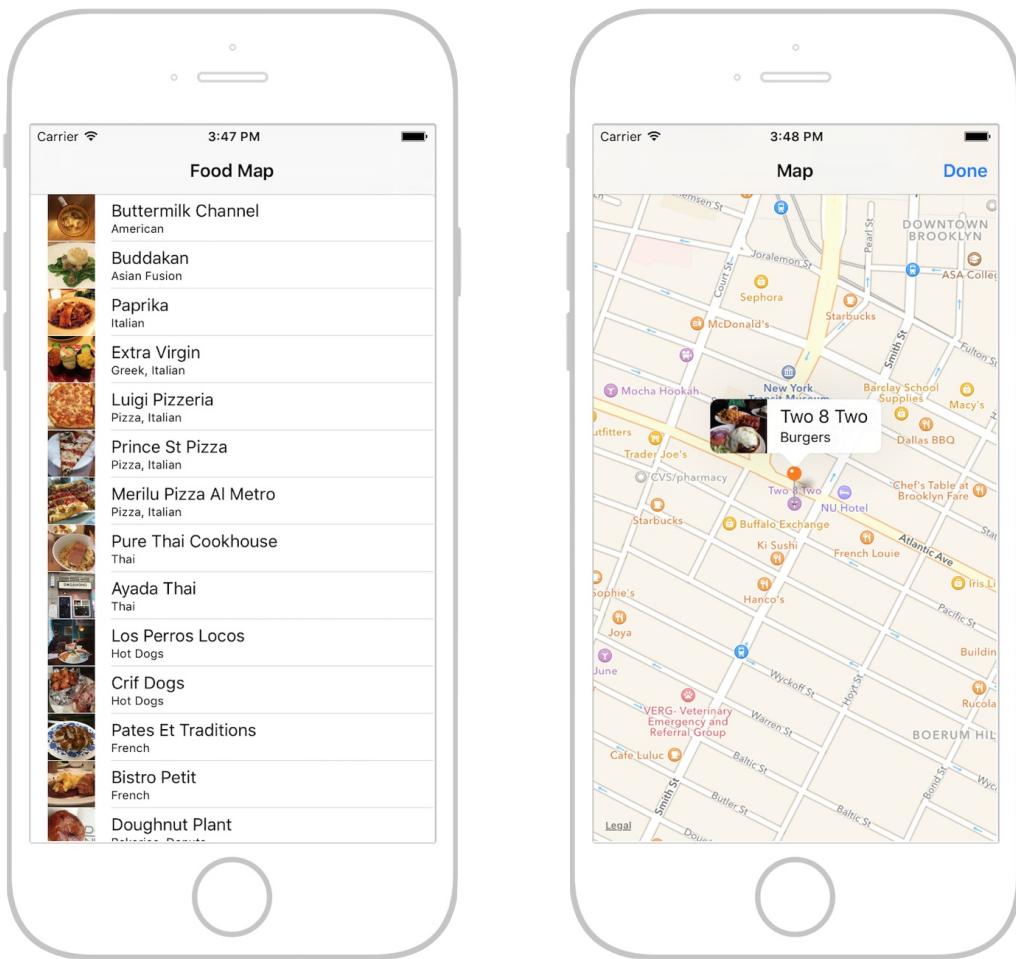
Since the release of the iOS 7 SDK, the `MapKit` framework includes the `MKDirections` API which allows iOS developers to access the route-based directions data from Apple's server. Typically you create an `MKDirections` instance with the start and end points of a route. The instance then automatically contacts Apple's server and retrieves the route-based data.

You can use the `MKDirections` API to get both driving and walking directions depending on your preset transport type. If you like, `MKDirections` can also provide you alternate routes. On top of all that, the API lets you calculate the travel time of a route.

Again we'll build a demo app to see how to utilize the `MKDrections` API.

Sample Route App

I have covered the basics of the `MapKit` framework in the [Beginning iOS 9 Programming with Swift book](#), so I expect you have some idea about how `MapKit` works, and understand how to pin a location on a map. To demonstrate the usage of the `MKDrections` API, we'll build a sample map app. You can start with this project template (<https://www.dropbox.com/s/da8ag58xcjqdfep/MapKitDirectionDemoTemplate.zip?dl=0>). If you build the template, you should have an app that shows a list of restaurants. By tapping a restaurant, the app brings you to the map view with the location of the restaurant annotated on the map. That's pretty much the same as what you have implemented in the FoodPin app. We'll enhance the demo app to get the user's current location, and display the directions to the selected restaurant.



There is one thing I want to point out. If you look into the `MapViewController` class, you will

find these lines of code:

```
// Pin color customization
if #available(iOS 9.0, *) {
    annotationView?.pinTintColor = UIColor.orangeColor()
}
```

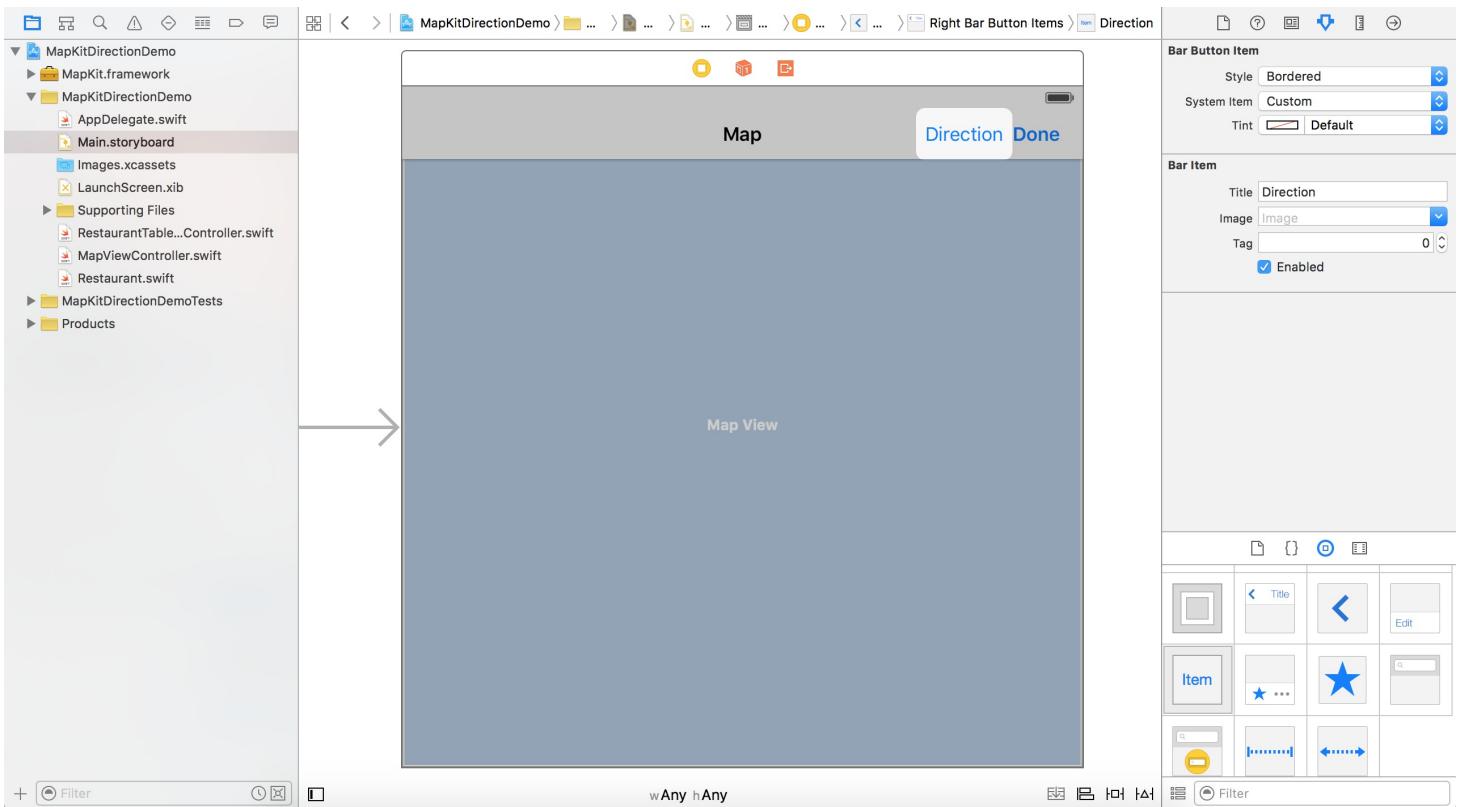
The iOS 9 SDK provides a new API for customizing the pin color. The `MKPinAnnotationView` class now lets you change the pin color using the `pinTintColor` property. Like this project, if your app is going to support both iOS 8 and 9, you will need to check the OS version before changing the pin color. Otherwise, this will cause errors when the app runs on older versions of iOS.

Starting from Swift 2, it has built-in support for checking API availability. You can easily define an availability condition so that the block of code will only be executed on certain iOS versions. You use the `#available` keyword in a `if` statement. In the availability condition, you specify the OS versions (e.g. iOS 9, OSX 10.10) you want to verify. The asterisk (*) is required and indicates that the `if` clause is executed on the minimum deployment target and any other versions of OS. In the above example, we will execute the code block only if the device is running on iOS 9 (or up).

Adding a Direction Button in the Navigation Bar

Now, open the Xcode project and go to `Main.storyboard`. Let's add a `Direction` button to the navigation bar of the map view controller. Drag a Bar Button Item from the Object library and add it to the navigation bar. Set the title of the button to Direction. When this button is tapped, the app will get the user's current location and display the directions to the restaurant.

Quick note: Do you notice that you can now place two bar button items next to each other using Interface Builder? It is a new feature in Xcode 7. Prior to that, you cannot do that without writing your own code.

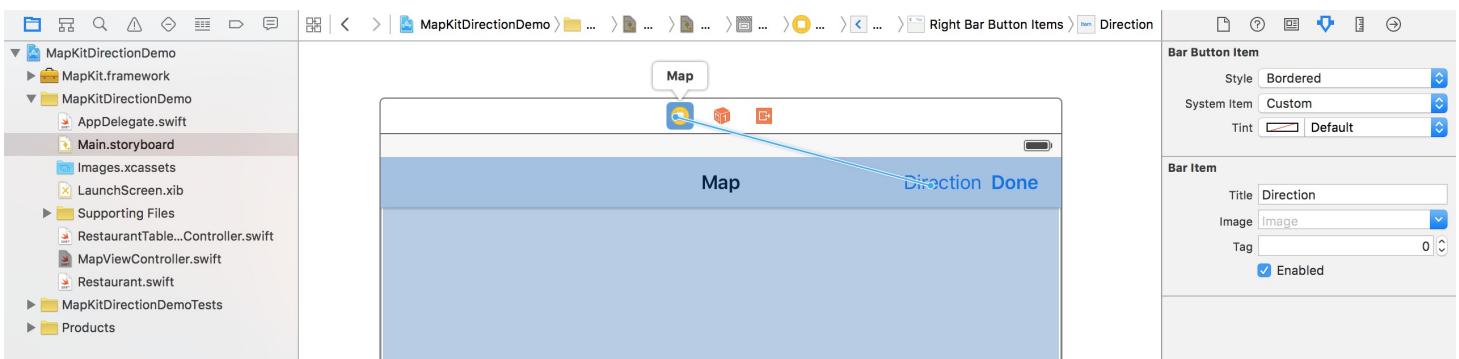


Creating an Action Method for the Direction Button

Next, create an empty action method named `showDirection` in the `MapViewController` class. We'll provide the implementation in the later section.

```
@IBAction func showDirection(sender: AnyObject) {
```

In the storyboard, establish a connection between the `Direction` button and the action method. Control-drag from the `Direction` button to the view controller icon in the dock. Select `showDirection:` to connect with the action method.



Displaying the User Location on Maps

Since our app is going to display a route from the user's current location to the selected restaurant, we have to enable the map view to show the user's current location. By default, the `MKMapView` class doesn't display the user's location on the map. You can set the `showsUserLocation` property of the `MKMapView` class to `true` to enable it. Because the option is set to true, the map view uses the built-in Core Location framework to search for the current location and display it on the map.

In the `viewDidLoad` method of the `MapViewController` class, insert the following line of code:

```
mapView.showsUserLocation = true
```

If you can't wait to test the app and see how it displays the user location, you can compile and run the app. Select any of the restaurants to bring up the map. Unfortunately, it doesn't work as expected. If you look into the console, you should find the following error:

```
MapKitDirectionDemo[2337:222562] Trying to start MapKit location updates  
without prompting for location authorization. Must call -[CLLocationManager  
requestWhenInUseAuthorization] or -[CLLocationManager  
requestAlwaysAuthorization] first.
```

Starting from iOS 8, Core Location introduces a new feature known as *Location Authorization*. You have to explicitly ask for a user's permission to grant your app location services. Basically, you need to implement these two things to get the location working:

- Request a user's authorization by calling the `requestWhenInUseAuthorization` or `requestAlwaysAuthorization` method of `CLLocationManager`.
- Add a key (`NSLocationWhenInUseUsageDescription` / `NSLocationAlwaysUsageDescription`) to your `Info.plist`.

There are two types of authorization: `requestWhenInUseAuthorization` and `requestAlwaysAuthorization`. You use the former if your app only needs location updates when it's in use. The latter is designed for apps that use location services in the background (suspended or terminated). For example, a social app that tracks a user's location requires location updates even if it's not running in the foreground. Obviously, `requestWhenInUseAuthorization` is good enough for our demo app.

To do that, you will need to add a key to your `Info.plist`. Depending on the authorization type, you can either add the `NSLocationWhenInUseUsageDescription` or `NSLocationAlwaysUsageDescription` key to `Info.plist`. Both keys contain a message telling a user why your app needs location services. For example, you can enter a string like *Location is required to find out your current location*.

MapKitDirectionDemo > MapKitDirectionDemo > Supporting Files > Info.plist > No Selection			
	Key	Type	Value
MapKitDirectionDemo	Information Property List	Dictionary	(15 items)
MapKit.framework	Localization native development region	String	en
MapKitDirectionDemo	Executable file	String	\$(EXECUTABLE_NAME)
AppDelegate.swift	Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
Main.storyboard	InfoDictionary version	String	6.0
Images.xcassets	Bundle name	String	\$(PRODUCT_NAME)
LaunchScreen.xib	Bundle OS Type code	String	APPL
Supporting Files	Bundle versions string, short	String	1.0
Info.plist	Bundle creator OS Type code	String	????
RestaurantTable...Controller.swift	Bundle version	String	1
MapViewController.swift	Application requires iPhone environment	Boolean	YES
Restaurant.swift	Launch screen interface file base name	String	LaunchScreen
MapKitDirectionDemoTests	Main storyboard file base name	String	Main
Products	► Required device capabilities	Array	(1 item)
	► Supported interface orientations	Array	(3 items)
	NSLocationWhenInUseUsageDescri... (selected)	String	Location is required to find out your current location.

Now we are ready to modify the code again. First, declare a location manager variable in the `MapViewController` class:

```
let locationManager = CLLocationManager()
```

Insert the following lines of code in the `viewDidLoad` method right after `super.viewDidLoad()`:

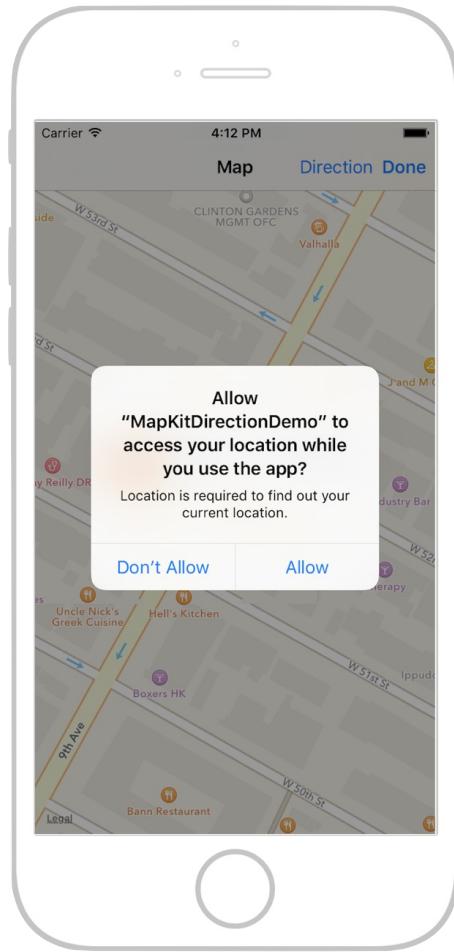
```
// Request for a user's authorization for location services
locationManager.requestWhenInUseAuthorization()
let status = CLLocationManager.authorizationStatus()

if status == CLAuthorizationStatus.AuthorizedWhenInUse {
    mapView.showsUserLocation = true
}
```

The first line of code calls the `requestWhenInUseAuthorization` method. The method first checks the current authorization status. If the user has not yet been asked to authorize location updates, it automatically prompts the user to authorize the use of location services.

Once the user makes a choice, we check the authorization status to see if the user granted permission. If yes, we enable `showsUserLocation` in the app.

Now run the app again and have a quick test. When you launch the map view, you'll be prompted to authorize location services. As you can see, the message shown is the one we specified in the `NSLocationWhenInUseUsageDescription` key. Remember to hit the *Allow* button to enable the location updates.



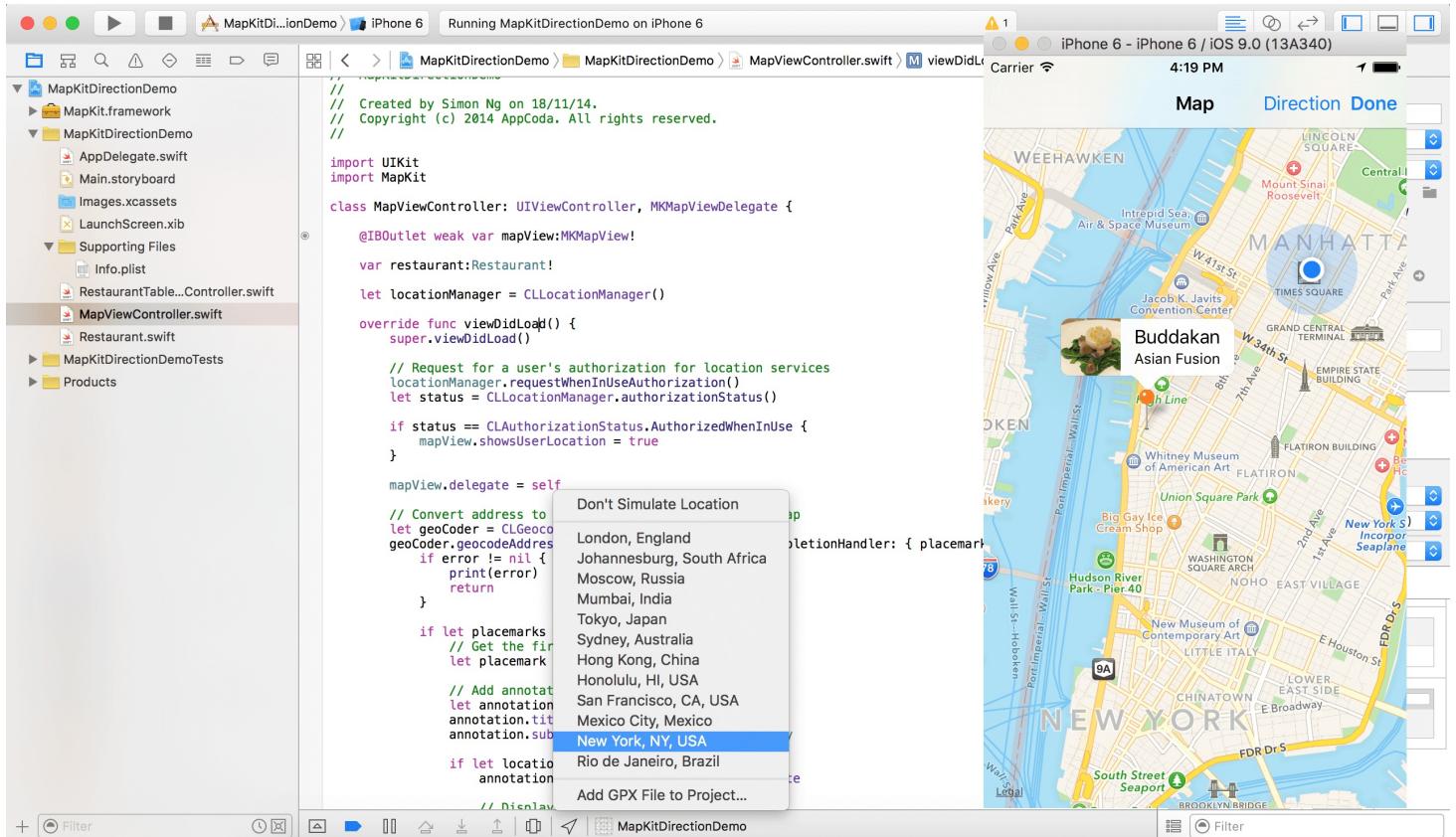
Testing Location Using the Simulator

Wait! How can we simulate the current location using the built-in simulator? How can you tell the simulator where you are?

There is no way for the simulator to get the current location of your computer. However, the simulator allows you to fake its location. By default, the simulator doesn't simulate the location. You have to enable it manually. While running the app, you can use the *Simulate location* button (arrow button) in the toolbar of the debug area. Xcode comes with a number of preset locations. Just change it to your preferred location (e.g. New York). Alternatively, you can set the default location of your simulator. Just click your scheme > Edit Scheme to bring

up the scheme editor. Select the *Options* tab and set the default location.

Once you set the location, the simulator will display a blue dot in the map which indicates the current user location. If you can't find the blue dot on the map, simply zoom out. In the simulator, you can hold down the *option* key to simulate the pinch-in and pinch-out gestures. For details, you can refer to [Apple's official document](#).



Using MKDirections API to Get the Route info

With the user location enabled, we will compute the route between the current location and the location of the restaurant. First declare a `placemark` variable in the `MapViewController` class:

```
var currentPlacemark: CLPlacemark?
```

This variable is used to save the current placemark. In other words, it is the `placemark` object of the selected restaurant. In the `viewDidLoad` method, locate the following line:

```
let placemark = placemarks[0]
```

Add the following code right below it:

```
self.currentPlacemark = placemark
```

Next we'll implement the `showDirection` method and use the `MKDrections` API to get the route data. Update the method by using the following code snippet:

```
@IBAction func showDirection(sender: AnyObject) {

    guard let currentPlacemark = currentPlacemark else {
        return
    }

    let directionRequest = MKDirectionsRequest()

    // Set the source and destination of the route
    directionRequest.source = MKMapItem.mapItemForCurrentLocation()
    let destinationPlacemark = MKPlacemark(placemark: currentPlacemark)
    directionRequest.destination = MKMapItem(placemark: destinationPlacemark)
    directionRequest.transportType = MKDirectionsTransportType.Automobile

    // Calculate the direction
    let directions = MKDirections(request: directionRequest)

    directions.calculateDirectionsWithCompletionHandler { (routeResponse,
    routeError) -> Void in

        guard let routeResponse = routeResponse else {
            if let routeError = routeError {
                print("Error: \(routeError)")
            }
        }

        return
    }

    let route = routeResponse.routes[0]
    self.mapView.addOverlay(route.polyline, level:
    MKOverlayLevel.AboveRoads)

}
```

At the beginning of the method, we make sure if `currentPlacemark` contains a value. Otherwise, we just skip everything. To request directions, we first create an instance of `MKDrectionsRequest`. The class is used to store the source and destination of a route. There are

a few optional parameters you can configure such as transport type, alternate routes, etc. In the above code, we just set the source, destination and transport type while using default values for the rest of the options. The starting point is set to the user's current location. We use `MKMapItem.mapItemForCurrentLocation` to retrieve the current location. The end point of the route is set to the destination of the selected restaurant. The transport type is set to `automobile`.

With the `MKDrectionsRequest` object created, we instantiate an `MKDrections` object and call the `calculateDirectionsWithCompletionHandler` method. The method initiates an asynchronous request for directions and calls your completion handler when the request is completed. The `MKDrections` object simply passes your request to the Apple servers and asks for route-based directions data. Once the request completes, the completion handler is called. The route information returned by the Apple servers is returned as an `MKDrectionsResponse` object. `MKDrectionsResponse` provides a container for saving the route information so that the routes are saved in the `routes` property.

In the completion handler block, we first check if the route response contains a value. Otherwise, we just print the error. If we can successfully get the route response, we retrieve the first `MKRoute` object. By default, only one route is returned. Apple may return multiple routes if the `requestsAlternateRoutes` property of the `MKDrectionsRequest` object is enabled. Because we didn't enable the alternate route option, we just pick the first route.

With the route, we add it to the map by calling the `addOverlay` method of the `MKMapView` class. The detailed route geometry (i.e. `route.polyline`) is represented by an `MKPolyline` object. The `addOverlay` method is used to add an `MKPolyline` object to the existing map view. Optionally, we configure the map view to overlay the route above roadways but below map labels or point-of-interest icons.

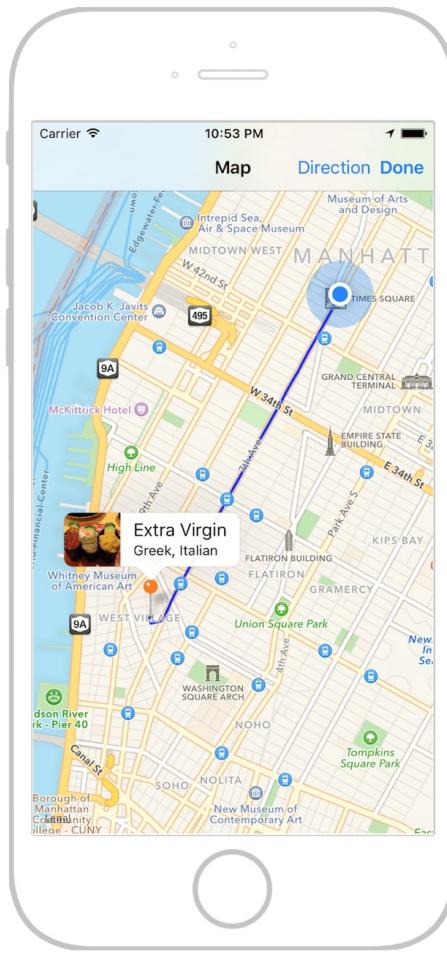
That's how you construct a direction request and overlay a route on map. If you run the app now, you will not see a route when the Direction button is tapped. There is still one thing left. We need to implement the `rendererForOverlay` method which actually draws the route:

```
func mapView(mapView: MKMapView!, rendererForOverlay overlay: MKOverlay!) ->
MKOverlayRenderer! {
    let renderer = MKPolylineRenderer(overlay: overlay)
    renderer.strokeColor = UIColor.blueColor()
    renderer.lineWidth = 3.0
```

```
        return renderer  
    }
```

In the method, we create an `MKPolylineRenderer` object which provides the visual representation for the specified MKPolyline overlay object. Here the overlay object is the one we added earlier. The renderer object provides various properties to control the appearance of the route path. We simply change the stroke color and line width.

Okay, let's run the app again and you should be able to see the route after pressing the `Direction` button. If you can't view the path, remember to check if you set the simulated location to New York.



Scale the Map to Make the Route Fit Perfectly

You should notice a problem with the current implementation. The demo app does indeed draw the route on the map, but you may need to zoom out manually in order to show the route.

Could we scale the map automatically?

You can use the `boundingMapRect` property of the polyline to determine the smallest rectangle that completely encompasses the overlay and changes the visible region of the map view.

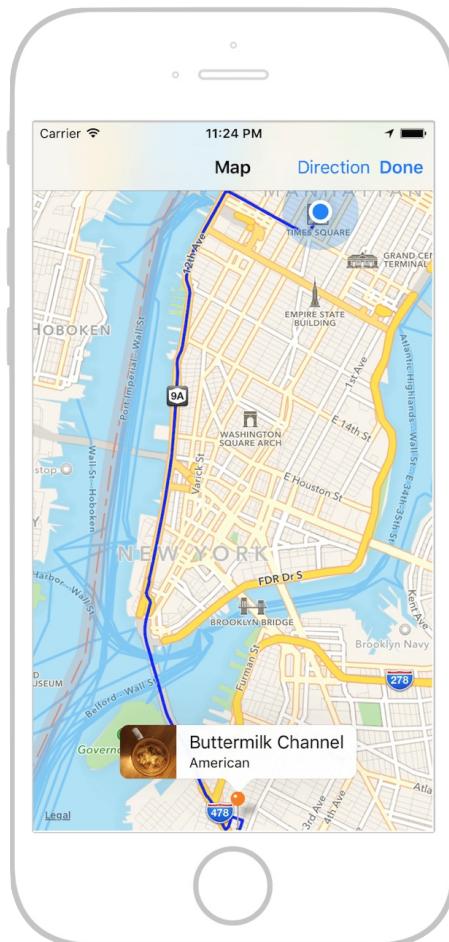
Insert the following lines of code in the `showDirection` method:

```
let rect = route.polyline.boundingMapRect  
mapView.setRegion(MKCoordinateRegionForMapRect(rect), animated: true)
```

And place them right after the following line of code:

```
self.mapView.addOverlay(route.polyline, level: MKOverlayLevel.AboveRoads)
```

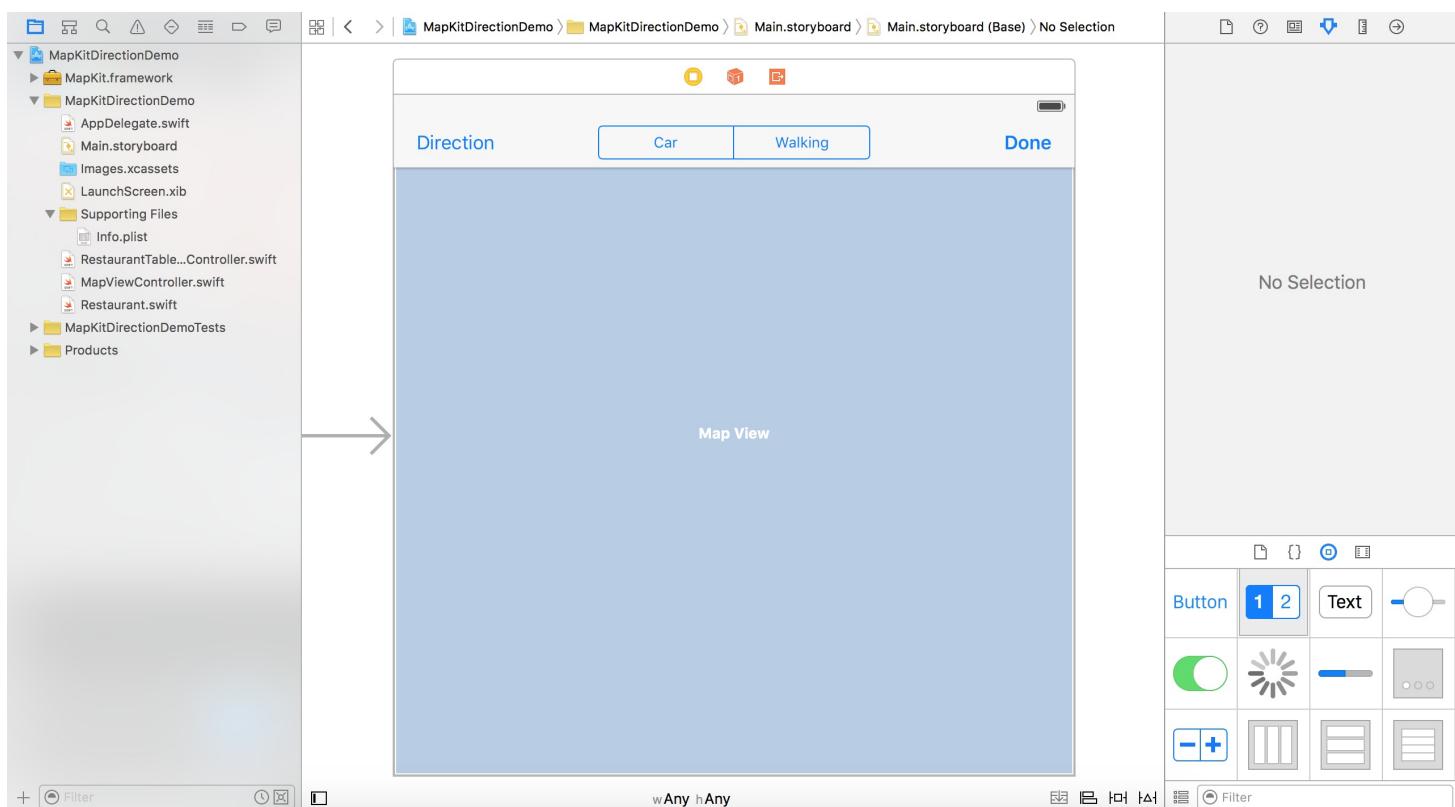
Compile and run the app again. The map should now scale automatically to display the route within the screen real estate.



Using Segmented control

Presently, the app only provides route information for automobile. Wouldn't it be great if the app supported walking directions? We'll add a segmented control in the app such that users can choose between driving and walking directions. A segmented control is a horizontal control made of multiple segments. Each segment of the control functions like a button.

Now go to the storyboard. Drag a segmented control from the Object library to the navigation bar of the map view controller. Select the segmented control and go to the Attributes inspector. Change the title of the first item to `Car` and the second item to `Walking`. Optionally, move the `Direction` button to the left of the navigation bar. The user interface will look better.



Next, go to `MapViewController.swift`. Declare an outlet variable for the segmented control:

```
@IBOutlet var segmentedControl: UISegmentedControl!
```

Go back to the storyboard and connect the segmented control with the outlet variable. In the `viewDidLoad` method of `MapViewController.swift`, put this line of code right after `super.viewDidLoad()`:

```
segmentedControl.hidden = true
```

We only want to display the control when a user taps the `Direction` button. This is why we hide it when the view controller is first loaded up.

Next, declare a new instance variable in the `MapViewController` class:

```
var currentTransportType = MKDirectionsTransportType.Automobile
```

The variable indicates the selected transport type. By default, it is set to `automobile` (i.e. car). Due to the introduction of this variable, we have to change the following line of code in the `showDirection` method:

```
directionRequest.transportType = MKDirectionsTransportType.Automobile
```

Simply replace `MKDirectionsTransportType.Automobile` with `currentTransportType`.

Okay, you've got everything in place. But how can you detect the user's selection of a segmented control? When a user presses one of the segments, the control sends a `valueChanged` event. So all you need to do is register the event and perform the corresponding action when the event is triggered.

You can register the event by control-dragging the segmented control's Value Changed event from the Connections inspector to the action method. But since you're now an intermediate programmer, let's see how you can register the event by writing code. Typically, you register the target-action methods for a segmented control like this:

```
segmentedControl.addTarget(self, action: "showDirection:", forControlEvents: .ValueChanged)
```

You use the `addTarget` method to register the `.ValueChanged` event. When the event is triggered, we instruct the control to call the `showDirection` method of the current object (i.e. `MapViewController`).

Since we need to check the selected segment, insert the following code snippet at the very beginning of the `showDirection` method:

```
switch segmentedControl.selectedSegmentIndex {
```

```
case 0: currentTransportType = MKDirectionsTransportType.Automobile
case 1: currentTransportType = MKDirectionsTransportType.Walking
default: break
}

segmentedControl.hidden = false
```

The `selectedSegmentedIndex` property of the segmented control indicates the index of the selected segment. If the first segment (i.e. Car) is selected, we set the current transport type to automobile. Otherwise, it is set to walking. We also unhide the segmented control.

Lastly, insert the following line of code (highlighted in yellow) in the `calculateDirectionsWithCompletionHandler` closure:

```
self.mapView.removeOverlays(self.mapView.overlays)
```

Place the line of code right before calling the `addOverlay` method. Your closure should look like this:

```
directions.calculateDirectionsWithCompletionHandler { (routeResponse,
routeError) -> Void in

    guard let routeResponse = routeResponse else {
        if let routeError = routeError {
            print("Error: \(routeError)")
        }
    }

    return
}

let route = routeResponse.routes[0]
self.mapView.removeOverlays(self.mapView.overlays)
self.mapView.addOverlay(route.polyline, level: MKOverlayLevel.AboveRoads)

let rect = route.polyline.boundingMapRect
self.mapView.setRegion(MKCoordinateRegionForMapRect(rect), animated: true)
}
```

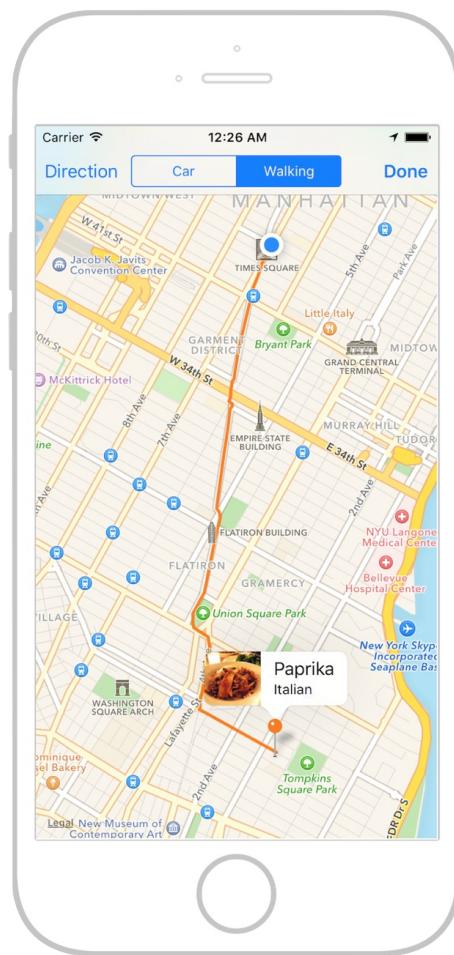
The line of code simply asks the map view to remove all the overlays. This is to avoid both *Car* and *Walk* routes overlapping with each other.

You can now test the app. In the map view, tap the Direction button and the segmented control should appear. You're free to select the Walking segment to display the walking directions. For

now, both types of routes are shown in blue. You can make a minor change in the `rendererForOverlay` method of the `MapViewController` class to display a different color. Simply change this line of code:

```
renderer.strokeColor = (currentTransportType == .Automobile) ?  
    UIColor.blueColor() : UIColor.orangeColor()
```

We use blue color for the *Car* route and orange color for the *Walking* route. After the change, run the app again. When walking is selected, the route is displayed in orange.



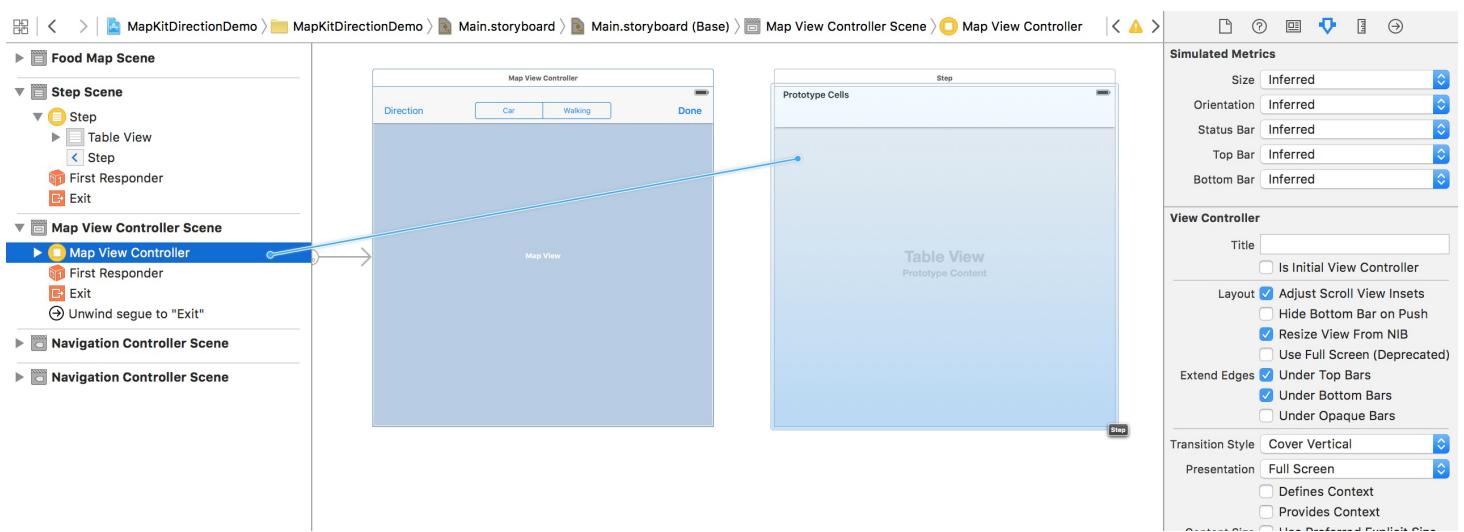
Showing Route Steps

Now that you know how to display a route in a map, wouldn't it be great if you can provide detailed driving (or walking) directions for your users? The `MKRoute` object provides a property called `steps`, which contains an array of `MKRouteStep` objects. An `MKRouteStep` object represents one part of an overall route. Each step in a route corresponds to a single instruction

that would need to be followed by the user.

Okay, let's tweak the demo. When someone taps the annotation, the app will display the detailed driving/walking instructions.

First, add a table view controller to the storyboard and set the identifier of the prototype cell as `toCell`. Next, connect the map view controller with the new table view controller using a segue. In the Document Outline of Interface Builder, control-drag the map view controller to the table view controller. Select `show` for the segue type and set the segue's identifier to `showSteps`.



The UI design is ready. Now create a new class file using the Cocoa Touch class template. Name it `RouteTableViewController` and make it a subclass of `UITableViewController`. Once the class is created, go back to the storyboard. Select the Steps table view controller. Under the Identity inspector, set the custom class to `RouteTableViewController`.

You may have these two questions in your head:

- How can we get the detailed steps from the route?
- How do we know if a user touches the annotation in a map?

As I mentioned earlier, the `steps` property of an `MKRoute` object contains an array of `MKRouteStep` objects. Each `MKRouteStep` object comes with an `instructions` property that stores the written instructions (e.g. Turn right onto Charles St) for following the path of a particular step. So all we need to do is loop through all the `MKRouteStep` objects to display the

written instructions in the Steps table view.

Similar to a table view, `MKAnnotationView` provides an optional accessory view displayed on the right side of a standard callout bubble. Once you create the accessory view, the `calloutAccessoryControlTapped` method of your map view's delegate will be called when a user taps the accessory view.

Now that you should have a better idea of the implementation, let's continue to develop the app. First open the `RouteTableViewController.swift` file and import MapKit:

```
import MapKit
```

Then declare an instance variable:

```
var routeSteps = [MKRouteStep]()
```

This variable is used for storing an array of `MKRouteStep` object of a selected route. Replace the method of table view data source with the following:

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

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

override func tableView(tableView: UITableView, cellForRowAt indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath)

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

    return cell
}
```

The above code is very straightforward. We simply display the written instructions of the route steps in the table view. Next, open `MapViewController.swift`. We're going to add a few lines of

code to handle the touch of an annotation.

At the very beginning of the class, declare a new variable to store the current route:

```
var currentRoute:MKRoute?
```

In the `viewForAnnotation` method, insert the following line of code before `return annotationView :`

```
annotationView?.rightCalloutAccessoryView = UIButton(type:  
UIButtonType.DetailDisclosure)
```

Here we add a detail disclosure button to the right side of an annotation. To handle a touch, we implement the `mapView(_:calloutAccessoryControlTapped:_)` method like this:

```
func mapView(mapView: MKMapView, annotationView view: MKAnnotationView,  
calloutAccessoryControlTapped control: UIControl) {  
    performSegueWithIdentifier("showSteps", sender: view)  
}
```

In iOS, you're allowed to trigger a segue programmatically by calling the `performSegueWithIdentifier` method. Earlier we created a segue between the map view controller and the navigation controller and set the segue's identifier to `showSteps`. The app will navigate to the Steps table view controller when the above `performSegueWithIdentifier` method is called.

Lastly, we have to pass the current route steps to the `RouteTableViewController` class.

In the body of the `calculateDirectionsWithCompletionHandler` closure, insert a line of code to update the current route:

```
self.currentRoute = route
```

It should be placed right before calling the `removeOverlays` method. The closure should look like this after the modification:

```
directions.calculateDirectionsWithCompletionHandler { (routeResponse,  
routeError) -> Void in  
  
    guard let routeResponse = routeResponse else {
```

```

    if let routeError = routeError {
        print("Error: \(routeError)")
    }

    return
}

let route = routeResponse.routes[0]
self.currentRoute = route
self.mapView.removeOverlays(self.mapView.overlays)
self.mapView.addOverlay(route.polyline, level: MKOverlayLevel.AboveRoads)

let rect = route.polyline.boundingMapRect
self.mapView.setRegion(MKCoordinateRegionForMapRect(rect), animated: true)
}

```

To pass the route steps to `RouteTableViewController`, implement the `prepareForSegue` method like this:

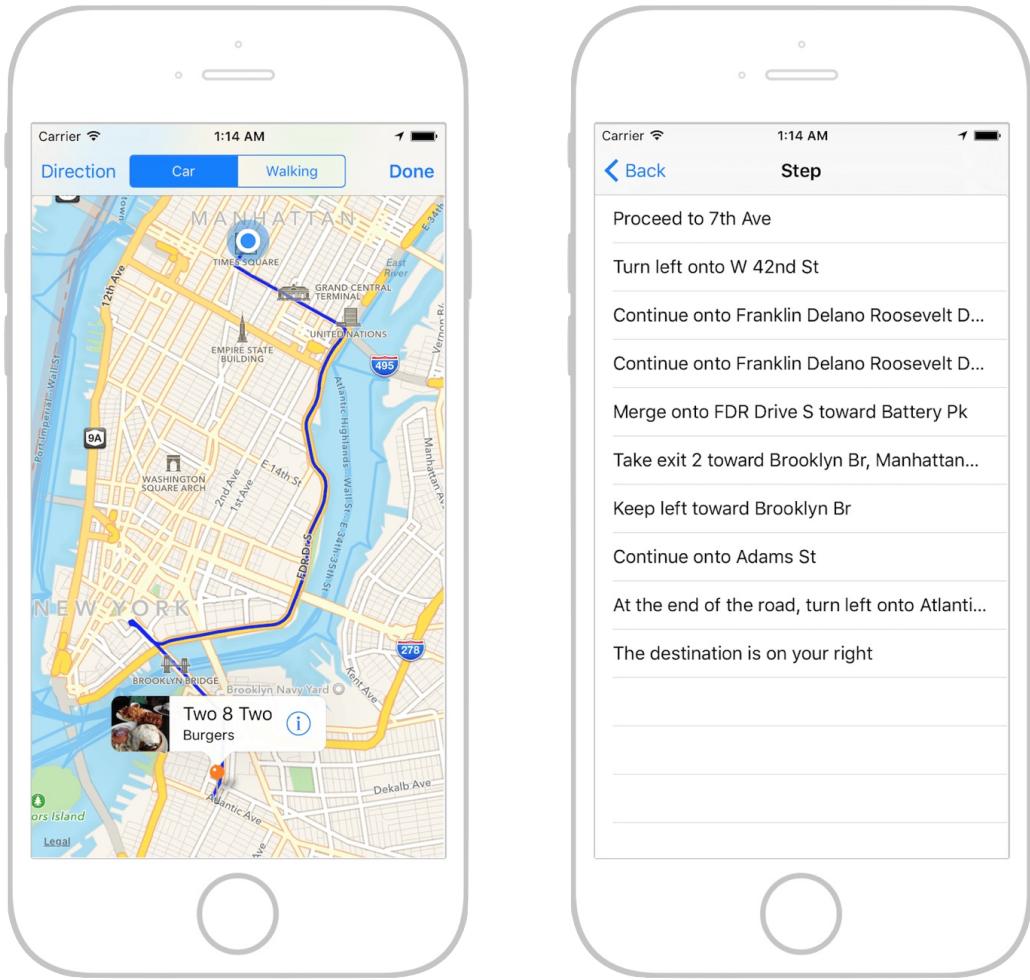
```

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.
    if segue.identifier == "showSteps" {
        let routeTableViewController = segue.destinationViewController as!
RouteTableViewController
        if let steps = currentRoute?.steps {
            routeTableViewController.routeSteps = steps
        }
    }
}

```

The above code snippet should be very familiar to you. We first get the destination controller, which is the `RouteTableViewController` object, and then pass it the route steps to the controller.

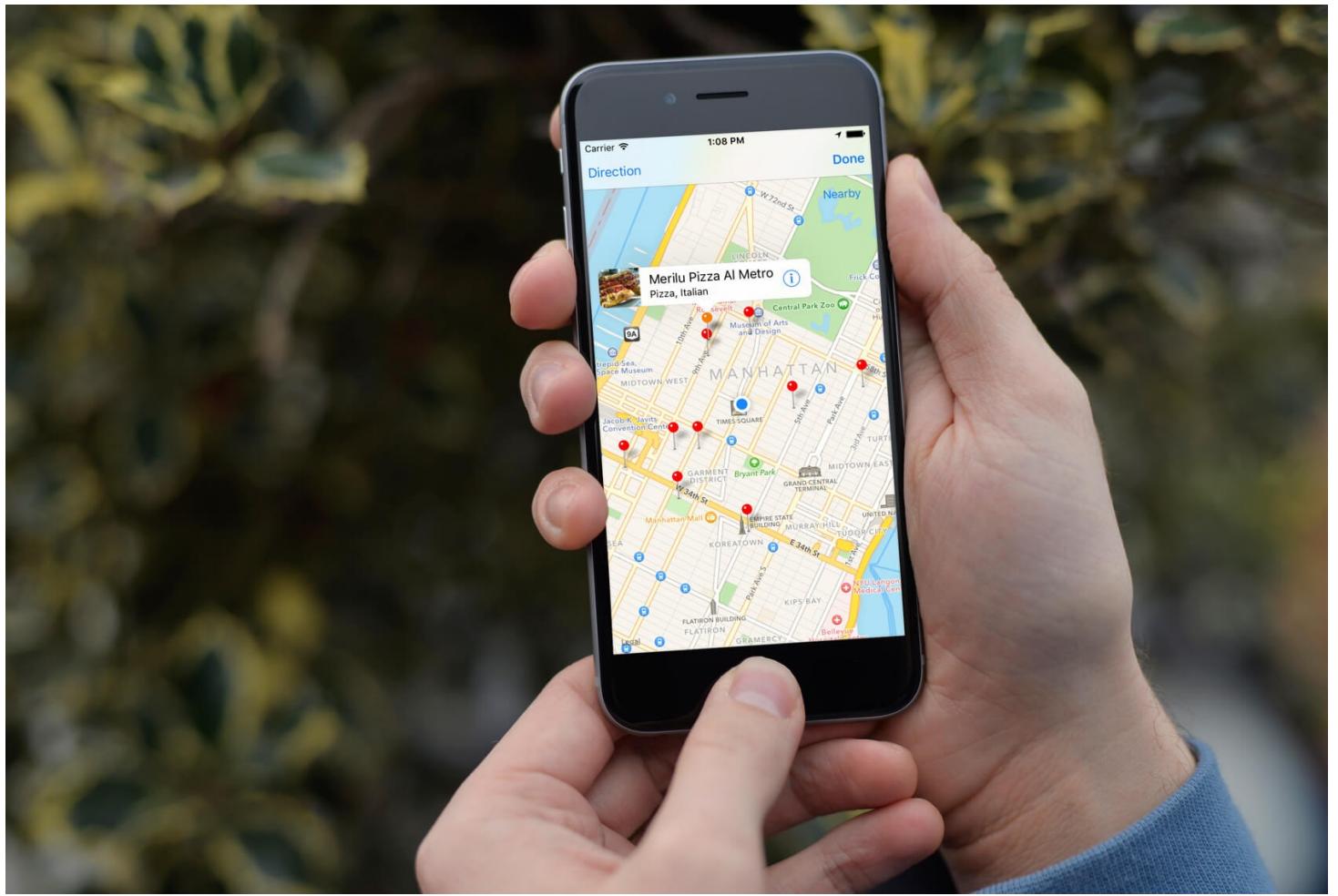
The app is now ready to run. When you tap the annotation on the map, the app shows you a list of steps to follow.



For reference, you can download the complete Xcode project from
<https://www.dropbox.com/s/kzkyokepakkzxoi/MapKitDirectionDemo.zip?dl=0>.

Chapter 9

Search for Nearby Points of Interest Using Local Search



Introduced in iOS 7, the new Search API (i.e `MKLocalSearch`) allows iOS developers to search for points of interest and display them on maps. App developers can use this API to perform searches for locations, which can be name, address, or type, such as coffee or pizza.

The use of `MKLocalSearch` is very similar to the `MKDrections` API covered in the previous chapter. You'll first need to create an `MKLocalSearchRequest` object that bundles your search query. You can also specify the map region to narrow down the search result. You then use the configured object to initialize an `MKLocalSearch` object and perform the search.

The search is performed remotely in an asynchronous way. Once Apple returns the search result (as an `MKLocalSearchResponse` object) to your app, the complete handler will be executed. In general, you'll need to parse the response object and display the search results on the map.

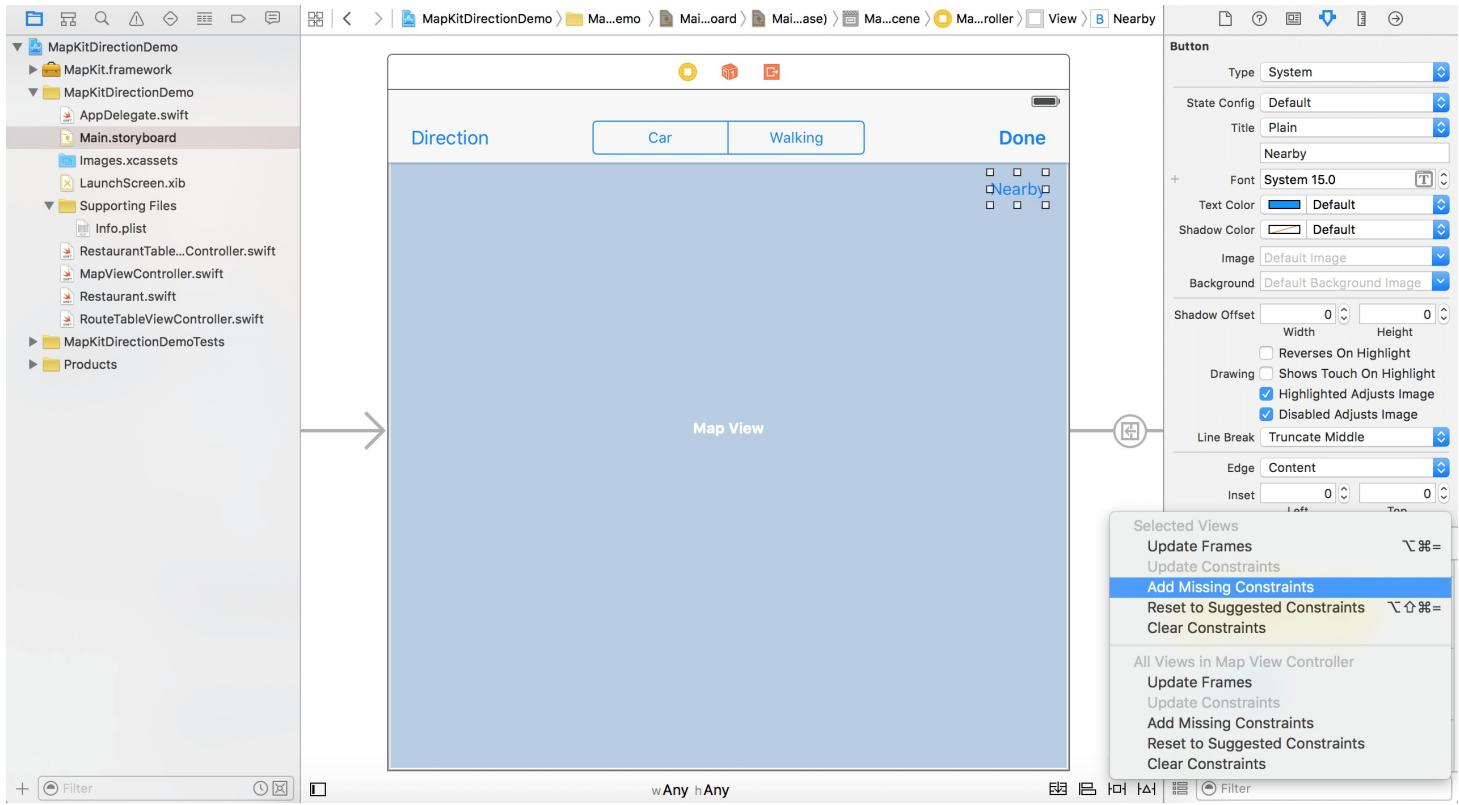
Local Search Demo App

There is no better way to understand local search than working on a demo project. Again we will not start from scratch but build on top of the previous project to add a *Nearby* feature. When you tap the *Nearby* button, the app searches for nearby restaurants and pins the places on the map.

To start with, first download the Xcode project template from
<https://www.dropbox.com/s/omakovehpe4a9ag/MapKitLocalSearchTemplate.zip?dl=0>.

Adding a Nearby Button in Storyboard

Okay, let's get started. First go to `Main.storyboard` and add a button item to the map view. Name the button *Nearby*. Click the *Issues* button in the layout bar and select *Add Missing Constraints*.



Search Nearby Restaurants and Adding annotations

Now open the `MapViewController.swift` file, we will create an action method called `showNearby` for the `Nearby` button. Insert the following code snippet in the class:

```
@IBAction func showNearby(sender: AnyObject) {
    let searchRequest = MKLocalSearchRequest()
    searchRequest.naturalLanguageQuery = restaurant.category
    searchRequest.region = mapView.region

    let localSearch = MKLocalSearch(request: searchRequest)
    localSearch.startWithCompletionHandler { (response, error) -> Void in
        guard let response = response else {
            if let error = error {
                print(error)
            }
        }
        return
    }

    let mapItems = response.mapItems
    var nearbyAnnotations:[MKAnnotation] = []
    if mapItems.count > 0 {
        for item in mapItems {
            // Add annotation
        }
    }
}
```

```

        let annotation = MKPointAnnotation()
        annotation.title = item.name
        annotation.subtitle = item.phoneNumber
        if let location = item.placemark.location {
            annotation.coordinate = location.coordinate
        }
        nearbyAnnotations.append(annotation)
    }
}

self.mapView.showAnnotations(nearbyAnnotations, animated: true)
}
}

```

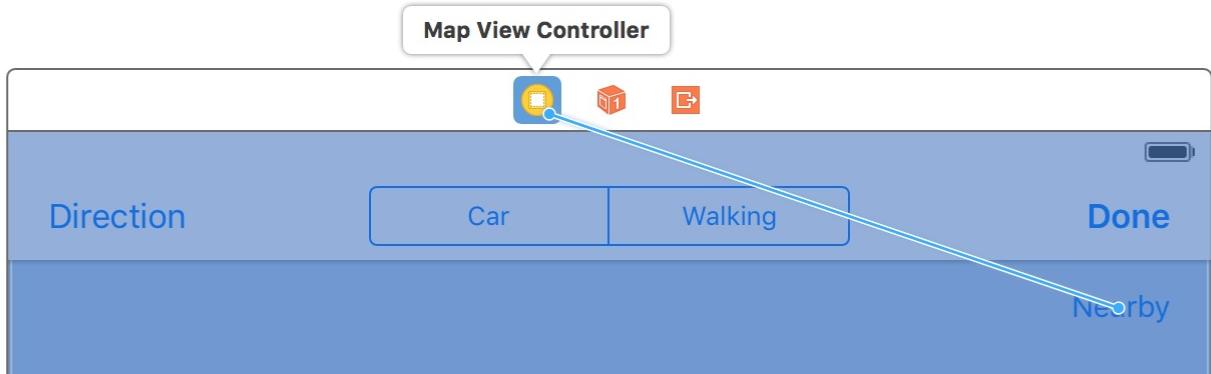
To perform a local search, here are the two things you need to do:

- Specify your search parameters in an `MKLocalSearchRequest` object. You are allowed to specify the search criteria in natural language by using the `naturalLanguageQuery` parameter. For example, if you want to search for a nearby cafe, you can specify `cafe` in the search parameter. Since we want to search for similar types of restaurants, we specify `restaurant.category` in the query.
- Initiate the local search by creating an `MKLocalSearch` object with the search parameters. An `MKLocalSearch` object is used to initiate a map-based search operation and delivers the results back to your app asynchronously.

In the `showNearby` method, we lookup the nearby restaurants that are of the same type (e.g. Italian). Furthermore, we specify the current region of the map view as the search region.

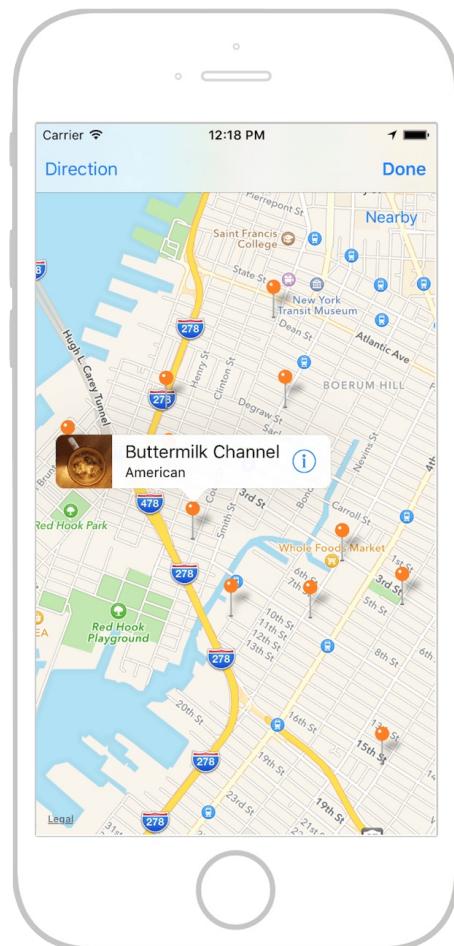
We then initialize the search by creating the `MKLocalSearch` object and invoking the `startWithCompletionHandler:` method. When the search completes, the closure will be called and the results are delivered as an array of `MKMapItem`. In the body of the closure, we loop through the items (i.e. nearby restaurants) and highlight them on the map using annotations.

Okay, you're almost ready to test the app. Just go to the storyboard and connect the *Nearby* button with the `showNearby` method. Simply control-drag from the *Nearby* button to the view controller icon in the scene dock and select the `showNearby:` action method.



Testing the Demo App

Now hit the Run button to compile and run your app. Select a restaurant to bring up the map view. Tap the *Nearby* button and the app should show you the nearby restaurants.



Cool, right? With just a few lines of code, you took your Map app to the next level. If you're going to embed a map within your app, try to explore the local search API.

Modifying the Pin Color for the Nearby Restaurants

With the current implementation, all the pins are in the same color. If you tap a nearby restaurant, the callout bubble still displays the current restaurant image. How can we make it better? You probably want to differentiate the nearby restaurants with the current selection by using a different pin color. To do so, you can modify the `mapView(_:viewForAnnotation:_)` method, which is responsible for the appearance of an annotation.

```
func mapView(mapView: MKMapView, viewForAnnotation annotation: MKAnnotation) ->
MKAnnotationView? {
    let identifier = "MyPin"

    if annotation is MKUserLocation {
        return nil
    }

    // Reuse the annotation if possible
    var annotationView: MKPinAnnotationView? =
mapView.dequeueReusableCell(withIdentifier: identifier) as?
MKPinAnnotationView

    if annotationView == nil {
        annotationView = MKPinAnnotationView(annotation: annotation,
reuseIdentifier: identifier)
        annotationView?.canShowCallout = true
    }

    if let currentPlacemarkCoordinate = currentPlacemark?.location?.coordinate
{
        if currentPlacemarkCoordinate.latitude ==
annotation.coordinate.latitude &&
            currentPlacemarkCoordinate.longitude ==
annotation.coordinate.longitude {

            let leftIconView = UIImageView(frame: CGRect(x: 0, y: 0, width: 53, height: 53))
            leftIconView.image = UIImage(named: "restaurant.image")!
            annotationView?.leftCalloutAccessoryView = leftIconView

            // Pin color customization
            if #available(iOS 9.0, *) {
                annotationView?.pinTintColor = UIColor.orangeColor()
            }
        } else {
            // Pin color customization
            if #available(iOS 9.0, *) {
                annotationView?.pinTintColor = UIColor.redColor()
            }
        }
    }
}
```

```

        }
    }

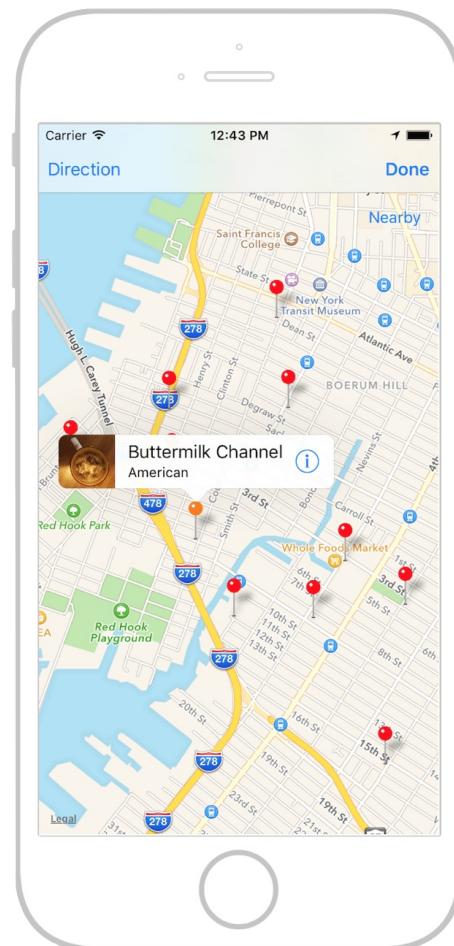
    annotationView?.rightCalloutAccessoryView = UIButton(type:
UIButtonType.DetailDisclosure)

    return annotationView
}

```

We have modified the method a bit by adding a condition block. The code checks if the annotation, which is about to display, is the same as the current placemark. In other words, we check if the coordinate of the annotation is equal to the coordinate of the selected restaurant. If the result is positive, we display the restaurant image in the callout bubble and keep the pin color to orange. Otherwise, for those nearby restaurants, we set the pin color to red.

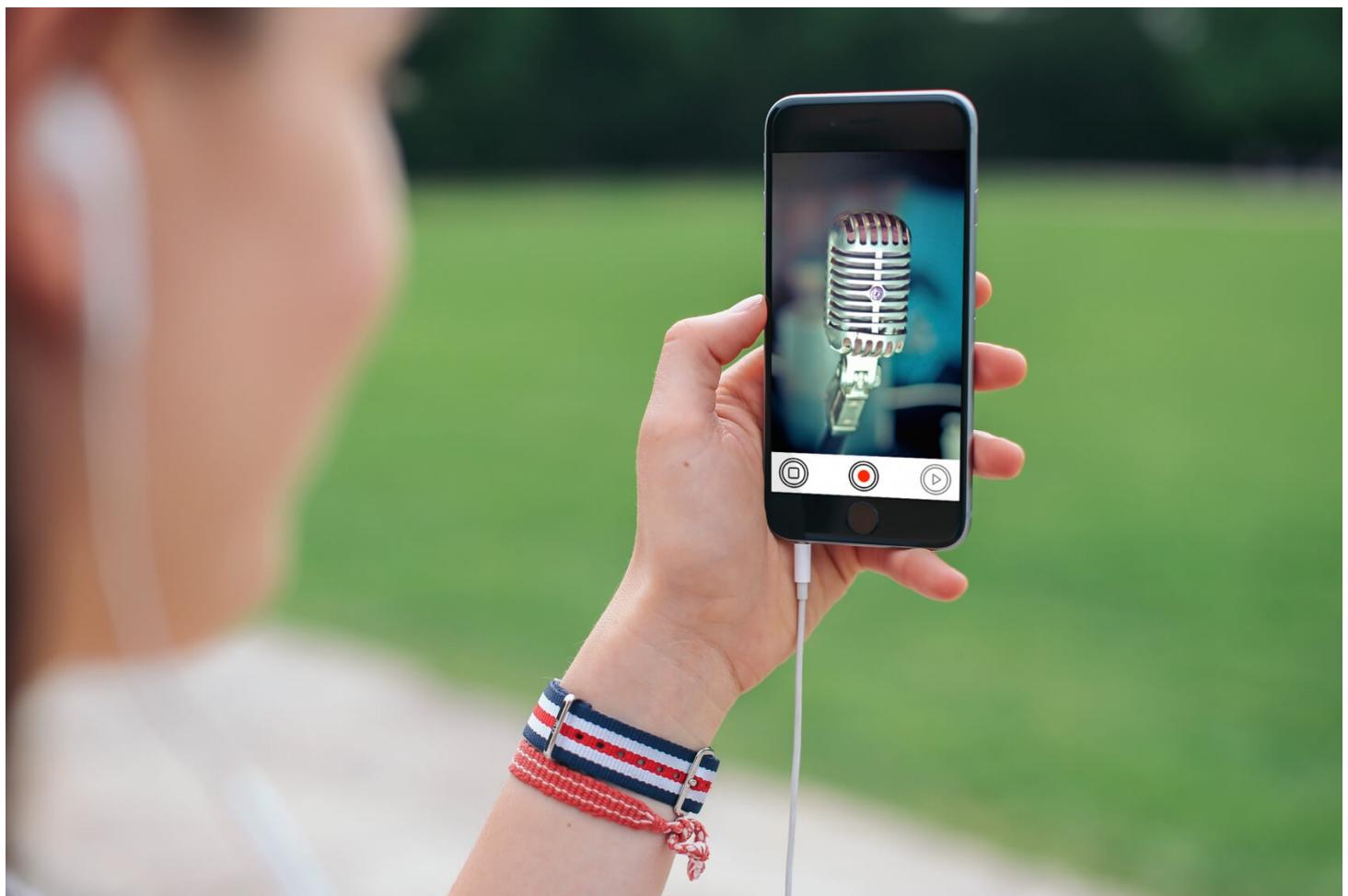
Now run the project and try out the Nearby feature again. The pin color of the nearby restaurants is now in red.



For reference, you can download the complete Xcode project from
<https://www.dropbox.com/s/la3872e8jffou9q/MapKitLocalSearch.zip?dl=0>.

Chapter 10

Audio Recording and Playback



The iOS SDK provides various frameworks to let you work with sounds in your app. One of the frameworks that you can use to play and record audio files is the **AV Foundation framework**. In this chapter, I will walk you through the basics of the framework and show you how to manage audio playback and recording.

The AV Foundation provides essential APIs for developers to deal with audio on iOS. In this demo, we mainly use these two classes of the framework:

- **AVAudioPlayer** – think of it as an audio player for playing sound files. By using the player, you can play sounds of any duration and in one of the many audio formats available in

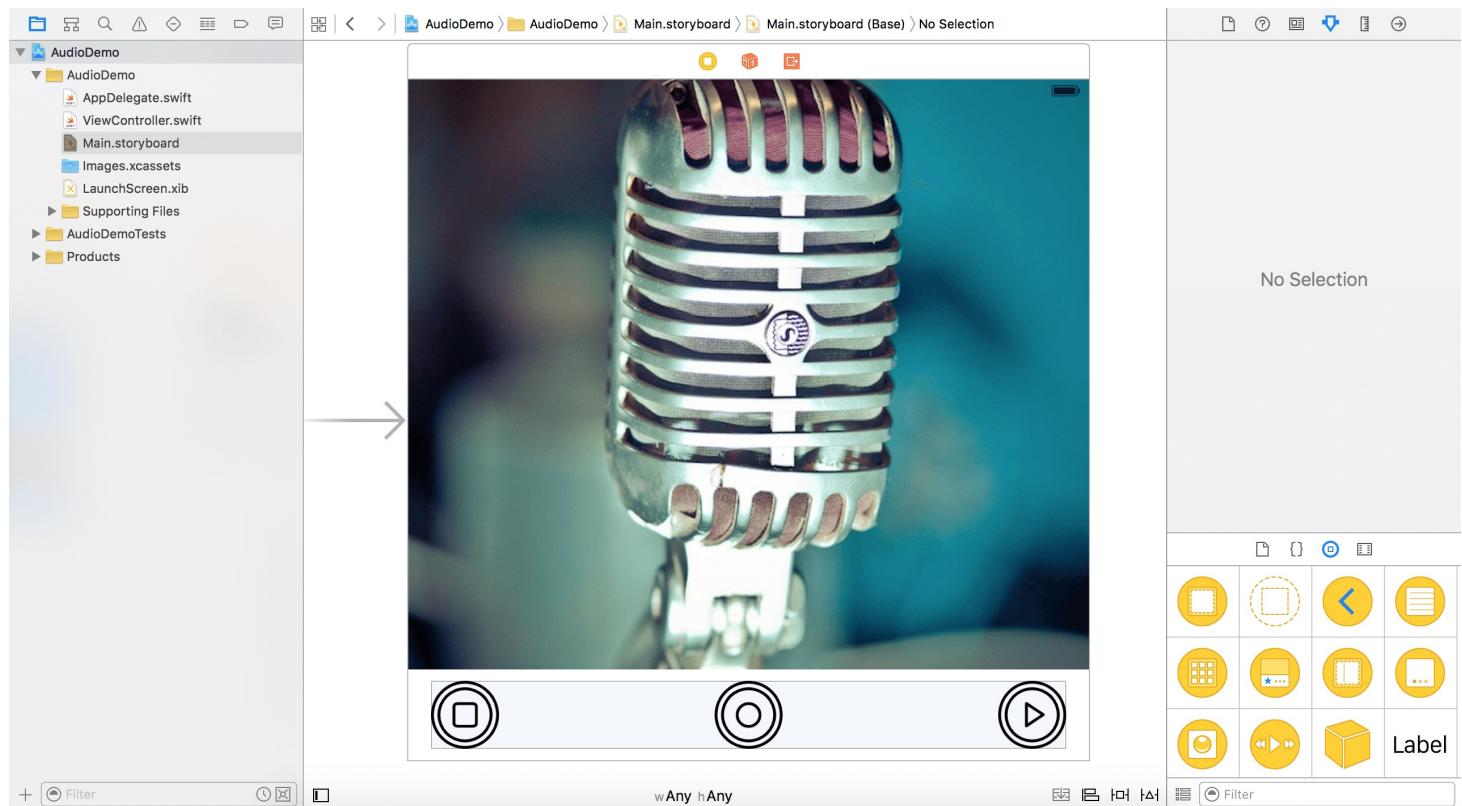
iOS.

- **AVAudioRecorder** – an audio recorder for recording audio.

A Simple Demo App

To understand how to use the API, we will build a simple audio app that allows users to record and play audio. Our primary focus is to demonstrate the AVFoundation framework so the app's user interface will be very simple.

First, create an app using the *Single View Application* template and name it *AudioDemo*. To free you from setting up the user interface and custom classes, you can download the project template from <https://www.dropbox.com/s/4hwmuk49mgjsdma/audiodemotemplate.zip?dl=0>. I've created the storyboard and custom classes for you. The interface contains three buttons: *Record*, *Stop*, and *Play*. The buttons are also linked to the corresponding action methods in the `ViewController` class.



Credit: The photo used in the demo app is courtesy of [Eric May](#).

Audio Recording using AVAudioRecorder

The `AVAudioRecorder` class of the AV Foundation framework allows your app to provide audio recording capability. In iOS, the audio being recorded comes from the built-in microphone or headset microphone of the iOS device. These devices include the iPhone, iPad or iPod touch.

First, let's take a look at how we can use the `AVAudioRecorder` class to record audio. Like most of the APIs in the SDK, `AVAudioRecorder` makes use of the delegate pattern. You can implement a delegate object for an audio recorder to respond to audio interruptions and to the completion of a recording. The delegate of an `AVAudioRecorder` object must adopt the `AVAudioRecorderDelegate` protocol. For the AudioDemo app, the `ViewController` class serves as the delegate object. Therefore, we implement the `AVAudioRecorderDelegate` protocol in the class:

```
class ViewController: UIViewController, AVAudioRecorderDelegate
```

Because the delegate is defined in the AV Foundation framework, you have to import the AVFoundation:

```
import AVFoundation
```

Next, declare an instance variable of the type `AVAudioRecorder` and an instance variable of the type `AVAudioPlayer` in `ViewController.swift`:

```
var audioRecorder:AVAudioRecorder?  
var audioPlayer:AVAudioPlayer?
```

Let's focus on `AVAudioRecorder` first. We will use the `audioPlayer` variable later. The `AVAudioRecorder` class provides an easy way to record sounds in your app. To use the recorder, you have to prepare a few things:

- Specify a sound file URL
- Set up an audio session
- Configure the audio recorder's initial state

We will do the setup in the `viewDidLoad` method. Simply update the method with the following code:

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

```

// Disable Stop/Play button when application launches
stopButton.enabled = false
playButton.enabled = false

// Get the document directory. If fails, just skip the rest of the code
guard let directoryURL =
    NSFileManager.defaultManager().URLsForDirectory(NSSearchPathDirectory.DocumentDi
    inDomains: NSSearchPathDomainMask.UserDomainMask).first else {

    let alertMessage = UIAlertController(title: "Error", message: "Failed
        to get the document directory for recording the audio. Please try again
        later.", preferredStyle: .Alert)
    alertMessage.addAction(UIAlertAction(title: "OK", style: .Default,
    handler: nil))
    presentViewController(alertMessage, animated: true, completion: nil)

    return
}

// Set the default audio file
let audioFileURL =
directoryURL.URLByAppendingPathComponent("MyAudioMemo.m4a")

// Setup audio session
let audioSession = AVAudioSession.sharedInstance()

do {
    try audioSession.setCategory(AVAudioSessionCategoryPlayAndRecord,
withOptions: AVAudioSessionCategoryOptions.DefaultToSpeaker)

    // Define the recorder setting
    let recorderSetting: [String: AnyObject] = [
        AVFormatIDKey: Int(kAudioFormatMPEG4AAC),
        AVSampleRateKey: 44100.0,
        AVNumberOfChannelsKey: 2,
        AVEncoderAudioQualityKey: AVAudioQuality.High.rawValue
    ]

    // Initiate and prepare the recorder
    audioRecorder = try AVAudioRecorder(URL: audioFileURL, settings:
recorderSetting)
    audioRecorder?.delegate = self
    audioRecorder?.meteringEnabled = true
    audioRecorder?.prepareToRecord()

} catch {
    print(error)
}

```

```
}
```

In the above code, we first disable both the *Stop* and *Play* buttons when the app is launched. We then define the URL of the sound file for saving the recording - you can use `NSFileManager` to interact with the file system. We simply ask for the document directory in the user's home directory (`NSSearchPathDomainMask.UserDomainMask`). After retrieving the file path, we create the audio file URL and name the audio file `MyAudioMemo.m4a`. In case of failure, the app shows an alert message to the users.

Next, we configure the audio session. So what's the audio session for? iOS handles audio behavior of an app by using audio sessions. In brief, it acts as a middle man between your app and the system's media service. Through the shared audio session object, you tell the system how you're going to use audio in your app. The audio session provides answers to questions like: *Should the system disable the existing music being played by the Music app? Should your app be allowed to record audio and music playback?*

Since the `AudioDemo` app is used for audio recording and playback, we will set the audio session category to `AVAudioSessionCategoryPlayAndRecord` which enables both audio input and output, and uses the built-in speaker for recording and playback.

The `AVAudioRecorder` uses dictionary-based settings for the configuration. In the code snippet, we use the option keys to configure the audio data format, sample rate, number of channels and audio quality. After defining the audio settings, we initialize an `AVAudioRecorder` object and set the delegate to itself. Lastly, we call the `prepareToRecord` method to create the audio file and get prepared for recording. Note that the recording has not yet started; the recording will not begin until the `record` method is called.

As you may notice, we've put a `try` keyword in front of the `audioSession.setCategory` call. In Swift 2, Apple has changed some of the APIs in favour of the *do-try-catch* error handling model. Prior to Swift 2, it lacked a proper error handling model. When calling a method that may cause a failure, you normally pass it with an `NSError` object (as a pointer) like this:

```
audioSession.setCategory(AVAudioSessionCategoryPlayAndRecord, withOptions:  
AVAudioSessionCategoryOptions.DefaultToSpeaker, error: &error)
```

If there is an error, the object will be set to the corresponding error. You then check if the error object is `nil` or not and respond to the error accordingly. This is how you handle errors in Swift 1.2. Starting from Swift 2, it comes with an exception-like model. The same method call no longer takes in an error parameter. Instead, it throws an error for any failures. To invoke a method call that throws an error, you have to enclose it in a do-catch block like this:

```
do {
    try audioSession.setCategory(AVAudioSessionCategoryPlayAndRecord,
withOptions: AVAudioSessionCategoryOptions.DefaultToSpeaker)

} catch {
    print(error)
}
```

In the `do` clause, you can call the method by putting a `try` keyword in front of it. If there is an error, it will be caught and the `catch` block will be executed. By default, the error is bundled in an `error` object.

Implementing the Record Button

We've completed the recording preparation. Let's move on to the implementation of the action method of the *Record* button. Before we dive into the code, let me explain how the *Record* button works. When a user taps the *Record* button, the app will start recording. The *Record* button will be changed to a *Recording* button, indicated by a red dot. If the user taps the *Record* button again, the app will pause the audio recording until the button is tapped again. The audio recording will only be stopped when the user taps the *Stop* button.

Now update the record method like this:

```
@IBAction func record(sender: AnyObject) {
    // Stop the audio player before recording
    if let player = audioPlayer {
        if player.playing {
            player.stop()
            playButton.setImage(UIImage(named: "play"), forState:
UIControlState.Normal)
            playButton.selected = false
        }
    }

    if let recorder = audioRecorder {
```

```

if !recorder.recording {
    let audioSession = AVAudioSession.sharedInstance()

    do {
        try audioSession.setActive(true)

        // Start recording
        recorder.record()
        recordButton.setImage(UIImage(named: "recording"), forState:
UIControlState.Selected)
        recordButton.selected = true
    } catch {
        print(error)
    }

} else {
    // Pause recording
    recorder.pause()
    recordButton.setImage(UIImage(named: "pause"), forState:
UIControlState.Normal)
    recordButton.selected = false
}
}

stopButton.enabled = true
playButton.enabled = false
}

```

In the above code, we first check whether the audio player is playing. You probably don't want to play an audio file while you're recording, so we simply stop any audio playback by using the `stop` method.

If `audioRecorder` is not in recording mode, the app activates the audio sessions and starts the recording by calling the `record` method of the audio recorder. To make the recorder work, remember to set audio session to `active`. Otherwise, the audio recording will not be activated.

```
audioSession.setActive(true)
```

Once the recording starts, we change the *Record* button to a recording button (a different image). In case the user taps the *Record* button while the recorder is in recording mode, we pause it by calling the `pause` method.

In general, you can use the following methods of `AVAudioRecorder` class to control the recording:

- Record – start/resume a recording
- Pause – pause a recording
- Stop – stop a recording

Implementing the Stop Button

The `stop` action method is called when the user taps the *Stop* button. This method is pretty simple. All we need to do is reset the *Record* and *Play* buttons to the normal state. Furthermore, we will call the `stop` method of the `AVAudioRecorder` object to stop the recording, and deactivate the audio session. Update the `stop` action method with the following code:

```
@IBAction func stop(sender: AnyObject) {
    recordButton.setImage(UIImage(named: "record"), forState:
    UIControlState.Normal)
    recordButton.selected = false
    playButton.setImage(UIImage(named: "play"), forState:
    UIControlState.Normal)
    playButton.selected = false

    stopButton.enabled = false
    playButton.enabled = true

    audioRecorder?.stop()

    let audioSession = AVAudioSession.sharedInstance()

    do {
        try audioSession.setActive(false)
    } catch {
        print(error)
    }
}
```

Implementing the AVAudioRecorderDelegate Protocol

You can make use of the `AVAudioRecorderDelegate` protocol to handle audio interruptions (say, a phone call during audio recording) as well as to complete the recording. In the example, `viewController` is the delegate. The methods defined in the `AVAudioRecorderDelegate` protocol are optional. For demo purposes, we'll only implement the `audioRecorderDidFinishRecording`

method to handle the completion of recording. Add the following code to the `ViewController.swift` file:

```
func audioRecorderDidFinishRecording(recorder: AVAudioRecorder, successfully
flag: Bool) {
    if flag {
        let alertMessage = UIAlertController(title: "Finish Recording",
message: "Successfully recorded the audio!", preferredStyle: .Alert)
        alertMessage.addAction(UIAlertAction(title: "OK", style: .Default,
handler: nil))
        presentViewController(alertMessage, animated: true, completion: nil)
    }
}
```

After the recording completes, the app displays an alert dialog with a success message.

Playing Audio Using AVAudioPlayer

Finally, we come to the implementation of the *Play* button. `AVAudioPlayer` is the class which is responsible for audio playback. Typically, there are a few things you have to implement in order to use `AVAudioPlayer`:

- Initialize the audio player and assign a sound file to it. In this case, it's the audio file of the recording (i.e. `MyAudioMemo.m4a`). You can use the `URL` property of an `AVAudioRecorder` object to get the file URL of the recording.
- Designate an audio player delegate object, which handles interruptions as well as the playback-completed event.
- Call the `play` method to play the sound file.

In the `ViewController` class, edit the play action method using the following code:

```
@IBAction func play(sender: AnyObject) {
    if let recorder = audioRecorder {
        if !recorder.recording {
            audioPlayer = AVAudioPlayer(contentsOfURL: recorder.url, error:
nil)
            audioPlayer?.delegate = self
            audioPlayer?.play()
            playButton.setImage(UIImage(named: "playing"), forState:
UIControlState.Selected)
            playButton.selected = true
    }
}
```

```
    }
}
```

The above code is very straightforward. We first initialize an instance of `AVAudioPlayer` with the URL of an audio file (`recorder.url`). To play the audio, you just need to call the `play` method. In the `viewDidLoad` method, we configured the audio session to use the built-in speaker. Thus, the player will use the speaker for audio playback.

Implementing the AVAudioPlayerDelegate Protocol

The delegate of an `AVAudioPlayer` object must adopt the `AVAudioPlayerDelegate` protocol. Again, `viewController` is set as the delegate, so update the `viewController` declaration to adopt the protocol:

```
class ViewController: UIViewController, AVAudioRecorderDelegate,  
AVAudioPlayerDelegate
```

The delegate allows you to handle interruptions, audio decoding errors, and update the user interface when an audio file finishes playing. All methods in the `AVAudioPlayerDelegate` protocol are optional, however. To demonstrate how it works, we'll implement the `audioPlayerDidFinishPlaying` method to display an alert message after the completion of audio playback. For usage of the other methods, you can refer to the [official documentation of AVAudioPlayerDelegate protocol](#).

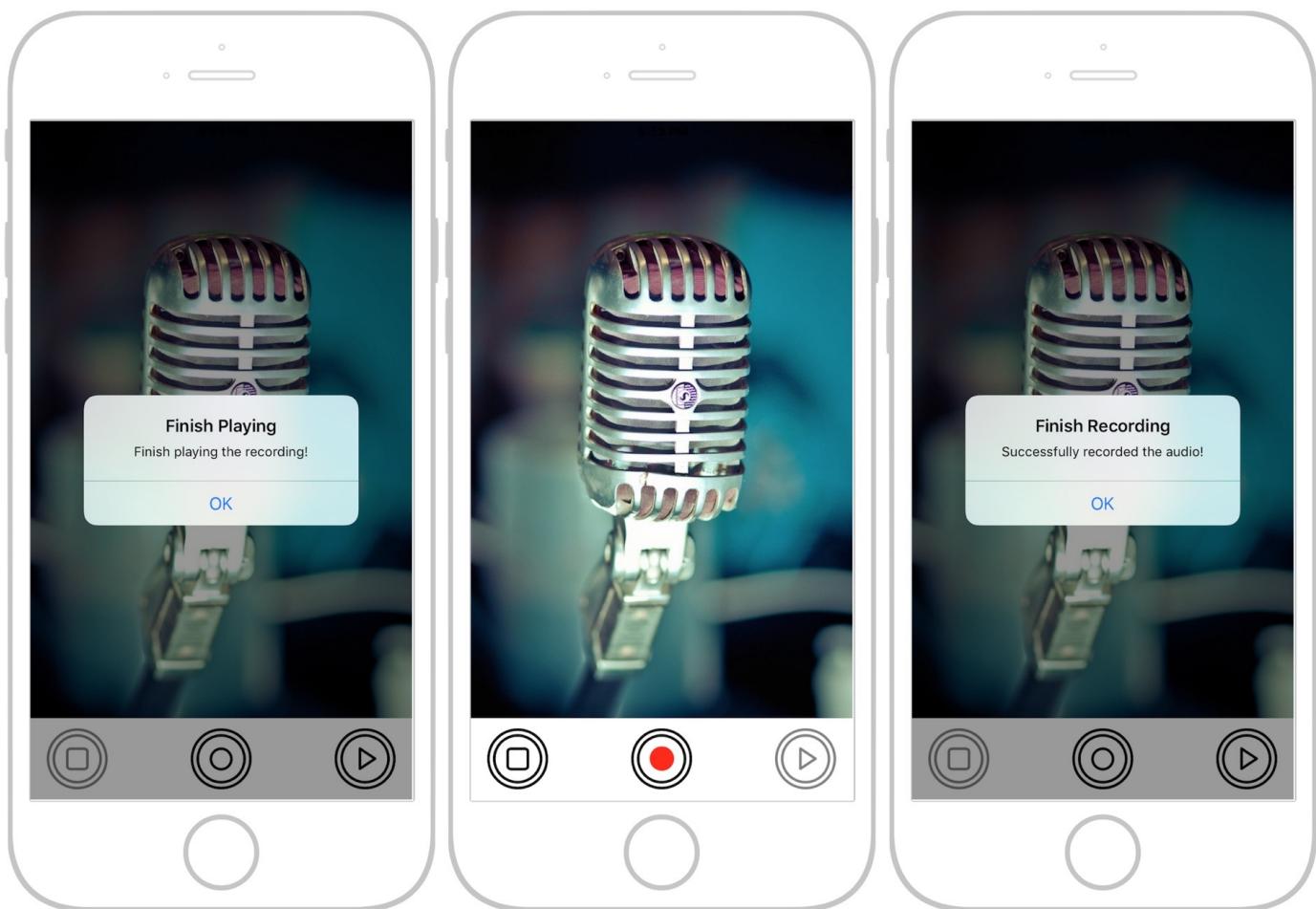
Insert the following code in `ViewController.swift`:

```
func audioPlayerDidFinishPlaying(player: AVAudioPlayer, successfully flag:  
Bool) {  
    playButton.setImage(UIImage(named: "play"), forState:  
 UIControlState.Normal)  
    playButton.selected = false  
  
    let alertMessage = UIAlertController(title: "Finish Playing", message:  
    "Finish playing the recording!", preferredStyle: .Alert)  
    alertMessage.addAction(UIAlertAction(title: "OK", style: .Default, handler:  
    nil))  
    presentViewController(alertMessage, animated: true, completion: nil)  
}
```

Compile and Run Your App

You can test audio recording and playback using a real device or the simulator. If you test the app using an actual device (e.g. iPhone), the audio being recorded comes from the device's built-in microphone. On the other hand, if you test the app by using the simulator, the audio comes from the system's default audio input device as set in the System Preferences.

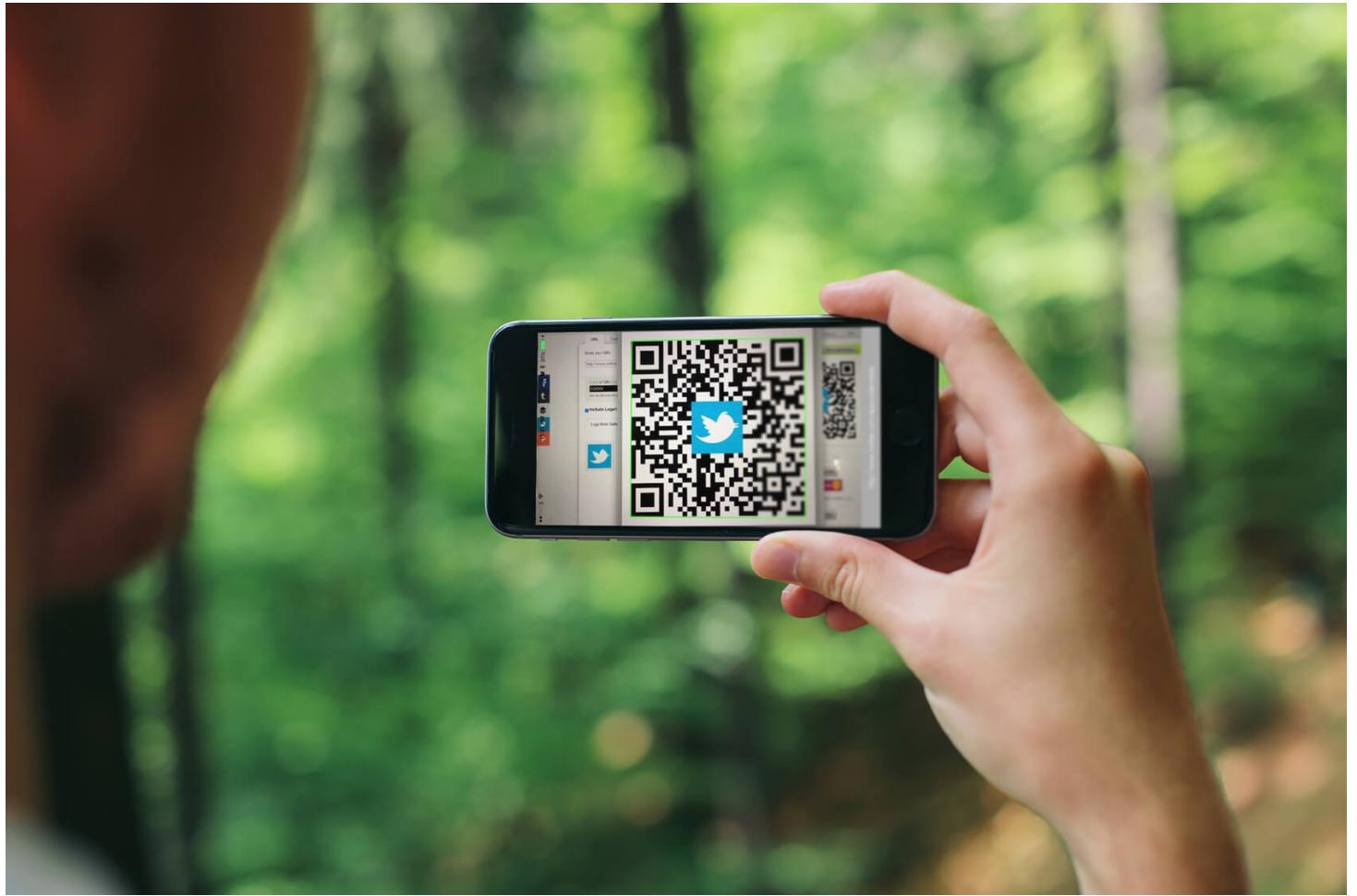
Go ahead to compile and run the app! Tap the Record button to start recording. Say something, tap the Stop button and then select the Play button to playback the recording.



For your reference, you can download the Xcode project from
<https://www.dropbox.com/s/tz223o7whsemyn/audiodemo.zip?dl=0>.

Chapter 11

Scan QR Code Using AVFoundation Framework



So, what's QR code? I believe most of you know what a QR code is. In case you haven't heard of it, just take a look at the above image - that's a QR code.

QR (short for Quick Response) code is a kind of two-dimensional bar code developed by Denso. Originally designed for tracking parts in manufacturing, QR code has gained popularity in consumer space in recent years as a way to encode the URL of a landing page or marketing information. Unlike the basic barcode that you're familiar with, a QR code contains information in both the horizontal and vertical direction. Thus this contributes to its capability of storing a larger amount of data in both numeric and letter form. I don't want to go into the

technical details of the QR code here. If you're interested in learning more, you can check out the official website of QR code.

With the rising prevalence of iPhone and Android phones, the use of QR codes has been increased dramatically. In some countries QR codes can be found nearly everywhere. They appear in magazines, newspapers, advertisements, billboards and even name cards. As an iOS developer, you may wonder how you can empower your app to read a QR code. Prior to iOS 7, you had to rely on third party libraries to implement the scanning feature. Now, you can use the built-in AVFoundation framework to discover and read barcodes in real-time.

Creating an app for scanning and translating QR codes has never been so easy.

Quick tip: You can generate your own QR code. Simply go to <http://www.qrcode-monkey.com> to create one.

Creating a QR Code Reader App

The demo app that we're going to build is fairly simple and straightforward. Before we proceed to build the demo app, it's important to understand that any barcode scanning in iOS, including QR code scanning, is totally based on video capturing. That's why the barcode scanning feature is added in the AVFoundation framework. Keep this point in mind, as it'll help you understand the entire chapter.

So, how does the demo app work?

Take a look at the screenshot below. This is how the app UI looks. The app works pretty much like a video capturing app but without the recording feature. When the app is launched, it takes advantage of the iPhone's rear camera to spot the QR code and recognizes it automatically. The decoded information (e.g. an URL) is displayed right at the bottom of the screen.



It's that simple.

To build the app, you can start by downloading the project template from <https://www.dropbox.com/s/osojn1mia56soxg/QRReaderDemoTemplate.zip?dl=0>. I have pre-built the storyboard and linked up the message label for you.

Okay. Let's get started and develop the QR scanning feature in the app.

Import AVFoundation Framework

I have created the user interface of the app in the project template. The label in the UI will be used to display the decoded information of the QR code and it is associated with the `messageLabel` property of the `ViewController` class.

As I mentioned earlier, we rely on the `AVFoundation` framework to implement the QR code scanning feature. First, open the `ViewController.swift` file and import the framework:

```
import AVFoundation
```

Later, we need to implement the `AVCaptureMetadataOutputObjectsDelegate` protocol. We'll talk about that in a while. For now, update the following line of code:

```
class ViewController: UIViewController, AVCaptureMetadataOutputObjectsDelegate
```

Before moving on, declare the following variables in the `ViewController` class. We'll talk about them one by one later.

```
var captureSession:AVCaptureSession?  
var videoPreviewLayer:AVCaptureVideoPreviewLayer?  
var qrCodeFrameView:UIView?
```

Implementing Video Capture

As mentioned in the earlier section, QR code reading is totally based on video capturing. To perform a real-time capture, all we need to do is instantiate an `AVCaptureSession` object with the input set to the appropriate `AVCaptureDevice` for video capturing. Insert the following code in the `viewDidLoad` method of the `ViewController` class:

```
// Get an instance of the AVCaptureDevice class to initialize a device object  
and provide the video  
// as the media type parameter.  
let captureDevice =  
AVCaptureDevice.defaultDeviceWithMediaType(AMediaTypeVideo)  
  
do {  
    // Get an instance of the AVCaptureDeviceInput class using the previous  
device object.  
    let input = try AVCaptureDeviceInput(device: captureDevice)  
  
    // Initialize the captureSession object.  
    captureSession = AVCaptureSession()  
    // Set the input device on the capture session.  
    captureSession?.addInput(input)  
  
} catch {  
    // If any error occurs, simply print it out and don't continue any more.  
    print(error)  
    return  
}
```

An `AVCaptureDevice` object represents a physical capture device. You use a capture device to configure the properties of the underlying hardware. Since we are going to capture video data, we call the `defaultDeviceWithMediaType` method, passing it `AVMediaTypeVideo` to get the video capture device. To perform a real-time capture, we instantiate an `AVCaptureSession` object and add the input of the video capture device. The `AVCaptureSession` object is used to coordinate the flow of data from the video input device to our output.

In this case, the output of the session is set to an `AVCaptureMetadataOutput` object. The `AVCaptureMetadataOutput` class is the core part of QR code reading. This class, in combination with the `AVCaptureMetadataOutputObjectsDelegate` protocol, is used to intercept any metadata found in the input device (the QR code captured by the device's camera) and translate it to a human-readable format. Don't worry if something sounds weird or if you don't totally understand it right now - everything will become clear in a while. For now, continue to add the following lines of code in the `do` block of the `viewDidLoad` method:

```
// Initialize a AVCaptureMetadataOutput object and set it as the output device
// to the capture session.
let captureMetadataOutput = AVCaptureMetadataOutput()
captureSession?.addOutput(captureMetadataOutput)
```

Next, proceed to add the lines of code shown below. We set `self` as the delegate of the `captureMetadataOutput` object. This is the reason why the `QRReaderViewController` class adopts the `AVCaptureMetadataOutputObjectsDelegate` protocol. When new metadata objects are captured, they are forwarded to the delegate object for further processing. Here we also need to specify the dispatch queue on which to execute the delegate's methods. According to Apple's documentation, the queue must be a serial queue. So we simply use the `dispatch_get_main_queue()` function to get the default serial queue.

```
// Set delegate and use the default dispatch queue to execute the call back
captureMetadataOutput.setMetadataObjectsDelegate(self, queue:
dispatch_get_main_queue())
captureMetadataOutput.metadataObjectTypes = [AVMetadataObjectTypeQRCode]
```

The `metadataObjectTypes` property is also quite important; as this is the point where we tell the app what kind of metadata we are interested in. The `AVMetadataObjectTypeQRCode` clearly indicates our purpose.

Now that we have set and configured an `AVCaptureMetadataOutput` object, we need to display

the video captured by the device's camera on screen. This can be done using an `AVCaptureVideoPreviewLayer`, which actually is a `CALayer`. You use this preview layer in conjunction with an AV capture session to display video. The preview layer is added as a sublayer of the current view. Here is the code:

```
// Initialize the video preview layer and add it as a sublayer to the
viewPreview view's layer.
videoPreviewLayer = AVCaptureVideoPreviewLayer(session: captureSession)
videoPreviewLayer?.videoGravity = AVLayerVideoGravityResizeAspectFill
videoPreviewLayer?.frame = view.layer.bounds
view.layer.addSublayer(videoPreviewLayer!)
```

Finally, we start the video capture by calling the `startRunning` method of the capture session:

```
// Start video capture.
captureSession?.startRunning()
```

If you compile and run the app on a real iOS device, it should start capturing video when launched. However, at this point the message label is still hidden. You can fix it by adding the following line of code:

```
// Move the message label to the top view
view.bringSubviewToFront(messageLabel)
```

Re-run the app after making the changes. The message label *No QR code is detected* should now appear on screen.

Implementing QR Code Reading

As of now, the app looks pretty much like a video capture app. So how can it scan QR codes and translate the code into something meaningful? The app itself is already capable of detecting QR codes. We just aren't aware of that, however. Here is how we are going to tweak the app:

- When a QR code is detected, the app will highlight the code using a green box
- The QR code will be decoded and the decoded information will be displayed at the bottom of the screen

Initializing the Green Box

In order to highlight the QR code, we'll first create a `UIView` object and set its border to green. Add the following code in the `do` block of the `viewDidLoad` method:

```
// Initialize QR Code Frame to highlight the QR code
qrCodeFrameView = UIView()

if let qrCodeFrameView = qrCodeFrameView {
    qrCodeFrameView.layer.borderColor = UIColor.greenColor().CGColor
    qrCodeFrameView.layer.borderWidth = 2
    view.addSubview(qrCodeFrameView)
    view.bringSubviewToFront(qrCodeFrameView)
}
```

The `qrCodeFrameView` variable is invisible on screen because the size of the `UIView` object is set to zero by default. Later, when a QR code is detected, we will change its size and turn it into a green box.

Decoding the QR Code

As mentioned earlier, when the `AVCaptureMetadataOutput` object recognizes a QR code, the following delegate method of `AVCaptureMetadataOutputObjectsDelegate` will be called:

```
optional func captureOutput(captureOutput: AVCaptureOutput!,
didOutputMetadataObjects metadataObjects: [AnyObject]!, fromConnection
connection: AVCaptureConnection)
```

So far we haven't implemented the method; this is why the app can't translate the QR code. In order to capture the QR code and decode the information, we need to implement the method to perform additional processing on metadata objects. Here is the code:

```
func captureOutput(captureOutput: AVCaptureOutput!, didOutputMetadataObjects
metadataObjects: [AnyObject]!, fromConnection connection: AVCaptureConnection!) {
    // Check if the metadataObjects array is not nil and it contains at least
    one object.
    if metadataObjects == nil || metadataObjects.count == 0 {
        qrCodeFrameView?.frame = CGRectZero
        messageLabel.text = "No QR code is detected"
```

```

        return
    }

    // Get the metadata object.
    let metadataObj = metadataObjects[0] as!
AVMetadataMachineReadableCodeObject

    if metadataObj.type == AVMetadataObjectTypeQRCode {
        // If the found metadata is equal to the QR code metadata then update
        the status label's text and set the bounds
        let barCodeObject =
videoPreviewLayer?.transformedMetadataObjectForMetadataObject(metadataObj)
qrCodeFrameView?.frame = barCodeObject!.bounds

        if metadataObj.stringValue != nil {
            messageLabel.text = metadataObj.stringValue
        }
    }
}

```

The second parameter (i.e. `metadataObjects`) of the method is an array object, which contains all the metadata objects that have been read. The very first thing we need to do is make sure that this array is not `nil`, and it contains at least one object. Otherwise, we reset the size of `qrCodeFrameView` to zero and set `messageLabel` to its default message.

If a metadata object is found, we check to see if it is a QR code. If that's the case, we'll proceed to find the bounds of the QR code. These couple lines of code are used to set up the green box for highlighting the QR code. By calling the `transformedMetadataObjectForMetadataObject` method of `viewPreviewLayer`, the metadata object's visual properties are converted to layer coordinates. From that, we can find the bounds of the QR code for constructing the green box.

Lastly, we decode the QR code into human-readable information. This step should be fairly simple. The decoded information can be accessed by using the `stringValue` property of an `AVMetadataMachineReadableCode` object.

Now you're ready to go! Hit the Run button to compile and run the app on a real device. Once launched, point it to a QR code like the one on your left. The app immediately detects the code and decodes the information.



Your Exercise - Barcode Reader

The demo app is currently capable of scanning a QR code. Wouldn't it be great if you could turn it into a general barcode reader? Other than the QR code, the AVFoundation framework supports the following types of barcodes:

- UPC-E (AVMetadataObjectTypeUPCECode)
- Code 39 (AVMetadataObjectTypeCode39Code)
- Code 39 mod 43 (AVMetadataObjectTypeCode39Mod43Code)
- Code 93 (AVMetadataObjectTypeCode93Code)
- Code 128 (AVMetadataObjectTypeCode128Code)
- EAN-8 (AVMetadataObjectTypeEAN8Code)
- EAN-13 (AVMetadataObjectTypeEAN13Code)
- Aztec (AVMetadataObjectTypeAztecCode)
- PDF417 (AVMetadataObjectTypePDF417Code)



Your task is to tweak the existing Xcode project and enable the demo to scan other types of barcodes. You'll need to instruct `captureMetadataOutput` to identify an array of barcode types rather than just QR codes.

```
captureMetadataOutput.metadataObjectTypes = barcodeTypes
```

I'll leave it for you to figure out the solution. While I include the solution in the Xcode project below, I encourage you to try to sort out the problem on your own before moving on. It's gonna be fun and this is the best way to really understand how the code operates.

If you've given it your best shot and are still stumped, you can download the solution from <https://www.dropbox.com/s/c1fo4cr4ksp147d/QRReaderDemo.zip?dl=0>.

Chapter 12

Working with URL Schemes



The URL scheme is an interesting feature provided by the iOS SDK that allows developers to launch system apps and third-party apps through URLs. For example, let's say your app displays a phone number, and you want to make a call whenever a user taps that number. You can use a specific URL scheme to launch the built-in phone app and dial the number automatically. Similarly, you can use another URL scheme to launch the Message app for sending an SMS. Additionally, you can create a custom URL scheme for your own app so that other applications can launch your app via a URL. You'll see what I mean in a minute.

As usual, we will build an app to demonstrate the use of URL schemes. We will reuse the QR code reader app that was built in the previous chapter. If you haven't read the previous chapter, go back and read it before continuing on.

So far, the demo app is capable of decoding a QR code and displaying the decoded message on screen. In this chapter, we'll make it even better. When the QR code is decoded, the app will launch the corresponding app based on the type of the URL.

To start with, first download the QRCodeReader app from <https://www.dropbox.com/s/c1fo4cr4ksp147d/QRReaderDemo.zip?dl=0>. If you compile and run the app, you'll have a simple QR code reader app. Note that the app only works on a real iOS device.

Quick tip: How do you know the custom URL of third-party iOS apps? You can check out <http://handleopenurl.com> and http://wiki.akosma.com/IPhone_URL_Schemes. Both websites provide an extensive list of URLs of third-party iOS apps. Alternatively, you can refer to the official documentation of the third-party apps (e.g. Facebook, Whatsapp)

Sample QR Codes

Here I include some sample QR codes that you can use to test the app. Alternatively, you can create your QR code using online services like www.qrcode-monkey.com. Open the demo app and point your device's camera at one of the codes. You should see the decoded message.



http://www.appcoda.com



tel://743234028



fb://feed



sms://89234234



mailto:support@appcoda.com



whatsapp://send?text=Hello!

Using URL Schemes

For most of the built-in applications Apple provides support URL schemes. For instance, you use the `mailto` scheme to open the Mail app (e.g. `mailto:support@appcoda.com`) or the `tel` scheme to initiate a phone call (e.g. `tel://743234028`). To open an application with a custom URL scheme, all you need to do is call the `openURL` method of the `UIApplication` class. Here is the line of code:

```
UIApplication.sharedApplication().openURL(url)
```

Now we will modify the demo to open the corresponding app when a QR code is decoded. Open the Xcode project and select the `ViewController.swift` file. Add a helper method called `launchApp` in the class:

```
func launchApp(decodedURL: String) {
    let alertPrompt = UIAlertController(title: "Open App", message: "You're
going to open \(decodedURL)", preferredStyle: .ActionSheet)
    let confirmAction = UIAlertAction(title: "Confirm", style:
UIAlertActionStyle.Default, handler: { (action) -> Void in
```

```
if let url = NSURL(string: decodedURL) {
    if UIApplication.sharedApplication().canOpenURL(url) {
        UIApplication.sharedApplication().openURL(url)
    }
}
let cancelAction = UIAlertAction(title: "Cancel", style:
UIAlertActionStyle.Cancel, handler: nil)

alertPrompt.addAction(confirmAction)
alertPrompt.addAction(cancelAction)

presentViewController(alertPrompt, animated: true, completion: nil)
}
```

The `launchApp` method takes in a URL decoded from the QR code and creates an alert prompt. If the user taps the Confirm button, the app then creates an `NSURL` object and opens it accordingly. iOS will then open the corresponding app based on the given URL.

In the `captureOutput` method, which is called when a QR code is detected, insert a line of code to call the `launchApp` method:

```
launchApp(metadataObj.stringValue)
```

Place the above line of code right after:

```
messageLabel.text = metadataObj.stringValue
```

Now compile and run the app. Point your device's camera at one of the sample QR codes (e.g. `tel://743234028`). The app will prompt you with an action sheet when the QR code is decoded. Once you tap the Confirm button, it opens the Phone app and initiates the call.



But there is a minor issue with the current app. If you look into the console, you should find the following warning:

```
2015-10-09 13:23:47.911 QRReaderDemo[4931:2581028] Warning: Attempt to present <UIAlertController: 0x135e5a660> on <QRReaderDemo.ViewController: 0x135e3a230> which is already presenting <UIAlertController: 0x135f0f740>
```

The `launchApp` method is called every time when a barcode or QR code is scanned. So the app may present another `UIAlertController` when there is already a `UIAlertController` presented. To resolve the issue, we have to check if the app has presented a `UIAlertController` object before calling the `presentViewController` method.

In iOS, when you present a view controller modally using the `presentViewController` method, the presented view controller is stored in the `presentedViewController` property of the current view controller. For example, when the `viewController` object calls the `presentViewController` method to present the `UIAlertController` object, the `presentedViewController` property is set to the `UIAlertController` object. When the `UIAlertController` object is dismissed, the

```
presentedViewController
```

 property will be set to `nil`.

With this property, it is quite easy for us to fix the warning issue. All you need to do is put the following code at the beginning of the `launchApp` method:

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

We simply check if the property is set to a specific view controller, and present the `UIAlertController` object only if there is no presented view controller. Now run the app again. The warning should go away.

For reference, you can download the Xcode project from

<https://www.dropbox.com/s/6vxj9qabsr98vox/QRReaderDemoURLScheme.zip?dl=0>.

Quick note: You may notice that the app cannot open these two URLs: `fb://feed` and `whatsapp://send?text=Hello!`. These URLs are known as custom URL schemes created by third-party apps. In iOS 9, the app is not able to open these custom URLs. Apple has made a small change to the handling of URL scheme, specifically for the `canOpenURL()` method. If the URL scheme is not registered in the whitelist, the method returns `false`. This explains why the app cannot open Facebook and Whatsapp even it can decode their URLs. We will discuss more about custom URL scheme in the next section and show you how to workaround this issue.

Creating Your Custom URL Scheme

In the sample QR codes, I included two QR codes from third party apps:

- Facebook - `fb://feed`
- Whatsapp - `whatsapp://send?text>Hello!`

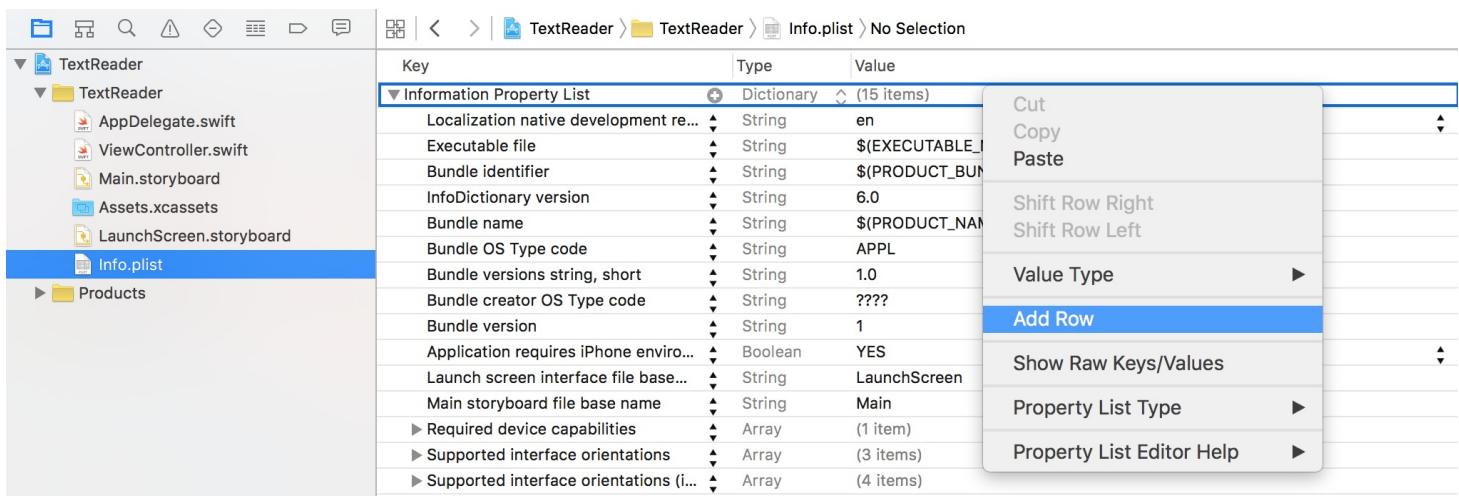
The first URL is used to open the news feed of the user's Facebook app. The other URL is for sending a text message using Whatsapp. Interestingly, Apple allows developers to create their own URLs for communicating between apps. Let's see how we can add a custom URL to our QR Reader app.

We're going to create another app called *TextReader*. This app serves as a receiver app that

defines a custom URL and accepts a text message from other apps. The custom URL will look like this:

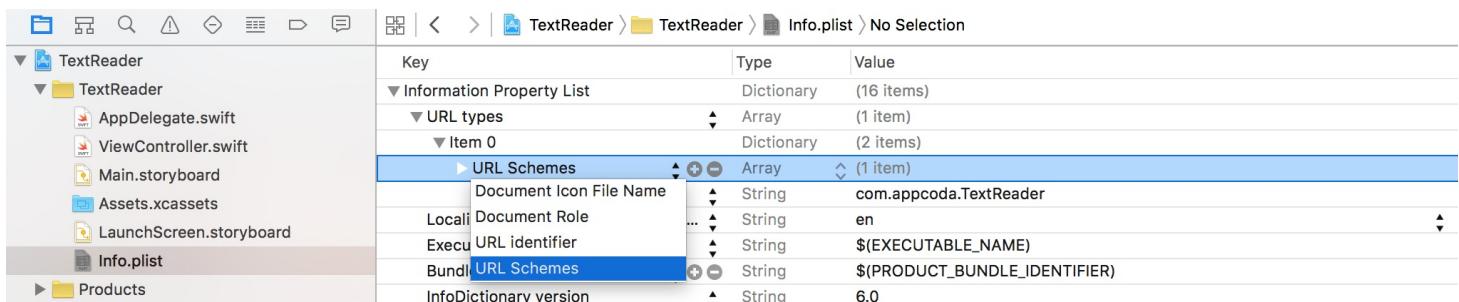
```
textreader://Hello!
```

When an app (e.g. QR Code Reader) launches the URL, iOS will open the TextReader app and pass it the *Hello!* message. In Xcode, create a new project using the Single View Application template and name it `TextReader`. Once the project is created, expand the Supporting Files folder in the project navigator and select `Info.plist`. Right click any of the rows and select `Add Row` to create a new key.



You'll be prompted to select a key from a drop-down menu. Scroll to the bottom and select `URL types`. This creates an array item so you can click the disclosure icon (i.e. triangle) to expand it. You'll then select Item 0. Click the disclosure icon next to the item and expand it to show the URL identifier line. Double-click the value field to fill in your identifier. Typically, you set the value to be the same as the bundle ID (e.g. com.appcoda.TextReader).

Next, right click on `Item 0` and select `Add Row` from the context menu. In the dropdown menu, select `URL Schemes` to add the item.



Again, click the disclosure icon of URL Schemes to expand the item. Double click the value box of Item 0 and key in `textreader`. If you've followed the procedures correctly, your *URL types* settings should look like this:

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
▼ URL types	Array	(1 item)
▼ Item 0	Dictionary	(2 items)
▼ URL Schemes	Array	(1 item)
Item 0	String	textreader
URL identifier	String	com.appcoda.TextReader
Localization native development re...	String	en

That's it. We have configured a custom URL scheme in the TextReader app. Now the app takes the URL in the form of `textreader://<message>`. We still need to write a few lines of code so that it knows what to do when another app launches the custom URL (e.g.

```
textreader://Hello! ).
```

As you know, the `AppDelegate` class implements the `UIApplicationDelegate` protocol. The method defined in the protocol gives you a chance to interact with important events during the lifetime of your app. When an *Open a URL* event is sent to your app, the system calls the `application(_:openURL:sourceApplication:annotation:)` method of the app delegate. You'll need to implement this method in respond to the launch of the custom URL.

Open `AppDelegate.swift` and insert the following method:

```
func application(application: UIApplication, openURL url: NSURL,
sourceApplication: String?, annotation: AnyObject) -> Bool {
    let message = url.host?.stringByRemovingPercentEncoding
    let alertController = UIAlertController(title: "Incoming Message", message:
message, preferredStyle: .Alert)
    let okAction = UIAlertAction(title: "OK", style:
UIAlertCellStyle.Default, handler: nil)
    alertController.addAction(okAction)

    window?.rootViewController?.presentViewController(alertController,
animated: true, completion: nil)

    return true
}
```

From the arguments of the `openURL` method, you can get the URL resource to open. For

instance, if another app launches `textreader://Hello!`, then the URL will be embedded in the URL object. The first line of code extracts the message by using the host property of the NSURL object.

URLs can only contain ASCII characters, spaces are not allowed. For characters outside the ASCII character set, they should be encoded using URL encoding. URL encoding replaces unsafe ASCII characters with a % followed by two hexadecimal digits and a space with `%20`. For example, “Hello World!” is encoded to *Hello%20World!* The `stringByRemovingPercentEncoding` method is used to decode the message by removing the URL percent encoding. The rest of the code is very straightforward. We instantiate a `UIAlertController` and present the message on screen.

If you compile and run the app, you should see a blank screen. That's normal because the TextReader app is triggered by another app using the custom URL. You have two ways to test the app. You can open mobile Safari and enter `textreader://Great!%20It%20works!` in the address bar - you'll be prompted to open the TextReader app. Once confirmed, the system should redirect you to the TextReader app and displays the `Great! It works!` message.

Alternatively, you can use the QR Code Reader app for testing. If you open the app and point the camera to the QR code shown below, the app should be able to decode the message but fails to open the TextReader app.



textreader://Great!%20It%20works!

The console should show you the following error:

```
2015-10-09 17:11:45.650 QRReaderDemo[5057:2632457] -canOpenURL: failed for URL:  
"textreader://Great!%20It%20works!" - error: "This app is not allowed to query  
for scheme textreader"
```

As explained earlier, iOS 9 has made some changes to the `canOpenURL` method. You have to register the custom URL schemes before the method returns `true`. To register a custom scheme, open `Info.plist` of the *QRReaderDemo* project and add a new key named `LSApplicationQueriesSchemes`. Set the type to `Array` and add the following items:

- textreader
- fb
- whatsapp

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
▼ LSApplicationQueriesSchemes	Array	◊ (3 items)
Item 0	String	textreader
Item 1	String	fb
Item 2	String	whatsapp
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)

Once you've made the change, test the QR Reader app again. Point to a QR code with a custom URL scheme (e.g. textreader). The app should be able to launch the corresponding app.



For reference, you can download the TextReader project from
<https://www.dropbox.com/s/ld05m7m5kj1ioo2/TextReader.zip?dl=0>.

Chapter 13

Building a Full Screen Camera



iOS provides two ways for developers to access the built-in camera for taking photos. The simple approach is to use `UIImagePickerController`, which I briefly covered in the [Beginning iOS 9 Programming book](#). This class is very handy and comes with a standard camera interface. Alternatively, you can control the built-in cameras and capture images using AVFoundation framework. Compared to `UIImagePickerController`, AVFoundation framework is more complicated, but also far more flexible and powerful for building a fully custom camera interface.

In this chapter, we will see how to use the AVFoundation framework for capturing still images. You will learn a lot of stuff including:

- How to create a camera interface using the AVFoundation framework
- How to capture a still image using both the front-facing and back-facing camera
- How to use gesture recognizers to detect a swipe gesture
- How to provide a zoom feature for the camera app
- How to save an image to the camera roll

The core of AV Foundation media capture is an `AVCaptureSession` object. It controls the data flow between an input (e.g. back-facing camera) and an output (e.g. image file). In general, to capture a still image using the AVFoundation framework, you'll need to:

- Get an instance of `AVCaptureDevice` that represents the underlying input device such as the back-facing camera
- Create an instance of `AVCaptureDeviceInput` with the device
- Create an instance of `AVCaptureStillImageOutput` to manage the output to a still image
- Use `AVCaptureSession` to coordinate the data flow from the input and the output
- Create an instance of `AVCaptureVideoPreviewLayer` with the session to show a camera preview

If you still have questions at this point, no worries. The best way to learn any new concept is by trying it out - following along with the demo creation should help to clear up any confusion surrounding the AV Foundation framework.

Demo App

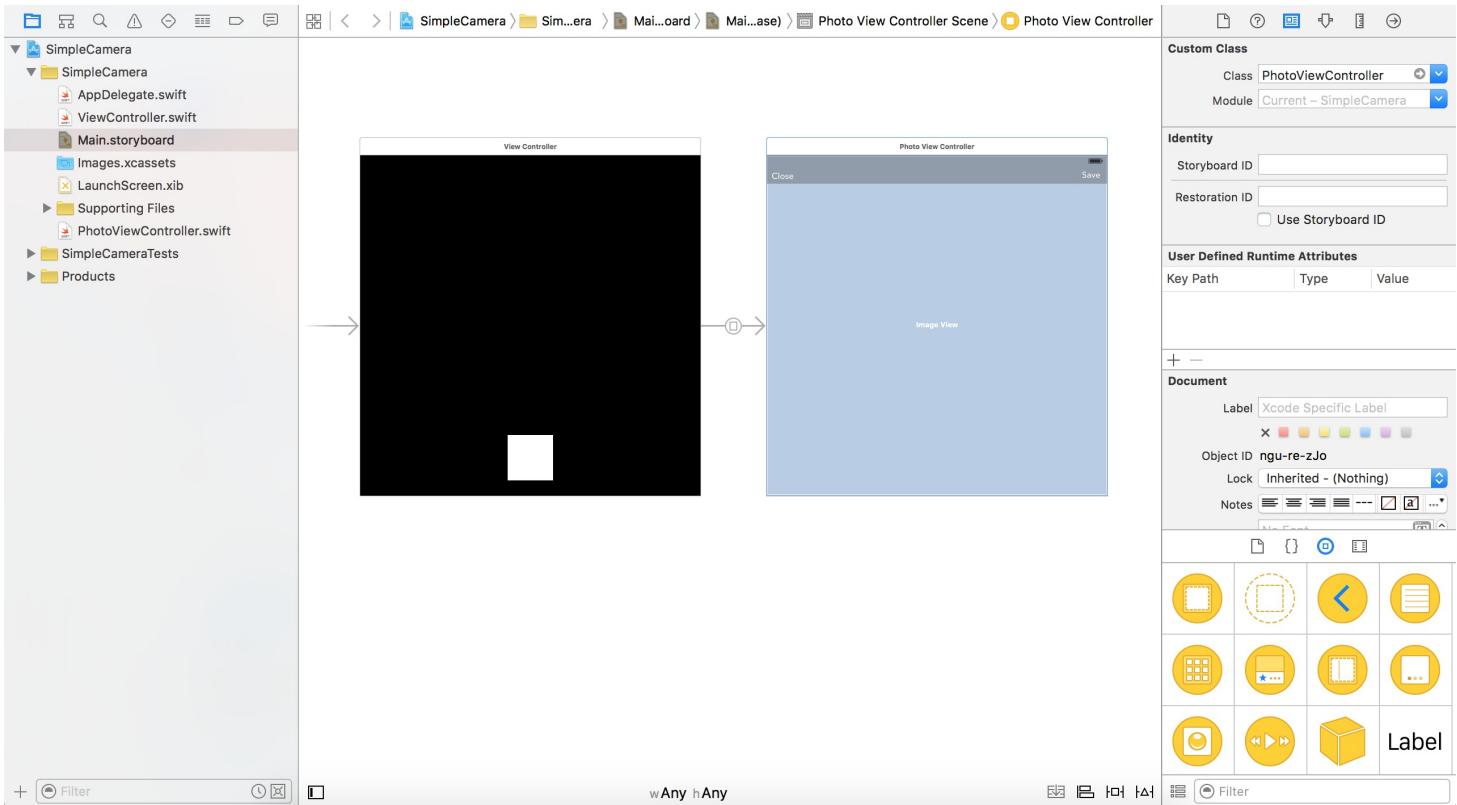
We're going to build a simple camera app that offers a full-screen experience and gesture-based controls. The app provides a minimalistic UI with a single capture button at the bottom of the screen. Users can swipe up the screen to switch between the front-facing and back-facing cameras. The camera offers up to 5x digital zoom. Users can swipe the screen from left to right to zoom in or from right to left to zoom out.

When the user taps the capture button, it should capture the photo in full resolution. Optionally, the user can save to the photo album.



To begin, you can download the Xcode project template from <https://www.dropbox.com/s/pklclny9sbsod16/SimpleCameraTemplate.zip?dl=0>. The template includes a pre-built storyboard and custom classes. If you open the Storyboard, you will find two view controllers. The Main View Controller is used to display the camera interface, while the Photo View Controller is designed for displaying a still image after taking the photo. The view controller has been associated with the `ViewController` class. When the capture button is tapped, it will call the `capture` action method.

The Photo View Controller is associated with the `PhotoViewController` class. The *Save* button is associated with the `save` action method, which is now without any implementation.



Configuring a Session

As mentioned, the heart of AVFoundation media capture is the `AVCaptureSession` object. So open `ViewController.swift` and declare a variable of the type `AVCaptureSession`:

```
let captureSession = AVCaptureSession()
```

In the `viewDidLoad` method, insert the following lines of code to configure the session:

```
// Preset the session for taking photo in full resolution
captureSession.sessionPreset = AVCaptureSessionPresetPhoto
```

You use the `sessionPreset` property to specify the image quality and resolution you want. Here we preset it to `AVCaptureSessionPresetPhoto`, which indicates a full photo resolution.

Selecting the Input Device

The next step is to find out the camera devices for taking photos. First declare the following instance variables in the `ViewController` class:

```
var backFacingCamera:AVCaptureDevice?  
var frontFacingCamera:AVCaptureDevice?  
var currentDevice:AVCaptureDevice?
```

Since the camera app supports both front and back-facing cameras, we create two separate variables for storing the `AVCaptureDevice` objects. The `currentDevice` variable is used for storing the current device that is selected by the user.

Next, insert the following code in the `viewDidLoad` method:

```
let devices = AVCaptureDevice.devices(withMediaType: AVMediaTypeVideo) as!  
[AVCaptureDevice]  
// Get the front and back-facing camera for taking photos  
for device in devices {  
    if device.position == AVCaptureDevicePosition.Back {  
        backFacingCamera = device  
    } else if device.position == AVCaptureDevicePosition.Front {  
        frontFacingCamera = device  
    }  
}  
currentDevice = backFacingCamera  
  
do {  
    let captureDeviceInput = try AVCaptureDeviceInput(device: currentDevice)  
} catch {  
    print(error)  
}
```

In the AVFoundation framework, a physical device is abstracted by an `AVCaptureDevice` object. Apparently, an iPhone has more than one input (audio and video). The `AVCaptureDevice` class provides a couple of methods for querying the available capture devices. You can use the `devicesWithMediaType` method to retrieve the available devices used to capture data of a specific media type. In the code snippet, we ask `AVCaptureDevice` to return us an array of capture devices that are capable of capturing video/still image (i.e. `AVMediaTypeVideo`).

With the cameras returned, we examine its `position` property to determine if it is a front-facing or back-facing camera. By default, the camera app uses the back-facing camera when it's first started. Thus, we set the `currentDevice` to the back-facing camera.

Lastly, we create an instance of `AVCaptureDeviceInput` with the current device so that you can capture data from the device.

Configuring an Output Device

With the input device configured, declare the following variable in the `ViewController` class for the device output:

```
var stillImageOutput:AVCaptureStillImageOutput?  
var stillImage:UIImage?
```

Insert the following code in the `viewDidLoad` method:

```
// Configure the session with the output for capturing still images  
stillImageOutput = AVCaptureStillImageOutput()  
stillImageOutput?.outputSettings = [AVVideoCodecKey: AVVideoCodecJPEG]
```

Here we create an instance of `AVCaptureStillImageOutput` and configure it to use the JPEG encoder. The `AVCaptureStillImageOutput` is specially designed for capturing still images.

Coordinating the Input and Output using Session

Now that you have configured both input and output, you'll need to assign them to the capture session so that it can coordinate the flow of data between them. Continue to insert the following lines of code in the `do` block of the `viewDidLoad` method:

```
// Configure the session with the input and the output devices  
captureSession.addInput(captureDeviceInput)  
captureSession.addOutput(stillImageOutput)
```

Creating a Preview Layer and Start the Session

You have now configured the `AVCaptureSession` object and are ready to present the camera preview. First, declare an instance variable:

```
var cameraPreviewLayer:AVCaptureVideoPreviewLayer?
```

And insert the following code in the `viewDidLoad` method:

```
// Provide a camera preview  
cameraPreviewLayer = AVCaptureVideoPreviewLayer(session: captureSession)  
view.layer.addSublayer(cameraPreviewLayer!)
```

```
cameraPreviewLayer?.videoGravity = AVLayerVideoGravityResizeAspectFill
cameraPreviewLayer?.frame = view.layer.frame

// Bring the camera button to front
view.bringSubviewToFront(cameraButton)
captureSession.startRunning()
```

You use the `AVCaptureVideoPreviewLayer` to display video as it is being captured by an input device. The layer is then added to the view's layer to display on the screen. The preview layer object provides a property named `videoGravity` that indicates how the video preview is displayed. In order to give a full-screen camera interface, we set it to `AVLayerVideoGravityResizeAspectFill`. You're free to explore the other two options (`AVLayerVideoGravityResize` and `AVLayerVideoGravityResizeAspect`) and see how the camera interface is presented.

As you add the preview layer to the view, it should cover the camera button. To unhide the button, we simply bring it to the front. Lastly, we call the `startRunning` method of the session to start capturing data.

If you compile and run the app on a real device, you should see the camera preview, though the camera button doesn't work yet.

Capture a Still Image

To capture a still image when the camera button is tapped, update the `capture` method of the `ViewController.swift` file to the following:

```
@IBAction func capture(sender: AnyObject) {
    let videoConnection =
stillImageOutput?.connectionWithMediaType(AMediaTypeVideo)

stillImageOutput?.captureStillImageAsynchronouslyFromConnection(videoConnection,
completionHandler: { (imageDataSampleBuffer, error) -> Void in

    let imageData =
AVCaptureStillImageOutput.jpegStillImageNSDataRepresentation(imageDataSampleBuff

        self.stillImage = UIImage(data: imageData)
        self.performSegueWithIdentifier("showPhoto", sender: self)

})
```

```
}
```

When you add the input and the output to the session, the session forms connections between them. To capture a still image, we first retrieve the corresponding connection (i.e. the one with video media type). We then call the `captureStillImageAsynchronouslyFromConnection` method to capture a still image asynchronously. When the capture completes, the system will call the complete handler with the corresponding image data.

With the data captured in the form of a `CMSampleBuffer`, we call the `jpegStillImageNSDataRepresentation` method to convert the buffer to an `NSData` object of a still image. Lastly, we invoke the `showPhoto` segue to display the still image in the Photo View Controller. Remember to add the `prepareForSegue` method in the `viewController` class:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {  
    // Get the new view controller using segue.destinationViewController.  
    // Pass the selected object to the new view controller.  
    if segue.identifier == "showPhoto" {  
        let photoViewController = segue.destinationViewController as!  
PhotoViewController  
        photoViewController.image = stillImage  
    }  
}
```

Now you're ready to test the app. Hit the Run button and test out the camera button. It should now work and be able to capture a still image.

Toggle between Front and Back Facing Camera Using Gestures

The camera app is expected to support both front and back-facing cameras. Instead of using a button for the switching, we will implement a gesture-based control. When a user swipes up the screen, the app toggles between the cameras. The iOS SDK provides various gesture recognizers for detecting common gestures such as tap and pinch. To recognize swiping gestures, you use the `UISwipeGestureRecognizer` class. First let's declare an instance variable of the swipe recognizer:

```
var toggleCameraGestureRecognizer = UISwipeGestureRecognizer()
```

Then insert the following code in the `viewDidLoad` method:

```
// Toggle Camera recognizer
toggleCameraGestureRecognizer.direction = .Up
toggleCameraGestureRecognizer.addTarget(self, action: "toggleCamera")
view.addGestureRecognizer(toggleCameraGestureRecognizer)
```

The `UISwipeGestureRecognizer` object is capable of recognizing swiping gestures in one or more directions. Since we look for swipe-up gestures, we configure the recognizer for the `.Up` direction only. You use the `addTarget` method to tell the recognizer what to do when the gesture is detected. Here we instruct it to call the `toggleCamera` method, which will be implemented shortly. Once you've configured the recognizer object, you have to attach it to a view; that is the view that receives touches. We simply call the `addGestureRecognizer` method of the view to attach the swipe recognizer.

Now create a new method called `toggleCamera` in the `ViewController` class:

```
func toggleCamera() {
    captureSession.beginConfiguration()

    // Change the device based on the current camera
    let newDevice = (currentDevice?.position == AVCaptureDevicePosition.Back) ?
frontFacingCamera : backFacingCamera

    // Remove all inputs from the session
    for input in captureSession.inputs {
        captureSession.removeInput(input as! AVCaptureDeviceInput)
    }

    // Change to the new input
    let cameraInput:AVCaptureDeviceInput
    do {
        cameraInput = try AVCaptureDeviceInput(device: newDevice)
    } catch {
        print(error)
        return
    }

    if captureSession.canAddInput(cameraInput) {
        captureSession.addInput(cameraInput)
    }

    currentDevice = newDevice
    captureSession.commitConfiguration()
}
```

The method is used to toggle between front-facing and back-facing cameras. To switch the input device of a session, we first call the `beginConfiguration` method of the capture session. This indicates the start of the configuration change. Next, we determine the new device to use. Before adding the new device input to the session, you have to remove all existing inputs from the session. You can simply access the `inputs` property of the session to get the existing inputs. We simply loop through them and remove them from the session by calling the `removeInput` method.

Once all the inputs are removed, we add the new device input (i.e. front / back facing camera) to the session. Lastly, we call the `commitConfiguration` method of the session to commit the changes. Note that no changes are actually made until you invoke the method.

It's time to have a quick test. Run the app on a real iOS device. You should be able to switch between cameras by swiping up the screen.

Zoom in and Out

The camera app also provides a digital zoom feature that allows up to 5x magnification. Again, we will not use a button for controlling the zooming. Instead, the app allows users to zoom by using a swipe gesture. To zoom into a particular subject, just swipe the screen from left to right. To zoom out, swipe the screen from right to left.

In the `viewController` class, declare two instance variables of `UISwipeGestureRecognizer`:

```
var zoomInGestureRecognizer = UISwipeGestureRecognizer()
var zoomOutGestureRecognizer = UISwipeGestureRecognizer()
```

Next, insert the following lines of code in the `viewDidLoad` method:

```
// Zoom In recognizer
zoomInGestureRecognizer.direction = .Right
zoomInGestureRecognizer.addTarget(self, action: "zoomIn")
view.addGestureRecognizer(zoomInGestureRecognizer)

// Zoom Out recognizer
zoomOutGestureRecognizer.direction = .Left
zoomOutGestureRecognizer.addTarget(self, action: "zoomOut")
view.addGestureRecognizer(zoomOutGestureRecognizer)
```

Here we define the `direction` property and the corresponding action method of the swipe gesture recognizers. I will not go into the details because the implementation is pretty much the same as that covered in the previous section.

Now create two new methods for `zoomIn` and `zoomOut`:

```
func zoomIn() {
    if let zoomFactor = currentDevice?.videoZoomFactor {
        if zoomFactor < 5.0 {
            let newZoomFactor = min(zoomFactor + 1.0, 5.0)
            do {
                try currentDevice?.lockForConfiguration()
                currentDevice?.rampToVideoZoomFactor(newZoomFactor, withRate:
1.0)
                currentDevice?.unlockForConfiguration()
            } catch {
                print(error)
            }
        }
    }
}

func zoomOut() {
    if let zoomFactor = currentDevice?.videoZoomFactor {
        if zoomFactor > 1.0 {
            let newZoomFactor = max(zoomFactor - 1.0, 1.0)
            do {
                try currentDevice?.lockForConfiguration()
                currentDevice?.rampToVideoZoomFactor(newZoomFactor, withRate:
1.0)
                currentDevice?.unlockForConfiguration()
            } catch {
                print(error)
            }
        }
    }
}
```

To change the zoom level of a camera device, all you need to do is adjust the `videoZoomFactor` property. The property controls the enlargement of images captured by the device. For example, a value of `2.0` doubles the size of an image. If it is set to `1.0`, it resets to display a full field of view. You can directly modify the value of the property to achieve a zoom effect. However, to provide a smooth transition from one zoom factor to another, we use the `rampToVideoZoomFactor` method. By providing a new zoom factor and a rate of transition, the

method delivers a smooth zooming transition.

With some basic understanding of the zooming effect, let's look further into both methods. In the `zoomIn` method, we first check if the zoom factor is larger than 5.0 (the camera app only supports up to 5x magnification.) If zooming is allowed, we then increase the zoom factor by 1.0. We use the `min()` function to ensure the new zoom factor does not exceed 5.0. To change a property of a capture device, you have to first call the `lockForConfiguration` method to acquire a lock of the device. Then we call the `rampToVideoZoomFactor` method with the new zoom factor to achieve the zooming effect. Once done, we release the lock by calling the `unlockForConfiguration` method.

The `zoomOut` method works pretty much the same as the `zoomIn` method. Instead of increasing the zoom factor, the method reduces the zoom factor when called. The minimum value of the zoom factor is 1.0; this is why we have to ensure the zoom factor is not set to any value less than 1.0.

Now hit the Run button to test the app on your iOS device. When the camera app is launched, try out the zoom feature by swiping the screen from left to right.

Saving Images to the Photo Album

The `PhotoViewController` class is used to display a still image captured by the device. For now the image is stored in memory. You can't save the image to the built-in photo album because we haven't implemented the *Save* button yet. If you open the `PhotoViewController.swift` file, the `save` action method, which is connected to the Save button, is empty.

It is very simple to save a still image to the Camera Roll album. UIKit provides the following function to let you add an image to the user's Camera Roll album:

```
func UIImageWriteToSavedPhotosAlbum(_ image: UIImage!, _ completionTarget:  
AnyObject!, _ completionSelector: Selector , _ contextInfo:  
UnsafeMutablePointer<Void>)
```

So in the `save` method of the `PhotoViewController` class, insert a couple lines of code. Your `save` method should look like this:

```
@IBAction func save(sender: AnyObject) {
```

```
guard let imageToSave = image else {
    return
}

UIImageWriteToSavedPhotosAlbum(imageToSave, nil, nil, nil)
dismissViewControllerAnimated(true, completion: nil)
}
```

We first check if the image is ready to save. And then call the `UIImageWriteToSavedPhotosAlbum` function to save the still image to the camera roll, followed by dismissing the view controller.

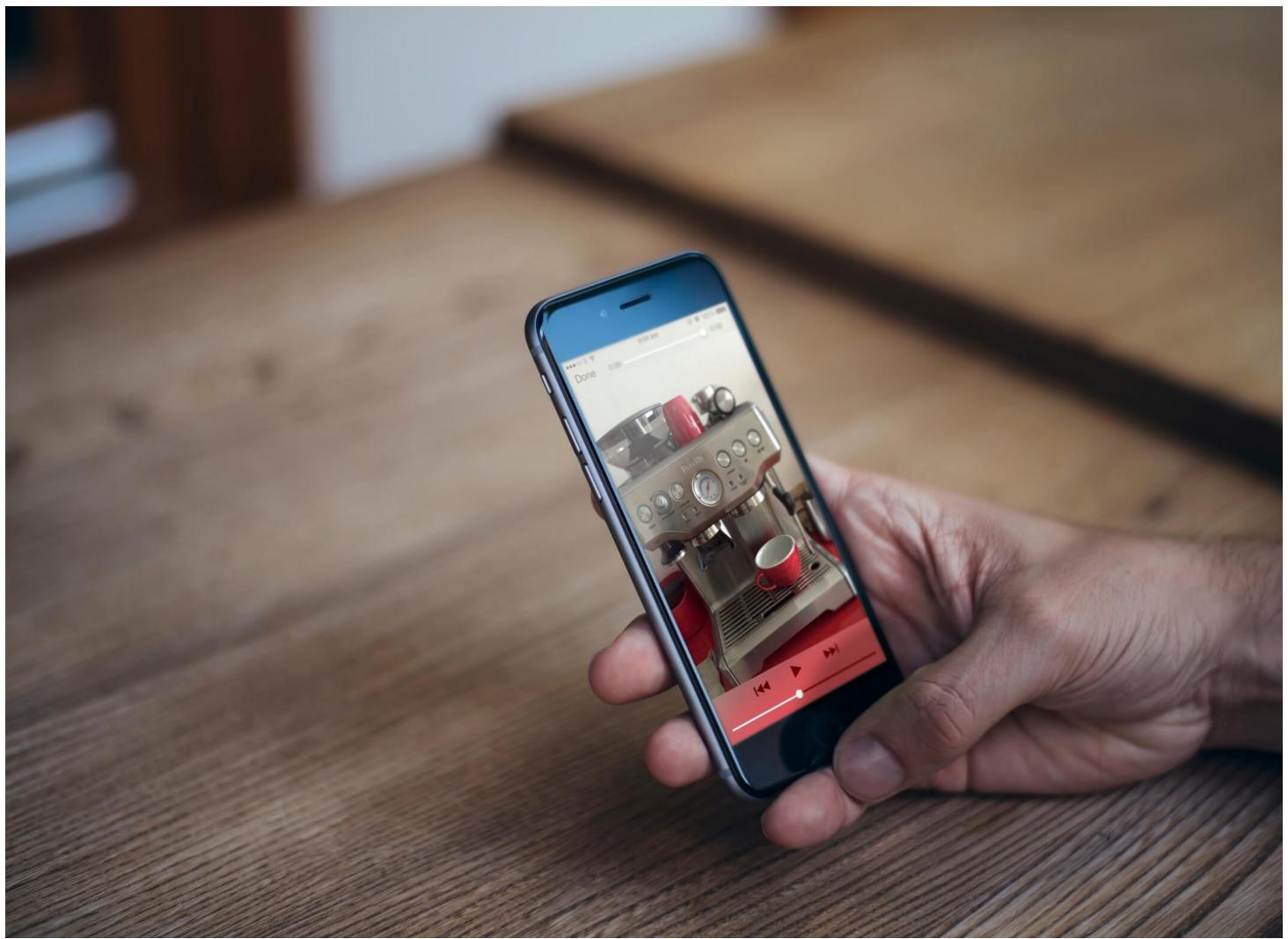
Hit the Run button again to test the app. The Camera app should now be able to save photos to your photo album. Just use Photos app to verify the result.



Congratulations! You've managed to use the AVFoundation framework and build a camera app for capturing photos. To further explore the framework, I recommend you check out the official documentation from Apple. For your reference, you can download the complete Xcode project from <https://www.dropbox.com/s/1zr5c9wqy9ys2u3/SimpleCamera.zip?dl=0>.

Chapter 14

Video Capturing and Playback Using AVKit



Previously, we built a simple camera app using the AVFoundation framework. You are not limited to using the framework for capturing still images. By changing the input and the output of `AVCaptureSession`, you can easily turn the simple camera app into a video-capturing app.

In this chapter, we will develop a simple video app that allows users to record videos. Not only we will explore video capturing, but I will also show you a new framework known as *AVKit*. The framework was first introduced in iOS 8 and can be used to play video content in your iOS

app. You will discover how easy it is to integrate AVKit into your app for video playback.

To get started, download the project template from

<https://www.dropbox.com/s/af9fu0329r87r2q/SimpleVideoCameraTemplate.zip?dl=0>. The template is very similar to the one you worked on in the previous chapter. If you run the template, you will see a blank screen with a red button (which is the record button) at the bottom part of the screen.



Configuring a Session

Similar to image capturing, the first thing to do is import the `AVFoundation` framework and prepare the `AVCaptureSession` object. In the `ViewController.swift` file, insert the following statement at the beginning of the file:

```
import AVFoundation
```

And, declare an instance variable of `AVCaptureSession`:

```
let captureSession = AVCaptureSession()
```

In the `viewDidLoad` method, insert the following lines of code to configure the session:

```
// Preset the session for taking photo in full resolution  
captureSession.sessionPreset = AVCaptureSessionPresetHigh
```

Here, we define the session and preset it to `AVCaptureSessionPresetHigh`, which indicates a

high quality output. Alternatively, you can set the value to `AVCaptureSessionPresetMedium`, which is suitable for capturing videos that can be shared over WiFi. If you need to share the video over a 3G network, you may set the value to `AVCaptureSessionPresetLow`.

Selecting the Input Device

Next, we have to find out the camera devices for shooting videos. First declare the following instance variable in the `ViewController` class:

```
var currentDevice:AVCaptureDevice?
```

Then, insert the following code in the `viewDidLoad` method:

```
// Preset the session for taking photo in full resolution
captureSession.sessionPreset = AVCaptureSessionPresetHigh

// Get the available devices that is capable of taking video
let devices = AVCaptureDevice.devicesWithMediaType(AMediaTypeVideo) as!
[AVCaptureDevice]

// Get the back-facing camera for taking videos
for device in devices {
    if device.position == AVCaptureDevicePosition.Back {
        currentDevice = device
    }
}

let captureDeviceInput:AVCaptureDeviceInput
do {
    captureDeviceInput = try AVCaptureDeviceInput(device: currentDevice)
} catch {
    print(error)
    return
}
```

For this demo app, we only support the back-facing camera for capturing videos. Once the device is retrieved, we create an instance of `AVCaptureDeviceInput`.

Configuring an Output Device

With the input device configured, declare the following variable in the `ViewController` class

for the device output:

```
var videoFileOutput:AVCaptureMovieFileOutput?
```

Insert the following code in the `viewDidLoad` method:

```
// Configure the session with the output for capturing video  
videoFileOutput = AVCaptureMovieFileOutput()
```

Here, we create an instance of `AVCaptureMovieFileOutput`. This output is used to save data to a QuickTime movie file. `AVCaptureMovieFileOutput` provides a couple of properties for controlling the length and size of the recording. For example, you can use the `maxRecordedDuration` property to specify the longest duration allowed for the recording. In this demo, we just use the default settings.

Coordinating the Input and Output using the Capture Session

Now that you have configured both input and output, the next step is to assign them to the capture session so that it can coordinate the flow of data between them. Continue to insert the following lines of code in the `viewDidLoad` method:

```
// Configure the session with the input and the output devices  
captureSession.addInput(captureDeviceInput)  
captureSession.addOutput(videoFileOutput)
```

Creating a Preview Layer and Starting the Session

With the session configured, it's time to create a preview layer for the camera preview. First, declare an instance variable:

```
var cameraPreviewLayer:AVCaptureVideoPreviewLayer?
```

And insert the following code in the `viewDidLoad` method:

```
// Provide a camera preview  
cameraPreviewLayer = AVCaptureVideoPreviewLayer(session: captureSession)  
view.layer.addSublayer(cameraPreviewLayer!)
```

```
cameraPreviewLayer?.videoGravity = AVLayerVideoGravityResizeAspectFill
cameraPreviewLayer?.frame = view.layer.frame

// Bring the camera button to front
view.bringSubviewToFront(cameraButton)
captureSession.startRunning()
```

You use `AVCaptureVideoPreviewLayer` to display video as it is being captured by an input device. The layer is then added to the view's layer to display on the screen. This is pretty much the same as what we implemented in the Camera app.

When you add the preview layer to the view, it will cover the record button. To unhide the button, we simply bring it to the front. Lastly, we call the `startRunning` method of the session to start capturing data. If you compile and run the app on a real device, you should see the camera preview.

Let's continue to implement the video capturing.

Saving Video Data to a Movie File

For the demo app, the recording process starts once the red button is tapped. First, declare a Boolean variable to indicate whether video recording is in process:

```
var isRecording = false
```

Now the output of the session is configured for capturing data to a movie file. However, the saving process will not start until the `startRecordingToOutputFileURL` method of `AVCaptureMovieFileOutput` is invoked. Presently, the `capture` method is empty. Update the method with the following code:

```
@IBAction func capture(sender: AnyObject) {
    if !isRecording {
        isRecording = true

        UIView.animateWithDuration(0.5, delay: 0.0, options: [.Repeat,
        .Autoreverse, .AllowUserInteraction], animations: { () -> Void in
            self.cameraButton.transform = CGAffineTransformMakeScale(0.5, 0.5)
        }, completion: nil)

        let outputPath = NSTemporaryDirectory() + "output.mov"
        let outputFileURL = NSURL(fileURLWithPath: outputPath)
```

```

        videoFileOutput?.startRecordingToOutputFileURL(outputFileURL,
recordingDelegate: self)
    } else {
        isRecording = false

        UIView.animateWithDuration(0.5, delay: 1.0, options: [], animations: {
() -> Void in
            self.cameraButton.transform = CGAffineTransformMakeScale(1.0, 1.0)
        }, completion: nil)
        cameraButton.layer.removeAllAnimations()
        videoFileOutput?.stopRecording()
    }
}

```

We first check if the app is doing any recordings. If not, we initiate video capturing. Once recording starts, we create a simple animation for the button to indicate recording is in progress. If you've read over Chapter 13 of the [Beginning iOS 9 Programming book](#), the `animateWithDuration` method shouldn't be new to you. What's new to you are the animation options. Here I want to create a pulse animation for the button. In order to create such an effect, here is what needs to be done:

- First, reduce the size of the button by 50%
- Then grow the button to the original size
- Keep repeating step #1 and #2

If we write the above steps in code, this is the code snippet you need:

```

UIView.animateWithDuration(0.5, delay: 0.0, options: [.Repeat, .Autoreverse,
.AllowUserInteraction], animations: { () -> Void in
    self.cameraButton.transform = CGAffineTransformMakeScale(0.5, 0.5)
}, completion: nil)

```

For step #1, we use the `CGAffineTransformMakeScale` function to scale down the button. With `UIView` animation, the button will reduce its size by half smoothly.

For step #2, we use the `.Autoreverse` animation option to run the animation backward. The button will grow to its original size.

To repeat step #1 and #2, we specify the `.Repeat` animation option to repeat the animation indefinitely. While animating the button, users should still be able to interact with it. This is why we also specify the `.AllowUserInteraction` option.

Now let's get back to the code for saving video data. The `AVCaptureMovieFileOutput` class provides a convenient method called `startRecordingToOutputFileURL` to capture data to a movie file. All you need to do is specify an output file path and the delegate object. Here, we simply save the video to the temporary folder. Once the recording is completely written to the movie file, it will notify the delegate object by calling the following method:

```
captureOutput:didFinishRecordingToOutputFileAtURL:fromConnections:error:
```

Conversely, if the recording is in progress when the button is tapped, we simply reset the button to its original size, remove all animations, and then stop the recording.

Using AVKit for Video Playback

Now the app should be able to capture a video to a movie file. But how can you play the video in your app? iOS 9 provides the AVKit framework for handling video playback. If you have some experience with older version of the iOS SDK, you might be using

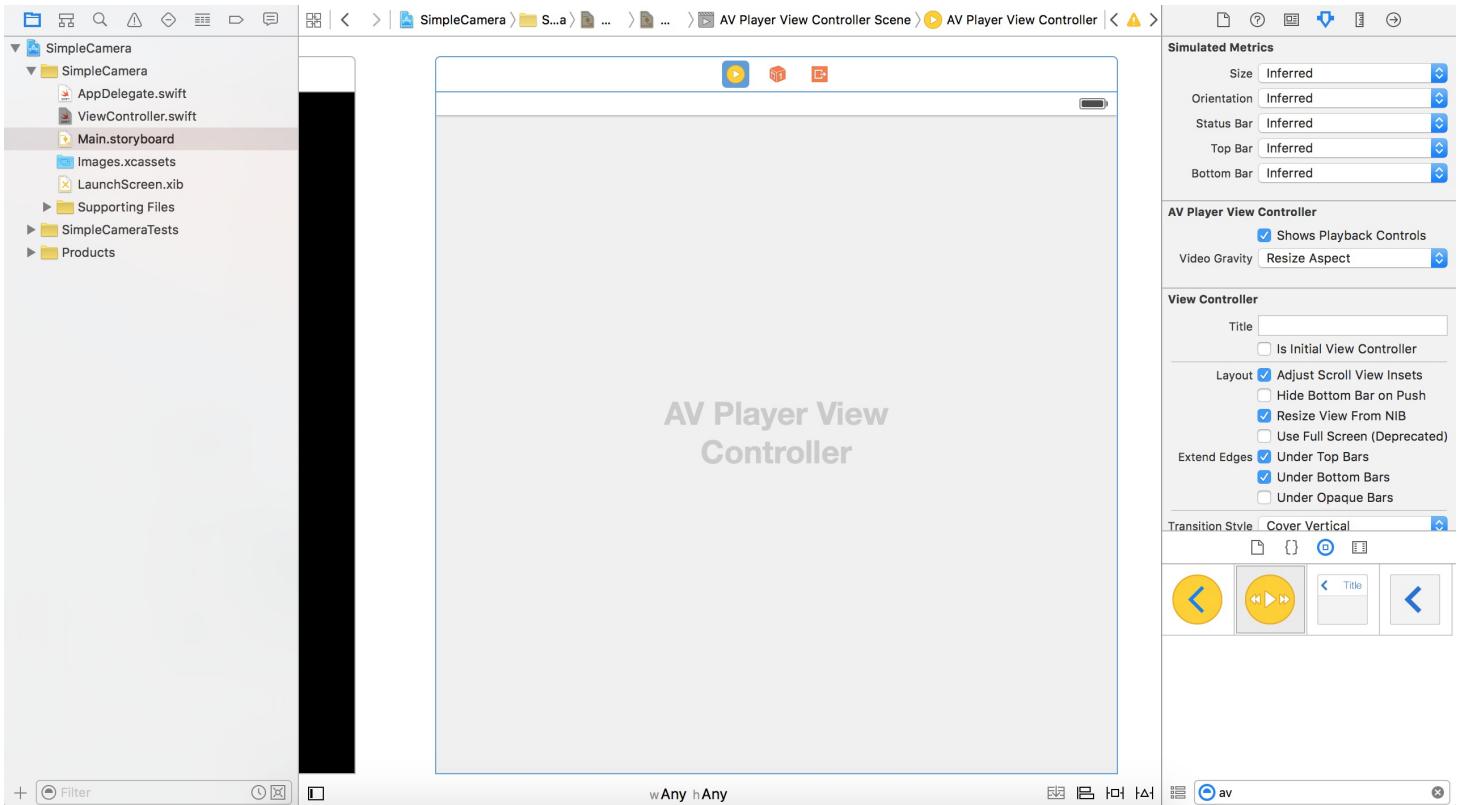
`MPMoviePlayerController` in your applications for displaying video content. You are now encouraged to replace it with the new `AVPlayerViewController`.

AVKit is a very simple framework with just two classes. `AVPlayerViewController` is one of them. The class is a subclass of `UIViewController` with additional features for displaying video content and playback controls. The heart of the `AVPlayerViewController` class is the `player` property, which provides video content to the view controller. The player is of the type `AVPlayer`, which is a class from the AVFoundation framework for controlling playback. To use `AVPlayerViewController` for video playback, you just need to set the player property to an `AVPlayer` object.

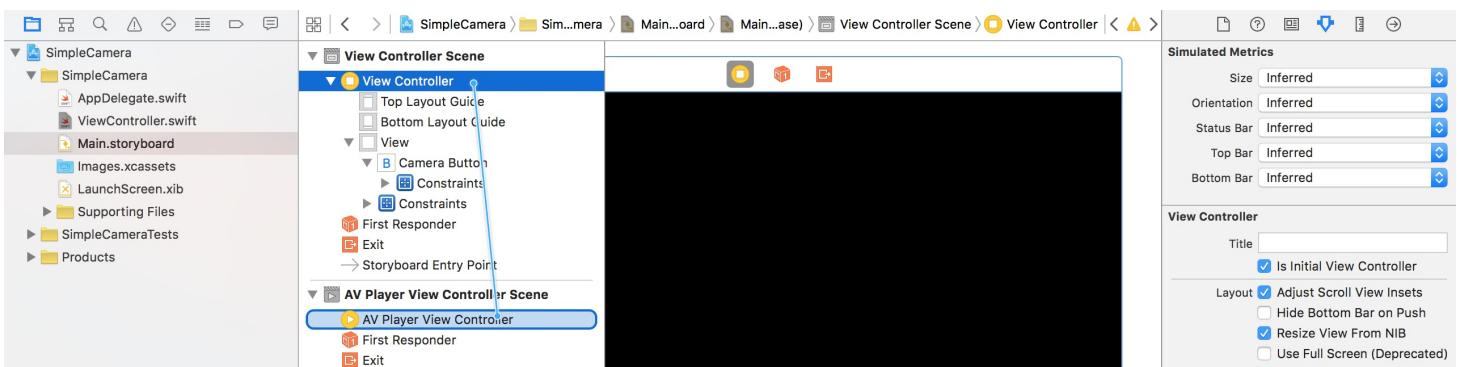
Apple has made it easy for you to integrate `AVPlayerViewController` in your apps. If you go to the Interface Builder and open the Object library, you will find an `AVPlayerViewController` object. You can drag the object onto the storyboard and connect it with other view controllers.

Okay, let's continue to develop the video camera app.

First, open `Main.storyboard` and drag an `AVPlayerViewController` object to the storyboard.



Next, connect the original View Controller to the AV Player View Controller using a segue. In the Document Outline, control-drag from the View Controller to the AV Player View Controller. When prompted, select *Present Modally* as the segue type. Select the segue and go to Attributes inspector. Set the identifier of the segue to `playVideo`.



Implement the `AVCaptureFileOutputRecordingDelegate` Protocol

Now that you have created the UI of `AVPlayerController`, the real question is: *when will we bring it up for video playback?* For the demo app, we'll play the movie file right after the user

stops the recording.

As mentioned earlier, `AVCaptureMovieFileOutput` will call the `captureOutput` method of the delegate object once the video has been completely written to a movie file. Here, the `ViewController` is the delegate object, which should implement the `AVCaptureFileOutputRecordingDelegate` protocol.

So open the `ViewController.swift` file, import `AVKit` and implement the protocol:

```
import AVKit

class ViewController: UIViewController, AVCaptureFileOutputRecordingDelegate
```

Next, implement the delegate method and segue method like this:

```
func captureOutput(captureOutput: AVCaptureFileOutput!, didFinishRecordingToOutputFileAtURL outputFileURL: NSURL!, fromConnections connections: [AnyObject]!, error: NSError!) {

    if error != nil {
        print(error)
        return
    }

    performSegueWithIdentifier("playVideo", sender: outputFileURL)
}

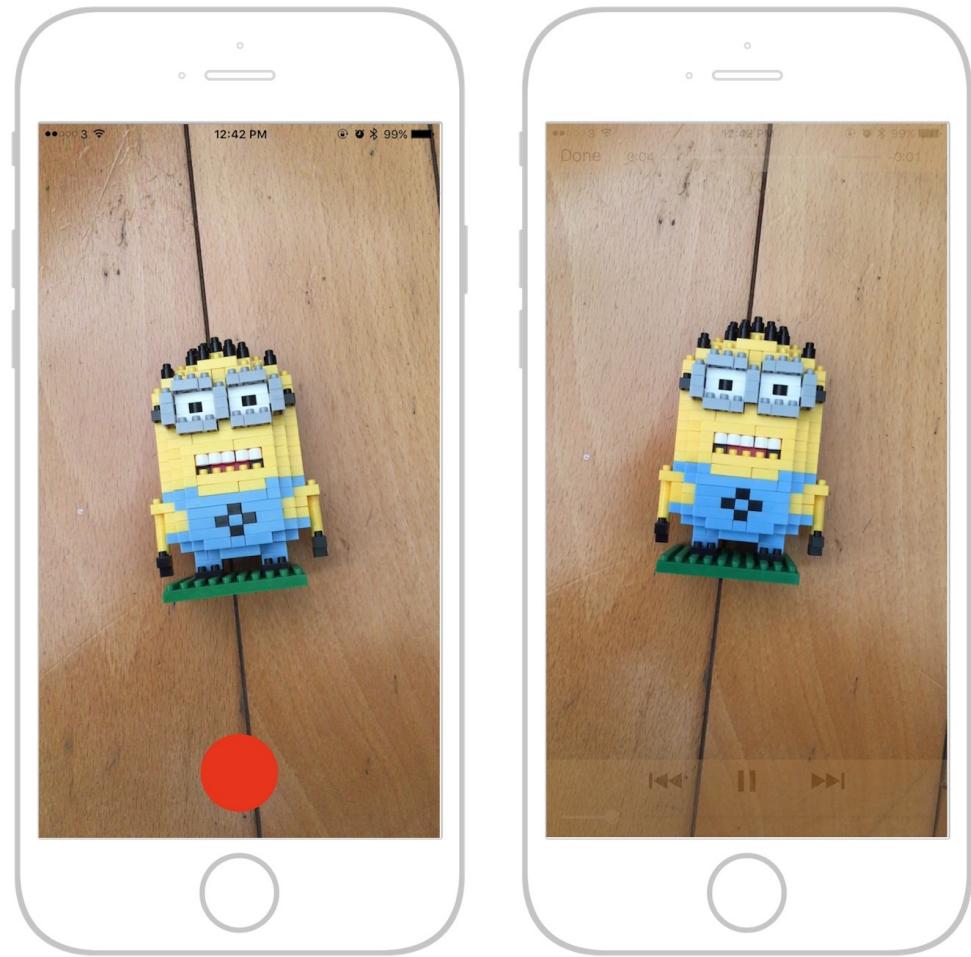
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "playVideo" {
        let videoPlayerViewController = segue.destinationViewController as!
        AVPlayerViewController
        let videoFileURL = sender as! NSURL
        videoPlayerViewController.player = AVPlayer(URL: videoFileURL)
    }
}
```

When a video is captured and written to a file, the above method is invoked. We simply determine if there are any errors and bring up the AV Player View Controller by calling the `performSegueWithIdentifier` method with the video file URL.

In the `performSegueWithIdentifier` method, we pick the video file URL and create an instance of `AVPlayer` with the URL. Setting the `player` property with the `AVPlayer` object is all you

need to perform video playback. AVFoundation then takes care of opening the video URL, buffering the content and playing it back.

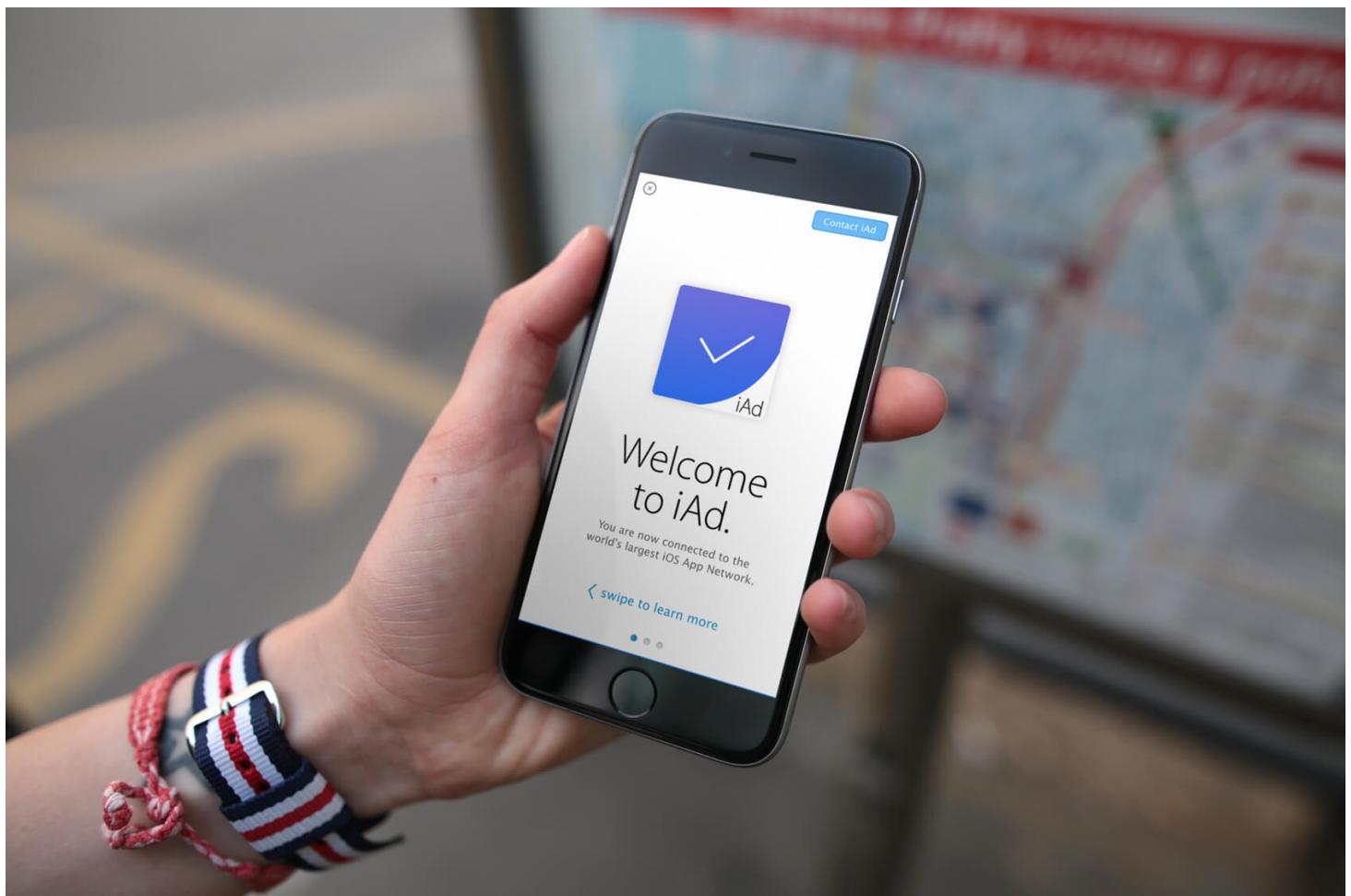
Now you're ready to test the video camera app. Hit Run and capture a video. Once you stop the video capturing, the app automatically plays the video in the AV Player View Controller.



For reference, you can download the Xcode project from
<https://www.dropbox.com/s/hegmhb1326fcli/SimpleVideoCamera.zip?dl=0>.

Chapter 15

Displaying Banner Ads using iAd

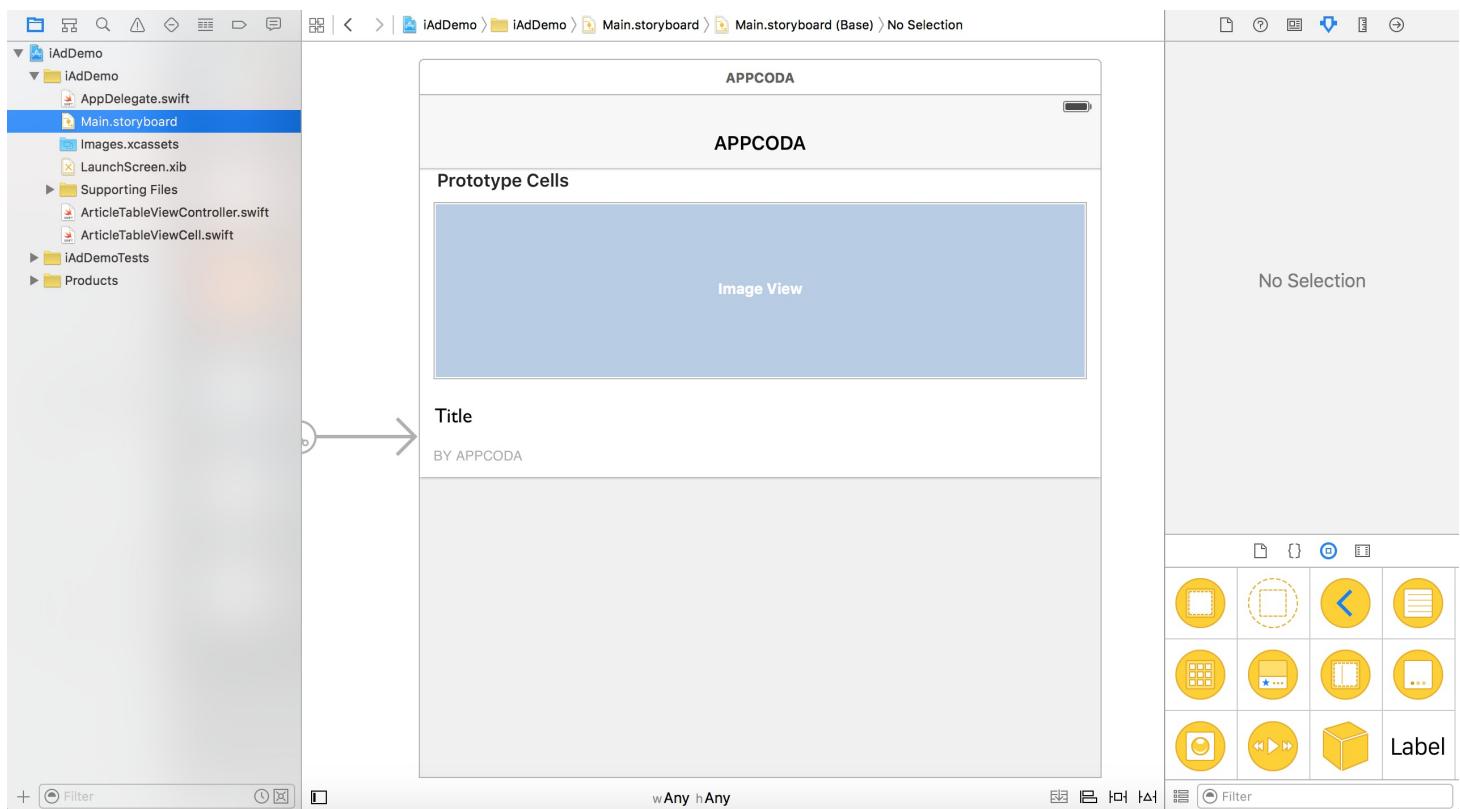


Like most developers, you're probably looking for ways to make extra money from your app. The most straightforward way is to put your app in the App Store and sell it for \$0.99 or more. This paid model works really well for some apps. But this is not the only monetization model. In this chapter, we'll discuss how to monetize your app using iAd.

iAd is an advertising platform developed by Apple for its iOS devices including the iPhone, iPod touch, and iPad. It allows developers to embed ads in their iOS apps without relying on third-party advertisers. Apple sells the advertising space (e.g. banner) within your app to a bunch of advertisers. You earn 70% of the ad revenue when a user views or clicks your ads.

What separates iAd from other banner ads is that it promises to deliver more interactive ads on mobile. When a user taps on an ad banner, a full-screen interactive ad appears on the screen. But from a developer's perspective, it provides the easiest way to deliver advertisements in an iOS app. iAd is a part of the iOS SDK so you do not need to rely on other third party libraries. All the required libraries are already bundled in Xcode. You'll discover how easy it is to integrate an iAd banner into your app in a minute. It only takes a few lines (or even a single line) of code to start making a profit from your app.

There is no better way to learn the iAd integration than by trying it out. As usual, we'll work on a sample project and then add a banner ad. The sample app is similar to the one we built in earlier chapter. You can download the Xcode project template from <https://www.dropbox.com/s/r504aapiy822jmr/iAdDemoTemplate.zip?dl=0>.

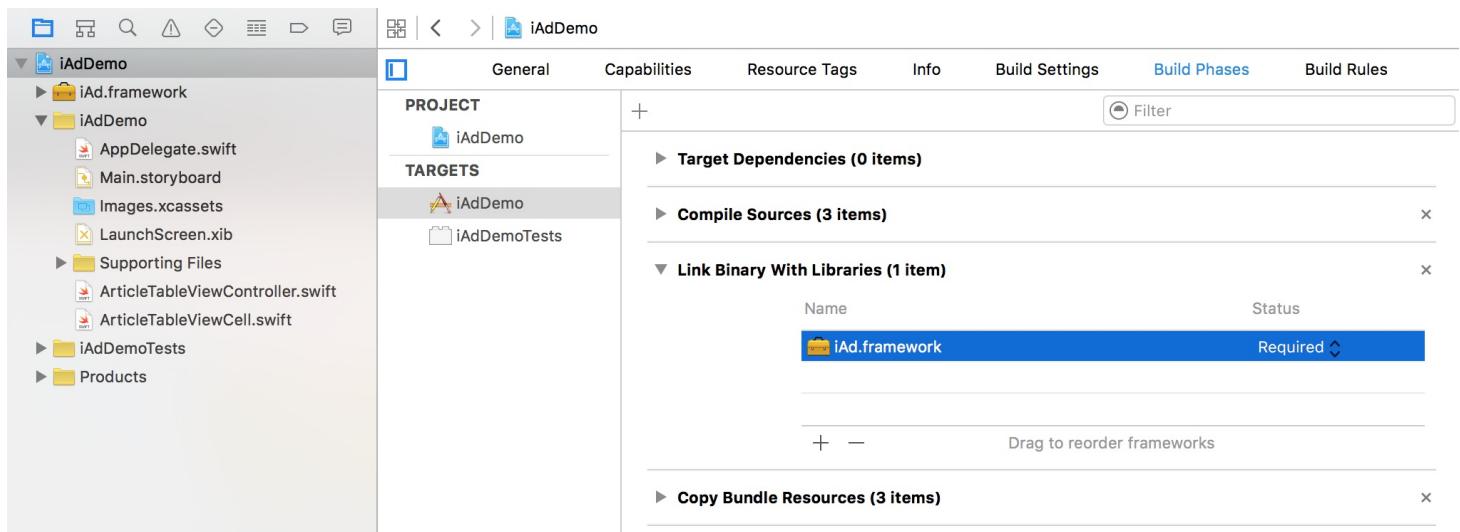


Using iAd Framework

Okay, let's get started. Fire up Xcode and open the `iAdDemo` project you just downloaded. I suggest you compile and run the project template so that you have a basic idea of the demo app; it's a simple table-based app that displays a list of articles. We will tweak it to show an advertisement to earn some extra revenue. You do not need to enroll in the Apple Developer

Program before testing iAd. The ad service can send test advertisements to help you verify that your integration is correct.

To integrate iAd into your app, the first thing you need to do is add the iAd framework into the Xcode project. Select `iAdDemo` project in the project navigator. Then select `iAdDemo target > Build Phases`. Expand `Link Binary with Libraries` by clicking the disclosure icon. To add the iAd framework, click the `+` button and search for iAd. Select `iAd.framework` and click the Add button to add the framework into the project.



In the class that uses iAd, you should add an import statement in order to import the framework. For our demo app, we will display a banner advertisement in the table view. Open the `ArticleTableViewController.swift` file and insert the following statement at the very beginning:

```
import iAd
```

Displaying Banner Ads

The iAd framework is tightly integrated with `UIViewController`. When iAd is used, `UIViewController` allows you to access additional properties that are specifically designed for iAd.

To display a banner ad at the default position, all you need to do is set the `canDisplayBannerAds` property of `UIViewController` to `true`. The controller will take care of resizing the content, grabbing the banner ad from Apple, and displaying a banner ad below the

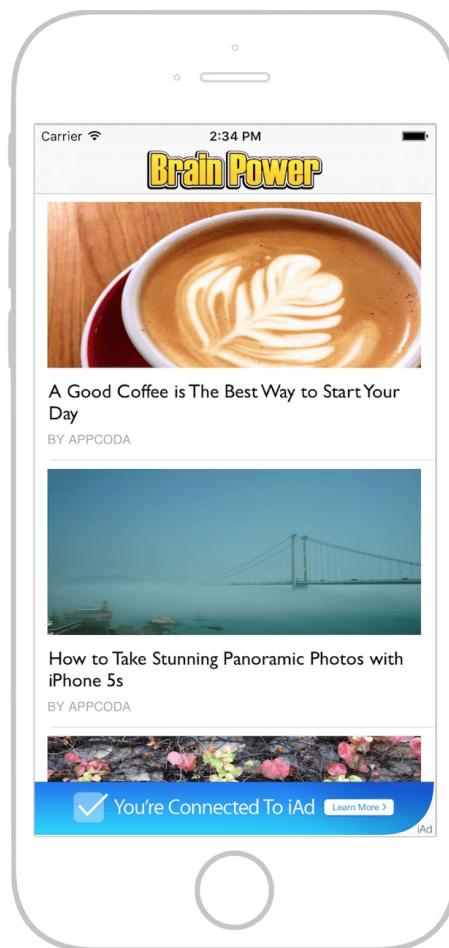
content.

Now open `ArticleTableViewController.swift` and add the following line of code in the `viewDidLoad` method:

```
canDisplayBannerAds = true
```

Just one line of code? Yes, you read it correctly. That's all you need to add an ad banner.

Once you set the property to `true`, the banner ad is automatically displayed. On top of that, the content is resized based on the size of the banner. For our demo app, you'll see that the table view has been resized appropriately. Try to run the demo app and play around with it. For testing purpose, the iAd framework will load a test ad.



Displaying Interstitial ads

Not only can you include banner ads in your apps, but `UIViewController` also lets you easily display interstitial ads (i.e. full screen ads) by changing the `ADInterstitialPresentationPolicy` property. The property is set to *none* by default, which means no interstitial ads are allowed. According to Apple's documentation, you can set the policy to *automatic* and the framework automatically displays an interstitial ad on screen. The frequency of ad presentation is, however, completely controlled by the iAd framework. You have no idea when the ad is displayed.

Alternatively, you can set the policy to *manual*. You then call the `requestInterstitialAdPresentation` method to show an interstitial ad at an appropriate time. For example, let's say you want to display an interstitial ad 30 seconds after the app is launched. We can implement it like this. First, insert the following lines of code in the `viewDidLoad` method of the `ArticleTableViewController` class:

```
interstitialPresentationPolicy = .Manual  
UIViewController.prepareInterstitialAds()
```

We simply change the ad policy to `.Manual`. Before an ad can be displayed, the app needs to fetch and download an advertisement. The second line of the code tells the framework to prepare for the ad presentation. To display an interstitial ad, we create a helper method:

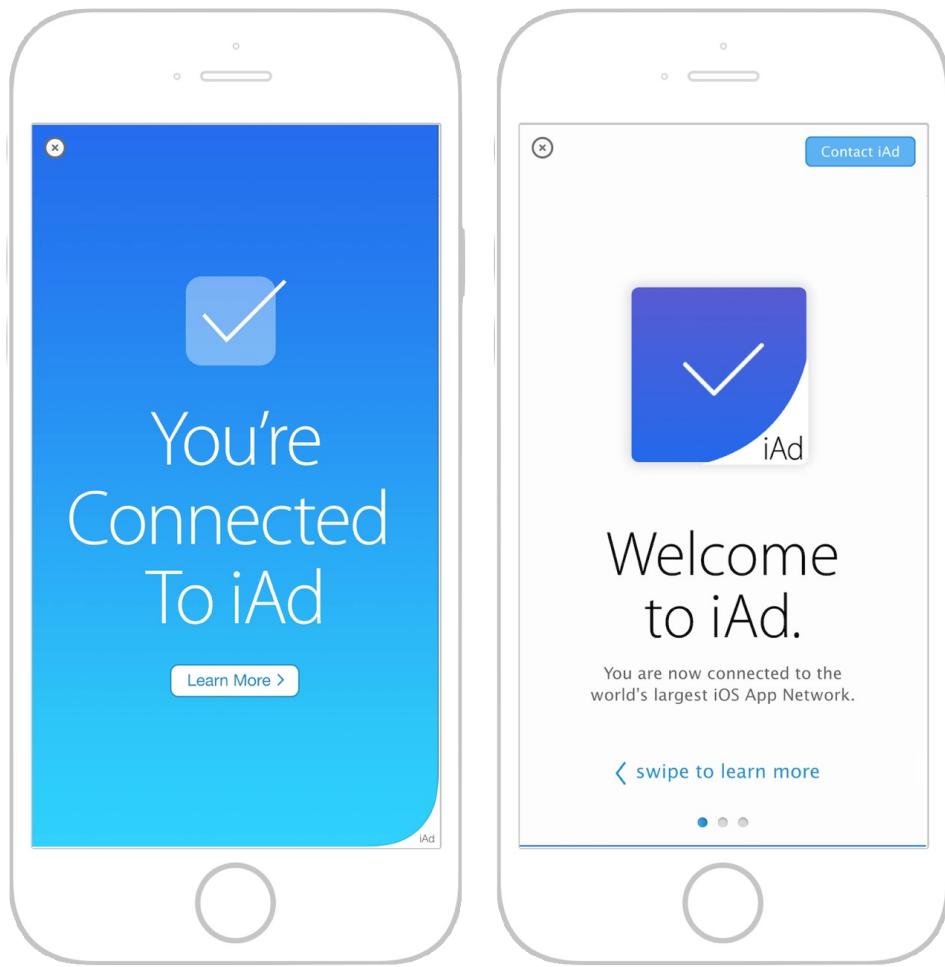
```
func displayInterstitialAds() {  
    if displayingBannerAd {  
        canDisplayBannerAds = false  
    }  
  
    requestInterstitialAdPresentation()  
    canDisplayBannerAds = true  
}
```

We first disable the current banner ad if it's active. Then we call the `requestInterstitialAdPresentation` method to request an ad. Once it's displayed, we re-enable the banner ad. To trigger an interstitial ad at a certain time, we will create and configure an `NSTimer` object. Continue to insert the following lines of code in the `viewDidLoad` method:

```
let timer = NSTimer(fireDate: NSDate(timeIntervalSinceNow: 30), interval: 0,  
target: self, selector: "displayInterstitialAds", userInfo: nil, repeats:  
false)  
NSRunLoop.currentRunLoop().addTimer(timer, forMode: NSRunLoopCommonModes)
```

The `NSTimer` class takes several parameters. Here we specify the fire date (i.e. the time at which the time is triggered) to *30 seconds since now*. When it's fired, the time will call the `displayInterstitialAds` method to display the ad. We set the repeats option to false, since we only want to display the interstitial ad once.

Now you're ready to test the app. After launching the app, wait for 30 seconds and you should see a full-screen ad.



Quick note: If you can't make it work on the simulator (e.g. iPhone 6), try to test the app on a real device or select an alternate simulator (e.g. iPhone 5s).

Display Banner Ads in Other Positions

The `canDisplayBannerAds` property provides the easiest way to display a banner ad. That said, it limits the position of banner ad to the bottom of the content. What if you want to display the

banner ad in other positions such as above the content or on the top part of the screen? In this case you can't just rely on the `canDisplayBannerAds` property; you have to create the banner ad on your own.

Indeed it's not difficult to render the banner ad with your own code. As a demonstration, we'll modify the sample project and see how we can display the banner ad as a section header of the table view. First, remove the following line code from the `viewDidLoad:` method of `iAdDemoTableViewController.swift`:

```
canDisplayBannerAds = true
```

Next, since we're going to display the banner ad using our own implementation, implement `ADBannerViewDelegate` and add the two instance variables in the `ArticleTableViewController` class:

```
class ArticleTableViewController: UITableViewController, ADBannerViewDelegate {  
  
    var adBannerView: ADBannerView?  
    var isAdDisplayed = false
```

The `ADBannerView` class provides a view to display banner ads on the screen. Later we'll use the banner view as the section header view of the table. The `iAdDisplayed` property is a Boolean variable indicating whether the ad is loaded and displayed properly. We'll use this value to determine whether we should hide/show the ad. Any events that occur during the display of a banner ad are communicated via the `ADBannerViewDelegate` protocol. Here, `ArticleTableViewController` is the delegate of the `adBannerView`. Later we'll implement some of the methods defined by the protocol to handle certain ad events (e.g the ad has loaded successfully).

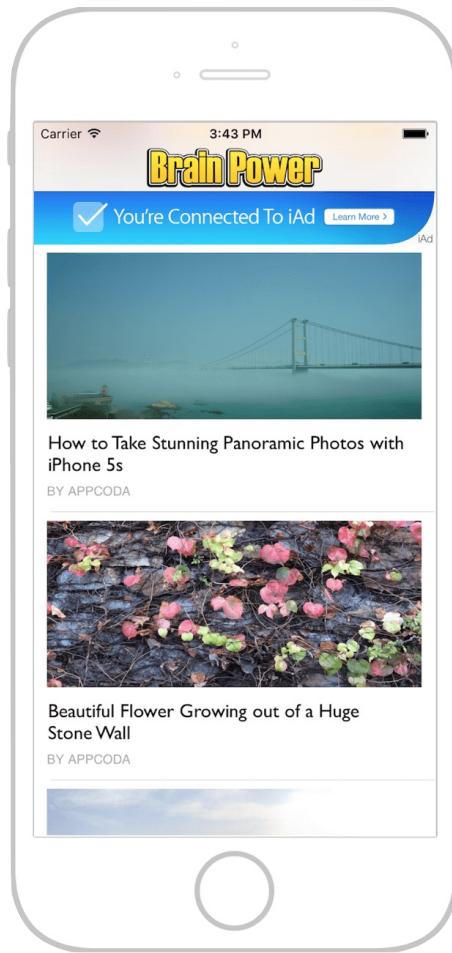
Open the `ArticleTableViewController.swift` file and add the following code in the `viewDidLoad` method. Here we initialize `adBannerView` with banner type and define its delegate:

```
adBannerView = ADBannerView(adType: ADAdType.Banner)  
adBannerView?.delegate = self
```

Next, add the following code in the same class:

```
override func tableView(tableView: UITableView, viewForHeaderInSection section: Int) -> UIView? {
    return adBannerView
}
```

This tells the table view to use the ad banner view as the section header. Now you're ready to go! Hit the Run button and test the app. Wait a few seconds and you should see the banner ad.



Show/Hide Banner Ad

Great! You've managed to change the ad position. But wouldn't it be even better if the app only displayed the banner ad when it's available? Presently, the app displays a blank ad at the time when it's first launched - it can take up to 10 seconds before the ad is shown properly.

To enhance the user experience, it would be great if the banner ad was displayed only when it has been completely loaded. Until then, we just hide the banner ad. Let's tweak the demo project and make the app even better.

So far we haven't implemented any methods of the `ADBannerViewDelegate` protocol. Actually, the banner view calls its delegate when a new ad is loaded or when an error occurs. Insert the following code in the `ArticleTableViewController` class:

```
override func tableView(tableView: UITableView, heightForHeaderInSection section: Int) -> CGFloat {
    if isAdDisplayed {
        if let bannerView = adBannerView {
            return bannerView.frame.size.height
        }
    }
    return 0
}

func bannerViewDidLoadAd(banner: ADBannerView!) {
    print("Banner ad loaded successfully")
    isAdDisplayed = true

    // Reload table section to show the banner ad
    let indexSet = NSIndexSet(index: 0)
    tableView.reloadSections(indexSet, withRowAnimation: .Automatic)
}

func bannerView(banner: ADBannerView!, didFailToReceiveAdWithError error: NSError!) {
    print("Failed to load banner ad")
    isAdDisplayed = false

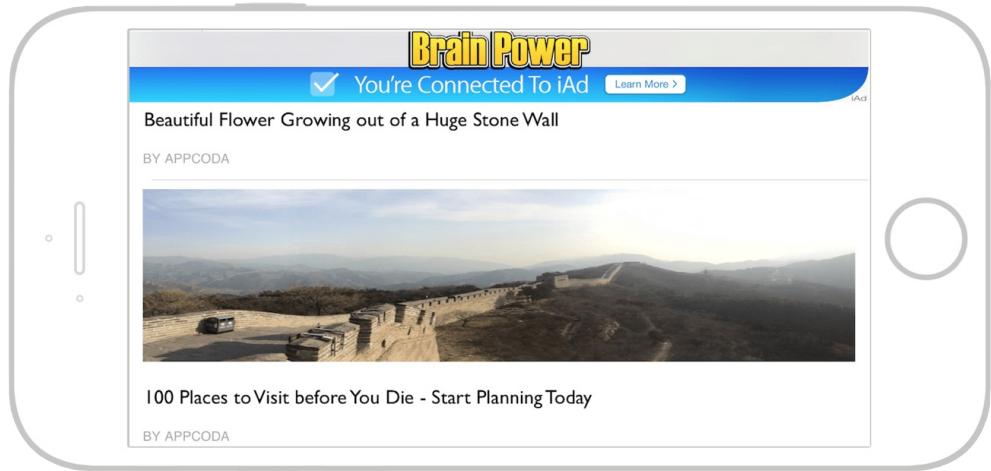
    // Reload table section to hide the banner ad
    let indexSet = NSIndexSet(index: 0)
    tableView.reloadSections(indexSet, withRowAnimation: .Automatic)
}
```

We use the `heightForHeaderInSection` method to control whether the banner ad is shown/hidden. If the ad is ready to display (i.e. `isAdDisplayed=true`), we return the height of the banner view (e.g. 50 points for ads in portrait orientation). Otherwise, the height is set to zero. By setting the height to zero, the app will not be displayed in the section header. In other words, the banner ad is hidden. The `bannerViewDidLoadAd` method is called when a new ad is successfully loaded. Here we set the `isAdDisplayed` value to `true` and tell the table view to reload the section. When we reload the section, the `heightForHeaderInSection` method will be called to show the ad.

When there are any errors loading an ad, the `didFailToReceiveAdWithError:` method is called.

We then set the `isAdDisplayed` value to `false`. Again we reload the table section to hide the ad.

Alright, the app is ready to test. When the app is first launched, the banner is not shown. Once the ad is ready, the section header is resized to display the banner. To simulate the error case, you can disable your Wi-Fi or unplug your LAN cable. Wait 10-20 seconds till the ad reloads, and you should see the message *Failed to load banner ad* shown in the console as the banner ad is automatically hidden. Try to turn your iPhone sideways; the demo app can also handle banner ads in the landscape orientation.



For your reference, you can download the complete Xcode project from
<https://www.dropbox.com/s/4yyodyltdmpnvhi/iAdDemo.zip?dl=0>.

Chapter 16

Working with Custom Fonts



When you add a label to a view, Xcode allows you to change the font type using the Attribute inspector. From there, you can pick a system font or custom font from the pre-defined font family.

What if you can't find any font from the default font family that fits into your app? Typography is an important aspect of app design. Proper use of a typeface makes your app superior, so you may want to use some custom fonts that are created by third parties but not bundled in Mac OS. Just perform a simple search on Google and you'll find tons of free fonts for app development. However, this still leaves you with the problem of bundling the font in your Xcode project. You may think that we can just add the font file into the project, but it's a little more difficult than that. In this chapter, I'll focus on how to bundle new fonts and go through the procedures with you.

As always, I'll give you a demo and build the demo app together. The demo app is very simple; it just displays a set of labels using different custom fonts. You can start by building the demo from scratch or downloading the template from

<https://www.dropbox.com/s/knfrdrwaxmyggtk/CustomFontTemplate.zip?dl=0>.

Download Custom Fonts

We'll use a few fonts that are available for free. However, we're not allowed to bundle and distribute them via our project template. Before proceeding, download the following fonts via the below links:

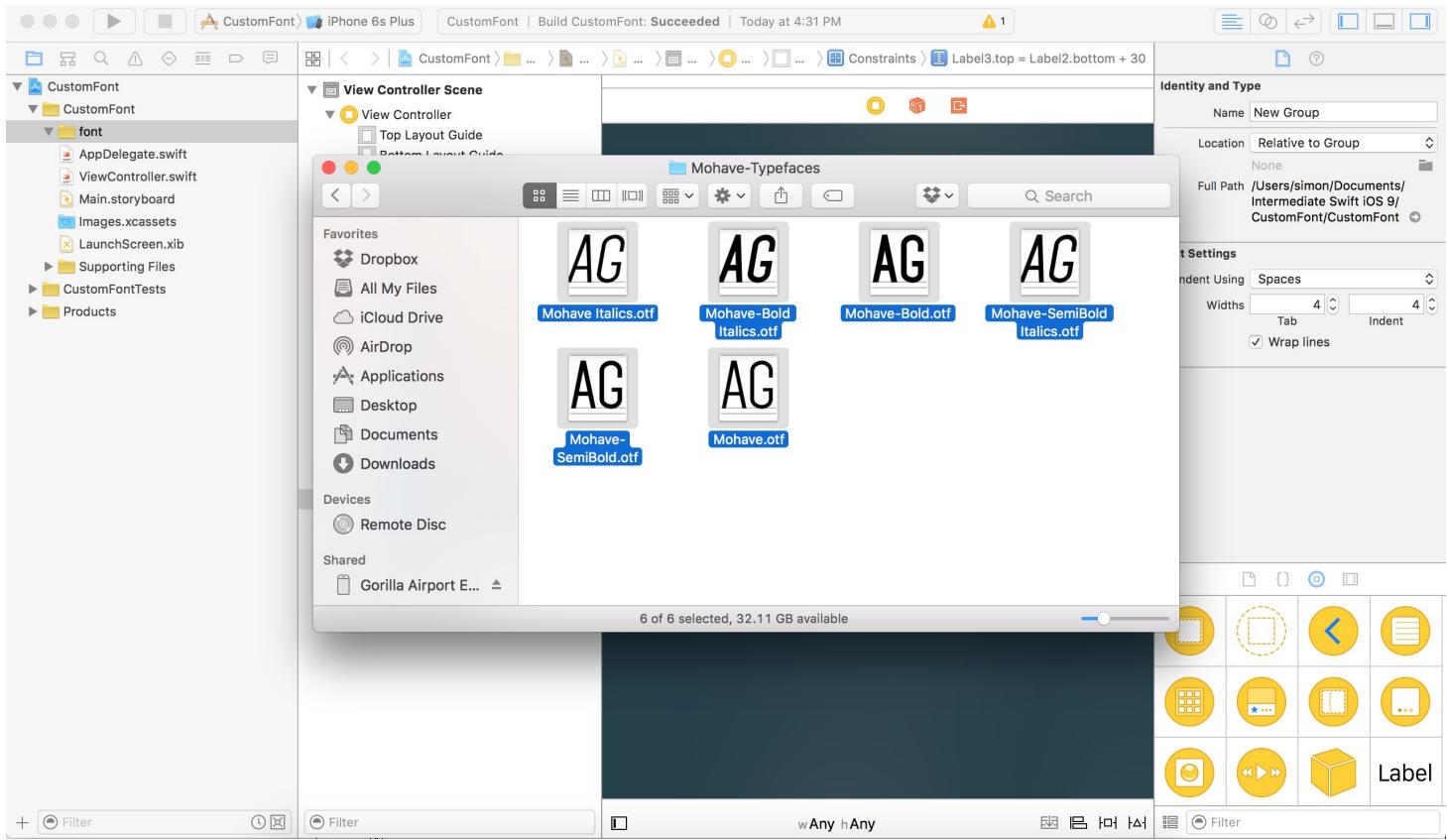
- <https://dribbble.com/shots/1371629-Mohave-Typefaces?list=users&offset=3>
- <http://fredrikstaurland.com/hallo-sans-free-font/>
- <http://fontfabric.com/canter-free-font/>

Alternatively, you can use any fonts that you own for the project. Or you are free to use some of the beautifully designed fonts from:

- <http://www.awwwards.com/100-greatest-free-fonts-collection-for-2015.html>
- <http://creativeshory.com/75-best-free-fonts-2014/>
- <http://www.awwwards.com/100-greatest-free-fonts-collection-for-2013.html>
- <http://www.creativebloq.com/graphic-design-tips/best-free-fonts-for-designers-1233380>

Adding Font Files to the Project

Just like any other resource file (e.g. image), you have to first add the font files to your Xcode project. I like to keep all the resource files under a *font* folder. In the project navigator, right click the *CustomFont* folder and select *New Group* to create a folder. Change the name to *font*. Then drag the font files that you have downloaded into the folder.



When Xcode prompts you for confirmation, make sure to check the box of your targets (i.e. CustomFont) and enable the *Copy items if needed* option. This instructs Xcode to copy the font files to your app's folder. If you have this option unchecked, your Xcode project will only add a reference to the font files.

Choose options for adding these files:

Destination: Copy items if needed

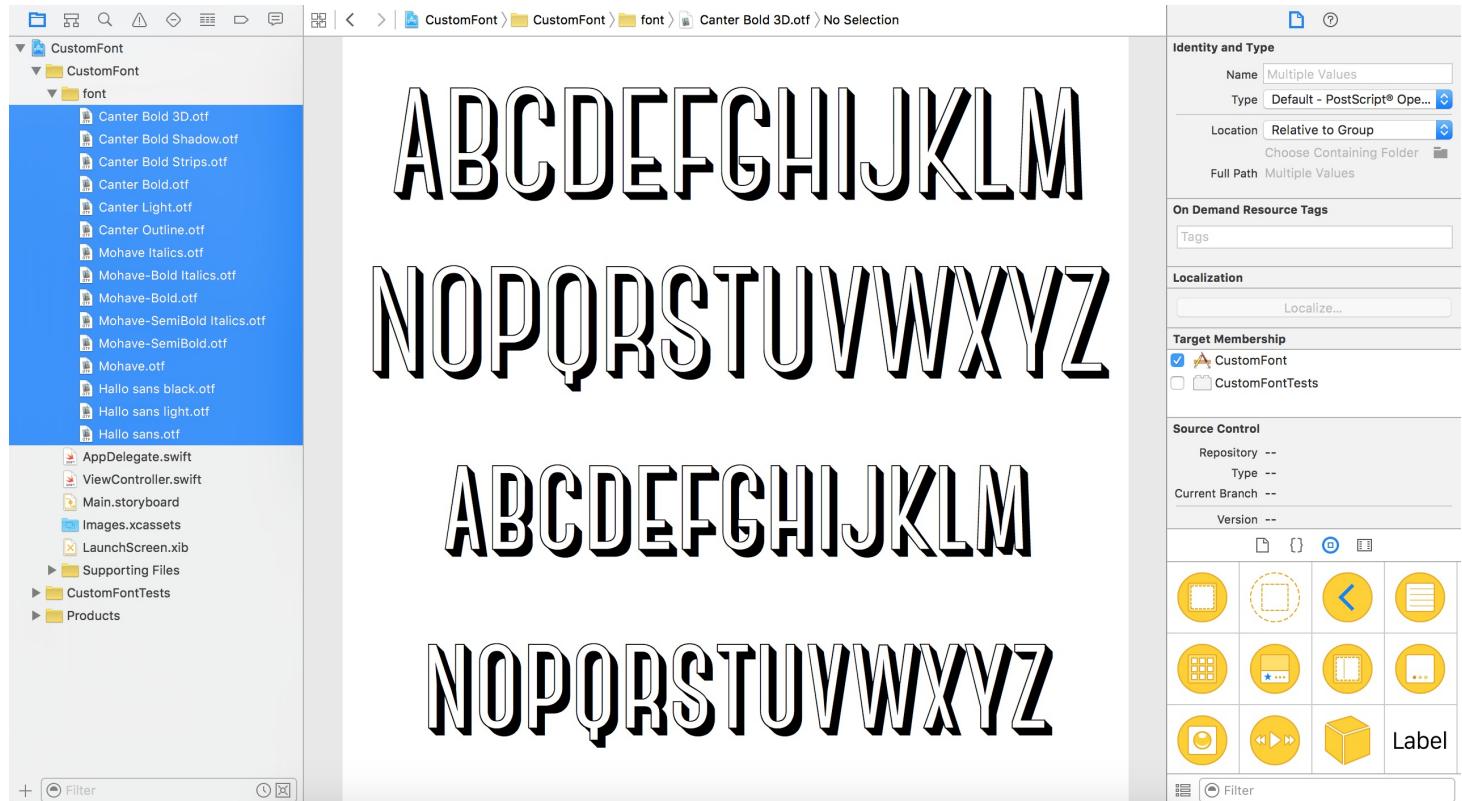
Added folders: Create groups
 Create folder references

Add to targets:  CustomFont
  CustomFontTests

Cancel

Finish

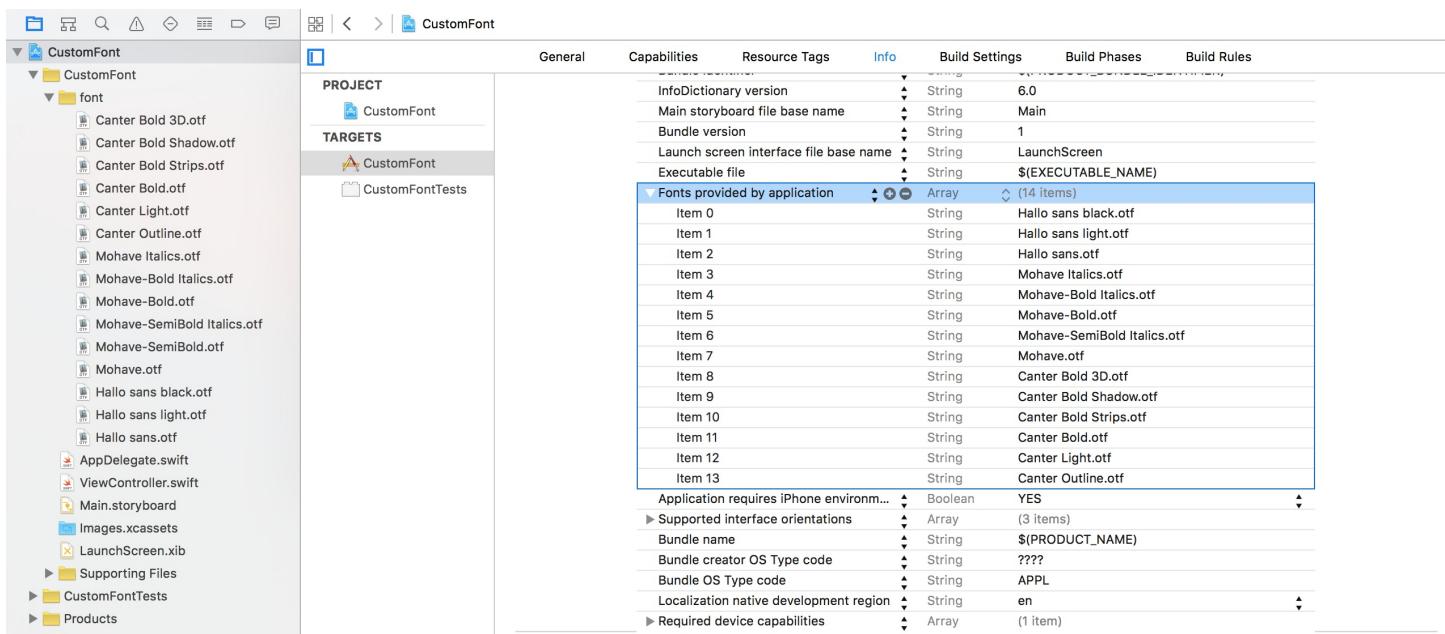
The font files are usually in *.ttf* or *.otf* format. Once you add all the files, you should find them in the project navigator under the *font* folder.



Register the Fonts in the Project Info Settings

Before using the font faces, you have to register them in the project settings. Select the CustomFont project in the project navigator and then select CustomFont under Targets. Under the Info tab, add a new property named *Fonts provided by application*. This is an array key that allows you to register the custom font files.

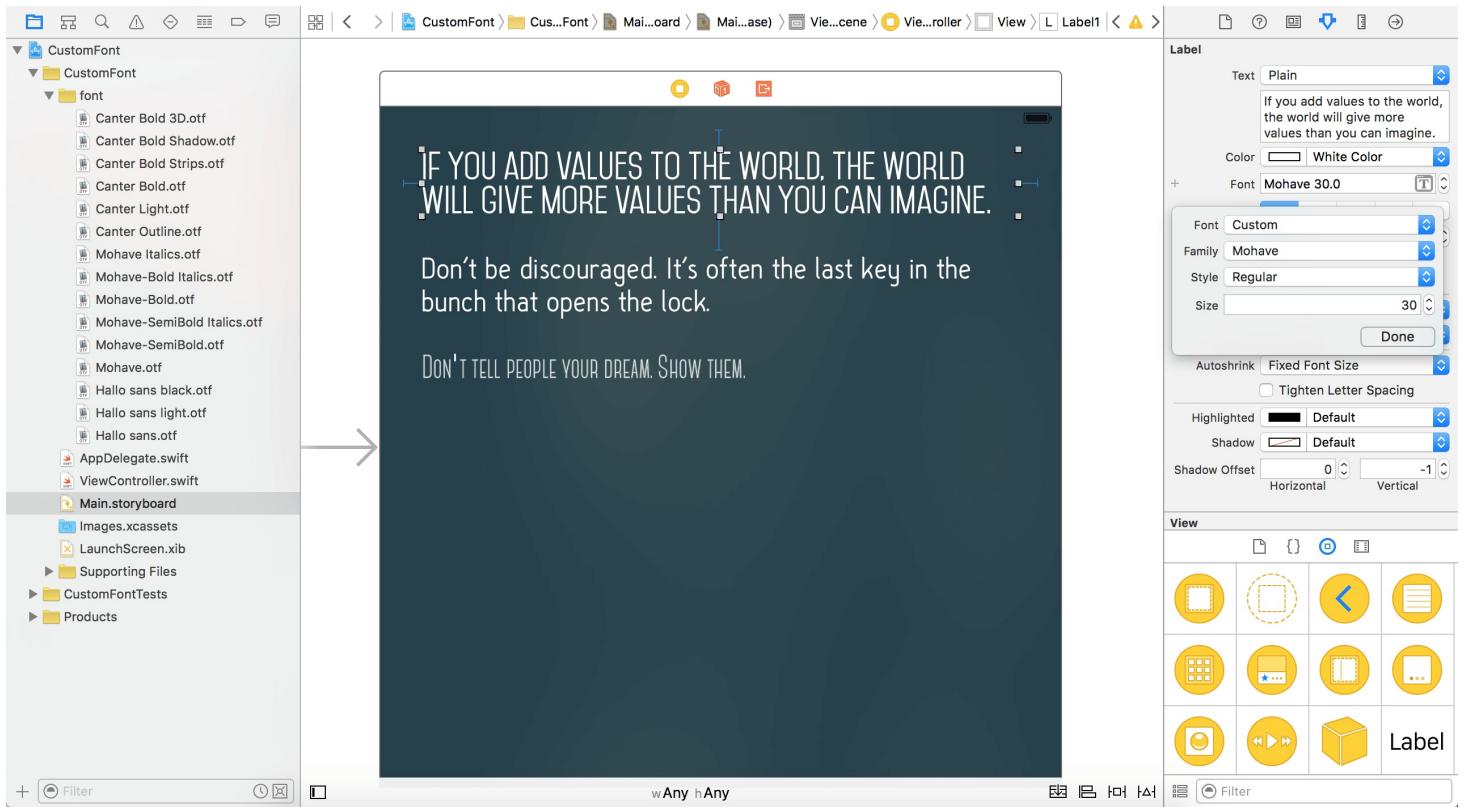
Right click one of the keys and select *Add Row* from the context menu. Scroll and select *Fonts provided by application* from the dropdown menu. Click the disclosure icon (i.e. triangle) to expand the key. You should see Item 0. Double click the value field and enter *Haloo sans black.otf*. Then click the + button next to Item 0 to add another font file. Repeat the same step until all the font files are registered - you'll end up with a screenshot like the one shown below. Make sure you key in the file names correctly. Otherwise, you won't be able to use the fonts.



Using Custom Fonts in Interface Builder

You can embed custom fonts in older version of Xcode. However, you can't see a preview of how the finished app will look in Interface Builder; you can only see the resulting design by testing the app in the simulator. Xcode 7 allows developers to preview the fonts in Interface Builder. Any custom fonts added to your project will be made available in Interface Builder. You can change the font of an object (e.g. label) in the Attributes inspector and Interface

Builder will render the result in real-time.



Using Custom Fonts in Code

Alternatively, you can use the custom font through code. Simply instantiate a `UIFont` object with your desired custom font and assign it to a UI object such as `UILabel`. Here is an example:

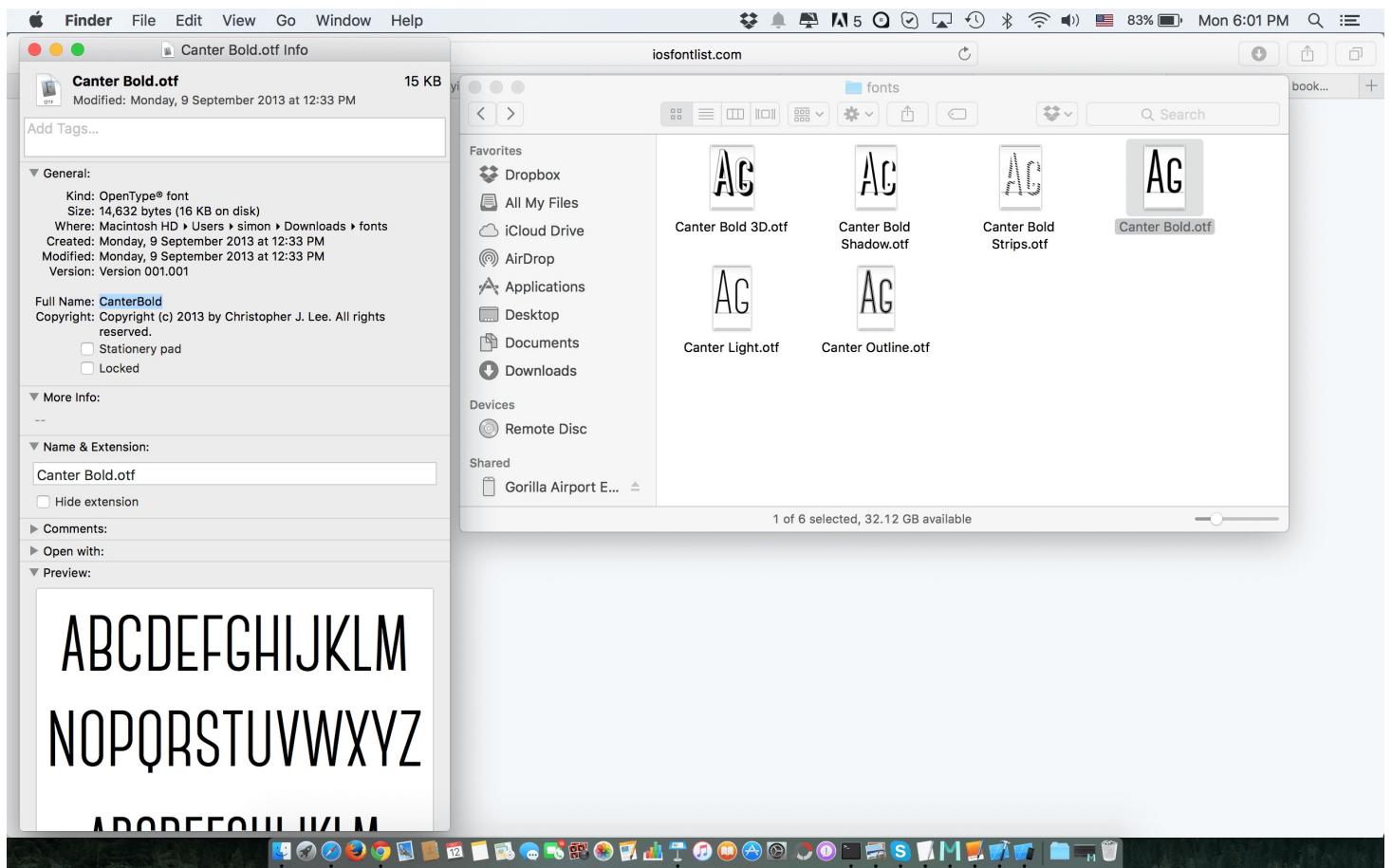
```
label1.font = UIFont(name: "Mohave-Italic", size: 25.0)
label2.font = UIFont(name: "Hallo sans", size: 30.0)
label3.font = UIFont(name: "CanterLight", size: 35.0)
```

If you insert the above code in the `viewDidLoad` method of the `ViewController` class, and run the app, all the labels should change to the specified custom fonts accordingly.

For starters, you may have a question in your mind: *how can you find out the font name?* It seems that the font names differ from the file names.

That's a very good observation. When initializing a `UIFont` object, you should specify the font

name instead of the filename of the font. To find the name of the font, you can right-click a font file in Finder and select *Get Info* from the context menu. The value displayed in the Full Name field is the font name used in UIFont. In the sample screenshot, the font name is *CanterBold*.



For your reference, you can download the complete project from
<https://www.dropbox.com/s/aj9qf428wvx5sbo/CustomFont.zip?dl=0>.

Chapter 17

Working with AirDrop and UIActivityViewController



AirDrop is Apple's answer to file and data sharing. Prior to iOS 7, users had to rely on third-party apps like Bump to share files between iOS devices. Since the release of iOS 7, iOS users are allowed to use a feature called AirDrop to share data with nearby iOS devices. In brief, the feature allows you to share photos, videos, contacts, URLs, Passbook passes, app listings on the App Store, media listings on iTunes Store, location in Maps, etc.

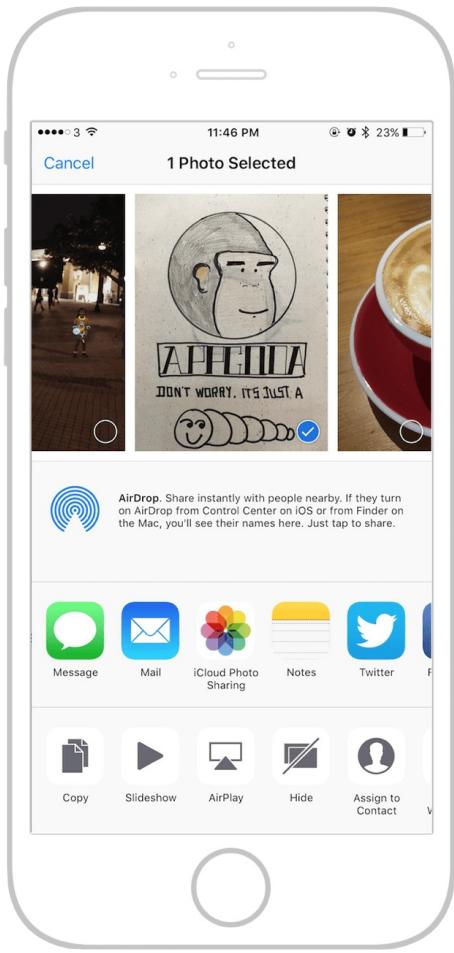
Wouldn't it be great if you could integrate AirDrop into your app? Your users could easily share

photos, text files, or any other type of document with nearby devices. The `UIActivityViewController` class bundled in the iOS SDK makes it easy for you to embed AirDrop into your apps. The class shields you from the underlying details of file sharing. All you need to do is tell the class the objects you want to share and the controller handles the rest. In this chapter, we'll demonstrate the usage of `UIActivityViewController` and see how to use it to share images and documents via AirDrop.

To activate AirDrop, simply bring up Control Center and tap AirDrop. Depending on whom you want to share the data with, you can either select *Contact Only* or *Everyone*. If you choose the *Contact Only* option, your device will only be discovered by people listed in your contacts. If the Everyone option is selected your device can be discovered from any other device.

AirDrop uses Bluetooth to scan for nearby devices. When a connection is established via Bluetooth, it will create an ad-hoc Wi-Fi network to link the two devices together, allowing for faster data transmission. This doesn't mean you need to connect the devices to a Wi-Fi network in order to use AirDrop; your WiFi just needs to be on for the data transfer to occur.

For example, let's say you want to transfer a photo in the Photos app from one iPhone to another. Assuming you have enabled AirDrop on both devices, you can share the photo with another device by tapping the Share button (the one with an arrow pointing up) in the lower-left corner of the screen. In the AirDrop area, you should see the name of the devices that are eligible for sharing. AirDrop is not available when the screen is turned off, so make sure the device on the receiving side is switched on. You can then select the device with which you want to share the photo. On the receiving device, you should see a preview of the photo and a confirmation request. The recipient can accept or decline to receive the image. If they choose the accept option, the photo is then transferred and automatically saved in their camera roll.



AirDrop doesn't just work with the Photos app. You can also share items in your Contacts, iTunes, App Store, and Safari browser, to name a few. If you're new to AirDrop, you should now have a better idea of how it works.

Let's see how we can integrate AirDrop into an app to share various types of data.

UIActivityViewController Overview

You might think it would take a hundred lines of code to implement the AirDrop feature. Conversely, you just need a few lines of code to embed AirDrop. The `UIActivityViewController` class provided by the UIKit framework streamlines the integration process.

The `UIActivityViewController` class is a standard view controller that provides several standard services, such as copying items to the clipboard, sharing content to social media sites, sending items via Messages, etc. Since iOS 7, the class adds the support of AirDrop sharing. In iOS 8 or later, the activity view controller adds the support of app extensions. However, we will

not discuss it in this chapter.

The class is very simple to use. Let's say you have an array of objects to share using AirDrop. All you need to do is create an instance of `UIActivityViewController` with the array of objects and then present the controller on screen. Here is the code snippet:

```
let objectsToShare = [fileURL]
let activityController = UIActivityViewController(activityItems:
objectsToShare, applicationActivities: nil)
presentViewController(activityController, animated: true, completion: nil)
```

As you can see, with just three lines of code you can bring up an activity view with the AirDrop option. Whenever there is a nearby device detected, the activity controller automatically displays the device and handles the data transfer if you choose to share the data. By default, the activity controller includes sharing options such as Messages, Flickr and Sina Weibo. Optionally, you can exclude these types of activities by setting the `excludedActivityTypes` property of the controller. Here is the sample code snippet:

```
let excludedActivities = [UIActivityTypePostToWeibo, UIActivityTypeMessage,
UIActivityTypePostToTencentWeibo]
activityController.excludedActivityTypes = excludedActivities
```

You can use `UIActivityViewController` to share different types of data including `String`, `UIImage`, and `NSURL`. Not only you can use `NSURL` to share a link, but it also allows developers to transfer any type of files by using the file URL.

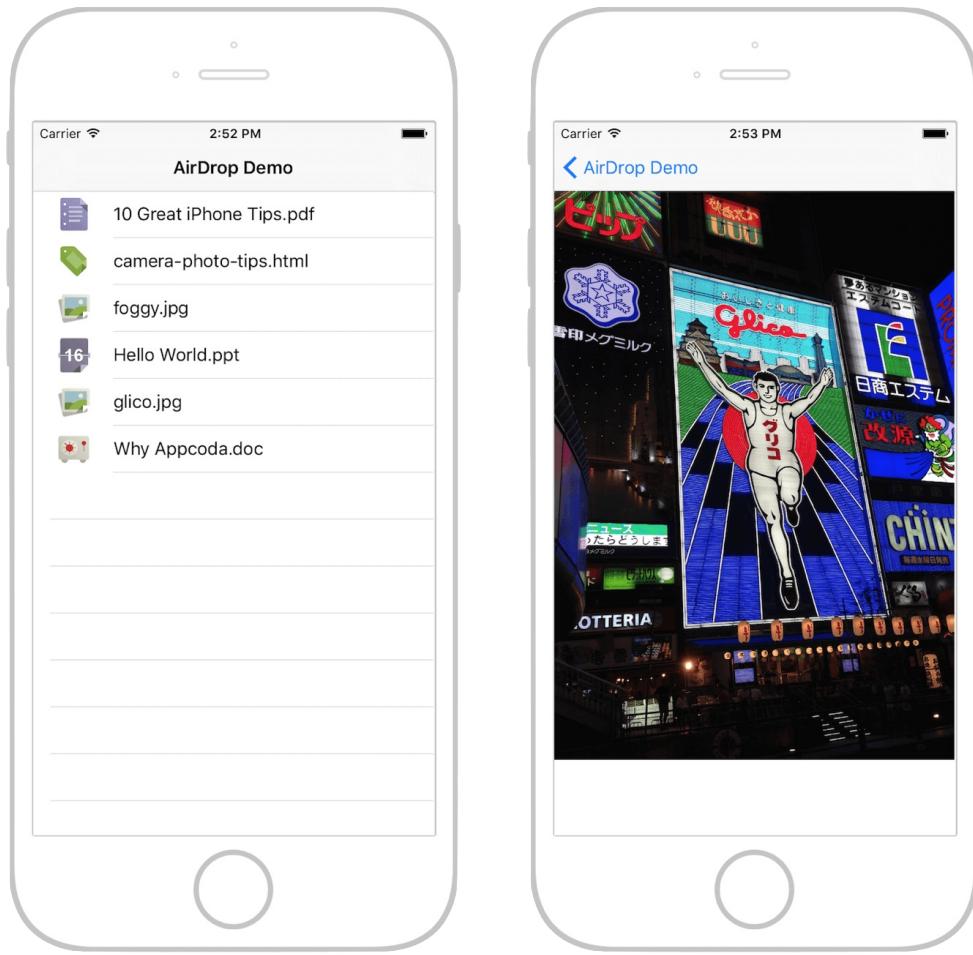
When the other device receives the data, it will automatically open an app based on the data type. So, if a `UIImage` object is transferred, the received image will be displayed in the Photos app. When you transfer a PDF file, the other device will open it in Safari. If you just share a `String` object, the data will be presented in the Notes app.

Demo App

To give you a better idea of `UIActivityViewController` and AirDrop, we'll build a demo app as usual. Once again, the app is very simple. When it is launched, you'll see a table view listing a few files including image files, a PDF file, a document, and a Powerpoint. You can tap a file and view its content in the detail view. In the content view, there is a toolbar at the bottom of the

screen. Tapping the Share action button in the toolbar will bring up the AirDrop option for sharing the file with a nearby device.

To keep you focused on implementing the AirDrop feature, you can download the project template from <https://www.dropbox.com/s/74hripl2eefprkbq/AirdropDemoTemplate.zip?dl=0>. After downloading the template, open it and have a quick look.

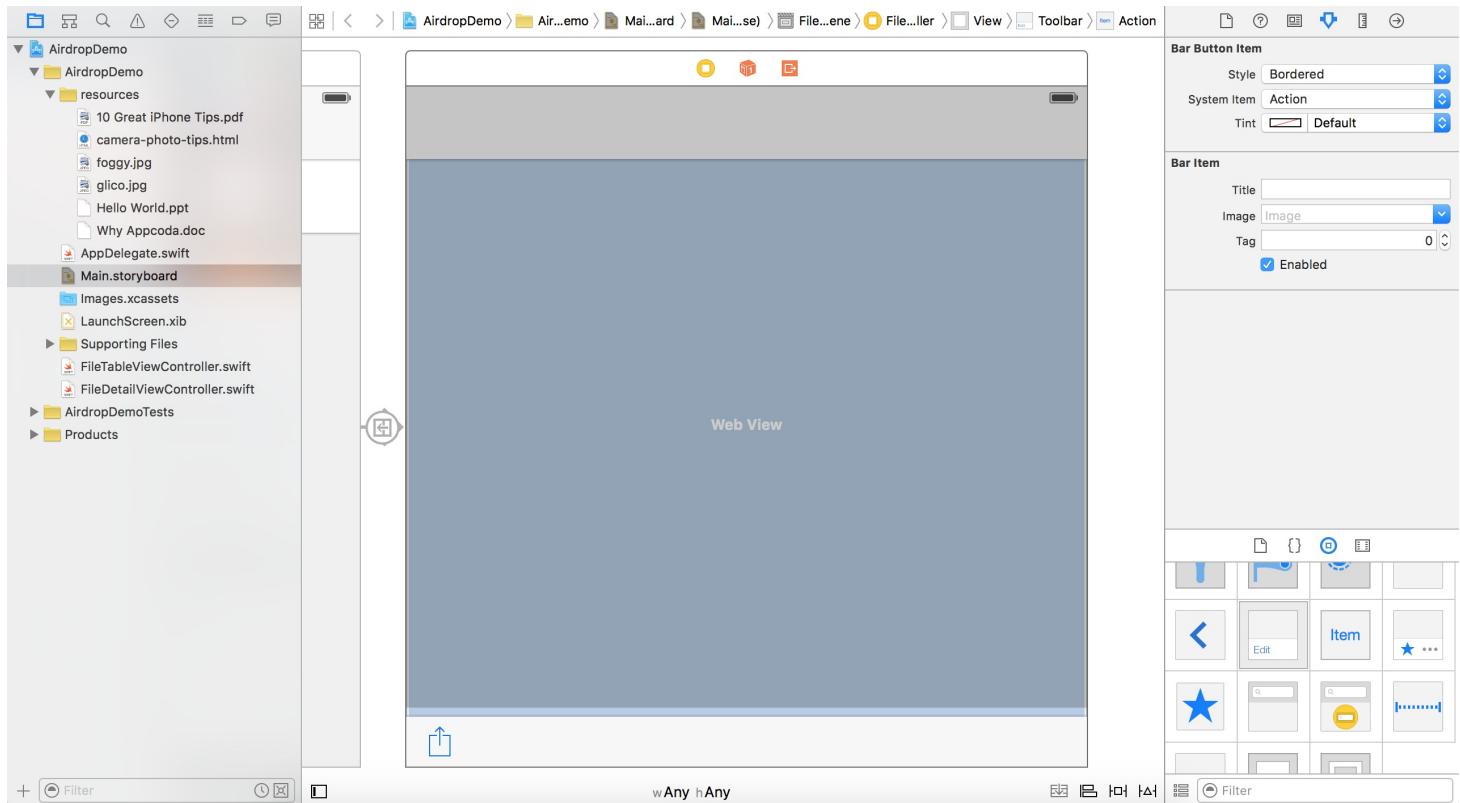


The project template already includes the storyboard and the custom classes. The table view controller is associated with `FileTableViewController`, while the detail view controller is connected with `FileDetailViewController`. The `FileDetailViewController` object simply makes use of `UIWebView` to display the file content. What we are going to do is add a *Share* button in the detail view to activate AirDrop.

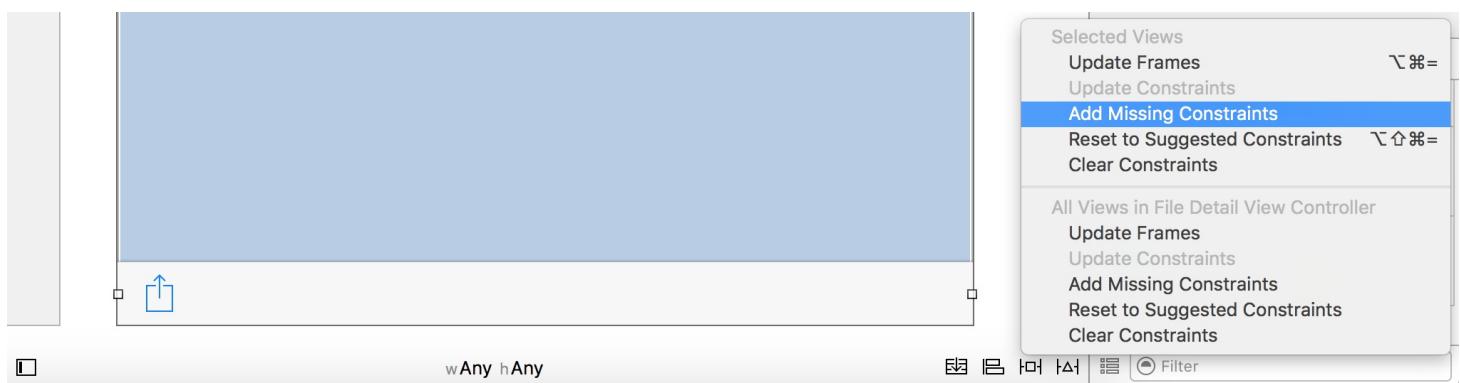
Let's get started.

Adding a Share Button in Interface Builder

First, let's go to the storyboard. Drag a toolbar from the Object library to the detail view controller and place it at the bottom part of the controller. Select the default bar button item and change its identifier to `Action` in the Attributes inspector. Your screen should look like this:



Next you'll need to add some layout constraints for the toolbar, otherwise, it will not be properly displayed on some devices. In the layout bar, click the Issues button. Choose the *Add Missing Constraints* option under the *Selected Views* section. Interface Builder will then generate the layout constraints for the toolbar.



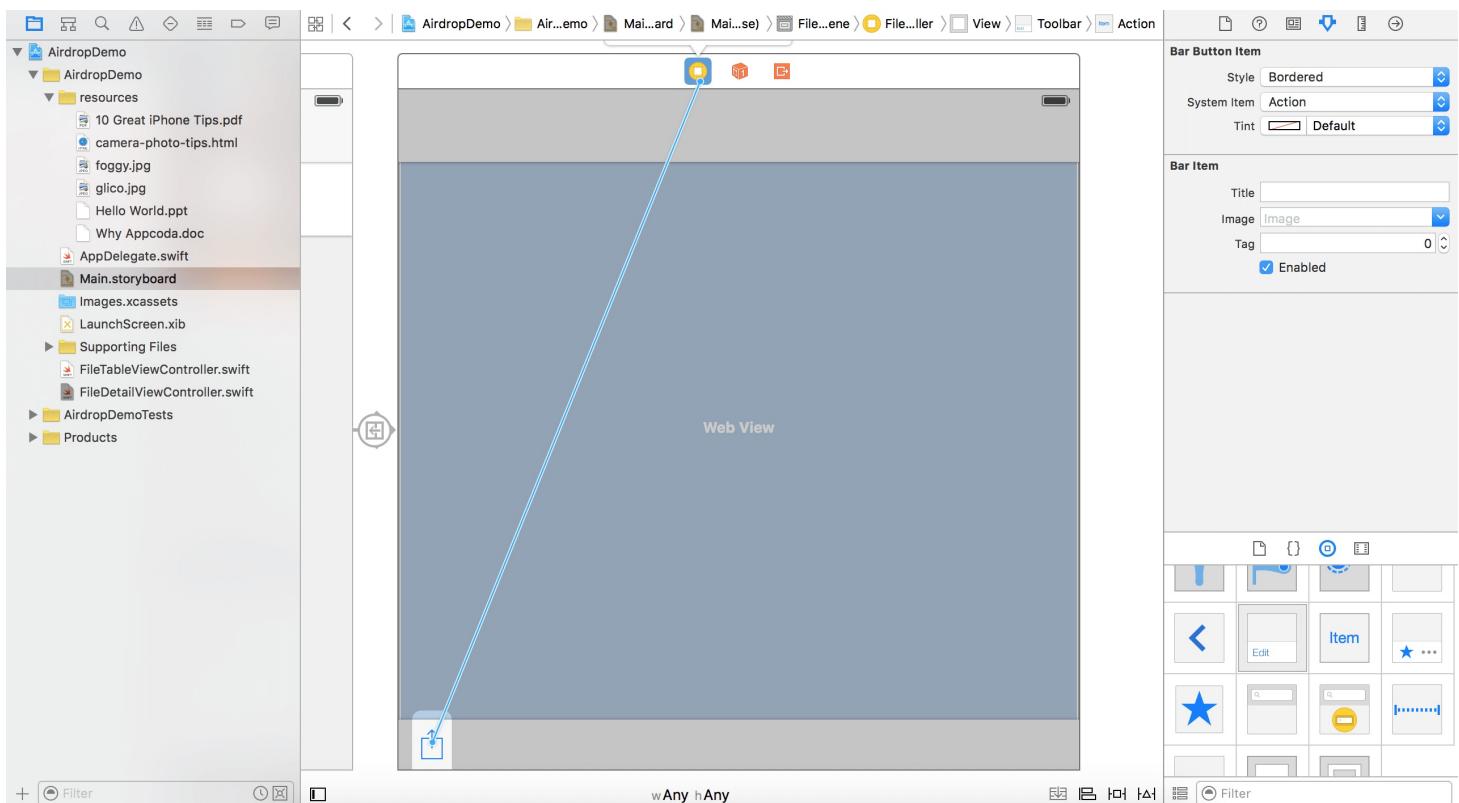
The constraints added ensure the toolbar is always displayed at the bottom part of the view

controller. And the leading and trailing edges are aligned with the view.

Now go back to `FileDetailViewController` and add an action method for the *Share* action:

```
@IBAction func share(sender: AnyObject) {  
}
```

Go back to `Main.storyboard` and connect the *Share* button with the action method. Control-drag from the *Share* button to the view controller icon of the scene dock, and select `share:` from the pop-up menu.



Implementing AirDrop for File Sharing

Now that you have completed the UI design, we will move on to the coding part. Update the `share` method of the `FileDetailViewController` class to the following:

```
@IBAction func share(sender: AnyObject) {  
    if let fileURL = fileToURL(filename) {  
        let objectsToShare = [fileURL]  
        let activityController = UIActivityViewController(activityItems:  
objectsToShare, applicationActivities: nil)
```

```

        let excludedActivities = [UIActivityTypePostToFlickr,
UIActivityTypePostToWeibo, UIActivityTypeMessage, UIActivityTypeMail,
UIActivityTypePrint, UIActivityTypeCopyToPasteboard,
UIActivityTypeAssignToContact, UIActivityTypeSaveToCameraRoll,
UIActivityTypeAddToReadingList, UIActivityTypePostToFlickr,
UIActivityTypePostToVimeo, UIActivityTypePostToTencentWeibo]

        activityController.excludedActivityTypes = excludedActivities
        presentViewController(activityController, animated: true, completion:
nil)
    }
}

```

The above code should be very familiar to you; we discussed it at the very beginning of the chapter. The code creates an instance of `UIActivityViewController`, excludes some of the activities and presents the controller on the screen. The tricky part is how you define the objects to share.

The `filename` property of `FileDetailViewController` contains the file name to share. We need to first find the full path of the file before passing it to the activity view controller. In the project template, I already include a helper method for this purpose:

```

func fileToURL(file: String) -> NSURL? {
    // Get the full path of the file
    let fileComponents = file.componentsSeparatedByString(".")

    if let filePath = NSBundle mainBundle().pathForResource(fileComponents[0],
ofType: fileComponents[1]) {
        return NSURL(fileURLWithPath: filePath)
    }

    return nil
}

```

The code is very straightforward. For example, the image file `glico.jpg` will be transformed to `file:///Users/simon/Library/Application%20Support/iPhone%20Simulator/8.1/Applications/A5321493-318A-4A3B-8B37-E56B8B4405FC/AirdropDemo.app/glico.jpg`. The file URL varies depending on the device you're running. But the URL should begin with the `file://` protocol. With the file URL object, we create the corresponding array and pass it to `UIActivityViewController` for AirDrop sharing.

Build and Run the AirDrop Demo

That's all you need to implement AirDrop sharing. You're now ready to test the app. Compile and run it on a real iPhone. Yes, you need a real device to test AirDrop sharing; the sharing feature will not work in a simulator. Furthermore, you need to have at least two iOS devices or a Mac running Mac OS Yosemite to test the sharing feature.

Once you launch the app, select a file, tap the Share action button, and enable AirDrop. Make sure the receiving device has AirDrop enabled. The app should recognize the receiving device for file transfer.



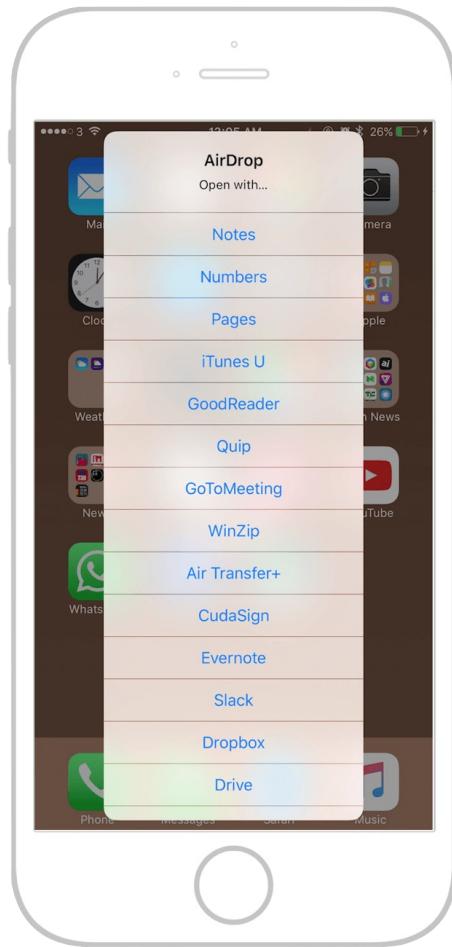
For reference, you can download the complete Xcode project from
<https://www.dropbox.com/s/l1sv18bumf12frh/AirdropDemo.zip?dl=0>.

Uniform Type Identifiers

When you share an image with another iOS device, the receiving side automatically opens the Photos app and saves the image. If you transfer a PDF or document file, the receiving device

may prompt you to pick an app for opening the file or open it directly in iBooks. How can iOS know which app to use for a particular data type?

UTIs (short for Uniform Type Identifiers) is Apple's answer to identifying data within the system. In brief, a uniform type identifier is a unique identifier for a particular type of data or file. For instance, com.adobe.pdf represents a PDF document and public.png represents a PNG image. You can find the full list of registered UTIs here. An application that is capable of opening a specific type of file will be registered to handle that UTI with the iOS. So whenever that type of file is opened, iOS hands off that file to the specific app.



The system allows multiple apps to register the same UTI. In this case, iOS will prompt user with the list of capable apps for opening the file. For example, when you share a document, the receiving device will prompt a menu for user's selection.

Summary

AirDrop is a very handy feature, which offers a great way to share data between devices. Best of all, the built-in `UIActivityViewController` has made it easy for developers to add AirDrop support in their apps. As you can see from the demo app, you just need a few lines of code to implement the feature. I highly recommend you to integrate this sharing feature into your app.

Chapter 18

Building Grid Layouts Using Collection Views



If you have no idea about what grid-like layout is, just take a look at the built-in Photos app. The app presents photos in grid format. Before Apple introduced `UICollectionView`, you had to write a lot of code or make use of third-party libraries to build a similar layout.

`UICollectionView`, in my opinion, is one of the most spectacular APIs in the iOS SDK. Not only can it simplify the way to arrange visual elements in a grid layout, it even lets developers customize the layout (e.g. circular, cover flow style layout) without changing the data.

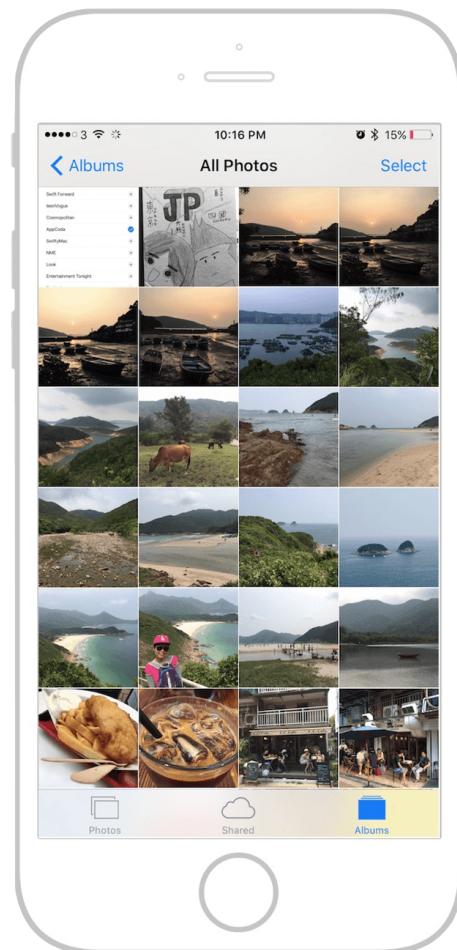
In this chapter, we will build a simple app to display a collection of recipe photos in grid layout. Here is what you're going to learn:

- An introduction to `UICollectionView`
- How to use `UICollectionView` to build a simple grid-based layout
- How to customize the background of a collection view cell

Let's get started.

Getting Started with UICollectionView and UICollectionViewController

`UICollectionView` operates pretty much like the `UITableView` class. While `UITableView` manages a collection of data items and displays them on screen in a single-column layout, the `UICollectionView` class offers developers the flexibility to present items using customizable layouts. You can present items in multi-column grids, tiled layout, circular layout, etc.



By default, the SDK comes with the `UICollectionViewFlowLayout` class that organizes items into a grid with optional header and footer views for each section. Later, we'll use the layout class to build the demo app.

The UICollectionView is composed of several components:

- Cells – instances of `UICollectionViewCell`. Like `UITableViewCell`, a cell represents a single item in the data collection. The cells are the main elements organized by the associated layout. If `UICollectionViewFlowLayout` is used, the cells are arranged in a grid-like format.
- Supplementary views – Optional. It's usually used for implementing the header or footer views of sections.
- Decoration views – think of it as another type of supplementary view but for decoration purpose only. The decoration view is unrelated to the data collection. We simply create decoration views to enhance the visual appearance of the collection view.

Like the table view, you have two ways to implement a collection view. You can add a collection view (UICollectionView) to your user interface and provide an object that conforms to the `UICollectionViewDataSource` protocol. The object is responsible for providing and managing the data associated with the collection view.

Alternatively, you can add a collection view controller from the Object library to your storyboard. The collection view controller has a view controller with a collection view built-in and provides a default implementation of the `UICollectionViewDataSource` protocol.

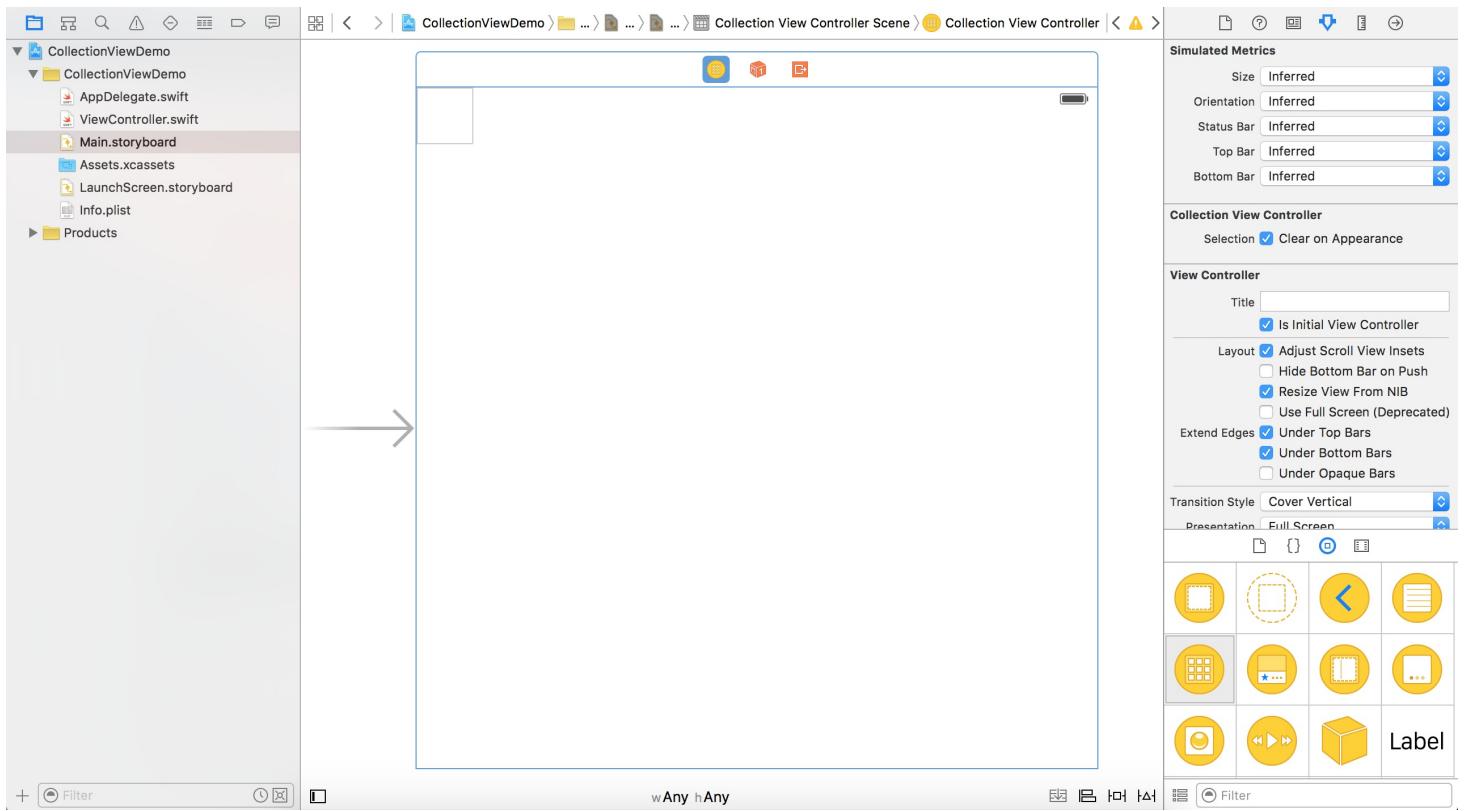
For the demo project, we will use the latter approach.

Creating a New Project

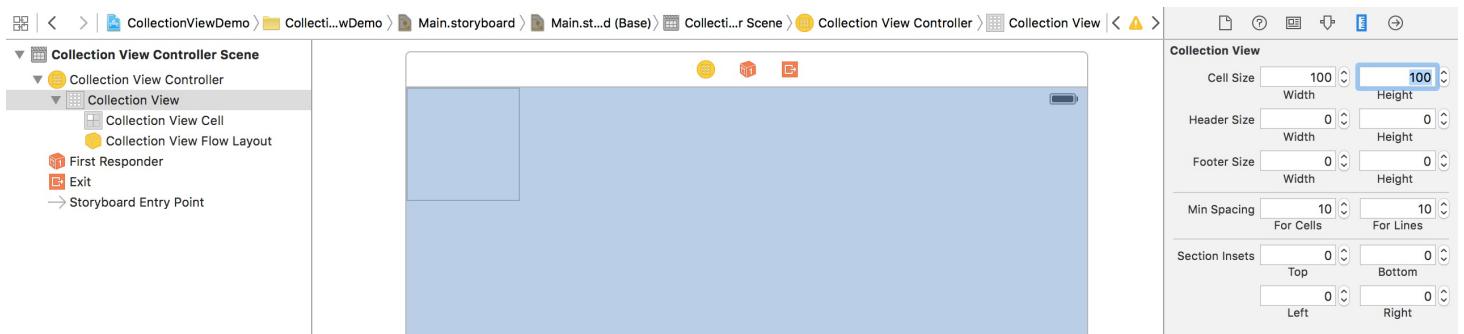
First, fire up Xcode and create a new project using the Single View Application template. Name the project `CollectionViewDemo`, set the device to iPhone and make sure you select Swift for the programming language. Once you've created the project, open `Main.storyboard` in the project navigator. Delete the default view controller and drag a Collection View Controller from the Object library to the storyboard. The controller already has a collection view built-in. You should see a collection view cell in the controller, which is similar to the prototype cell of a

table view.

Under the Attributes inspector, set the collection view controller as the initial view controller.



Next, open the Document Outline and select the collection view. Under the Size inspector, change the width and height of the cell to 100 points. Also change the min spacing of both *for cells* and *for lines* to 10 points.



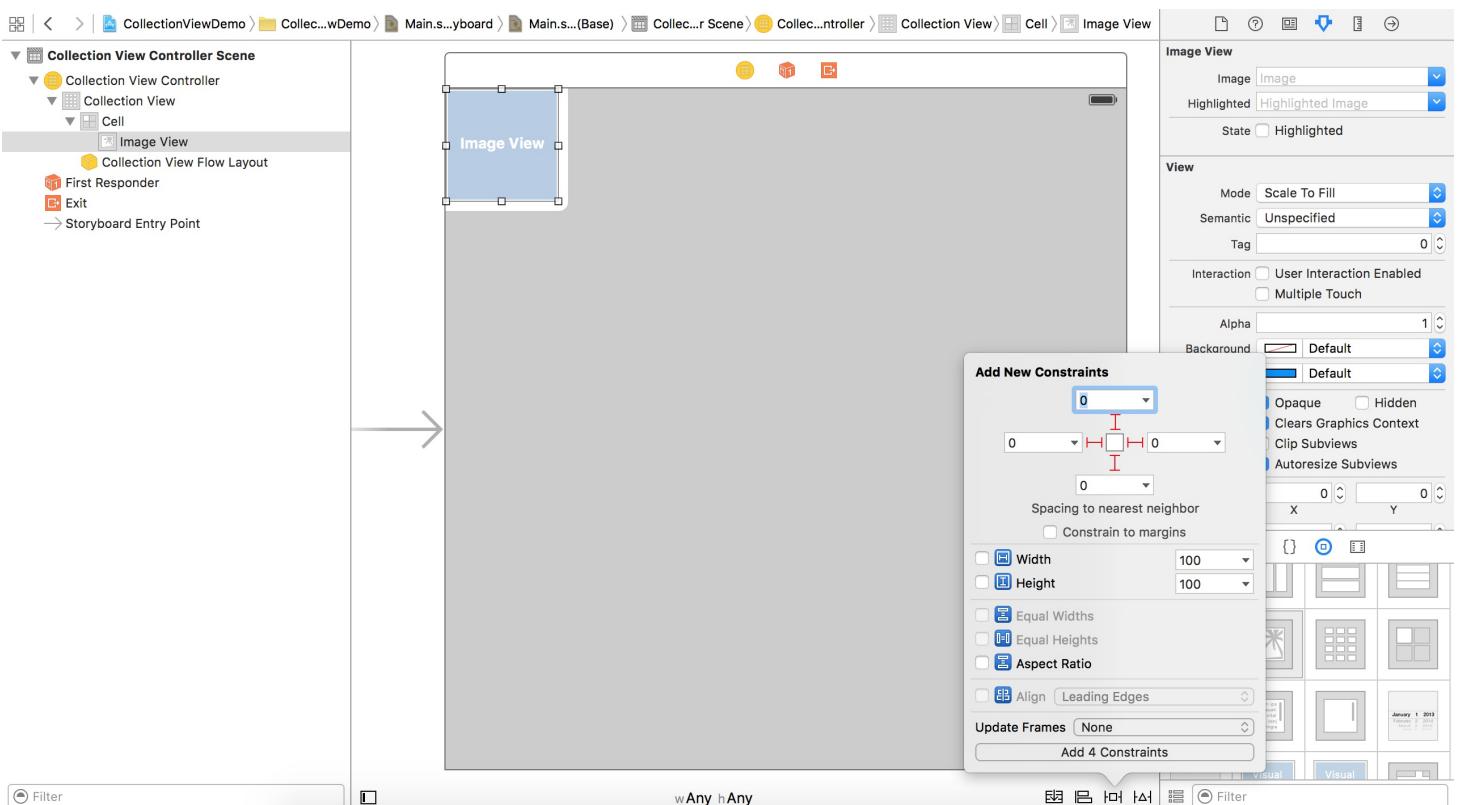
The *for cells* value defines the minimum spacing between items in the same row, while the *for lines* value defines the minimum spacing between successive rows.

Next, select the collection view cell and set the identifier to `cell` in the Attribute inspector.

This looks familiar, right?



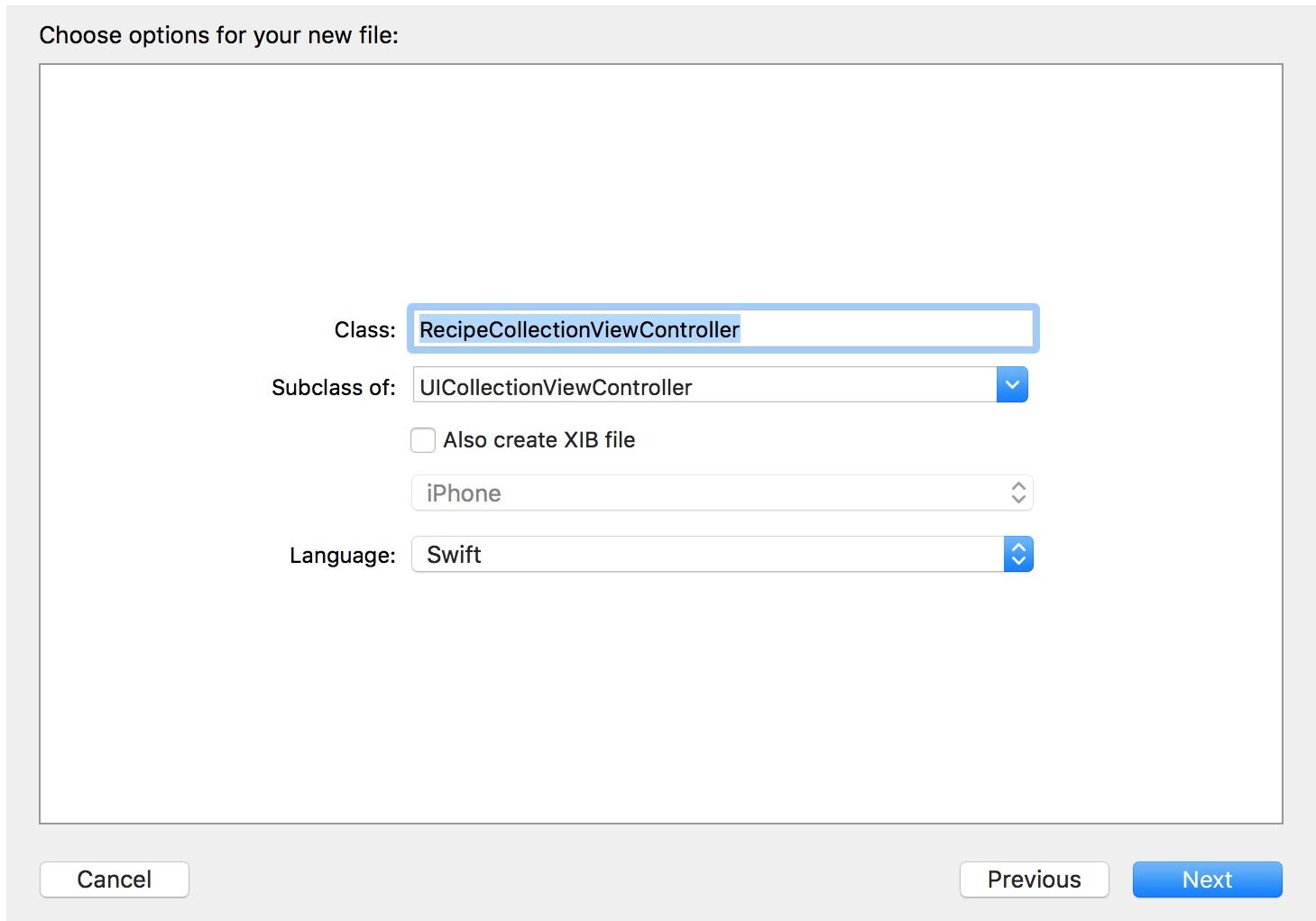
Now drag an image view from the Object library to the cell. You then manually resizes the image view and makes it fit the cell. As usual, you will need to add some layout constraints for the image view. Select the image view, click the *Pin* button in the layout bar and add 4 spacing constraints (refer to the figure below for details).



Lastly, embed the collection view controller in a navigation controller. Go up to the Xcode menu, select *Editor > Embed In > Navigation Controller*. Set the title of the navigation bar to `Recipes`. That's it. We have completed the user interface design. The next step is to create the custom classes for the collection view controller and the collection view cell.

Creating Custom Classes for the Collection View

In the project navigator, delete ViewController.swift file that was generated by Xcode. We do not need it because we will create our own classes. Right click the *CollectionViewDemo* folder and select `New File...`. Create a new class using the Cocoa Touch Class template. Name the class `RecipeCollectionViewCell` and set the subclass to `UICollectionViewCell`.



Repeat the process to create another class. Name the new class `RecipeCollectionViewController` and set the subclass to `UICollectionViewController`. Now open the `RecipeCollectionViewCell.swift` file and insert the following line of code to declare an outlet variable for the image view. Your class should look like this:

```
class RecipeCollectionViewCell: UICollectionViewCell {  
    @IBOutlet var recipeImageView: UIImageView!  
}
```

Go back to storyboard and select the collection view cell. Under the Identity inspector, change the custom class to `RecipeCollectionViewCell`. Then right click the cell and connect the outlet

variable with the image view.



Next, select the collection view controller (Recipes). Under the Identity inspector, set the custom class to `RecipeCollectionViewController`.

Implementing the Collection View Controller

As mentioned before, `UICollectionView` operates very similarly to `UITableView`. To populate data in a table view, all you have to do is implement two methods defined in the `UITableViewDataSource` protocol. Like `UITableView`, the `UICollectionViewDataSource` protocol defines a number of data source methods to interact with the collection view. You have to implement at least two methods:

- `func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int`
- `func collectionView(_ collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell`

First, download this image pack

(<https://www.dropbox.com/s/971c43jqbu9w4uj/RecipePhotoImagePack.zip?dl=0>), unzip it and add all the images to the image asset.

Next, declare a `recipeImages` array in the `RecipeCollectionViewController` class:

```
var recipeImages = ["angry_birds_cake", "creme_brelee", "egg_benedict",
"full_breakfast", "green_tea", "ham_and_cheese_panini", "ham_and_egg_sandwich",
"hamburger", "instant_noodle_with_egg.jpg", "japanese_noodle_with_pork",
"mushroom_risotto", "noodle_with_bbq_pork", "starbucks_coffee",
"thai_shrimp_cake", "vegetable_curry", "white_chocolate_donut"]
```

By default, Xcode generates a statement in the `viewDidLoad` method to register a collection view cell for reuse purpose. Since we already use a prototype cell in storyboard, this line of code is no longer required. Remove it from the `viewDidLoad` method:

```
self.collectionView!.registerClass(UICollectionViewCell.self,  
forCellWithReuseIdentifier: reuseIdentifier)
```

Similar to what you did when implementing a table view, update these data source methods of the `UICollectionViewDataSource` protocol to the following:

```
override func numberOfSectionsInCollectionView(collectionView:  
UICollectionView) -> Int {  
    return 1  
}  
  
override func collectionView(collectionView: UICollectionView,  
numberOfItemsInSection section: Int) -> Int {  
    return recipeImages.count  
}  
  
override func collectionView(collectionView: UICollectionView,  
cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell {  
    let cell =  
collectionView.dequeueReusableCellWithIdentifier(reuseIdentifier,  
forIndexPath: indexPath) as! RecipeCollectionViewCell  
  
    // Configure the cell  
    cell.recipeImageView.image = UIImage(named: recipeImages[indexPath.row])  
  
    return cell  
}
```

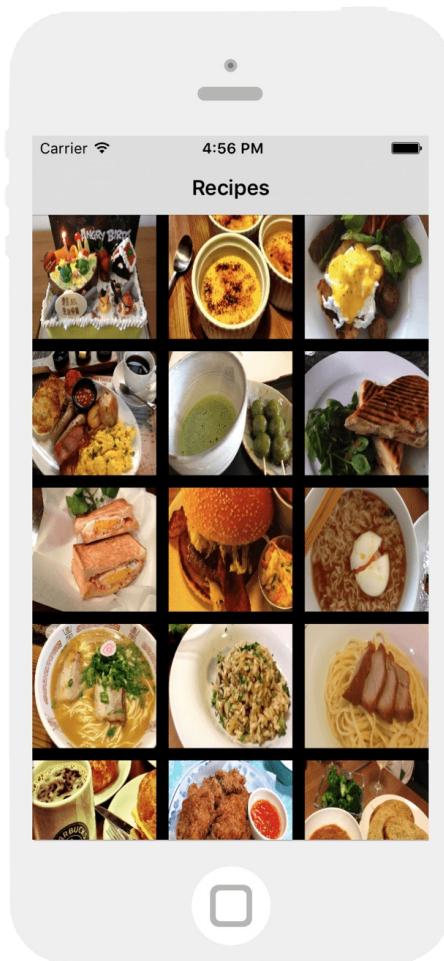
The `numberOfSectionsInCollectionView` method returns the total number of sections. In this case, we only have one section in the collection view. The

`collectionView(_:numberOfItemsInSection:_)` method returns the total number of recipe images.

The `collectionView(_:cellForItemAtIndexPath:_)` method manages the data for the collection view cells. We first define a cell identifier and then request `collectionView` to dequeue a reusable cell using the cell's identifier. The `dequeueReusableCellWithIdentifier` method will either automatically create a cell or return a cell from the re-use queue. The cell returned is

downcast to the type `RecipeCollectionViewCell`. Finally, we get the corresponding recipe image and assign it to the image view to display.

Now compile and run the app using the iPhone 5/5s simulator. You should have a grid-based photo app like this.



Quick note: If you run the app on iPhone 6/6 Plus simulator, the result will be slightly difference. I will explain it in the later chapter.

Customizing the Collection Cell Background

Cool, right? With a few lines of code, you can create a grid-based photo app. What if you want to add a picture frame to the photos? Like other UI elements, `UICollectionViewCell` lets developers easily customize its background.

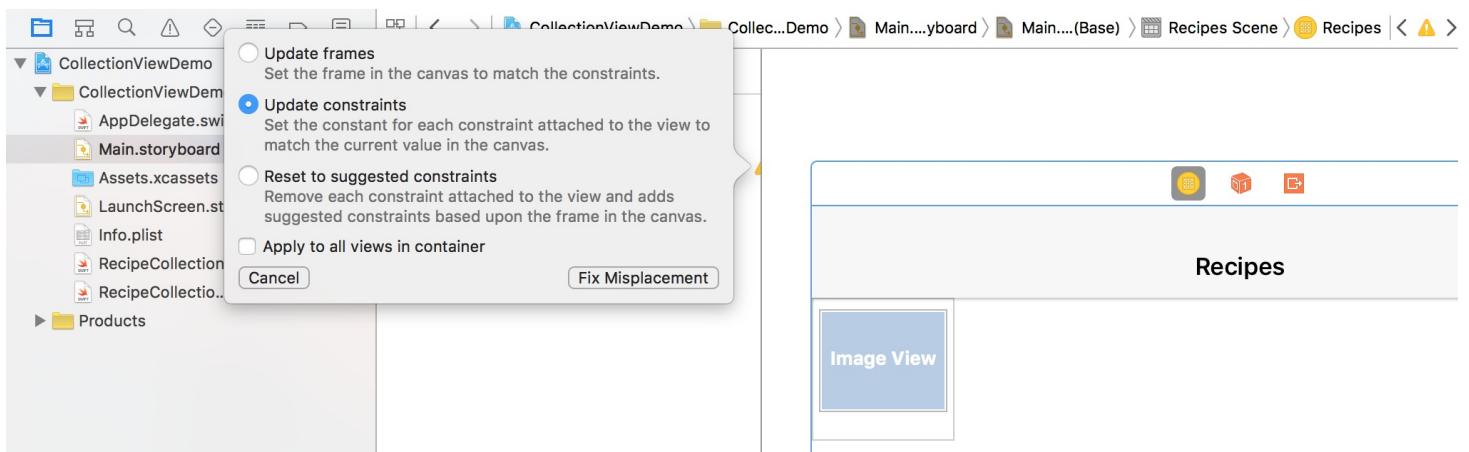
A collection view cell is comprised of three different views including background, selected

background and content view:

- Background View – background view of the cell
- Selected Background View – the background view when the cell is selected. When the user selects the cell, this selected background view will be layered above the background view.
- Content View – obviously, it's the cell content.

We have used the content view to display the recipe image. What we are going to do is use the background view to display a picture frame. In the image pack you downloaded earlier, it includes a file named *photo_frame.png*, which is the picture frame. The size of the frame is 100 by 100 pixel. In order to frame the recipe photo, we'll first resize the image view of the cell and change its position. Go to `Main.storyboard` and select the image view. Under the Size inspector, set X to 5 and Y to 8. The width and height should be changed to 90 and 72 pixels respectively.

As the size of the image view is changed, the existing layout constraints no longer apply. In the Document Outline, you should notice a yellow arrow indicating that there are some issues with the constraints. Click the arrow and then click the yellow indicator to bring up a menu. Select *Update constraints* and click *Fix Misplacement* to update the constraints.



Next, go back to the `RecipeCollectionViewController.swift` file. In the `collectionView(_:cellForItemAtIndexPath:)` method, insert the following line of code before `return cell :`

```
cell.backgroundView = UIImageView(image: UIImage(named: "photo-frame"))
```

We simply load the photo frame image and set it as the background view of the collection view

cell. Now compile and run the app again. Your app now displays a photo frame for each of the cell item.



For reference, you can download the Xcode project from
<https://www.dropbox.com/s/o1anhgdxmoxz1p/CollectionViewDemo.zip?dl=0>.

Chapter 19

Interacting with Collection Views



In the previous chapter I covered the basics of `UICollectionView`. You should now understand how to display items in a collection view. However, you may not know how to interact with the collection view cells. As mentioned before, a collection view works pretty much like a table view. To give you a better idea, I will show you how to interact with the collection view cells, specially about how to handle single and multiple item selections.

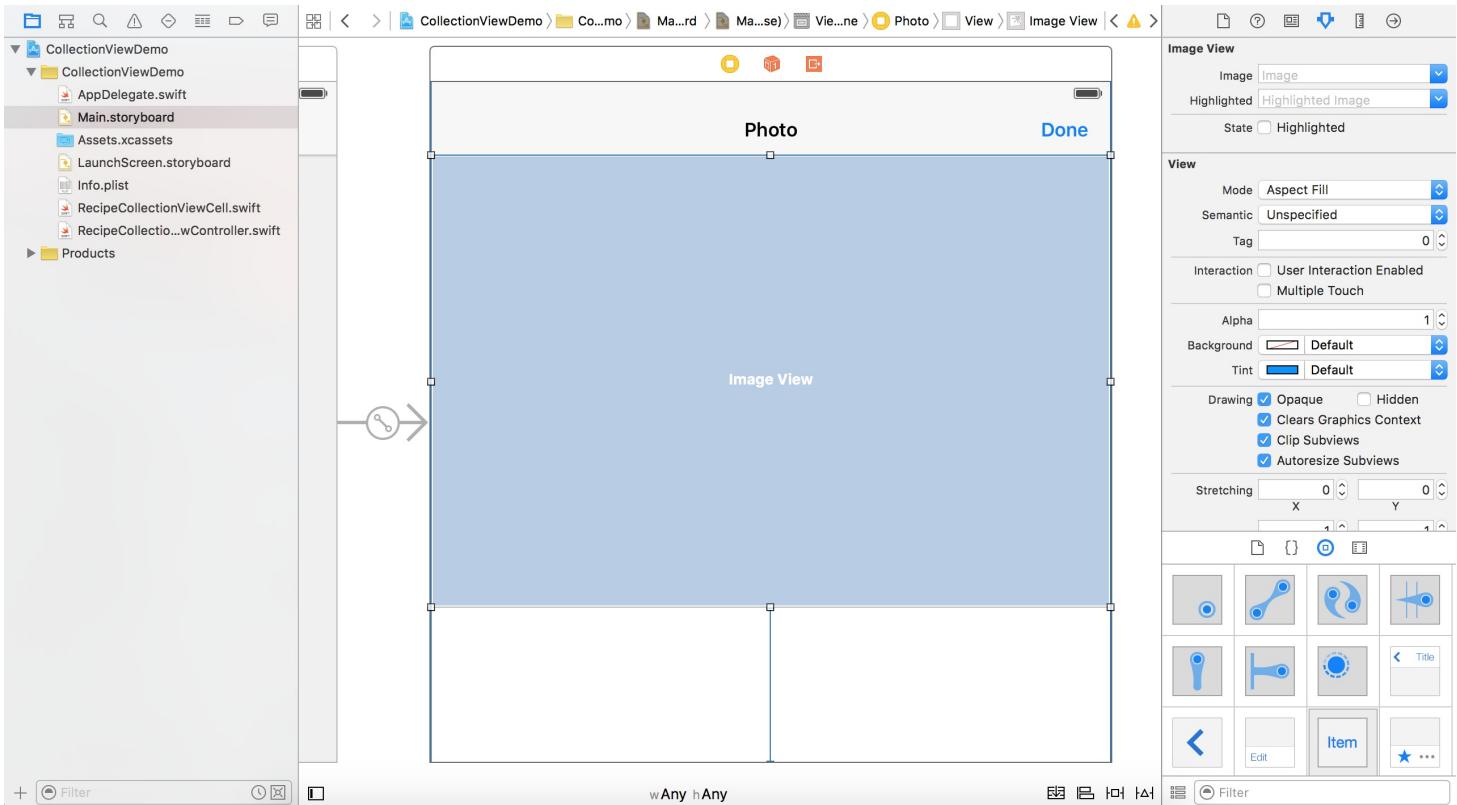
We'll continue to improve the collection view demo app. Here is what we're going to implement:

- When a user taps a recipe photo, the app will bring up a modal view and display the photo in a larger size.
- We'll also implement Facebook sharing in the app in order to demonstrate multiple item selections. Users are allowed to select multiple photos and share them on Facebook.

Let's first see how to implement the first feature to handle single selection. When the user taps any of the recipe photos, the app will bring up a modal view to display the photo in a higher resolution. If you didn't go through the previous, you can start by downloading the project template from <https://www.dropbox.com/s/o1anhgdxmoxz1p/CollectionViewDemo.zip?dl=0>.

First, let's design the view controller that is used to display the recipe photo. Go to `Main.storyboard`, drag a View Controller from the Object library to the storyboard. Then add an Image View to it and set the width and height to `600` and `400` respectively. Under the Attributes inspector, change the mode of the image view to `Aspect Fill` and enable `Clip Subviews`.

Lastly, embed it in a navigation controller and add a Bar Button Item to the navigation bar. Change the title of the navigation bar to `Photo`. Under the Attributes inspector, set the identifier of the button item to `Done`. Remember to define auto layout constraints for the image view. Simply select the image view object. Click the *Issues* button of the layout bar, followed by choosing the *Add Missing Constraints* option. Your view controller should look similar to the screenshot below.



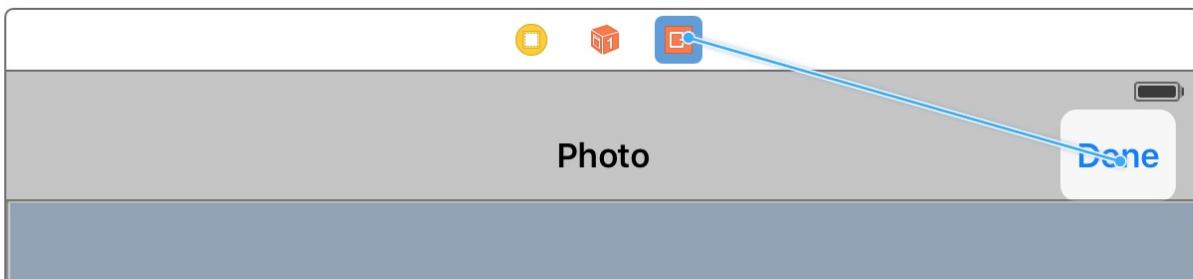
Since we want to display the view controller when a user taps any of the recipe photos in the collection view, we have to connect the collection view with the view controller using a segue. Control-drag from the cell of the collection view in the Document Outline to the navigation controller we just added. Select `Present Modally` for the style and set the segue identifier to `showRecipePhoto`.



When the user taps the *Done* button in the Photo View Controller, the controller will be dismissed. In order to do that, we will add an unwind segue. In `RecipeCollectionViewController.swift`, insert the following unwind segue method:

```
@IBAction func unwindToHome(segue: UIStoryboardSegue) {
```

Go back to the storyboard. Control-drag from the *Done* button to the Exit icon of the scene dock. Select `unwindToHome:` segue when prompted. This creates an unwind segue. When the current view controller is dismissed, the user will be brought back to the collection view controller.



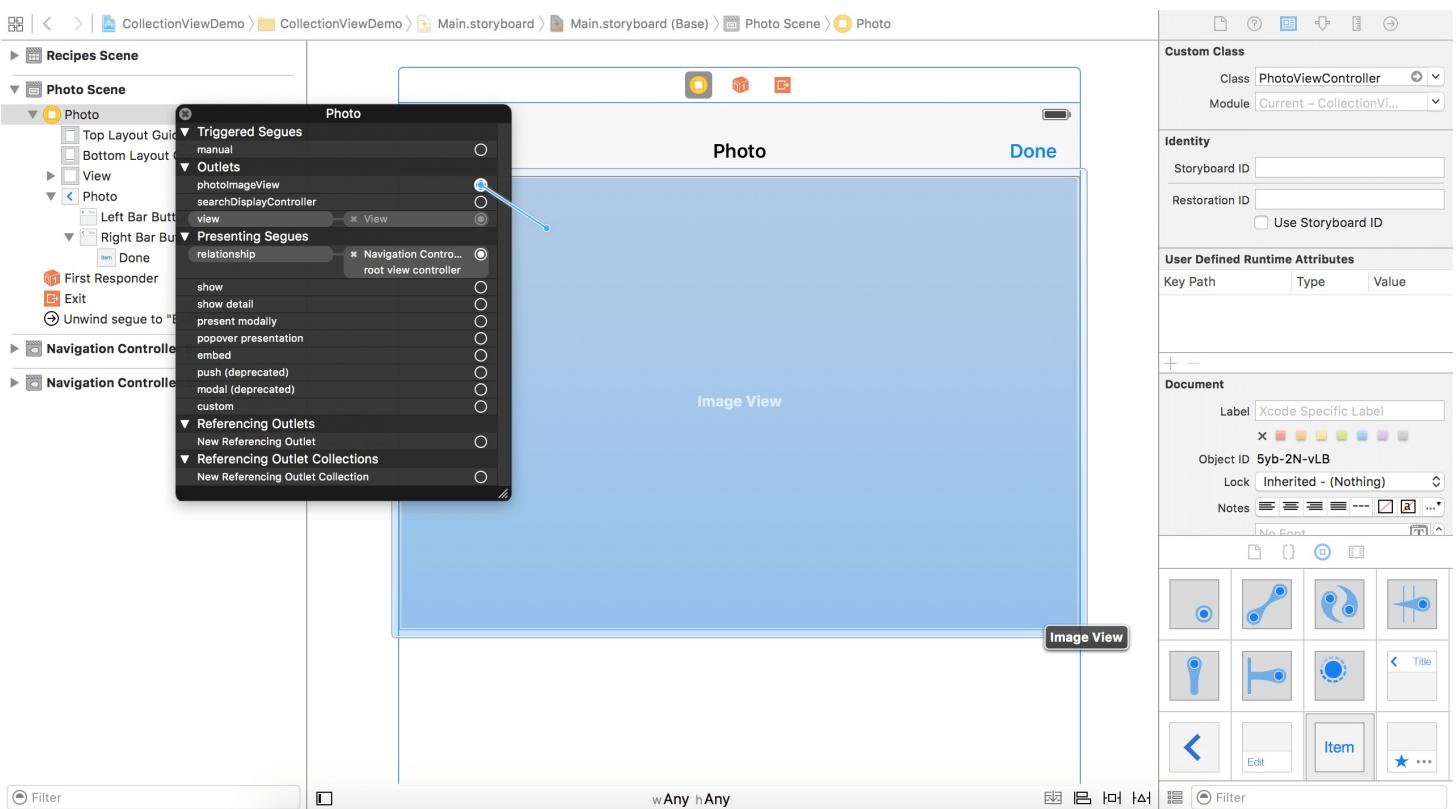
If you compile and run the app, you'll end up with an empty view when selecting any of the

recipe photos. Tapping the *Done* button will dismiss the view.

Since we haven't written any code, the modal view controller knows nothing about the selected recipe photo. Create a new class and name it `PhotoViewController`. Make it a subclass of `UIViewController`. Once created, add an outlet variable in the `PhotoViewController` class:

```
@IBOutlet var photoImageView: UIImageView!
```

In Interface Builder, select the view controller we just created and set the custom class to `PhotoViewController`. Then establish a connection between the image view and the `photoImageView` outlet variable.



Data Passing

In order to let other controllers pass the image name, we'll add an `imageName` property in the `PhotoViewController` class. Declare the property like this:

```
var imageName:String = ""
```

To load the specified recipe image in the image view, change the `viewDidLoad` method of

`PhotoViewController` to the following:

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

    photoImageView.image = UIImage(named: imageName)
}
```

Now the `PhotoViewController` class should be able to load the recipe image in the image view - but there's still one thing left. *How can we identify the selected item of the collection view and pass the image name to the PhotoViewController?*

If you understand how data passing works via a segue, you know we must implement the `prepareForSegue` method in the `RecipeCollectionViewController` class. Select `RecipeCollectionViewController.swift` and add the following code:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showRecipePhoto" {
        if let indexPaths = collectionView?.indexPathsForSelectedItems() {
            let destinationViewController = segue.destinationViewController as!
UINavigationController
            let photoViewController =
destinationViewController.viewControllers[0] as! PhotoViewController
            photoViewController.imageName = recipeImages[indexPaths[0].row]
            collectionView?.deselectItemAtIndexPath(indexPaths[0], animated:
false)
        }
    }
}
```

Just like `UITableView`, the `UICollectionView` class provides the `indexPathsForSelectedItems` method that returns the index paths of the selected items. You may wonder why multiple index paths are returned. The reason is that `UICollectionView` supports multiple selections. Each of the index paths corresponds to one of the selected items. For this demo, we only have single item selection. Therefore, we just pick the first index path, retrieve the selected image, and pass it to the photo view controller.

When a user taps a collection cell in the collection view, the cell changes to the highlighted state and then to the selected state. The last line of code is to deselect the selected item once the image is displayed in the modal view controller.

Now, let's build and run the app. After the app is launched, tap any of the recipes. You should see a modal view showing the selected recipe image.

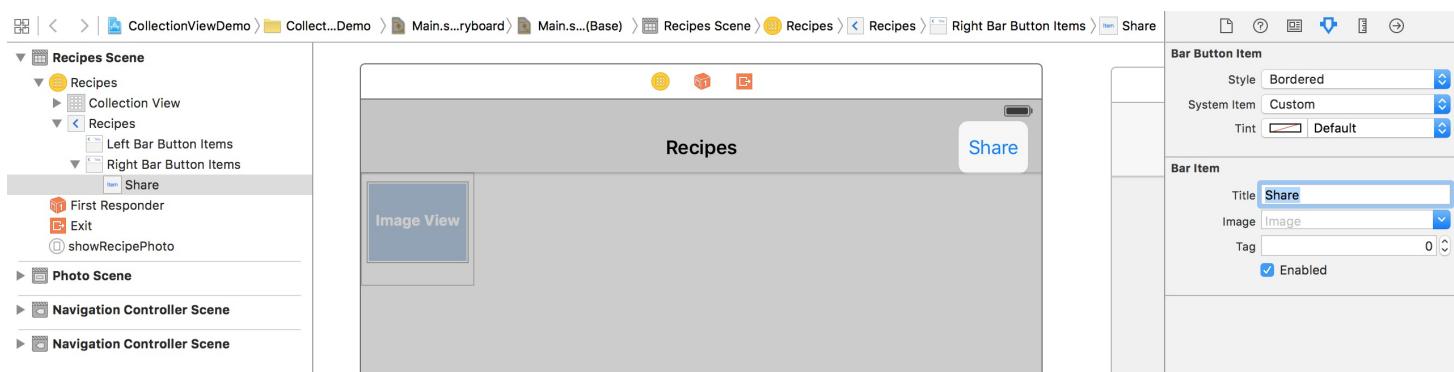
Handling Multiple Selections

`UICollectionView` supports both single and multiple selections. By default, the app only allows users to select a single item. The `allowsMultipleSelection` property of the `UICollectionView` class controls whether multiple items can be selected simultaneously. To enable multiple selections, the trick is to set the property to `true`.

To give you a better idea of how multiple selections work, we'll continue to tweak the demo app. Users are allowed to select multiple recipes and share them on Facebook in the following ways:

- A user taps the *Share* button in the navigation bar. Once the sharing starts, the button title is automatically changed to *Upload*.
- The user selects the recipe photos to share.
- After selection, the user taps the *Upload* button. The app will bring up a dialog for sharing the photos on Facebook.

We'll first add the *Share* button in the navigation bar of Recipe Collection View Controller. Go to `Main.storyboard`, drag a Bar Button Item from the Object library, and put it in the navigation bar of Recipe Collection View Controller.



In `RecipeCollectionViewController.swift`, insert an outlet variable for the *Share* button:

```
@IBOutlet var shareButton: UIBarButtonItem!
```

Also, add an action method:

```
@IBAction func shareButtonTapped(sender: AnyObject) {  
}
```

As usual, go to Interface Builder. Establish a connection between the *Share* button and the `shareButtonTapped` method. Also, associate it with the `shareButton` outlet.



The demo app now offers two modes: *single selection* and *multiple selections*. When a user taps the *Share* button, the app goes into multiple selection mode. This allows users to select multiple photos for sharing. To support multiple selection mode, we'll add two variables in the `RecipeCollectionViewController` class:

- `shareEnabled` – this is a boolean variable to indicate the selection mode. If it's set to `true`, it indicates the *Share* button was tapped and multiple selection is enabled.
- `selectedRecipes` – this is an array to store the selected recipes

Your code should look like this:

```
var shareEnabled = false  
var selectedRecipes:[String] = []
```

The `UICollectionViewDelegate` protocol defines methods that allow you to manage the selection and highlight items in a collection view. When a user selects an item, the `collectionView(_:didSelectItemAtIndexPath:)` method will be called. We'll implement this method and add the selected items into the `selectedRecipes` array. Insert the following method in the `RecipeCollectionViewController` class:

```
override func collectionView(collectionView: UICollectionView,  
didSelectItemAtIndexPath indexPath: NSIndexPath) {
```

```

// Check if the sharing mode is enabled, otherwise, just leave this method
guard shareEnabled else {
    return
}

// Determine the selected items by using the indexPath
let selectedRecipe = recipeImages[indexPath.row]

// Add the selected item into the array
selectedRecipes.append(selectedRecipe)
}

```

The `UICollectionViewCell` class provides a property named `selectedBackgroundView` for setting the background view of a selected item. To indicate a selected item, we'll change the background image of a collection cell. I've included the *photo-frame-selected.png* file in the project template. If you didn't use the template, you can download the image from <https://www.dropbox.com/s/owpro2wk24aq7n7/photo-frame-selected.png?dl=0>, and add it to the image asset. Edit the `collectionView(_:cellForItemAtIndexPath:)` method and insert the following line of code:

```

cell.selectedBackgroundView = UIImageView(image: UIImage(named: "photo-frame-
selected"))

```

Now when a user selects a recipe item, the selected cell will be highlighted.



Not only do we have to handle item selection, we also need to account for deselection. A user

may deselect an item from the collection view. When an item is deselected, it should be removed from the `selectedRecipes` array.

Insert the following code in the `RecipeCollectionViewController` class:

```
override func collectionView(collectionView: UICollectionView,
didDeselectItemAtIndexPath indexPath: NSIndexPath) {

    // Check if the sharing mode is enabled, otherwise, just leave this method
    guard shareEnabled else {
        return
    }

    let deSelectedRecipe = recipeImages[indexPath.row]
    if let index = recipeImages.indexOf(deSelectedRecipe) {
        recipeImages.removeAtIndex(index)
    }
}
```

Next, we'll move onto the implementation of the `shareButtonTouched` method. The method is called when a user taps the *Share* button. Update the method to the following code:

```
@IBAction func shareButtonTapped(sender: AnyObject) {
    if shareEnabled {

        // Post selected photos to Facebook
        if selectedRecipes.count > 0 {
            if
SLComposeViewController.isAvailableForServiceType(SLServiceTypeFacebook) {
                let facebookComposer = SLComposeViewController(forServiceType:
SLServiceTypeFacebook)
                    facebookComposer.setInitialText("Check out my recipes!")

                    for recipePhoto in selectedRecipes {
                        facebookComposer.addImage(UIImage(named: recipePhoto))
                    }

                    presentViewController(facebookComposer, animated: true,
completion: nil)

            }
        }

        // Deselect all selected items
        for indexPath in collectionView?.indexPathsForSelectedItems() as!
[NSIndexPath] {
    }
```

```

        collectionView?.deselectItemAtIndexPath(indexPath, animated: false)
    }

    // Remove all items from selectedRecipes array
    selectedRecipes.removeAll(keepCapacity: true)

    // Change the sharing mode to NO
    shareEnabled = false
    collectionView?.allowsMultipleSelection = false
    shareButton.title = "Share"
    shareButton.style = UIBarButtonItemStyle.Plain

} else {

    // Change shareEnabled to YES and change the button text to Upload
    shareEnabled = true
    collectionView?.allowsMultipleSelection = true
    shareButton.title = "Upload"
    shareButton.style = UIBarButtonItemStyle.Done

}
}

```

Let's take a look at the above code line by line.

We have two `guard` statements at the beginning of the method. We first check if the sharing mode is enabled. If not, we'll put the app into sharing mode and enable multiple selections. To enable multiple selections, all you need to do is set the `allowsMultipleSelection` property to `true`. Finally, we change the title of the button to *Upload*. The second `guard` statement check if the user has selected at least one recipe. If no recipe is selected, we just leave the method and do nothing.

When the app is in sharing mode (i.e. `shareEnabled` is set to `true`), after the user taps the *Upload* button, we'll bring up the Facebook composer. The iOS SDK allows you to integrate the social sharing feature in your app via the `SLComposeViewController` class. The `SLComposeViewController` class comes with some built-in methods to compose a post for various social networking services such as Facebook and Twitter. Here we use some of its built-in methods to upload multiple photos to Facebook. We first use the `isAvailableForServiceType` method to check if Facebook is configured on the current iOS device. If it's configured, we create a Facebook composer and attach the selected images using the `addImage` method. Finally, we call up the `presentViewController` method to display the composer on screen.

Quick note: If you do not understand how `SLComposeViewController` works, refer back to Chapter 5 for details.

After uploading the photos to Facebook, we deselect the selected items and remove them from the `selectedRecipes` array. Lastly, we switch back to single selection mode and change the button title.

Because `SLComposeViewController` is a class provided by the Social framework, remember to import the Social framework at the very beginning of `RecipeCollectionViewController.swift`:

```
import Social
```

The app is almost ready. However, if you run the app now, you will end up with a bug. After switching to sharing mode, the modal view still appears when you select any of the recipe photos - the result is not what we expected. The segue is invoked every time a collection view cell is tapped. Obviously, we don't want to trigger the segue when the app is in sharing mode. We only want to trigger the segue when it's in single selection mode. The `shouldPerformSegueWithIdentifier` method allows you to control the performance of a segue. Insert the following code snippet in `RecipeCollectionViewController.swift`:

```
override func shouldPerformSegueWithIdentifier(identifier: String, sender: AnyObject?) -> Bool {
    if identifier == "showRecipePhoto" {
        if shareEnabled {
            return false
        }
    }
    return true
}
```

Ready to Test Your App

Great! Now compile and run the app. Tap the *Share* button, select a few recipe photos and tap the *Upload* button to share them on Facebook. If the app can't bring up the Facebook composer, you probably forgot to sign in your Facebook account in the Simulator. Go back to the home screen, select Settings > Facebook. Sign in with your Facebook account. Once configured, re-run the app and try it out again.

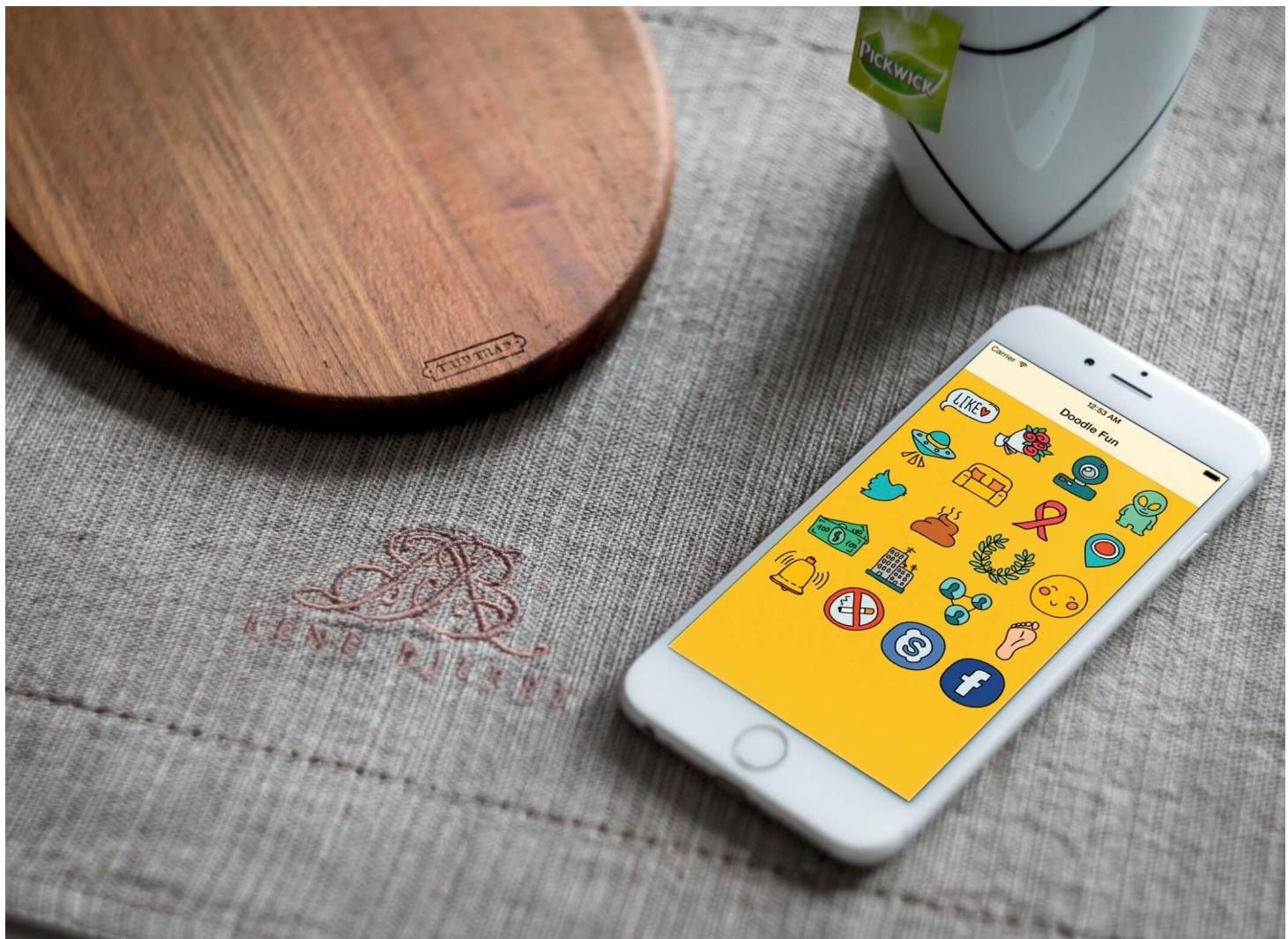


For reference, you can download the Xcode project from

<https://www.dropbox.com/s/nr478zimuol50q7/CollectionViewSelection.zip?dl=0>.

Chapter 20

Adaptive Collection Views Using Size Classes and UITraitCollection



In the previous two chapters, you learned to build a demo app using a collection view. The app works perfectly on iPhone 5/5s. But if you run the app on iPhone 6/6 Plus, your screen should look like the screenshot shown below. The recipes are displayed in grid format but with a large space between items. The screen of iPhone 6 and 6 Plus is wider than that of their predecessors. As the size of the collection view cell was fixed, the app rendered extra space between cells according to the screen width of the test device.



So how can we fix the issue? As mentioned in the first chapter of the book, iOS 8/9 comes with a new concept called *Adaptive User Interfaces*. You will need to make use of *Size Classes* and *UITraitCollection* to adapt the collection view to a particular device and device orientation. If you haven't read Chapter 1, I would recommend you to take a pause here and go back to the first chapter. Everything I will cover here is based on the material covered in the very beginning of the book.

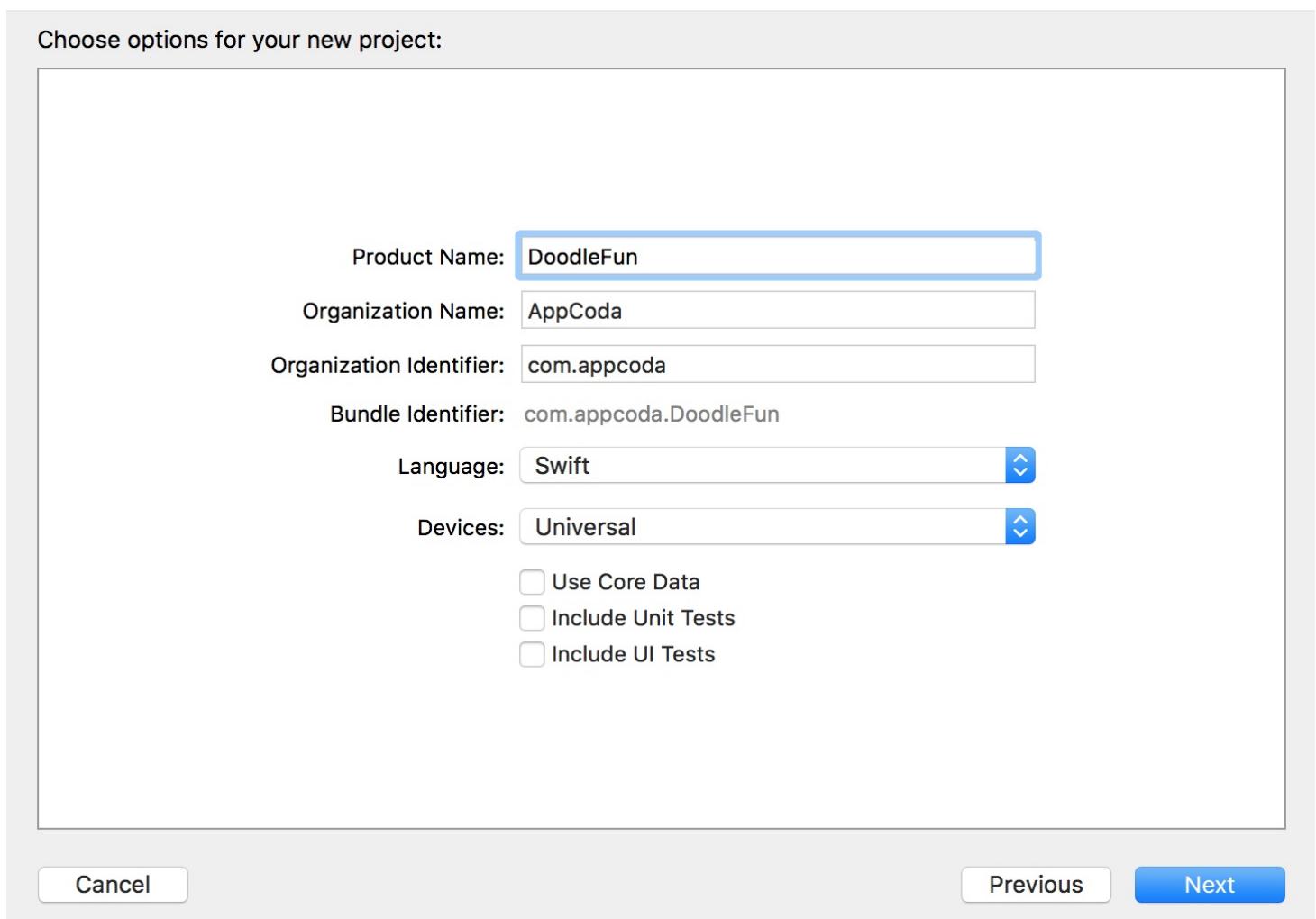
As usual, we will build a demo app to walk you through the concept. You are going to create an app similar to the one before but with the following changes:

- The cell is adaptive - The size of the collection view cell changes according to a particular device and orientation. You will learn how to use *size classes* and *UITraitCollection* to make the collection view adaptive.
- The app is universal - It is a universal app that supports both iPhone and iPad.
- We will use `UICollectionView` - Instead of using `UICollectionViewController`, you will learn how to use `UICollectionView` to build a similar UI.

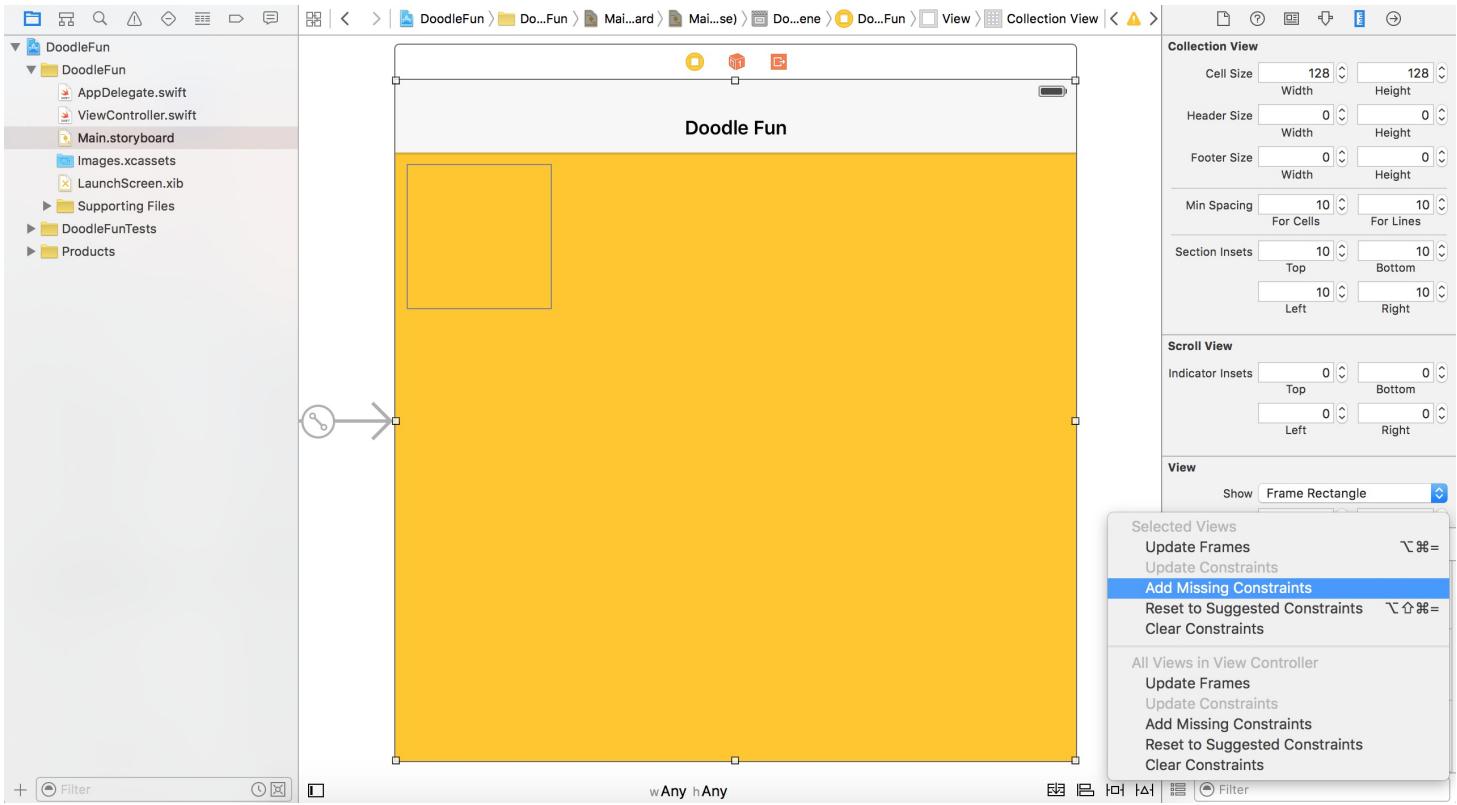
Creating the Demo Project

To get started, download the project template called *DoodleFun* from <https://www.dropbox.com/s/jbntfxvg4vthwm7/DoodleFunTemplate.zip?dl=0>. I have included a set of Doodle images (provided by the team at RoundIcons) and prebuilt the storyboard for you.

If you prefer to create the project from scratch, make sure to select Universal for the Devices option as we're going to build a universal app.

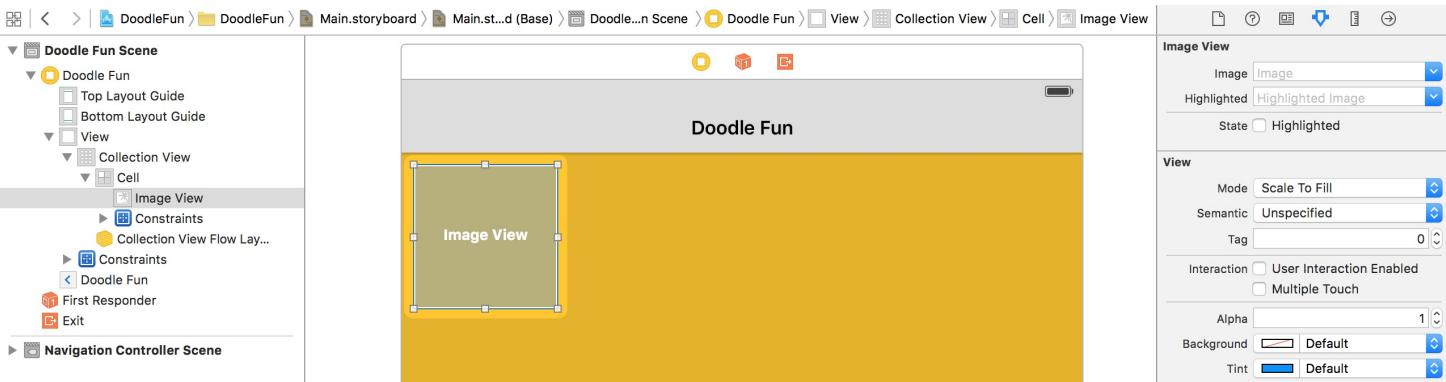


Assuming you've downloaded the template, open the project in Xcode and go to `Main.storyboard`. Drag a Collection View object from the Object library to the View Controller. Resize it (600x600) to make it fit the whole view.



In the Attributes inspector, change the background color to `yellow`. Next, go to the Size inspector. Set the cell size to `128` by `128`. Change the values of Section Insets (Top, Bottom, Left and Right) to `10` points. The insets define the margins applied to the content of the section. Because we are using a Collection View instead of Collection View Controller, we have to deal with auto layout constraints on our own. The simplest way to do this is to select the collection view and then click the *Issues* button of the auto layout menu, followed by selecting the *Add Missing Constraints* option. Xcode automatically defines the constraints for you.

Next, select the collection view cell and set its identifier to `cell` under the Attributes inspector. Drag an image view to the cell and resize it to make it fit the cell. Again, click the *Issues* button of the auto layout menu and select the *Add Missing Constraints* option to define the layout constraints.

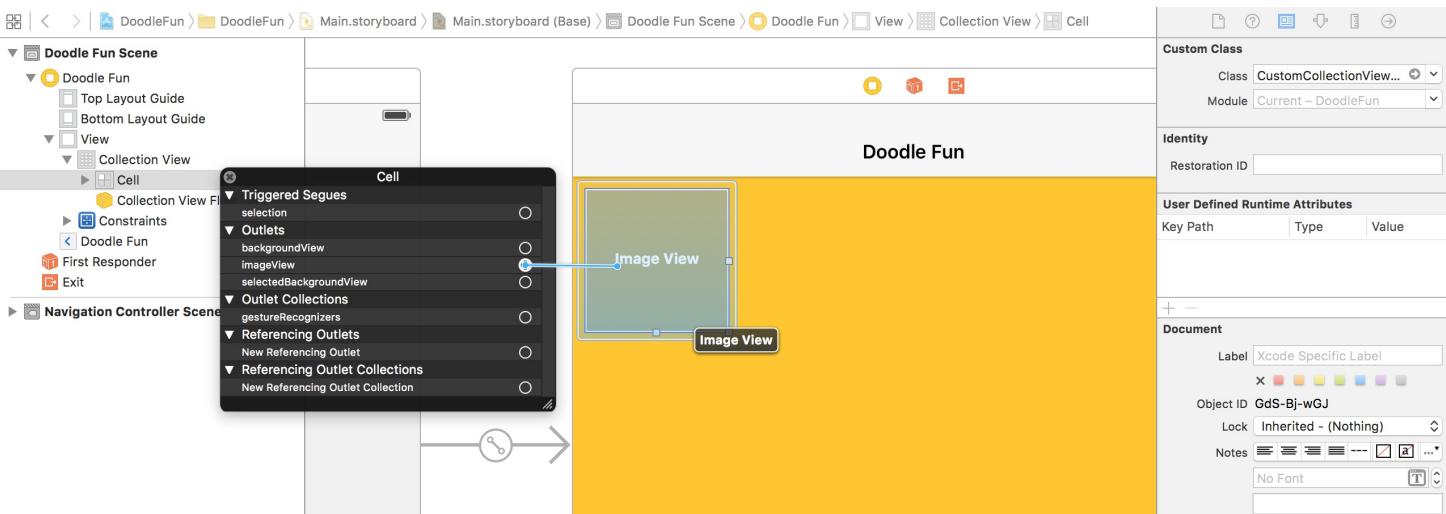


Diving into the Code

Now that you've created the collection view in the storyboard, let's move on to the coding part. First create a new file named `customCollectionViewCell` and set it as a subclass of `UICollectionViewCell`. Once the file was created, declare an outlet variable for the image view:

```
class CustomCollectionViewCell: UICollectionViewCell {
    @IBOutlet var imageView: UIImageView!
}
```

Switch to the storyboard. Select the collection view cell and change its custom class (under the Identity inspector) to `CustomCollectionViewCell`. Then right click the cell and connect the `imageView` outlet variable with the image view.



The `viewController` class is associated with the main view controller in the storyboard. As we want to present a set of images using the collection view, we have to implement both the `UICollectionViewDataSource` and `UICollectionViewDelegate` protocols.

```
class ViewController: UIViewController, UICollectionViewDataSource,  
UICollectionViewDelegate
```

Next, declare an array for the images and an outlet variable for the collection view:

```
var doodleImages = ["DoodleIcons-1", "DoodleIcons-2", "DoodleIcons-3",  
"DoodleIcons-4", "DoodleIcons-5", "DoodleIcons-6", "DoodleIcons-7",  
"DoodleIcons-8", "DoodleIcons-9", "DoodleIcons-10", "DoodleIcons-11",  
"DoodleIcons-12", "DoodleIcons-13", "DoodleIcons-14", "DoodleIcons-15",  
"DoodleIcons-16", "DoodleIcons-17", "DoodleIcons-18", "DoodleIcons-19",  
"DoodleIcons-20"]
```

```
@IBOutlet var collectionView: UICollectionView!
```

Like what we did before, we will have to implement two required methods of the `UICollectionViewDataSource` protocol:

- `func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int`
- `func collectionView(_ collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell`

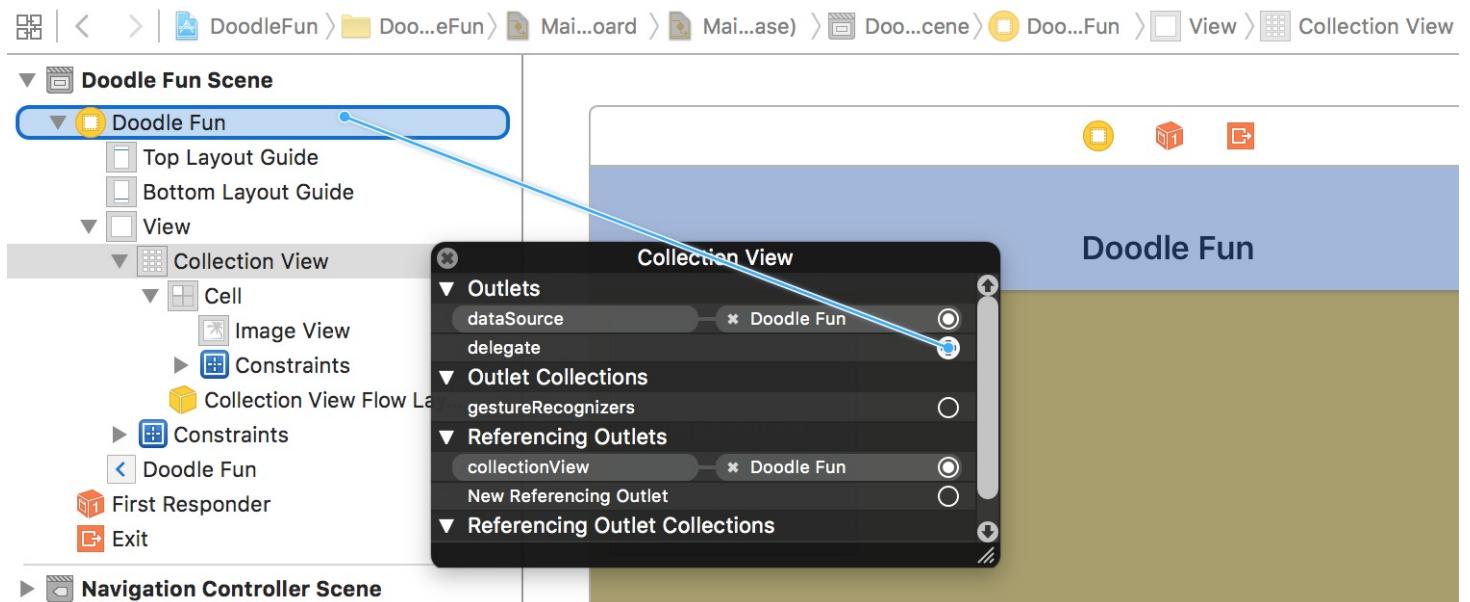
Okay, let's implement the methods like this:

```
func collectionView(collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {  
    return doodleImages.count  
}  
  
func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell {  
  
    let cell = collectionView.dequeueReusableCellWithIdentifier("Cell",  
forIndexPath: indexPath) as! CustomCollectionViewCell  
    cell.imageView.image = UIImage(named: doodleImages[indexPath.row])  
  
    return cell  
}
```

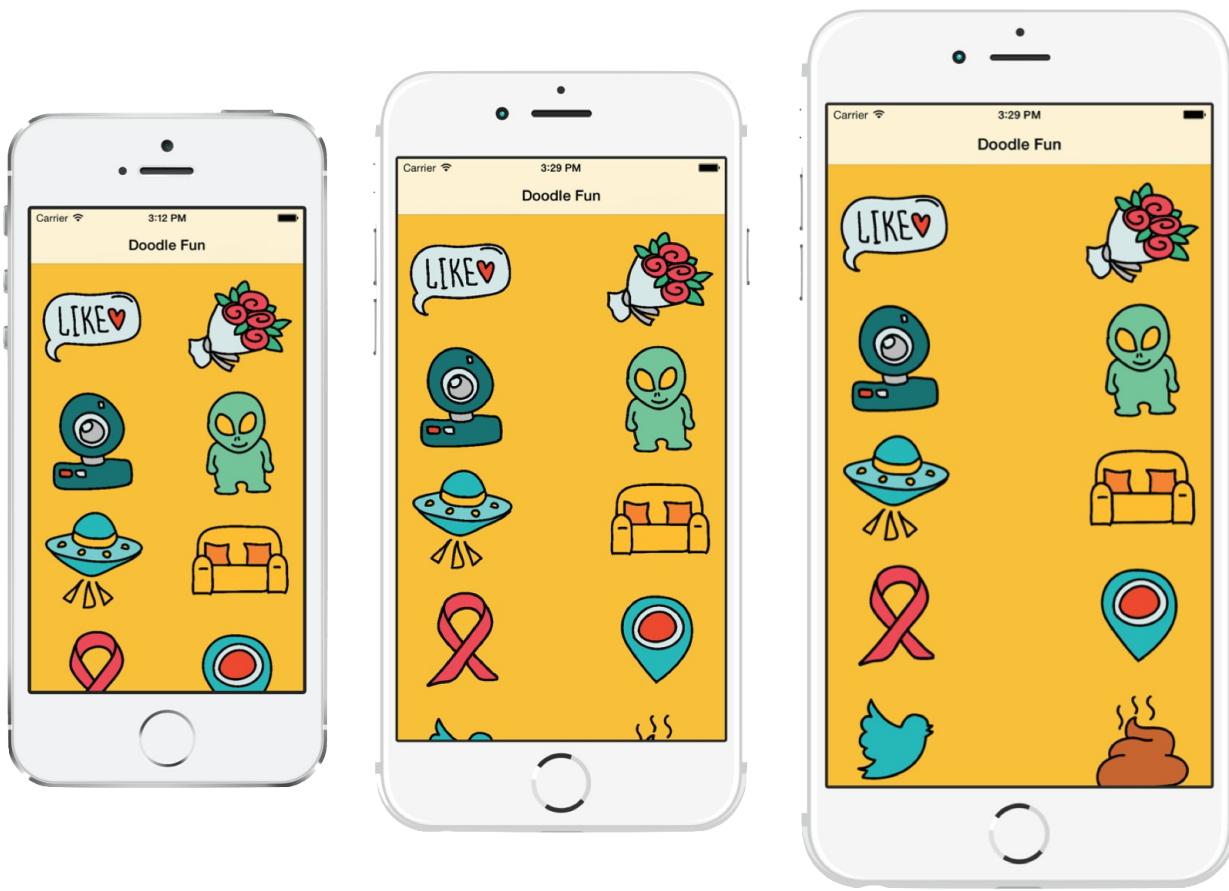
The above code is very straightforward. We return the total number of images in the first method and set the image of the image view in the latter method.

Now switch over to the storyboard. Establish a connection between the collection view and the

collectionView outlet variable. Also, connect the dataSource and delegate with the view controller.



That's it! We're ready to test the app. Compile and run the app on iPhone 5/5s simulator. The app looks pretty good, right? Now try to test the app on other iOS devices including the iPad and in landscape orientation. The app looks great on most devices but falls short on iPhone 6 and 6 Plus. `UICollectionView` can automatically determine the number of columns that best fits its contents according to the cell size. As you can see below, the number of columns varies depending on the screen size of a particular device. In portrait mode, the screen width of an iPhone 6 and iPhone 6 Plus is 370 points and 414 points respectively. If you do a simple calculation for the iPhone 6 Plus (e.g. $[414 - 20 \text{ (margin)} - 20 \text{ (cell spacing)}] / 128 = 2.9$), you should understand why it can only display cells in two columns, leaving a large gap between columns.



Designing for size classes

So how can you fix this issue on the iPhone 6 and 6 Plus? Obviously you can reduce the cell size so that it fits well on all Apple devices. A better way to resolve the issue, however, is to make the cell size adaptive.

The collection view works pretty well in landscape orientation regardless of device types. To fix the display issue, we are going to keep the size of the cell the same (i.e. 128x128 points) for devices in landscape mode but minimize the cell for iPhones in portrait mode.

The real question is how do you find out the current device and its orientation? In the past, you would determine the device type and orientation using code like this:

```
let device = UIDevice.currentDevice()
let orientation = device.orientation
let isPhone = (device.userInterfaceIdiom == UIUserInterfaceIdiom.Phone) ? true
: false
```

```

if isPhone {
    if orientation.isPortrait {
        // Change cell size
    }
}

```

Starting from iOS 8, the above code is not ideal. You're discouraged from using `UIUserInterfaceIdiom` to verify the device type. Instead, you should use size classes to handle issues related to idiom and orientation. I covered size classes in Chapter 1, so I won't go into the details here. In short, it boils down to this two by two grid:

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

There is no concept of orientation. For iPhones in portrait mode, it is indicated by a compact horizontal class and regular vertical class. So how can you access the current size class from code?

Understanding Trait Collections

Well, you use a new system called *Traits*. The horizontal and vertical size classes are considered traits. Together with other properties like `userInterfaceIdiom` and display scale they make up a so-called *trait collection*.

In iOS 8, Apple introduced trait environments (i.e. `UITraitEnvironment`). This is a new protocol that is able to return the current trait collection. Because `UIViewController` conforms to the `UITraitEnvironment` protocol, you can access the current trait collection through the

`traitCollection` property. If you put the following line of code in the `viewDidLoad` method to print its content to console:

```
print("\(traitCollection)")
```

You should have something like this when running the app on an iPhone 6 Plus:

```
<UITraitCollection: 0x7fae9a435a60; _UITraitNameUserInterfaceIdiom = Phone,  
_UITraitNameDisplayScale = 3.000000, _UITraitNameHorizontalSizeClass = Compact,  
_UITraitNameVerticalSizeClass = Regular, _UITraitNameTouchLevel = 0,  
_UITraitNameInteractionModel = 1, _UITraitNameForceTouchCapability = 1>
```

From the above information, you discover the device is an iPhone that is in the *Compact* horizontal and *Regular* vertical size classes. The display scale of *3x* indicates a Retina HD 5.5 display.

Adaptive Collection View

With a basic understanding of trait collection, you should know how to determine the current size class of a device. Now it's time to make the collection view adaptive. The `UICollectionViewDelegateFlowLayout` protocol provides an optional method for specifying the size of a cell:

```
collectionView(_:layout:sizeForItemAtIndexPath:)
```

All you need to do is override the method and return the cell size at runtime. Recall that we only want to alter the cell size for iPhones in portrait mode, so we will implement the method like this:

```
func collectionView(collectionView: UICollectionView, layout  
collectionViewLayout: UICollectionViewLayout, sizeForItemAtIndexPath indexPath:  
NSIndexPath) -> CGSize {  
  
    let sideSize = (traitCollection.horizontalSizeClass == .Compact &&  
    traitCollection.verticalSizeClass == .Regular) ? 80.0 : 128.0  
    return CGSize(width: sideSize, height: sideSize)  
}
```

For devices with a *Compact* horizontal and a *Regular* vertical size class (i.e. iPhone Portrait),

we set the size of the cell to 80x80 points. Otherwise, we just keep the cell size the same. Run the app again on an iPhone 6 / 6 Plus. It should look much better now.



Respond to the Change of Size Class

Did you try to test the app in landscape mode? When you turned the iPhone sideways, the cell size was unchanged. There is one thing missing in the current implementation; we have not implemented the method that responds to size and trait changes. Insert the following method in the View Controller class:

```
override func viewWillTransitionToSize(size: CGSize,  
    withTransitionCoordinator coordinator:  
UIViewControllerTransitionCoordinator) {  
    collectionView.reloadData()  
}
```

When the size of the view is about to change (e.g. rotation), UIKit will call the method. Here we simply update the collection view by reloading its data. Now test the app again. When your iPhone is put in landscape mode, the cell size should be changed accordingly.



For reference, you can download the complete project from
<https://www.dropbox.com/s/wgt056afoporaq4/DoodleFunFinal.zip?dl=0>.

Your Exercise

In some scenarios, you may want all images to be visible in the collection view without scrolling. In this case, you'll need to perform some calculations to adjust the cell size based on the area of the collection view. To calculate the total area of the collection view, you can use the code like this:

```
let collectionViewSize = collectionView.frame.size
let collectionViewArea = Double(collectionViewSize.width *  
collectionViewSize.height)
```

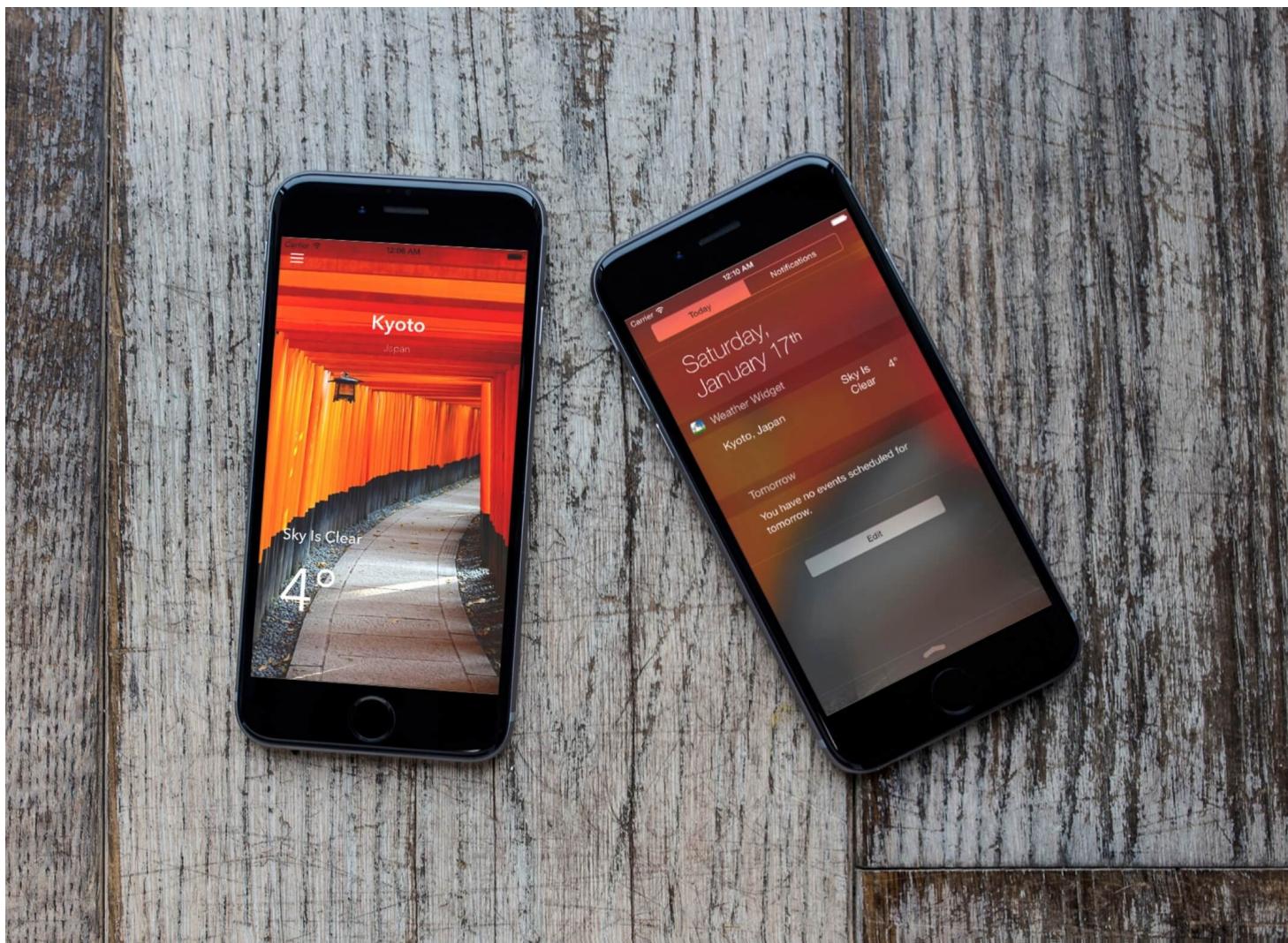


With the total area and total number of images, you can calculate the new size of a cell. For the rest of the implementation, I will leave it as an exercise for you. Take some time and try to

figure it out on your own before checking out the solution at
<https://www.dropbox.com/s/562118zu12z1oyd/DoodleFunExercise.zip?dl=0>.

Chapter 21

Building a Today Widget



In iOS 8, Apple introduced app extensions, which let you extend functionality beyond your app and make it available to users from other parts of the system (such as from other apps or the Notification Center). For example, you can provide a widget for users to put in Notification Center. This widget can display the latest information from your app (i.e. weather, sports scores, stock quotes, etc.).

iOS defines different types of extensions, each of which is tied to an area of the system such as the keyboard, Notification Center, etc. A system area that supports extensions is called an

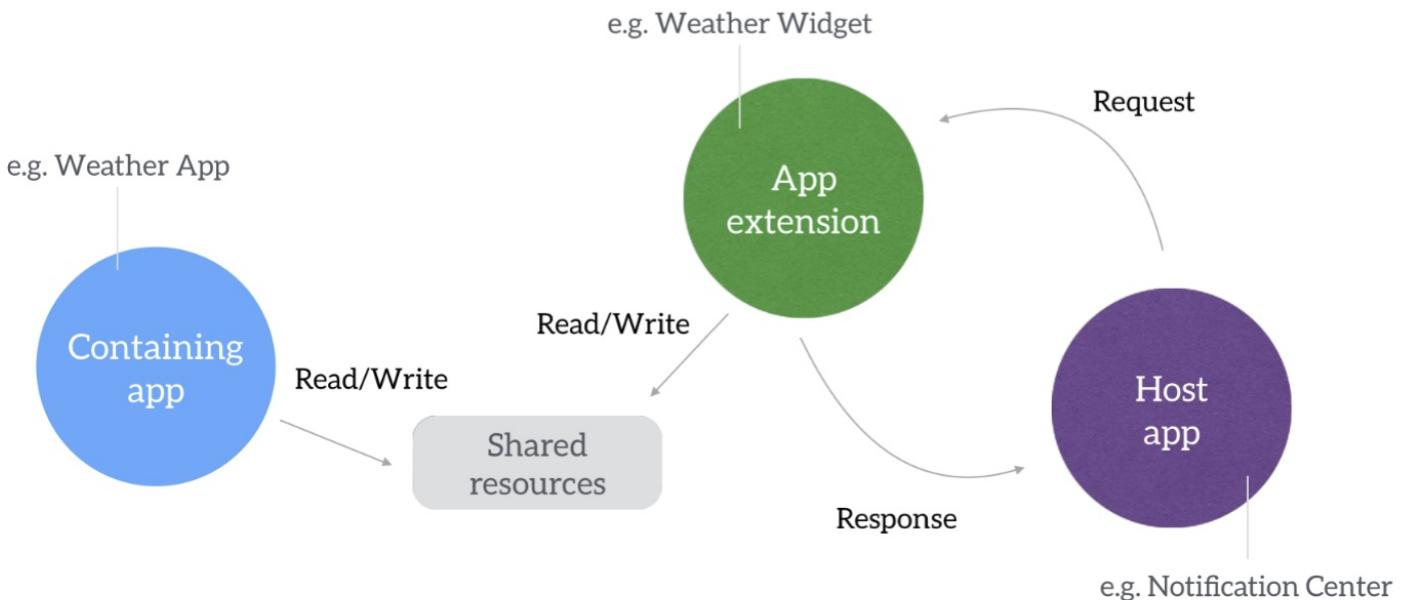
extension point. Below is a list of the extension points in iOS:

- Today – Shows brief information and can allow performing of quick tasks in the Today view of Notification Center (also known as widgets)
- Share – Shares content with others or post to a sharing website
- Action – Manipulates content in a host app
- Photo Editing – Allows user to edit photos or videos within the Photos app
- Document Provider – Provides access to and manage a repository of files
- Custom Keyboard – Provides a custom keyboard to replace the iOS system keyboard

In this chapter, I will show you how to add a weather widget in the notification center. For other extensions, we'll explore them in later chapters.

Understanding How App Extensions Work

Before getting started with the Today Extensions, let's first take a look at how extensions work. To start off, app extensions are not standalone apps. An extension is delivered via the App Store as a part of an app bundle. The app that the extension is bundled with is known as the *container app*, while the app that invokes the extension is the *host app*. For example, if you're building a weather app that bundles a weather extension, the weather extension will appear in the Today View of the Notification Center. Here the weather app is the container app and the Notification Center is the extension's host app. Usually, you bundle a single app extension in a container but you're allowed to have more than one extension.



When an extension is running, it doesn't run in the same process as the container app. Every instance of your extension runs as its own process. It is also possible to have one extension run in multiple processes at the same time. For example, let's say you have a sharing extension which is invoked in Safari. An instance of the extension, a new process, is going to be created to serve Safari. Now, if the user goes over to Mail and launches your share extension, a new process of the extension is created again. These two processes don't share address space.

An extension cannot communicate directly with its container app, nor can it enable communication between the host app and container app. However, indirect communication with its container app is possible via either `openURL()` or a shared data container like the use of `NSUserDefaults` to store data which both extension and container apps can read and write to.

The Today Extension

Today extensions, also known as widgets, appear on the Today View of the Notification Center. They provide brief pieces of information to the user and they even allow some interaction, though limited, right from the Notification Center. You've seen these available in previous versions of iOS, for e.g. Reminders, Stocks, and Weather. Starting from iOS 8, third party apps can now have their own widgets.

We are going to explore how to create a widget by creating a simple weather app. To keep you

focused on building an extension instead of creating an app from scratch, I have provided a starter project that you can download at <https://www.dropbox.com/s/s1eu2fsoobboypo/WeatherDemo.zip?dl=0>. The project is a simple weather app, showing the various weather information of a particular location. You will need an internet connection for the data to be fetched. The app is very simple and doesn't include any geoLocation functionality. The default location is assumed to be Paris, France. The app, however, provides a setting screen for altering the default location. It relies on a free API provided by openweathermap.org to aggregate weather information. The API returns weather data of a particular location in JSON format. If you have no idea about JSON parsing in iOS, refer to Chapter 4 for details.

When you open the app, you should see a visual that shows the weather information for the default location. You can simply tap the menu button to change the location.



We are going to create a Today extension of the app that will show a brief summary of the weather in the Today View. You'll also learn how to share data between the container app and

extension. We'll use this shared data to let a user choose the location they want weather information about.

Code Sharing with Embedded Framework

Extensions are created in their own targets separate from the container app. This means that you can't access common code files as you normally would in your project. While you can duplicate the common code files in the extension, this is not a good habit to get into. To avoid code repetition, make the common code files available to both the container app and the extension.

For example, both the weather app and the weather extension are required to use the `WeatherService` class to retrieve the latest weather information. You can replicate the files in both targets. But this is not a good practice. When developing an app or an extension, you should always consider code reuse.

To allow for code reuse, you create an embedded framework, which can be used across both targets. You can place the common code that will need to be used by both the container app and extension in the framework.

In the demo app, both the extension and container app make a call to a weather API and retrieve the weather data. Without using a framework we would have to duplicate the code, which would be inefficient and difficult to maintain.

Creating an Embedded Framework

To create a framework, select your project in the Project Navigator and add a new target by selecting *Editor > Add Target*. From the window that appears, select *iOS > Framework & Library > Cocoa Touch Framework*.

Choose a template for your new target:

<p>iOS</p> <ul style="list-style-type: none">ApplicationFramework & LibraryApplication ExtensionApple WatchTest <p>watchOS</p> <ul style="list-style-type: none">ApplicationFramework & Library <p>OS X</p> <ul style="list-style-type: none">ApplicationFramework & LibraryApplication ExtensionTestSystem Plug-inOther	 Cocoa Touch Framework	 Cocoa Touch Static Library
	<p>Cocoa Touch Framework</p> <p>This template creates a framework that uses UIKit.</p>	

[Cancel](#)

[Previous](#)

[Next](#)

Set its name to `WeatherInfoKit` and check that the language is `Swift`. Leave the rest of the options as they are and click `Finish`.

Choose options for your new target:

Product Name: `WeatherInfoKit`

Organization Name: `AppCoda`

Organization Identifier: `com.appcoda`

Bundle Identifier: `com.appcoda.WeatherInfoKit`

Language: `Swift`

Include Unit Tests

Project: `WeatherDemo`

Embed in Application: `WeatherDemo`

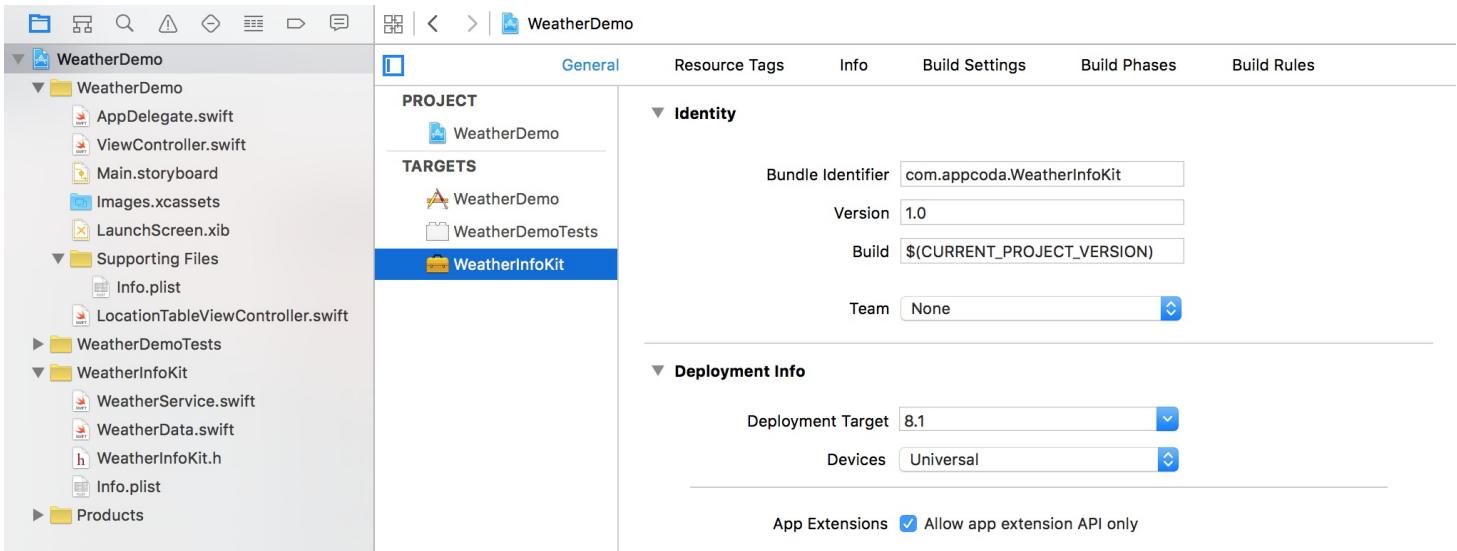
[Cancel](#)

[Previous](#)

[Finish](#)

You will see a new target appear in the list of targets as well as a new group folder in the Project Navigator. When you expand the `WeatherInfoKit` group, you will see `WeatherInfoKit.h`. If you are using Objective-C, or if you have any Objective-C files in your framework, you will have to include all public headers of your frameworks here. Because we're now using Swift, we do not need to edit this file.

Next, on the General tab of the `WeatherInfoKit` target, under the Deployment Info section, check `Allow app extension API only`. Also change the deployment target to `8.1` because this Xcode project is set to support iOS 8.1 (or up).



You should note that app extensions are somewhat limited in what they can do and therefore not all Cocoa Touch APIs are available for use in extensions. For instance extensions cannot do the following:

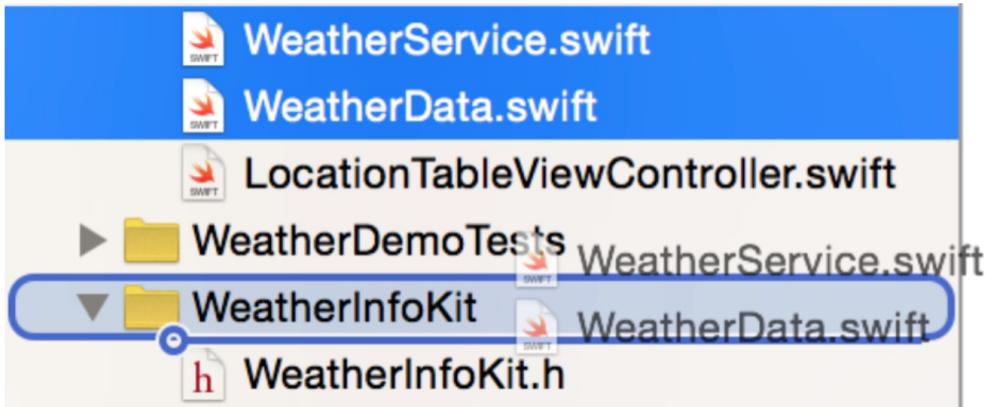
- Access the camera or microphone on an iOS device
- Receive data using AirDrop (however they can send data using AirDrop)
- Perform long-running background tasks
- Use any API marked in header files with the `NS_EXTENSION_UNAVAILABLE` macro, similar unavailability macro, or any API in an unavailable framework (for example EventKit or HealthKit) are unavailable to app extensions.
- Access a sharedApplication object or use any of the methods on that object. For example, both the HealthKit framework and EventKit UI framework are unavailable to app extensions.

Because the framework we're creating will be used by an app extension, it's important to check the *Allow app extension API only* option.

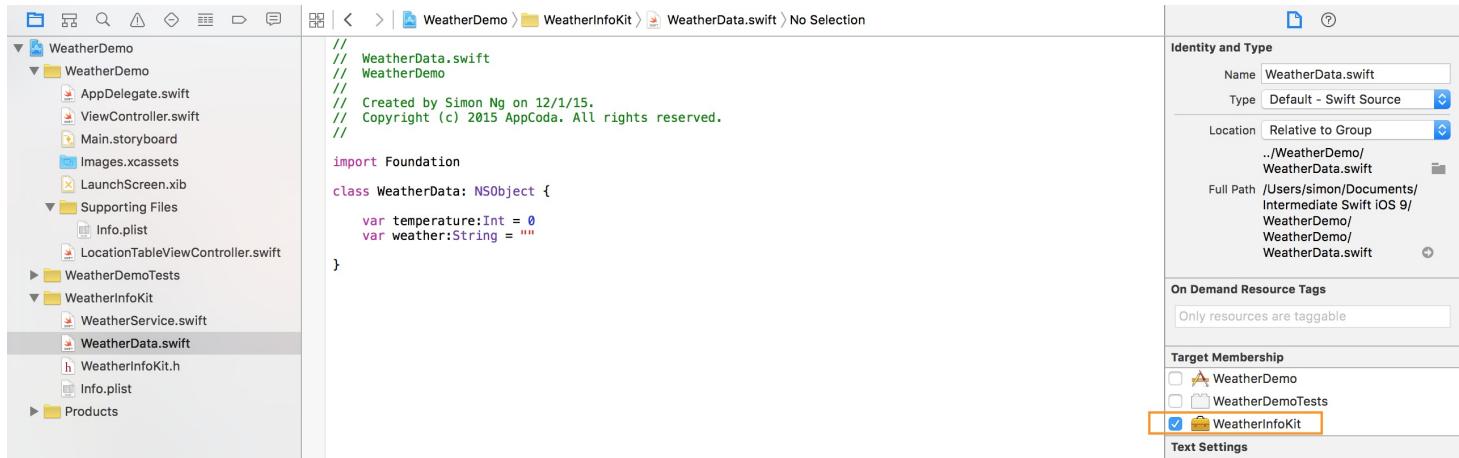
Moving Common Files to the Framework

In the starter project, both `WeatherService.swift` and `WeatherData.swift` are common code files. The `WeatherData` class represents the weather information including temperature (in Celsius) and weather description (e.g. Sky is clear). The `WeatherService` class is a common service class that is responsible for calling up the weather API and parsing the returned JSON data.

To put these two files (or classes) into the `WeatherInfoKit` framework, all you need to do is drag these two files into the `WeatherInfoKit` group under the Project Navigator.



Merely dragging a file from one target to another doesn't make the file part of that target - you have to change the file's target membership yourself. To do this, select the `WeatherService.swift` file from the Project Navigator. Then open the File Inspector and change the files target in the Target Membership section by unchecking the `Weather` target and checking the `WeatherInfoKit` target. Repeat the process for the `WeatherData.swift` file.



Because the `WeatherService` and `WeatherData` classes were removed from the `WeatherDemo` target, you'll end up with an error in `ViewController.swift`.

If you're new to Swift, it's important to know that it provides three access levels for entities in your code: *public*, *internal* and *private*. By default, all entities (e.g. classes, variables) are defined with the *internal* access level. That means the entities can only be used within any source file from the same module/target. Now that the `WeatherService` and `WeatherData` classes were moved to another target (i.e. `WeatherInfoKit`), the `ViewController` of the

`WeatherDemo` target can no longer access both classes as the access level of the classes is set to *internal*.

To resolve the error, we have to change the access level of these classes to *public*.

Public access allows entities to be used in source files from another module. When you're developing a framework, typically, your classes should be accessible by source files of any modules. In this case, you use public access to specify the public interface of a framework.

Therefore, open `WeatherData.swift` and add the `public` access modifier to the class declaration:

```
public class WeatherData: NSObject {  
  
    public var temperature:Int = 0  
    public var weather:String = ""  
  
}
```

Apply the same change to the class, method and typealias declarations of

`WeatherService.swift`:

```
public class WeatherService {  
    public typealias WeatherDataCompletionBlock = (data: WeatherData?) -> ()  
  
    let openWeatherBaseAPI = "http://api.openweathermap.org/data/2.5/weather?  
appid=bd82977b86bf27fb59a04b61b657fb6f&units=metric&q="  
    let urlSession:NSURLSession = NSURLSession.sharedSession()  
  
    public class func sharedWeatherService() -> WeatherService {  
        return _sharedWeatherService  
    }  
  
    public func getCurrentWeather(location:String, completion:  
WeatherDataCompletionBlock) {  
        ...  
    }  
}
```

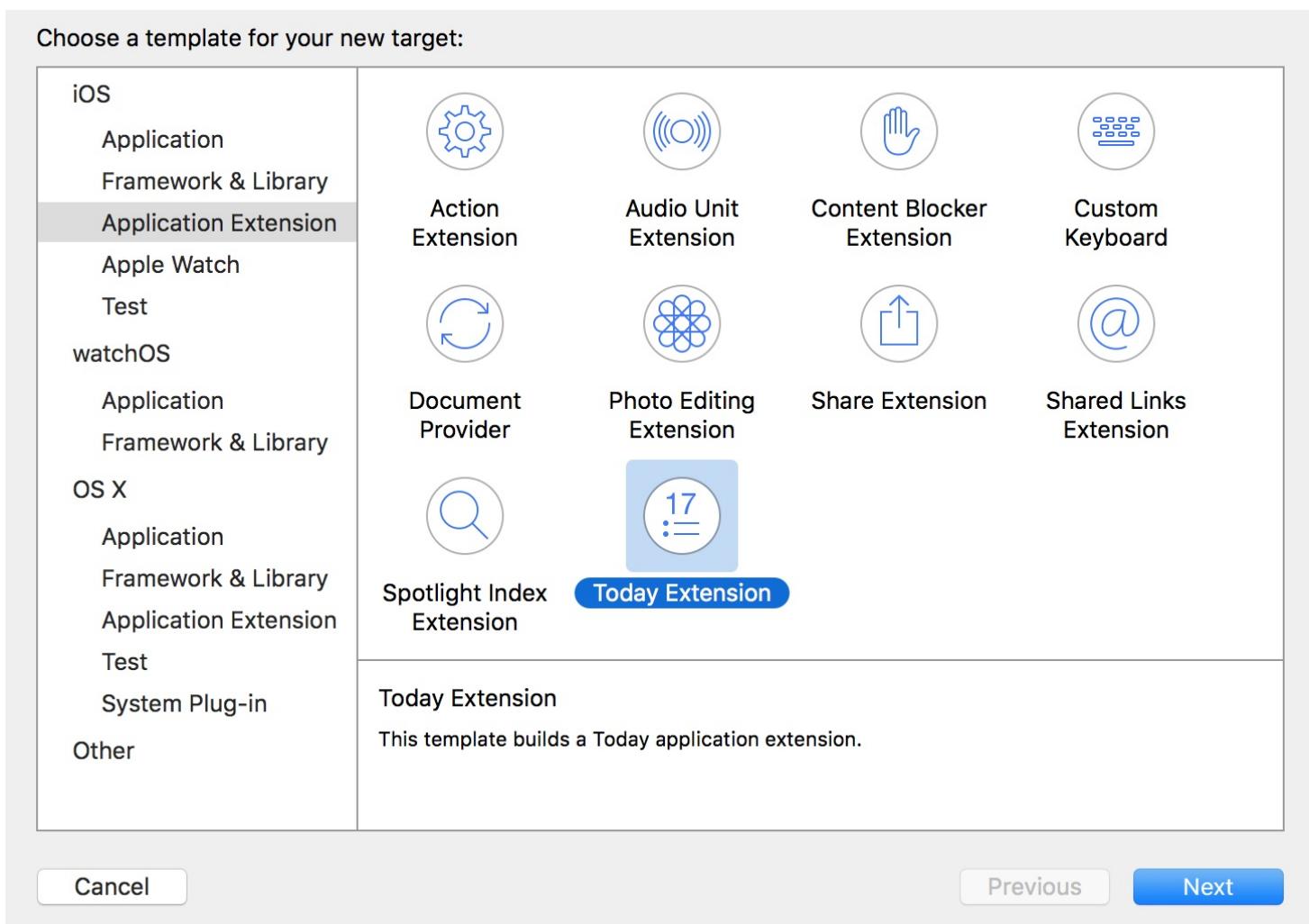
After doing this, the errors in `ViewController.swift` still appear. Include the following import statement at the top of the file to import the framework we just created:

```
import WeatherInfoKit
```

Now compile the project again. You should be able to run the WeatherDemo without errors. The app is still the same but the common files are now put into a framework.

Creating the Today Widget

You're now ready to create the widget. To create a widget, we'll use the Today extension point template provided by Xcode. Select the project in the Project Navigator and add a new target by selecting *Editor > Add Target*. Select *iOS > Application Extension > Today Extension* and then click `Next`.



Set the Product Name to `Weather Widget` and leave the rest of the settings as they are. Click `Finish`.

Choose options for your new target:

Product Name:	Weather Widget
Organization Name:	AppCoda
Organization Identifier:	com.appcoda.WeatherDemo
Bundle Identifier:	com.appcoda.WeatherDemo.Weather-Widget
Language:	Swift
Project:	WeatherDemo
Embed in Application:	WeatherDemo

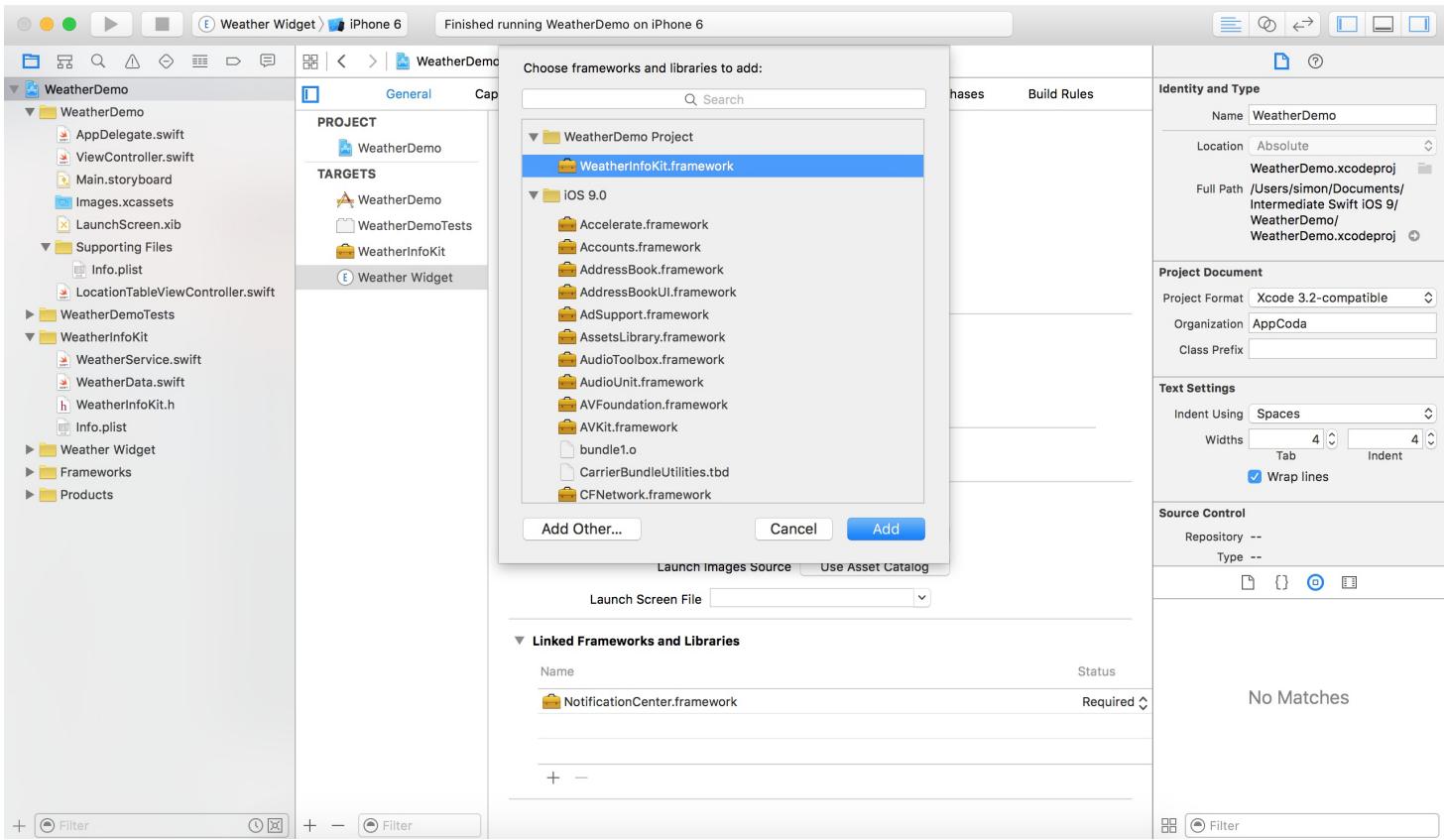
Cancel

Previous

Finish

At this point you should see a prompt asking if you want to activate the `Weather Widget` scheme. Press `Activate`. Another Xcode scheme has been created for you and you can switch schemes by navigating to *Product > Scheme* and then selecting the scheme you want to switch to. You can also switch schemes from the Xcode toolbar.

Next, select `WeatherDemo` from the Project Navigator. From the list of available targets, select `Weather Widget` and change the deployment target to `8.1`. Then on the General tab press the `+` button under Linked Frameworks and Libraries. Select `weatherInfoKit.framework` and press `Add`.



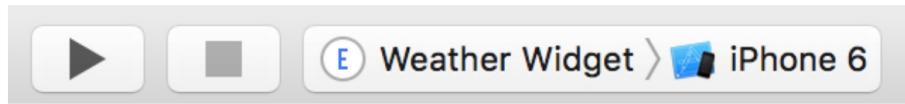
With the framework linked, we can now implement the extension.

In the Project Navigator you will see that a new group with the widget's name was created. This contains the extension's storyboard, view controller, and property list file. The plist file contains information about the widget and most often you won't need to edit this file, but an important key that you should be aware of is the `NSExtension` dictionary. This contains the `NSExtensionMainStoryboard` key with a value of the widget's storyboard name, in our case `MainInterface`. If you don't want to use the storyboard file provided by the template, you will have to change this value to the name of your storyboard file. For this demo, we just keep it intact.

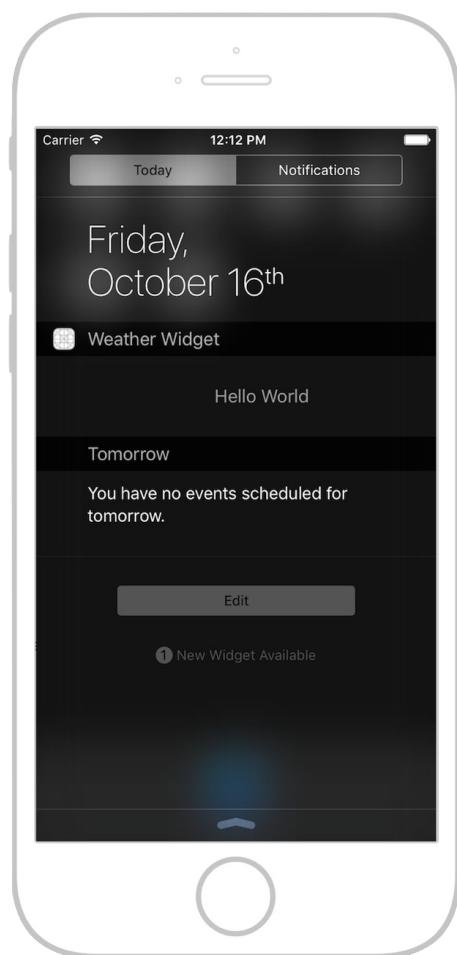
Open `MainInterface.storyboard`. You'll see a simple view with a Hello World label.



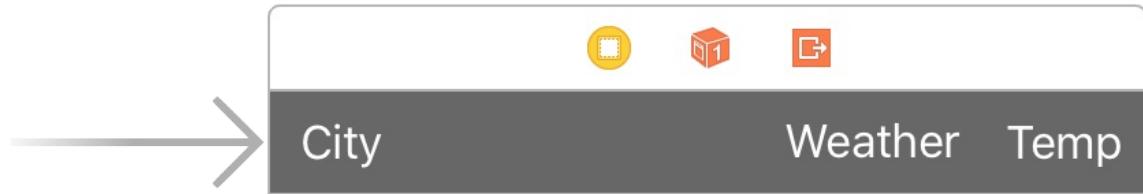
Let's have a quick test before redesigning the widget. To run the extension, make sure the `Weather Widget` scheme is selected in Xcode's toolbar and hit Run.



A window will pop up, allowing you to choose an app to run. This lets Xcode know which host app to run. Choose `Today`. With this selection, iOS will know to open Notification Center in the Today view, which in turn launches your widget. Notification Center is the Today Extension's host app. Click Run and you should see the widget on your simulator's/device's Notification Center.



To display the weather data in Today view, we first redesign the Today View Controller in storyboard like this:



All you need to do is delete the Hello World label and add the City, Weather, and Temperature labels. Remember to set the number of lines for the labels to zero and define auto layout constraints so that it fits for multiple screen resolutions. If you're not familiar with auto layout, the easiest way is to let Xcode add the rules for you. Once you place the labels in the view controller, select City and click the Issues button in the layout bar. Select the *Add Missing Constraints* option. Xcode then automatically adds the rules for you.

Now go to the `TodayViewController.swift` file and add the following import statement:

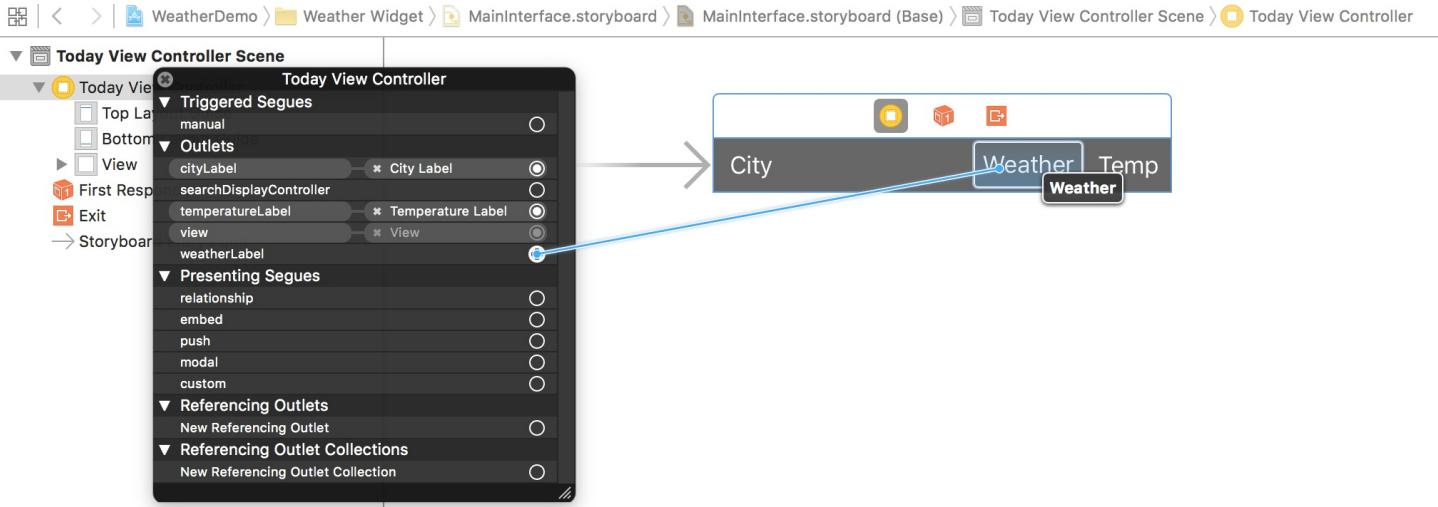
```
import WeatherInfoKit
```

Next, declare three outlet variables and an instance variable for the default location in `TodayViewController.swift`:

```
@IBOutlet var cityLabel:UILabel!
@IBOutlet var weatherLabel:UILabel!
@IBOutlet var temperatureLabel:UILabel!

var location = "Paris, France"
```

Go back to `MainInterface.storyboard` of the widget. Right click the Today View Controller in the Document Outline. Connect the outlet variables with the labels.



To present the weather information in the widget's view controller, insert the `viewDidAppear` method in `TodayViewController.swift`:

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

    cityLabel.text = location

    // Invoke weather service to get the weather data
    WeatherService.sharedWeatherService().getCurrentWeather(location) { (data)
-> () in
        NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
            if let weatherData = data {
                self.weatherLabel.text = weatherData.weather.capitalizedString
                self.temperatureLabel.text = String(format: "%d",
weatherData.temperature) + "\u{00B0}"
            }
        })
    }
}
```

In the method, we simply call the API provided by the `WeatherInfoKit` framework that we created earlier to secure the weather information. To enable the widget to update its view when it's off-screen, make the following changes to the `widgetPerformUpdateWithCompletionHandler` method:

```
func widgetPerformUpdateWithCompletionHandler(completionHandler:
((NCUpdateResult) -> Void)) {
    // Perform any setup necessary in order to update the view.

    // If an error is encountered, use NCUpdateResult.Failed
```

```

// If there's no update required, use NCUpdateResult.NoData
// If there's an update, use NCUpdateResult.NewData

cityLabel.text = location

WeatherService.sharedWeatherService().getCurrentWeather(location) { (data)
-> () in
    guard let weatherData = data else {
        completionHandler(NCUpdateResult.NoData)
        return
    }

    NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
        self.weatherLabel.text = weatherData.weather.capitalizedString
        self.temperatureLabel.text = String(format: "%d",
weatherData.temperature) + "\u{00B0}"
    })

    completionHandler(NCUpdateResult.NewData)
}
}

```

The method is called automatically to give you an opportunity to update the widget's content. Here we retrieve the latest weather information for our location. If it updates successfully, the function calls the system-provided completion block with the `NCUpdateResult.NewData` enumeration. If the update wasn't successful, then the existing snapshot is used, which is indicated by `NCUpdateResult.NoData`.

Now compile and run the widget on iOS 9. You will end up with this exception in the console:

```

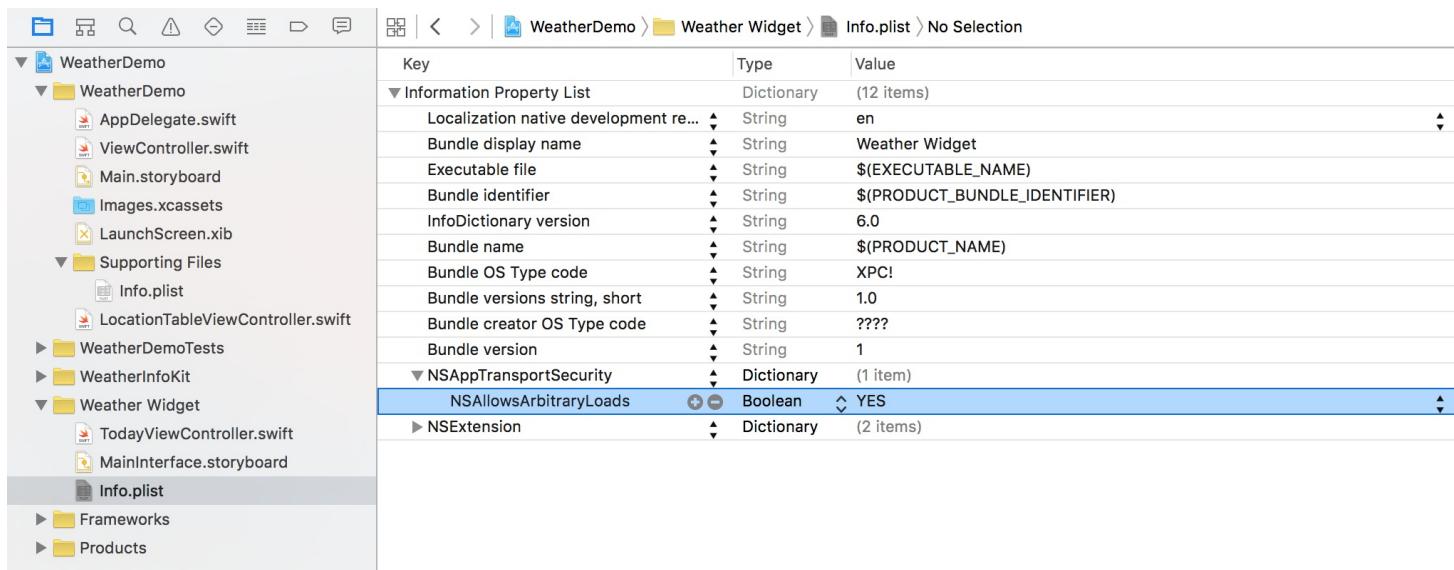
Optional(Error Domain=NSURLErrorDomain Code=-1022 "The resource could not be
loaded because the App Transport Security policy requires the use of a secure
connection." UserInfo={NSErrorUnderlyingError=0x7fcab15f3860 {Error
Domain=kCFErrorDomainCFNetwork Code=-1022 "(null)"},
NSErrorFailingURLStringKey=http://api.openweathermap.org/data/2.5/weather?
appid=bd82977b86bf27fb59a04b61b657fb6f&units=metric&q=Paris,%20France,
NSErrorFailingURLKey=http://api.openweathermap.org/data/2.5/weather?
appid=bd82977b86bf27fb59a04b61b657fb6f&units=metric&q=Paris,%20France,
NSErrorLocalizedDescription=The resource could not be loaded because the App
Transport Security policy requires the use of a secure connection.})

```

App Transport Security is first introduced in iOS 9. The purpose of the feature is to improve the security of connections between an app and web services by enforcing some of the best practices. One of them is the use of secure connections. With ATS, all network requests should

now be sent over HTTPS. If you make a network connection using HTTP, ATS will block the request and display the error. For the API provided by openweathermap.org, it only comes with the support of HTTP. To resolve the issue, one way is to opt out of App Transport Security. To do so, you need to add a specific key in the widget's Info.plist to disable ATS.

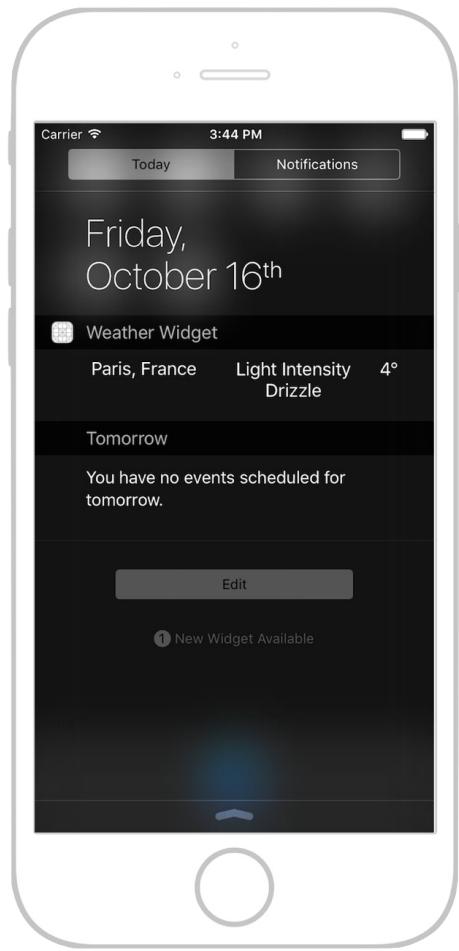
Select `Info.plist` under the `Weather Widget` folder in the project navigator to display the content in a property list editor. To add a new key, right click the editor and select *Add Row*. For the key column, enter `NSAppTransportSecurity`. Set the type to `Dictionary`. Next, add the `NSAllowsArbitraryLoads` key with the type `Boolean`. By setting `NSAllowsArbitraryLoads` to `YES`, you explicitly disable App Transport Security.



The screenshot shows the Xcode project navigator on the left and a property list editor on the right. The project navigator lists several files and folders: WeatherDemo, WeatherDemo (with AppDelegate.swift, ViewController.swift, Main.storyboard, Images.xcassets, LaunchScreen.xib), Supporting Files (Info.plist, LocationTableViewCellController.swift), WeatherDemoTests, WeatherInfoKit, Weather Widget (TodayViewController.swift, MainInterface.storyboard), and Frameworks/Products. The `Info.plist` file is selected in the Weather Widget folder. The property list editor shows the following keys:

Key	Type	Value
Localization native development region	String	en
Bundle display name	String	Weather Widget
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 creator OS Type code	String	????
Bundle version	String	1
NSAppTransportSecurity	Dictionary	(1 item)
NSAllowsArbitraryLoads	Boolean	YES
NSExtension	Dictionary	(2 items)

Now run the app again. It should be able to load the widget. The weather widget should look like this:



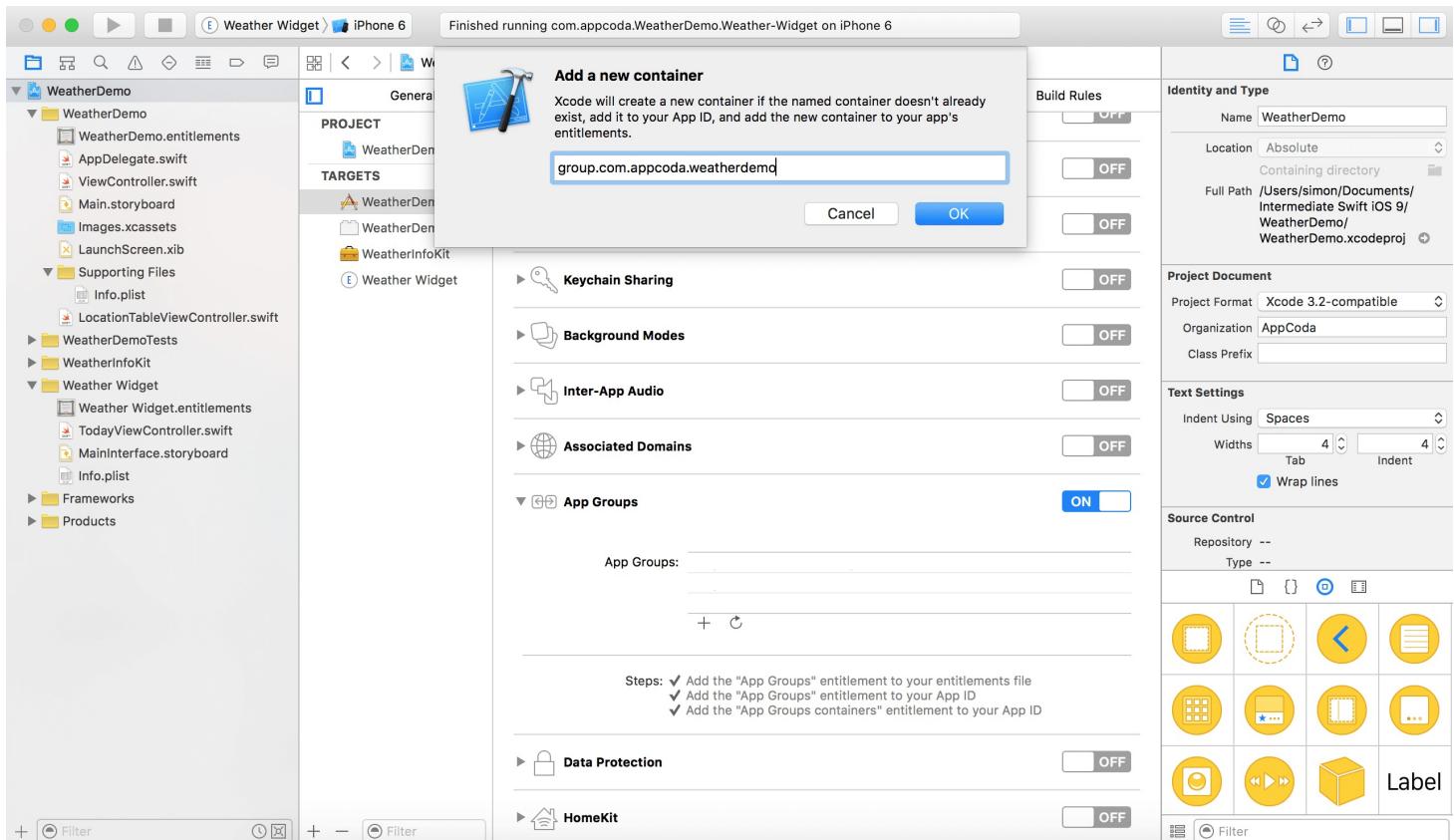
Sharing Data with the Container App

The WeatherDemo app (i.e. the container app) provides a Setting screen for users to change the default location. Tap the hamburger button in the top-left corner of the screen and change the default location (say, New York) of the app. If you've done everything correctly so far, the WeatherDemo app should now display the weather information of your preferred location.

However, the weather widget is not updated accordingly. We need to figure out a way to pass the default location to the weather widget. Currently the default location of the widget is hardcoded to *Paris, France*. As mentioned before, your extension and its containing app have no direct access to each other's containers. You can, however, share the setting through `NSUserDefaults`. To enable data sharing you have to enable app groups for the containing app (i.e. WeatherDemo) and its app extension (i.e. Weather Widget).

To get started, select your main app target (i.e. WeatherDemo) and choose the *Capabilities* tab. Switch on `App Groups` (you will need a developer account for this). Click the `+` button to

create a new container and give it a unique name. Commonly, the name starts with `group`. I set the name to `group.com.appcoda.weatherdemo`. Select the `Weather Widget` target and repeat the above procedures to set the App Groups. Don't create a new container for it though - use the one you had created for the WeatherDemo target.



After you enable app groups, an app extension and its containing app can both use the `NSUserDefaults` API to share access to user settings. Open `LocationTableViewController.swift` and add the following property to the class:

```
var defaults = UserDefaults(suiteName: "group.com.appcoda.weatherdemo")!
```

The `LocationTableViewController` class is the controller for handling the location selection. To enable data sharing, we create a new `UserDefaults` object with the suite name set to the group name. Update the following method:

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let cell = tableView.cellForRowAtIndexPath(indexPath)
    cell?.accessoryType = .Checkmark
    if let location = cell?.textLabel?.text {
```

```
        selectedLocation = location
        defaults.setValue(selectedLocation, forKey: "location")
    }

    tableView.reloadData()
}
```

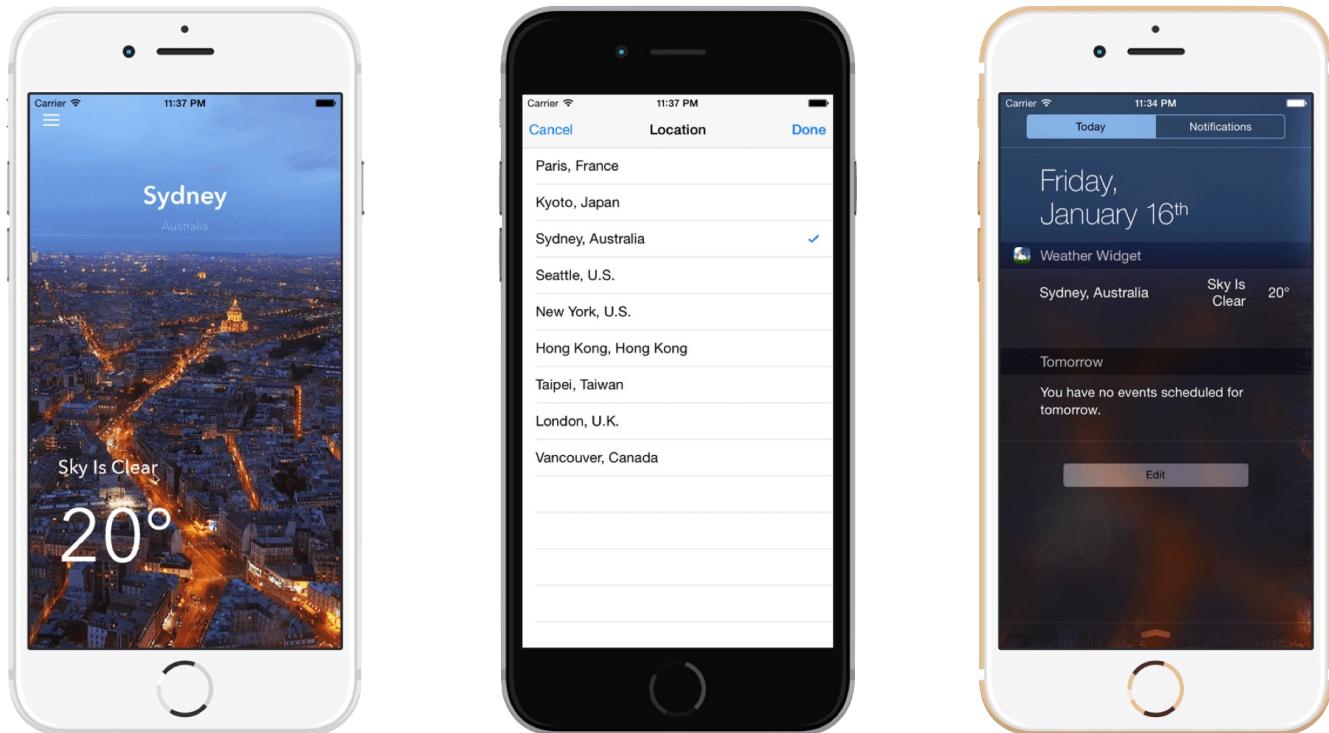
We only add a line of code in the `if let` block to save the selected location to the defaults. Next, open `TodayViewController.swift` and add the following variable:

```
var defaults = UserDefaults(suiteName: "group.com.appcoda.weatherdemo")!
```

In the `widgetPerformUpdateWithCompletionHandler` method, insert the following lines of code at the beginning:

```
// Get the location from defaults
if let defaultLocation = defaults.value(forKey("location")) as? String {
    location = defaultLocation
}
```

Here we simply retrieve the location from `UserDefaults`, which is the location set by the user. Now we are ready to test the widget again. Run the app and change the default location. Once the location is set, activate the Notification Center to review the weather widget, which should be updated according to your preference.

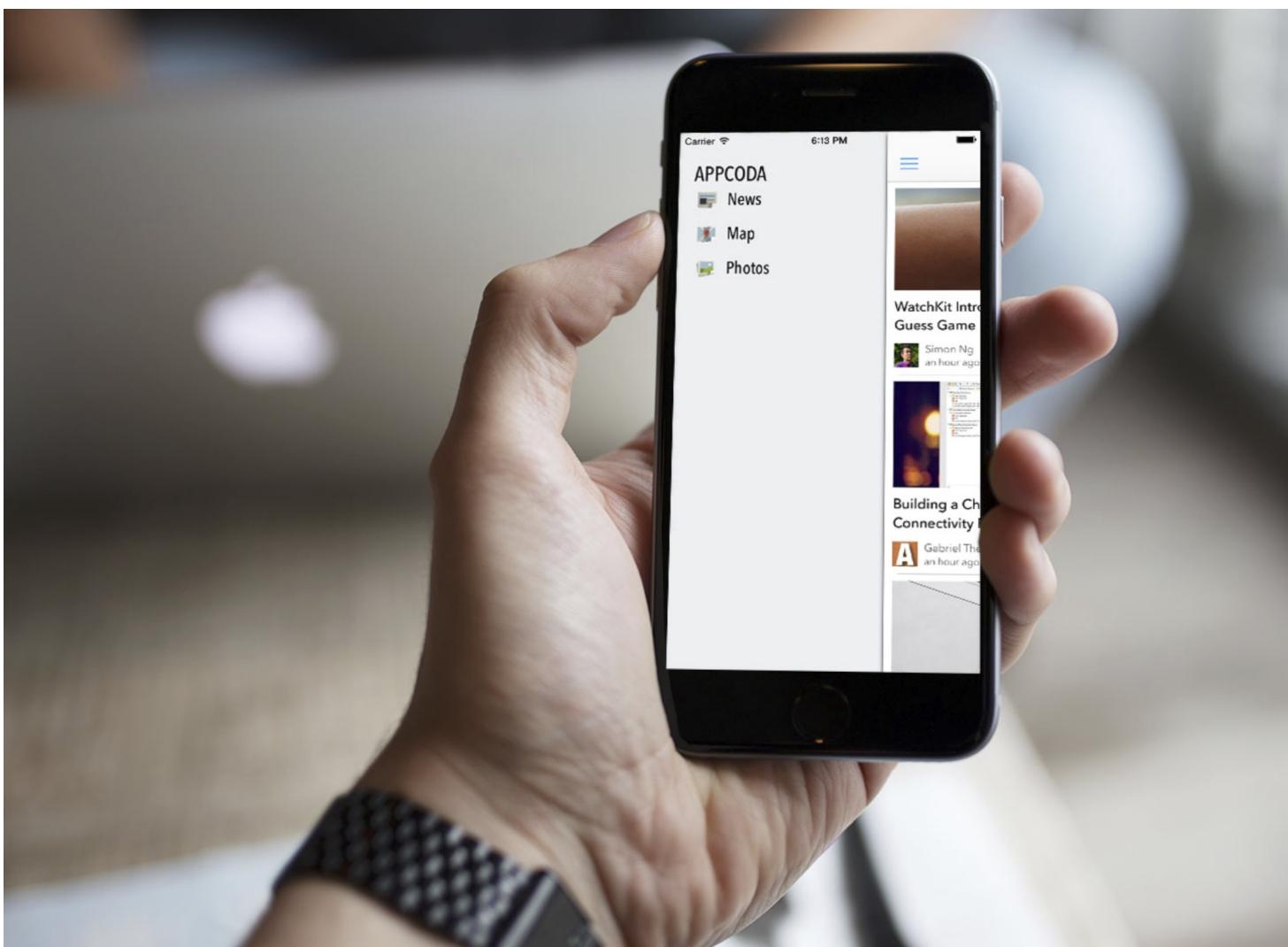


For reference, you can download the complete Xcode project from

<https://www.dropbox.com/s/do6xodquqwzbknq/WeatherDemoFinal.zip?dl=0>.

Chapter 22

Building Slide Out Sidebar Menus



In this chapter I will show you how create a slide-out navigation menu similar to the one you find in the Gmail app. If you're unfamiliar with slide out navigation menus, take a look at the figure above. Ken Yarmost (<http://kenyarmosh.com/ios-pattern-slide-out-navigation/>) gave a good explanation and defined it as follows:

Slide-out navigation consists of a panel that slides out from underneath the left or the right of the main content area, revealing a vertically independent scroll view that serves as the primary navigation for the application.

The slide-out sidebar menu (also known as a hamburger menu) has been around for a few years now. It was first introduced by Facebook in 2011. Since then it has become a standard way to implement a navigation menu. The slide-out design pattern lets you build a navigation menu in your apps but without wasting the screen real estate. Normally, the navigation menu is hidden behind the front view. The menu can then be triggered by tapping a list button in the navigation bar. Once the menu is expanded and becomes visible, users can close it by using the list button or simply swiping left on the content area.

Lately, there are some debates (<https://lmjabreu.com/post/why-and-how-to-avoid-hamburger-menus/>) about this kind of menu that it doesn't provide a good user experience and less efficient. In most cases, you should prefer tab bars over sidebar menus for navigation. Being that said, you can still easily find this design pattern in some popular content-related apps, including Google Maps, Pocket, LinkedIn, etc. The purpose of this chapter is not to discuss with you whether you should kill the hamburger menu. There are already a lot of discussions out there:

- Kill The Hamburger Button (<http://techcrunch.com/2014/05/24/before-the-hamburger-button-kills-you/>)
- Why and How to Avoid Hamburger Menus by Luis Abreu (<https://lmjabreu.com/post/why-and-how-to-avoid-hamburger-menus/>)
- Hamburger vs Menu: The Final AB Test (<http://exisweb.net/menu-eats-hamburger>)

So our focus in this chapter is on how. I want to show you how to create a slide-out sidebar menu using a free library.

You can build the sidebar menu from the ground up. But with so many free pre-built solutions on GitHub, we're not going to build it from scratch. Instead, we'll make use of a library called SWRevealViewController (<https://github.com/John-Lluch/SWRevealViewController>). Developed by John Lluch, this excellent library provides a quick and easy way to put up a slide-out navigation menu in your apps. Best of all, the library is available for free.

The library was written in Objective-C. By going through the tutorial, you will also learn how to use Objective-C in a Swift project.

A Glance at the Demo App

As usual, we'll build a demo app to show you how to apply `SWRevealViewController`. This app is very simple but not fully functional. The primary purpose of this app is to walk you through the implementation of slide-out navigation menu. The navigation menu will work like this:

- The user triggers the menu by tapping the list button at the top-left of navigation bar.
- The user can also bring up the menu by swiping right on the main content area.
- Once the menu appears, the user can close it by tapping the list button again.
- The user can also close the menu by dragging left on the content area.

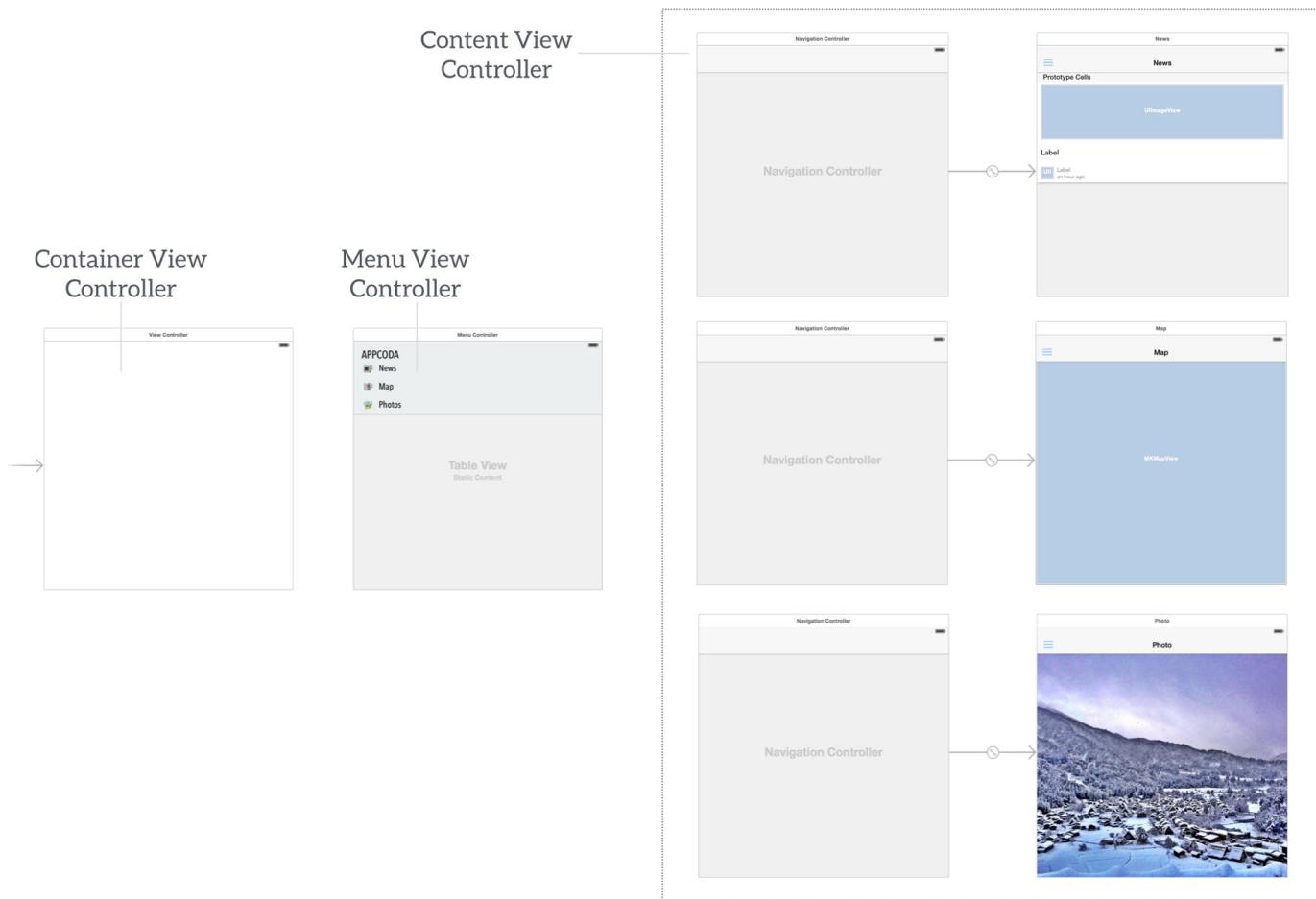


Creating the Xcode Project

This chapter focuses on the sidebar implementation. If you want to save time and avoid building the project from scratch, you can download the Xcode project template from <https://www.dropbox.com/s/6noglr15ibcsl4/SidebarMenuStart.zip?dl=0>.

The project comes with a pre-built storyboard with all the required view controllers. If you've downloaded the template, open the storyboard to take a look. To use `SWRevealViewController` for building a sidebar menu, you create a container view controller, which is actually an empty

view controller, to hold both the menu view controller and a set of content view controllers.



I have already created the menu view controller for you. It is just a static table view with three menu items. There are three content view controllers for displaying *news*, *maps*, and *photos*. For demo purposes, there are three content view controllers, and they show only static data. If you need to have a few more controllers, simply insert them into the storyboard.

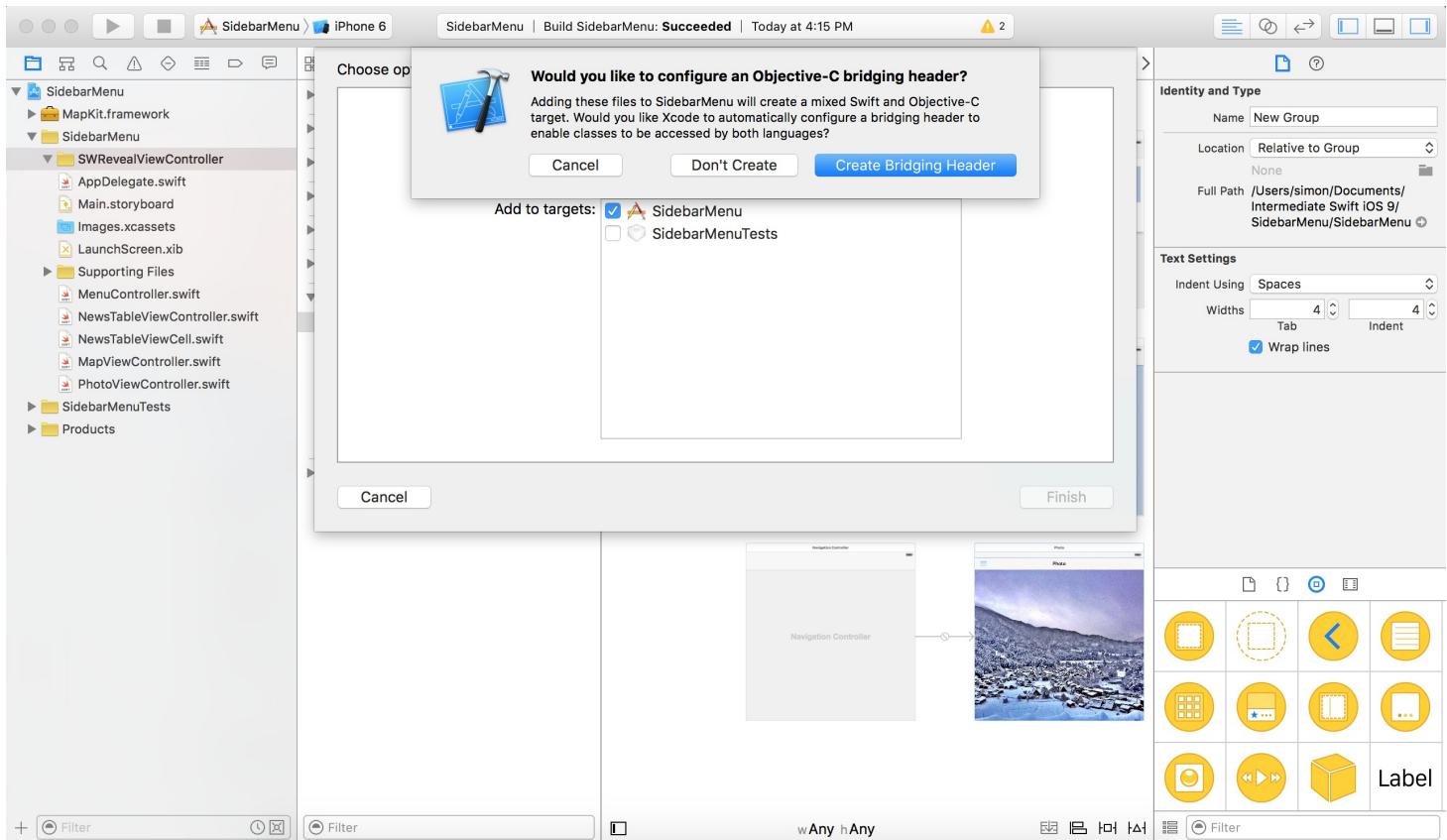
All icons and images are included in the project template (credit: thanks to [Pixeden](#) for the free icon).

Using the SWRevealViewController Library

As mentioned, we'll use the free `SWRevealViewController` library to implement the slide-out menu. To begin, download the library from GitHub (<https://github.com/John-Lluch/SWRevealViewController/archive/master.zip>) and extract the zipped file - you should see the `SWRevealViewController` folder. In that folder there are two files:

`SWRevealViewController.h` and `SWRevealViewController.m`. If you don't have a background in Objective-C, you might wonder why the file extension is not `.swift`. As mentioned before, the `SWRevealViewController` library was written in Objective-C; the file extension differs from that of Swift's source file. We will add both files to the project.

In the project navigator, right-click `SidebarMenu` folder and select `New Group`. Name the group `SWRevealViewController`. Drag both files to the `SWRevealViewController` group. When prompted, make sure the *Copy items if needed* option is checked. As soon as you confirm to add the files, Xcode prompts you to configure an Objective-C bridging header.



By creating the header file, you'll be able to access the Objective-C code from Swift. Click *Create Bridging Header* to proceed. Xcode then generates a header file named `SidebarMenu-Bridging-Header.h` under the `SWRevealViewController` folder. Open the `SidebarMenu-Bridging-Header.h` and insert the following line:

```
#import "SWRevealViewController.h"
```

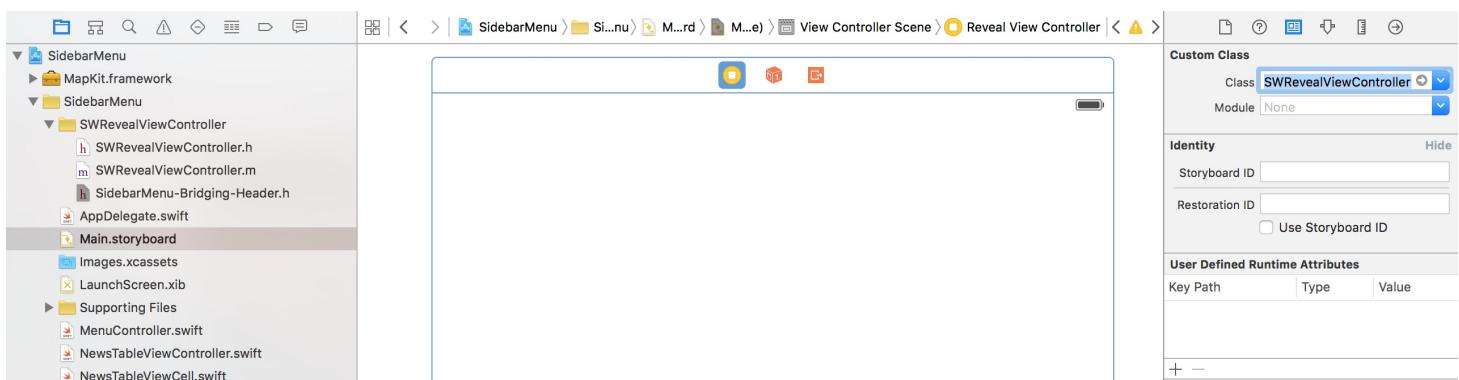
By adding the header file of `SWRevealViewController`, our Swift project will be able to access the Objective-C library. This is all you need to do when using an external library written in

Objective-C.

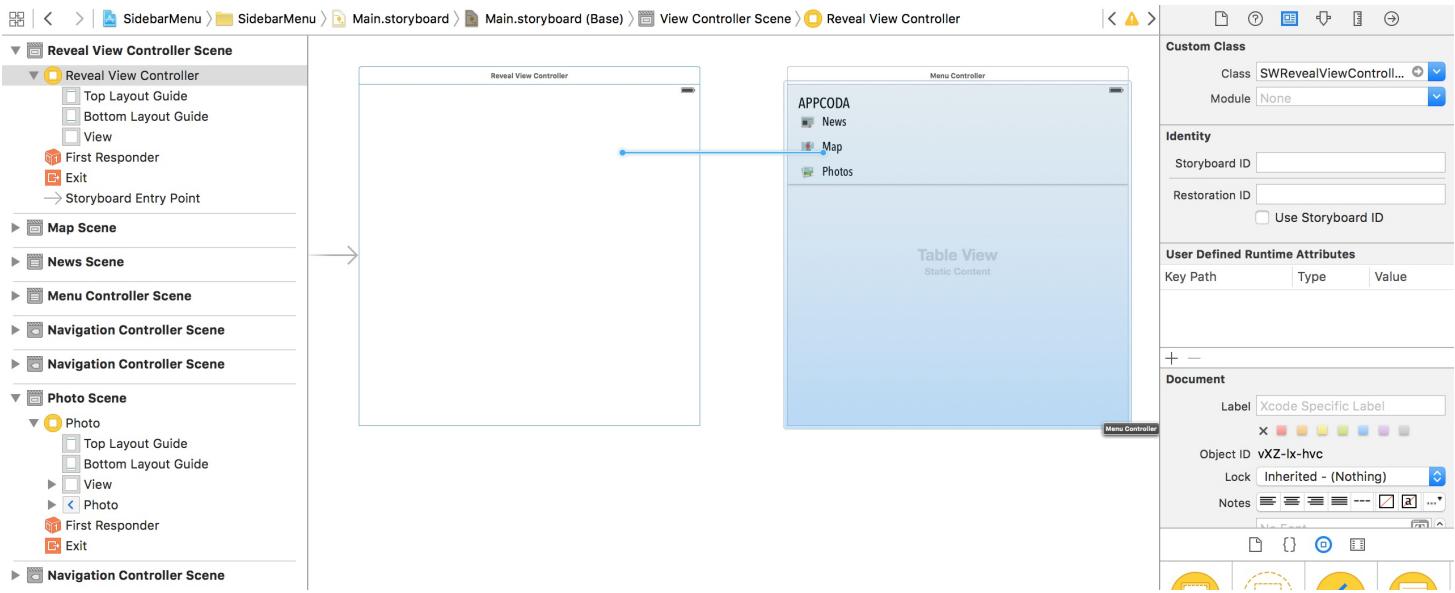
Associate the Front View and Rear View Controller

The `SWRevealViewController` library provides built-in support for Interface Builder. When implementing a sidebar menu, all you need to do is associate the `SWRevealViewController` object with a front and a rear view controller using segues. The front view controller is the main controller for displaying content. In our storyboard this is the navigation controller which associates with a view controller for presenting news. The rear view controller is the controller that shows the navigation menu. Here it is the Sidebar View Controller.

Go to the storyboard. First, select the empty view controller (i.e. container view controller) and change its class to `SWRevealViewController`.

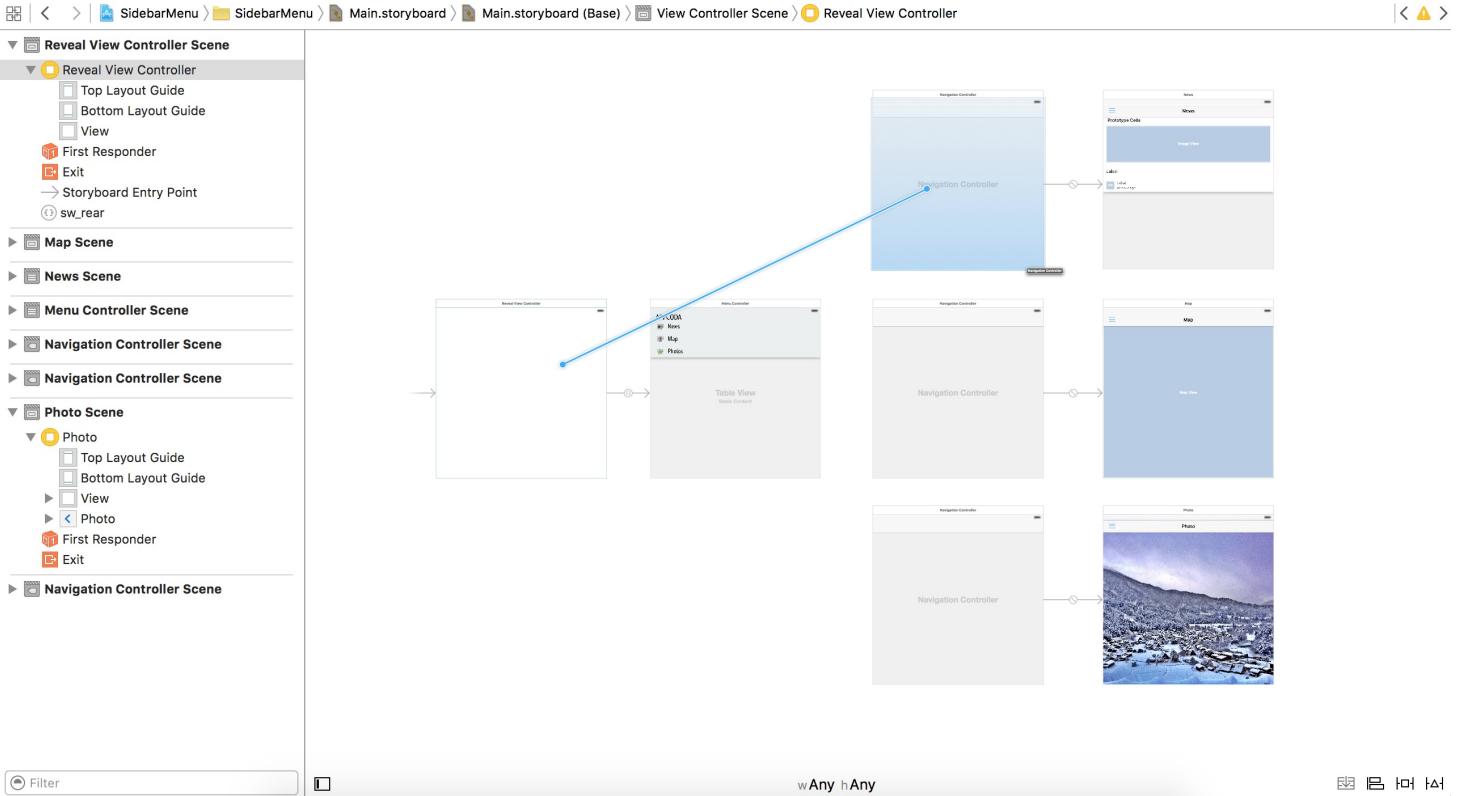


Next, control-drag from `SWRevealViewController` to the Menu view controller. After releasing the button, you will see a context menu for segue selection. In this case, select `reveal view controller set segue`.



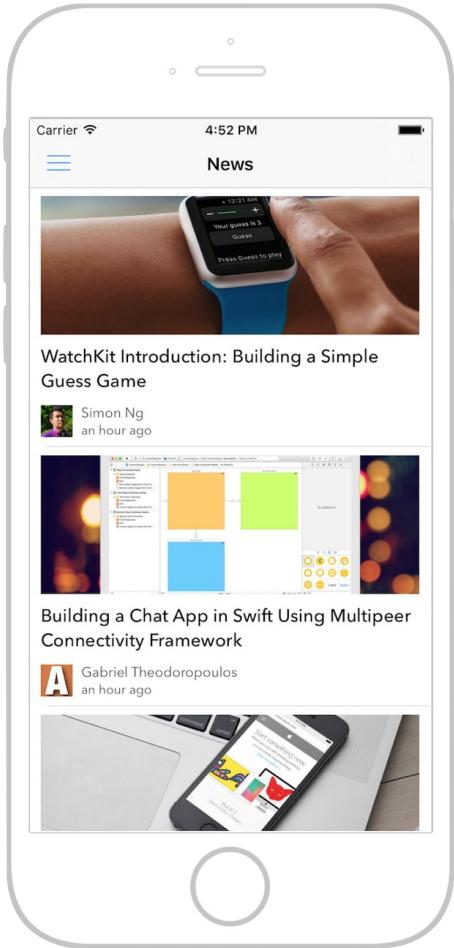
This defines a custom segue known as `SWRevealViewControllerSegue`. Select this segue and change its identifier to `sw_rear` under the Identity inspector. By setting the identifier, you tell `SWRevealViewController` that the menu view controller is the *rear view controller*. This means that the sidebar menu will be hidden behind a content view controller.

Next, repeat the same procedures to connect `SWRevealViewController` with the navigation controller of the News View Controller. Again, select `reveal view controller set segue` when prompted.



Set the identifier of the segue to `sw_front`. This tells the `SWRevealViewController` that the navigation controller is the front view controller.

You can now compile the app and test it before moving on. At this point, your app should display the News view. You will not be able to pull out the sidebar menu when tapping the menu button (aka the hamburger button) because we haven't implemented that action method yet.



If your app works properly, let's continue with the implementation. If it doesn't work properly, go back to the beginning of the chapter and work through step-by-step to figure out where you went wrong.

Open `NewsTableViewController.swift`, which is the controller class of News Controller. In the `viewDidLoad` method, insert the following lines of code:

```
if revealViewController() != nil {
    menuButton.target = revealViewController()
    menuButton.action = "revealToggle:"}

view.addGestureRecognizer(self.revealViewController().panGestureRecognizer())
}
```

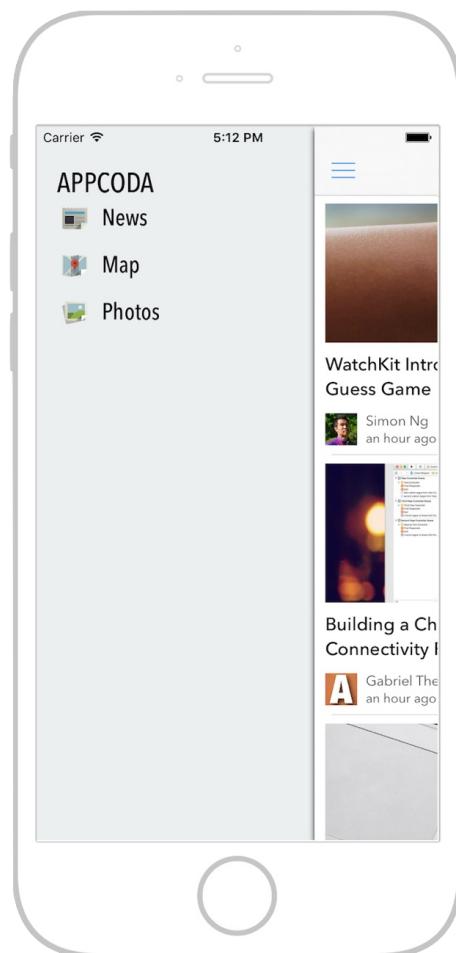
The `SWRevealViewController` class provides a method called `revealViewController()` to get the parent `SWRevealViewController` from any child controller. It also provides the `revealToggle:` method to handle the expansion and contraction of the sidebar menu. As you know, Cocoa uses the target-action mechanism for communication between a control and another object. We set

the target of the menu button to the reveal view controller and action to the `revealToggle:` method. So when the menu button is tapped, it will call the `revealToggle:` method to display the sidebar menu.

Using Objective-C from Swift: The action property of the menu button accepts an Objective-C selector. An Objective-C selector is a type that refers to the name of an Objective-C method. In Swift, you just need to specify the method name as a string literal to construct a selector.

Lastly, we add a gesture recognizer. Not only you can use the menu button to bring out the sidebar menu, but the user can swipe the content area to activate the sidebar as well.

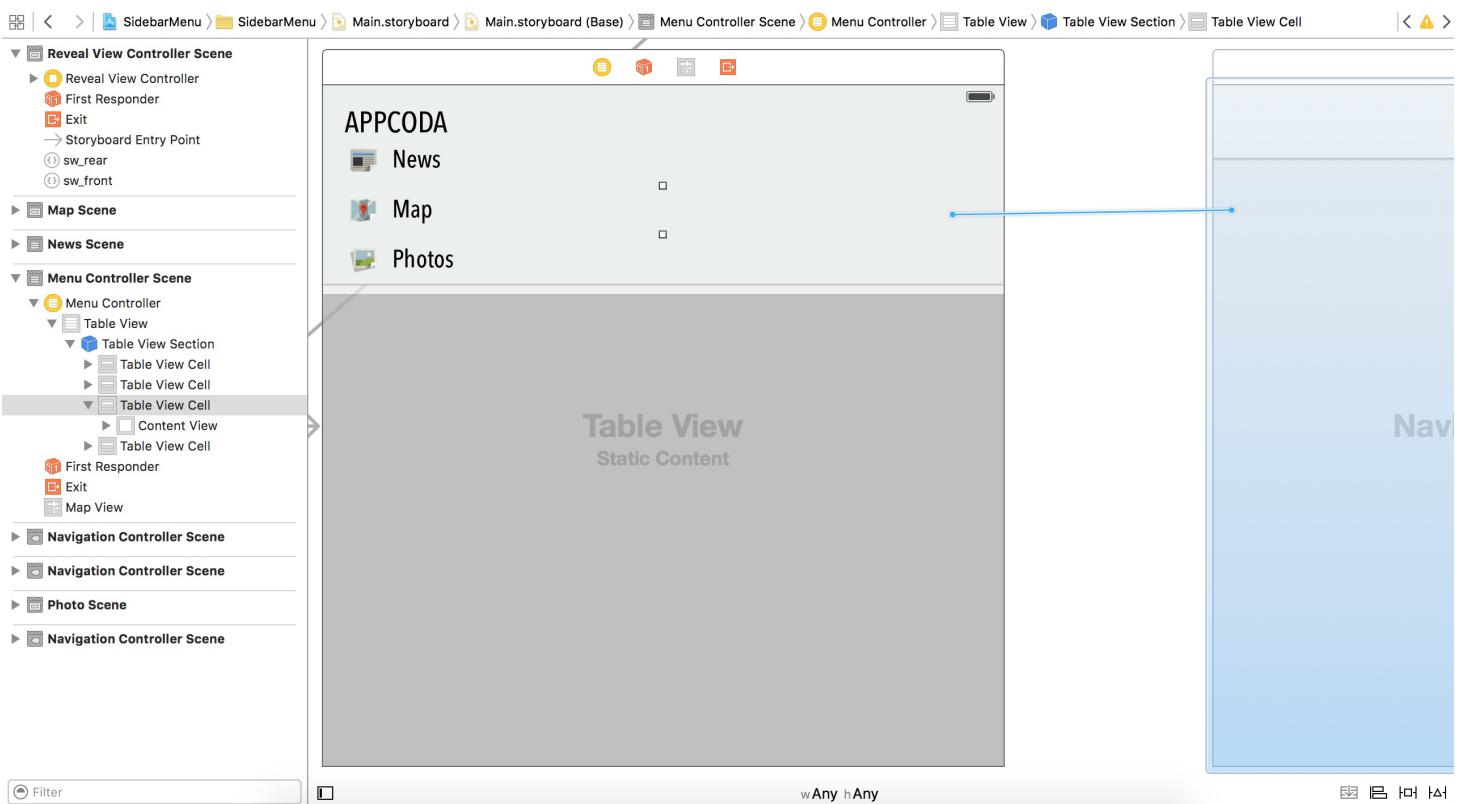
Cool! Let's compile and run the app in the simulator. Tap the menu button and the sidebar menu should appear. You can hide the sidebar menu by tapping the menu button again. You can also open the menu by using gesture. Try to swipe right on the content area and see what you get.



Handling Menu Item Selection

Now that you've already built a visually appealing sidebar menu, there's only one thing left. For now, we haven't defined any segues for the menu items. When you select any of the menu items, they will not transit to the corresponding view.

Okay, go back to `Main.storyboard`. First, select the map cell. Control-drag from the map cell to the navigation controller of the map view controller, and then select the `reveal view controller push controller` segue under Selection Segue. Repeat the procedure for the News and Photos items, but connect them with the navigation controllers of the news view controller and photos view controller respectively.



The custom `SWRevealViewControllerSeguePushController` segue automatically handles the switching of the controllers. Similarly, insert the following lines of code in the `viewDidLoad` method of `MapViewController.swift` and `PhotoViewController.swift` to toggle the sidebar menu:

```
if revealViewController() != nil {  
    menuButton.target = revealViewController()  
    menuButton.action = "revealToggle:"
```

```
view.addGestureRecognizer(self.revealViewController().panGestureRecognizer())
}
```

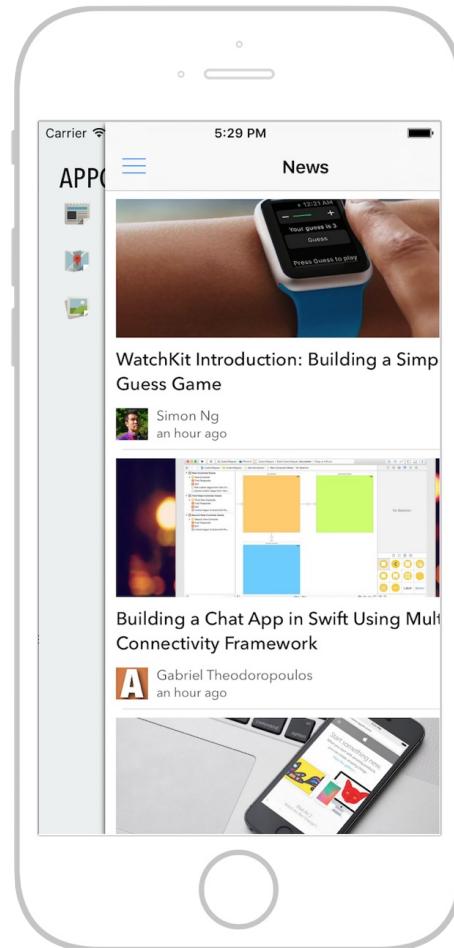
That's it! Hit the Run button and test out the app.

Customizing the Menu

The `SWRevealViewController` class provides a number of options for configuring the sidebar menu. For example, if you want to change the width of the menu you can update the value of the `rearViewRevealWidth` property. Try to insert the following line of code in the `viewDidLoad` method of `NewsTableViewController`:

```
revealViewController().rearViewRevealWidth = 62
```

When you run the app, you should have a sidebar menu like the one shown below. You can look into the `SWRevealViewController.h` file to explore more customizable options.

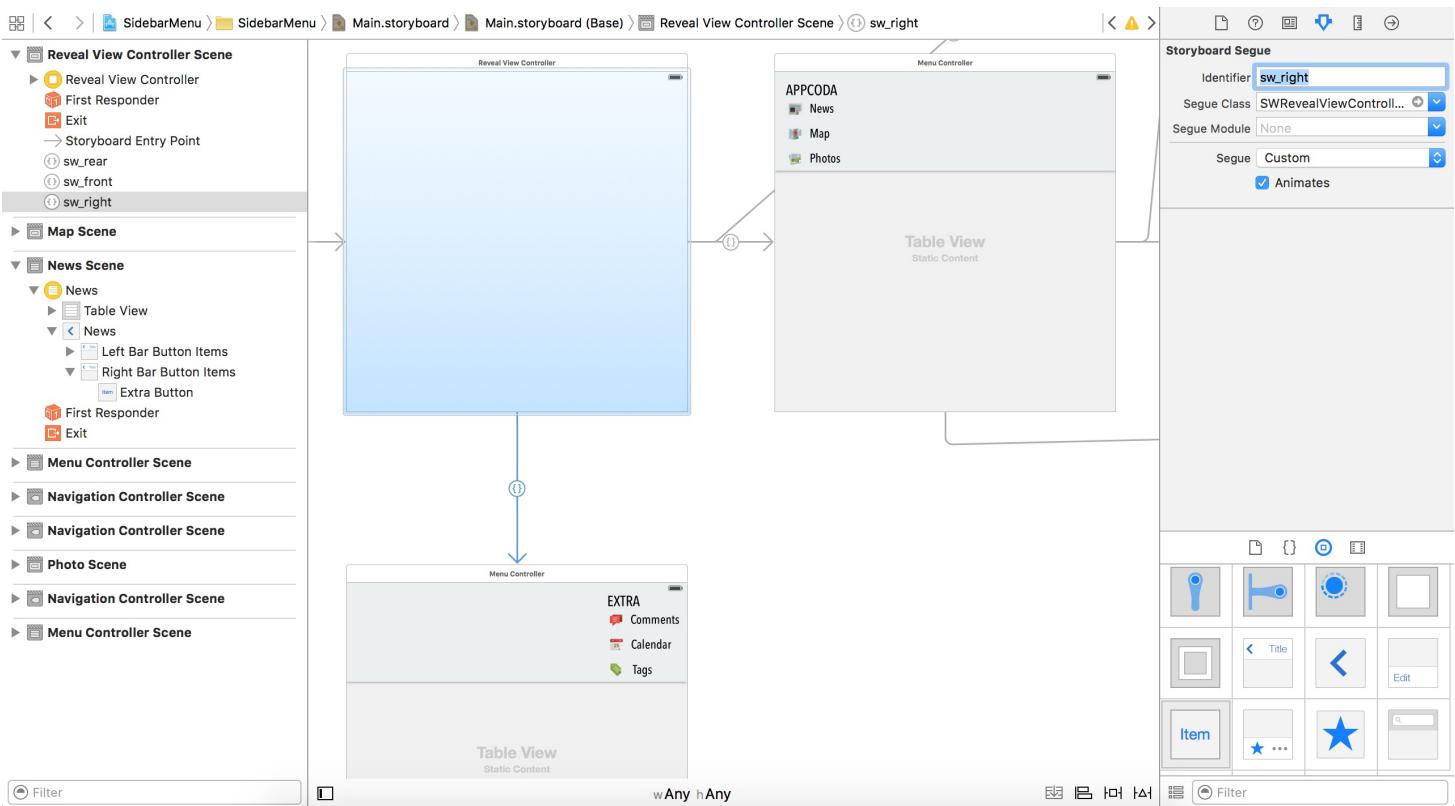


Adding a Right Sidebar

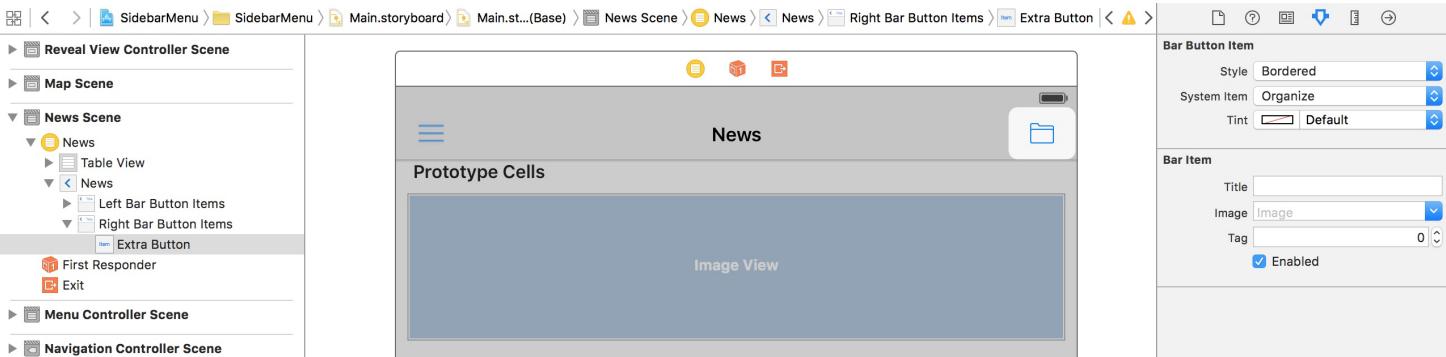
Sometimes, you may need an extra sidebar. For example, the Facebook app allows users to swipe from the right to reveal a right sidebar, showing a list of the favorite contacts. With `SWRevealViewController`, it is quite simple to add an extra sidebar.

In the demo storyboard, it already comes with an *Extra* menu view controller. The procedures to add a right sidebar is very similar to what we have already did. The trick is to change the identifier of the segue from `sw_rear` to `sw_right`. Let's see how to get it done.

In `Main.storyboard`, control-drag from `SWRevealViewController` to the *Extra* menu view controller. In the pop-over menu, select `reveal view controller set segue`. Then select the segue and go to the Identity inspector. Set the identifier to `sw_right`. This tells `SWRevealViewController` that the *Extra* menu view controller should be slided from right.



Now drag a bar button item to the News view controller, and place it at the right side of the navigation bar. In the Identity inspector, set the *System Item* option to `organize`.



In the project navigator, select `NewsTableViewController.swift`. Declare an outlet variable for the bar button item:

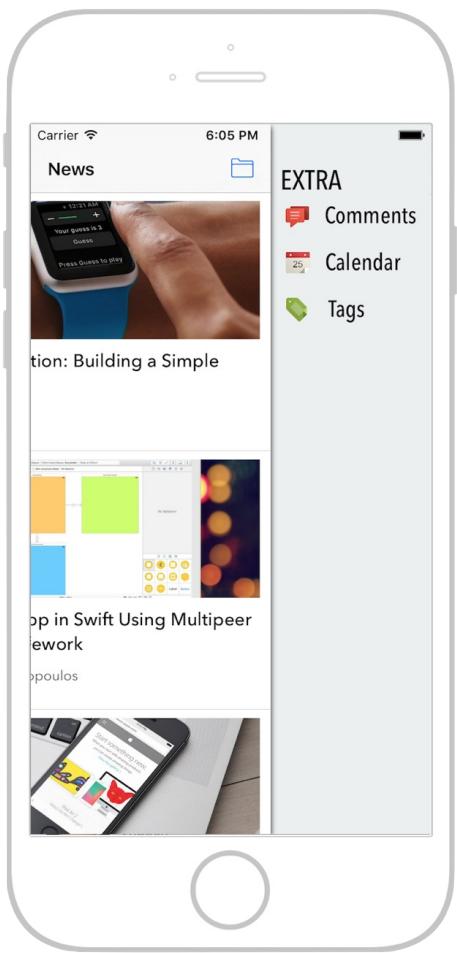
```
@IBOutlet var extraButton: UIBarButtonItem!
```

In the `viewDidLoad` method, insert these lines of code right before the `view.addGestureRecognizer` method call:

```
revealViewController().rightViewRevealWidth = 150
extraButton.target = revealViewController()
extraButton.action = "rightRevealToggle:"
```

Here we change the width of the extra menu to `150`, and set the corresponding `target` / `action` properties. Instead of calling the `revealToggle:` method when the extra button is tapped, we call the `rightRevealToggle:` method to display the menu from the right.

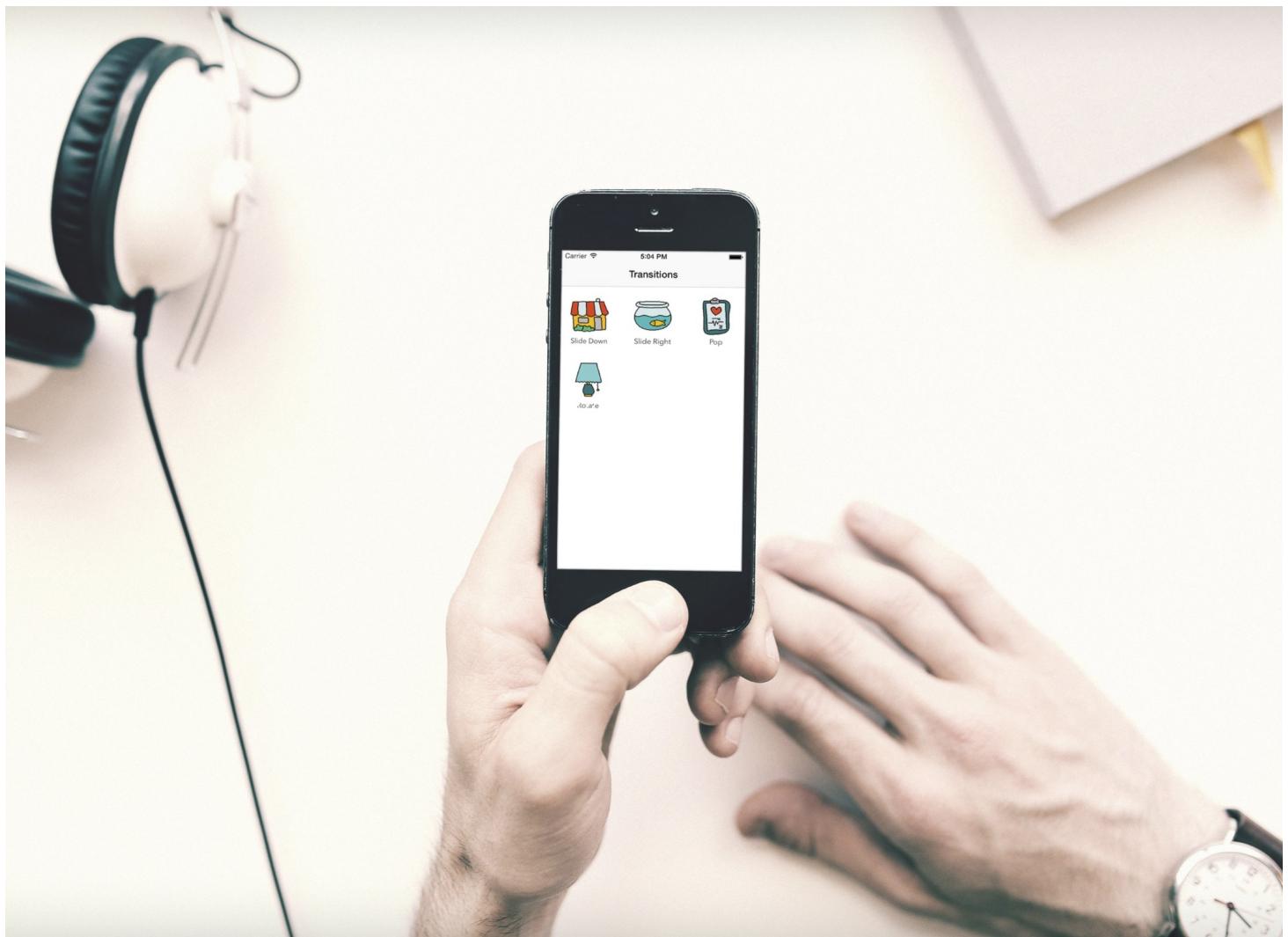
Lastly, go back to the storyboard. Connect the `organize` button with the outlet variable. That's it! Run the project again. Now the app has a right sidebar.



For your reference, you can download the final project from
<https://www.dropbox.com/s/axozhk655j204zu/SidebarMenu.zip?dl=0>.

Chapter 23

View Controller Transitions and Animations



Wouldn't it be great if you could define the transition style between view controllers? Apple provides a handful of default animations for view controller transitions. Presenting a view controller modally usually uses a slide-up animation. The transition between two view controllers in navigation controller is predefined too. Pushing or popping a controller from the navigation controller's stack uses a standard slide animation. In older versions of iOS, there was no easy way to customize the transitions of two view controllers. Starting from iOS 7, iOS developers are allowed to implement our own transitions through the **View Controller**

Transitioning API. The API gives you full control over how one view controller presents another.

There are two types of view controller transitions: interactive and non-interactive. In iOS 7 (or up), you can pan from the leftmost edge of the screen and drag the current view to the right to pop a view controller from the navigation controller's stack. This is a great example of interactive transition. In this chapter, we are going to focus on the non-interactive transition first, as it is easier to understand.

The concept of custom transition is pretty simple. You create an animator object (or so called *transition manager*), which implements the required custom transition. This animator object is called by the UIKit framework when one view controller starts to present or transit to another. It then performs the animations and informs the framework when the transition completes.

When implementing non-interactive view controller transitions, you basically deal with the following protocols:

- `UIViewControllerAnimatedTransitioning` - your animator object must adopt this protocol to create the animations for transitioning a view controller on or off screen.
- `UIViewControllerTransitioningDelegate` - you adopt this protocol to vend the animator objects used to present and dismiss a view controller. Interestingly, you can provide different animator objects to manage the transition between two view controllers.
- `UIViewControllerContextTransitioning` - this protocol provides methods for accessing contextual information for transition animations between view controllers. You do not need to adopt this protocol in your own class. Your animator object will receive the context object, provided by the system, during the transition.

It looks a bit complicated, right? Actually, it's not. Once you go through a simple project, you will have a better idea about how to build custom transitions between view controllers.

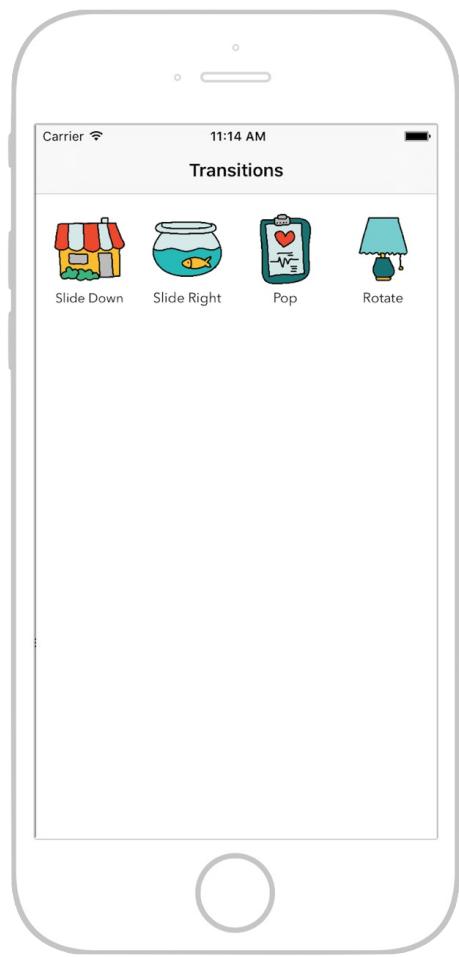
Demo Project

We are going to build a simple demo app. To keep your focus on building the animations, download the project template from

<https://www.dropbox.com/s/nnsj9doreuu2uzm/NavTransitionStart.zip?dl=0>. The template comes with prebuilt storyboard and view controller classes.

Quick note: The user interface was built using collection view. If you do not have any experience with UICollectionView, read over chapter 18 and 20.

If you have a trial run, you will end up with a screen similar to the one shown below.

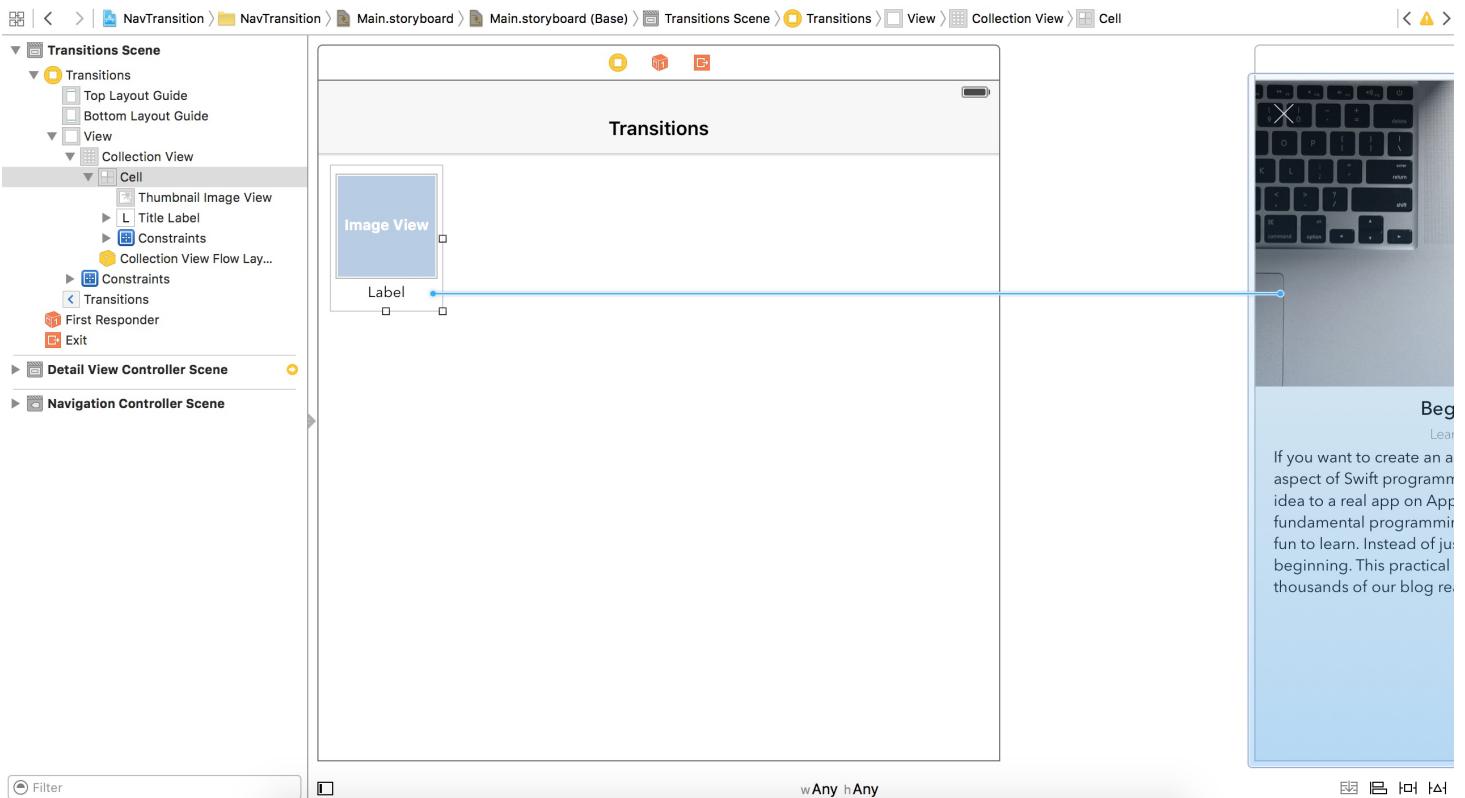


Each icon indicates a unique custom transition. For now, the icons are not functional. Coming up next, you will learn how to implement all the transitions using the View Controller Transitioning API.

Applying the Standard Transition

First, open `Main.storyboard`. You should find the transitions view controller (embedded in a navigation controller) and the detail view controller showing product information. These two controllers are not connected yet. Control-drag from the collection view cell of the transition

view controller to the detail view controller. Select `Present modally` as the selection segue.



If you run the demo app again, it will bring up the detail view controller using the standard slide-up animation. What we are going to do is implement our own animator object to replace that animation.

Creating a Slide Down Animator

In the project navigator, right-click to create a new file. Name the class `SlideDownTransitionAnimator` and set it as a subclass of `NSObject`. As mentioned earlier, the animator object should adopt both the `UIViewControllerAnimatedTransitioning` and the `UIViewControllerTransitioningDelegate` protocols. So update the class declaration like this:

```
class SlideDownTransitionAnimator: NSObject,  
UIViewControllerAnimatedTransitioning, UIViewControllerTransitioningDelegate {  
}
```

Let's first talk about the `UIViewControllerTransitioningDelegate` protocol. You adopt this protocol to vend the animator objects that manage the transition between view controllers. You have to implement the following methods in the `SlideDownTransitionAnimator` class:

```

func animationControllerForPresentedController(presented: UIViewController,
presentingController presenting: UIViewController, sourceController source:
UIViewController) -> UIViewControllerAnimatedTransitioning? {
    return self
}

func animationControllerForDismissedController(dismissed: UIViewController) ->
UIViewControllerAnimatedTransitioning? {
    return self
}

```

The `animationControllerForPresentedController` method returns the animator object to use when presenting the view controller. Here we return `self`, as the `slideDownTransitionAnimator` object is the animator.

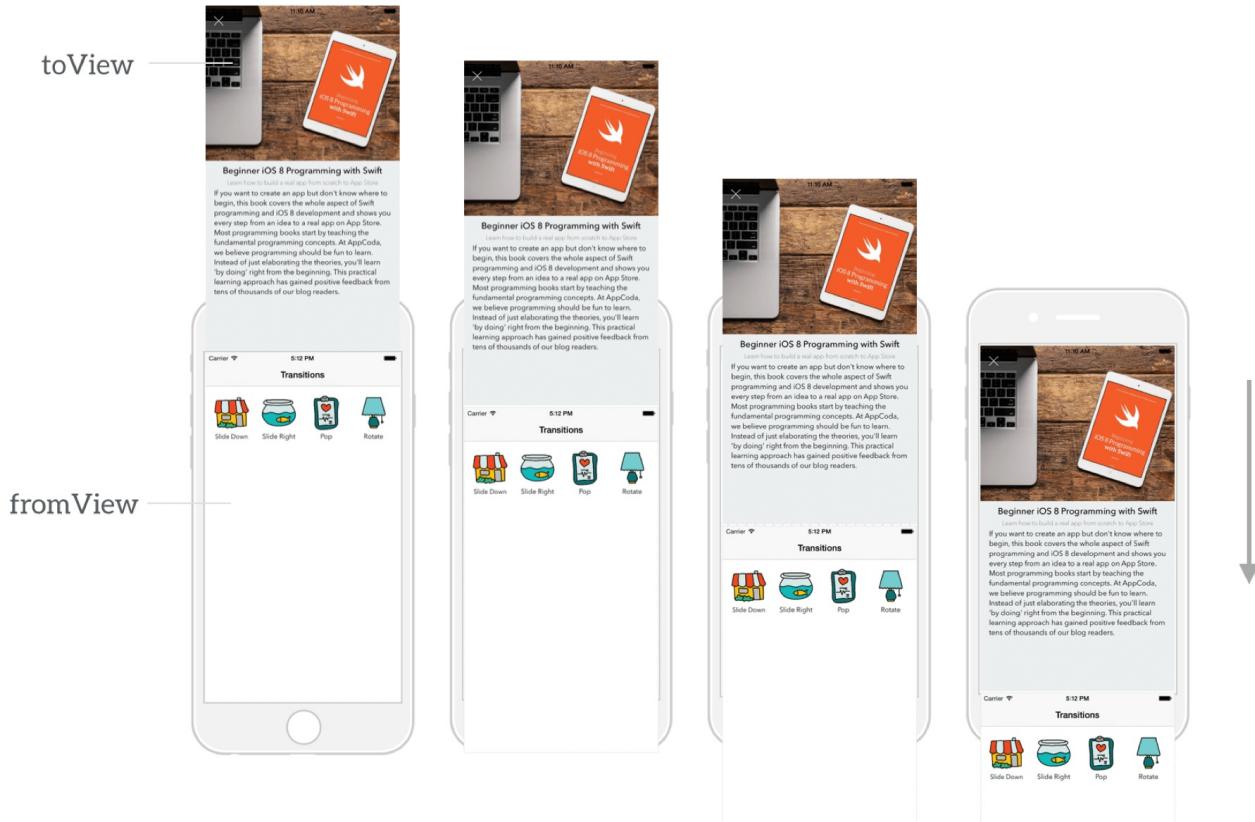
The `animationControllerForDismissedController` method, on the other hand, indicates the animator object to use when dismissing the view controller. In the above code, we simply return the current animator object.

Okay, let's move onto the implementation of `UIViewControllerAnimatedTransitioning` protocol, which provides the actual animation for the transition. When adopting the protocol, you have to provide the implementation of the following required methods:

- `transitionDuration(_:)`
- `animateTransition(_ transitionContext: UIViewControllerContextTransitioning)`

The first method is simple. You just return the duration (in seconds) of the transition animation. The second method is where the transition animations take place. When presenting or dismissing a view controller, UIKit calls the `animateTransition` method to perform the animations.

Before we dive into the code, let me explain how our own version of slide-down animation works. Take a look at the illustration below.



When a user taps the Slide Down icon, the current view controller begins to slide down off the screen. The detail view controller will also slide down from the top of the screen. When the animation ends, the detail view controller completely replaces the current view controller.

Okay, how can we implement an animation like that in code? First, insert the following code snippet in the `slideDownTransitionAnimator`. I will go through with you line by line later.

```

let duration = 0.5

func transitionDuration(transitionContext: UIViewControllerContextTransitioning?) -> NSTimeInterval {
    return duration
}

func animateTransition(transitionContext: UIViewControllerContextTransitioning) {
    // Get reference to our fromView, toView and the container view
    let fromView =
    transitionContext.viewForKey(UIViewTransitionContextFromViewKey)!
    let toView = transitionContext.viewForKey(UIViewTransitionContextToViewKey)!

    // Set up the transform we'll use in the animation
    guard let container = transitionContext.containerView() else {
        return
    }
}
```

```

    }

    let offScreenUp = CGAffineTransformMakeTranslation(0, -  

container.frame.height)  

    let offScreenDown = CGAffineTransformMakeTranslation(0,  

container.frame.height)

    // Make the toView off screen  

    toView.transform = offScreenUp

    // Add both views to the container view  

    container.addSubview(fromView)  

    container.addSubview(toView)

    // Perform the animation  

    UIView.animateWithDuration(duration, delay: 0.0, usingSpringWithDamping:  

0.8, initialSpringVelocity: 0.8, options: [], animations: {

        fromView.transform = offScreenDown  

        fromView.alpha = 0.5  

        toView.transform = CGAffineTransformIdentity

    }, completion: { finished in

        transitionContext.completeTransition(true)

    })
}

```

At the beginning, we set the transition duration to 0.5 seconds. The first method simply returns the duration.

Let's take a closer look at the `animateTransition` method. During the transition, there are two view controllers involved: the current view controller and the detail view controller. When UIKit calls the `animateTransition` method, it passes a context object (which adopts the `UIViewControllerContextTransitioning` protocol) containing information about the transition. From the context object, we can retrieve the view controllers involved in the transition using the `viewControllerForKey` method. The current view controller, which is the view controller that appears at the start of the transition, is referred to as the "from view controller". The detail view controller, which is going to replace the current view controller, is referred to as the "to view controller".

We then configure two transforms for moving the views. To implement the slide-down

animation, `toView` should be first moved off the screen. The `offScreenUp` variable is used for this purpose. The `offScreenDown` transform will later be used to move `fromView` off the screen during the transition. The context object also provides a container view that acts as the superview for the view involved in the transition. It is your responsibility to add both views to the container view using the `addSubview` method.

Lastly, we use the `animateWithDuration` method of `UIView` to perform a spring animation. In the animation block, we specify the changes of `fromView` and `toView`. By applying the `offScreenDown` transform to `fromView` to move the view off the screen, and restoring `toView` to the original position, this creates the slide-down animation.

Okay, we've created the animator object. The next step is to use this class to replace the standard transition. In the `ViewController.swift` file, declare the following variable to hold the animator object:

```
let slideDownTransition = SlideDownTransitionAnimator()
```

Next, implement the `prepareForSegue` method:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    let toViewController = segue.destinationViewController
    let sourceViewController = segue.sourceViewController as! ViewController

    if let selectedIndexPaths =
sourceViewController.collectionView.indexPathsForSelectedItems() {
        switch selectedIndexPaths[0].row {
        case 0: toViewController.transitioningDelegate = slideDownTransition
        default: break
    }
}
```

The app only performs the slide-down transition when the user taps the Slide Down icon, so we first verify whether the first cell is selected. When the cell is selected, we set our `SlideDownTransitionAnimator` object as the transitioning delegate.

Now compile and run the app. Tap on the Slide Down icon, you should get a nice slide-down transition to the detail view. However, the reverse transition doesn't work properly when you tap on the close button.



The resulting view, after transition, is dimmed. Obviously, the alpha value is not restored to the original value. And we expect the main view controller slides from the bottom of the screen instead of from the top.

Reversing the Transition

To reverse the transition, we just need to add some simple logic to the `slideDownTransitionAnimator` class to keep track of whether the app is presenting the view controller or dismissing it and we perform the animation accordingly.

First, declare the `isPresenting` variable in the class:

```
var isPresenting = false
```

This variable keeps track of whether we're presenting the view controller or dismissing one. Update the `animationControllerForDismissedController` and

`animationControllerForPresentedController` methods like this:

```
func animationControllerForPresentedController(presented: UIViewController,
presentingController presenting: UIViewController, sourceController source:
UIViewController) -> UIViewControllerAnimatedTransitioning? {

    isPresenting = true
    return self
}

func animationControllerForDismissedController(dismissed: UIViewController) ->
UIViewControllerAnimatedTransitioning? {

    isPresenting = false
    return self
}
```

We simply set `isPresenting` to `true` when the view controller is presented, and set it to `false` when the controller is dismissed. Next, update the `animateTransition` method as shown below:

```
func animateTransition(transitionContext: UIViewControllerContextTransitioning) {
    // Get reference to our fromView, toView and the container view
    let fromView =
transitionContext.viewForKey(UITransitionContextFromViewKey)!
    let toView = transitionContext.viewForKey(UITransitionContextToViewKey)!

    // Set up the transform we'll use in the animation
    guard let container = transitionContext.containerView() else {
        return
    }

    let offScreenUp = CGAffineTransformMakeTranslation(0, -
container.frame.height)
    let offScreenDown = CGAffineTransformMakeTranslation(0,
container.frame.height)

    // Make the toView off screen
    if isPresenting {
        toView.transform = offScreenUp
    }

    // Add both views to the container view
    container.addSubview(fromView)
    container.addSubview(toView)
```

```
// Perform the animation
UIView.animateWithDuration(duration, delay: 0.0, usingSpringWithDamping:
0.8, initialSpringVelocity: 0.8, options: [], animations: {

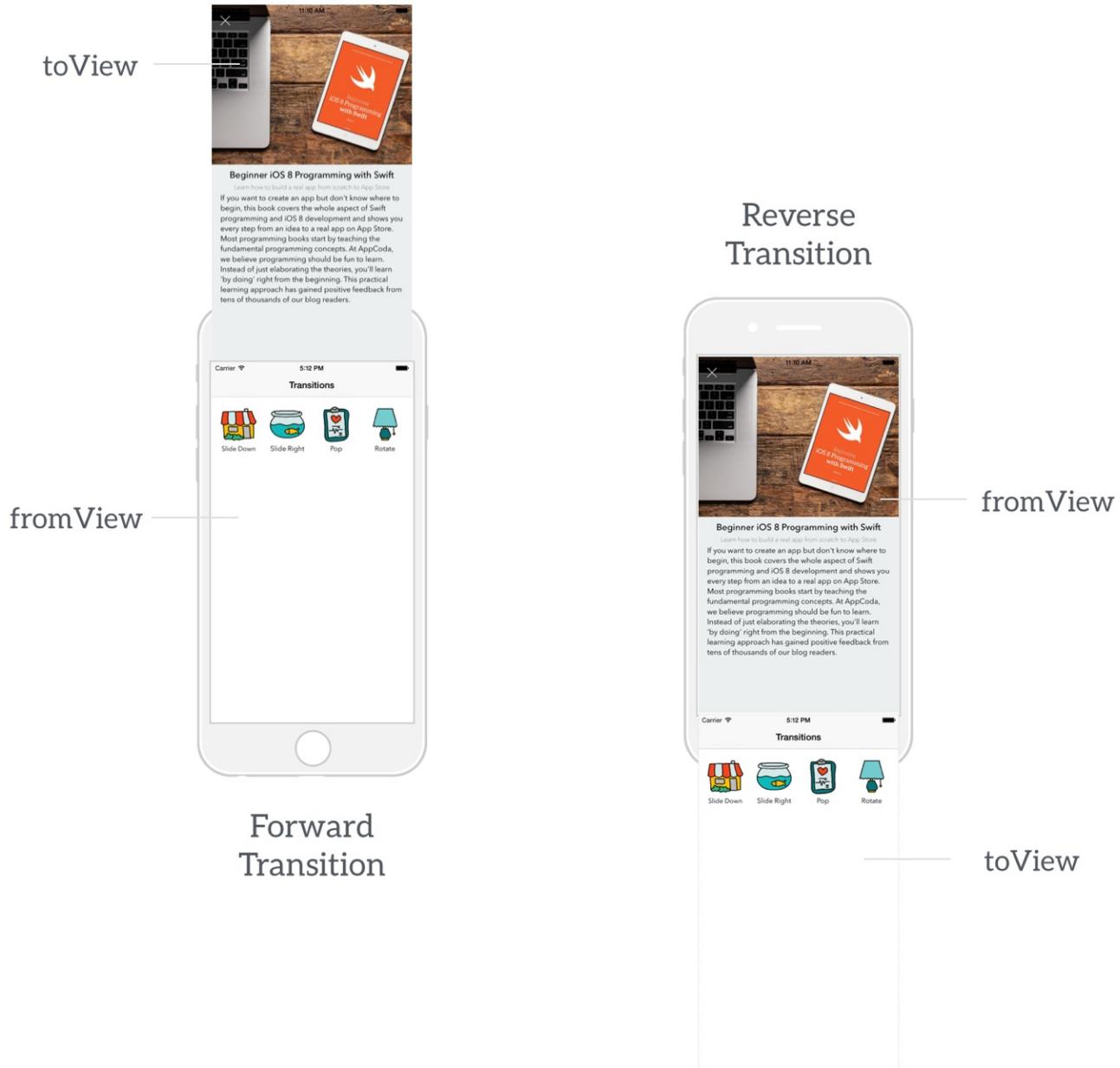
    if self.isPresenting {
        fromView.transform = offScreenDown
        fromView.alpha = 0.5
        toView.transform = CGAffineTransformIdentity
    } else {
        fromView.transform = offScreenUp
        toView.alpha = 1.0
        toView.transform = CGAffineTransformIdentity
    }

}, completion: { finished in

    transitionContext.completeTransition(true)

})
}
```

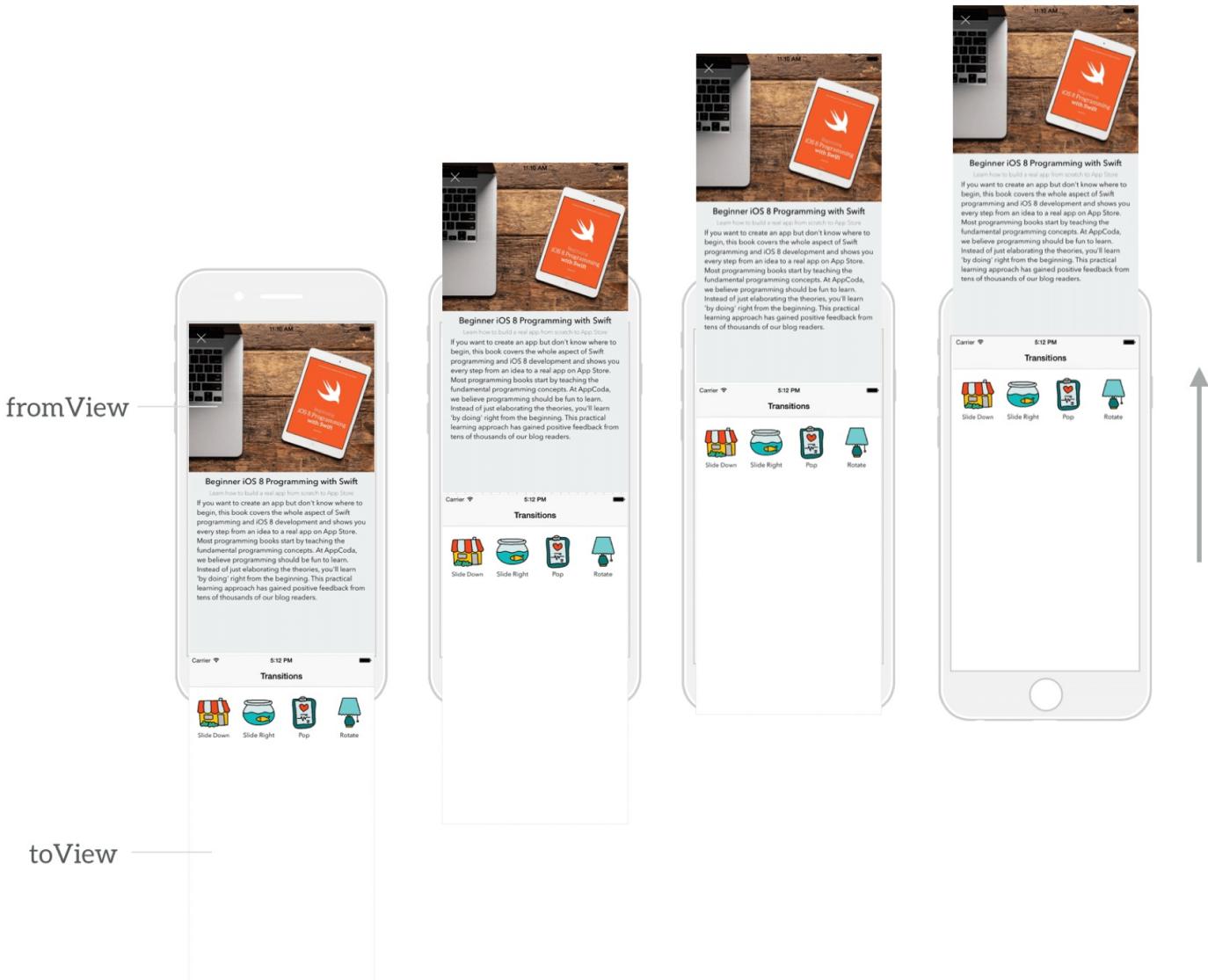
For reverse transition, `fromView` and `toView` are reversed. In other words, the detail view is now `fromView`, while the main view becomes `toView`.



We only want to make `toview` (i.e. detail view) off the screen in forward transition. So the `offScreenUp` transform is applied when the `isPresenting` variable is set to `true`.

In the animation block, the code is unchanged when `isPresenting` is set to `true`. But for reverse transition, we perform a different animation. We move the detail view (i.e. `fromView`) off the screen by applying the `offScreenUp` transform. For `toView`, it is restored to the original position and its `alpha` value is reset to `1.0`.

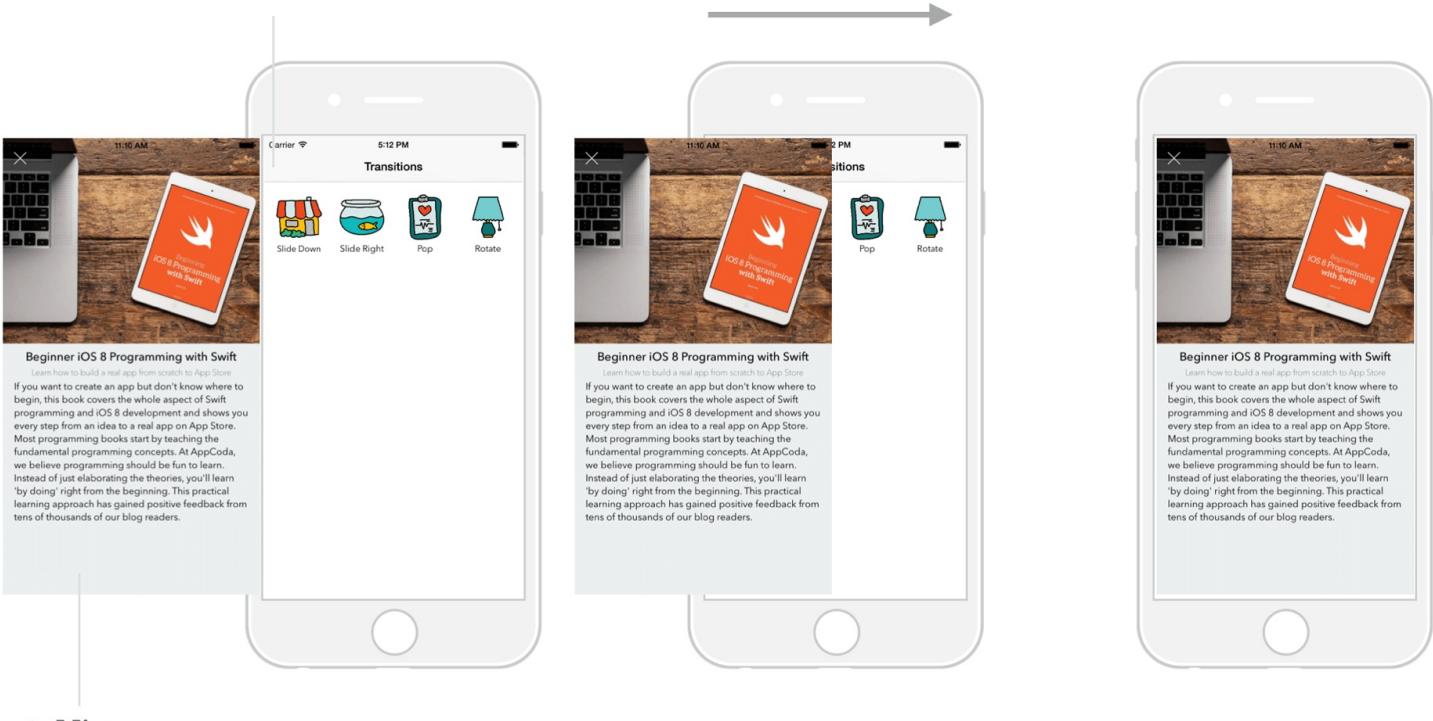
Now run the app again. When you close the detail view, the animation should work like this.



Creating the Slide Right Transition Animator

If you understand the material I've covered so far, it is pretty straightforward for you to create the slide right transition animator. The slide right animation will work like this:

fromView



toView

The detail view controller is first moved off the screen to the left. When the user taps on the Slide Right icon, the detail view slides into the screen to replace the main view. This time we keep the main view intact.

Okay, let's go into the implementation. In the project navigator, create a new class named `SlideRightTransitionAnimator` and set it as a subclass of `NSObject`. Update the `SlideRightTransitionAnimator` class to the following:

```
class SlideRightTransitionAnimator: NSObject,
UIViewControllerAnimatedTransitioning, UIViewControllerTransitioningDelegate {

    let duration = 0.5
    var isPresenting = false

    func transitionDuration(transitionContext:
    UIViewControllerContextTransitioning?) -> NSTimeInterval {
        return duration
    }

    func animateTransition(transitionContext:
    UIViewControllerContextTransitioning) {
        // Get reference to our fromView, toView and the container view
        let fromView =

```

```

transitionContext.viewForKey(UIKitTransitionContextFromViewKey)!

    let toView =
transitionContext.viewForKey(UIKitTransitionContextToViewKey)!

    // Set up the transform we'll use in the animation
    guard let container = transitionContext.containerView() else {
        return
    }

    let offScreenLeft = CGAffineTransformMakeTranslation(
        -container.frame.width, 0)

    // Make the toView off screen
    if isPresenting {
        toView.transform = offScreenLeft
    }

    // Add both views to the container view
    if isPresenting {
        container.addSubview(fromView)
        container.addSubview(toView)
    } else {
        container.addSubview(toView)
        container.addSubview(fromView)
    }

    // Perform the animation
    UIView.animateWithDuration(duration, delay: 0.0,
usingSpringWithDamping: 0.8, initialSpringVelocity: 0.8, options: [],
animations: {

    if self.isPresenting {
        toView.transform = CGAffineTransformIdentity
    } else {
        fromView.transform = offScreenLeft
    }

}, completion: { finished in

    transitionContext.completeTransition(true)

})
}

func animationControllerForPresentedController(presented: UIViewController,
presentingController presenting: UIViewController, sourceController source:
UIViewController) -> UIViewControllerAnimatedTransitioning? {

    isPresenting = true
}

```

```

        return self
    }

    func animationControllerForDismissedController(dismissed: UIViewController)
-> UIViewControllerAnimatedTransitioning? {
    isPresenting = false
    return self
}

}

```

The code is very similar to the one we developed previously. The changes are highlighted in yellow.

First, we move the detail view (i.e. `toview`) off the screen to the left by applying the `offScreenLeft` transform. If the `isPresenting` variable is set to `true`, `toview` should be placed on top of `fromView`. This is why we first add `fromView` to the container view, followed by `toview`. Conversely, for reverse transition, the detail view (i.e. `fromView`) should be placed above the main view before the transition begins.

For the animation block, the code is simple. When presenting the detail view (i.e. `toview`), we set its `transform` property to `CGAffineTransformIdentity` in order to move the view to the original position. When dismissing the detail view (i.e. `fromView`), we move it off screen again.

Before testing the animation, there is still one thing left. We need to hook up this animator to the transition delegate. Open the `ViewController.swift` file and declare the `slideRightTransition` variable:

```
let slideRightTransition = SlideRightTransitionAnimator()
```

Change the `prepareSegue` method by updating the `switch` statement like this:

```

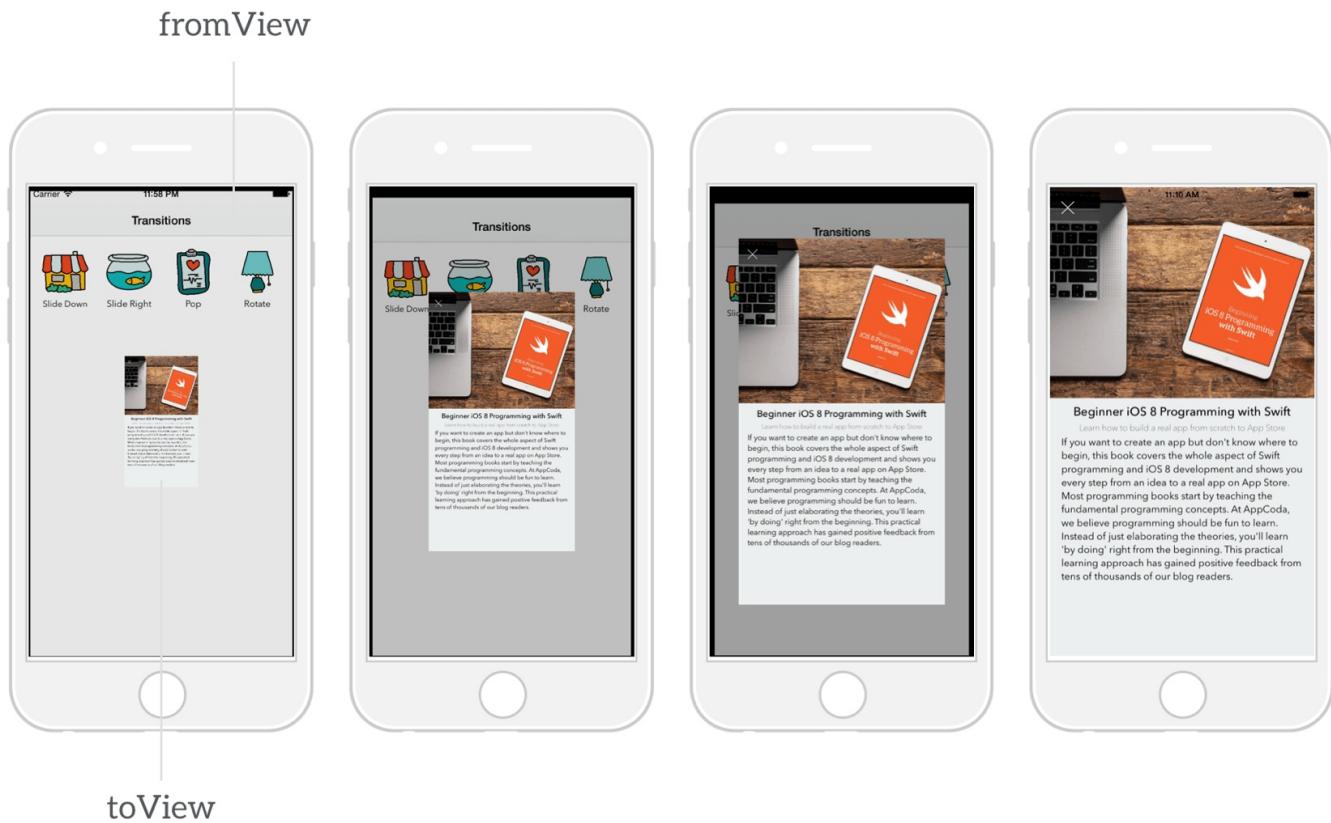
switch selectedIndexPaths[0].row {
case 0: toViewController.transitioningDelegate = slideDownTransition
case 1: toViewController.transitioningDelegate = slideRightTransition
default: break
}

```

Now when you run the project, you should see a slide right transition when tapping the Slide Right icon.

Creating a Pop Transition Animator

Let's move onto the pop transition. The pop animation is illustrated as below. When the user taps the Pop icon, the detail view will pop up from the screen. At the same time, the main view will change to a smaller size.



Similar to the slide animation, to implement the pop animation, the detail view (i.e. `toview`) is first minimized. Once a user taps the Pop icon, the detail view grows in size till it is restored to its original size.

Now create a new class and name it `PopTransitionAnimator`. Make sure you set it as a subclass of `NSObject`. Implement the class like this:

```
class PopTransitionAnimator: NSObject, UIViewControllerAnimatedTransitioning, UIViewControllerTransitioningDelegate {

    let duration = 0.5
    var isPresenting = false

    func transitionDuration(transitionContext: UIViewControllerContextTransitioning?) -> NSTimeInterval {
```

```

        return duration
    }

    func animateTransition(transitionContext:
UIViewControllerAnimatedTransitioning) {
    // Get reference to our fromView, toView and the container view
    let fromView =
transitionContext.viewForKey(UIViewTransitionFromViewKey)!
    let toView =
transitionContext.viewForKey(UIViewTransitionToViewKey)!

    // Set up the transform we'll use in the animation
    guard let container = transitionContext.containerView() else {
        return
    }

    let minimize = CGAffineTransformMakeScale(0, 0)
    let offScreenDown = CGAffineTransformMakeTranslation(0,
container.frame.height)
    let shiftDown = CGAffineTransformMakeTranslation(0, 15)
    let scaleDown = CGAffineTransformScale(shiftDown, 0.95, 0.95)

    // Change the initial size of the toView
    toView.transform = minimize

    // Add both views to the container view
    if isPresenting {
        container.addSubview(fromView)
        container.addSubview(toView)
    } else {
        container.addSubview(toView)
        container.addSubview(fromView)
    }

    // Perform the animation
    UIView.animateWithDuration(duration, delay: 0.0,
usingSpringWithDamping: 0.8, initialSpringVelocity: 0.8, options: [],
animations: {

        if self.isPresenting {
            fromView.transform = scaleDown
            fromView.alpha = 0.5
            toView.transform = CGAffineTransformIdentity
        } else {
            fromView.transform = offScreenDown
            toView.alpha = 1.0
            toView.transform = CGAffineTransformIdentity
        }
    })
}

```

```

    }, completion: { finished in
        transitionContext.completeTransition(true)
    })
}

func animationControllerForPresentedController(presented: UIViewController,
presentingController presenting: UIViewController, sourceController source:
UIViewController) -> UIViewControllerAnimatedTransitioning? {
    isPresenting = true
    return self
}

func animationControllerForDismissedController(dismissed: UIViewController)
-> UIViewControllerAnimatedTransitioning? {
    isPresenting = false
    return self
}
}

```

Again I will not walk you through the code line by line because you should now have a better understanding of the view controller transition. The logic is very similar to that of the previous two examples. Here we just define a different set of transforms. For example, we use `CGAffineTransformMakeScale` function to minimize the detail view.

In the animation block, when presenting the detail view, the main view (i.e. `fromView`) is shifted down a little bit and reduced in size. In the case of dismissing the detail view, we simply move the detail view off the screen.

Now go to the `viewController.swift` file and declare the `popTransitionAnimator` variable:

```
let popTransition = PopTransitionAnimator()
```

Update the `switch` block like this to configure the transitioning delegate:

```

switch selectedIndexPaths[0].row {
case 0: toViewController.transitioningDelegate = slideDownTransition
case 1: toViewController.transitioningDelegate = slideRightTransition
case 2: toViewController.transitioningDelegate = popTransition
default: break
}

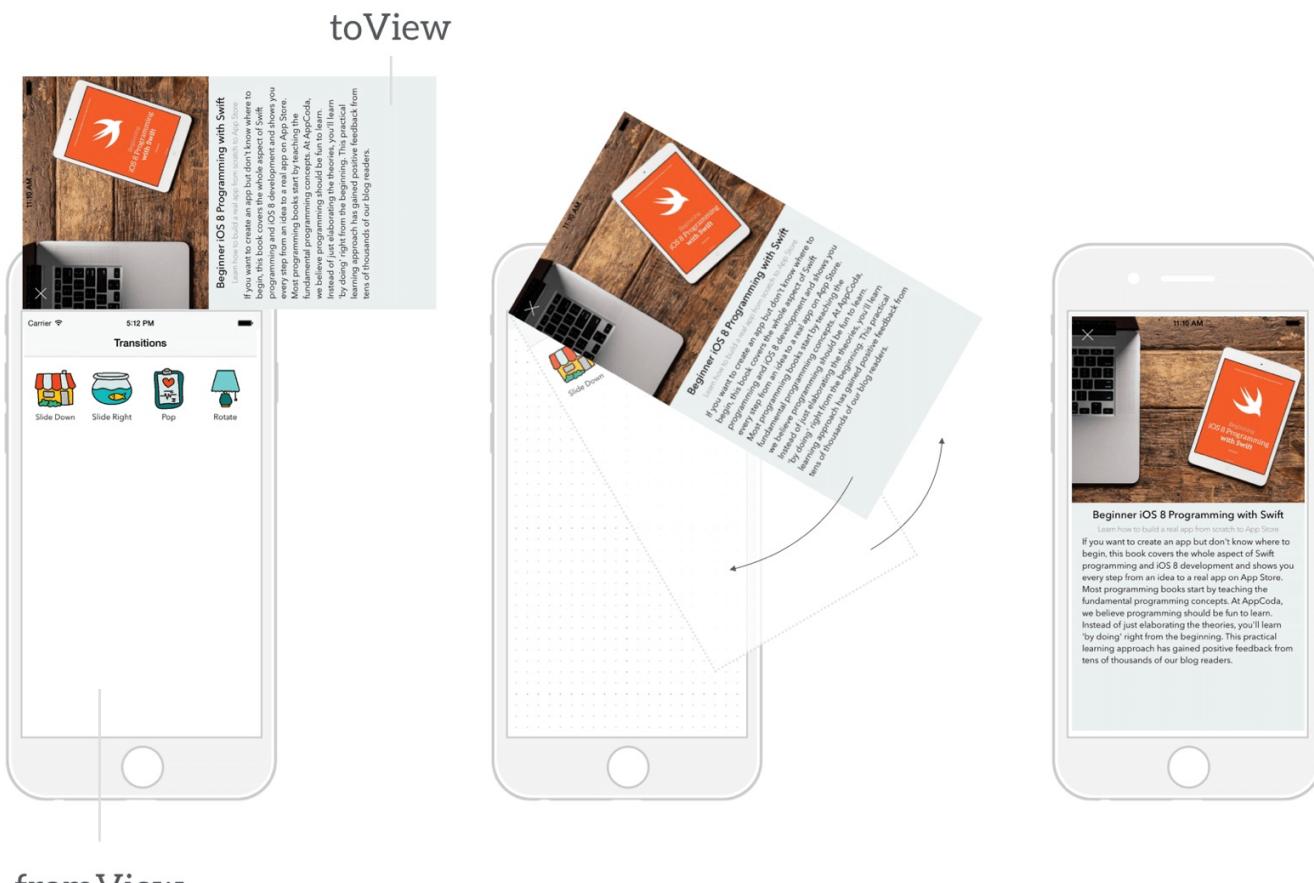
```

{}

Now hit the Run button to test out the transition. When you tap the Pop icon, you will get a nice pop animation.

Creating a Rotation Transition Animator

Now that we have created three custom transitions, we come to the last animation, which is a bit more complicated than the previous one. While I name the animation Rotation Transition, the effect actually works like the example below.



The detail view is initially turned sideways and moved off the screen. When the transition begins, the detail view swings back to the original position, while the main view rotates counterclockwise and swings off the screen. Okay, let's first create a new class called `RotateTransitionAnimator`. Set its subclass to `NSObject` and save the file. Implement the class like this:

```
class RotateTransitionAnimator: NSObject,
```

```

UIViewControllerAnimatedTransitioning, UIViewControllerTransitioningDelegate {

    let duration = 0.5
    var isPresenting = false

    func transitionDuration(transitionContext:
        UIViewControllerContextTransitioning?) -> NSTimeInterval {
        return duration
    }

    func animateTransition(transitionContext:
        UIViewControllerContextTransitioning) {
        // Get reference to our fromView, toView and the container view
        let fromView =
            transitionContext.viewForKey( UITransitionContextFromViewKey )!
        let toView =
            transitionContext.viewForKey( UITransitionContextToViewKey )!

        // Set up the transform we'll use in the animation
        guard let container = transitionContext.containerView() else {
            return
        }

        // Set up the transform for rotation
        // The angle is in radian. To convert from degree to radian, use this
formula
        // radian = angle * pi / 180
        let rotateOut = CGAffineTransformMakeRotation(-90 * CGFloat(M_PI) /
180)

        // Change the anchor point and position
        toView.layer.anchorPoint = CGPointMake(x:0, y:0)
        fromView.layer.anchorPoint = CGPointMake(x:0, y:0)
        toView.layer.position = CGPointMake(x:0, y:0)
        fromView.layer.position = CGPointMake(x:0, y:0)

        // Change the initial position of the toView
        toView.transform = rotateOut

        // Add both views to the container view
        container.addSubview(toView)
        container.addSubview(fromView)

        // Perform the animation
        UIView.animateWithDuration(duration, delay: 0.0,
usingSpringWithDamping: 0.8, initialSpringVelocity: 0.8, options: [],
animations: {

            if self.isPresenting {

```

```

        fromView.transform = rotateOut
        fromView.alpha = 0
        toView.transform = CGAffineTransformIdentity
        toView.alpha = 1.0
    } else {
        fromView.alpha = 0
        fromView.transform = rotateOut
        toView.alpha = 1.0
        toView.transform = CGAffineTransformIdentity
    }

}, completion: { finished in

    transitionContext.completeTransition(true)

})

}

func animationControllerForPresentedController(presented: UIViewController,
presentingController presenting: UIViewController, sourceController source:
UIViewController) -> UIViewControllerAnimatedTransitioning? {

    isPresenting = true
    return self
}

func animationControllerForDismissedController(dismissed: UIViewController)
-> UIViewControllerAnimatedTransitioning? {

    isPresenting = false
    return self
}

}

```

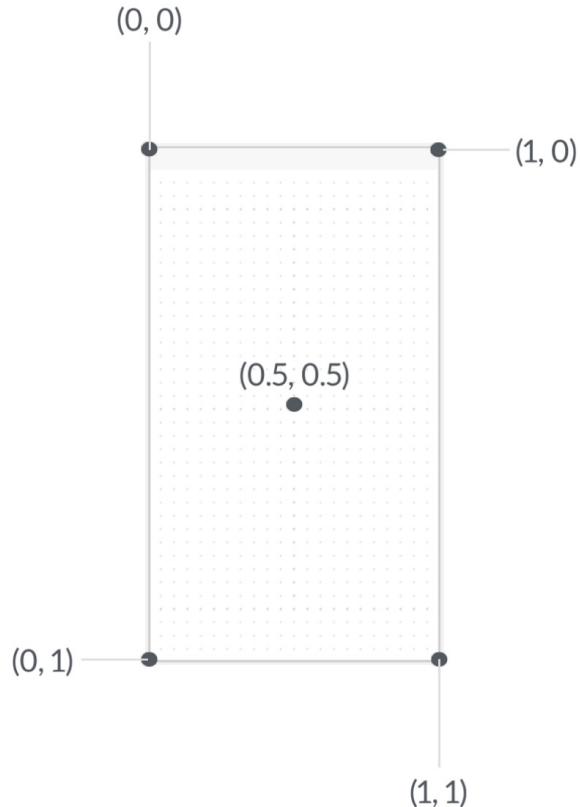
Let's discuss the first code snippet highlighted in yellow. To build the animation, the first thing that comes to your mind is to create a rotation transform using the `CGAffineTransformMakeRotation` function. You provide the angle of rotation in radian, for which a positive value indicates a clockwise direction while a negative value specifies a counter clockwise rotation. Here is an example:

```
let rotateOut = CGAffineTransformMakeRotation(-90 * CGFloat(M_PI) / 180)
```

If you apply the above transform to the detail view, you will rotate the view by 90 degrees counter clockwise. However, the rotation happens around the center of the screen. Obviously,

to perform our expected animation, the detail view should be rotated around the top-left corner of the screen.

By default, the anchor point of a view's layer (`CALayer` class) is set to the center. You specify the value for this property using the unit coordinate space.



To change the anchor point to the top left corner of the layer, we set it to (0, 0) for both `fromView` and `toView`.

```
toView.layer.anchorPoint = CGPointMake(x:0, y:0)  
fromView.layer.anchorPoint = CGPointMake(x:0, y:0)
```

But why do we need to change the layer's position in addition to the anchor point? The layer's position is set to the center of the view. For instance, if you are using iPhone 5, the position of the layer is set to (160, 284). Without altering the position, you will end up with an animation like this:



Since the layer's anchor point was changed to `(0, 0)` and the position is unchanged, the layer moves so that its new anchor point is at the unchanged position. This is why we have to change the position of both `fromView` and `toView` to `(0, 0)`.

For the animation block, we simply apply the rotation transform to `fromView` and `toView` accordingly. When presenting the detail view (i.e. `toView`), we restore it to the original position and rotate the main view off the screen. We do the reverse when dismissing the detail view.

Go to the `ViewController.swift` file and declare a variable for the `RotateTransitionAnimator` object:

```
let rotateTransition = RotateTransitionAnimator()
```

Lastly, update the switch block to hook up the `RotateTransitionAnimator` object:

```
switch selectedIndexPaths[0].row {
case 0: toViewController.transitioningDelegate = slideDownTransition
case 1: toViewController.transitioningDelegate = slideRightTransition
case 2: toViewController.transitioningDelegate = popTransition
case 3: toViewController.transitioningDelegate = rotateTransition
```

```
default: break  
}
```

Now compile and run the project again. Tap the Rotate icon, and you will get an interesting transition.

In this chapter, I showed you the basics of custom view controller transitions. Now it is time to create your own animation in your apps. Good design is much more than visuals. Your app has to feel right. By implementing proper and engaging view controller transitions, you will take your app to the next level.

For reference, you can download the final project from

<https://www.dropbox.com/s/8aiyts5yayxzdr/NavTransition.zip?dl=0>.

Chapter 24

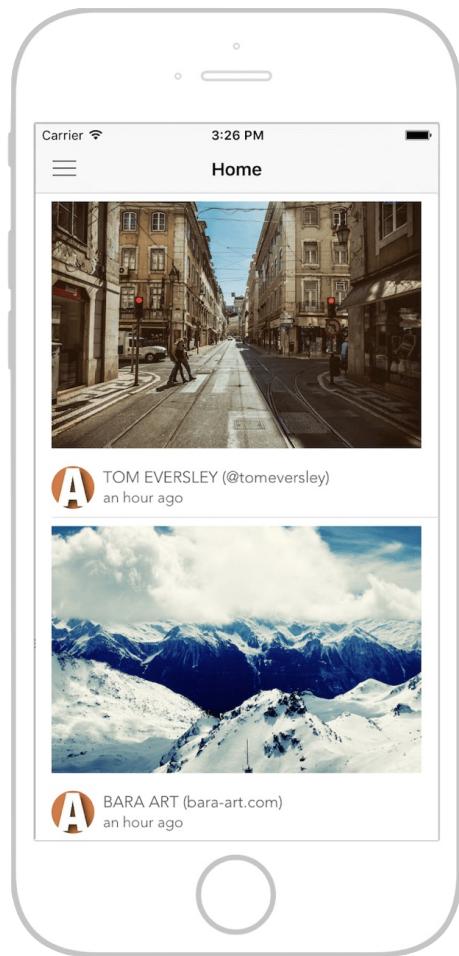
Building a Slide Down Menu



Navigation is an important part of every user interface. There are multiple ways to present a menu for your users to access the app's features. The sidebar menu that we discussed earlier is an example. Slide down menu is another common menu design. When a user taps the menu button, the main screen slides down to reveal the menu. The screen below shows a sample slide down menu used in the older version of the Medium app.

If you have gone through the previous chapter, you should have a basic understanding of custom view controller transition. In this chapter, you will apply what you have learned to build an animated slide down menu.

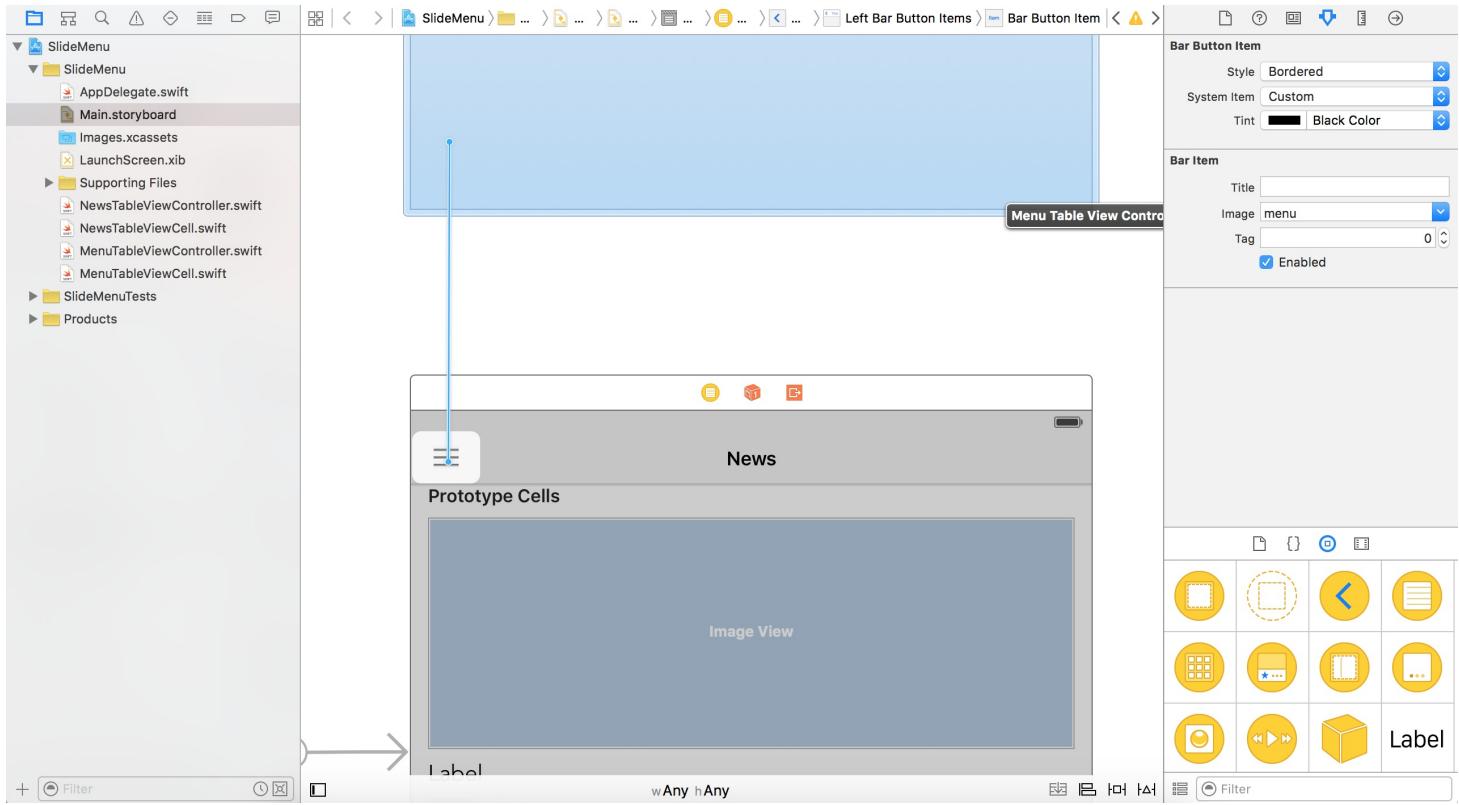
As usual, I don't want you to start from scratch. You can download the project template from <https://www.dropbox.com/s/cflrrp9wy6v37f4/SlideMenuStart.zip?dl=0>. It includes the storyboard and view controller classes. You will find two table view controllers. One is for the main screen (embedded in a navigation controller) and the other is for the navigation menu. If you run the project, the app should present you the main interface with some dummy data.



Before moving on, take a few minutes to browse through the code template to familiarize yourself with the project.

Presenting the Menu Modally

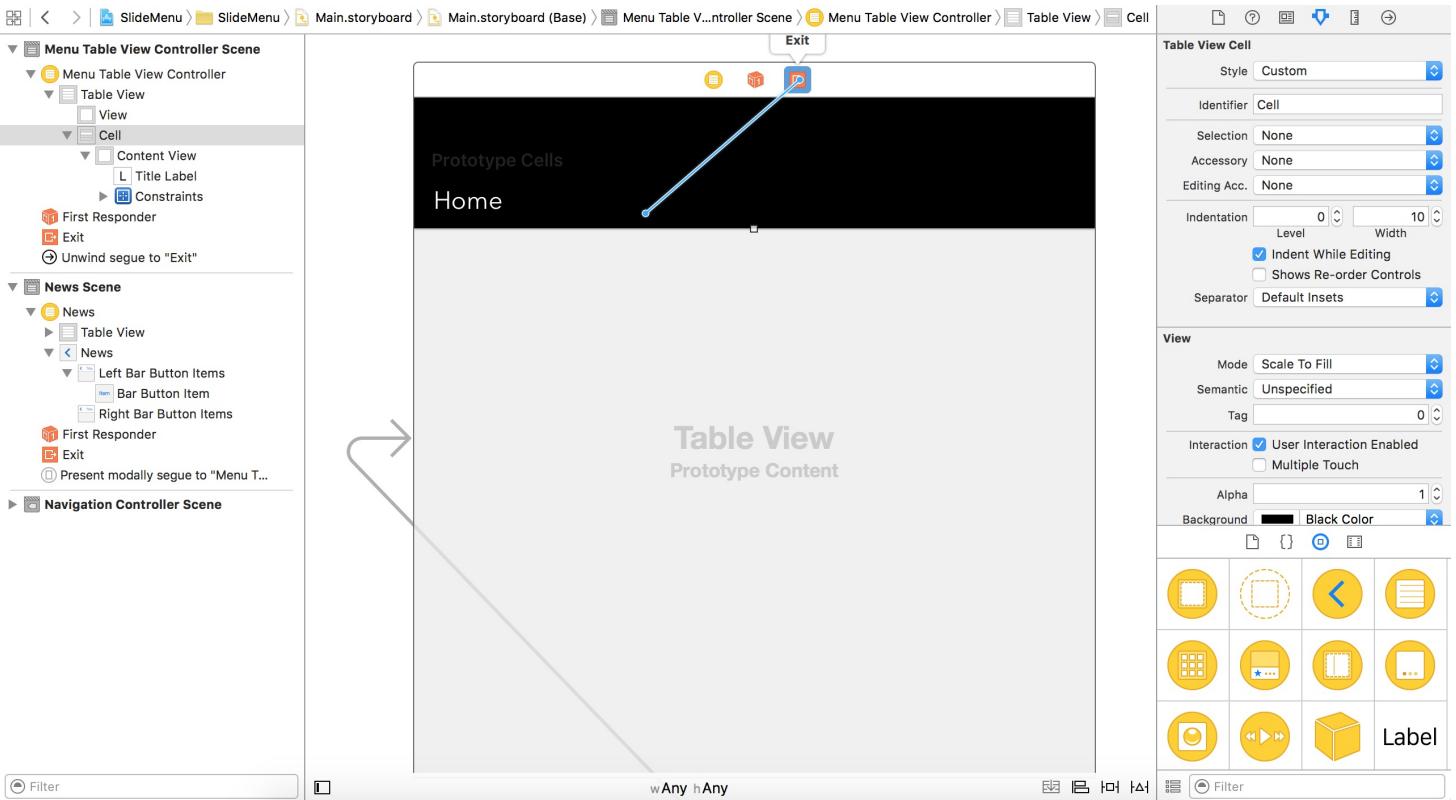
Okay, let's get started. First, open the `Main.storyboard` file. You should find two table view controllers, which are not connected with any segue yet. In order to bring up the menu when a user taps the menu button, control-drag from the menu button to the menu table view controller. Release the buttons and select `present modally` under action segue.



If you run the project now, the menu will be presented as a modal view. In order to dismiss the menu, we will add an unwind segue. Open the `NewsTableViewCell.swift` file and insert an unwind action method:

```
@IBAction func unwindToHome(segue: UIStoryboardSegue) {
    let sourceController = segue.sourceViewController as!
MenuTableViewCell
    self.title = sourceController.currentItem
}
```

Next, go to the storyboard. Control-drag from the prototype cell of the Menu table view controller to the exit icon. When prompted, select the `unwindToHome:` option under selection segue.



Now when a user taps any menu item, the menu controller will dismiss to reveal the main screen. Through the `unwindToHome:` action method, the main view controller (i.e. `NewsTableViewController`) retrieves the menu item selected by the user and changes the title of the navigation bar. To keep things simple, we just change the title of the navigation bar and will not alter the content of the main screen.

However, the app can't change the title yet. The reason is that the `currentItem` variable of the `MenuTableViewController` object is not updated properly. To make it work, there are a couple of methods we need to implement.

Insert the following method in the `MenuTableViewController` class:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    let menuTableViewController = segue.sourceViewController as!
    MenuTableViewController
    if let selectedIndexPath =
        menuTableViewController.tableView.indexPathForSelectedRow {
        currentItem = menuItems[selectedIndexPath.row]
    }
}
```

Here, we just update the `currentItem` variable to the selected menu item. Later the `NewsTableViewController` class can pick the value of `currentItem` to update the title of the navigation bar.

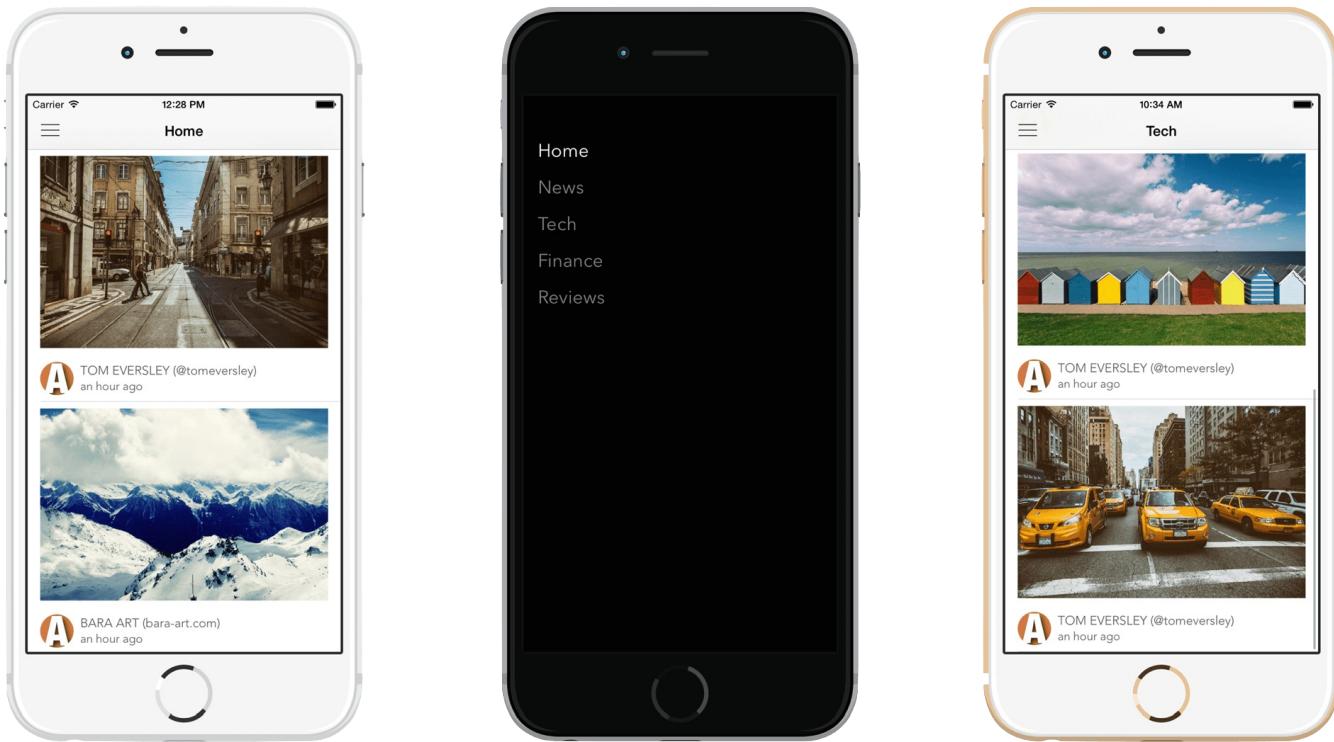
Now the app should be able to update the title of navigation bar. But there is still one thing left. For example, say you select *News* in the menu, the app then changes the title to *News*. If you tap the menu button again, the menu controller still highlights *Home* in white, instead of *News*.

Let's fix the issue. In the `NewsTableViewController.swift` file, insert the following method to pass the current title to the menu controller:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {  
    let menuTableViewController = segue.destinationViewController as!  
    MenuTableViewController  
    menuTableViewController.currentItem = self.title!  
}
```

When the menu button is tapped, the `prepareForSegue` method will be called before transitioning to the menu view controller. Here we just update the current item of the controller, so it can highlight the item in white.

Now compile and run the project. Tap the menu item and the app will present you the menu modally. When you select a menu item, the menu will dismiss and the navigation bar title will change accordingly.



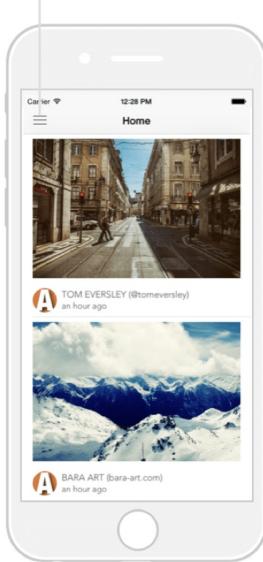
Creating the Animated Slide Down Menu

Now that the menu is presented using the standard animation, let's begin to create a custom transition. As I mentioned in the previous chapter, the core of a custom view controller animation is to create an animator object, that conforms to both

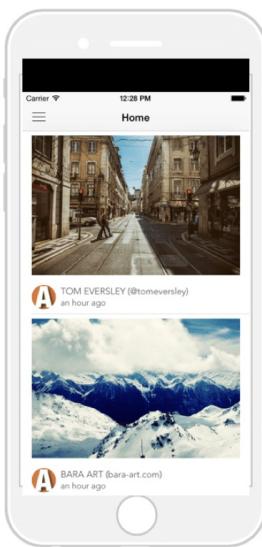
`UIViewControllerAnimatedTransitioning` and `UIViewControllerTransitioningDelegate` protocols. We are going to implement the class. But first, let's take a look at how the slide down menu works.

When a user taps the menu, the main view begins to slide down until it reaches the predefined location, which is 150 points away from the bottom of the screen. The below illustration should give you a better idea of the sliding menu.

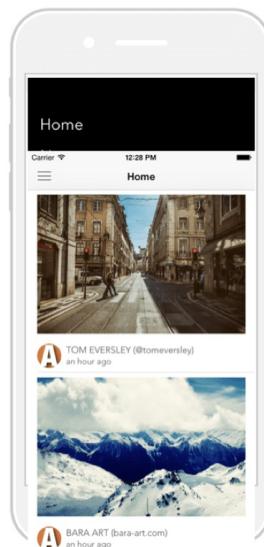
1
Tap the menu button



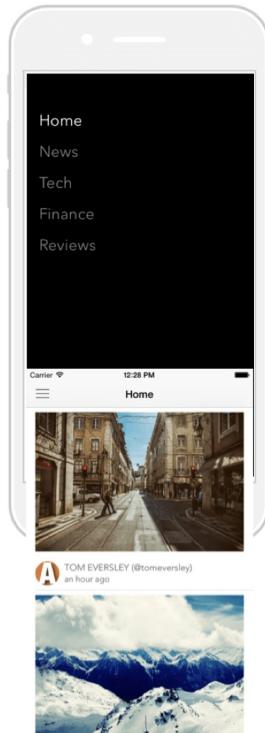
2
The menu moves up a little bit and the main view starts to move down



3
The menu returns to its original position and the main view continues to slide down



4
The main keeps sliding down until it reaches the preset position



Building the Slide Down Menu Animator

To create the slide down animation, we will create a slide down animator called `MenuTransitionManager`. In the project navigator, right click to create a new file. Name the class `MenuTransitionManager` and set it as a subclass of `NSObject`. Update the class like this:

```
class MenuTransitionManager: NSObject, UIViewControllerAnimatedTransitioning, UIViewControllerTransitioningDelegate {

    let duration = 0.5
    var isPresenting = false

    var snapshot:UIView?

    func transitionDuration(transitionContext: UIViewControllerContextTransitioning?) -> NSTimeInterval {
        return duration
    }

    func animateTransition(transitionContext:
```

```

UIViewControllerAnimatedTransitioning) {
    // Get reference to our fromView, toView and the container view
    let fromView =
transitionContext.viewForKey(UIViewContextFromViewKey)!
    let toView =
transitionContext.viewForKey(UIViewContextToViewKey)!

    // Set up the transform we'll use in the animation
    guard let container = transitionContext.containerView() else {
        return
    }

    let moveDown = CGAffineTransformMakeTranslation(0,
container.frame.height - 150)
    let moveUp = CGAffineTransformMakeTranslation(0, -50)

    // Add both views to the container view
    if isPresenting {
        toView.transform = moveUp
        snapshot = fromView.snapshotViewAfterScreenUpdates(true)
        container.addSubview(toView)
        container.addSubview(snapshot!)
    }

    // Perform the animation
    UIView.animateWithDuration(duration, delay: 0.0,
usingSpringWithDamping: 0.8, initialSpringVelocity: 0.8, options: [],
animations: {

        if self.isPresenting {
            self.snapshot?.transform = moveDown
            toView.transform = CGAffineTransformIdentity
        } else {
            self.snapshot?.transform = CGAffineTransformIdentity
            fromView.transform = moveUp
        }

    }, completion: { finished in

        transitionContext.completeTransition(true)

        if !self.isPresenting {
            self.snapshot?.removeFromSuperview()
        }
    })
}

func animationControllerForPresentedController(presented: UIViewController,
presentingController presenting: UIViewController, sourceController source:

```

```
UIViewController) -> UIViewControllerAnimatedTransitioning? {  
  
    isPresenting = true  
    return self  
}  
  
func animationControllerForDismissedController(dismissed: UIViewController)  
-> UIViewControllerAnimatedTransitioning? {  
  
    isPresenting = false  
    return self  
}  
  
}
```

The class implements both `UIViewControllerAnimatedTransitioning` and `UIViewControllerTransitioningDelegate` protocols. I will not go into the details of the methods, as they are explained in the previous chapter. Let's focus on the animation block (i.e. the `animateTransition` method).

Referring to the illustration displayed earlier, during the transition, the main view is the `fromView`, while the menu view is the `toView`.

To create the animations, we configure two transforms. The first transform (i.e. `moveDown`) is used to move down the main view. The other transform (i.e. `moveUp`) is configured to move up the menu view a bit so that it will also have a slide-down effect when restoring to its original position. You will understand what I mean when you run the project later.

From iOS 7 and onwards, you can use the *UIView-Snapshotting API* to quickly and easily create a light-weight snapshot of a view.

```
snapshot = fromView.snapshotViewAfterScreenUpdates(true)
```

By calling the `snapshotViewAfterScreenUpdates` method, you have a snapshot of the main view. With the snapshot, we can add it to the container view to perform the animation. Note that the snapshot is added on top of the menu view.

For the actual animation when presenting the menu, the implementation is really simple. We just apply the `moveDown` transform to the snapshot of the main view and restore the menu view to its default position.

```
self.snapshot?.transform = moveDown  
toView.transform = CGAffineTransformIdentity
```

When dismissing the menu, the reverse happens. The snapshot of the main view slides up and returns to its default position. Additionally, the snapshot is removed from its super view so that we can bring the actual main view back.

Now open `NewsTableViewController.swift` and declare a variable for the `MenuTransitionManager` object:

```
let menuTransitionManager = MenuTransitionManager()
```

In the `prepareForSegue` method, add a line of code to hook up the animation:

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {  
    let menuTableViewController = segue.destinationViewController as!  
    MenuTableViewController  
    menuTableViewController.currentItem = self.title!  
    menuTableViewController.transitioningDelegate = menuTransitionManager  
}
```

That's it! You can now compile and run the project. Tap the menu button and you will have a slide down menu.

Detecting Tap Gesture

For now, the only way to dismiss the menu is to select a menu item. From a user's perspective, tapping the snapshot should dismiss the menu too. However, the snapshot of the main view is non-responsive.

The snapshot is actually a `UIView` object. So we can create a `UITapGestureRecognizer` object and add it to the snapshot. When instantiating a `UITapGestureRecognizer` object, we need to pass it the target object that is the recipient of action messages sent by the receiver, and the action method to be called. Obviously, you can hardcode a particular object as the target object to dismiss the view, but to keep our design flexible, we will define a protocol and let the delegate object implement it.

In `MenuTransitionManager.swift`, define the following protocol:

```
@objc protocol MenuTransitionManagerDelegate {
    func dismiss()
}
```

Here we define a `MenuTransitionManagerDelegate` protocol with a required method called `dismiss()`. The beauty of a protocol is that you do not need to provide any implementation for the methods. Instead the implementation is left to the delegate that implements the protocol. In other words, the delegate should implement the `dismiss` method and provide the actual logic for dismissing the view.

Quick note: Here, the protocol must be exposed to the Objective-C runtime, as it will be accessed by `UITapGestureRecognizer`. This is why we prefix the protocol with the `@objc` attribute.

In the `MenuTransitionManager` class, declare a delegate variable for storing the delegate object:

```
var delegate:MenuTransitionManagerDelegate?
```

Later, the object which is responsible to handle the tap gesture should be set as the delegate object. Lastly, we need to create a `UITapGestureRecognizer` object and add it to the snapshot. A good way to do this is define a `didSet` method within the `snapshot` variable. Change the `snapshot` declaration to the following:

```
var snapshot:UIView? {
    didSet {
        if let delegate = delegate {
            let tapGestureRecognizer = UITapGestureRecognizer(target: delegate,
action: "dismiss")
            snapshot?.addGestureRecognizer(tapGestureRecognizer)
        }
    }
}
```

Property observer is one of the powerful features in Swift. The observer (`willSet` / `didSet`) is called every time a property's value is set. This provides us a convenient way to perform certain actions immediately before or after an assignment. The `willset` method is called right before the value is stored, while the `didset` method is called immediately after the assignment.

In the above code, we make use of the property observer to create a gesture recognizer and set it to the snapshot. So every time we assign the `snapshot` variable an object, it will immediately

configure with a tap gesture recognizer.

We are almost done. Now go back to `NewsTableViewController.swift`, which is the class to implement the `MenuTransitionManagerDelegate` protocol. First, change the class declaration to the following:

```
class NewsTableViewController: UITableViewController,  
MenuTransitionManagerDelegate
```

Next, implement the required method of the protocol:

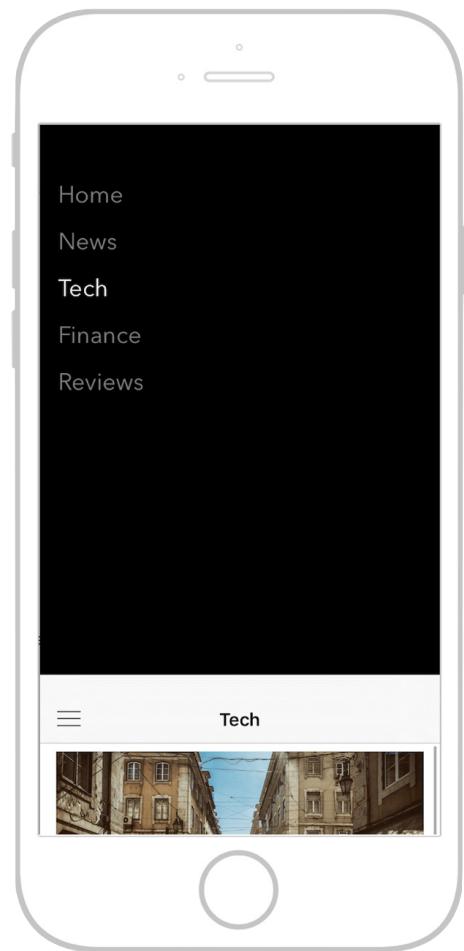
```
func dismiss() {  
    dismissViewControllerAnimated(true, completion: nil)  
}
```

Here, we simply dismiss the view controller by calling the `dismissViewControllerAnimated` method.

Lastly, insert a line of code in the `prepareForSegue` method of the `NewsTableViewController` class to set itself as the delegate object:

```
menuTransitionManager.delegate = self
```

Great! You're now ready to test the app again. Hit the Run button to try it out. You should be able to dismiss the menu by tapping the snapshot of the main view.

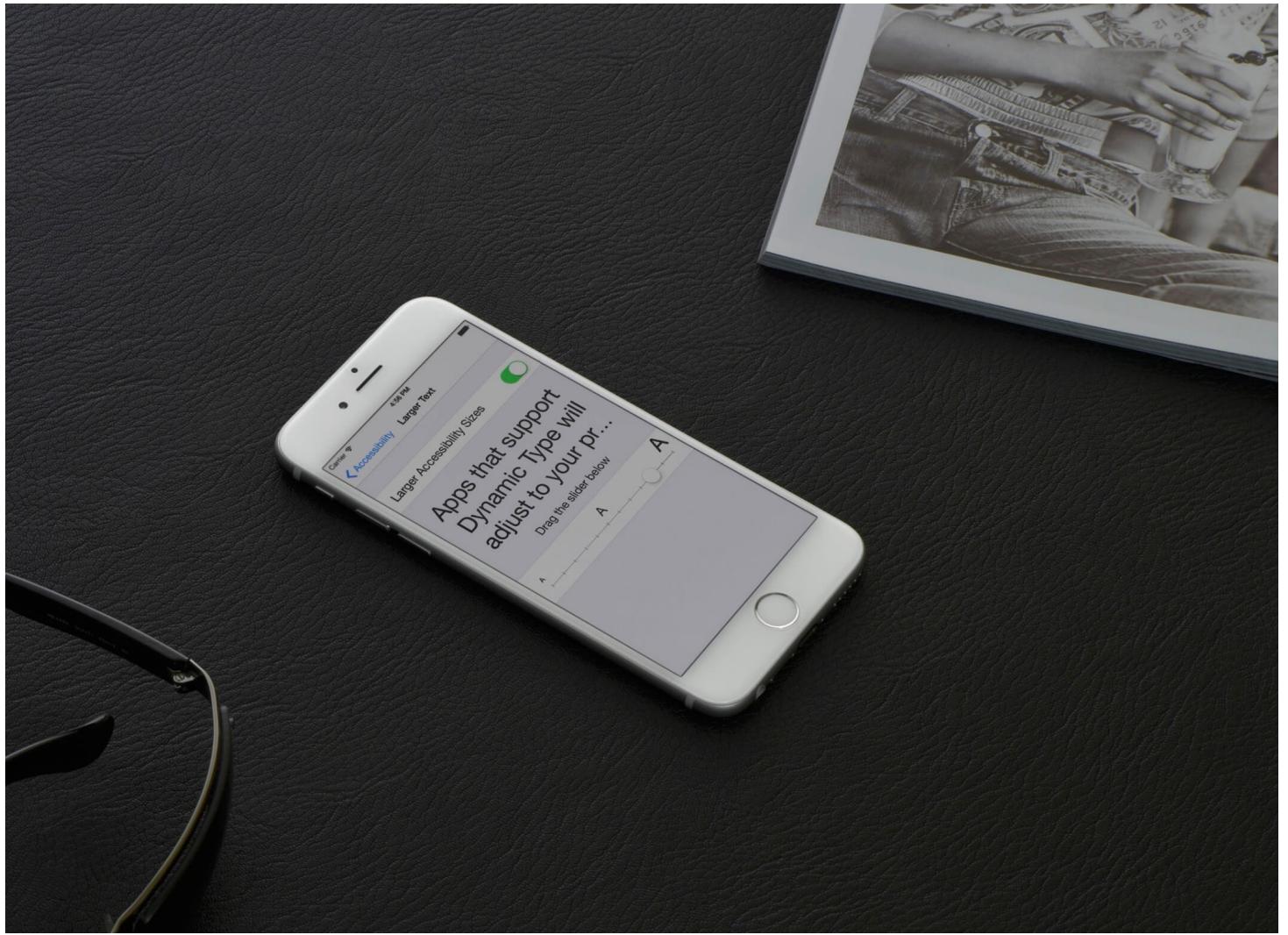


By applying custom view controller transitions properly, you can greatly improve the user experience and set your app apart from the crowd. The slide down menu is just an example, so try to create your own animation in your next app.

For reference, you can download the final project from
<https://www.dropbox.com/s/yxxuidy9veqiva8/SlideMenu.zip?dl=0>.

Chapter 25

Self Sizing Cells and Dynamic Type



In iOS 8, Apple introduced a new feature for `UITableView` known as *Self Sizing Cells*. To me, this is seriously one of the most exciting features for the SDK. Prior to iOS 8, if you wanted to display dynamic content in table view with variable height you would need to calculate the row height manually. Now with iOS 8 and iOS 9, self sizing cells provide a solution for displaying dynamic content. In brief, here is what you need to do when using self sizing cells:

- Define auto layout constraints for your prototype cell
- Specify the `estimatedRowHeight` property of your table view

- Set the `rowHeight` property of your table view to `UITableViewAutomaticDimension`

If we express the last point in code, it looks like this:

```
tableView.estimatedRowHeight = 95.0  
tableView.rowHeight = UITableViewAutomaticDimension
```

With just two lines of code, you instruct the table view to calculate the cell's size to match its content and render it dynamically. This self sizing cell feature should save you tons of code and time. You're going to love it.

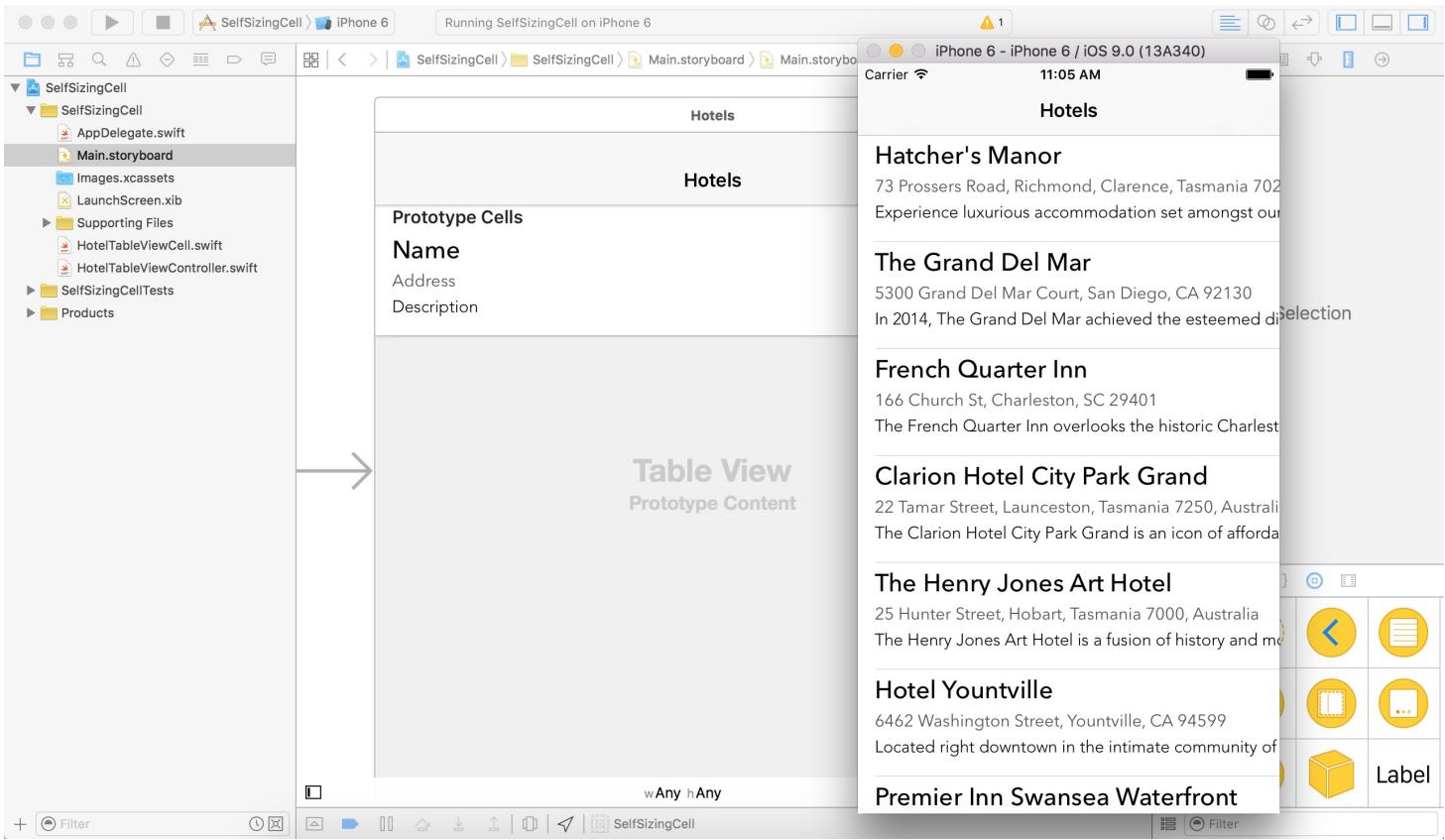
In the next section we'll develop a simple demo app to demonstrate self sizing cell. There is no better way to learn a new feature than to use it. In addition to self sizing cell, I will also talk about *Dynamic Type*. Dynamic Type was first introduced in iOS 7 - it allows users to customize the text size to fit their own needs. However, only apps that adopt Dynamic Type respond to the text change. As of right now, only a fraction of third-party apps have adopted the feature.

You're encouraged to adopt Dynamic Type so as to give your users the flexibility to change text sizes, and to improve the user experience for vision-challenged users. Therefore, in the later section you will learn how to adopt dynamic type in your apps.

Building a Simple Demo

We will start with a project template for a simple table-based app showing a list of hotels. The prototype cell contains three one-line text labels for the name, address and description of a hotel. If you download the project from

<https://www.dropbox.com/s/7itd2bt7kd31b6t/SelfSizingCellStart.zip?dl=0> and run it, you will have an app like the one shown below.



As you can see, some of the addresses and descriptions are truncated; you may have faced the same issue when developing table-based apps. To fix the issue, one option is to simply reduce the font size or increase the number of lines of a label. However, this solution is not perfect. As the length of the addresses and descriptions varies, it will probably result in an imperfect UI with redundant white spaces. A better solution is to adapt the cell size with respect to the size of its inner content. Prior to iOS 8, you would need to manually compute the size of each label and adjust the cell size accordingly, which would involve a lot of code and subsequently a lot of time.

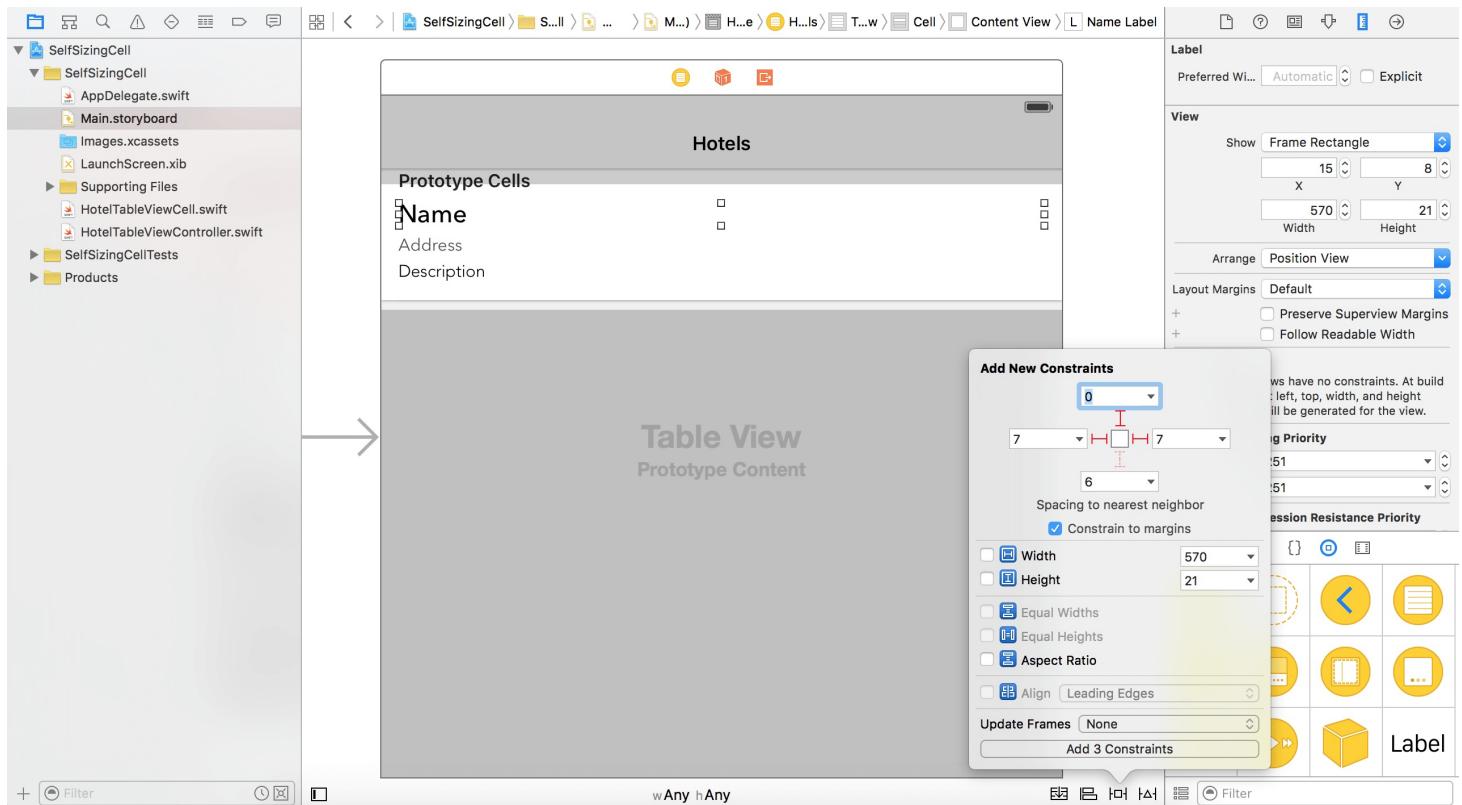
Starting from iOS 8, all you need to do is use self sizing cells and the cell size can be adapted automatically. Currently, the project template creates a prototype cell with a fixed height of 95 points. What we are going to do is turn the cells into self sizing cells so that the cell content can be displayed perfectly.

Adding Auto Layout Constraints

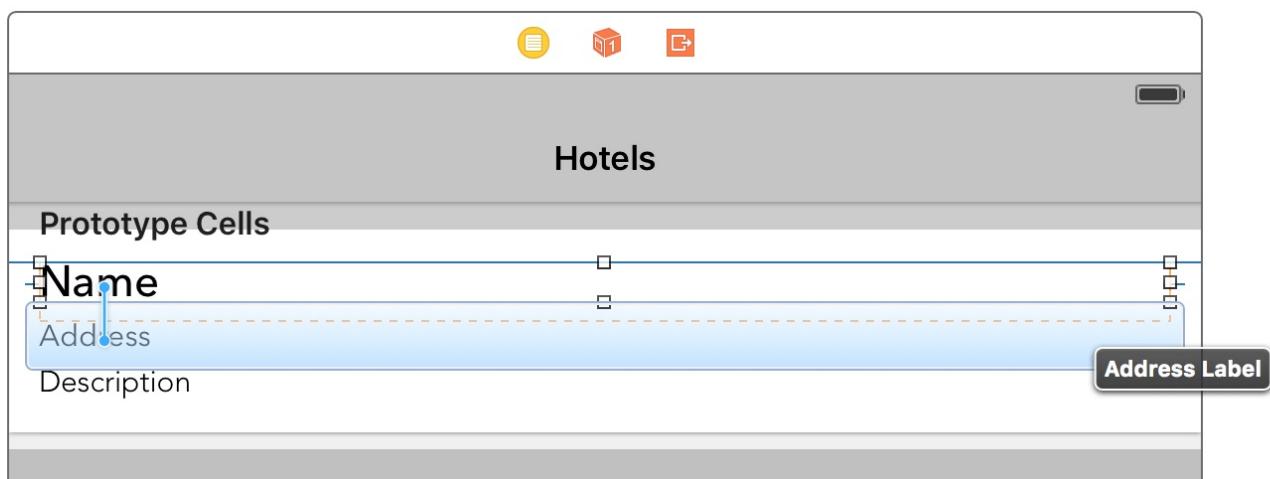
Many developers hate auto layout and avoid using it whenever possible. However, without auto layout self sizing cells will not work, because they rely on the constraints to determine the

proper row height. In fact, table view calls `systemLayoutSizeFittingSize` on your cell and that returns the size of the cell based on the layout constraints. If this is the first time you're working with auto layout, I recommend that you quickly review chapter 1 about adaptive UI before continuing.

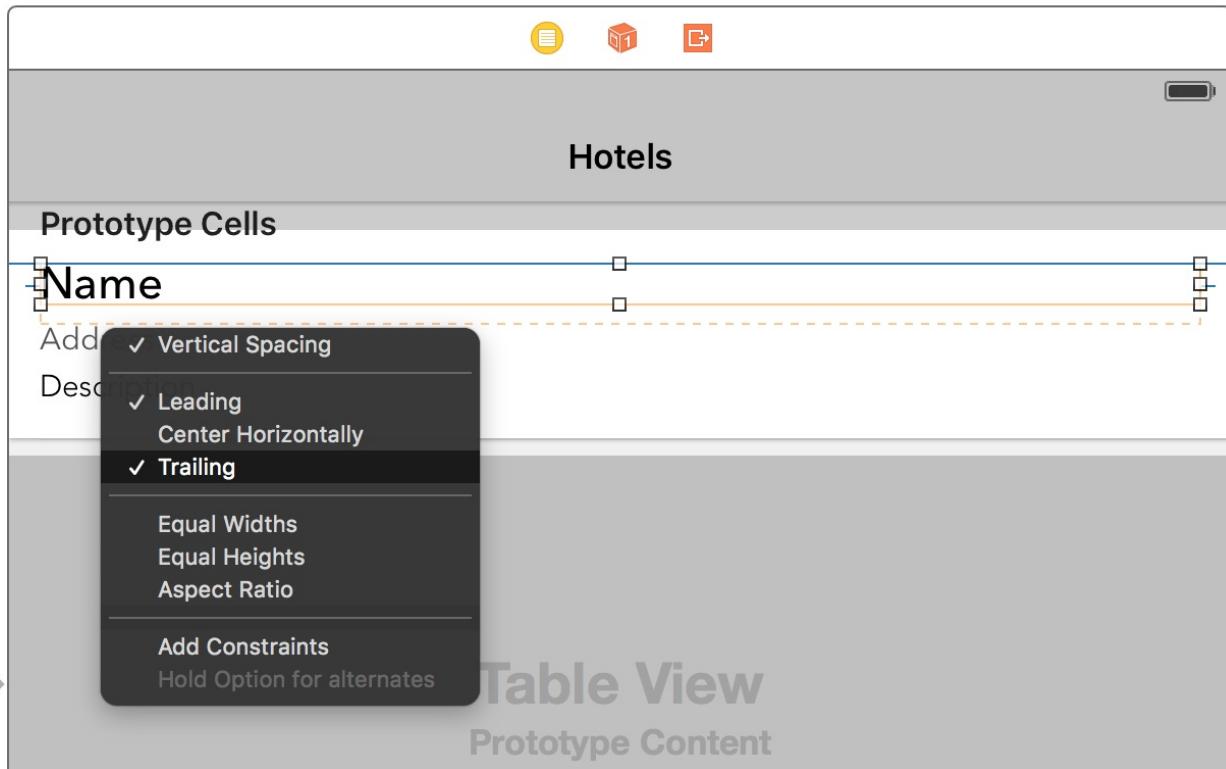
For the project template I did not define any auto layout constraints for the prototype cell; let's add a few constraints to the cell before beginning. Select the name label and click the Pin button of the auto layout menu. Add three spacing constraints for the top, left, and right sides.



Next, control-drag from the name label to the address label.



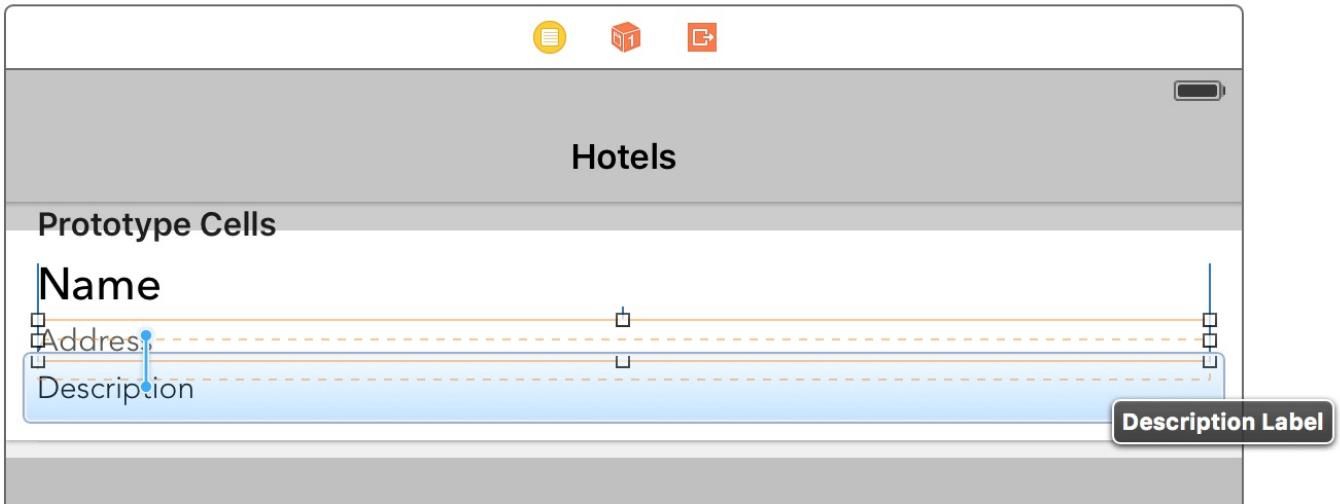
Once you release the button, Xcode shows a context menu. Hold down the shift key and select the *Vertical Spacing*, *Left* and *Right* options, and then hit return to add the constraints.



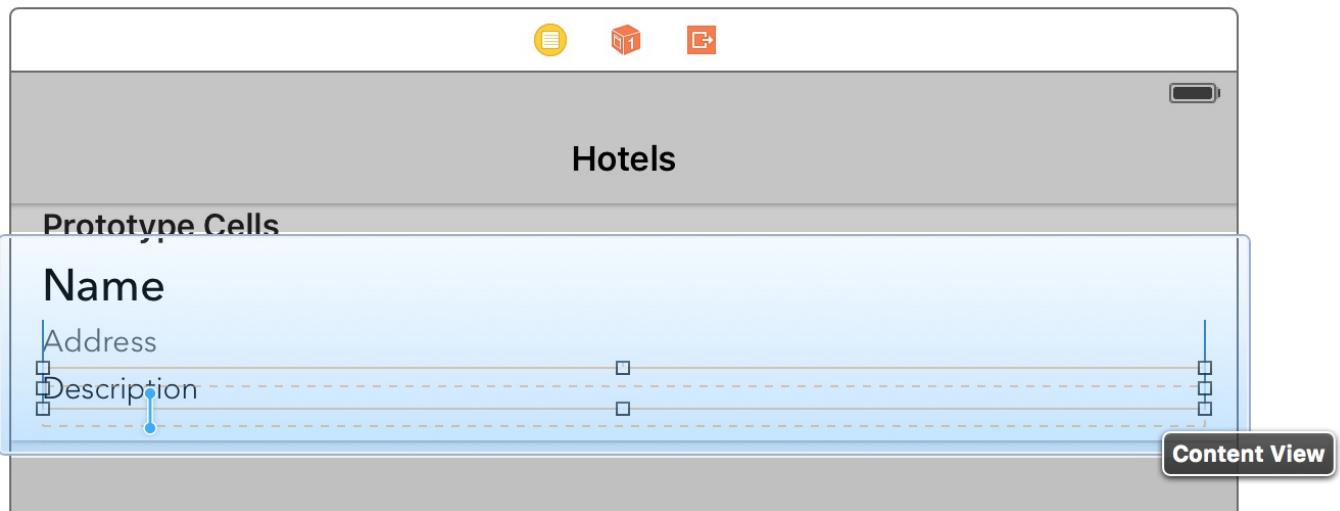
This defines three layout constraints between the name and address labels:

- A vertical spacing constraint between the name and address labels. For example, the name label should be five points away from the address label.
- The left side of the address label should align with that of the name label.
- The right side of the address label should align with that of the name label.

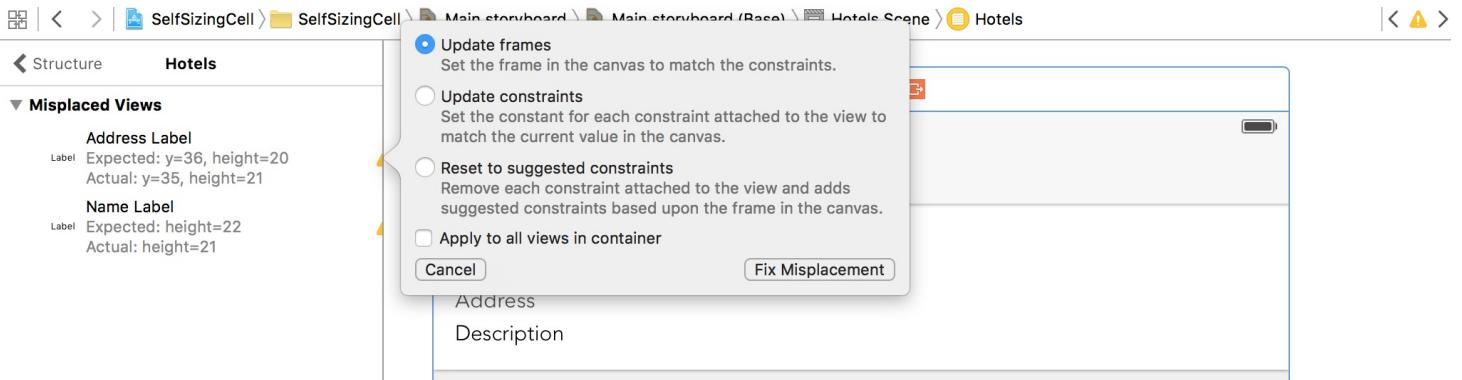
Similarly, control-drag from the address label to the description label. Again, select the *Vertical Spacing*, *Left*, and *Right* options in the context menu. This defines a similar set of layout constraints for the address and description labels.



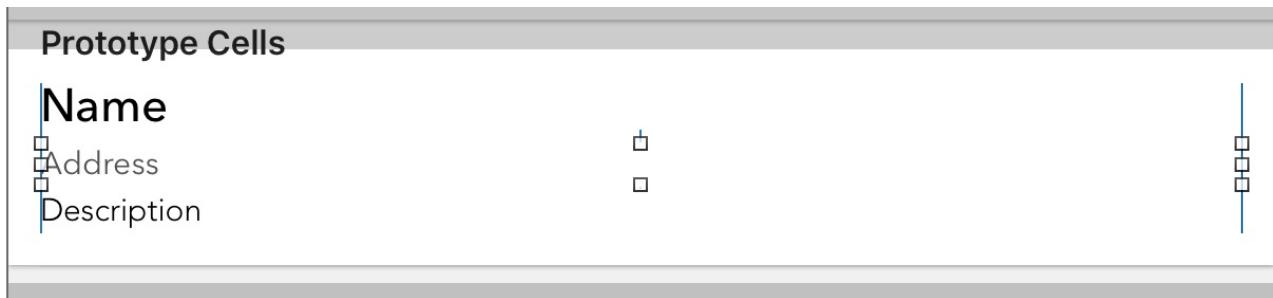
Lastly, we have to define a spacing constraint between the description label and the bottom part of the cell's content view. Control-drag from the description label to the content view of the cell.



Once releasing the button, select the “Bottom space to container margin” option. After you configured all the layout constraints, Xcode should detect several auto layout issues. Click the disclosure arrow in the Interface Builder outline view and you will see a list of the issues. Click the warning or error symbol to fix the issue (either by adding the missing constraints or updating the frame).



If you have configured the constraints correctly, your final layout should look similar to this:



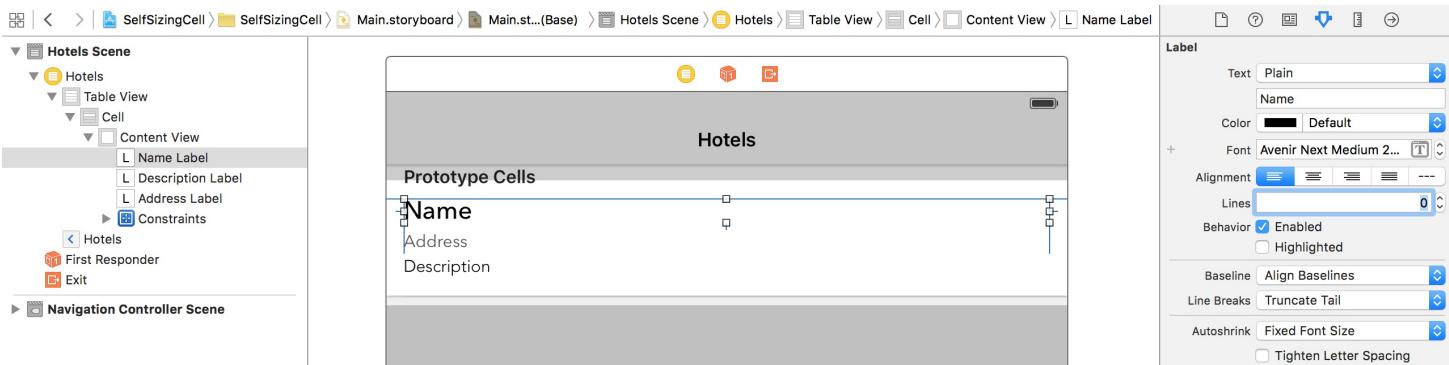
Setting Row Height

With the layout constraints configured, you now need to add the following code in the `viewDidLoad` method of `HotelTableViewController`:

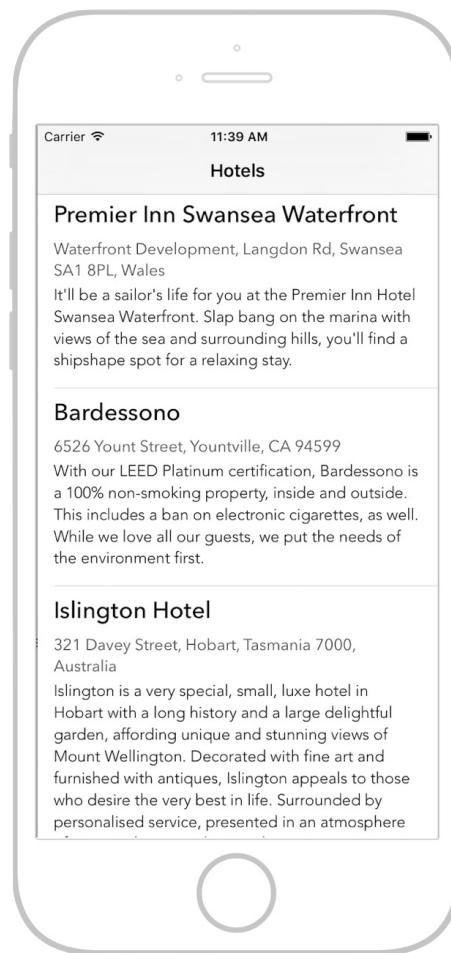
```
tableView.estimatedRowHeight = 95.0
tableView.rowHeight = UITableViewAutomaticDimension
```

The lines of code set the `estimatedRowHeight` property to 95 points, which is the current row height, and the `rowHeight` property to `UITableViewAutomaticDimension`, which is the default row height in iOS 9. In other words, you ask table view to figure out the appropriate cell size based on the provided information.

If you test the app now, the cells are still not resized. This is because all labels are restricted to display one line only. Select the *Name* label and set the number of lines under the attributes inspector to 0. By doing this, the label should now adjust itself automatically. Repeat the same procedures for both the *Address* and *Description* labels.



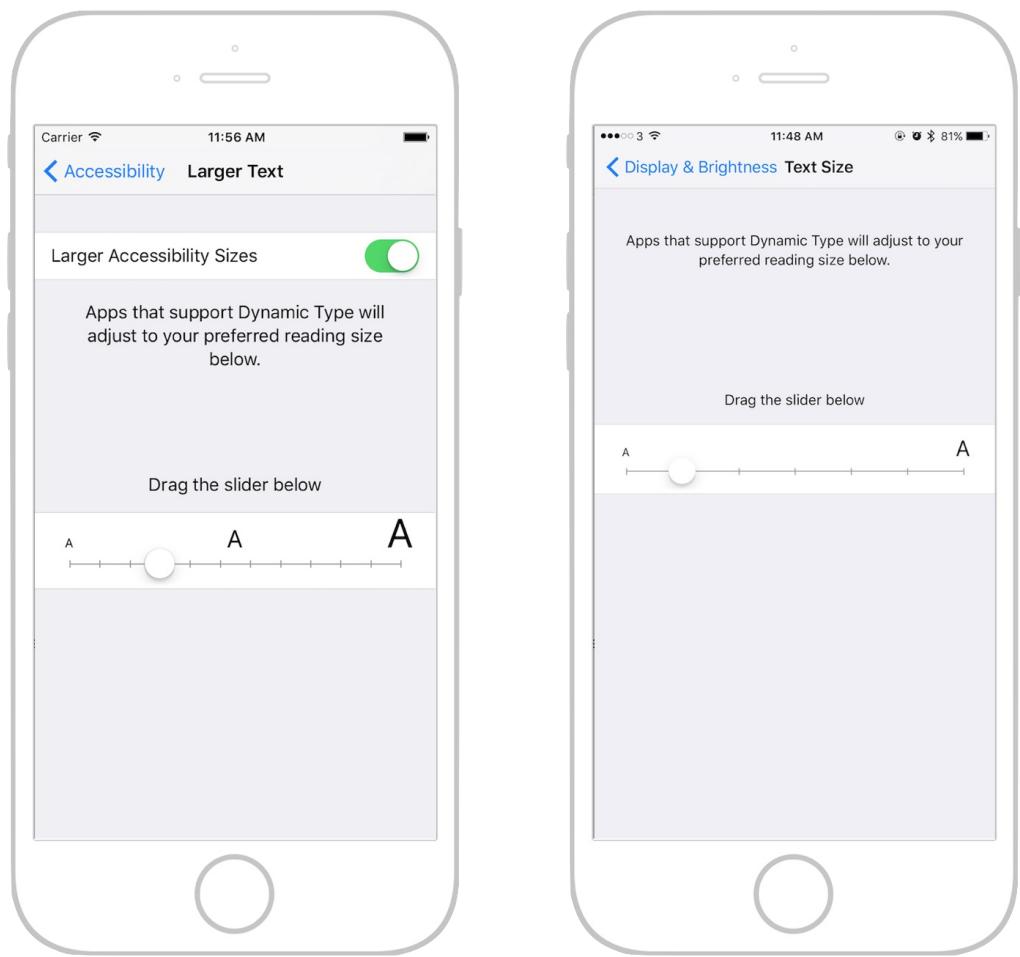
Once you made the changes, you can run the project again. This time the cells should be resized properly with respect to the content.



For reference, you can download the final project from
<https://www.dropbox.com/s/zqpdwokgkolju9g/SelfSizingCell.zip?dl=0>.

Dynamic Type Introduction

Self sizing cells are particularly useful to support Dynamic Type. You may not have heard of Dynamic Type but you probably see the setting screen (*Settings > General > Accessibility > Larger Text* or *Settings > Display & Brightness > Text Size*) shown below.



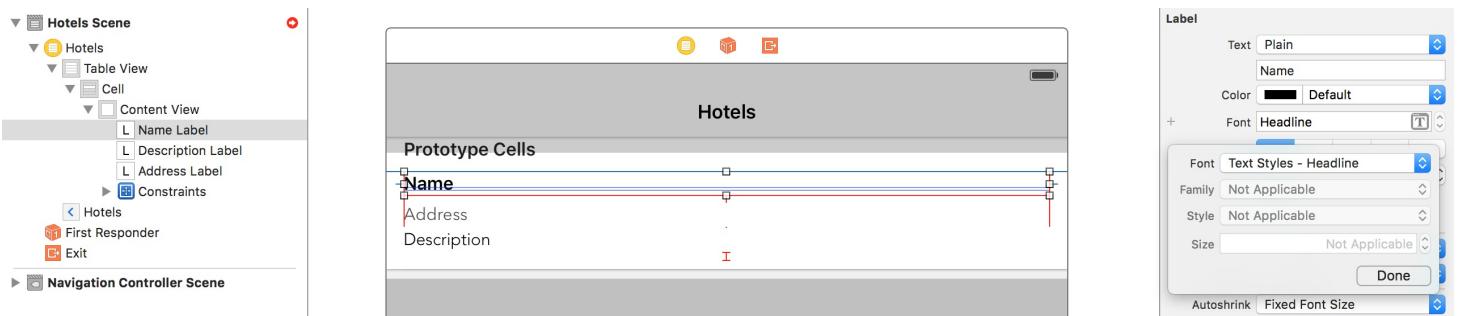
Dynamic Type was first introduced in iOS 7 - it allows users to customize the text size to fit their own needs. However, only apps that adopt dynamic type respond to the text change. I believe most of the users are not aware of this feature because only a fraction of third-party apps have adopted the feature. Starting from iOS 8, Apple wants to encourage developers to adopt Dynamic Type. All of the system applications have already adopted Dynamic Type and the built-in labels automatically have dynamic fonts. When the user changes the text size, the size of labels are going to change as well.

Furthermore, the introduction of Self Sizing Cells is a way to facilitate the widespread adoption of Dynamic Type. It saves you a lot of code from developing your own solution to adjust the row height. Once the cell is self-sized, adopting Dynamic Type is just a piece of cake.

Let's explore how to apply dynamic type to the demo app.

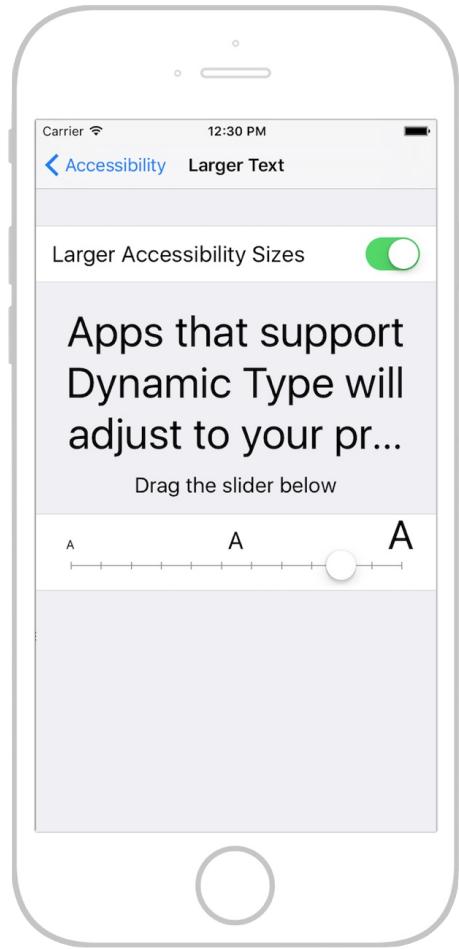
Adopting Dynamic Type

We will change the font of the label in the demo project from a custom font to a preferred font for text style (i.e. headline, body, etc). First, select the *Name* label and go the Attributes inspector. Change the font option to `Headline`.



Next, select the *Address* label and change the font to `Subhead`. Repeat the same procedure but change the font of the *Description* label to `Body`. As the font style is changed, Xcode should detect some auto layout issues. Just click the disclosure indicator on the Interface Builder outline menu to fix the issues.

That's it. Before testing the app, you should first change the text size. In the simulator, go to *Settings > General > Accessibility > Larger Text* and enable the *Larger Accessibility Sizes* option. Drag the slider to set to your preferred font size.



Now run the app and it should adapt to the text size change.



Responding to Text Size Change

Now that the demo app has adopted dynamic type, but it doesn't respond to text size change yet. While running the demo app, go to Settings and change the text size. When you switch back to the app, the font size will not be adjusted accordingly; you must quit the app and relaunch it to effectuate the change.

In order to respond to the size change, the app has to listen to the `UIContentSizeCategoryDidChangeNotification` event and reload the table view accordingly. The notification is posted when the user changes the preferred content size setting.

All you need to do now is add an observer in the `viewDidLoad` method of the `HotelTableViewController` class:

```
NSNotificationCenter.defaultCenter().addObserver(self, selector:  
"onTextSizeChange:", name: UIContentSizeCategoryDidChangeNotification, object:  
nil)
```

The code asks the Notification Center to call the `onTextSizeChange` method when the `UIContentSizeCategoryDidChangeNotification` event is posted. Implement the `onTextSizeChange` method like this:

```
func onTextSizeChange(notification: NSNotification) {
    tableView.reloadData()
}
```

When the text size changes, we simply reload the table view to refresh the content. As we use custom table view cells, we have to explicitly set the fonts each time when the cells are loaded. Update the `tableView(_:cellForRowAtIndexPath:)` method like this:

```
override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
forIndexPath: indexPath) as! HotelTableViewCell

    // Configure the cell...
    let hotel = hotels[indexPath.row]
    cell.nameLabel.text = hotel.name
    cell.addressLabel.text = hotel.address
    cell.descriptionLabel.text = hotel.description

    // Set the font style
    cell.nameLabel.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleHeadline)
    cell.addressLabel.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleSubheadline)
    cell.descriptionLabel.font =
UIFont.preferredFontForTextStyle(UIFontTextStyleBody)

    return cell
}
```

The `preferredFontForTextStyle` method of `UIFont` returns the system font with size based on the user's current content size for a specified text style. Here are the available text style options:

- `UIFontTextStyleHeadline`
- `UIFontTextStyleSubheadline`
- `UIFontTextStyleBody`
- `UIFontTextStyleFootnote`

- UIFontTextStyleCaption1
- UIFontTextStyleCaption2

Lastly, add the deinit method to remove the observer:

```
deinit {
    NSNotificationCenter.defaultCenter().removeObserver(self)
}
```

Quick note: A deinitializer, with the keyword `deinit`, is called immediately before a class instance is deallocated. When the table view controller is deallocated, we remove the observer.

Now you can test the app again. When the app is launched in the simulator, press command+shift+h to go back to the home screen. Then go to *Settings > General > Accessibility > Larger Text* and enable the *Larger Accessibility Sizes* option. Drag the slider to set to change the text size.

Once changed, press `command+shift+h` again to go back to the home screen. Launch the `SelfSizingCell` app and the size of the labels should change automatically.

Using Custom Font

Now that you have learned how to adopt Dynamic Type and enable your app to respond to text size changes. However, there is one issue with dynamic type that we haven't covered yet. The font of the text style is defaulted to *San Francisco*, Apple's new font in iOS 9. Fortunately, you can use `UIFontDescriptor` to get a font descriptor for a given text style. Based on the font descriptor, you can create your own font using `UIFont`. Here is an example:

```
var bodyFontDescriptor =
UIFontDescriptor.preferredFontDescriptorWithTextStyle(UIFontTextStyleBody)
let bodyFont = UIFont(name: "Avenir-Medium", size:
bodyFontDescriptor.pointSize)
cell.descriptionLabel.font = bodyFont
```

`UIFontDescriptor` provides a class method called `preferredFontDescriptorWithTextStyle` that returns a font descriptor of a given text style. In the sample, we ask for the font descriptor of body text style (`UIFontTextStyleBody`). The font descriptor contains the font size of the system font with respect to the current content size setting. With the font size, we can create our own

font using `UIFont` and assign it to the cell label.

This solution works, but there is an even better way to apply your preferred fonts with Dynamic Type. Swift offers a feature called *extensions* that allows developers to add new functionality to an existing class. You can even extend the functionality of a built-in class (e.g. `UIFont`). If you have Objective-C background, extensions are similar to categories in Objective-C.

We will create an extension for `UIFont` and add a new method called `preferredCustomFontForTextStyle` to the class. To create an extension, you declare the extension with the `extension` keyword. Insert this extension in the `HotelTableViewController.swift` file and put it right after the `import UIKit` statement:

```
extension UIFont {
    class func preferredCustomFontForTextStyle (textStyle: NSString) -> UIFont {
        let font =
        UIFontDescriptor.preferredFontDescriptorWithTextStyle(textStyle as String)
        let fontSize: CGFloat = font.pointSize

        if textStyle == UIFontTextStyleHeadline {
            return UIFont(name: "AvenirNext-Medium", size: fontSize)!
        } else if (textStyle == UIFontTextStyleSubheadline) {
            return UIFont(name: "Avenir-Medium", size: fontSize)!
        } else {
            return UIFont(name: "Avenir-Light", size: fontSize)!
        }
    }
}
```

The method simply returns a custom font based on the given text style. For instance, we used *AvenirNext-Medium* font for the headline text style. You can use the method of the extension just like any other methods. Simply call up the `UIFont.preferredCustomFontForTextStyle` method instead of `UIFont.preferredFontForTextStyle`. So update the `tableView(_:cellForRowAtIndexPath:)` method like this:

```
override func tableView(tableView: UITableView, cellForRowAt indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell",
forIndexPath: indexPath) as! HotelTableViewCell
    // Configure the cell...
    let hotel = hotels[indexPath.row]
```

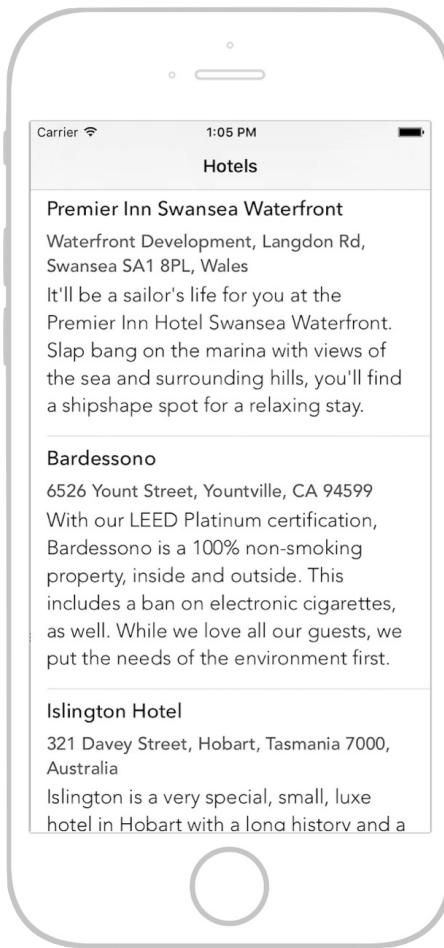
```

cell.nameLabel.text = hotel.name
cell.addressLabel.text = hotel.address
cell.descriptionLabel.text = hotel.description

// Set the font style
cell.nameLabel.font =
UIFont.preferredCustomFontForTextStyle(UIFontTextStyleHeadline)
cell.addressLabel.font =
UIFont.preferredCustomFontForTextStyle(UIFontTextStyleSubheadline)
cell.descriptionLabel.font =
UIFont.preferredCustomFontForTextStyle(UIFontTextStyleBody)

return cell
}

```



Now compile and run the project again. The app should use the custom font instead of the system font. For reference, you can download the final project from <https://www.dropbox.com/s/q5qtvw1adk83jhv/SelfSizingCellDynamicType.zip?dl=0>.

Chapter 26

XML Parsing and RSS



One of the most important tasks that a developer has to deal with when creating applications is data handling and manipulation. Data can be expressed in many different formats, and mastering at least the most common of them is a key ability for every single programmer. Speaking of mobile applications specifically now, it's quite common nowadays for them to exchange data with web applications. In such cases, the way that data is expressed may vary, but usually uses either the JSON or the XML format.

The iOS SDK provides classes for handling both of them. For managing JSON data, there is the `NSJSONSerialization` class. This one allows developers to easily convert JSON data into a Foundation object, and the other way round. I have covered JSON parsing in chapter 4. In this

chapter, we will look into the APIs for parsing XML data.

iOS offers the `NSXMLParser` class, which takes charge of doing all the hard work and, through some useful delegate methods gives us the tools we need for handling each step of the parsing. I have to say that `NSXMLParser` is a very convenient class and makes the parsing of XML data a piece of cake.

Being more specific, let me introduce you the `NSXMLParserDelegate` protocol we'll use, and what each of the methods is for. The protocol defines the optional methods that should be implemented for XML parsing. For clarification purpose, every XML data is considered as an XML document in iOS. Here are the core methods that you will usually deal with:

- `parserDidStartDocument` - This one is called when the parsing actually starts. Obviously, it is called just once per XML document.
- `parserDidEndDocument` - This one is the complement of the first one, and is called when the parser reaches the end of the XML document.
- `parser(_:parseErrorOccurred:)` - This delegate method is called when an error occurs during the parsing. The method contains an error object, which you can use to define the actual error.
- `parser(_:didStartElement:namespaceURI:qualifiedName:attributes:)` - This one is called when the opening tag of an element (e.g.
- `parser(_:didEndElement:namespaceURI:qualifiedName:)` - Contrary to the above method, this is called when the closing tag of an element (e.g.) is found.
- `parser(_:foundCharacters:)` - This method is called during the parsing of the contents of an element. Its second argument is a string value containing the character that was just parsed.

To help you understand the usage of the methods, we will build a simple RSS reader app together. The app will consume an RSS feed (in XML format), parse its content and display the data in a table view.

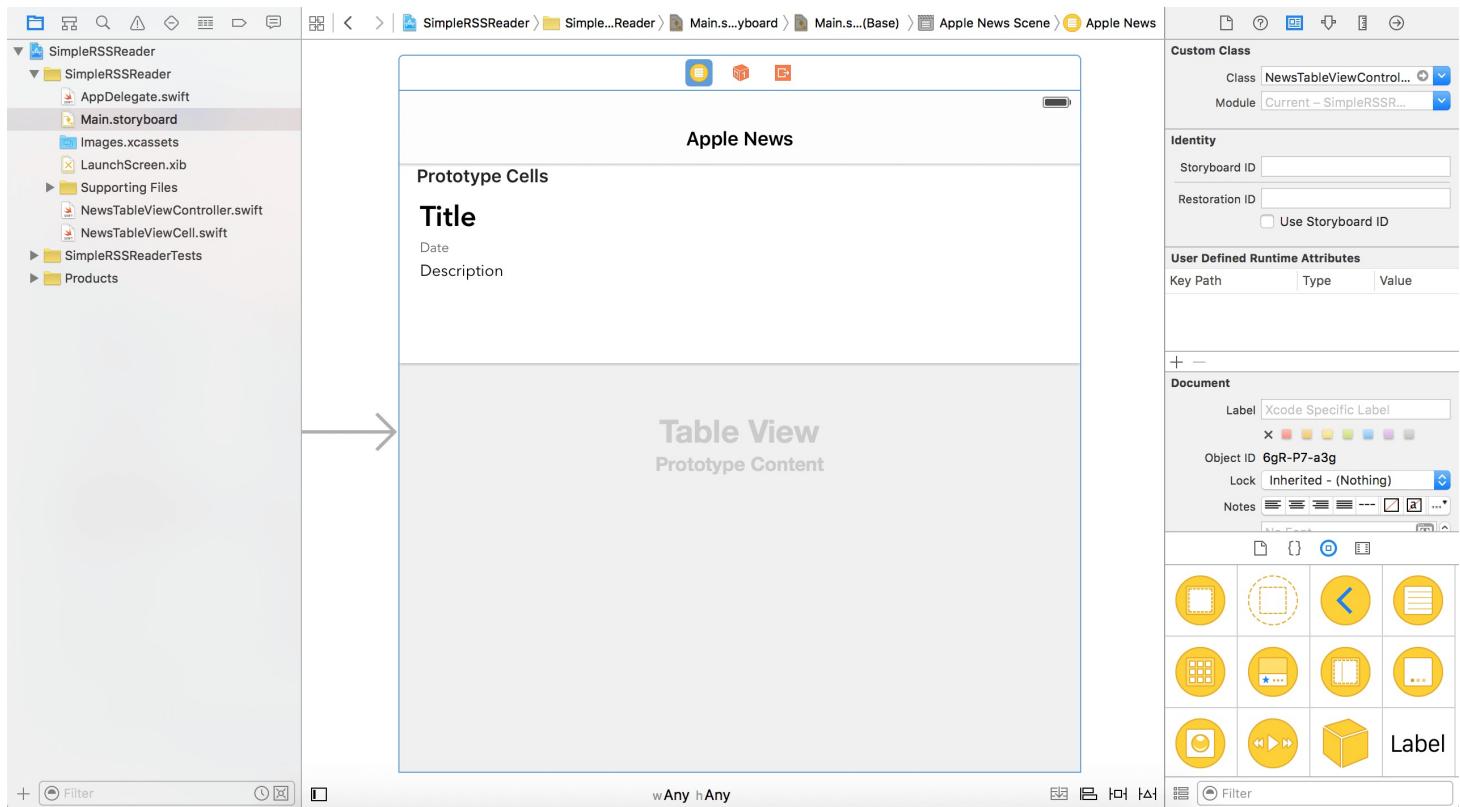
Demo App

I can show you how to build a plain XML parser that reads an XML file but that would be boring. Wouldn't it be better to create a simple RSS reader? The RSS Reader app reads an RSS

feed of Apple, which is essentially XML formatted plain text. It then parses the content, extracts the news articles and shows them in a table view.

To help you get started, I have created the project template that comes with a prebuilt storyboard and view controller classes. You can download the template from <https://www.dropbox.com/s/kaptncr3mp2oibe/SimpleRSSReaderStart.zip?dl=0>.

The `NewsTableViewController` class is associated with the table view controller in the storyboard, while the `NewsTableViewCell` class is connected with the custom cell. The custom cell is designed to display the title, date and description of a news article. I have also configured the auto layout constraints of the cell so that it can be self-sized.



A Sample RSS Feed

We will use a free RSS feed from Apple (<https://www.apple.com/main/rss/hotnews/hotnews.rss>) as the source of XML data. If you load the feed into any browser (e.g. Chrome), you will get a sample of the XML data, as shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
<channel>
<title>Apple Hot News</title>
<link>http://www.apple.com/hotnews/</link>
<description>Hot News provided by Apple.</description>
<language>en-us</language>
<copyright>Copyright 2015, Apple Inc.</copyright>
<pubDate>Tue, 13 Oct 2015 14:26:41 PDT</pubDate>
<lastBuildDate>Tue, 13 Oct 2015 14:26:41 PDT</lastBuildDate>
<category>Apple</category>
<generator>In house</generator>
<docs>http://blogs.law.harvard.edu/tech/rss/</docs><item>
<title>Apple Updates iMac Family with Stunning New Retina Displays</title>
<link>http://www.apple.com/imac/?sr=hotnews.rss</link>
<description>Apple has updated the entire iMac family, bringing a new Retina 4K display to the 21.5-inch iMac for the first time and the Retina 5K display to every 27-inch iMac. The updated iMacs also feature more powerful processors and graphics, two Thunderbolt 2 ports, and new storage options that make the high-performance Fusion Drive even more affordable. Apple also introduced a new lineup of wireless accessories including the all-new Magic Keyboard, Magic Mouse 2, and Magic Trackpad 2. "From the first iMac to today, the spirit of iMac has never wavered – deliver the ultimate desktop experience with the latest technologies, gorgeous displays, and cutting-edge designs," said Philip Schiller, Apple's senior vice president of Worldwide Marketing. "These are the most stunning iMacs we've ever made. With our gorgeous new Retina displays, more powerful processors and graphics, and all-new Magic accessories, the new iMac continues to redefine the ultimate desktop experience."</description>
<pubDate>Tue, 13 Oct 2015 11:12:45 PDT</pubDate>
</item>
<item>
<title>OS X El Capitan Available as Free Update Tomorrow</title>
<link>http://www.apple.com/pr/library/2015/09/29OS-X-El-Capitan-Available-as-a-Free-Update-Tomorrow.html?sr=hotnews.rss</link>
<description>OS X El Capitan, the latest major release of the world's most advanced desktop operating system, will be available Wednesday, September 30, as a free update for Mac users. El Capitan builds on the groundbreaking features and beautiful design of OS X Yosemite. "People love using their Macs, and one of the biggest reasons is the power and ease of use of OS X," said Craig Federighi, Apple's senior vice president of Software Engineering. "El Capitan refines the Mac experience and improves performance in a lot of little ways that make a very big difference. Feedback from our OS X beta program has been incredibly positive and we think customers are going to love their Macs even more with El Capitan."</description>
<pubDate>Tue, 29 Sep 2015 14:03:20 PDT</pubDate>
</item>
.
.
```

```
</channel>  
</rss>
```

As I said before, an RSS feed is essentially XML formatted plain text. It's human readable. Every RSS feed should conform to a certain format. I will not go into the details of RSS format. If you want to learn more about RSS, you can refer to <http://en.wikipedia.org/wiki/RSS>. The part that we are particularly interested in are those elements within the item tag. The section highlighted in yellow represents a single article. Each article basically includes the title, description, published date and link. For our RSS Reader app, the nodes that we are interested in are:

- title
- description
- pubDate

Our job is to parse the XML data and get all the items so as to display them in the table view. When we talk about XML parsing, there are two general approaches: *Tree-based* and *Event-driven*. The `NSXMLParser` class adopts the event-driven approach. It generates a message for each type of parsing event to its delegate, that adopts the `NSXMLParserDelegate` protocol. To better elaborate the concept, let's consider the following simplified XML content:

```
<item>  
<title>Apple Announces Record iPhone 6s and iPhone 6s Plus Sales</title>  
<pubDate>Mon, 28 Sep 2015 14:37:05 PDT</pubDate>  
</item>
```

When parsing the above XML, the `NSXMLParser` object would inform its delegate of the following events:

Event No.	Event Description	Invoked method of the delegate
1	Started parsing the XML document	<code>parserDidStartDocument(_:)</code>
2	Found the start tag for	<code>parser(_:didStartElement:namespaceURI:qualifiedName:attribu</code>

	element <i>item</i>	
3	Found the start tag for element <i>title</i>	<code>parser(_:didStartElement:namespaceURI:qualifiedName:attribu</code>
4	Found the characters <i>Apple Announces Record iPhone 6s and iPhone 6s Plus Sales</i>	<code>parser(_:foundCharacters:)</code>
5	Found the end tag for element <i>title</i>	<code>parser(_:didEndElement:namespaceURI:qualifiedName:)</code>
6	Found the start tag for element <i>pubDate</i>	<code>parser(_:didStartElement:namespaceURI:qualifiedName:attribu</code>
7	Found the characters <i>Mon, 28 Sep 2015 14:37:05 PDT</i>	<code>parser(_:foundCharacters:)</code>
8	Found the end tag for element <i>pubDate</i>	<code>parser(_:didEndElement:namespaceURI:qualifiedName:)</code>
9	Found the end tag for element <i>item</i>	<code>parser(_:didEndElement:namespaceURI:qualifiedName:)</code>
10	Ended parsing the XML document	<code>parserDidEndDocument(_:)</code>

By implementing the methods of the `NSXMLParserDelegate` protocol, you can retrieve the data

you need (e.g. title) and save them accordingly.

Building the Feed Parser

Having an idea of XML parsing in iOS, we can now proceed to create the `FeedParser` class. Right click the Project Navigator and select `New File...` to create a new class. Name the class `FeedParser` and set it as a subclass of `NSObject`.

Here are a few things we will implement in the class:

1. Download the content of RSS Feed asynchronously. The `NSXMLParser` class provides a convenient method to specify a URL of the XML content. If you use the method, the class automatically downloads the content for further parsing. However, it only works synchronously. That means that the main thread (or any UI update) is blocked while retrieving the feed. We don't want to block the UI so we will use `NSURLSession` to download the content asynchronously.
2. Once the XML content is downloaded, we will initialize an `NSXMLParser` object to start the parsing.
3. Use the `NSXMLParser` delegate methods to handle the parsed data. During the parsing, we look for the value of *title*, *description* and *pubDate* tags, and then we group them into a tuple and save the tuple in the `rssItems` array.
4. Lastly, we call the `parserCompletionHandler` closure when the parsing completes.

Let's see everything step by step. Initially, open the `FeedParser.swift` file, and adopt the `NSXMLParserDelegate` protocol. It's necessary to do that in order to handle the data later.

```
class FeedParser: NSObject, NSXMLParserDelegate
```

Now declare an array of tuples to store the items:

```
private var rssItems:[(title: String, description: String, pubDate: String)] = []
```

We use tuples to temporarily store the parsed items. If you haven't heard of Tuple, this is one of the nifty features of Swift. It groups multiple values into a single compound value. Here we group *title*, *description* and *pubDate* into a single item.

Quick Tip: To learn more about Tuples, check out Apple's official documentation at https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Tuples.html CH5-ID329.

Next, declare the following variables in the `FeedParser` class:

```
private var currentElement = ""
private var currentTitle:String = "" {
    didSet {
        currentTitle =
currentTitle.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewline
                                              CharacterSet())
    }
}
private var currentDescription:String = "" {
    didSet {
        currentDescription =
currentDescription.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewline
                                              CharacterSet())
    }
}
private var currentPubDate:String = "" {
    didSet {
        currentPubDate =
currentPubDate.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewline
                                              CharacterSet())
    }
}

private var parserCompletionHandler:([(title: String, description: String,
pubDate: String)] -> Void)?
```

The `currentElement` variable is used as a temporary variable for storing the currently parsed element (e.g.

The `currentTitle`, `currentDescription` and `currentPubDate` variables are used to store the value of an element (e.g. the value within the `<title>` tags) when the `parser(_:foundCharacters:)` method is called. Because the value may contain white space and new line characters, we add a property observer to trim these characters.

The `parserCompletionHandler` variable is a closure to be specified by the caller class. You can

think of it as a callback function. When the parsing finishes, there are certain actions we should take, such as displaying the items in a table view. This completion handler will be called to perform the actions at the end of the parsing. We will talk more about it in a later section.

Next, let's add a new method called `parseFeed` :

```
func parseFeed(feedUrl: String, completionHandler: (([title: String,
description: String, pubDate: String]) -> Void)?) -> Void {

    self.parserCompletionHandler = completionHandler

    let request = NSURLRequest(URL: NSURL(string: feedUrl)!)
    let urlSession = NSURLSession.sharedSession()
    let task = urlSession.dataTaskWithRequest(request, completionHandler: {
        (data, response, error) -> Void in

        guard let data = data else {
            if let error = error {
                print(error)
            }
            return
        }

        // Parse XML data
        let parser = NSXMLParser(data: data)
        parser.delegate = self
        parser.parse()

    })
    task.resume()
}
```

This method takes in two variables: `feedUrl` and `completionHandler`. The feed URL is a `String` object containing the link of the RSS feed. The completion handler is the one we just discussed, and will be called when the parsing finishes. In this method, we create an `NSURLSession` object and a download task to retrieve the XML content asynchronously. When the download completes, we initialize the parser object with the XML data, set the delegate to itself, and start the parsing.

Now let's implement the delegate methods one by one. Referring to the event table I mentioned before, the first delegate method to be invoked is the `parserDidStartDocument` method. Implement the method like this:

```
func parserDidStartDocument(parser: NSXMLParser) {
    rssItems = []
}
```

To begin, here we just initialize an empty `rssItems` array. When a new element (e.g. `<item>`) is found, the `didStartElement` method is called. Insert this method in the class:

```
func parser(parser: NSXMLParser, didStartElement elementName: String,
namespaceURI: String?, qualifiedName qName: String?, attributes attributeDict:
[NSObject : AnyObject]) {

    currentElement = elementName

    if currentElement == "item" {
        currentTitle = ""
        currentDescription = ""
        currentPubDate = ""
    }
}
```

We simply assign the name of the element to the `currentElement` variable. If the `<item>` tag is found, we reset the temporary variables of *title*, *description*, *pubDate* to blank for later use.

When the value of an element is parsed, the `parser(_:foundCharacters:)` method is called with a string representing all or part of the characters of the current element. Implement the method like this:

```
func parser(parser: NSXMLParser, foundCharacters string: String) {

    switch currentElement {
    case "title": currentTitle += string
    case "description": currentDescription += string
    case "pubDate": currentPubDate += string
    default: break
    }
}
```

Note that the `string` object may only contain part of the characters of the element. Instead of assigning the `string` object to the temporary variable, we append it to the end.

When the closing tag (e.g. `</item>`) is found, the

`parser(_:didEndElement:namespaceURI:qualifiedName:)` method is called. Here we only perform

actions when the tag is item.

```
func parser(parser: NSXMLParser, didEndElement elementName: String,  
namespaceURI: String?, qualifiedName qName: String?) {  
  
    if elementName == "item" {  
        var rssItem = (title: currentTitle, description: currentDescription,  
pubDate: currentPubDate)  
        rssItems += [rssItem]  
    }  
}
```

We create a tuple using the *title*, *description* and *pubDate* tags just parsed, and then we add the tuple to the `rssItems` array.

Lastly, we come to the `parserDidEndDocument` method. When the parsing is completed successfully, the method is invoked. In this method, we will call up `parseCompletionHandler` as specified by the caller to perform any follow-up actions:

```
func parserDidEndDocument(parser: NSXMLParser) {  
    parserCompletionHandler?(rssItems)  
}
```

Optionally, we also implement the following in the `FeedParser` class:

```
func parser(parser: NSXMLParser, parseErrorOccurred parseError: NSError) {  
    print(parseError.localizedDescription)  
}
```

This method is called when the parser encounters a fatal error. Now that we have completed the implementation of `FeedParser`. Let's go to the `NewsTableViewController.swift` file, which is the caller of the `FeedParser` class. Declare a variable to store the article items:

```
private var rssItems:[(title: String, description: String, pubDate: String)]?
```

In the `viewDidLoad` method, insert the following lines of code:

```
let feedParser = FeedParser()  
feedParser.parseFeed("https://www.apple.com/main/rss/hotnews/hotnews.rss",  
completionHandler: {  
    (rssItems: [(title: String, description: String, pubDate: String)]) -> Void  
in
```

```

self.rssItems = rssItems
NSOperationQueue.mainQueue().addOperationWithBlock({ () -> Void in
    self.tableView.reloadSections(NSIndexSet(index: 0), withRowAnimation:
.None)
})
})
}

```

Here we create a `FeedParser` object and call up the `parseFeed` method to parse the specified RSS feed. As said before, the `completionHandler`, which is a closure, will be called when the parsing completes. So we save `rssItems` and ask the table view to display them by reloading the table data. Note that the UI update should be performed in the main thread.

Lastly, update the following methods to load the items in the table view:

```

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    // Return the number of rows in the section.
    guard let rssItems = rssItems else {
        return 0
    }

    return rssItems.count
}

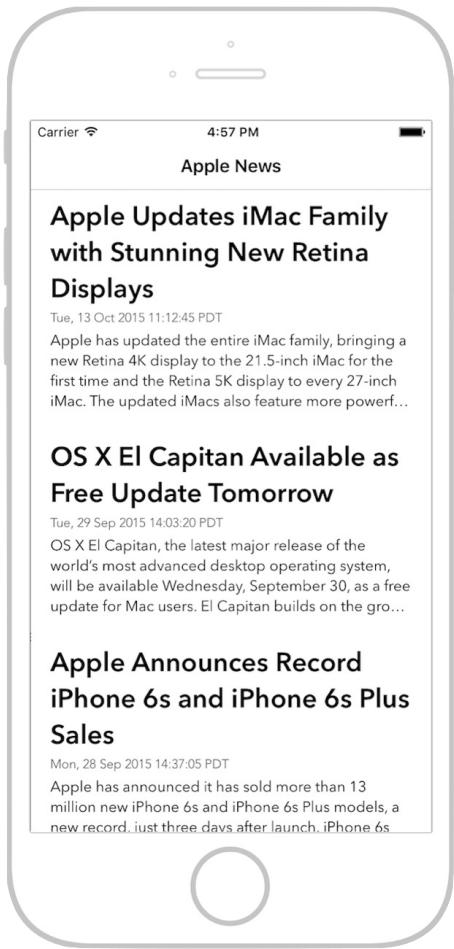
override func tableView(tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
forIndexPath: indexPath) as! NewsTableViewCell

    // Configure the cell...
    if let item = rssItems?[indexPath.row] {
        cell.titleLabel.text = item.title
        cell.descriptionLabel.text = item.description
        cell.dateLabel.text = item.pubDate
    }

    return cell
}

```

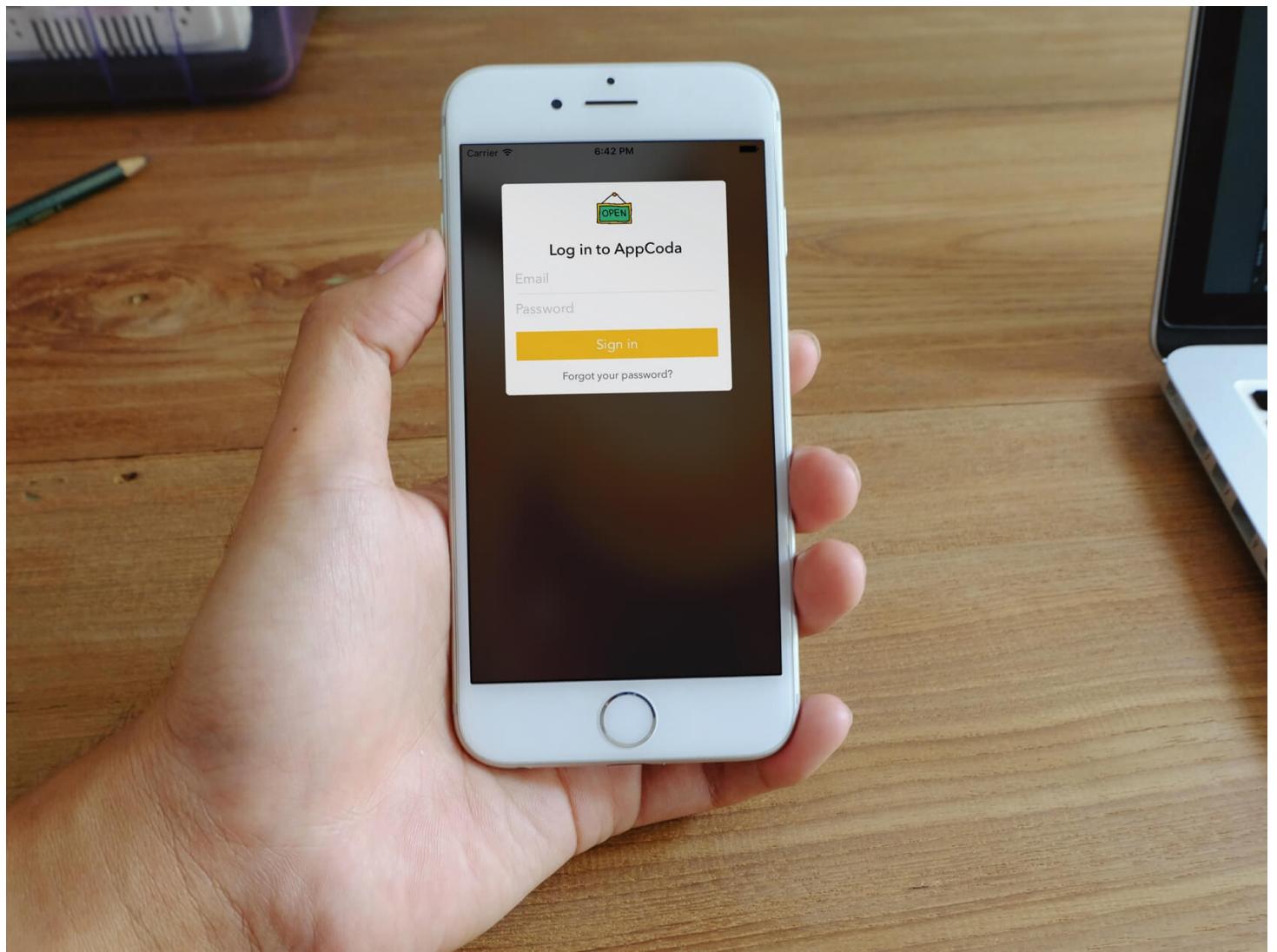
Great! You can now run the project. If you're testing the app using the simulator, make sure your computer is connected to the Internet. The RSS Reader app should be able to retrieve the news feed of Apple.



For reference, you can download the final Xcode project from
<https://www.dropbox.com/s/2x4j2e56dqs4qdk/SimpleRSSReader.zip?dl=0>.

Chapter 27

Applying a Blurred Background Using UIVisualEffect



It's been two years now. I still remembered how Jonathan Ive described the user interface redesign of iOS 7. Other than "Flat" design, the mobile operating system introduced *new types of depth* in the words of the Apple's renowned design guru.

One of the ways to achieve depth is to use layering and translucency in the view hierarchy. The use of blurred backgrounds could be found throughout the mobile operating system. For

instance, when you swipe up the Control Center, its background is blurred. Moreover, the blurring effect is dynamic and keeps changing with respect to the background image. At that time, Apple did not provide APIs for developers to create such stunning visual effects. To replicate the effects, developers were required to create their own solutions or make use of the third-party libraries.



Starting from iOS 8, Apple provided a new method which makes it very easy to create translucent and blurring-style effects. It introduced a new visual effect API that lets developers apply visual effects to a view. You can now easily add blurring effect to an existing image.

In this chapter, I will go through the new API with you. Again, we will build a simple app to see how to apply the blurring effect.

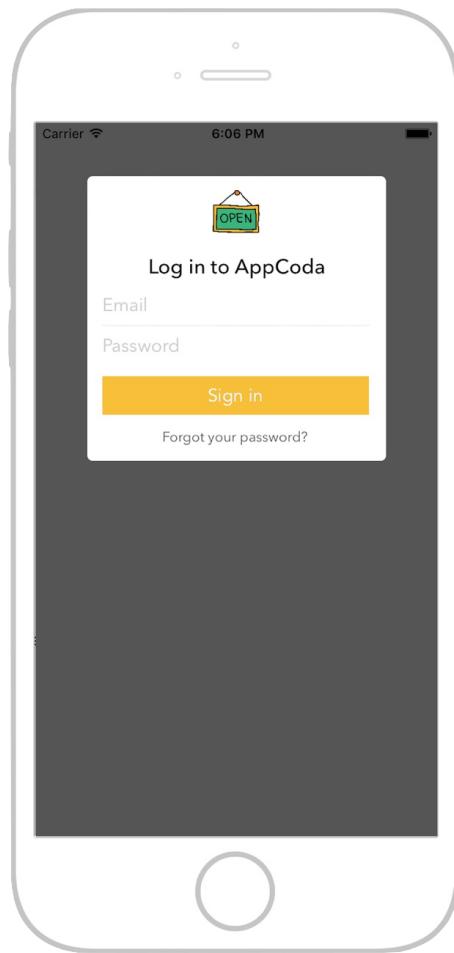
The Demo App

The demo app is not fully functional and primarily used for demonstrating the blurring effect.

It will work like this: when launched, it randomly picks an image from its image set. The selected image, with blurring effect applied, is used as a background image for the login screen.

To keep your focus on learning the `UIVisualEffect` API, I have created the project template for you. Firstly, download the project from

<https://www.dropbox.com/s/vvhmjf6cfp6kk47/VisualStyleStart.zip?dl=0> and have a trial run. The resulting app should look like the screenshot shown below. It now only displays a background view in gray. Next up, we will change it to an image background with a live blurring effect.



Understanding `UIVisualEffect` and `UIVisualEffectView`

The iOS SDK provides two UIKit classes for applying visual effects:

- `UIVisualEffect` - There are only two kinds of visual effects including blur and vibrant. The `UIBlurEffect` class is used to apply a blurring effect to the content layered behind a

`UIVisualEffectView`. A blur effect comes with three types of style: *ExtraLight*, *Light* and *Dark*. The `UIVibrantEffect` class is designed for adjusting the color of the content, such that the element (e.g. label) inside a blurred view looks sharper.



- `UIVisualEffectView` - This is the view that actually applies the visual effect. The class takes in a `UIVisualEffect` object as a parameter. Depending on the parameter passed, it performs a blur or vibrant effect to the existing view.

To apply a blurring effect, you first create a `UIBlurEffect` object like this:

```
let blurEffect = UIBlurEffect(style: UIBlurEffectStyle.Light)
```

Here we create a blurring effect with a *Light* style. Once you have created the visual effect, you initialize a `UIVisualEffectView` object like this:

```
let blurEffectView = UIVisualEffectView(effect: blurEffect)
```

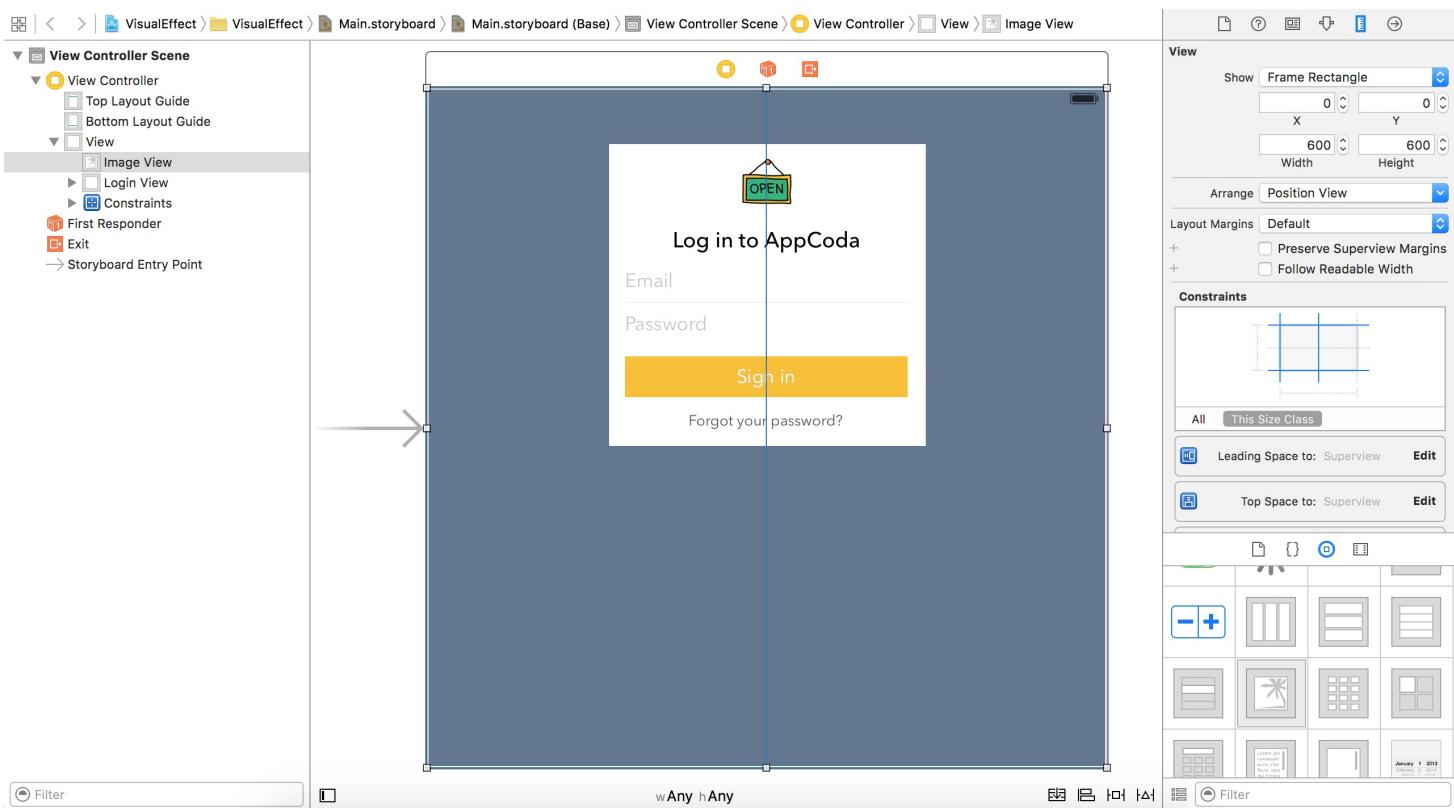
Suppose you have a `UIImageView` object serving as a background view; you can simply add the `blurEffectView` to the view hierarchy using the `addSubview` method and the background view will be blurred:

```
backgroundImageView.addSubview(blurEffectView)
```

Now that you have some ideas about the visual effect API, let's continue to work on the demo app.

Adding a Background Image View

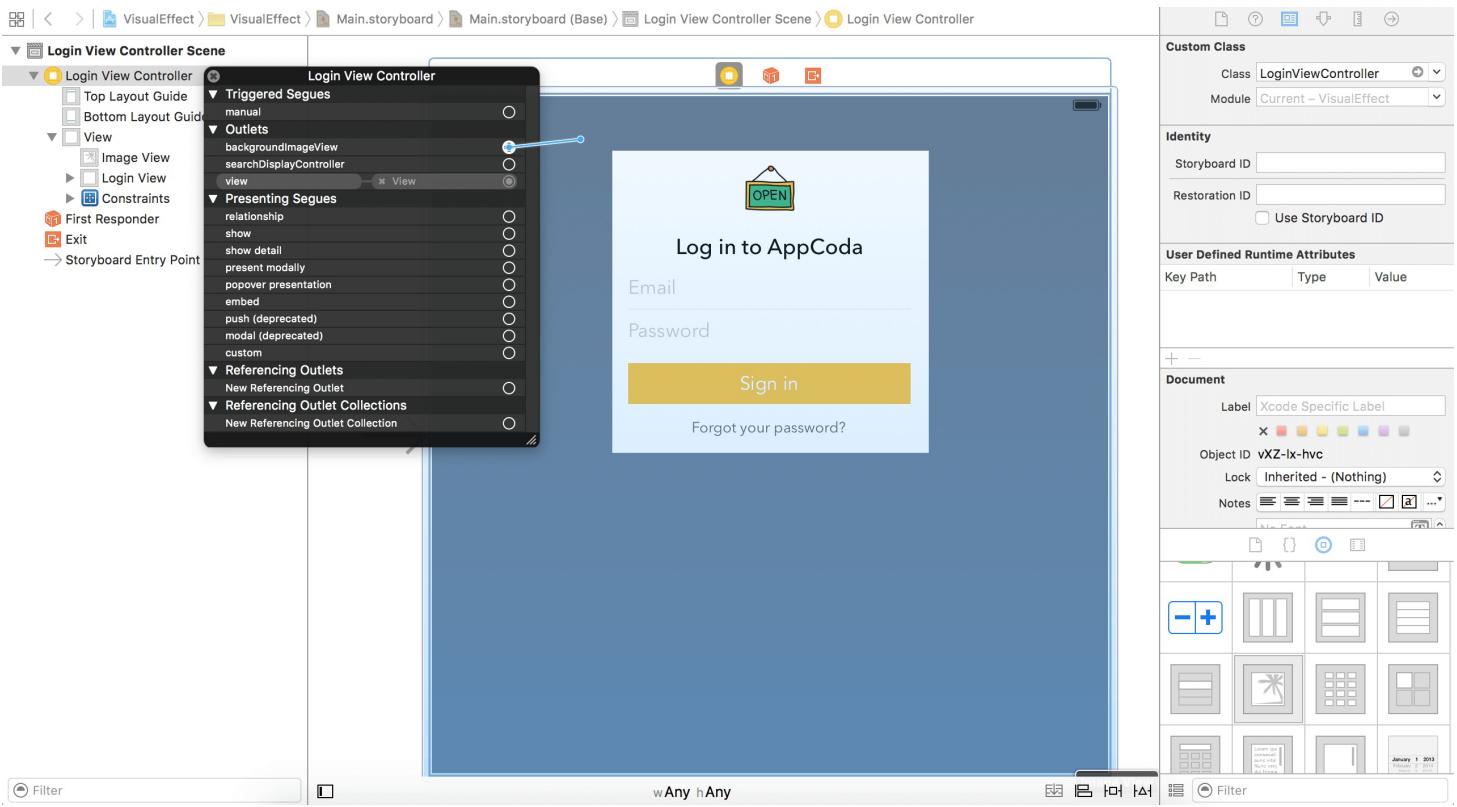
First, open `Main.storyboard` and drag an image view to the view controller. As this image view is used as a background image, make sure you put the image view beneath the login view (refer to the document outline).



Once you have added the image view, select it and add the auto layout constraints. You can simply click the Issue button of the auto layout menu in the Interface Builder. Then select *Add Missing Constraints* under Selected Views. Xcode should automatically add the required layout constraints to the image view. Next, go to the `LoginViewController.swift` file and add an outlet variable for the background image view:

```
@IBOutlet var backgroundImageView: UIImageView!
```

Now go back to the storyboard and establish a connection between the outlet variable and the image view.



Applying a Blurring Effect

The project template already includes five background images. Every time the app is loaded, it will randomly pick one and use it as the background image. Now declare the following array in `LoginViewController.swift`:

```
let imageSet = ["cloud", "coffee", "food", "pmq", "temple"]
```

Also declare a variable to hold the `UIVisualEffectView` object:

```
var blurEffectView: UIVisualEffectView?
```

Next, update the `viewDidLoad` method with the following code:

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

    // Randomly pick an image
    let selectedImageIndex = Int(arc4random_uniform(5))

    // Apply blurring effect
    backgroundImageView.image = UIImage(named: imageSet[selectedImageIndex])
```

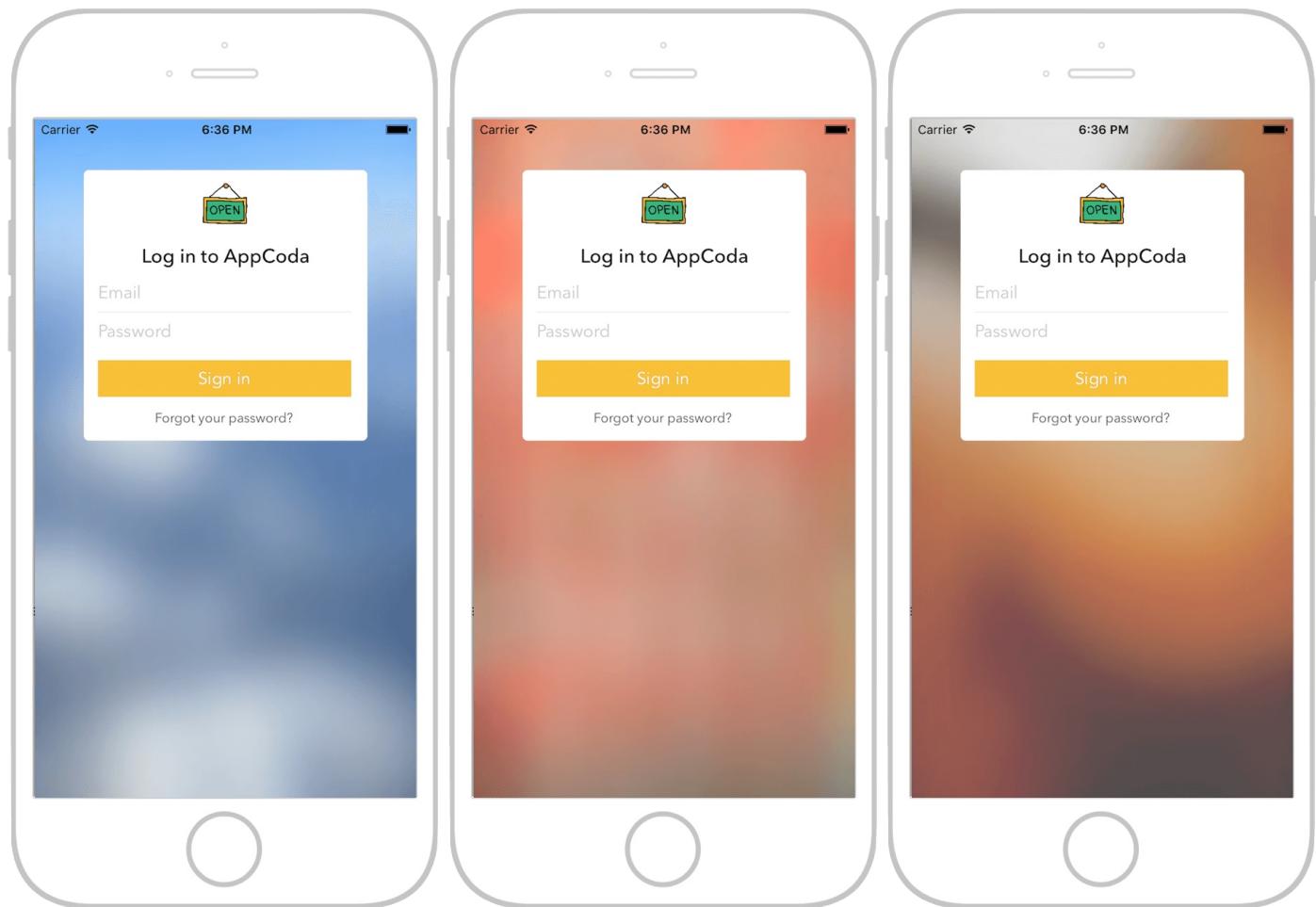
```
let blurEffect = UIBlurEffect(style: UIBlurEffectStyle.Light)
blurEffectView = UIVisualEffectView(effect: blurEffect)
blurEffectView?.frame = view.bounds
backgroundImageView.addSubview(blurEffectView!)
}
```

To pick an image randomly, we use the `arc4random_uniform()` function to generate a random number. By specifying 5 as the parameter, the function returns a value between 0 and 4. The rest of the code is similar to what we have discussed in the previous section. You are free to configure other blurring styles such as Dark.

To ensure the blur effect works in landscape mode, we have to update the `frame` property when the device's orientation changes. Insert the following method in the class:

```
override func traitCollectionDidChange(previousTraitCollection:
UITraitCollection?) {
    blurEffectView?.frame = view.bounds
}
```

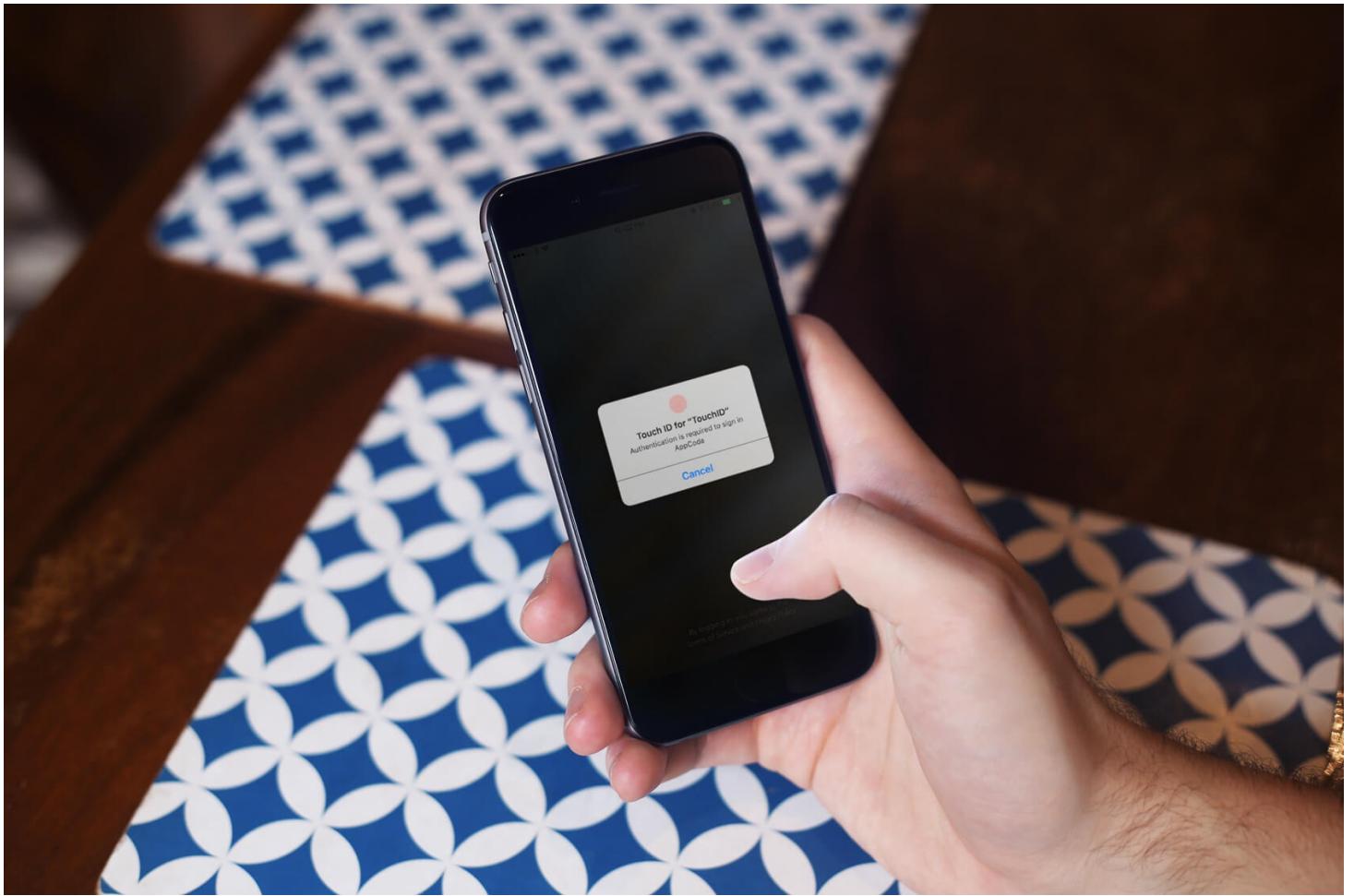
When the orientation is altered, the `traitCollectionDidChange` method will be called. We then update the frame property accordingly. Now run the project and see what you get. If you followed everything correctly, your app will display a blurred background.



For reference, you can download the final project from
<https://www.dropbox.com/s/u7ubjamxl1vq6t4/VisualEffect.zip?dl=0>.

Chapter 28

Using Touch ID For Authentication



Touch ID is Apple's biometric fingerprint authentication technology, which was first seen on the iPhone 5S in 2013. As of today, the feature is available on most iOS devices including iPhones and iPads. Touch ID is built into the home button and very simple to use. Once the steel ring surrounding the home button detects your finger, the Touch ID sensor immediately reads your fingerprint, analyses it, and provides you access to your phone.

When it was first unveiled, along with the release of iOS 7, you could only use Touch ID to unlock your iPhone, and authorize your purchases on the App Store or iTunes Store.

Security and privacy are the two biggest concerns for the fingerprint sensor. According to

Apple, your device does not store any images of your fingerprints; the scan of your fingerprint is translated into a mathematical representation, which is encrypted and stored on the Secure Enclave of the A7, A8, or A8X chip. The fingerprint data is used by the Secure Enclave only for fingerprint verifications; even the iOS itself has no way of accessing the fingerprint data.

Quick note: To learn more about Secure Enclave, you can refer to the [Apple's Security White Paper for iOS](#).

In iOS 7, Apple did not allow developers to get hold of the APIs to implement Touch ID authentication in their own apps. With every major version release of iOS, Apple ships along a great number of new technologies and frameworks. iOS 8 brings quite exciting new innovations, among them, is the release of the public APIs for Touch ID authentication. You can now integrate your apps with fingerprint authentication, potentially replacing passwords or PINs. The usage of the TouchID is based on a new framework, named *Local Authentication*. The framework provides methods to prompt a user to authenticate. It offers a ton of opportunities for developers. You can use Touch ID authentication for login, or authorize secure access to sensitive information within an app.

The heart of the Local Authentication framework is the `LAContext` class, which provides two methods:

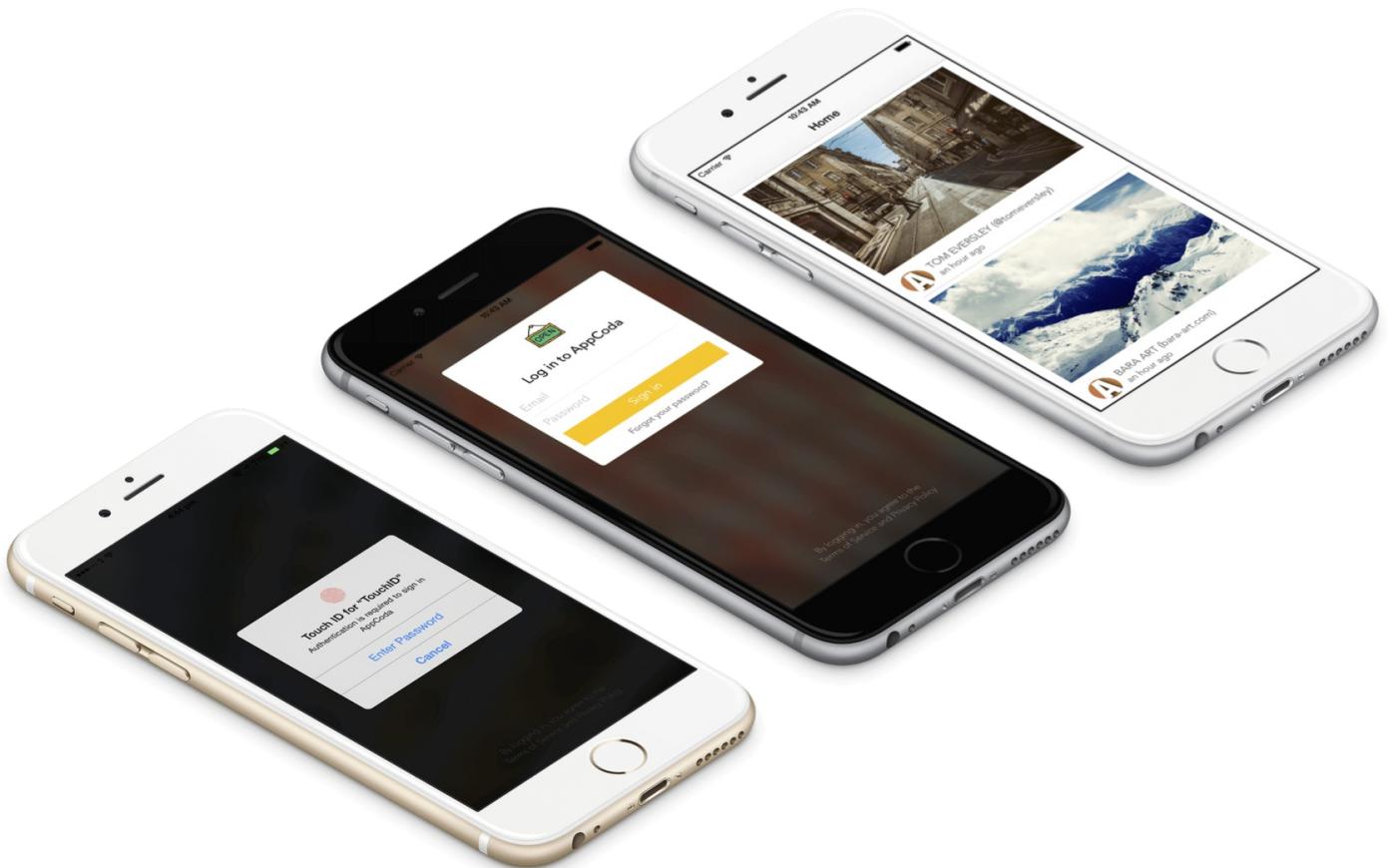
- `canEvaluatePolicy(_:error:)` - this method evaluates the given authentication policy to see if we can proceed with the authentication. At the time of this writing, `DeviceOwnerAuthenticationWithBiometrics` (i.e. Touch ID) is the only policy. If the method returns a positive result, this means the device supports Touch ID authentication. On the other hand, if a negative result is returned, the device may not have the Touch ID sensor, or the user has not enabled the fingerprint authentication feature.
- `evaluatePolicy(_:localizedReason:reply:)` - when this method is invoked, it presents an authentication dialog to the user, requesting a finger scan. The authentication is performed asynchronously. When it finishes, the reply block will be called along with the authentication result. If the authentication fails, it returns you a `LAError` object indicating the reason of the failure.

Enough of the theories. It is time for us to work on a demo app. In the process, you will fully understand how to use Local Authentication framework.

Touch ID Demo App

To begin with, you can download the project template from <https://www.dropbox.com/s/e44lhb5wjsiz6in/TouchIDStart.zip?dl=0>. If you run the project, you will have a demo app displaying a login dialog. What we are going to do is replace the password authentication with Touch ID. If the authentication is successful, the app will show you the Home screen. However, the fingerprint authentication is not always successful. The user may use a device without Touch ID support, or has disabled the feature. In some cases, the user may opt for password authentication, or simply cancel the authentication prompt. It is crucial that you handle all these exceptional cases. So this is how the app works:

1. When it is first launched, the app presents a Touch ID dialog and requests a finger scan.
2. For whatever reasons the authentication fails or the user chooses to use a password, the app will display a login dialog and falls back to password-based authentication.
3. When the authentication is successful, the app will display the Home screen.



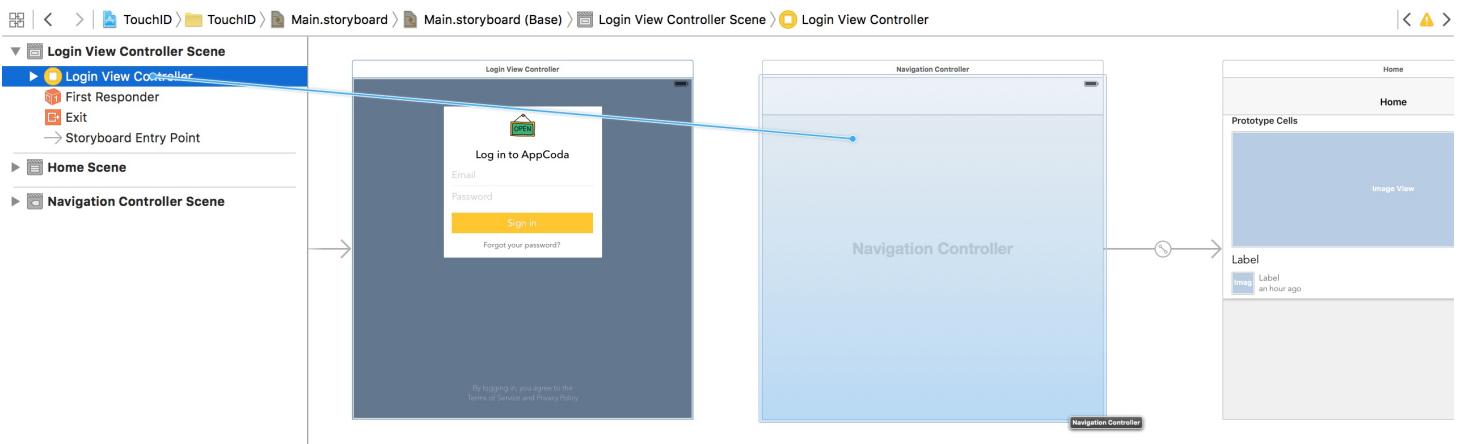
The project template is very similar to the demo app we built in the previous chapter. I just

added a new method named `showLoginDialog` in `LoginViewController.swift` to create a simple slide-down animation for the dialog.

Okay, let's get started building the demo app.

Designing the User Interface

The project template comes with a prebuilt storyboard that includes the login view controller and the table view controller of the Home screen; however, there is no connection between them. The very first thing we have to do is connect the login view controller with the navigation controller of the Home screen using a segue. Control-drag from the Login View Controller to the Navigation Controller. When prompted, select `present modally` for the segue option.



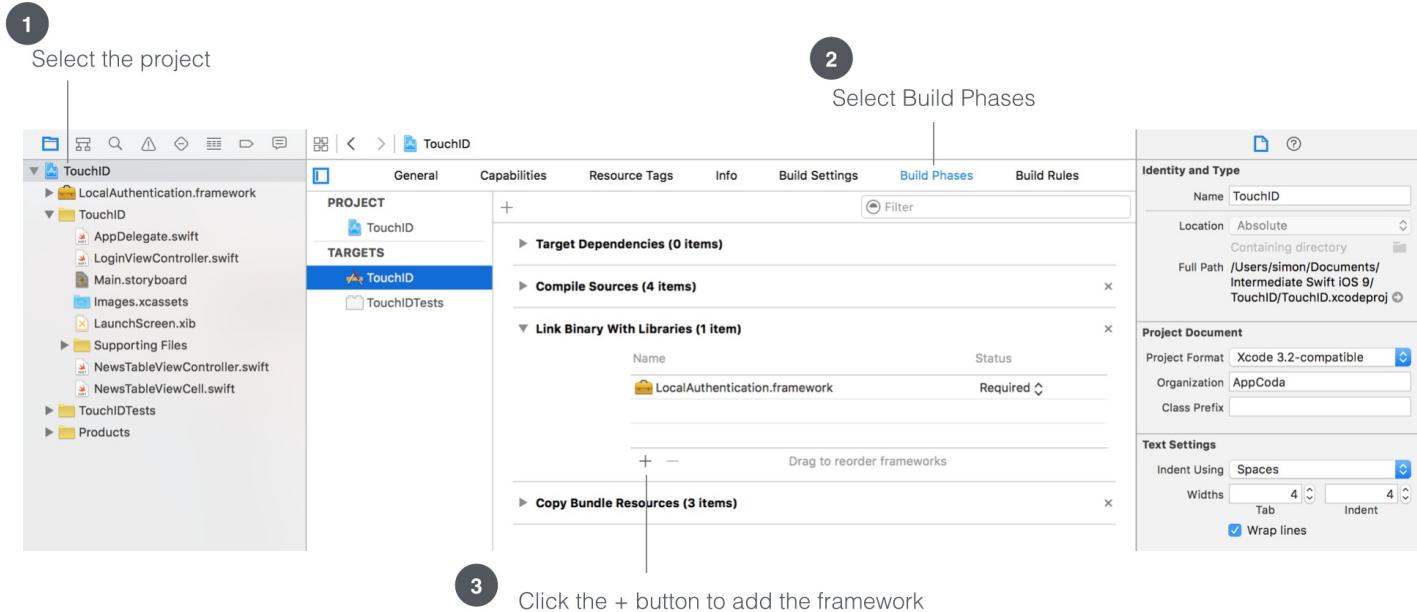
Typically, we use the default transition (i.e. slide-up). This time, let's change it to `cross dissolve`. Select the segue and go to the Attributes inspector. Change the transition option from Default to `Cross Dissolve`. Also, set the identifier to `showHomeScreen`. Later, we will perform the segue programmatically.

Using Local Authentication Framework

As I mentioned at the beginning of the chapter, the use of Touch ID is based on the Local Authentication framework, but for the time being, this framework doesn't exist in our project. We must manually add it to the project first.

In the Project Navigator, click on the TouchID Project and then select the Build Phases tab on the right side of the project window. Next, click on the disclosure icon of the Link Binary with

Libraries to expand the section and then click on the small plus icon. When prompted, search for the Local Authentication framework and add it to the project.



To use the framework, all you need is to import it using the following statement:

```
import LocalAuthentication
```

Open the `LoginViewController.swift` file and insert the above statement at the very beginning. Next, replace the following statement in the `viewDidLoad` method:

```
showLoginDialog()
```

with:

```
loginView.hidden = true
```

Normally, the app displays a login dialog when it is launched. Since we are going to replace the password-based authentication with Touch ID, the login view is hidden by default.

To implement Touch ID, we will create a new method called `authenticateWithTouchID` in the class. Let's start with the code snippet:

```
func authenticateWithTouchID() {
    // Get the local authentication context.
    let localAuthContext = LAContext()
```

```

let reasonText = "Authentication is required to sign in AppCoda"
var authError: NSError?

if
!localAuthContext.canEvaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiomet
error: &authError) {

    println(authError?.localizedDescription)

    // Display the login dialog when Touch ID is not available (e.g. in
simulator)
    showLoginDialog()

    return
}

}

```

The core of the Local Authentication framework is the `LAContext` class. To use Touch ID, the very first thing is to instantiate a `LAContext` object. The next step is to ask the framework if the Touch ID authentication can be performed on the device by calling the `canEvaluatePolicy` method. As mentioned earlier, the framework now only supports the `DeviceOwnerAuthenticationWithBiometrics` policy. If the method returns a `true` value, this indicates the device is capable to use Touch ID and the user has enabled Touch ID as the authentication mechanism. If a `false` value is returned, that means you cannot use Touch ID to authenticate the user. In this case, you should provide an alternative authentication method. Here we just call up the `showLoginDialog` method to fallback to password authentication.

Once we've confirmed that the Touch ID is supported, we can proceed to perform the Touch ID authentication. Insert the following lines of code in the `authenticateWithTouchID` method:

```

// Perform the Touch ID authentication
localAuthContext.evaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics
localizedReason: reasonText, reply: { (success: Bool, error: NSError?) -> Void
in

    if success {
        print("Successfully authenticated")
        NSOperationQueue.mainQueue().addOperationWithBlock({
            self.performSegueWithIdentifier("showHomeScreen", sender: nil)
        })
    } else {

```

```

if let error = error {
    switch error.code {
        case LAError.AuthenticationFailed.rawValue:
            print("Authentication failed")
        case LAError.PasscodeNotSet.rawValue:
            print("Passcode not set")
        case LAError.SystemCancel.rawValue:
            print("Authentication was canceled by system")
        case LAError.UserCancel.rawValue:
            print("Authentication was canceled by the user")
        case LAError.TouchIDNotEnrolled.rawValue:
            print("Authentication could not start because Touch ID has no
enrolled fingers.")
        case LAError.TouchIDNotAvailable.rawValue:
            print("Authentication could not start because Touch ID is not
available.")
        case LAError.UserFallback.rawValue:
            print("User tapped the fallback button (Enter Password).")

        default:
            print(error.localizedDescription)
    }

    // Fallback to password authentication
    NSOperationQueue.mainQueue().addOperationWithBlock({
        self.showLoginDialog()
    })

}
}
}

```

The `evaluatePolicy` method of the local authentication context object handles all the heavy lifting of the user authentication. When `DeviceOwnerAuthenticationWithBiometrics` is specified as the policy, the method automatically presents a dialog, requesting a finger scan from the user. You can provide a reason text, which will be displayed in the sub-title of the authentication dialog. The method performs Touch ID authentication in an asynchronous manner. When it finishes, the reply block (i.e. closure in Swift) will be called with the authentication result and error passed as parameters. In the closure, we first check if the authentication is successful. If it is true, we simply call up the `performSegueWithIdentifier` method and navigate to the Home screen.

If the authentication fails, the error object will incorporate the reason for failure. You can use

the code property of the error object to reveal the possible cause, which includes:

- `AuthenticationFailed` - the authentication failed because the fingerprint does not match up with those enrolled.
- `UserCancel` - the user has canceled the authentication (e.g. by tapping the Cancel button in the dialog).
- `UserFallback` - the user chooses to use password authentication instead of Touch ID. In the authentication dialog, there is a button called *Enter Password*. When a user taps the button, this error code will be returned.
- `SystemCancel` - the authentication is canceled by the system. For example, if another application came to the foreground while the authentication dialog was up.
- `PasscodeNotSet` - the authentication failed because the passcode is not configured.
- `TouchIDNotAvailable` - Touch ID is not available on the device.
- `TouchIDNotEnrolled` - the user has not configured Touch ID yet.

In the implementation, we simply log the error to the console. And when any error occurs, the app will fallback to password authentication by calling `showLoginDialog()`:

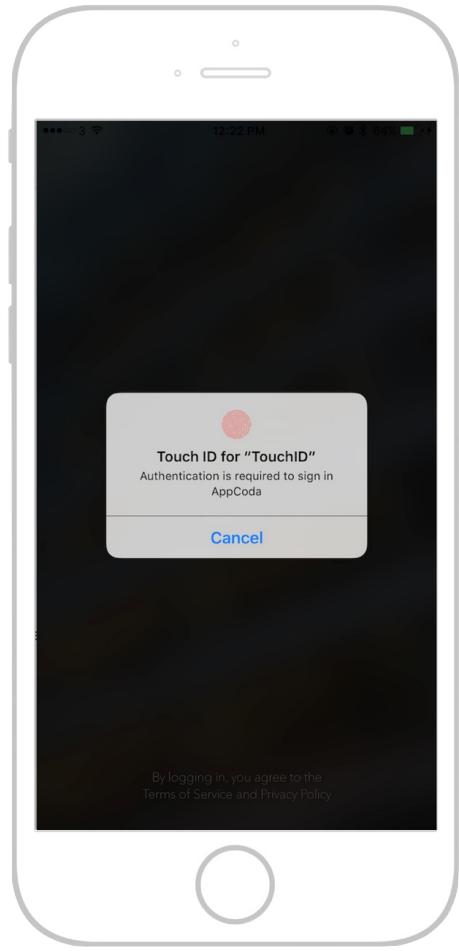
```
// Fallback to password authentication
NSOperationQueue.mainQueue().addOperationWithBlock({
    self.showLoginDialog()
})
```

Because the reply block is run in background, we have to explicitly perform the visual change in the main thread. This is why we execute the `showLoginDialog` method in the main thread to ensure a responsive UI update.

Lastly, insert the following line of code at the end of the `viewDidLoad` method to initiate the authentication:

```
authenticateWithTouchID()
```

Now you're ready to test the app. Make sure you run the app on a real device with Touch ID support (e.g. iPhone 6). Once launched, the app should ask for Touch ID authentication.



If the authentication is successful, you will be able to access the Home screen. If you run the app on the simulator, you should see the login dialog with the following error shown in the console:

```
Optional("No fingers are enrolled with Touch ID.")
```

Password Authentication

Now you have implemented the Touch ID authentication. However, when the user opts for password authentication, the login dialog is not fully functional yet. Let's create an action method called `authenticateWithPassword` :

```
@IBAction func authenticateWithPassword() {  
  
    if emailTextField.text == "hi@appcoda.com" && passwordTextField.text ==  
    "1234" {  
        performSegueWithIdentifier("showHomeScreen", sender: nil)  
    } else {
```

```

// Shake to indicate wrong login ID/password
loginView.transform = CGAffineTransformMakeTranslation(25, 0)
UIView.animateWithDuration(0.2, delay: 0.0, usingSpringWithDamping:
0.15, initialSpringVelocity: 0.3, options: .CurveEaseInOut, animations: {

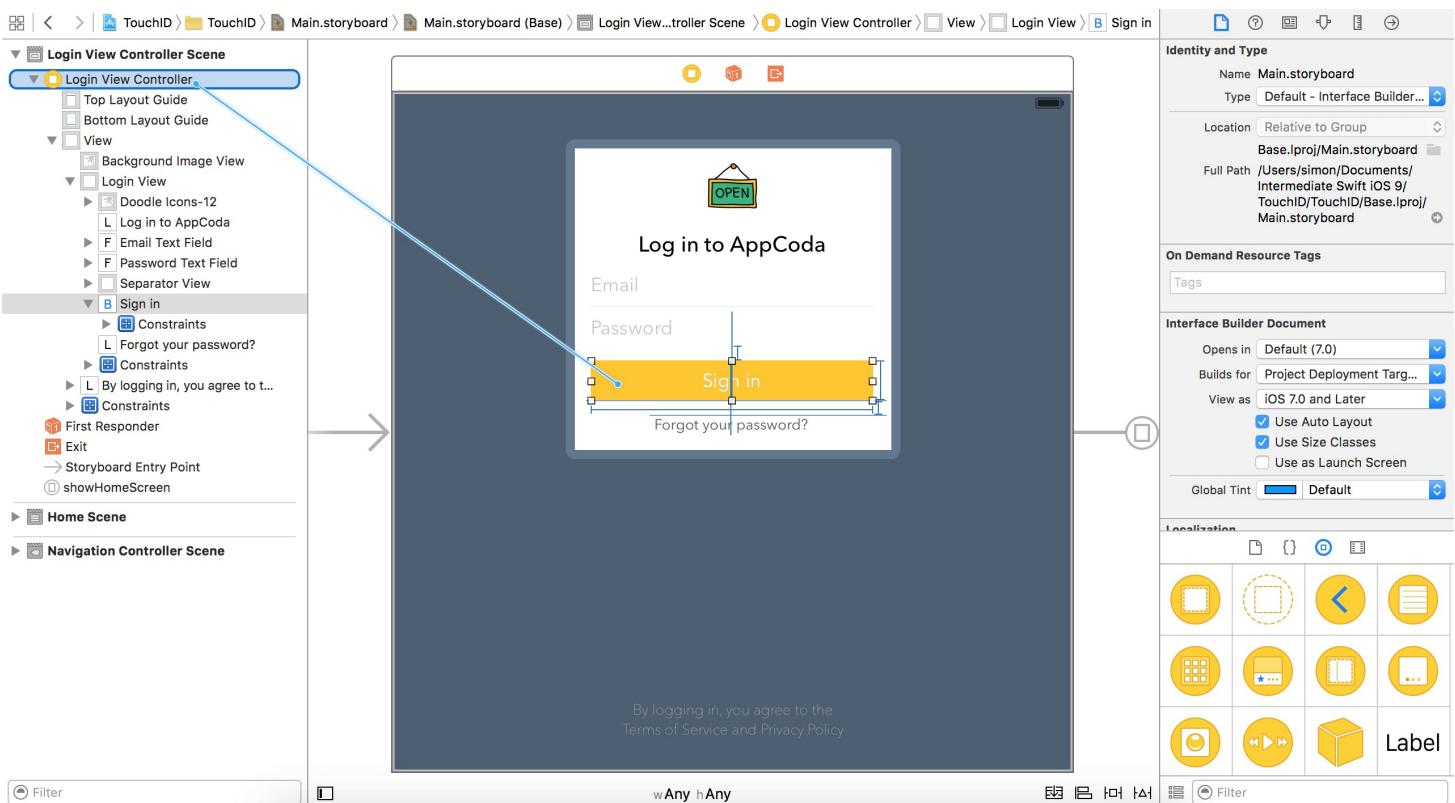
    self.loginView.transform = CGAffineTransformIdentity

}, completion: nil)
}
}

```

In reality, you may store the user profiles in your backend and authenticate the user using web service call. To keep things simple, we just hardcode the login ID and password to `hi@appcoda.com` and `1234` respectively. When the user enters a wrong combination of login ID and password, the dialog performs a "Shake" animation to indicate the error.

Now go back to the storyboard to connect the Sign In button with the method. Control-drag from the Sign In button to the Login View Controller and select `authenticateWithPassword` under Sent Events.



Build and run the project again. You should now be able to login the app even if you choose to fallback to the password authentication. Tapping the Sign In button without entering the

password will "shake" the login dialog.

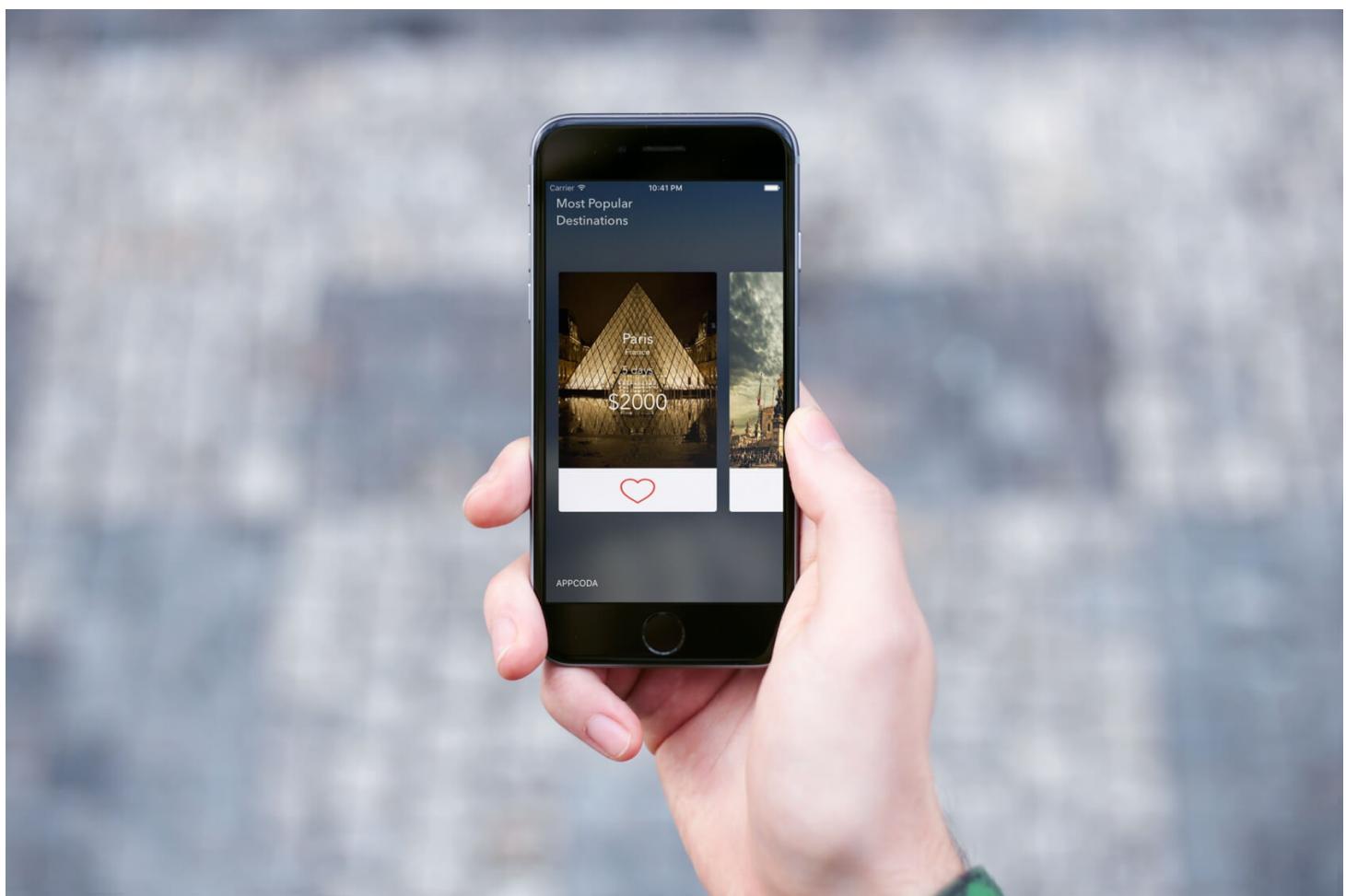
The introduction of Touch ID API in iOS 8 is huge. As Touch ID is more widespread, I expect more and more apps will use Touch ID as the authentication mechanism. Have you begun to integrate the feature in your app?

For reference, you can download the final project from

<https://www.dropbox.com/s/wp93yucct9nqaz5/TouchID.zip?dl=0>.

Chapter 29

Building a Carousel-Like User Interface



Have you used Kickstarter's iOS app? If not, [download it from App Store](#) and give it a shot. Kickstarter is one of my favorite crowdfunding services. Earlier this year, the company revamped its iOS app with a beautifully redesigned user interface. As soon as you open up the app, it displays the featured projects in a carousel, with which you can flick left or right through the cards to discover more Kickstarter projects. Themed with vivid colors, the carousel design of the app looks plain awesome.

Carousel is a popular way to showcase a variety of featured content. Not only can you find carousel design in mobile apps, but it has also been applied to web applications for many years. A carousel arranges a set of items horizontally, where each item usually includes a thumbnail.

Users can scroll through the list of items by flicking left or right.

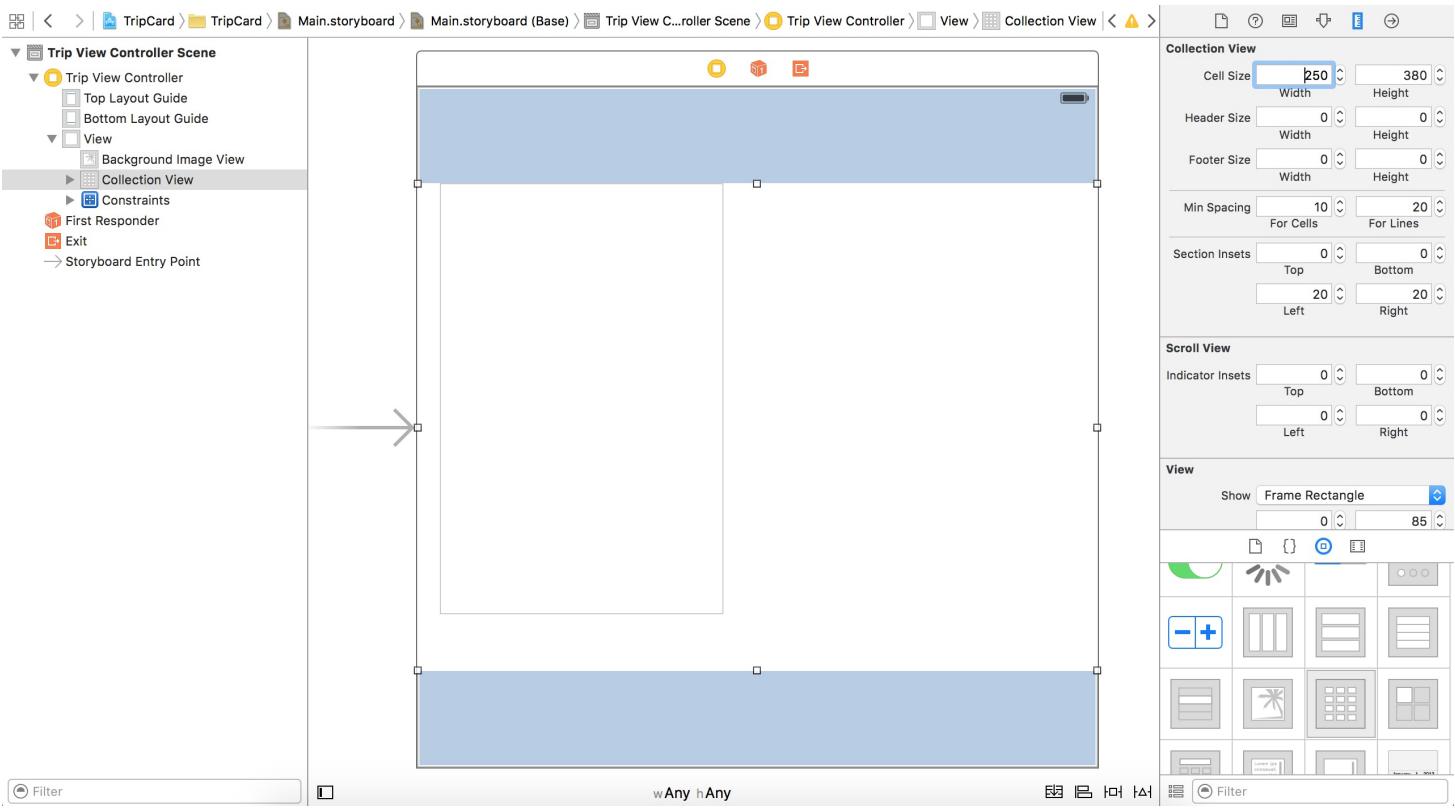


In this chapter, I will show you how to build a carousel in iOS apps. It's not as hard as you might think. All you need to do is to implement a `UICollectionView`. If you do not know how to create a collection view, I recommend you take a look at chapter 18. As usual, to walk you through the feature we will build a demo app with a simple carousel that displays a list of trips.

Designing the Storyboard

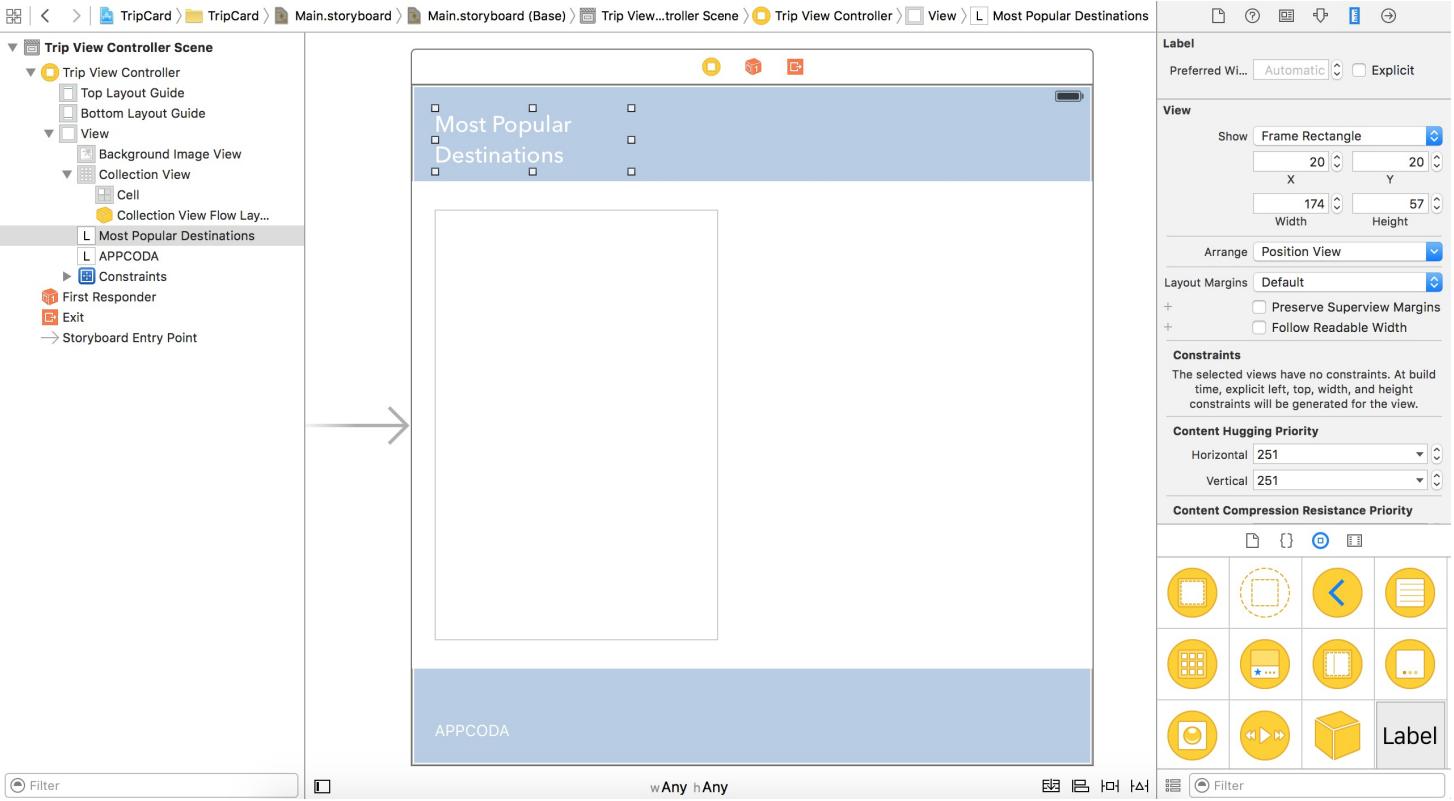
To begin with, you can first download the project template named *TripCard* from <https://www.dropbox.com/s/hj425zgq8tqcpeo/TripCardStart.zip?dl=0>. After the download, compile it and have a trial run of the project using the built-in simulator. You should have an app showing a blurred background (if you want to learn how to apply a blurring effect, check out chapter 27). The template already incorporates the necessary resources including images and icons. We will build upon the template by adding a collection view for it.

Okay, go to `Main.storyboard`. Drag a collection view from the Object library to the view controller. Resize its width to `600` points and height to `430` points. Place it at the center of the view controller. Next, go to the Size inspector. In the cell size option, set the width to `250` points and height to `380` points. Also change minimum spacing for lines to `20` points to add some spacing between cell items. Lastly, set the left and right values of section insets to `20` points.

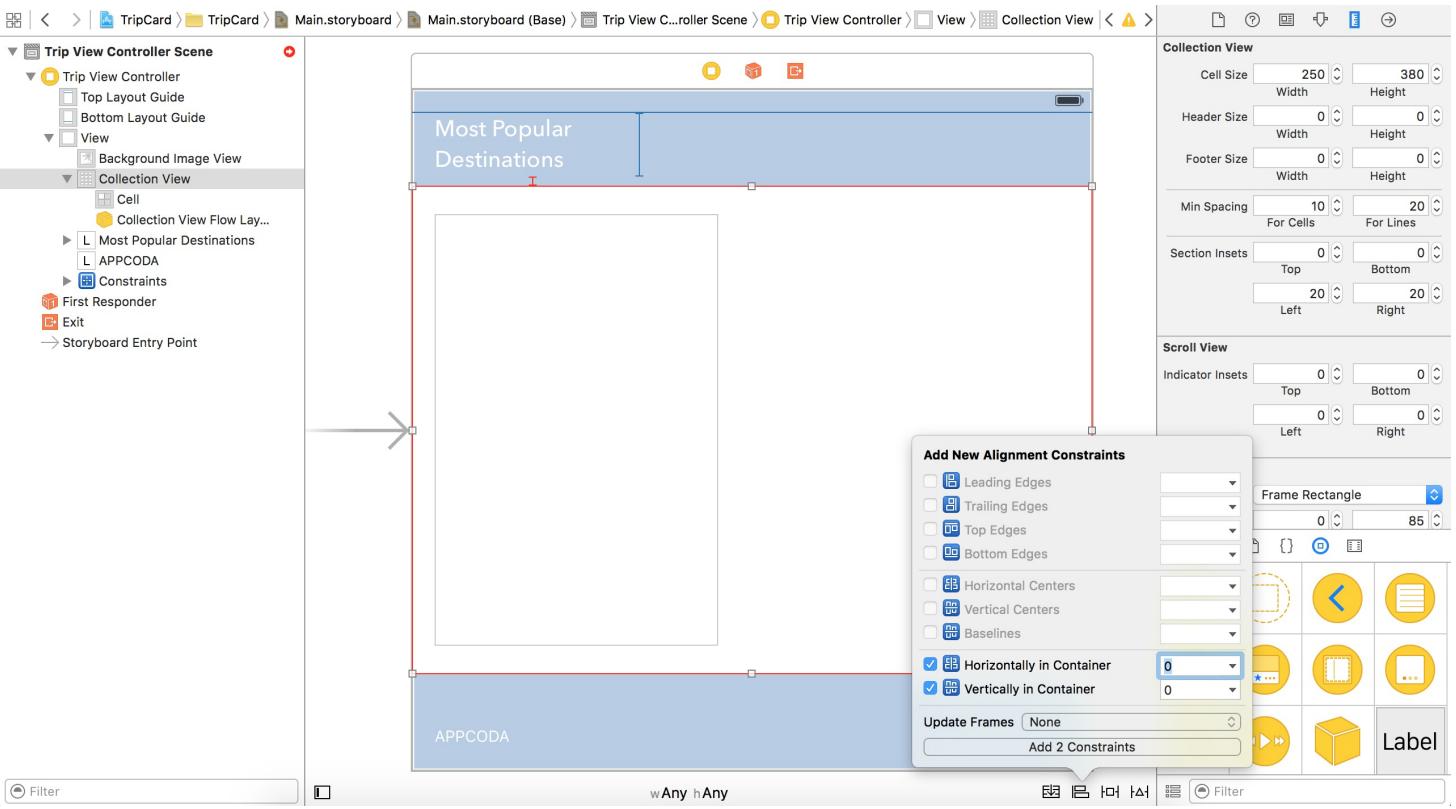


Your storyboard should now look similar to the screenshot above. Now select the collection view and go to the Attributes inspector. Change the scroll direction from `vertical` to `horizontal`. Once you have made this change, users will be able to scroll through the collection view horizontally instead of vertically. This is the real trick to building a carousel. Don't forget to set the identifier of the collection view cell to `cell`.

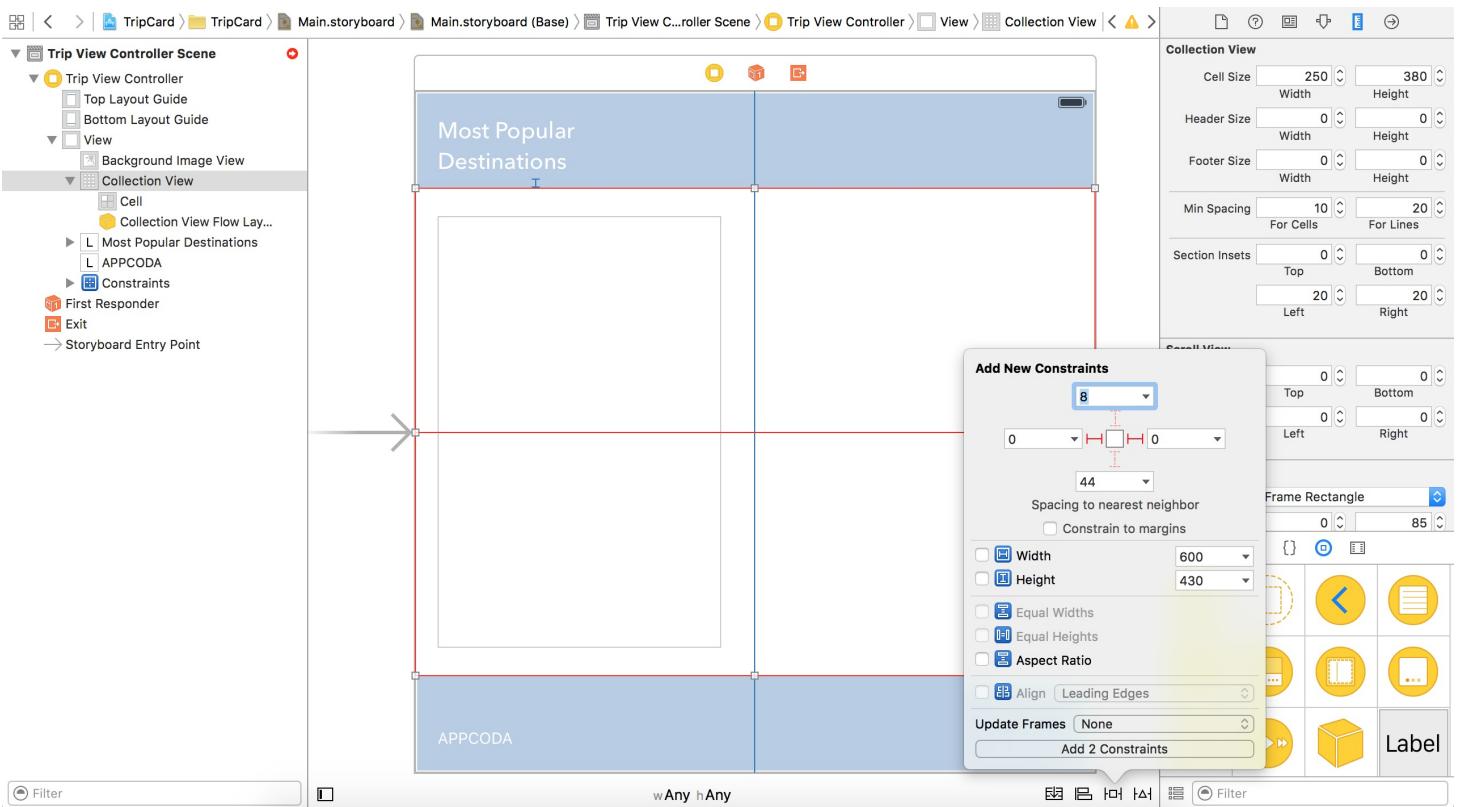
Next, drag a label to the view controller and place it at the top-left corner of the view. Set the text to `Most Popular Destinations` and the color to `white`. Change to your preferred font and size. Then, add another label to the view controller but put it below the view controller. Change its text to `APPCODA` or whatever you prefer. Your view controller will look similar to this:



So far we haven't configured any auto layout constraint. First, select the *Most Popular Destinations* label. Click the *Issues* button and select *Adding Missing Constraints*. Now let's add a few layout constraints for the collection view. Select the collection view and click the *Align* button of the auto layout menu. Check both the *Horizontal Center in Container* and *Vertical Center in Container* options, and click *Add 2 Constraints*. This will align the collection view to the center of the view.



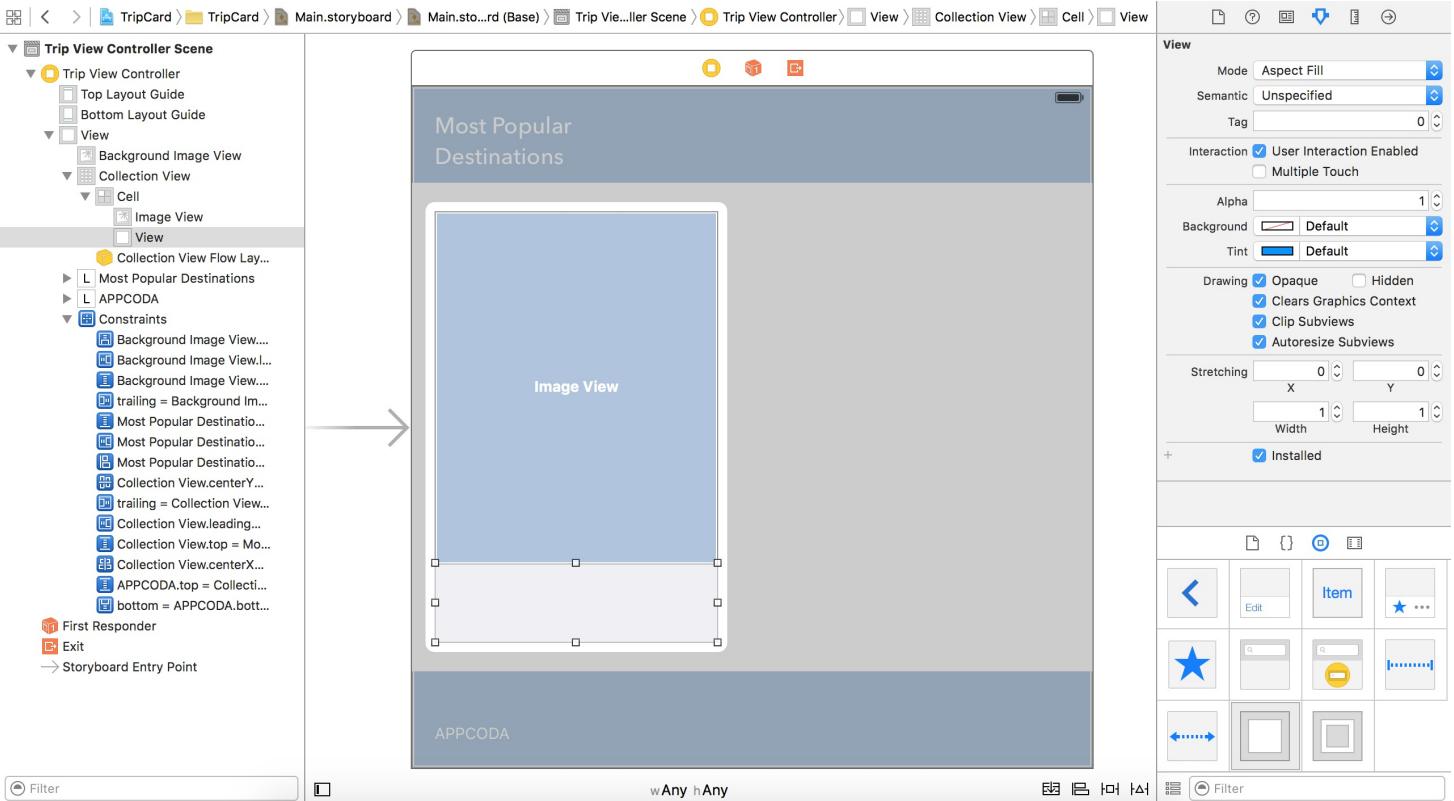
Xcode should indicate some missing constraints. Click the *Pin* button and select the dashed red line corresponding to the left and right sides. Uncheck the *Constrain to margins* option and click *Add 2 Constraints*. This ensures that the left and right sides of the collection view align perfectly with the background image view.



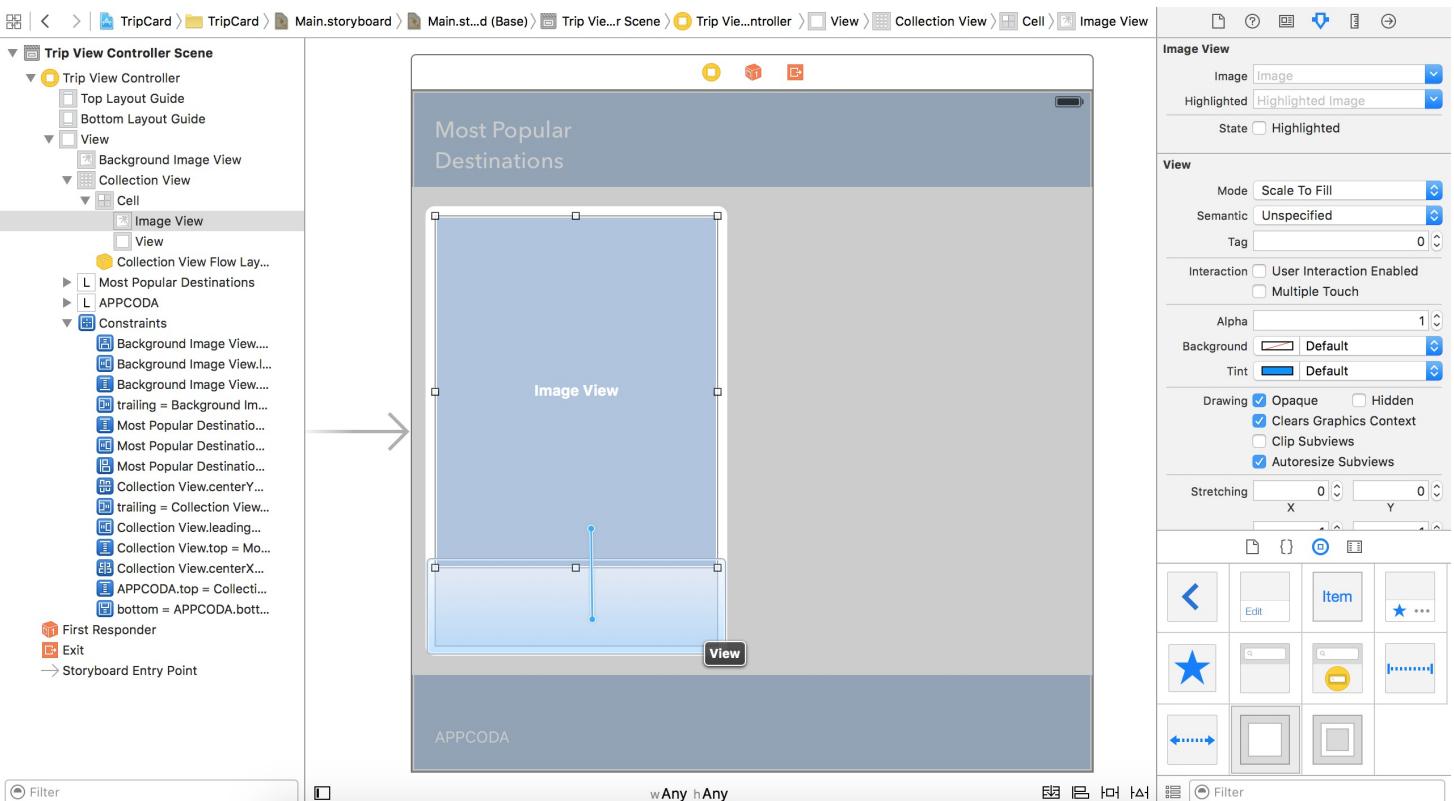
Lastly, let's add the layout constraints for the *APPCODA* label. Select the label and click the *Issues* button. Select the *Add Missing Constraints* option and let Xcode configure the layout constraints for you. Now that you have created the skeleton of the collection view, let's configure the cell content, which will be used to display trip information. First, select the cell and change its background to `light gray`. Then drag an image view to the cell and change its size to `250x311` points.

Next, drag a view from the Object Library and place it right below the image view. In the Attributes inspector, change its background color to Default, set the mode to `Aspect Fill` and enable the `Clip Subviews` option. This view serves as a container to hold other UI elements. Sometimes it is good to use a view to group multiple UI elements together so that it is easier for you to define the layout constraints later.

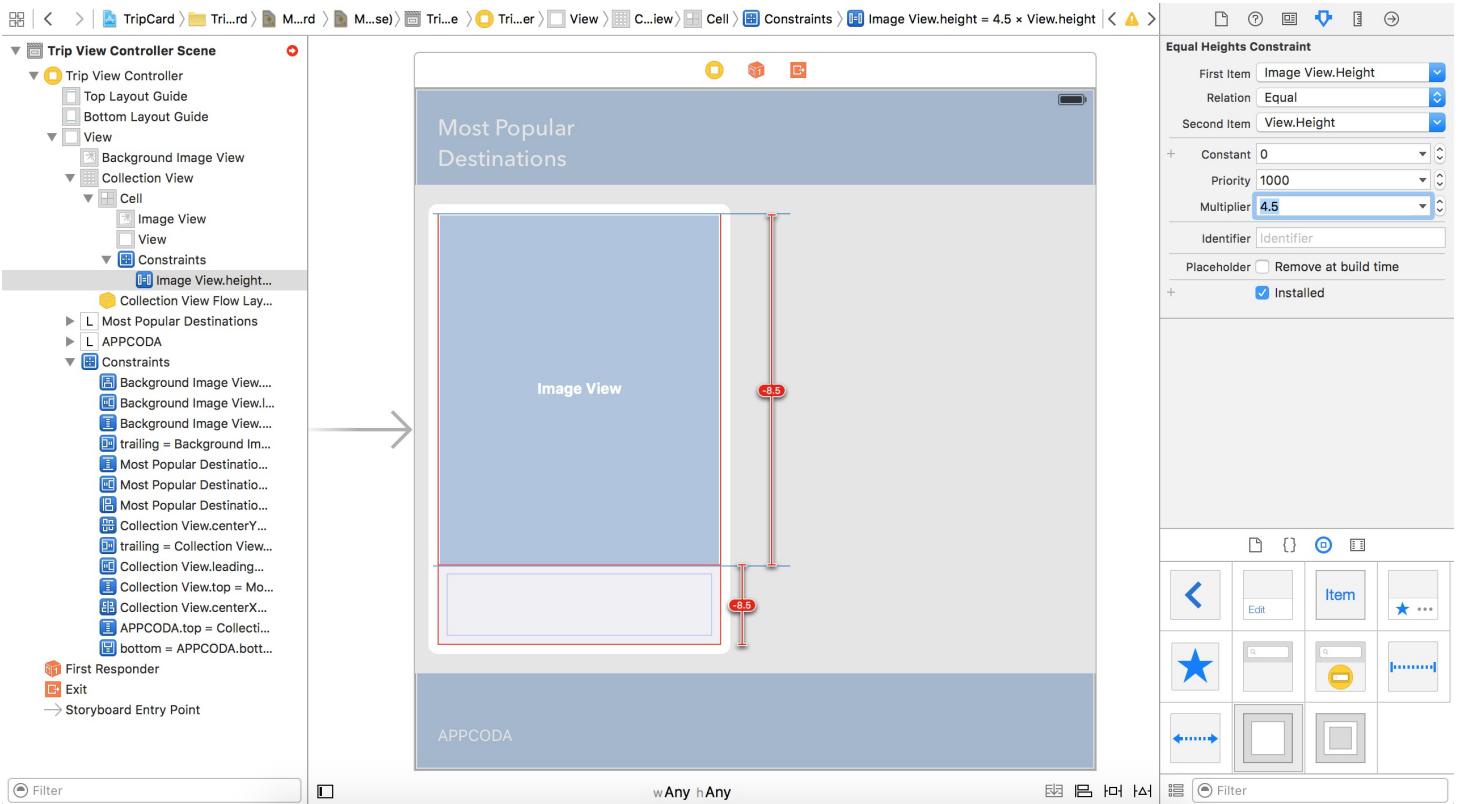
If you follow the procedures correctly, your storyboard should look similar to this:



Later we will change the size of the collection view according to the screen height. But I still want to keep the height of the image view and the view inside the cell proportional. To do that, control-drag from the image view to the view and select Equal Heights.

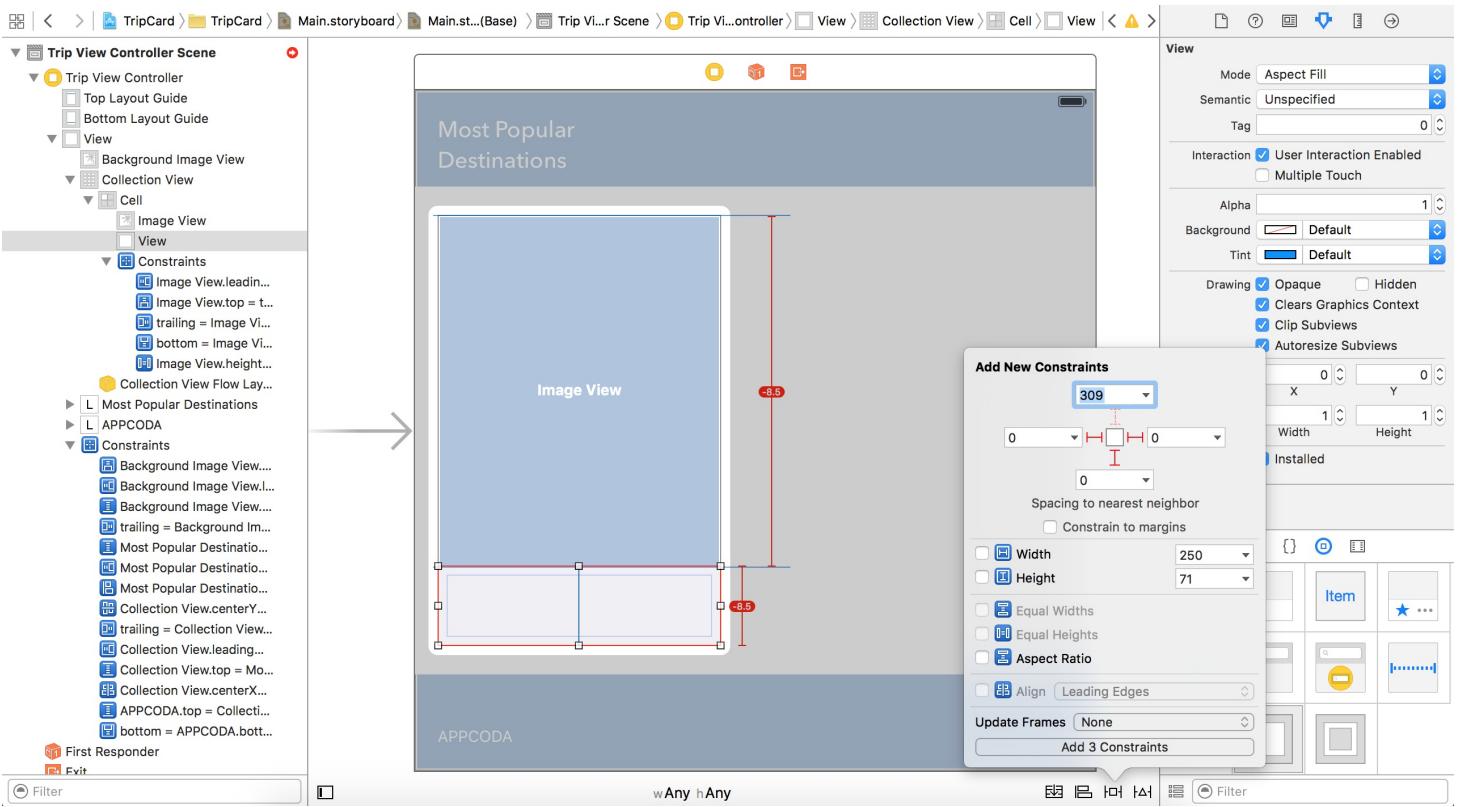


Next, select the constraint just created and go to the Size inspector. Change the multiplier from `1` to `4.5`. Make sure the first and second items are set to `Image View.height` and `View.height` respectively. This defines a constraint so that the height of the image view is always 4.5 times taller than the view.



Now select the image view and define the spacing constraints. Click the *Pin* button and select the dashed red lines of all sides. Click the *Add 4 Constraints* button to define the layout constraints.

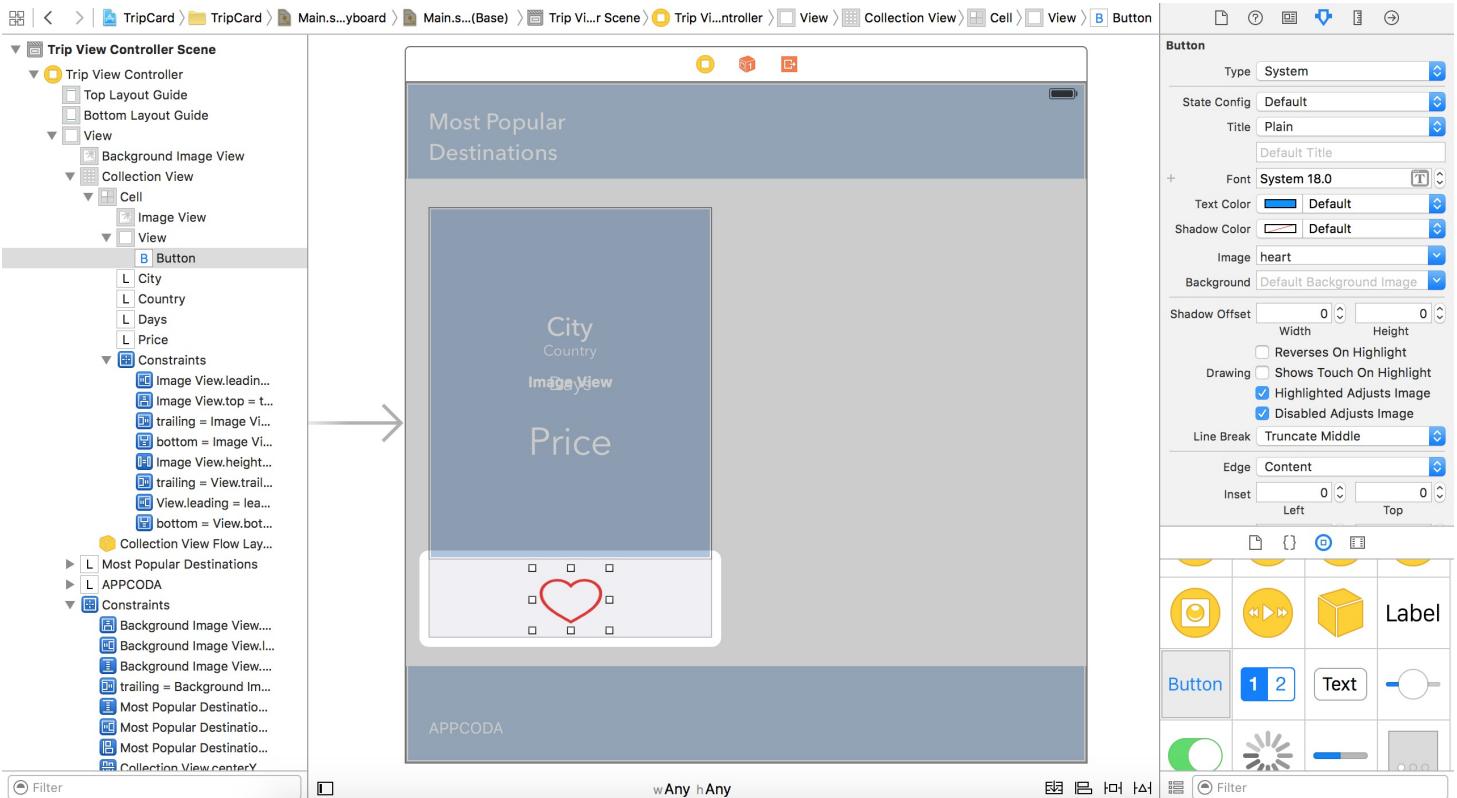
Select the view inside the collection view cell and click the *Pin* button. Click the dashed red lines that correspond to the left, right and bottom sides.



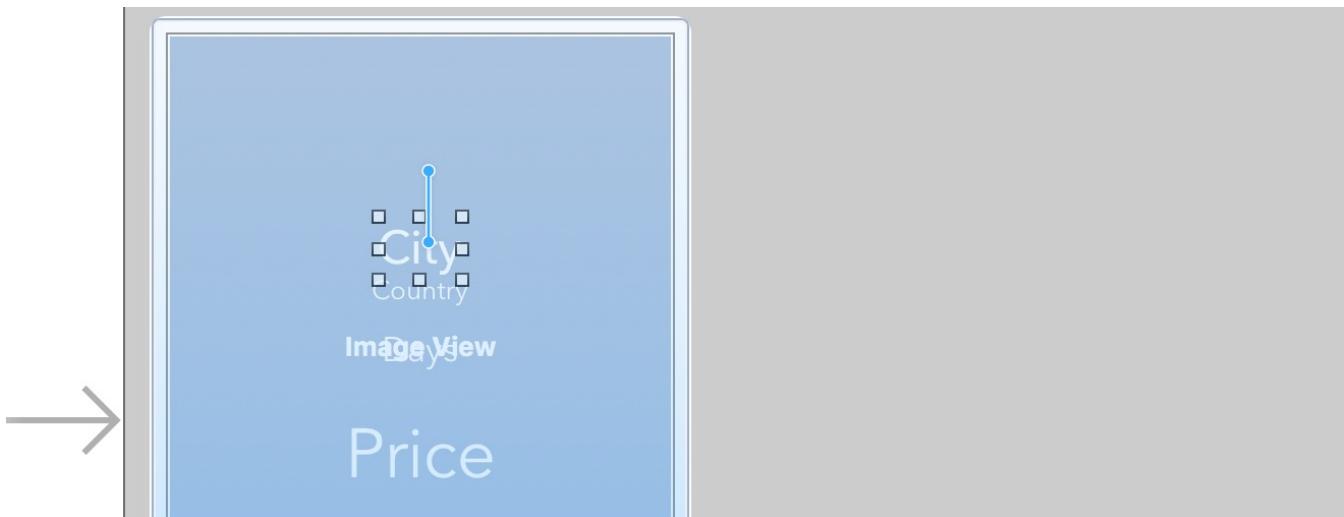
If the document outline displays the layout issue indication, simply click on it and follow the procedures to fix the layout issue.

Okay, you have defined the layout constraints for these two views. It's now time to add some UI elements to the image view for displaying the trip information.

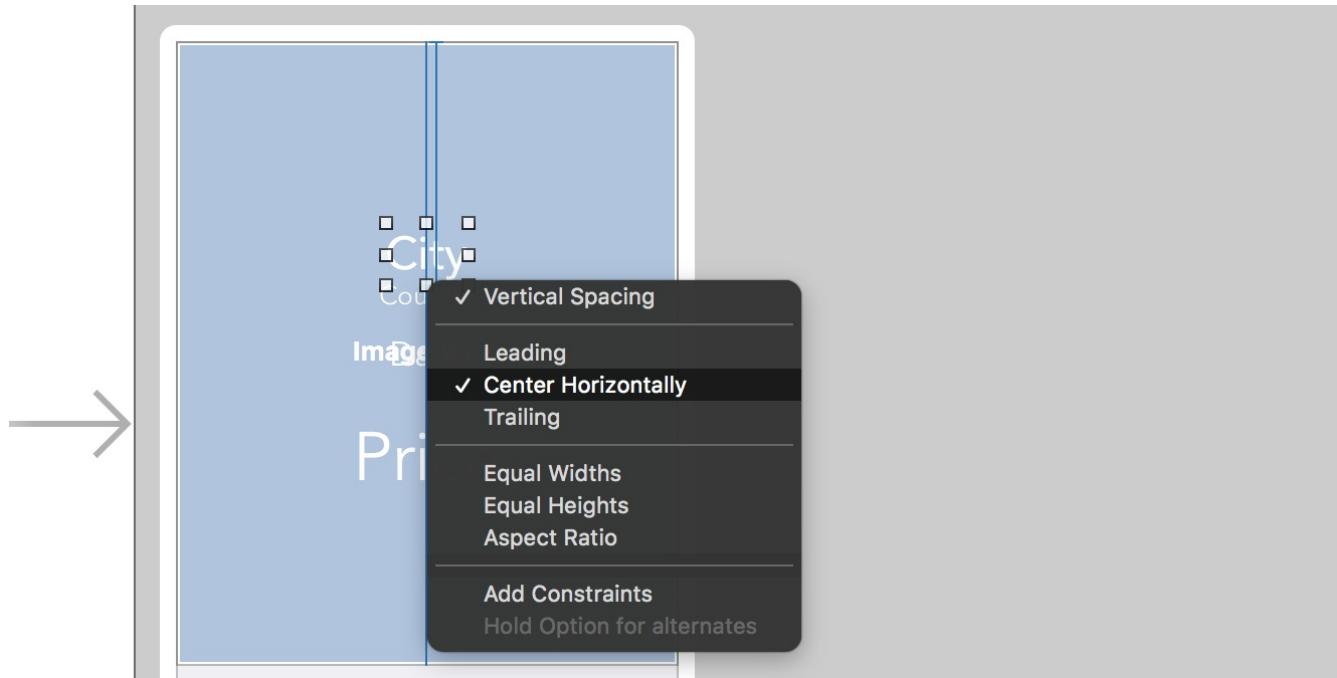
- First, add a label to the image view of the cell. Name it *City* and change its color to `white`. You may change its font and size.
- Second, drag another label to the image view. Name it *Country* and set the color to `white`. Again, change its font to whatever you like
- Next, add another label to the image view. Name it *Days* and set the color to `white`. Change the font to whatever you like (e.g. Avenir Next), but make it larger than the other two labels.
- Drag another label to the image view. Name it *Price* and set the color to `white`. Change its size such that it is larger than the rest of the labels.
- Finally, add a button object to the view (below the image view) and place it at the center of the view. In the Attributes inspector, change its title to blank and set the image to heart. Also change its type to System and tint color to `red`. In the Size inspector, set its width to `69` points and height to `56` points.



The UI design is almost complete. We simply need to add a few layout constraints for the elements we just added. First, control-drag from the *City* label to the image view of the cell.



In the popover menu, select both *Vertical Spacing* and *Center Horizontally* (simply hold the shift key to select multiple options). Next, control-drag from the *Country* label to the *City* label. Release the buttons and select both the *Vertical Spacing* and *Center Horizontally* options.

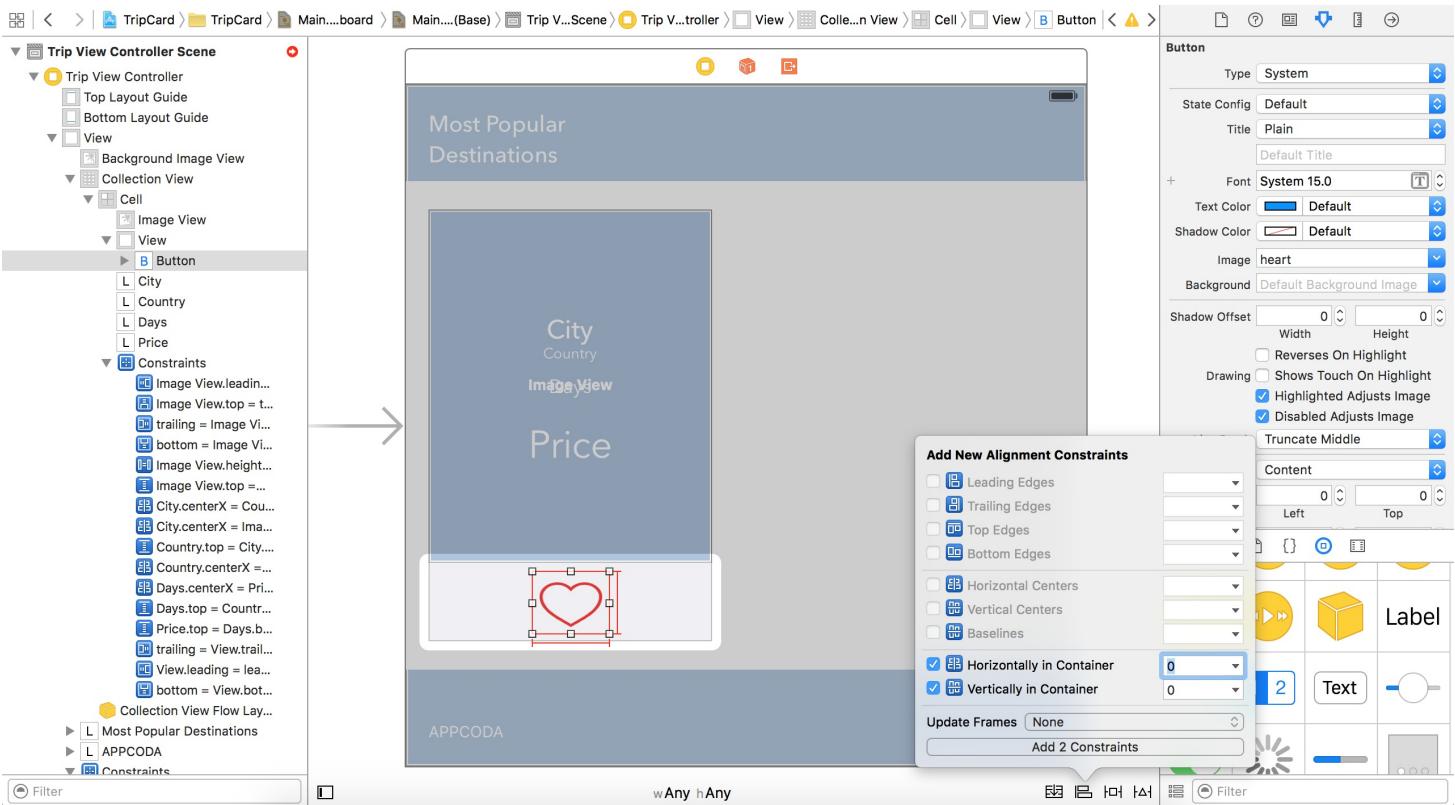


Then, control-drag from the *Days* label to the *Country* label. Repeat the procedure and set the same set of constraints. Lastly, control-drag from the *Price* label to the *Days* label and define the same layout constraints.

For the heart button, I want it to be a fixed size. Control-drag to the right (see below) and set the *Width* constraint. Next, control-drag vertically to set the *Height* constraint for the button.



To ensure the heart button is always displayed at the center of the view, click the *Align* button and select *Horizontal Center in Container* and *Vertical Center in Container*.



Great! You have completed the UI design. Now we will move onto the coding part.

Creating a Custom Class for the Collection View Cell

As the collection view cell is customized, we will first create a custom class for it. In the Project Navigator, right-click the *TripCard* folder and select *New File...*. Choose the Cocoa Touch Class template and proceed. Name the class `TripCollectionCell` and set it as a subclass of `UICollectionViewCell`. Once the class is created, open up `TripCollectionCell.swift` and update the code to the following:

```
class TripCollectionCell: UICollectionViewCell {
    @IBOutlet var imageView:UIImageView!
    @IBOutlet var cityLabel:UILabel!
    @IBOutlet var countryLabel:UILabel!
    @IBOutlet var totalDaysLabel:UILabel!
    @IBOutlet var priceLabel:UILabel!
    @IBOutlet var likeButton:UIButton!

    var isLiked:Bool = false
        didSet {
            if isLiked {
                likeButton.setImage(UIImage(named: "heartfull"), forState:
.Normal)
```

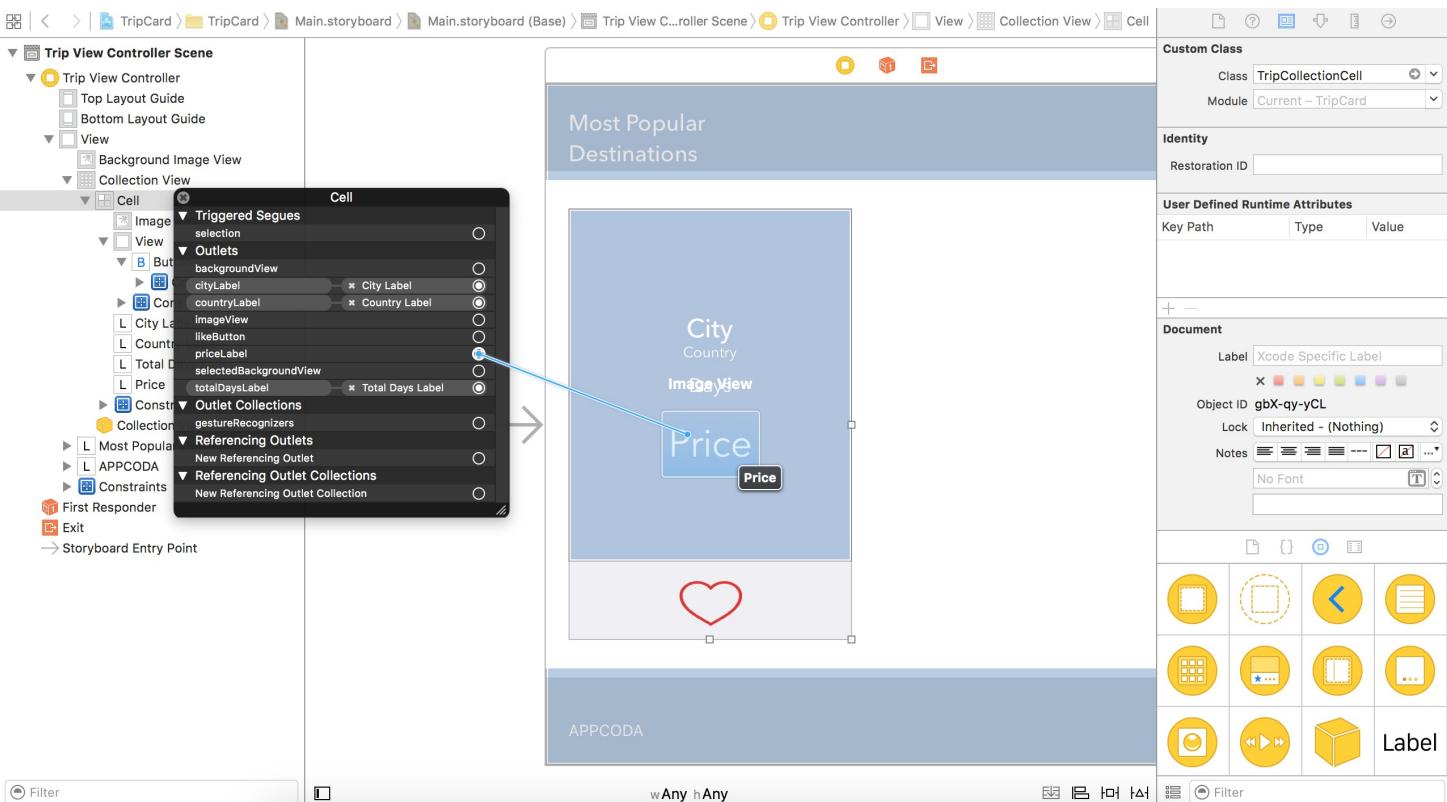
```

        } else {
            likeButton.setImage(UIImage(named: "heart"), forState: .Normal)
        }
    }
}

```

The above lines of code should be very familiar to you. We simply define the outlet variables to associate with the labels, image view and button of the collection view cell in storyboard. The `isLiked` variable is a boolean to indicate whether a user favors a trip or not. In the above code, we declare a `didSet` observer for the `isLiked` property. If this is the first time you have heard of property observer, it is a great feature of Swift. When the `isLiked` property is stored, the `didSet` observer will be called immediately. Here we simply set the image of the like button according to the value of `isLiked`.

Now go back to the storyboard and select the collection view cell. In the Identity inspector, set the custom class to `TripCollectionCell`. Right click the Cell in Document Outline. Connect each of the outlet variables to the corresponding visual element.



Creating the Model Class

Before we implement the `TripViewController` class to populate the data, we will create a model class named `Trip` to represent a trip. Create a new file using the *Swift File template* and name the class `Trip`. Proceed to create and save the `Trip.swift` file. Open `Trip.swift` and update the code to the following:

```
import Foundation
import UIKit

class Trip {
    var tripId = ""
    var city = ""
    var country = ""
    var featuredImage: UIImage?
    var price: Int = 0
    var totalDays: Int = 0
    var isLiked = false

    init(tripId: String, city: String, country: String, featuredImage: UIImage!, price: Int, totalDays: Int, isLiked: Bool) {
        self.tripId = tripId
        self.city = city
        self.country = country
        self.featuredImage = featuredImage
        self.price = price
        self.totalDays = totalDays
        self.isLiked = isLiked
    }
}
```

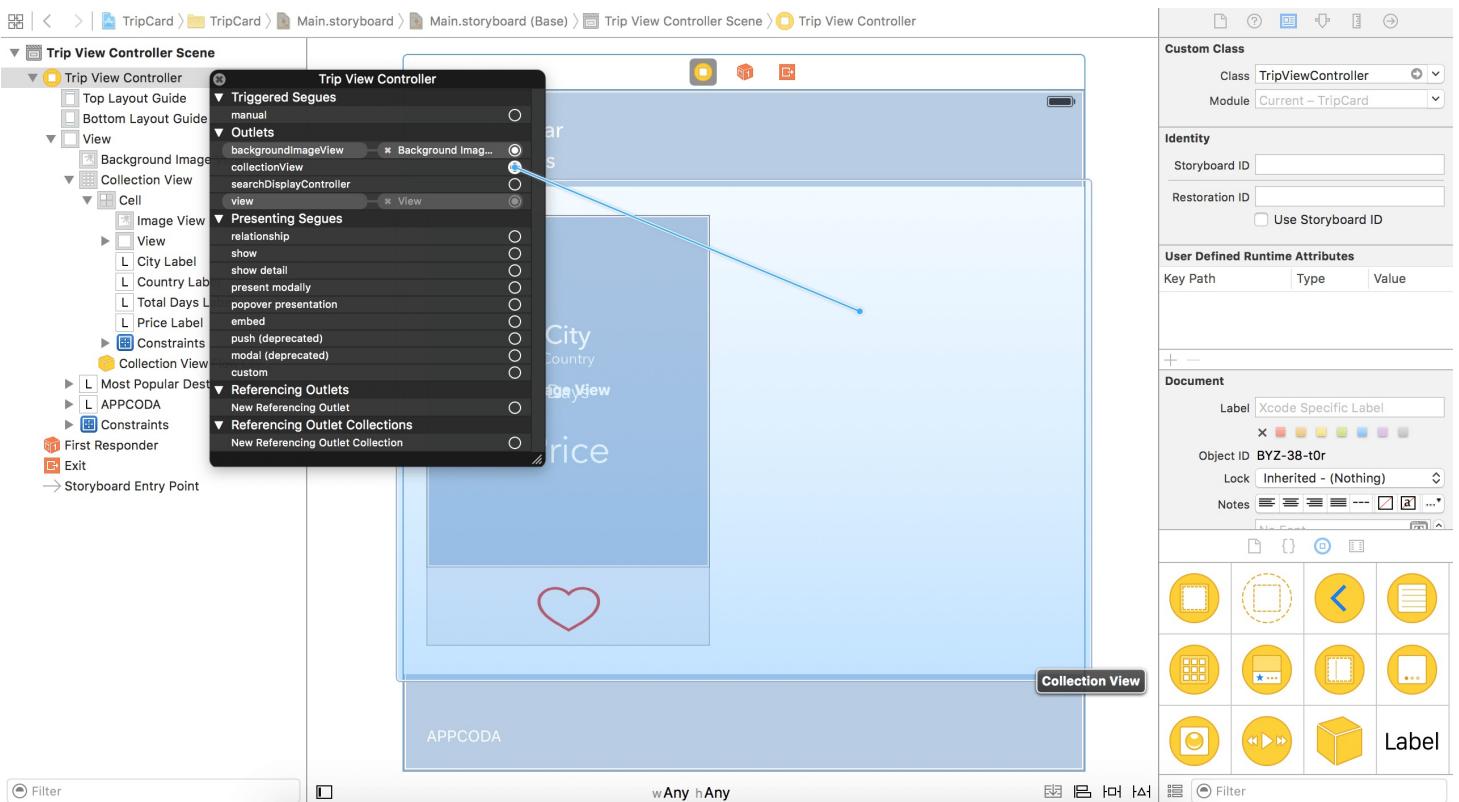
The `Trip` class contains a few properties for holding the trip data including *ID*, *city*, *country*, *featured image*, *price*, *total number of days* and *isLiked*. Other than the *ID* and *isLiked* properties, the rest of the properties are self-explanatory. Regarding the trip ID property, it is used for holding a unique ID of a trip. `isLiked` is a boolean variable that indicates whether a user favors the trip.

Populating the Collection View

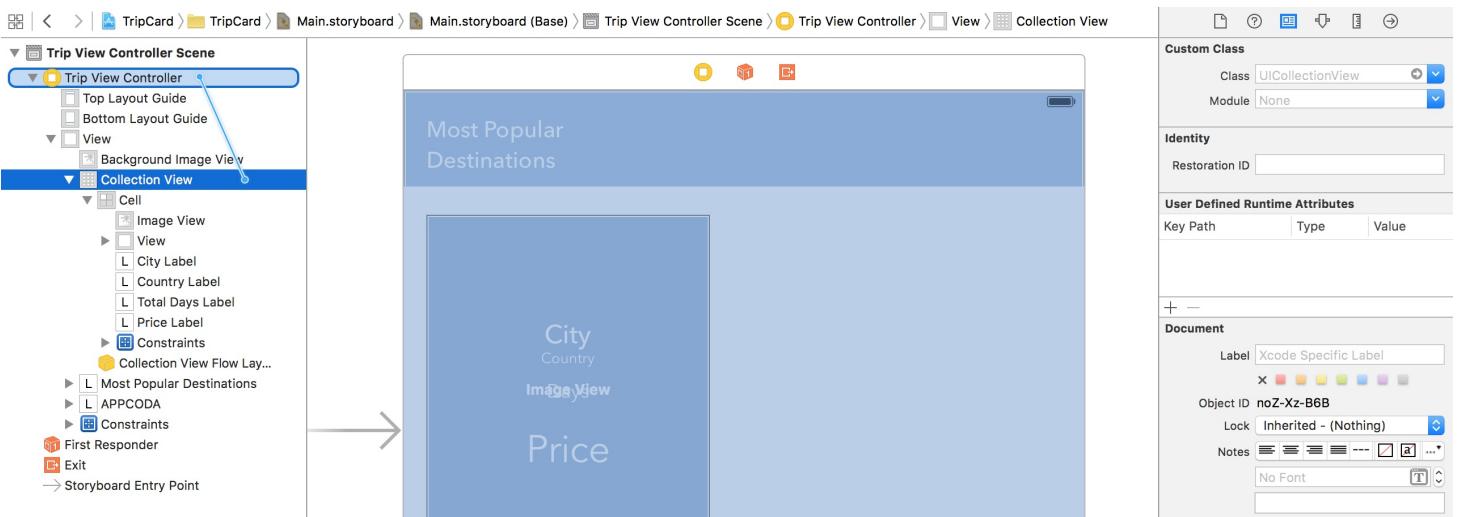
Now we are ready to populate the collection view with some trip data. First, declare an outlet variable for the collection view in `TripViewController.swift`:

```
@IBOutlet var collectionView: UICollectionView!
```

Go to the storyboard. In the Document Outline, right click Trip View Controller. Connect collectionView outlet variable with the collection view.



Furthermore, control-drag from collection view to trip view controller to connect the data source and delegate.



To keep things simple, we will just put the trip data into an array. Declare the following variable in `TripViewController.swift`:

```

private var trips = [Trip(tripId: "Paris001", city: "Paris", country: "France",
featuredImage: UIImage(named: "paris"), price: 2000, totalDays: 5, isLiked:
false),
    Trip(tripId: "Rome001", city: "Rome", country: "Italy", featuredImage:
UIImage(named: "rome"), price: 800, totalDays: 3, isLiked: false),
    Trip(tripId: "Istanbul001", city: "Istanbul", country: "Turkey",
featuredImage: UIImage(named: "istanbul"), price: 2200, totalDays: 10, isLiked:
false),
    Trip(tripId: "London001", city: "London", country: "United Kingdom",
featuredImage: UIImage(named: "london"), price: 3000, totalDays: 4, isLiked:
false),
    Trip(tripId: "Sydney001", city: "Sydney", country: "Australia",
featuredImage: UIImage(named: "sydney"), price: 2500, totalDays: 8, isLiked:
false),
    Trip(tripId: "Santorini001", city: "Santorini", country: "Greece",
featuredImage: UIImage(named: "santorini"), price: 1800, totalDays: 7, isLiked:
false),
    Trip(tripId: "NewYork001", city: "New York", country: "United States",
featuredImage: UIImage(named: "newyork"), price: 900, totalDays: 3, isLiked:
false),
    Trip(tripId: "Kyoto001", city: "Kyoto", country: "Japan", featuredImage:
UIImage(named: "kyoto"), price: 1000, totalDays: 5, isLiked: false)
]

```

To manage the data in the collection view, the `TripViewController` class has to adopt the `UICollectionViewDelegate` and `UICollectionViewDataSource` protocols. So change the class declaration like this:

```

class TripViewController: UIViewController, UICollectionViewDelegate,
UICollectionViewDataSource

```

Next, implement the required methods of the protocols:

```

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

func collectionView(collectionView: UICollectionView, numberOfItemsInSection
section: Int) -> Int {
    return trips.count
}

func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath
indexPath: NSIndexPath) -> UICollectionViewCell {

```

```
let cell = collectionView.dequeueReusableCell(withIdentifier: "Cell",
forIndexPath: indexPath) as! TripCollectionCell

// Configure the cell
cell.cityLabel.text = trips[indexPath.row].city
cell.countryLabel.text = trips[indexPath.row].country
cell.imageView.image = trips[indexPath.row].featuredImage
cell.priceLabel.text = "$\$(String(trips[indexPath.row].price))"
cell.totalDaysLabel.text = "\$(trips[indexPath.row].totalDays) days"
cell.isLiked = trips[indexPath.row].isLiked

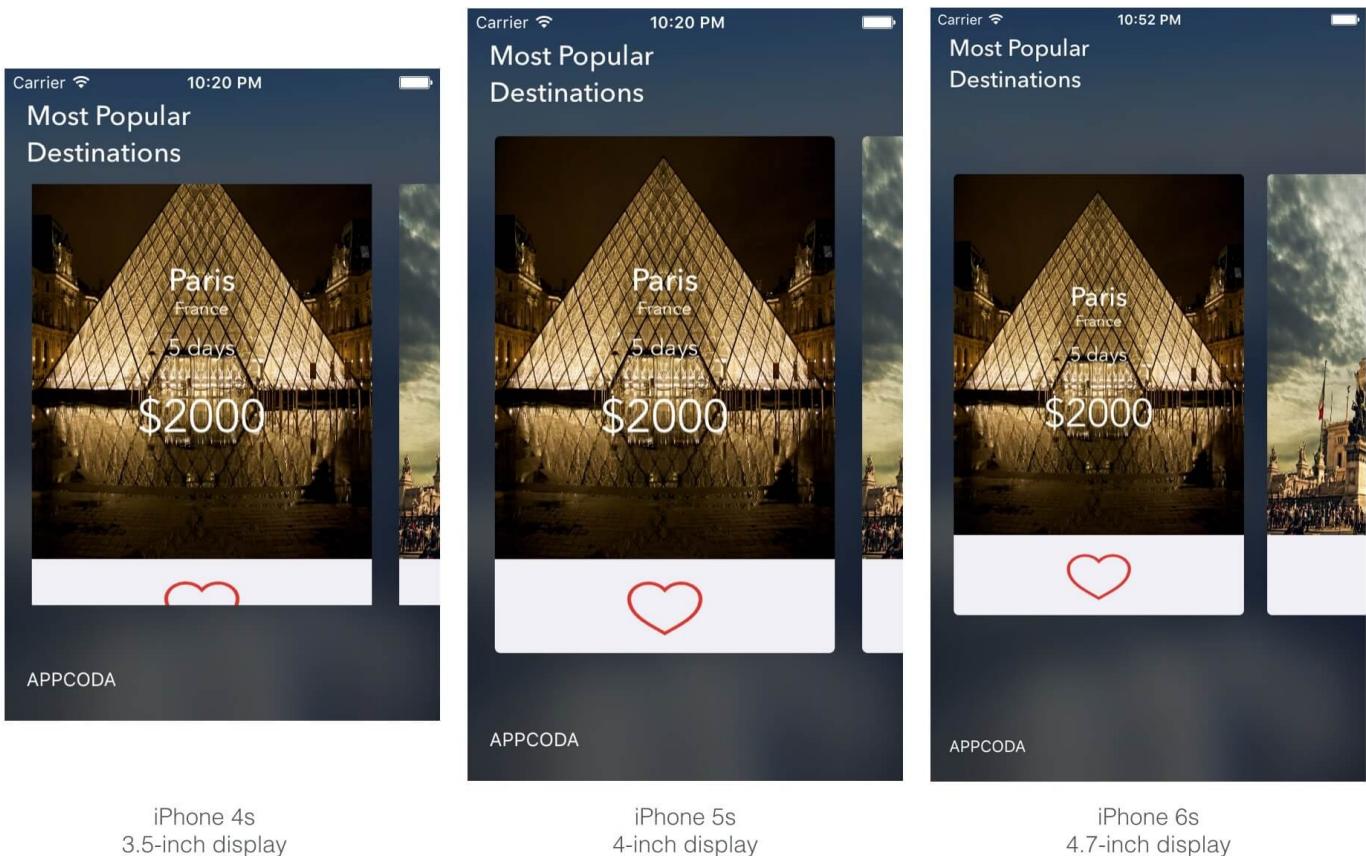
// Apply round corner
cell.layer.cornerRadius = 4.0

return cell
}
```

I will not go into the details of the implementation as you should be very familiar with the methods. Finally, insert this line of code in the `viewDidLoad` method to make the collection view transparent:

```
collectionView.backgroundColor = UIColor.clearColor()
```

Now it's time to test the app. Hit the Run button, and you should have a carousel showing a list of trips. The app works properly on devices with at least 4-inch display. If you run the app on iPhone 4s, however, parts of the collection view are blocked.



We have to reduce the height of the collection view specifically for 3.5-inch devices. Insert the following block of code in the `viewDidLoad` method of `TripViewController.swift`:

```
if UIScreen.mainScreen().bounds.size.height == 480.0 {
    let flowLayout = self.collectionView.collectionViewLayout as!
UICollectionViewFlowLayout
    flowLayout.itemSize = CGSizeMake(250.0, 300.0)
}
```

Base on the screen height, we can deduce if the device has a 3.5-inch screen. If it meets the criteria, we adjust the height of the collection view from `380` points to `300` points. Once you have made the change, try to test the app on iPhone 4s again. This should work now.

Handling the Like Button

In chapter 19, I showed you how to interact with collection views. You can apply the same techniques to handle the cell selections. However, it is a bit different for the TripCard app. We only want to toggle the heart button when a user taps on it. We don't want to toggle it when a user taps on the featured image or the price label.

To fit the requirement, we are going to define a new protocol named `TripCollectionCellDelegate` in the `TripCollectionCell` class:

```
protocol TripCollectionCellDelegate {  
    func didLikeButtonPressed(cell: TripCollectionCell)  
}
```

And declare a variable in the class to hold the delegate object:

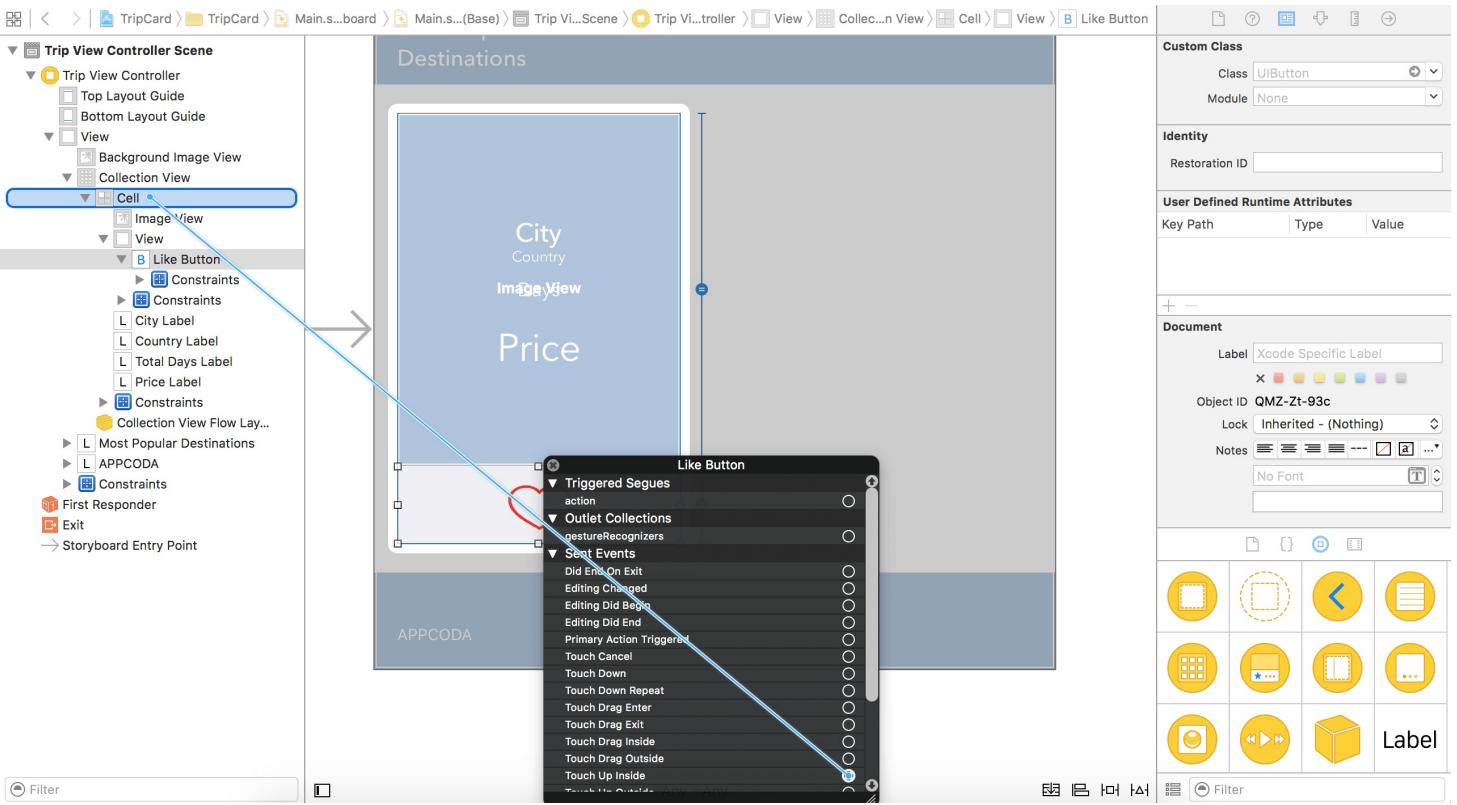
```
var delegate:TripCollectionCellDelegate?
```

In the protocol, we define a method called `didLikeButtonPressed`, which will be invoked when the heart button is tapped. The object that implements the delegate protocol is responsible for handling the button press.

Add the following action method, which is triggered when a user taps the heart button:

```
@IBAction func likeButtonTapped(sender: AnyObject) {  
    delegate?.didLikeButtonPressed(self)  
}
```

Now go back to the storyboard to associate the heart button with this method. Right-click the heart button and drag from *Touch Up Inside* to the Cell in the Document Outline. Select `likeButtonTapped:` when the popover appears.



Now open `TripViewController.swift`. It is the object that adopts the `TripCollectionCellDelegate` protocol:

```
class TripViewController: UIViewController, UICollectionViewDelegate,
UICollectionViewDataSource, TripCollectionCellDelegate
```

In the class, implement the required method of the protocol like this:

```
func didLikeButtonPressed(cell: TripCollectionCell) {
    if let indexPath = collectionView.indexPathForCell(cell) {
        trips[indexPath.row].isLiked = trips[indexPath.row].isLiked ? false :
true
        cell.isLiked = trips[indexPath.row].isLiked
    }
}
```

When the heart button is tapped, the `didLikeButtonPressed` method is called, along with the selected cell. Based on selected cell, we can determine the index path using the `indexPathForCell` method and toggle the status of `isLiked` accordingly.

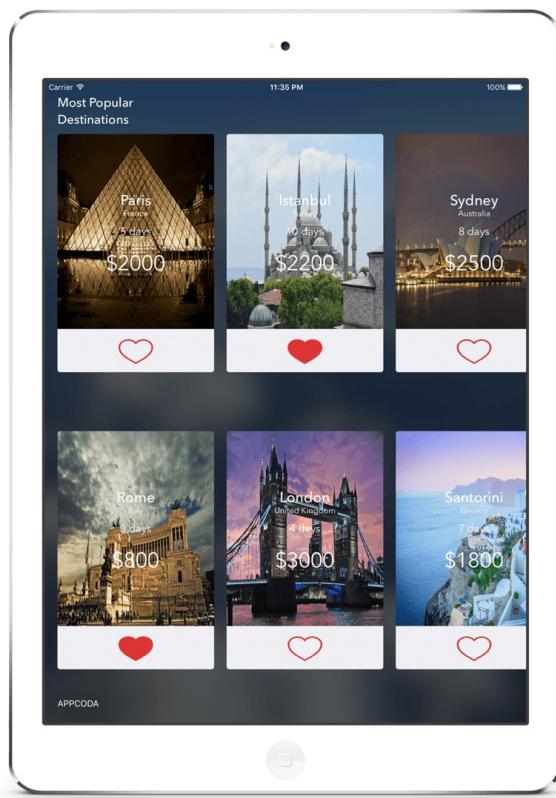
Recall that we have defined a `didSet` observer for the `isLiked` property of `TripCollectionCell`. The heart button will change its images according to the value of

`isLiked`. For instance, the app displays an empty heart if `isLiked` is set to `false`.

Lastly, insert a line of code in the `cellForItemAtIndexPath` method to set the cell's delegate:

```
cell.delegate = self
```

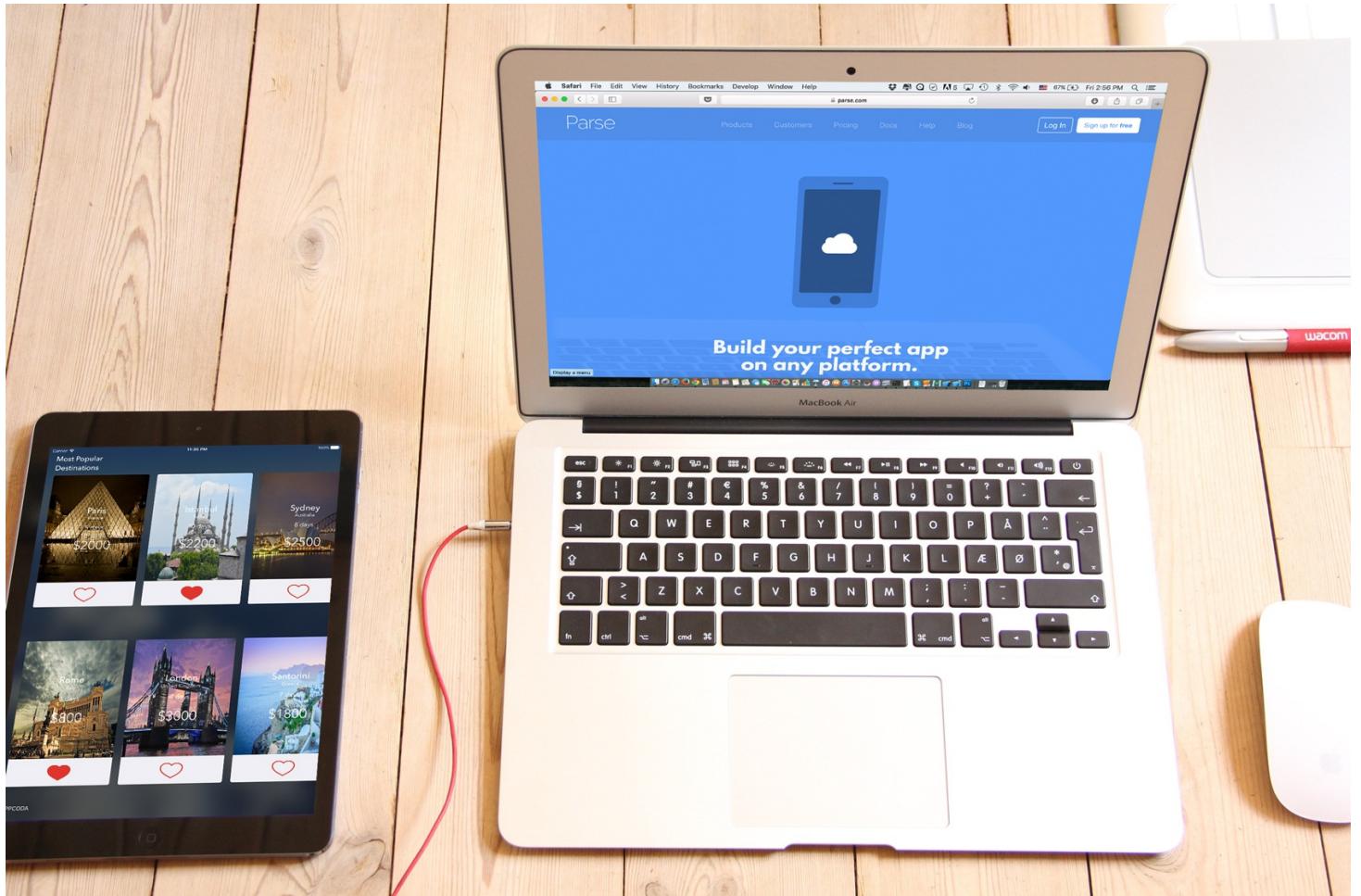
Okay, let's test the app again. When it launches, tapping the heart button of a trip can now favor the trip.



For reference, you can download the final project from
<https://www.dropbox.com/s/5j7mp4l6cgl7nor/TripCard.zip?dl=0>.

Chapter 30

Working with Parse



Some of your apps may need to store data on a server. Take the *TripCard* app that we developed in the previous chapter as an example. The app stored the trip information locally using an array. If you were building a real-world app, you would not keep the data in that way. The reason is quite obvious: You want the data to be manageable and updatable without re-releasing your app on App Store. The best solution is put your data onto a backend server that allows your app to communicate with it in order to get or update the data. Here you have several options:

- You can come up with your own home-brewed backend server, plus server-side APIs for data transfer, user authentication, etc.

- You can use CloudKit (which was introduced in iOS 8) to store the data in iCloud.
- You can make use of a third-party Backend as a Service provider (BaaS) to manage your data.

The downside of the first option is that you have to develop the backend service on your own. This requires a different skill set and a huge amount of work. As an iOS developer, you may want to focus on app development rather than server side development. This is one of the reasons why Apple introduced CloudKit, which makes developers' lives easier by eliminating the need to develop their own server solutions. With minimal setup and coding, CloudKit empowers your app to store data (including structured data and assets) in its new public database, where the shared data would be accessible by all users of the app. CloudKit works pretty well and is very easy to integrate (*note: it is covered in the [Beginning iOS 9 Programming with Swift](#) book*). However, CloudKit is only available for iOS. If you are going to port your app to Android that utilizes the shared data, CloudKit is not a viable option.

Parse, acquired by Facebook in late April 2013, is one of the BaaS that works across nearly all platforms including iOS, Android, Windows phone and web application. By providing an easy-to-use SDK, Parse allows iOS developers to easily manage the app data on the Parse cloud. This should save you development costs and time spent creating your own backend service. The service is free (with limits) and quick to set up.

In this chapter, I will walk you through the integration process of Parse. We will use the TripCard app as a demo and see how to put its trip data onto the Parse cloud. To begin with, you can download the TripCard project from

<https://www.dropbox.com/s/pnnf5inm54ksbom/ParseDemoStart.zip?dl=0>.

If you haven't read chapter 29, I highly recommend you to check it out first. It would be better to have some basic understandings of the demo app.

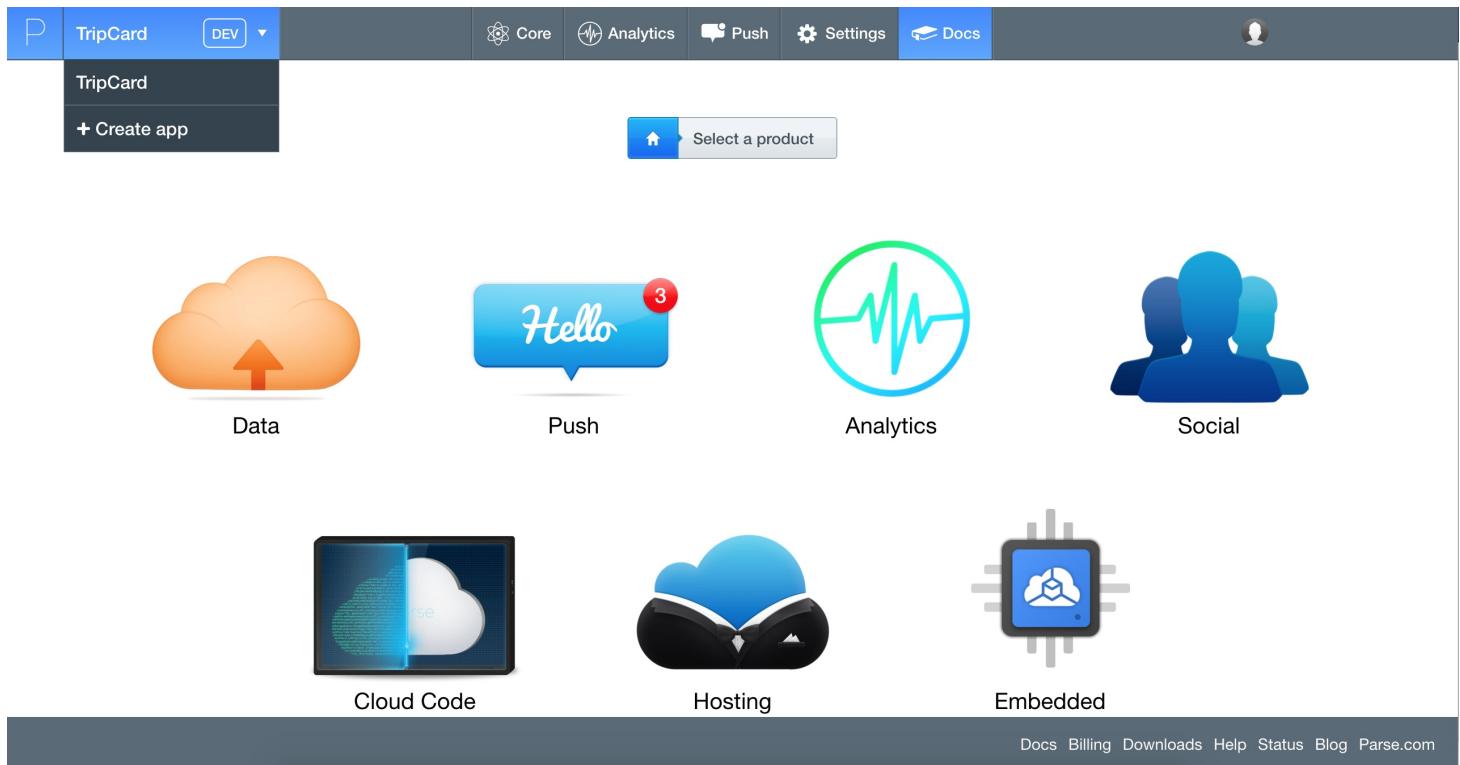
Okay, let's get started.

Creating Your App on Parse

First, you have to sign up for a free account on <http://parse.com>. During the signup process, you'll be prompted to create your first app. Simply use `TripCard` as the name. Once registered,

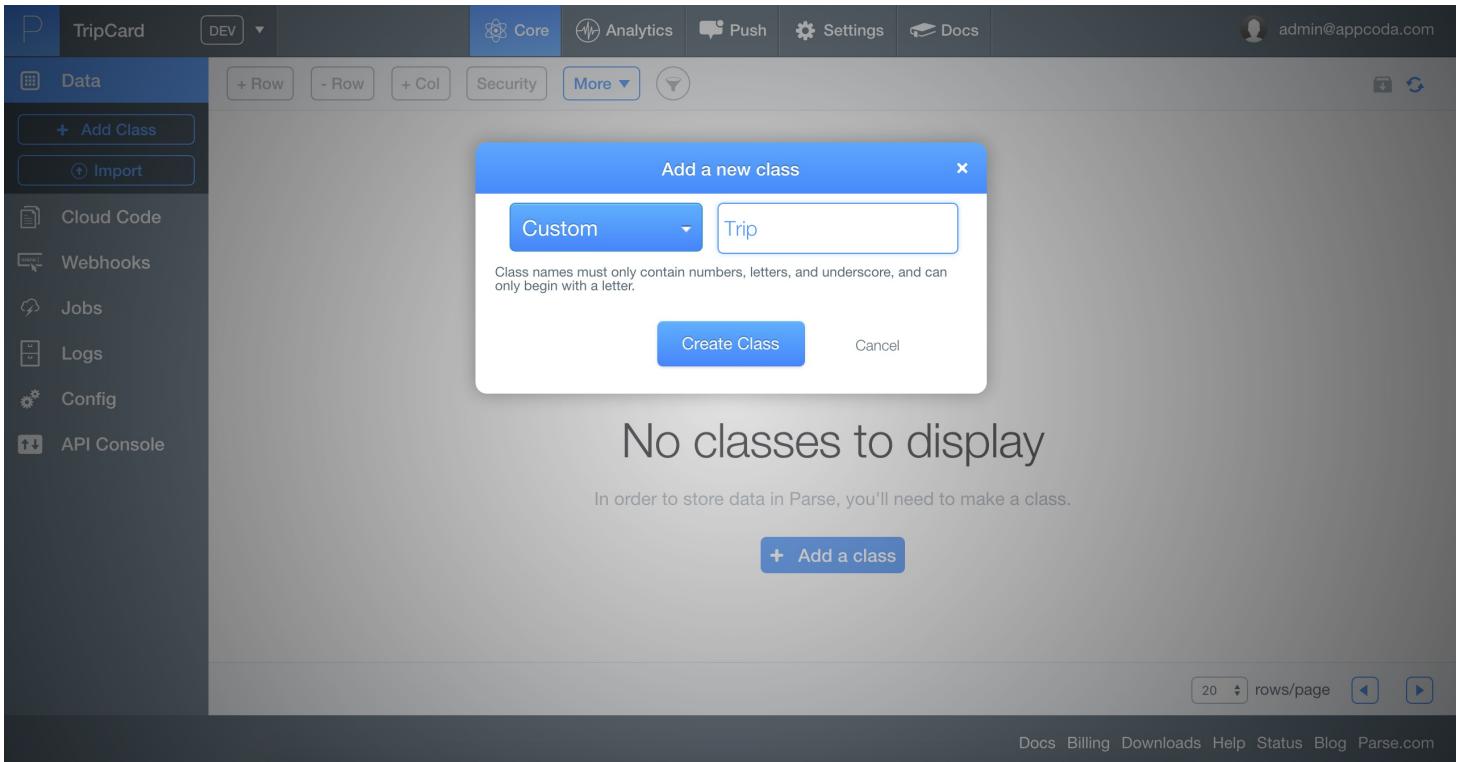
you'll be brought to the Parse dashboard. If you already have an account, you can move your mouse cursor to the dropdown menu at the top-left corner to create a new app.

Parse offers various backend services including data, push notification and many more. What we will focus on in this chapter is the data service. Make sure you've selected `TripCard` from the dropdown menu to configure your app.



Setting up Your Data

Now, select `Core` from the top menu. You will be brought to the data browser. By default, you do not have any data in the TripCard app. You will need to create and upload the trip data manually. But before that, you will have to define a `Trip` class in the data browser. The `Trip` class defined in Parse is the cloud version of the counterpart class that we have declared in our code. Each property of the class (e.g. `city`) will be mapped to a table column of the `Trip` class defined in Parse. Now click the *Add a Class* button to create a new class. Set the name to `Trip` and type to `Custom`. Once created, you should see the class under the Data section of the sidebar menu.



In the `TripCard` app, a trip consists of the following properties:

- Trip ID
- City
- Country
- Featured image
- Price
- Total number of days
- isLiked

With the exception of the trip ID, each of the properties should be mapped to a corresponding column of the `Trip` class in the data browser. Select the `Trip` class and click the `+ Col` button to add a new column.

The screenshot shows the AppCoda Parse.com dashboard. On the left sidebar, there are several navigation items: Data, + Add Class, Import, Cloud Code, Webhooks, Jobs, Logs, Config, and API Console. The main area displays a table with columns: objectId, String, createdAt, Date, updatedAt, Date, and ACL, ACL. A modal window titled "Add a Column" is open in the center. It has a dropdown menu set to "String" and a text input field containing "city". Below the input field is a note: "Must only contain alphanumeric or underscore characters, and must begin with a letter or number." At the bottom of the modal are two buttons: "Create Column" (highlighted in blue) and "Cancel".

When prompted, set the column name to `city` and type to `String`. Repeat the above procedures to add the rest of properties with the following column names and types:

- **Country:** Set the column name to `country` and type to `String`.
- **Featured image:** Set the column name to `featuredImage` and type to `File`. The `File` type is used for storing binary data such as image.
- **Price:** Set the column name to `price` and type to `Number`.
- **Total number of days:** Set the column name to `totalDays` and type to `Number`.
- **isLiked:** Set the column name to `isLiked` and type to `Boolean`.

Once you have added the columns, your table should look similar to the screenshot below.

The screenshot shows the Parse Data Browser interface. On the left, there's a sidebar with options like 'Add Class', 'Import', 'Cloud Code', 'Webhooks', 'Jobs', 'Logs', 'Config', and 'API Console'. The main area shows a table with the following columns: objectId, city, country, featuredImage, price, totalDays, isLiked, and createdAt. Below the table, it says 'No data to display' and 'Add a row to store an object in this class.' There's also a '+ Add a row' button.

You may wonder why we do not create a column for the trip ID. As you can see from the table, there is a default column named `objectId`. For each new row (or object), Parse automatically generates a unique ID. We will simply use this ID as the trip ID. You may also be wondering how we can convert the data stored in the Parse cloud to objects in our code? The Parse SDK is smart enough to handle the translation of native types. For instance, if you retrieve a `String` type from Parse, it will be translated into a `string` object in the app. We will discuss this in details later.

Now let's add some trip data into the data browser.

Click the `+ Row` button to create a new row. Each row represents a single Trip object. You only need to upload the image of a trip and fill in the city, country, price, totalDays and isLiked columns. For the `objectId`, `createdAt` and `updatedAt` columns, the values will be generated by Parse.

If you look into `TripViewController.swift`, the trips array is defined as follows:

```
private var trips = [Trip(tripId: "Paris001", city: "Paris", country: "France",
featuredImage: UIImage(named: "paris"), price: 2000, totalDays: 5, isLiked:
false),
    Trip(tripId: "Rome001", city: "Rome", country: "Italy", featuredImage:
UIImage(named: "rome"), price: 800, totalDays: 3, isLiked: false),
    Trip(tripId: "Istanbul001", city: "Istanbul", country: "Turkey",
featuredImage: UIImage(named: "istanbul"), price: 2200, totalDays: 10, isLiked:
false),
    Trip(tripId: "London001", city: "London", country: "United Kingdom",
featuredImage: UIImage(named: "london"), price: 3000, totalDays: 4, isLiked:
false),
```

```

    Trip(tripId: "Sydney001", city: "Sydney", country: "Australia",
featuredImage: UIImage(named: "sydney"), price: 2500, totalDays: 8, isLiked:
false),
    Trip(tripId: "Santorini001", city: "Santorini", country: "Greece",
featuredImage: UIImage(named: "santorini"), price: 1800, totalDays: 7, isLiked:
false),
    Trip(tripId: "NewYork001", city: "New York", country: "United States",
featuredImage: UIImage(named: "newyork"), price: 900, totalDays: 3, isLiked:
false),
    Trip(tripId: "Kyoto001", city: "Kyoto", country: "Japan", featuredImage:
UIImage(named: "kyoto"), price: 1000, totalDays: 5, isLiked: false)
]

```

To put the first item of the array into Parse, fill in the values of the row like this:

The screenshot shows the Parse Data Browser interface. On the left, there's a sidebar with buttons for 'Data' (selected), '+ Add Class', and 'Import'. The main area shows a table for the 'Trip' class. The table has columns: objectId, city, country, featuredImage, price, totalDays, isLiked, and createdAt. A single row is selected, corresponding to the first trip in the array above. The 'objectId' column contains 'jokGseXhx2', 'city' is 'Paris', 'country' is 'France', 'featuredImage' is 'paris.jpg', 'price' is 2000, 'totalDays' is 5, 'isLiked' is false, and 'createdAt' is 'Oct 23, 2015, 03:4'.

	objectId	city	country	featuredImage	price	totalDays	isLiked	createdAt
Trip	jokGseXhx2	Paris	France	paris.jpg	2000	5	false	Oct 23, 2015, 03:4

This is very straightforward. We just map the property of the `Trip` class to the column values of its Parse counterpart. Just note that Parse stores the actual image of the trip in the `featuredImage` column. You should have to upload the `paris.jpg` file by clicking the *Upload File* button.

Quick note: You can find the images in the `Images.xcassets` folder of the Xcode project.

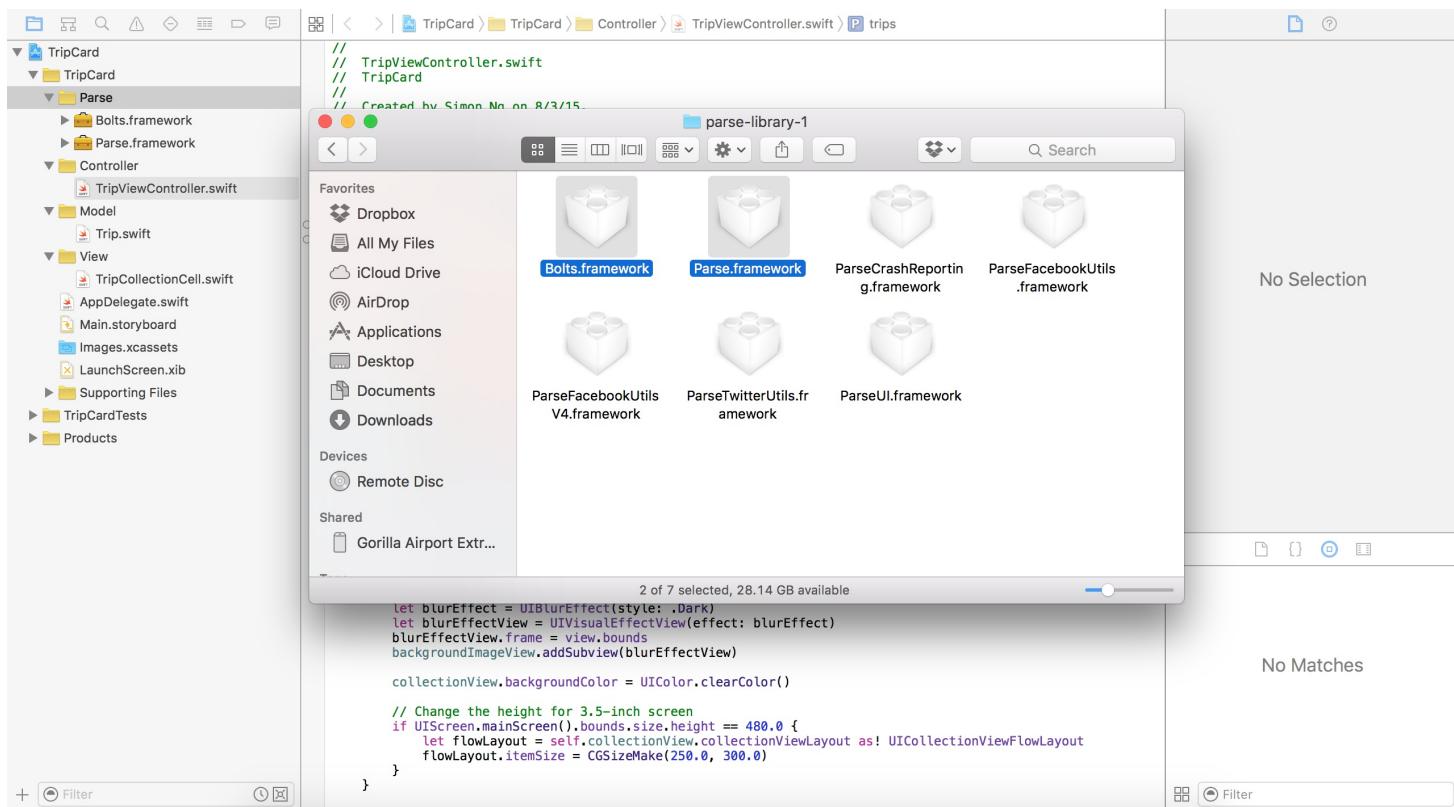
Repeat the above procedures and add the rest of the trip data. You will end up with a screen similar to this:

The screenshot shows the Parse Data Browser interface with 8 rows of trip data. The table structure is identical to the one in the previous screenshot, but it includes more rows. The last row, which corresponds to the Paris trip, has its 'totalDays' value highlighted in blue, indicating it's currently being edited. The rows represent different trips with various details like city, country, price, and duration.

	objectId	city	country	featuredImage	price	totalDays	isLiked	createdAt
Trip	WqnrVZLcqs	Rome	Italy	rome.jpg	800	3	true	Oct 23, 2015, 04:0
+ Add Class	grmF4V3gpz	Istanbul	Turkey	istanbul.jpg	2200	10	false	Oct 23, 2015, 04:0
Import	sgl4kdclkJ	London	United Kingdom	london.jpg	3000	4	false	Oct 23, 2015, 04:0
Cloud Code	F1RZSPXbeC	Sydney	Australia	sydney.jpg	2500	8	false	Oct 23, 2015, 04:0
Webhooks	rFIHERGaRc	Santorini	Greece	santorini.jpg	1800	7	false	Oct 23, 2015, 04:0
Jobs	YgMNk5rXKK	New York	United States	newyork.jpg	900	3	true	Oct 23, 2015, 04:0
	Z3jyuwCjmc	Kyoto	Japan	kyoto.jpg	1000	5	true	Oct 23, 2015, 03:5
	jokGseXhx2	Paris	France	paris.jpg	2000	5	false	Oct 23, 2015, 03:4

Configuring the Xcode Project for Parse

Now that you have configured the trip data on the Parse cloud, we will start to integrate the *TripCard* project with Parse. To begin with, download the Parse SDK for iOS from <https://www.parse.com/downloads/ios/parse-library/latest>. Unzip the file and drag both *Bolts.framework* and *Parse.framework* into the *TripCard* project. Optionally, you can create a new group called *Parse* to better organize the files. When prompted, make sure you enable the *Copy items if needed* option and click *Finish* to proceed.

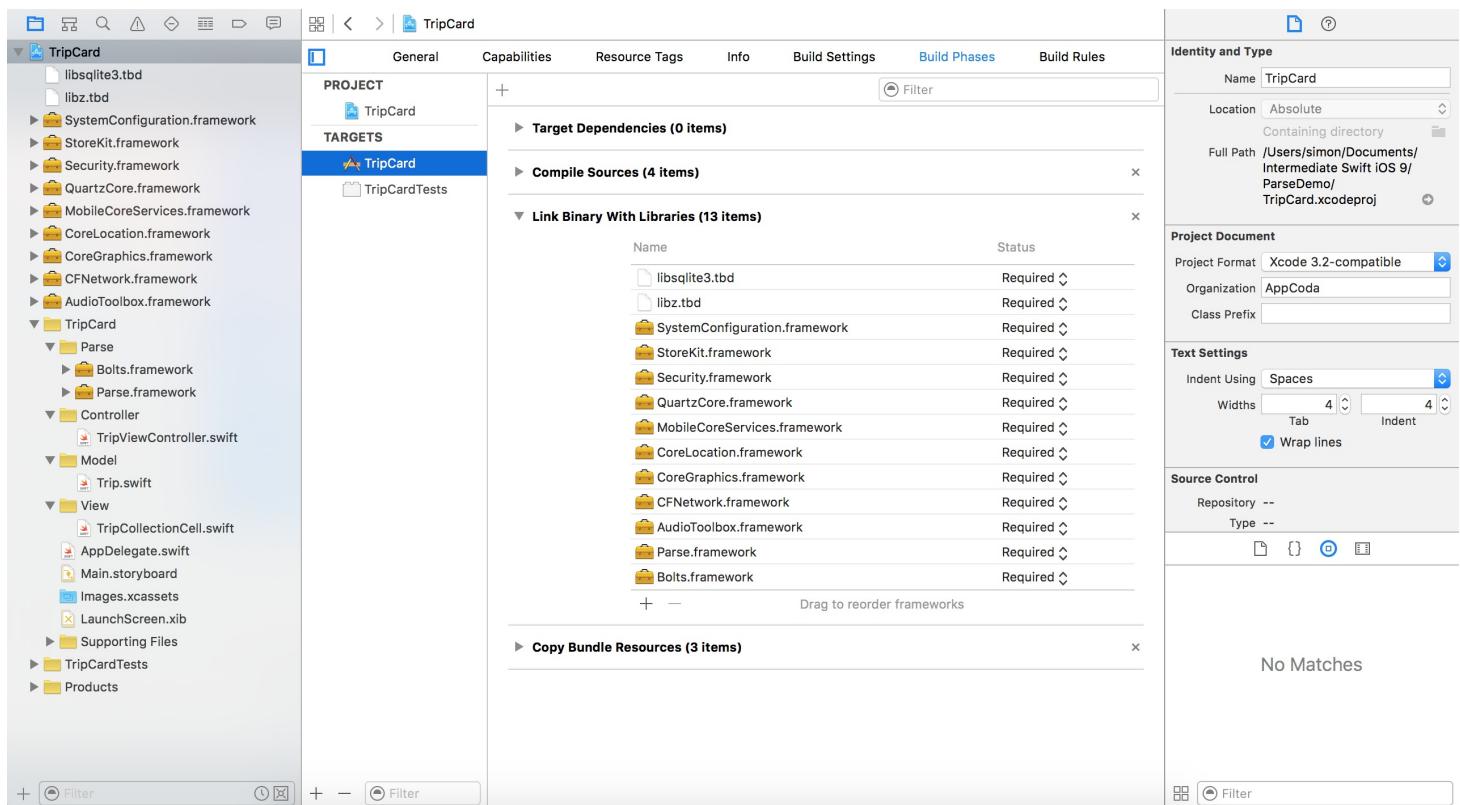


The Parse SDK depends on other frameworks in iOS SDK. You will need to add the following libraries into the project:

- *AudioToolbox.framework*
- *CFNetwork.framework*
- *CoreGraphics.framework*
- *CoreLocation.framework*
- *MobileCoreServices.framework*
- *QuartzCore.framework*
- *Security.framework*
- *StoreKit.framework*
- *SystemConfiguration.framework*

- libz.dylib
- libsqlite3.dylib

Select the TripCard project in the project navigator. Under the TripCard target, select *Build Phases* and expand the *Link Binary with Libraries*. Click the **+** button and add the above libraries one by one.



Creating the Bridging Header

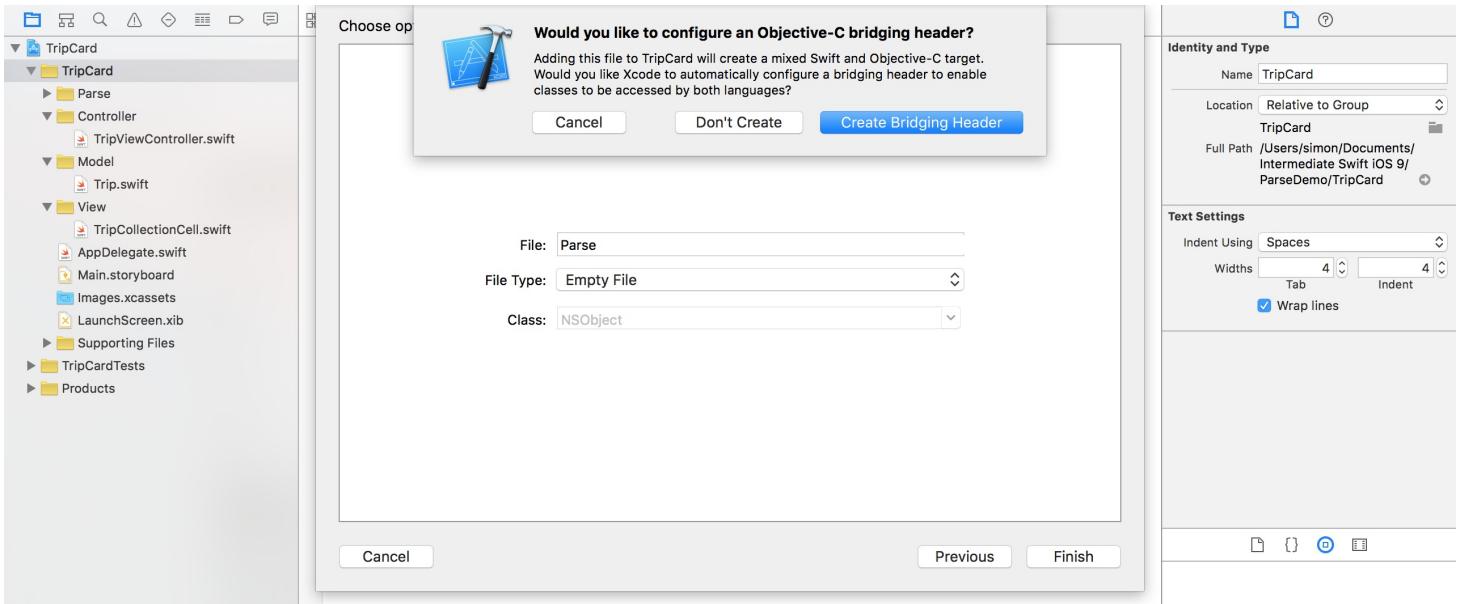
The Parse framework was written in Objective-C. One of the great things about Swift is that it is fully compatible with any Objective-C libraries. To use Parse in your Swift project, all you need to do is create a new file using the Objective-C File template. In the project navigator, right click the *TripCard* folder and select **New File**.

Choose a template for your new file:

iOS	 Cocoa Touch Class	 UI Test Case Class	 Unit Test Case Class	 Playground
	 Swift File	 Objective-C File	 Header File	 C File
watchOS	 C++ File	 Metal File		
OS X	<p>Objective-C File An empty Objective-C file, category, protocol or extension.</p>			

[Cancel](#) [Previous](#) [Next](#)

Choose the Objective-C File template and proceed. Name the class `Parse` (or whatever you like) and set the file type to `Empty file`. Then press continue to proceed and save the file. When prompted, click `Create Bridging Header` to configure an Objective-C bridging header.



Xcode creates two new files: `Parse.m` and `TripCard-Bridging-Header.h`. We do not need the `Parse.m` file, so you can simply delete it. In `TripCard-Bridging-Header.h`, add this line of code:

```
#import <Parse/Parse.h>
```

That's it. Now you should be able to access the Parse framework from the Swift project.

Connecting with Parse

To access your app data on Parse, you first need to find out the Application Key and the Client Key. Go to the Parse Dashboard and select Settings -> Keys.

Application Keys			
Application ID	GJpqNiRQW9gve2jT8gfPt	9gxDL	<button>Copy</button>
Client Key	aN2iqplAGRgnacf0ykQoGc	XZeM	<button>Copy</button>
JavaScript Key	fh1QH1ShJCU1p8R7FCeysmj	X8P	<button>Copy</button>
.NET Key	Tx8bAV3s1KwdI6RW00TsEyi	iv0	<button>Copy</button>
Webhook Key	ZBNhc9roaTJ2R7FXiLUpAZL	DxG	<button>Copy</button>
REST API Key	RyppftPs5pa9xeJgFFvRQ	26Tz	<button>Copy</button>
Master Key	sGfk1aART71D5E7jmN1kd>	eIrz	<button>Copy</button>

Here you can reveal the application ID and client key. Remember to keep these keys safe, as one can access your Parse data with them.

Open up `AppDelegate.m` and add the following code in the `didFinishLaunchingWithOptions` method to initialize Parse:

```
Parse.setApplicationId("R9h4YgyAQHTXIwLmzw7NRl1sy04T1Cij8HQ6cjz", clientKey:  
"unSLRwwfleGB6DUxPz8Mwdyvf2A4vPLStWXYYus9")
```

Note that you should replace the Application ID and the Client Key with your own keys. With just a line of code, your app is ready to connect to Parse. Try to compile and run it. If you get everything correct, you should be able to run the app without any error.

Retrieving Data from Parse

Next, we'll replace the trips array with the cloud data. To do that, you will have to retrieve the Trip objects that were just created on Parse. The Parse SDK provides a class called `PFQuery` for retrieving a list of objects (`PFObjects`) from Parse. The general usage is like this:

```
let query = PFQuery(className:"Trip")
query.findObjectsInBackgroundWithBlock {
    (objects: [AnyObject]?, error: NSError?) -> Void in
    if error == nil {
```

```

    // Successfully retrieved the objects from Parse
    if let objects = objects as? [PFObject] {
        // Do something
    }
} else {
    // Log details of the failure
    println("Error: \(error) \(error!.userInfo)")
}
}

```

You create a `PFQuery` object with a specific class name that matches the one created on Parse. For example, for the *TripCard* app, the class name is `Trip`. By calling the `findObjectsInBackgroundWithBlock` method of the query object, the app will go up to Parse and retrieve the available Trip objects. The method works in an asynchronous manner. When it finishes, the block of code will be called and you can perform additional processing based on the returned results.

With a basic understanding of data retrieval, we will modify the *TripCard* app to get the data from the Parse cloud.

First, open the `TripViewController.swift` file and change the declaration of trips array to this:

```
private var trips = [Trip]()
```

Instead of populating the array with static data, we initialize an empty array. Later we will get the trip data from Parse at runtime and save them into the array.

If you look into the `Trip` class (i.e. `Trip.swift`), you may notice that the `featuredImage` property is of `UIImage`. As we have defined the `featuredImage` column as a `File` type on Parse, we have to change the type of the `featuredImage` property accordingly. This would allow us to convert a `PFObject` to a `Trip` object easily.

The corresponding class of a `File` type in Parse, that lets you store application files (e.g. images) in the cloud, is `PFFFile`. Now open `Trip.swift` and update it to the following:

```

class Trip {
    var tripId = ""
    var city = ""
    var country = ""
    var featuredImage:PFFFile?
}

```

```

var price:Int = 0
var totalDays:Int = 0
var isLiked = false

init(tripId: String, city: String, country: String, featuredImage: PFFile!,  

price: Int, totalDays: Int, isLiked: Bool) {
    self.tripId = tripId
    self.city = city
    self.country = country
    self.featuredImage = featuredImage
    self.price = price
    self.totalDays = totalDays
    self.isLiked = isLiked
}

init(pfObject: PFObject) {
    self.tripId = pfObject.objectId!
    self.city = pfObject["city"] as! String
    self.country = pfObject["country"] as! String
    self.price = pfObject["price"] as! Int
    self.totalDays = pfObject["totalDays"] as! Int
    self.featuredImage = pfObject["featuredImage"] as? PFFile
    self.isLiked = pfObject["isLiked"] as! Bool
}

func toPFObject() -> PFObject {
    let tripObject = PFObject(className: "Trip")
    tripObject.objectId = tripId
    tripObject["city"] = city
    tripObject["country"] = country
    tripObject["featuredImage"] = featuredImage
    tripObject["price"] = price
    tripObject["totalDays"] = totalDays
    tripObject["isLiked"] = isLiked

    return tripObject
}
}

```

Here we added another initialization method for `PFObject` and a new method called `toPFObject`. In this method, we change the type of `featuredImage` from `UIImage` to `PFFile`. For the purpose of convenience, we create a new initialization method for `PFObject` and another method for `PFObject` conversion.

Next, open the `TripViewController.swift` file and add the following method:

```
func loadTripsFromParse() {
```

```

// Clear up the array
trips.removeAll(keepCapacity: true)
collectionView.reloadData()

// Pull data from Parse
let query = PFQuery(className:"Trip")
query.findObjectsInBackgroundWithBlock { (objects, error) -> Void in

    if let error = error {
        print("Error: \(error) \(error.userInfo)")
        return
    }

    if let objects = objects {
        for (index, object) in objects.enumerate() {
            // Convert PFObject into Trip object
            let trip = Trip(pfObject: object)
            self.trips.append(trip)

            let indexPath = NSIndexPath(forRow: index, inSection: 0)
            self.collectionView.insertItemsAtIndexPaths([indexPath])
        }
    }
}

}

```

The `loadTripsFromParse` method is created for retrieving trip information from Parse. At the very beginning, we clear out the `trips` array so as to have a fresh start. We then pull the trip data from Parse using `PFQuery`. If the objects are successfully retrieved from the cloud, we convert each of the `PFObjects` into `Trip` objects and append them to the `trips` array. Lastly, we insert the trip to the collection view by calling the `insertItemsAtIndexPaths` method.

For the `cellForItemAtIndexPath` method, you will need to change the following line of code:

```
cell.imageView.image = trips[indexPath.row].featuredImage
```

to:

```

// Load image in background
cell.imageView.image = UIImage()
if let featuredImage = trips[indexPath.row].featuredImage {
    featuredImage.getDataInBackgroundWithBlock({ (imageData: NSData?, error: NSError?) -> Void in
        if let tripImageData = imageData {

```

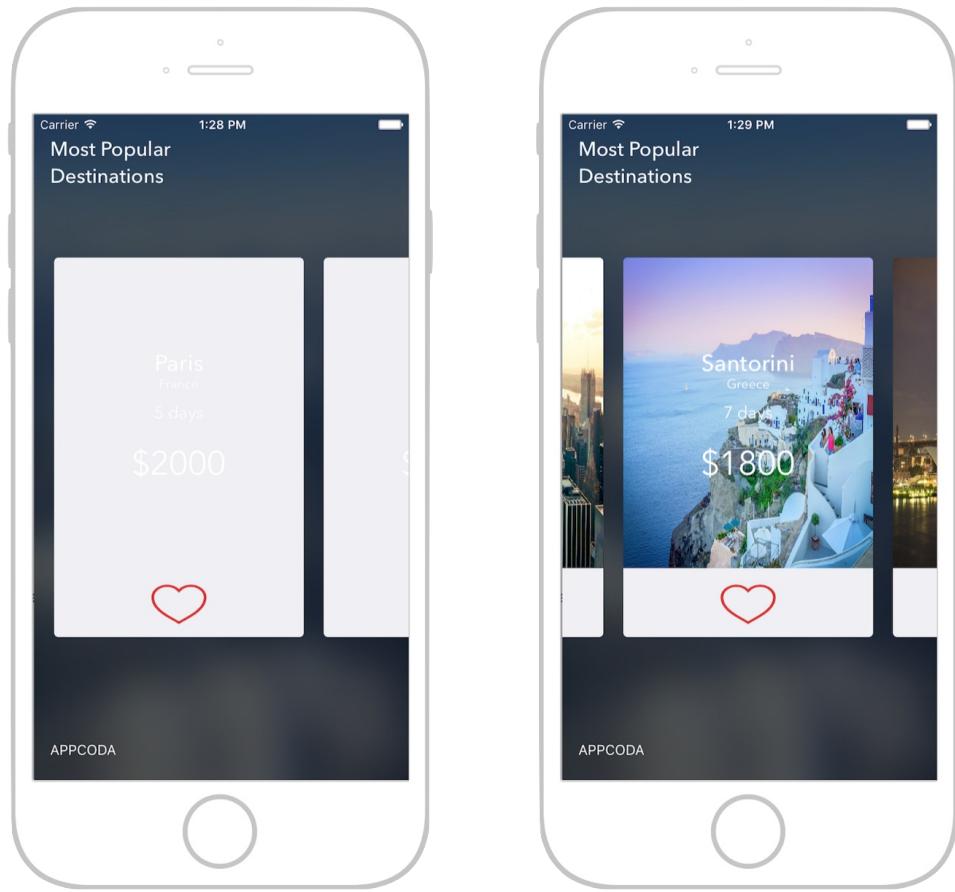
```
    cell.imageView.image = UIImage(data:tripImageData)
}
})
}
```

The trip images are no longer bundled in the app. Instead, we will pull them from the Parse cloud. The time required to load the images varies depending on the network speed. This is why we handle the image download in background. Parse stores files (such as images, audio and documents) in the cloud in the form of PFFFile. We use PFFFile to reference the featured image. The class provides the `getDataInBackgroundWithBlock` method to perform the file download in background. Once the download completes, we load it onto the screen.

Finally insert this line of code in the `viewDidLoad` method to start the data retrieval:

```
loadTripsFromParse()
```

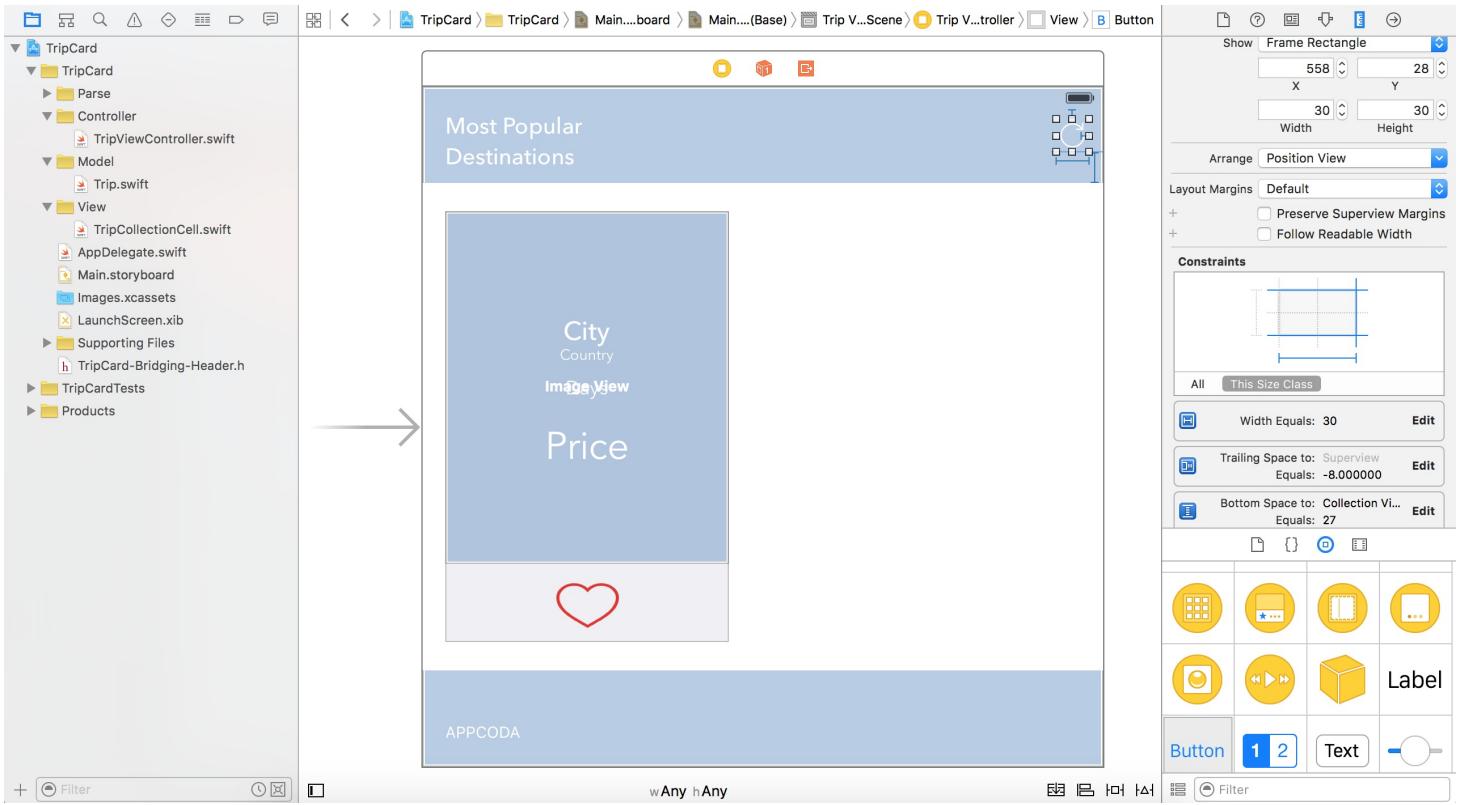
Now you are ready to go! Hit the “Run” button to test the app. Make sure your computer/device is connected to the Internet. The TripCard app should now retrieve the trip information from Parse. Depending on your network speed, it will take a few seconds for the images to load.



Refreshing Data

Currently, there is no way to refresh the data. Let's add a button to the Trip View Controller in storyboard. When a user taps the button, the app will go up to Parse and refresh the trip information.

The project template already bundled a reload image for the button. Simply open `Main.storyboard` and drag a button object to the view controller. Set its width and height to 30 points. Also, change its image to `reload` and tint color to `white`. Finally, click the *Issues* button of the auto layout menu and select *Add Missing Constraints* to add the layout constraints.

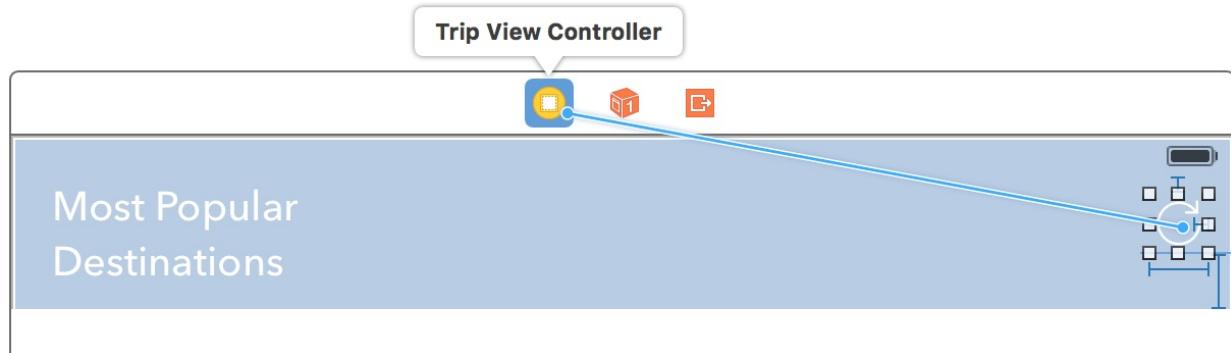


Your design should be similar to the screen above. Next, add an action method in

`TripViewController.swift` :

```
@IBAction func reloadButtonTapped(sender: AnyObject) {
    loadTripsFromParse()
}
```

Go back to the storyboard and associate the refresh button with this action method. Control-drag from the `refresh` button to the Trip View Controller. After releasing the buttons, select `reloadButtonTapped:`.



Now run the app again. Once it's launched, go to the Parse dashboard and add a new trip. Your app should now retrieve the new trip when the refresh button is tapped.

Caching for Speed and Offline Access

Try to close the app and re-launch it. Every time when it is launched, the app starts to download the trips from Parse. What if there is no network access? Let's give it a try. Disable your iPhone or the simulator's network connection and run the app again. The app will not be able to display any trips with the following error in the console:

```
2015-10-23 14:26:51.056 TripCard[17982:1783498] [Error]: The Internet  
connection appears to be offline. (Code: 100, Version: 1.9.0)  
2015-10-23 14:26:51.057 TripCard[17982:1783498] [Error]: Network connection  
failed. Making attempt 1 after sleeping for 1.215702 seconds.  
2015-10-23 14:26:52.392 TripCard[17982:1783498] [Error]: The Internet  
connection appears to be offline. (Code: 100, Version: 1.9.0)  
2015-10-23 14:26:52.393 TripCard[17982:1783498] [Error]: Network connection  
failed. Making attempt 2 after sleeping for 2.431403 seconds.  
2015-10-23 14:26:55.068 TripCard[17982:1783499] [Error]: The Internet  
connection appears to be offline. (Code: 100, Version: 1.9.0)  
2015-10-23 14:26:55.068 TripCard[17982:1783499] [Error]: Network connection  
failed. Making attempt 3 after sleeping for 4.862806 seconds.
```

There is a better way to handle this situation. Parse has a built-in support for caching that makes it a lot easier to save query results on local disk. In case Internet access is not available, your app can load the result from cache. Caching also improves the app's performance. Instead of loading data from Parse every time when the app runs, it retrieves the data from cache upon startup.

In the default setting, caching is disabled. But it can easily be enabled by using a single line of code. Add the following code in the `loadTripsFromParse` method after the initialization of `PFQuery`:

```
query.cachePolicy = PFCachePolicy.NetworkElseCache
```

The Parse query supports various types of cache policy. The `NetworkElseCache` policy is just one of them. It first loads data from the network, then if that fails, it loads results from the cache.

Now compile and run the app again. After you run it once (with WiFi enabled), disable the WiFi or other network connections and launch the app again. This time, your app should be able to show the trips even if the network is unavailable.

Updating Data on Parse

When you like a trip by tapping the heart button, the result is not saved to the Parse cloud. Updating a `PFObject` is pretty simple. Recalled that each `PFObject` comes with a unique object ID. All you need to do is to set some new data to an existing `PFObject` and then call the `saveInBackgroundWithBlock` method to upload the changes to the cloud. Based on the object ID, Parse updates the data of the specific object.

Open `TripViewController.swift` and update the `didLikeButtonPressed` method, like this:

```
func didLikeButtonPressed(cell: TripCollectionCell) {
    if let indexPath = collectionView.indexPathForCell(cell) {
        trips[indexPath.row].isLiked = trips[indexPath.row].isLiked ? false :
true
        cell.isLiked = trips[indexPath.row].isLiked

        // Update the trip on Parse
        trips[indexPath.row].toPFObject().saveInBackgroundWithBlock({ (success,
error) -> Void in
            if (success) {
                print("Successfully updated the trip")
            } else {
                print("Error: \(error?.description)")
            }
        })
    }
}
```

We first call the `toPFObject` method of the selected `Trip` object to convert itself to a `PFObject`. If you look into the `toPFObject` method of the `Trip` class, you will notice that the trip ID is set as the object ID of the `PFObject`.

Once we have the `PFObject`, we simply call the `saveInBackgroundWithBlock` method to upload the changes to Parse.

That's it.

You can now run the app again. Tap the heart button of a trip and go up to the data browser of Parse. You should find that the `isLiked` value of the selected trip (say, Santorini) is changed to `true`.

Deleting Data from Parse

Similarly, `PFObject` provides various methods for object deletion. In short, you call up the `deleteInBackgroundWithBlock` method of the `PFObject` class to delete the object from Parse.

Currently, the *TripCard* app does not allow users to remove a trip. We will modify the app to let users swipe up a trip item to delete it. iOS provides the `UISwipeGestureRecognizer` class to recognize swipe gestures. In the `viewDidLoad` method of the `TripViewController` class, insert the following lines of code to initialize a gesture recognizer:

```
// Setup swipe gesture
let swipeUpRecognizer = UISwipeGestureRecognizer(target: self, action:
"handleSwipe:")
swipeUpRecognizer.direction = .Up
swipeUpRecognizer.delegate = self
self.collectionView.addGestureRecognizer(swipeUpRecognizer)
```

When creating the `UISwipeGestureRecognizer` object, we specify the action method to call when the swipe gesture is recognized. Here we will invoke the `handleSwipe` method of the current object, which will be implemented later.

Because we only want to look for the swipe-up gesture, we specify the direction property of the recognizer as `up`. When using a gesture recognizer, you must associate it with a certain view that the touches happen. In the above code, we invoke the `addGestureRecognizer` method to associate the collection view with the recognizer.

The delegate of the recognizer should adopt the `UIGestureRecognizerDelegate` protocol. Thus, add it to the class declaration:

```
class TripViewController: UIViewController, UICollectionViewDelegate,
UICollectionViewDataSource, TripCollectionCellDelegate,
UIGestureRecognizerDelegate
```

Next, implement the `handleSwipe` method like this:

```
func handleSwipe(gesture: UISwipeGestureRecognizer) {
    let point = gesture.locationInView(self.collectionView)
    if (gesture.state == UIGestureRecognizerState.Ended) {
        if let indexPath = collectionView.indexPathForItemAtPoint(point) {
            // Remove trip from Parse, array and collection view
    }
}
```

```

        trips[indexPath.row].toPFObject().deleteInBackgroundWithBlock({
(success, error) -> Void in
    if (success) {
        print("Successfully updated the trip")
    } else {
        print("Error: \(error?.description)")
    }

    self.trips.removeAtIndex(indexPath.row)
    self.collectionView.deleteItemsAtIndexPaths([indexPath])
})
}
}
}
}

```

When a user swipes up a trip item (i.e. a collection view cell), we first need to determine which cell is going to be removed. The `locationInView` method provides the location of the gesture in the form of `CGPoint`. From the point returned, we can compute the index path of the collection cell by using the `indexPathForItemAtPoint` method. Once we have the index path of the cell to be removed, we simply call the `deleteInBackgroundWithBlock` method to delete it from Parse.

Great! You've implemented the delete feature. Hit the Run button to launch the app and try to delete a record from Parse.

I hope that this chapter gave you an idea of how to connect your app to the cloud. Parse, as well as other cloud providers (e.g. Apple's CloudKit), simplifies the whole development of backend services. Additionally, the startup cost of using a cloud is nearly zero. Most of the BaaS providers are free to use, and you only need to pay when the number of requests reaches a certain volume. If you think it's too hard to integrate your app with the cloud, think again! Consider building your existing apps with some cloud features.

For reference, you can download the final project from
<https://www.dropbox.com/s/vlhzu4c33e6ex57/ParseDemo.zip?dl=0>.

Chapter 31

How to Preload a SQLite Database Using Core Data



When working with Core Data, you may have asked these two questions:

- How can you preload existing data into the SQLite database?
- How can you use an existing SQLite database in my Xcode project?

I recently met a friend who is now working on a dictionary app for a particular industry. He got the same questions. He knows how to save data into the database and retrieve them back from

the Core Data store. The real question is: *how could he preload the existing dictionary data into the database?*

I believe some of you may have the same question. This is why I devote a full chapter to talk about data preloading in Core Data. I will answer the above questions and show you how to preload your app with existing data.

So how can you preload existing data into the built-in SQLite database of your app? In general you bundle a data file (in CSV or JSON format or whatever format you like). When the user launches the app for the very first time, it preloads the data from the data file and puts them into the database. At the time when the app is fully launched, it will be able to use the database, which has been pre-filled with data. The data file can be either bundled in the app or hosted on a cloud server. By storing the file in the cloud or other external sources, this would allow you to update the data easily, without rebuilding the app. I will walk you through both approaches by building a simple demo app.

Once you understand how data preloading works, I will show you how to use an existing SQLite database (again pre-filled with data) in your app. Note that I assume you have a basic understanding of Core Data. You should know how to insert and retrieve data through Core Data. If you have no ideas about these operations, you can refer to the [Beginning iOS 9 Programming with Swift](#) book.

A Simple Demo App

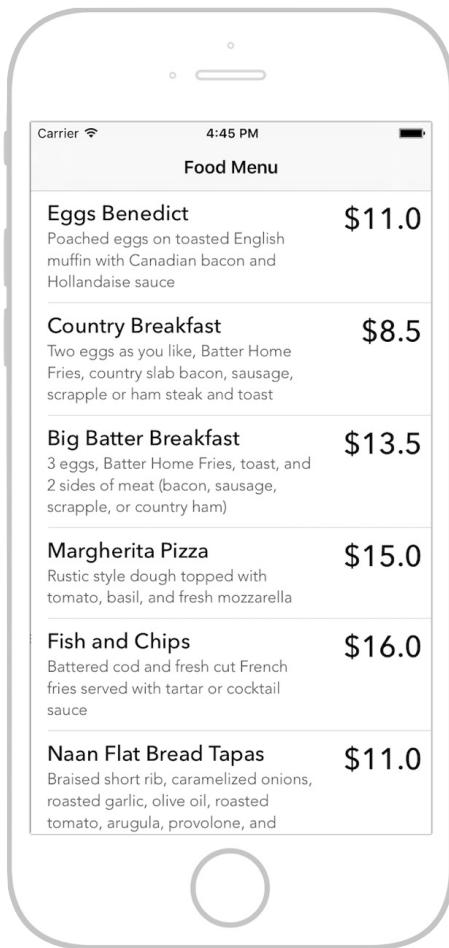
To keep your focus on learning data preloading, I have created the project template for you. Firstly, download the project from
<https://www.dropbox.com/s/l3m5gky5qu959jt/CoreDataPreloadDemoStart.zip?dl=0> and have a trial run.

It's a very simple app showing a list of food. By default, the starter project comes with an empty database. When you compile and launch the app, your app will end up a blank table view. What we are going to do is to preload the database with existing data.

I have already built the data model and provided the implementation of the table view. You can look into the `MenuItemTableViewController` class and `coreDataDemo.xcdatamodeld` for details.

The data model is pretty simple. I have defined a `MenuItem` entity, which includes three attributes: *name*, *detail*, and *price*.

Once you're able to preload the database with the food menu items, the app will display them accordingly, with the resulting user interface similar to the screenshot shown below.



The CSV File

In this demo I use a CSV file to store the existing data. CSV files are often used to store tabular data and can be easily created using text editor, Numbers or MS Excel. They are sometimes known as comma delimited files. Each record is one line and fields are separated with commas. In the project template, you should find the `menudata.csv` file. It contains all the food items for the demo app in CSV format. Here is a part of the file:

```
Eggs Benedict,"Poached eggs on toasted English muffin with Canadian bacon and Hollandaise sauce",11.0
```

```

Country Breakfast,"Two eggs as you like, Batter Home Fries, country slab bacon,
sausage, scrapple or ham steak and toast", 8.5
Big Batter Breakfast,"3 eggs, Batter Home Fries, toast, and 2 sides of meat
(bacon, sausage, scrapple, or country ham)",13.5
Margherita Pizza,"Rustic style dough topped with tomato, basil, and fresh
mozzarella",15.0
Fish and Chips,Battered cod and fresh cut French fries served with tartar or
cocktail sauce,16.0

```

The first field represents the name of the food menu item. The next field is the detail of the food, while the last field is the price. Each food item is one line, separated with a new line separator.



Parsing CSV Files

It's not required to use CSV files to store your data. JSON and XML are two common formats for data interchange and flat file storage. As compared to CSV format, they are more readable and suitable for storing structured data. Anyway, CSV has been around for a long time and is supported by most spreadsheet applications. At some point of time, you will have to deal with this type of file. So I pick it as an example. Let's see how we can parse the data from CSV.

The `AppDelegate` object is normally used to perform tasks during application startup (and shutdown). To preload data during the app launch, we will add a few methods in the `AppDelegate` class. First, insert the following method for parsing the CSV file:

```
func parseCSV (contentsOfURL: NSURL, encoding: NSStringEncoding) ->
```

```

[(name:String, detail:String, price: String)]? {
    // Load the CSV file and parse it
    let delimiter = ","
    var items:[(name:String, detail:String, price: String)]?

    do {
        let content = try String(contentsOfURL: contentsOfURL, encoding:
encoding)
        items = []
        let lines:[String] =
content.componentsSeparatedByCharactersInSet(NSCharacterSet.newlineCharacterSet(
as [String]

        for line in lines {
            var values:[String] = []
            if line != "" {
                // For a line with double quotes
                // we use NSScanner to perform the parsing
                if line.rangeOfString("\"") != nil {
                    var textToScan:String = line
                    var value:NSStrng?
                    var textScanner:NSScanner = NSScanner(string: textToScan)
                    while textScanner.string != "" {

                        if (textScanner.string as NSString).substringToIndex(1)
== "\"" {
                            textScanner.scanLocation += 1
                            textScanner.scanUpToString("\"", intoString:
&value)
                            textScanner.scanLocation += 1
                        } else {
                            textScanner.scanUpToString(delimiter, intoString:
&value)
                        }

                        // Store the value into the values array
                        values.append(value as! String)

                        // Retrieve the unscanned remainder of the string
                        if textScanner.scanLocation <
textScanner.string.characters.count {
                            textToScan = (textScanner.string as
NSString).substringFromIndex(textScanner.scanLocation + 1)
                        } else {
                            textToScan = ""
                        }
                        textScanner = NSScanner(string: textToScan)
                    }
                }
            }
        }
    }
}

```

```

        // For a line without double quotes, we can simply separate
the string
        // by using the delimiter (e.g. comma)
    } else {
        values = line.componentsSeparatedByString(delimiter)
    }

    // Put the values into the tuple and add it to the items array
let item = (name: values[0], detail: values[1], price:
values[2])
    items?.append(item)
}
}

} catch {
    print(error)
}

return items
}

```

The method takes in three parameters: *the file's URL* and *encoding*. It first loads the file content into memory, reads the lines into an array and then performs the parsing line by line. At the end of the method, it returns an array of food menu items in the form of tuples.

A simple CSV file only uses a comma to separate values. Parsing such kind of CSV files shouldn't be difficult. You can call the `componentsSeparatedByString` method to split a comma-delimited string. It'll then return you an array of strings that have been divided by the separator. For some CSV files, they are more complicated. Field values containing reserved characters (e.g. comma) are surrounded by double quotes. Here is another example:

```
Country Breakfast,"Two eggs as you like, Batter Home Fries, country slab bacon,
sausage, scrapple or ham steak and toast", 8.5
```

In this case, we cannot simply use the `componentsSeparatedByString` method to separate the field values. Instead, we use `NSScanner` to go through each character of the string and retrieve the field values. If the field value begins with a double quote, we scan through the string until we find the next double quote character by calling the `scanUpToString` method. The method is smart enough to extract the value surrounded by the double quotes. Once a field value is retrieved, we then repeat the same procedure for the remainder of the string.

After all the field values are retrieved, we save them into a tuple and then put it into the `items` array.

Preloading the Data and Saving it into Database

Now that you've created the method for CSV parsing, we now move onto the implementation of data preloading. The preloading will work like this: First, we will remove all the existing data from the database. This operation is optional if you can ensure the database is empty.

Next, we will call up the `parseCSV` method to parse `menudata.csv`. Once the parsing completes, we insert the food menu items into the database.

Insert the following code snippets in the `AppDelegate.swift` file:

```
func preloadData () {

    // Load the data file. For any reasons it can't be loaded, we just return
    guard let contentsOfURL = NSBundle mainBundle().URLForResource("menudata",
withExtension: "csv") else {
        return
    }

    // Remove all the menu items before preloading
    removeData()
    if let items = parseCSV(contentsOfURL, encoding: NSUTF8StringEncoding) {
        // Preload the menu items
        for item in items {
            let menuItem =
NSEntityDescription.insertNewObjectForEntityForName("MenuItem",
inManagedObjectContext: managedObjectContext) as! MenuItem
            menuItem.name = item.name
            menuItem.detail = item.detail
            menuItem.price = (item.price as NSString).doubleValue

            do {
                try managedObjectContext.save()
            } catch {
                print(error)
            }
        }
    }
}
```

```

func removeData () {
    // Remove the existing items
    let fetchRequest = NSFetchedRequest(entityName: "MenuItem")

    do {
        let menuItems = try
            managedObjectContext.executeFetchRequest(fetchRequest) as! [MenuItem]
        for menuItem in menuItems {
            managedObjectContext.deleteObject(menuItem)
        }
    } catch {
        print(error)
    }
}

```

The `removeData` method is used to remove any existing menu items from the database. I want to ensure the database is empty before populating the data extracted from the `menudata.csv` file. The implementation of the method is very straightforward if you have a basic understanding of Core Data. We first execute a query to retrieve all the menu items from the database and call the `deleteObject` method to delete the item one by one. Okay, now let's talk about the `preloadData` method.

In the method we first retrieve the file URL of the `menudata.csv` file using this line of code:

```
NSBundle mainBundle().URLForResource("menudata", withExtension: "csv")
```

After calling the `removeData` method, we execute the `parseCSV` method to parse the `menudata.csv` file. With the returned items, we insert them one by one by calling the `NSEntityDescription.insertNewObjectForEntityForName` method.

Lastly, execute the `preloadData()` method in the `didFinishLaunchingWithOptions` method:

```

func application(application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [NSObject: AnyObject]?) -> Bool {

    preloadData()

    return true
}

```

Now you're ready to test your app. Hit the Run button to launch the app. If you've followed the

implementation correctly, the app should be preloaded with the food items.

But there is an issue with the current implementation. Every time you launch the app, it preloads the data from the CSV file. Apparently, you only want to perform the preloading once. Change the `application:didFinishLaunchingWithOptions:` method to the following:

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {

    let defaults = NSUserDefaults.standardUserDefaults()
    let isPreloaded = defaults.boolForKey("isPreloaded")
    if !isPreloaded {
        preloadData()
        defaults.setBool(true, forKey: "isPreloaded")
    }

    return true
}
```

To indicate that the app has preloaded the data, we save a setting to the defaults system using a specific key (i.e. `isPreloaded`). Every time when the app is launched, we will first check if the value of the `isPreloaded` key. If it's set to `true`, we will skip the data preloading operation.

Using External Data Files

So far the CSV file is bundled in the app. If your data is static, it is completely fine. But what if you're going to change the data frequently? In this case, whenever there is a new update for the data file, you will have to rebuild the app and redeploy it to the app store.

There is a better way to handle this.

Instead of embedding the data file in the app, you put it in an external source. For example, you can store it on a cloud server. Every time when a user opens the app, it goes up to the server and download the data file. Then the app parses the file and loads the data into the database as usual. I have uploaded the sample data file to Google Drive and share it as a public file. You can access it through the URL below:

<https://googledrive.com/host/0ByZhaKOAvtNGTHhXUUUpGS3VqZnM/menudata.csv>

Quick note: If you also want to host your file using Google Drive, you can

follow [this guide](#) to create a public folder to store your files. Once you create the public folder, you can use the following direct link to access the file:

<https://googledrive.com/host/>

Please replace `with` your folder ID. You can look up the folder ID by clicking your public folder. The URL will be something like this:

<https://drive.google.com/drive/folders/0ByZhaKOAvtNGTHhXUUpGS3VqZnM>

In the example, "`0ByZhaKOAvtNGTHhXUUpGS3VqZnM`" is the folder ID.

This is just for demo purpose. If you have your own server, feel free to upload the file to the server and use your own URL. To load the data file from the remote server, all you need to do is make a little tweak to the code. First, update the `preloadData` method to the following:

```
func loadData () {

    // Load the data file. For any reasons it can't be loaded, we just return
    guard let remoteURL = NSURL(string:
"https://googledrive.com/host/0ByZhaKOAvtNGTHhXUUpGS3VqZnM/menudata.csv") else
{
    return
}

// Remove all the menu items before preloading
removeData()

if let items = parseCSV(remoteURL, encoding: NSUTF8StringEncoding) {
    // Preload the menu items
    for item in items {
        let menuItem =
NSEntityDescription.insertNewObjectForEntityForName("MenuItem",
inManagedObjectContext: managedObjectContext) as! MenuItem
        menuItem.name = item.name
        menuItem.detail = item.detail
        menuItem.price = (item.price as NSString).doubleValue

        do {
            try managedObjectContext.save()
        } catch {
            print(error)
        }
    }
}
```

```
    }  
}
```

The code is very similar to the original one. Instead loading the data file from the bundle, we specify the remote URL and pass it to the `parseCSV` method. That's it. The `parseCSV` method will handle the file download and perform the data parsing accordingly.

Before running the app, you have to update the `application:didFinishLaunchingWithOptions:` method so that the app will load the data every time it runs:

```
func application(application: UIApplication, didFinishLaunchingWithOptions  
launchOptions: [NSObject: AnyObject]?) -> Bool {  
  
    preloadData()  
  
    return true  
}
```

You're ready to go. Hit the Run button and test the app again. The menu items should be different from those shown previously.



For reference, you can download the complete Xcode project from

<https://www.dropbox.com/s/ow12gmg1ldxui5/CoreDataPreloadDemo.zip?dl=0>.

Using An Existing Database in Your Project

Now that you should know how to populate a database with external data, you may wonder if you can use an existing SQLite database directly. In some situations, you probably do not want to preload the data during app launch. For example, you need to preload hundreds of thousands of records. This will take some time to load the data and results a poor user experience. Apparently, you want to pre-filled the database beforehand and bundle it directly in the app.

Suppose you've already pre-filled an existing database with data, how can you bundle it in your app?

Before I show you the procedures, please download the starter project again from <https://www.dropbox.com/s/l3m5gky5qu959jt/CoreDataPreloadDemoStart.zip?dl=0>. As a demo, we will copy the existing database created in the previous section to this starter project.

Now open up the Xcode project that you have worked on earlier. If you've followed me along, your database should be pre-filled with data. We will now copy it to the starter project that you have just downloaded.

But where is the SQLite database?

The database is not bundled in the Xcode project but automatically created when you run the app in the simulator. To locate the database, you will need to add a line of code to reveal the file path. Update the `application:didFinishLaunchingWithOptions:` method to the following:

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {  
    print(applicationDocumentsDirectory.path)  
    preloadData()  
  
    return true  
}
```

The SQLite database is generated under the application's document directory. To find the file path, we simply print out the value of `applicationDocumentsDirectory.path` variable.

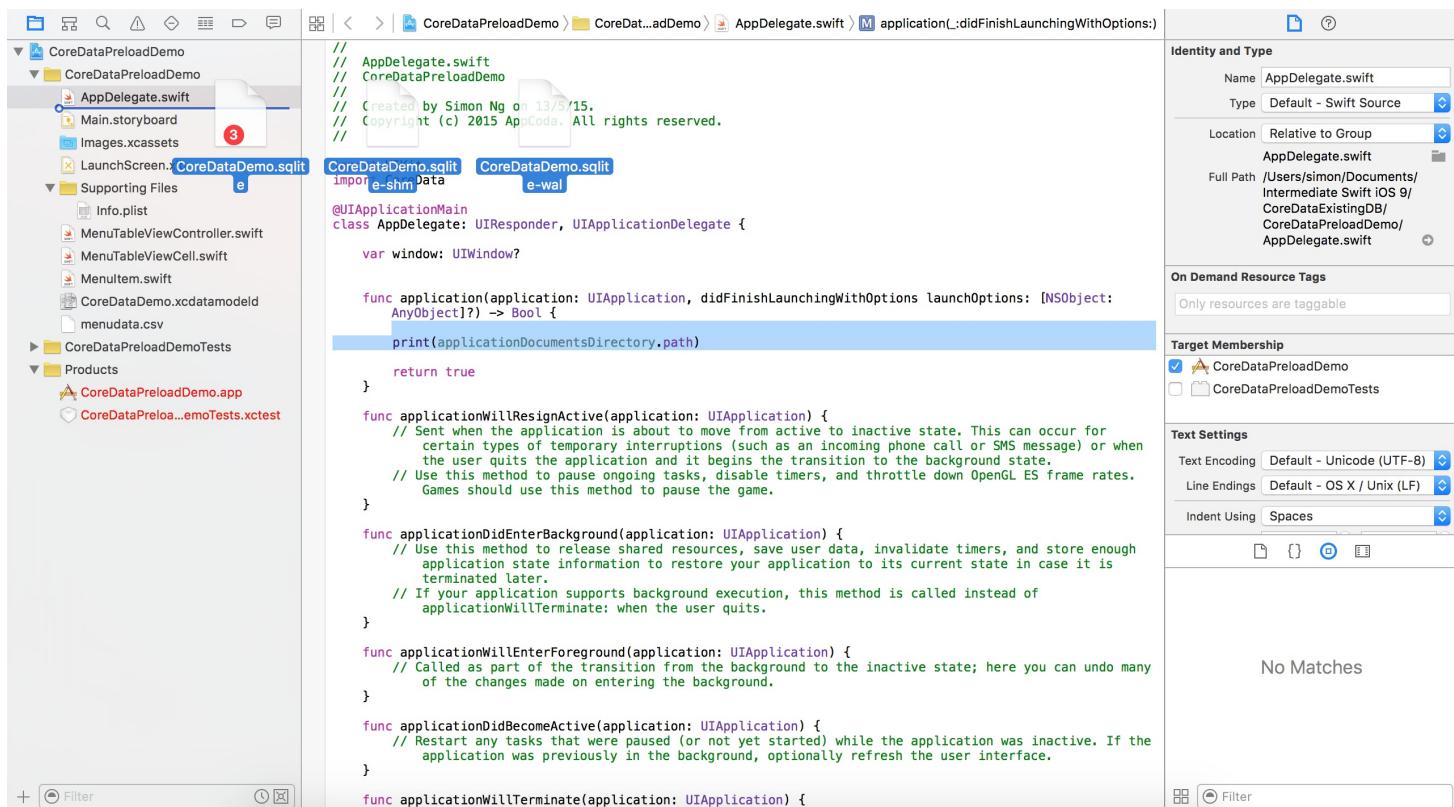
Now run the app again. You should see an output in the console window showing the full path of the document directory like this:

```
Optional("/Users/simon/Library/Developer/CoreSimulator/Devices/6BC1B40E-BB8C-  
4CA8-B7FF-54A1684E3ACF/data/Containers/Data/Application/4DD3C821-D5BB-4647-  
9169-6AD9705F4E36/Documents")
```

Copy the file path and go to Finder. In the menu select Go > Go to Folder... and then paste the path in the pop-up. Click `Go` to confirm. Once you open the document folder in Finder, you will find three files: `CoreDataDemo.sqlite`, `CoreDataDemo.sqlite-wal` and `CoreDataDemo.sqlite-shm`. Starting from iOS 7, the default journaling mode for Core Data SQLite stores is set to Write-Ahead Logging (WAL). With the WAL mode, Core Data keeps the main .sqlite file untouched and appends transactions to a .sqlite-wal file in the same folder.

When running WAL mode, SQLite will also create a shared memory file with .sqlite-shm extension. In order to backup the database or use it to in other projects, you will need copy these three files. If you just copy the CoreDataDemo.sqlite file, you will probably end up with an empty database.

Now, drag these three files to the starter project you just download.



When prompted, please ensure the *Copy item if needed* option is checked and the `CoreDataPreloadDemo` option of *Add to Targets* is selected. Then click `Finish` to confirm. Now that you've bundled an existing database in your Xcode project. When you build the app, this database will be embedded in the app. But you will have to tweak the code a bit before the app is able to use the database.

By default, the app will create an empty SQLite store if there is no database found in the document directory. So all you need to do is copy the database files bundled in the app to that directory. In the `AppDelegate` class update the declaration of the `persistentStoreCoordinator` variable like this:

```
lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator = {
    // The persistent store coordinator for the application. This
```

implementation creates and returns a coordinator, having added the store for the application to it. This property is optional since there are legitimate error conditions that could cause the creation of the store to fail.

```

// Create the coordinator and store
let coordinator = NSPersistentStoreCoordinator(managedObjectModel:
self.managedObjectModel)
let url =
self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataDemo.sql")

// Load the existing database
if !NSFileManager.defaultManager().fileExistsAtPath(url.path!) {
    let sourceSqliteURLS =
[NSBundle mainBundle().URLForResource("CoreDataDemo", withExtension:
"sqlite")!, mainBundle mainBundle().URLForResource("CoreDataDemo", withExtension:
"sqlite-wal")!, mainBundle mainBundle().URLForResource("CoreDataDemo",
withExtension: "sqlite-shm")!]
    let destSqliteURLS =
[self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataDemo.sc
self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataDemo.sql
wal"),
self.applicationDocumentsDirectory.URLByAppendingPathComponent("CoreDataDemo.sql
shm")]

    for var index = 0; index < sourceSqliteURLS.count; index++ {
        do {
            try
NSFileManager.defaultManager().copyItemAtURL(sourceSqliteURLS[index], toURL:
destSqliteURLS[index])
        } catch {
            print(error)
        }
    }
}

var failureReason = "There was an error creating or loading the
application's saved data."
do {
    try coordinator.addPersistentStoreWithType(NSSQLiteStoreType,
configuration: nil, URL: url, options: nil)
} catch {
    // Report any error we got.
    var dict = [String: AnyObject]()
    dict[NSLocalizedDescriptionKey] = "Failed to initialize the
application's saved data"
    dict[NSLocalizedFailureReasonErrorKey] = failureReason

    dict[NSUnderlyingErrorKey] = error as NSError
}

```

```

    let wrappedError = NSError(domain: "YOUR_ERROR_DOMAIN", code: 9999,
userInfo: dict)
    // Replace this with code to handle the error appropriately.
    // abort() causes the application to generate a crash log and
terminate. You should not use this function in a shipping application, although
it may be useful during development.
    NSLog("Unresolved error \(wrappedError), \(wrappedError.userInfo)")
    abort()
}

return coordinator
}()

```

We first verify if the database exists in the document folder. If not, we copy the SQLite files from the bundle folder to the document folder by calling the `copyItemAtURL` method of `NSFileManager`.

That's it! Before you hit the Run button to test the app, you better delete the `CoreDataPreloadDemo` app from the simulator or simply reset it (select iOS Simulator > Reset Content and Settings). This is to remove any existing SQLite databases from the simulator.

Okay, now you're good to go. When the app is launched, it should be able to use the database bundled in the Xcode project. For reference, you can download the final Xcode project from <https://www.dropbox.com/s/oxo878t9kogrsek/CoreDataExistingDB.zip?dl=0>.